

Table of Contents

[Introduction](#)

[Get started](#)

[Create a web app](#)

[Create a Web API](#)

[Tutorials](#)

[Create a Razor Pages web app](#)

[Get started with Razor Pages](#)

[Add a model](#)

[Scaffolded Razor Pages](#)

[SQL Server LocalDB](#)

[Update the pages](#)

[Add search](#)

[Add a new field](#)

[Add validation](#)

[Upload files](#)

[Create an MVC web app](#)

[Get started](#)

[Add a controller](#)

[Add a view](#)

[Add a model](#)

[Work with SQL Server LocalDB](#)

[Controller methods and views](#)

[Add search](#)

[Add a new field](#)

[Add validation](#)

[Examine the Details and Delete methods](#)

[Data access - Razor Pages with EF Core](#)

[Get started](#)

[Create, Read, Update, and Delete operations](#)

Sort, filter, page, and group

Migrations

Create a complex data model

Read related data

Update related data

Handle concurrency conflicts

Data access - MVC with EF Core

Get started

Create, Read, Update, and Delete operations

Sort, filter, page, and group

Migrations

Create a complex data model

Read related data

Update related data

Handle concurrency conflicts

Inheritance

Advanced topics

Create backend services for mobile apps

Build Web APIs

Create a Web API

ASP.NET Core Web API help pages using Swagger

Create backend services for native mobile apps

Fundamentals

Application startup

Dependency injection (services)

Middleware

Work with static files

Routing

URL rewriting middleware

Work with multiple environments

Configuration and options

Configuration

Options

Logging

Logging with LoggerMessage

Error handling

File providers

Hosting

Session and app state

Servers

Kestrel

ASP.NET Core Module

HTTP.sys

Globalization and localization

Configure Portable Object localization with Orchard Core

Request features

Primitives

Change tokens

Open Web Interface for .NET (OWIN)

WebSockets

Microsoft.AspNetCore.All metapackage

Choose between .NET Core and .NET Framework

Choose between ASP.NET Core and ASP.NET

MVC

Razor Pages

Razor syntax

Route and app convention features

Model binding

Model validation

Views

Razor syntax

View compilation

Layout

Tag Helpers

Partial views

Dependency injection into views

View components

Controllers

Route to controller actions

File uploads

Dependency injection into controllers

Test controllers

Advanced

Work with the app model

Filters

Areas

Application parts

Custom model binding

Custom formatters

Format response data

Test and debug

Unit testing

Integration testing

Razor Pages testing

Test controllers

Remote debugging

Snapshot debugging

Data access and storage

Get started with Razor Pages and Entity Framework Core using Visual Studio

Get started with ASP.NET Core and Entity Framework Core using Visual Studio

ASP.NET Core with EF Core - new database

ASP.NET Core with EF Core - existing database

Get started with ASP.NET Core and Entity Framework 6

Azure Storage

Add Azure Storage by using Visual Studio Connected Services

Get started with Blob storage and Visual Studio Connected Services

[Get Started with Queue Storage and Visual Studio Connected Services](#)

[Get Started with Table Storage and Visual Studio Connected Services](#)

Client-side development

[Use Gulp](#)

[Use Grunt](#)

[Manage client-side packages with Bower](#)

[Build responsive sites with Bootstrap](#)

[Style apps with LESS, Sass, and Font Awesome](#)

[Bundle and minify](#)

[Use Browser Link](#)

[Use JavaScriptServices for SPAs](#)

[Use the SPA project templates \(RC\)](#)

[Angular project template](#)

[React project template](#)

[React with Redux project template](#)

Mobile

[Create backend services for native mobile apps](#)

Host and deploy

[Host on Azure App Service](#)

[Publish to Azure with Visual Studio](#)

[Publish to Azure with CLI tools](#)

[Continuous deployment to Azure with Visual Studio and Git](#)

[Continuous deployment to Azure with VSTS](#)

[Troubleshoot ASP.NET Core on Azure App Service](#)

[Host on Windows with IIS](#)

[Troubleshoot ASP.NET Core on IIS](#)

[ASP.NET Core Module configuration reference](#)

[Development-time IIS support in Visual Studio for ASP.NET Core](#)

[IIS Modules with ASP.NET Core](#)

[Host in a Windows service](#)

[Host on Linux with Nginx](#)

[Host on Linux with Apache](#)

Host in Docker

[Build Docker images](#)

[Visual Studio Tools for Docker](#)

[Publish to a Docker image](#)

[Visual Studio publish profiles](#)

[Directory structure](#)

[Common errors reference for Azure App Service and IIS](#)

[Add app features from an external assembly using IHostingStartup](#)

Security

Authentication

[Community OSS authentication options](#)

[Introduction to Identity](#)

[Configure Identity](#)

[Configure Windows Authentication](#)

[Configure primary key type for Identity](#)

[Custom storage providers for Identity](#)

[Enable authentication using Facebook, Google, and other external providers](#)

[Account confirmation and password recovery](#)

[Enable QR code generation in Identity](#)

[Two-factor authentication with SMS](#)

[Use Cookie Authentication without Identity](#)

[Azure Active Directory](#)

[Secure ASP.NET Core apps with IdentityServer4](#)

[Secure ASP.NET Core apps with Azure App Service authentication \(Easy Auth\)](#)

Authorization

[Introduction](#)

[Create an app with user data protected by authorization](#)

[Razor Pages authorization](#)

[Simple authorization](#)

[Role-based authorization](#)

[Claims-based authorization](#)

[Policy-based authorization](#)

Dependency injection in requirement handlers

Resource-based authorization

View-based authorization

Limit identity by scheme

Data protection

Introduction to data protection

Get started with the Data Protection APIs

Consumer APIs

Configuration

Extensibility APIs

Implementation

Compatibility

Enforce SSL

Safe storage of app secrets during development

Azure Key Vault configuration provider

Anti-request forgery

Prevent open redirect attacks

Prevent Cross-Site Scripting

Enable Cross-Origin Requests (CORS)

Performance

Caching

In-memory caching

Work with a distributed cache

Response caching

Response caching middleware

Response compression middleware

Migration

ASP.NET to ASP.NET Core 1.x

Configuration

Authentication and Identity

Web API

HTTP modules to middleware

[ASP.NET to ASP.NET Core 2.0](#)

[ASP.NET Core 1.x to 2.0](#)

[Authentication and Identity](#)

[API Reference](#)

[2.0 release notes](#)

[1.1 release notes](#)

[Earlier release notes](#)

[VS 2015/project.json docs](#)

[Contribute](#)

Introduction to ASP.NET Core

1/10/2018 • 2 min to read • [Edit Online](#)

By [Daniel Roth](#), [Rick Anderson](#), and [Shaun Luttin](#)

ASP.NET Core is a cross-platform, high-performance, [open-source](#) framework for building modern, cloud-based, Internet-connected applications. With ASP.NET Core, you can:

- Build web apps and services, [IoT](#) apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on [.NET Core](#) or [.NET Framework](#).

Why use ASP.NET Core?

Millions of developers have used (and continue to use) [ASP.NET 4.x](#) to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, with architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Integration of [modern, client-side frameworks](#) and development workflows.
- A cloud-ready, environment-based [configuration system](#).
- Built-in [dependency injection](#).
- A lightweight, [high-performance](#), and modular HTTP request pipeline.
- Ability to host on [IIS](#), [Nginx](#), [Apache](#), [Docker](#), or self-host in your own process.
- Side-by-side app versioning when targeting [.NET Core](#).
- Tooling that simplifies modern web development.
- Ability to build and run on Windows, macOS, and Linux.
- Open-source and [community-focused](#).

ASP.NET Core ships entirely as [NuGet](#) packages. This allows you to optimize your app to include only the necessary NuGet packages. In fact, ASP.NET Core 2.x apps targeting .NET Core only require a [single NuGet package](#). The benefits of a smaller app surface area include tighter security, reduced servicing, and improved performance.

Build web APIs and web UI using ASP.NET Core MVC

ASP.NET Core MVC provides features to build [web APIs](#) and [web apps](#):

- The [Model-View-Controller \(MVC\) pattern](#) helps make your web APIs and web apps [testable](#).
- [Razor Pages](#) (new in ASP.NET Core 2.0) is a page-based programming model that makes building web UI easier and more productive.
- [Razor markup](#) provides a productive syntax for [Razor Pages](#) and [MVC views](#).
- [Tag Helpers](#) enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for [multiple data formats and content negotiation](#) lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- [Model binding](#) automatically maps data from HTTP requests to action method parameters.
- [Model validation](#) automatically performs client- and server-side validation.

Client-side development

ASP.NET Core integrates seamlessly with popular client-side frameworks and libraries, including [Angular](#), [React](#), and [Bootstrap](#). See [Client-side development](#) for more details.

Next steps

For more information, see the following resources:

- [ASP.NET Core tutorials](#)
- [ASP.NET Core fundamentals](#)
- [The weekly ASP.NET community standup](#) covers the team's progress and plans. It features new blogs and third-party software.

Get Started with ASP.NET Core

12/21/2017 • 1 min to read • [Edit Online](#)

NOTE

These instructions are for the latest version of ASP.NET Core. Looking to get started with an earlier version? See [the 1.1 version of this tutorial](#).

1. Install [.NET Core](#).
2. Create a new .NET Core project.

On macOS and Linux, open a terminal window. On Windows, open a command prompt. Enter the following command:

```
dotnet new razor -o aspnetcoreapp
```

3. Run the app.

Use the following commands to run the app:

```
cd aspnetcoreapp
dotnet run
```

4. Browse to <http://localhost:5000>
5. Open *Pages/About.cshtml* and modify the page to display the message "Hello, world! The time on the server is @DateTime.Now ":

```
@page
@model AboutModel
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"]</h2>
<h3>@Model.Message</h3>

<p>Hello, world! The time on the server is @DateTime.Now</p>
```

6. Browse to <http://localhost:5000/About> and verify the changes.

Next steps

For getting-started tutorials, see [ASP.NET Core Tutorials](#)

For an introduction to ASP.NET Core concepts and architecture, see [ASP.NET Core Introduction](#) and [ASP.NET Core Fundamentals](#).

An ASP.NET Core app can use the .NET Core or .NET Framework Base Class Library and runtime. For more information, see [Choosing between .NET Core and .NET Framework](#).

Introduction to Razor Pages in ASP.NET Core

12/19/2017 • 16 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Ryan Nowak](#)

Razor Pages is a new feature of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

If you're looking for a tutorial that uses the Model-View-Controller approach, see [Getting started with ASP.NET Core MVC](#).

This document provides an introduction to Razor Pages. It's not a step by step tutorial. If you find some of the sections difficult to follow, see [Getting started with Razor Pages](#).

ASP.NET Core 2.0 prerequisites

Install [.NET Core](#) 2.0.0 or later.

If you're using Visual Studio, install [Visual Studio](#) 2017 version 15.3 or later with the following workloads:

- **ASP.NET and web development**
- **.NET Core cross-platform development**

Creating a Razor Pages project

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)

See [Getting started with Razor Pages](#) for detailed instructions on how to create a Razor Pages project using Visual Studio.

Razor Pages

Razor Pages is enabled in *Startup.cs*:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Includes support for Razor Pages and controllers.
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
    }
}
```

Consider a basic page:

```
@page
```

```
<h1>Hello, world!</h1>  
<h2>The time on the server is @DateTime.Now</h2>
```

The preceding code looks a lot like a Razor view file. What makes it different is the `@page` directive. `@page` makes the file into an MVC action - which means that it handles requests directly, without going through a controller.

`@page` must be the first Razor directive on a page. `@page` affects the behavior of other Razor constructs.

A similar page, using a `PageModel` class, is shown in the following two files. The *Pages/Index2.cshtml* file:

```
@page  
@using RazorPagesIntro.Pages  
@model IndexModel2  
  
<h2>Separate page model</h2>  
<p>  
    @Model.Message  
</p>
```

The *Pages/Index2.cshtml.cs* "code-behind" file:

```
using Microsoft.AspNetCore.Mvc.RazorPages;  
using System;  
  
namespace RazorPagesIntro.Pages  
{  
    public class IndexModel2 : PageModel  
    {  
        public string Message { get; private set; } = "PageModel in C#";  
  
        public void OnGet()  
        {  
            Message += $" Server time is { DateTime.Now }";  
        }  
    }  
}
```

By convention, the `PageModel` class file has the same name as the Razor Page file with `.cs` appended. For example, the previous Razor Page is *Pages/Index2.cshtml*. The file containing the `PageModel` class is named *Pages/Index2.cshtml.cs*.

The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Page path and the matching URL:

FILE NAME AND PATH	MATCHING URL
<i>/Pages/Index.cshtml</i>	<code>/</code> Or <code>/Index</code>
<i>/Pages/Contact.cshtml</i>	<code>/Contact</code>
<i>/Pages/Store/Contact.cshtml</i>	<code>/Store/Contact</code>
<i>/Pages/Store/Index.cshtml</i>	<code>/Store</code> Or <code>/Store/Index</code>

Notes:

- The runtime looks for Razor Pages files in the *Pages* folder by default.
- `Index` is the default page when a URL doesn't include a page.

Writing a basic form

Razor Pages features are designed to make common patterns used with web browsers easy. [Model binding](#), [Tag Helpers](#), and HTML helpers all *just work* with the properties defined in a Razor Page class. Consider a page that implements a basic "contact us" form for the `Contact` model:

For the samples in this document, the `DbContext` is initialized in the [Startup.cs](#) file.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesContacts.Data;

namespace RazorPagesContacts
{
    public class Startup
    {
        public IHostingEnvironment HostingEnvironment { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(options =>
                options.UseInMemoryDatabase("name"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

The data model:

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Data
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
        public string Name { get; set; }
    }
}
```

The db context:

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions options)
            : base(options)
        {
        }

        public DbSet<Customer> Customers { get; set; }
    }
}
```

The *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

The *Pages/Create.cshtml.cs* code-behind file for the view:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class CreateModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }
    }
}

```

By convention, the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page.

The `PageModel` class allows separation of the logic of a page from its presentation. It defines page handlers for requests sent to the page and the data used to render the page. This separation allows you to manage page dependencies through [dependency injection](#) and to [unit test](#) the pages.

The page has an `OnPostAsync` *handler method*, which runs on `POST` requests (when a user posts the form). You can add handler methods for any HTTP verb. The most common handlers are:

- `OnGet` to initialize state needed for the page. [OnGet](#) sample.
- `OnPost` to handle form submissions.

The `Async` naming suffix is optional but is often used by convention for asynchronous functions. The `OnPostAsync` code in the preceding example looks similar to what you would normally write in a controller. The preceding code is typical for Razor Pages. Most of the MVC primitives like [model binding](#), [validation](#), and action results are shared.

The previous `OnPostAsync` method:

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}

```

The basic flow of `OnPostAsync` :

Check for validation errors.

- If there are no errors, save the data and redirect.
- If there are errors, show the page again with validation messages. Client-side validation is identical to traditional ASP.NET Core MVC applications. In many cases, validation errors would be detected on the client, and never submitted to the server.

When the data is entered successfully, the `OnPostAsync` handler method calls the `RedirectToPage` helper method to return an instance of `RedirectToPageResult`. `RedirectToPage` is a new action result, similar to `RedirectToAction` or `RedirectToRoute`, but customized for pages. In the preceding sample, it redirects to the root Index page (`/Index`). `RedirectToPage` is detailed in the [URL generation for Pages](#) section.

When the submitted form has validation errors (that are passed to the server), the `OnPostAsync` handler method calls the `Page` helper method. `Page` returns an instance of `PageResult`. Returning `Page` is similar to how actions in controllers return `View`. `PageResult` is the default return type for a handler method. A handler method that returns `void` renders the page.

The `Customer` property uses `[BindProperty]` attribute to opt in to model binding.

```
public class CreateModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateModel(AppDbContext db)
    {
        _db = db;
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        return RedirectToPage("/Index");
    }
}
```

Razor Pages, by default, bind properties only with non-GET verbs. Binding to properties can reduce the amount of code you have to write. Binding reduces code by using the same property to render form fields (`<input asp-for="Customer.Name" />`) and accept the input.

The home page (`Index.cshtml`):

```

@page
@model RazorPagesContacts.Pages.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts</h1>
<form method="post">
  <table class="table">
    <thead>
      <tr>
        <th>ID</th>
        <th>Name</th>
      </tr>
    </thead>
    <tbody>
      @foreach (var contact in Model.Customers)
      {
        <tr>
          <td>@contact.Id</td>
          <td>@contact.Name</td>
          <td>
            <a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
            <button type="submit" asp-page-handler="delete"
              asp-route-id="@contact.Id">delete</button>
          </td>
        </tr>
      }
    </tbody>
  </table>

  <a asp-page="./Create">Create</a>
</form>

```

The code behind *Index.cshtml.cs* file:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Pages
{
    public class IndexModel : PageModel
    {
        private readonly AppDbContext _db;

        public IndexModel(AppDbContext db)
        {
            _db = db;
        }

        public IList<Customer> Customers { get; private set; }

        public async Task OnGetAsync()
        {
            Customers = await _db.Customers.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostDeleteAsync(int id)
        {
            var contact = await _db.Customers.FindAsync(id);

            if (contact != null)
            {
                _db.Customers.Remove(contact);
                await _db.SaveChangesAsync();
            }

            return RedirectToPage();
        }
    }
}

```

The *Index.cshtml* file contains the following markup to create an edit link for each contact:

```
<a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
```

The [Anchor Tag Helper](#) used the `asp-route-{value}` attribute to generate a link to the Edit page. The link contains route data with the contact ID. For example, `http://localhost:5000/Edit/1`.

The *Pages/Edit.cshtml* file:

```
@page "{id:int}"
@model RazorPagesContacts.Pages.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@{
    ViewData["Title"] = "Edit Customer";
}

<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name" ></span>
        </div>
    </div>

    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

The first line contains the `@page "{id:int}"` directive. The routing constraint `"{id:int}"` tells the page to accept requests to the page that contain `int` route data. If a request to the page doesn't contain route data that can be converted to an `int`, the runtime returns an HTTP 404 (not found) error.

The *Pages/Edit.cshtml.cs* file:

```

using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Customer = await _db.Customers.FindAsync(id);

            if (Customer == null)
            {
                return RedirectToPage("/Index");
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Attach(Customer).State = EntityState.Modified;

            try
            {
                await _db.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                throw new Exception($"Customer {Customer.Id} not found!");
            }

            return RedirectToPage("/Index");
        }
    }
}

```

The *Index.cshtml* file also contains markup to create a delete button for each customer contact:

```

<button type="submit" asp-page-handler="delete"
        asp-route-id="@contact.Id">delete</button>

```

When the delete button is rendered in HTML, its `formaction` includes parameters for:

- The customer contact ID specified by the `asp-route-id` attribute.

- The `handler` specified by the `asp-page-handler` attribute.

Here is an example of a rendered delete button with a customer contact ID of `1`:

```
<button type="submit" formaction="/?id=1&handler=delete">delete</button>
```

When the button is selected, a form `POST` request is sent to the server. By convention, the name of the handler method is selected based the value of the `handler` parameter according to the scheme `OnPost[handler]Async`.

Because the `handler` is `delete` in this example, the `OnPostDeleteAsync` handler method is used to process the `POST` request. If the `asp-page-handler` is set to a different value, such as `remove`, a page handler method with the name `OnPostRemoveAsync` is selected.

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _db.Customers.FindAsync(id);

    if (contact != null)
    {
        _db.Customers.Remove(contact);
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

The `OnPostDeleteAsync` method:

- Accepts the `id` from the query string.
- Queries the database for the customer contact with `FindAsync`.
- If the customer contact is found, they're removed from the list of customer contacts. The database is updated.
- Calls `RedirectToPage` to redirect to the root Index page (`/Index`).

XSRF/CSRF and Razor Pages

You don't have to write any code for [antiforgery validation](#). Antiforgery token generation and validation are automatically included in Razor Pages.

Using Layouts, partials, templates, and Tag Helpers with Razor Pages

Pages work with all the features of the Razor view engine. Layouts, partials, templates, Tag Helpers, `_ViewStart.cshtml`, `_ViewImports.cshtml` work in the same way they do for conventional Razor views.

Let's declutter this page by taking advantage of some of those features.

Add a [layout page](#) to `Pages/_Layout.cshtml`:

```

<!DOCTYPE html>
<html>
<head>
  <title>Razor Pages Sample</title>
</head>
<body>
  <a asp-page="/Index">Home</a>
  @RenderBody()
  <a asp-page="/Customers/Create">Create</a> <br />
</body>
</html>

```

The [Layout](#):

- Controls the layout of each page (unless the page opts out of layout).
- Imports HTML structures such as JavaScript and stylesheets.

See [layout page](#) for more information.

The [Layout](#) property is set in *Pages/_ViewStart.cshtml*:

```

@{
  Layout = "_Layout";
}

```

Note: The layout is in the *Pages* folder. Pages look for other views (layouts, templates, partials) hierarchically, starting in the same folder as the current page. A layout in the *Pages* folder can be used from any Razor page under the *Pages* folder.

We recommend you **not** put the layout file in the *Views/Shared* folder. *Views/Shared* is an MVC views pattern. Razor Pages are meant to rely on folder hierarchy, not path conventions.

View search from a Razor Page includes the *Pages* folder. The layouts, templates, and partials you're using with MVC controllers and conventional Razor views *just work*.

Add a *Pages/_ViewImports.cshtml* file:

```

@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

```

`@namespace` is explained later in the tutorial. The `@addTagHelper` directive brings in the [built-in Tag Helpers](#) to all the pages in the *Pages* folder.

When the `@namespace` directive is used explicitly on a page:

```

@page
@namespace RazorPagesIntro.Pages.Customers

@model NameSpaceModel

<h2>Name space</h2>
<p>
  @Model.Message
</p>

```

The directive sets the namespace for the page. The `@model` directive doesn't need to include the namespace.

When the `@namespace` directive is contained in *_ViewImports.cshtml*, the specified namespace supplies the prefix for

the generated namespace in the Page that imports the `@namespace` directive. The rest of the generated namespace (the suffix portion) is the dot-separated relative path between the folder containing `_ViewImports.cshtml` and the folder containing the page.

For example, the code behind file `Pages/Customers/Edit.cshtml.cs` explicitly sets the namespace:

```
namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly ApplicationDbContext _db;

        public EditModel(ApplicationDbContext db)
        {
            _db = db;
        }

        // Code removed for brevity.
    }
}
```

The `Pages/_ViewImports.cshtml` file sets the following namespace:

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The generated namespace for the `Pages/Customers/Edit.cshtml` Razor Page is the same as the code behind file. The `@namespace` directive was designed so the C# classes added to a project and pages-generated code *just work* without having to add an `@using` directive for the code behind file.

Note: `@namespace` also works with conventional Razor views.

The original `Pages/Create.cshtml` view file:

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

The updated `Pages/Create.cshtml` view file:

```

@page
@model CreateModel

<html>
<body>
  <p>
    Enter your name.
  </p>
  <div asp-validation-summary="All"></div>
  <form method="POST">
    <div>Name: <input asp-for="Customer.Name" /></div>
    <input type="submit" />
  </form>
</body>
</html>

```

The [Razor Pages starter project](#) contains the *Pages/_ValidationScriptsPartial.cshtml*, which hooks up client-side validation.

URL generation for Pages

The `Create` page, shown previously, uses `RedirectToPage` :

```

public async Task<IActionResult> OnPostAsync()
{
  if (!ModelState.IsValid)
  {
    return Page();
  }

  _db.Customers.Add(Customer);
  await _db.SaveChangesAsync();
  return RedirectToPage("/Index");
}

```

The app has the following file/folder structure:

- */Pages*
 - *Index.cshtml*
 - */Customer*
 - *Create.cshtml*
 - *Edit.cshtml*
 - *Index.cshtml*

The *Pages/Customers/Create.cshtml* and *Pages/Customers/Edit.cshtml* pages redirect to *Pages/Index.cshtml* after success. The string `/Index` is part of the URI to access the preceding page. The string `/Index` can be used to generate URIs to the *Pages/Index.cshtml* page. For example:

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">My Index Page`
- `RedirectToPage("/Index")`

The page name is the path to the page from the root */Pages* folder (including a leading `/`, for example `/Index`). The preceding URL generation samples are much more feature rich than just hardcoding a URL. URL generation uses [routing](#) and can generate and encode parameters according to how the route is defined in the destination path.

URL generation for pages supports relative names. The following table shows which Index page is selected with different `RedirectToPage` parameters from `Pages/Customers/Create.cshtml`:

REDIRECTTOPAGE(X)	PAGE
<code>RedirectToPage("/Index")</code>	<code>Pages/Index</code>
<code>RedirectToPage("./Index");</code>	<code>Pages/Customers/Index</code>
<code>RedirectToPage("../Index")</code>	<code>Pages/Index</code>
<code>RedirectToPage("Index")</code>	<code>Pages/Customers/Index</code>

`RedirectToPage("Index")`, `RedirectToPage("./Index")`, and `RedirectToPage("../Index")` are *relative names*. The `RedirectToPage` parameter is *combined* with the path of the current page to compute the name of the destination page.

Relative name linking is useful when building sites with a complex structure. If you use relative names to link between pages in a folder, you can rename that folder. All the links still work (because they didn't include the folder name).

TempData

ASP.NET Core exposes the `TempData` property on a `controller`. This property stores data until it is read. The `Keep` and `Peek` methods can be used to examine the data without deletion. `TempData` is useful for redirection, when data is needed for more than a single request.

The `[TempData]` attribute is new in ASP.NET Core 2.0 and is supported on controllers and pages.

The following code sets the value of `Message` using `TempData`:

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

The following markup in the *Pages/Customers/Index.cshtml* file displays the value of `Message` using `TempData`.

```
<h3>Msg: @Model.Message</h3>
```

The *Pages/Customers/Index.cshtml.cs* code-behind file applies the `[TempData]` attribute to the `Message` property.

```
[TempData]  
public string Message { get; set; }
```

See [TempData](#) for more information.

Multiple handlers per page

The following page generates markup for two page handlers using the `asp-page-handler` Tag Helper:

```
@page  
@model CreateFATHModel  
  
<html>  
<body>  
  <p>  
    Enter your name.  
  </p>  
  <div asp-validation-summary="All"></div>  
  <form method="POST">  
    <div>Name: <input asp-for="Customer.Name" /></div>  
    <input type="submit" asp-page-handler="JoinList" value="Join" />  
    <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />  
  </form>  
</body>  
</html>
```

The form in the preceding example has two submit buttons, each using the `FormActionTagHelper` to submit to a different URL. The `asp-page-handler` attribute is a companion to `asp-page`. `asp-page-handler` generates URLs that submit to each of the handler methods defined by a page. `asp-page` is not specified because the sample is linking to the current page.

The code-behind file:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpper();
            return await OnPostJoinListAsync();
        }
    }
}

```

The preceding code uses *named handler methods*. Named handler methods are created by taking the text in the name after `On<HTTP Verb>` and before `Async` (if present). In the preceding example, the page methods are `OnPostJoinListAsync` and `OnPostJoinListUCAsync`. With `OnPost` and `Async` removed, the handler names are `JoinList` and `JoinListUC`.

```



```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `http://localhost:5000/Customers/CreateFATH?handler=JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `http://localhost:5000/Customers/CreateFATH?handler=JoinListUC`.

Customizing Routing

If you don't like the query string `?handler=JoinList` in the URL, you can change the route to put the handler name in the path portion of the URL. You can customize the route by adding a route template enclosed in double quotes after the `@page` directive.

```

@page "{handler?}"
@model CreateRouteModel

<html>
<body>
  <p>
    Enter your name.
  </p>
  <div asp-validation-summary="All"></div>
  <form method="POST">
    <div>Name: <input asp-for="Customer.Name" /></div>
    <input type="submit" asp-page-handler="JoinList" value="Join" />
    <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
  </form>
</body>
</html>

```

The preceding route puts the handler name in the URL path instead of the query string. The `?` following `handler` means the route parameter is optional.

You can use `@page` to add additional segments and parameters to a page's route. Whatever's there is **appended** to the default route of the page. Using an absolute or virtual path to change the page's route (like `"~/Some/Other/Path"`) is not supported.

Configuration and settings

To configure advanced options, use the extension method `AddRazorPagesOptions` on the MVC builder:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.RootDirectory = "/MyPages";
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        });
}

```

Currently you can use the `RazorPagesOptions` to set the root directory for pages, or add application model conventions for pages. We'll enable more extensibility this way in the future.

To precompile views, see [Razor view compilation](#).

[Download or view sample code.](#)

See [Getting started with Razor Pages in ASP.NET Core](#), which builds on this introduction.

Specify that Razor Pages are at the content root

By default, Razor Pages are rooted in the `/Pages` directory. Add `WithRazorPagesAtContentRoot` to `AddMvc` to specify that your Razor Pages are at the content root (`ContentRootPath`) of the app:

```

services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesAtContentRoot();

```

Specify that Razor Pages are at a custom root directory

Add [WithRazorPagesRoot](#) to [AddMvc](#) to specify that your Razor Pages are at a custom root directory in the app (provide a relative path):

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesRoot("/path/to/razor/pages");
```

See also

- [Getting started with Razor Pages](#)
- [Razor Pages authorization conventions](#)
- [Razor Pages custom route and page model providers](#)
- [Razor Pages unit and integration testing](#)

Create a web API with ASP.NET Core and Visual Studio for Windows

12/5/2017 • 8 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Mike Wasson](#)

This tutorial builds a web API for managing a list of "to-do" items. A user interface (UI) is not created.

There are 3 versions of this tutorial:

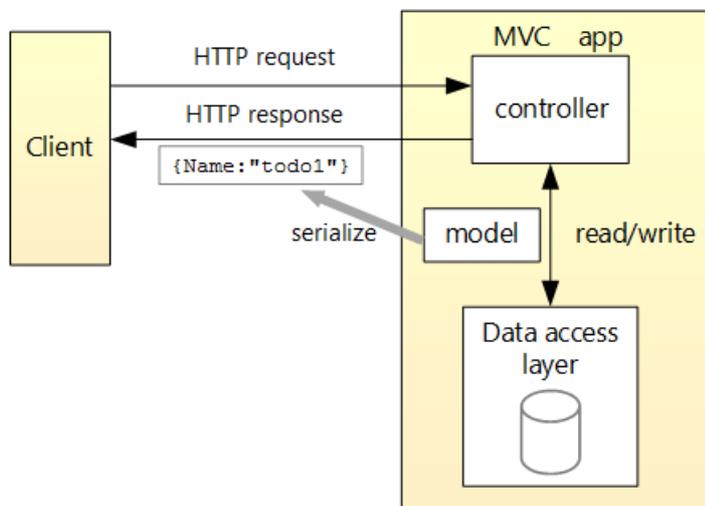
- Windows: Web API with Visual Studio for Windows (This tutorial)
- macOS: [Web API with Visual Studio for Mac](#)
- macOS, Linux, Windows: [Web API with Visual Studio Code](#)

Overview

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/todo	Get all to-do items	None	Array of to-do items
GET /api/todo/{id}	Get an item by ID	None	To-do item
POST /api/todo	Add a new item	To-do item	To-do item
PUT /api/todo/{id}	Update an existing item	To-do item	None
DELETE /api/todo/{id}	Delete an item	None	None

The following diagram shows the basic design of the app.



- The client is whatever consumes the web API (mobile app, browser, etc.). This tutorial doesn't create a client. [Postman](#) or [curl](#) is used as the client to test the app.

- A *model* is an object that represents the data in the app. In this case, the only model is a to-do item. Models are represented as C# classes, also known as **Plain Old C# Object** (POCOs).
- A *controller* is an object that handles HTTP requests and creates the HTTP response. This app has a single controller.
- To keep the tutorial simple, the app doesn't use a persistent database. The sample app stores to-do items in an in-memory database.

Prerequisites

Install the following:

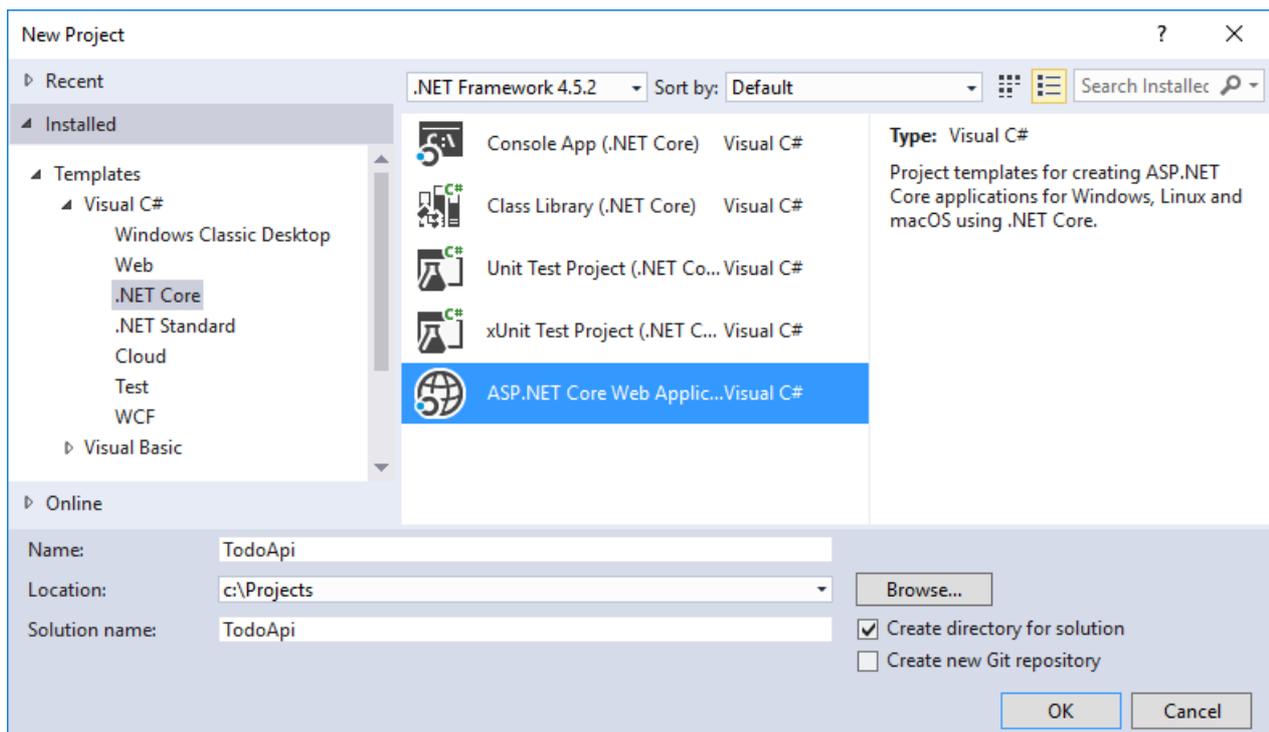
- [.NET Core 2.0.0 SDK](#) or later.
- [Visual Studio 2017](#) version 15.3 or later with the **ASP.NET and web development** workload.

See [this PDF](#) for the ASP.NET Core 1.1 version.

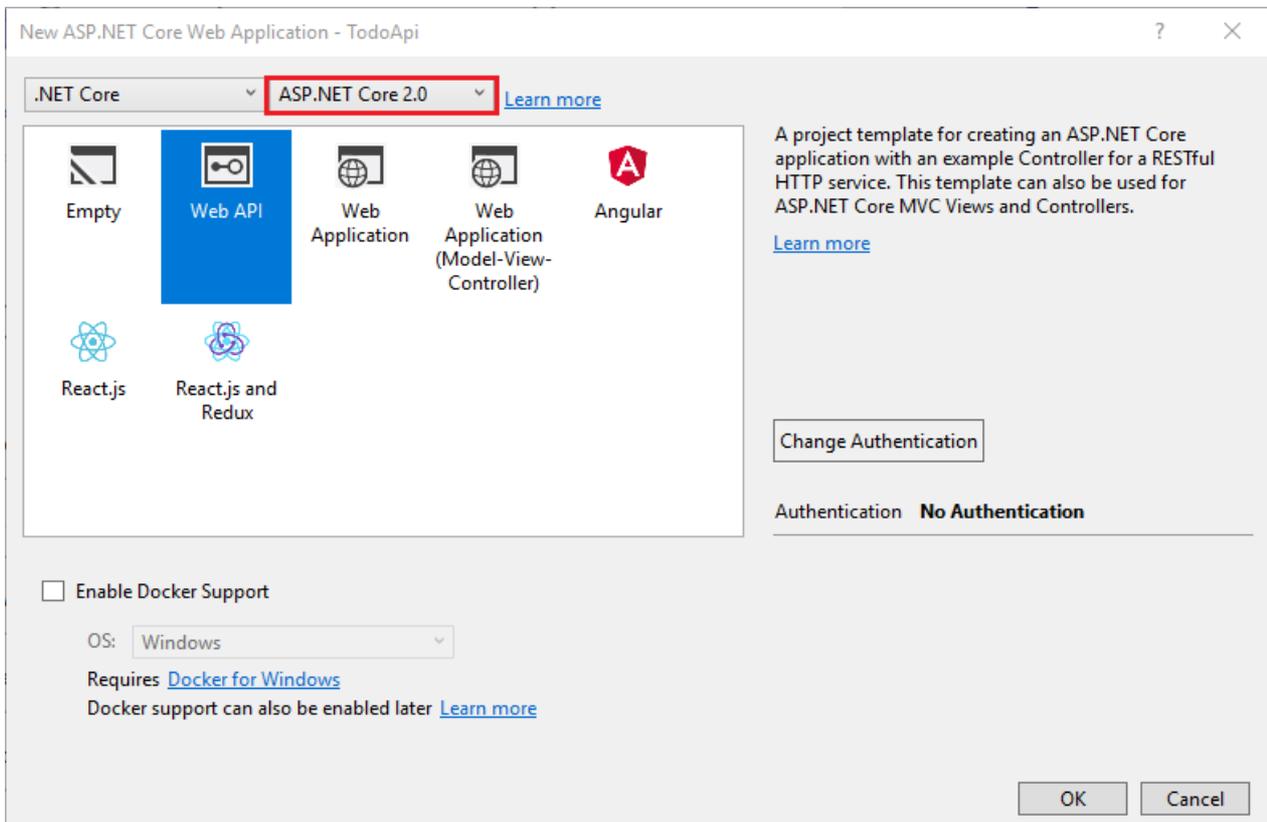
Create the project

From Visual Studio, select **File** menu, > **New > Project**.

Select the **ASP.NET Core Web Application (.NET Core)** project template. Name the project `TodoApi` and select **OK**.



In the **New ASP.NET Core Web Application - TodoApi** dialog, select the **Web API** template. Select **OK**. Do not select **Enable Docker Support**.



Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/values`, where *port* is a randomly chosen port number. Chrome, Microsoft Edge, and Firefox display the following output:

```
["value1", "value2"]
```

Add a model class

A model is an object that represents the data in the app. In this case, the only model is a to-do item.

Add a folder named "Models". In Solution Explorer, right-click the project. Select **Add > New Folder**. Name the folder *Models*.

Note: The model classes go anywhere in the project. The *Models* folder is used by convention for model classes.

Add a `TodoItem` class. Right-click the *Models* folder and select **Add > Class**. Name the class `TodoItem` and select **Add**.

Update the `TodoItem` class with the following code:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

The database generates the `Id` when a `TodoItem` is created.

Create the database context

The *database context* is the main class that coordinates Entity Framework functionality for a given data model. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

Add a `TodoContext` class. Right-click the *Models* folder and select **Add** > **Class**. Name the class `TodoContext` and select **Add**.

Replace the class with the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context

In this step, the database context is registered with the [dependency injection](#) container. Services (such as the DB context) that are registered with the dependency injection (DI) container are available to the controllers.

Register the DB context with the service container using the built-in support for [dependency injection](#). Replace the contents of the *Startup.cs* file with the following code:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

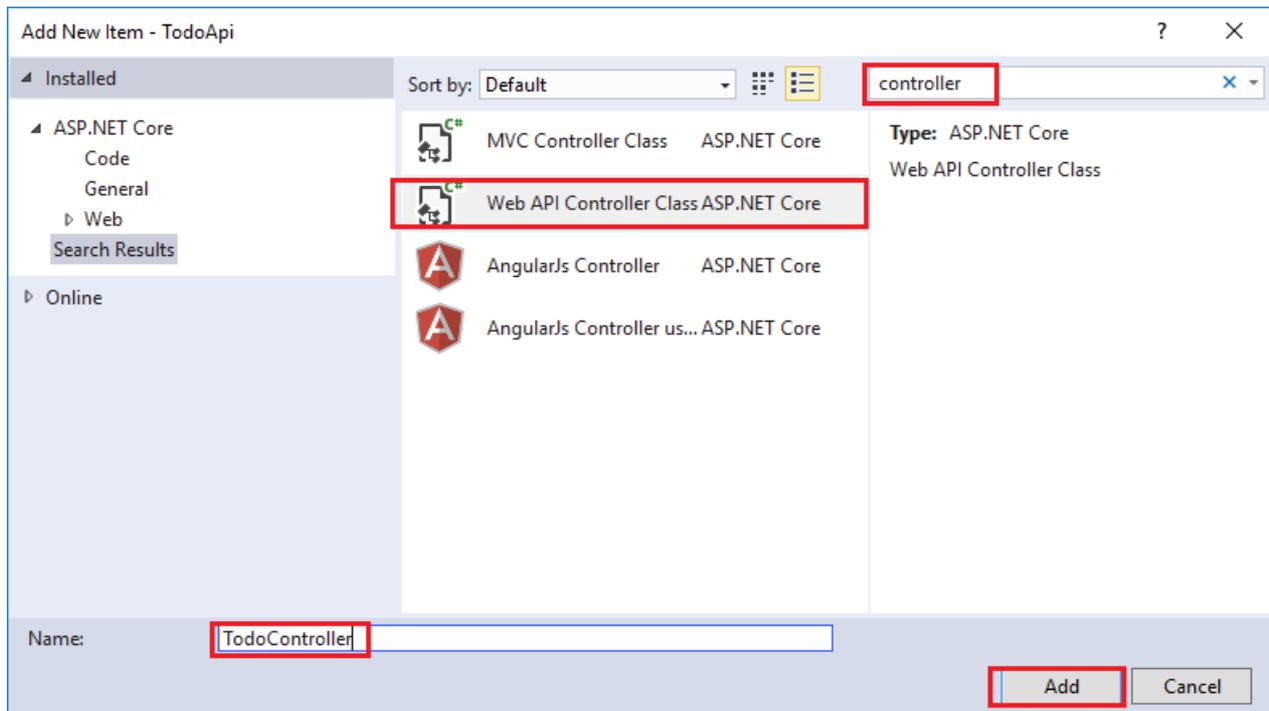
The preceding code:

- Removes the code that is not used.
- Specifies an in-memory database is injected into the service container.

Add a controller

In Solution Explorer, right-click the *Controllers* folder. Select **Add** > **New Item**. In the **Add New Item** dialog,

select the **Web API Controller Class** template. Name the class `TodoController`.



Replace the class with the following code:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using TodoApi.Models;
using System.Linq;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TODOItems.Count() == 0)
            {
                _context.TODOItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

- Defines an empty controller class. In the next sections, methods are added to implement the API.
- The constructor uses [Dependency Injection](#) to inject the database context (`TodoContext`) into the controller. The database context is used in each of the [CRUD](#) methods in the controller.
- The constructor adds an item to the in-memory database if one doesn't exist.

Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.

```

[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return _context.TODOItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}

```

These methods implement the two GET methods:

- GET /api/todo
- GET /api/todo/{id}

Here is an example HTTP response for the `GetAll` method:

```

[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]

```

Later in the tutorial I'll show how the HTTP response can be viewed with [Postman](#) or [curl](#).

Routing and URL paths

The `[HttpGet]` attribute specifies an HTTP GET method. The URL path for each method is constructed as follows:

- Take the template string in the controller's `Route` attribute:

```

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
    }
}

```

- Replace `[controller]` with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoController** and the root name is "todo". ASP.NET Core [routing](#) is not case sensitive.
- If the `[HttpGet]` attribute has a route template (such as `[HttpGet("/products")]`), append that to the path. This sample doesn't use a template. See [Attribute routing with Http\[Verb\] attributes](#) for more information.

In the `GetById` method:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

"{id}" is a placeholder variable for the ID of the `todo` item. When `GetById` is invoked, it assigns the value of "{id}" in the URL to the method's `id` parameter.

`Name = "GetTodo"` creates a named route. Named routes:

- Enable the app to create an HTTP link using the route name.
- Are explained later in the tutorial.

Return values

The `GetAll` method returns an `IEnumerable`. MVC automatically serializes the object to [JSON](#) and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

In contrast, the `GetById` method returns the more general `IActionResult` type, which represents a wide range of return types. `GetById` has two different return types:

- If no item matches the requested ID, the method returns a 404 error. Returning `NotFound` returns an HTTP 404 response.
- Otherwise, the method returns 200 with a JSON response body. Returning `ObjectResult` returns an HTTP 200 response.

Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to

`http://localhost:port/api/values`, where *port* is a randomly chosen port number. Navigate to the `Todo` controller at `http://localhost:port/api/todo`.

Implement the other CRUD operations

In the following sections, `Create`, `Update`, and `Delete` methods are added to the controller.

Create

Add the following `Create` method.

```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TODOItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

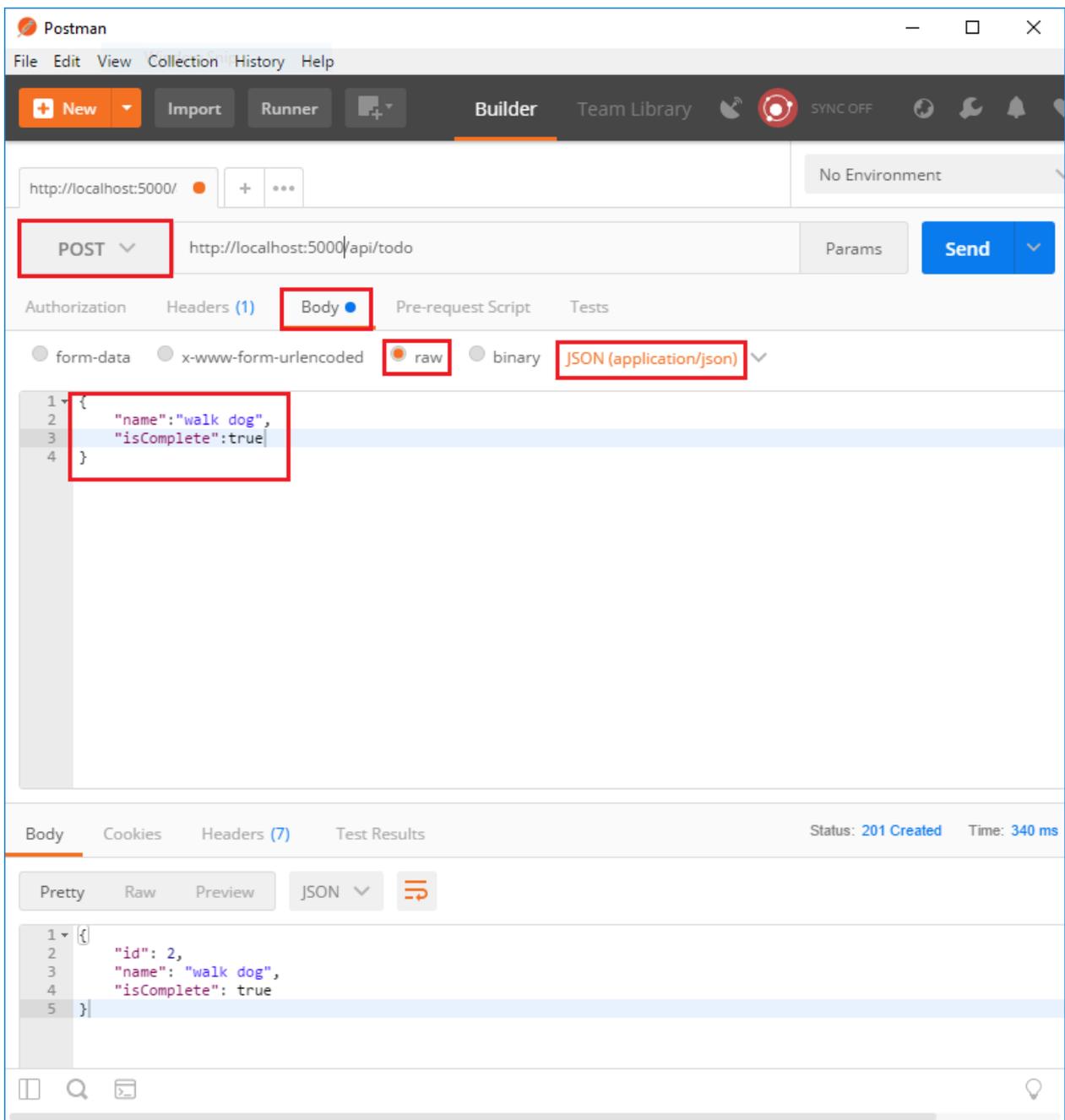
The preceding code is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method:

- Returns a 201 response. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See [10.2.2 201 Created](#).
- Uses the "GetTodo" named route to create the URL. The "GetTodo" named route is defined in `GetById`:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

Use Postman to send a Create request



- Set the HTTP method to `POST`
- Select the **Body** radio button
- Select the **raw** radio button
- Set the type to JSON
- In the key-value editor, enter a Todo item such as

```
{  
  "name": "walk dog",  
  "isComplete": true  
}
```

- Select **Send**
- Select the Headers tab in the lower pane and copy the **Location** header:

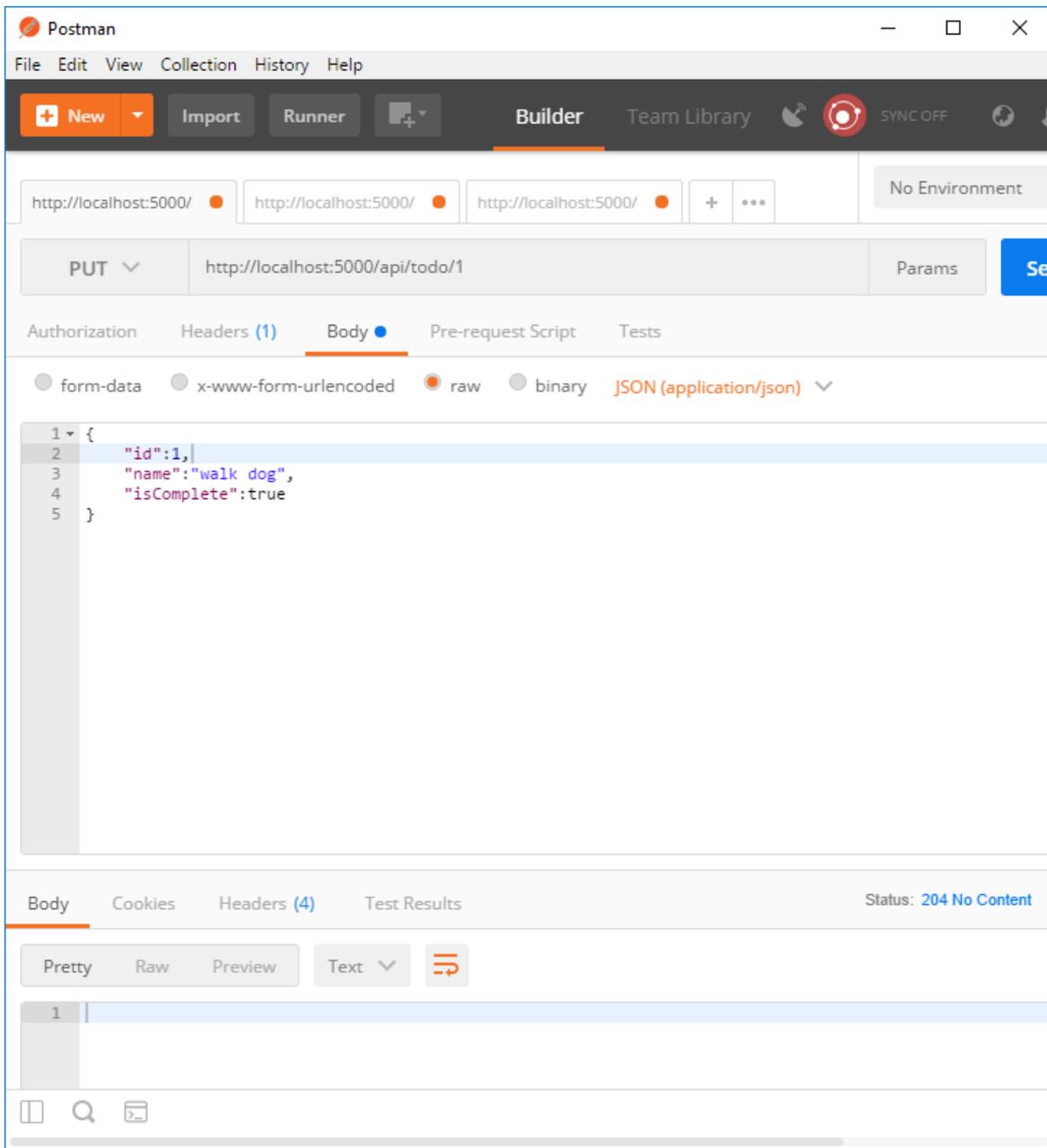

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] TodoItem item)
{
    if (item == null || item.Id != id)
    {
        return BadRequest();
    }

    var todo = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    todo.IsComplete = item.IsComplete;
    todo.Name = item.Name;

    _context.TODOItems.Update(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

`Update` is similar to `Create`, but uses HTTP PUT. The response is [204 \(No Content\)](#). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.



Delete

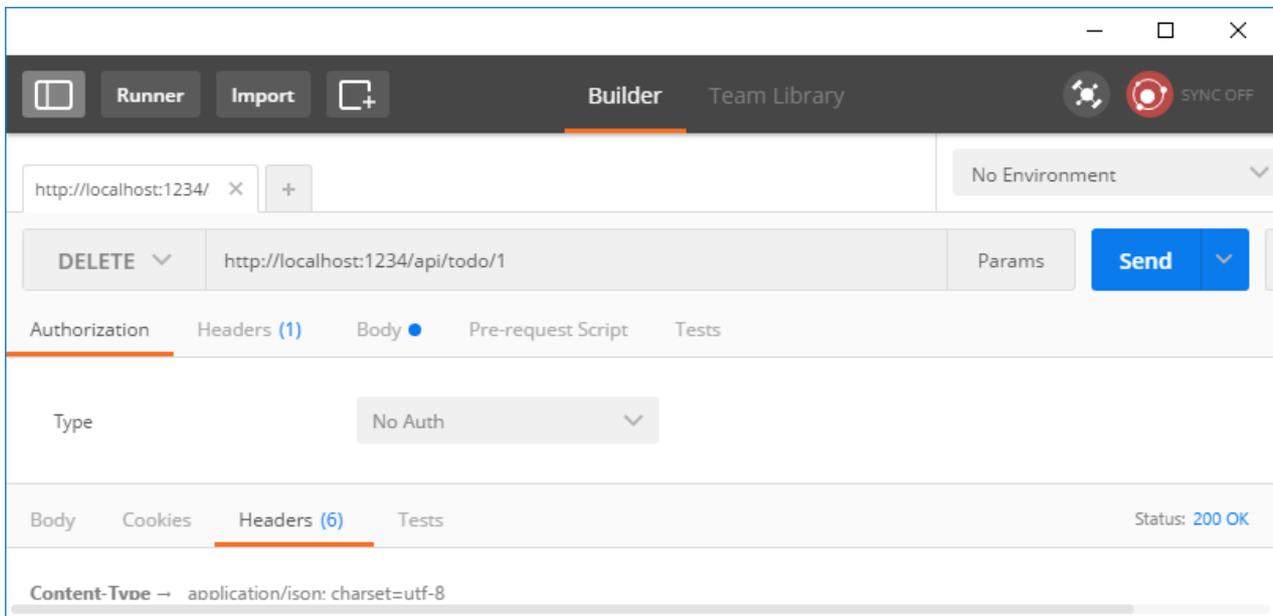
Add the following `Delete` method:

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TODOItems.Remove(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

The `Delete` response is [204 \(No Content\)](#).

Test `Delete` :



Next steps

- [ASP.NET Core Web API Help Pages using Swagger](#)
- [Routing to Controller Actions](#)
- For information about deploying an API, including to Azure App Service, see [Host and deploy](#).
- [View or download sample code](#). See [how to download](#).
- [Postman](#)

ASP.NET Core tutorials

1/10/2018 • 1 min to read • [Edit Online](#)

The following step-by-step guides for developing ASP.NET Core applications are available:

Build web apps

[Razor Pages](#) is the recommended approach to create a new Web UI app with ASP.NET Core 2.0.

- [Introduction to Razor Pages in ASP.NET Core](#)
- Create a Razor Pages web app with ASP.NET Core
 - [Razor Pages on Windows](#)
 - [Razor Pages on Mac](#)
 - [Razor Pages with VS Code](#)
- Create an ASP.NET Core MVC web app
 - [Web app with Visual Studio for Windows](#)
 - [Web app with Visual Studio for Mac](#)
 - [Web app with Visual Studio Code on Mac or Linux](#)
- [Get started with ASP.NET Core and Entity Framework Core using Visual Studio](#)
- [Create Tag Helpers](#)
- [Create a simple view component](#)
- [Develop ASP.NET Core apps using dotnet watch](#)

Build Web APIs

- Create a Web API with ASP.NET Core
 - [Web API with Visual Studio for Windows](#)
 - [Web API with Visual Studio for Mac](#)
 - [Web API with Visual Studio Code](#)
- [ASP.NET Core Web API help pages using Swagger](#)
- [Create backend web services for native mobile apps](#)

Data access and storage

- [Get started with ASP.NET Core and Entity Framework Core using Visual Studio](#)
- [ASP.NET Core with EF Core - new database](#)
- [ASP.NET Core with EF Core - existing database](#)

Authentication and authorization

- [Enable authentication using Facebook, Google, and other external providers](#)
- [Account confirmation and password recovery](#)
- [Two-factor authentication with SMS](#)

Client-side development

- [Use Gulp](#)
- [Use Grunt](#)
- [Manage client-side packages with Bower](#)
- [Build responsive sites with Bootstrap](#)

Test

- [Unit testing in .NET Core using dotnet test](#)

Publish and deploy

- [Deploy an ASP.NET Core web app to Azure using Visual Studio](#)
- [Deploy an ASP.NET Core web app to Azure using the command line](#)
- [Publish to an Azure Web App with continuous deployment](#)
- [Deploy an ASP.NET container to a remote Docker host](#)
- [ASP.NET Core on Nano Server](#)
- [ASP.NET Core and Azure Service Fabric](#)

How to download a sample

1. [Download the ASP.NET repository zip file.](#)
2. Unzip the *Docs-master.zip* file.
3. Use the URL in the sample link to help you navigate to the sample directory.

Create a Razor Pages web app with ASP.NET Core

9/30/2017 • 1 min to read • [Edit Online](#)

This series explains the basics of building a Razor Pages web app with ASP.NET Core using Visual Studio. For the Mac version, see [this](#). For the Visual Studio Code version, see [this](#).

1. [Getting started with Razor Pages](#)
2. [Adding a model to a Razor Pages app](#)
3. [Scaffolded Razor Pages](#)
4. [Working with SQL Server LocalDB](#)
5. [Updating the pages](#)
6. [Adding search](#)
7. [Adding a new field](#)
8. [Adding validation](#)
9. [Uploading files](#)

Get started with Razor Pages in ASP.NET Core

1/9/2018 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#)

This tutorial teaches the basics of building an ASP.NET Core Razor Pages web app. Razor Pages is the recommended way to build UI for web apps in ASP.NET Core.

There are three versions of this tutorial:

- Windows: This tutorial
- MacOS: [Getting started with Razor Pages with Visual Studio for Mac](#)
- macOS, Linux, and Windows: [Getting started with Razor Pages in ASP.NET Core with Visual Studio Code](#)

[View or download sample code \(how to download\)](#)

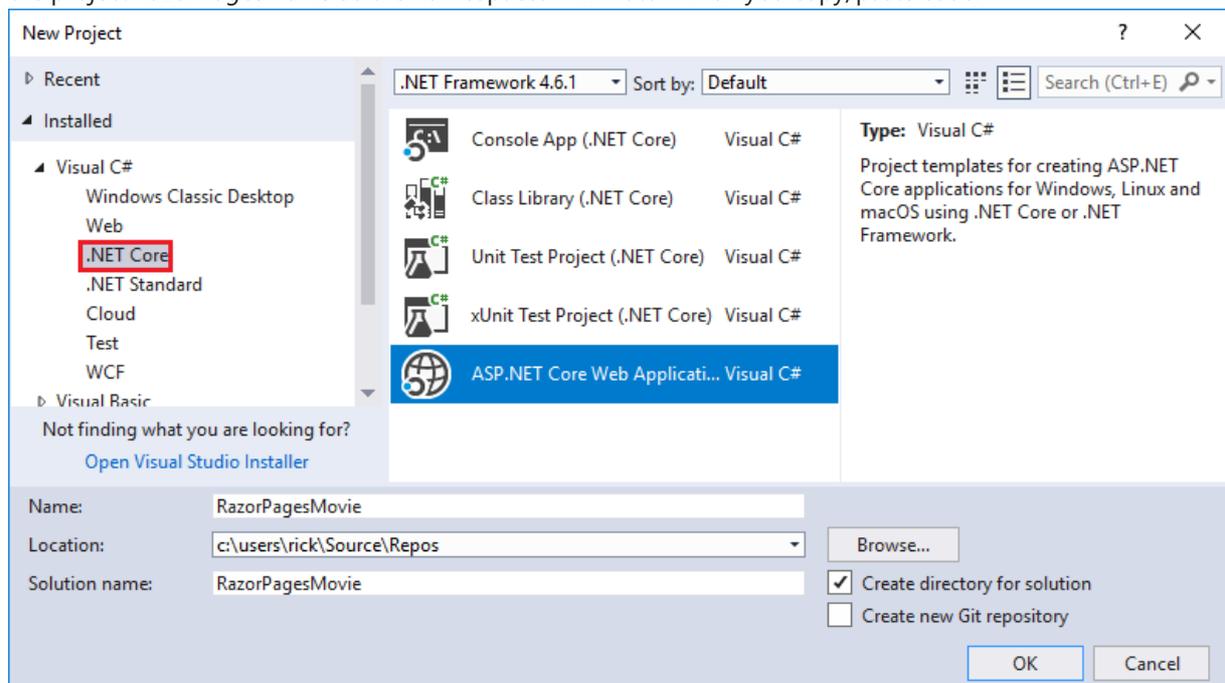
Prerequisites

Install the following:

- [.NET Core 2.0.0 SDK](#) or later.
- [Visual Studio 2017](#) version 15.3 or later with the **ASP.NET and web development** workload.

Create a Razor web app

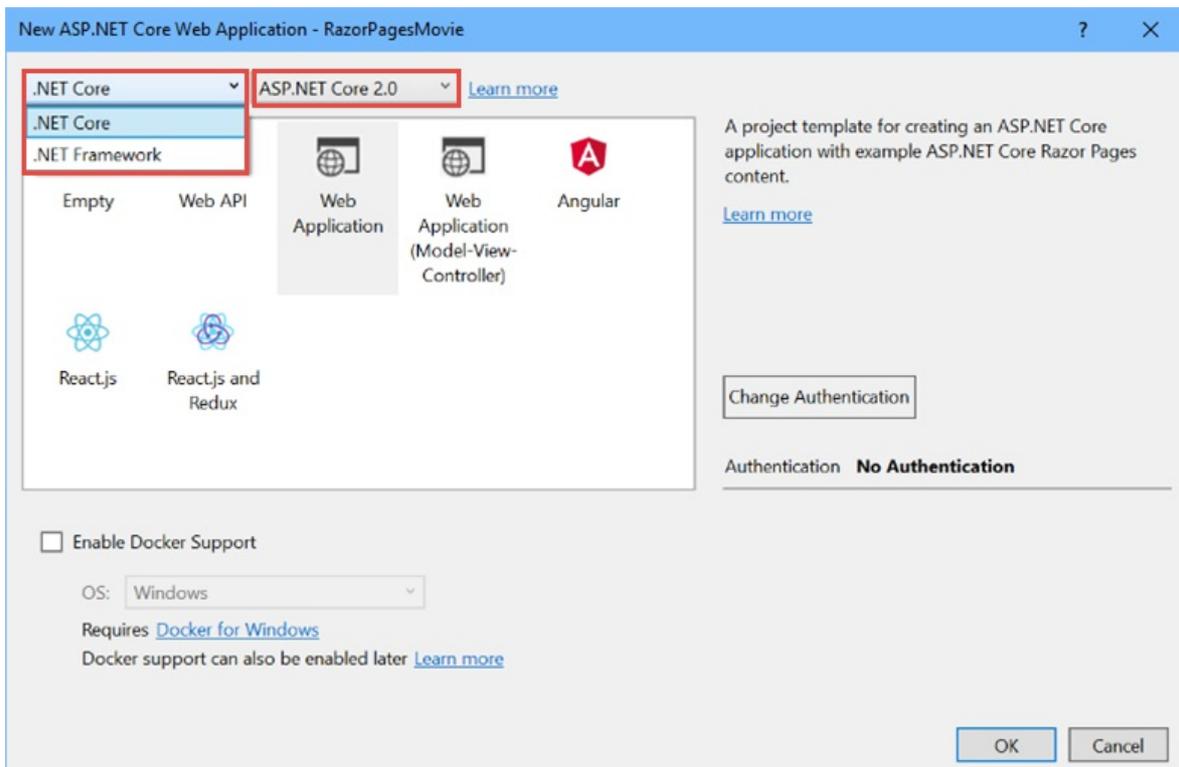
- From the Visual Studio **File** menu, select **New > Project**.
- Create a new ASP.NET Core Web Application. Name the project **RazorPagesMovie**. It's important to name the project *RazorPagesMovie* so the namespaces will match when you copy/paste code.



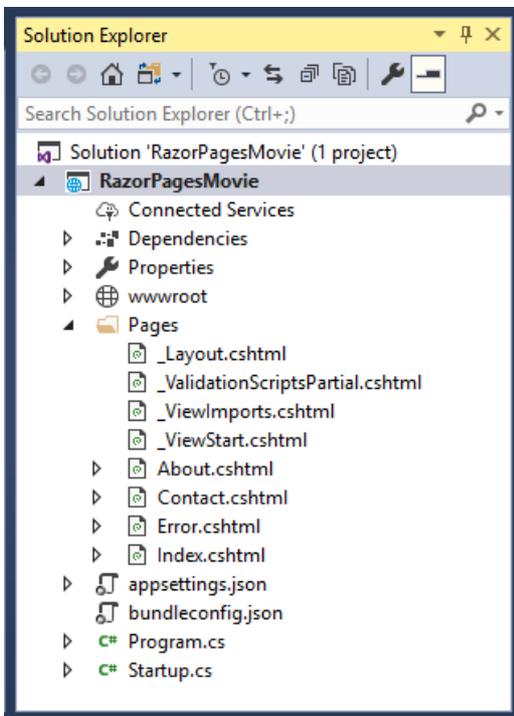
- Select **ASP.NET Core 2.0** in the dropdown, and then select **Web Application**.

NOTE

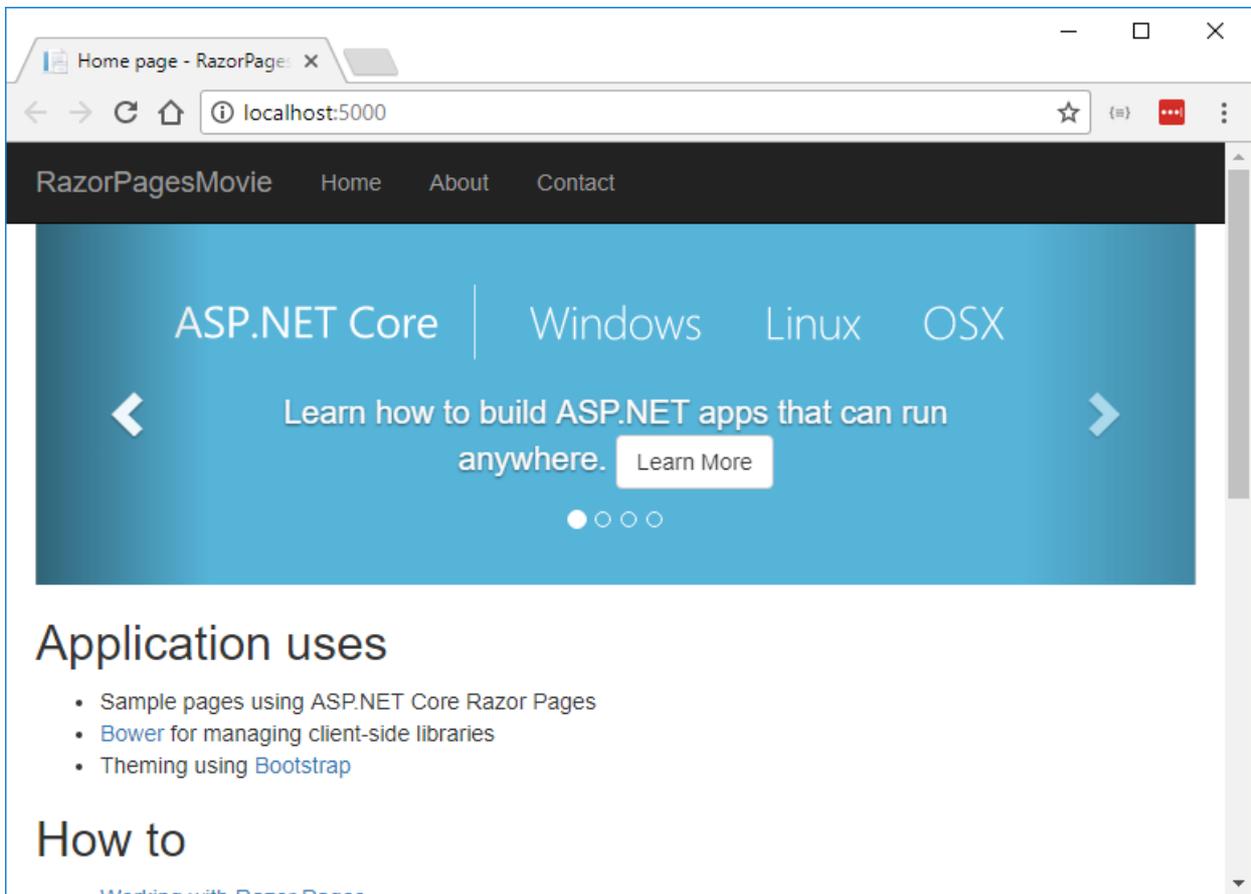
To use ASP.NET Core with .NET Framework, you must first select **.NET Framework** from the leftmost drop-down in the dialog, then you can select the desired ASP.NET Core version.



The Visual Studio template creates a starter project:

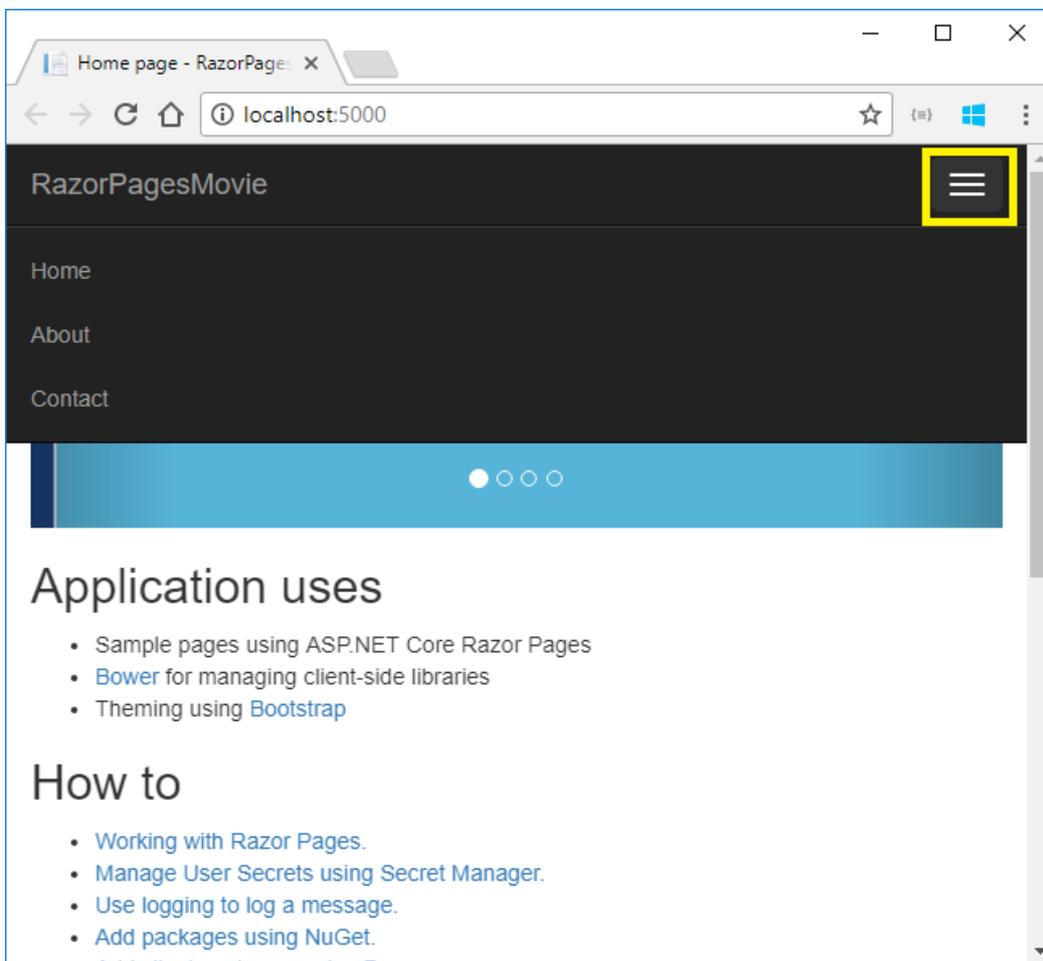


Press **F5** to run the app in debug mode or **Ctrl-F5** to run without attaching the debugger



- Visual Studio starts [IIS Express](#) and runs your app. The address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for your local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server. In the preceding image, the port number is 5000. When you run the app, you'll see a different port number.
- Launching the app with **Ctrl+F5** (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.

The default template creates **RazorPagesMovie**, **Home**, **About** and **Contact** links and pages. Depending on the size of your browser window, you might need to click the navigation icon to show the links.



Test the links. The **RazorPagesMovie** and **Home** links go to the Index page. The **About** and **Contact** links go to the `About` and `Contact` pages, respectively.

Project files and folders

The following table lists the files and folders in the project. For this tutorial, the `Startup.cs` file is the most important to understand. You don't need to review each link provided below. The links are provided as a reference when you need more information on a file or folder in the project.

FILE OR FOLDER	PURPOSE
<code>wwwroot</code>	Contains static files. See Working with static files .
<code>Pages</code>	Folder for Razor Pages .
<code>appsettings.json</code>	Configuration
<code>Program.cs</code>	Hosts the ASP.NET Core app.
<code>Startup.cs</code>	Configures services and the request pipeline. See Startup .

The Pages folder

The `_Layout.cshtml` file contains common HTML elements (scripts and stylesheets) and sets the layout for the application. For example, when you click on **RazorPagesMovie**, **Home**, **About** or **Contact**, you see the same elements. The common elements include the navigation menu on the top and the header on the bottom of the window. See [Layout](#) for more information.

The `_ViewStart.cshtml` sets the Razor Pages `Layout` property to use the `_Layout.cshtml` file. See [Layout](#) for more

information.

The `_ViewImports.cshtml` file contains Razor directives that are imported into each Razor Page. See [Importing Shared Directives](#) for more information.

The `_ValidationScriptsPartial.cshtml` file provides a reference to [jQuery](#) validation scripts. When we add `Create` and `Edit` pages later in the tutorial, the `_ValidationScriptsPartial.cshtml` file will be used.

The `About`, `Contact` and `Index` pages are basic pages you can use to start an app. The `Error` page is used to display error information.

NEXT: ADDING A
MODEL

NEXT: ADDING A
MODEL

Adding a model to a Razor Pages app

11/1/2017 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this section, you add classes for managing movies in a database. You use these classes with [Entity Framework Core](#) (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes you create are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They define the properties of the data that are stored in the database.

In this tutorial, you write the model classes first, and EF Core creates the database. An alternate approach not covered here is to [generate model classes from an existing database](#).

[View or download](#) sample.

Add a data model

In Solution Explorer, right-click the **RazorPagesMovie** project > **Add** > **New Folder**. Name the folder *Models*.

Right click the *Models* folder. Select **Add** > **Class**. Name the class **Movie** and add the following properties:

Add the following properties to the `Movie` class:

```
using System;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `ID` field is required by the database for the primary key.

Add a database context class

Add the following `DbContext` derived class named *MovieContext.cs* to the *Models* folder:

```

using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
    }
}

```

The preceding code creates a `DbSet` property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.

Add a database connection string

Add a connection string to the `appsettings.json` file.

```

{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "ConnectionStrings": {
    "MovieContext": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}

```

Register the database context

Register the database context with the [dependency injection](#) container in the `Startup.cs` file.

```

public void ConfigureServices(IServiceCollection services)
{
    // requires
    // using RazorPagesMovie.Models;
    // using Microsoft.EntityFrameworkCore;

    services.AddDbContext<MovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("MovieContext")));
    services.AddMvc();
}

```

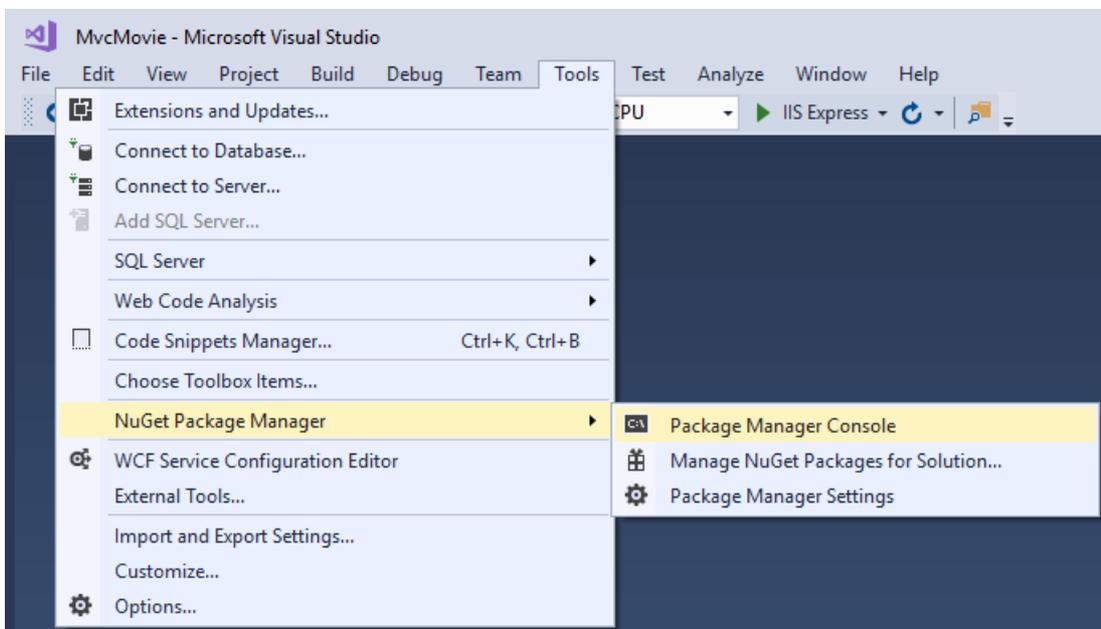
Build the project to verify you don't have any errors.

Add scaffold tooling and perform initial migration

In this section, you use the Package Manager Console (PMC) to:

- Add the Visual Studio web code generation package. This package is required to run the scaffolding engine.
- Add an initial migration.
- Update the database with the initial migration.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.



In the PMC, enter the following commands:

```
Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design -Version 2.0.0
Add-Migration Initial
Update-Database
```

The `Install-Package` command installs the tooling required to run the scaffolding engine.

The `Add-Migration` command generates code to create the initial database schema. The schema is based on the model specified in the `DbContext` (In the `Models/MovieContext.cs` file). The `Initial` argument is used to name the migrations. You can use any name, but by convention you choose a name that describes the migration. See [Introduction to migrations](#) for more information.

The `Update-Database` command runs the `Up` method in the `Migrations/<time-stamp>_InitialCreate.cs` file, which creates the database.

Scaffold the Movie model

- Open a command window in the project directory (The directory that contains the `Program.cs`, `Startup.cs`, and `.csproj` files).
- Run the following command:

```
dotnet aspnet-codegenerator razorpage -m Movie -dc MovieContext -udl -outDir Pages\Movies --
referenceScriptLibraries
```

If you get the error:

```
No executable found matching command "dotnet-aspnet-codegenerator"
```

Open a command window in the project directory (The directory that contains the `Program.cs`, `Startup.cs`, and `.csproj` files).

If you get the error:

```
The process cannot access the file
'RazorPagesMovie/bin/Debug/netcoreapp2.0/RazorPagesMovie.dll'
because it is being used by another process.
```

Exit Visual Studio and run the command again.

The following table details the ASP.NET Core code generators` parameters:

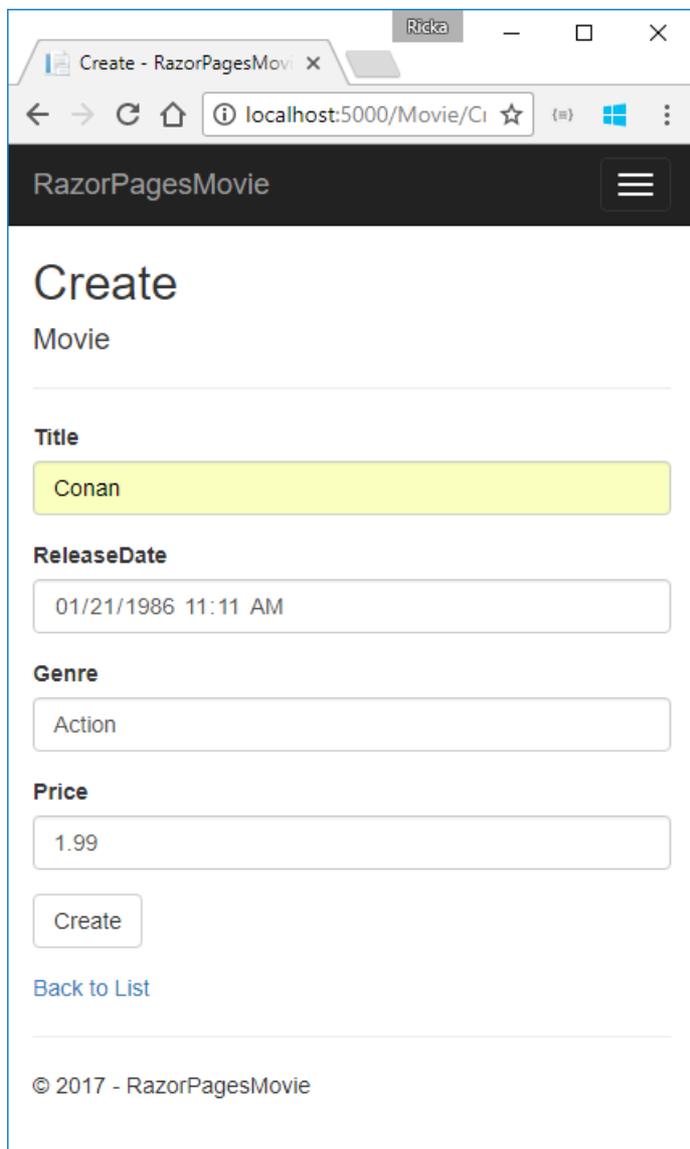
PARAMETER	DESCRIPTION
-m	The name of the model.
-dc	The data context.
-udl	Use the default layout.
-outDir	The relative output folder path to create the views.
--referenceScriptLibraries	Adds <code>_ValidationScriptsPartial</code> to Edit and Create pages

Use the `h` switch to get help on the `aspnet-codegenerator razorpage` command:

```
dotnet aspnet-codegenerator razorpage -h
```

Test the app

- Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).
- Test the **Create** link.



- Test the **Edit**, **Details**, and **Delete** links.

If you get a SQL exception, verify you have run migrations and updated the database:

The next tutorial explains the files created by scaffolding.

PREVIOUS: GETTING
STARTED

NEXT: SCAFFOLDED RAZOR
PAGES

Scaffolded Razor Pages in ASP.NET Core

12/13/2017 • 6 min to read • [Edit Online](#)

By [Rick Anderson](#)

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial topic [Adding a model](#).

[View or download sample](#).

The Create, Delete, Details, and Edit pages.

Examine the *Pages/Movies/Index.cshtml.cs* code-behind file:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public IndexModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get;set; }

        public async Task OnGetAsync()
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor Pages are derived from `PageModel`. By convention, the `PageModel`-derived class is called `<PageName>Model`. The constructor uses [dependency injection](#) to add the `MovieContext` to the page. All the scaffolded pages follow this pattern. See [Asynchronous code](#) for more information on asynchronous programming with Entity Framework.

When a request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. `OnGetAsync` or `OnGet` is called on a Razor Page to initialize the state for the page. In this case, `OnGetAsync` gets a list of movies to display.

Examine the *Pages/Movies/Index.cshtml* Razor Page:

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a [Razor reserved keyword](#), it transitions into Razor-specific markup, otherwise it transitions into C#.

The `@page` Razor directive makes the file into an MVC action — which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` is an example of transitioning into Razor-specific markup. See [Razor syntax](#) for more information.

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movie[0].Title))
```

The `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` are `null` or empty. When the lambda expression is evaluated (for example, with `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

The @model directive

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel`-derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayName` [HTML Helpers](#) on the page.

ViewData and layout

Consider the following code:

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel  
  
@{  
    ViewData["Title"] = "Index";  
}
```

The preceding highlighted code is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class has a `ViewData` dictionary property that can be used to add data that you want to pass to a View. You add objects into the `ViewData` dictionary using a key/value pattern. In the preceding sample, the "Title" property is added to the `ViewData` dictionary. The "Title" property is used in the `Pages/_Layout.cshtml` file. The following markup shows the first few lines of the `Pages/_Layout.cshtml` file.

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
    <title>@ViewData["Title"] - RazorPagesMovie</title>  
  
    @*Markup removed for brevity.*@
```

The line `@*Markup removed for brevity.*@` is a Razor comment. Unlike HTML comments (`<!-- -->`), Razor comments are not sent to the client.

Run the app and test the links in the project (**Home**, **About**, **Contact**, **Create**, **Edit**, and **Delete**). Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark. `Pages/Index.cshtml` and `Pages/Movies/Index.cshtml` currently have the same title, but you can modify them to have different values.

The `Layout` property is set in the `Pages/_ViewStart.cshtml` file:

```
@{
    Layout = "_Layout";
}
```

The preceding markup sets the layout file to *Pages/_Layout.cshtml* for all Razor files under the *Pages* folder. See [Layout](#) for more information.

Update the layout

Change the `<title>` element in the *Pages/_Layout.cshtml* file to use a shorter string.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie</title>
```

Find the following anchor element in the *Pages/_Layout.cshtml* file.

```
<a asp-page="/Index" class="navbar-brand">RazorPagesMovie</a>
```

Replace the preceding element with the following markup.

```
<a asp-page="/Movies/Index" class="navbar-brand">RpMovie</a>
```

The preceding anchor element is a [Tag Helper](#). In this case, it's the [Anchor Tag Helper](#). The

`asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page.

Save your changes, and test the app by clicking on the **RpMovie** link. See the [_Layout.cshtml](#) file in GitHub.

The Create code-behind page

Examine the *Pages/Movies/Create.cshtml.cs* code-behind file:

```

// Unused usings removed.
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public CreateModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}

```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize.

The `Page` method creates a `PageResult` object that renders the `Create.cshtml` page.

The `Movie` property uses the `[BindProperty]` attribute to opt-in to [model binding](#). When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date

field that cannot be converted to a date. We'll talk more about client-side validation and model validation later in the tutorial.

If there are no model errors, the data is saved, and the browser is redirected to the Index page.

The Create Razor Page

Examine the *Pages/Movies/Create.cshtml* Razor Page file:

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Visual Studio displays the `<form method="post">` tag in a distinctive font used for Tag Helpers. The `<form method="post">` element is a [Form Tag Helper](#). The Form Tag Helper automatically includes an [antiforgery token](#).

```

1  @page
2  @model RazorPagesMovie.Pages_Movie.CreateModel
3
4  @{
5      ViewData["Title"] = "Create";
6  }
7
8  <h2>Create</h2>
9
10 <h4>Movie</h4>
11 <hr />
12 <div class="row">
13     <div class="col-md-4">
14         <form method="post">
15
16             The form element represents a collection of form-associated elements, some of which can represent editable
17             values that can be submitted to a server for processing.
18
19             Learn more (F1)
20
21             Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper
22             Microsoft.AspNetCore.Razor.TagHelpers.ITagHelper implementation targeting <form> elements.
23
24             <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
25             </div>
26             <div class="form-group">
27                 <label asp-for="Movie.Genre" class="control-label"></label>
28                 <input asp-for="Movie.Genre" class="form-control" />
29                 <span asp-validation-for="Movie.Genre" class="text-danger"></span>
30             </div>
31             <div class="form-group">
32                 <label asp-for="Movie.Price" class="control-label"></label>
33                 <input asp-for="Movie.Price" class="form-control" />
34                 <span asp-validation-for="Movie.Price" class="text-danger"></span>
35             </div>
36             <div class="form-group">
37                 <input type="submit" value="Create" class="btn btn-default" />
38             </div>
39         </form>
40     </div>

```

The scaffolding engine creates Razor markup for each field in the model (except the ID) similar to the following:

```

<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>

```

The [Validation Tag Helpers](#) (`<div asp-validation-summary>` and ``) display validation errors. Validation is covered in more detail later in this series.

The [Label Tag Helper](#) (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `for` attribute for the `Title` property.

The [Input Tag Helper](#) (`<input asp-for="Movie.Title" class="form-control" />`) uses the [DataAnnotations](#) attributes and produces HTML attributes needed for jQuery Validation on the client-side.

The next tutorial explains SQL Server LocalDB and seeding the database.

Working with SQL Server LocalDB and ASP.NET Core

11/29/2017 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Joe Audette](#)

The `MovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the [Dependency Injection](#) container in the `ConfigureServices` method in the `Startup.cs` file:

```
public void ConfigureServices(IServiceCollection services)
{
    // requires
    // using RazorPagesMovie.Models;
    // using Microsoft.EntityFrameworkCore;

    services.AddDbContext<MovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("MovieContext")));
    services.AddMvc();
}
```

The ASP.NET Core [Configuration](#) system reads the `ConnectionString`. For local development, it gets the connection string from the `appsettings.json` file:

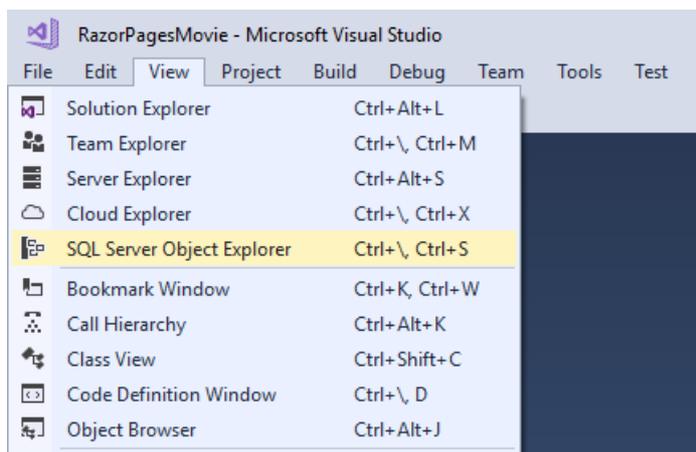
```
"ConnectionStrings": {
  "MovieContext": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

When you deploy the app to a test or production server, you can use an environment variable or another approach to set the connection string to a real SQL Server. See [Configuration](#) for more information.

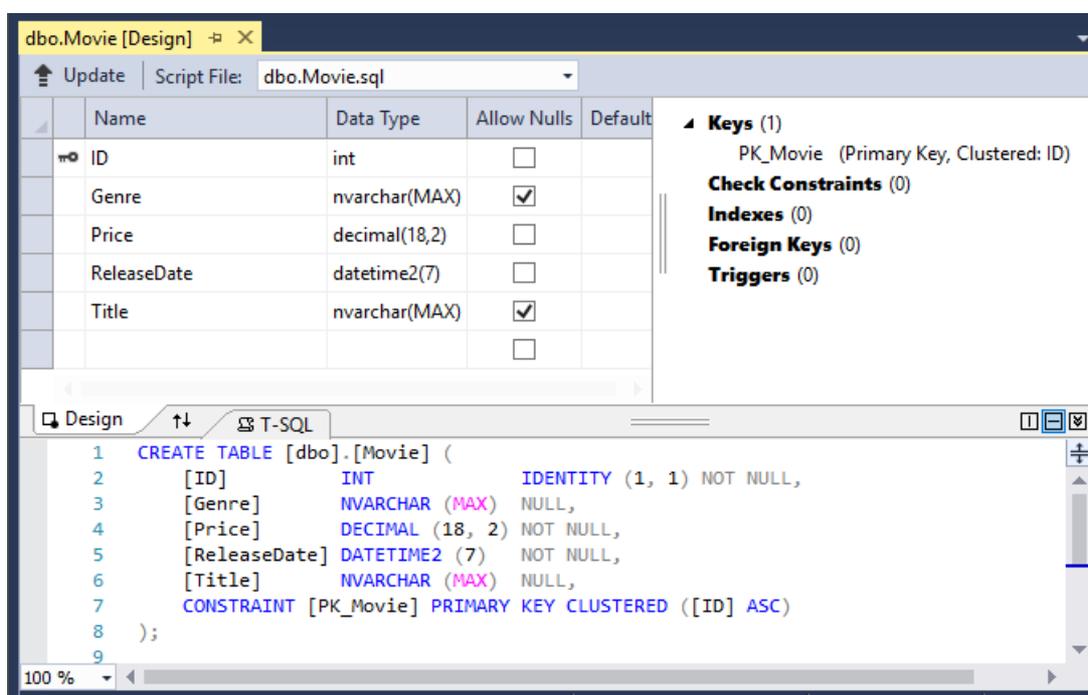
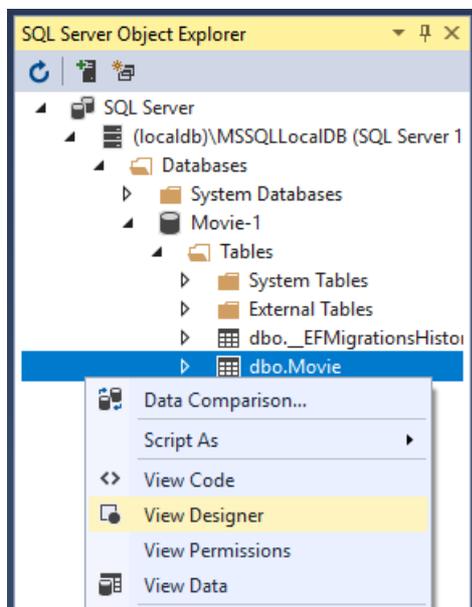
SQL Server Express LocalDB

LocalDB is a lightweight version of the SQL Server Express Database Engine that is targeted for program development. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:/Users/<user>` directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).

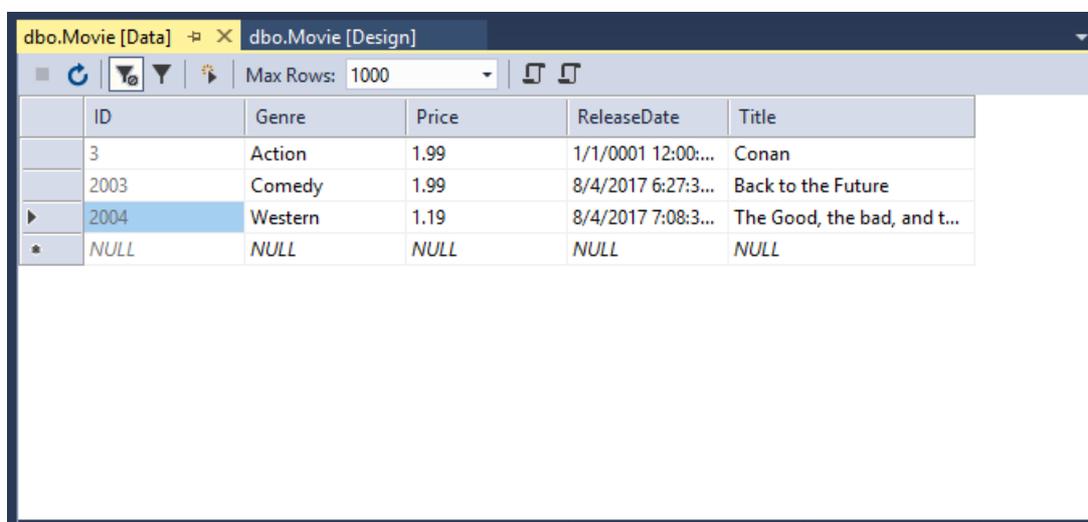


- Right click on the `Movie` table and select **View Designer**:



Note the key icon next to ID. By default, EF creates a property named ID for the primary key.

- Right click on the Movie table and select **View Data**:



Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

Add the seed initializer

Add the seed initializer to the end of the `Main` method in the `Program.cs` file:

```
// Unused usings removed.
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

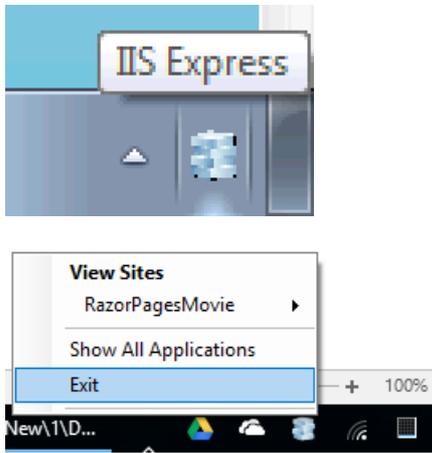
                try
                {
                    var context = services.GetRequiredService<MovieContext>();
                    // requires using Microsoft.EntityFrameworkCore;
                    context.Database.Migrate();
                    // Requires using RazorPagesMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

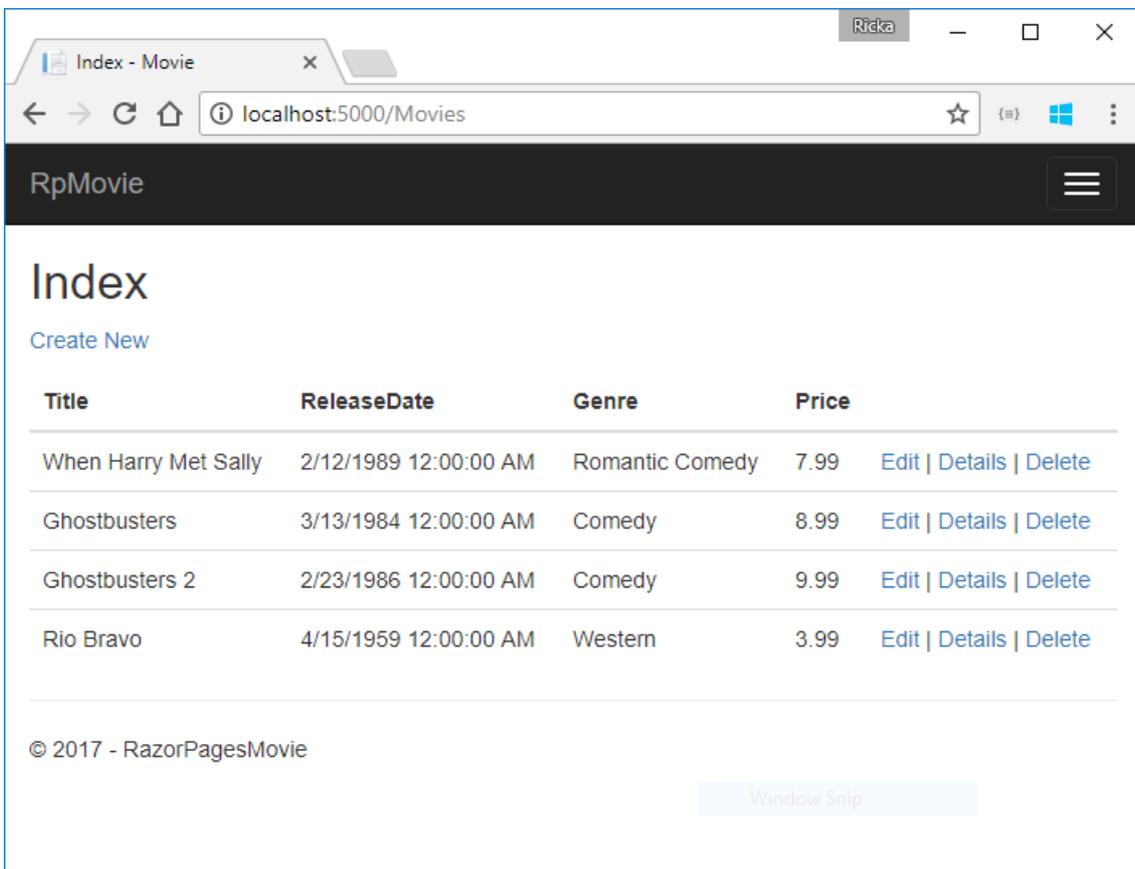
Test the app

- Delete all the records in the DB. You can do this with the delete links in the browser or from [SSOX](#)
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:
 - Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**:



- If you were running VS in non-debug mode, press F5 to run in debug mode.
- If you were running VS in debug mode, stop the debugger and press F5.

The app shows the seeded data:



The next tutorial will clean up the presentation of the data.

PREVIOUS: SCAFFOLDED RAZOR
PAGES

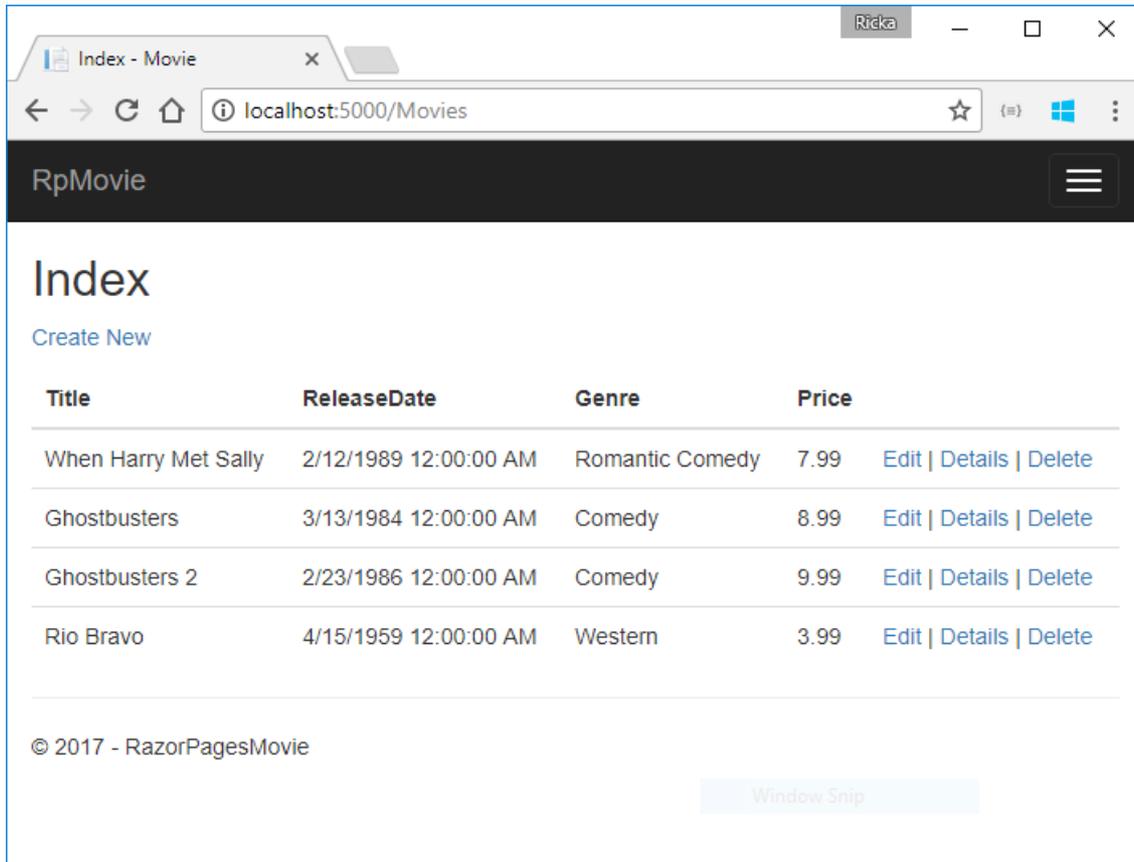
NEXT: UPDATING THE
PAGES

Updating the generated pages

12/6/2017 • 4 min to read • [Edit Online](#)

By [Rick Anderson](#)

We have a good start to the movie app, but the presentation is not ideal. We don't want to see the time (12:00:00 AM in the image below) and **ReleaseDate** should be **Release Date** (two words).



Update the generated code

Open the `Models/Movie.cs` file and add the highlighted lines shown in the following code:

```
using System;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

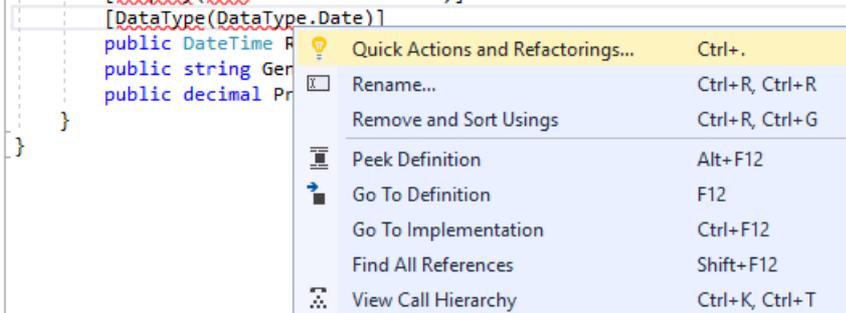
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

Right click on a red squiggly line > **Quick Actions and Refactorings**.

```
using System;

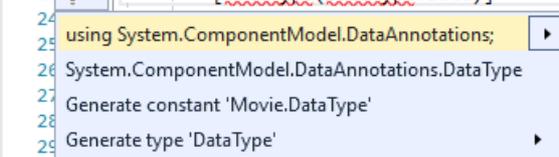
namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```



Select `using System.ComponentModel.DataAnnotations;`

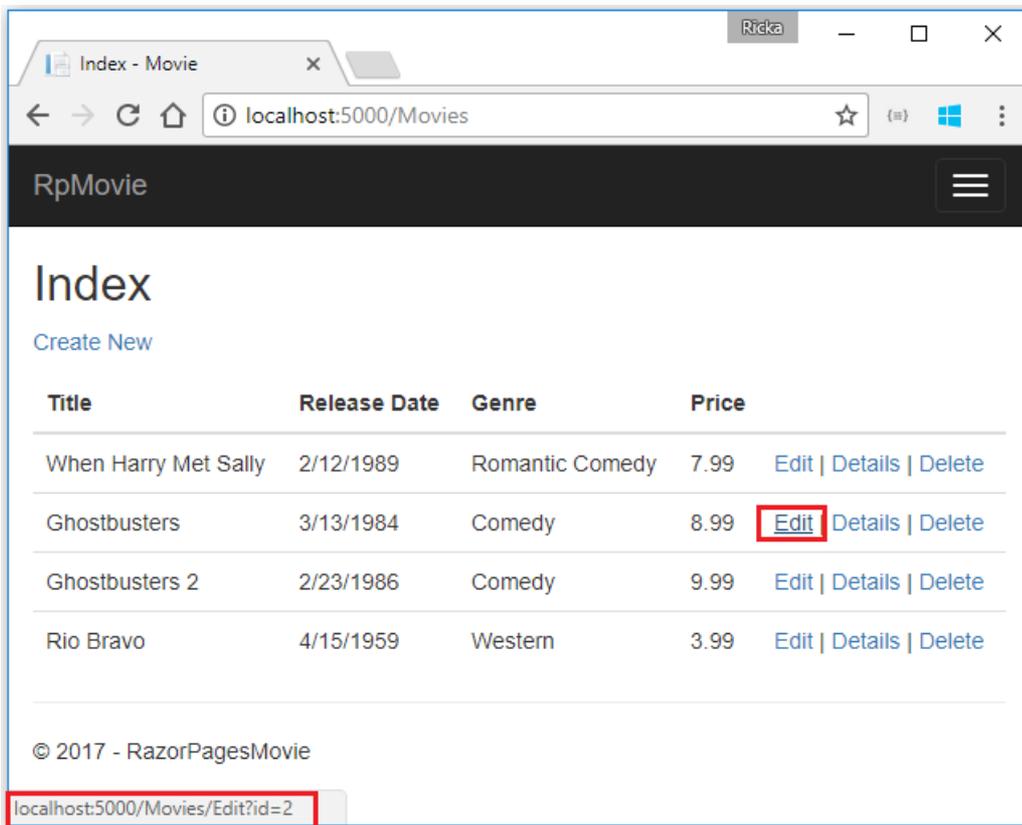
```
13 using System;
14
15 namespace RazorPagesMovie.Models
16 {
17     public class Movie
18     {
19         public int ID { get; set; }
20         public string Title { get; set; }
21
22         [Display(Name = "Release Date")]
23         [DataType(DataType.Date)]
24         public DateTime ReleaseDate { get; set; }
25         public string Genre { get; set; }
26         public decimal Price { get; set; }
27     }
28 }
29
```



Visual studio adds `using System.ComponentModel.DataAnnotations;`

We'll cover [DataAnnotations](#) in the next tutorial. The [Display](#) attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The [DataType](#) attribute specifies the type of the data (Date), so the time information stored in the field is not displayed.

Browse to Pages/Movies and hover over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the [Anchor Tag Helper](#) in the `Pages/Movies/Index.cshtml` file.

```
@foreach (var item in Model.Movie) {  
    <tr>  
        <td>  
            @Html.DisplayFor(modelItem => item.Title)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.ReleaseDate)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Genre)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Price)  
        </td>  
        <td>  
            <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |  
            <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |  
            <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>  
        </td>  
    </tr>  
}  
</tbody>  
</table>
```

[Tag Helpers](#) enable server-side code to participate in creating and rendering HTML elements in Razor files. In the preceding code, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route id (`asp-route-id`). See [URL generation for Pages](#) for more information.

Use **View Source** from your favorite browser to examine the generated markup. A portion of the generated HTML is shown below:

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically-generated links pass the movie ID with a query string (for example,

```
http://localhost:5000/Movies/Details?id=2 ).
```

Update the Edit, Details, and Delete Razor Pages to use the "{id:int}" route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`. Run the app and then view source. The generated HTML adds the ID to the path portion of the URL:

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the "{id:int}" route template that does **not** include the integer will return an HTTP 404 (not found) error. For example, `http://localhost:5000/Movies/Details` will return a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

Update concurrency exception handling

Update the `OnPostAsync` method in the `Pages/Movies/Edit.cshtml.cs` file. The following highlighted code shows the changes:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!_context.Movie.Any(e => e.ID == Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("../Index");
}
```

The previous code only detects concurrency exceptions when the first concurrent client deletes the movie, and the

second concurrent client posts changes to the movie.

To test the `catch` block:

- Set a breakpoint on `catch (DbUpdateConcurrencyException)`
- Edit a movie.
- In another browser window, select the **Delete** link for the same movie, and then delete the movie.
- In the previous browser window, post changes to the movie.

Production code would generally detect concurrency conflicts when two or more clients concurrently updated a record. See [Handling concurrency conflicts](#) for more information.

Posting and binding review

Examine the *Pages/Movies/Edit.cshtml.cs* file:

```

public class EditModel : PageModel
{
    private readonly RazorPagesMovie.Models.MovieContext _context;

    public EditModel(RazorPagesMovie.Models.MovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);

        if (Movie == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_context.Movie.Any(e => e.ID == Movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return RedirectToPage("../Index");
    }
}

```

When an HTTP GET request is made to the Movies/Edit page (for example, <http://localhost:5000/Movies/Edit/2>):

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the `Pages/Movies/Edit.cshtml` Razor Page. The `Pages/Movies/Edit.cshtml` file contains the model directive (`@model RazorPagesMovie.Pages.Movies.EditModel`), which makes the movie model available on the page.
- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables [Model binding](#).

```
[BindProperty]  
public Movie Movie { get; set; }
```

- If there are errors in the model state (for example, `ReleaseDate` cannot be converted to a date), the form is posted again with the submitted values.
- If there are no model errors, the movie is saved.

The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Search is added in the next tutorial.

PREVIOUS: WORKING WITH SQL SERVER
LOCALDB

ADDING
SEARCH

Adding search to a Razor Pages app

1/8/2018 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this document, search capability is added to the Index page that enables searching movies by *genre* or *name*.

Update the Index page's `OnGetAsync` method with the following code:

```
public async Task OnGetAsync(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a [LINQ](#) query to select the movies:

```
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

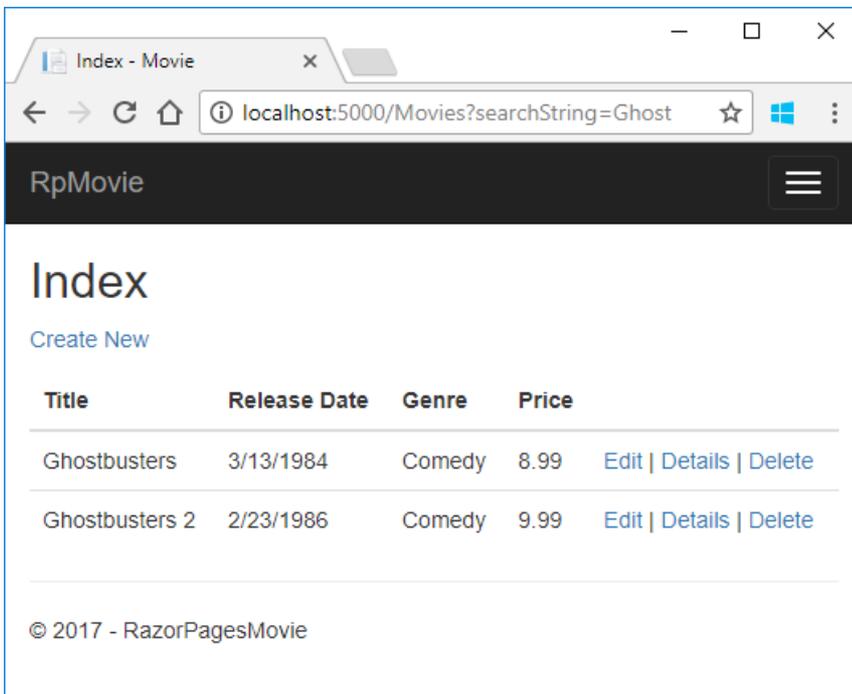
If the `searchString` parameter contains a string, the movies query is modified to filter on the search string:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

The `s => s.Title.Contains()` code is a [Lambda Expression](#). Lambdas are used in method-based [LINQ](#) queries as arguments to standard query operator methods such as the [Where](#) method or `Contains` (used in the preceding code). LINQ queries are not executed when they are defined or when they are modified by calling a method (such as `Where`, `Contains` or `OrderBy`). Rather, query execution is deferred. That means the evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See [Query Execution](#) for more information.

Note: The `Contains` method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to [SQL LIKE](#), which is case insensitive. In SQLite, with the default collation, it's case sensitive.

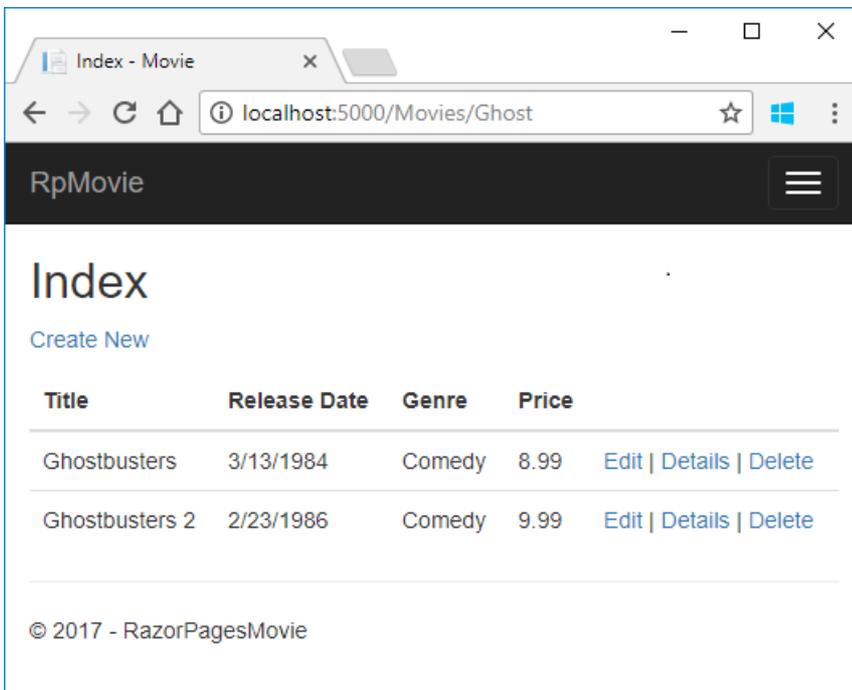
Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL (for example, `http://localhost:5000/Movies?searchString=Ghost`). The filtered movies are displayed.



If the following route template is added to the Index page, the search string can be passed as a URL segment (for example, `http://localhost:5000/Movies/ghost`).

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



However, you can't expect users to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the `Pages/Movies/Index.cshtml` file, and add the `<form>` markup highlighted in the following code:

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

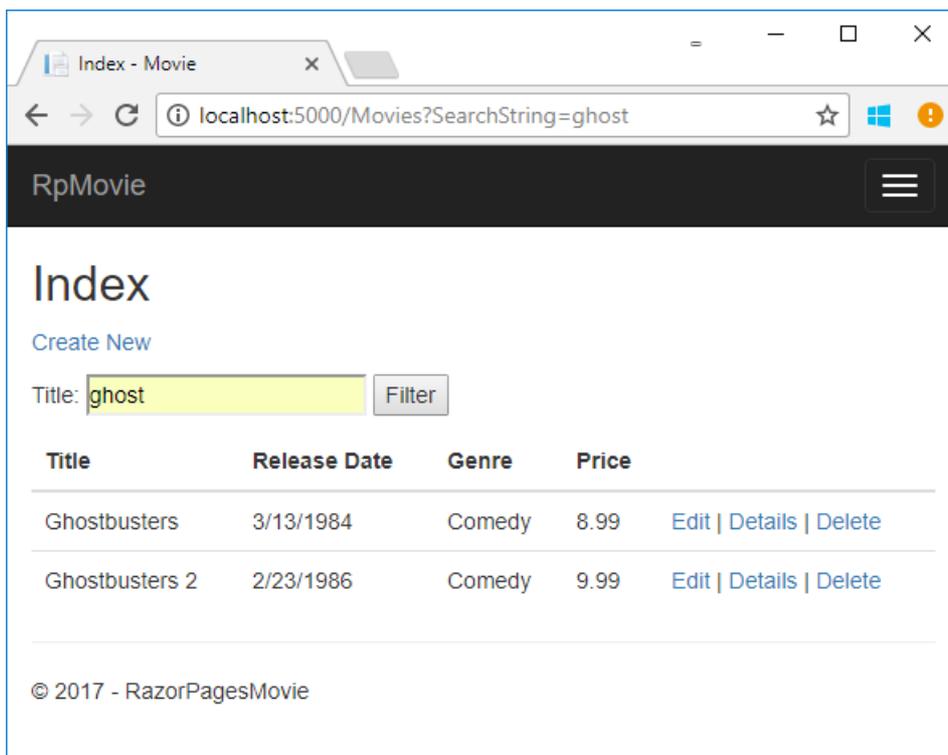
<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
@*Markup removed for brevity.*@

```

The HTML `<form>` tag uses the [Form Tag Helper](#). When the form is submitted, the filter string is sent to the `Pages/Movies/Index` page. Save the changes and test the filter.



Search by genre

Add the the following highlighted properties to `Pages/Movies/Index.cshtml.cs`:

```

public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Models.MovieContext _context;

    public IndexModel(RazorPagesMovie.Models.MovieContext context)
    {
        _context = context;
    }

    public List<Movie> Movie;
    public SelectList Genres;
    public string MovieGenre { get; set; }
}

```

The `SelectList Genres` contains the list of genres. This allows the user to select a genre from the list.

The `MovieGenre` property contains the specific genre the user selects (for example, "Western").

Update the `OnGetAsync` method with the following code:

```

// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task OnGetAsync(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}

```

The following code is a LINQ query that retrieves all the genres from the database.

```

// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;

```

The `SelectList` of genres is created by projecting the distinct genres.

```

Genres = new SelectList(await genreQuery.Distinct().ToListAsync());

```

Adding search by genre

Update *Index.cshtml* as follows:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

Test the app by searching by genre, by movie title, and by both.

PREVIOUS: UPDATING THE
PAGES

NEXT: ADDING A NEW
FIELD

Adding a new field to a Razor Page

10/6/2017 • 4 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this section you'll use [Entity Framework Code First Migrations](#) to add a new field to the model and migrate that change to the database.

When you use EF Code First to automatically create a database, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app (Ctrl+Shift+B).

Edit *Pages/Movies/Index.cshtml*, and add a `Rating` field:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
```

```

<thead>
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.Movie[0].Title)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movie[0].Genre)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movie[0].Price)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movie[0].Rating)
    </th>
  </tr>
</thead>
<tbody>
@foreach (var item in Model.Movie) {
  <tr>
    <td>
      @Html.DisplayFor(modelItem => item.Title)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.ReleaseDate)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.Genre)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.Price)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.Rating)
    </td>
    <td>
      <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
      <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
      <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
    </td>
  </tr>
}
</tbody>
</table>

```

Add the `Rating` field to the Delete and Details pages.

Update `Create.cshtml` with a `Rating` field. You can copy/paste the previous `<div>` element and let IntelliSense help you update the fields. IntelliSense works with [Tag Helpers](#).

```

</div>
<div class="form-group">
  <label asp-for="Movie.Price" class="control-label"></label>
  <input asp-for="Movie.Price" class="form-control" />
  <span asp-validation-for="Movie.Price" class="text-danger"></span>
</div>
<div class="form-group">
  <label asp-for="Movie.Price" class="control-label"></label>
  <input asp-for="Movie.Price" class="form-control" />
  <span asp-validation-for="Movie.Price" class="text-danger"></span>
</div>
<div class="form-group">
  <input type="submit" value="Submit" class="btn btn-default" />
</div>
</div>
</div>
<div>
  <a asp-page="Index">Back to List</a>
</div>

```



The following code shows *Create.cshtml* with a `Rating` field:

```

@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Rating" class="control-label"></label>
                <input asp-for="Movie.Rating" class="form-control" />
                <span asp-validation-for="Movie.Rating" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Add the `Rating` field to the Edit Page.

The app won't work until the DB is updated to include the new field. If run now, the app throws a `SqlException` :

```
SqlException: Invalid column name 'Rating'.
```

This error is caused by the updated Movie model class being different than the schema of the Movie table of the database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle; it allows you to quickly evolve the model and database schema together. The downside is that you lose existing data in the database. You don't want to use this approach on a production database! Dropping the DB on schema changes and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie` block.

```
context.Movie.AddRange(  
    new Movie  
    {  
        Title = "When Harry Met Sally",  
        ReleaseDate = DateTime.Parse("1989-2-12"),  
        Genre = "Romantic Comedy",  
        Price = 7.99M,  
        Rating = "R"  
    },  
);
```

See the [completed SeedData.cs file](#).

Build the solution.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**. In the PMC, enter the following commands:

```
Add-Migration Rating  
Update-Database
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` DB schema.
- Create code to migrate the DB schema to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If you delete all the records in the DB, the initializer will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from [Sql Server Object Explorer](#) (SSOX). To delete the database from SSOX:

- Select the database in SSOX.
- Right click on the database, and select *Delete*.
- Check **Close existing connections**.
- Select **OK**.
- In the [PMC](#), update the database:

Update-Database

Run the app and verify you can create/edit/display movies with a `Rating` field. If the database is not seeded, stop IIS Express, and then run the app.

PREVIOUS: ADDING
SEARCH

NEXT: ADDING
VALIDATION

Adding validation to a Razor Page

11/21/2017 • 7 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this section validation logic is added to the `Movie` model. The validation rules are enforced any time a user creates or edits a movie.

Validation

A key tenet of software development is called **DRY** ("Don't Repeat Yourself"). Razor Pages encourages development where functionality is specified once, and it's reflected throughout the app. DRY can help reduce the amount of code in an app. DRY makes the code less error prone, and easier to test and maintain.

The validation support provided by Razor Pages and Entity Framework is a good example of the DRY principle. Validation rules are declaratively specified in one place (in the model class), and the rules are enforced everywhere in the app.

Adding validation rules to the movie model

Open the `Movie.cs` file. [DataAnnotations](#) provides a built-in set of validation attributes that are applied declaratively to a class or property. `DataAnnotations` also contains formatting attributes like `DataType` that help with formatting and don't provide validation.

Update the `Movie` class to take advantage of the `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [DisplayName = "Release Date"]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'\s]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'\s]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

Validation attributes specify behavior that's enforced on model properties:

- The `Required` and `MinimumLength` attributes indicate that a property must have a value. However, nothing

prevents a user from entering whitespace to satisfy the validation constraint for a nullable type. Non-nullable **value types** (such as `decimal`, `int`, `float`, and `DateTime`) are inherently required and don't need the `Required` attribute.

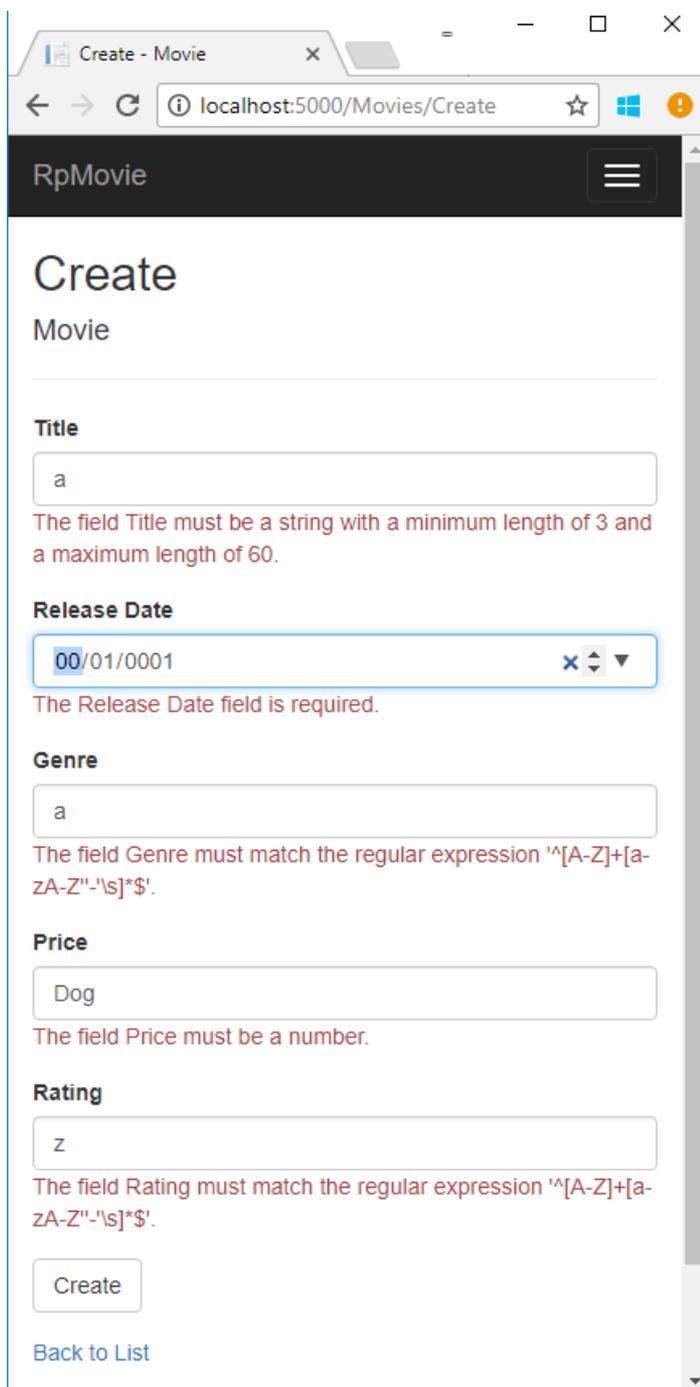
- The `RegularExpression` attribute limits the characters that the user can enter. In the preceding code, `Genre` and `Rating` must use only letters (whitespace, numbers, and special characters aren't allowed).
- The `Range` attribute constrains a value to a specified range.
- The `StringLength` attribute sets the maximum length of a string, and optionally the minimum length.

Having validation rules automatically enforced by ASP.NET Core helps make an app more robust. Automatic validation on models helps protect the app because you don't have to remember to apply them when new code is added.

Validation Error UI in Razor Pages

Run the app and navigate to Pages/Movies.

Select the **Create New** link. Complete the form with some invalid values. When jQuery client-side validation detects the error, it displays an error message.



NOTE

You may not be able to enter decimal points or commas in the `Price` field. To support [jQuery validation](#) in non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See [Additional resources](#) for more information. For now, just enter whole numbers like 10.

Notice how the form has automatically rendered a validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (when a user has JavaScript disabled).

A significant benefit is that **no** code changes were necessary in the Create or Edit pages. Once DataAnnotations were applied to the model, the validation UI was enabled. The Razor Pages created in this tutorial automatically picked up the validation rules (using validation attributes on the properties of the `Movie` model class). Test validation using the Edit page, the same validation is applied.

The form data is not posted to the server until there are no client-side validation errors. Verify form data is not posted by one or more of the following approaches:

- Put a break point in the `OnPostAsync` method. Submit the form (select **Create** or **Save**). The break point is never hit.
- Use the [Fiddler tool](#).
- Use the browser developer tools to monitor network traffic.

Server-side validation

When JavaScript is disabled in the browser, submitting the form with errors will post to the server.

Optional, test server-side validation:

- Disable JavaScript in the browser. If you can't disable JavaScript in the browser, try another browser.
- Set a break point in the `OnPostAsync` method of the Create or Edit page.
- Submit a form with validation errors.
- Verify the model state is invalid:

```
if (!ModelState.IsValid)
{
    return Page();
}
```

The following code shows a portion of the `Create.cshtml` page that you scaffolded earlier in the tutorial. It's used by the Create and Edit pages to display the initial form and to redisplay the form in the event of an error.

```
<form method="post">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
  </div>
```

The [Input Tag Helper](#) uses the [DataAnnotations](#) attributes and produces HTML attributes needed for jQuery Validation on the client-side. The [Validation Tag Helper](#) displays validation errors. See [Validation](#) for more information.

The Create and Edit pages have no validation rules in them. The validation rules and the error strings are specified only in the `Movie` class. These validation rules are automatically applied to Razor Pages that edit the `Movie` model.

When validation logic needs to change, it's done only in the model. Validation is applied consistently throughout the application (validation logic is defined in one place). Validation in one place helps keep the code clean, and makes it easier to maintain and update.

Using DataType Attributes

Examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. The `DataType` attribute is applied to the `ReleaseDate` and `Price` properties.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supplies attributes such as `<a>` for URL's and `` for email). Use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that is more specific than the database intrinsic type. `DataType` attributes are not validation attributes. In the sample application, only the date is displayed, without time.

The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`. A date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers consume. The `DataType` attributes do **not** provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should be applied when the value is displayed for editing. You might not want that behavior for some fields. For example, in currency values, you probably do not want the currency symbol in the edit UI.

The `DisplayFormat` attribute can be used by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable the ASP.NET Core framework to choose the right field template to render the data. The `DisplayFormat` if used by itself uses the string template.

Note: jQuery validation does not work with the `Range` attribute and `DateTime`. For example, the following code

will always display a client-side validation error, even when the date is in the specified range:

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

[Getting started with Razor Pages and EF Core](#) shows more advanced EF Core operations with Razor Pages.

Publish to Azure

See [Publish an ASP.NET Core web app to Azure App Service using Visual Studio](#) for instructions on how to publish this app to Azure.

Additional resources

- [Working with Forms](#)
- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Authoring Tag Helpers](#)

PREVIOUS: ADDING A NEW
FIELD

NEXT: UPLOADING
FILES

Uploading files to a Razor Page in ASP.NET Core

10/19/2017 • 11 min to read • [Edit Online](#)

By [Luke Latham](#)

In this section, uploading files with a Razor Page is demonstrated.

The [Razor Pages Movie sample app](#) in this tutorial uses simple model binding to upload files, which works well for uploading small files. For information on streaming large files, see [Uploading large files with streaming](#).

In the steps below, you add a movie schedule file upload feature to the sample app. A movie schedule is represented by a `Schedule` class. The class includes two versions of the schedule. One version is provided to customers, `PublicSchedule`. The other version is used for company employees, `PrivateSchedule`. Each version is uploaded as a separate file. The tutorial demonstrates how to perform two file uploads from a page with a single POST to the server.

Add a FileUpload class

Below, you create a Razor page to handle a pair of file uploads. Add a `FileUpload` class, which is bound to the page to obtain the schedule data. Right click the `Models` folder. Select **Add > Class**. Name the class **FileUpload** and add the following properties:

```
using Microsoft.AspNetCore.Http;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class FileUpload
    {
        [Required]
        [Display(Name="Title")]
        [StringLength(60, MinimumLength = 3)]
        public string Title { get; set; }

        [Required]
        [Display(Name="Public Schedule")]
        public IFormFile UploadPublicSchedule { get; set; }

        [Required]
        [Display(Name="Private Schedule")]
        public IFormFile UploadPrivateSchedule { get; set; }
    }
}
```

The class has a property for the schedule's title and a property for each of the two versions of the schedule. All three properties are required, and the title must be 3-60 characters long.

Add a helper method to upload files

To avoid code duplication for processing uploaded schedule files, add a static helper method first. Create a `Utilities` folder in the app and add a `FileHelpers.cs` file with the following content. The helper method, `ProcessFormFile`, takes an `IFormFile` and `ModelStateDictionary` and returns a string containing the file's size and content. The content type and length are checked. If the file doesn't pass a validation check, an error is added to the `ModelState`.

```
using System;
```

```

using System;
using System.ComponentModel.DataAnnotations;
using System.IO;
using System.Net;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Utilities
{
    public class FileHelpers
    {
        public static async Task<string> ProcessFormFile(IFormFile formFile, ModelStateDictionary modelState)
        {
            var fieldDisplayName = string.Empty;

            // Use reflection to obtain the display name for the model
            // property associated with this IFormFile. If a display
            // name isn't found, error messages simply won't show
            // a display name.
            MemberInfo property =
                typeof(FileUpload).GetProperty(formFile.Name.Substring(formFile.Name.IndexOf(".") + 1));

            if (property != null)
            {
                var displayAttribute =
                    property.GetCustomAttribute(typeof(DisplayAttribute)) as DisplayAttribute;

                if (displayAttribute != null)
                {
                    {
                        fieldDisplayName = $"{displayAttribute.Name} ";
                    }
                }
            }

            // Use Path.GetFileName to obtain the file name, which will
            // strip any path information passed as part of the
            // FileName property. HtmlEncode the result in case it must
            // be returned in an error message.
            var fileName = WebUtility.HtmlEncode(Path.GetFileName(formFile.FileName));

            if (formFile.ContentType.ToLower() != "text/plain")
            {
                modelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) must be a text file.");
            }

            // Check the file length and don't bother attempting to
            // read it if the file contains no content. This check
            // doesn't catch files that only have a BOM as their
            // content, so a content length check is made later after
            // reading the file's content to catch a file that only
            // contains a BOM.
            if (formFile.Length == 0)
            {
                modelState.AddModelError(formFile.Name, $"The {fieldDisplayName}file ({fileName}) is empty.");
            }
            else
            {
                try
                {
                    string fileContents;

                    // The StreamReader is created to read files that are UTF-8 encoded.
                    // If uploads require some other encoding, provide the encoding in the
                    // using statement. To change to 32-bit encoding, change
                    // new UTF8Encoding(...) to new UTF32Encoding().
                }
            }
        }
    }
}

```

```

        using (
            var reader =
                new StreamReader(
                    formFile.OpenReadStream(),
                    new UTF8Encoding(encoderShouldEmitUTF8Identifier: false, throwOnInvalidBytes:
true),
                        detectEncodingFromByteOrderMarks: true))
        {
            fileContents = await reader.ReadToEndAsync();

            // Check the content length in case the file's only
            // content was a BOM and the content is actually
            // empty after removing the BOM.
            if (fileContents.Length > 0)
            {
                return fileContents;
            }
            else
            {
                ModelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) is empty.");
            }
        }
    }
    catch (Exception ex)
    {
        ModelState.AddModelError(formFile.Name,
            $"The {fieldDisplayName}file ({fileName}) upload failed. " +
            $"Please contact the Help Desk for support. Error:
{ex.Message}");
        // Log the exception
    }
}

return string.Empty;
}
}
}

```

Add the Schedule class

Right click the *Models* folder. Select **Add** > **Class**. Name the class **Schedule** and add the following properties:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Schedule
    {
        public int ID { get; set; }
        public string Title { get; set; }

        public string PublicSchedule { get; set; }

        [Display(Name = "Public Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PublicScheduleSize { get; set; }

        public string PrivateSchedule { get; set; }

        [Display(Name = "Private Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PrivateScheduleSize { get; set; }

        [Display(Name = "Uploaded (UTC)")]
        [DisplayFormat(DataFormatString = "{0:F}")]
        public DateTime UploadDT { get; set; }
    }
}

```

The class uses `Display` and `DisplayFormat` attributes, which produce friendly titles and formatting when the schedule data is rendered.

Update the MovieContext

Specify a `DbSet` in the `MovieContext` (*Models/MovieContext.cs*) for the schedules:

```

using Microsoft.EntityFrameworkCore;

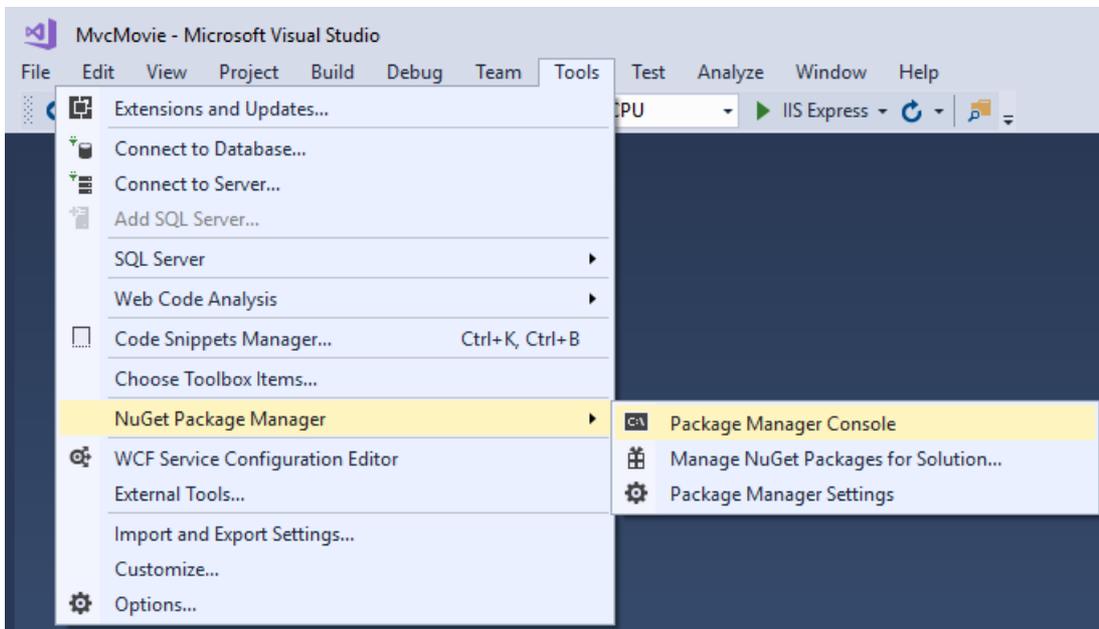
namespace RazorPagesMovie.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
        public DbSet<Schedule> Schedule { get; set; }
    }
}

```

Add the Schedule table to the database

Open the Package Manger Console (PMC): **Tools > NuGet Package Manager > Package Manager Console.**



In the PMC, execute the following commands. These commands add a `Schedule` table to the database:

```
Add-Migration AddScheduleTable
Update-Database
```

Add a file upload Razor Page

In the *Pages* folder, create a *Schedules* folder. In the *Schedules* folder, create a page named *Index.cshtml* for uploading a schedule with the following content:

```
@page
@model RazorPagesMovie.Pages.Schedules.IndexModel

@{
    ViewData["Title"] = "Schedules";
}

<h2>Schedules</h2>
<hr />

<h3>Upload Schedules</h3>
<div class="row">
    <div class="col-md-4">
        <form method="post" enctype="multipart/form-data">
            <div class="form-group">
                <label asp-for="FileUpload.Title" class="control-label"></label>
                <input asp-for="FileUpload.Title" type="text" class="form-control" />
                <span asp-validation-for="FileUpload.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FileUpload.UploadPublicSchedule" class="control-label"></label>
                <input asp-for="FileUpload.UploadPublicSchedule" type="file" class="form-control"
                style="height:auto" />
                <span asp-validation-for="FileUpload.UploadPublicSchedule" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FileUpload.UploadPrivateSchedule" class="control-label"></label>
                <input asp-for="FileUpload.UploadPrivateSchedule" type="file" class="form-control"
                style="height:auto" />
                <span asp-validation-for="FileUpload.UploadPrivateSchedule" class="text-danger"></span>
            </div>
            <input type="submit" value="Upload" class="btn btn-default" />
        </form>
    </div>
</div>
```

```

    </div>
</div>

<h3>Loaded Schedules</h3>
<table class="table">
  <thead>
    <tr>
      <th></th>
      <th>
        @Html.DisplayNameFor(model => model.Schedule[0].Title)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Schedule[0].UploadDT)
      </th>
      <th class="text-center">
        @Html.DisplayNameFor(model => model.Schedule[0].PublicScheduleSize)
      </th>
      <th class="text-center">
        @Html.DisplayNameFor(model => model.Schedule[0].PrivateScheduleSize)
      </th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model.Schedule) {
      <tr>
        <td>
          <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.UploadDT)
        </td>
        <td class="text-center">
          @Html.DisplayFor(modelItem => item.PublicScheduleSize)
        </td>
        <td class="text-center">
          @Html.DisplayFor(modelItem => item.PrivateScheduleSize)
        </td>
      </tr>
    }
  </tbody>
</table>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Each form group includes a **<label>** that displays the name of each class property. The `Display` attributes in the `FileUpload` model provide the display values for the labels. For example, the `UploadPublicSchedule` property's display name is set with `[Display(Name="Public Schedule")]` and thus displays "Public Schedule" in the label when the form renders.

Each form group includes a validation ****. If the user's input fails to meet the property attributes set in the `FileUpload` class or if any of the `ProcessFormFile` method file validation checks fail, the model fails to validate. When model validation fails, a helpful validation message is rendered to the user. For example, the `Title` property is annotated with `[Required]` and `[StringLength(60, MinimumLength = 3)]`. If the user fails to supply a title, they receive a message indicating that a value is required. If the user enters a value less than three characters or more than sixty characters, they receive a message indicating that the value has an incorrect length. If a file is provided that has no content, a message appears indicating that the file is empty.

Add the code-behind file

Add the code-behind file (*Index.cshtml.cs*) to the *Schedules* folder:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using RazorPagesMovie.Utilities;

namespace RazorPagesMovie.Pages.Schedules
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public IndexModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public FileUpload FileUpload { get; set; }

        public IList<Schedule> Schedule { get; private set; }

        public async Task OnGetAsync()
        {
            Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            // Perform an initial check to catch FileUpload class
            // attribute violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var publicScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPublicSchedule, ModelState);

            var privateScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPrivateSchedule, ModelState);

            // Perform a second check to catch ProcessFormFile method
            // violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var schedule = new Schedule()
            {
                PublicSchedule = publicScheduleData,
                PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
                PrivateSchedule = privateScheduleData,
                PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
                Title = FileUpload.Title,
                UploadDT = DateTime.UtcNow
            };

            _context.Schedule.Add(schedule);
            await context.SaveChangesAsync();
        }
    }
}
```

```
        return RedirectToPage("./Index");
    }
}
```

The page model (`IndexModel` in `Index.cshtml.cs`) binds the `FileUpload` class:

```
[BindProperty]
public FileUpload FileUpload { get; set; }
```

The model also uses a list of the schedules (`IList<Schedule>`) to display the schedules stored in the database on the page:

```
public IList<Schedule> Schedule { get; private set; }
```

When the page loads with `OnGetAsync`, `Schedules` is populated from the database and used to generate an HTML table of loaded schedules:

```
public async Task OnGetAsync()
{
    Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
}
```

When the form is posted to the server, the `ModelState` is checked. If invalid, `Schedule` is rebuilt, and the page renders with one or more validation messages stating why page validation failed. If valid, the `FileUpload` properties are used in `OnPostAsync` to complete the file upload for the two versions of the schedule and to create a new `Schedule` object to store the data. The schedule is then saved to the database:

```

public async Task<IActionResult> OnPostAsync()
{
    // Perform an initial check to catch FileUpload class
    // attribute violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var publicScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPublicSchedule, ModelState);

    var privateScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPrivateSchedule, ModelState);

    // Perform a second check to catch ProcessSchedule method
    // violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var schedule = new Schedule()
    {
        PublicSchedule = publicScheduleData,
        PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
        PrivateSchedule = privateScheduleData,
        PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
        Title = FileUpload.Title,
        UploadDT = DateTime.UtcNow
    };

    _context.Schedule.Add(schedule);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

```

Link the file upload Razor Page

Open *_Layout.cshtml* and add a link to the navigation bar to reach the file upload page:

```

<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li><a asp-page="/Index">Home</a></li>
    <li><a asp-page="/Schedules/Index">Schedules</a></li>
    <li><a asp-page="/About">About</a></li>
    <li><a asp-page="/Contact">Contact</a></li>
  </ul>
</div>

```

Add a page to confirm schedule deletion

When the user clicks to delete a schedule, you want them to have a chance to cancel the operation. Add a delete confirmation page (*Delete.cshtml*) to the *Schedules* folder:

```

@page "{id:int}"
@model RazorPagesMovie.Pages.Schedules.DeleteModel

@{
    ViewData["Title"] = "Delete Schedule";
}

<h2>Delete Schedule</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Schedule</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PublicScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PublicScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PrivateScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PrivateScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.UploadDT)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.UploadDT)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Schedule.ID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>

```

The code-behind file (*Delete.cshtml.cs*) loads a single schedule identified by `id` in the request's route data. Add the *Delete.cshtml.cs* file to the *Schedules* folder:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Schedules
{
    public class DeleteModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public DeleteModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Schedule Schedule { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Schedule = await _context.Schedule.SingleOrDefaultAsync(m => m.ID == id);

            if (Schedule == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Schedule = await _context.Schedule.FindAsync(id);

            if (Schedule != null)
            {
                _context.Schedule.Remove(Schedule);
                await _context.SaveChangesAsync();
            }

            return RedirectToPage("./Index");
        }
    }
}

```

The `OnPostAsync` method handles deleting the schedule by its `id` :

```

public async Task<IActionResult> OnPostAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Schedule = await _context.Schedule.FindAsync(id);

    if (Schedule != null)
    {
        _context.Schedule.Remove(Schedule);
        await _context.SaveChangesAsync();
    }

    return RedirectToPage("./Index");
}

```

After successfully deleting the schedule, the `RedirectToPage` sends the user back to the schedules *Index.cshhtml* page.

The working Schedules Razor Page

When the page loads, labels and inputs for schedule title, public schedule, and private schedule are rendered with a submit button:

Upload Schedules

Title

Public Schedule

Private Schedule

Selecting the **Upload** button without populating any of the fields violates the `[Required]` attributes on the model. The `ModelState` is invalid. The validation error messages are displayed to the user:

Upload Schedules

Title

The Title field is required.

Public Schedule

The Public Schedule field is required.

Private Schedule

The Private Schedule field is required.

Type two letters into the **Title** field. The validation message changes to indicate that the title must be between 3-60 characters:

Title

The field Title must be a string with a minimum length of 3 and a maximum length of 60.

When one or more schedules are uploaded, the **Loaded Schedules** section renders the loaded schedules:

Loaded Schedules				
	Title	Uploaded (UTC)	Public Schedule Size (bytes)	Private Schedule Size (bytes)
Delete	Schedule #1	Monday, September 11, 2017 9:09:22 PM	3,805.0	2,417.0
Delete	Schedule #2	Monday, September 11, 2017 9:09:35 PM	2,023.0	997.0

The user can click the **Delete** link from there to reach the delete confirmation view, where they have an opportunity to confirm or cancel the delete operation.

Troubleshooting

For troubleshooting information with `IFormFile` uploading, see the [File uploads in ASP.NET Core: Troubleshooting](#).

Thanks for completing this introduction to Razor Pages. We appreciate any comments you leave. [Getting started with MVC and EF Core](#) is an excellent follow up to this tutorial.

Additional resources

- [File uploads in ASP.NET Core](#)

- IFormFile

PREVIOUS:
VALIDATION

Create a web app with ASP.NET Core MVC using Visual Studio

11/30/2017 • 1 min to read • [Edit Online](#)

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the [Razor Pages](#) tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

There are 3 versions of this tutorial:

- Windows: This series
- macOS: [Create an ASP.NET Core MVC app with Visual Studio for Mac](#)
- macOS, Linux, and Windows: [Create an ASP.NET Core MVC app with Visual Studio Code](#) The tutorial series includes the following:

1. [Getting started](#)
2. [Adding a controller](#)
3. [Adding a view](#)
4. [Adding a model](#)
5. [Working with SQL Server LocalDB](#)
6. [Controller methods and views](#)
7. [Adding Search](#)
8. [Adding a New Field](#)
9. [Adding Validation](#)
10. [Examining the Details and Delete methods](#)

Getting started with ASP.NET Core MVC and Visual Studio

1/12/2018 • 2 min to read • [Edit Online](#)

By [Rick Anderson](#)

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the [Razor Pages](#) tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

There are 3 versions of this tutorial:

- macOS: [Create an ASP.NET Core MVC app with Visual Studio for Mac](#)
- Windows: [Create an ASP.NET Core MVC app with Visual Studio](#)
- macOS, Linux, and Windows: [Create an ASP.NET Core MVC app with Visual Studio Code](#)

Install Visual Studio and .NET Core

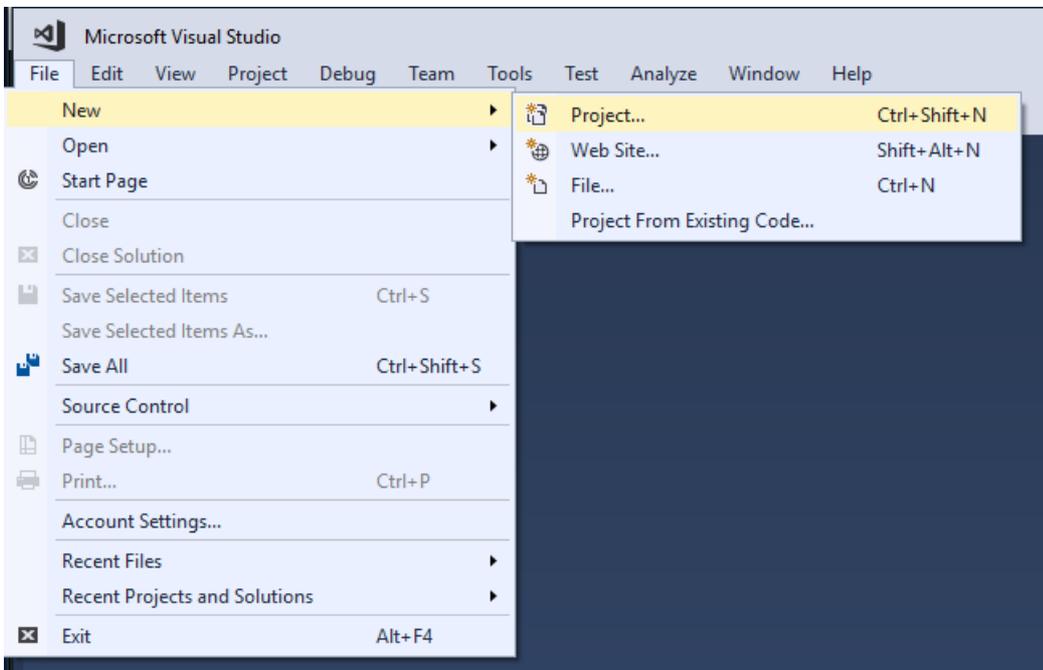
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Install the following:

- [.NET Core 2.0.0 SDK](#) or later.
- [Visual Studio 2017](#) version 15.3 or later with the **ASP.NET and web development** workload.

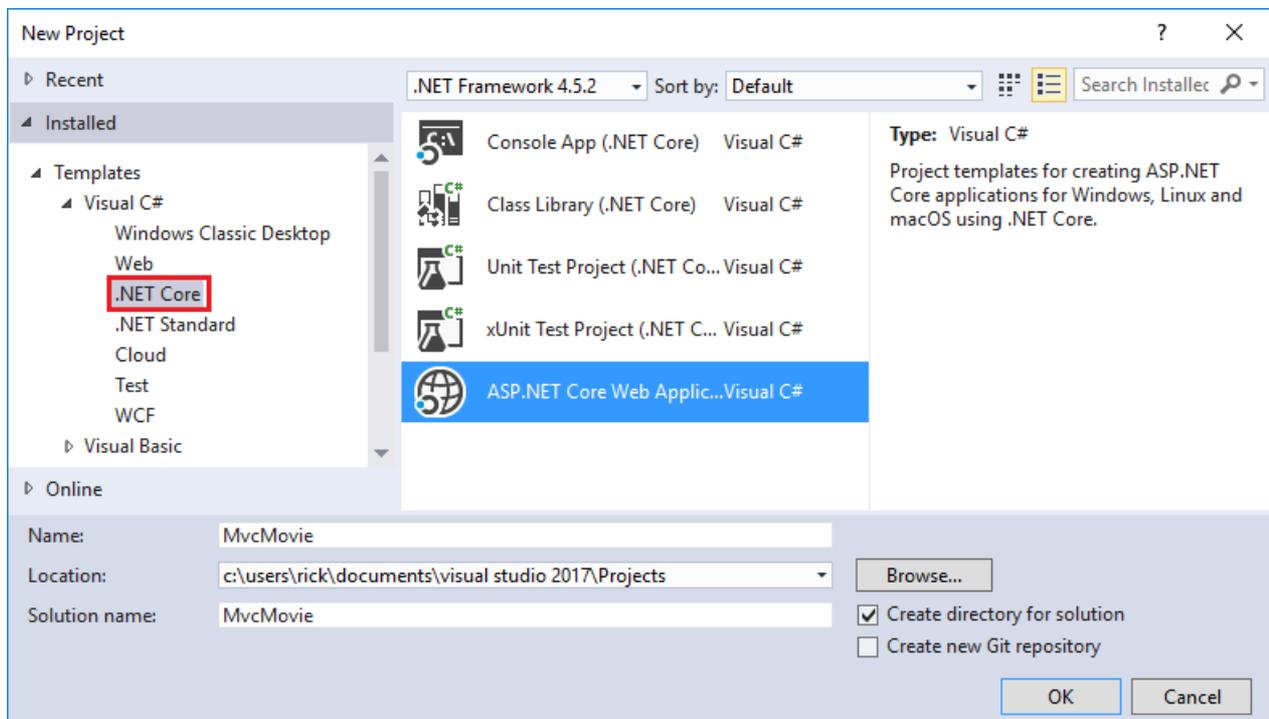
Create a web app

From Visual Studio, select **File > New > Project**.



Complete the **New Project** dialog:

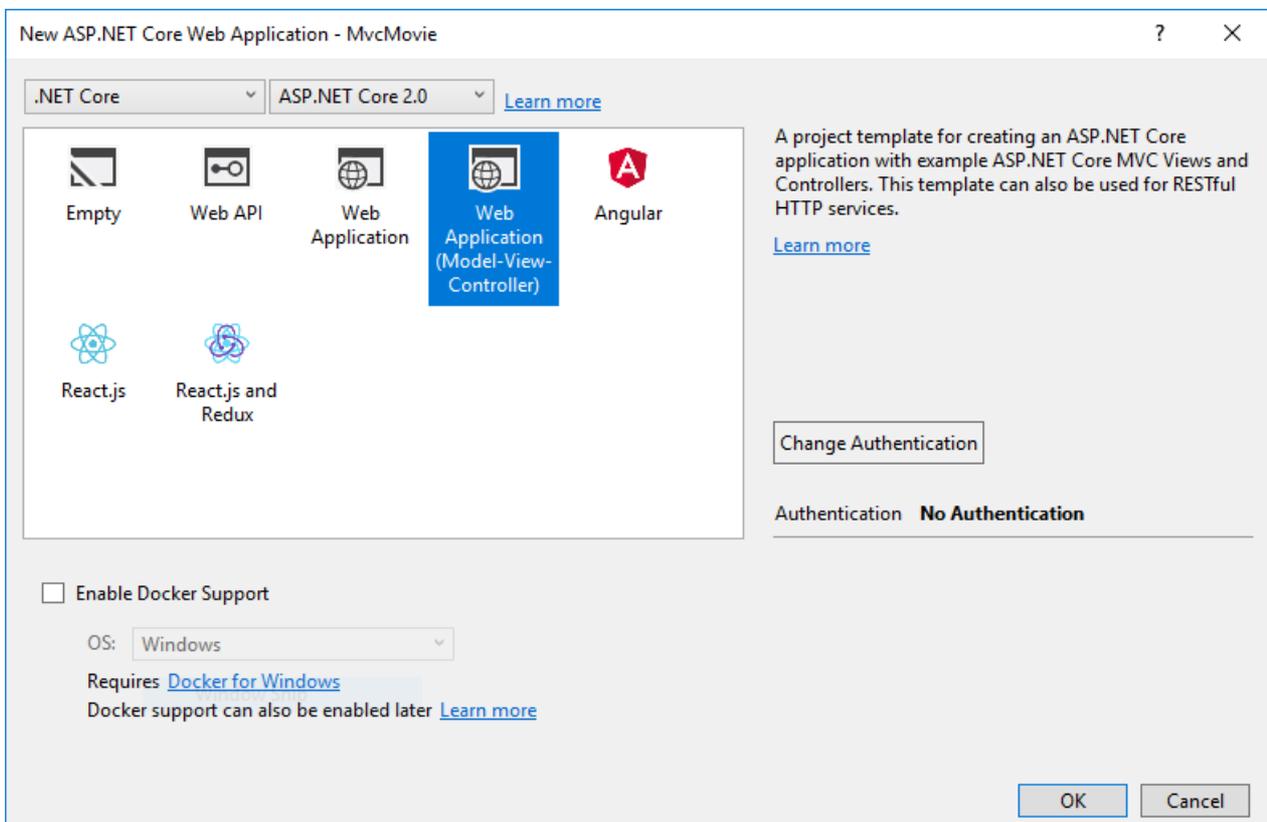
- In the left pane, tap **.NET Core**
- In the center pane, tap **ASP.NET Core Web Application (.NET Core)**
- Name the project "MvcMovie" (It's important to name the project "MvcMovie" so when you copy code, the namespace will match.)
- Tap **OK**



- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

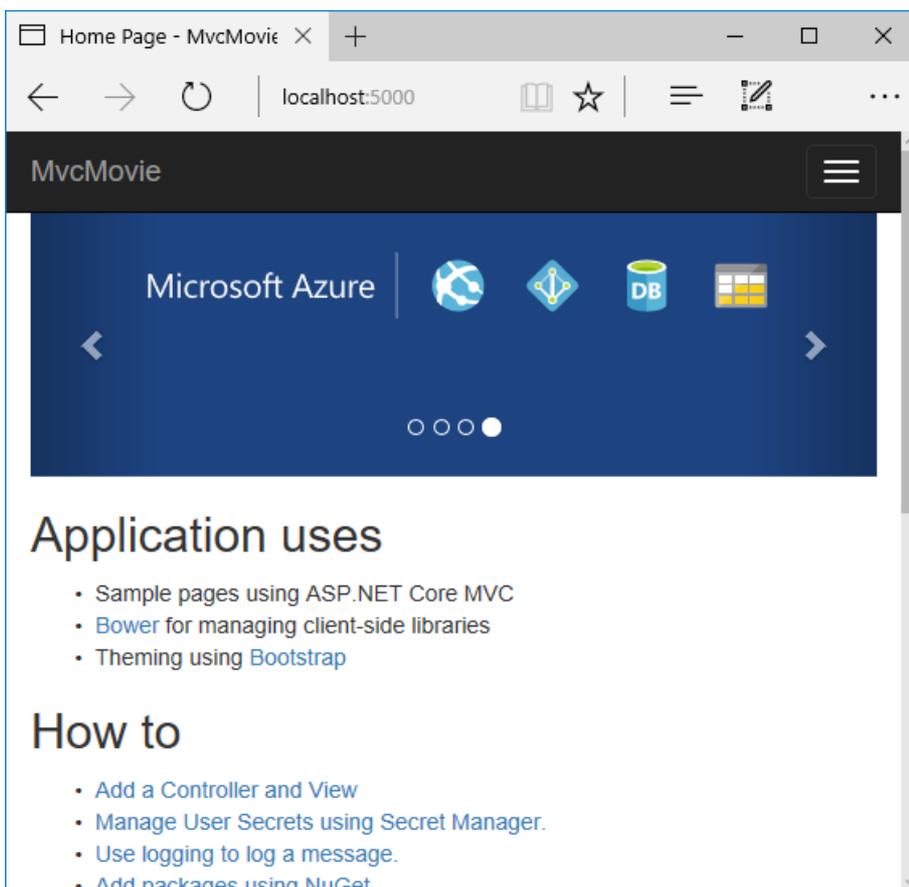
Complete the **New ASP.NET Core Web Application (.NET Core) - MvcMovie** dialog:

- In the version selector drop-down box select **ASP.NET Core 2.-**
- Select **Web Application(Model-View-Controller)**
- Tap **OK**.



Visual Studio used a default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a simple starter project, and it's a good place to start,

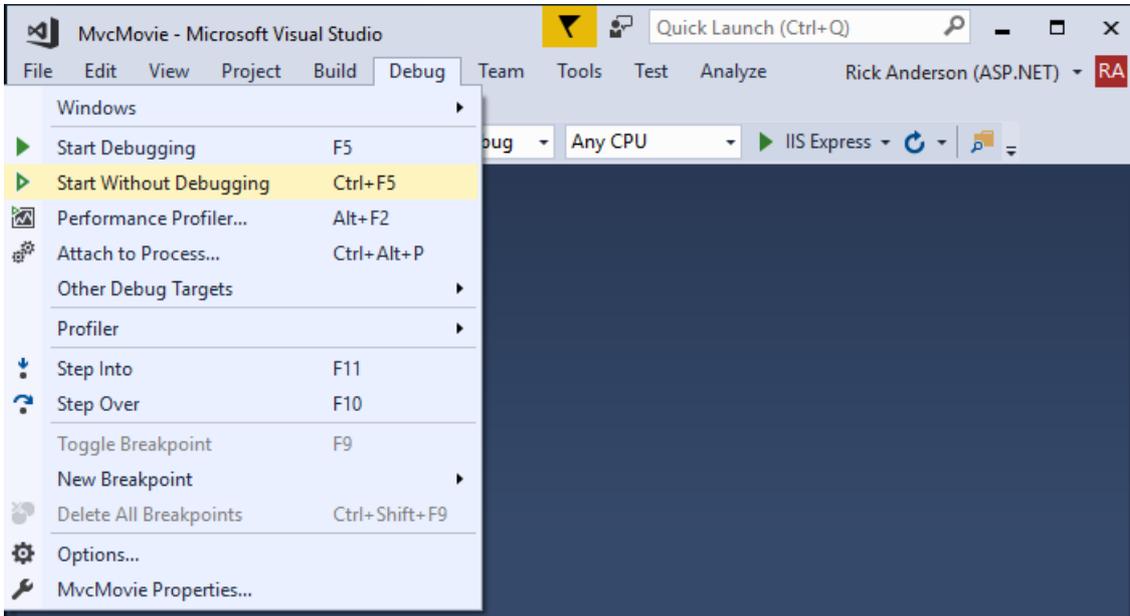
Tap **F5** to run the app in debug mode or **Ctrl-F5** in non-debug mode.



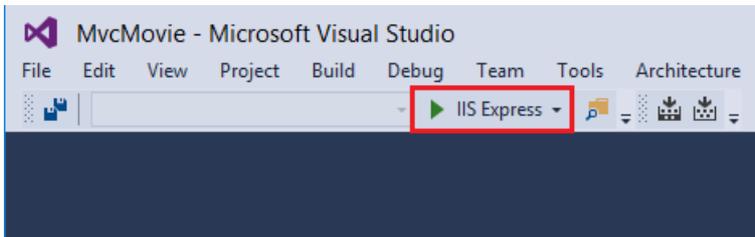
- Visual Studio starts [IIS Express](#) and runs your app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for your local computer. When Visual Studio creates a web project, a random port is used for the web server. In the image above, the

port number is 5000. The URL in the browser shows `localhost:5000`. When you run the app, you'll see a different port number.

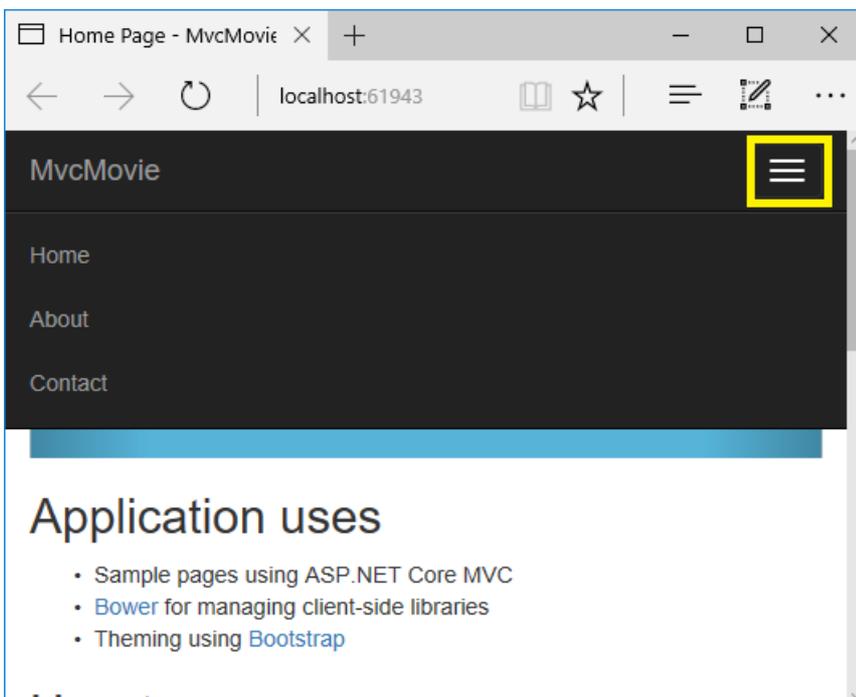
- Launching the app with **Ctrl+F5** (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
- You can launch the app in debug or non-debug mode from the **Debug** menu item:



- You can debug the app by tapping the **IIS Express** button



The default template gives you working **Home**, **About** and **Contact** links. The browser image above doesn't show these links. Depending on the size of your browser, you might need to click the navigation icon to show them.



If you were running in debug mode, tap **Shift-F5** to stop debugging.

In the next part of this tutorial, we'll learn about MVC and start writing some code.

NEXT

Adding a controller to a ASP.NET Core MVC app with Visual Studio

9/22/2017 • 5 min to read • [Edit Online](#)

By [Rick Anderson](#)

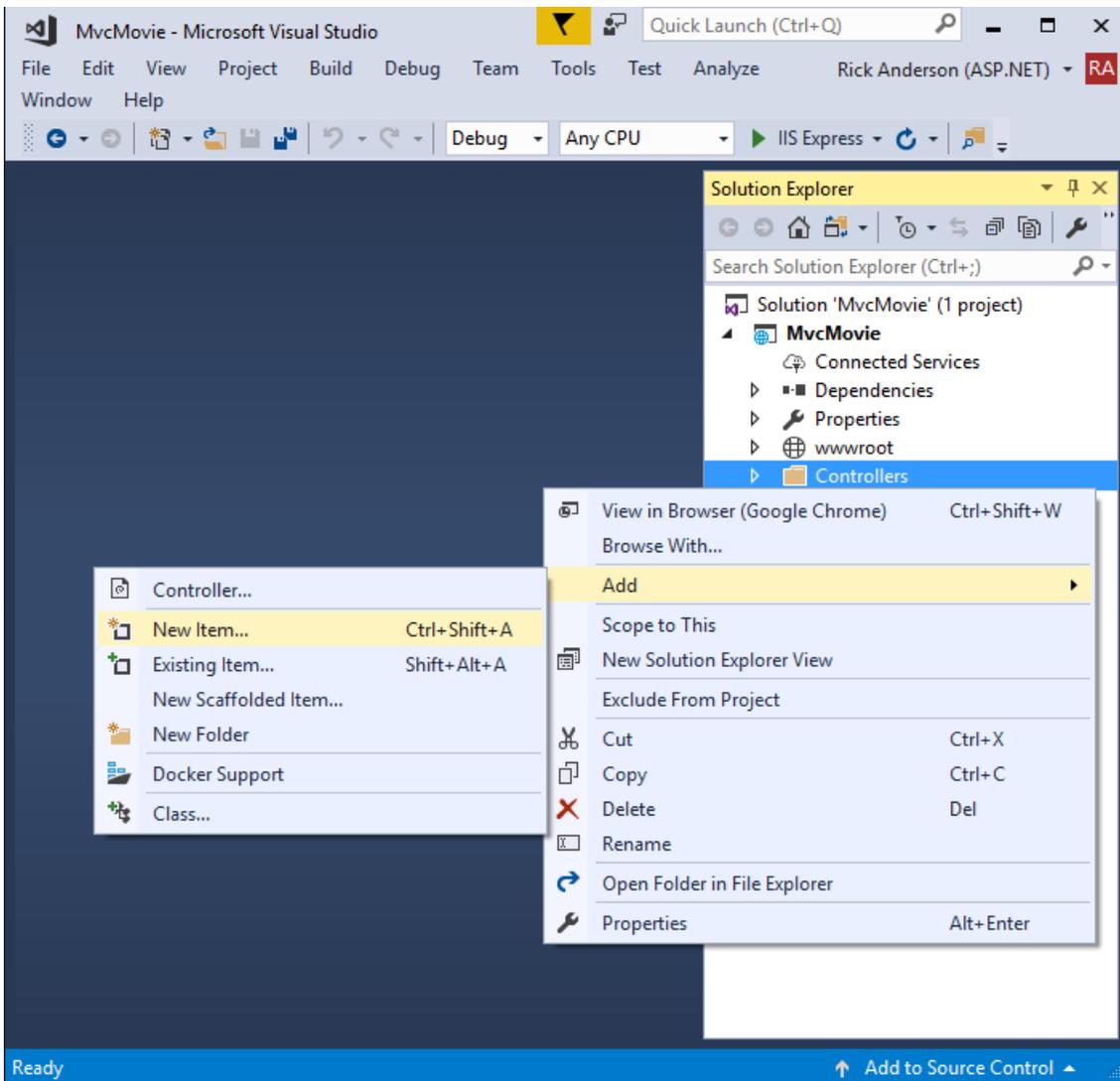
The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps. MVC-based apps contain:

- **Models:** Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **Views:** Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **Controllers:** Classes that handle browser requests. They retrieve model data and call view templates that return a response. In an MVC app, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database. For example, `http://localhost:1234/Home/About` has route data of `Home` (the controller) and `About` (the action method to call on the home controller). `http://localhost:1234/Movies/Edit/5` is a request to edit the movie with ID=5 using the movie controller. We'll talk about route data later in the tutorial.

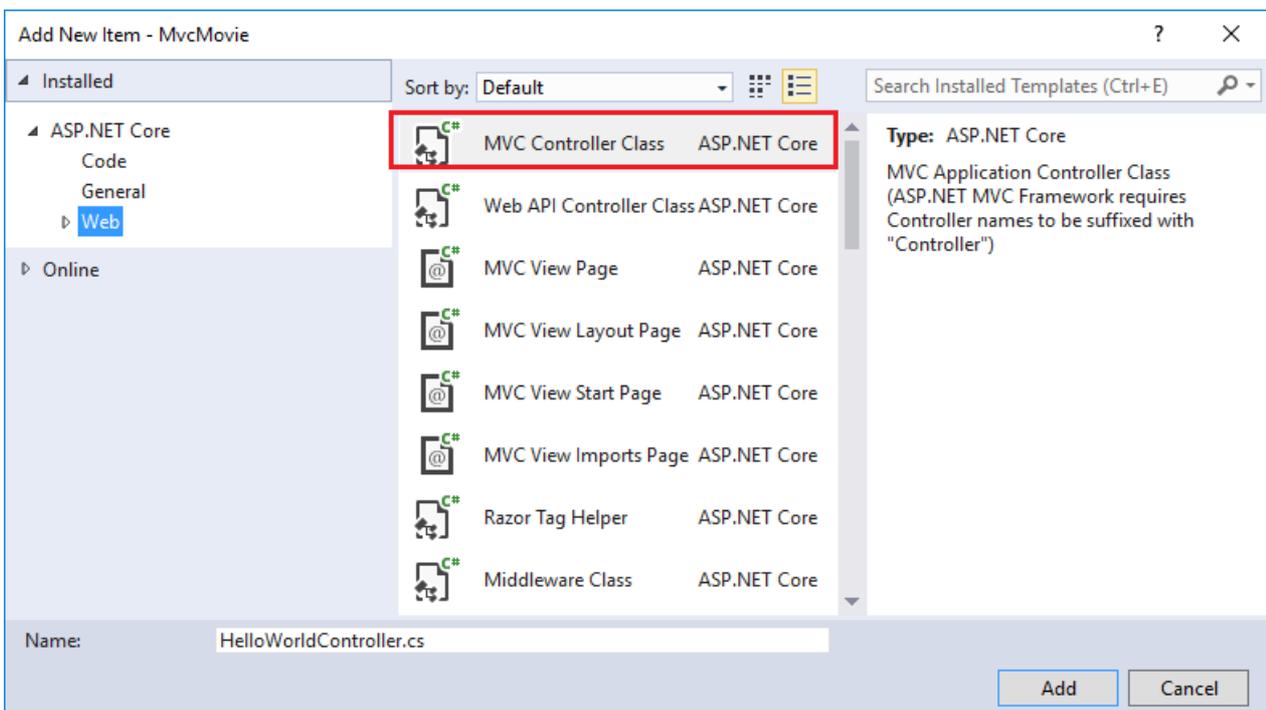
The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

We cover these concepts in this tutorial series and show you how to use them to build a movie app. The MVC project contains folders for the *Controllers* and *Views*.

- In **Solution Explorer**, right-click **Controllers** > **Add** > **New Item**



- Select **MVC Controller Class**
- In the **Add New Item** dialog, enter **HelloWorldController**.



Replace the contents of *Controllers/HelloWorldController.cs* with the following:

```

using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}

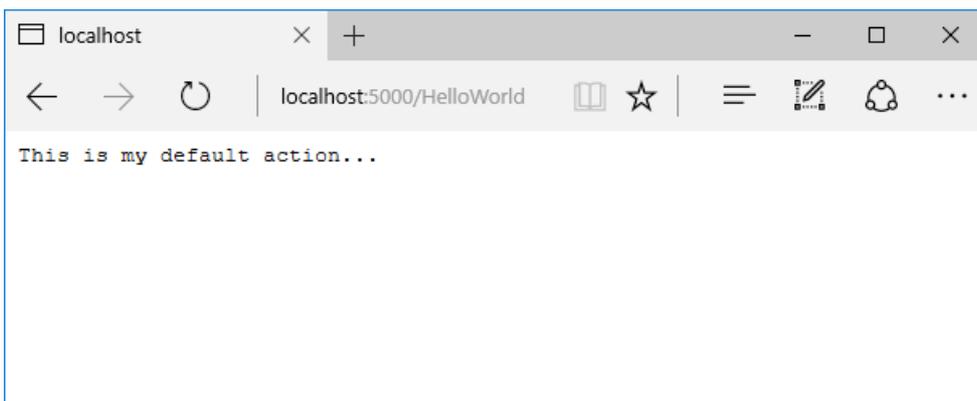
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint is a targetable URL in the web application, such as `http://localhost:1234/HelloWorld`, and combines the protocol used: `HTTP`, the network location of the web server (including the TCP port): `localhost:1234` and the target URI `HelloWorld`.

The first comment states this is an [HTTP GET](#) method that is invoked by appending `/HelloWorld/` to the base URL. The second comment specifies an [HTTP GET](#) method that is invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial you'll use the scaffolding engine to generate `HTTP POST` methods.

Run the app in non-debug mode and append `HelloWorld` to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default [URL routing logic](#) used by MVC uses a format like this to determine what code to invoke:

```
/[Controller]/[ActionName]/[Parameters]
```

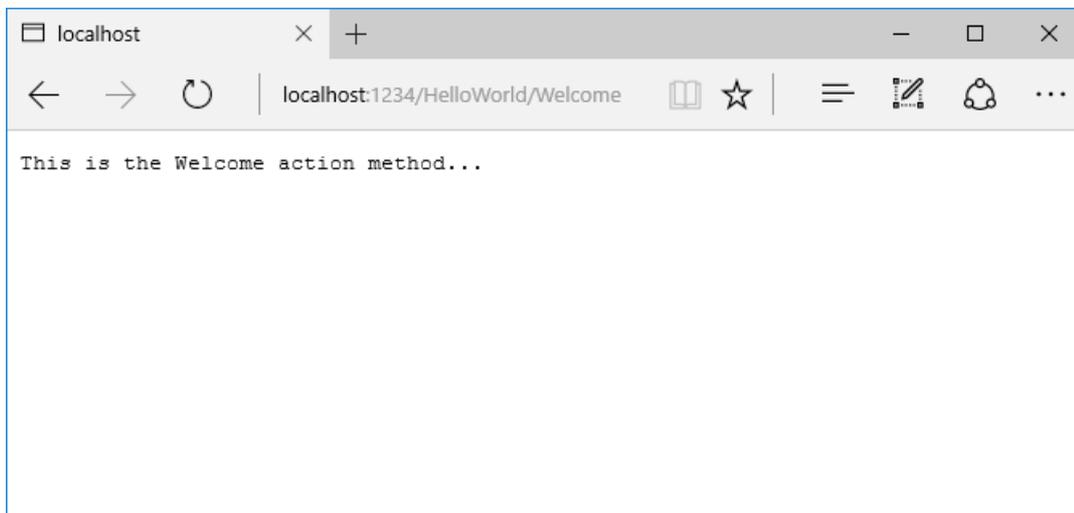
You set the format for routing in the `Startup.cs` file.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

When you run the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above.

The first URL segment determines the controller class to run. So `localhost:xxxx/HelloWorld` maps to the `HelloWorldController` class. The second part of the URL segment determines the action method on the class. So `localhost:xxxx/HelloWorld/Index` would cause the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:xxxx/HelloWorld` and the `Index` method was called by default. This is because `Index` is the default method that will be called on a controller if a method name is not explicitly specified. The third part of the URL segment (`{id}`) is for route data. You'll see route data later on in this tutorial.

Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numTimes=4`. Change the `Welcome` method to include two parameters as shown in the following code.

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript).
- Uses [Interpolated Strings](#).

Run your app and browse to:

```
http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4
```

(Replace xxxx with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC [model binding](#) system automatically maps the named parameters from the query string in the address bar to parameters in your method. See [Model Binding](#) for more information.



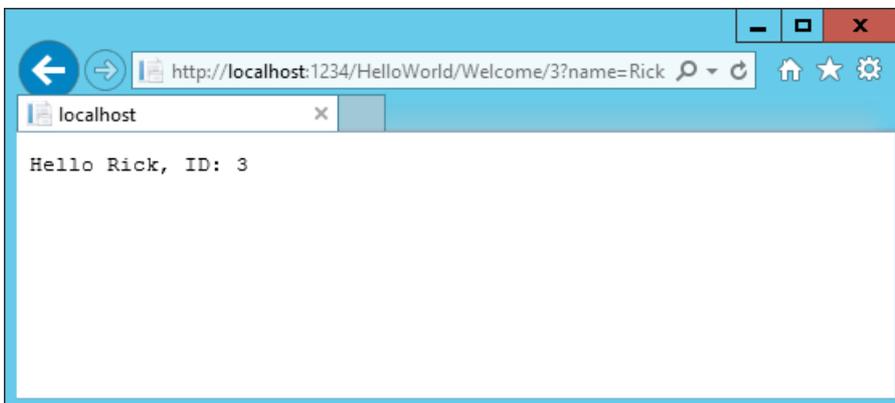
In the image above, the URL segment (`Parameters`) is not used, the `name` and `numTimes` parameters are passed as [query strings](#). The `?` (question mark) in the above URL is a separator, and the query strings follow. The `&` character separates query strings.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL:

```
http://localhost:xxx/HelloWorld/Welcome/3?name=Rick
```



This time the third URL segment matched the route parameter `id`. The `Welcome` method contains a parameter `id` that matched the URL template in the `MapRoute` method. The trailing `?` (in `id?`) indicates the `id` parameter is optional.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

In these examples the controller has been doing the "VC" portion of MVC - that is, the view and controller work. The controller is returning HTML directly. Generally you don't want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead you typically use a separate Razor view template file to

help generate the HTML response. You do that in the next tutorial.

In Visual Studio, in non-debug mode (Ctrl+F5), you don't need to build the app after changing code. Just save the file, refresh your browser and you can see the changes.

[PREVIOUS](#)[NEXT](#)

Adding a view to an ASP.NET Core MVC app

9/20/2017 • 8 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this section you modify the `HelloWorldController` class to use Razor view template files to cleanly encapsulate the process of generating HTML responses to a client.

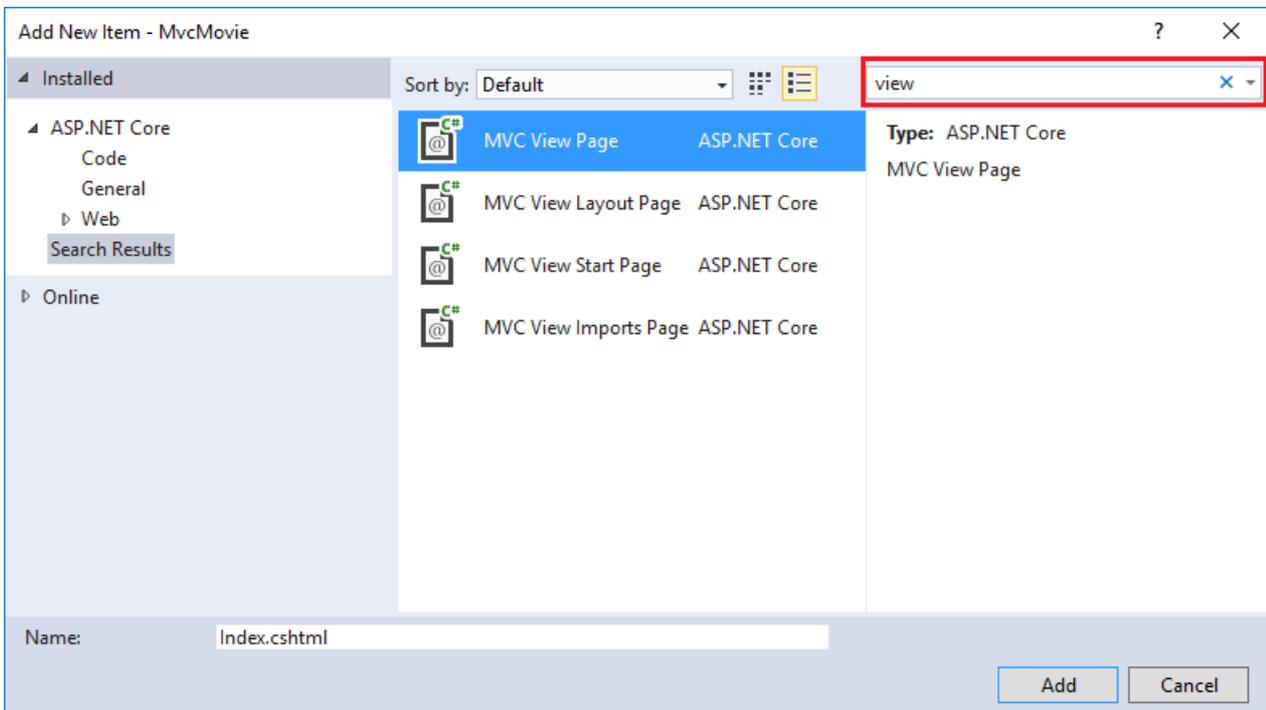
You create a view template file using Razor. Razor-based view templates have a `.cshtml` file extension. They provide an elegant way to create HTML output using C#.

Currently the `Index` method returns a string with a message that is hard-coded in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

```
public IActionResult Index()
{
    return View();
}
```

The preceding code returns a `View` object. It uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods) such as the `Index` method above, generally return an [ActionResult](#) (or a class derived from `ActionResult`), not a type like string.

- Right click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.
- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item - MvcMovie** dialog
 - In the search box in the upper-right, enter *view*
 - Tap **MVC View Page**
 - In the **Name** box, change the name if necessary to *Index.cshtml*.
 - Tap **Add**



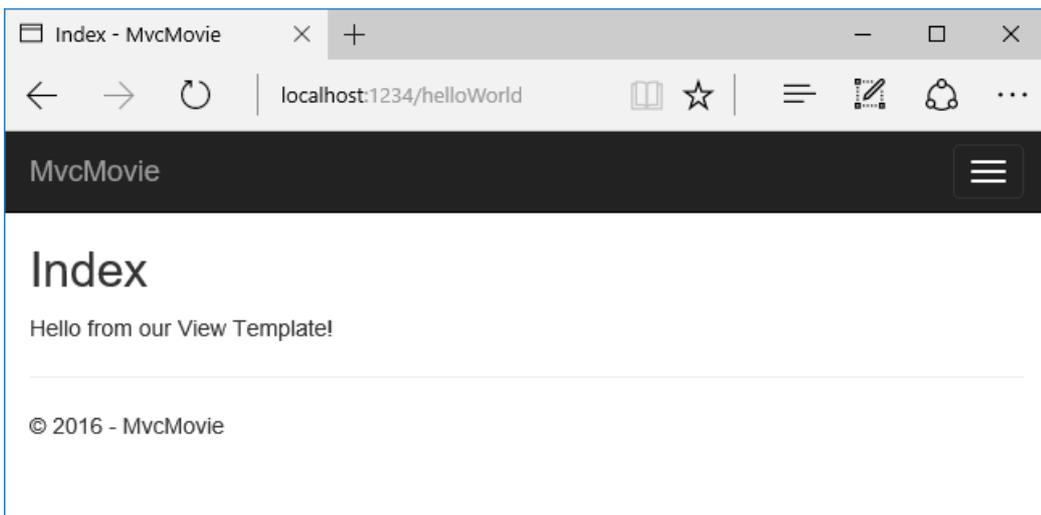
Replace the contents of the `Views/HelloWorld/Index.cshtml` Razor view file with the following:

```
@{
    ViewData["Title"] = "Index";
}

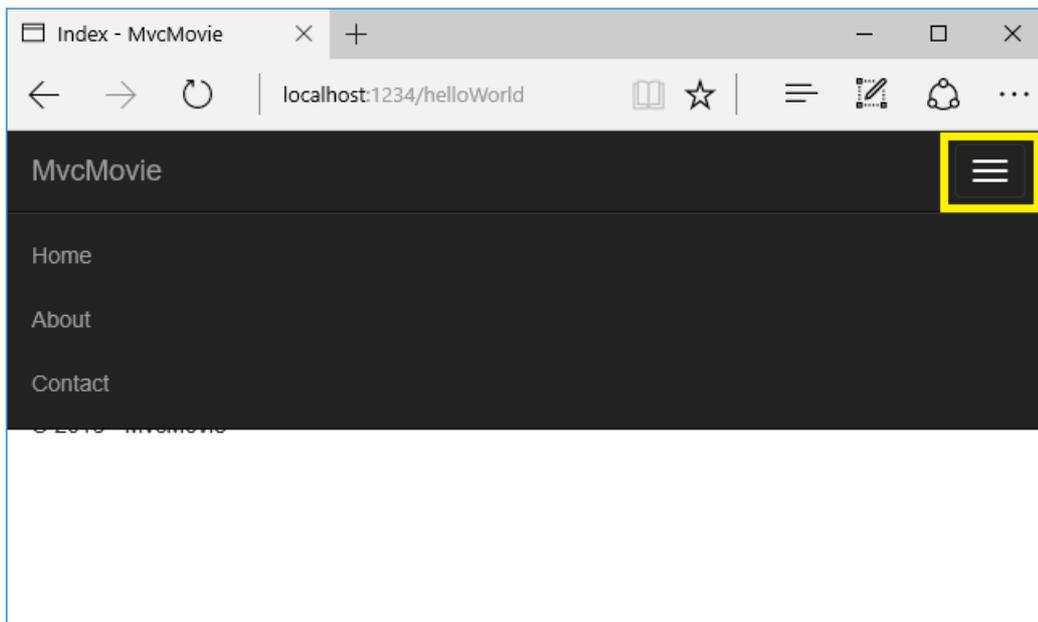
<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `http://localhost:xxxx/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much; it ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file, MVC defaulted to using the `Index.cshtml` view file in the `/Views/HelloWorld` folder. The image below shows the string "Hello from our View Template!" hard-coded in the view.



If your browser window is small (for example on a mobile device), you might need to toggle (tap) the [Bootstrap navigation button](#) in the upper right to see the **Home**, **About**, and **Contact** links.



Changing views and layout pages

Tap the menu links (**MvcMovie**, **Home**, **About**). Each page shows the same menu layout. The menu layout is implemented in the `Views/Shared/_Layout.cshtml` file. Open the `Views/Shared/_Layout.cshtml` file.

[Layout](#) templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **About** link, the **Views/Home/About.cshtml** view is rendered inside the `RenderBody` method.

Change the title and menu link in the layout file

In the title element, change `MvcMovie` to `Movie App`. Change the anchor text in the layout template from `MvcMovie` to `Mvc Movie` and the controller from `Home` to `Movies` as highlighted below:

Note: The ASP.NET Core 2.0 version is slightly different. It doesn't contain `@inject ApplicationInsights` and `@Html.Raw(JavaScriptSnippet.FullScript)`.

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - Movie App</title>

  <environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment names="Staging,Production">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
      value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>
  @Html.Raw(JavaScriptSnippet.FullScript)
</head>
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
```

```

<div class="container">
  <div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a asp-area="" asp-controller="Movies" asp-action="Index" class="navbar-brand">Movie App</a>
  </div>
  <div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
      <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
      <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
      <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
    </ul>
  </div>
</div>
</nav>
<div class="container body-content">
  @RenderBody()
  <hr />
  <footer>
    <p>&copy; 2017 - MvcMovie</p>
  </footer>
</div>

<environment names="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
asp-fallback-test="window.jQuery"
crossorigin="anonymous"
integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsFRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
</script>
  <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
crossorigin="anonymous"
integrity="sha384-Tc5IQib027qvyjSMFhJOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
</script>
  <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

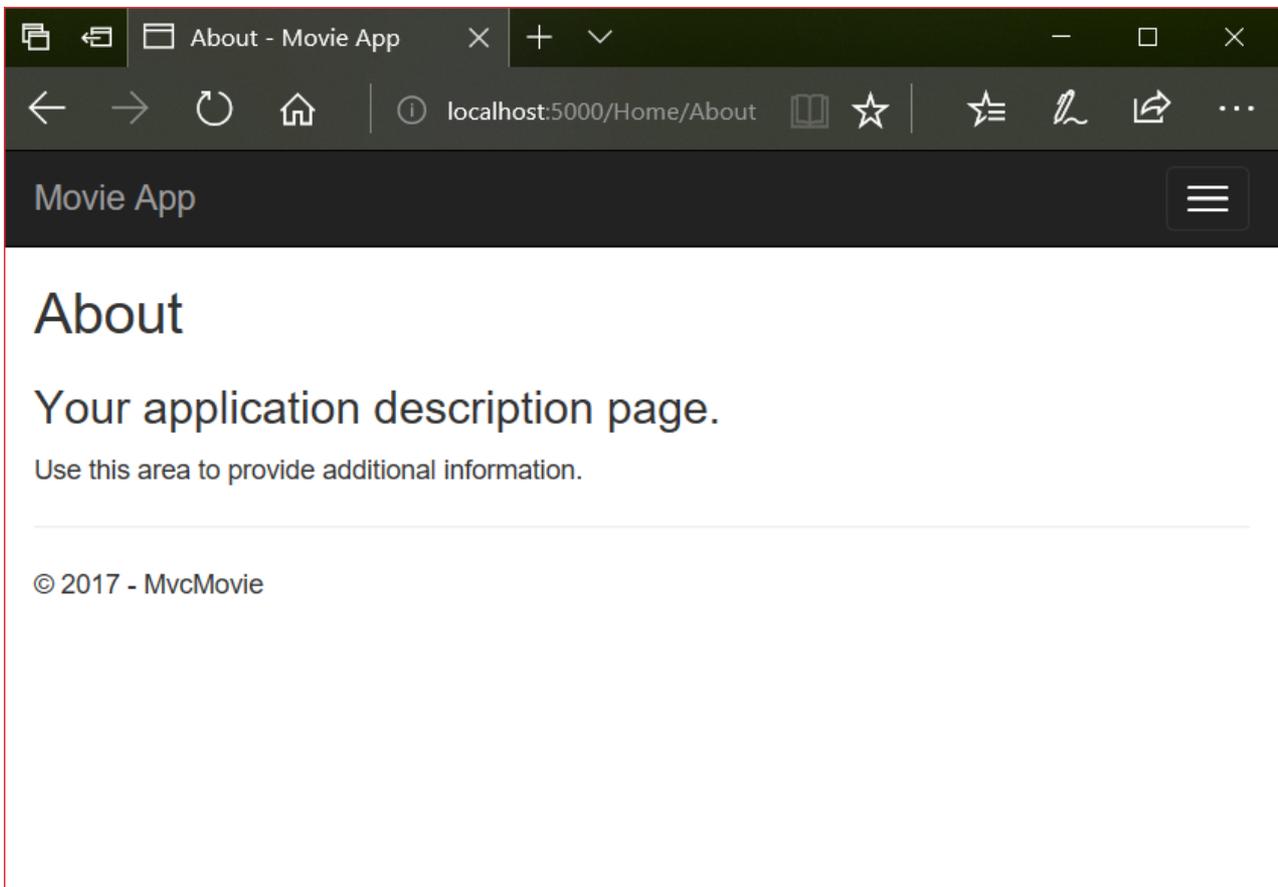
  @RenderSection("Scripts", required: false)
</body>
</html>

```

WARNING

We haven't implemented the `Movies` controller yet, so if you click on that link, you'll get a 404 (Not found) error.

Save your changes and tap the **About** link. Notice how the title on the browser tab now displays **About - Movie App** instead of **About - Mvc Movie**:



Tap the **Contact** link and notice that the title and anchor text also display **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the *Views/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view. You can use the `Layout` property to set a different layout view, or set it to `null` so no layout file will be used.

Change the title of the `Index` view.

Open *Views/HelloWorld/Index.cshtml*. There are two places to make a change:

- The text that appears in the title of the browser.
- The secondary header (`<h2>` element).

You'll make them slightly different so you can see which bit of code changes which part of the app.

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

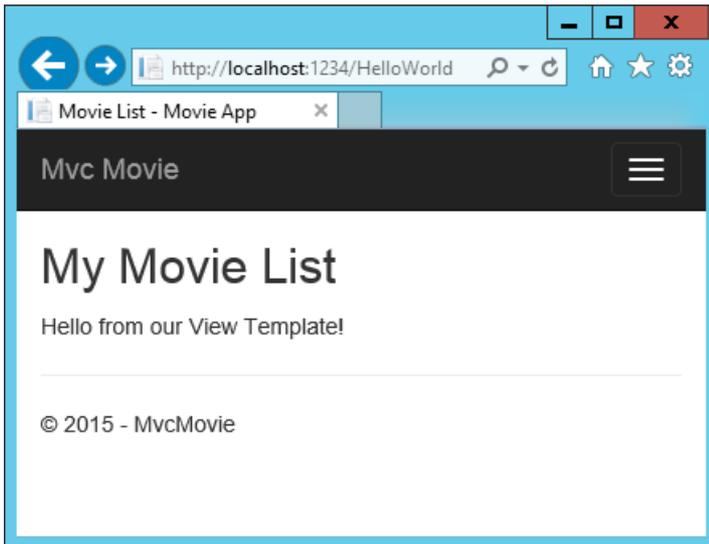
<p>Hello from our View Template!</p>
```

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

Save your change and navigate to `http://localhost:xxxx/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press `Ctrl+F5` in your browser to force the response from the server to be loaded.) The browser title is created with `ViewData["Title"]` we set in the `Index.cshtml` view template and the additional "- Movie App" added in the layout file.

Also notice how the content in the `Index.cshtml` view template was merged with the `Views/Shared/_Layout.cshtml` view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application. To learn more see [Layout](#).



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where you write the code that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep your code clean, testable, and maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and a `ID` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let's change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary that the view template can then access.

Return to the `HelloWorldController.cs` file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means you can put whatever you want in to it; the `ViewData` object has no defined properties until you put something inside it. The [MVC model binding system](#) automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete `HelloWorldController.cs` file looks like this:

```

using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}

```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named *Views/HelloWorld/Welcome.cshtml*.

You'll create a loop in the *Welcome.cshtml* view template that displays "Hello" `NumTimes`. Replace the contents of *Views/HelloWorld/Welcome.cshtml* with the following:

```

@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

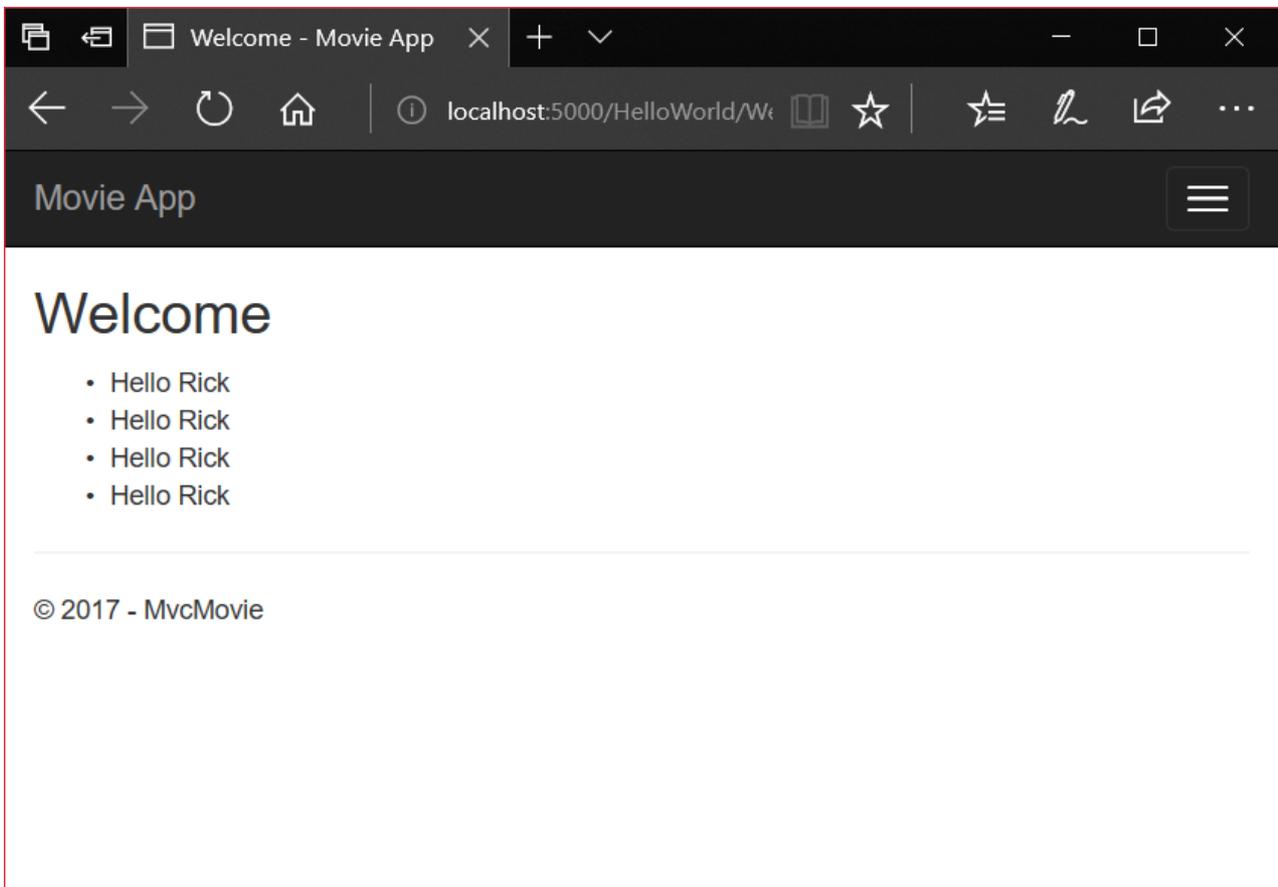
<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>

```

Save your changes and browse to the following URL:

```
http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4
```

Data is taken from the URL and passed to the controller using the [MVC model binder](#). The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the sample above, we used the `ViewData` dictionary to pass data from the controller to a view. Later in the tutorial, we will use a view model to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See [ViewModel vs ViewData vs ViewBag vs TempData vs Session in MVC](#) for more information.

Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

[PREVIOUS](#)

[NEXT](#)

Adding a model to an ASP.NET Core MVC app

12/9/2017 • 8 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Tom Dykstra](#)

In this section, you'll add some classes for managing movies in a database. These classes will be the "**Model**" part of the **MVC** app.

You use these classes with [Entity Framework Core](#) (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write. [EF Core supports many database engines](#).

The model classes you'll create are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They just define the properties of the data that will be stored in the database.

In this tutorial you'll write the model classes first, and EF Core will create the database. An alternate approach not covered here is to generate model classes from an already-existing database. For information about that approach, see [ASP.NET Core - Existing Database](#).

Add a data model class

Note: The ASP.NET Core 2.0 templates contain the *Models* folder.

Right-click the *Models* folder > **Add** > **Class**. Name the class **Movie** and add the following properties:

```
using System;

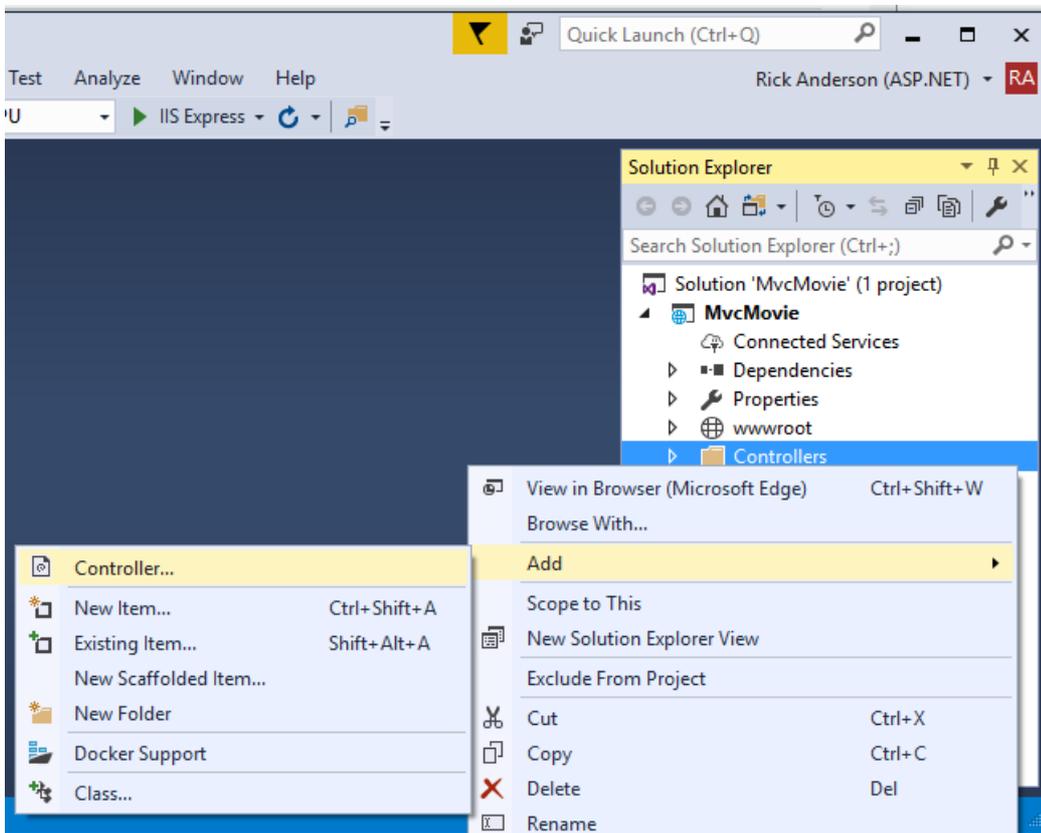
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `ID` field is required by the database for the primary key.

Build the project to verify you don't have any errors. You now have a **Model** in your **MVC** app.

Scaffolding a controller

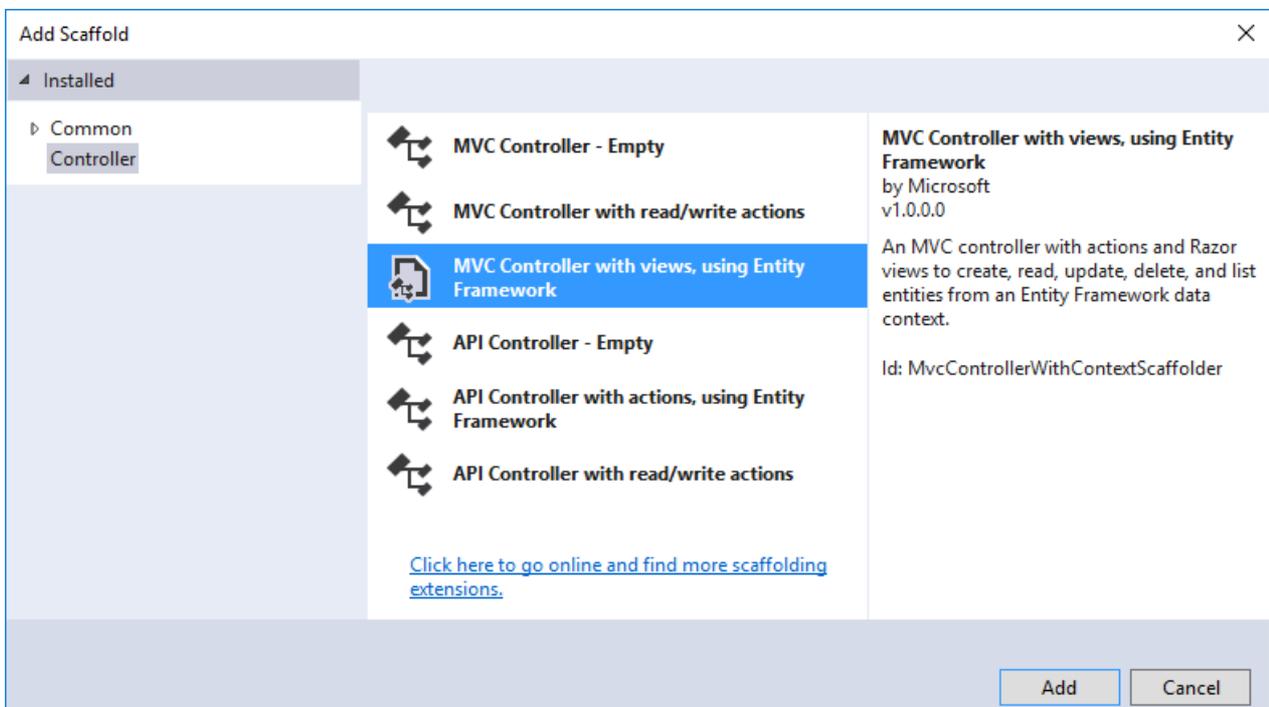
In **Solution Explorer**, right-click the *Controllers* folder > **Add** > **Controller**.



If the **Add MVC Dependencies** dialog appears:

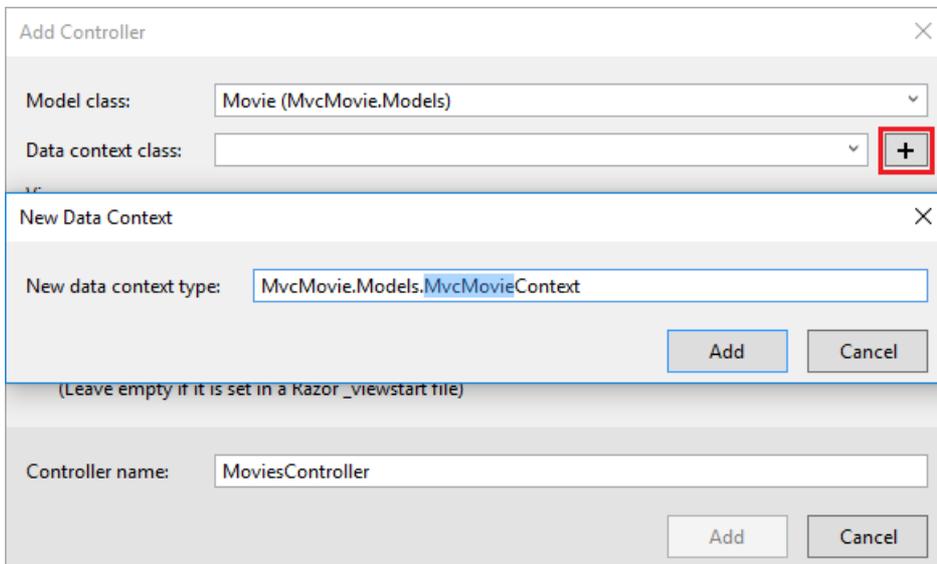
- [Update Visual Studio to the latest version](#). Visual Studio versions prior to 15.5 show this dialog.
- If you can't update, select **ADD**, and then follow the add controller steps again.

In the **Add Scaffold** dialog, tap **MVC Controller with views, using Entity Framework** > **Add**.

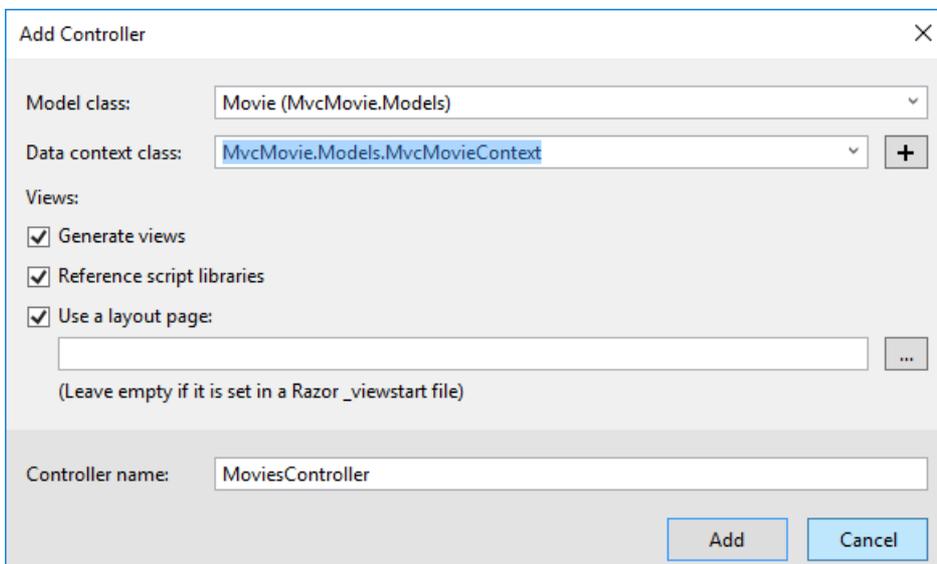


Complete the **Add Controller** dialog:

- **Model class:** *Movie* (*MvcMovie.Models*)
- **Data context class:** Select the + icon and add the default **MvcMovie.Models.MvcMovieContext**



- **Views:** Keep the default of each option checked
- **Controller name:** Keep the default *MoviesController*
- Tap **Add**

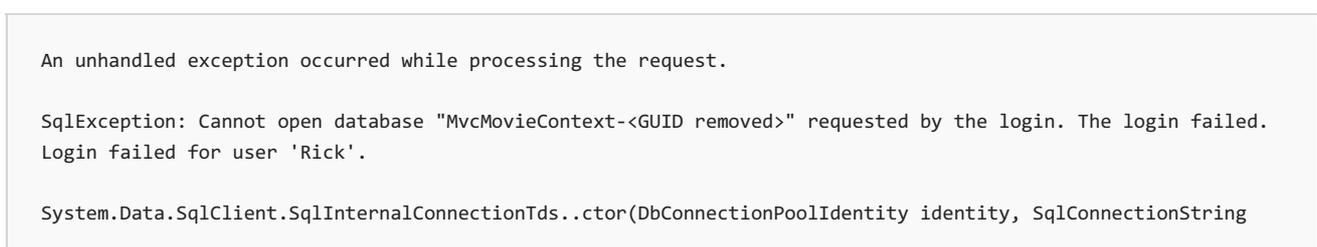


Visual Studio creates:

- An Entity Framework Core [database context class](#) (*Data/MvcMovieContext.cs*)
- A movies controller (*Controllers/MoviesController.cs*)
- Razor view files for Create, Delete, Details, Edit, and Index pages (*Views/Movies/*.cshtml*)

The automatic creation of the database context and **CRUD** (create, read, update, and delete) action methods and views is known as *scaffolding*. You'll soon have a fully functional web application that lets you manage a movie database.

If you run the app and click on the **Mvc Movie** link, you get an error similar to the following:



You need to create the database, and you'll use the EF Core [Migrations](#) feature to do that. Migrations lets you

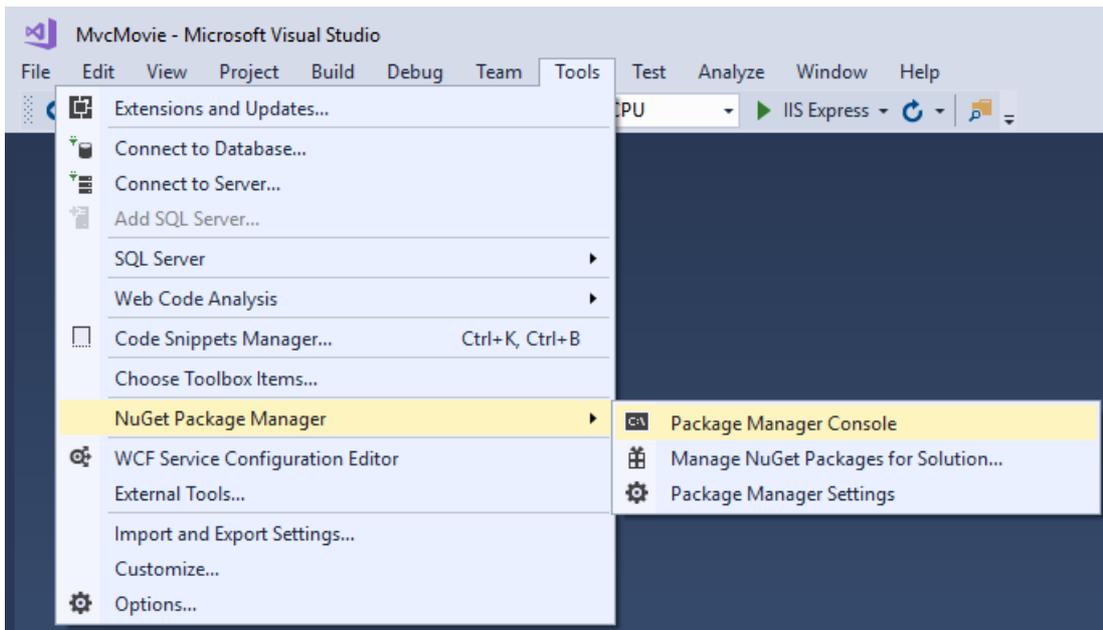
create a database that matches your data model and update the database schema when your data model changes.

Add EF tooling and perform initial migration

In this section you'll use the Package Manager Console (PMC) to:

- Add the Entity Framework Core Tools package. This package is required to add migrations and update the database.
- Add an initial migration.
- Update the database with the initial migration.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.



In the PMC, enter the following commands:

```
Install-Package Microsoft.EntityFrameworkCore.Tools
Add-Migration Initial
Update-Database
```

Note: If you receive an error with the `Install-Package` command, open NuGet Package Manager and search for the `Microsoft.EntityFrameworkCore.Tools` package. This allows you to install the package or check if it is already installed. Alternatively, see the [CLI approach](#) if you have problems with the PMC.

The `Add-Migration` command creates code to create the initial database schema. The schema is based on the model specified in the `DbContext` (In the `Data/MvcMovieContext.cs` file). The `Initial` argument is used to name the migrations. You can use any name, but by convention you choose a name that describes the migration. See [Introduction to migrations](#) for more information.

The `Update-Database` command runs the `up` method in the `Migrations/<time-stamp>_Initial.cs` file, which creates the database.

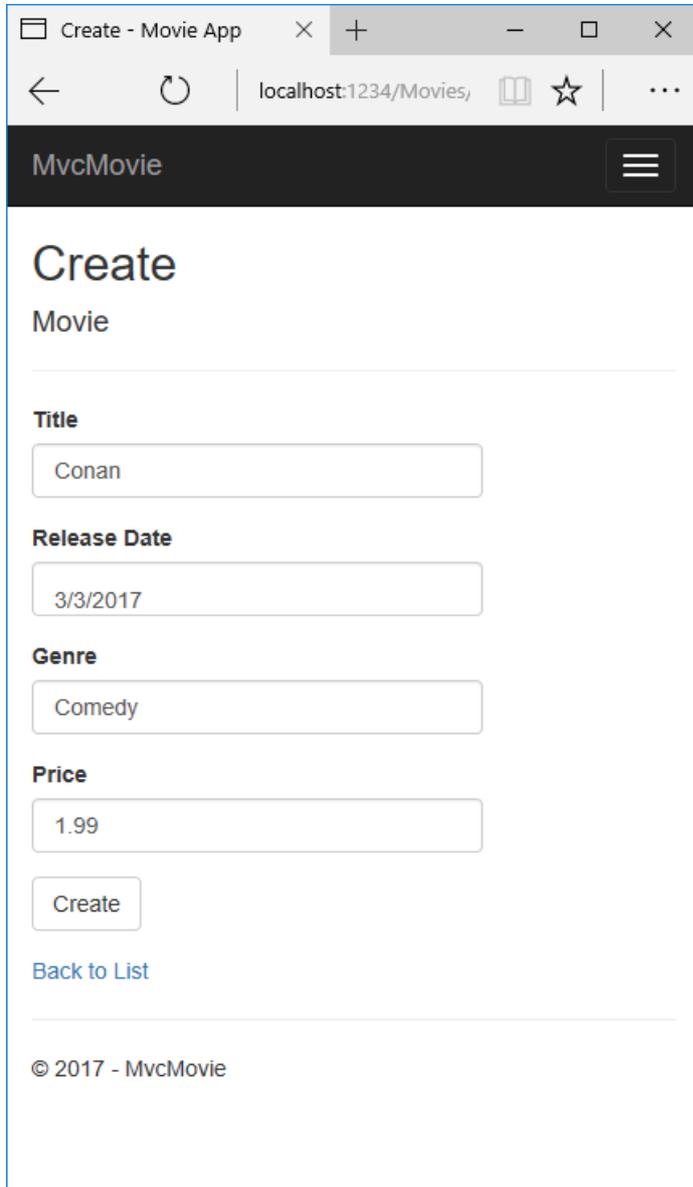
You can perform the preceding steps using the command-line interface (CLI) rather than the PMC:

- Add [EF Core tooling](#) to the `.csproj` file.
- Run the following commands from the console (in the project directory):

```
dotnet ef migrations add Initial
dotnet ef database update
```

Test the app

- Run the app and tap the **Mvc Movie** link.
- Tap the **Create New** link and create a movie.



The screenshot shows a mobile browser window with the address bar displaying 'localhost:1234/Movies'. The page title is 'MvcMovie'. The main content area is titled 'Create Movie' and contains a form with the following fields:

- Title:** Conan
- Release Date:** 3/3/2017
- Genre:** Comedy
- Price:** 1.99

Below the form is a 'Create' button and a 'Back to List' link. At the bottom of the page, there is a copyright notice: '© 2017 - MvcMovie'.

- You may not be able to enter decimal points or commas in the `Price` field. To support [jQuery validation](#) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See <https://github.com/aspnet/Docs/issues/4076> and [Additional resources](#) for more information. For now, just enter whole numbers like 10.
- In some locales you need to specify the date format. See the highlighted code below.

```

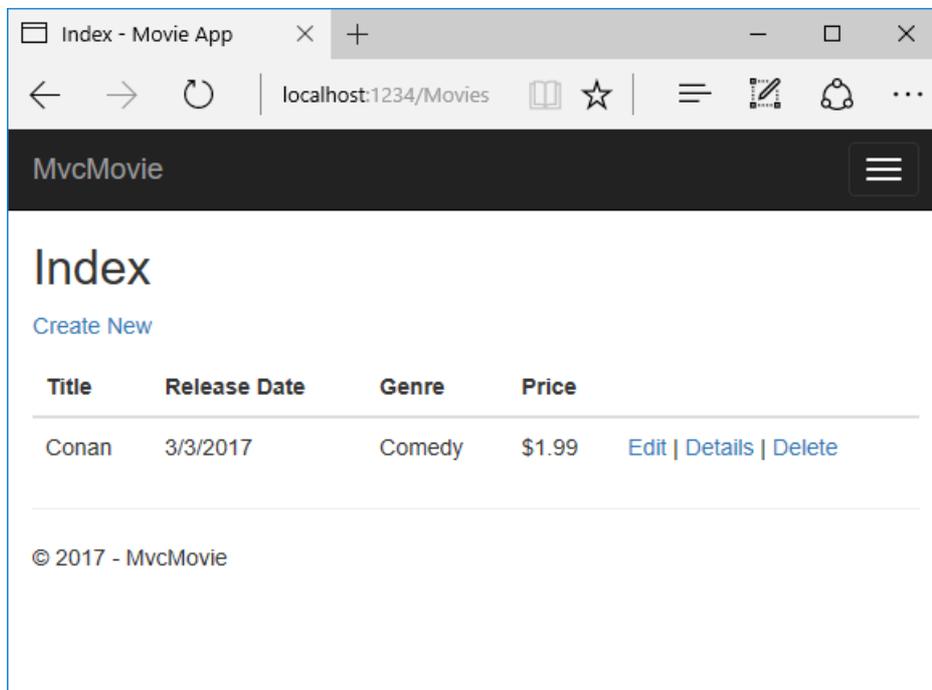
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}

```

We'll talk about `DataAnnotations` later in the tutorial.

Tapping **Create** causes the form to be posted to the server, where the movie information is saved in a database. The app redirects to the `/Movies` URL, where the newly created movie information is displayed.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

```

public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}

```

The highlighted code above shows the movie database context being added to the [Dependency Injection](#) container (In the `Startup.cs` file). `services.AddDbContext<MvcMovieContext>(options =>` specifies the database to use and the connection string. `=>` is a [lambda operator](#).

Open the `Controllers/MoviesController.cs` file and examine the constructor:

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
}
```

The constructor uses [Dependency Injection](#) to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the [CRUD](#) methods in the controller.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables better compile-time checking of your code. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the `MoviesController` class and views when it created the methods and views.

Examine the generated `Details` method in the `Controllers/MoviesController.cs` file:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example `http://localhost:5000/movies/details/1` sets:

- The controller to the `movies` controller (the first URL segment).
- The action to `details` (the second URL segment).
- The id to 1 (the last URL segment).

You can also pass in the `id` with a query string as follows:

```
http://localhost:1234/movies/details?id=1
```

The `id` parameter is defined as a [nullable type](#) (`int?`) in case an ID value is not provided.

A [lambda expression](#) is passed in to `SingleOrDefaultAsync` to select movie entities that match the route data or query string value.

```
var movie = await _context.Movie
    .SingleOrDefaultAsync(m => m.ID == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the `Views/Movies/Details.cshtml` file:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.ID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

By including a `@model` statement at the top of the view file, you can specify the type of object that the view expects. When you created the movie controller, Visual Studio automatically included the following `@model` statement at the top of the `Details.cshtml` file:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows you to access the movie that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the `Details.cshtml` view, the code passes each movie field to the

`DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the `Index.cshtml` view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

When you created the movies controller, scaffolding automatically included the following `@model` statement at the top of the `Index.cshtml` file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the `Index.cshtml` view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

```

@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

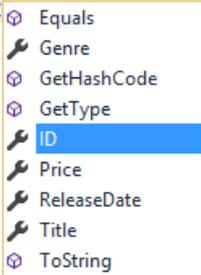
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
    }
    </tbody>
</table>

```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each item in the loop is typed as `Movie`. Among other benefits, this means that you get compile-time checking of the code:

```
@foreach (var item in Model) {  
    <tr>  
        <td>  
            @Html.DisplayFor(modelItem => item.Genre)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Price)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.ReleaseDate)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Title)  
        </td>  
        <td>  
            <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |  
            <a asp-action="Details" asp-route-id="@item.ID">Details</a> |  
            <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>  
        </td>  
    </tr>  
}  
</table>
```



Additional resources

- [Tag Helpers](#)
- [Globalization and localization](#)

PREVIOUS ADDING A

VIEW

NEXT WORKING WITH

SQL

Working with SQL Server LocalDB

11/29/2017 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#)

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the [Dependency Injection](#) container in the `ConfigureServices` method in the `Startup.cs` file:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The ASP.NET Core [Configuration](#) system reads the `ConnectionString`. For local development, it gets the connection string from the `appsettings.json` file:

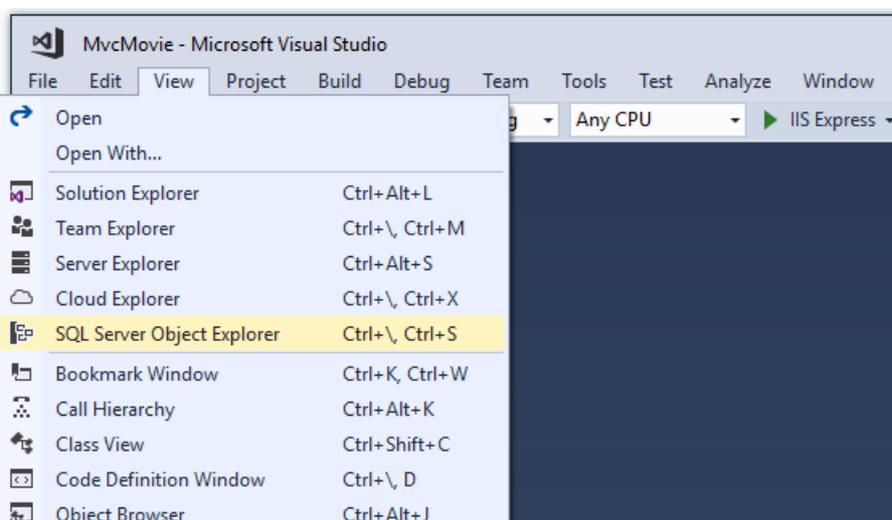
```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-2;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

When you deploy the app to a test or production server, you can use an environment variable or another approach to set the connection string to a real SQL Server. See [Configuration](#) for more information.

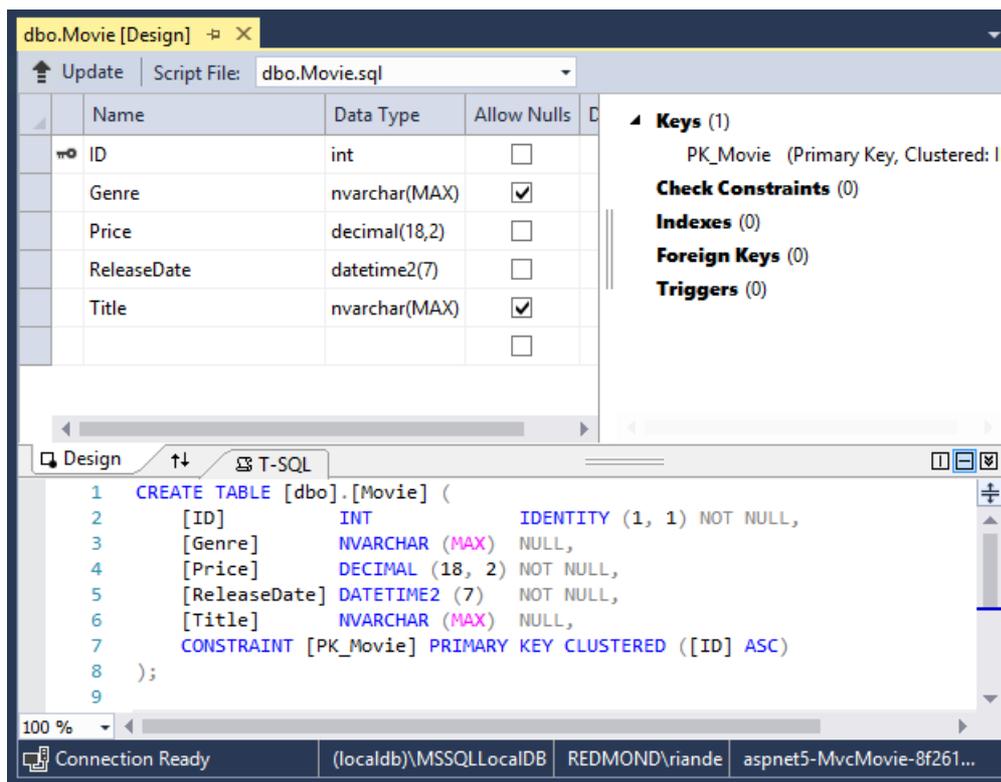
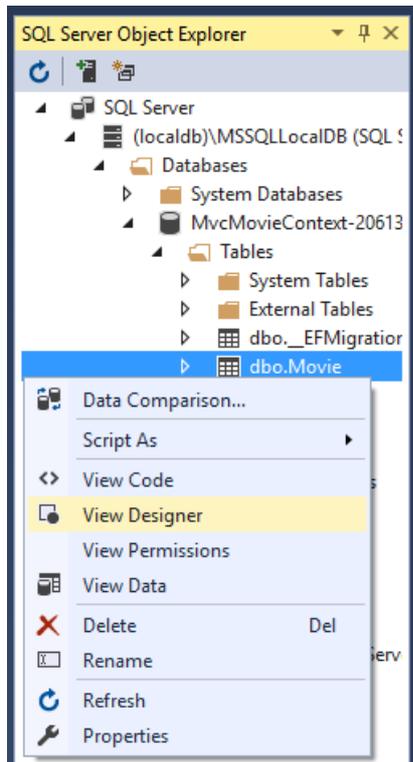
SQL Server Express LocalDB

LocalDB is a lightweight version of the SQL Server Express Database Engine that is targeted for program development. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:/Users/<user>` directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).

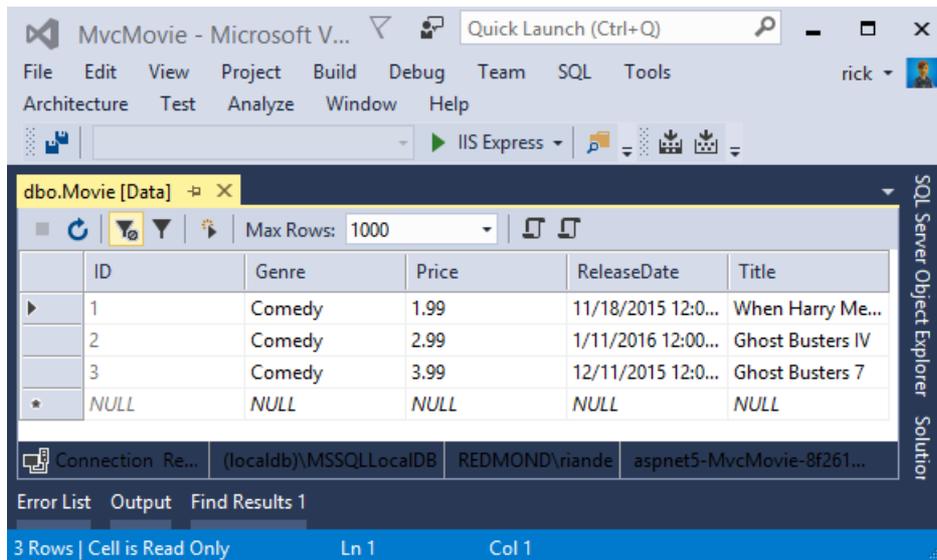
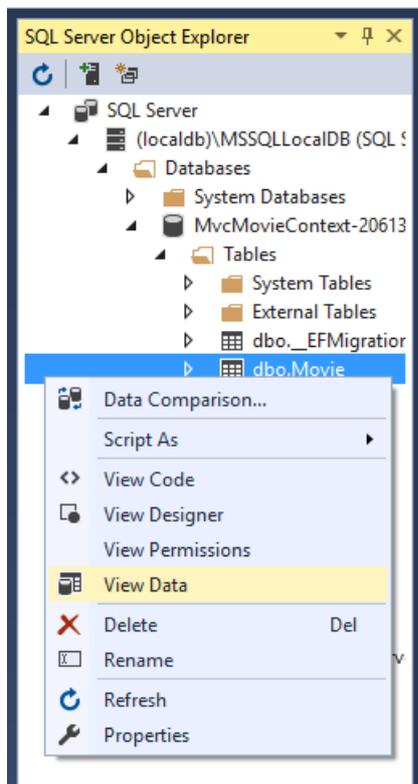


- Right click on the `Movie` table > **View Designer**



Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.

- Right click on the `Movie` table > **View Data**



Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-1-11"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}

```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

Add the seed initializer

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Add the seed initializer to the `Main` method in the `Program.cs` file:

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using MvcMovie.Models;
using System;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

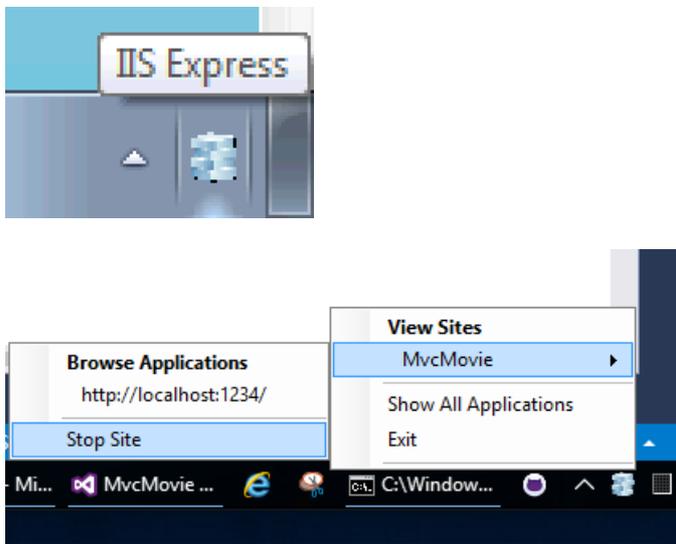
                try
                {
                    // Requires using MvcMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

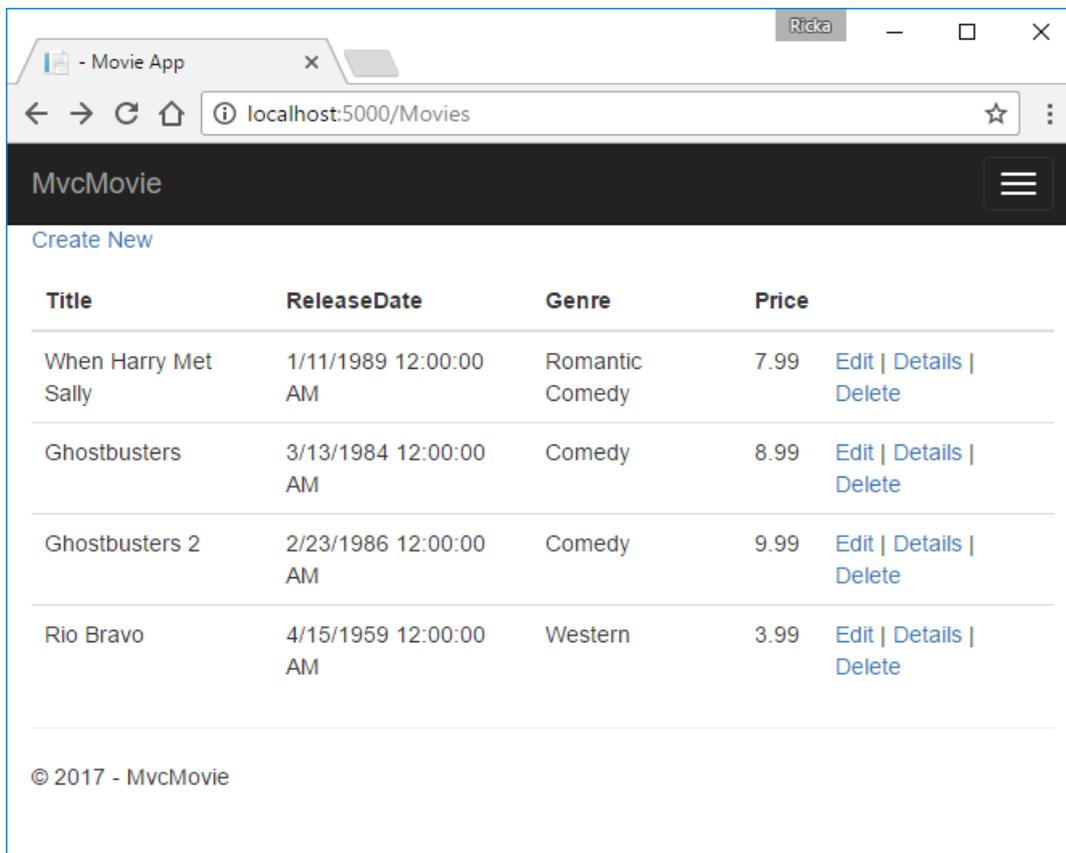
Test the app

- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:
 - Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**



- If you were running VS in non-debug mode, press F5 to run in debug mode
- If you were running VS in debug mode, stop the debugger and press F5

The app shows the seeded data.



PREVIOUS

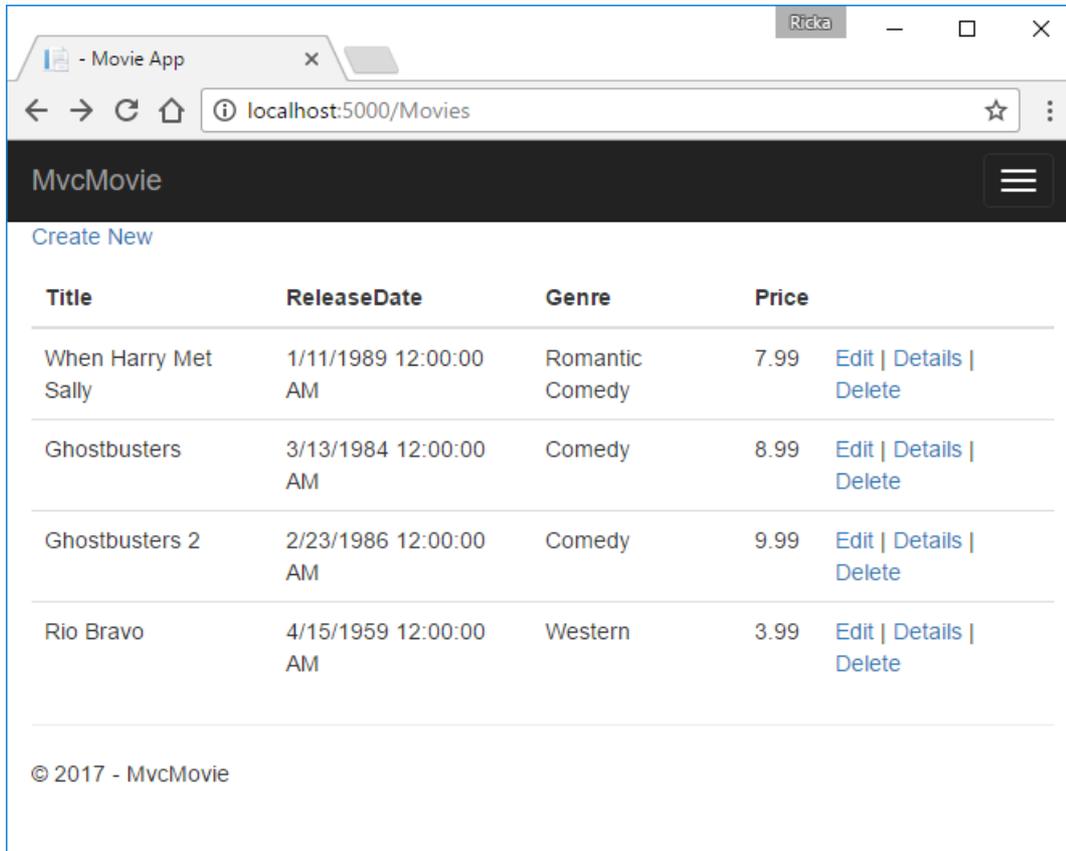
NEXT

Controller methods and views

7/5/2017 • 9 min to read • [Edit Online](#)

By [Rick Anderson](#)

We have a good start to the movie app, but the presentation is not ideal. We don't want to see the time (12:00:00 AM in the image below) and **ReleaseDate** should be two words.



Open the *Models/Movie.cs* file and add the highlighted lines shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

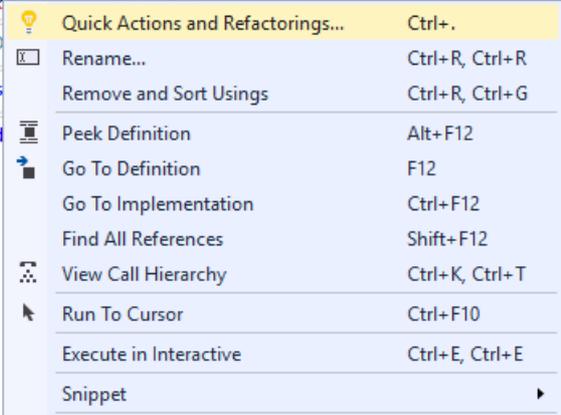
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

Right click on a red squiggly line > **Quick Actions and Refactorings**.

```
using System;
```

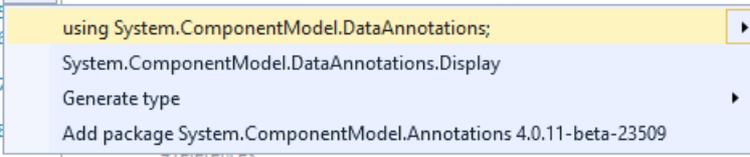
```
namespace MvcMovie.Models
{
    8 references | 0 changes | 0 authors, 0 changes
    public class Movie
    {
        7 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public int ID { get; set; }
        4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
    }
}
```



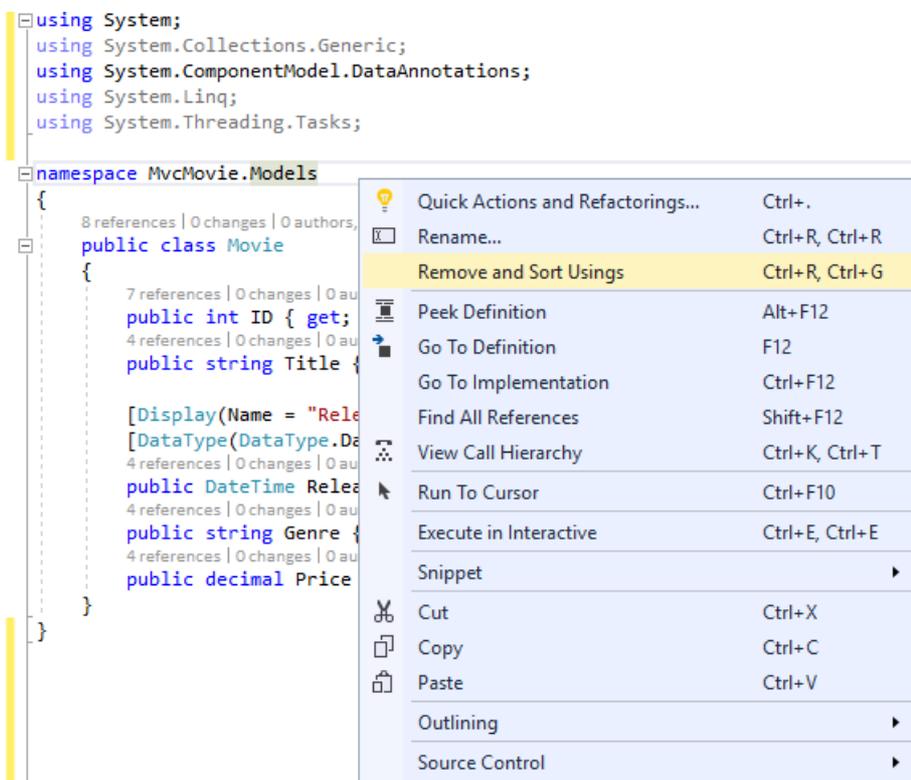
Tap `using System.ComponentModel.DataAnnotations;`

```
12         4 references
    public string Title { get; set; }
13
14     [Display(Name = "Release Date")]
15     [DataType(DataType.Date)]
16     using System.ComponentModel.DataAnnotations;
17     System.ComponentModel.DataAnnotations.Display
18     Generate type
19     Add package System.ComponentModel.Annotations 4.0.11-beta-23509
20     4 references
    public decimal Price { get; set; }
21 }
22
```



Visual studio adds `using System.ComponentModel.DataAnnotations;`

Let's remove the `using` statements that are not needed. They show up by default in a light grey font. Right click anywhere in the *Movie.cs* file > **Remove and Sort Usings**.



The updated code:

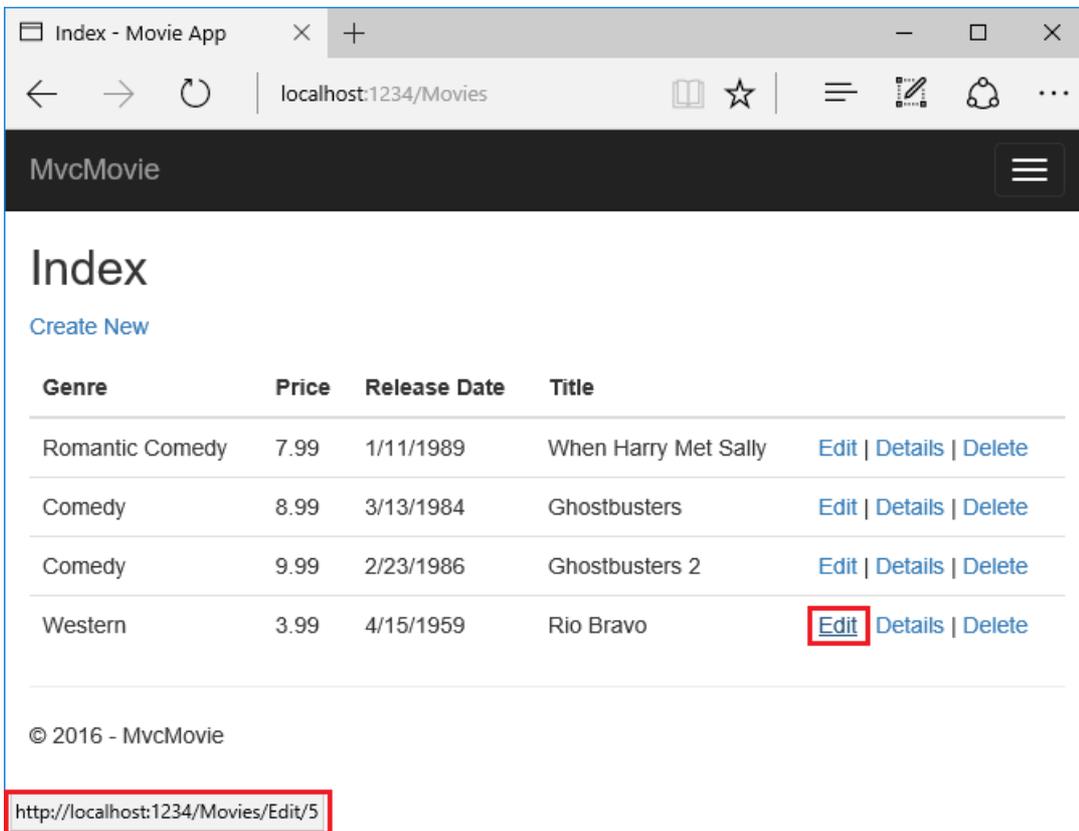
```
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

We'll cover [DataAnnotations](#) in the next tutorial. The [Display](#) attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The [DataType](#) attribute specifies the type of the data (Date), so the time information stored in the field is not displayed.

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Core MVC Anchor Tag Helper in the `Views/Movies/Index.cshtml` file.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
<a asp-action="Details" asp-route-id="@item.ID">Details</a> |
<a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
</tr>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

```
<td>
<a href="/Movies/Edit/4"> Edit </a> |
<a href="/Movies/Details/4"> Details </a> |
<a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for **routing** set in the `Startup.cs` file:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

ASP.NET Core translates `http://localhost:1234/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. See [Additional resources](#) for more

information.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the `Edit.cshtml` Razor file.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `[Bind]` attribute is one way to protect against [over-posting](#). You should only include properties in the `[Bind]` attribute that you want to change. See [Protect your controller from over-posting](#) for more information. [ViewModels](#) provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to [prevent forgery of a request](#) and is paired up with an anti-forgery token generated in the edit view file (*Views/Movies/Edit.cshtml*). The edit view file generates the anti-forgery token with the [Form Tag Helper](#).

```
<form asp-action="Edit">
```

The [Form Tag Helper](#) generates a hidden anti-forgery token that must match the `[ValidateAntiForgeryToken]` generated anti-forgery token in the `Edit` method of the `Movies` controller. For more information, see [Anti-Request Forgery](#).

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `SingleOrDefaultAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

```

@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <input type="hidden" asp-for="ID" />
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Genre" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Price" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file.

`@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The [Label Tag Helper](#) displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The [Input Tag Helper](#) renders an HTML `<input>` element. The [Validation Tag Helper](#) displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

```
<form action="/Movies/Edit/7" method="post">
  <div class="form-horizontal">
    <h4>Movie</h4>
    <hr />
    <div class="text-danger" />
    <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID"
value="7" />
    <div class="form-group">
      <label class="control-label col-md-2" for="Genre" />
      <div class="col-md-10">
        <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
        <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-
replace="true"></span>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-md-2" for="Price" />
      <div class="col-md-10">
        <input class="form-control" type="text" data-val="true" data-val-number="The field Price must
be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" />
        <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-
replace="true"></span>
      </div>
    </div>
    <!-- Markup removed for brevity -->
    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Save" class="btn btn-default" />
      </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIISduCRx9jDQC1rV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRd10ri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
  </form>
```

The `<input>` elements are in an `HTML <form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the `Save` button is clicked. The last line before the closing `</form>` element shows the hidden `XSRF` token generated by the [Form Tag Helper](#).

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

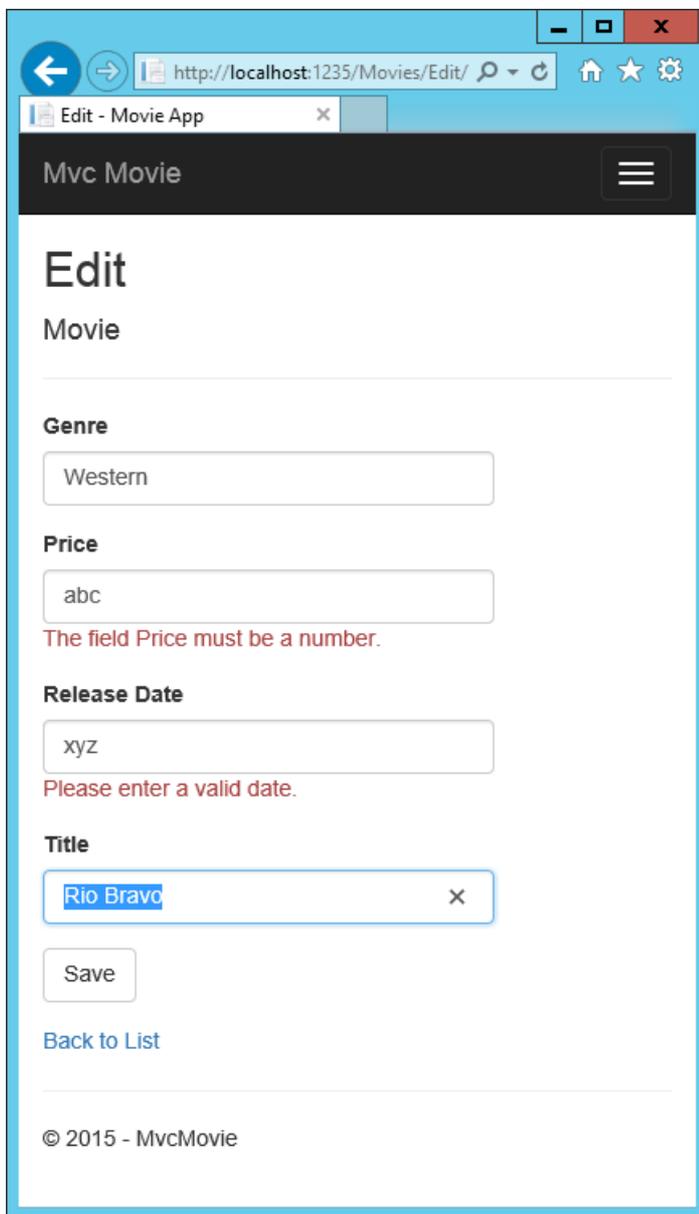
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

The `[ValidateAntiForgeryToken]` attribute validates the hidden [XSRF](#) token generated by the anti-forgery token generator in the [Form Tag Helper](#)

The [model binding](#) system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` method verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form is not posted. If JavaScript is disabled, you won't have client side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine [Model Validation](#) in more detail. The [Validation Tag Helper](#) in the `Views/Movies/Edit.cshtml` view template takes care of displaying appropriate error messages.



All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural [REST](#) pattern, which specifies that GET requests should not change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Authoring Tag Helpers](#)
- [Anti-Request Forgery](#)
- [Protect your controller from over-posting](#)
- [ViewModels](#)
- [Form Tag Helper](#)
- [Input Tag Helper](#)
- [Label Tag Helper](#)
- [Select Tag Helper](#)

- [Validation Tag Helper](#)

[PREVIOUS](#)

[NEXT](#)

Adding Search to an ASP.NET Core MVC app

7/5/2017 • 7 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this section you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method with the following code:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

The first line of the `Index` action method creates a [LINQ](#) query to select the movies:

```
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

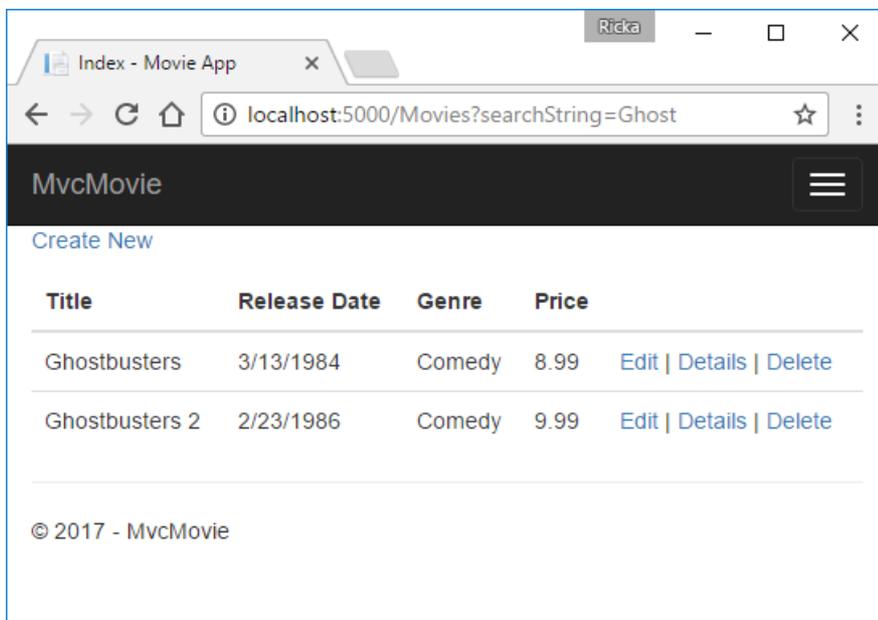
If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

The `s => s.Title.Contains()` code above is a [Lambda Expression](#). Lambdas are used in method-based [LINQ](#) queries as arguments to standard query operator methods such as the [Where](#) method or `Contains` (used in the code above). LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where`, `Contains` or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see [Query Execution](#).

Note: The `Contains` method is run on the database, not in the `c#` code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to [SQL LIKE](#), which is case insensitive. In SQLite, with the default collation, it's case sensitive.

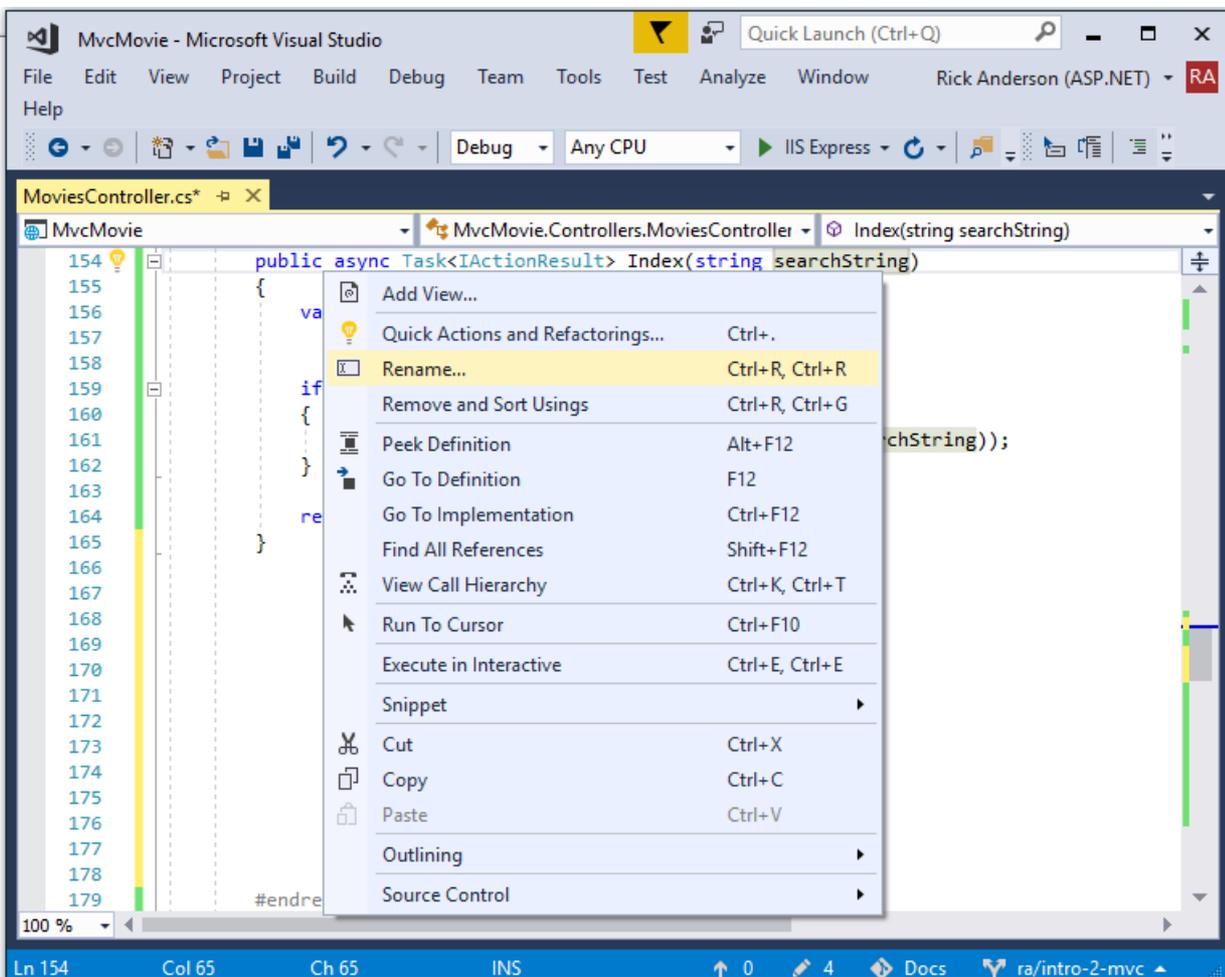
Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in `Startup.cs`.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

You can quickly rename the `searchString` parameter to `id` with the **rename** command. Right click on `searchString` > **Rename**.



The rename targets are highlighted.

```
public ActionResult Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

Change the parameter to `id` and all occurrences of `searchString` change to `id`.

```
public ActionResult Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(movies);
}
```

The previous `Index` method:

```

public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}

```

The updated `Index` method with `id` parameter:

```

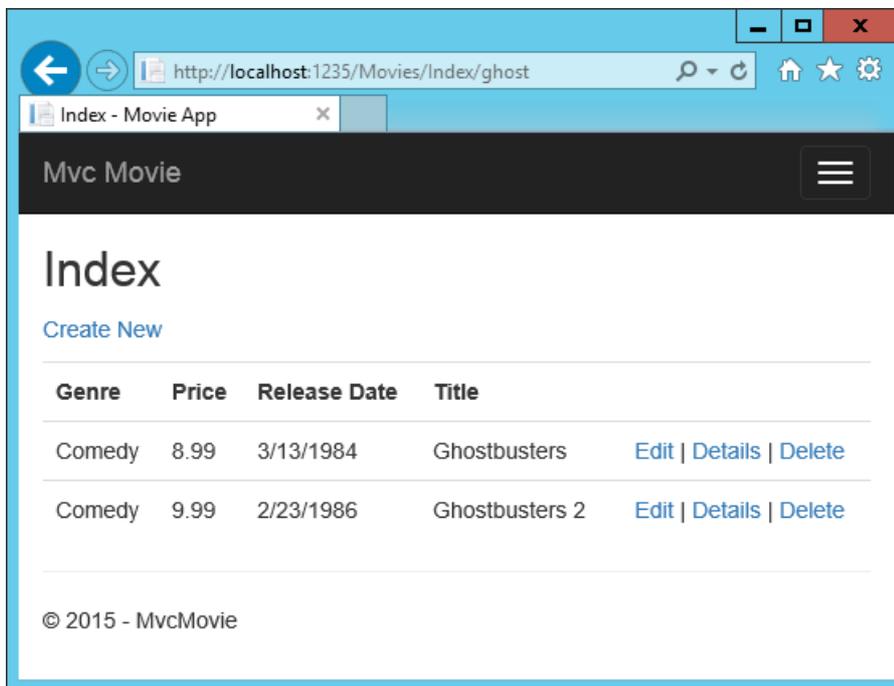
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}

```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

```

public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}

```

Open the `Views/Movies/Index.cshtml` file, and add the `<form>` markup highlighted below:

```

    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>

```

The HTML `<form>` tag uses the [Form Tag Helper](#), so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.

The screenshot shows a web browser window with the address bar at `localhost:1899/Movies`. The page displays the MvcMovie application. At the top, there is a "Create New" link. Below it is a search form with the label "Title:" and an input field containing the text "ghost". To the right of the input field is a "Filter" button. Below the search form is a table with the following columns: Genre, Price, Release Date, and Title. The table contains four rows of movie data:

Genre	Price	Release Date	Title
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally
Comedy	8.99	3/13/1984	Ghostbusters
Comedy	9.99	2/23/1986	Ghostbusters 2
Western	3.99	4/15/1959	Rio Bravo

Each row in the table has links for "Edit", "Details", and "Delete" next to the title. At the bottom of the page, there is a footer that reads "© 2016 - MvcMovie".

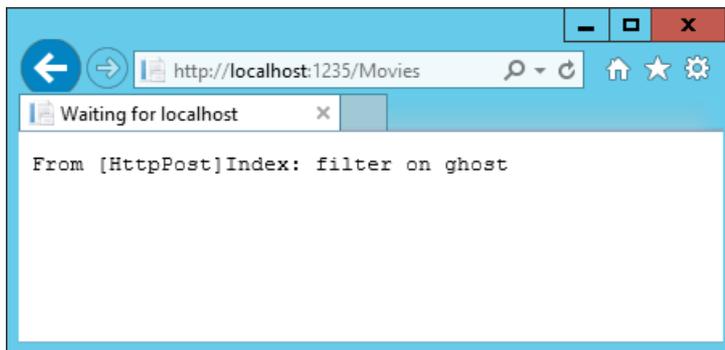
There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following `[HttpPost] Index` method.

```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (`localhost:xxxx/Movies/Index`) -- there's no search information in the URL. The search string information is sent to the server as a [form field value](#). You can verify that with the browser Developer tools or the excellent [Fiddler tool](#). The image below shows the Chrome browser Developer tools:

The screenshot shows a web browser window with the address bar at `localhost:5000/Movies`. The Network developer tool is open, displaying a POST request to `http://localhost:5000/Movies`. The status code is `200 OK`. The request headers include `Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8`, `Accept-Encoding: gzip, deflate, br`, `Accept-Language: en-US,en;q=0.8`, `Cache-Control: max-age=0`, `Connection: keep-alive`, `Content-Length: 201`, `Content-Type: application/x-www-form-urlencoded`, `Cookie: .AspNetCore.Antiforgery.txPTUN878m8=CfDJ8B98MxUFL5pAq2aeCj59HP3vvTrLPK1krIeXsJFchnazwJLRXzWQjTwt-tsEJDQr8bQg-xCdy7DbpfcQ-Hi2HAX0in1R838CvFU6oyWz7VIKmiKjUuXI371S-YZR0pBdNaP0mTxZX9JRMj_xj_x_ziig; .AspNetCore.Antiforgery.Mkg1_D_R5qY=CfDJ8B98MxUFL5pAq2aeCj59HP2tNs0B01ew8FztibCokKfe2wX09rL6Z-9YP41jarbKyJ_I5aQvz9BMWGFpPwwbH71Jw4I8qgC-CkWyxAsFwKQyP0MYsYab98gk-z_M4jH1_Hw9OD8ZJuBVS0fzF4tM8`, `Host: localhost:5000`, `Origin: http://localhost:5000`, `Referer: http://localhost:5000/Movies`, and `Upgrade-Insecure-Requests: 1`. The user agent is `Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36`. The form data includes `SearchString: Ghost` and a long `_RequestVerificationToken`.

You can see the search parameter and [XSRF](#) token in the request body. Note, as mentioned in the previous tutorial, the [Form Tag Helper](#) generates an [XSRF](#) anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. We'll fix this by specifying the request should be `HTTP GET`.

Notice how IntelliSense helps us update the markup.

```
<form asp-controller="Movies" asp-action="Index" m>
  <p>
    Title: <input type="text" name="SearchStr" />
    <input type="submit" value="Filter" />
  </p>
</form>
```

- autocomplete
- contextmenu
- itemid
- itemprop
- itemref
- itemscope
- itemtype
- method
- name

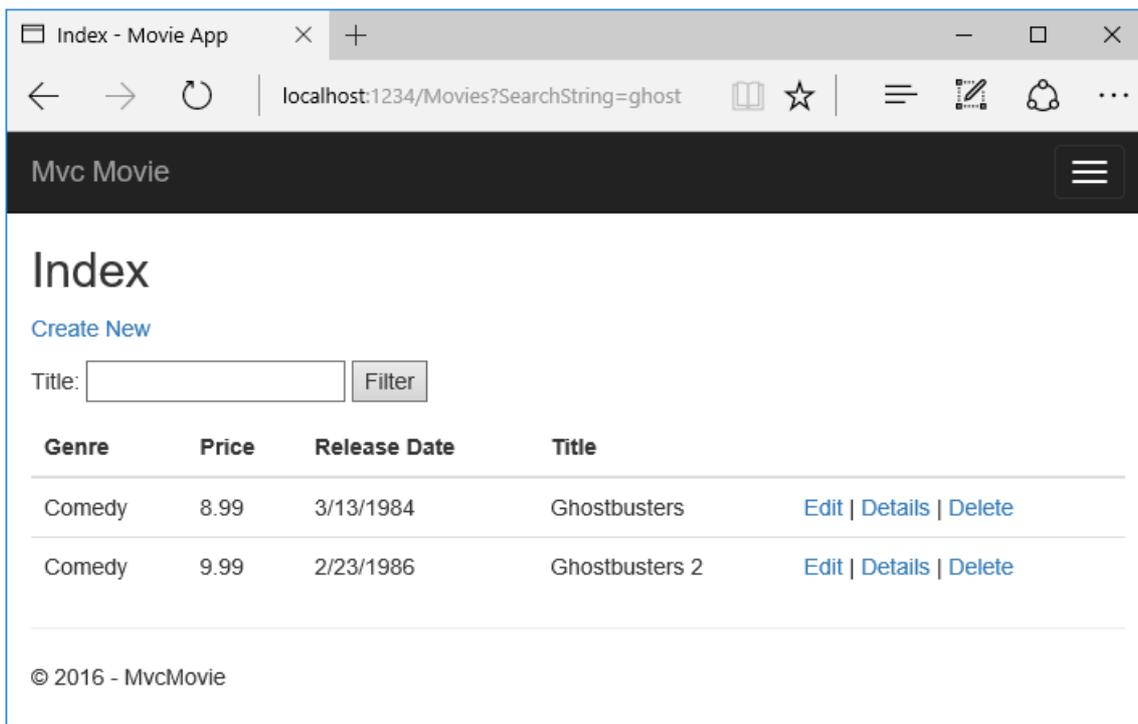
```
<form asp-controller="Movies" asp-action="Index" method="get">
  <p>
    Title: <input type="text" name="SearchString">
    <input type="submit" value="Filter" />
  </p>
</form>
```

- delete
- get
- post
- put

Notice the distinctive font in the `<form>` tag. That distinctive font indicates the tag is supported by [Tag Helpers](#).

```
<form asp-controller="Movies" asp-action="Index">
  <p>
    Title: <input type="text" name="SearchString">
    <input type="submit" value="Filter" />
  </p>
</form>
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



The following markup shows the change to the `form` tag:

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Adding Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

```

using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> movies;
        public SelectList genres;
        public string movieGenre { get; set; }
    }
}

```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This will allow the user to select a genre from the list.
- `movieGenre`, which contains the selected genre.

Replace the `Index` method in `MoviesController.cs` with the following code:

```

// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel();
    movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    movieGenreVM.movies = await movies.ToListAsync();

    return View(movieGenreVM);
}

```

The following code is a `LINQ` query that retrieves all the genres from the database.

```

// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;

```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

```
movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync())
```

Adding search by genre to the Index view

Update `Index.cshtml` as follows:

```

@model MvcMovie.Models.MovieGenreViewModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <select asp-for="movieGenre" asp-items="Model.genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.movies[0].Title)
```

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.movies`, or `model.movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

Test the app by searching by genre, by movie title, and by both.

[PREVIOUS](#)[NEXT](#)

Adding a New Field

10/6/2017 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this section you'll use [Entity Framework Code First Migrations](#) to add a new field to the model and migrate that change to the database.

When you use EF Code First to automatically create a database, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app (Ctrl+Shift+B).

Because you've added a new field to the `Movie` class, you also need to update the binding white list so this new property will be included. In *MoviesController.cs*, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

You also need to update the view templates in order to display, create and edit the new `Rating` property in the browser view.

Edit the */Views/Movies/Index.cshtml* file and add a `Rating` field:

```

<table class="table">
  <thead>
    <tr>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Title)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Genre)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Price)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Rating)
      </th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model.movies)
    {
      <tr>
        <td>
          @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Rating)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Rating)
        </td>
      </tr>
    }
  </tbody>
</table>

```

Update the `/Views/Movies/Create.cshtml` with a `Rating` field. You can copy/paste the previous "form group" and let IntelliSense help you update the fields. IntelliSense works with [Tag Helpers](#). Note: In the RTM version of Visual Studio 2017 you need to install the [Razor Language Services](#) for Razor IntelliSense. This will be fixed in the next release.

```

</div>
<div class="form-group">
  <label asp-for="Title" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="Title" class="form-control" />
    <span asp-validation-for="Title" class="text-danger" />
  </div>
</div>
<div class="form-group">
  <label asp-for="Rating" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="Rating" class="form-control" />
    <span asp-validation-for="Rating" class="text-danger" />
  </div>
</div>
<div class="form-group">
  <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input type="text" class="form-control" />
    <span class="text-danger" />
  </div>
</div>
</div>
</form>

```



The app won't work until we update the DB to include the new field. If you run it now, you'll get the following

```
SQLException :
```

```
SQLException: Invalid column name 'Rating'.
```

You're seeing this error because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no Rating column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you are doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

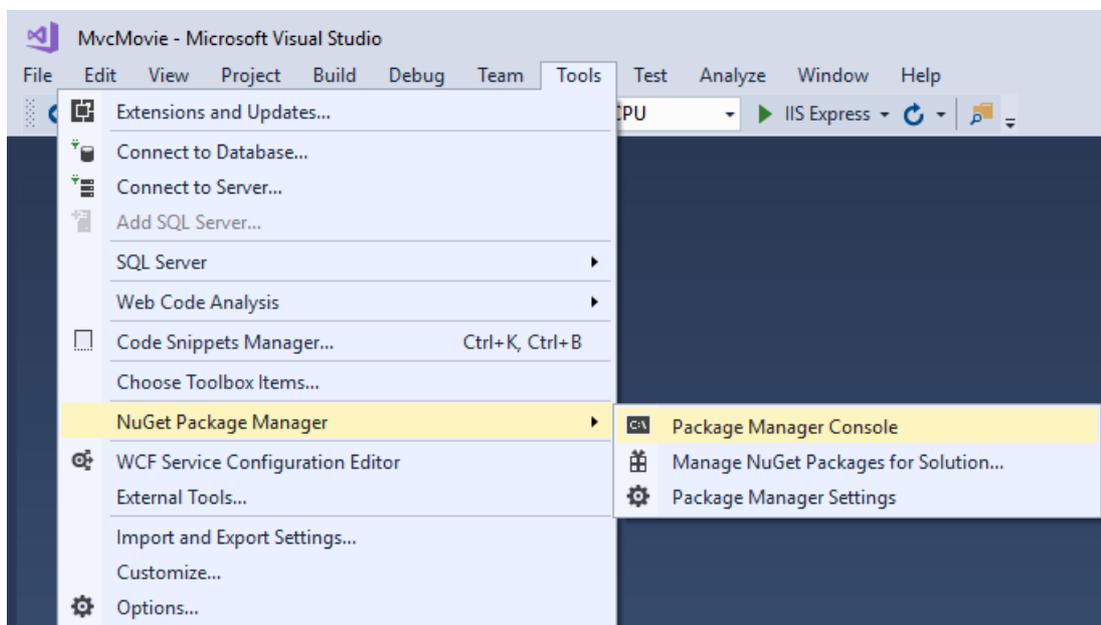
For this tutorial, we'll use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

Build the solution.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model. The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If you delete all the records in the DB, the initialize will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from SSOX.

Run the app and verify you can create/edit/display movies with a `Rating` field. You should also add the `Rating` field to the `Edit`, `Details`, and `Delete` view templates.

[PREVIOUS](#)

[NEXT](#)

Adding validation

9/22/2017 • 9 min to read • [Edit Online](#)

By [Rick Anderson](#)

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is **DRY** ("Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Adding validation rules to the movie model

Open the `Movie.cs` file. `DataAnnotations` provides a built-in set of validation attributes that you apply declaratively to any class or property. (It also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.)

Update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [DisplayName = "Release Date"]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-\s]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-\s]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they are applied to.

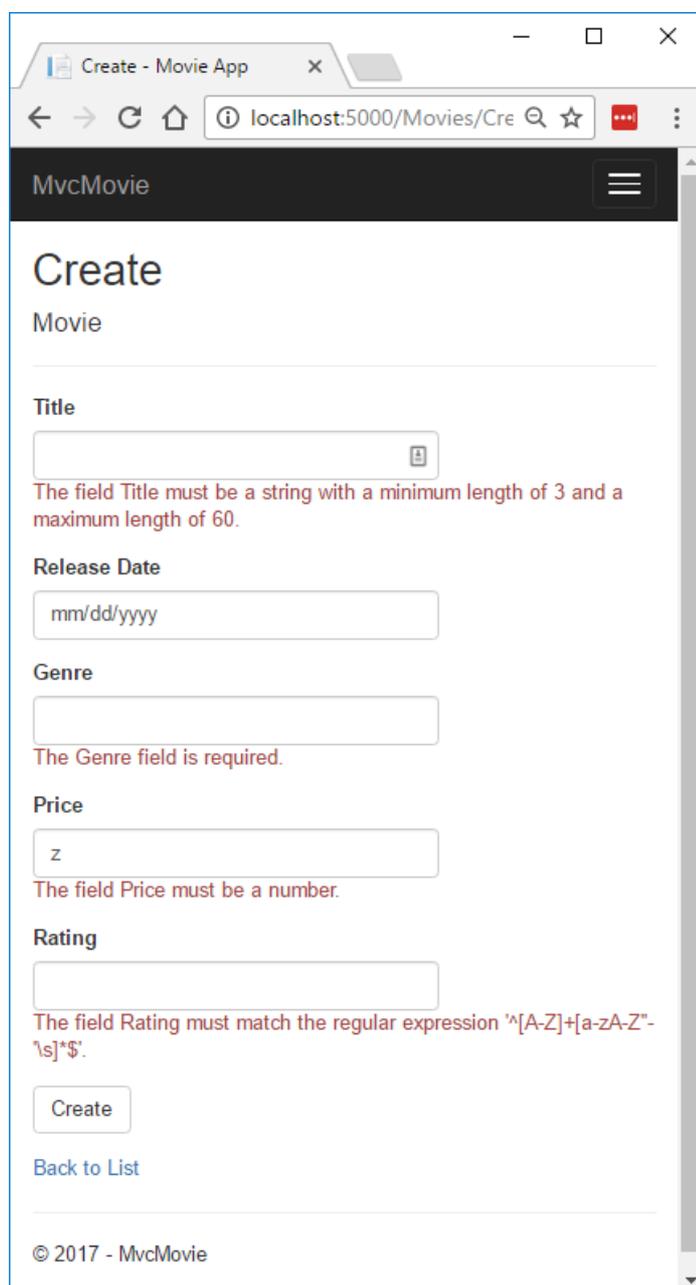
The `Required` and `MinimumLength` attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation. The `RegularExpression` attribute is used to limit what characters can be input. In the code above, `Genre` and `Rating` must use only letters (white space, numbers and special characters are not allowed). The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI in MVC

Run the app and navigate to the Movies controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



The screenshot shows a web browser window with the address bar at `localhost:5000/Movies/Cre`. The page title is "MvcMovie" and the main heading is "Create Movie". The form contains several fields with validation errors:

- Title:** The field is empty. The error message is: "The field Title must be a string with a minimum length of 3 and a maximum length of 60."
- Release Date:** The field contains the placeholder text "mm/dd/yyyy".
- Genre:** The field is empty. The error message is: "The Genre field is required."
- Price:** The field contains the character "z". The error message is: "The field Price must be a number."
- Rating:** The field is empty. The error message is: "The field Rating must match the regular expression `^[A-Z]+[a-zA-Z'-\s]*$`."

At the bottom of the form, there is a "Create" button and a "Back to List" link. The footer of the page reads "© 2017 - MvcMovie".

NOTE

You may not be able to enter decimal commas in the `Price` field. To support [jQuery validation](#) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. This [GitHub issue 4076](#) for instructions on adding decimal comma.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the `Create.cshtml` view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data is not sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the [Fiddler tool](#), or the [F12 Developer tools](#).

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form is not posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.


```

74     // POST: Movies/Create
75     // To protect from overposting attacks, please enable t
76     // more details see http://go.microsoft.com/fwlink/?Lin
77     [HttpPost]
78     [ValidateAntiForgeryToken]
79     0 references
80     public async Task<IActionResult> Create([Bind("ID,Title
81     {
82         if (ModelState.IsValid)
83         {
84             _context.Add(movie);
85             await _context.SaveChangesAsync();
86             return RedirectToAction("Index");
87         }
88         return View(movie);
89     }

```

Below is portion of the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

```

<form asp-action="Create">
  <div class="form-horizontal">
    <h4>Movie</h4>
    <hr />

    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
      <label asp-for="Title" class="col-md-2 control-label"></label>
      <div class="col-md-10">
        <input asp-for="Title" class="form-control" />
        <span asp-validation-for="Title" class="text-danger"></span>
      </div>
    </div>

    @*Markup removed for brevity.*@
  </div>
</form>

```

The [Input Tag Helper](#) uses the [DataAnnotations](#) attributes and produces HTML attributes needed for jQuery Validation on the client side. The [Validation Tag Helper](#) displays validation errors. See [Validation](#) for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the *Movie.cs* file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supply attributes such as `<a>` for URL's and `` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that is more specific than the database intrinsic type, they are not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably do not want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

NOTE

jQuery validation does not work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [DisplayName = "Release Date", DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-\s]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-\s]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- [Working with Forms](#)
- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Authoring Tag Helpers](#)

[PREVIOUS](#)[NEXT](#)

Examining the Details and Delete methods

11/6/2017 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#)

Open the Movie controller and examine the `Details` method:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method and an `id` value. Recall these segments are defined in *Startup.cs*.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

EF makes it easy to search for data using the `SingleOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:xxxx/Movies/Details/1` to something like `http://localhost:xxxx/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you did not check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

```

// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}

```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```

// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{

```

```

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{

```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes `/Delete/` for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

```
// POST: Movies/Delete/6
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Publish to Azure

See [Publish an ASP.NET Core web app to Azure App Service using Visual Studio](#) for instructions on how to publish this app to Azure using Visual Studio. The app can also be published from the [command line](#).

Thanks for completing this introduction to ASP.NET Core MVC. We appreciate any comments you leave. [Getting started with MVC and EF Core](#) is an excellent follow up to this tutorial.

[PREVIOUS](#)

Working with Data in ASP.NET Core

1/10/2018 • 1 min to read • [Edit Online](#)

- [Get started with Razor Pages and Entity Framework Core using Visual Studio](#)
 - [Get started with Razor Pages and EF](#)
 - [Create, Read, Update, and Delete operations](#)
 - [Sort, filter, page, and group](#)
 - [Migrations](#)
 - [Create a complex data model](#)
 - [Read related data](#)
 - [Update related data](#)
 - [Handle concurrency conflicts](#)
- [Get started with ASP.NET Core MVC and Entity Framework Core using Visual Studio](#)
 - [Get started](#)
 - [Create, Read, Update, and Delete operations](#)
 - [Sort, filter, page, and group](#)
 - [Migrations](#)
 - [Create a complex data model](#)
 - [Read related data](#)
 - [Update related data](#)
 - [Handle concurrency conflicts](#)
 - [Inheritance](#)
 - [Advanced topics](#)
- [ASP.NET Core with EF Core - new database](#) (Entity Framework Core documentation site)
- [ASP.NET Core with EF Core - existing database](#) (Entity Framework Core documentation site)
- [Get started with ASP.NET Core and Entity Framework 6](#)
- [Azure Storage](#)
 - [Add Azure Storage by using Visual Studio Connected Services](#)
 - [Get started with Azure Blob storage and Visual Studio Connected Services](#)
 - [Get started with Queue Storage and Visual Studio Connected Services](#)
 - [Get started with Azure Table Storage and Visual Studio Connected Services](#)

Getting started with Razor Pages and Entity Framework Core using Visual Studio (1 of 8)

1/8/2018 • 17 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web app demonstrates how to create ASP.NET Core 2.0 MVC web applications using Entity Framework (EF) Core 2.0 and Visual Studio 2017.

The sample app is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This page is the first in a series of tutorials that explain how to build the Contoso University sample app.

[Download or view the completed app.](#) [Download instructions.](#)

Prerequisites

Install the following:

- [.NET Core 2.0.0 SDK](#) or later.
- [Visual Studio 2017](#) version 15.3 or later with the **ASP.NET and web development** workload.

Familiarity with [Razor Pages](#). New programmers should complete [Get started with Razor Pages](#) before starting this series.

Troubleshooting

If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the [completed stage](#) or [completed project](#). For a list of common errors and how to solve them, see [the Troubleshooting section of the last tutorial in the series](#). If you don't find what you need there, you can post a question to StackOverflow.com for [ASP.NET Core](#) or [EF Core](#).

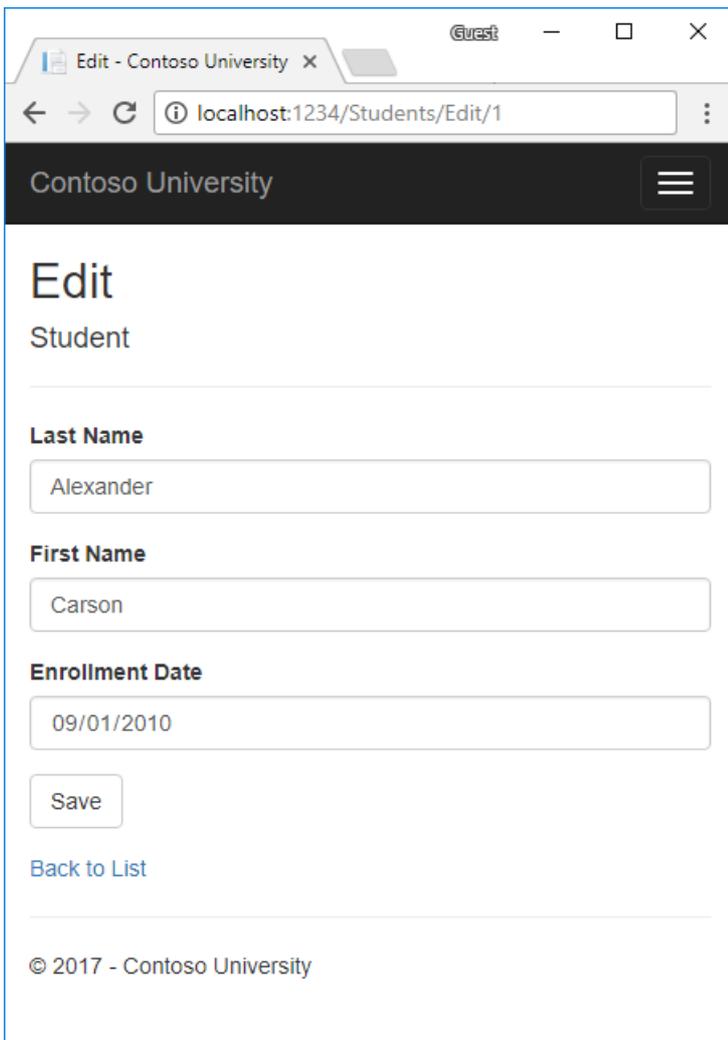
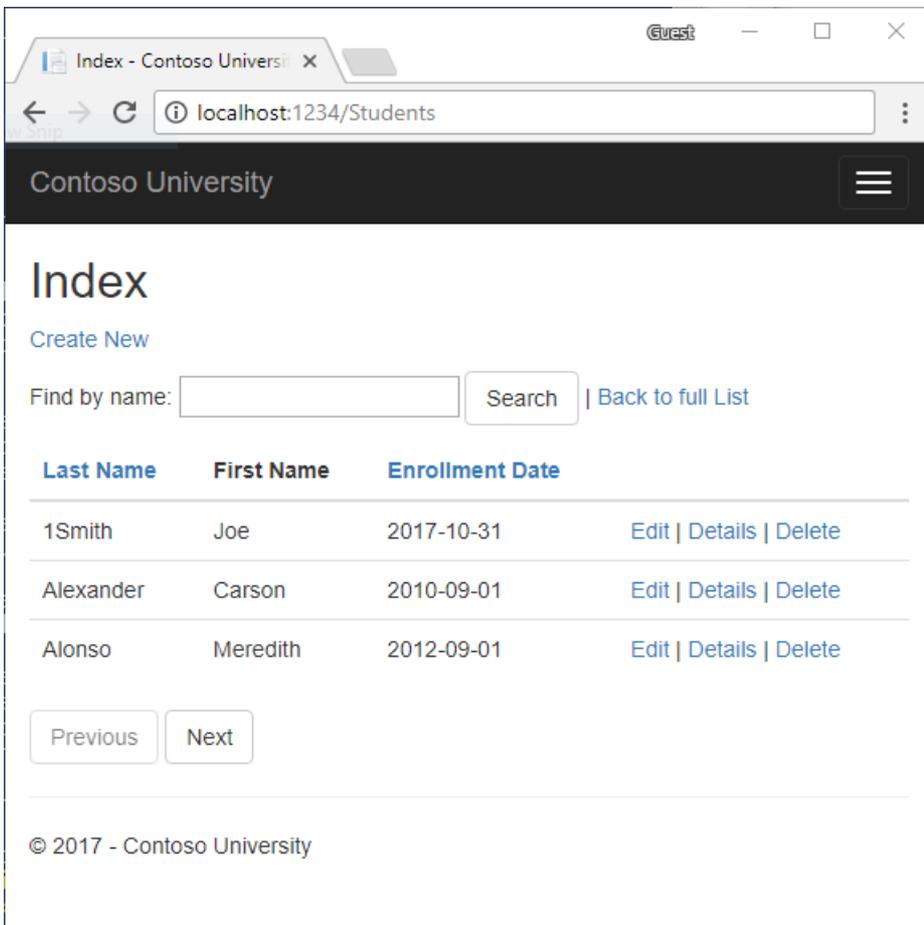
TIP

This series of tutorials builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. If you run into problems, you can start over from the previous tutorial instead of going back to the beginning. Alternatively, you can download a [completed stage](#) and start over using the completed stage.

The Contoso University web app

The app built in these tutorials is a basic university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.

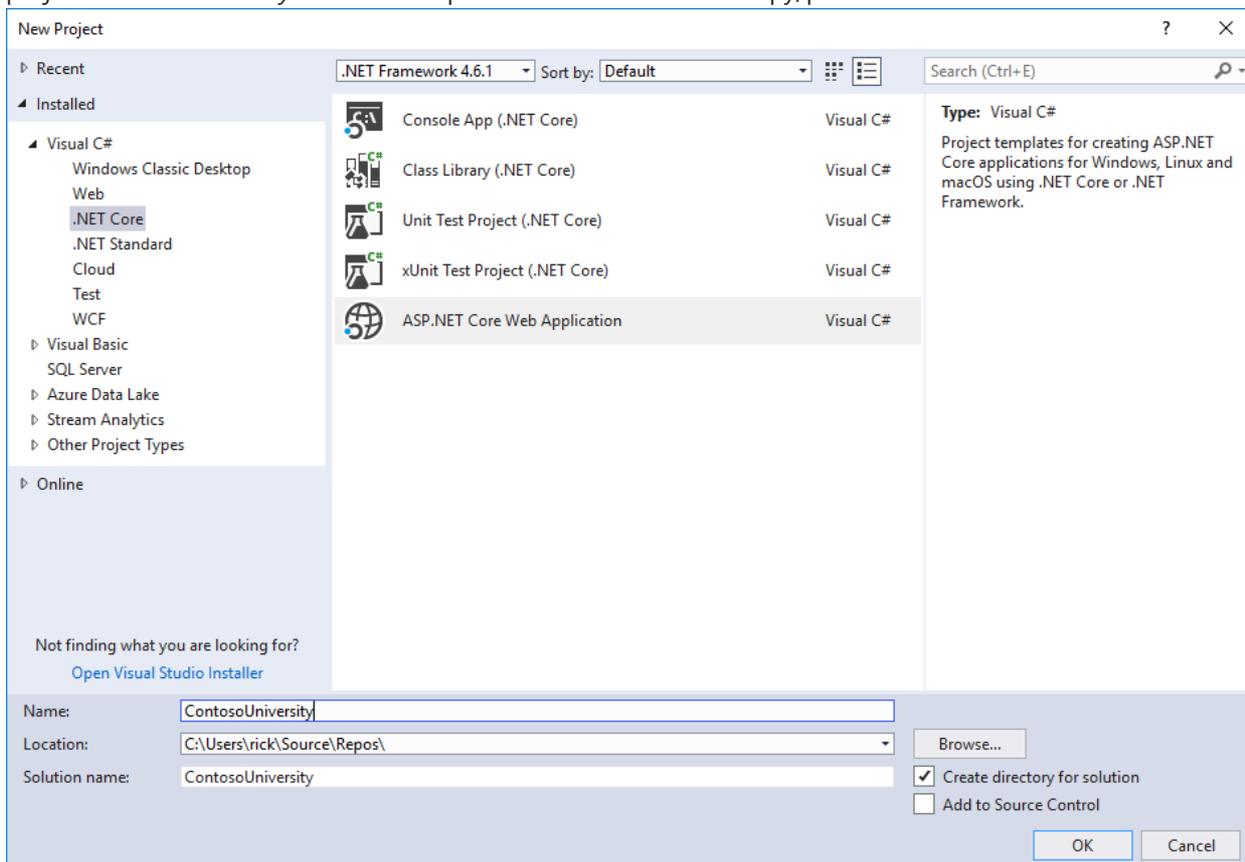


The UI style of this site is close to what's generated by the built-in templates. The tutorial focus is on EF Core with

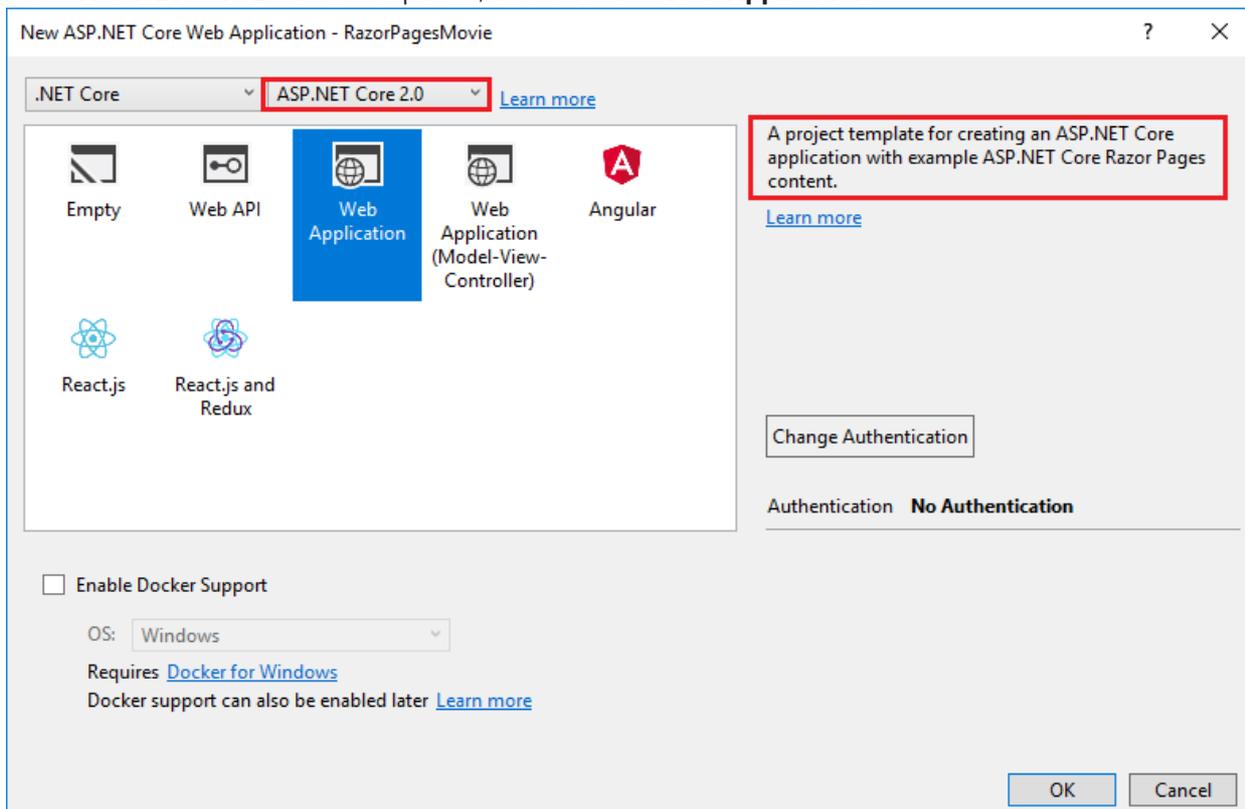
Razor Pages, not the UI.

Create a Razor Pages web app

- From the Visual Studio **File** menu, select **New > Project**.
- Create a new ASP.NET Core Web Application. Name the project **ContosoUniversity**. It's important to name the project *ContosoUniversity* so the namespaces match when code is copy/pasted.



- Select **ASP.NET Core 2.0** in the dropdown, and then select **Web Application**.



Press **F5** to run the app in debug mode or **Ctrl-F5** to run without attaching the debugger

Set up the site style

A few changes set up the site menu, layout, and home page.

Open `Pages/_Layout.cshtml` and make the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University." There are three occurrences.
- Add menu entries for **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Contact** menu entry.

The changes are highlighted. (All the markup is *not* displayed.)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - Contoso University</title>

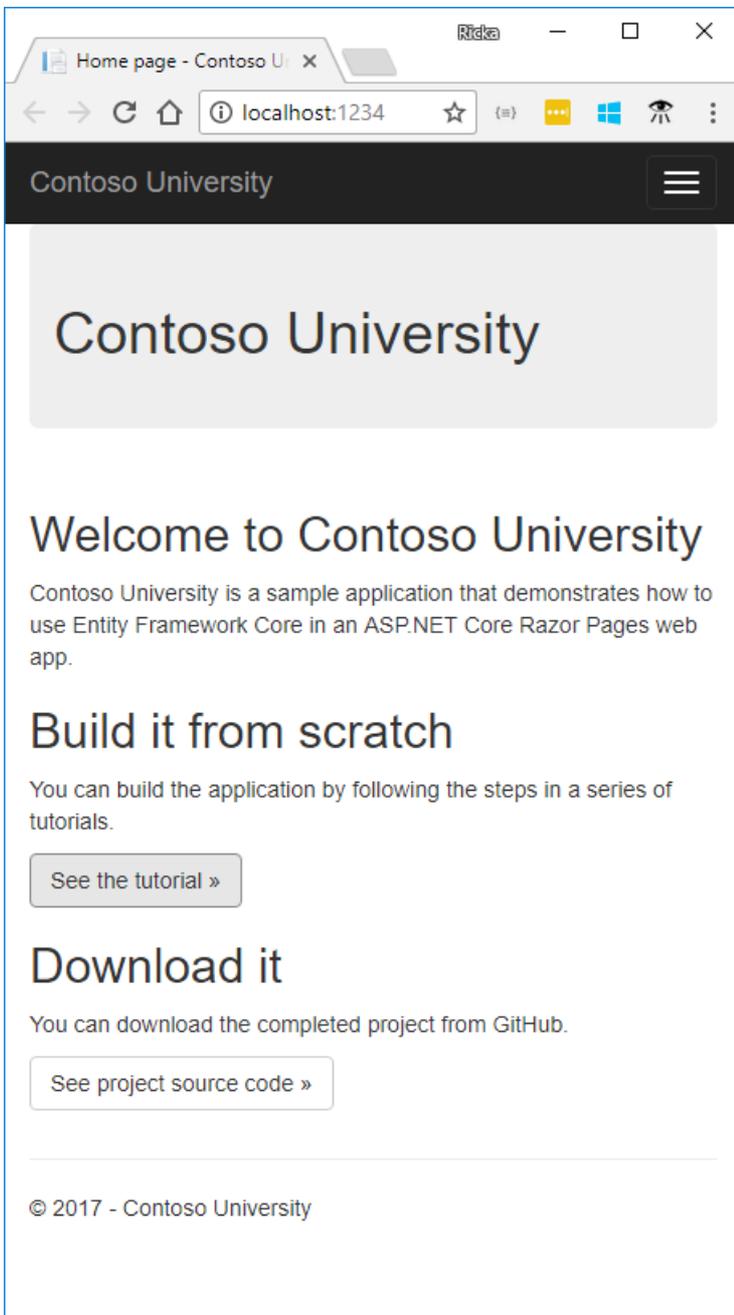
  <environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment exclude="Development">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>
</head>
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-page="/Index" class="navbar-brand">Contoso University</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-page="/Index">Home</a></li>
          <li><a asp-page="/About">About</a></li>
          <li><a asp-page="/Students/Index">Students</a></li>
          <li><a asp-page="/Courses/Index">Courses</a></li>
          <li><a asp-page="/Instructors/Index">Instructors</a></li>
          <li><a asp-page="/Departments/Index">Departments</a></li>
        </ul>
      </div>
    </div>
  </nav>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; 2017 - Contoso University</p>
    </footer>
  </div>
```

In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this app:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

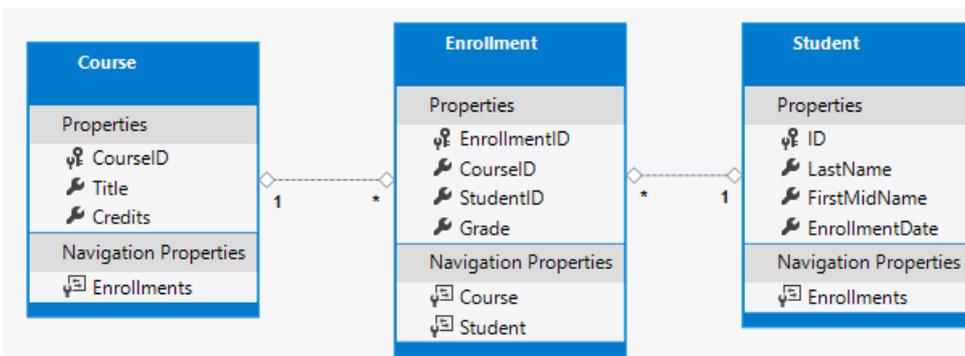
<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core Razor Pages web app.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default"
            href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro">
            See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
            href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-rp/intro/samples/cu-final">
            See project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project. The home page is displayed with tabs created in the following tutorials:



Create the data model

Create entity classes for the Contoso University app. Start with the following three entities:



There's a one-to-many relationship between `Student` and `Enrollment` entities. There's a one-to-many relationship between `Course` and `Enrollment` entities. A student can enroll in any number of courses. A course can have any number of students enrolled in it.

In the following sections, a class for each one of these entities is created.

The Student entity

Student
Properties
PK ID
LastName
FirstMidName
EnrollmentDate
Navigation Properties
Enrollments

Create a *Models* folder. In the *Models* folder, create a class file named *Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property becomes the primary key column of the database (DB) table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key.

The `Enrollments` property is a navigation property. Navigation properties link to other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that `Student`. For example, if a `Student` row in the DB has two related `Enrollment` rows, the `Enrollments` navigation property contains those two `Enrollment` entities. A related `Enrollment` row is a row that contains that student's primary key value in the `StudentID` column. For example, suppose the student with `ID=1` has two rows in the `Enrollment` table. The `Enrollment` table has two rows with `StudentID = 1`. `StudentID` is a foreign key in the `Enrollment` table that specifies the student in the `Student` table.

If a navigation property can hold multiple entities, the navigation property must be a list type, such as `ICollection<T>`. `ICollection<T>` can be specified, or a type such as `List<T>` or `HashSet<T>`. When `ICollection<T>` is used, EF Core creates a `HashSet<T>` collection by default. Navigation properties that hold multiple entities come from many-to-many and one-to-many relationships.

The Enrollment entity

Enrollment
Properties
PK EnrollmentID
CourseID
StudentID
Grade
Navigation Properties
Course
Student

In the *Models* folder, create *Enrollment.cs* with the following code:

```

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}

```

The `EnrollmentID` property is the primary key. This entity uses the `classnameID` pattern instead of `ID` like the `Student` entity. Typically developers choose one pattern and use it throughout the data model. In a later tutorial, using ID without classname is shown to make it easier to implement inheritance in the data model.

The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity. The `Student` entity differs from the `Student.Enrollments` navigation property, which contains multiple `Enrollment` entities.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity

Course	
Properties	
	CourseID
	Title
	Credits
Navigation Properties	
	Enrollments

In the `Models` folder, create `Course.cs` with the following code:

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the DB generate it.

Create the SchoolContext DB context

The main class that coordinates EF Core functionality for a given data model is the DB context class. The data context is derived from `Microsoft.EntityFrameworkCore.DbContext`. The data context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named *Data*.

In the *Data* folder create *SchoolContext.cs* with the following code:

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}

```

This code creates a `DbSet` property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a DB table.
- An entity corresponds to a row in the table.

`DbSet<Enrollment>` and `DbSet<Course>` can be omitted. EF Core includes them implicitly because the `Student` entity references the `Enrollment` entity, and the `Enrollment` entity references the `Course` entity. For this tutorial, keep `DbSet<Enrollment>` and `DbSet<Course>` in the `SchoolContext`.

When the DB is created, EF Core creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural (Students rather than Student). Developers disagree about whether table names should be plural. For these tutorials, the default behavior is overridden by specifying singular table names in

the DbContext. To specify singular table names, add the following highlighted code:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Register the context with dependency injection

ASP.NET Core includes [dependency injection](#). Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a db context instance is shown later in the tutorial.

To register `SchoolContext` as a service, open `Startup.cs`, and add the highlighted lines to the `ConfigureServices` method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the [ASP.NET Core configuration system](#) reads the connection string from the `appsettings.json` file.

Add `using` statements for `ContosoUniversity.Data` and `Microsoft.EntityFrameworkCore` namespaces. Build the project.

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

Open the `appsettings.json` file and add a connection string as shown in the following code:

```

{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\mssqllocaldb;Database=ContosoUniversity1;ConnectRetryCount=0;Trusted_Connection=True;MultipleActiveR
esultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
}

```

The preceding connection string uses `ConnectRetryCount=0` to prevent [SQLClient](#) from hanging.

SQL Server Express LocalDB

The connection string specifies a SQL Server LocalDB DB. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB creates `.mdf` DB files in the `C:/Users/<user>` directory.

Add code to initialize the DB with test data

EF Core creates an empty DB. In this section, a *Seed* method is written to populate it with test data.

In the *Data* folder, create a new class file named *DbInitializer.cs* and add the following code:

```

using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return; // DB has been seeded
            }

            var students = new Student[]
            {
                new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();
        }
    }
}

```

```

var courses = new Course[]
{
    new Course{CourseID=1050,Title="Chemistry",Credits=3},
    new Course{CourseID=4022,Title="Microeconomics",Credits=3},
    new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
    new Course{CourseID=1045,Title="Calculus",Credits=4},
    new Course{CourseID=3141,Title="Trigonometry",Credits=4},
    new Course{CourseID=2021,Title="Composition",Credits=3},
    new Course{CourseID=2042,Title="Literature",Credits=4}
};
foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment{StudentID=1, CourseID=1050, Grade=Grade.A},
    new Enrollment{StudentID=1, CourseID=4022, Grade=Grade.C},
    new Enrollment{StudentID=1, CourseID=4041, Grade=Grade.B},
    new Enrollment{StudentID=2, CourseID=1045, Grade=Grade.B},
    new Enrollment{StudentID=2, CourseID=3141, Grade=Grade.F},
    new Enrollment{StudentID=2, CourseID=2021, Grade=Grade.F},
    new Enrollment{StudentID=3, CourseID=1050},
    new Enrollment{StudentID=4, CourseID=1050},
    new Enrollment{StudentID=4, CourseID=4022, Grade=Grade.F},
    new Enrollment{StudentID=5, CourseID=4041, Grade=Grade.C},
    new Enrollment{StudentID=6, CourseID=1045},
    new Enrollment{StudentID=7, CourseID=3141, Grade=Grade.A},
};
foreach (Enrollment e in enrollments)
{
    context.Enrollments.Add(e);
}
context.SaveChanges();
}
}
}

```

The code checks if there are any students in the DB. If there are no students in the DB, the DB is seeded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

The `EnsureCreated` method automatically creates the DB for the DB context. If the DB exists, `EnsureCreated` returns without modifying the DB.

In *Program.cs*, modify the `Main` method to do the following:

- Get a DB context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method completes.

The following code shows the updated *Program.cs* file.

```

// Unused usings removed
using System;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred while seeding the database.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

The first time the app is run, the DB is created and seeded with test data. When the data model is updated:

- Delete the DB.
- Update the seed method.
- Run the app and a new seeded DB is created.

In later tutorials, the DB is updated when the data model changes, without deleting and re-creating the DB.

Add scaffold tooling

In this section, the Package Manager Console (PMC) is used to add the Visual Studio web code generation package. This package is required to run the scaffolding engine.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.

In the Package Manager Console (PMC), enter the following commands:

```

Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design
Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Utils

```

The previous command adds the NuGet packages to the *.csproj file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Utils" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>
```

Scaffold the model

- Open a command window in the project directory (The directory that contains the *Program.cs*, *Startup.cs*, and *.csproj* files).
- Run the following commands:

```
dotnet restore
dotnet aspnet-codegenerator razorpage -m Student -dc SchoolContext -udl -outDir Pages\Students --
referenceScriptLibraries
```

If the following error is generated:

```
Unhandled Exception: System.IO.FileNotFoundException:
Could not load file or assembly
'Microsoft.VisualStudio.Web.CodeGeneration.Utils,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60'.
The system cannot find the file specified.
```

Run the command again and leave a comment at the bottom of the page.

If you get the error:

```
No executable found matching command "dotnet-aspnet-codegenerator"
```

Open a command window in the project directory (The directory that contains the *Program.cs*, *Startup.cs*, and *.csproj* files).

Build the project. The build generates errors like the following:

```
1>Pages\Students\Index.cshtml.cs(26,38,26,45): error CS1061: 'SchoolContext' does not contain a definition for 'Student'
```

Globally change `_context.Student` to `_context.Students` (that is, add an "s" to `Student`). 7 occurrences are found and updated. We hope to fix [this bug](#) in the next release.

The following table details the ASP.NET Core code generators` parameters:

PARAMETER	DESCRIPTION
-m	The name of the model.
-dc	The data context.

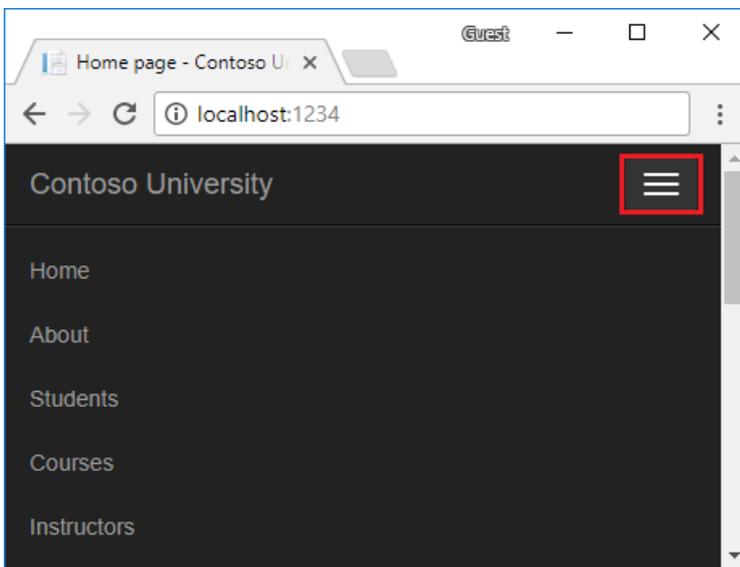
PARAMETER	DESCRIPTION
-udl	Use the default layout.
-outDir	The relative output folder path to create the views.
--referenceScriptLibraries	Adds <code>_ValidationScriptsPartial</code> to Edit and Create pages

Use the `h` switch to get help on the `aspnet-codegenerator razorpage` command:

```
dotnet aspnet-codegenerator razorpage -h
```

Test the app

Run the app and select the **Students** link. Depending on the browser width, the **Students** link appears at the top of the page. If the **Students** link is not visible, click the navigation icon in the upper right corner.



Test the **Create**, **Edit**, and **Details** links.

View the DB

When the app is started, `DbInitializer.Initialize` calls `EnsureCreated`. `EnsureCreated` detects if the DB exists, and creates one if necessary. If there are no Students in the DB, the `Initialize` method adds students.

Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio. In SSOX, click **(localdb)\MSSQLLocalDB > Databases > ContosoUniversity1**.

Expand the **Tables** node.

Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.

The `.mdf` and `.ldf` DB files are in the `C:\Users\` folder.

`EnsureCreated` is called on app start, which allows the following work flow:

- Delete the DB.
- Change the DB schema (for example, add an `EmailAddress` field).
- Run the app.

`EnsureCreated` creates a DB with the `EmailAddress` column.

Conventions

The amount of code written in order for EF Core to create a complete DB is minimal because of the use of conventions, or assumptions that EF Core makes.

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or `classNameID` are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can be named (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

Conventional behavior can be overridden. For example, the table names can be explicitly specified, as shown earlier in this tutorial. The column names can be explicitly set. Primary keys and foreign keys can be explicitly set.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the `async` keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Student = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
 - Generate callbacks for parts of the method body.
 - Automatically create the `Task` object that is returned. For more information, see [Task Return Type](#).
- The implicit return type `Task` represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the DB are executed asynchronously. That includes, `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It does not

include statements that just change an `IQueryable`, such as

```
var students = context.Students.Where(s => s.LastName == "Davolio").
```

- An EF Core context is not threaded safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the DB.

For more information about asynchronous programming in .NET, see [Async Overview](#).

In the next tutorial, basic CRUD (create, read, update, delete) operations are examined.

NEXT

Create, Read, Update, and Delete - EF Core with Razor Pages (2 of 8)

12/15/2017 • 11 min to read • [Edit Online](#)

By [Tom Dykstra](#), [Jon P Smith](#), and [Rick Anderson](#)

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see [the first tutorial](#).

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

Note: To minimize complexity and keep these tutorials focused on EF Core, EF Core code is used in the Razor Pages code-behind files. Some developers use a service layer or repository pattern in to create an abstraction layer between the UI (Razor Pages) and the data access layer.

In this tutorial, the Create, Edit, Delete, and Details Razor Pages in the *Student* folder are modified.

The scaffolded code uses the following pattern for Create, Edit, and Delete pages:

- Get and display the requested data with the HTTP GET method `OnGetAsync`.
- Save changes to the data with the HTTP POST method `OnPostAsync`.

The Index and Details pages get and display the requested data with the HTTP GET method `OnGetAsync`.

Replace `SingleOrDefaultAsync` with `FirstOrDefaultAsync`

The generated code uses `SingleOrDefaultAsync` to fetch the requested entity. `FirstOrDefaultAsync` is more efficient at fetching one entity:

- Unless the code needs to verify that there is not more than one entity returned from the query.
- `SingleOrDefaultAsync` fetches more data and does unnecessary work.
- `SingleOrDefaultAsync` throws an exception if there is more than one entity that fits the filter part.
- `FirstOrDefaultAsync` doesn't throw if there is more than one entity that fits the filter part.

Globally replace `SingleOrDefaultAsync` with `FirstOrDefaultAsync`. `SingleOrDefaultAsync` is used in 5 places:

- `OnGetAsync` in the Details page.
- `OnGetAsync` and `OnPostAsync` in the Edit and Delete pages.

FindAsync

In much of the scaffolded code, `FindAsync` can be used in place of `FirstOrDefaultAsync` or `SingleOrDefaultAsync`.

`FindAsync` :

- Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it is returned without a request to the DB.
- Is simple and concise.
- Is optimized to look up a single entity.
- Can have perf benefits in some situations, but they rarely come into play for normal web scenarios.
- Implicitly uses `FirstAsync` instead of `SingleAsync`. But if you want to Include other entities, then Find is no longer appropriate. This means that you may need to abandon Find and move to a query as your app progresses.

Customize the Details page

Browse to `Pages/Students` page. The **Edit**, **Details**, and **Delete** links are generated by the [Anchor Tag Helper](#) in the `Pages/Students/Index.cshtml` file.

```
<td>
  <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
  <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
  <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
</td>
```

Select a Details link. The URL is of the form `http://localhost:5000/Students/Details?id=2`. The Student ID is passed using a query string (`?id=2`).

Update the Edit, Details, and Delete Razor Pages to use the `"{id:int}"` route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`.

A request to the page with the `"{id:int}"` route template that does **not** include an integer route value returns an HTTP 404 (not found) error. For example, `http://localhost:5000/Students/Details` returns a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

Run the app, click on a Details link, and verify the URL is passing the ID as route data (`http://localhost:5000/Students/Details/2`).

Don't globally change `@page` to `@page "{id:int}"`, doing so breaks the links to the Home and Create pages.

Add related data

The scaffolded code for the Students Index page does not include the `Enrollments` property. In this section, the contents of the `Enrollments` collection is displayed in the Details page.

The `OnGetAsync` method of `Pages/Students/Details.cshtml.cs` uses the `FirstOrDefaultAsync` method to retrieve a single `Student` entity. Add the following highlighted code:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync (m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the reading-related data tutorial.

The `AsNoTracking` method improves performance in scenarios when the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display related enrollments on the Details page

Open *Pages/Students/Details.cshtml*. Add the following highlighted code to display a list of enrollments:

```

@page "{id:int}"
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd>
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>

```

If code indentation is wrong after the code is pasted, press CTRL-K-D to correct it.

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for the selected student is displayed.

Update the Create page

Update the `OnPostAsync` method in `Pages/Students/Create.cshtml.cs` with the following code:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student", // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return null;
}
```

TryUpdateModelAsync

Examine the `TryUpdateModelAsync` code:

```
var emptyStudent = new Student();

if (await TryUpdateModelAsync<Student>(
    emptyStudent,
    "student", // Prefix for form value.
    s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
{
```

In the preceding code, `TryUpdateModelAsync<Student>` tries to update the `emptyStudent` object using the posted form values from the `PageContext` property in the `PageModel`. `TryUpdateModelAsync` only updates the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).

In the preceding sample:

- The second argument (`"student", // Prefix`) is the prefix used to look up values. It's not case sensitive.
- The posted form values are converted to the types in the `Student` model using [model binding](#).

Overposting

Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the `Student` entity includes a `Secret` property that this web page should not update or add:

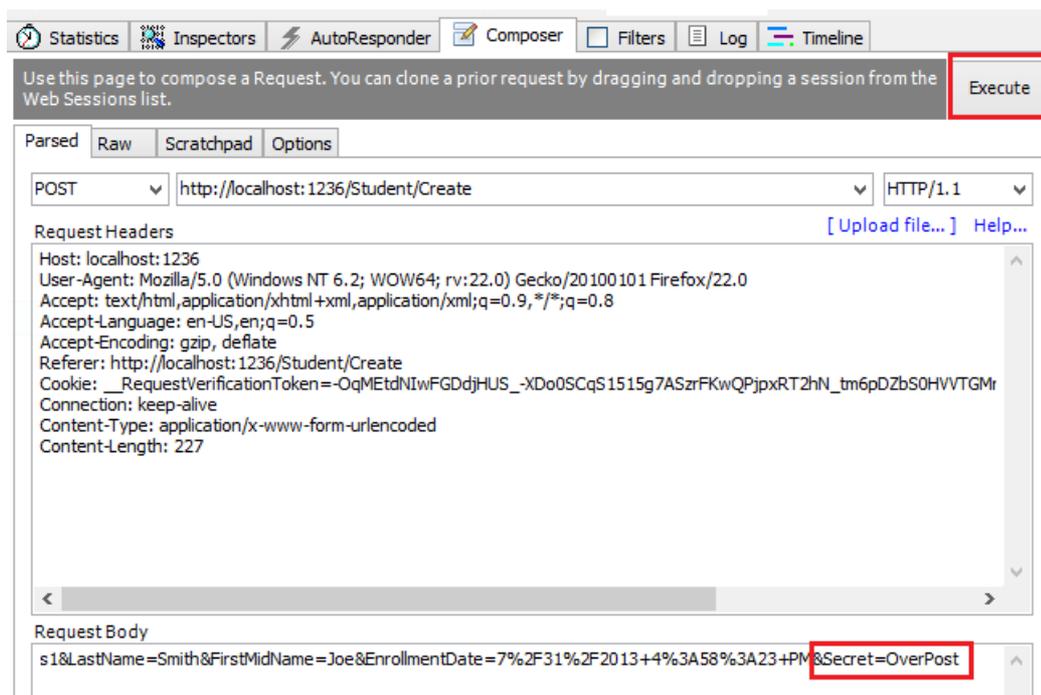
```

public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}

```

Even if the app doesn't have a `Secret` field on the create/update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the DB. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. The app designer never intended the `Secret` property to be set with the Create page.

View model

A view model typically contains a subset of the properties included in the model used by the application. The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the DB. The view model contains only the properties needed for the UI layer (for example, the Create page). In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages code-behind class and the browser. Consider the following `Student` view model:

```
using System;

namespace ContosoUniversity.Models
{
    public class StudentVM
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }
    }
}
```

View models provide an alternative way to prevent overposting. The view model contains only the properties to view (display) or update.

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("../Index");
}
```

The `SetValues` method sets the values of this object by reading values from another `PropertyValues` object.

`SetValues` uses property name matching. The view model type doesn't need to be related to the model type, it just needs to have properties that match.

Using `StudentVM` requires `CreateVM.cshtml` be updated to use `StudentVM` rather than `Student`.

In Razor Pages, the `PageModel` derived class is the view model.

Update the Edit page

Update the Edit page code-behind file:

```

public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Student = await _context.Students.FindAsync(id);

        if (Student == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var studentToUpdate = await _context.Students.FindAsync(id);

        if (await TryUpdateModelAsync<Student>(
            studentToUpdate,
            "student",
            s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
        {
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }

        return Page();
    }
}

```

The code changes are similar to the Create page with a few exceptions:

- `OnPostAsync` has an optional `id` parameter.
- The current student is fetched from the DB, rather than creating an empty student.
- `FirstOrDefaultAsync` has been replaced with `FindAsync`. `FindAsync` is a good choice when selecting an entity from the primary key. See [FindAsync](#) for more information.

Test the Edit and Create pages

Create and edit a few student entities.

Entity States

The DB context keeps track of whether entities in memory are in sync with their corresponding rows in the DB. The

DB context sync information determines what happens when `SaveChanges` is called. For example, when a new entity is passed to the `Add` method, that entity's state is set to `Added`. When `SaveChanges` is called, the DB context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`: The entity does not yet exist in the DB. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it is read from the DB.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`: The entity isn't being tracked by the DB context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state to automatically be changed to `Modified`. Calling `SaveChanges` generates a SQL UPDATE statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Re-reading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, code is added to implement a custom error message when the call to `SaveChanges` fails. Add a string to contain possible error messages:

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }
    public string ErrorMessage { get; set; }
```

Replace the `OnGetAsync` method with the following code:

```

public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ErrorMessage = "Delete failed. Try again";
    }

    return Page();
}

```

The preceding code contains the optional parameter `saveChangesError`. `saveChangesError` indicates whether the method was called after a failure to delete the student object. The delete operation might fail because of transient network problems. Transient network errors are more likely in the cloud. `saveChangesError` is false when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` (because the delete operation failed), the `saveChangesError` parameter is true.

The Delete pages `OnPostAsync` method

Replace the `OnPostAsync` with the following code:

```

public async Task<IActionResult> OnPostAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("./Delete",
            new { id = id, saveChangesError = true });
    }
}

```

The preceding code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated. If `Remove` fails:

- The DB exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Update the Delete Razor Page

Add the following highlighted error message to the Delete Razor Page.

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
```

Test Delete.

Common errors

Student/Home or other links don't work:

Verify the Razor Page contains the correct `@page` directive. For example, The Student/Home Razor Page should **not** contain a route template:

```
@page "{id:int}"
```

Each Razor Page must include the `@page` directive.

[PREVIOUS](#)

[NEXT](#)

Sorting, filtering, paging, and grouping - EF Core with Razor Pages (3 of 8)

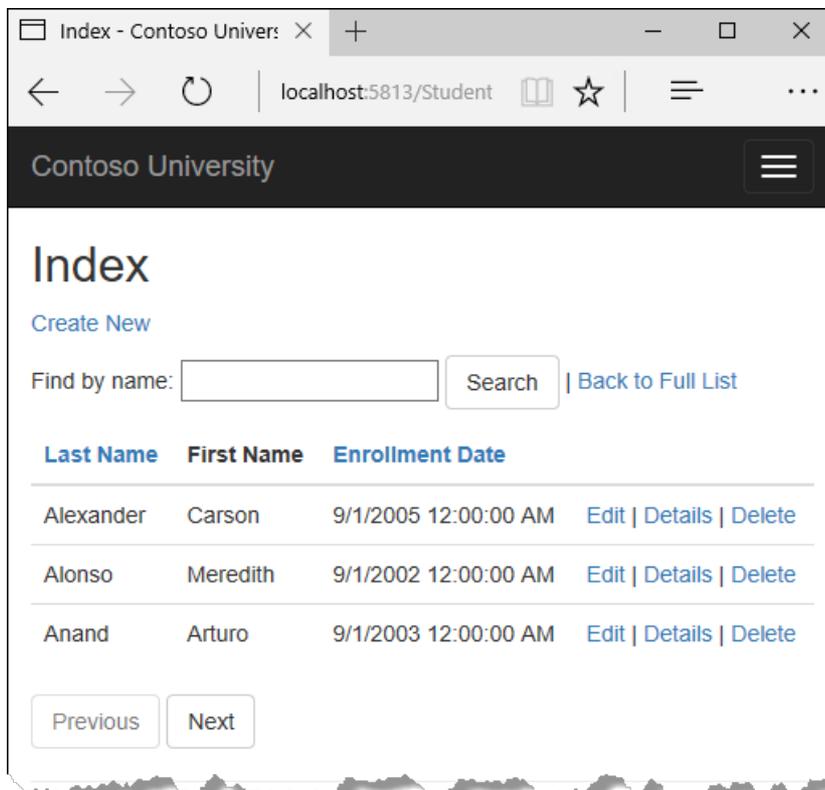
12/5/2017 • 15 min to read • [Edit Online](#)

By [Tom Dykstra](#), [Rick Anderson](#), and [Jon P Smith](#)

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see [the first tutorial](#).

In this tutorial, sorting, filtering, grouping, and paging, functionality is added.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Clicking a column heading repeatedly switches between ascending and descending sort order.



If you run into problems you can't solve, download the [completed app for this stage](#).

Add sorting to the Index page

Add strings to the `Students/Index.cshtml.cs` `PageModel` to contain the sorting parameters:

```

public class IndexModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public IndexModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }
}

```

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```

public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Students
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}

```

The preceding code receives a `sortOrder` parameter from the query string in the URL. The URL (including the query string) is generated by the [Anchor Tag Helper](#)

The `sortOrder` parameter is either "Name" or "Date." The `sortOrder` parameter is optionally followed by "_desc" to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the default (fall-through case) in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```

public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Students
                                   select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}

```

The following code contains the C# `?:` operator:

```

NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";

```

The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to "name_desc." If `sortOrder` is **not** null or empty, `NameSort` is set to an empty string.

The `?:` operator is also known as the ternary operator.

These two statements enable the view to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

```

public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Students
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}

```

When an `IQueryable` is created or modified, no query is sent to the database. The query is not executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that is not executed until the following statement:

```
Student = await studentIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of columns.

Add column heading hyperlinks to the Student Index view

Replace the code in `Students/Index.cshhtml`, with the following highlighted code:

```

@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click **Last Name**.
- Click **Enrollment Date**.

To get a better understanding of the code:

- In *Student/Index.cshtml.cs*, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort` and `DateSort`.
- In *Student/Index.cshtml*, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Add a Search Box to the Students Index page

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The code-behind file is updated to use the text box value.

Add filtering functionality to the Index method

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder, string searchString)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Students
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
    }

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method. The search string value is received from a text box that is added in the next section.
- Added to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is executed only if there's a value to search for.

Note: The preceding code calls the `Where` method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory

collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper()))
```

The preceding code would ensure that results are case-insensitive if the code changes to use `IEnumerable`. When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used. Returning an `IEnumerable` from a repository can have a significant performance penalty:

1. All the rows are returned from the DB server.
2. The filter is applied to all the returned rows in the application.

There is a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

Add a Search Box to the Student Index View

In `Views/Student/Index.cshtml`, add the following highlighted code to create a **Search** button and assorted chrome.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form sp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
```

The preceding code uses the `<form>` [tag helper](#) to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The [W3C guidelines](#) recommend that GET should be used when the action does not result in an update.

Test the app:

- Select the **Students** tab and enter a search string.
- Select **Search**.

Notice that the URL contains the search string.

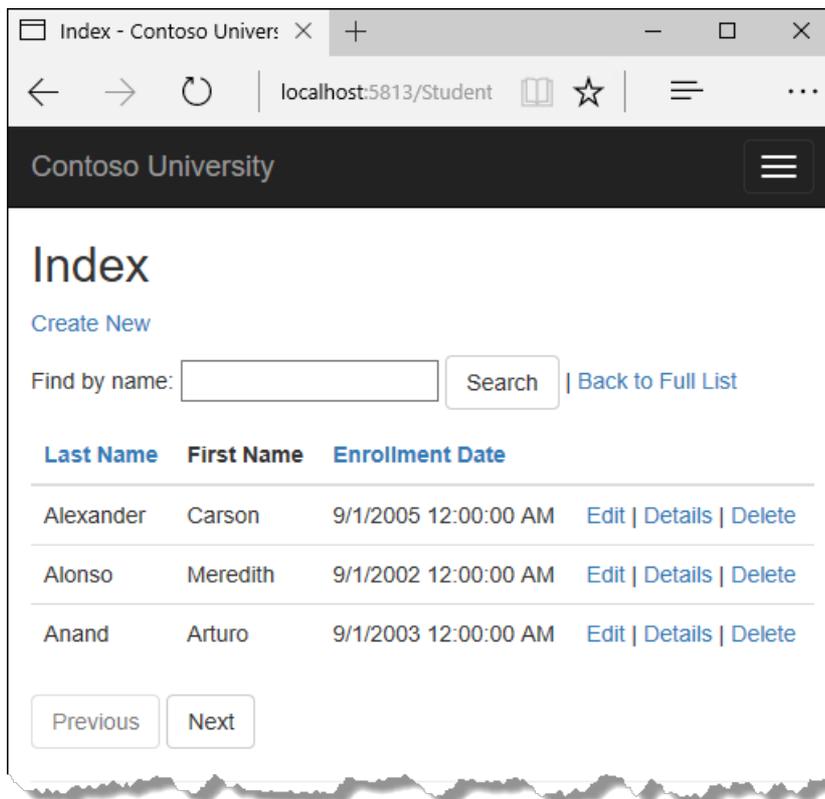
```
http://localhost:5000/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging functionality to the Students Index page

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs` with the following code:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}

```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object, constructors can't run asynchronous code.

Add paging functionality to the Index method

In `Students/Index.cshtml.cs`, update the type of `Student` from `IList<Student>` to `PaginatedList<Student>`:

```
public PaginatedList<Student> Student { get; set; }
```

Update the *Students/Index.cshhtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
{
    CurrentSort = sortOrder;
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    if (searchString != null)
    {
        pageIndex = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Students
        select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
            || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    Student = await PaginatedList<Student>.CreateAsync(
        studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
}
```

The preceding code adds the page index, the current `sortOrder`, and the `currentFilter` to the method signature.

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
```

All the parameters are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

`CurrentSort` provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

`CurrentFilter` provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter is not null.

```
if (searchString != null)
{
    pageIndex = 1;
}
else
{
    searchString = currentFilter;
}
```

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

```
Student = await PaginatedList<Student>.CreateAsync(
    studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
```

The two question marks in `PaginatedList.CreateAsync` represent the [null-coalescing operator](#). The null-coalescing operator defines a default value for a nullable type. The expression `(pageIndex ?? 1)` means return the value of `pageIndex` if it has a value. If `pageIndex` doesn't have a value, return 1.

Add paging links to the student Razor Page

Update the markup in `Students/Index.cshtml`. The changes are highlighted:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                    href="@Model.SortOrder" class="text-decoration: none">
                    @Model.SortOrder
                </a>
            </th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>
                <a href="#">
                    <input type="checkbox" />
                </a>
            </td>
            <td>
                <input type="text" value="@Model.CurrentFilter" />
            </td>
        </tr>
    </tbody>
</table>
```

```

                asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                    asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
        </thead>
        <tbody>
        @foreach (var item in Model.Student) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
        </tbody>
    </table>

    @{
        var prevDisabled = !Model.Student.HasPreviousPage ? "disabled" : "";
        var nextDisabled = !Model.Student.HasNextPage ? "disabled" : "";
    }

    <a asp-page="./Index"
        asp-route-sortOrder="@Model.CurrentSort"
        asp-route-pageIndex="@((Model.Student.PageIndex - 1))"
        asp-route-currentFilter="@Model.CurrentFilter"
        class="btn btn-default @prevDisabled">
        Previous
    </a>
    <a asp-page="./Index"
        asp-route-sortOrder="@Model.CurrentSort"
        asp-route-pageIndex="@((Model.Student.PageIndex + 1))"
        asp-route-currentFilter="@Model.CurrentFilter"
        class="btn btn-default @nextDisabled">
        Next
    </a>

```

The column header links use the query string to pass the current search string to the `OnGetAsync` method so that the user can sort within filter results:

```

<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
    asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Student[0].LastName)
</a>

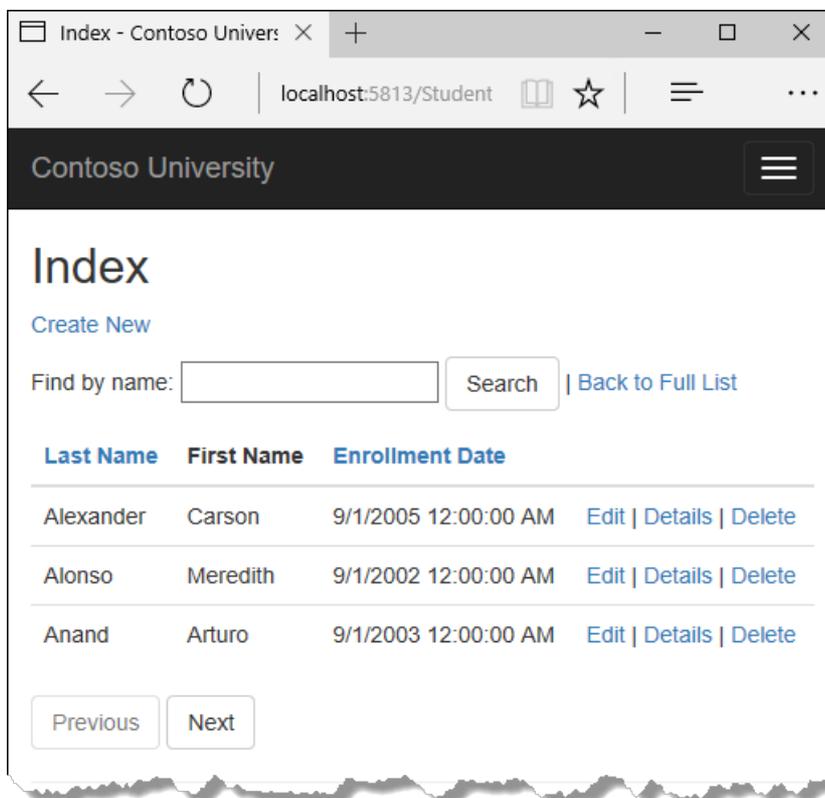
```

The paging buttons are displayed by tag helpers:

```
<a asp-page="./Index"
  asp-route-sortOrder="@Model.CurrentSort"
  asp-route-pageIndex="@((Model.Student.PageIndex - 1))"
  asp-route-currentFilter="@Model.CurrentFilter"
  class="btn btn-default @prevDisabled">
  Previous
</a>
<a asp-page="./Index"
  asp-route-sortOrder="@Model.CurrentSort"
  asp-route-pageIndex="@((Model.Student.PageIndex + 1))"
  asp-route-currentFilter="@Model.CurrentFilter"
  class="btn btn-default @nextDisabled">
  Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



To get a better understanding of the code:

- In *Student/Index.cshtml.cs*, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort`, `DateSort`, `CurrentSort`, and `Model.Student.PageIndex`.
- In *Student/Index.cshtml*, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Update the About page to show student statistics

In this step, *Pages/About.cshtml* is updated to display how many students have enrolled for each enrollment date. The update uses grouping, and includes the following steps:

- Create a view model class for the data used by the **About** Page.
- Modify the About Razor Page and code-behind file.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the *SchoolViewModels* folder, add a *EnrollmentDateGroup.cs* with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Update the About code-behind page

Update the *Pages/About.cshtml.cs* file with the following code:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Student { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Student = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each

group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Note: The LINQ `group` command isn't currently supported by EF Core. In the preceding code, all the student records are returned from SQL Server. The `group` command is applied in the Razor Pages app, not on the SQL Server. EF Core 2.1 will support this LINQ `group` operator, and the grouping occurs on the SQL Server. See [Relational: Support translating GroupBy\(\) to SQL](#). EF Core 2.1 will be released with .NET Core 2.1. For more information, see the [.NET Core Roadmap](#).

Modify the About Razor Page

Replace the code in the `Views/Home/About.cshtml` file with the following code:

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

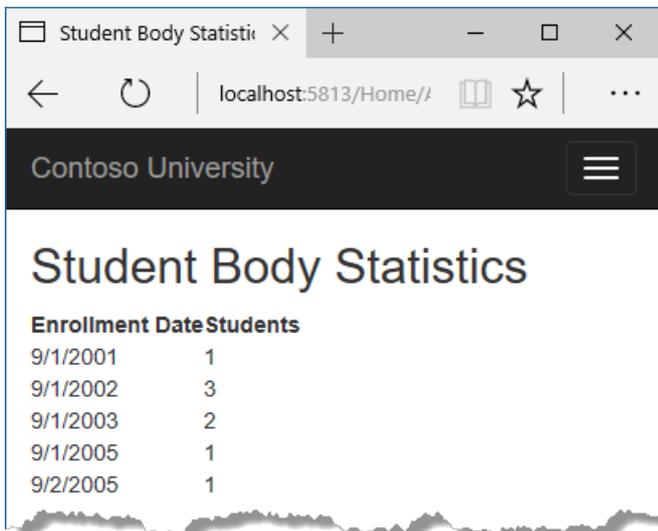
<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Student)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.

If you run into problems you can't solve, download the [completed app for this stage](#).



Additional Resources

- [Debugging ASP.NET Core 2.x source](#)

In the next tutorial, the app uses migrations to update the data model.

PREVIOUS

NEXT

Migrations - EF Core with Razor Pages tutorial (4 of 8)

12/2/2017 • 9 min to read • [Edit Online](#)

By [Tom Dykstra](#), [Jon P Smith](#), and [Rick Anderson](#)

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see [the first tutorial](#).

In this tutorial, the EF Core migrations feature for managing data model changes is used.

If you run into problems you can't solve, download the [completed app for this stage](#).

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes:

- The DB is dropped.
- EF creates a new one that matches the model.
- The app seeds the DB with test data.

This approach to keeping the DB in sync with the data model works well until you deploy the app to production. When the app is running in production, it is usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the DB schema instead of creating a new DB.

Rather than dropping and recreating the DB when the data model changes, migrations updates the schema and retains existing data.

Entity Framework Core NuGet packages for migrations

To work with migrations, use the **Package Manager Console** (PMC) or the command-line interface (CLI). These tutorials show how to use CLI commands. Information about the PMC is at [the end of this tutorial](#).

The EF Core tools for the command-line interface (CLI) are provided in [Microsoft.EntityFrameworkCore.Tools.DotNet](#). To install this package, add it to the `DotNetCliToolReference` collection in the `.csproj` file, as shown. **Note:** This package must be installed by editing the `.csproj` file. The `install-package` command or the package manager GUI cannot be used to install this package. Edit the `.csproj` file by right-clicking the project name in **Solution Explorer** and selecting **Edit ContosoUniversity.csproj**.

The following markup shows the updated `.csproj` file with the EF Core CLI tools highlighted:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Utils" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  </ItemGroup>
</Project>
```

The version numbers in the preceding example were current when the tutorial was written. Use the same version for the EF Core CLI tools found in the other packages.

Change the connection string

In the `appsettings.json` file, change the name of the DB in the connection string to `ContosoUniversity2`.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity2;ConnectRetryCount=0;Trusted_Connection=True;MultipleActive
ResultSets=true"
  },
}
```

Changing the DB name in the connection string causes the first migration to create a new DB. A new DB is created because one with that name does not exist. Changing the connection string isn't required for getting started with migrations.

An alternative to changing the DB name is deleting the DB. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

```
dotnet ef database drop
```

The following section explains how to run CLI commands.

Create an initial migration

Build the project.

Open a command window and navigate to the project folder. The project folder contains the `Startup.cs` file.

Enter the following in the command window:

```
dotnet ef migrations add InitialCreate
```

The command window displays information similar to the following:

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

If the migration fails with the message *"cannot access the file ... ContosoUniversity.dll because it is being used by another process."* is displayed:

- Stop IIS Express.
 - Exit and restart Visual Studio, or
 - Find the IIS Express icon in the Windows System Tray.
 - Right-click the IIS Express icon, and then click **ContosoUniversity > Stop Site**.

If the error message "Build failed." is displayed, run the command again. If you get this error, leave a note at the bottom of this tutorial.

Examine the Up and Down methods

The EF Core command `migrations add` generated code to create the DB from. This migrations code is in the `Migrations<timestamp>_InitialCreate.cs` file. The `Up` method of the `InitialCreate` class creates the DB tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(type: "int", nullable: false),
                Credits = table.Column<int>(type: "int", nullable: false),
                Title = table.Column<string>(type: "nvarchar(max)", nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");

        migrationBuilder.DropTable(
            name: "Course");

        migrationBuilder.DropTable(
            name: "Student");
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, migrations calls the `Down` method.

The preceding code is for the initial migration. That code was created when the `migrations add InitialCreate`

command was run. The migration name parameter ("InitialCreate" in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

If the initial migration is created and the DB exists:

- The DB creation code is generated.
- The DB creation code doesn't need to run because the DB already matches the data model. If the DB creation code is run, it doesn't make any changes because the DB already matches the data model.

When the app is deployed to a new environment, the DB creation code must be run to create the DB.

Previously the connection string was changed to use a new name for the DB. The specified DB doesn't exist, so migrations creates the DB.

Examine the data model snapshot

Migrations creates a *snapshot* of the current DB schema in *Migrations/SchoolContextModelSnapshot.cs*:

```
[DbContext(typeof(SchoolContext))]
partial class SchoolContextModelSnapshot : ModelSnapshot
{
    protected override void BuildModel(ModelBuilder modelBuilder)
    {
        modelBuilder
            .HasAnnotation("ProductVersion", "2.0.0-rtm-26452")
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
                SqlServerValueGenerationStrategy.IdentityColumn);

        modelBuilder.Entity("ContosoUniversity.Models.Course", b =>
        {
            b.Property<int>("CourseID");

            b.Property<int>("Credits");

            b.Property<string>("Title");

            b.HasKey("CourseID");

            b.ToTable("Course");
        });

        // Additional code for Enrollment and Student tables not shown.

        modelBuilder.Entity("ContosoUniversity.Models.Enrollment", b =>
        {
            b.HasOne("ContosoUniversity.Models.Course", "Course")
                .WithMany("Enrollments")
                .HasForeignKey("CourseID")
                .OnDelete(DeleteBehavior.Cascade);

            b.HasOne("ContosoUniversity.Models.Student", "Student")
                .WithMany("Enrollments")
                .HasForeignKey("StudentID")
                .OnDelete(DeleteBehavior.Cascade);
        });
    }
}
```

Because the current DB schema is represented in code, EF Core doesn't have to interact with the DB to create migrations. When you add a migration, EF Core determines what changed by comparing the data model to the snapshot file. EF Core interacts with the DB only when it has to update the DB.

The snapshot file must be in sync with the migrations that created it. A migration can't be removed by deleting the file named `<timestamp>_<migrationname>.cs`. If that file is deleted, the remaining migrations are out of sync with the DB snapshot file. To delete the last migration added, use the [dotnet ef migrations remove](#) command.

Remove EnsureCreated

For early development, the `EnsureCreated` command was used. In this tutorial, migrations is used. `EnsureCreated` has the following limitations:

- Bypasses migrations and creates the DB and schema.
- Does not create a migrations table.
- Can *not* be used with migrations.
- Is designed for testing or rapid prototyping where the DB is dropped and re-created frequently.

Remove the following line from `DbInitializer` :

```
context.Database.EnsureCreated();
```

Apply the migration to the DB in development

In the command window, enter the following to create the DB and tables.

```
dotnet ef database update
```

Note: If the `update` command returns the error "Build failed.":

- Run the command again.
- If it fails again, exit Visual Studio and then run the `update` command.
- Leave a message at the bottom of the page.

The output from the command is similar to the `migrations add` command output. In the preceding command, logs for the SQL commands that set up the DB are displayed. Most of the logs are omitted in the following sample output:

```

info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
      repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
      'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (467ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );

<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20170816151242_InitialCreate', N'2.0.0-rtm-26452');
Done.

```

To reduce the level of detail in log messages, can change the log levels in the *appsettings.Development.json* file. For more information, see [Introduction to logging](#).

Use **SQL Server Object Explorer** to inspect the DB. Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the DB. View the data in the `__EFMigrationsHistory` table, it shows one row for the first migration. The last log in the preceding CLI output example shows the INSERT statement that creates this row.

Run the app and verify that everything works.

Applying migrations in production

We recommend production apps should **not** call `Database.Migrate` at application startup. `Migrate` should not be called from an app in server farm. For example, if the app has been cloud deployed with scale-out (multiple instances of the app are running).

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

EF Core uses the `__MigrationsHistory` table to see if any migrations need to run. If the DB is up to date, no migration is run.

Command-line interface (CLI) vs. Package Manager Console (PMC)

The EF Core tooling for managing migrations is available from:

- .NET Core CLI commands.
- The PowerShell cmdlets in the Visual Studio **Package Manager Console** (PMC) window.

This tutorial shows how to use the CLI, some developers prefer using the PMC.

The EF Core commands for the PMC are in the `Microsoft.EntityFrameworkCore.Tools` package. This package is

included in the [Microsoft.AspNetCore.All](#) metapackage, so you don't have to install it.

Important: This is not the same package as the one you install for the CLI by editing the `.csproj` file. The name of this one ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see [.NET Core CLI](#).

For more information about the PMC commands, see [Package Manager Console \(Visual Studio\)](#).

Troubleshooting

Download the [completed app for this stage](#).

The app generates the following exception:

```
`SqlException: Cannot open database "ContosoUniversity" requested by the login.  
The login failed.  
Login failed for user 'user name'.
```

Solution: Run `dotnet ef database update`

If the `update` command returns the error "Build failed.":

- Run the command again.
- Leave a message at the bottom of the page.

[PREVIOUS](#)

[NEXT](#)

Creating a complex data model - EF Core with Razor Pages tutorial (5 of 8)

12/2/2017 • 28 min to read • [Edit Online](#)

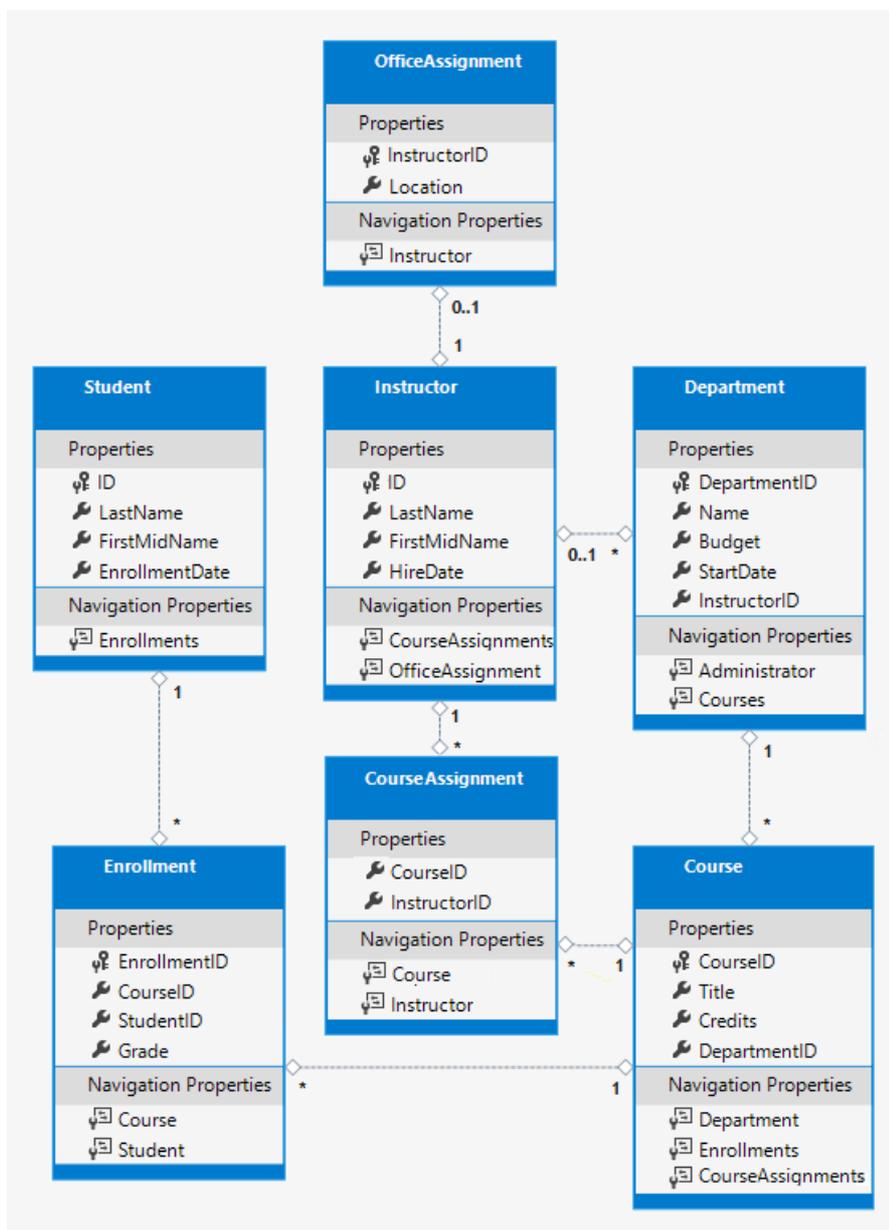
By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see [the first tutorial](#).

The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The entity classes for the completed data model is shown in the following illustration:



If you run into problems you can't solve, download the [completed app for this stage](#).

Customize the data model with attributes

In this section, the data model is customized using attributes.

The `DataType` attribute

The student pages currently displays the time of the enrollment date. Typically, date fields show only the date and not the time.

Update `Models/Student.cs` with the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `DataType` attribute specifies a data type that is more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The [DataType Enumeration](#) provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers consume. The `DataType` attributes do not provide validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the date field is displayed according to the default formats based on the server's [CultureInfo](#).

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

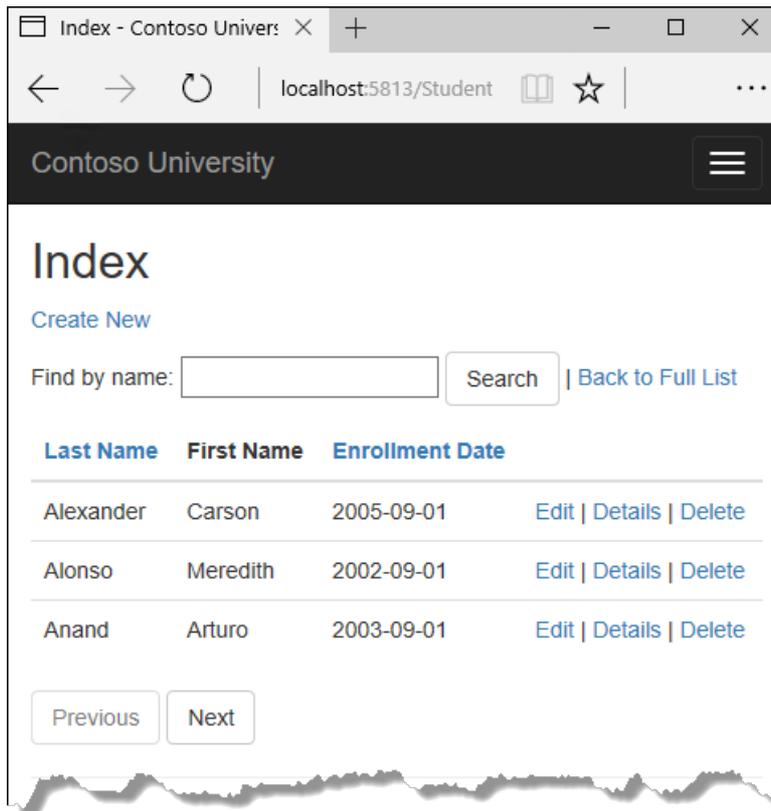
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields should not use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, client-side input validation, etc.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the [<input> Tag Helper documentation](#).

Run the app. Navigate to the Students Index page. Times are no longer displayed. Every view that uses the `Student` model displays the date without time.



The `StringLength` attribute

Data validation rules and validation error messages can be specified with attributes. The `StringLength` attribute specifies the minimum and maximum length of characters that are allowed in a data field. The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

Update the `Student` model with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code limits names to no more than 50 characters. The `StringLength` attribute doesn't prevent a user from entering white space for a name. The `RegularExpression` attribute is used to apply restrictions to the

input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z' '-'\s]*$")]
```

Run the app:

- Navigate to the Students page.
- Select **Create New**, and enter a name longer than 50 characters.
- Select **Create**, client-side validation shows an error message.

The screenshot shows a web browser window with the URL localhost:5813/Student. The page title is 'Contoso University' and the main heading is 'Create Student'. There are three input fields: 'LastName' with the value 'Davolio very long last name longer than !', 'FirstMidName' with the value 'Nancy very long first name longer than 5', and 'EnrollmentDate' with the value '2/15/2017'. Below the 'LastName' field, there is a red error message: 'The field LastName must be a string with a maximum length of 50.' Below the 'FirstMidName' field, there is another red error message: 'First name cannot be longer than 50 characters.' A 'Create' button is visible at the bottom of the form.

In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.

The screenshot shows the SQL Server Object Explorer (SSOX) interface. The 'dbo.Student [Design]' table is selected, and the 'Design' tab is active. The table structure is displayed in a grid with columns for Name, Data Type, and Allow Nulls. The 'ID' column is the primary key. The 'EnrollmentDate' column is of type datetime2(7) and is not null. The 'FirstMidName' and 'LastName' columns are of type nvarchar(MAX) and are nullable. The 'Create Table' script is visible in the T-SQL tab below the design grid.

Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>
EnrollmentDate	datetime2(7)	<input type="checkbox"/>
FirstMidName	nvarchar(MAX)	<input checked="" type="checkbox"/>
LastName	nvarchar(MAX)	<input checked="" type="checkbox"/>

```
1 CREATE TABLE [dbo].[Student] (  
2     [ID] INT IDENTITY (1, 1) NOT NULL,  
3     [EnrollmentDate] DATETIME2 (7) NOT NULL,  
4     [FirstMidName] NVARCHAR (MAX) NULL,  
5     [LastName] NVARCHAR (MAX) NULL,  
6     CONSTRAINT [PK_Student] PRIMARY KEY CLUSTERED ([ID] ASC)  
7 );  
8
```

The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)` because migrations has not been run on the DB. When migrations are run later in this tutorial, the name fields become `nvarchar(50)`.

The Column attribute

Attributes can control how classes and properties are mapped to the database. In this section, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the DB.

When the DB is created, property names on the model are used for column names (except when the `Column` attribute is used).

The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

Update the `Student.cs` file with the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

With the preceding change, `Student.FirstMidName` in the app maps to the `FirstName` column of the `Student` table.

The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. If the app is run before applying migrations, the following exception is generated:

```
SqlException: Invalid column name 'FirstName'.
```

To update the DB:

- Build the project.
- Open a command window in the project folder. Enter the following commands to create a new migration and update the DB:

```
dotnet ef migrations add ColumnFirstName
dotnet ef database update
```

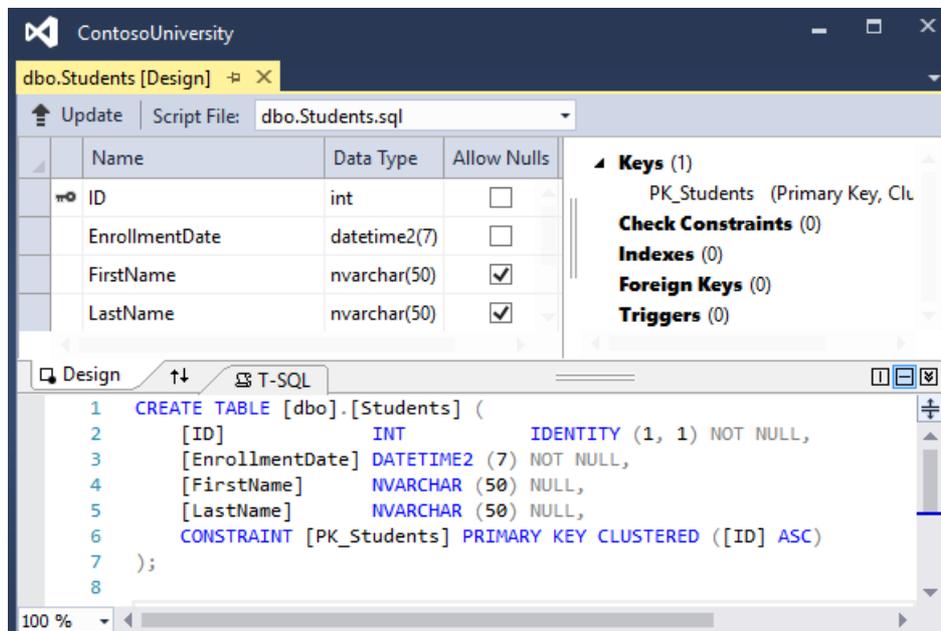
The `dotnet ef migrations add ColumnFirstName` command generates the following warning message:

An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.

The warning is generated because the name fields are now limited to 50 characters. If a name in the DB had more than 50 characters, the 51 to last character would be lost.

- Test the app.

Open the Student table in SSOX:



Before migration was applied, the name columns were of type `nvarchar(MAX)`. The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

NOTE

In the following section, building the app at some stages generates compiler errors. The instructions specify when to build the app.

Student entity update

Student
Properties
ID
LastName
FirstMidName
EnrollmentDate
Navigation Properties
Enrollments

Update `Models/Student.cs` with the following code:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute is not needed for non-nullable types such as value types (`DateTime`, `int`, `double`, etc.). Types that can't be null are automatically treated as required fields.

The `Required` attribute could be replaced with a minimum length parameter in the `StringLength` attribute:

```

[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }

```

The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties.

`FullName` cannot be set, it has only a get accessor. No `FullName` column is created in the database.

Create the Instructor Entity

Instructor	
Properties	
PK	ID
	LastName
	FirstMidName
	HireDate
Navigation Properties	
1	CourseAssignments
1	OfficeAssignment

Create *Models/Instructor.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the `Student` and `Instructor` entities. In the [Implementing Inheritance](#) tutorial later in this series, this code is refactored to eliminate the redundancy.

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The `CourseAssignments` and `OfficeAssignment` navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

If a navigation property holds multiple entities:

- It must be a list type where the entries can be added, deleted, and updated.

Navigation property types include:

- `ICollection<T>`
- `List<T>`
- `HashSet<T>`

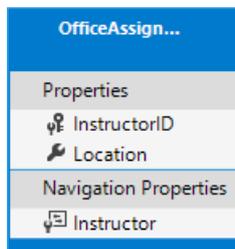
If `ICollection<T>` is specified, EF Core creates a `HashSet<T>` collection by default.

The `CourseAssignment` entity is explained in the section on many-to-many relationships.

Contoso University business rules state that an instructor can have at most one office. The `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than `classNameID` or `ID`.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office

assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity. EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because:

- `InstructorID` doesn't follow the ID or classnameID naming convention.

Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The `OfficeAssignment` navigation property for the `Instructor` entity is nullable because:

- Reference types (such as classes) are nullable.
- An instructor might not have an office assignment.

The `OfficeAssignment` entity has a non-nullable `Instructor` navigation property because:

- `InstructorID` is non-nullable.
- An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The `[Required]` attribute could be applied to the `Instructor` navigation property:

```
[Required]
public Instructor Instructor { get; set; }
```

The preceding code specifies that there must be a related instructor. The preceding code is unnecessary because the `InstructorID` foreign key (which is also the PK) is non-nullable.

Modify the Course Entity

Course
Properties
🔑 CourseID
🔑 Title
🔑 Credits
🔑 DepartmentID
Navigation Properties
🔑 Department
🔑 Enrollments
🔑 CourseAssignments

Update `Models/Course.cs` with the following code:

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}

```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a FK property for a data model when the model has a navigation property for a related entity.

EF Core automatically creates FKs in the database wherever they are needed. EF Core creates [shadow properties](#) for automatically created FKs. Having the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` entity is null if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there is no need to fetch the `Department` entity before an update.

The `DatabaseGenerated` attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

```

[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }

```

By default, EF Core assumes that PK values are generated by the DB. DB generated PK values is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the DB can automatically generate a date field to record the date a row was created or updated. For more information, see [Generated Properties](#).

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection:

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

`CourseAssignment` is explained [later](#).

Create the Department entity

Department
Properties
DepartmentID
Name
Budget
StartDate
InstructorID
Navigation Properties
Administrator
Courses

Create `Models/Department.cs` with the following code:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}

```

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the DB:

```

[Column(TypeName="money")]
public decimal Budget { get; set; }

```

Column mapping is generally not required. EF Core generally chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

```

public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }

```

The question mark (?) in the preceding code specifies the property is nullable.

A department may have many courses, so there's a `Courses` navigation property:

```
public ICollection<Course> Courses { get; set; }
```

Note: By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. Cascading delete can result in circular cascade delete rules. Circular cascade delete rules causes an exception when a migration is added.

For example, if the `Department.InstructorID` property was not defined as nullable:

- EF Core configures a cascade delete rule to delete the instructor when the department is deleted.
- Deleting the instructor when the department is deleted is not the intended behavior.

If business rules required the `InstructorID` property be non-nullable, use the following fluent API statement:

```
modelBuilder.Entity<Department>()  
    .HasOne(d => d.Administrator)  
    .WithMany()  
    .OnDelete(DeleteBehavior.Restrict)
```

The preceding code disables cascade delete on the department-instructor relationship.

Update the Enrollment entity

An enrollment record is for a one course taken by one student.

Enrollment	
Properties	
PK	EnrollmentID
FK	CourseID
FK	StudentID
FK	Grade
Navigation Properties	
FK	Course
FK	Student

Update `Models/Enrollment.cs` with the following code:

```
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;  
  
namespace ContosoUniversity.Models  
{  
    public enum Grade  
    {  
        A, B, C, D, F  
    }  
  
    public class Enrollment  
    {  
        public int EnrollmentID { get; set; }  
        public int CourseID { get; set; }  
        public int StudentID { get; set; }  
        [DisplayFormat(NullDisplayText = "No grade")]  
        public Grade? Grade { get; set; }  
  
        public Course Course { get; set; }  
        public Student Student { get; set; }  
    }  
}
```

Foreign key and navigation properties

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

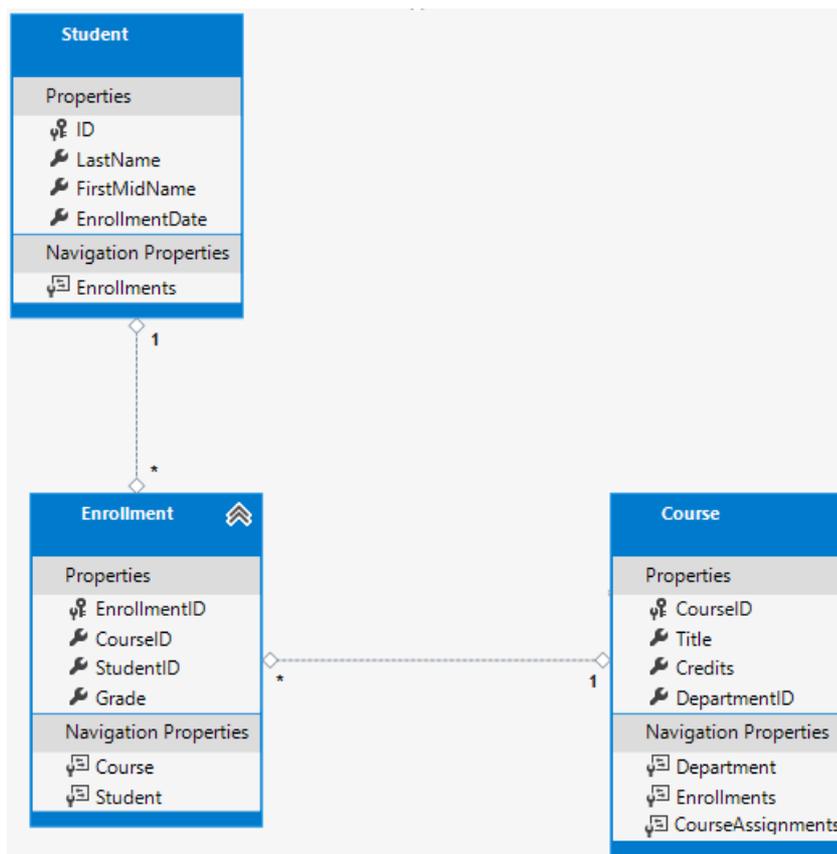
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities. The `Enrollment` entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the `Enrollment` table contains additional data besides FKs for the joined tables (in this case, the PK and `Grade`).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



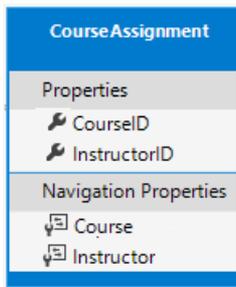
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs (`CourseID` and `StudentID`). A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a pure join table.

Note: EF 6.x supports implicit join tables for many-to-many relationships, but EF Core does not. For more information, see [Many-to-many relationships in EF Core 2.0](#).

The CourseAssignment entity

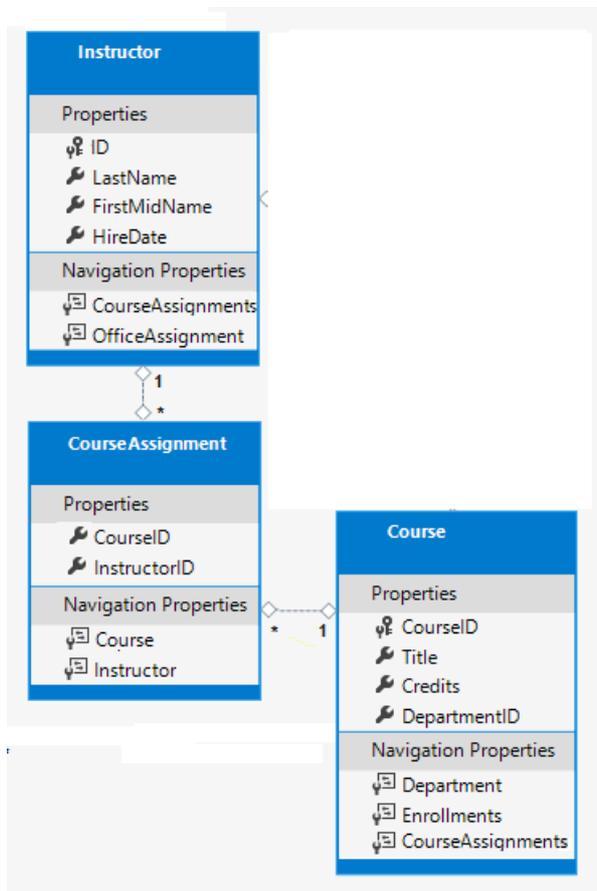


Create *Models/CourseAssignment.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Instructor-to-Courses



The Instructor-to-Courses many-to-many relationship:

- Requires a join table that must be represented by an entity set.

- Is a pure join table (table without payload).

It's common to name a join entity `EntityName1EntityName2`. For example, the Instructor-to-Courses join table using this pattern is `CourseInstructor`. However, we recommend using a name that describes the relationship.

Data models start out simple and grow. No-payload joins (PJT) frequently evolve to include payload. By starting with a descriptive entity name, the name doesn't need to change when the join table changes. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked with a join entity called Ratings. For the Instructor-to-Courses many-to-many relationship, `CourseAssignment` is preferred over `CourseInstructor`.

Composite key

FKs are not nullable. The two FKs in `CourseAssignment` (`InstructorID` and `CourseID`) together uniquely identify each row of the `CourseAssignment` table. `CourseAssignment` doesn't require a dedicated PK. The `InstructorID` and `CourseID` properties function as a composite PK. The only way to specify composite PKs to EF Core is with the *fluent API*. The next section shows how to configure the composite PK.

The composite key ensures:

- Multiple rows are allowed for one course.
- Multiple rows are allowed for one instructor.
- Multiple rows for the same instructor and course is not allowed.

The `Enrollment` join entity defines its own PK, so duplicates of this sort are possible. To prevent such duplicates:

- Add a unique index on the FK fields, or
- Configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see [Indexes](#).

Update the DB context

Add the following highlighted code to `Data/SchoolContext.cs`:

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}

```

The preceding code adds the new entities and configures the `CourseAssignment` entity's composite PK.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement. The [following code](#) is an example of the fluent API:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}

```

In this tutorial, the fluent API is used only for DB mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API (specifying a composite PK). There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

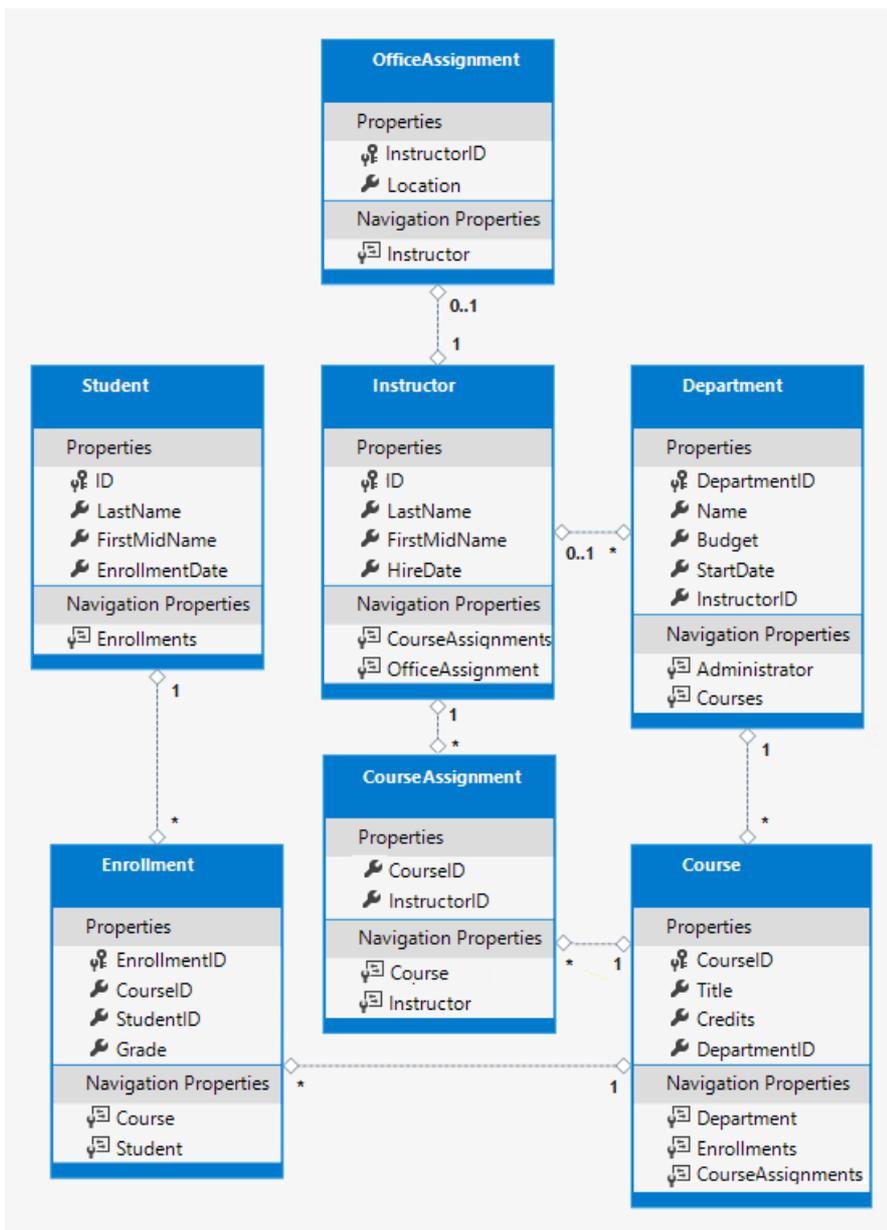
Some of the attributes used in the this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see [Methods of configuration](#).

Entity Diagram Showing Relationships

The following illustration shows the diagram that EF Power Tools create for the completed School model.



The preceding diagram shows:

- Several one-to-many relationship lines (1 to *).
- The one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment` entities.
- The zero-or-one-to-many relationship line (0..1 to *) between the `Instructor` and `Department` entities.

Seed the DB with Test Data

Update the code in *Data/DbInitializer.cs*:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return; // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson", LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo", LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis", LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan", LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy", LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura", LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino", LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim", LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi", LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger", LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger", LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            foreach (Instructor i in instructors)
            {
                context.Instructors.Add(i);
            }
        }
    }
}
```

```

}
context.SaveChanges();

var departments = new Department[]
{
    new Department { Name = "English",      Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID = instructors.Single( i => i.LastName == "Abercrombie").ID },
    new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID },
    new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID = instructors.Single( i => i.LastName == "Harui").ID },
    new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID }
};

foreach (Department d in departments)
{
    context.Departments.Add(d);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
    },
    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 1045, Title = "Calculus",      Credits = 4,
                DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 3141, Title = "Trigonometry",  Credits = 4,
                DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 2021, Title = "Composition",   Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
    new Course {CourseID = 2042, Title = "Literature",    Credits = 4,
                DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
};

foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var officeAssignments = new OfficeAssignment[]
{
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
        Location = "Smith 17" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
        Location = "Gowan 27" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
        Location = "Thompson 304" },
};

```

```

foreach (OfficeAssignment o in officeAssignments)
{
    context.OfficeAssignments.Add(o);
}
context.SaveChanges();

var courseInstructors = new CourseAssignment[]
{
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    },
};

foreach (CourseAssignment ci in courseInstructors)
{
    context.CourseAssignments.Add(ci);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        Grade = Grade.A
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        Grade = Grade.C
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        Grade = Grade.B
    }
};

```

```

    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Li").ID,
        CourseID = courses.Single(c => c.Title == "Composition").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Justice").ID,
        CourseID = courses.Single(c => c.Title == "Literature").CourseID,
        Grade = Grade.B
    }
    }
};

foreach (Enrollment e in enrollments)
{
    var enrollmentInDataBase = context.Enrollments.Where(
        s =>
            s.Student.ID == e.StudentID &&
            s.Course.CourseID == e.CourseID).SingleOrDefault();
    if (enrollmentInDataBase == null)
    {
        context.Enrollments.Add(e);
    }
}
context.SaveChanges();
}
}
}

```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing. The preceding code creates the following many-to-many relationships:

- `Enrollments`
- `CourseAssignment`

Note: [EF Core 2.1](#) will support [data seeding](#).

Add a migration

Build the project. Open a command window in the project folder and enter the following command:

```
dotnet ef migrations add ComplexDataModel
```

The preceding command displays a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If the `database update` command is run, the following error is produced:

```
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint
"FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in
database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.
```

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. For this tutorial, a new DB is created, so there are no FK constraint violations. See [Fixing foreign key constraints with legacy data](#) for instructions on how to fix the FK violations on the current DB.

Change the connection string and update the DB

The code in the updated `DbInitializer` adds seed data for the new entities. To force EF Core to create a new empty DB:

- Change the DB connection string name in `appsettings.json` to `ContosoUniversity3`. The new name must be a name that hasn't been used on the computer.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
}
```

- Alternatively, delete the DB using:
 - **SQL Server Object Explorer** (SSOX).
 - The `database drop` CLI command:

```
dotnet ef database drop
```

Run `database update` in the command window:

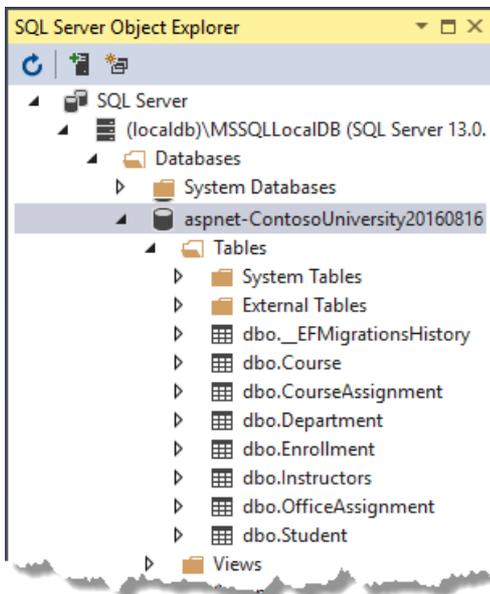
```
dotnet ef database update
```

The preceding command runs all the migrations.

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new DB.

Open the DB in SSOX:

- Expand the **Tables** node. The created tables are displayed.
- If SSOX was opened previously, click the **Refresh** button.



Examine the **CourseAssignment** table:

- Right-click the **CourseAssignment** table and select **View Data**.
- Verify the **CourseAssignment** table contains data.

	CourseID	InstructorID
▶	2021	1
	2042	1
	1045	2
	1050	3
	3141	3
	1050	4
	4022	5
	4041	5
*	NULL	NULL

Fixing foreign key constraints with legacy data

This section is optional.

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. With production data, steps must be taken to migrate the existing data. This section provides an example of fixing FK constraint violations. Don't make these code changes without a backup. Don't make these code changes if you completed the previous section and updated the database.

The `{timestamp}_ComplexDataModel.cs` file contains the following code:

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

The preceding code adds a non-nullable `DepartmentID` FK to the `Course` table. The DB from the previous tutorial contains rows in `Course`, so that table cannot be updated by migrations.

To make the `ComplexDataModel` migration work with existing data:

- Change the code to give the new column (`DepartmentID`) a default value.
- Create a fake department named "Temp" to act as the default department.

Fix the foreign key constraints

Update the `ComplexDataModel` classes `Up` method:

- Open the `{timestamp}_ComplexDataModel.cs` file.
- Comment out the line of code that adds the `DepartmentID` column to the `Course` table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

Add the following highlighted code. The new code goes after the `.CreateTable(name: "Department"` block:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(type: "int", nullable: true),
        Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(type: "datetime2", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

With the preceding changes, existing `Course` rows will be related to the "Temp" department after the `ComplexDataModel` `Up` method runs.

A production app would:

- Include code or scripts to add `Department` rows and related `Course` rows to the new `Department` rows.
- Would not use the "Temp" department or the default value for `Course.DepartmentID`.

The next tutorial covers related data.

[PREVIOUS](#)[NEXT](#)

Reading related data - EF Core with Razor Pages (6 of 8)

12/5/2017 • 14 min to read • [Edit Online](#)

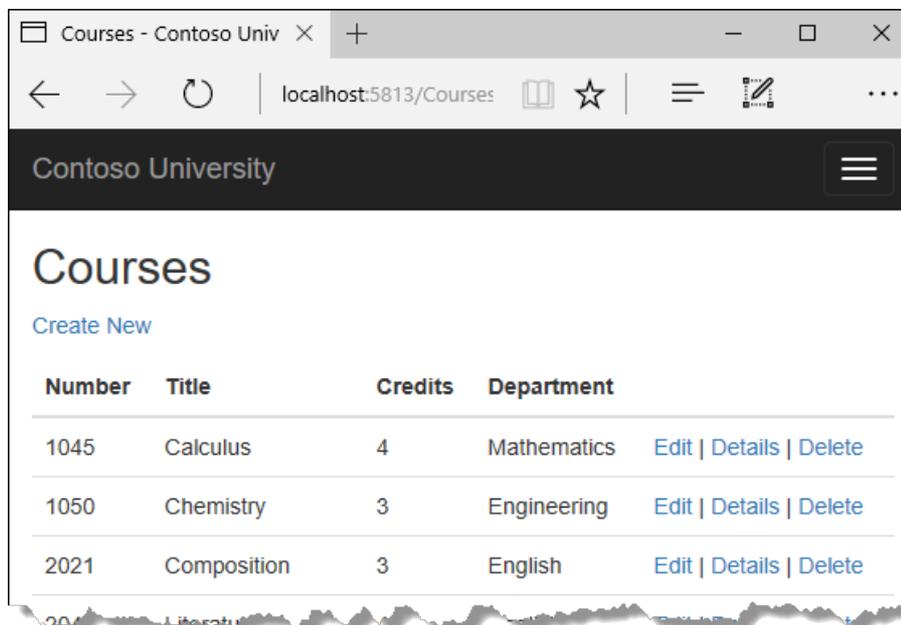
By [Tom Dykstra](#), [Jon P Smith](#), and [Rick Anderson](#)

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see [the first tutorial](#).

In this tutorial, related data is read and displayed. Related data is data that EF Core loads into navigation properties.

If you run into problems you can't solve, download the [completed app for this stage](#).

The following illustrations show the completed pages for this tutorial:



Instructors - Contoso University

localhost:1234/Instructors/3?courseID=1050&action=OnGetAsync

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete
Kapoor	Candace	2001-01-15	Thompson 304	1050 Chemistry	Select Edit Details Delete
Zheng	Roger	2004-02-12		4022 Microeconomics 4041 Macroeconomics	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
Alexander, Carson	A
Anand, Arturo	No grade
Barzdukas, Gytis	B

© 2017 - Contoso University

Eager, explicit, and lazy Loading of related data

There are several ways that EF Core can load related data into the navigation properties of an entity:

- **Eager loading.** Eager loading is when a query for one type of entity also loads related entities. When the entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than was the case for some queries in EF6 where there was a single query.

Eager loading is specified with the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

Eager loading sends multiple queries when a collection navigation is included:

- One query for the main query
- One query for each collection "edge" in the load tree.
- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "fixes up" means that EF Core automatically populates the navigation properties. Separate queries with `Load` is more like explicit loading than eager loading.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- **Explicit loading.** When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the DB. With explicit loading, the code specifies the navigation properties to be loaded. Use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

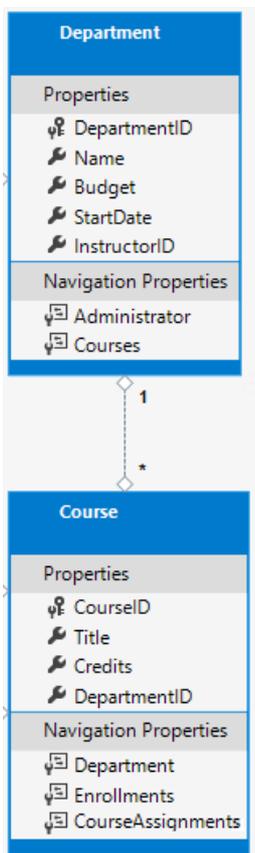
- **Lazy loading.** EF Core does not currently support lazy loading. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the DB each time a navigation property is accessed for the first time.
- The `Select` operator loads only the related data needed.

Create a Courses page that displays department name

The Course entity includes a navigation property that contains the `Department` entity. The `Department` entity contains the department that the course is assigned to.

To display the name of the assigned department in a list of courses:

- Get the `Name` property from the `Department` entity.
- The `Department` entity comes from the `Course.Department` navigation property.



Scaffold the Course model

- Exit Visual Studio.
- Open a command window in the project directory (The directory that contains the *Program.cs*, *Startup.cs*, and *.csproj* files).
- Run the following command:

```
dotnet aspnet-codegenerator razorpage -m Course -dc SchoolContext -udl -outDir Pages\Courses --referenceScriptLibraries
```

The preceding command scaffolds the `Course` model. Open the project in Visual Studio.

Build the project. The build generates errors like the following:

```
1>Pages/Courses/Index.cshtml.cs(26,37,26,43): error CS1061: 'SchoolContext' does not contain a definition for 'Course' and no extension method 'Course' accepting a first argument of type 'SchoolContext' could be found (are you missing a using directive or an assembly reference?)
```

Globally change `_context.Course` to `_context.Courses` (that is, add an "s" to `course`). 7 occurrences are found and updated.

Open *Pages/Courses/Index.cshtml.cs* and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which is not useful.

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The preceding code adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities are not tracked because they are not updated in the current context.

Update *Views/Courses/Index.cshtml* with the following highlighted markup:

```

@page
@model ContosoUniversity.Pages.Courses.IndexModel
@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

<p>
    <a asp-page="TestCreate">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Department)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Course)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

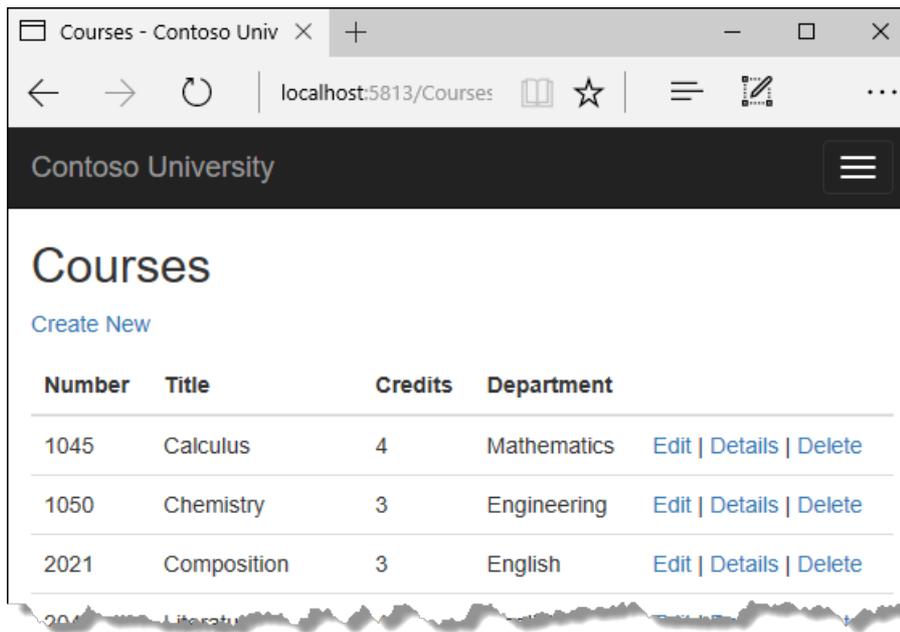
```

The following changes have been made to the scaffolded code:

- Changed the heading from Index to Courses.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they are meaningless to end users. However, in this case the primary key is meaningful.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The `Select` operator loads only the related data needed. For single items, like the `Department.Name` it uses a SQL INNER JOIN. For collections it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
        .Select(p => new CourseViewModel
        {
            CourseID = p.CourseID,
            Title = p.Title,
            Credits = p.Credits,
            DepartmentName = p.Department.Name
        }).ToListAsync();
}
```

The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See [IndexSelect.cshtml](#) and [IndexSelect.cshtml.cs](#) for a complete example.

Create an Instructors page that shows Courses and Enrollments

In this section, the Instructors page is created.

Instructors - Contoso University

localhost:1234/Instructors/3?courseID=1050&action=OnGetAsync

Instructors

Create New

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete
Kapoor	Candace	2001-01-15	Thompson 304	1050 Chemistry	Select Edit Details Delete
Zheng	Roger	2004-02-12		4022 Microeconomics 4041 Macroeconomics	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
Alexander, Carson	A
Anand, Arturo	No grade
Barzdukas, Gytis	B

© 2017 - Contoso University

This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor (Harui in the preceding image), related `Course` entities are displayed. The

`Instructor` and `Course` entities are in a many-to-many relationship. Eager loading for the `Course` entities and their related `Department` entities is used. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.

- When the user selects a course (Chemistry in the preceding image), related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model for the Instructor Index view

The instructors page shows data from three different tables. A view model is created that includes the three entities representing the three tables.

In the `SchoolViewModels` folder, create `InstructorIndexData.cs` with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Scaffold the Instructor model

- Exit Visual Studio.
- Open a command window in the project directory (The directory that contains the `Program.cs`, `Startup.cs`, and `.csproj` files).
- Run the following command:

```
dotnet aspnet-codegenerator razorpage -m Instructor -dc SchoolContext -udl -outDir Pages\Instructors --referenceScriptLibraries
```

The preceding command scaffolds the `Instructor` model. Open the project in Visual Studio.

Build the project. The build generates errors.

Globally change `_context.Instructor` to `_context.Instructors` (that is, add an "s" to `Instructor`). 7 occurrences are found and updated.

Run the app and navigate to the instructors page.

Replace `Pages/Instructors/Index.cshhtml.cs` with the following code:

```

using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels; // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData Instructor { get; set; }
        public int InstructorID { get; set; }

        public async Task OnGetAsync(int? id)
        {
            Instructor = new InstructorIndexData();
            Instructor.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                .AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
            }
        }
    }
}

```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query on the `Pages/Instructors/Index.cshtml` page:

```

Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

The query has two includes:

- `OfficeAssignment`: Displayed in the [instructors view](#).
- `CourseAssignments`: Which brings in the courses taught.

Update the instructors Index page

Update `Pages/Instructors/Index.cshtml` with the following markup:

```

@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructor.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @: @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

The preceding markup makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor when the page directive produces a URL like the following:

```
http://localhost:1234/Instructors?id=2
```

When the page directive is `@page "{id:int?}"`, the previous URL is:

```
http://localhost:1234/Instructors/2
```

- Page title is **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` is not null. Because this is a one-to-zero-or-one relationship, there might not be a related `OfficeAssignment` entity.

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

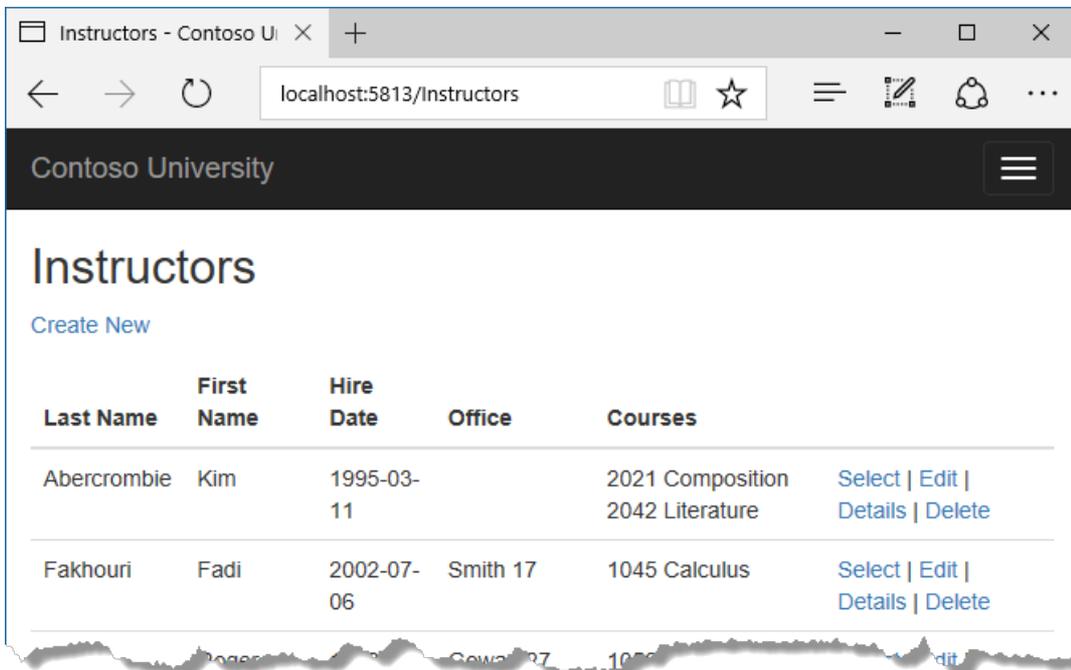
- Added a **Courses** column that displays courses taught by each instructor. See [Explicit Line Transition with @:](#) for more about this razor syntax.
- Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Added a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If `OfficeAssignment` is null, an empty table cell is displayed.



Click on the **Select** link. The row style changes.

Add courses taught by selected instructor

Update the `OnGetAsync` method in `Pages/Instructors/Index.cshtml.cs` with the following code:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }
}
```

Examine the updated query:

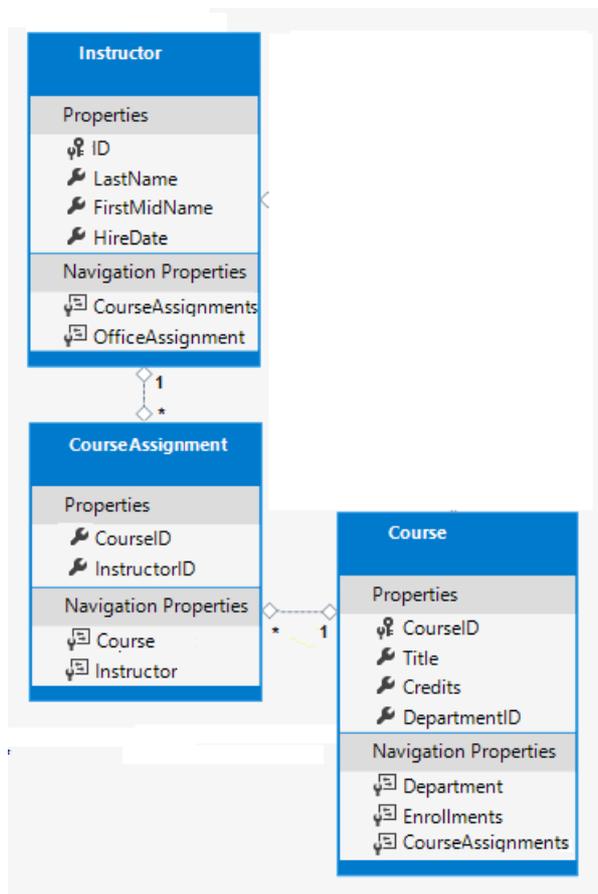
```
Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The preceding query adds the `Department` entities.

The following code executes when an instructor is selected (`id != null`). The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = Instructor.Instructors.Where(
        i => i.ID == id.Value).Single();
    Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `where` method returns a collection. In the preceding `where` method, only a single `Instructor` entity is returned. The `Single` method converts the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `CourseAssignments` property. `CourseAssignments` provides access to the related `Course` entities.



The `Single` method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. Using `SingleOrDefault` on an empty collection:

- Results in an exception (from trying to find a `Courses` property on a null reference).
- The exception message would less clearly indicate the cause of the problem.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    Instructor.Enrollments = Instructor.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Add the following markup to the end of the `Pages/Courses/Index.cshtml` Razor Page:

```

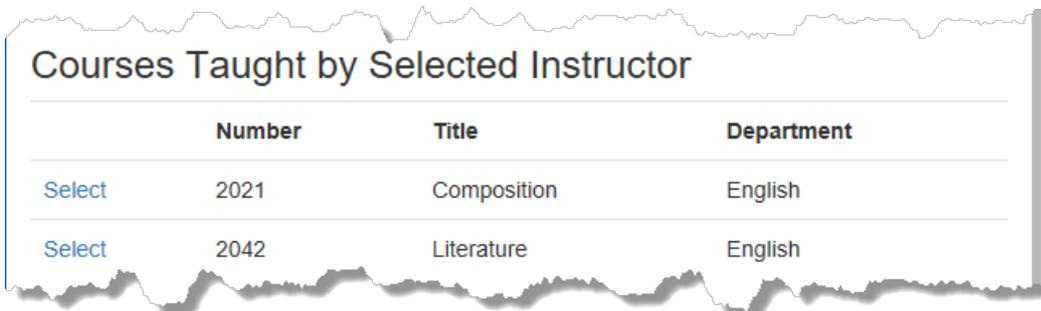
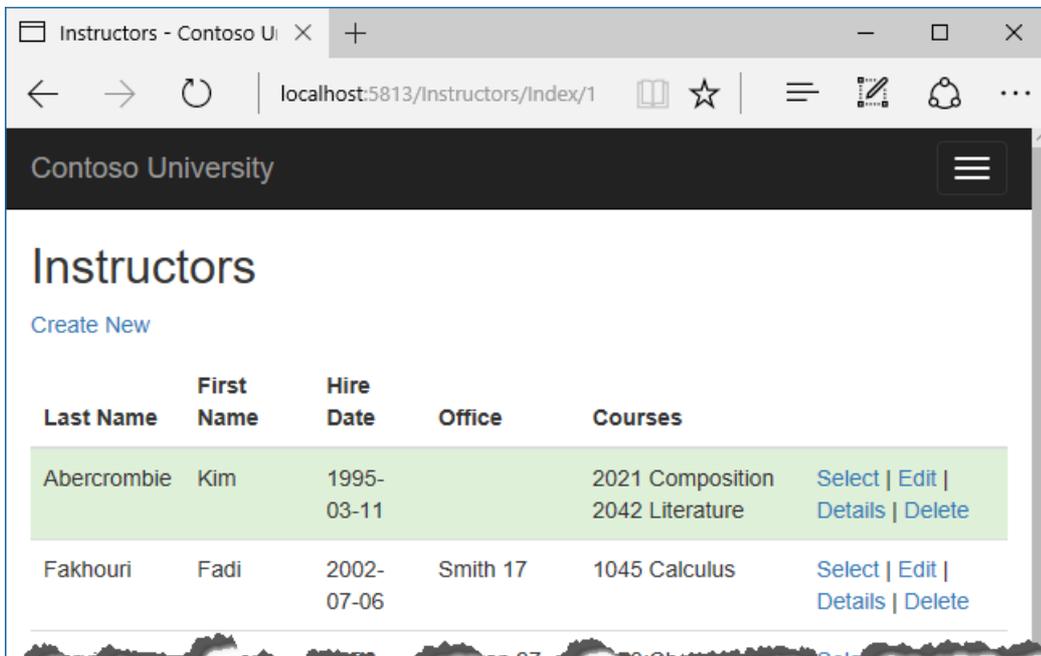
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>

@if (Model.Instructor.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Instructor.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "OnGetAsync",
                                    new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td> <td>
                    @item.Department.Name
                </td>
            </tr>
        }
    </table>
}
```

The preceding markup displays a list of courses related to an instructor when an instructor is selected.

Test the app. Click on a **Select** link on the instructors page.



Show student data

In this section, the app is updated to show the student data for a selected course.

Update the query in the `OnGetAsync` method in `Pages/Instructors/Index.cshtml.cs` with the following code:

```
Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

Update `Pages/Instructors/Index.cshtml`. Add the following markup to the end of the file:

```
@if (Model.Instructor.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Instructor.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding markup displays a list of the students who are enrolled in the selected course.

Refresh the page and select an instructor. Select a course to see the list of enrolled students and their grades.

Instructors - Contoso University

localhost:1234/Instructors/3?courseID=1050&action=OnGetAsync

Instructors

Create New

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete
Kapoor	Candace	2001-01-15	Thompson 304	1050 Chemistry	Select Edit Details Delete
Zheng	Roger	2004-02-12		4022 Microeconomics 4041 Macroeconomics	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
Alexander, Carson	A
Anand, Arturo	No grade
Barzdukas, Gytis	B

© 2017 - Contoso University

Using Single

The `Single` method can pass in the `where` condition instead of calling the `where` method separately:

```

public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();

    Instructor.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
        .ThenInclude(i => i.Student)
        .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Single(
            i => i.ID == id.Value);
        Instructor.Courses = instructor.CourseAssignments.Select(
            s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}

```

The preceding `Single` approach provides no benefits over using `Where`. Some developers prefer the `Single` approach style.

Explicit loading

The current code specifies eager loading for `Enrollments` and `Students`:

```

Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

Suppose users rarely want to see enrollments in a course. In that case, an optimization would be to only load the enrollment data if it's requested. In this section, the `OnGetAsync` is updated to use explicit loading of `Enrollments` and `Students`.

Update the `OnGetAsync` with the following code:

```

public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        // .Include(i => i.CourseAssignments)
        //     .ThenInclude(i => i.Course)
        //         .ThenInclude(i => i.Enrollments)
        //             .ThenInclude(i => i.Student)
        // .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        var selectedCourse = Instructor.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        Instructor.Enrollments = selectedCourse.Enrollments;
    }
}

```

The preceding code drops the *ThenInclude* method calls for enrollment and student data. If a course is selected, the highlighted code retrieves:

- The `Enrollment` entities for the selected course.
- The `Student` entities for each `Enrollment`.

Notice the preceding code comments out `.AsNoTracking()`. Navigation properties can only be explicitly loaded for tracked entities.

Test the app. From a users perspective, the app behaves identically to the previous version.

The next tutorial shows how to update related data.

[PREVIOUS](#)

[NEXT](#)

Updating related data - EF Core Razor Pages (7 of 8)

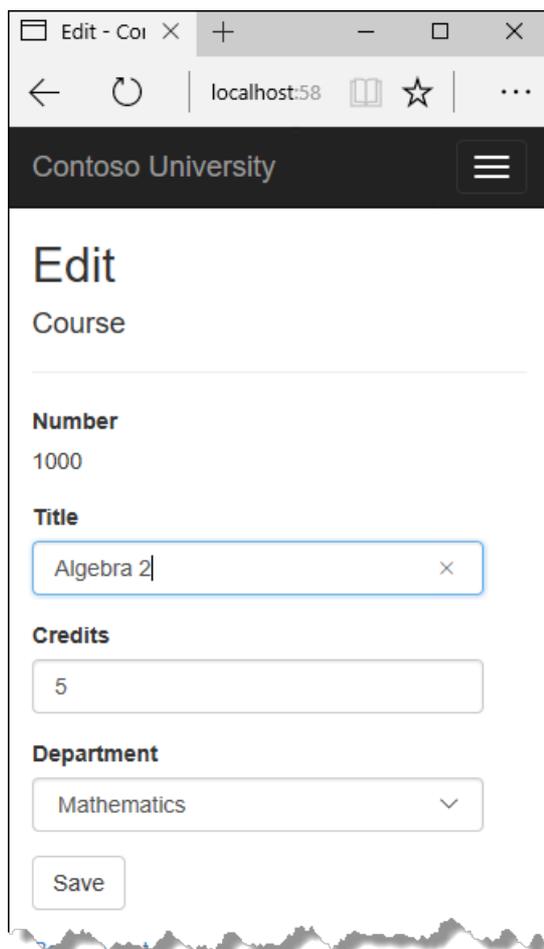
12/2/2017 • 16 min to read • [Edit Online](#)

By [Tom Dykstra](#), and [Rick Anderson](#)

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see [the first tutorial](#).

This tutorial demonstrates updating related data. If you run into problems you can't solve, download the [completed app for this stage](#).

The following illustrations shows some of the completed pages.



Edit - Contoso Universit × + - □ ×

localhost:5813/Instruct

Contoso University

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

1000 Algebra 2 1045 Calculus 1050 Chemistry
 2021 Composition 2042 Literature 3141 Trigonometry
 4022 Microeconomics 4041 Macroeconomics

Examine and test the Create and Edit course pages. Create a new course. The department is selected by its primary key (an integer), not its name. Edit the new course. When you have finished testing, delete the new course.

Create a base class to share common code

The Courses/Create and Courses/Edit pages each need a list of department names. Create the *Pages/Courses/DepartmentNamePageModel.cshtml.cs* base class for the Create and Edit pages:

```

using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                orderby d.Name // Sort by name.
                select d;

            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                "DepartmentID", "Name", selectedDepartment);
        }
    }
}

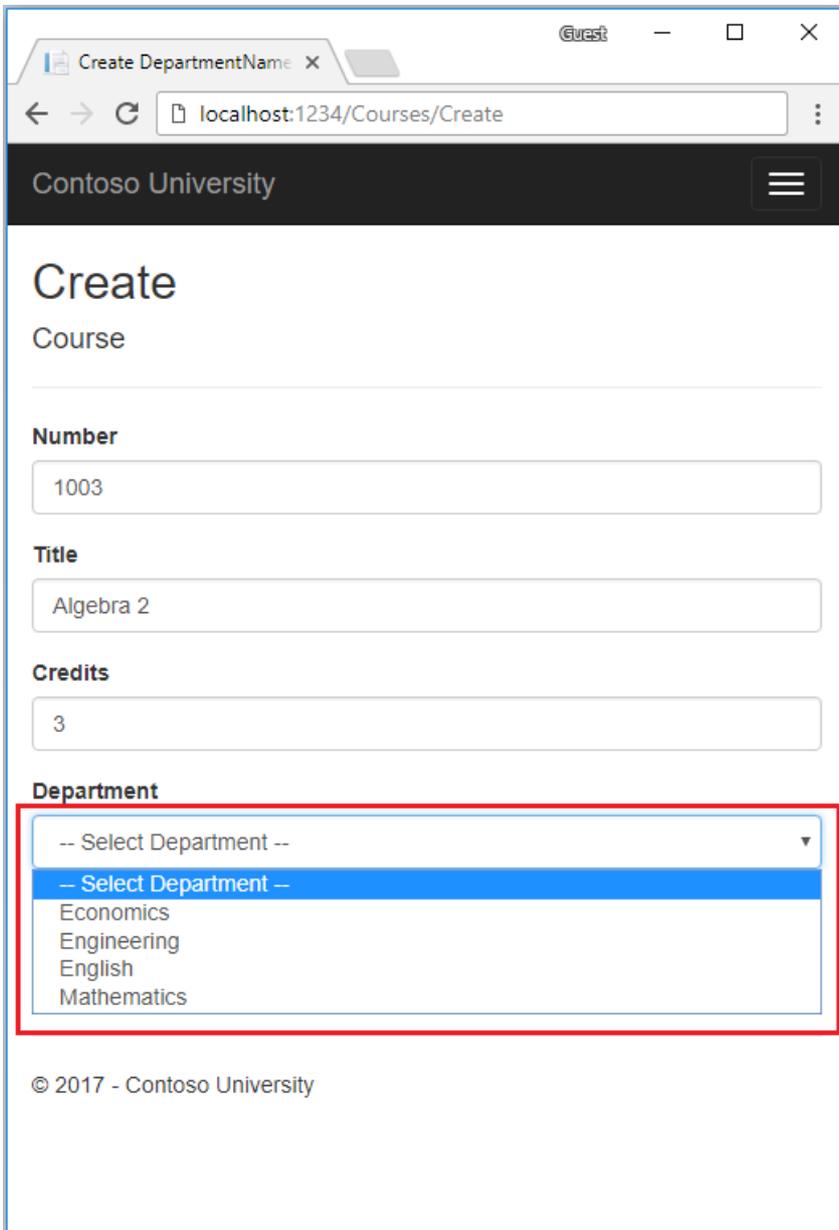
```

The preceding code creates a [SelectList](#) to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Customize the Courses Pages

When a new course entity is created, it must have a relationship to an existing department. To add a department while creating a course, the base class for Create and Edit contains a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update the Create page model with the following code:

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                emptyCourse,
                "course", // Prefix for form value.
                s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
    }
}

```

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses `TryUpdateModelAsync` to prevent [overposting](#).
- Replaces `ViewData["DepartmentID"]` with `DepartmentNameSL` (from the base class).

`ViewData["DepartmentID"]` is replaced with the strongly typed `DepartmentNameSL`. Strongly typed models are preferred over weakly typed. For more information, see [Weakly typed data \(ViewData and ViewBag\)](#).

Update the Courses Create page

Update `Pages/Courses/Create.cshtml` with the following markup:

```

@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                    asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

The preceding markup makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" rather than the first department.
- Adds a validation message when the department is not selected.

The Razor Page uses the [Select Tag Helper](#):

```
<div class="form-group">
  <label asp-for="Course.Department" class="control-label"></label>
  <select asp-for="Course.DepartmentID" class="form-control"
    asp-items="@Model.DepartmentNameSL">
    <option value="">-- Select Department --</option>
  </select>
  <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Courses Edit page.

Update the edit page model with the following code:

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (await TryUpdateModelAsync<Course>(
                courseToUpdate,
                "course", // Prefix for form value.
                c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}

```

The changes are similar to those made in the Create page model. In the preceding code,

`PopulateDepartmentsDropDownList` passes in the department ID, which select the department specified in the drop-

down list.

Update *Pages/Courses/Edit.cshtml* with the following markup:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                    asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity is not displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" rather than the first department.
- Adds a validation message when the department is not selected.

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for the course number to be included in the posted data when the user clicks **Save**.

Test the updated code. Create, edit, and delete a course.

Add AsNoTracking to the Details and Delete page models

[AsNoTracking](#) can improve performance when tracking is not required. Add `AsNoTracking` to the Delete and Details page model. The following code shows the updated Delete page model:

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Course Course { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .Include(c => c.Department)
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course != null)
        {
            _context.Courses.Remove(Course);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage("../Index");
    }
}
```

Update the `OnGetAsync` method in the `Pages/Courses/Details.cshtml.cs` file:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Course = await _context.Courses
        .AsNoTracking()
        .Include(c => c.Department)
        .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

Modify the Delete and Details pages

Update the Delete Razor page with the following markup:

```

@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Department.DepartmentID)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>

```

Make the same changes to the Details page.

Test the Course pages

Test create, edit, details, and delete.

Update the instructor pages

The following sections update the instructor pages.

Add office location

When editing an instructor record, you may want to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity. The instructor code must handle:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.

- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the instructors Edit page model with the following code:

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrEmpty(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");
    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property.
- Updates the retrieved `Instructor` entity with values from the model binder. `TryUpdateModel` prevents [overposting](#).
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.

Update the instructor Edit page

Update `Pages/Instructors/Edit.cshtml` with the office location:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Verify you can change an instructors office location.

Add Course assignments to the instructor Edit page

Instructors may teach any number of courses. In this section, you add the ability to change course assignments.

The following image shows the updated instructor Edit page:

Contoso University

Edit Instructor

Last Name
Abercrombie

First Name
Kim

Hire Date
3/11/1995

Office Location
44/3P

1000 Algebra 2 1045 Calculus 1050 Chemistry
 2021 Composition 2042 Literature 3141 Trigonometry
 4022 Microeconomics 4041 Macroeconomics

Save

`Course` and `Instructor` has a many-to-many relationship. To add and remove relationships, you add and remove entities from the `CourseAssignments` join entity set.

Check boxes enable changes to courses an instructor is assigned to. A check box is displayed for every course in the database. Courses that the instructor is assigned to are checked. The user can select or clear check boxes to change course assignments. If the number of courses were much greater:

- You'd probably use a different user interface to display the courses.
- The method of manipulating a join entity to create or delete relationships would not change.

Add classes to support Create and Edit instructor pages

Create `SchoolViewModels/AssignedCourseData.cs` with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the check boxes for assigned courses by an instructor.

Create the *Pages/Instructors/InstructorCoursesPageModel.cshtml.cs* base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {
        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
            Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            string[] selectedCourses, Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>(
                instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
            foreach (var course in context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.CourseAssignments.Add(
                            new CourseAssignment
                            {
                                InstructorID = instructorToUpdate.ID,
                                CourseID = course.CourseID
                            });
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        CourseAssignment courseToRemove
                            = instructorToUpdate
                                .CourseAssignments
                                .SingleOrDefault(i => i.CourseID == course.CourseID);
```

```
        context.Remove(courseToRemove);
    }
}
}
}
```

The `InstructorCoursesPageModel` is the base class you will use for the Edit and Create page models.

`PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A [HashSet](#) is used to create efficient lookups.

Instructors Edit page model

Update the instructor Edit page model with the following code:

```

public class EditModel : InstructorCoursesPageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        PopulateAssignedCourseData(_context, Instructor);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrEmpty(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            await _context.SaveChangesAsync();
            return RedirectToPage("../Index");
        }
        UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(_context, instructorToUpdate);
        return Page();
    }
}

```

The preceding code handles office assignment changes.

Update the instructor Razor View:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            <td>
                                @{
                                    int cnt = 0;

                                    foreach (var course in Model.AssignedCourseDataList)
                                    {
                                        if (cnt++ % 3 == 0)
                                        {
                                            @:</tr><tr>
                                        }
                                        @:<td>
                                            <input type="checkbox"
                                                name="selectedCourses"
                                                value="@course.CourseID"
                                                @(Html.Raw(course.Assigned ? "checked=\"checked\" : \"\")) />
                                            @course.CourseID @: @course.Title
                                        @:</td>
                                    }
                                @:</tr>
                            </td>
                        </tr>
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
```

```

</div>

<div>
  <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
  @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

NOTE

When you paste the code in Visual Studio, line breaks are changed in a way that breaks the code. Press Ctrl+Z one time to undo the automatic formatting. Ctrl+Z fixes the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown. With the block of new code selected, press Tab three times to line up the new code with the existing code. Vote on or review the status of this bug [with this link](#).

The preceding code creates an HTML table that has three columns. Each column has a check box and a caption containing the course number and title. The check boxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each check box is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the check boxes that are selected.

When the check boxes are initially rendered, courses assigned to the instructor have checked attributes.

Run the app and test the updated instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Note: The approach taken here to edit instructor course data works well when there is a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Update the instructors Create page

Update the instructor Create page model with the following code:

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
  public class CreateModel : InstructorCoursesPageModel
  {
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public CreateModel(ContosoUniversity.Data.SchoolContext context)
    {
      _context = context;
    }

    public IActionResult OnGet()
    {
      var instructor = new Instructor();
      instructor.CourseAssignments = new List<CourseAssignment>();

      // Provides an empty collection for the foreach loop
      // foreach (var course in Model.AssignedCourseDataList)
      // in the Create Razor page.
      PopulateAssignedCourseData( context, instructor);
    }
  }
}

```

```

        return Page();
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var newInstructor = new Instructor();
        if (selectedCourses != null)
        {
            newInstructor.CourseAssignments = new List<CourseAssignment>();
            foreach (var course in selectedCourses)
            {
                var courseToAdd = new CourseAssignment
                {
                    CourseID = int.Parse(course)
                };
                newInstructor.CourseAssignments.Add(courseToAdd);
            }
        }

        if (await TryUpdateModelAsync<Instructor>(
            newInstructor,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            _context.Instructors.Add(newInstructor);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        PopulateAssignedCourseData(_context, newInstructor);
        return Page();
    }
}
}

```

The preceding code is similar to the *Pages/Instructors/Edit.cshtml.cs* code.

Update the instructor Create Razor page with the following markup:

```

@page
@model ContosoUniversity.Pages.Instructors.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
        </form>
    </div>
</div>

```

```

<div class="form-group">
  <label asp-for="Instructor.FirstMidName" class="control-label"></label>
  <input asp-for="Instructor.FirstMidName" class="form-control" />
  <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
</div>
<div class="form-group">
  <label asp-for="Instructor.HireDate" class="control-label"></label>
  <input asp-for="Instructor.HireDate" class="form-control" />
  <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
</div>

<div class="form-group">
  <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
  <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
  <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
</div>
<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <table>
      <tr>
        @{
          int cnt = 0;

          foreach (var course in Model.AssignedCourseDataList)
          {
            if (cnt++ % 3 == 0)
            {
              @:</tr><tr>
            }
            @:<td>
              <input type="checkbox"
                name="selectedCourses"
                value="@course.CourseID"
                @(Html.Raw(course.Assigned ? "checked=\"checked\" : \"\")) />
              @course.CourseID @: @course.Title
            @:</td>
          }
        @:</tr>
      }
    </table>
  </div>
</div>
<div class="form-group">
  <input type="submit" value="Create" class="btn btn-default" />
</div>
</form>
</div>
</div>

<div>
  <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
  @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Test the instructor Create page.

Update the Delete page

Update the Delete page model with the following code:

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.SingleAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}

```

The preceding code makes the following changes:

- Uses eager loading for the `CourseAssignments` navigation property. `CourseAssignments` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

[PREVIOUS](#)

[NEXT](#)

en-us/

Handling concurrency conflicts - EF Core with Razor Pages (8 of 8)

By [Rick Anderson](#), [Tom Dykstra](#), and [Jon P Smith](#)

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see [the first tutorial](#).

This tutorial shows how to handle conflicts when multiple users update an entity concurrently (at the same time). If you run into problems you can't solve, download the [completed app for this stage](#).

Concurrency conflicts

A concurrency conflict occurs when:

- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the DB.

If concurrency detection is not enabled, when concurrent updates occur:

- The last update wins. That is, the last update values are saved to the DB.
- The first of the current updates are lost.

Optimistic concurrency

Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from \$350,000.00 to \$0.00.

Browser window: Edit - Cont x localhost:581

Contoso University

Edit

Department

Budget

Administrator

Abercrombie, Kim

Name

English

Start Date

9/1/2007

Save

Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.

Browser window: Edit - Cont x localhost:581

Contoso University

Edit

Department

Budget

Administrator

Abercrombie, Kim

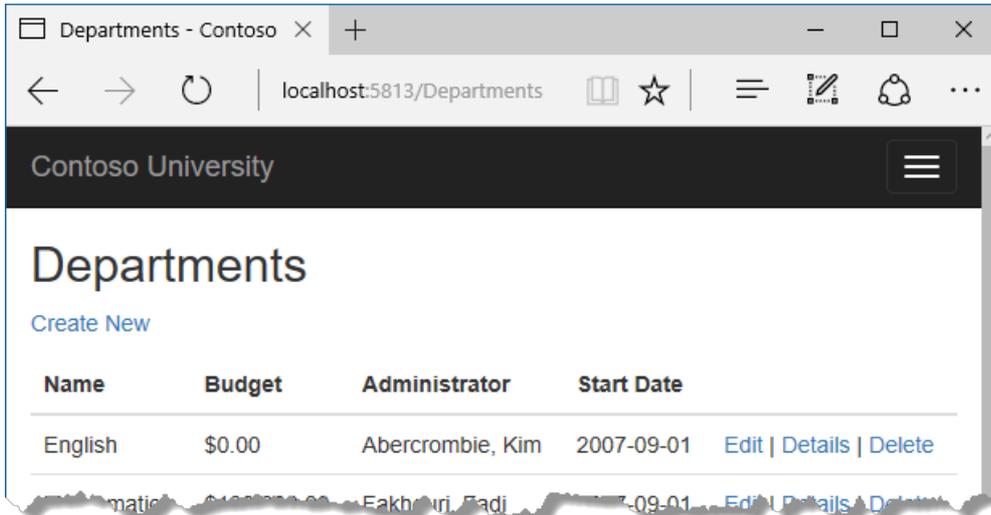
Name

English

Start Date

Save

Jane clicks **Save** first and sees her change when the browser displays the Index page.



John clicks **Save** on an Edit page that still shows a budget of \$350,000.00. What happens next is determined by how you handle concurrency conflicts.

Optimistic concurrency includes the following options:

- You can keep track of which property a user has modified and update only the corresponding columns in the DB.

In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they'll see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach:

- Can't avoid data loss if competing changes are made to the same property.
 - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.
 - Can increase app complexity compared to concurrency detection on an entity.
- You can let John's change overwrite Jane's change.

The next time someone browses the English department, they'll see 9/1/2013 and the fetched \$350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) If you don't do any coding for concurrency handling, Client Wins happens automatically.

- You can prevent John's change from being updated in the DB. Typically, the app would:
 - Display an error message.
 - Show the current state of the data.
 - Allow the user to reapply the changes.

This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Handling concurrency

When a property is configured as a [concurrency token](#):

- EF Core verifies that property has not been modified after it was fetched. The check occurs when [SaveChanges](#) or [SaveChangesAsync](#) is called.
- If the property has been changed after it was fetched, a [DbUpdateConcurrencyException](#) is thrown.

The DB and data model must be configured to support throwing `DbUpdateConcurrencyException`.

Detecting concurrency conflicts on a property

Concurrency conflicts can be detected at the property level with the `ConcurrencyCheck` attribute. The attribute can be applied to multiple properties on the model. For more information, see [Data Annotations-ConcurrencyCheck](#).

The `[ConcurrencyCheck]` attribute is not used in this tutorial.

Detecting concurrency conflicts on a row

To detect concurrency conflicts, a `rowversion` tracking column is added to the model. `rowversion` :

- Is SQL Server specific. Other databases may not provide a similar feature.
- Is used to determine that an entity has not been changed since it was fetched from the DB.

The DB generates a sequential `rowversion` number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the fetched value of `rowversion`. If the row being updated has changed:

- `rowversion` doesn't match the fetched value.
- The `Update` or `Delete` commands don't find a row because the `Where` clause includes the fetched `rowversion`.
- A `DbUpdateConcurrencyException` is thrown.

In EF Core, when no rows have been updated by an `Update` or `Delete` command, a concurrency exception is thrown.

Add a tracking property to the Department entity

In `Models/Department.cs`, add a tracking property named `RowVersion`:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}

```

The `Timestamp` attribute specifies that this column is included in the `where` clause of `Update` and `Delete` commands. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` type replaced it.

The fluent API can also specify the tracking property:

```

modelBuilder.Entity<Department>()
    .Property<byte[]>("RowVersion")
    .IsRowVersion();

```

The following code shows a portion of the T-SQL generated by EF Core when the Department name is updated:

```

SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;

```

The preceding highlighted code shows the `WHERE` clause containing `RowVersion`. If the DB `RowVersion` doesn't equal the `RowVersion` parameter (`@p2`), no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

```

SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;

```

`@@ROWCOUNT` returns the number of rows affected by the last statement. In no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

You can see the T-SQL EF Core generates in the output window of Visual Studio.

Update the DB

Adding the `RowVersion` property changes the DB model, which requires a migration.

Build the project. Enter the following in a command window:

```

dotnet ef migrations add RowVersion
dotnet ef database update

```

The preceding commands:

- Adds the `Migrations/{time stamp}_RowVersion.cs` migration file.
- Updates the `Migrations/SchoolContextModelSnapshot.cs` file. The update adds the following highlighted code to the `BuildModel` method:

```

modelBuilder.Entity("ContosoUniversity.Models.Department", b =>
{
    b.Property<int>("DepartmentID")
        .ValueGeneratedOnAdd();

    b.Property<decimal>("Budget")
        .HasColumnType("money");

    b.Property<int?>("InstructorID");

    b.Property<string>("Name")
        .HasMaxLength(50);

    b.Property<byte[]>("RowVersion")
        .IsConcurrencyToken()
        .ValueGeneratedOnAddOrUpdate();

    b.Property<DateTime>("StartDate");

    b.HasKey("DepartmentID");

    b.HasIndex("InstructorID");

    b.ToTable("Department");
});

```

- Runs migrations to update the DB.

Scaffold the Departments model

- Exit Visual Studio.
- Open a command window in the project directory (The directory that contains the `Program.cs`, `Startup.cs`, and `.csproj` files).

- Run the following command:

```
dotnet aspnet-codegenerator razorpage -m Department -dc SchoolContext -udl -outDir Pages\Departments --referenceScriptLibraries
```

The preceding command scaffolds the `Department` model. Open the project in Visual Studio.

Build the project. The build generates errors like the following:

```
1>Pages/Departments/Index.cshtml.cs(26,37,26,43): error CS1061: 'SchoolContext' does not contain a definition for 'Department' and no extension method 'Department' accepting a first argument of type 'SchoolContext' could be found (are you missing a using directive or an assembly reference?)
```

Globally change `_context.Department` to `_context.Departments` (that is, add an "s" to `Department`). 7 occurrences are found and updated.

Update the Departments Index page

The scaffolding engine created a `RowVersion` column for the Index page, but that field shouldn't be displayed. In this tutorial, the last byte of the `RowVersion` is displayed to help understand concurrency. The last byte is not guaranteed to be unique. A real app wouldn't display `RowVersion` or the last byte of `RowVersion`.

Update the Index page:

- Replace Index with Departments.
- Replace the markup containing `RowVersion` with the last byte of `RowVersion`.
- Replace FirstMidName with FullName.

The following markup shows the updated page:

```

@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Administrator)
            </th>
            <th>
                RowVersion
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Department) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    @item.RowVersion[7]
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Update the Edit page model

Update `pages\departments\edit.cshtml.cs` with the following code:

```
using ContosoUniversity.Data;
```

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator) // eager loading
                .AsNoTracking() // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            // null means Department was deleted by another user.
            if (departmentToUpdate == null)
            {
                return await HandleDeletedDepartment();
            }

            // Update the RowVersion to the value when this entity was
            // fetched. If the entity has been updated after it was
            // fetched, RowVersion won't match the DB RowVersion and
            // a DbUpdateConcurrencyException is thrown.
            // A second postback will make them match, unless a new
            // concurrency issue happens.
            _context.Entry(departmentToUpdate)
                .Property("RowVersion").OriginalValue = Department.RowVersion;
        }
    }
}

```

```

if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await setDbErrorMessage(dbValues, clientValues, _context);

        // Save the current RowVersion so next postback
        // matches unless an new concurrency issue happens.
        Department.RowVersion = (byte[])dbValues.RowVersion;
        // Must clear the model error for the next postback.
        ModelState.Remove("Department.RowVersion");
    }
}

InstructorNameSL = new SelectList(_context.Instructors,
    "ID", "FullName", departmentToUpdate.InstructorID);

return Page();
}

private async Task<IActionResult> HandleDeletedDepartment()
{
    Department deletedDepartment = new Department();
    // ModelState contains the posted data because of the deletion error and will override the
    Department instance values when displaying Page().
    ModelState.AddModelError(string.Empty,
        "Unable to save. The department was deleted by another user.");
    InstructorNameSL = new SelectList(_context.Instructors, "ID", "FullName",
    deletedDepartment.InstructorID);
    return Page();
}

private async Task setDbErrorMessage(Department dbValues,
    Department clientValues, SchoolContext context)
{
    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
}

```

```

    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
            .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
        + "was modified by another user after you. The "
        + "edit operation was canceled and the current values in the database "
        + "have been displayed. If you still want to edit this record, click "
        + "the Save button again.");
    }
}
}
}

```

To detect a concurrency issue, the `OriginalValue` is updated with the `rowVersion` value from the entity it was fetched. EF Core generates a SQL UPDATE command with a WHERE clause containing the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), a `DbUpdateConcurrencyException` exception is thrown.

```

public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    // null means Department was deleted by another user.
    if (departmentToUpdate == null)
    {
        return await HandleDeletedDepartment();
    }

    // Update the RowVersion to the value when this entity was
    // fetched. If the entity has been updated after it was
    // fetched, RowVersion won't match the DB RowVersion and
    // a DbUpdateConcurrencyException is thrown.
    // A second postback will make them match, unless a new
    // concurrency issue happens.
    _context.Entry(departmentToUpdate)
        .Property("RowVersion").OriginalValue = Department.RowVersion;
}

```

In the preceding code, `Department.RowVersion` is the value when the entity was fetched. `OriginalValue` is the value in the DB when `FirstOrDefaultAsync` was called in this method.

The following code gets the client values (the values posted to this method) and the DB values:

```

try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless an new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
}

```

The following code adds a custom error message for each column that has DB values different from what was posted to `OnPostAsync`:

```

private async Task setDbErrorMessage(Department dbValues,
    Department clientValues, SchoolContext context)
{
    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
            .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
        + "was modified by another user after you. The "
        + "edit operation was canceled and the current values in the database "
        + "have been displayed. If you still want to edit this record, click "
        + "the Save button again.");
}
}

```

The following highlighted code sets the `RowVersion` value to the new value retrieved from the DB. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless a new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
}
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Edit page

Update `Pages/Departments/Edit.cshtml` with the following markup:

```

@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.RowVersion" />
            <div class="form-group">
                <label>RowVersion</label>
                @Model.Department.RowVersion[7]
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label"></label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label"></label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label"></label>
                <input asp-for="Department.StartDate" class="form-control" />
                <span asp-validation-for="Department.StartDate" class="text-danger">
                    </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
                <select asp-for="Department.InstructorID" class="form-control"
                    asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID" class="text-danger">
                    </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

The preceding markup:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `RowVersion` must be added so post back binds the value.
- Displays the last byte of `RowVersion` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

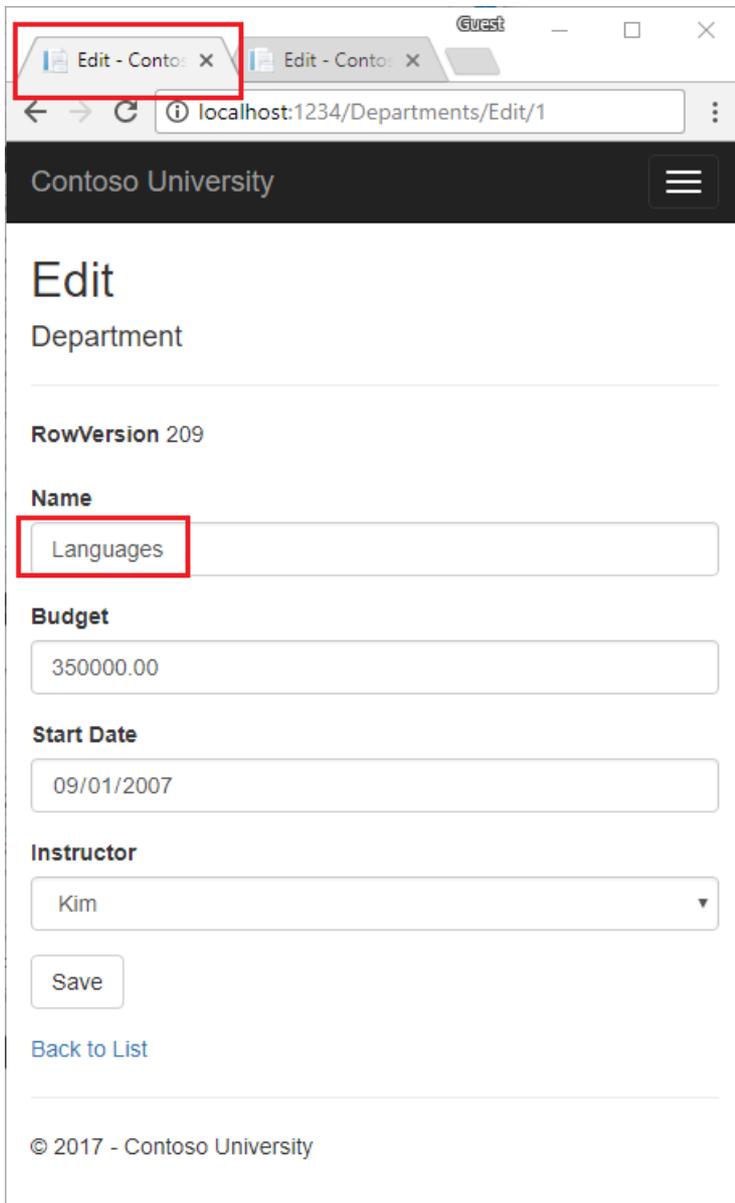
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.



The screenshot shows a web browser window with two tabs. The active tab is titled 'Edit - Contoso' and shows the URL 'localhost:1234/Departments/Edit/1'. The page content includes a header for 'Contoso University' and a main heading 'Edit Department'. Below this, there is a 'RowVersion 209' indicator. The 'Name' field is highlighted with a red box and contains the text 'Languages'. Other fields include 'Budget' (350000.00), 'Start Date' (09/01/2007), and 'Instructor' (Kim). A 'Save' button is visible at the bottom of the form, along with a 'Back to List' link. The footer of the page reads '© 2017 - Contoso University'.

The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it is displayed on the second postback in the other tab.

Change a different field in the second browser tab.

Department: x Edit - Contoso: x

localhost:1234/Departments/Edit/1

Contoso University

Edit

Department

RowVersion 209

Name

English

Budget

5000000

Start Date

09/01/2007

Instructor

Kim

Save

[Back to List](#)

© 2017 - Contoso University

Click **Save**. You see error messages for all fields that don't match the DB values:

Departments x Edit - Contos x

localhost:1234/Departments/Edit/1

Contoso University

Edit

Department

- The record you attempted to edit was modified by another user after you. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again.

RowVersion 21

Name

Current value: Languages

Budget

Current value: \$350,000.00

Start Date

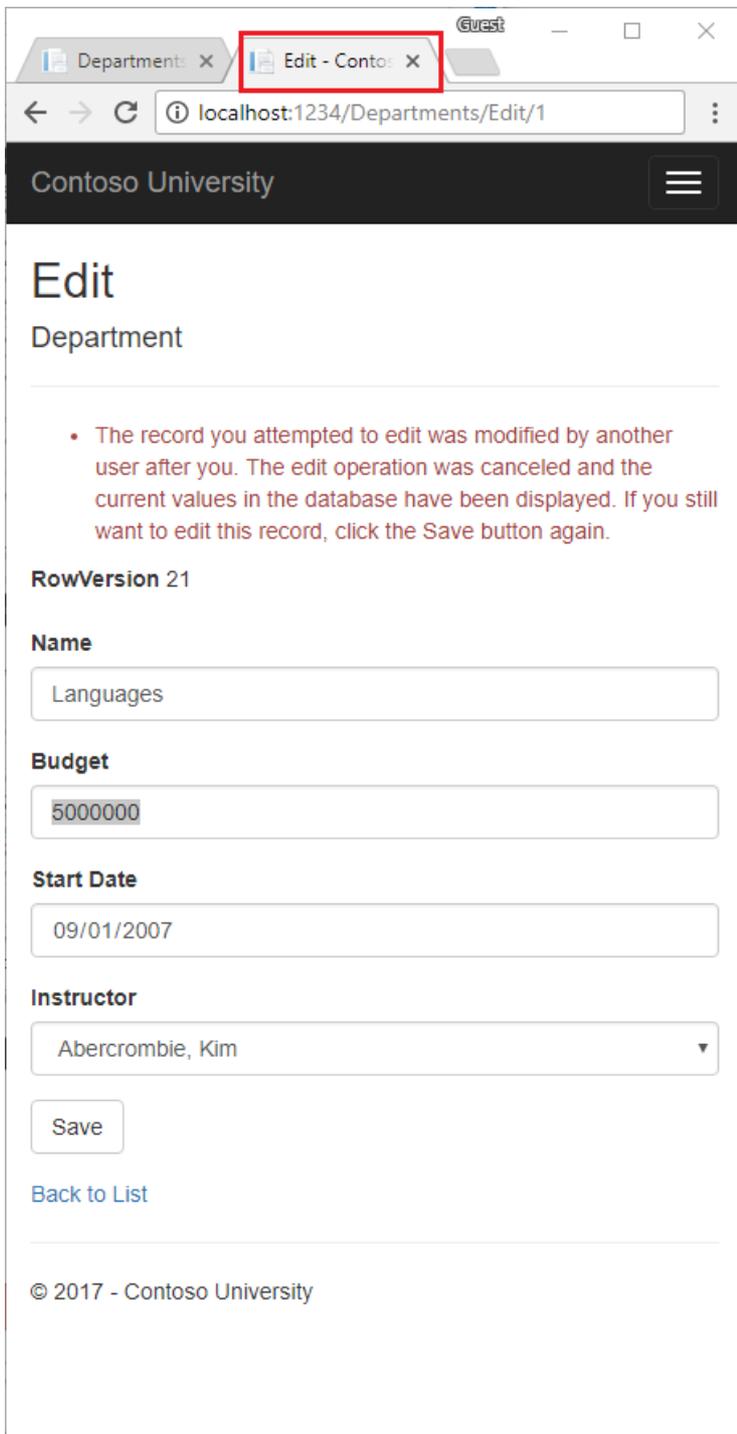
Instructor

Save

[Back to List](#)

© 2017 - Contoso University

This browser window did not intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page

Update the Delete page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
```

```

public DeleteModel(ContosoUniversity.Data.SchoolContext context)
{
    _context = context;
}

[BindProperty]
public Department Department { get; set; }
public string ConcurrencyErrorMessage { get; set; }

public async Task<IActionResult> OnGetAsync(int id, bool? concurrencyError)
{
    Department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (Department == null)
    {
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ConcurrencyErrorMessage = "The record you attempted to delete "
            + "was modified by another user after you selected delete. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again.";
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    try
    {
        if (await _context.Departments.AnyAsync(
            m => m.DepartmentID == id))
        {
            // Department.RowVersion value is from when the entity
            // was fetched. If it doesn't match the DB, a
            // DbUpdateConcurrencyException exception is thrown.
            _context.Departments.Remove(Department);
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException)
    {
        return RedirectToPage("./Delete",
            new { concurrencyError = true, id = id });
    }
}
}

```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched.

`Department.RowVersion` is the row version when the entity was fetched. When EF Core creates the SQL DELETE command, it includes a WHERE clause with `RowVersion`. If the SQL DELETE command results in zero rows affected:

- The `RowVersion` in the SQL DELETE command doesn't match `RowVersion` in the DB.
- A `DbUpdateConcurrencyException` exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete page

Update *Pages/Departments/Delete.cshtml* with the following code:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.RowVersion)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.RowVersion[7])
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-page="./Index">Back to List</a>
        </div>
    </form>
</div>
```

The preceding markup makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.
- Replaces `FirstMidName` with `FullName` in the **Administrator** field.

- Changes `RowVersion` to display the last byte.
- Adds a hidden row version. `RowVersion` must be added so post back binds the value.

Test concurrency conflicts with the Delete page

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it is displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the DB. Clicking **Delete** deletes the entity, unless `RowVersion` has been updated.department has been deleted.

See [Inheritance](#) on how to inherit a data model.

Additional resources

- [Concurrency Tokens in EF Core](#)
- [Handling concurrency in EF Core](#)

PREVIOUS

Getting started with ASP.NET Core MVC and Entity Framework Core using Visual Studio

12/13/2017 • 1 min to read • [Edit Online](#)

Note: A Razor Pages version of this tutorial is available [here](#). The Razor Pages version is easier to follow and covers more EF features.

This series of tutorials teaches you how to create ASP.NET Core MVC web applications that use Entity Framework Core for data access. The tutorials require Visual Studio 2017.

1. [Getting started](#)
2. [Create, Read, Update, and Delete operations](#)
3. [Sorting, filtering, paging, and grouping](#)
4. [Migrations](#)
5. [Creating a complex data model](#)
6. [Reading related data](#)
7. [Updating related data](#)
8. [Handling concurrency conflicts](#)
9. [Inheritance](#)
10. [Advanced topics](#)

Getting started with ASP.NET Core MVC and Entity Framework Core using Visual Studio (1 of 10)

12/9/2017 • 20 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

A Razor Pages version of this tutorial is available [here](#). The Razor Pages version is easier to follow and covers more EF features. We recommend you follow the [Razor Pages version of this tutorial](#).

The Contoso University sample web application demonstrates how to create ASP.NET Core 2.0 MVC web applications using Entity Framework (EF) Core 2.0 and Visual Studio 2017.

The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This is the first in a series of tutorials that explain how to build the Contoso University sample application from scratch.

[Download or view the completed application.](#)

EF Core 2.0 is the latest version of EF but does not yet have all the features of EF 6.x. For information about how to choose between EF 6.x and EF Core, see [EF Core vs. EF6.x](#). If you choose EF 6.x, see [the previous version of this tutorial series](#).

NOTE

- For the ASP.NET Core 1.1 version of this tutorial, see the [VS 2017 Update 2 version of this tutorial in PDF format](#).
- For the Visual Studio 2015 version of this tutorial, see the [VS 2015 version of ASP.NET Core documentation in PDF format](#).

Prerequisites

Install the following:

- [.NET Core 2.0.0 SDK](#) or later.
- [Visual Studio 2017](#) version 15.3 or later with the **ASP.NET and web development** workload.

Troubleshooting

If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the [completed project](#). For a list of common errors and how to solve them, see [the Troubleshooting section of the last tutorial in the series](#). If you don't find what you need there, you can post a question to [StackOverflow.com](#) for [ASP.NET Core](#) or [EF Core](#).

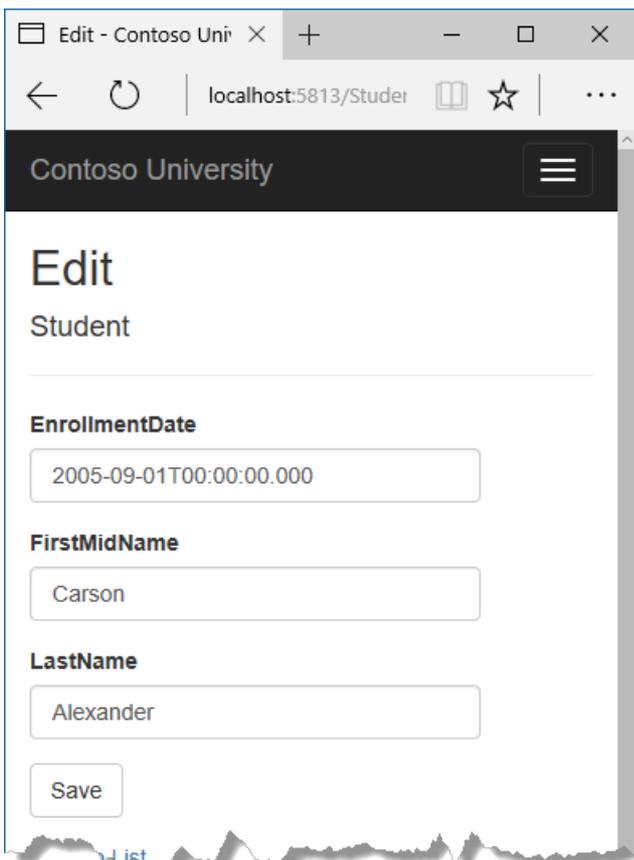
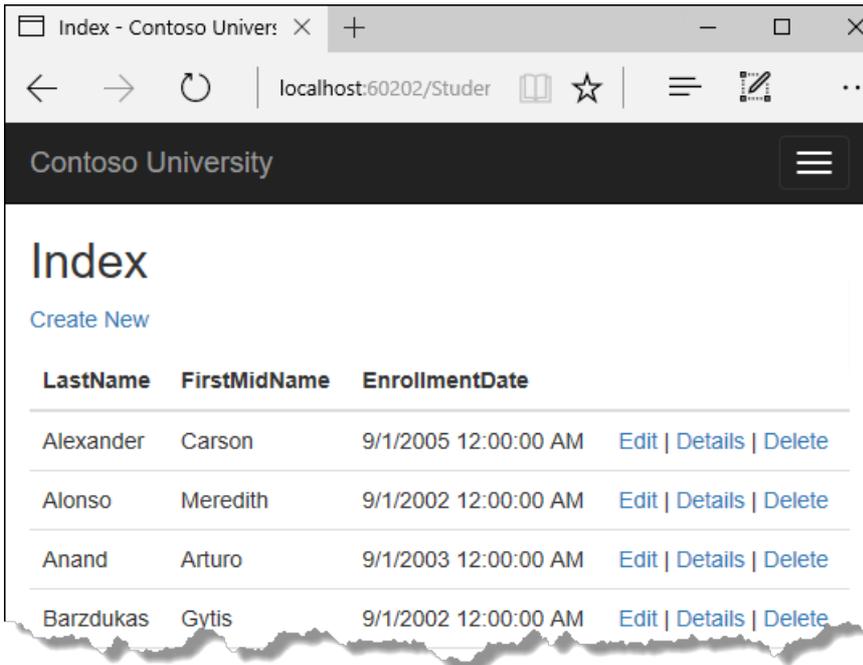
TIP

This is a series of 10 tutorials, each of which builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. Then if you run into problems, you can start over from the previous tutorial instead of going back to the beginning of the whole series.

The Contoso University web application

The application you'll be building in these tutorials is a simple university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens you'll create.



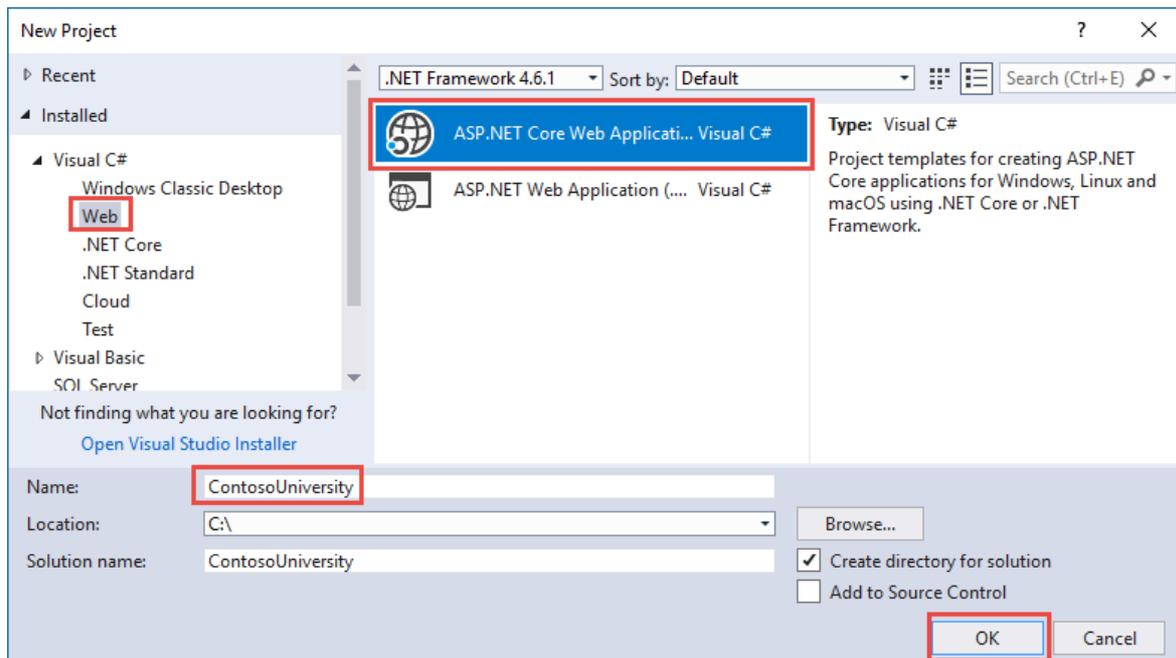
The UI style of this site has been kept close to what's generated by the built-in templates, so that the tutorial can focus mainly on how to use the Entity Framework.

Create an ASP.NET Core MVC web application

Open Visual Studio and create a new ASP.NET Core C# web project named "ContosoUniversity".

- From the **File** menu, select **New > Project**.

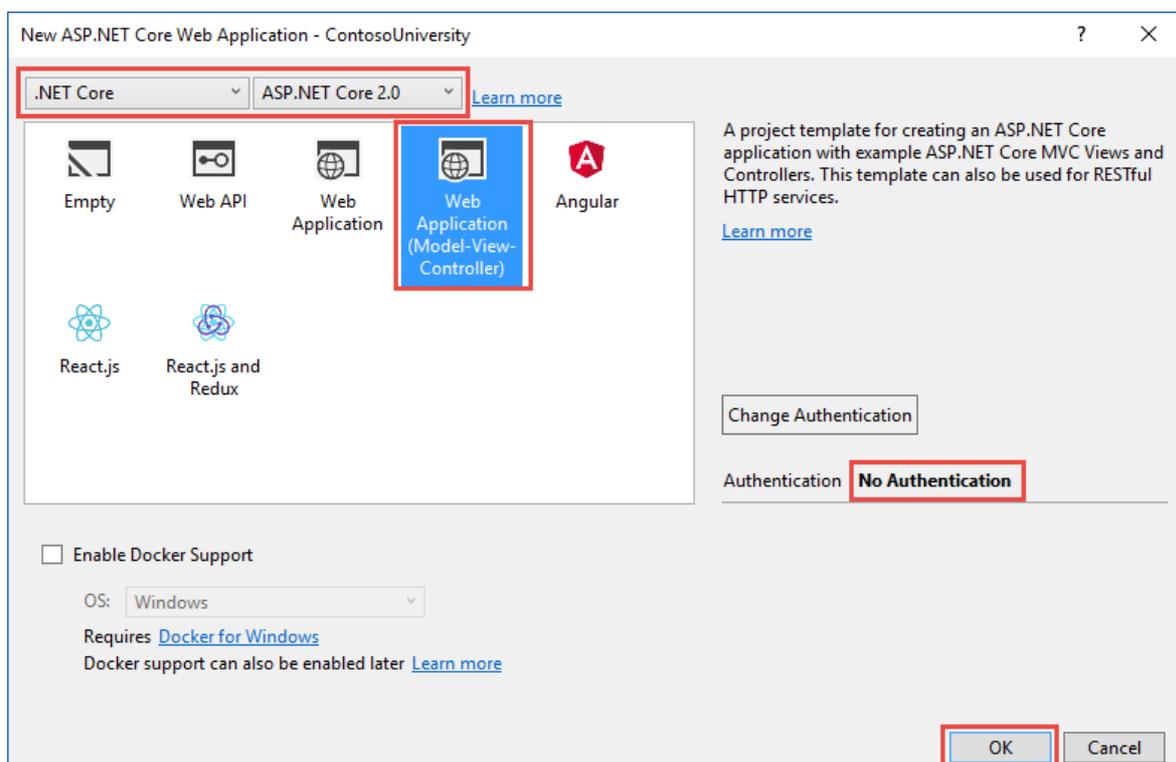
- From the left pane, select **Installed > Visual C# > Web**.
- Select the **ASP.NET Core Web Application** project template.
- Enter **ContosoUniversity** as the name and click **OK**.



- Wait for the **New ASP.NET Core Web Application (.NET Core)** dialog to appear
- Select **ASP.NET Core 2.0** and the **Web Application (Model-View-Controller)** template.

Note: This tutorial requires ASP.NET Core 2.0 and EF Core 2.0 or later -- make sure that **ASP.NET Core 1.1** is not selected.

- Make sure **Authentication** is set to **No Authentication**.
- Click **OK**



Set up the site style

A few simple changes will set up the site menu, layout, and home page.

Open *Views/Shared/_Layout.cshtml* and make the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Add menu entries for **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Contact** menu entry.

The changes are highlighted.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - Contoso University</title>

  <environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment names="Staging,Production">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>

</head>
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">

          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">Contoso
University</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
          <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
          <li><a asp-area="" asp-controller="Students" asp-action="Index">Students</a></li>
          <li><a asp-area="" asp-controller="Courses" asp-action="Index">Courses</a></li>
          <li><a asp-area="" asp-controller="Instructors" asp-action="Index">Instructors</a></li>
          <li><a asp-area="" asp-controller="Departments" asp-action="Index">Departments</a></li>
        </ul>
      </div>
    </div>
  </nav>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; 2017 - Contoso University</p>
    </footer>
  </div>
```

```

<environment names="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery"
    crossorigin="anonymous"
    integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
    asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
    crossorigin="anonymous"
    integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIpG9mGCD8wGNIcPD7Txa">
  </script>
  <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

  @RenderSection("Scripts", required: false)
</body>
</html>

```

In *Views/Home/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this application:

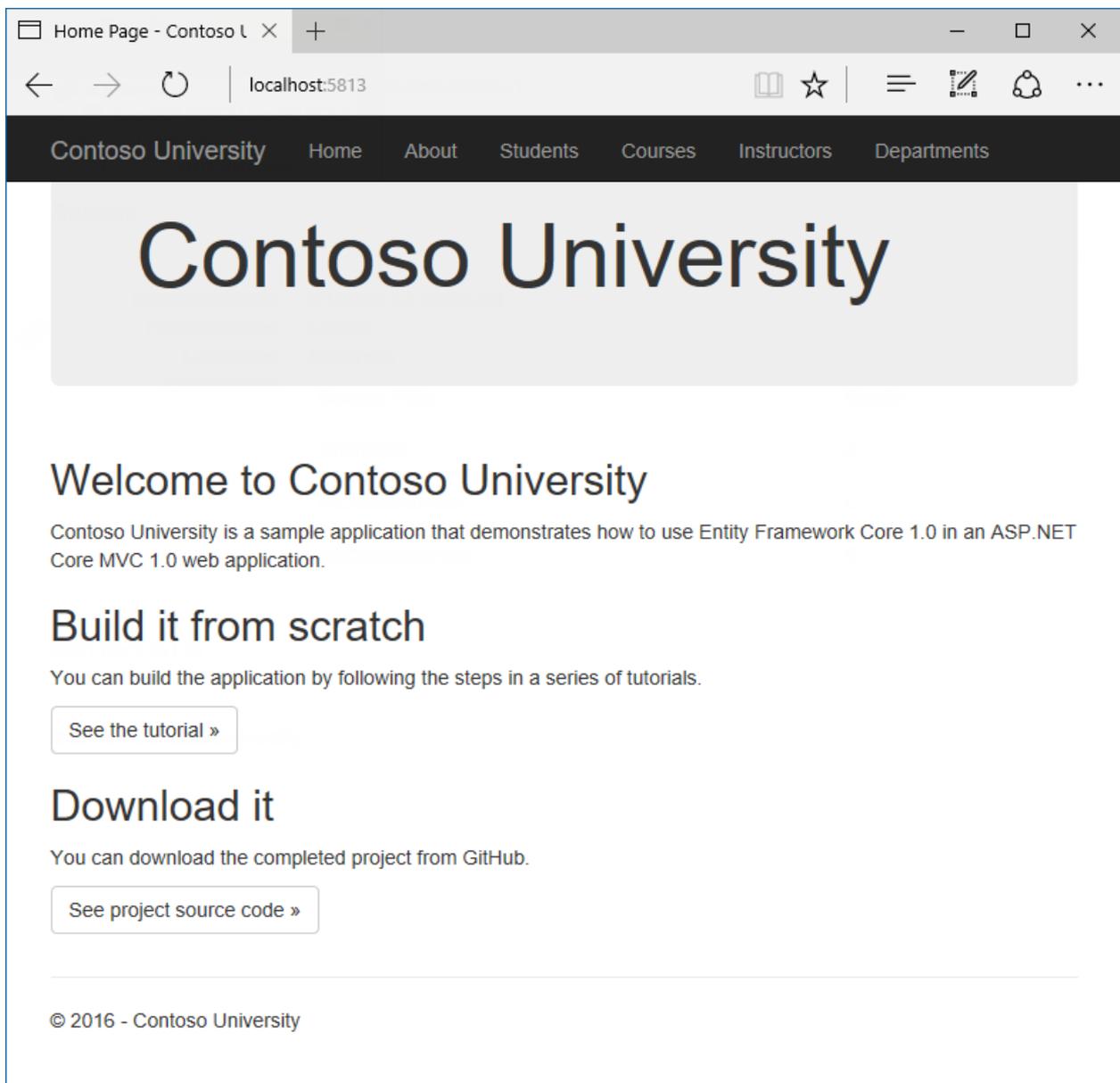
```

@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default" href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
        tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default" href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-
        mvc/intro/samples/cu-final">See project source code &raquo;</a></p>
    </div>
</div>

```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu. You see the home page with tabs for the pages you'll create in these tutorials.



Entity Framework Core NuGet packages

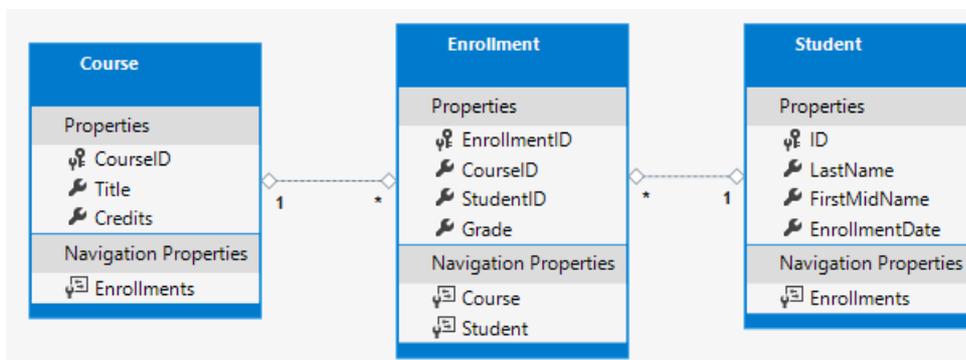
To add EF Core support to a project, install the database provider that you want to target. This tutorial uses SQL Server, and the provider package is [Microsoft.EntityFrameworkCore.SqlServer](#). This package is included in the [Microsoft.AspNetCore.All](#) metapackage, so you don't have to install it.

This package and its dependencies (`Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`) provide runtime support for EF. You'll add a tooling package later, in the [Migrations](#) tutorial.

For information about other database providers that are available for Entity Framework Core, see [Database providers](#).

Create the data model

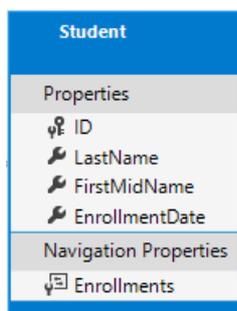
Next you'll create entity classes for the Contoso University application. You'll start with the following three entities.



There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections you'll create a class for each one of these entities.

The Student entity



In the `Models` folder, create a class file named `Student.cs` and replace the template code with the following code.

```

using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
  
```

The `ID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or `classnameID` as the primary key.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given `Student` row in the database has two related `Enrollment` rows (rows that contain that student's primary key value in their `StudentID` foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection<T>`. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The Enrollment entity

Enrollment	
Properties	
	EnrollmentID
	CourseID
	StudentID
	Grade
Navigation Properties	
	Course
	Student

In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property will be the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself as you saw in the `Student` entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a [later tutorial](#), you'll see how using `ID` without classname makes it easier to implement inheritance in the data model.

The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a foreign key property if it's named `<navigation property name><primary key property name>` (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply `<primary key property name>` (for example, `CourseID` since the `Course` entity's primary key is `CourseID`).

The Course entity

Course
Properties
<ul style="list-style-type: none"> CourseID Title Credits
Navigation Properties
<ul style="list-style-type: none"> Enrollments

In the *Models* folder, create *Course.cs* and replace the existing code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the `DatabaseGenerated` attribute in a [later tutorial](#) in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the Database Context

The main class that coordinates Entity Framework functionality for a given data model is the database context class. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. In your code you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named *Data*.

In the *Data* folder create a new class file named *SchoolContext.cs*, and replace the template code with the following code:

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}

```

This code creates a `DbSet` property for each entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.

You could have omitted the `DbSet<Enrollment>` and `DbSet<Course>` statements and it would work the same. The Entity Framework would include them implicitly because the `Student` entity references the `Enrollment` entity and the `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural (Students rather than Student), but developers disagree about whether table names should be pluralized or not. For these tutorials you'll override the default behavior by specifying singular table names in the `DbContext`. To do that, add the following highlighted code after the last `DbSet` property.

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}

```

Register the context with dependency injection

ASP.NET Core implements [dependency injection](#) by default. Services (such as the EF database context) are registered with dependency injection during application startup. Components that require these services (such as MVC controllers) are provided these services via constructor parameters. You'll see the controller constructor code that gets a context instance later in this tutorial.

To register `SchoolContext` as a service, open `Startup.cs`, and add the highlighted lines to the `ConfigureServices` method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the [ASP.NET Core configuration system](#) reads the connection string from the `appsettings.json` file.

Add `using` statements for `ContosoUniversity.Data` and `Microsoft.EntityFrameworkCore` namespaces, and then build the project.

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

Open the `appsettings.json` file and add a connection string as shown in the following example.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

SQL Server Express LocalDB

The connection string specifies a SQL Server LocalDB database. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for application development, not production use. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB creates `.mdf` database files in the `C:/Users/<user>` directory.

Add code to initialize the database with test data

The Entity Framework will create an empty database for you. In this section, you write a method that is called after the database is created in order to populate it with test data.

Here you'll use the `EnsureCreated` method to automatically create the database. In a [later tutorial](#) you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the `Data` folder, create a new class file named `DbInitializer.cs` and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
using ContosoUniversity.Models;
using System;
using System.Linq;
```

```

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return; // DB has been seeded
            }

            var students = new Student[]
            {
                new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")},
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {

```

```

        context.Enrollments.Add(e);
    }
    context.SaveChanges();
}
}
}

```

The code checks if there are any students in the database, and if not, it assumes the database is new and needs to be seeded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

In *Program.cs*, modify the `Main` method to do the following on application startup:

- Get a database context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method is done.

```

public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}

```

Add `using` statements:

```

using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;

```

In older tutorials, you may see similar code in the `Configure` method in *Startup.cs*. We recommend that you use the `Configure` method only to set up the request pipeline. Application startup code belongs in the `Main` method.

Now the first time you run the application, the database will be created and seeded with test data. Whenever you change your data model, you can delete the database, update your seed method, and start afresh with a new database the same way. In later tutorials, you'll see how to modify the database when the data model changes, without deleting and re-creating it.

Create a controller and views

Next, you'll use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding. Scaffolding differs from

code generation in that the scaffolded code is a starting point that you can modify to suit your own requirements, whereas you typically don't modify generated code. When you need to customize generated code, you use partial classes or you regenerate the code when things change.

- Right-click the **Controllers** folder in **Solution Explorer** and select **Add > New Scaffolded Item**.

If the **Add MVC Dependencies** dialog appears:

- [Update Visual Studio to the latest version](#). Visual Studio versions prior to 15.5 show this dialog.
- If you can't update, select **ADD**, and then follow the add controller steps again.

- In the **Add Scaffold** dialog box:

- Select **MVC controller with views, using Entity Framework**.
- Click **Add**.

- In the **Add Controller** dialog box:

- In **Model class** select **Student**.
- In **Data context class** select **SchoolContext**.
- Accept the default **StudentsController** as the name.
- Click **Add**.

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Student (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. Under the 'Views:' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. The 'Controller name' text box contains 'StudentsController'. At the bottom right, the 'Add' button is highlighted with a red box.

When you click **Add**, the Visual Studio scaffolding engine creates a *StudentsController.cs* file and a set of views (*.cshtml* files) that work with the controller.

(The scaffolding engine can also create the database context for you if you don't create it manually first as you did earlier for this tutorial. You can specify a new context class in the **Add Controller** box by clicking the plus sign to the right of **Data context class**. Visual Studio will then create your `DbContext` class as well as the controller and views.)

You'll notice that the controller takes a `SchoolContext` as a constructor parameter.

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
    }
}
```

ASP.NET dependency injection will take care of passing an instance of `SchoolContext` into the controller. You configured that in the `Startup.cs` file earlier.

The controller contains an `Index` action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the `Students` property of the database context instance:

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

You'll learn about the asynchronous programming elements in this code later in the tutorial.

The `Views/Students/Index.cshtml` view displays this list in a table:

```

@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

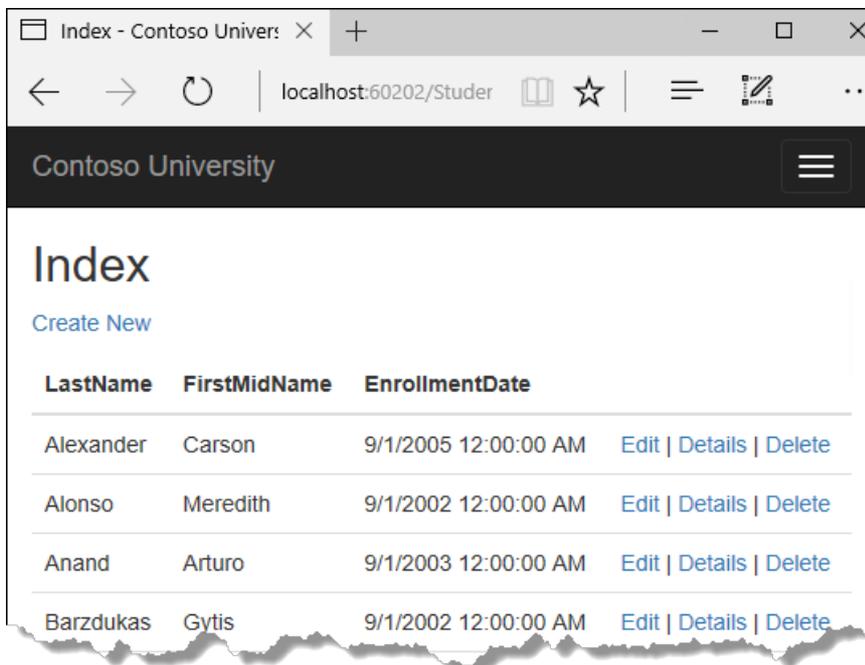
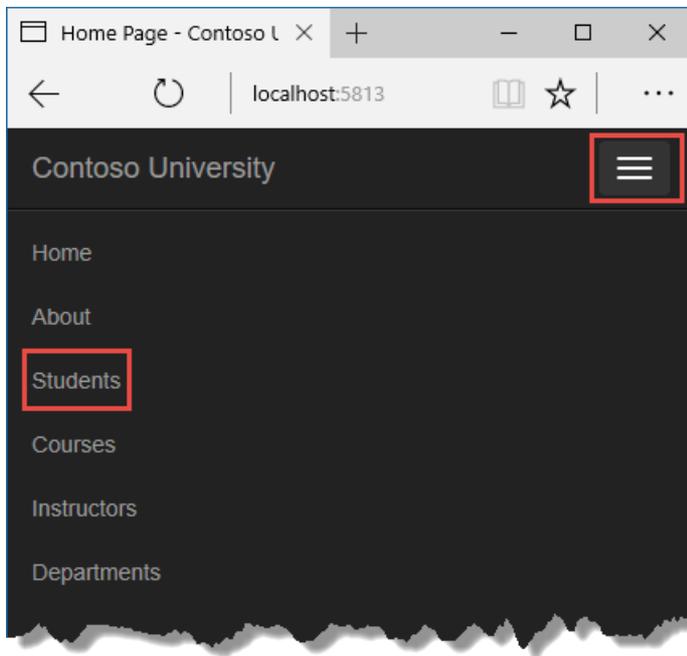
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.EnrollmentDate)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu.

Click the Students tab to see the test data that the `DbInitializer.Initialize` method inserted. Depending on how narrow your browser window is, you'll see the `student` tab link at the top of the page or you'll have to click the navigation icon in the upper right corner to see the link.



View the Database

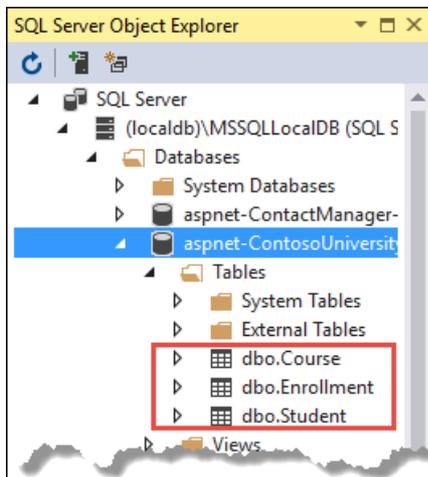
When you started the application, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database and so it created one, then the remainder of the `Initialize` method code populated the database with data. You can use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio.

Close the browser.

If the SSOX window isn't already open, select it from the **View** menu in Visual Studio.

In SSOX, click **(localdb)\MSSQLLocalDB > Databases**, and then click the entry for the database name that is in the connection string in your `appsettings.json` file.

Expand the **Tables** node to see the tables in your database.



Right-click the **Student** table and click **View Data** to see the columns that were created and the rows that were inserted into the table.

ID	EnrollmentDate	FirstMidName	LastName
1	9/1/2005 12:00:...	Carson	Alexander
2	9/1/2002 12:00:...	Meredith	Alonso
3	9/1/2003 12:00:...	Arturo	Anand
4	9/1/2002 12:00:...	Gytis	Barzdukas
5	9/1/2002 12:00:...	Yan	Li

The *.mdf* and *.ldf* database files are in the `C:\Users<yourusername>` folder.

Because you're calling `EnsureCreated` in the initializer method that runs on app start, you could now make a change to the `Student` class, delete the database, run the application again, and the database would automatically be re-created to match your change. For example, if you add an `EmailAddress` property to the `Student` class, you'll see a new `EmailAddress` column in the re-created table.

Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of conventions, or assumptions that the Entity Framework makes.

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or `classNameID` are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

Conventional behavior can be overridden. For example, you can explicitly specify table names, as you saw earlier in this tutorial. And you can set column names and set any property as primary key or foreign key, as you'll see in a [later tutorial](#) in this series.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the `async` keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that is returned.
- The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when you are writing asynchronous code that uses the Entity Framework:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes, for example, `ToListAsync`, `SingleOrDefaultAsync`, and `SaveChangesAsync`. It does not include, for example, statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio");`
- An EF context is not thread safe: don't try to do multiple operations in parallel. When you call any async EF method, always use the `await` keyword.
- If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

For more information about asynchronous programming in .NET, see [Async Overview](#).

Summary

You've now created a simple application that uses the Entity Framework Core and SQL Server Express LocalDB to store and display data. In the following tutorial, you'll learn how to perform basic CRUD (create, read, update, delete) operations.

Create, Read, Update, and Delete - EF Core with ASP.NET Core MVC tutorial (2 of 10)

9/22/2017 • 19 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

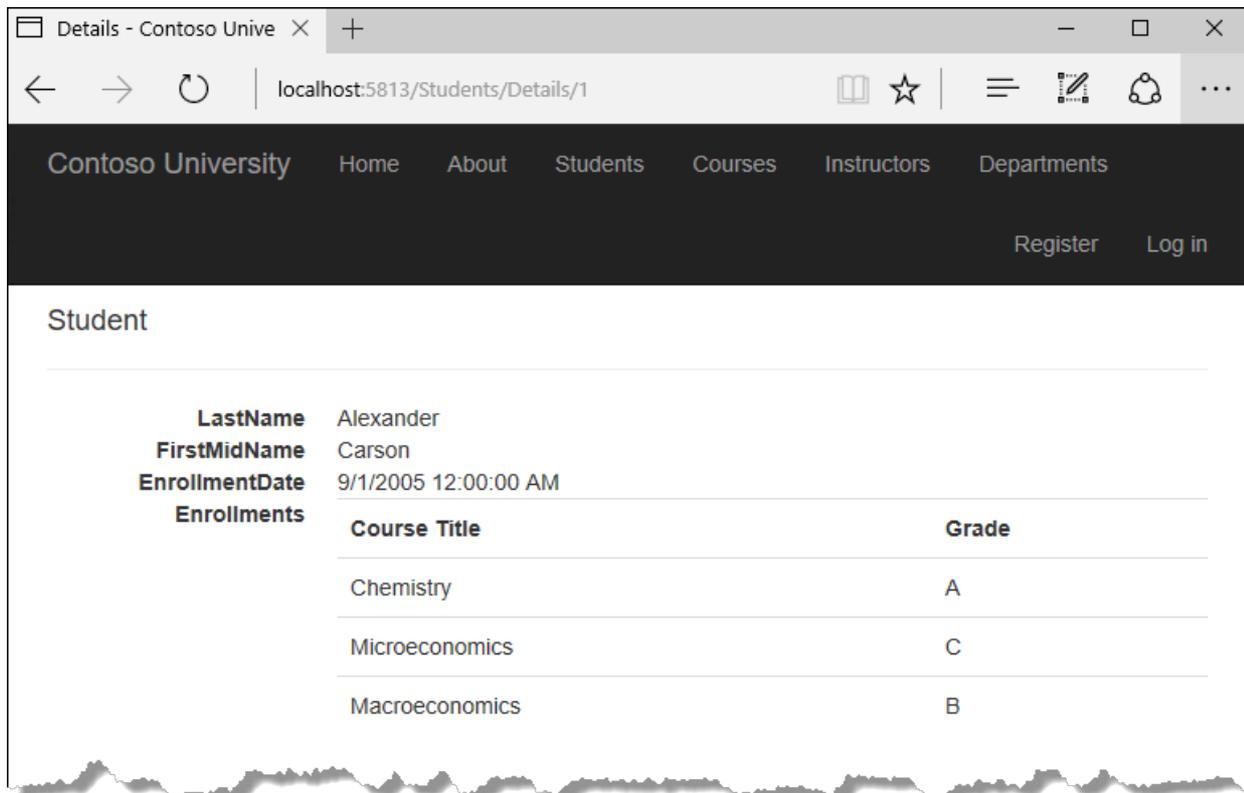
The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial, you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial, you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

NOTE

It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about repositories with EF, see [the last tutorial in this series](#).

In this tutorial, you'll work with the following web pages:



The screenshot shows a web browser window with the address bar displaying 'localhost:5813/Students/Details/1'. The page title is 'Details - Contoso Unive'. The navigation menu includes 'Home', 'About', 'Students', 'Courses', 'Instructors', and 'Departments'. There are also 'Register' and 'Log in' links. The main content area is titled 'Student' and displays the following information:

LastName	Alexander
FirstMidName	Carson
EnrollmentDate	9/1/2005 12:00:00 AM

Enrollments	Course Title	Grade
	Chemistry	A
	Microeconomics	C
	Macroeconomics	B

Create x + - □ ×

← ↻ | s/Create | ☆ | ...

Contoso University ☰

Create

Student

LastName

FirstMidName

EnrollmentDate

Create

Edit - C x + - □ ×

← ↻ | localhost: | ☆ | ...

Contoso University ☰

Edit

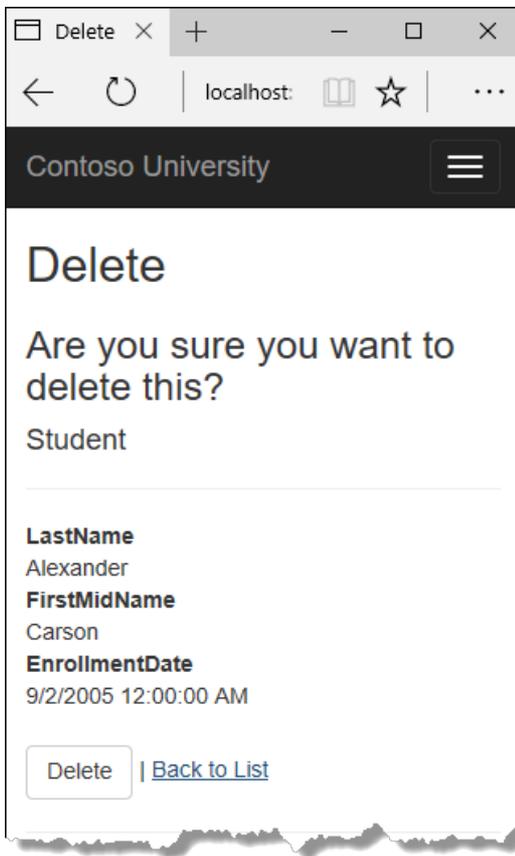
Student

LastName

FirstMidName

EnrollmentDate

Save



Customize the Details page

The scaffolded code for the Students Index page left out the `Enrollments` property, because that property holds a collection. In the **Details** page, you'll display the contents of the collection in an HTML table.

In `Controllers/StudentsController.cs`, the action method for the Details view uses the `SingleOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls `Include`, `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. You'll learn more about these methods in the [reading related data](#) tutorial.

The `AsNoTracking` method improves performance in scenarios where the entities returned will not be updated in the current context's lifetime. You'll learn more about `AsNoTracking` at the end of this tutorial.

Route data

The key value that is passed to the `Details` method comes from *route data*. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the `Details` method `id` parameter if you pass it as a query string value:

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the `id` parameter matches the default route, so `id` is added to the route data.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, `studentID` doesn't match a parameter in the default route, so it's added as a query string.

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

For more information about tag helpers, see [Tag helpers in ASP.NET Core](#).

Add enrollments to the Details view

Open `Views/Students/Details.cshtml`. Each field is displayed using `DisplayNameFor` and `DisplayFor` helpers, as shown in the following example:

```
<dt>
  @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd>
  @Html.DisplayFor(model => model.LastName)
</dd>
```

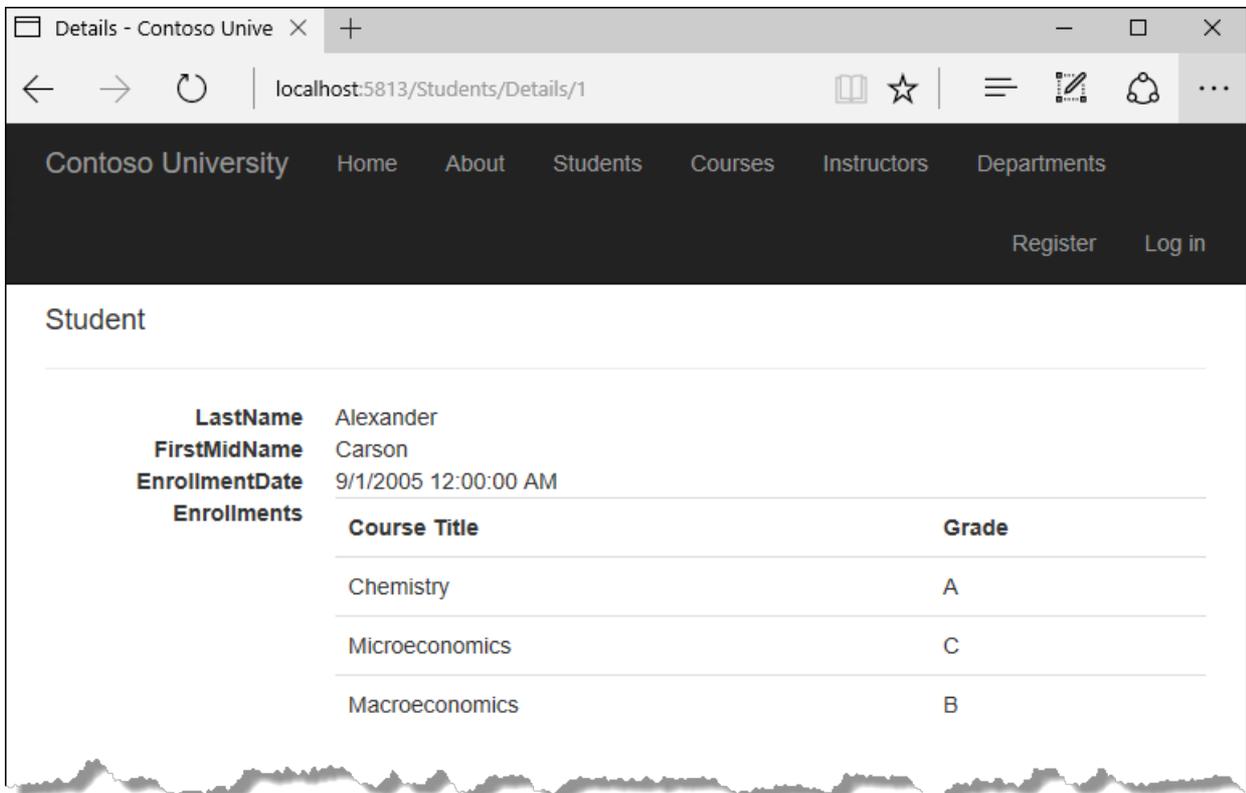
After the last field and immediately before the closing `</dl>` tag, add the following code to display a list of enrollments:

```
<dt>
  @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd>
  <table class="table">
    <tr>
      <th>Course Title</th>
      <th>Grade</th>
    </tr>
    @foreach (var item in Model.Enrollments)
    {
      <tr>
        <td>
          @Html.DisplayFor(modelItem => item.Course.Title)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Grade)
        </td>
      </tr>
    }
  </table>
</dd>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. You see the list of courses and grades for the selected student:



Update the Create page

In *StudentsController.cs*, modify the `HttpPost Create` method by adding a try-catch block and removing ID from the `Bind` attribute.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

This code adds the Student entity created by the ASP.NET MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed `ID` from the `Bind` attribute because ID is the primary key value which SQL Server will set

automatically when the row is inserted. Input from the user does not set the ID value.

Other than the `Bind` attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from `DbUpdateException` is caught while the changes are being saved, a generic error message is displayed. `DbUpdateException` exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in [Monitoring and Telemetry \(Building Real-World Cloud Apps with Azure\)](#).

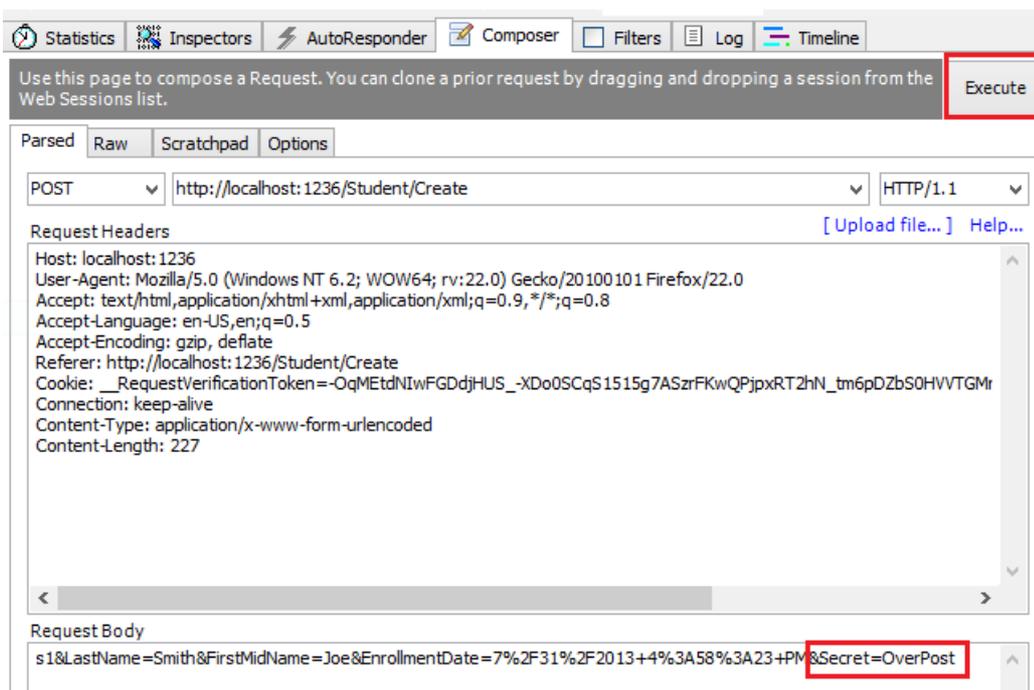
The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the `FormTagHelper` and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information about CSRF, see [Anti-Request Forgery](#).

Security note about overposting

The `Bind` attribute that the scaffolded code includes on the `Create` method is one way to protect against overposting in create scenarios. For example, suppose the Student entity includes a `Secret` property that you don't want this web page to set.

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a Student instance, the model binder would pick up that `Secret` form value and use it to create the Student entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you

never intended that the web page be able to set that property.

You can prevent overposting in edit scenarios by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That is the method used in these tutorials.

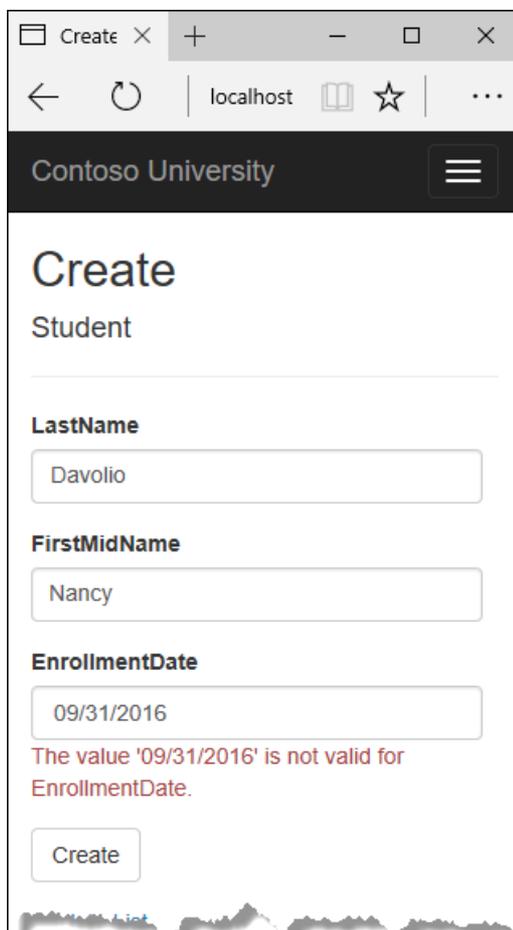
An alternative way to prevent overposting that is preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use `_context.Entry` on the entity instance to set its state to `Unchanged`, and then set `Property("PropertyName").IsModified` to true on each entity property that is included in the view model. This method works in both edit and create scenarios.

Test the Create page

The code in `Views/Students/Create.cshtml` uses `label`, `input`, and `span` (for validation messages) tag helpers for each field.

Run the app, select the **Students** tab, and click **Create New**.

Enter names and a date. Try entering an invalid date if your browser lets you do that. (Some browsers force you to use a date picker.) Then click **Create** to see the error message.



This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the `Create` method.

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}

```

Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

Update the Edit page

In *StudentController.cs*, the `HttpGet Edit` method (the one without the `HttpPost` attribute) uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the `Details` method. You don't need to change this method.

Recommended HttpPost Edit code: Read and update

Replace the `HttpPost Edit` action method with the following code.

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.SingleOrDefaultAsync(s => s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}

```

These changes implement a security best practice to prevent overposting. The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a `Modified` flag. That code is not recommended for many scenarios because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter.

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity [based on user input in the posted form data](#). The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

As a best practice to prevent overposting, the fields that you want to be updateable by the **Edit** page are whitelisted in the `TryUpdateModel` parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the `HttpPost Edit` method is the same as the `HttpGet Edit` method; therefore you've renamed the method `EditPost`.

Alternative HttpPost Edit code: Create and attach

The recommended `HttpPost` edit code ensures that only changed columns get updated and preserves data in properties that you don't want included for model binding. However, the read-first approach requires an extra database read, and can result in more complex code for handling concurrency conflicts. An alternative is to attach an entity created by the model binder to the EF context and mark it as modified. (Don't update your project with this code, it's only shown to illustrate an optional approach.)

```

public async Task<IActionResult> Edit(int id, [Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student
student)
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}

```

You can use this approach when the web page UI includes all of the fields in the entity and can update any of them.

The scaffolded code uses the create-and-attach approach but only catches `DbUpdateConcurrencyException` exceptions and returns 404 error codes. The example shown catches any database update exception and displays an error message.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the `Add` method, that entity's state is set to `Added`. Then when you call the `SaveChanges` method, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`. The entity does not yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.
- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. You read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the `HttpPost Edit` action method is called, a new web request is made and you have a new

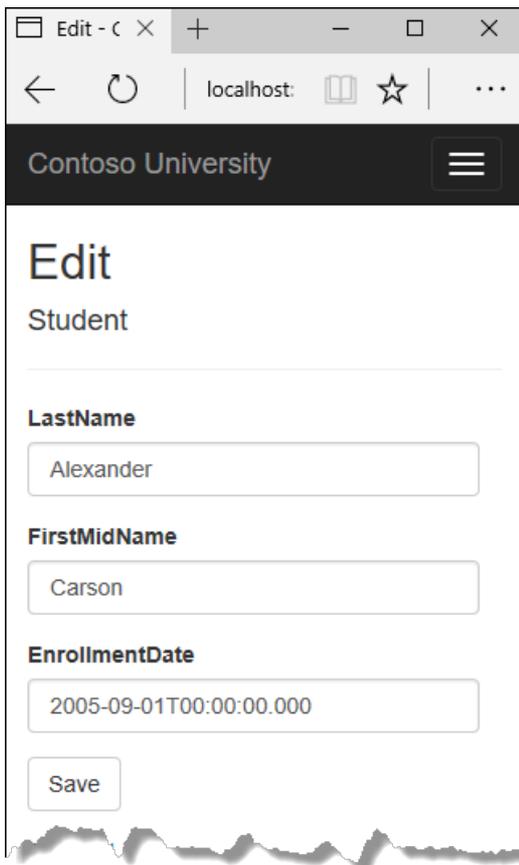
instance of the `DbContext`. If you re-read the entity in that new context, you simulate desktop processing.

But if you don't want to do the extra read operation, you have to use the entity object created by the model binder. The simplest way to do this is to set the entity state to Modified as is done in the alternative `HttpPost Edit` code shown earlier. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want to avoid the read-first approach, but you also want the SQL UPDATE statement to update only the fields that the user actually changed, the code is more complex. You have to save the original values in some way (such as by using hidden fields) so that they are available when the `HttpPost Edit` method is called. Then you can create a Student entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`.

Test the Edit page

Run the app, select the **Students** tab, then click an **Edit** hyperlink.



Change some of the data and click **Save**. The **Index** page opens and you see the changed data.

Update the Delete page

In `StudentController.cs`, the template code for the `HttpGet Delete` method uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the `HttpPost Delete` method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the `HttpPost Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the `HttpPost Delete` method calls the `HttpGet Delete` method, passing it a

parameter that indicates that an error has occurred. The `HttpGet Delete` method then redisplay the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

Replace the `HttpGet Delete` action method with the following code, which manages error reporting.

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the `HttpGet Delete` method is called without a previous failure. When it is called by the `HttpPost Delete` method in response to a database update error, the parameter is true and an error message is passed to the view.

The read-first approach to `HttpPost Delete`

Replace the `HttpPost Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

```

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}

```

This code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated.

The create-and-attach approach to HttpPost Delete

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query by instantiating a Student entity using only the primary key value and then setting the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity. (Don't put this code in your project; it's here just to illustrate an alternative.)

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}

```

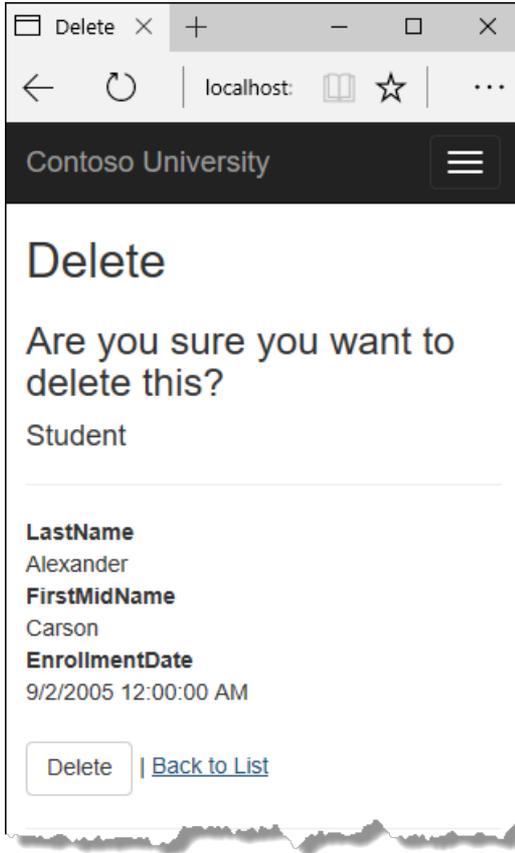
If the entity has related data that should also be deleted, make sure that cascade delete is configured in the database. With this approach to entity deletion, EF might not realize there are related entities to be deleted.

Update the Delete view

In `Views/Student/Delete.cshtml`, add an error message between the h2 heading and the h3 heading, as shown in the following example:

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the app, select the **Students** tab, and click a **Delete** hyperlink:



Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Closing database connections

To free up the resources that a database connection holds, the context instance must be disposed as soon as possible when you are done with it. The ASP.NET Core built-in [dependency injection](#) takes care of that task for you.

In *Startup.cs*, you call the [AddDbContext extension method](#) to provision the `DbContext` class in the ASP.NET DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handling Transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or they all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see [Transactions](#).

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps

track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by calling the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

- During the context lifetime you don't need to update any entities, and you don't need EF to [automatically load navigation properties with entities retrieved by separate queries](#). Frequently these conditions are met in a controller's `HttpGet` action methods.
- You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.
- You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see [Tracking vs. No-Tracking](#).

Summary

You now have a complete set of pages that perform simple CRUD operations for Student entities. In the next tutorial you'll expand the functionality of the **Index** page by adding sorting, filtering, and paging.

[PREVIOUS](#)[NEXT](#)

Sorting, filtering, paging, and grouping - EF Core with ASP.NET Core MVC tutorial (3 of 10)

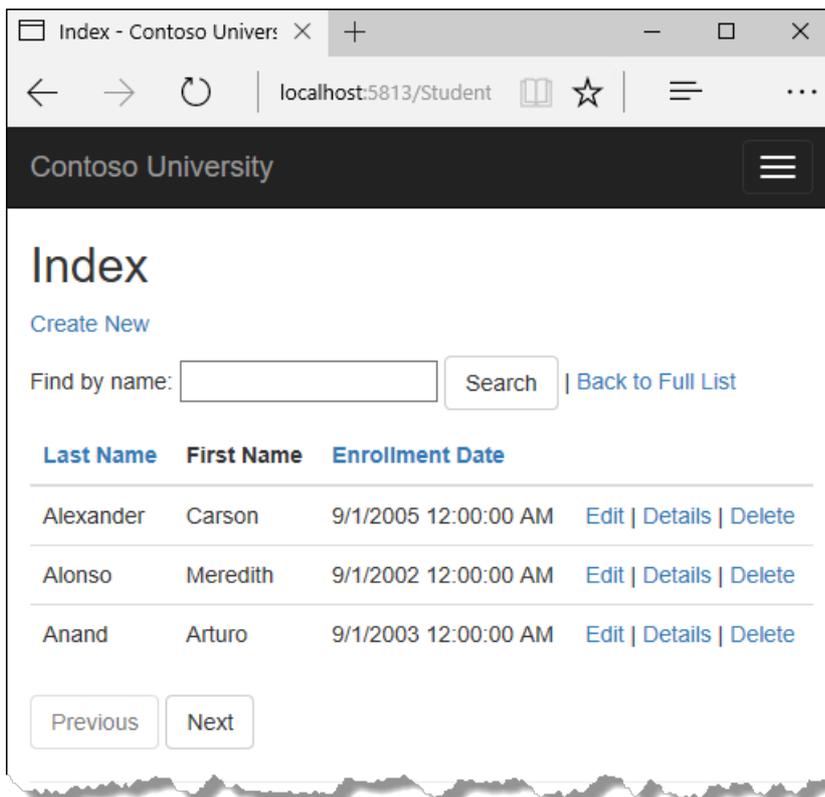
9/22/2017 • 14 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for Student entities. In this tutorial you'll add sorting, filtering, and paging functionality to the Students Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



Add Column Sort Links to the Students Index Page

To add sorting to the Student Index page, you'll change the `Index` method of the Students controller and add code to the Student Index view.

Add sorting Functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code:

```

public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}

```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewData` elements (NameSortParm and DateSortParm) are used by the view to configure the column heading hyperlinks with the appropriate query string values.

```

public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}

```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, NameSortParm should be set to "name_desc"; otherwise, it should be set to an empty string. These two

statements enable the view to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an `IQueryable` variable before the switch statement, modifies it in the switch statement, and calls the `ToListAsync` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query is not executed until you convert the `IQueryable` object into a collection by calling a method such as `ToListAsync`. Therefore, this code results in a single query that is not executed until the `return View` statement.

This code could get verbose with a large number of columns. [The last tutorial in this series](#) shows how to write code that lets you pass the name of the `OrderBy` column in a string variable.

Add column heading hyperlinks to the Student Index view

Replace the code in `Views/Students/Index.cshtml`, with the following code to add column heading hyperlinks. The changed lines are highlighted.

```

@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

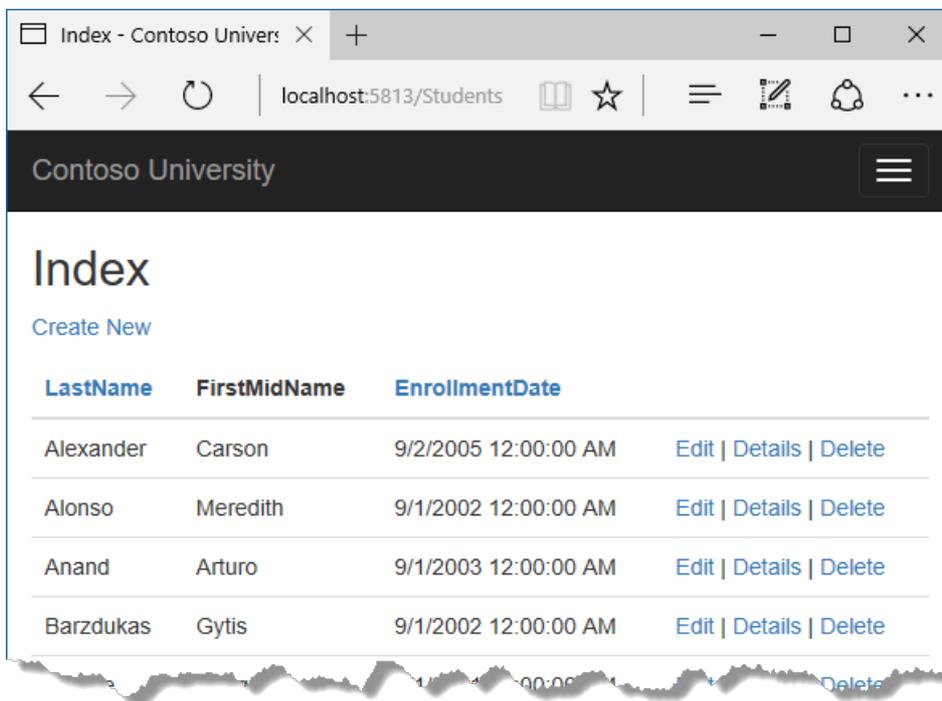
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

This code uses the information in `ViewData` properties to set up hyperlinks with the appropriate query string values.

Run the app, select the **Students** tab, and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Add a Search Box to the Students Index page

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

In `StudentsController.cs`, replace the `Index` method with the following code (the changes are highlighted).

```
public async Task<IActionResult> Index(string sortOrder, string searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

You've added a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

NOTE

Here you are calling the `Where` method on an `IQueryable` object, and the filter will be processed on the server. In some scenarios you might be calling the `Where` method as an extension method on an in-memory collection. (For example, suppose you change the reference to `_context.Students` so that instead of an EF `DbSet` it references a repository method that returns an `IEnumerable` collection.) The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: `Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper()))`. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there is a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

In `Views/Student/Index.cshtml`, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

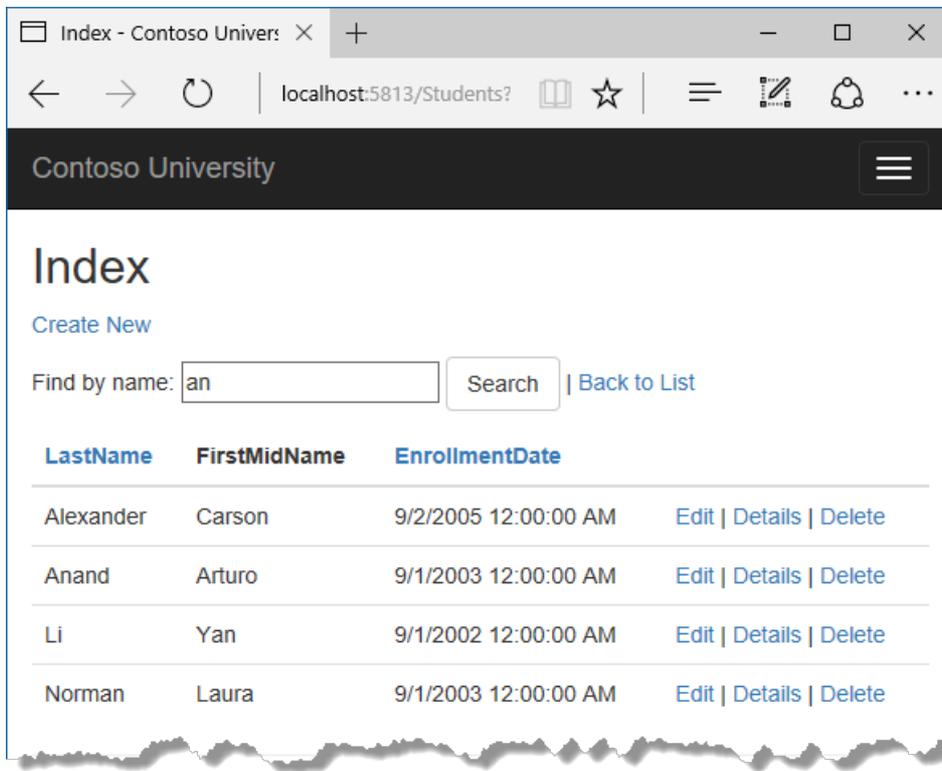
```
<p>
  <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
  <div class="form-actions no-color">
    <p>
      Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
      <input type="submit" value="Search" class="btn btn-default" /> |
      <a asp-action="Index">Back to Full List</a>
    </p>
  </div>
</form>

<table class="table">
```

This code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action does not result in an update.

Run the app, select the **Students** tab, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

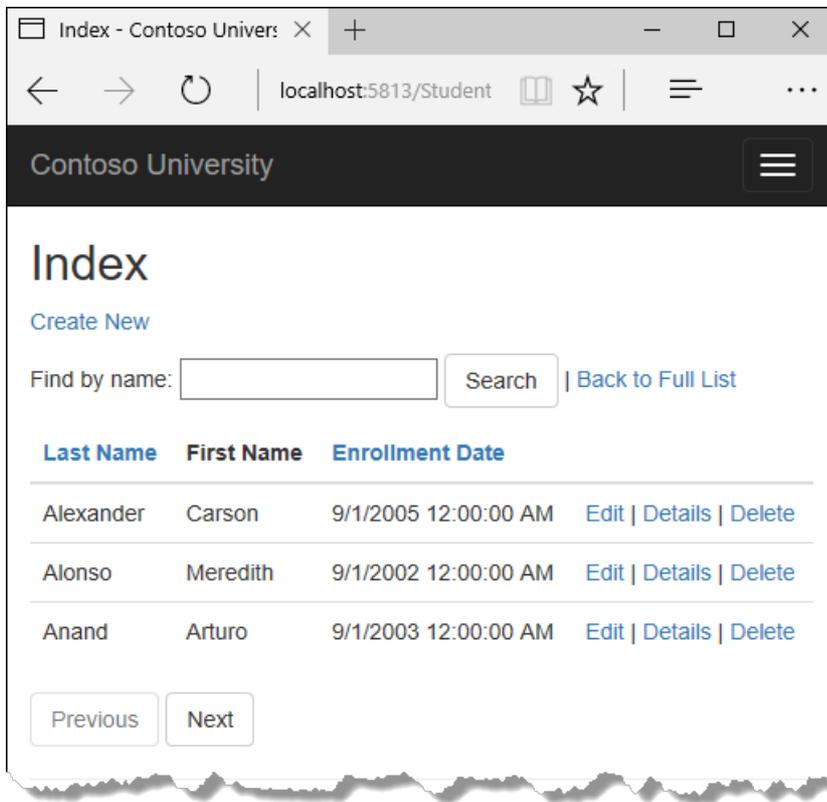
```
http://localhost:5813/Students?SearchString=an
```

If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding `method="get"` to the `form` tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the **Search** box. You'll fix that in the next section.

Add paging functionality to the Students Index page

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs`, and then replace the template code with the following code.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int
        pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}

```

The `CreateAsync` method in this code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it will return a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` can be used to enable or disable **Previous** and **Next** paging buttons.

A `CreateAsync` method is used instead of a constructor to create the `PaginatedList<T>` object because constructors can't run asynchronous code.

Add paging functionality to the Index method

In `StudentsController.cs`, replace the `Index` method with the following code.

```

public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
}

```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

```

public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)

```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The `ViewData` element named CurrentSort provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The `ViewData` element named `CurrentFilter` provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the `searchString` parameter is not null.

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the `Index` method, the `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
```

The `PaginatedList.CreateAsync` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(page ?? 1)` means return the value of `page` if it has a value, or return 1 if `page` is null.

Add paging links to the Student Index view

In `Views/Students/Index.cshtml`, replace the existing code with the following code. The changes are highlighted.

```
@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
        </tr>
    </thead>
</table>
```

```

        </th>
        <th>
            <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
        </th>
        <th></th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex - 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex + 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @nextDisabled">
    Next
</a>

```

The `@model` statement at the top of the page specifies that the view now gets a `PagedList<T>` object instead of a `List<T>` object.

The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```

<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter
="@ViewData["CurrentFilter"]">Enrollment Date</a>

```

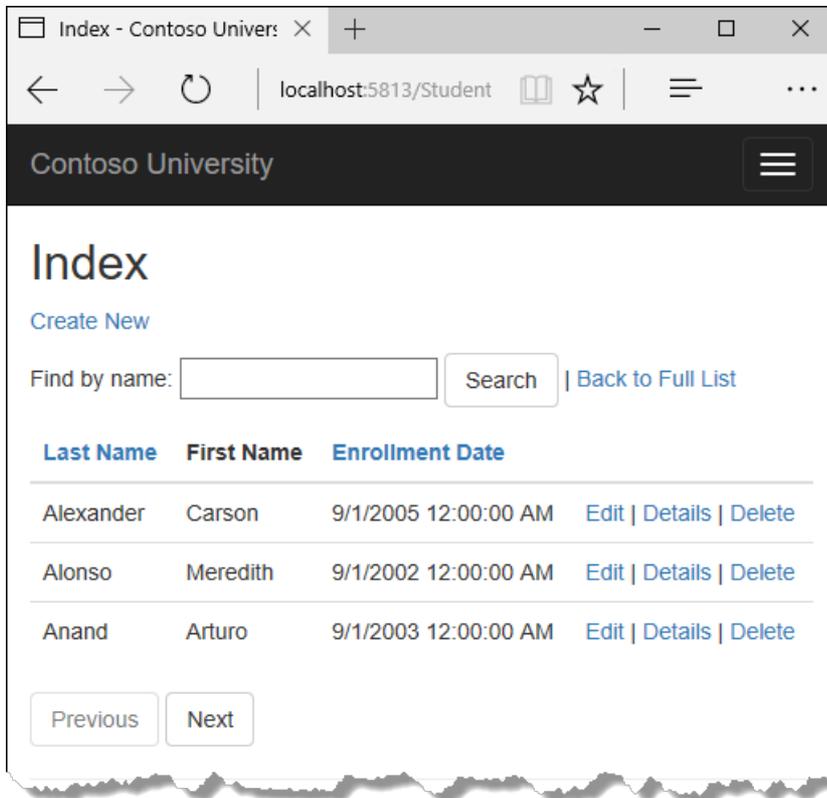
The paging buttons are displayed by tag helpers:

```

<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex - 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled">
    Previous
</a>

```

Run the app and go to the Students page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

Create an About page that shows Student statistics

For the Contoso University website's **About** page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Modify the About method in the Home controller.
- Modify the About view.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the new folder, add a class file *EnrollmentDateGroup.cs* and replace the template code with the following code:

```

using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}

```

Modify the Home Controller

In *HomeController.cs*, add the following using statements at the top of the file:

```

using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;

```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

```

public class HomeController : Controller
{
    private readonly SchoolContext _context;

    public HomeController(SchoolContext context)
    {
        _context = context;
    }
}

```

Replace the `About` method with the following code:

```

public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}

```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

NOTE

In the 1.0 version of Entity Framework Core, the entire result set is returned to the client, and grouping is done on the client. In some scenarios this could create performance problems. Be sure to test performance with production volumes of data, and if necessary use raw SQL to do the grouping on the server. For information about how to use raw SQL, see [the last tutorial in this series](#).

Modify the About View

Replace the code in the `Views/Home/About.cshtml` file with the following code:

```
@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

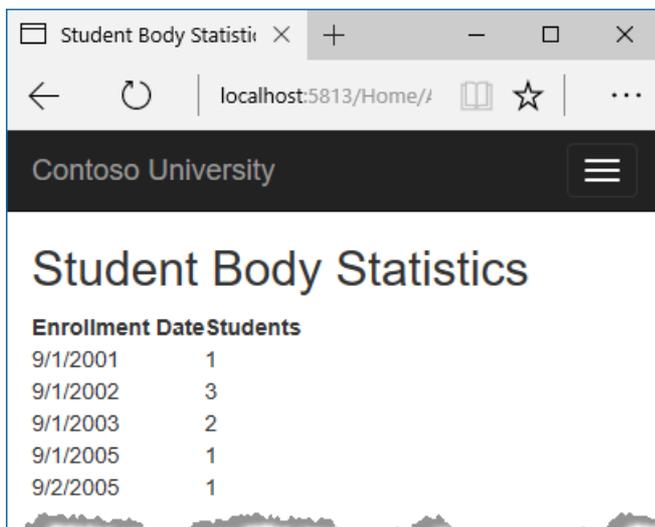
@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and go to the About page. The count of students for each enrollment date is displayed in a table.



Summary

In this tutorial, you've seen how to perform sorting, filtering, paging, and grouping. In the next tutorial, you'll learn how to handle data model changes by using migrations.

[PREVIOUS](#)[NEXT](#)

Migrations - EF Core with ASP.NET Core MVC tutorial (4 of 10)

11/7/2017 • 8 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In this tutorial, you start using the EF Core migrations feature for managing data model changes. In later tutorials, you'll add more migrations as you change the data model.

Introduction to migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You started these tutorials by configuring the Entity Framework to create the database if it doesn't exist. Then each time you change the data model -- add, remove, or change entity classes or change your DbContext class -- you can delete the database and EF creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The EF Core Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.

Entity Framework Core NuGet packages for migrations

To work with migrations, you can use the **Package Manager Console** (PMC) or the command-line interface (CLI). These tutorials show how to use CLI commands. Information about the PMC is at [the end of this tutorial](#).

The EF tools for the command-line interface (CLI) are provided in [Microsoft.EntityFrameworkCore.Tools.DotNet](#). To install this package, add it to the `DotNetCliToolReference` collection in the `.csproj` file, as shown. **Note:** You have to install this package by editing the `.csproj` file; you can't use the `install-package` command or the package manager GUI. You can edit the `.csproj` file by right-clicking the project name in **Solution Explorer** and selecting **Edit ContosoUniversity.csproj**.

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

(The version numbers in this example were current when the tutorial was written.)

Change the connection string

In the `appsettings.json` file, change the name of the database in the connection string to ContosoUniversity2 or some other name that you haven't used on the computer you're using.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
}
```

This change sets up the project so that the first migration will create a new database. This isn't required for getting started with migrations, but you'll see later why it's a good idea.

NOTE

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

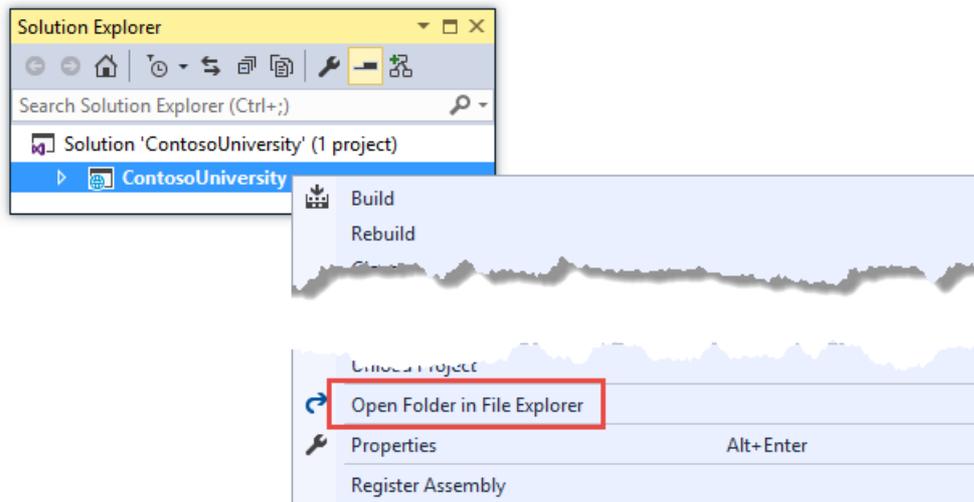
```
dotnet ef database drop
```

The following section explains how to run CLI commands.

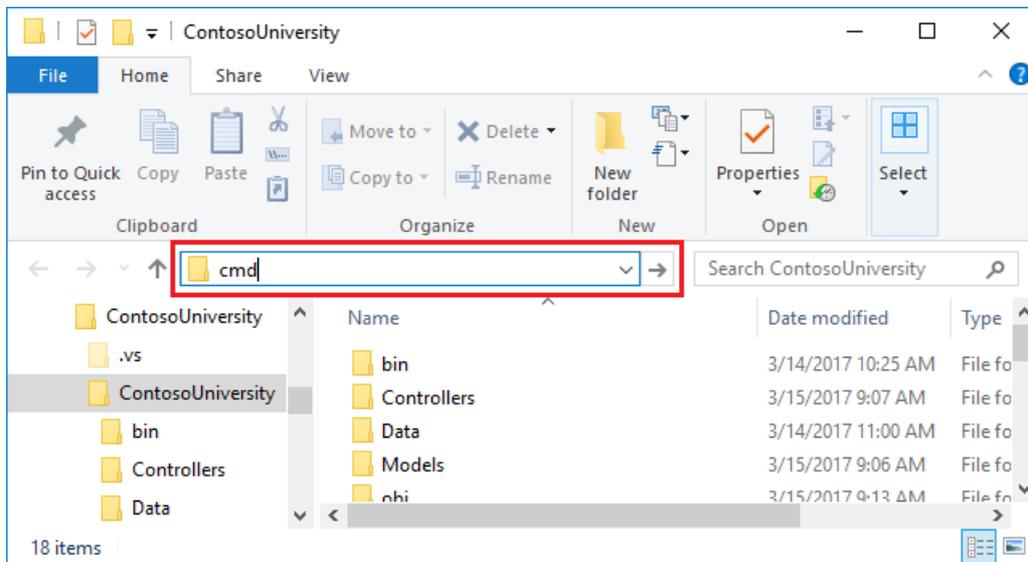
Create an initial migration

Save your changes and build the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

- In **Solution Explorer**, right-click the project and choose **Open in File Explorer** from the context menu.



- Enter "cmd" in the address bar and press Enter.



Enter the following command in the command window:

```
dotnet ef migrations add InitialCreate
```

You see output like the following in the command window:

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
      repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
      'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

NOTE

If you see an error message *No executable found matching command "dotnet-ef"*, see [this blog post](#) for help troubleshooting.

If you see an error message *"cannot access the file ... ContosoUniversity.dll because it is being used by another process."*, find the IIS Express icon in the Windows System Tray, and right-click it, then click **ContosoUniversity > Stop Site**.

Examine the Up and Down methods

When you executed the `migrations add` command, EF generated the code that will create the database from scratch. This code is in the `Migrations` folder, in the file named `<timestamp>_InitialCreate.cs`. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

```

public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");
        // Additional code not shown
    }
}

```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the `migrations add InitialCreate` command. The migration name parameter ("InitialCreate" in the example) is used for the file name and can be whatever you want. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you changed the name of the database in the connection string earlier -- so that migrations can create a new one from scratch.

Examine the data model snapshot

Migrations also creates a *snapshot* of the current database schema in `Migrations/SchoolContextModelSnapshot.cs`. Here's what that code looks like:

```

[DbContext(typeof(SchoolContext))]
partial class SchoolContextModelSnapshot : ModelSnapshot
{
    protected override void BuildModel(ModelBuilder modelBuilder)
    {
        modelBuilder
            .HasAnnotation("ProductVersion", "2.0.0-rtm-26452")
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
                SqlServerValueGenerationStrategy.IdentityColumn);

        modelBuilder.Entity("ContosoUniversity.Models.Course", b =>
        {
            b.Property<int>("CourseID");

            b.Property<int>("Credits");

            b.Property<string>("Title");

            b.HasKey("CourseID");

            b.ToTable("Course");
        });

        // Additional code for Enrollment and Student tables not shown

        modelBuilder.Entity("ContosoUniversity.Models.Enrollment", b =>
        {
            b.HasOne("ContosoUniversity.Models.Course", "Course")
                .WithMany("Enrollments")
                .HasForeignKey("CourseID")
                .OnDelete(DeleteBehavior.Cascade);

            b.HasOne("ContosoUniversity.Models.Student", "Student")
                .WithMany("Enrollments")
                .HasForeignKey("StudentID")
                .OnDelete(DeleteBehavior.Cascade);
        });
    }
}

```

Because the current database schema is represented in code, EF Core doesn't have to interact with the database to create migrations. When you add a migration, EF determines what changed by comparing the data model to the snapshot file. EF interacts with the database only when it has to update the database.

The snapshot file has to be kept in sync with the migrations that create it, so you can't remove a migration just by deleting the file named `<timestamp>_<migrationname>.cs`. If you delete that file, the remaining migrations will be out of sync with the database snapshot file. To delete the last migration that you added, use the [dotnet ef migrations remove](#) command.

Apply the migration to the database

In the command window, enter the following command to create the database and tables in it.

```
dotnet ef database update
```

The output from the command is similar to the `migrations add` command, except that you see logs for the SQL commands that set up the database. Most of the logs are omitted in the following sample output. If you prefer not to see this level of detail in log messages, you can change the log level in the `appsettings.Development.json` file. For more information, see [Introduction to logging](#).

```

info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (467ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
CREATE TABLE [__EFMigrationsHistory] (
  [MigrationId] nvarchar(150) NOT NULL,
  [ProductVersion] nvarchar(32) NOT NULL,
  CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
);

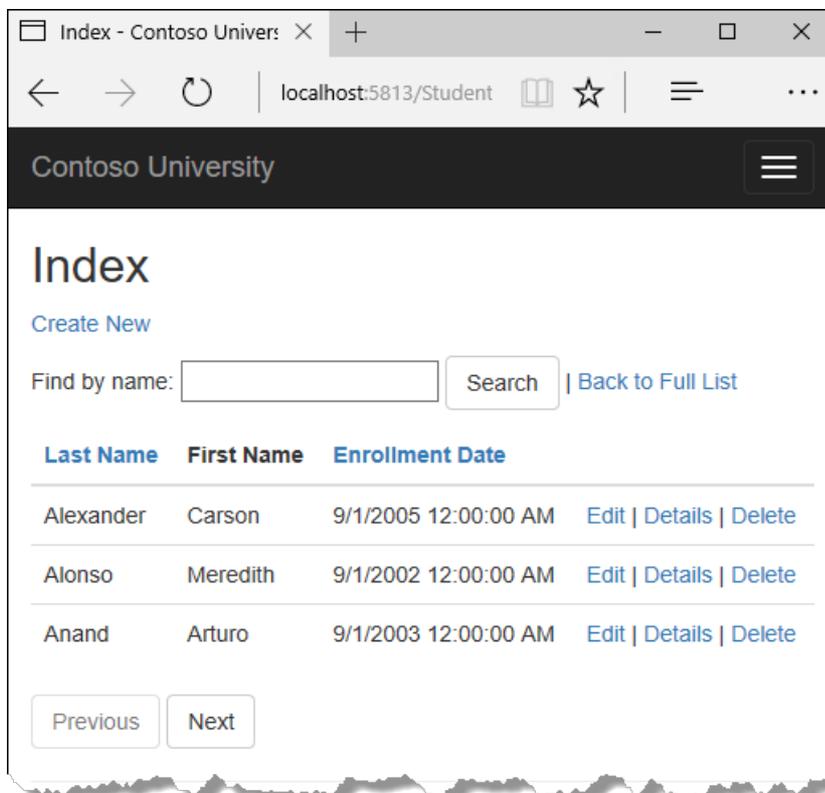
<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20170816151242_InitialCreate', N'2.0.0-rtm-26452');
Done.

```

Use **SQL Server Object Explorer** to inspect the database as you did in the first tutorial. You'll notice the addition of an `__EFMigrationsHistory` table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one row for the first migration. (The last log in the preceding CLI output example shows the INSERT statement that creates this row.)

Run the application to verify that everything still works the same as before.



Command-line interface (CLI) vs. Package Manager Console (PMC)

The EF tooling for managing migrations is available from .NET Core CLI commands or from PowerShell cmdlets in the Visual Studio **Package Manager Console** (PMC) window. This tutorial shows how to use the CLI, but you

can use the PMC if you prefer.

The EF commands for the PMC commands are in the [Microsoft.EntityFrameworkCore.Tools](#) package. This package is already included in the [Microsoft.AspNetCore.All](#) metapackage, so you don't have to install it.

Important: This is not the same package as the one you install for the CLI by editing the `.csproj` file. The name of this one ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see [.NET Core CLI](#).

For more information about the PMC commands, see [Package Manager Console \(Visual Studio\)](#).

Summary

In this tutorial, you've seen how to create and apply your first migration. In the next tutorial, you'll begin looking at more advanced topics by expanding the data model. Along the way you'll create and apply additional migrations.

[PREVIOUS](#)[NEXT](#)

Creating a complex data model - EF Core with ASP.NET Core MVC tutorial (5 of 10)

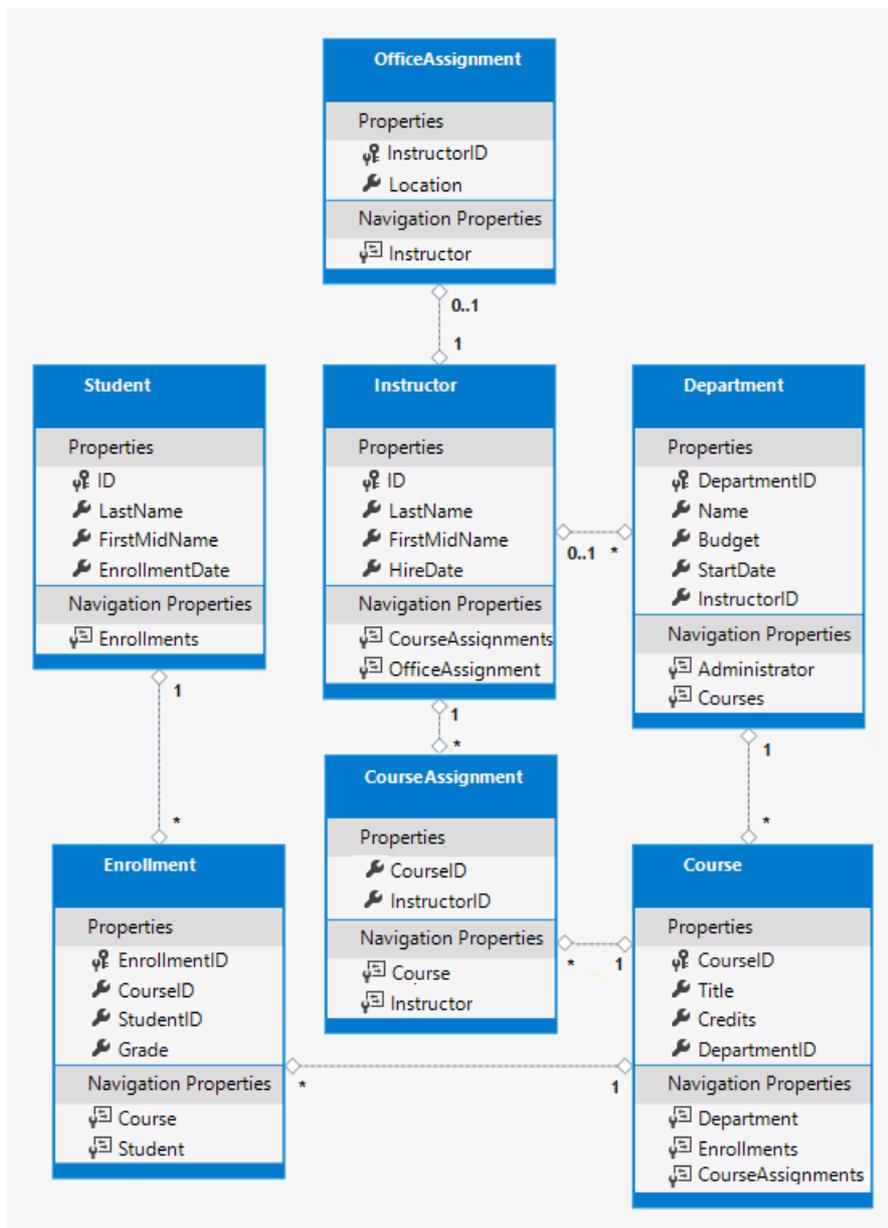
9/22/2017 • 30 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorials, you worked with a simple data model that was composed of three entities. In this tutorial, you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



Customize the Data Model by Using Attributes

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete School data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The `DataType` attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In `Models/Student.cs`, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `DataType` attribute is used to specify a data type that is more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The `DataType` Enumeration provides for many data types, such as `Date`, `Time`, `PhoneNumber`, `Currency`, `EmailAddress`, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

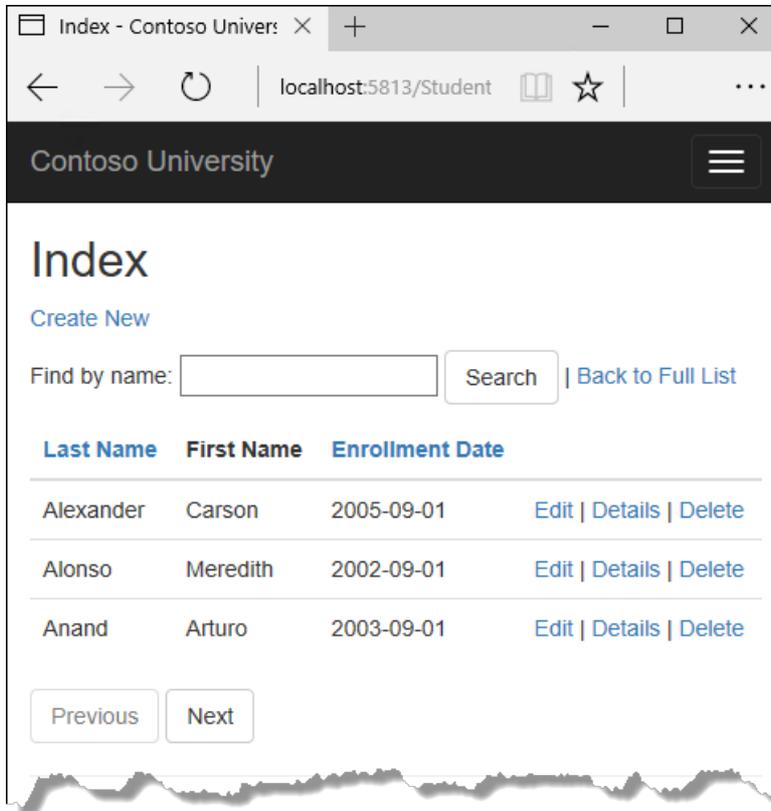
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).
- By default, the browser will render data using the correct format based on your locale.

For more information, see the [<input> tag helper documentation](#).

Run the app, go to the Students Index page and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the Student model.



The `StringLength` attribute

You can also specify data validation rules and validation error messages using attributes. The `StringLength` attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `StringLength` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

The `StringLength` attribute won't prevent a user from entering white space for a name. You can use the `RegularExpression` attribute to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z'-'\s]*$")]
```

The `MaxLength` attribute provides functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

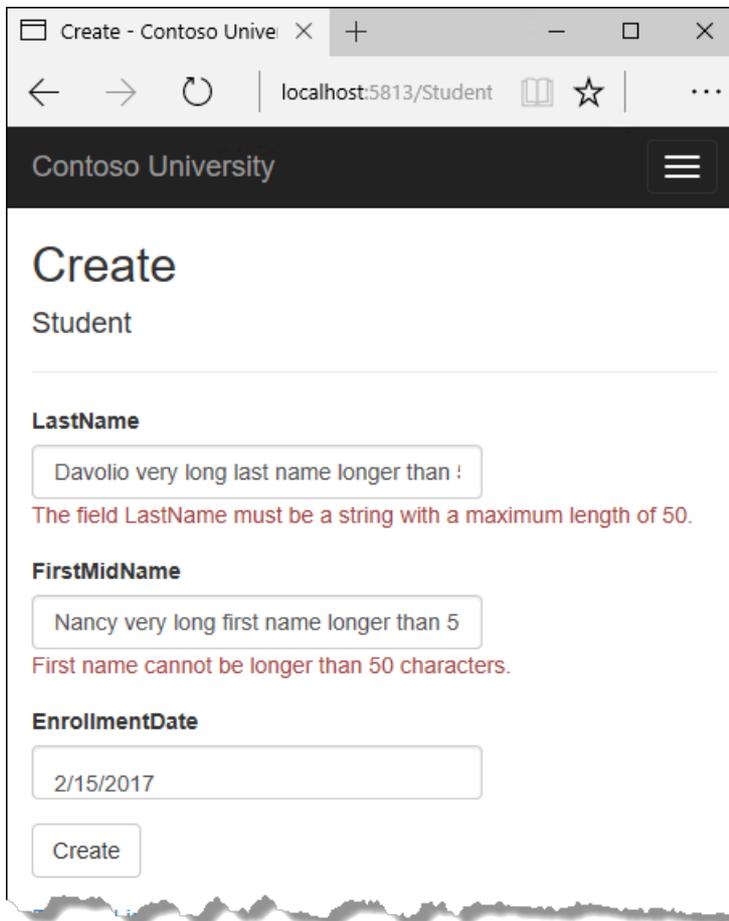
```
dotnet ef migrations add MaxLengthOnNames
```

```
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations creates a file named `<timestamp>_MaxLengthOnNames.cs`. This file contains code in the `Up` method that will update the database to match the current data model. The `database update` command ran that code.

The timestamp prefixed to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the update-database command, and then all of the migrations are applied in the order in which they were created.

Run the app, select the **Students** tab, click **Create New**, and enter either name longer than 50 characters. When you click **Create**, client side validation shows an error message.



The Column attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they are given the same name as the property name.

In the `Student.cs` file, add a `using` statement for `System.ComponentModel.DataAnnotations.Schema` and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

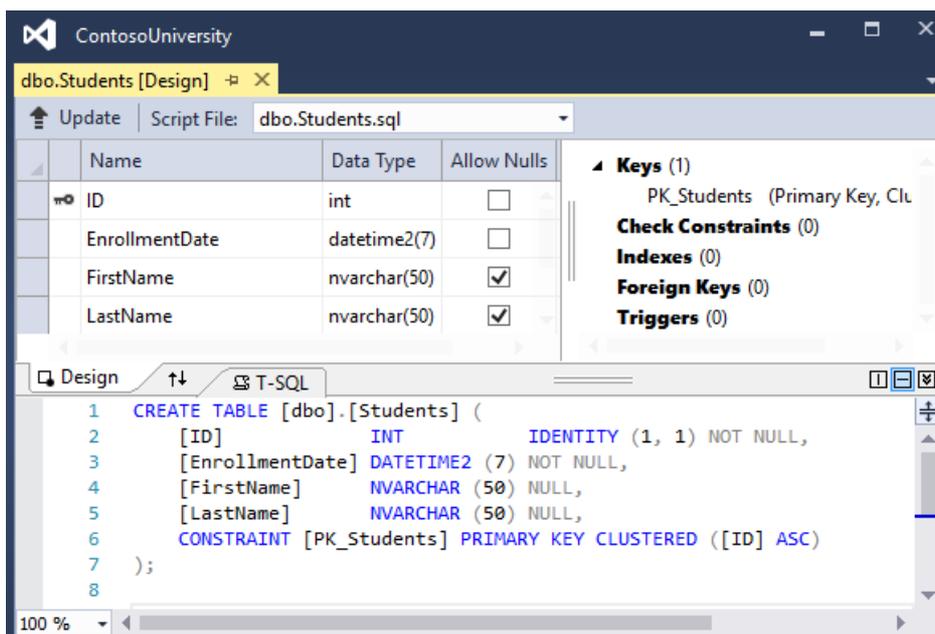
The addition of the `column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

```
dotnet ef migrations add ColumnFirstName
```

```
dotnet ef database update
```

In **SQL Server Object Explorer**, open the Student table designer by double-clicking the **Student** table.



Before you applied the first two migrations, the name columns were of type `nvarchar(MAX)`. They are now `nvarchar(50)` and the column name has changed from `FirstMidName` to `FirstName`.

NOTE

If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Final changes to the Student entity

Student
Properties
☑ ID
🔑 LastName
🔑 FirstMidName
🔑 EnrollmentDate
Navigation Properties
☑ Enrollments

In *Models/Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute is not needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically treated as required fields.

You could remove the `Required` attribute and replace it with a minimum length parameter for the `StringLength` attribute:

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a get accessor, and no `FullName` column will be generated in the database.

Create the Instructor Entity

Instructor	
Properties	
PK	ID
	LastName
	FirstMidName
	HireDate
Navigation Properties	
	CourseAssignments
	OfficeAssignment

Create `Models/Instructor.cs`, replacing the template code with the following code:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}

```

Notice that several properties are the same in the Student and Instructor entities. In the [Implementing Inheritance](#) tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the `HireDate` attributes as follows:

```

[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]

```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```

public ICollection<CourseAssignment> CourseAssignments { get; set; }

```

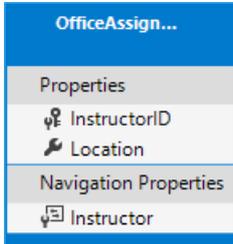
If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity (which may be null if no office is assigned).

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

There's a one-to-zero-or-one relationship between the `Instructor` and the `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the `Instructor` entity. But the Entity Framework can't automatically recognize `InstructorID` as the primary key of this entity because its name doesn't follow the `ID` or `classnameID` naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something other than `classnameID` or `ID`.

By default, EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The `Instructor` entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the `OfficeAssignment` entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an `Instructor` entity has a related `OfficeAssignment` entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the `InstructorID` foreign key (which is also the key to this table) is non-nullable.

Modify the Course Entity

Course
Properties
CourseID
Title
Credits
DepartmentID
Navigation Properties
Department
Enrollments
CourseAssignments

In `Models/Course.cs`, replace the code you added earlier with the following code. The changes are highlighted.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related Department entity and it has a `Department` navigation property.

The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they are needed and creates [shadow properties](#) for them. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the Department entity is null if you don't load it, so when you update the course entity, you would have to first fetch the Department entity.

When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the Department entity before you update.

The DatabaseGenerated attribute

The `DatabaseGenerated` attribute with the `None` parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for Course entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see [Generated Properties](#).

Foreign key and navigation properties

The foreign key properties and navigation properties in the Course entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained [later](#)):

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create the Department entity

Department
Properties
DepartmentID
Name
Budget
StartDate
InstructorID
Navigation Properties
Administrator
Courses

Create `Models/Department.cs` with the following code:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}

```

The Column attribute

Earlier you used the `Column` attribute to change column name mapping. In the code for the Department entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

```

[Column(TypeName="money")]
public decimal Budget { get; set; }

```

Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the Instructor entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation property is named `Administrator` but holds an Instructor entity:

```

public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }

```

A department may have many courses, so there's a Courses navigation property:

```

public ICollection<Course> Courses { get; set; }

```

NOTE

By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the `Department.InstructorID` property as nullable, EF would configure a cascade delete rule to delete the instructor when you delete the department, which is not what you want to have happen. If your business rules required the `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

Modify the Enrollment entity

Enrollment
Properties
• EnrollmentID
• CourseID
• StudentID
• Grade
Navigation Properties
• Course
• Student

In `Models/Enrollment.cs`, replace the code you added earlier with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The foreign key properties and navigation properties reflect the following relationships:

An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

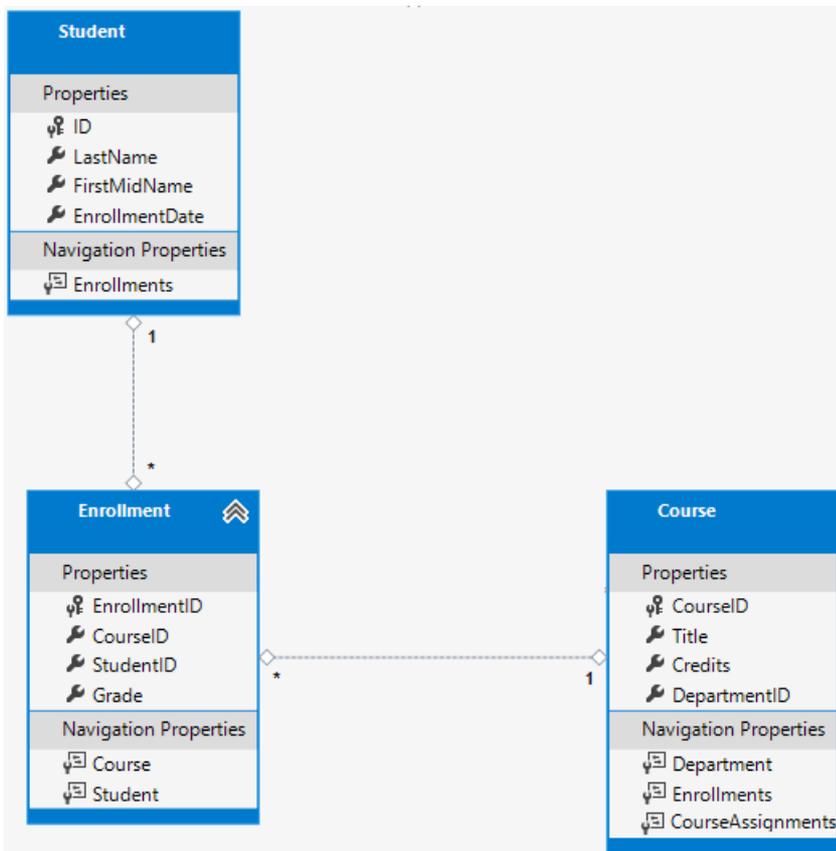
An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the Student and Course entities, and the Enrollment entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the Enrollment table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a Grade property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)



Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the Enrollment table didn't include grade information, it would only need to contain the two foreign keys `CourseID` and `StudentID`. In that case, it would be a many-to-many join table without payload (or a pure join table) in the database. The Instructor and Course entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

(EF 6.x supports implicit join tables for many-to-many relationships, but EF Core does not. For more information, see the [discussion in the EF Core GitHub repository](#).)

The CourseAssignment entity

CourseAssignment	
Properties	
	CourseID
	InstructorID
Navigation Properties	
	Course
	Instructor

Create *Models/CourseAssignment.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there is no need for a separate primary key. The `InstructorID` and `CourseID` properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see [Indexes](#).

Update the database context

Add the following highlighted code to the *Data/SchoolContext.cs* file:

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}

```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

Fluent API alternative to attributes

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the [EF Core documentation](#):

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}

```

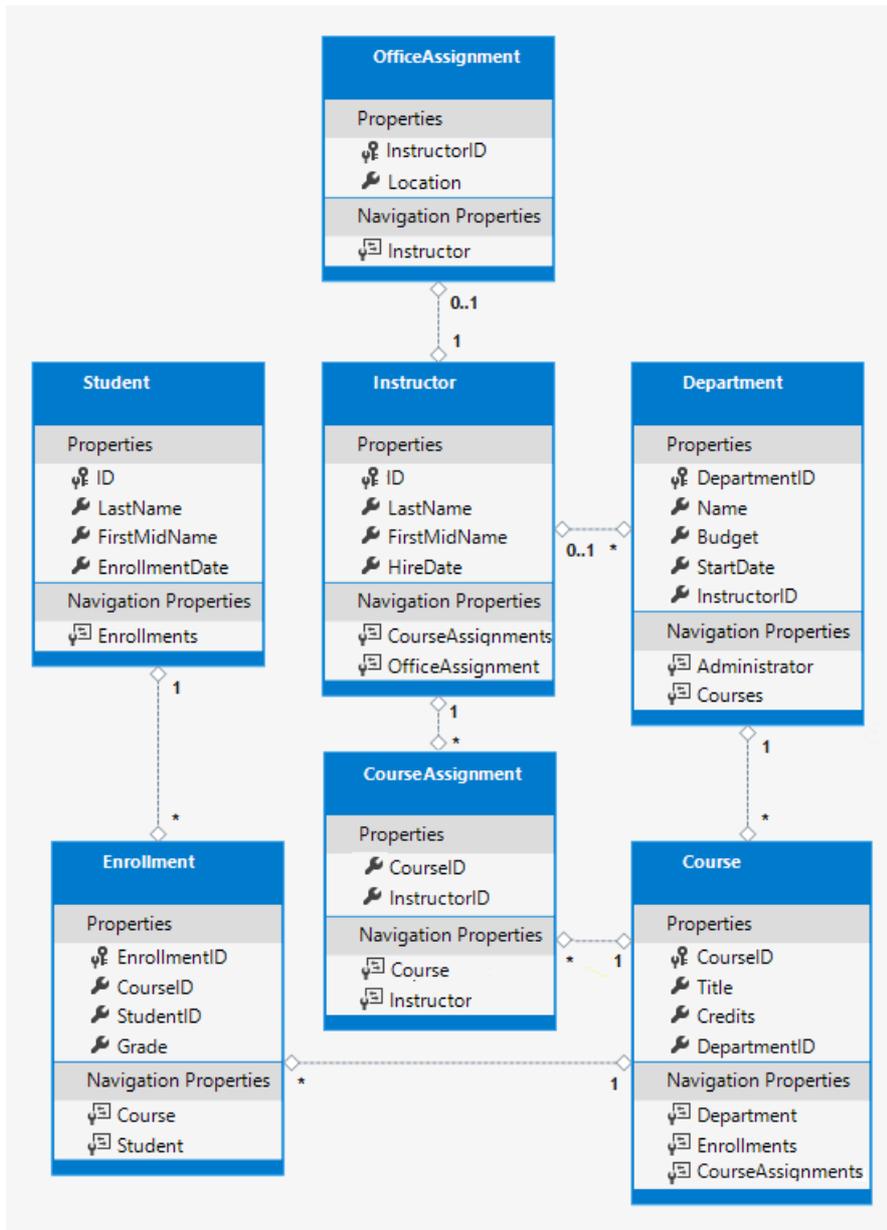
In this tutorial, you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible. If you do use both, note that wherever there is a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see [Methods of configuration](#).

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the Instructor and OfficeAssignment entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Seed the Database with Test Data

Replace the code in the *Data/DbInitializer.cs* file with the following code in order to provide seed data for the new entities you've created.

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
```

```

{
public static void Initialize(SchoolContext context)
{
    //context.Database.EnsureCreated();

    // Look for any students.
    if (context.Students.Any())
    {
        return; // DB has been seeded
    }

    var students = new Student[]
    {
        new Student { FirstMidName = "Carson", LastName = "Alexander",
            EnrollmentDate = DateTime.Parse("2010-09-01") },
        new Student { FirstMidName = "Meredith", LastName = "Alonso",
            EnrollmentDate = DateTime.Parse("2012-09-01") },
        new Student { FirstMidName = "Arturo", LastName = "Anand",
            EnrollmentDate = DateTime.Parse("2013-09-01") },
        new Student { FirstMidName = "Gytis", LastName = "Barzdukas",
            EnrollmentDate = DateTime.Parse("2012-09-01") },
        new Student { FirstMidName = "Yan", LastName = "Li",
            EnrollmentDate = DateTime.Parse("2012-09-01") },
        new Student { FirstMidName = "Peggy", LastName = "Justice",
            EnrollmentDate = DateTime.Parse("2011-09-01") },
        new Student { FirstMidName = "Laura", LastName = "Norman",
            EnrollmentDate = DateTime.Parse("2013-09-01") },
        new Student { FirstMidName = "Nino", LastName = "Olivetto",
            EnrollmentDate = DateTime.Parse("2005-09-01") }
    };

    foreach (Student s in students)
    {
        context.Students.Add(s);
    }
    context.SaveChanges();

    var instructors = new Instructor[]
    {
        new Instructor { FirstMidName = "Kim", LastName = "Abercrombie",
            HireDate = DateTime.Parse("1995-03-11") },
        new Instructor { FirstMidName = "Fadi", LastName = "Fakhouri",
            HireDate = DateTime.Parse("2002-07-06") },
        new Instructor { FirstMidName = "Roger", LastName = "Harui",
            HireDate = DateTime.Parse("1998-07-01") },
        new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
            HireDate = DateTime.Parse("2001-01-15") },
        new Instructor { FirstMidName = "Roger", LastName = "Zheng",
            HireDate = DateTime.Parse("2004-02-12") }
    };

    foreach (Instructor i in instructors)
    {
        context.Instructors.Add(i);
    }
    context.SaveChanges();

    var departments = new Department[]
    {
        new Department { Name = "English", Budget = 350000,
            StartDate = DateTime.Parse("2007-09-01"),
            InstructorID = instructors.Single( i => i.LastName == "Abercrombie").ID },
        new Department { Name = "Mathematics", Budget = 100000,
            StartDate = DateTime.Parse("2007-09-01"),
            InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID },
        new Department { Name = "Engineering", Budget = 350000,
            StartDate = DateTime.Parse("2007-09-01"),
            InstructorID = instructors.Single( i => i.LastName == "Harui").ID },
        new Department { Name = "Economics", Budget = 100000,
    }
}

```

```

        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID }
};

foreach (Department d in departments)
{
    context.Departments.Add(d);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course {CourseID = 1050, Title = "Chemistry", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
    },
    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 1045, Title = "Calculus", Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 3141, Title = "Trigonometry", Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 2021, Title = "Composition", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
    new Course {CourseID = 2042, Title = "Literature", Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
};

foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var officeAssignments = new OfficeAssignment[]
{
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
        Location = "Smith 17" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
        Location = "Gowan 27" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
        Location = "Thompson 304" },
};

foreach (OfficeAssignment o in officeAssignments)
{
    context.OfficeAssignments.Add(o);
}
context.SaveChanges();

var courseInstructors = new CourseAssignment[]
{
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    }
};

```

```

    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    },
    },
};

foreach (CourseAssignment ci in courseInstructors)
{
    context.CourseAssignments.Add(ci);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        Grade = Grade.A
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        Grade = Grade.C
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
    },
    },
};

```

```

        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Li").ID,
            CourseID = courses.Single(c => c.Title == "Composition").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Justice").ID,
            CourseID = courses.Single(c => c.Title == "Literature").CourseID,
            Grade = Grade.B
        }
    };

    foreach (Enrollment e in enrollments)
    {
        var enrollmentInDataBase = context.Enrollments.Where(
            s =>
                s.Student.ID == e.StudentID &&
                s.Course.CourseID == e.CourseID).SingleOrDefault();
        if (enrollmentInDataBase == null)
        {
            context.Enrollments.Add(e);
        }
    }
    context.SaveChanges();
}
}
}

```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If you tried to run the `database update` command at this point (don't do it yet), you would get the following error:

```
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint
"FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity",
table "dbo.Department", column 'DepartmentID'.
```

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints. The generated code in the `Up` method adds a non-nullable `DepartmentID` foreign key to the `Course` table. If there are already rows in the `Course` table when the code runs, the `AddColumn` operation fails because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing `Course` rows will all be related to the "Temp" department after the `Up` method runs.

- Open the `{timestamp}_ComplexDataModel.cs` file.
- Comment out the line of code that adds the `DepartmentID` column to the `Course` table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

- Add the following highlighted code after the code that creates the `Department` table:

```

migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
                SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
    GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);

```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the Course.DepartmentID column.

Save your changes and build the project.

Change the connection string and update the database

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in `appsettings.json` to `ContosoUniversity3` or some other name that you haven't used on the computer you're using.

```

{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },

```

Save your change to `appsettings.json`.

NOTE

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer (SSOX)** or the `database drop` CLI command:

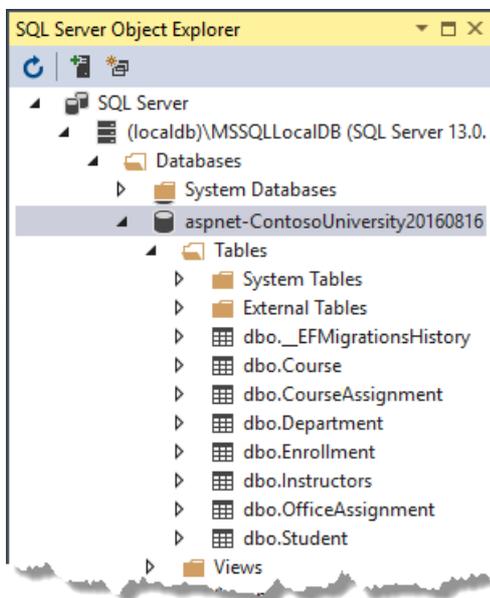
```
dotnet ef database drop
```

After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

```
dotnet ef database update
```

Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

Open the database in SSOX as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have SSOX open from the earlier time, click the **Refresh** button.)



Run the app to trigger the initializer code that seeds the database.

Right-click the **CourseAssignment** table and select **View Data** to verify that it has data in it.

The screenshot shows the ContosoUniversity application window. The 'dbo.CourseAssignment [Data]' table is displayed with the following data:

	CourseID	InstructorID
▶	2021	1
	2042	1
	1045	2
	1050	3
	3141	3
	1050	4
	4022	5
	4041	5
*	NULL	NULL

Summary

You now have a more complex data model and corresponding database. In the following tutorial, you'll learn more about how to access related data.

[PREVIOUS](#)[NEXT](#)

Reading related data - EF Core with ASP.NET Core MVC tutorial (6 of 10)

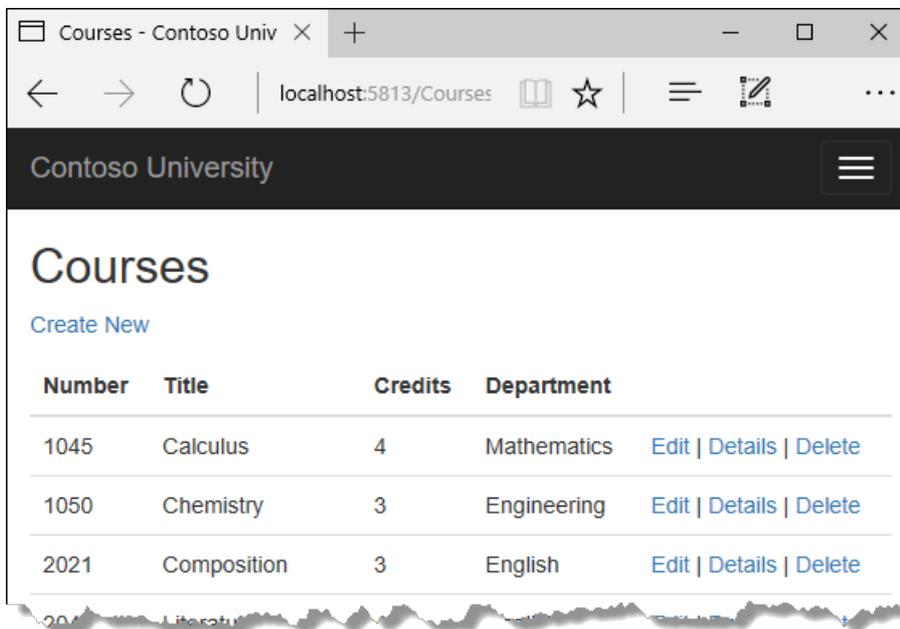
10/4/2017 • 14 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial, you completed the School data model. In this tutorial, you'll read and display related data -- that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.



Instructors - Contoso UI

localhost:5813/Instructors/Index/1?

Contoso University

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

Eager, explicit, and lazy Loading of related data

There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

- Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of

previously retrieved entities. For the query that retrieves related data, you can use the `Load` method instead of a method that returns a list or object, such as `ToList` or `Single`.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- **Explicit loading.** When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed. As in the case of eager loading with separate queries, explicit loading results in multiple queries sent to the database. The difference is that with explicit loading, the code specifies the navigation properties to be loaded. In Entity Framework Core 1.1 you can use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- **Lazy loading.** When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. A query is sent to the database each time you try to get data from a navigation property for the first time. Entity Framework Core 1.0 does not support lazy loading.

Performance considerations

If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager loading of all related data would result in just a single (join) query and a single round trip to the database. A separate query for courses for each department would result in eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, separate queries might perform better because eager loading of everything up front would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Create a Courses page that displays Department name

The Course entity includes a navigation property that contains the Department entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the Name property from the Department entity that is in the `Course.Department` navigation property.

Create a controller named `CoursesController` for the Course entity type, using the same options for the **MVC Controller with views, using Entity Framework** scaffolder that you did earlier for the Students controller, as shown in the following illustration:

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Course (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. The 'Controller name' text box contains 'CoursesController'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page'. The 'Add' button is highlighted with a red dashed border.

Open *CoursesController.cs* and examine the `Index` method. The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Replace the `Index` method with the following code that uses a more appropriate name for the `IQueryable` that returns `Course` entities (`courses` instead of `schoolContext`):

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Open *Views/Courses/Index.cshtml* and replace the template code with the following code. The changes are highlighted:

```

@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

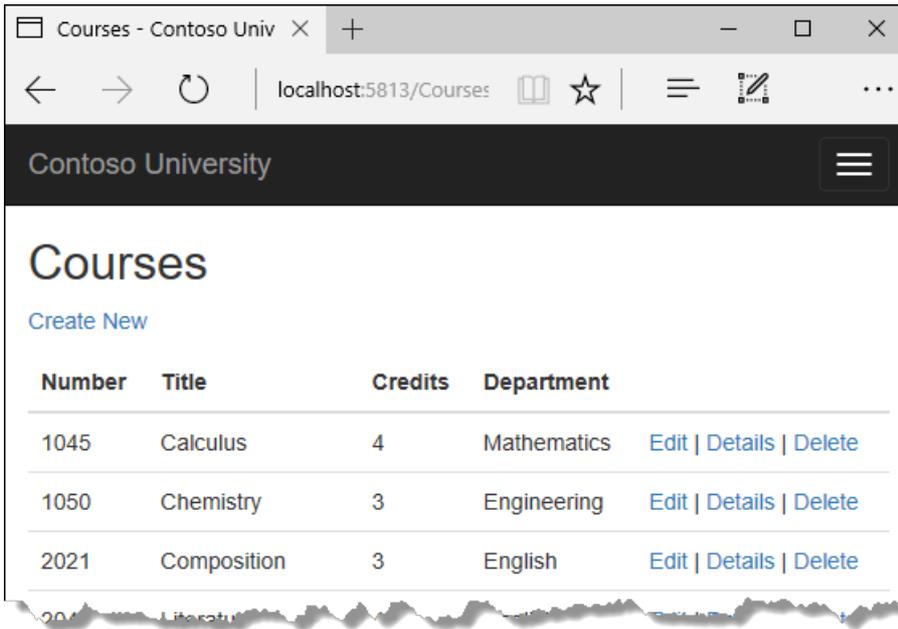
```

You've made the following changes to the scaffolded code:

- Changed the heading from Index to Courses.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they are meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the Department entity that's loaded into the `Department` navigation property:

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Create an Instructors page that shows Courses and Enrollments

In this section, you'll create a controller and view for the Instructor entity in order to display the Instructors page:

The screenshot shows a web browser window with the URL `localhost:5813/Instructors/Index/1?`. The page title is "Contoso University" and the main heading is "Instructors". There is a "Create New" link. Below is a table of instructors with columns: Last Name, First Name, Hire Date, Office, and Courses. Each row has "Select | Edit | Details | Delete" links. The first instructor, Kim Abercrombie, is selected. Below this is a section titled "Courses Taught by Selected Instructor" with a table showing course details (Number, Title, Department) and "Select" links. The first course, 2021 Composition, is selected. Below that is a section titled "Students Enrolled in Selected Course" with a table showing student names and grades.

Last Name	First Name	Hire Date	Office	Courses
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry

Number	Title	Department
2021	Composition	English
2042	Literature	English

Name	Grade
Alonso, Meredith	B
Li, Yan	B

This page reads and displays related data in the following ways:

- The list of instructors displays related data from the OfficeAssignment entity. The Instructor and OfficeAssignment entities are in a one-to-zero-or-one relationship. You'll use eager loading for the OfficeAssignment entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.
- When the user selects an instructor, related Course entities are displayed. The Instructor and Course entities are in a many-to-many relationship. You'll use eager loading for the Course entities and their related Department entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the Enrollments entity set is displayed. The Course and Enrollment entities are in a one-to-many relationship. You'll use separate queries for Enrollment entities and their related Student entities.

Create a view model for the Instructor Index view

The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

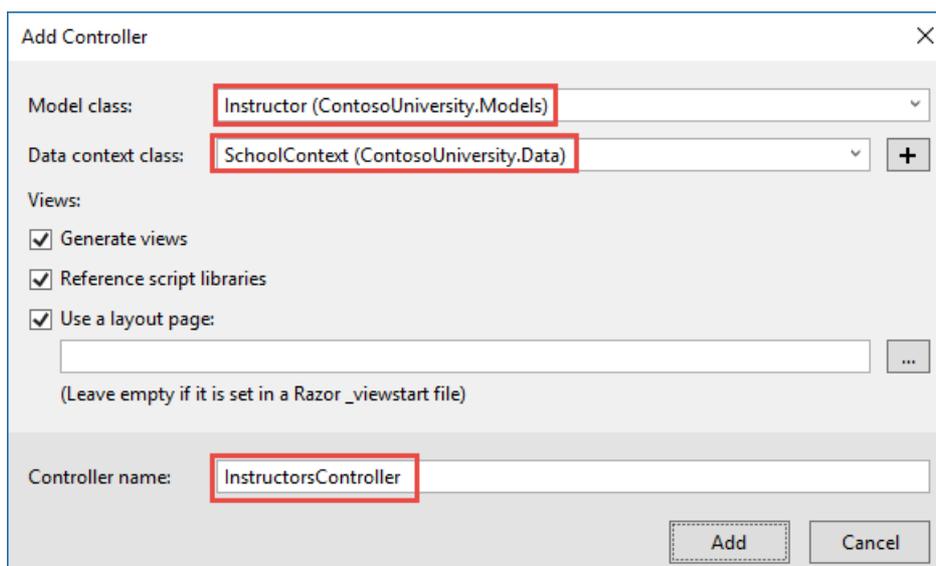
In the *SchoolViewModels* folder, create *InstructorIndexData.cs* and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:



The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Instructor (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. The 'Views' section has three checked options: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. The 'Controller name' text box contains 'InstructorsController'. The 'Add' and 'Cancel' buttons are at the bottom right.

Open *InstructorsController.cs* and add a using statement for the ViewModels namespace:

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

```

public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
        .ThenInclude(i => i.Student)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}

```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

```

viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

Since the view always requires the OfficeAssignment entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `Enrollment.Student`.

```
viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `Course.Enrollments`.

```
viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the Course entities from that instructor's `CourseAssignments` navigation property.

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single Instructor entity being returned. The `Single` method converts the collection into a single Instructor entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the `Where` condition instead of calling the `Where` method separately:

```
.Single(i => i.ID == id.Value)
```

Instead of:

```
.Where(i => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Modify the Instructor Index view

In `Views/Instructors/Index.cshtml`, replace the template code with the following code. The changes are highlighted.

```

@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @: @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` is not null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

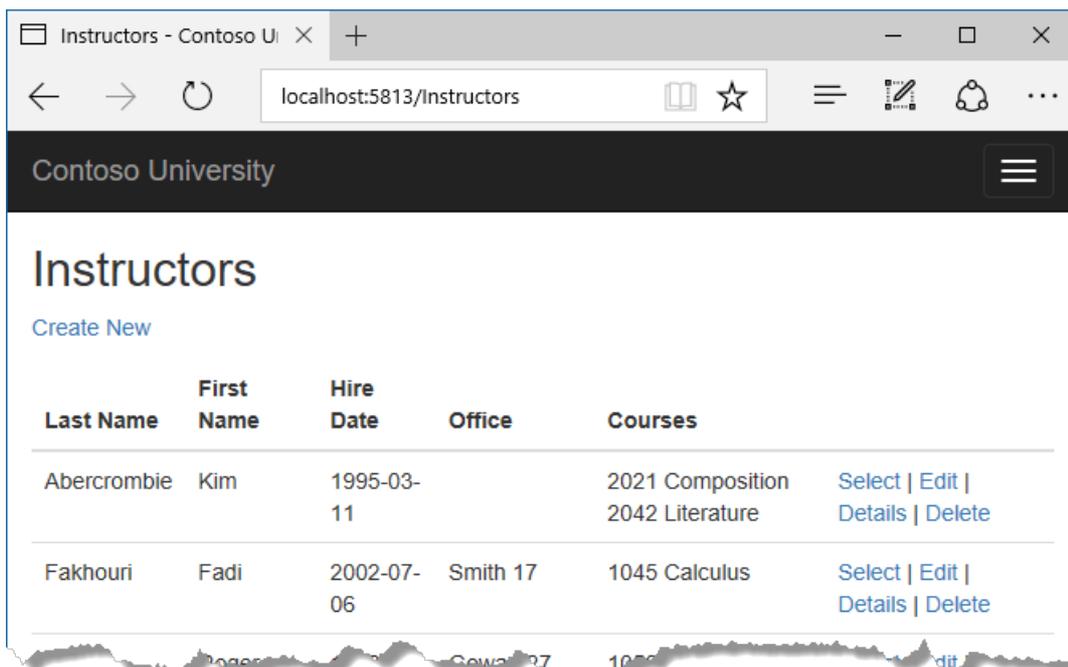
- Added a **Courses** column that displays courses taught by each instructor. See [Explicit Line Transition with @:](#) for more about this razor syntax.
- Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Added a new hyperlink labeled **Select** immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.



In the `Views/Instructors/Index.cshtml` file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

```

@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

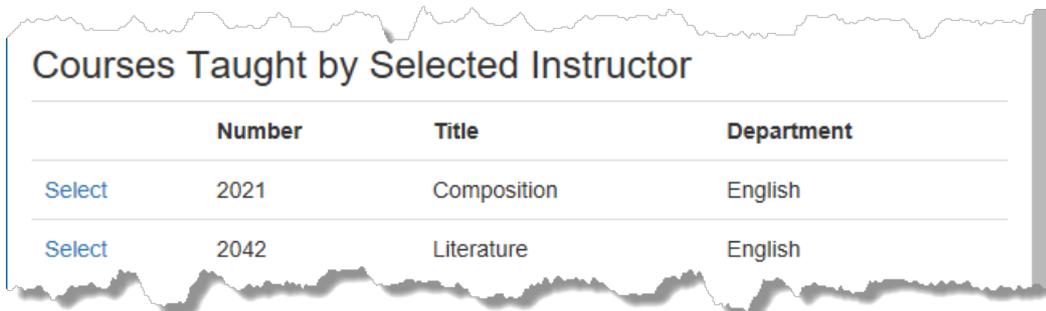
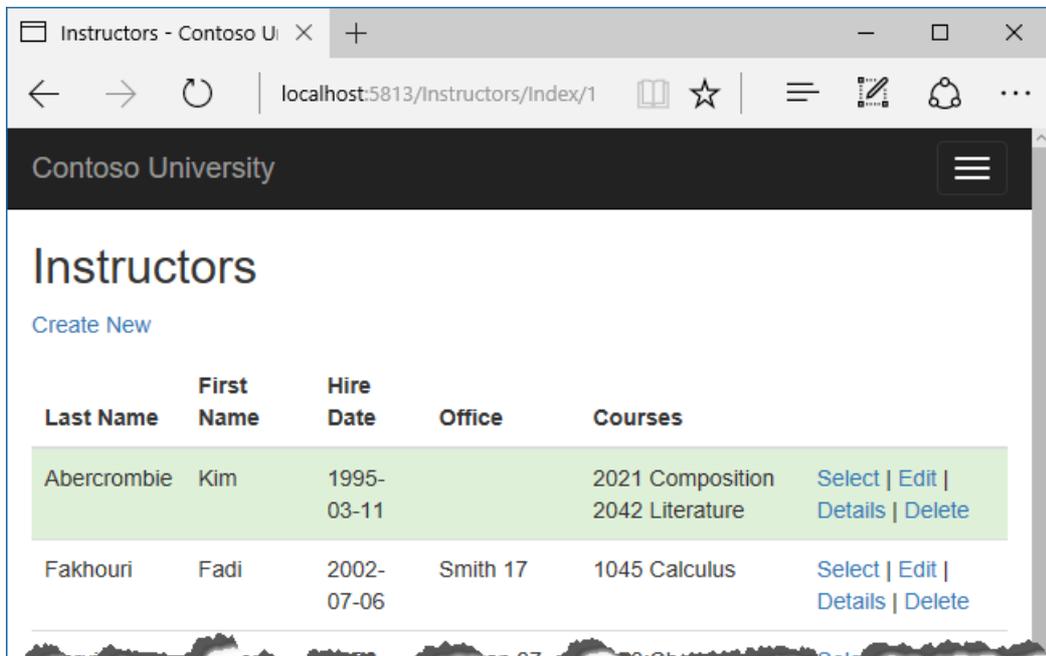
        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}

```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a **Select** hyperlink that sends the ID of the selected course to the `Index` action method.

Refresh the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



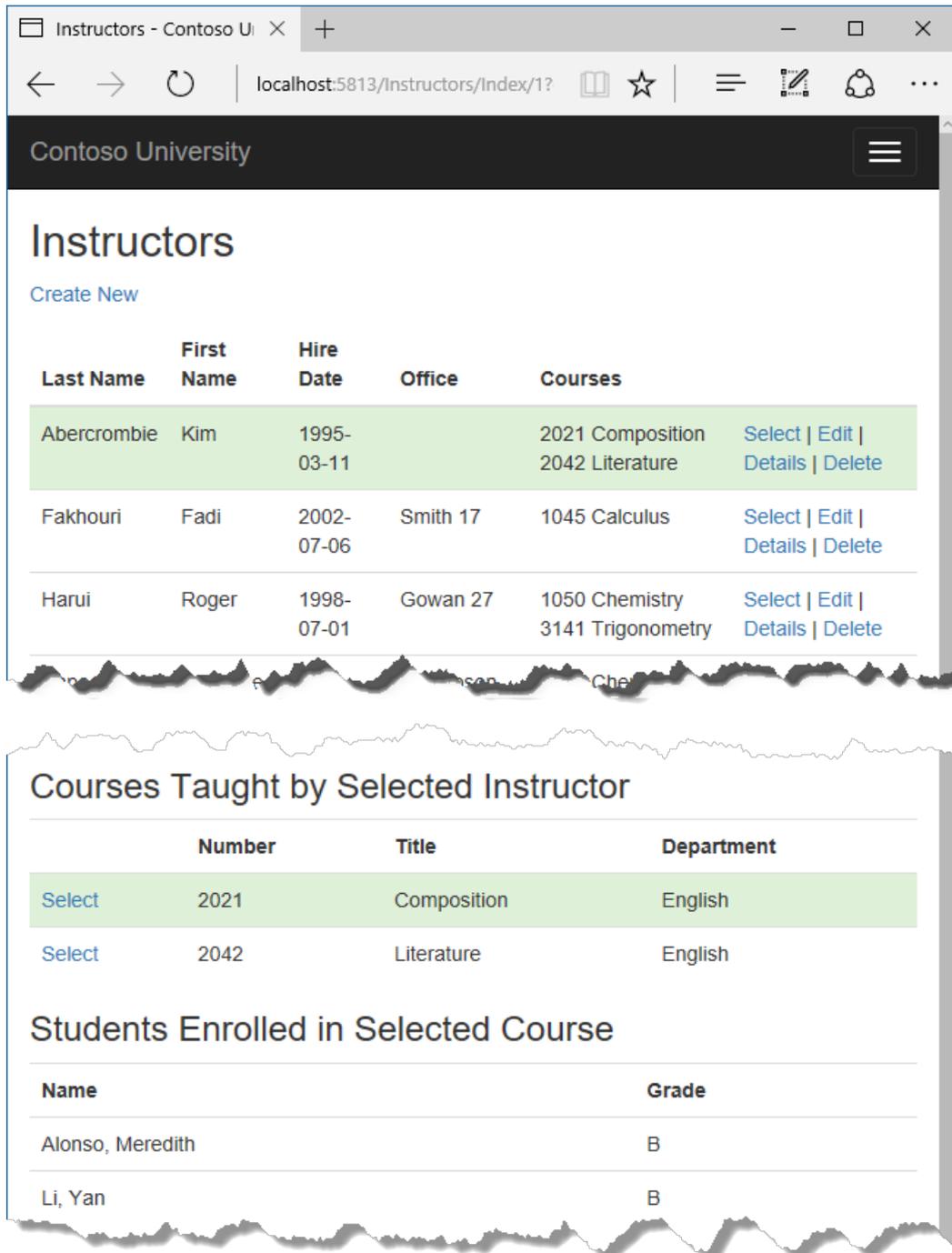
After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the Enrollments property of the view model in order to display a list of students enrolled in the course.

Refresh the page again and select an instructor. Then select a course to see the list of enrolled students and their

grades.



Explicit loading

When you retrieved the list of instructors in *InstructorsController.cs*, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for Enrollments and loads that property explicitly. The code changes are highlighted.

```

public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}

```

The new code drops the *ThenInclude* method calls for enrollment data from the code that retrieves instructor entities. If an instructor and course are selected, the highlighted code retrieves Enrollment entities for the selected course, and Student entities for each Enrollment.

Run the app, go to the Instructors Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Summary

You've now used eager loading with one query and with multiple queries to read related data into navigation properties. In the next tutorial you'll learn how to update related data.

[PREVIOUS](#)
[NEXT](#)

Updating related data - EF Core with ASP.NET Core MVC tutorial (7 of 10)

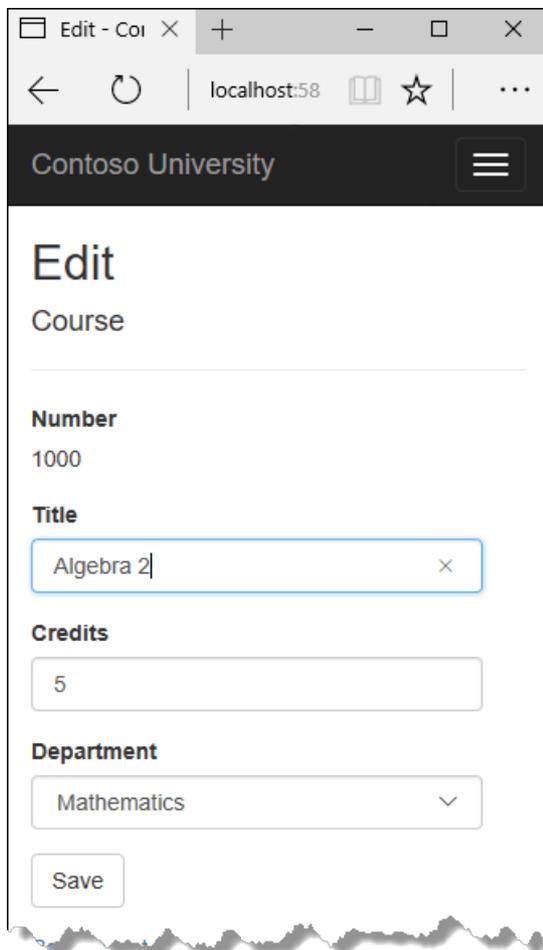
11/11/2017 • 18 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.



Edit - Contoso Universit × + - □ ×

localhost:5813/Instruct

Contoso University

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

1000 Algebra 2 1045 Calculus 1050 Chemistry
 2021 Composition 2042 Literature 3141 Trigonometry
 4022 Microeconomics 4041 Macroeconomics

Save

Customize the Create and Edit Pages for Courses

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate Department entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In `CoursesController.cs`, delete the four Create and Edit methods and replace them with the following code:

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .SingleOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}

```

After the `Edit` `HttpPost` method, create a new method that loads department info for the drop-down list.

```

private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in _context.Departments
        orderby d.Name
        select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(), "DepartmentID", "Name",
        selectedDepartment);
}

```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The `HttpGet Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department is not established yet:

```

public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}

```

The `HttpGet Edit` method sets the selected item, based on the ID of the department that is already assigned to

the course being edited:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The `HttpPost` methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add `.AsNoTracking` to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and `HttpGet Delete` methods.

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

```

public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}

```

Modify the Course views

In *Views/Courses/Create.cshtml*, add a "Select Department" option to the **Department** drop-down list, change the caption from **DepartmentID** to **Department**, and add a validation message.

```

<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />

```

In *Views/Courses/Edit.cshtml*, make the same change for the Department field that you just did in *Create.cshtml*.

Also in *Views/Courses/Edit.cshtml*, add a course number field before the **Title** field. Because the course number is the primary key, it's displayed, but it can't be changed.

```

<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>

```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the **Edit** page.

In *Views/Courses/Delete.cshtml*, add a course number field at the top and change department ID to department name.

```

@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

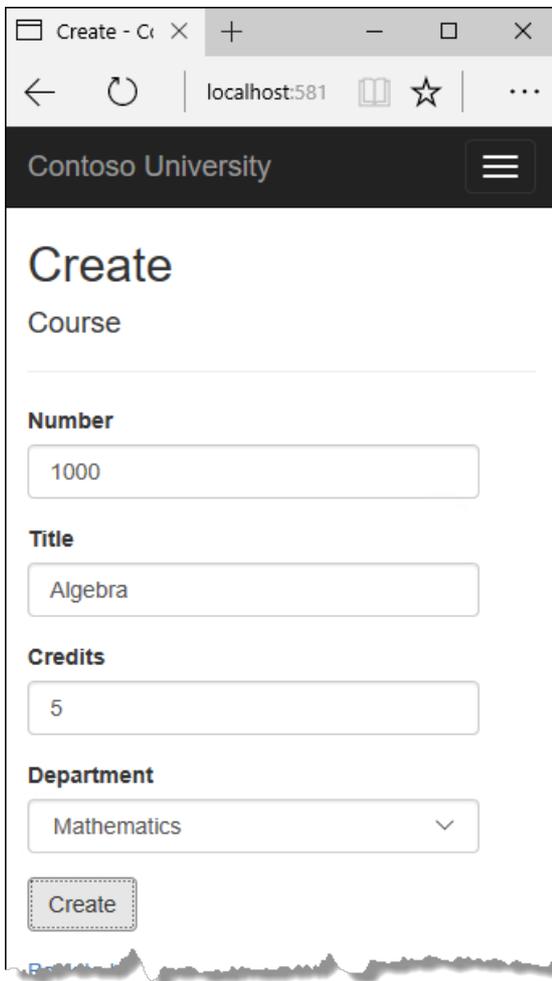
    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>

```

In *Views/Courses/Details.cshtml*, make the same change that you just did for *Delete.cshtml*.

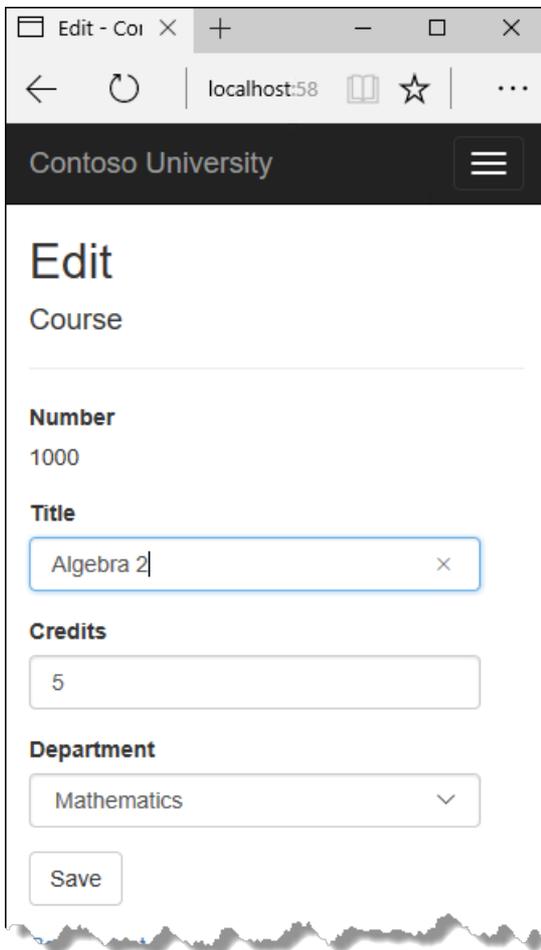
Test the Course pages

Run the app, select the **Courses** tab, click **Create New**, and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Click **Edit** on a course in the Courses Index page.



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add an Edit Page for Instructors

When you edit an instructor record, you want to be able to update the instructor's office assignment. The Instructor entity has a one-to-zero-or-one relationship with the OfficeAssignment entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the OfficeAssignment entity.
- If the user enters an office assignment value and it originally was empty, create a new OfficeAssignment entity.
- If the user changes the value of an office assignment, change the value in an existing OfficeAssignment entity.

Update the Instructors controller

In *InstructorsController.cs*, change the code in the `HttpGet Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

```

public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}

```

Replace the `HttpPost Edit` method with the following code to handle office assignment updates:

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .SingleOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}

```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the `HttpGet Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment`

navigation property. This is the same as what you did in the `HttpGet Edit` method.

- Updates the retrieved Instructor entity with values from the model binder. The `TryUpdateModel` overload enables you to whitelist the properties you want to include. This prevents over-posting, as explained in the [second tutorial](#).

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

```
if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

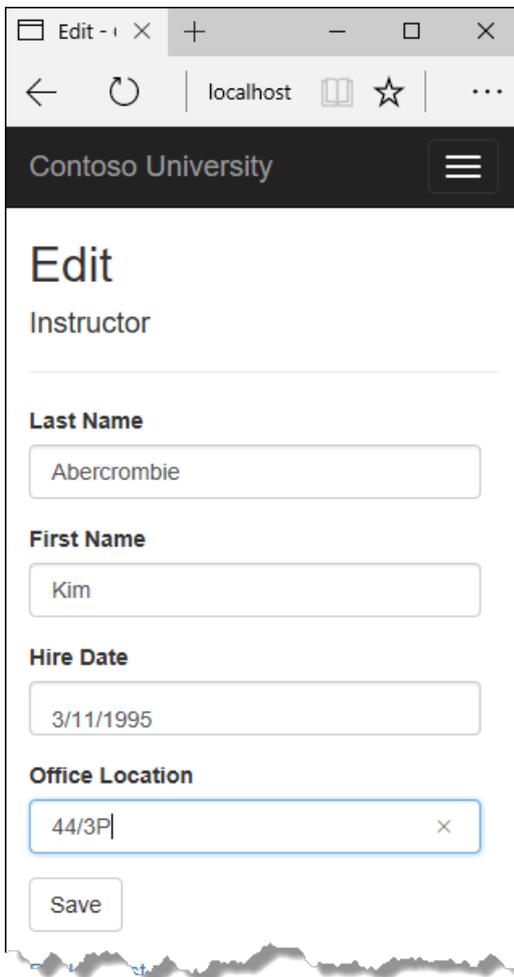
- Saves the changes to the database.

Update the Instructor Edit view

In `Views/Instructors/Edit.cshtml`, add a new field for editing the office location, at the end before the **Save** button:

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>
```

Run the app, select the **Instructors** tab, and then click **Edit** on an instructor. Change the **Office Location** and click **Save**.



Add Course assignments to the Instructor Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:

Edit - Contoso Universit x + - □ x

localhost:5813/Instruct

Contoso University

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

1000 Algebra 2 1045 Calculus 1050 Chemistry
 2021 Composition 2042 Literature 3141 Trigonometry
 4022 Microeconomics 4041 Macroeconomics

The relationship between the Course and Instructor entities is many-to-many. To add and remove relationships, you add and remove entities to and from the CourseAssignments join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of check boxes, you'll use a view model class.

Create *AssignedCourseData.cs* in the *SchoolViewModels* folder and replace the existing code with the following code:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}

```

In *InstructorsController.cs*, replace the `HttpGet` `Edit` method with the following code. The changes are highlighted.

```

public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}

```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set to true for courses the instructor is assigned to. The view will use this property to determine which check boxes must

be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .SingleOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    UpdateInstructorCourses(selectedCourses, instructorToUpdate);
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}
```

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

The method signature is now different from the `HttpGet Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of `Course` entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the whitelisting overload and `CourseAssignments` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

Update the Instructor views

In *Views/Instructors/Edit.cshtml*, add a **Courses** field with an array of check boxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

NOTE

When you paste the code in Visual Studio, line breaks will be changed in a way that breaks the code. Press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab three times to line up the new code with the existing code. You can check the status of this problem [here](#).

```

<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <table>
      <tr>
        @{
          int cnt = 0;
          List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

          foreach (var course in courses)
          {
            if (cnt++ % 3 == 0)
            {
              @:</tr><tr>
            }
            @:<td>
              <input type="checkbox"
                name="selectedCourses"
                value="@course.CourseID"
                @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                @course.CourseID @: @course.Title
            @:</td>
          }
        @:</tr>
      }
    </table>
  </div>
</div>

```

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they are to be treated as a group. The value attribute of each check box is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the app, select the **Instructors** tab, and click **Edit** on an instructor to see the **Edit** page.

Edit - Contoso Universit x + - □ x

localhost:5813/Instruct ☆ | ≡ ...

Contoso University ≡

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

1000 Algebra 2 1045 Calculus 1050 Chemistry
 2021 Composition 2042 Literature 3141 Trigonometry
 4022 Microeconomics 4041 Macroeconomics

Change some course assignments and click Save. The changes you make are reflected on the Index page.

NOTE

The approach taken here to edit instructor course data works well when there is a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update the Delete page

In *InstructorsController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

```

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}

```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to the Create page

In *InstructorsController.cs*, delete the `HttpGet` and `HttpPost` `Create` methods, and then add the following code in their place:

```

public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}

// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID =
int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The `HttpGet Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The `HttpPost Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the `Instructor` model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

```

private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}

```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In `Views/Instructor/Create.cshtml`, add an office location text box and check boxes for courses before the Submit button. As in the case of the Edit page, [fix the formatting if Visual Studio reformats the code when you paste it](#).

```

<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                name="selectedCourses"
                                value="@course.CourseID"
                                @(Html.Raw(course.Assigned ? "checked=\""checked\"" : "")) />
                            @course.CourseID @: @course.Title
                        @:</td>
                    }
                @:</tr>
            }
        </table>
    </div>
</div>

```

Test by running the app and creating an instructor.

Handling Transactions

As explained in the [CRUD tutorial](#), the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see [Transactions](#).

Summary

You have now completed the introduction to working with related data. In the next tutorial you'll see how to handle concurrency conflicts.

[PREVIOUS](#)[NEXT](#)

Handling concurrency conflicts - EF Core with ASP.NET Core MVC tutorial (8 of 10)

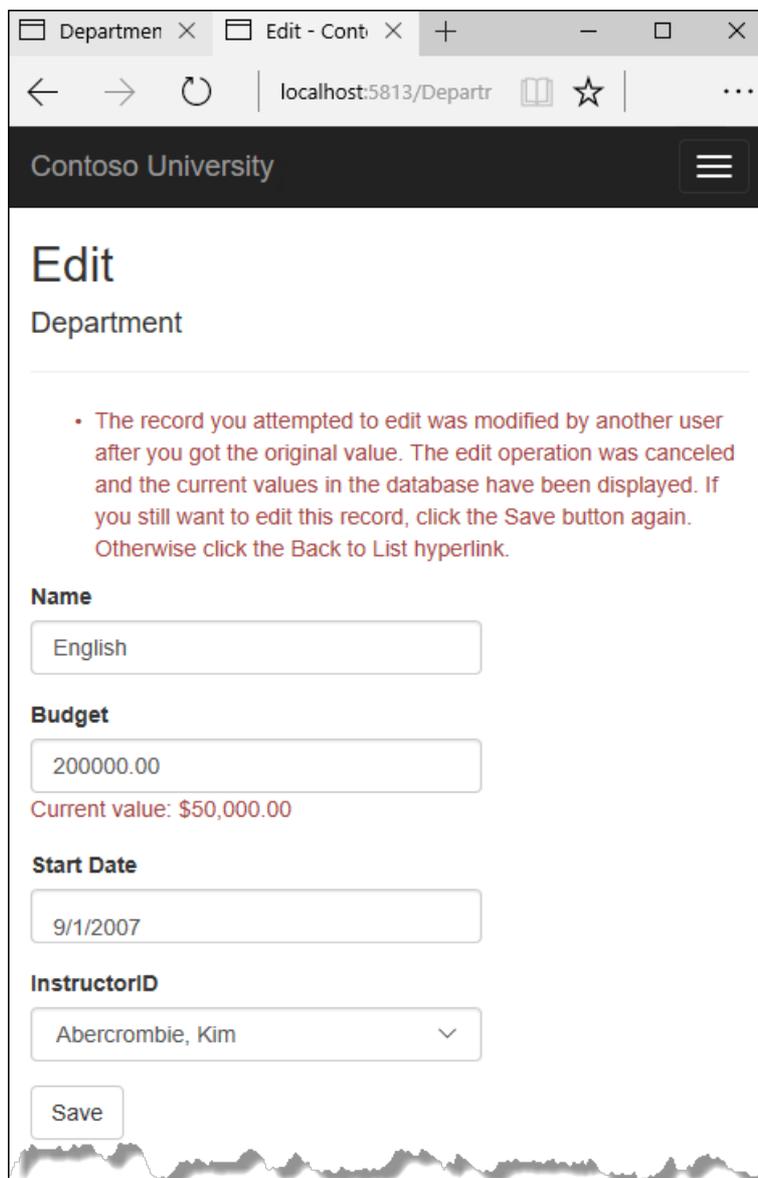
12/2/2017 • 18 min to read • [Edit Online](#)

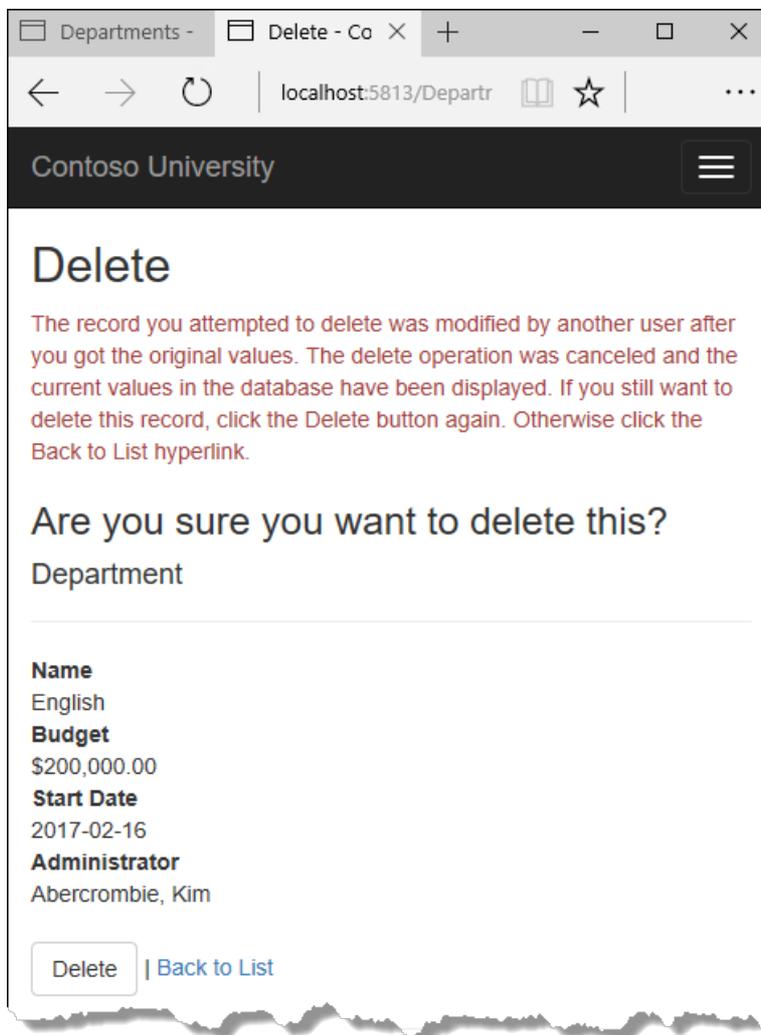
By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In earlier tutorials, you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the Department entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.





Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if it isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

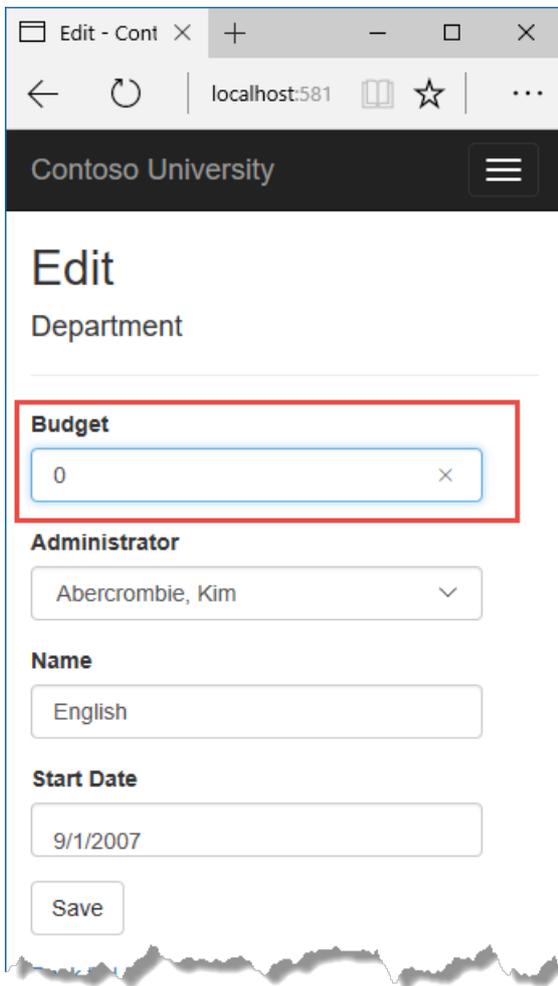
Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

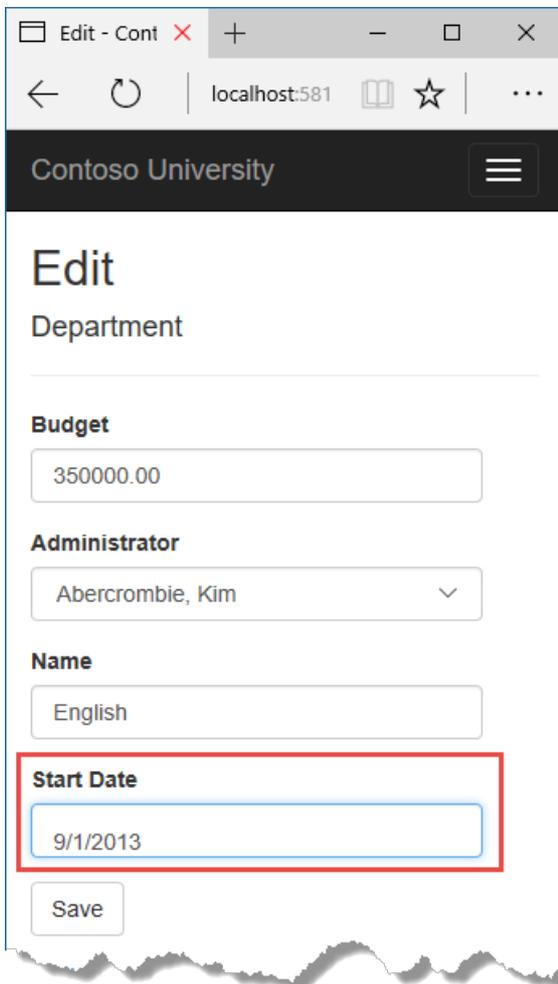
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

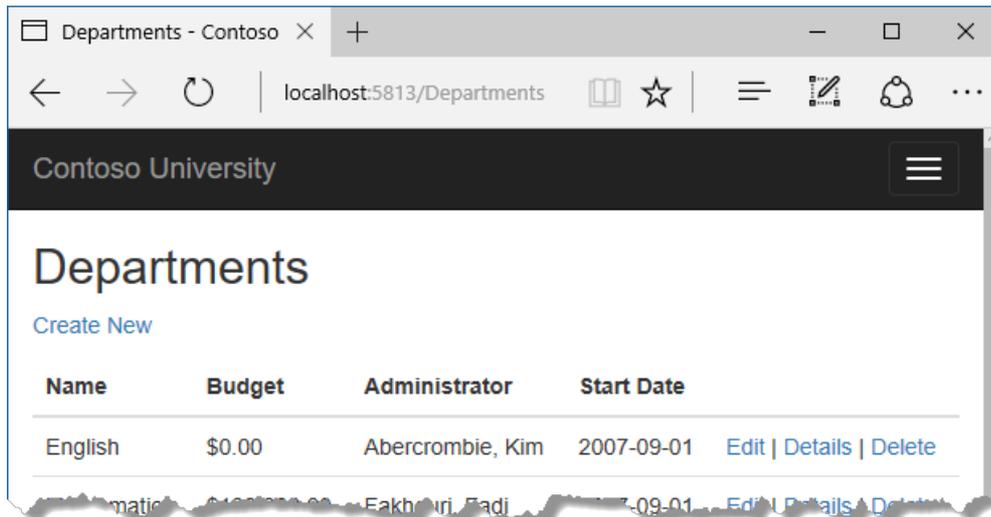
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from \$350,000.00 to \$0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



Then John clicks **Save** on an Edit page that still shows a budget of \$350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they'll see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let John's change overwrite Jane's change.

The next time someone browses the English department, they'll see 9/1/2013 and the restored \$350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent John's change from being updated in the database.

Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been

changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version). If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

Add a tracking property to the Department entity

In *Models/Department.cs*, add a tracking property named RowVersion:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}

```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsConcurrencyToken` method (in *Data/SchoolContext.cs*) to specify the tracking property, as shown in the following example:

```

modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();

```

By adding a property you changed the database model, so you need to do another migration.

Save your changes and build the project, and then enter the following commands in the command window:

```
dotnet ef migrations add RowVersion
```

```
dotnet ef database update
```

Create a Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Department (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. Below the 'Use a layout page' checkbox is an empty text box with a '...' button. The 'Controller name' text box contains 'DepartmentsController'. At the bottom right, there are 'Add' and 'Cancel' buttons. The 'Add' button is highlighted with a dashed border.

In the *DepartmentsController.cs* file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", department.InstructorID);
```

Update the Departments Index view

The scaffolding engine created a RowVersion column in the Index view, but that field shouldn't be displayed.

Replace the code in *Views/Departments/Index.cshtml* with the following code.

```

@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

This changes the heading to "Departments", deletes the RowVersion column, and shows full name instead of first name for the administrator.

Update the Edit methods in the Departments controller

In both the `HttpGet Edit` method and the `Details` method, add `AsNoTracking`. In the `HttpGet Edit` method, add eager loading for the Administrator.

```

var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .SingleOrDefaultAsync(m => m.DepartmentID == id);

```

Replace the existing code for the `HttpPost` `Edit` method with the following code:

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i => i.Administrator).SingleOrDefaultAsync(m
=> m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value: {databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {

```

```

        ModelState.AddModelError("StartDate", $"Current value: {databaseValues.StartDate:d}");
    }
    if (databaseValues.InstructorID != clientValues.InstructorID)
    {
        Instructor databaseInstructor = await _context.Instructors.SingleOrDefaultAsync(i => i.ID
== databaseValues.InstructorID);
        ModelState.AddModelError("InstructorID", $"Current value:
{databaseInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty, "The record you attempted to edit "
+ "was modified by another user after you got the original value. The "
+ "edit operation was canceled and the current values in the database "
+ "have been displayed. If you still want to edit this record, click "
+ "the Save button again. Otherwise click the Back to List hyperlink.");
    departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
    ModelState.Remove("RowVersion");
}
}
}
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
departmentToUpdate.InstructorID);
return View(departmentToUpdate);
}

```

The code begins by trying to read the department to be updated. If the `SingleOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a department entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the department entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

```

_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

```

var exceptionEntry = ex.Entries.Single();

```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.

```

var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();

```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
}
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Department Edit view

In *Views/Departments/Edit.cshtml*, make the following changes:

- Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.
- Add a "Select Administrator" option to the drop-down list.

```

@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

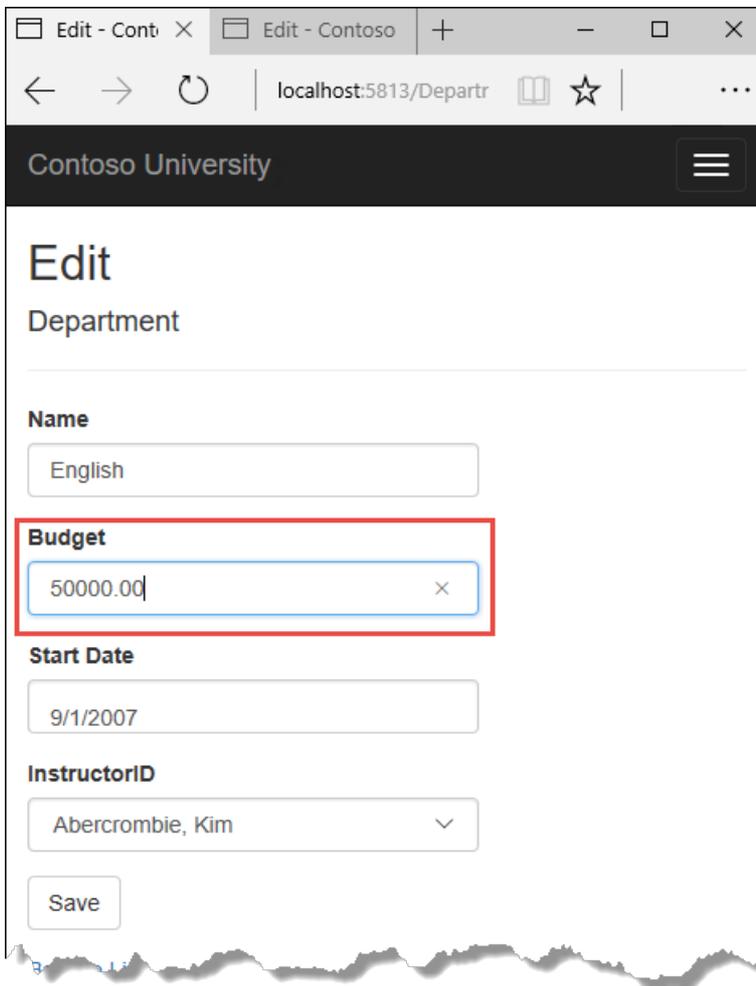
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Test concurrency conflicts in the Edit page

Run the app and go to the Departments Index page. Right-click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.



The browser shows the Index page with the changed value.

Change a field in the second browser tab.

Departments - Edit - Cont x + - □ x

localhost:5813/Departr ☆

Contoso University

Edit

Department

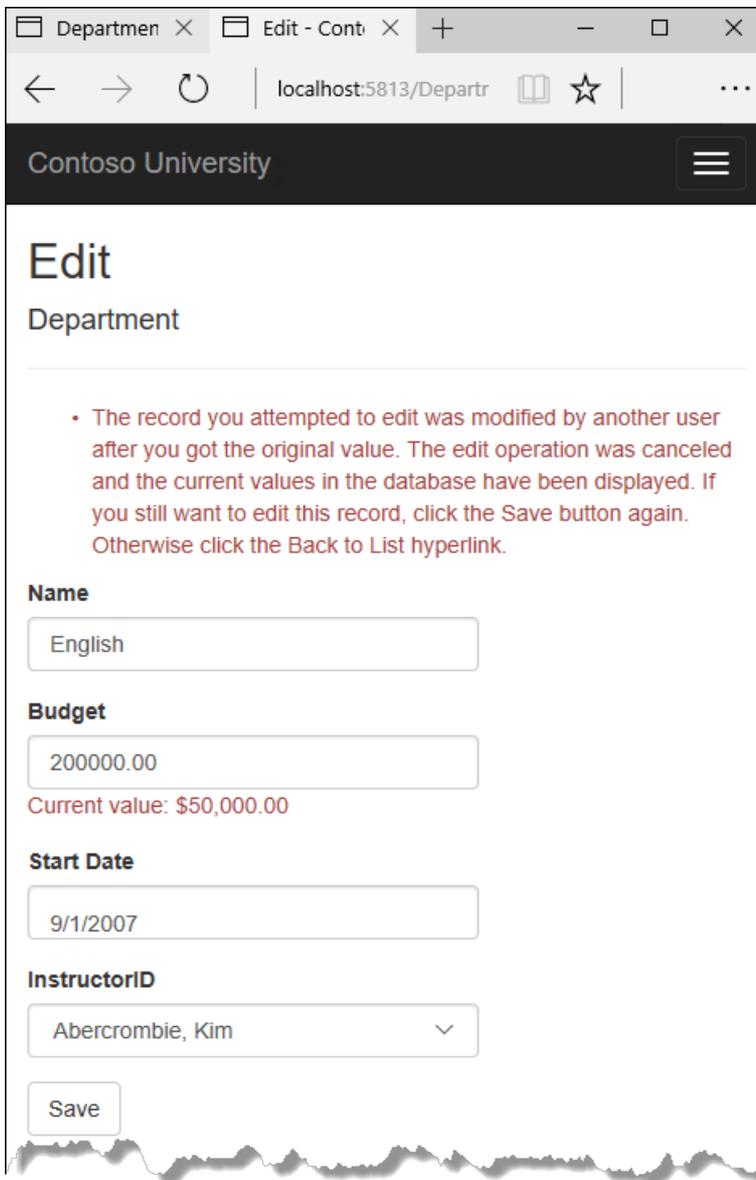
Name

Budget

Start Date

InstructorID

Click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the `HttpGet Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the `HttpPost Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the `HttpGet Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In `DepartmentsController.cs`, replace the `HttpGet Delete` method with the following code:

```

public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

    return View(department);
}

```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the Department does exist, it was changed by another user. In that case, the code sends an error message to the view using `ViewData`.

Replace the code in the `HttpPost Delete` method (named `DeleteConfirmed`) with the following code:

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID == department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof>Delete), new { concurrencyError = true, id = department.DepartmentID
    };
}
}

```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a Department entity instance created by the model binder. This gives EF access to the RowVersion property value in addition to the record key.

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplay the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In *Views/Departments/Delete.cshtml*, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

```

@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>

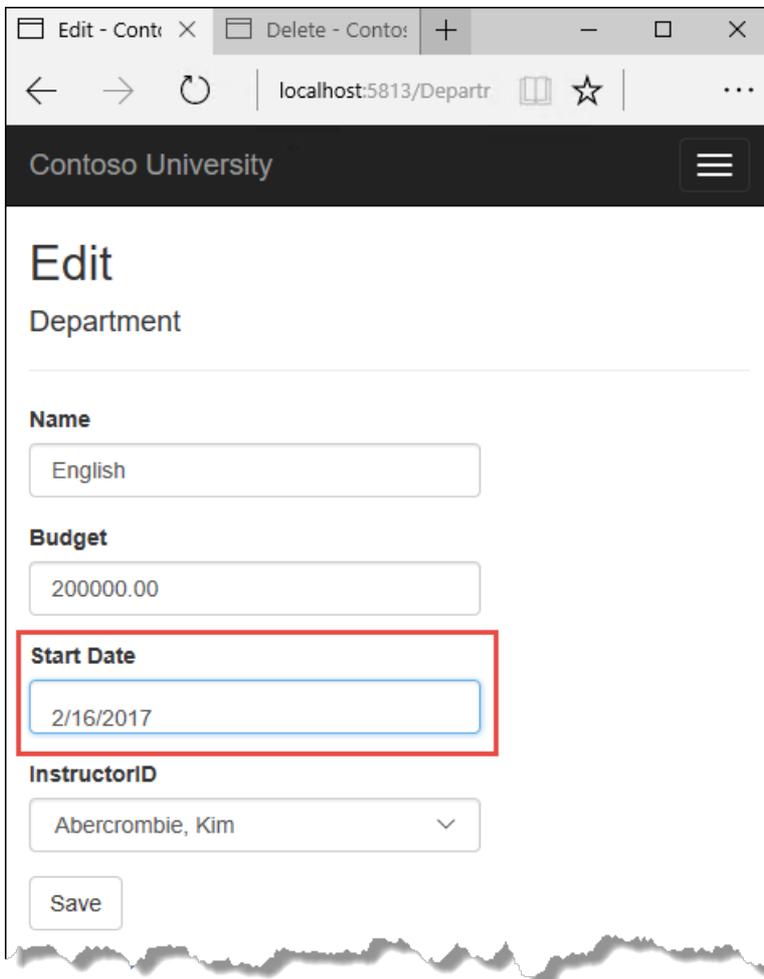
```

This makes the following changes:

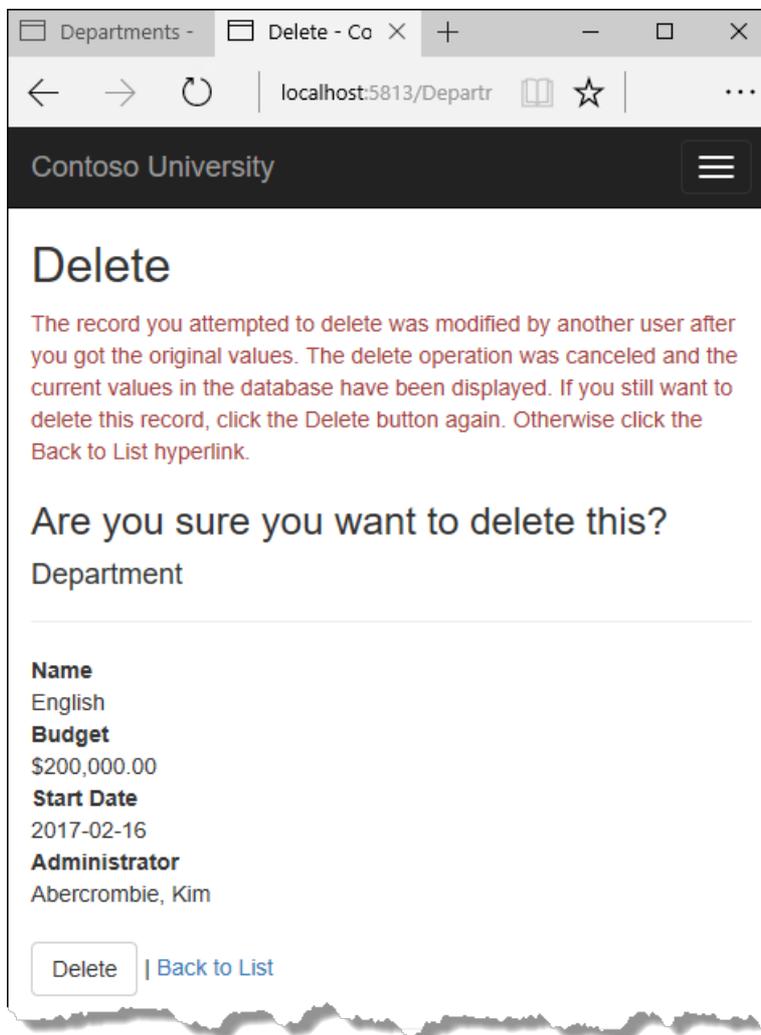
- Adds an error message between the `h2` and `h3` headings.
- Replaces `FirstMidName` with `FullName` in the **Administrator** field.
- Removes the `RowVersion` field.
- Adds a hidden field for the `RowVersion` property.

Run the app and go to the Departments Index page. Right-click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in *Views/Departments/Details.cshtml* to delete the RowVersion column and show the full name of the Administrator.

```

@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>

```

Replace the code in *Views/Departments/Create.cshtml* to add a Select option to the drop-down list.

```

@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Summary

This completes the introduction to handling concurrency conflicts. For more information about how to handle concurrency in EF Core, see [Concurrency conflicts](#). The next tutorial shows how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

[PREVIOUS](#)
[NEXT](#)

Inheritance - EF Core with ASP.NET Core MVC tutorial (9 of 10)

9/22/2017 • 7 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

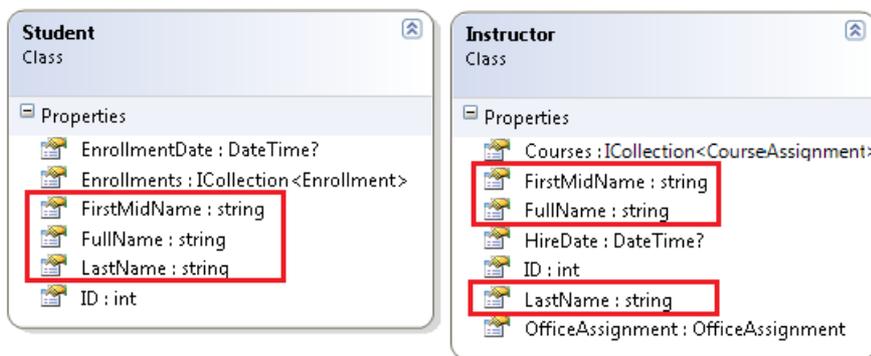
The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial, you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

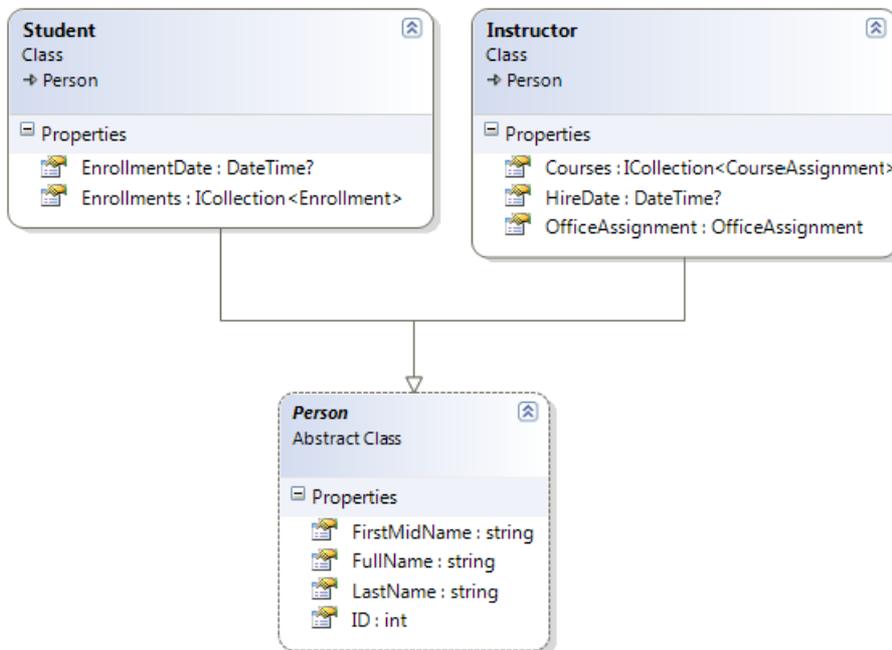
In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

Options for mapping inheritance to database tables

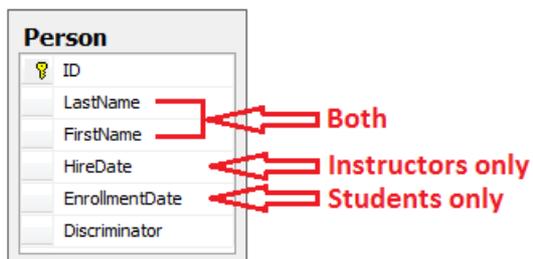
The `Instructor` and `Student` classes in the School data model have several properties that are identical:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:

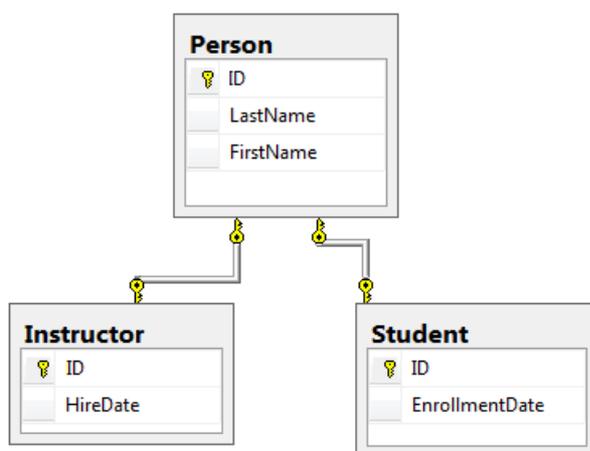


There are several ways this inheritance structure could be represented in the database. You could have a Person table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called table-per-hierarchy (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the Person table and have separate Instructor and Student tables with the date fields.



This pattern of making a database table for each entity class is called table per type (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including

inherited properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the Person, Student, and Instructor classes as shown earlier, the Student and Instructor tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

TIP

Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

Create the Person class

In the Models folder, create `Person.cs` and replace the template code with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }

        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

Make Student and Instructor classes inherit from Person

In `Instructor.cs`, derive the `Instructor` class from the `Person` class and remove the key and name fields. The code will look like the following example:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}

```

Make the same changes in *Student.cs*.

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

Add the Person entity type to the data model

Add the Person entity type to *SchoolContext.cs*. The new lines are highlighted.

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}

```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and customize migration code

Save your changes and build the project. Then open the command window in the project folder and enter the following command:

```
dotnet ef migrations add Inheritance
```

Don't run the `database update` command yet. That command will result in lost data because it will drop the Instructor table and rename the Student table to Person. You need to provide custom code to preserve existing data.

Open `Migrations/<timestamp>_Inheritance.cs` and replace the `Up` method with the following code:

```

protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table: "Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person", nullable: false, maxLength:
128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate,
Discriminator, OldId) SELECT LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS
Discriminator, ID AS OldId FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId =
Enrollment.StudentId AND Discriminator = 'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
        name: "IX_Enrollment_StudentID",
        table: "Enrollment",
        column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}

```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
- Adds nullable EnrollmentDate for students.
- Adds Discriminator column to indicate whether a row is for a student or an instructor.
- Makes HireDate nullable since student rows won't have hire dates.
- Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they'll get new primary key values.
- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command:

```
dotnet ef database update
```

(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

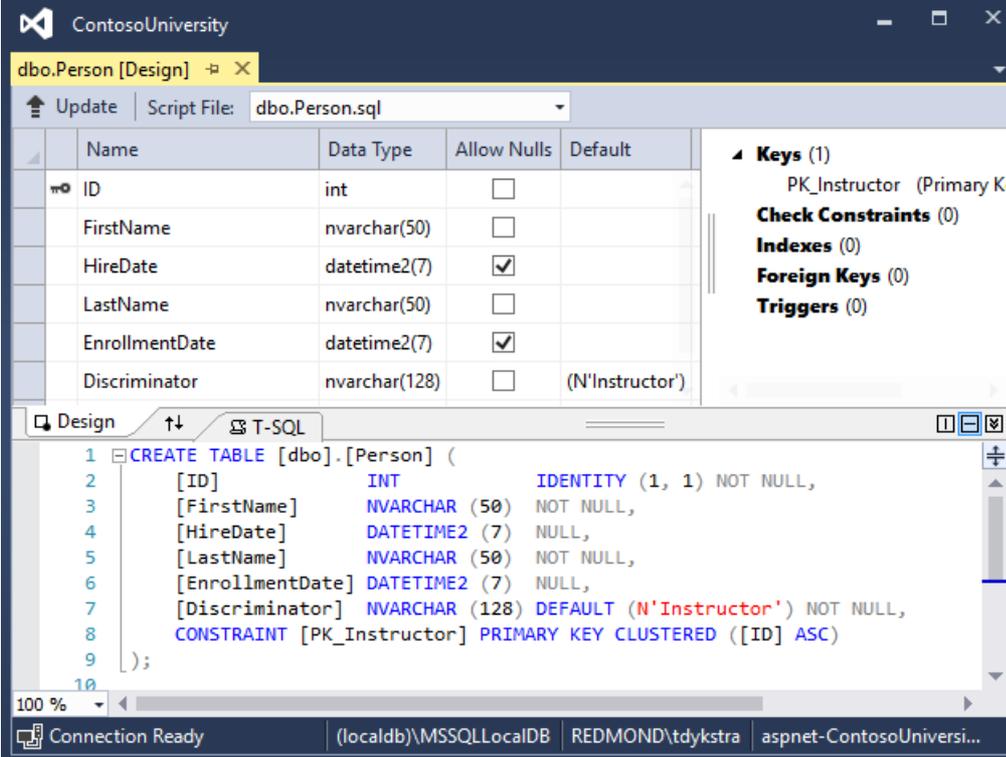
NOTE

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there is no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `database drop` CLI command.

Test with inheritance implemented

Run the app and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Name	Data Type	Allow Nulls	Default
ID	int	<input type="checkbox"/>	
FirstName	nvarchar(50)	<input type="checkbox"/>	
HireDate	datetime2(7)	<input checked="" type="checkbox"/>	
LastName	nvarchar(50)	<input type="checkbox"/>	
EnrollmentDate	datetime2(7)	<input checked="" type="checkbox"/>	
Discriminator	nvarchar(128)	<input type="checkbox"/>	(N'Instructor')

```
1 CREATE TABLE [dbo].[Person] (  
2     [ID] INT IDENTITY (1, 1) NOT NULL,  
3     [FirstName] NVARCHAR (50) NOT NULL,  
4     [HireDate] DATETIME2 (7) NULL,  
5     [LastName] NVARCHAR (50) NOT NULL,  
6     [EnrollmentDate] DATETIME2 (7) NULL,  
7     [Discriminator] NVARCHAR (128) DEFAULT (N'Instructor') NOT NULL,  
8     CONSTRAINT [PK_Instructor] PRIMARY KEY CLUSTERED ([ID] ASC)  
9 );  
10
```

Right-click the Person table, and then click **Show Table Data** to see the discriminator column.

ID	FirstName	HireDate	LastName	Enrollme...	Discriminator
1	Kim	3/11/1995...	Abercrombie	NULL	Instructor
2	Fadi	7/6/2002 ...	Fakhouri	NULL	Instructor
3	Roger	7/1/1998 ...	Harui	NULL	Instructor
4	Candace	1/15/2001...	Kapoor	NULL	Instructor
5	Roger	2/12/2004...	Zheng	NULL	Instructor
7	Nancy	8/17/2016...	Davolio	NULL	Instructor
8	Carson	NULL	Alexander	9/1/2010 ...	Student
9	Meredith	NULL	Alonso	9/1/2012 ...	Student
10	Arturo	NULL	Anand	9/1/2013 ...	Student
11	Gytis	NULL	Barzdukas	9/1/2012 ...	Student
12	Yan	NULL	Li	9/1/2012	Student

Summary

You've implemented table-per-hierarchy inheritance for the `Person`, `Student`, and `Instructor` classes. For more information about inheritance in Entity Framework Core, see [Inheritance](#). In the next tutorial you'll see how to handle a variety of relatively advanced Entity Framework scenarios.

[PREVIOUS](#)[NEXT](#)

Advanced topics - EF Core with ASP.NET Core MVC tutorial (10 of 10)

1/10/2018 • 12 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial, you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET Core web applications that use Entity Framework Core.

Raw SQL Queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

- Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they are automatically tracked by the database context unless you [turn tracking off](#).
- Use the `Database.ExecuteNonQueryCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Call a query that returns entities

The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In `DepartmentsController.cs`, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

```

public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

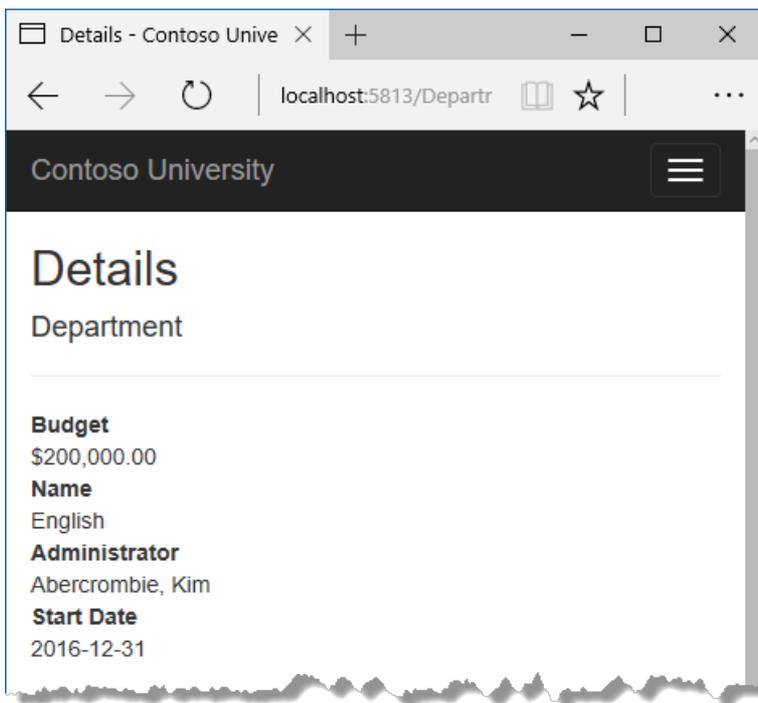
    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}

```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Call a query that returns other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is write ADO.NET code and get the database connection from EF.

In `HomeController.cs`, replace the `About` method with the following code:

```

public async Task<ActionResult> About()
{
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

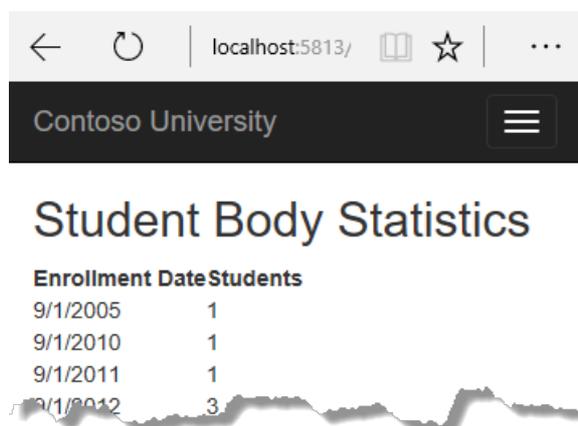
            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate = reader.GetDateTime(0), StudentCount
= reader.GetInt32(1) };
                    groups.Add(row);
                }
                reader.Dispose();
            }
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}

```

Add a using statement:

```
using System.Data.Common;
```

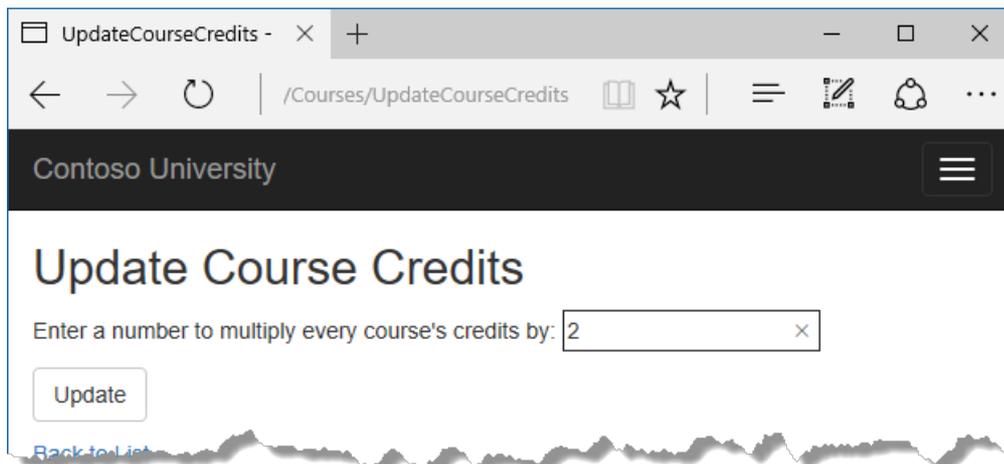
Run the app and go to the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the

change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In *CoursesController.cs*, add *UpdateCourseCredits* methods for *HttpGet* and *HttpPost*:

```
public IActionResult UpdateCourseCredits()
{
    return View();
}
```

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

When the controller processes an *HttpGet* request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the *HttpPost* method is called, and *multiplier* has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

In **Solution Explorer**, right-click the *Views/Courses* folder, and then click **Add > New Item**.

In the **Add New Item** dialog, click **ASP.NET** under **Installed** in the left pane, click **MVC View Page**, and name the new view *UpdateCourseCredits.cshtml*.

In *Views/Courses/UpdateCourseCredits.cshtml*, replace the template code with the following code:

```

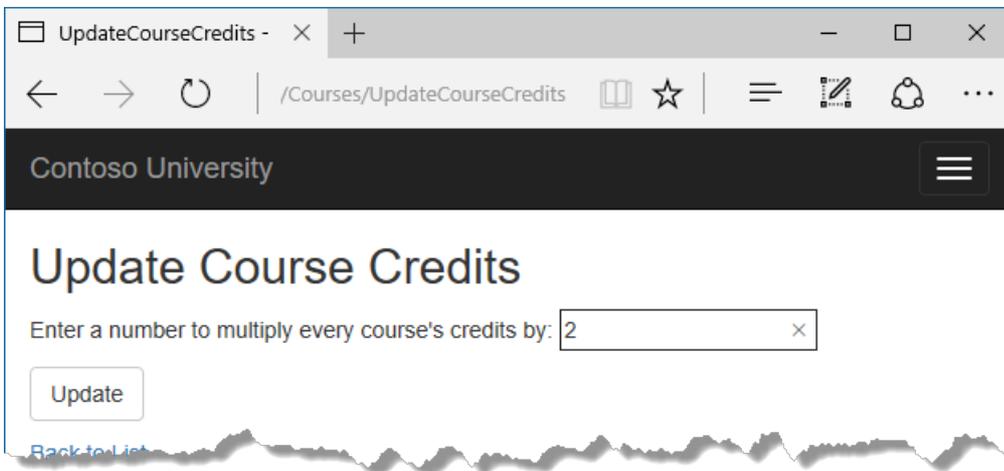
@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

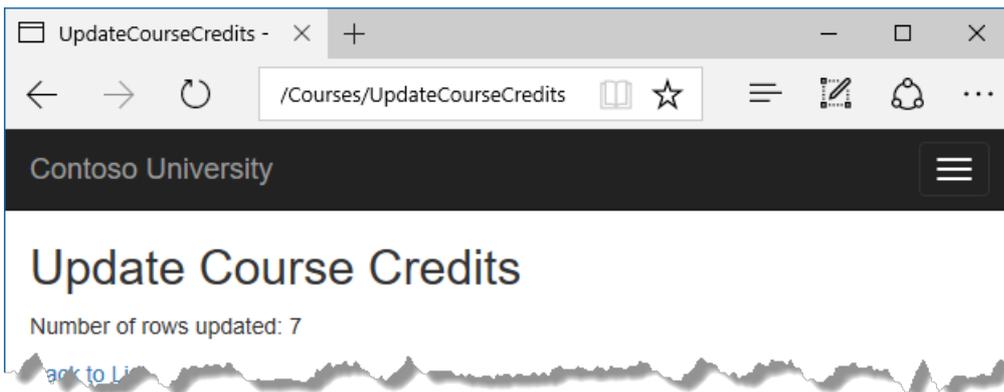
@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
            <p>
                Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default" />
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding `/UpdateCourseCredits` to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Courses/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see [Raw SQL Queries](#).

Examine SQL sent to the database

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open `StudentsController.cs` and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the app in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (56ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (122ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID], [s.Enrollments].[Grade], [s.Enrollments].
[StudentID], [e.Course].[CourseID], [e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] = [e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`) from the Person table. The `SingleOrDefaultAsync` method doesn't resolve to 1 row on the server. Here's why:

- If the query would return multiple rows, the method returns null.
- To determine whether the query would return multiple rows, EF has to check if it returns at least 2.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Repository and unit of work patterns

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns is not always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see [the Entity Framework 5 version of this tutorial series](#).

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see [Testing with InMemory](#).

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- `DbContext.SaveChanges`
- `DbContext.Entry`
- `ChangeTracker.Entries`

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

Entity Framework Core source code and development plans

The source code for Entity Framework Core is available at <https://github.com/aspnet/EntityFrameworkCore>. Besides source code, you can get nightly builds, issue tracking, feature specs, design meeting notes, [the roadmap for future development](#), and more. You can file bugs, and you can contribute your own enhancements to the EF source code.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the [scaffold-dbcontext](#) command. See the [getting-started tutorial](#).

Use dynamic LINQ to simplify sort selection code

The [third tutorial in this series](#) shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

```

public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] =
        String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
    ViewData["DateSortParm"] =
        sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" : "EnrollmentDate";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
        select s;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
            || s.FirstMidName.Contains(searchString));
    }

    if (string.IsNullOrEmpty(sortOrder))
    {
        sortOrder = "LastName";
    }

    bool descending = false;
    if (sortOrder.EndsWith("_desc"))
    {
        sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
        descending = true;
    }

    if (descending)
    {
        students = students.OrderByDescending(e => EF.Property<object>(e, sortOrder));
    }
    else
    {
        students = students.OrderBy(e => EF.Property<object>(e, sortOrder));
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
        page ?? 1, pageSize));
}

```

Next steps

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET MVC application.

For more information about EF Core, see the [Entity Framework Core documentation](#). A book is also available: [Entity Framework Core in Action](#).

For information on how to deploy a web app, see [Host and deploy](#).

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see the [ASP.NET Core documentation](#).

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials.

Common errors

ContosoUniversity.dll used by another process

Error message:

```
Cannot open '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.
```

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there is no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in *appsettings.json*. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the `database drop` CLI command:

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

```
A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is
```

configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

[PREVIOUS](#)

Creating Backend Services for Native Mobile Applications

9/30/2017 • 8 min to read • [Edit Online](#)

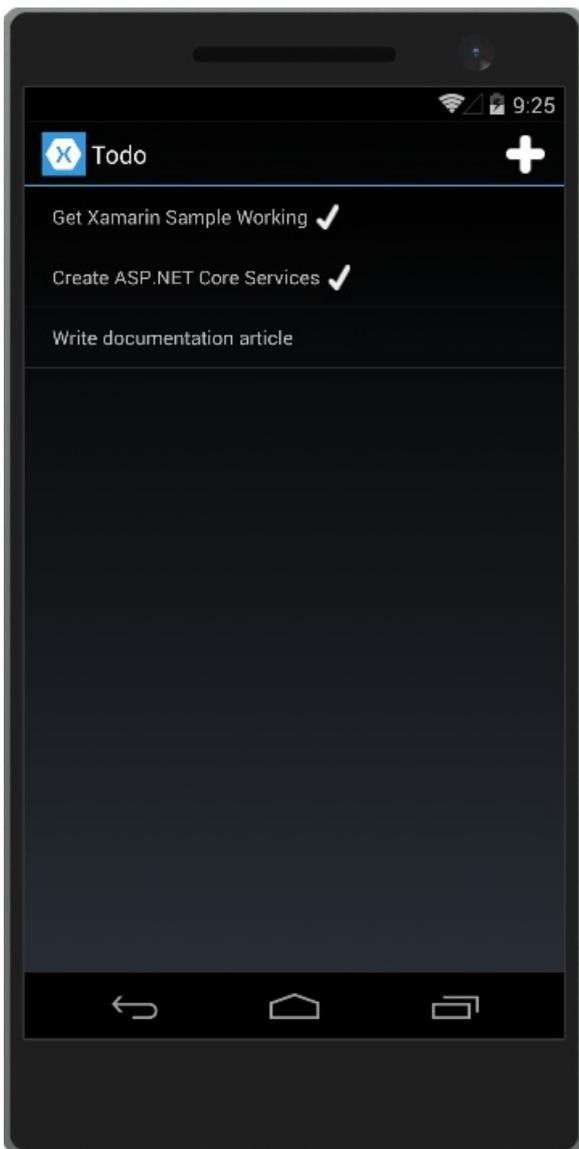
By [Steve Smith](#)

Mobile apps can easily communicate with ASP.NET Core backend services.

[View or download sample backend services code](#)

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core MVC to support native mobile apps. It uses the [Xamarin Forms ToDoRest app](#) as its native client, which includes separate native clients for Android, iOS, Windows Universal, and Window Phone devices. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), as well as download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Web API 2 services project, which this article's ASP.NET Core app replaces (with no changes required by the client).



Features

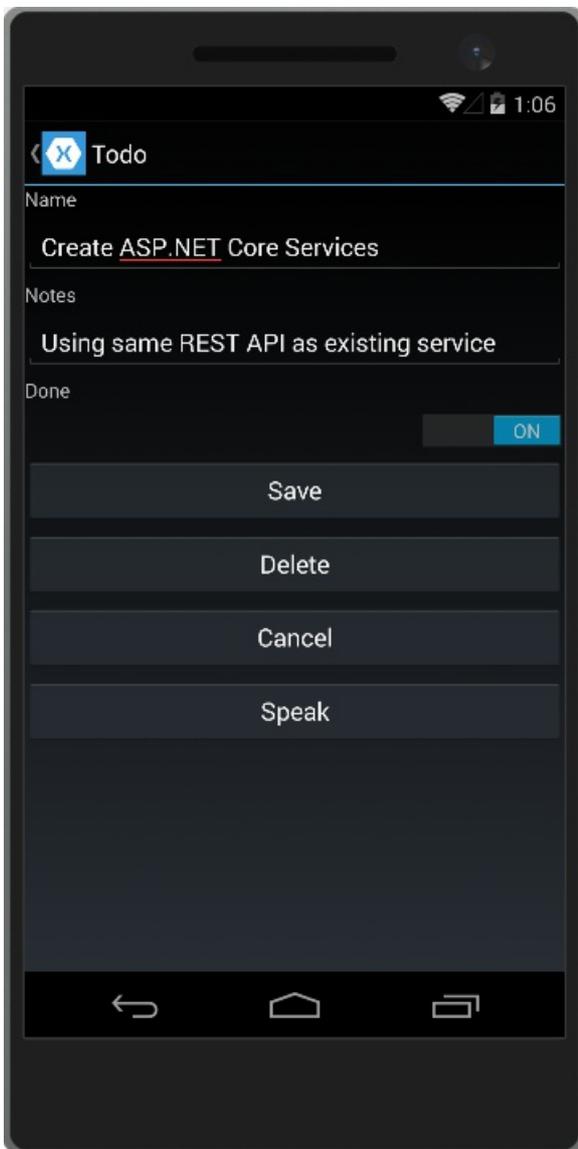
The ToDoRest app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

The main view of the items, as shown above, lists each item's name and indicates if it is done with a checkmark.

Tapping the  icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:



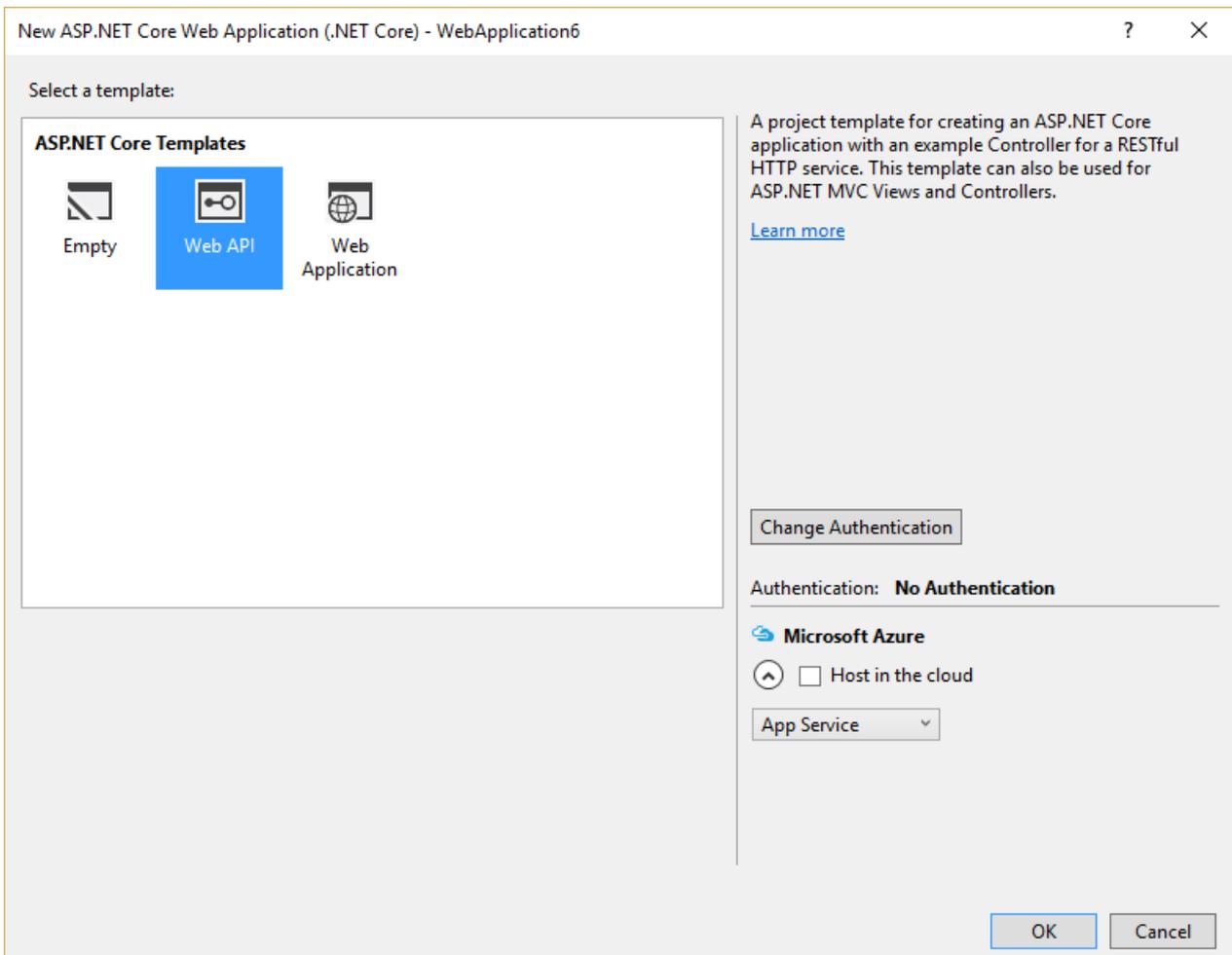
This sample is configured by default to use backend services hosted at `developer.xamarin.com`, which allow read-only operations. To test it out yourself against the ASP.NET Core app created in the next section running on your computer, you'll need to update the app's `RestUrl` constant. Navigate to the `ToDoREST` project and open the `Constants.cs` file. Replace the `RestUrl` with a URL that includes your machine's IP address (not localhost or 127.0.0.1, since this address is used from the device emulator, not from your machine). Include the port number as well (5000). In order to test that your services work with a device, ensure you don't have an active firewall blocking access to this port.

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template and No Authentication. Name the project *ToDoApi*.



The application should respond to all requests made to port 5000. Update *Program.cs* to include

`.UseUrls("http://*:5000")` to achieve this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

NOTE

Make sure you run the application directly, rather than behind IIS Express, which ignores non-local requests by default. Run `dotnet run` from a command prompt, or choose the application name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields using the `[Required]` attribute:

```

using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        public string Notes { get; set; }

        public bool Done { get; set; }
    }
}

```

The API methods require some way to work with data. Use the same `IToDoRepository` interface the original Xamarin sample uses:

```

using System.Collections.Generic;
using ToDoApi.Models;

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}

```

For this sample, the implementation just uses a private collection of items:

```

using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _todoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _todoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _todoList.Any(item => item.ID == id);
        }
    }
}

```

```

    }

    public TodoItem Find(string id)
    {
        return _todoList.FirstOrDefault(item => item.ID == id);
    }

    public void Insert(TodoItem item)
    {
        _todoList.Add(item);
    }

    public void Update(TodoItem item)
    {
        var todoItem = this.Find(item.ID);
        var index = _todoList.IndexOf(todoItem);
        _todoList.RemoveAt(index);
        _todoList.Insert(index, item);
    }

    public void Delete(string id)
    {
        _todoList.Remove(this.Find(id));
    }

    private void InitializeData()
    {
        _todoList = new List<TodoItem>();

        var todoItem1 = new TodoItem
        {
            ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
            Name = "Learn app development",
            Notes = "Attend Xamarin University",
            Done = true
        };

        var todoItem2 = new TodoItem
        {
            ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
            Name = "Develop apps",
            Notes = "Use Xamarin Studio/Visual Studio",
            Done = false
        };

        var todoItem3 = new TodoItem
        {
            ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
            Name = "Publish apps",
            Notes = "All app stores",
            Done = false,
        };

        _todoList.Add(todoItem1);
        _todoList.Add(todoItem2);
        _todoList.Add(todoItem3);
    }
}
}
}

```

Configure the implementation in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<ITodoRepository, TodoRepository>();
}
```

At this point, you're ready to create the *ToDoItemsController*.

TIP

Learn more about creating web APIs in [Building Your First Web API with ASP.NET Core MVC and Visual Studio](#).

Creating the Controller

Add a new controller to the project, *ToDoItemsController*. It should inherit from `Microsoft.AspNetCore.Mvc.Controller`. Add a `Route` attribute to indicate that the controller will handle requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about [routing](#).

The controller requires an `ITodoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance will be provided using the framework's support for [dependency injection](#).

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly ITodoRepository _todoRepository;

        public ToDoItemsController(ITodoRepository todoRepository)
        {
            _todoRepository = todoRepository;
        }
    }
}
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP GET request.

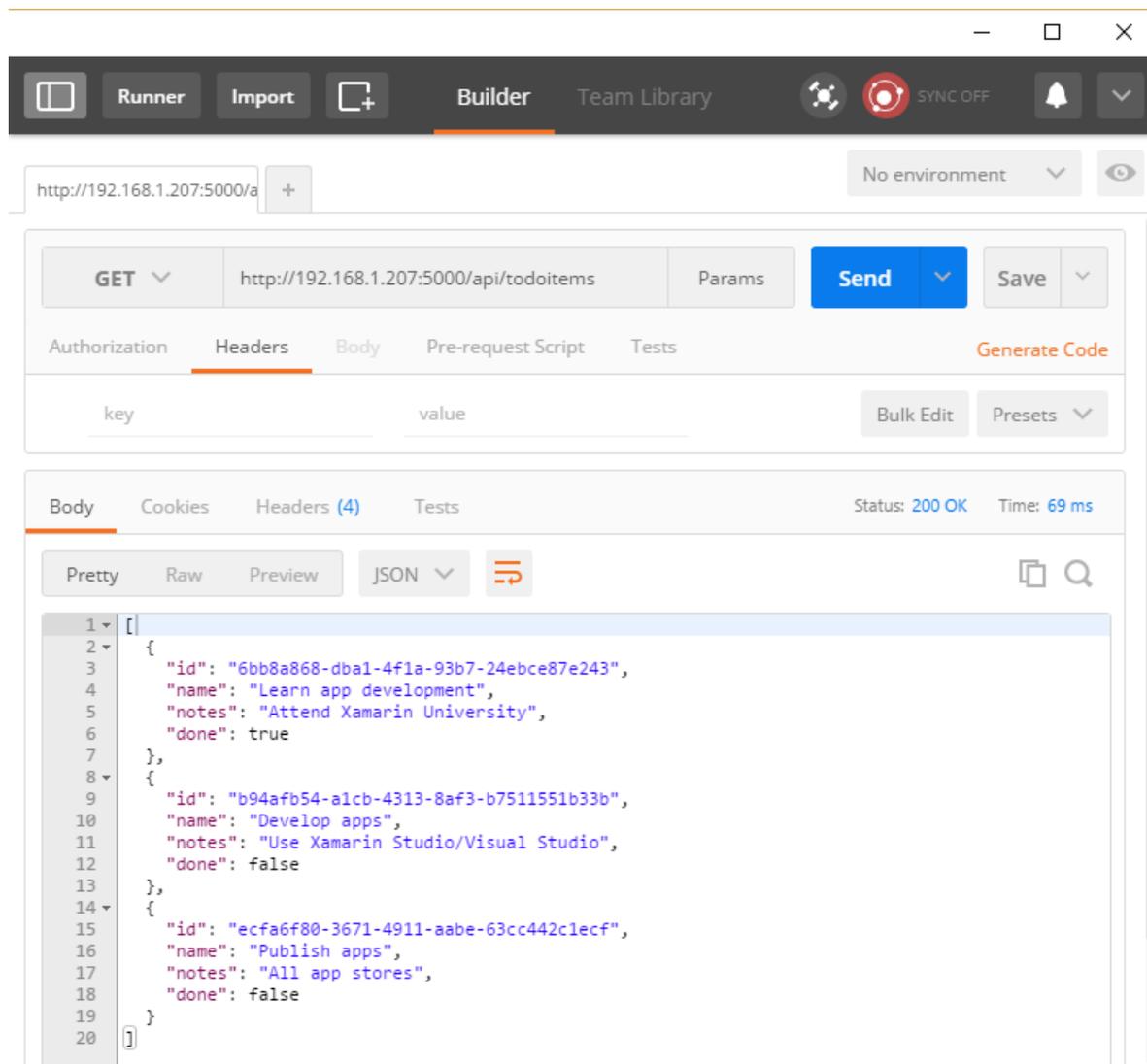
Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

```
[HttpGet]
public IActionResult List()
{
    return Ok(_todoRepository.All);
}
```

The `List` method returns a 200 OK response code and all of the `ToDo` items, serialized as JSON.

You can test your new API method using a variety of tools, such as [Postman](#), shown here:



Creating Items

By convention, creating new data items is mapped to the HTTP POST verb. The `Create` method has an `[HttpPost]` attribute applied to it, and accepts a `ToDoItem` instance. Since the `item` argument will be passed in the body of the POST, this parameter is decorated with the `[FromBody]` attribute.

Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it is added using the repository. Checking `ModelState.IsValid` performs [model validation](#), and should be done in every API method that accepts user input.

```

[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TODOItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _todoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TODOItemIDInUse.ToString());
        }
        _todoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}

```

The sample uses an enum containing error codes that are passed to the mobile client:

```

public enum ErrorCode
{
    TODOItemNameAndNotesRequired,
    TODOItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}

```

Test adding new items using Postman by choosing the POST verb providing the new object in JSON format in the Body of the request. You should also add a request header specifying a `Content-Type` of `application/json`.

The screenshot shows a REST client interface with a toolbar at the top containing buttons for 'Runner', 'Import', 'Builder', and 'Team Library'. A 'SYNC OFF' indicator is also present. Below the toolbar, a URL bar shows 'http://192.168.1.207:5000/a'. The main interface is divided into several sections:

- Request Section:** Shows a POST request to 'http://192.168.1.207:5000/api/todoitems'. The 'Body' tab is selected, showing a JSON payload:

```
{
  "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",
  "Name": "A Test Item",
  "Notes": "asdf",
  "Done": false
}
```
- Response Section:** Shows the response with a status of '200 OK' and a time of '227 ms'. The 'Body' tab is selected, showing the received JSON:

```
{
  "id": "6bb8b868-dba1-4f1a-93b7-24ebce87243",
  "name": "A Test Item",
  "notes": "asdf",
  "done": false
}
```

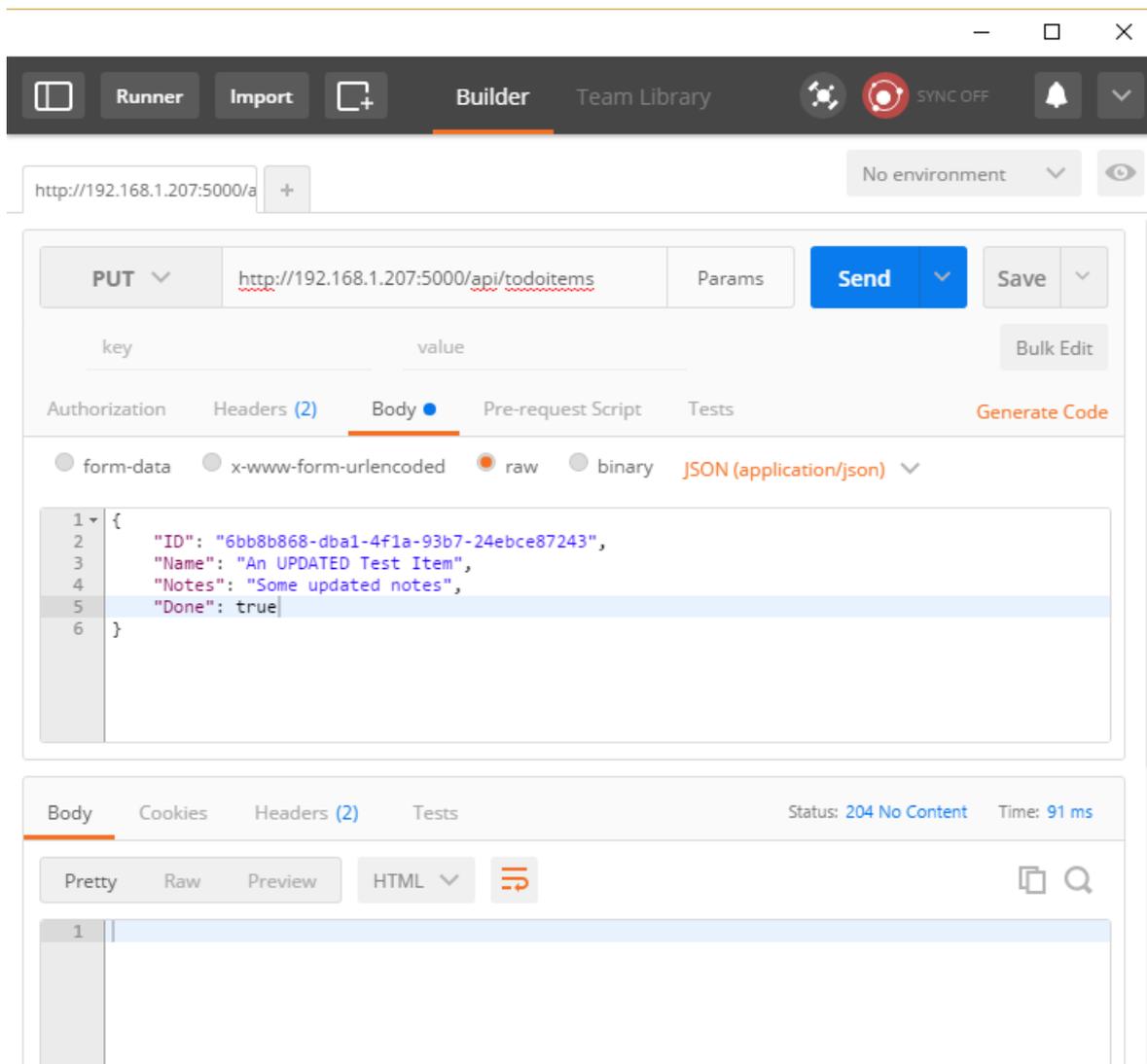
The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP PUT requests. Other than this change, the `Edit` method is almost identical to `Create`. Note that if the record isn't found, the `Edit` action will return a `NotFound` (404) response.

```
[HttpPut]
public IActionResult Edit([FromBody] TodoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TODOItemNameAndNotesRequired.ToString());
        }
        var existingItem = _todoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with Postman, change the verb to PUT. Specify the updated object data in the Body of the request.



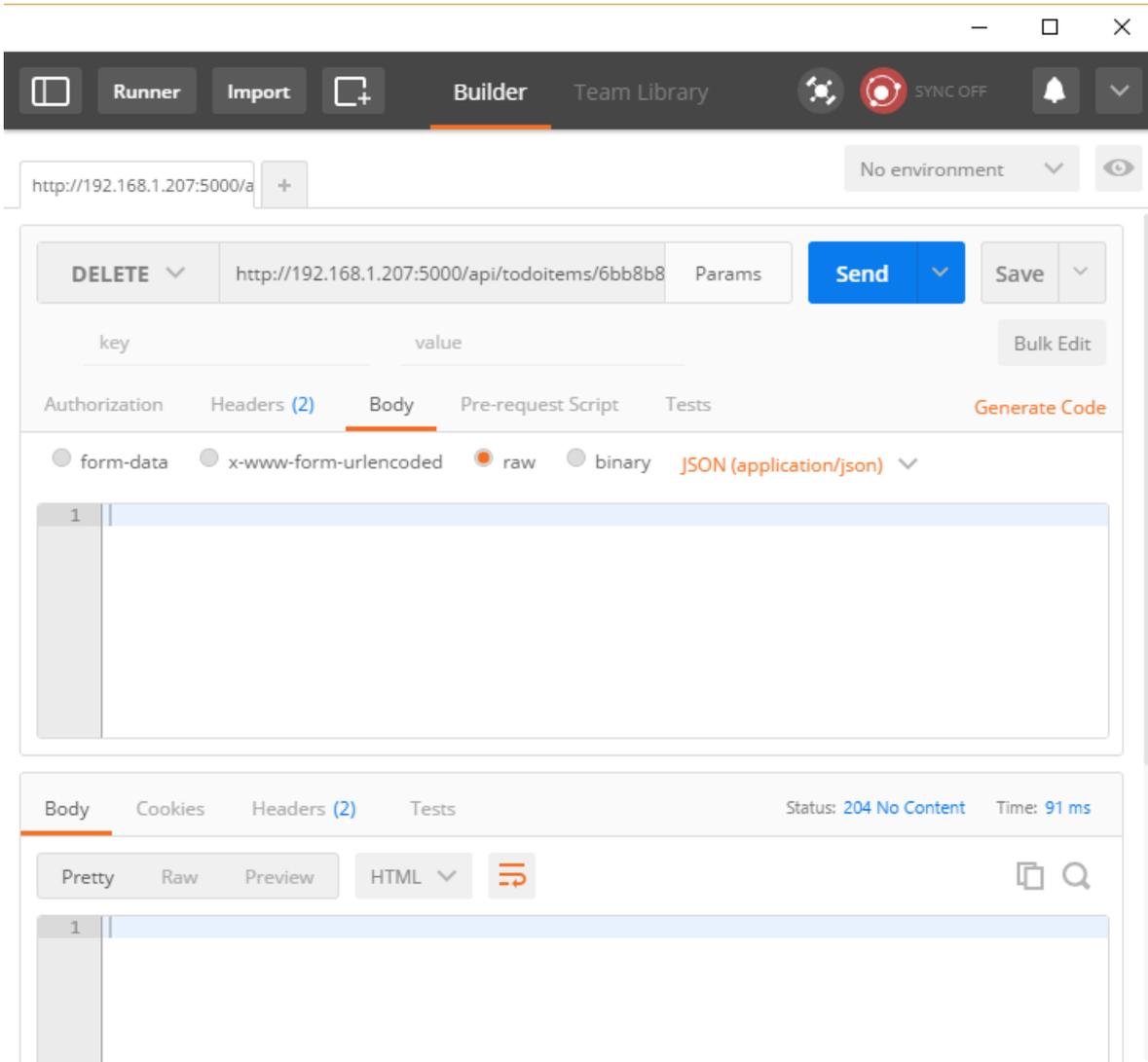
This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making DELETE requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist will receive `NotFound` responses. Otherwise, a successful request will get a `NoContent` (204) response.

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _todoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

Note that when testing the delete functionality, nothing is required in the Body of the request.



Common Web API Conventions

As you develop the backend services for your app, you will want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown above, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a [filter](#). Learn more about [how to encapsulate common API policies in ASP.NET Core MVC applications](#).

Building Web APIs

12/13/2017 • 1 min to read • [Edit Online](#)

- [Building your first Web API with ASP.NET Core using Visual Studio](#)
- [ASP.NET Core Web API Help Pages using Swagger](#)
- [Creating backend services for native mobile applications](#)
- [Formatting response data](#)
- [Custom formatters](#)

Create a Web API with ASP.NET Core MVC and Visual Studio for Mac

1/10/2018 • 8 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Mike Wasson](#)

In this tutorial, you'll build a web API for managing a list of "to-do" items. You won't build a UI.

There are 3 versions of this tutorial:

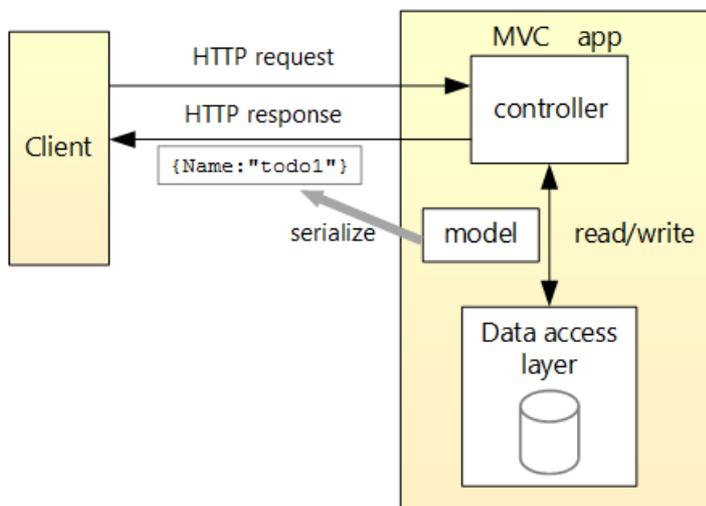
- macOS: Web API with Visual Studio for Mac (This tutorial)
- Windows: [Web API with Visual Studio for Windows](#)
- macOS, Linux, Windows: [Web API with Visual Studio Code](#)

Overview

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/todo	Get all to-do items	None	Array of to-do items
GET /api/todo/{id}	Get an item by ID	None	To-do item
POST /api/todo	Add a new item	To-do item	To-do item
PUT /api/todo/{id}	Update an existing item	To-do item	None
DELETE /api/todo/{id}	Delete an item	None	None

The following diagram shows the basic design of the app.



- The client is whatever consumes the web API (mobile app, browser, etc.). This tutorial doesn't create a client. [Postman](#) or [curl](#) is used as the client to test the app.

- A *model* is an object that represents the data in the app. In this case, the only model is a to-do item. Models are represented as C# classes, also known as **Plain Old C# Object** (POCOs).
- A *controller* is an object that handles HTTP requests and creates the HTTP response. This app has a single controller.
- To keep the tutorial simple, the app doesn't use a persistent database. The sample app stores to-do items in an in-memory database.
- See [Introduction to ASP.NET Core MVC on Mac or Linux](#) for an example that uses a persistent database.

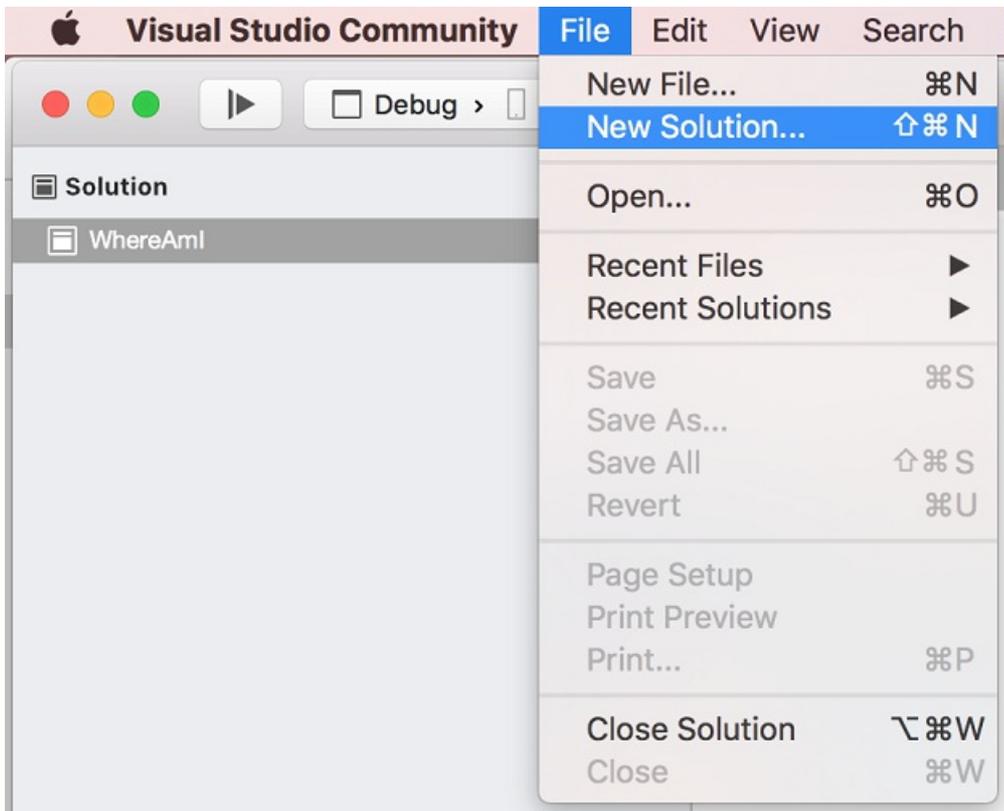
Prerequisites

Install the following:

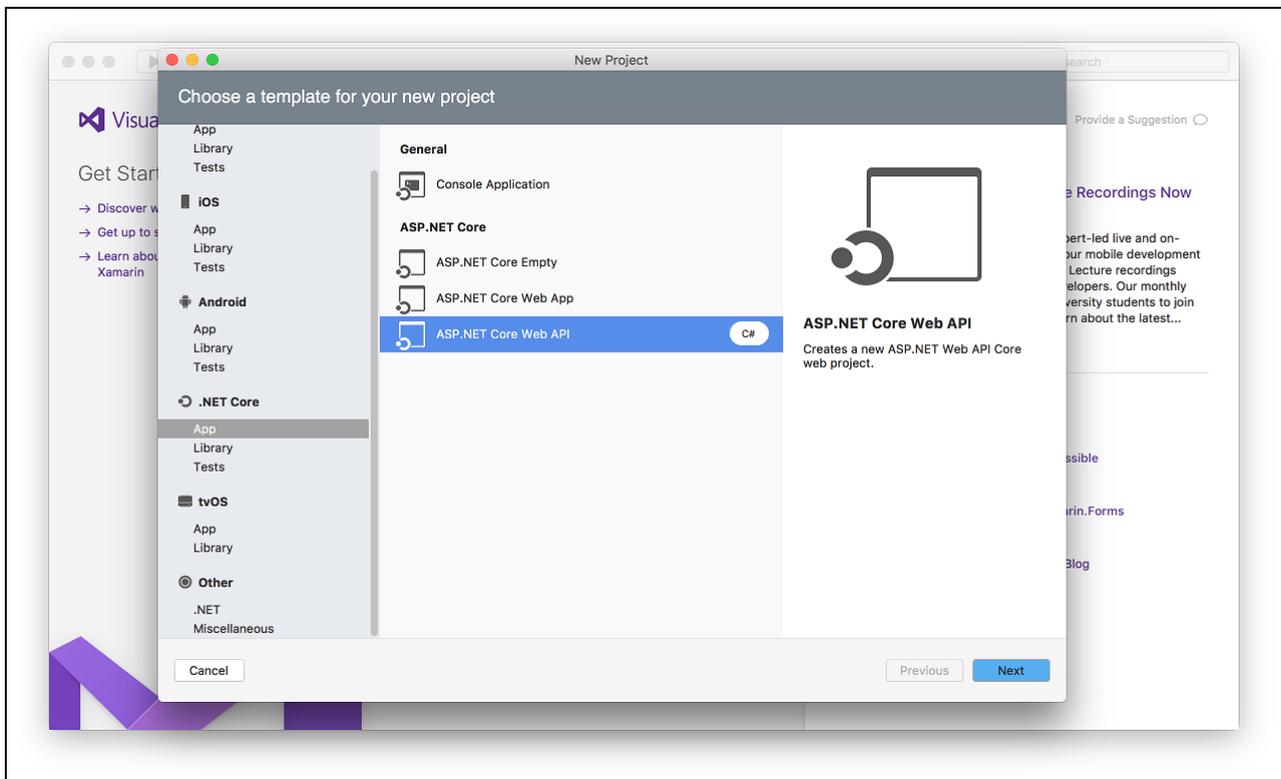
- [.NET Core 2.0.0 SDK](#) or later
- [Visual Studio for Mac](#)

Create the project

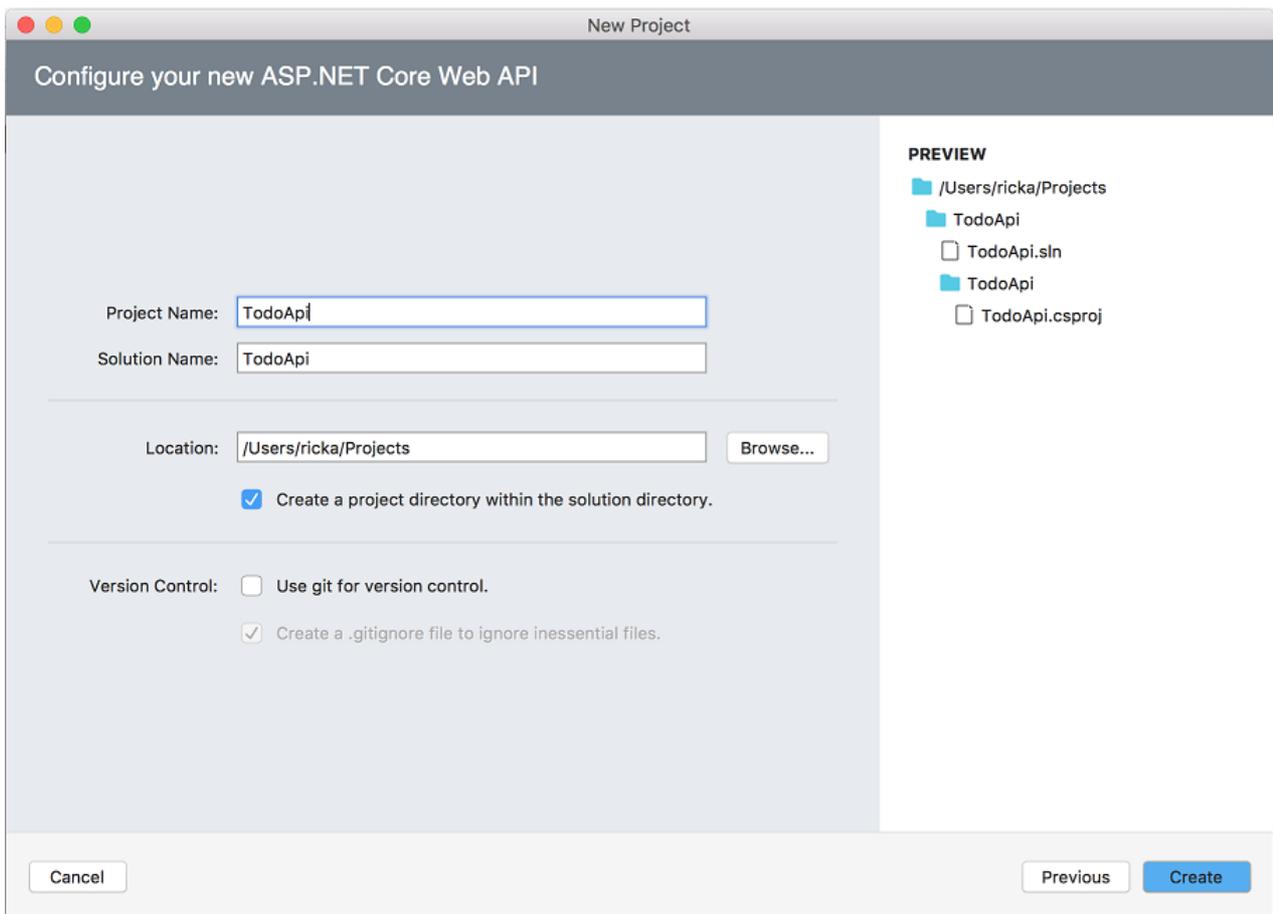
From Visual Studio, select **File > New Solution**.



Select **.NET Core App > ASP.NET Core Web API > Next**.



Enter **TodoApi** for the **Project Name**, and then select Create.



Launch the app

In Visual Studio, select **Run > Start With Debugging** to launch the app. Visual Studio launches a browser and navigates to `http://localhost:5000`. You get an HTTP 404 (Not Found) error. Change the URL to `http://localhost:port/api/values`. The `ValuesController` data will be displayed:

```
["value1","value2"]
```

Add support for Entity Framework Core

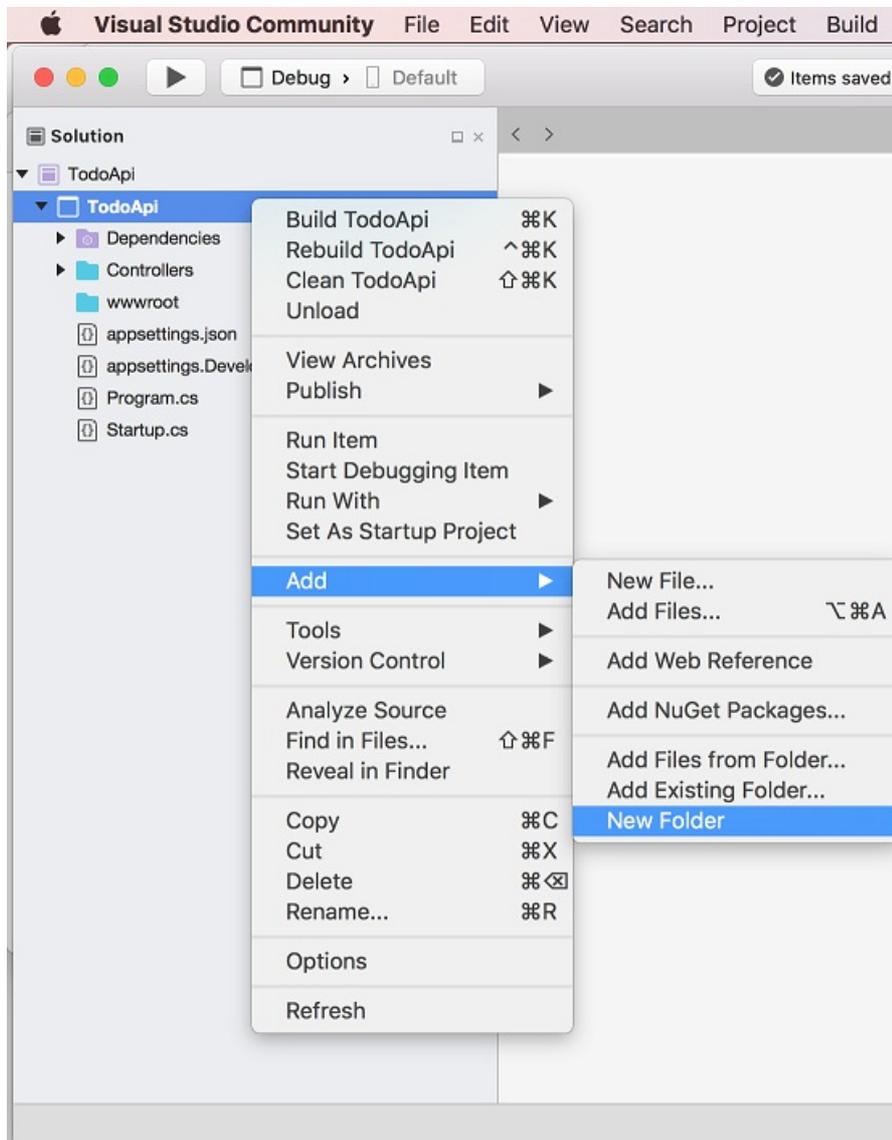
Install the [Entity Framework Core InMemory](#) database provider. This database provider allows Entity Framework Core to be used with an in-memory database.

- From the **Project** menu, select **Add NuGet Packages**.
 - Alternately, you can right-click **Dependencies**, and then select **Add Packages**.
- Enter `EntityFrameworkCore.InMemory` in the search box.
- Select `Microsoft.EntityFrameworkCore.InMemory`, and then select **Add Package**.

Add a model class

A model is an object that represents the data in your application. In this case, the only model is a to-do item.

Add a folder named *Models*. In Solution Explorer, right-click the project. Select **Add > New Folder**. Name the folder *Models*.



Note: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

Add a `TodoItem` class. Right-click the *Models* folder and select **Add > New File > General > Empty Class**. Name the class `TodoItem`, and then select **New**.

Replace the generated code with:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

The database generates the `Id` when a `TodoItem` is created.

Create the database context

The *database context* is the main class that coordinates Entity Framework functionality for a given data model. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

Add a `TodoContext` class to the *Models* folder.

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context

In this step, the database context is registered with the [dependency injection](#) container. Services (such as the DB context) that are registered with the dependency injection (DI) container are available to the controllers.

Register the DB context with the service container using the built-in support for [dependency injection](#). Replace the contents of the *Startup.cs* file with the following code:

```

using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}

```

The preceding code:

- Removes the code that is not used.
- Specifies an in-memory database is injected into the service container.

Add a controller

In Solution Explorer, in the *Controllers* folder, add the class `TodoController`.

Replace the generated code with the following (and add closing braces):

```

using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using TodoApi.Models;
using System.Linq;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TODOItems.Count() == 0)
            {
                _context.TODOItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}

```

The preceding code:

- Defines an empty controller class. In the next sections, methods are added to implement the API.
- The constructor uses [Dependency Injection](#) to inject the database context (`TodoContext`) into the controller. The

database context is used in each of the [CRUD](#) methods in the controller.

- The constructor adds an item to the in-memory database if one doesn't exist.

Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return _context.TODOItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

- `GET /api/todo`
- `GET /api/todo/{id}`

Here is an example HTTP response for the `GetAll` method:

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Later in the tutorial I'll show how the HTTP response can be viewed with [Postman](#) or [curl](#).

Routing and URL paths

The `[HttpGet]` attribute specifies an HTTP GET method. The URL path for each method is constructed as follows:

- Take the template string in the controller's `Route` attribute:

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
    }
}
```

- Replace `[controller]` with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoController** and the root name is "todo". ASP.NET Core [routing](#) is not case sensitive.
- If the `[HttpGet]` attribute has a route template (such as `[HttpGet("/products")]`), append that to the path. This sample doesn't use a template. See [Attribute routing with Http\[Verb\] attributes](#) for more information.

In the `GetById` method:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

`"{id}"` is a placeholder variable for the ID of the `todo` item. When `GetById` is invoked, it assigns the value of `{id}` in the URL to the method's `id` parameter.

`Name = "GetTodo"` creates a named route. Named routes:

- Enable the app to create an HTTP link using the route name.
- Are explained later in the tutorial.

Return values

The `GetAll` method returns an `IEnumerable`. MVC automatically serializes the object to [JSON](#) and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

In contrast, the `GetById` method returns the more general `IActionResult` type, which represents a wide range of return types. `GetById` has two different return types:

- If no item matches the requested ID, the method returns a 404 error. Returning `NotFound` returns an HTTP 404 response.
- Otherwise, the method returns 200 with a JSON response body. Returning `ObjectResult` returns an HTTP 200 response.

Launch the app

In Visual Studio, select **Run > Start With Debugging** to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port`, where *port* is a randomly chosen port number. You get an HTTP 404 (Not Found) error. Change the URL to `http://localhost:port/api/values`. The `valuesController` data will be displayed:

```
["value1", "value2"]
```

Navigate to the `Todo` controller at `http://localhost:port/api/todo`:

```
[{"key":1,"name":"Item1","isComplete":false}]
```

Implement the other CRUD operations

We'll add `Create`, `Update`, and `Delete` methods to the controller. These are variations on a theme, so I'll just show the code and highlight the main differences. Build the project after adding or changing code.

Create

```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

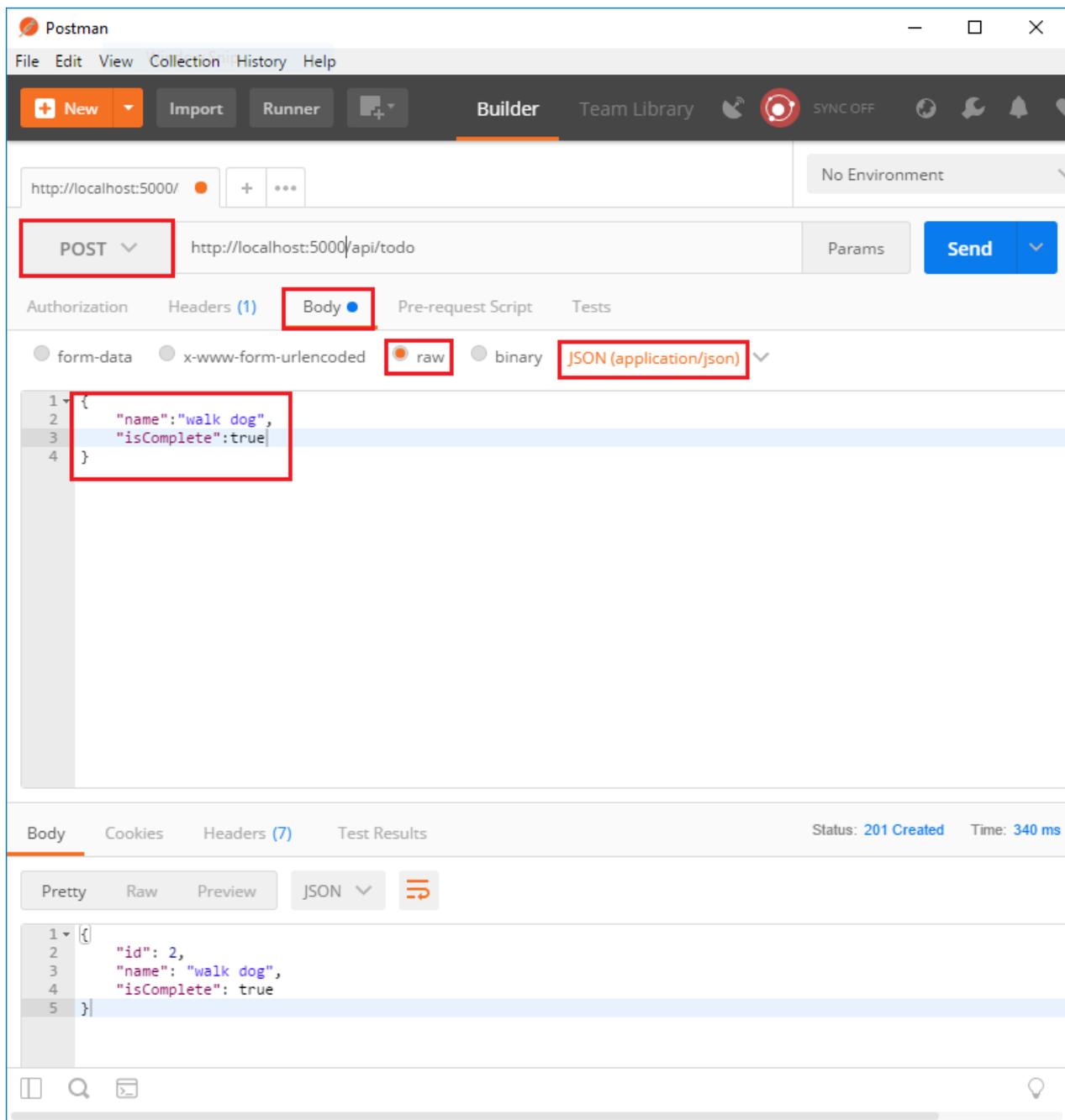
    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

This is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See [10.2.2 201 Created](#).

Use Postman to send a Create request

- Start the app (**Run > Start With Debugging**).
- Start Postman.



- Set the HTTP method to `POST`
- Select the **Body** radio button
- Select the **raw** radio button
- Set the type to JSON
- In the key-value editor, enter a Todo item such as

```
{
  "name": "walk dog",
  "isComplete": true
}
```

- Select **Send**
- Select the Headers tab in the lower pane and copy the **Location** header:

The screenshot shows a REST client interface with the following details:

- URL: `http://localhost:1234/`
- Method: `POST`
- Endpoint: `http://localhost:1234/api/todo`
- Body:

```
{
  "name": "walk dog",
  "isComplete": true
}
```
- Headers (7):
 - `Content-Type` → `application/json; charset=utf-8`
 - `Date` → `Sat, 04 Mar 2017 02:27:17 GMT`
 - `Location` → `http://localhost:1234/api/ToDo/2` (highlighted in red)
 - `Server` → `Kestrel`
 - `Transfer-Encoding` → `chunked`
 - `X-Powered-By` → `ASP.NET`
 - `X-SourceFiles` → `=?UTF-8?B?QzpcY3Nwcm9qTmV3XDRcRG9jc1xhc3BuZXRjb3JlXHR1dG9yaWFsc1xmaXJzdC13ZWltYXBpXHNhbXBsZVxUb2RvQXBpXGFwaVx0b2Rv?=</code>`
- Status: `201 Created`

You can use the Location header URI to access the resource you just created. Recall the `GetById` method created the `"GetTodo"` named route:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(string id)
```

Update

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] TodoItem item)
{
    if (item == null || item.Id != id)
    {
        return BadRequest();
    }

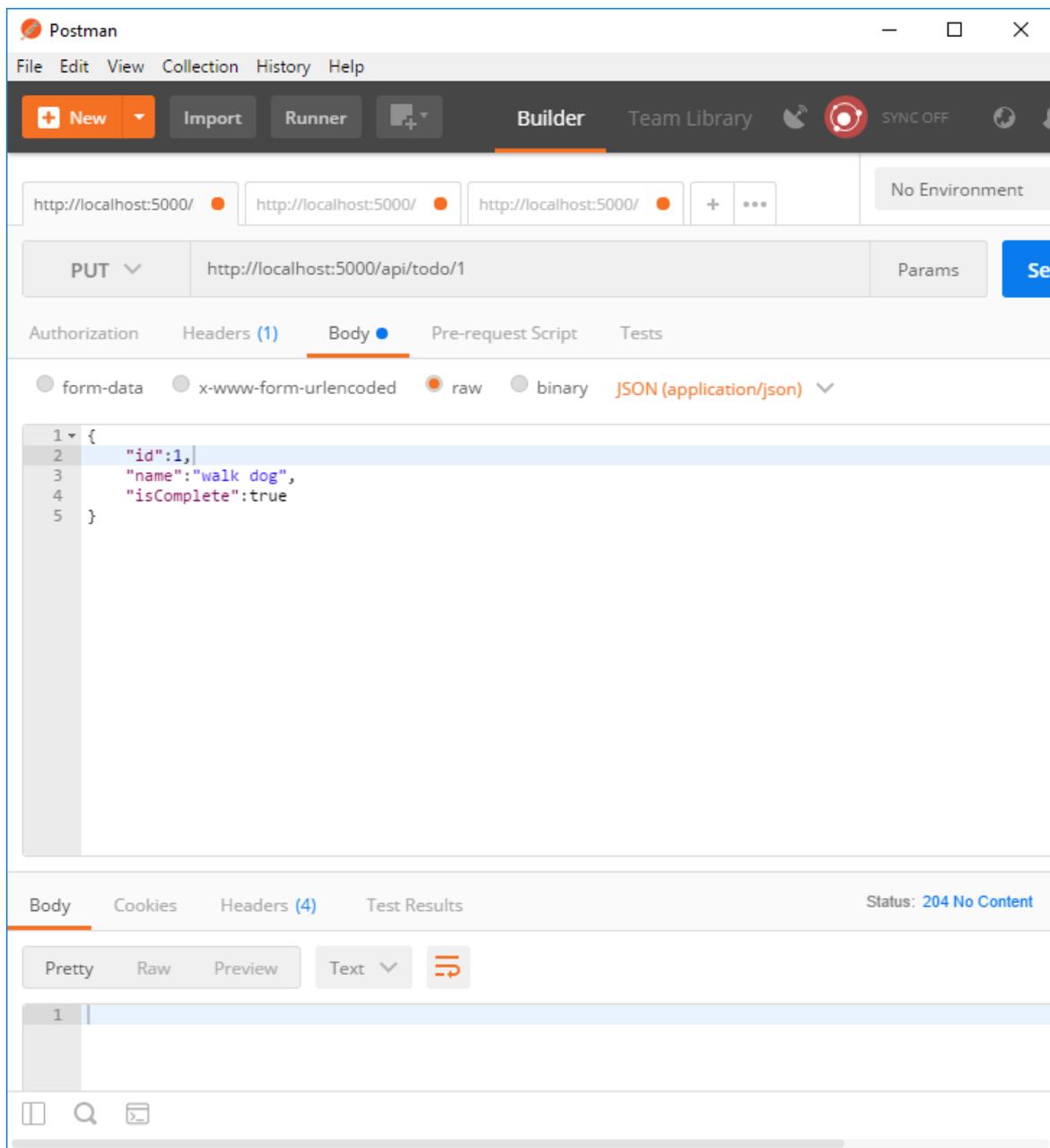
    var todo = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    todo.IsComplete = item.IsComplete;
    todo.Name = item.Name;

    _context.TODOItems.Update(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

`Update` is similar to `Create`, but uses HTTP PUT. The response is [204 \(No Content\)](#). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.

```
{
  "key": 1,
  "name": "walk dog",
  "isComplete": true
}
```

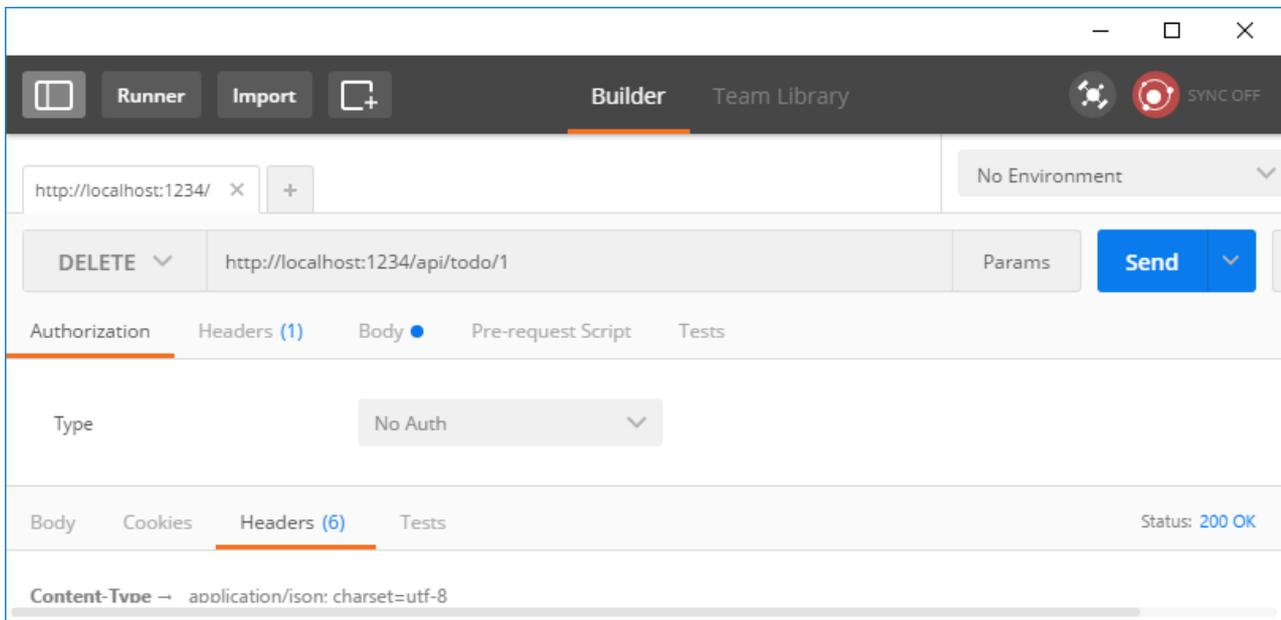


Delete

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TODOItems.Remove(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

The response is 204 (No Content).



Next steps

- [Routing to Controller Actions](#)
- For information about deploying your API, see [Host and deploy](#).
- [View or download sample code \(how to download\)](#)
- [Postman](#)
- [Fiddler](#)

ASP.NET Core Web API Help Pages using Swagger

1/8/2018 • 9 min to read • [Edit Online](#)

By [Shayne Boyer](#) and [Scott Addie](#)

Understanding the various methods of an API can be a challenge for a developer when building a consuming application.

Generating good documentation and help pages for your Web API, using [Swagger](#) with the .NET Core implementation [Swashbuckle.AspNetCore](#), is as easy as adding a couple of NuGet packages and modifying the *Startup.cs*.

- [Swashbuckle.AspNetCore](#) is an open source project for generating Swagger documents for ASP.NET Core Web APIs.
- [Swagger](#) is a machine-readable representation of a RESTful API that enables support for interactive documentation, client SDK generation, and discoverability.

This tutorial builds on the sample on [Building Your First Web API with ASP.NET Core MVC and Visual Studio](#). If you'd like to follow along, download the sample at <https://github.com/aspnet/Docs/tree/master/aspnetcore/tutorials/first-web-api/sample>.

Getting Started

There are three main components to Swashbuckle:

- `Swashbuckle.AspNetCore.Swagger`: a Swagger object model and middleware to expose `SwaggerDocument` objects as JSON endpoints.
- `Swashbuckle.AspNetCore.SwaggerGen`: a Swagger generator that builds `SwaggerDocument` objects directly from your routes, controllers, and models. It's typically combined with the Swagger endpoint middleware to automatically expose Swagger JSON.
- `Swashbuckle.AspNetCore.SwaggerUI`: an embedded version of the Swagger UI tool which interprets Swagger JSON to build a rich, customizable experience for describing the Web API functionality. It includes built-in test harnesses for the public methods.

NuGet Packages

Swashbuckle can be added with the following approaches:

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)
- From the **Package Manager Console** window:

```
Install-Package Swashbuckle.AspNetCore
```

- From the **Manage NuGet Packages** dialog:
 - Right-click your project in **Solution Explorer** > **Manage NuGet Packages**

- Set the **Package source** to "nuget.org"
- Enter "Swashbuckle.AspNetCore" in the search box
- Select the "Swashbuckle.AspNetCore" package from the **Browse** tab and click **Install**

Add and configure Swagger to the middleware

Add the Swagger generator to the services collection in the `ConfigureServices` method of `Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining one or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
    });
}
```

Add the following using statement for the `Info` class:

```
using Swashbuckle.AspNetCore.Swagger;
```

In the `Configure` method of `Startup.cs`, enable the middleware for serving the generated JSON document and the SwaggerUI:

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.), specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

Launch the app, and navigate to `http://localhost:<random_port>/swagger/v1/swagger.json`. The generated document describing the endpoints appears.

Note: Microsoft Edge, Google Chrome, and Firefox display JSON documents natively. There are extensions for Chrome that format the document for easier reading. *The following example is reduced for brevity.*

```

{
  "swagger": "2.0",
  "info": {
    "version": "v1",
    "title": "API V1"
  },
  "basePath": "/",
  "paths": {
    "/api/ToDo": {
      "get": {
        "tags": [
          "ToDo"
        ],
        "operationId": "ApiToDoGet",
        "consumes": [],
        "produces": [
          "text/plain",
          "application/json",
          "text/json"
        ],
        "responses": {
          "200": {
            "description": "Success",
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/ToDoItem"
              }
            }
          }
        }
      },
      "post": {
        ...
      }
    },
    "/api/ToDo/{id}": {
      "get": {
        ...
      },
      "put": {
        ...
      },
      "delete": {
        ...
      }
    },
    "definitions": {
      "ToDoItem": {
        "type": "object",
        "properties": {
          "id": {
            "format": "int64",
            "type": "integer"
          },
          "name": {
            "type": "string"
          },
          "isComplete": {
            "default": false,
            "type": "boolean"
          }
        }
      }
    }
  },
  "securityDefinitions": {}
}

```

This document drives the Swagger UI, which can be viewed by navigating to

`http://localhost:<random_port>/swagger :`

The header bar of the Swagger UI is a solid green horizontal bar. On the left side, there is a circular icon with a double arrow and the word "swagger" in white lowercase letters. In the center, there is a white text input field containing the URL "http://localhost:60201/swagger/v1/swagger.json". On the right side, there is a dark green button with the text "My API V1" and a small downward-pointing triangle icon.

My API

Todo

Show/Hide | List Operations | Expand Operations

GET	/api/ToDo
POST	/api/ToDo
DELETE	/api/ToDo/{id}
GET	/api/ToDo/{id}
PUT	/api/ToDo/{id}

[BASE URL: / , API VERSION: v1]

Each public action method in `TodoController` can be tested from the UI. Click a method name to expand the section. Add any necessary parameters, and click "Try it out!".

GET /api/ToDo

Response Class (Status 200)
Success

Model | Example Value

```
[
  {
    "id": 0,
    "name": "string",
    "isComplete": false
  }
]
```

Response Content Type [Hide Response](#)

Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:60201/api/ToDo'
```

Request URL

```
http://localhost:60201/api/ToDo
```

Response Body

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Response Code

```
200
```

Response Headers

```
{
  "date": "Thu, 31 Aug 2017 17:29:04 GMT",
  "server": "Kestrel",
  "transfer-encoding": "chunked",
  "content-type": "application/json; charset=utf-8"
}
```

Customization & Extensibility

Swagger provides options for documenting the object model and customizing the UI to match your theme.

API Info and Description

The configuration action passed to the `AddSwaggerGen` method can be used to add information such as the author, license, and description:

```
// Register the Swagger generator, defining one or more Swagger documents
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "A simple example ASP.NET Core Web API",
        TermsOfService = "None",
        Contact = new Contact { Name = "Shayne Boyer", Email = "", Url = "https://twitter.com/spboyer" },
        License = new License { Name = "Use under LICX", Url = "https://example.com/license" }
    });
    c.IncludeXmlComments(xmlPath);
});
```

The following image depicts the Swagger UI displaying the version information:

ToDo API

A simple example ASP.NET Core Web API

Created by Shayne Boyer
 See more at <https://twitter.com/spboyer>
[Use under LICX](#)

Todo

Show/Hide | List Operations | Expand Operations

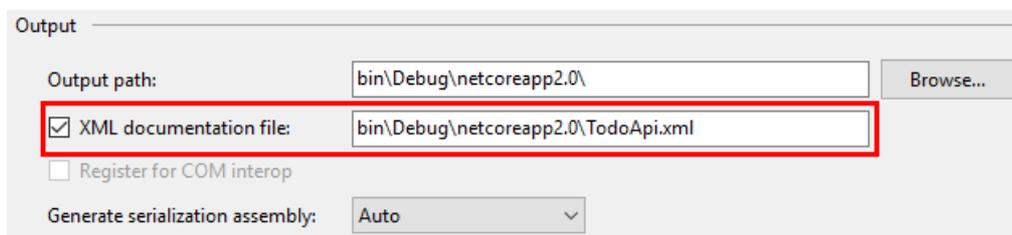
GET	/api/ToDo
POST	/api/ToDo
DELETE	/api/ToDo/{id}
GET	/api/ToDo/{id}
PUT	/api/ToDo/{id}

[BASE URL: / , API VERSION: v1]

XML Comments

XML comments can be enabled with the following approaches:

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)
- Right-click the project in **Solution Explorer** and select **Properties**
- Check the **XML documentation file** box under the **Output** section of the **Build** tab:



Enabling XML comments provides debug information for undocumented public types and members. Undocumented types and members are indicated by the warning message: *Missing XML comment for publicly visible type or member.*

Configure Swagger to use the generated XML file. For Linux or non-Windows operating systems, file names and paths can be case sensitive. For example, a *ToDoApi.XML* file would be found on Windows but not CentOS.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining one or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = "None",
            Contact = new Contact { Name = "Shayne Boyer", Email = "", Url = "https://twitter.com/spboyer" },
            License = new License { Name = "Use under LICX", Url = "https://example.com/license" }
        });

        // Set the comments path for the Swagger JSON and UI.
        var basePath = AppContext.BaseDirectory;
        var xmlPath = Path.Combine(basePath, "TodoApi.xml");
        c.IncludeXmlComments(xmlPath);
    });
}

```

In the preceding code, `ApplicationBasePath` gets the base path of the app. The base path is used to locate the XML comments file. `TodoApi.xml` only works for this example, since the name of the generated XML comments file is based on the application name.

Adding the triple-slash comments to the method enhances the Swagger UI by adding the description to the section header:

```

/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TODOItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TODOItems.Remove(todo);
    _context.SaveChanges();
    return new NoContentResult();
}

```

DELETE
/api/Todo/{id}
Deletes a specific TodoItem.

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text" value="(required)"/>		path	long

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

The UI is driven by the generated JSON file, which also contains these comments:

```

"delete": {
  "tags": [
    "Todo"
  ],
  "summary": "Deletes a specific TodoItem.",
  "operationId": "ApiTodoByIdDelete",
  "consumes": [],
  "produces": [],
  "parameters": [
    {
      "name": "id",
      "in": "path",
      "description": "",
      "required": true,
      "type": "integer",
      "format": "int64"
    }
  ],
  "responses": {
    "200": {
      "description": "Success"
    }
  }
}

```

Add a tag to the `Create` action method documentation. It supplements information specified in the `<summary>` tag and provides a more robust Swagger UI. The `<remarks>` tag content can consist of text, JSON, or XML.

```

/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///       "id": 1,
///       "name": "Item1",
///       "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly-created TodoItem</returns>
/// <response code="201">Returns the newly-created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), 201)]
[ProducesResponseType(typeof(TodoItem), 400)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TODOItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}

```

Notice the UI enhancements with these additional comments.

POST /api/ToDo Creates a TodoItem.

Implementation Notes

Sample request:

```
POST /ToDo
{
  "id": 1,
  "name": "Item1",
  "isComplete": true
}
```

Response Class (Status 201)
Success

Data Annotations

Decorate the model with attributes, found in `System.ComponentModel.DataAnnotations`, to help drive the Swagger UI components.

Add the `[Required]` attribute to the `Name` property of the `TodoItem` class:

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }

        [Required]
        public string Name { get; set; }

        [DefaultValue(false)]
        public bool IsComplete { get; set; }
    }
}
```

The presence of this attribute changes the UI behavior and alters the underlying JSON schema:

```
"definitions": {
  "TodoItem": {
    "required": [
      "name"
    ],
    "type": "object",
    "properties": {
      "id": {
        "format": "int64",
        "type": "integer"
      },
      "name": {
        "type": "string"
      },
      "isComplete": {
        "default": false,
        "type": "boolean"
      }
    }
  }
},
```

Add the `[Produces("application/json")]` attribute to the API controller. Its purpose is to declare that the

controller's actions support a return a content type of *application/json*:

```
namespace TodoApi.Controllers
{
    [Produces("application/json")]
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
```

The **Response Content Type** drop-down selects this content type as the default for the controller's GET actions:

Swagger UI for the **Todo** API endpoint `GET /api/Todo`. The response is a **Success** (Status 200) with a **Model** of **Example Value**. The response body is a JSON object:

```
{
  "id": 0,
  "name": "string",
  "isComplete": false
}
```

The **Response Content Type** dropdown is set to `application/json`. A **Try it out!** button is available below the dropdown.

As the usage of data annotations in the Web API increases, the UI and API help pages become more descriptive and useful.

Describing Response Types

Consuming developers are most concerned with what is returned — specifically response types and error codes (if not standard). These are handled in the XML comments and data annotations.

The `Create` action returns `201 Created` on success or `400 Bad Request` when the posted request body is null. Without proper documentation in the Swagger UI, the consumer lacks knowledge of these expected outcomes. That problem is fixed by adding the highlighted lines in the following example:

```

/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///         "id": 1,
///         "name": "Item1",
///         "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly-created TodoItem</returns>
/// <response code="201">Returns the newly-created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), 201)]
[ProducesResponseType(typeof(TodoItem), 400)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TODOItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}

```

The Swagger UI now clearly documents the expected HTTP response codes:

Response Class (Status 201)
Returns the newly-created item

Model | Example Value

```
{
  "id": 0,
  "name": "string",
  "isComplete": false
}
```

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
item	<input type="text"/>		body	Model Example Value

Parameter content type:

```
{
  "id": 0,
  "name": "string",
  "iscomplete": false
}
```

Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	If the item is null	Model Example Value	

Customizing the UI

The stock UI is both functional and presentable; however, when building documentation pages for your API, you want it to represent your brand or theme. Accomplishing that task with the Swashbuckle components requires adding the resources to serve static files and then building the folder structure to host those files.

If targeting .NET Framework, add the `Microsoft.AspNetCore.StaticFiles` NuGet package to the project:

```
<PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
```

Enable the static files middleware:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.), specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

Acquire the contents of the `dist` folder from the [Swagger UI GitHub repository](#). This folder contains the necessary assets for the Swagger UI page.

Create a `wwwroot/swagger/ui` folder, and copy into it the contents of the `dist` folder.

Create a `wwwroot/swagger/ui/css/custom.css` file with the following CSS to customize the page header:

```
.swagger-section #header
{
  border-bottom: 1px solid #000000;
  font-style: normal;
  font-weight: 400;
  font-family: "Segoe UI Light", "Segoe WP Light", "Segoe UI", "Segoe WP", Tahoma, Arial, sans-serif;
  background-color: black;
}

.swagger-section #header h1
{
  text-align: center;
  font-size: 20px;
  color: white;
}
```

Reference `custom.css` in the `index.html` file:

```
<link href='css/custom.css' media='screen' rel='stylesheet' type='text/css' />
```

Browse to the `index.html` page at `http://localhost:<random_port>/swagger/ui/index.html`. Enter

`http://localhost:<random_port>/swagger/v1/swagger.json` in the header's textbox, and click the **Explore** button. The resulting page looks as follows:

The screenshot shows the Swagger UI interface. At the top, there is a Swagger logo on the left, a text input field containing the URL `http://localhost:60201/swagger/v1/swagger.json`, and an **Explore** button on the right. Below this is the title **ToDo API** and a subtitle "A simple example ASP.NET Core Web API". It also includes the creator's name "Created by Shayne Boyer" and a link to his Twitter profile. The main section is titled **Todo** and lists several API endpoints with their methods and descriptions:

- GET** `/api/ToDo`
- POST** `/api/ToDo` - Creates a `TodoItem`.
- DELETE** `/api/ToDo/{id}` - Deletes a specific `TodoItem`.
- GET** `/api/ToDo/{id}`
- PUT** `/api/ToDo/{id}`

At the bottom, it shows the base URL and API version: `[BASE URL: / , API VERSION: v1]`.

There is much more you can do with the page. See the full capabilities for the UI resources at the [Swagger UI GitHub repository](#).

Creating Backend Services for Native Mobile Applications

9/30/2017 • 8 min to read • [Edit Online](#)

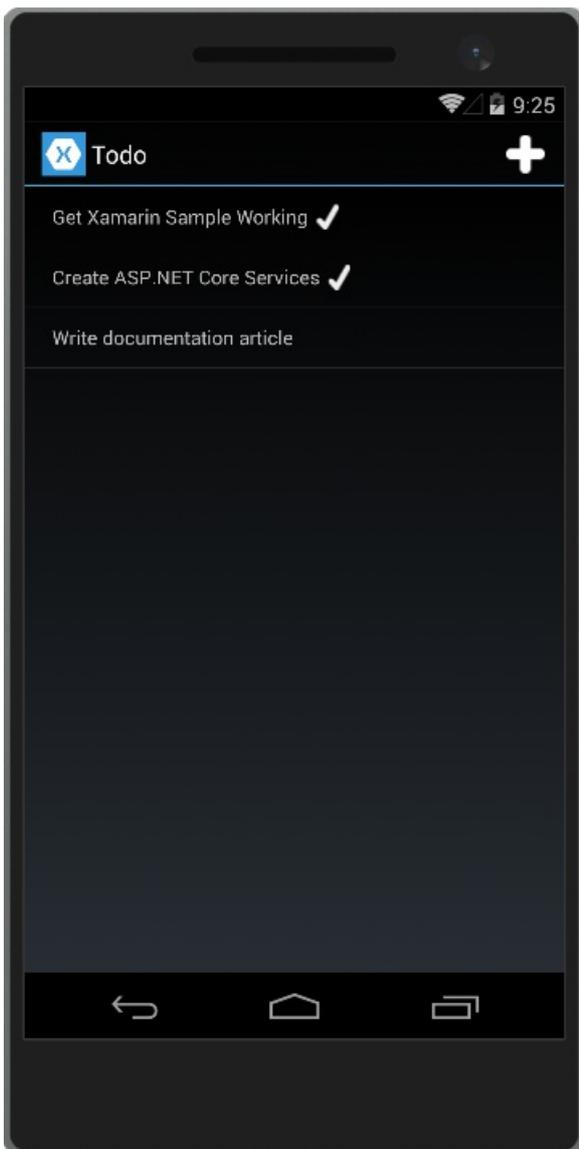
By [Steve Smith](#)

Mobile apps can easily communicate with ASP.NET Core backend services.

[View or download sample backend services code](#)

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core MVC to support native mobile apps. It uses the [Xamarin Forms ToDoRest app](#) as its native client, which includes separate native clients for Android, iOS, Windows Universal, and Window Phone devices. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), as well as download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Web API 2 services project, which this article's ASP.NET Core app replaces (with no changes required by the client).

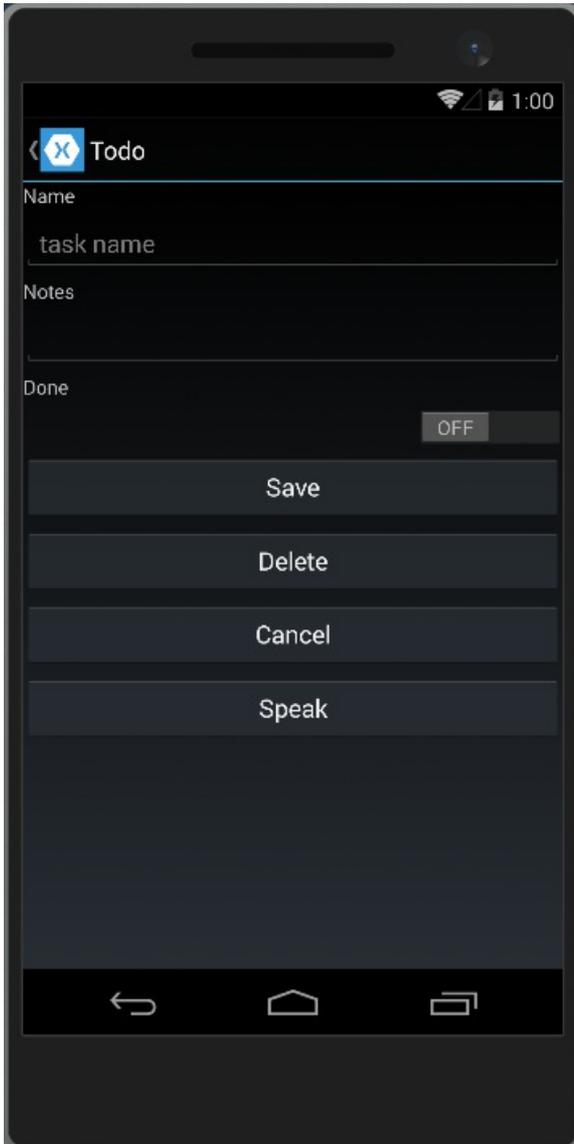


Features

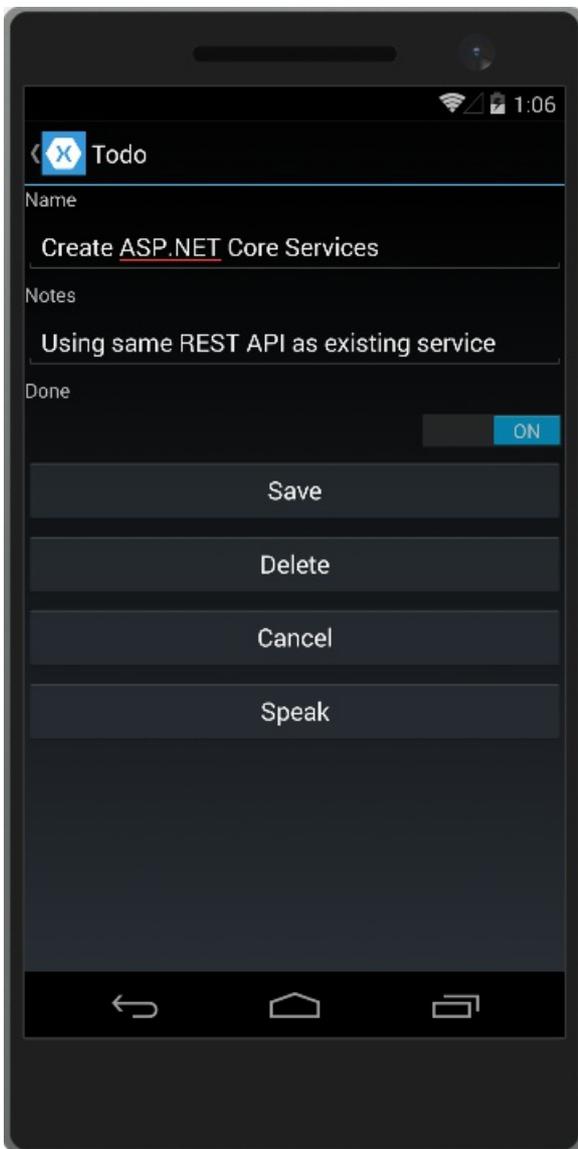
The ToDoRest app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

The main view of the items, as shown above, lists each item's name and indicates if it is done with a checkmark.

Tapping the  icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:



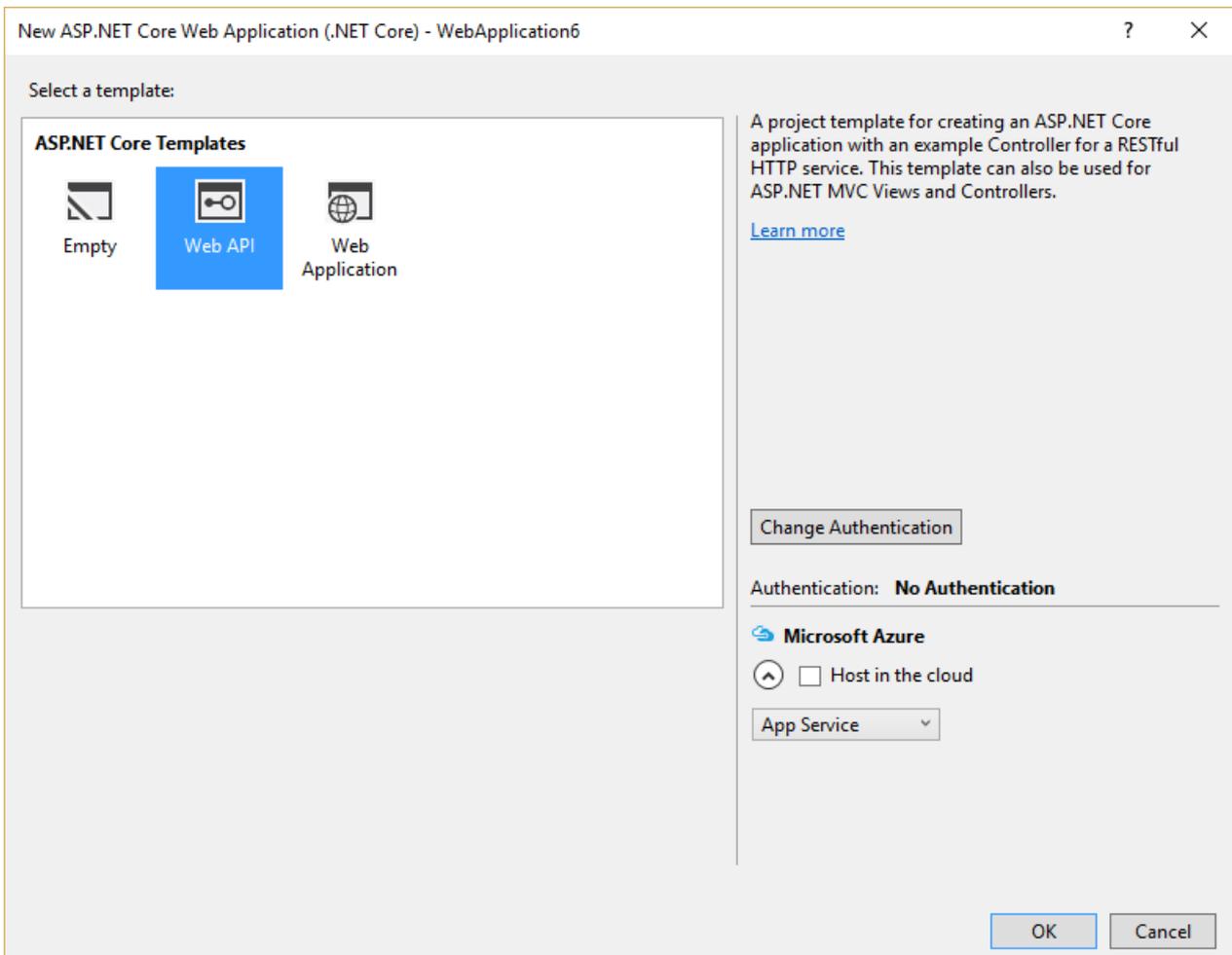
This sample is configured by default to use backend services hosted at `developer.xamarin.com`, which allow read-only operations. To test it out yourself against the ASP.NET Core app created in the next section running on your computer, you'll need to update the app's `RestUrl` constant. Navigate to the `ToDoREST` project and open the `Constants.cs` file. Replace the `RestUrl` with a URL that includes your machine's IP address (not localhost or 127.0.0.1, since this address is used from the device emulator, not from your machine). Include the port number as well (5000). In order to test that your services work with a device, ensure you don't have an active firewall blocking access to this port.

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template and No Authentication. Name the project *ToDoApi*.



The application should respond to all requests made to port 5000. Update *Program.cs* to include

`.UseUrls("http://*:5000")` to achieve this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

NOTE

Make sure you run the application directly, rather than behind IIS Express, which ignores non-local requests by default. Run `dotnet run` from a command prompt, or choose the application name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields using the `[Required]` attribute:

```

using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        public string Notes { get; set; }

        public bool Done { get; set; }
    }
}

```

The API methods require some way to work with data. Use the same `IToDoRepository` interface the original Xamarin sample uses:

```

using System.Collections.Generic;
using ToDoApi.Models;

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}

```

For this sample, the implementation just uses a private collection of items:

```

using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _todoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _todoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _todoList.Any(item => item.ID == id);
        }
    }
}

```

```

    }

    public TodoItem Find(string id)
    {
        return _todoList.FirstOrDefault(item => item.ID == id);
    }

    public void Insert(TodoItem item)
    {
        _todoList.Add(item);
    }

    public void Update(TodoItem item)
    {
        var todoItem = this.Find(item.ID);
        var index = _todoList.IndexOf(todoItem);
        _todoList.RemoveAt(index);
        _todoList.Insert(index, item);
    }

    public void Delete(string id)
    {
        _todoList.Remove(this.Find(id));
    }

    private void InitializeData()
    {
        _todoList = new List<TodoItem>();

        var todoItem1 = new TodoItem
        {
            ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
            Name = "Learn app development",
            Notes = "Attend Xamarin University",
            Done = true
        };

        var todoItem2 = new TodoItem
        {
            ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
            Name = "Develop apps",
            Notes = "Use Xamarin Studio/Visual Studio",
            Done = false
        };

        var todoItem3 = new TodoItem
        {
            ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
            Name = "Publish apps",
            Notes = "All app stores",
            Done = false,
        };

        _todoList.Add(todoItem1);
        _todoList.Add(todoItem2);
        _todoList.Add(todoItem3);
    }
}
}
}

```

Configure the implementation in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<ITodoRepository, TodoRepository>();
}
```

At this point, you're ready to create the *ToDoItemsController*.

TIP

Learn more about creating web APIs in [Building Your First Web API with ASP.NET Core MVC and Visual Studio](#).

Creating the Controller

Add a new controller to the project, *ToDoItemsController*. It should inherit from `Microsoft.AspNetCore.Mvc.Controller`. Add a `Route` attribute to indicate that the controller will handle requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about [routing](#).

The controller requires an `ITodoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance will be provided using the framework's support for [dependency injection](#).

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly ITodoRepository _todoRepository;

        public ToDoItemsController(ITodoRepository todoRepository)
        {
            _todoRepository = todoRepository;
        }
    }
}
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP GET request.

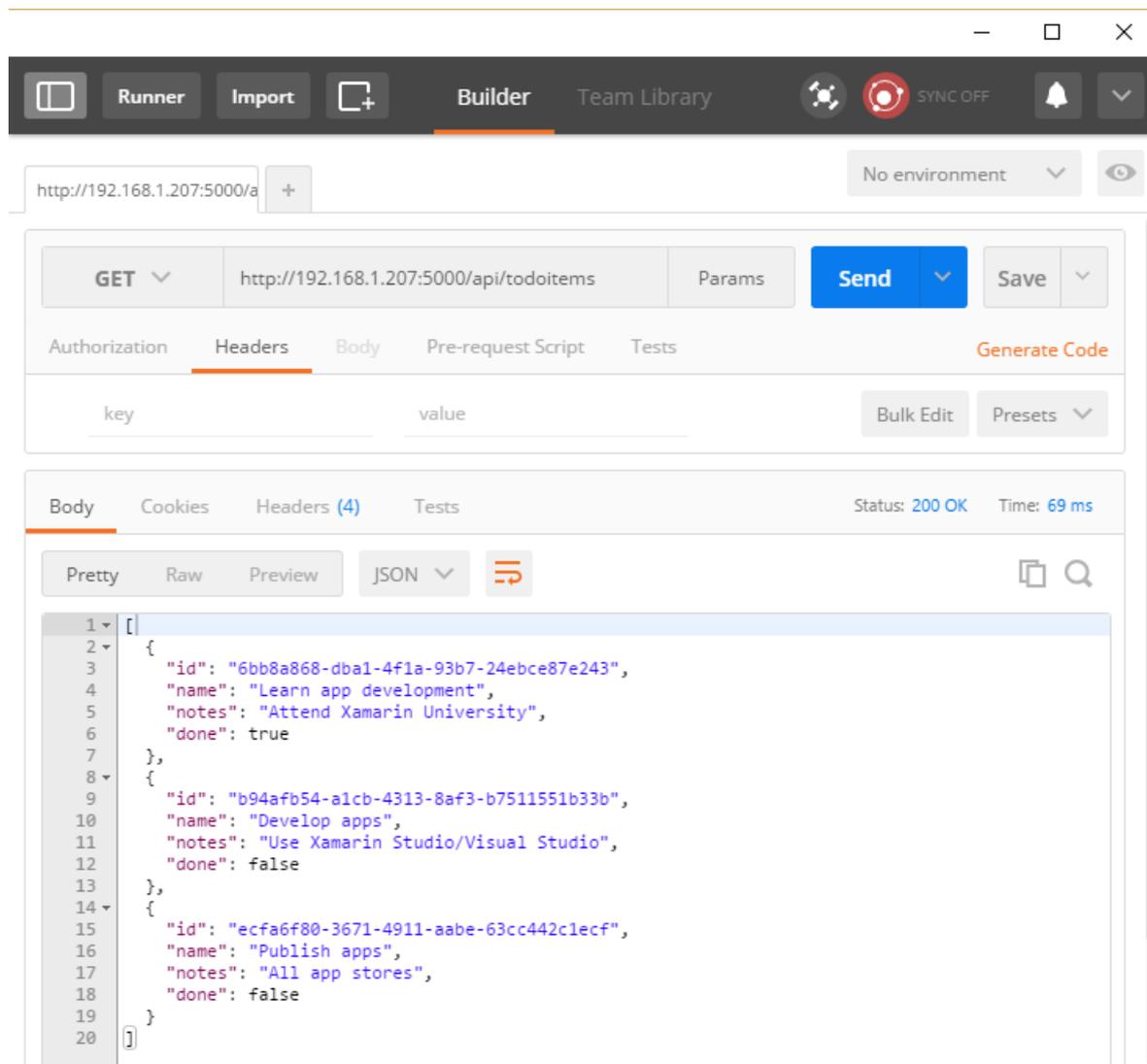
Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

```
[HttpGet]
public IActionResult List()
{
    return Ok(_todoRepository.All);
}
```

The `List` method returns a 200 OK response code and all of the `ToDo` items, serialized as JSON.

You can test your new API method using a variety of tools, such as [Postman](#), shown here:



Creating Items

By convention, creating new data items is mapped to the HTTP POST verb. The `Create` method has an `[HttpPost]` attribute applied to it, and accepts a `ToDoItem` instance. Since the `item` argument will be passed in the body of the POST, this parameter is decorated with the `[FromBody]` attribute.

Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it is added using the repository. Checking `ModelState.IsValid` performs [model validation](#), and should be done in every API method that accepts user input.

```

[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TODOItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _todoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TODOItemIDInUse.ToString());
        }
        _todoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}

```

The sample uses an enum containing error codes that are passed to the mobile client:

```

public enum ErrorCode
{
    TODOItemNameAndNotesRequired,
    TODOItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}

```

Test adding new items using Postman by choosing the POST verb providing the new object in JSON format in the Body of the request. You should also add a request header specifying a `Content-Type` of `application/json`.

The screenshot shows a REST client interface with a toolbar at the top containing buttons for Runner, Import, Builder, and Team Library. A URL bar shows `http://192.168.1.207:5000/a`. The main area is set to a POST request to `http://192.168.1.207:5000/api/todoitems`. The request body is a JSON object: `{ "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243", "Name": "A Test Item", "Notes": "asdf", "Done": false }`. Below the request, the response is shown with a status of 200 OK and a time of 227 ms. The response body is a JSON object: `{ "id": "6bb8b868-dba1-4f1a-93b7-24ebce87243", "name": "A Test Item", "notes": "asdf", "done": false }`.

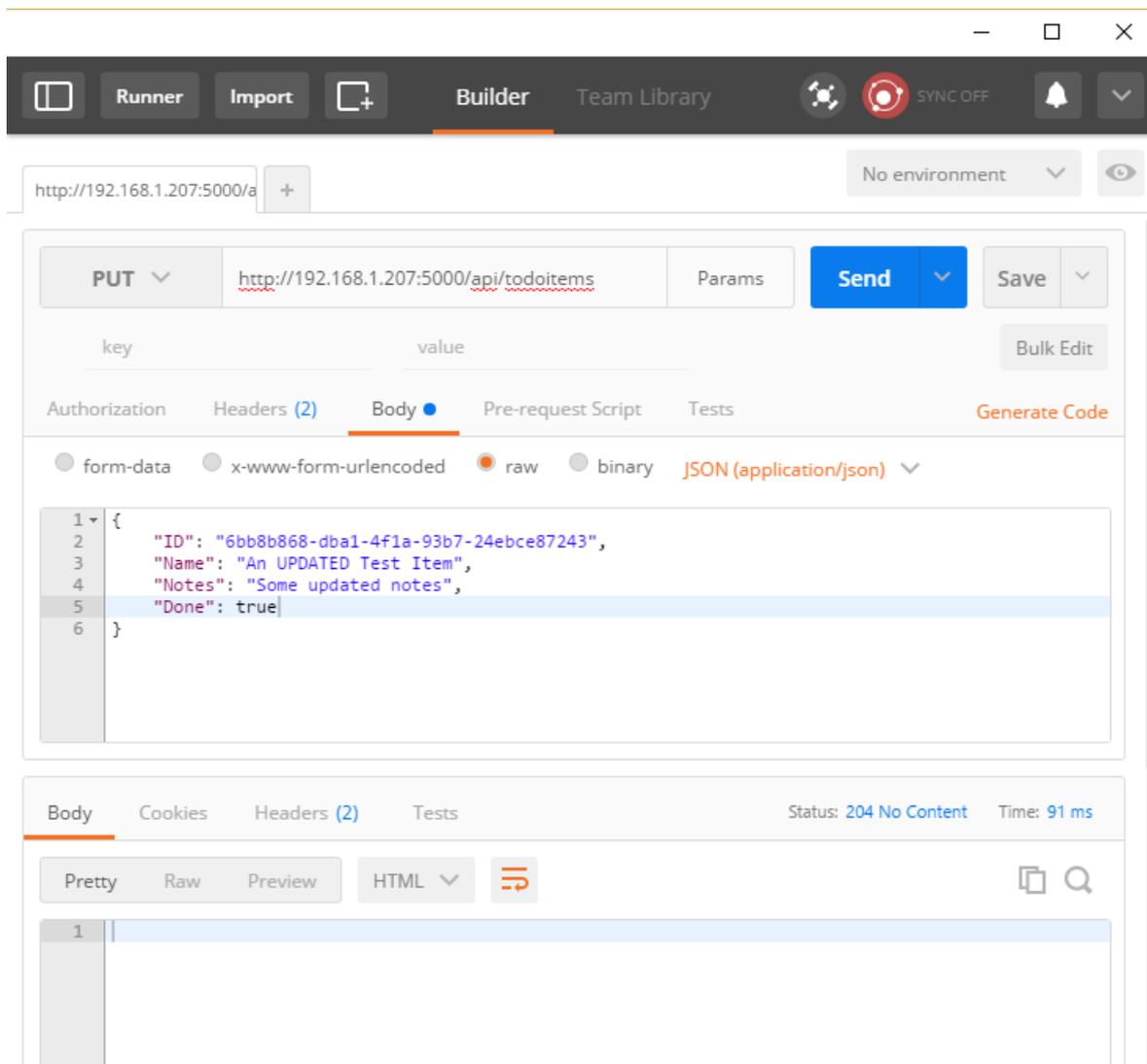
The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP PUT requests. Other than this change, the `Edit` method is almost identical to `Create`. Note that if the record isn't found, the `Edit` action will return a `NotFound` (404) response.

```
[HttpPut]
public IActionResult Edit([FromBody] TodoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TODOItemNameAndNotesRequired.ToString());
        }
        var existingItem = _todoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with Postman, change the verb to PUT. Specify the updated object data in the Body of the request.



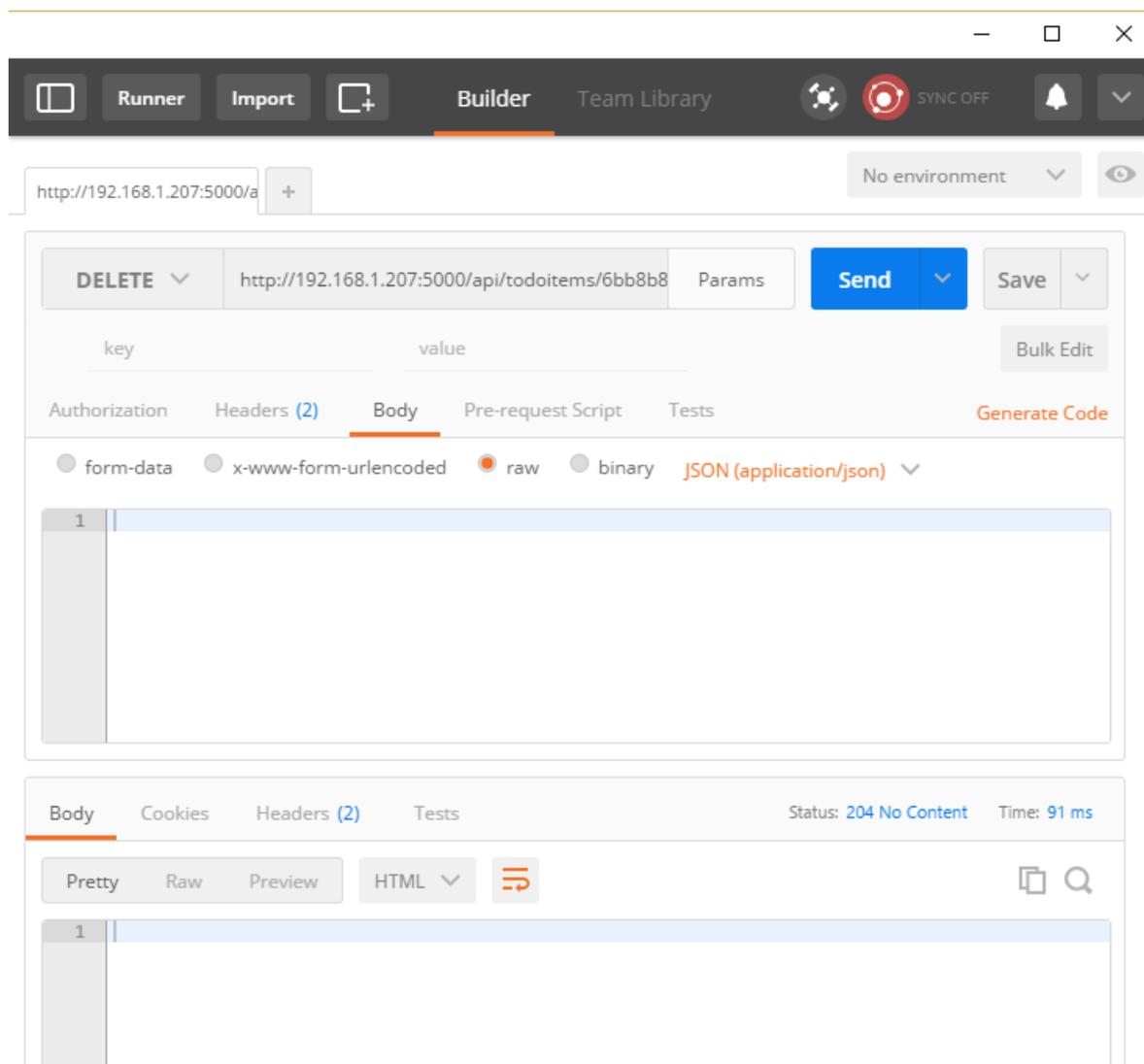
This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making DELETE requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist will receive `NotFound` responses. Otherwise, a successful request will get a `NoContent` (204) response.

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _todoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

Note that when testing the delete functionality, nothing is required in the Body of the request.



Common Web API Conventions

As you develop the backend services for your app, you will want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown above, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a [filter](#). Learn more about [how to encapsulate common API policies in ASP.NET Core MVC applications](#).

ASP.NET Core fundamentals

11/29/2017 • 6 min to read • [Edit Online](#)

An ASP.NET Core application is a console app that creates a web server in its `Main` method:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace aspnetcoreapp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The `Main` method invokes `WebHost.CreateDefaultBuilder`, which follows the builder pattern to create a web application host. The builder has methods that define the web server (for example, `UseKestrel`) and the startup class (`UseStartup`). In the preceding example, the [Kestrel](#) web server is automatically allocated. ASP.NET Core's web host attempts to run on IIS, if available. Other web servers, such as [HTTP.sys](#), can be used by invoking the appropriate extension method. `UseStartup` is explained further in the next section.

`IWebHostBuilder`, the return type of the `WebHost.CreateDefaultBuilder` invocation, provides many optional methods. Some of these methods include `UseHttpSys` for hosting the app in HTTP.sys and `UseContentRoot` for specifying the root content directory. The `Build` and `Run` methods build the `IWebHost` object that hosts the app and begins listening for HTTP requests.

Startup

The `UseStartup` method on `WebHostBuilder` specifies the `Startup` class for your app:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}

```

The `Startup` class is where you define the request handling pipeline and where any services needed by the app are configured. The `Startup` class must be public and contain the following methods:

```

public class Startup
{
    // This method gets called by the runtime. Use this method
    // to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method
    // to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
    }
}

```

`ConfigureServices` defines the [Services](#) used by your app (for example, ASP.NET Core MVC, Entity Framework Core, Identity). `Configure` defines the [middleware](#) for the request pipeline.

For more information, see [Application startup](#).

Content root

The content root is the base path to any content used by the app, such as views, [Razor Pages](#), and static assets. By default, the content root is the same as application base path for the executable hosting the app.

Web root

The web root of an app is the directory in the project containing public, static resources, such as CSS, JavaScript, and image files.

Dependency Injection (Services)

A service is a component that's intended for common consumption in an app. Services are made available through [dependency injection \(DI\)](#). ASP.NET Core includes a native **Inversion of Control (IoC)** container that supports [constructor injection](#) by default. You can replace the default native container if you wish. In addition to its loose coupling benefit, DI makes services available throughout your app (for example, [logging](#)).

For more information, see [Dependency injection](#).

Middleware

In ASP.NET Core, you compose your request pipeline using [middleware](#). ASP.NET Core middleware performs

asynchronous logic on an `HttpContext` and then either invokes the next middleware in the sequence or terminates the request directly. A middleware component called "XYZ" is added by invoking an `UseXYZ` extension method in the `Configure` method.

ASP.NET Core comes with a rich set of built-in middleware:

- [Static files](#)
- [Routing](#)
- [Authentication](#)
- [Response Compression Middleware](#)
- [URL Rewriting Middleware](#)

OWIN-based middleware is available for ASP.NET Core apps, and you can write your own custom middleware.

For more information, see [Middleware](#) and [Open Web Interface for .NET \(OWIN\)](#).

Environments

Environments, such as "Development" and "Production", are a first-class notion in ASP.NET Core and can be set using environment variables.

For more information, see [Working with Multiple Environments](#).

Configuration

ASP.NET Core uses a configuration model based on name-value pairs. The configuration model isn't based on `System.Configuration` or `web.config`. Configuration obtains settings from an ordered set of configuration providers. The built-in configuration providers support a variety of file formats (XML, JSON, INI) and environment variables to enable environment-based configuration. You can also write your own custom configuration providers.

For more information, see [Configuration](#).

Logging

ASP.NET Core supports a logging API that works with a variety of logging providers. Built-in providers support sending logs to one or more destinations. Third-party logging frameworks can be used.

[Logging](#)

Error handling

ASP.NET Core has built-in features for handling errors in apps, including a developer exception page, custom error pages, static status code pages, and startup exception handling.

For more information, see [Error Handling](#).

Routing

ASP.NET Core offers features for routing of app requests to route handlers.

For more information, see [Routing](#).

File providers

ASP.NET Core abstracts file system access through the use of File Providers, which offers a common interface for

working with files across platforms.

For more information, see [File Providers](#).

Static files

Static files middleware serves static files, such as HTML, CSS, image, and JavaScript.

For more information, see [Working with static files](#).

Hosting

ASP.NET Core apps configure and launch a *host*, which is responsible for app startup and lifetime management.

For more information, see [Hosting](#).

Session and application state

Session state is a feature in ASP.NET Core that you can use to save and store user data while the user browses your web app.

For more information, see [Session and application state](#).

Servers

The ASP.NET Core hosting model doesn't directly listen for requests. The hosting model relies on an HTTP server implementation to forward the request to the app. The forwarded request is wrapped as a set of feature objects that can be accessed through interfaces. ASP.NET Core includes a managed, cross-platform web server, called [Kestrel](#). Kestrel is often run behind a production web server, such as [IIS](#) or [nginx](#). Kestrel can be run as an edge server.

For more information, see [Servers](#) and the following topics:

- [Kestrel](#)
- [ASP.NET Core Module](#)
- [HTTP.sys](#) (formerly called [WebListener](#))

Globalization and localization

Creating a multilingual website with ASP.NET Core allows your site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

For more information, see [Globalization and localization](#).

Request features

Web server implementation details related to HTTP requests and responses are defined in interfaces. These interfaces are used by server implementations and middleware to create and modify the app's hosting pipeline.

For more information, see [Request Features](#).

Open Web Interface for .NET (OWIN)

ASP.NET Core supports the Open Web Interface for .NET (OWIN). OWIN allows web apps to be decoupled from web servers.

For more information, see [Open Web Interface for .NET \(OWIN\)](#).

WebSockets

[WebSocket](#) is a protocol that enables two-way persistent communication channels over TCP connections. It's used for apps such as chat, stock tickers, games, and anywhere you desire real-time functionality in a web app. ASP.NET Core supports web socket features.

For more information, see [WebSockets](#).

Microsoft.AspNetCore.All metapackage

The [Microsoft.AspNetCore.All](#) metapackage for ASP.NET Core includes:

- All supported packages by the ASP.NET Core team.
- All supported packages by the Entity Framework Core.
- Internal and 3rd-party dependencies used by ASP.NET Core and Entity Framework Core.

For more information, see [Microsoft.AspNetCore.All metapackage](#).

.NET Core vs. .NET Framework runtime

An ASP.NET Core app can target the .NET Core or .NET Framework runtime.

For more information, see [Choosing between .NET Core and .NET Framework](#).

Choose between ASP.NET Core and ASP.NET

For more information on choosing between ASP.NET Core and ASP.NET, see [Choose between ASP.NET Core and ASP.NET](#).

Application startup in ASP.NET Core

12/19/2017 • 5 min to read • [Edit Online](#)

By [Steve Smith](#), [Tom Dykstra](#), and [Luke Latham](#)

The `Startup` class configures services and the app's request pipeline.

The Startup class

ASP.NET Core apps use a `Startup` class, which is named `Startup` by convention. The `Startup` class:

- Can optionally include a [ConfigureServices](#) method to configure the app's services.
- Must include a [Configure](#) method to create the app's request processing pipeline.

`ConfigureServices` and `Configure` are called by the runtime when the app starts:

```
public class Startup
{
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
        ...
    }
}
```

Specify the `Startup` class with the [WebHostBuilderExtensions UseStartup<TStartup>](#) method:

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

The `Startup` class constructor accepts dependencies defined by the host. A common use of [dependency injection](#) into the `Startup` class is to inject [IHostingEnvironment](#) to configure services by environment:

```

public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        HostingEnvironment = env;
    }

    public IHostingEnvironment HostingEnvironment { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }
    }
}

```

An alternative to injecting `IHostingStartup` is to use a conventions-based approach. The app can define separate `Startup` classes for different environments (for example, `StartupDevelopment`), and the appropriate startup class is selected at runtime. The class whose name suffix matches the current environment is prioritized. If the app is run in the Development environment and includes both a `Startup` class and a `StartupDevelopment` class, the `StartupDevelopment` class is used. For more information see [Working with multiple environments](#).

To learn more about `WebHostBuilder`, see the [Hosting](#) topic. For information on handling errors during startup, see [Startup exception handling](#).

The ConfigureServices method

The `ConfigureServices` method is:

- Optional.
- Called by the web host before the `Configure` method to configure the app's services.
- Where [configuration options](#) are set by convention.

Adding services to the service container makes them available within the app and in the `Configure` method. The services are resolved via [dependency injection](#) or from `IApplicationBuilder.ApplicationServices`.

The web host may configure some services before `Startup` methods are called. Details are available in the [Hosting](#) topic.

For features that require substantial setup, there are `Add[Service]` extension methods on `IServiceCollection`. A typical web app registers services for Entity Framework, Identity, and MVC:

```

public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}

```

Services available in Startup

The web host provides some services that are available to the `Startup` class constructor. The app adds additional services via `ConfigureServices`. Both the host and app services are then available in `Configure` and throughout the application.

The Configure method

The `Configure` method is used to specify how the app responds to HTTP requests. The request pipeline is configured by adding [middleware](#) components to an `IApplicationBuilder` instance. `IApplicationBuilder` is available to the `Configure` method, but it isn't registered in the service container. Hosting creates an `IApplicationBuilder` and passes it directly to `Configure` ([reference source](#)).

The [ASP.NET Core templates](#) configure the pipeline with support for a developer exception page, [BrowserLink](#), error pages, static files, and ASP.NET MVC:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller}/{action=Index}/{id?}");
    });
}

```

Each `Use` extension method adds a middleware component to the request pipeline. For instance, the `UseMvc` extension method adds the [routing middleware](#) to the request pipeline and configures [MVC](#) as the default handler.

Additional services, such as `IHostingEnvironment` and `ILoggerFactory`, may also be specified in the method signature. When specified, additional services are injected if they're available.

For more information on how to use `IApplicationBuilder`, see [Middleware](#).

Convenience methods

[ConfigureServices](#) and [Configure](#) convenience methods can be used instead of specifying a `Startup` class.

Multiple calls to `ConfigureServices` append to one another. Multiple calls to `Configure` use the last method call.

```
public class Program
{
    public static IHostingEnvironment HostingEnvironment { get; set; }

    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                HostingEnvironment = hostingContext.HostingEnvironment;
            })
            .ConfigureServices(services =>
            {
                services.AddMvc();
            })
            .Configure(app =>
            {
                if (HostingEnvironment.IsDevelopment())
                {
                    app.UseDeveloperExceptionPage();
                }
                else
                {
                    app.UseExceptionHandler("/Error");
                }

                app.UseMvcWithDefaultRoute();
                app.UseStaticFiles();
            })
            .Build();
}
```

Startup filters

Use [IStartupFilter](#) to configure middleware at the beginning or end of an app's [Configure](#) middleware pipeline.

`IStartupFilter` is useful to ensure that a middleware runs before or after middleware added by libraries at the start or end of the app's request processing pipeline.

`IStartupFilter` implements a single method, [Configure](#), which receives and returns an

`Action<IApplicationBuilder>`. An [IApplicationBuilder](#) defines a class to configure an app's request pipeline. For more information, see [Creating a middleware pipeline with IApplicationBuilder](#).

Each `IStartupFilter` implements one or more middlewares in the request pipeline. The filters are invoked in the order they were added to the service container. Filters may add middleware before or after passing control to the next filter, thus they append to the beginning or end of the app pipeline.

The [sample app \(how to download\)](#) demonstrates how to register a middleware with `IStartupFilter`. The

sample app includes a middleware that sets an options value from a query string parameter:

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;
    private IOptions<AppOptions> _injectedOptions;

    public RequestSetOptionsMiddleware(
        RequestDelegate next, IOptions<AppOptions> injectedOptions)
    {
        _next = next;
        _injectedOptions = injectedOptions;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        Console.WriteLine("RequestSetOptionsMiddleware.Invoke");

        var option = httpContext.Request.Query["option"];

        if (!string.IsNullOrEmpty(option))
        {
            _injectedOptions.Value.Option = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

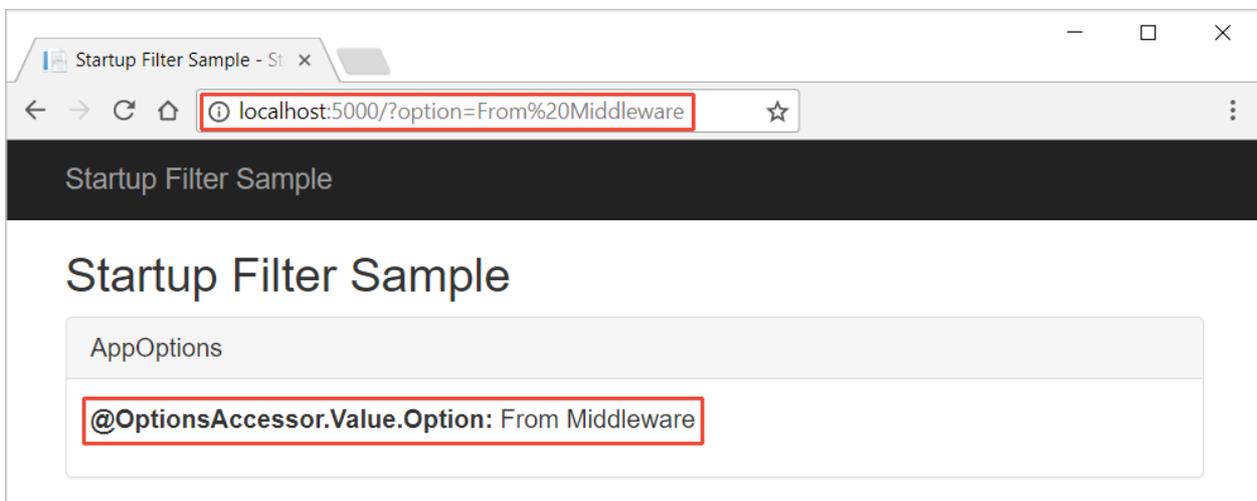
The `RequestSetOptionsMiddleware` is configured in the `RequestSetOptionsStartupFilter` class:

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

The `IStartupFilter` is registered in the service container in `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IStartupFilter, RequestSetOptionsStartupFilter>();
    services.AddMvc();
}
```

When a query string parameter for `option` is provided, the middleware processes the value assignment before the MVC middleware renders the response:



Middleware execution order is set by the order of `IStartupFilter` registrations:

- Multiple `IStartupFilter` implementations may interact with the same objects. If ordering is important, order their `IStartupFilter` service registrations to match the order that their middlewares should run.
- Libraries may add middleware with one or more `IStartupFilter` implementations that run before or after other app middleware registered with `IStartupFilter`. To invoke an `IStartupFilter` middleware before a middleware added by a library's `IStartupFilter`, position the service registration before the library is added to the service container. To invoke it afterward, position the service registration after the library is added.

Additional resources

- [Hosting](#)
- [Working with Multiple Environments](#)
- [Middleware](#)
- [Logging](#)
- [Configuration](#)
- [StartupLoader class: FindStartupType method \(reference source\)](#)

Introduction to Dependency Injection in ASP.NET Core

11/29/2017 • 16 min to read • [Edit Online](#)

By [Steve Smith](#) and [Scott Addie](#)

ASP.NET Core is designed from the ground up to support and leverage dependency injection. ASP.NET Core applications can leverage built-in framework services by having them injected into methods in the Startup class, and application services can be configured for injection as well. The default services container provided by ASP.NET Core provides a minimal feature set and is not intended to replace other containers.

[View or download sample code \(how to download\)](#)

What is Dependency Injection?

Dependency injection (DI) is a technique for achieving loose coupling between objects and their collaborators, or dependencies. Rather than directly instantiating collaborators, or using static references, the objects a class needs in order to perform its actions are provided to the class in some fashion. Most often, classes will declare their dependencies via their constructor, allowing them to follow the [Explicit Dependencies Principle](#). This approach is known as "constructor injection".

When classes are designed with DI in mind, they are more loosely coupled because they do not have direct, hard-coded dependencies on their collaborators. This follows the [Dependency Inversion Principle](#), which states that "*high level modules should not depend on low level modules; both should depend on abstractions.*" Instead of referencing specific implementations, classes request abstractions (typically `interfaces`) which are provided to them when the class is constructed. Extracting dependencies into interfaces and providing implementations of these interfaces as parameters is also an example of the [Strategy design pattern](#).

When a system is designed to use DI, with many classes requesting their dependencies via their constructor (or properties), it's helpful to have a class dedicated to creating these classes with their associated dependencies. These classes are referred to as *containers*, or more specifically, [Inversion of Control \(IoC\)](#) containers or Dependency Injection (DI) containers. A container is essentially a factory that is responsible for providing instances of types that are requested from it. If a given type has declared that it has dependencies, and the container has been configured to provide the dependency types, it will create the dependencies as part of creating the requested instance. In this way, complex dependency graphs can be provided to classes without the need for any hard-coded object construction. In addition to creating objects with their dependencies, containers typically manage object lifetimes within the application.

ASP.NET Core includes a simple built-in container (represented by the `IServiceProvider` interface) that supports constructor injection by default, and ASP.NET makes certain services available through DI. ASP.NET's container refers to the types it manages as *services*. Throughout the rest of this article, *services* will refer to types that are managed by ASP.NET Core's IoC container. You configure the built-in container's services in the `ConfigureServices` method in your application's `Startup` class.

NOTE

Martin Fowler has written an extensive article on [Inversion of Control Containers and the Dependency Injection Pattern](#). Microsoft Patterns and Practices also has a great description of [Dependency Injection](#).

NOTE

This article covers Dependency Injection as it applies to all ASP.NET applications. Dependency Injection within MVC controllers is covered in [Dependency Injection and Controllers](#).

Constructor Injection Behavior

Constructor injection requires that the constructor in question be *public*. Otherwise, your app will throw an `InvalidOperationException`:

A suitable constructor for type 'YourType' could not be located. Ensure the type is concrete and services are registered for all parameters of a public constructor.

Constructor injection requires that only one applicable constructor exist. Constructor overloads are supported, but only one overload can exist whose arguments can all be fulfilled by dependency injection. If more than one exists, your app will throw an `InvalidOperationException`:

Multiple constructors accepting all given argument types have been found in type 'YourType'. There should only be one applicable constructor.

Constructors can accept arguments that are not provided by dependency injection, but these must support default values. For example:

```
// throws InvalidOperationException: Unable to resolve service for type 'System.String'...
public CharactersController(ICharacterRepository characterRepository, string title)
{
    _characterRepository = characterRepository;
    _title = title;
}

// runs without error
public CharactersController(ICharacterRepository characterRepository, string title = "Characters")
{
    _characterRepository = characterRepository;
    _title = title;
}
```

Using Framework-Provided Services

The `ConfigureServices` method in the `Startup` class is responsible for defining the services the application will use, including platform features like Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `ConfigureServices` has the following services defined (depending on [how the host was configured](#)):

SERVICE TYPE	LIFETIME
Microsoft.AspNetCore.Hosting.IHostingEnvironment	Singleton

SERVICE TYPE	LIFETIME
Microsoft.Extensions.Logging.ILoggerFactory	Singleton
Microsoft.Extensions.Logging.ILogger<T>	Singleton
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	Transient
Microsoft.AspNetCore.Http.IHttpContextFactory	Transient
Microsoft.Extensions.Options.IOptions<T>	Singleton
System.Diagnostics.DiagnosticSource	Singleton
System.Diagnostics.DiagnosticListener	Singleton
Microsoft.AspNetCore.Hosting.IStartupFilter	Transient
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<T>	Transient
Microsoft.AspNetCore.Hosting.Server.IServer	Singleton
Microsoft.AspNetCore.Hosting.IStartup	Singleton
Microsoft.AspNetCore.Hosting.IApplicationLifetime	Singleton

Below is an example of how to add additional services to the container using a number of extension methods like `AddDbContext`, `AddIdentity`, and `AddMvc`.

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

The features and middleware provided by ASP.NET, such as MVC, follow a convention of using a single `AddServiceName` extension method to register all of the services required by that feature.

TIP

You can request certain framework-provided services within `Startup` methods through their parameter lists - see [Application Startup](#) for more details.

Registering Your Own Services

You can register your own application services as follows. The first generic type represents the type (typically an interface) that will be requested from the container. The second generic type represents the concrete type that will be instantiated by the container and used to fulfill such requests.

```
services.AddTransient<IEmailSender, AuthMessageSender>();  
services.AddTransient<ISmsSender, AuthMessageSender>();
```

NOTE

Each `services.Add<ServiceName>` extension method adds (and potentially configures) services. For example, `services.AddMvc()` adds the services MVC requires. It's recommended that you follow this convention, placing extension methods in the `Microsoft.Extensions.DependencyInjection` namespace, to encapsulate groups of service registrations.

The `AddTransient` method is used to map abstract types to concrete services that are instantiated separately for every object that requires it. This is known as the service's *lifetime*, and additional lifetime options are described below. It is important to choose an appropriate lifetime for each of the services you register. Should a new instance of the service be provided to each class that requests it? Should one instance be used throughout a given web request? Or should a single instance be used for the lifetime of the application?

In the sample for this article, there is a simple controller that displays character names, called `CharactersController`. Its `Index` method displays the current list of characters that have been stored in the application, and initializes the collection with a handful of characters if none exist. Note that although this application uses Entity Framework Core and the `ApplicationDbContext` class for its persistence, none of that is apparent in the controller. Instead, the specific data access mechanism has been abstracted behind an interface, `ICharacterRepository`, which follows the [repository pattern](#). An instance of `ICharacterRepository` is requested via the constructor and assigned to a private field, which is then used to access characters as necessary.

```

public class CharactersController : Controller
{
    private readonly ICharacterRepository _characterRepository;

    public CharactersController(ICharacterRepository characterRepository)
    {
        _characterRepository = characterRepository;
    }

    // GET: /characters/
    public IActionResult Index()
    {
        PopulateCharactersIfNoneExist();
        var characters = _characterRepository.ListAll();

        return View(characters);
    }

    private void PopulateCharactersIfNoneExist()
    {
        if (!_characterRepository.ListAll().Any())
        {
            _characterRepository.Add(new Character("Darth Maul"));
            _characterRepository.Add(new Character("Darth Vader"));
            _characterRepository.Add(new Character("Yoda"));
            _characterRepository.Add(new Character("Mace Windu"));
        }
    }
}

```

The `ICharacterRepository` defines the two methods the controller needs to work with `Character` instances.

```

using System.Collections.Generic;
using DependencyInjectionSample.Models;

namespace DependencyInjectionSample.Interfaces
{
    public interface ICharacterRepository
    {
        IEnumerable<Character> ListAll();
        void Add(Character character);
    }
}

```

This interface is in turn implemented by a concrete type, `CharacterRepository`, that is used at runtime.

NOTE

The way DI is used with the `CharacterRepository` class is a general model you can follow for all of your application services, not just in "repositories" or data access classes.

```

using System.Collections.Generic;
using System.Linq;
using DependencyInjectionSample.Interfaces;

namespace DependencyInjectionSample.Models
{
    public class CharacterRepository : ICharacterRepository
    {
        private readonly ApplicationDbContext _dbContext;

        public CharacterRepository(ApplicationDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        public IEnumerable<Character> ListAll()
        {
            return _dbContext.Characters.AsEnumerable();
        }

        public void Add(Character character)
        {
            _dbContext.Characters.Add(character);
            _dbContext.SaveChanges();
        }
    }
}

```

Note that `CharacterRepository` requests an `ApplicationDbContext` in its constructor. It is not unusual for dependency injection to be used in a chained fashion like this, with each requested dependency in turn requesting its own dependencies. The container is responsible for resolving all of the dependencies in the graph and returning the fully resolved service.

NOTE

Creating the requested object, and all of the objects it requires, and all of the objects those require, is sometimes referred to as an *object graph*. Likewise, the collective set of dependencies that must be resolved is typically referred to as a *dependency tree* or *dependency graph*.

In this case, both `ICharacterRepository` and in turn `ApplicationDbContext` must be registered with the services container in `ConfigureServices` in `Startup`. `ApplicationDbContext` is configured with the call to the extension method `AddDbContext<T>`. The following code shows the registration of the `CharacterRepository` type.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseInMemoryDatabase()
    );

    // Add framework services.
    services.AddMvc();

    // Register application services.
    services.AddScoped<ICharacterRepository, CharacterRepository>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
    services.AddTransient<OperationService, OperationService>();
}
```

Entity Framework contexts should be added to the services container using the `Scoped` lifetime. This is taken care of automatically if you use the helper methods as shown above. Repositories that will make use of Entity Framework should use the same lifetime.

WARNING

The main danger to be wary of is resolving a `Scoped` service from a singleton. It's likely in such a case that the service will have incorrect state when processing subsequent requests.

Services that have dependencies should register them in the container. If a service's constructor requires a primitive, such as a `string`, this can be injected by using [configuration](#) and the [options pattern](#).

Service Lifetimes and Registration Options

ASP.NET services can be configured with the following lifetimes:

Transient

Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.

Scoped

Scoped lifetime services are created once per request.

Singleton

Singleton lifetime services are created the first time they are requested (or when `ConfigureServices` is run if you specify an instance there) and then every subsequent request will use the same instance. If your application requires singleton behavior, allowing the services container to manage the service's lifetime is recommended instead of implementing the singleton design pattern and managing your object's lifetime in the class yourself.

Services can be registered with the container in several ways. We have already seen how to register a service implementation with a given type by specifying the concrete type to use. In addition, a factory can be specified, which will then be used to create the instance on demand. The third approach is to directly specify the instance of the type to use, in which case the container will never attempt to create an instance (nor will it dispose of the instance).

To demonstrate the difference between these lifetime and registration options, consider a simple

interface that represents one or more tasks as an *operation* with a unique identifier, `OperationId`. Depending on how we configure the lifetime for this service, the container will provide either the same or different instances of the service to the requesting class. To make it clear which lifetime is being requested, we will create one type per lifetime option:

```
using System;

namespace DependencyInjectionSample.Interfaces
{
    public interface IOperation
    {
        Guid OperationId { get; }
    }

    public interface IOperationTransient : IOperation
    {
    }
    public interface IOperationScoped : IOperation
    {
    }
    public interface IOperationSingleton : IOperation
    {
    }
    public interface IOperationSingletonInstance : IOperation
    {
    }
}
```

We implement these interfaces using a single class, `Operation`, that accepts a `Guid` in its constructor, or uses a new `Guid` if none is provided.

Next, in `ConfigureServices`, each type is added to the container according to its named lifetime:

```
services.AddScoped<ICharacterRepository, CharacterRepository>();
services.AddTransient<IOperationTransient, Operation>();
services.AddScoped<IOperationScoped, Operation>();
services.AddSingleton<IOperationSingleton, Operation>();
services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
services.AddTransient<OperationService, OperationService>();
}
```

Note that the `IOperationSingletonInstance` service is using a specific instance with a known ID of `Guid.Empty` so it will be clear when this type is in use (its `Guid` will be all zeroes). We have also registered an `OperationService` that depends on each of the other `Operation` types, so that it will be clear within a request whether this service is getting the same instance as the controller, or a new one, for each operation type. All this service does is expose its dependencies as properties, so they can be displayed in the view.

```

using DependencyInjectionSample.Interfaces;

namespace DependencyInjectionSample.Services
{
    public class OperationService
    {
        public IOperationTransient TransientOperation { get; }
        public IOperationScoped ScopedOperation { get; }
        public IOperationSingleton SingletonOperation { get; }
        public IOperationSingletonInstance SingletonInstanceOperation { get; }

        public OperationService(IOperationTransient transientOperation,
                                IOperationScoped scopedOperation,
                                IOperationSingleton singletonOperation,
                                IOperationSingletonInstance instanceOperation)
        {
            TransientOperation = transientOperation;
            ScopedOperation = scopedOperation;
            SingletonOperation = singletonOperation;
            SingletonInstanceOperation = instanceOperation;
        }
    }
}

```

To demonstrate the object lifetimes within and between separate individual requests to the application, the sample includes an `OperationsController` that requests each kind of `IOperation` type as well as an `OperationService`. The `Index` action then displays all of the controller's and service's `OperationId` values.

```

using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;
using Microsoft.AspNetCore.Mvc;

namespace DependencyInjectionSample.Controllers
{
    public class OperationsController : Controller
    {
        private readonly OperationService _operationService;
        private readonly IOperationTransient _transientOperation;
        private readonly IOperationScoped _scopedOperation;
        private readonly IOperationSingleton _singletonOperation;
        private readonly IOperationSingletonInstance _singletonInstanceOperation;

        public OperationsController(OperationService operationService,
            IOperationTransient transientOperation,
            IOperationScoped scopedOperation,
            IOperationSingleton singletonOperation,
            IOperationSingletonInstance singletonInstanceOperation)
        {
            _operationService = operationService;
            _transientOperation = transientOperation;
            _scopedOperation = scopedOperation;
            _singletonOperation = singletonOperation;
            _singletonInstanceOperation = singletonInstanceOperation;
        }

        public IActionResult Index()
        {
            // viewbag contains controller-requested services
            ViewBag.Transient = _transientOperation;
            ViewBag.Scoped = _scopedOperation;
            ViewBag.Singleton = _singletonOperation;
            ViewBag.SingletonInstance = _singletonInstanceOperation;

            // operation service has its own requested services
            ViewBag.Service = _operationService;
            return View();
        }
    }
}

```

Now two separate requests are made to this controller action:

DependencyInjectionSample Home Characters Register Log in

Lifetimes

Request One

Controller Operations

Transient	e6fee2c8-2122-4d10-aa05-cb376042e2c7
Scoped	661bff78-5ecb-4758-ae43-ec22a3e0babe
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

OperationService Operations

Transient	a379336b-3fd0-49ac-b176-bae7c27c5de5
Scoped	661bff78-5ecb-4758-ae43-ec22a3e0babe
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

DependencyInjectionSample Home Characters Register Log in

Lifetimes

Request Two

Controller Operations

Transient	d0d9cf4c-9677-491e-a633-c6b961af938d
Scoped	2a801c7e-d9ac-41ac-8a4f-259e6ff0f92c
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

OperationService Operations

Transient	11d7cfa8-e4e9-43e1-bb0e-b164a83854e2
Scoped	2a801c7e-d9ac-41ac-8a4f-259e6ff0f92c
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6
Instance	00000000-0000-0000-0000-000000000000

Observe which of the `OperationId` values vary within a request, and between requests.

- *Transient* objects are always different; a new instance is provided to every controller and every service.
- *Scoped* objects are the same within a request, but different across different requests
- *Singleton* objects are the same for every object and every request (regardless of whether an instance is provided in `ConfigureServices`)

Request Services

The services available within an ASP.NET request from `HttpContext` are exposed through the `RequestServices` collection.

```
public IActionResult Index()
{
    this.HttpContext.RequestServices
}
RequestServices
RequestServices
Gets or sets the System.IServiceProvider that provides access to the request's
service container.
```

Request Services represent the services you configure and request as part of your application. When your objects specify dependencies, these are satisfied by the types found in `RequestServices`, not `ApplicationServices`.

Generally, you shouldn't use these properties directly, preferring instead to request the types your classes you require via your class's constructor, and letting the framework inject these dependencies. This yields classes that are easier to test (see [Testing](#)) and are more loosely coupled.

NOTE

Prefer requesting dependencies as constructor parameters to accessing the `RequestServices` collection.

Designing Your Services For Dependency Injection

You should design your services to use dependency injection to get their collaborators. This means avoiding the use of stateful static method calls (which result in a code smell known as [static cling](#)) and the direct instantiation of dependent classes within your services. It may help to remember the phrase, [New is Glue](#), when choosing whether to instantiate a type or to request it via dependency injection. By following the [SOLID Principles of Object Oriented Design](#), your classes will naturally tend to be small, well-factored, and easily tested.

What if you find that your classes tend to have way too many dependencies being injected? This is generally a sign that your class is trying to do too much, and is probably violating SRP - the [Single Responsibility Principle](#). See if you can refactor the class by moving some of its responsibilities into a new class. Keep in mind that your `Controller` classes should be focused on UI concerns, so business rules and data access implementation details should be kept in classes appropriate to these [separate concerns](#).

With regards to data access specifically, you can inject the `DbContext` into your controllers (assuming you've added EF to the services container in `ConfigureServices`). Some developers prefer to use a repository interface to the database rather than injecting the `DbContext` directly. Using an interface to encapsulate the data access logic in one place can minimize how many places you will have to change when your database changes.

Disposing of services

The container will call `Dispose` for `IDisposable` types it creates. However, if you add an instance to the container yourself, it will not be disposed.

Example:

```

// Services implement IDisposable:
public class Service1 : IDisposable {}
public class Service2 : IDisposable {}
public class Service3 : IDisposable {}

public interface ISomeService {}
public class SomeServiceImplementation : ISomeService, IDisposable {}

public void ConfigureServices(IServiceCollection services)
{
    // container will create the instance(s) of these types and will dispose them
    services.AddScoped<Service1>();
    services.AddSingleton<Service2>();
    services.AddSingleton<ISomeService>(sp => new SomeServiceImplementation());

    // container did not create instance so it will NOT dispose it
    services.AddSingleton<Service3>(new Service3());
    services.AddSingleton(new Service3());
}

```

NOTE

In version 1.0, the container called dispose on *all* `IDisposable` objects, including those it did not create.

Replacing the default services container

The built-in services container is meant to serve the basic needs of the framework and most consumer applications built on it. However, developers can replace the built-in container with their preferred container. The `ConfigureServices` method typically returns `void`, but if its signature is changed to return `IServiceProvider`, a different container can be configured and returned. There are many IOC containers available for .NET. In this example, the [Autofac](#) package is used.

First, install the appropriate container package(s):

- `Autofac`
- `Autofac.Extensions.DependencyInjection`

Next, configure the container in `ConfigureServices` and return an `IServiceProvider`:

```

public IServiceProvider ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    // Add other framework services

    // Add Autofac
    var containerBuilder = new ContainerBuilder();
    containerBuilder.RegisterModule<DefaultModule>();
    containerBuilder.Populate(services);
    var container = containerBuilder.Build();
    return new AutofacServiceProvider(container);
}

```

NOTE

When using a third-party DI container, you must change `ConfigureServices` so that it returns `IServiceProvider` instead of `void`.

Finally, configure Autofac as normal in `DefaultModule` :

```
public class DefaultModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterType<CharacterRepository>().As<ICharacterRepository>();
    }
}
```

At runtime, Autofac will be used to resolve types and inject dependencies. [Learn more about using Autofac and ASP.NET Core.](#)

Thread safety

Singleton services need to be thread safe. If a singleton service has a dependency on a transient service, the transient service may also need to be thread safe depending how it's used by the singleton.

Recommendations

When working with dependency injection, keep the following recommendations in mind:

- DI is for objects that have complex dependencies. Controllers, services, adapters, and repositories are all examples of objects that might be added to DI.
- Avoid storing data and configuration directly in DI. For example, a user's shopping cart shouldn't typically be added to the services container. Configuration should use the [options pattern](#). Similarly, avoid "data holder" objects that only exist to allow access to some other object. It's better to request the actual item needed via DI, if possible.
- Avoid static access to services.
- Avoid service location in your application code.
- Avoid static access to `HttpContext` .

NOTE

Like all sets of recommendations, you may encounter situations where ignoring one is required. We have found exceptions to be rare -- mostly very special cases within the framework itself.

Remember, dependency injection is an *alternative* to static/global object access patterns. You will not be able to realize the benefits of DI if you mix it with static object access.

Additional Resources

- [Application Startup](#)
- [Testing](#)
- [Writing Clean Code in ASP.NET Core with Dependency Injection \(MSDN\)](#)
- [Container-Managed Application Design, Prelude: Where does the Container Belong?](#)
- [Explicit Dependencies Principle](#)
- [Inversion of Control Containers and the Dependency Injection Pattern \(Fowler\)](#)

ASP.NET Core Middleware Fundamentals

10/13/2017 • 8 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Steve Smith](#)

[View or download sample code](#) ([how to download](#))

What is middleware

Middleware is software that is assembled into an application pipeline to handle requests and responses. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline is invoked.

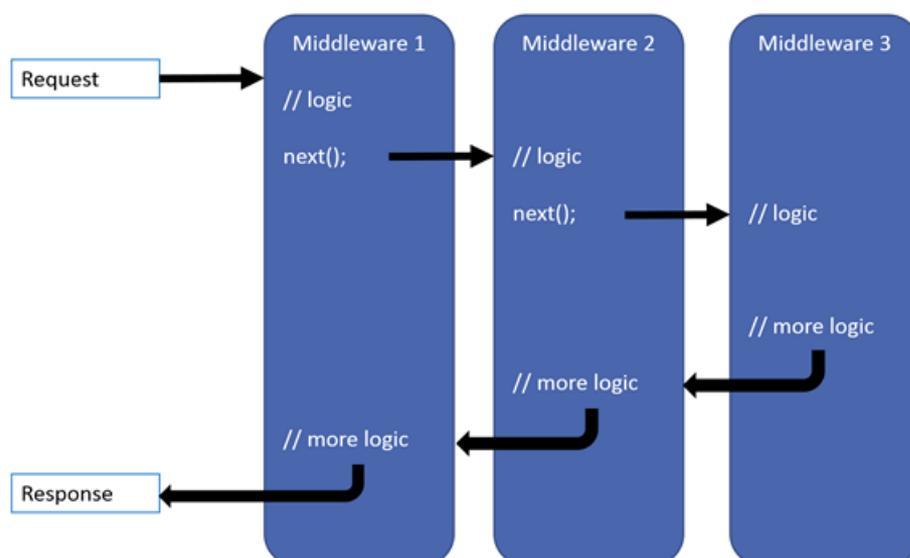
Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using [Run](#), [Map](#), and [Use](#) extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, or *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline, or short-circuiting the chain if appropriate.

[Migrating HTTP Modules to Middleware](#) explains the difference between request pipelines in ASP.NET Core and the previous versions and provides more middleware samples.

Creating a middleware pipeline with IApplicationBuilder

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other, as this diagram shows (the thread of execution follows the black arrows):



Each delegate can perform operations before and after the next delegate. A delegate can also decide to not pass a request to the next delegate, which is called short-circuiting the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, the static file middleware can return a request for a static file and short-circuit the rest of the pipeline. Exception-handling delegates need to be called

early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

The first `app.Run` delegate terminates the pipeline.

You can chain multiple request delegates together with `app.Use`. The `next` parameter represents the next delegate in the pipeline. (Remember that you can short-circuit the pipeline by *not* calling the `next` parameter.) You can typically perform actions both before and after the next delegate, as this example demonstrates:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

WARNING

Do not call `next.Invoke` after the response has been sent to the client. Changes to `HttpResponse` after the response has started will throw an exception. For example, changes such as setting headers, status code, etc, will throw an exception. Writing to the response body after calling `next` :

- May cause a protocol violation. For example, writing more than the stated `content-length` .
- May corrupt the body format. For example, writing an HTML footer to a CSS file.

`HttpResponse.HasStarted` is a useful hint to indicate if headers have been sent and/or the body has been written to.

Ordering

The order that middleware components are added in the `Configure` method defines the order in which they

are invoked on requests, and the reverse order for the response. This ordering is critical for security, performance, and functionality.

The Configure method (shown below) adds the following middleware components:

1. Exception/error handling
2. Static file server
3. Authentication
4. MVC

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void Configure(IApplicationBuilder app)
{
    app.UseExceptionHandler("/Home/Error"); // Call first to catch exceptions
                                           // thrown in the following middleware.

    app.UseStaticFiles(); // Return static files and end pipeline.

    app.UseAuthentication(); // Authenticate before you access
                             // secure resources.

    app.UseMvcWithDefaultRoute(); // Add MVC to the request pipeline.
}
```

In the code above, `UseExceptionHandler` is the first middleware component added to the pipeline—therefore, it catches any exceptions that occur in later calls.

The static file middleware is called early in the pipeline so it can handle requests and short-circuit without going through the remaining components. The static file middleware provides **no** authorization checks. Any files served by it, including those under `wwwroot`, are publicly available. See [Working with static files](#) for an approach to secure static files.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

If the request is not handled by the static file middleware, it's passed on to the Identity middleware (`app.UseAuthentication`), which performs authentication. Identity does not short-circuit unauthenticated requests. Although Identity authenticates requests, authorization (and rejection) occurs only after MVC selects a specific Razor Page or controller and action.

The following example demonstrates a middleware ordering where requests for static files are handled by the static file middleware before the response compression middleware. Static files are not compressed with this ordering of the middleware. The MVC responses from `UseMvcWithDefaultRoute` can be compressed.

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // Static files not compressed
                         // by middleware.

    app.UseResponseCompression();
    app.UseMvcWithDefaultRoute();
}
```

Use, Run, and Map

You configure the HTTP pipeline using `Use`, `Run`, and `Map`. The `Use` method can short-circuit the pipeline (that is, if it does not call a `next` request delegate). `Run` is a convention, and some middleware components

may expose `Run[Middleware]` methods that run at the end of the pipeline.

`Map*` extensions are used as a convention for branching the pipeline. `Map` branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

```
public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code:

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

When `Map` is used, the matched path segment(s) are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

`MapWhen` branches the request pipeline based on the result of the given predicate. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a predicate is used to detect the presence of a query string variable `branch`:

```

public class Startup
{
    private static void HandleBranch(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            var branchVer = context.Request.Query["branch"];
            await context.Response.WriteAsync($"Branch used = {branchVer}");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
            HandleBranch);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}

```

The following table shows the requests and responses from `http://localhost:1234` using the previous code:

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/?branch=master	Branch used = master

`Map` supports nesting, for example:

```

app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a"
        //...
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b"
        //...
    });
});

```

`Map` can also match multiple segments at once, for example:

```

app.Map("/level1/level2", HandleMultiSeg);

```

Built-in middleware

ASP.NET Core ships with the following middleware components:

MIDDLEWARE	DESCRIPTION
Authentication	Provides authentication support.
CORS	Configures Cross-Origin Resource Sharing.

MIDDLEWARE	DESCRIPTION
Response Caching	Provides support for caching responses.
Response Compression	Provides support for compressing responses.
Routing	Defines and constrains request routes.
Session	Provides support for managing user sessions.
Static Files	Provides support for serving static files and directory browsing.
URL Rewriting Middleware	Provides support for rewriting URLs and redirecting requests.

Writing middleware

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following middleware, which sets the culture for the current request from the query string:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use((context, next) =>
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrEmpty(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            return next();
        });

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });
    }
}
```

Note: The sample code above is used to demonstrate creating a middleware component. See [Globalization and localization](#) for ASP.NET Core's built-in localization support.

You can test the middleware by passing in the culture, for example `http://localhost:7997/?culture=no`.

The following code moves the middleware delegate to a class:

```

using Microsoft.AspNetCore.Http;
using System.Globalization;
using System.Threading.Tasks;

namespace Culture
{
    public class RequestCultureMiddleware
    {
        private readonly RequestDelegate _next;

        public RequestCultureMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public Task Invoke(HttpContext context)
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrEmpty(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            return this._next(context);
        }
    }
}

```

The following extension method exposes the middleware through [ApplicationBuilder](#):

```

using Microsoft.AspNetCore.Builder;

namespace Culture
{
    public static class RequestCultureMiddlewareExtensions
    {
        public static IApplicationBuilder UseRequestCulture(
            this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<RequestCultureMiddleware>();
        }
    }
}

```

The following code calls the middleware from `Configure`:

```

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRequestCulture();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });
    }
}

```

Middleware should follow the [Explicit Dependencies Principle](#) by exposing its dependencies in its constructor. Middleware is constructed once per *application lifetime*. See *Per-request dependencies* below if you need to share services with middleware within a request.

Middleware components can resolve their dependencies from dependency injection through constructor parameters. `UseMiddleware<T>` can also accept additional parameters directly.

Per-request dependencies

Because middleware is constructed at app startup, not per-request, *scoped* lifetime services used by middleware constructors are not shared with other dependency-injected types during each request. If you must share a *scoped* service between your middleware and other types, add these services to the `Invoke` method's signature. The `Invoke` method can accept additional parameters that are populated by dependency injection. For example:

```

public class MyMiddleware
{
    private readonly RequestDelegate _next;

    public MyMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext, IMyScopedService svc)
    {
        svc.MyProperty = 1000;
        await _next(httpContext);
    }
}

```

Resources

- [Sample code used in this doc](#)
- [Migrating HTTP Modules to Middleware](#)
- [Application Startup](#)
- [Request Features](#)

Working with static files in ASP.NET Core

11/15/2017 • 8 min to read • [Edit Online](#)

By [Rick Anderson](#)

Static files, such as HTML, CSS, image, and JavaScript, are assets that an ASP.NET Core app can serve directly to clients.

[View or download sample code](#) (how to download)

Serving static files

Static files are typically located in the `web root` (`<content-root>/wwwroot`) folder. See [Content root](#) and [Web root](#) for more information. You generally set the content root to be the current directory so that your project's `web root` will be found while in development.

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();

    host.Run();
}
```

Static files can be stored in any folder under the `web root` and accessed with a relative path to that root. For example, when you create a default Web application project using Visual Studio, there are several folders created within the `wwwroot` folder - `css`, `images`, and `js`. The URI to access an image in the `images` subfolder:

- `http://<app>/images/<imageFileName>`
- `http://localhost:9189/images/banner3.svg`

In order for static files to be served, you must configure the [Middleware](#) to add static files to the pipeline. The static file middleware can be configured by adding a dependency on the `Microsoft.AspNetCore.StaticFiles` package to your project and then calling the `UseStaticFiles` extension method from `Startup.Configure`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

`app.UseStaticFiles();` makes the files in `web root` (`wwwroot` by default) servable. Later I'll show how to make other directory contents servable with `UseStaticFiles`.

You must include the NuGet package "Microsoft.AspNetCore.StaticFiles".

NOTE

`web root` defaults to the `wwwroot` directory, but you can set the `web root` directory with `UseWebRoot`.

Suppose you have a project hierarchy where the static files you wish to serve are outside the `web root`. For example:

- `wwwroot`
 - `css`
 - `images`
 - ...
- `MyStaticFiles`
 - `test.png`

For a request to access `test.png`, configure the static files middleware as follows:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"MyStaticFiles")),
        RequestPath = new PathString("/StaticFiles")
    });
}
```

A request to `http://<app>/StaticFiles/test.png` will serve the `test.png` file.

`StaticFileOptions()` can set response headers. For example, the code below sets up static file serving from the `wwwroot` folder and sets the `Cache-Control` header to make them publicly cacheable for 10 minutes (600 seconds):

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions()
    {
        OnPrepareResponse = ctx =>
        {
            ctx.Context.Response.Headers.Append("Cache-Control", "public,max-age=600");
        }
    });
}
```

The `HeaderDictionaryExtensions.Append` method is available from the `Microsoft.AspNetCore.Http` package. Add `using Microsoft.AspNetCore.Http;` to your `csharp` file if the method is unavailable.

```
▼ Response Headers view source
Accept-Ranges: bytes
Cache-Control: public,max-age=600
Content-Length: 143058
Content-Type: image/png
Date: Wed, 01 Feb 2017 01:27:01 GMT
ETag: "1d1909cf92bacd2"
Last-Modified: Thu, 07 Apr 2016 07:13:24 GMT
```

Static file authorization

The static file module provides **no** authorization checks. Any files served by it, including those under `wwwroot` are publicly available. To serve files based on authorization:

- Store them outside of `wwwroot` and any directory accessible to the static file middleware **and**

- Serve them through a controller action, returning a `FileResult` where authorization is applied

Enabling directory browsing

Directory browsing allows the user of your web app to see a list of directories and files within a specified directory. Directory browsing is disabled by default for security reasons (see [Considerations](#)). To enable directory browsing, call the `UseDirectoryBrowser` extension method from `Startup.Configure`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

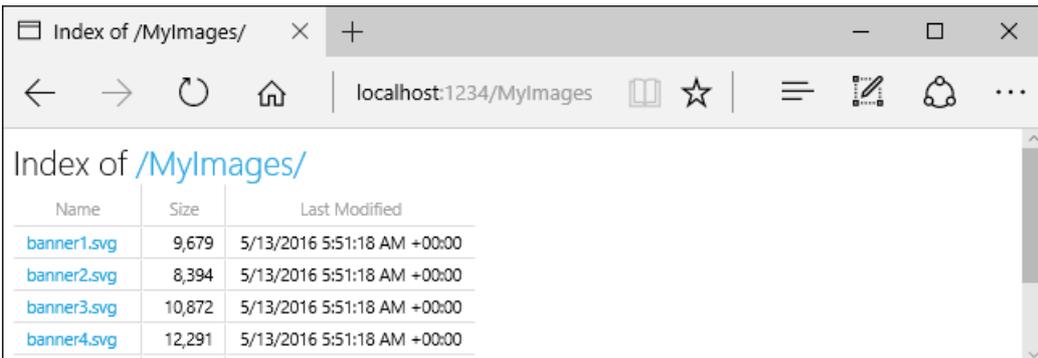
    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });
}
```

And add required services by calling `AddDirectoryBrowser` extension method from `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

The code above allows directory browsing of the `wwwroot/images` folder using the URL `http://<app>/MyImages`, with links to each file and folder:



Name	Size	Last Modified
banner1.svg	9,679	5/13/2016 5:51:18 AM +00:00
banner2.svg	8,394	5/13/2016 5:51:18 AM +00:00
banner3.svg	10,872	5/13/2016 5:51:18 AM +00:00
banner4.svg	12,291	5/13/2016 5:51:18 AM +00:00

See [Considerations](#) on the security risks when enabling browsing.

Note the two `app.UseStaticFiles` calls. The first one is required to serve the CSS, images and JavaScript in the `wwwroot` folder, and the second call for directory browsing of the `wwwroot/images` folder using the URL `http://<app>/MyImages`:

```

public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });
}
}

```

Serving a default document

Setting a default home page gives site visitors a place to start when visiting your site. In order for your Web app to serve a default page without the user having to fully qualify the URI, call the `UseDefaultFiles` extension method from `Startup.Configure` as follows.

```

public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}

```

NOTE

`UseDefaultFiles` must be called before `UseStaticFiles` to serve the default file. `UseDefaultFiles` is a URL re-writer that doesn't actually serve the file. You must enable the static file middleware (`UseStaticFiles`) to serve the file.

With `UseDefaultFiles`, requests to a folder will search for:

- default.htm
- default.html
- index.htm
- index.html

The first file found from the list will be served as if the request was the fully qualified URI (although the browser URL will continue to show the URI requested).

The following code shows how to change the default file name to *mydefault.html*.

```

public void Configure(IApplicationBuilder app)
{
    // Serve my app-specific default file, if present.
    DefaultFilesOptions options = new DefaultFilesOptions();
    options.DefaultFileNames.Clear();
    options.DefaultFileNames.Add("mydefault.html");
    app.UseDefaultFiles(options);
    app.UseStaticFiles();
}

```

UseFileServer

`UseFileServer` combines the functionality of `UseStaticFiles`, `UseDefaultFiles`, and `UseDirectoryBrowser`.

The following code enables static files and the default file to be served, but does not allow directory browsing:

```
app.UseFileServer();
```

The following code enables static files, default files and directory browsing:

```
app.UseFileServer(enableDirectoryBrowsing: true);
```

See [Considerations](#) on the security risks when enabling browsing. As with `UseStaticFiles`, `UseDefaultFiles`, and `UseDirectoryBrowser`, if you wish to serve files that exist outside the `web root`, you instantiate and configure an `FileServerOptions` object that you pass as a parameter to `UseFileServer`. For example, given the following directory hierarchy in your Web app:

- wwwroot
 - css
 - images
 - ...
- MyStaticFiles
 - test.png
 - default.html

Using the hierarchy example above, you might want to enable static files, default files, and browsing for the `MyStaticFiles` directory. In the following code snippet, that is accomplished with a single call to `FileServerOptions`.

```

public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseFileServer(new FileServerOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"MyStaticFiles")),
        RequestPath = new PathString("/StaticFiles"),
        EnableDirectoryBrowsing = true
    });
}

```

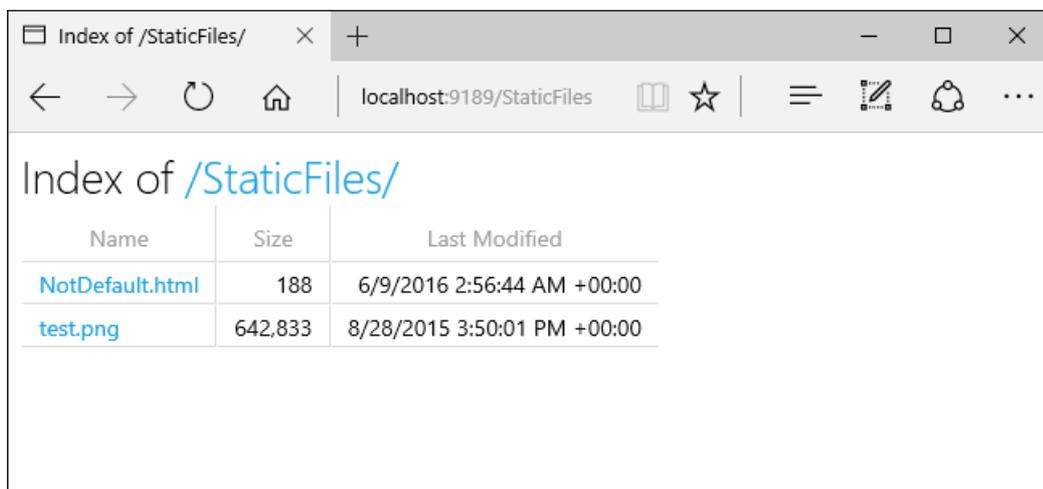
If `enableDirectoryBrowsing` is set to `true` you are required to call `AddDirectoryBrowser` extension method from `Startup.ConfigureServices` :

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

Using the file hierarchy and code above:

URI	RESPONSE
<code>http://<app>/StaticFiles/test.png</code>	MyStaticFiles/test.png
<code>http://<app>/StaticFiles</code>	MyStaticFiles/default.html

If no default named files are in the *MyStaticFiles* directory, `http://<app>/StaticFiles` returns the directory listing with clickable links:



NOTE

`UseDefaultFiles` and `UseDirectoryBrowser` will take the url `http://<app>/StaticFiles` without the trailing slash and cause a client side redirect to `http://<app>/StaticFiles/` (adding the trailing slash). Without the trailing slash relative URLs within the documents would be incorrect.

FileExtensionContentTypeProvider

The `FileExtensionContentTypeProvider` class contains a collection that maps file extensions to MIME content types. In the following sample, several file extensions are registered to known MIME types, the ".rtf" is replaced, and ".mp4" is removed.

```

public void Configure(IApplicationBuilder app)
{
    // Set up custom content types -associating file extension to MIME type
    var provider = new FileExtensionContentTypeProvider();
    // Add new mappings
    provider.Mappings[".myapp"] = "application/x-msdownload";
    provider.Mappings[".htm3"] = "text/html";
    provider.Mappings[".image"] = "image/png";
    // Replace an existing mapping
    provider.Mappings[".rtf"] = "application/x-msdownload";
    // Remove MP4 videos.
    provider.Mappings.Remove(".mp4");

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages"),
        ContentTypeProvider = provider
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });
}

```

See [MIME content types](#).

Non-standard content types

The ASP.NET static file middleware understands almost 400 known file content types. If the user requests a file of an unknown file type, the static file middleware returns a HTTP 404 (Not found) response. If directory browsing is enabled, a link to the file will be displayed, but the URI will return an HTTP 404 error.

The following code enables serving unknown types and will render the unknown file as an image.

```

public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions()
    {
        ServeUnknownFileTypes = true,
        DefaultContentType = "image/png"
    });
}

```

With the code above, a request for a file with an unknown content type will be returned as an image.

WARNING

Enabling `ServeUnknownFileTypes` is a security risk and using it is discouraged. `FileExtensionContentTypeProvider` (explained above) provides a safer alternative to serving files with non-standard extensions.

Considerations

WARNING

`UseDirectoryBrowser` and `UseStaticFiles` can leak secrets. We recommend that you **not** enable directory browsing in production. Be careful about which directories you enable with `UseStaticFiles` or `UseDirectoryBrowser` as the entire directory and all sub-directories will be accessible. We recommend keeping public content in its own directory such as `<content root>/wwwroot`, away from application views, configuration files, etc.

- The URLs for content exposed with `UseDirectoryBrowser` and `UseStaticFiles` are subject to the case sensitivity and character restrictions of their underlying file system. For example, Windows is case insensitive, but Mac and Linux are not.
- ASP.NET Core applications hosted in IIS use the ASP.NET Core Module to forward all requests to the application including requests for static files. The IIS static file handler is not used because it doesn't get a chance to handle requests before they are handled by the ASP.NET Core Module.
- To remove the IIS static file handler (at the server or website level):
 - Navigate to the **Modules** feature
 - Select **StaticFileModule** in the list
 - Tap **Remove** in the **Actions** sidebar

WARNING

If the IIS static file handler is enabled **and** the ASP.NET Core Module (ANCM) is not correctly configured (for example if `web.config` was not deployed), static files will be served.

- Code files (including `c#` and Razor) should be placed outside of the app project's `web root` (`wwwroot` by default). This creates a clean separation between your app's client side content and server side source code, which prevents server side code from being leaked.

Additional Resources

- [Middleware](#)
- [Introduction to ASP.NET Core](#)

Routing in ASP.NET Core

11/7/2017 • 19 min to read • [Edit Online](#)

By [Ryan Nowak](#), [Steve Smith](#), and [Rick Anderson](#)

Routing functionality is responsible for mapping an incoming request to a route handler. Routes are defined in the ASP.NET app and configured when the app starts up. A route can optionally extract values from the URL contained in the request, and these values can then be used for request processing. Using route information from the ASP.NET app, the routing functionality is also able to generate URLs that map to route handlers. Therefore, routing can find a route handler based on a URL, or the URL corresponding to a given route handler based on route handler information.

IMPORTANT

This document covers the low level ASP.NET Core routing. For ASP.NET Core MVC routing, see [Routing to Controller Actions](#)

[View or download sample code \(how to download\)](#)

Routing basics

Routing uses *routes* (implementations of [IRouter](#)) to:

- map incoming requests to *route handlers*
- generate URLs used in responses

Generally, an app has a single collection of routes. When a request arrives, the route collection is processed in order. The incoming request looks for a route that matches the request URL by calling the `RouteAsync` method on each available route in the route collection. By contrast, a response can use routing to generate URLs (for example, for redirection or links) based on route information, and thus avoid having to hard-code URLs, which helps maintainability.

Routing is connected to the [middleware](#) pipeline by the `RouterMiddleware` class. [ASP.NET MVC](#) adds routing to the middleware pipeline as part of its configuration. To learn about using routing as a standalone component, see [using-routing-middleware](#).

URL matching

URL matching is the process by which routing dispatches an incoming request to a *handler*. This process is generally based on data in the URL path, but can be extended to consider any data in the request. The ability to dispatch requests to separate handlers is key to scaling the size and complexity of an application.

Incoming requests enter the `RouterMiddleware`, which calls the `RouteAsync` method on each route in sequence. The `IRouter` instance chooses whether to *handle* the request by setting the `RouteContext.Handler` to a non-null `RequestDelegate`. If a route sets a handler for the request, route processing stops and the handler will be invoked to process the request. If all routes are tried and no handler is found for the request, the middleware calls *next* and the next middleware in the request pipeline is invoked.

The primary input to `RouteAsync` is the `RouteContext.HttpContext` associated with the current request. The `RouteContext.Handler` and `RouteContext.RouteData` are outputs that will be set after a route matches.

A match during `RouteAsync` will also set the properties of the `RouteContext.RouteData` to appropriate values

based on the request processing done so far. If a route matches a request, the `RouteContext.RouteData` will contain important state information about the *result*.

`RouteData.Values` is a dictionary of *route values* produced from the route. These values are usually determined by tokenizing the URL, and can be used to accept user input, or to make further dispatching decisions inside the application.

`RouteData.DataTokens` is a property bag of additional data related to the matched route. `DataTokens` are provided to support associating state data with each route so the application can make decisions later based on which route matched. These values are developer-defined and do **not** affect the behavior of routing in any way. Additionally, values stashed in data tokens can be of any type, in contrast to route values, which must be easily convertible to and from strings.

`RouteData.Routers` is a list of the routes that took part in successfully matching the request. Routes can be nested inside one another, and the `Routers` property reflects the path through the logical tree of routes that resulted in a match. Generally the first item in `Routers` is the route collection, and should be used for URL generation. The last item in `Routers` is the route handler that matched.

URL generation

URL generation is the process by which routing can create a URL path based on a set of route values. This allows for a logical separation between your handlers and the URLs that access them.

URL generation follows a similar iterative process, but starts with user or framework code calling into the `GetVirtualPath` method of the route collection. Each *route* will then have its `GetVirtualPath` method called in sequence until a non-null `VirtualPathData` is returned.

The primary inputs to `GetVirtualPath` are:

- `VirtualPathContext.HttpContext`
- `VirtualPathContext.Values`
- `VirtualPathContext.AmbientValues`

Routes primarily use the route values provided by the `Values` and `AmbientValues` to decide where it is possible to generate a URL and what values to include. The `AmbientValues` are the set of route values that were produced from matching the current request with the routing system. In contrast, `Values` are the route values that specify how to generate the desired URL for the current operation. The `HttpContext` is provided in case a route needs to get services or additional data associated with the current context.

Tip: Think of `Values` as being a set of overrides for the `AmbientValues`. URL generation tries to reuse route values from the current request to make it easy to generate URLs for links using the same route or route values.

The output of `GetVirtualPath` is a `VirtualPathData`. `VirtualPathData` is a parallel of `RouteData`; it contains the `VirtualPath` for the output URL as well as some additional properties that should be set by the route.

The `VirtualPathData.VirtualPath` property contains the *virtual path* produced by the route. Depending on your needs you may need to process the path further. For instance, if you want to render the generated URL in HTML you need to prepend the base path of the application.

The `VirtualPathData.Router` is a reference to the route that successfully generated the URL.

The `VirtualPathData.DataTokens` properties is a dictionary of additional data related to the route that generated the URL. This is the parallel of `RouteData.DataTokens`.

Creating routes

Routing provides the `Route` class as the standard implementation of `IRouter`. `Route` uses the *route template*

syntax to define patterns that will match against the URL path when `RouteAsync` is called. `Route` will use the same route template to generate a URL when `GetVirtualPath` is called.

Most applications will create routes by calling `MapRoute` or one of the similar extension methods defined on `IRouteBuilder`. All of these methods will create an instance of `Route` and add it to the route collection.

Note: `MapRoute` doesn't take a route handler parameter - it only adds routes that will be handled by the `DefaultHandler`. Since the default handler is an `IRouter`, it may decide not to handle the request. For example, ASP.NET MVC is typically configured as a default handler that only handles requests that match an available controller and action. To learn more about routing to MVC, see [Routing to Controller Actions](#).

This is an example of a `MapRoute` call used by a typical ASP.NET MVC route definition:

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id?}");
```

This template will match a URL path like `/Products/Details/17` and extract the route values `{ controller = Products, action = Details, id = 17 }`. The route values are determined by splitting the URL path into segments, and matching each segment with the *route parameter* name in the route template. Route parameters are named. They are defined by enclosing the parameter name in braces `{ }`.

The template above could also match the URL path `/` and would produce the values `{ controller = Home, action = Index }`. This happens because the `{controller}` and `{action}` route parameters have default values, and the `id` route parameter is optional. An equals `=` sign followed by a value after the route parameter name defines a default value for the parameter. A question mark `?` after the route parameter name defines the parameter as optional. Route parameters with a default value *always* produce a route value when the route matches - optional parameters will not produce a route value if there was no corresponding URL path segment.

See [route-template-reference](#) for a thorough description of route template features and syntax.

This example includes a *route constraint*:

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id:int}");
```

This template will match a URL path like `/Products/Details/17`, but not `/Products/Details/Apples`. The route parameter definition `{id:int}` defines a *route constraint* for the `id` route parameter. Route constraints implement `IRouteConstraint` and inspect route values to verify them. In this example the route value `id` must be convertible to an integer. See [route-constraint-reference](#) for a more detailed explanation of route constraints that are provided by the framework.

Additional overloads of `MapRoute` accept values for `constraints`, `dataTokens`, and `defaults`. These additional parameters of `MapRoute` are defined as type `object`. The typical usage of these parameters is to pass an anonymously typed object, where the property names of the anonymous type match route parameter names.

The following two examples create equivalent routes:

```

routes.MapRoute(
    name: "default_route",
    template: "{controller}/{action}/{id?}",
    defaults: new { controller = "Home", action = "Index" });

routes.MapRoute(
    name: "default_route",
    template: "{controller=Home}/{action=Index}/{id?}");

```

Tip: The inline syntax for defining constraints and defaults can be more convenient for simple routes. However, there are features such as data tokens which are not supported by inline syntax.

This example demonstrates a few more features:

```

routes.MapRoute(
    name: "blog",
    template: "Blog/{*article}",
    defaults: new { controller = "Blog", action = "ReadArticle" });

```

This template will match a URL path like `/Blog/All-About-Routing/Introduction` and will extract the values `{ controller = Blog, action = ReadArticle, article = All-About-Routing/Introduction }`. The default route values for `controller` and `action` are produced by the route even though there are no corresponding route parameters in the template. Default values can be specified in the route template. The `article` route parameter is defined as a *catch-all* by the appearance of an asterisk `*` before the route parameter name. Catch-all route parameters capture the remainder of the URL path, and can also match the empty string.

This example adds route constraints and data tokens:

```

routes.MapRoute(
    name: "us_english_products",
    template: "en-US/Products/{id}",
    defaults: new { controller = "Products", action = "Details" },
    constraints: new { id = new IntRouteConstraint() },
    dataTokens: new { locale = "en-US" });

```

This template will match a URL path like `/Products/5` and will extract the values `{ controller = Products, action = Details, id = 5 }` and the data tokens `{ locale = en-US }`.

The screenshot shows the Visual Studio Locals window with the following structure:

Name	Value	Type
Response	{Microsoft.AspNetCore.Http.Internal.DefaultHttpResponse}	Microsoft.AspNetCore.Http.Internal.DefaultHttpResponse
RouteData	{Microsoft.AspNetCore.Routing.RouteData}	Microsoft.AspNetCore.Routing.RouteData
DataTokens	{Microsoft.AspNetCore.Routing.RouteValueDictionary}	Microsoft.AspNetCore.Routing.RouteValueDictionary
Comparer	{System.OrdinalComparer}	System.OrdinalComparer
Count	1	int
Keys	{string[1]}	System.Collections.Generic.StringCollection
[0]	"locale"	string
Values	{object[1]}	System.Collections.Generic.Dictionary
[0]	"en-US"	object {string, object}
Non-Public members		
Results View	Expanding the Results View will enumerate the IEnumerable	
Routers	Count = 3	System.Collections.Generic.List
[0]	{Microsoft.AspNetCore.Routing.RouteCollection}	Microsoft.AspNetCore.Routing.RouteCollection
[1]	{en-US/Products/{id}}	Microsoft.AspNetCore.Routing.RouteTemplate
[2]	{Microsoft.AspNetCore.Mvc.Internal.MvcRouteHandler}	Microsoft.AspNetCore.Mvc.Internal.MvcRouteHandler

URL generation

The `Route` class can also perform URL generation by combining a set of route values with its route template. This is logically the reverse process of matching the URL path.

Tip: To better understand URL generation, imagine what URL you want to generate and then think about how a route template would match that URL. What values would be produced? This is the rough equivalent of how URL generation works in the `Route` class.

This example uses a basic ASP.NET MVC style route:

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id?}");
```

With the route values `{ controller = Products, action = List }`, this route will generate the URL `/Products/List`. The route values are substituted for the corresponding route parameters to form the URL path. Since `id` is an optional route parameter, it's no problem that it doesn't have a value.

With the route values `{ controller = Home, action = Index }`, this route will generate the URL `/`. The route values that were provided match the default values so the segments corresponding to those values can be safely omitted. Note that both URLs generated would round-trip with this route definition and produce the same route values that were used to generate the URL.

Tip: An app using ASP.NET MVC should use `UrlHelper` to generate URLs instead of calling into routing directly.

For more details about the URL generation process, see [url-generation-reference](#).

Using Routing Middleware

Add the NuGet package "Microsoft.AspNetCore.Routing".

Add routing to the service container in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddRouting();  
}
```

Routes must be configured in the `Configure` method in the `Startup` class. The sample below uses these APIs:

- `RouteBuilder`
- `Build`
- `MapGet` Matches only HTTP GET requests
- `UseRouter`

```

public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    var trackPackageRouteHandler = new RouteHandler(context =>
    {
        var routeValues = context.GetRouteData().Values;
        return context.Response.WriteAsync(
            $"Hello! Route values: {string.Join(", ", routeValues)}");
    });

    var routeBuilder = new RouteBuilder(app, trackPackageRouteHandler);

    routeBuilder.MapRoute(
        "Track Package Route",
        "package/{operation:regex:^(track|create|detonate$)}/{id:int}");

    routeBuilder.MapGet("hello/{name}", context =>
    {
        var name = context.GetRouteValue("name");
        // This is the route handler when HTTP GET "hello/<anything>" matches
        // To match HTTP GET "hello/<anything>/<anything>",
        // use routeBuilder.MapGet("hello/{*name}")
        return context.Response.WriteAsync($"Hi, {name}!");
    });

    var routes = routeBuilder.Build();
    app.UseRouter(routes);
}

```

The table below shows the responses with the given URIs.

URI	RESPONSE
/package/create/3	Hello! Route values: [operation, create], [id, 3]
/package/track/-3	Hello! Route values: [operation, track], [id, -3]
/package/track/-3/	Hello! Route values: [operation, track], [id, -3]
/package/track/	<Fall through, no match>
GET /hello/Joe	Hi, Joe!
POST /hello/Joe	<Fall through, matches HTTP GET only>
GET /hello/Joe/Smith	<Fall through, no match>

If you are configuring a single route, call `app.UseRouter` passing in an `IRouter` instance. You won't need to call `RouteBuilder`.

The framework provides a set of extension methods for creating routes such as:

- `MapRoute`
- `MapGet`
- `MapPost`
- `MapPut`
- `MapDelete`
- `MapVerb`

Some of these methods such as `MapGet` require a `RequestDeLegate` to be provided. The `RequestDeLegate` will be used as the *route handler* when the route matches. Other methods in this family allow configuring a middleware pipeline which will be used as the route handler. If the *Map* method doesn't accept a handler, such as `MapRoute`, then it will use the `DefaultHandler`.

The `Map[Verb]` methods use constraints to limit the route to the HTTP Verb in the method name. For example, see [MapGet](#) and [MapVerb](#).

Route Template Reference

Tokens within curly braces (`{ }`) define *route parameters* which will be bound if the route is matched. You can define more than one route parameter in a route segment, but they must be separated by a literal value. For example `{controller=Home}{action=Index}` would not be a valid route, since there is no literal value between `{controller}` and `{action}`. These route parameters must have a name, and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URL's path. To match the literal route parameter delimiter `{` or `}`, escape it by repeating the character (`{{` or `}}`).

URL patterns that attempt to capture a filename with an optional file extension have additional considerations. For example, using the template `files/{filename}.{ext?}` - When both `filename` and `ext` exist, both values will be populated. If only `filename` exists in the URL, the route matches because the trailing period `.` is optional. The following URLs would match this route:

- `/files/myFile.txt`
- `/files/myFile.`
- `/files/myFile`

You can use the `*` character as a prefix to a route parameter to bind to the rest of the URI - this is called a *catch-all* parameter. For example, `blog/{*slug}` would match any URI that started with `/blog` and had any value following it (which would be assigned to the `slug` route value). Catch-all parameters can also match the empty string.

Route parameters may have *default values*, designated by specifying the default after the parameter name, separated by an `=`. For example, `{controller=Home}` would define `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. In addition to default values, route parameters may be optional (specified by appending a `?` to the end of the parameter name, as in `id?`). The difference between optional and "has default" is that a route parameter with a default value always produces a value; an optional parameter has a value only when one is provided.

Route parameters may also have constraints, which must match the route value bound from the URL. Adding a colon `:` and constraint name after the route parameter name specifies an *inline constraint* on a route parameter. If the constraint requires arguments those are provided enclosed in parentheses `()` after the constraint name. Multiple inline constraints can be specified by appending another colon `:` and constraint name. The constraint name is passed to the `IInlineConstraintResolver` service to create an instance of `IRouteConstraint` to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies the `minlength` constraint with the argument `10`. For more description route constraints, and a listing of the constraints provided by the framework, see [route-constraint-reference](#).

The following table demonstrates some route templates and their behavior.

ROUTE TEMPLATE	EXAMPLE MATCHING URL	NOTES
hello	/hello	Only matches the single path <code>/hello</code>
{Page=Home}	/	Matches and sets <code>Page</code> to <code>Home</code>
{Page=Home}	/Contact	Matches and sets <code>Page</code> to <code>Contact</code>
{controller}/{action}/{id?}	/Products/List	Maps to <code>Products</code> controller and <code>List</code> action
{controller}/{action}/{id?}	/Products/Details/123	Maps to <code>Products</code> controller and <code>Details</code> action. <code>id</code> set to 123
{controller=Home}/{action=Index}/{id?}	/	Maps to <code>Home</code> controller and <code>Index</code> method; <code>id</code> is ignored.

Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

Tip: Enable [Logging](#) to see how the built in routing implementations, such as `Route`, match requests.

Route Constraint Reference

Route constraints execute when a `Route` has matched the syntax of the incoming URL and tokenized the URL path into route values. Route constraints generally inspect the route value associated via the route template and make a simple yes/no decision about whether or not the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the

`HttpMethodRouteConstraint` can accept or reject a request based on its HTTP verb.

WARNING

Avoid using constraints for **input validation**, because doing so means that invalid input will result in a 404 (Not Found) instead of a 400 with an appropriate error message. Route constraints should be used to **disambiguate** between similar routes, not to validate the inputs for a particular route.

The following table demonstrates some route constraints and their expected behavior.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
<code>int</code>	<code>{id:int}</code>	<code>123456789</code> , <code>-123456789</code>	Matches any integer
<code>bool</code>	<code>{active:bool}</code>	<code>true</code> , <code>FALSE</code>	Matches <code>true</code> or <code>false</code> (case-insensitive)
<code>datetime</code>	<code>{dob:datetime}</code>	<code>2016-12-31</code> , <code>2016-12-31 7:32pm</code>	Matches a valid <code>DateTime</code> value (in the invariant culture - see warning)
<code>decimal</code>	<code>{price:decimal}</code>	<code>49.99</code> , <code>-1,000.01</code>	Matches a valid <code>decimal</code> value (in the invariant culture - see warning)

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
<code>double</code>	<code>{weight:double}</code>	1.234 , -1,001.01e8	Matches a valid <code>double</code> value (in the invariant culture - see warning)
<code>float</code>	<code>{weight:float}</code>	1.234 , -1,001.01e8	Matches a valid <code>float</code> value (in the invariant culture - see warning)
<code>guid</code>	<code>{id:guid}</code>	CD2C1638-1638-72D5-1638-DEADBEEF1638 , {CD2C1638-1638-72D5-1638-DEADBEEF1638}	Matches a valid <code>Guid</code> value
<code>long</code>	<code>{ticks:long}</code>	123456789 , -123456789	Matches a valid <code>long</code> value
<code>minlength(value)</code>	<code>{username:minlength(4)}</code>	Rick	String must be at least 4 characters
<code>maxlength(value)</code>	<code>{filename:maxlength(8)}</code>	Richard	String must be no more than 8 characters
<code>length(length)</code>	<code>{filename:length(12)}</code>	somefile.txt	String must be exactly 12 characters long
<code>length(min,max)</code>	<code>{filename:length(8,16)}</code>	somefile.txt	String must be at least 8 and no more than 16 characters long
<code>min(value)</code>	<code>{age:min(18)}</code>	19	Integer value must be at least 18
<code>max(value)</code>	<code>{age:max(120)}</code>	91	Integer value must be no more than 120
<code>range(min,max)</code>	<code>{age:range(18,120)}</code>	91	Integer value must be at least 18 but no more than 120
<code>alpha</code>	<code>{name:alpha}</code>	Rick	String must consist of one or more alphabetical characters (a - z, case-insensitive)
<code>regex(expression)</code>	<code>{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}\$)}</code>	123-45-6789	String must match the regular expression (see tips about defining a regular expression)
<code>required</code>	<code>{name:required}</code>	Rick	Used to enforce that a non-parameter value is present during URL generation

WARNING

Route constraints that verify the URL can be converted to a CLR type (such as `int` or `DateTime`) always use the invariant culture - they assume the URL is non-localizable. The framework-provided route constraints do not modify the values stored in route values. All route values parsed from the URL will be stored as strings. For example, the [Float route constraint](#) will attempt to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

Regular expressions

The ASP.NET Core framework adds

`RegexOptions.IgnoreCase` | `RegexOptions.Compiled` | `RegexOptions.CultureInvariant` to the regular expression constructor. See [RegexOptions Enumeration](#) for a description of these members.

Regular expressions use delimiters and tokens similar to those used by Routing and the C# language. Regular expression tokens must be escaped. For example, to use the regular expression `^\d{3}-\d{2}-\d{4}$` in Routing, it needs to have the `\` characters typed in as `\\` in the C# source file to escape the `\` string escape character (unless using [verbatim string literals](#). The `{`, `}`, `'` and `]` characters need to be escaped by doubling them to escape the Routing parameter delimiter characters. The table below shows a regular expression and the escaped version.

EXPRESSION	NOTE
<code>^\d{3}-\d{2}-\d{4}\$</code>	Regular expression
<code>^\\d{3}-\\d{2}-\\d{4}\$</code>	Escaped
<code>^[a-z]{2}\$</code>	Regular expression
<code>^[[a-z]]{2}\$</code>	Escaped

Regular expressions used in routing will often start with the `^` character (match starting position of the string) and end with the `$` character (match ending position of the string). The `^` and `$` characters ensure that the regular expression match the entire route parameter value. Without the `^` and `$` characters the regular expression will match any sub-string within the string, which is often not what you want. The table below shows some examples and explains why they match or fail to match.

EXPRESSION	STRING	MATCH	COMMENT
<code>[a-z]{2}</code>	hello	yes	substring matches
<code>[a-z]{2}</code>	123abc456	yes	substring matches
<code>[a-z]{2}</code>	mz	yes	matches expression
<code>[a-z]{2}</code>	MZ	yes	not case sensitive
<code>^[a-z]{2}\$</code>	hello	no	see <code>^</code> and <code>\$</code> above
<code>^[a-z]{2}\$</code>	123abc456	no	see <code>^</code> and <code>\$</code> above

Refer to [.NET Framework Regular Expressions](#) for more information on regular expression syntax.

To constrain a parameter to a known set of possible values, use a regular expression. For example `{action:regex:^(list|get|create)$}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string `^(list|get|create)$` would be equivalent. Constraints that are passed in the constraints dictionary (not inline within a template) that don't match one of the known constraints are also treated as regular expressions.

URL Generation Reference

The example below shows how to generate a link to a route given a dictionary of route values and a

`RouteCollection`.

```
app.Run(async (context) =>
{
    var dictionary = new RouteValueDictionary
    {
        { "operation", "create" },
        { "id", 123}
    };

    var vpc = new VirtualPathContext(context, null, dictionary, "Track Package Route");
    var path = routes.GetVirtualPath(vpc).VirtualPath;

    context.Response.ContentType = "text/html";
    await context.Response.WriteAsync("Menu<hr/>");
    await context.Response.WriteAsync($"<a href='{path}'>Create Package 123</a><br/>");
});
```

The `VirtualPath` generated at the end of the sample above is `/package/create/123`.

The second parameter to the `VirtualPathContext` constructor is a collection of *ambient values*. Ambient values provide convenience by limiting the number of values a developer must specify within a certain request context. The current route values of the current request are considered ambient values for link generation. For example, in an ASP.NET MVC app if you are in the `About` action of the `HomeController`, you don't need to specify the controller route value to link to the `Index` action (the ambient value of `Home` will be used).

Ambient values that don't match a parameter are ignored, and ambient values are also ignored when an explicitly-provided value overrides it, going from left to right in the URL.

Values that are explicitly provided but which don't match anything are added to the query string. The following table shows the result when using the route template `{controller}/{action}/{id?}`.

AMBIENT VALUES	EXPLICIT VALUES	RESULT
controller="Home"	action="About"	<code>/Home/About</code>
controller="Home"	controller="Order",action="About"	<code>/Order/About</code>
controller="Home",color="Red"	action="About"	<code>/Home/About</code>
controller="Home"	action="About",color="Red"	<code>/Home/About?color=Red</code>

If a route has a default value that doesn't correspond to a parameter and that value is explicitly provided, it must match the default value. For example:

```
routes.MapRoute("blog_route", "blog/{*slug}",  
    defaults: new { controller = "Blog", action = "ReadPost" });
```

Link generation would only generate a link for this route when the matching values for controller and action are provided.

URL Rewriting Middleware in ASP.NET Core

1/10/2018 • 16 min to read • [Edit Online](#)

By [Luke Latham](#) and [Mikael Mengistu](#)

[View or download sample code \(how to download\)](#)

URL rewriting is the act of modifying request URLs based on one or more predefined rules. URL rewriting creates an abstraction between resource locations and their addresses so that the locations and addresses are not tightly linked. There are several scenarios where URL rewriting is valuable:

- Moving or replacing server resources temporarily or permanently while maintaining stable locators for those resources
- Splitting request processing across different apps or across areas of one app
- Removing, adding, or reorganizing URL segments on incoming requests
- Optimizing public URLs for Search Engine Optimization (SEO)
- Permitting the use of friendly public URLs to help people predict the content they will find by following a link
- Redirecting insecure requests to secure endpoints
- Preventing image hotlinking

You can define rules for changing the URL in several ways, including regex, Apache `mod_rewrite` module rules, IIS Rewrite Module rules, and using custom rule logic. This document introduces URL rewriting with instructions on how to use URL Rewriting Middleware in ASP.NET Core apps.

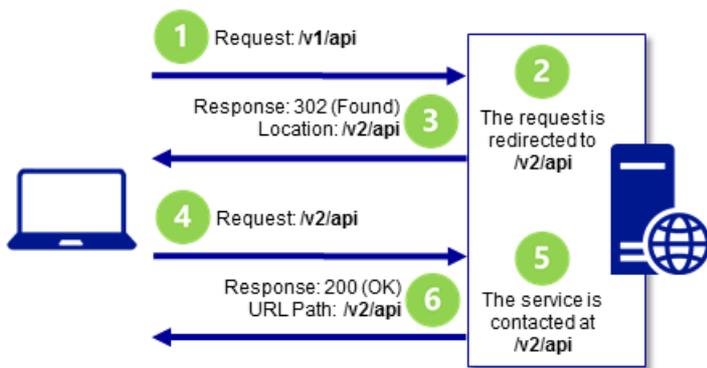
NOTE

URL rewriting can reduce the performance of an app. Where feasible, you should limit the number and complexity of rules.

URL redirect and URL rewrite

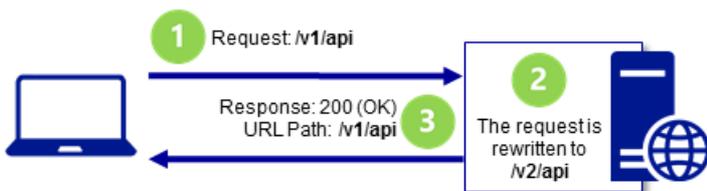
The difference in wording between *URL redirect* and *URL rewrite* may seem subtle at first but has important implications for providing resources to clients. ASP.NET Core's URL Rewriting Middleware is capable of meeting the need for both.

A *URL redirect* is a client-side operation, where the client is instructed to access a resource at another address. This requires a round-trip to the server, and the redirect URL returned to the client appears in the browser's address bar when the client makes a new request for the resource. If `/resource` is *redirected* to `/different-resource`, the client requests `/resource`, and the server responds that the client should obtain the resource at `/different-resource` with a status code indicating that the redirect is either temporary or permanent. The client executes a new request for the resource at the redirect URL.



When redirecting requests to a different URL, you indicate whether the redirect is permanent or temporary. The 301 (Moved Permanently) status code is used where the resource has a new, permanent URL and you wish to instruct the client that all future requests for the resource should use the new URL. *The client may cache the response when a 301 status code is received.* The 302 (Found) status code is used where the redirection is temporary or generally subject to change, such that the client shouldn't store and reuse the redirect URL in the future. For more information, see [RFC 2616: Status Code Definitions](#).

A *URL rewrite* is a server-side operation to provide a resource from a different resource address. Rewriting a URL doesn't require a round-trip to the server. The rewritten URL isn't returned to the client and won't appear in the browser's address bar. When `/resource` is *rewritten* to `/different-resource`, the client requests `/resource`, and the server *internally* fetches the resource at `/different-resource`. Although the client might be able to retrieve the resource at the rewritten URL, the client won't be informed that the resource exists at the rewritten URL when it makes its request and receives the response.



URL rewriting sample app

You can explore the features of the URL Rewriting Middleware with the [URL rewriting sample app](#). The app applies rewrite and redirect rules and shows the rewritten or redirected URL.

When to use URL Rewriting Middleware

Use URL Rewriting Middleware when you are unable to use the [URL Rewrite module](#) with IIS on Windows Server, the [Apache mod_rewrite module](#) on Apache Server, [URL rewriting on Nginx](#), or your app is hosted on [HTTP.sys server](#) (formerly called [WebListener](#)). The main reasons to use the server-based URL rewriting technologies in IIS, Apache, or Nginx are that the middleware doesn't support the full features of these modules and the performance of the middleware probably won't match that of the modules. However, there are some features of the server modules that don't work with ASP.NET Core projects, such as the `IsFile` and `IsDirectory` constraints of the IIS Rewrite module. In these scenarios, use the middleware instead.

Package

To include the middleware in your project, add a reference to the [Microsoft.AspNetCore.Rewrite](#) package. This feature is available for apps that target ASP.NET Core 1.1 or later.

Extension and options

Establish your URL rewrite and redirect rules by creating an instance of the `RewriteOptions` class with extension

methods for each of your rules. Chain multiple rules in the order that you would like them processed. The

`RewriteOptions` are passed into the URL Rewriting Middleware as it's added to the request pipeline with `app.UseRewriter(options);`.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
using (StreamReader apacheModRewriteStreamReader = File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader = File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXMLRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}
```

URL redirect

Use `AddRedirect` to redirect requests. The first parameter contains your regex for matching on the path of the incoming URL. The second parameter is the replacement string. The third parameter, if present, specifies the status code. If you don't specify the status code, it defaults to 302 (Found), which indicates that the resource is temporarily moved or replaced.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
using (StreamReader apacheModRewriteStreamReader = File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader = File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXMLRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}
```

In a browser with developer tools enabled, make a request to the sample app with the path

`/redirect-rule/1234/5678`. The regex matches the request path on `redirect-rule/(.*)`, and the path is replaced

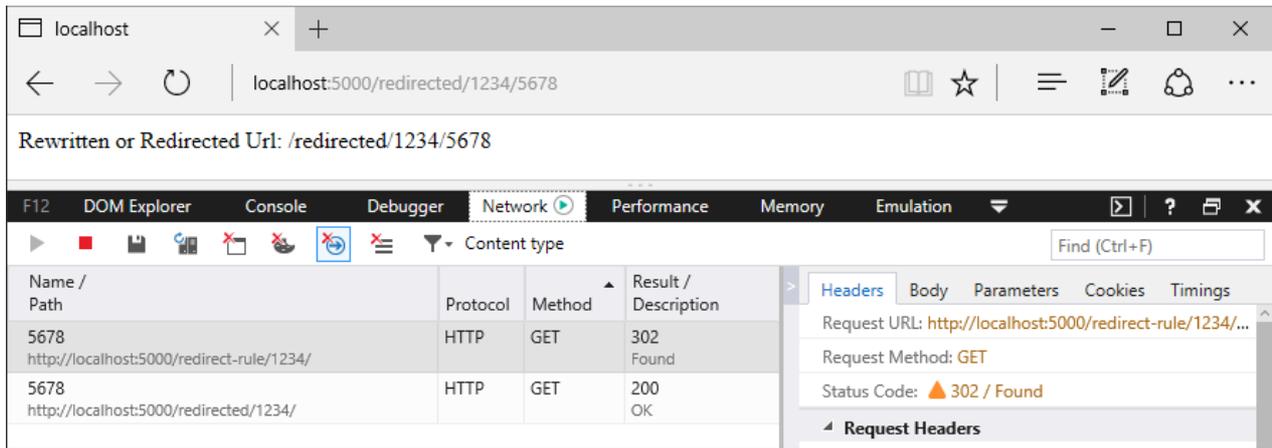
with `/redirected/1234/5678`. The redirect URL is sent back to the client with a 302 (Found) status code. The

browser makes a new request at the redirect URL, which appears in the browser's address bar. Since no rules in the sample app match on the redirect URL, the second request receives a 200 (OK) response from the app and the body of the response shows the redirect URL. A roundtrip is made to the server when a URL is *redirected*.

WARNING

Be cautious when establishing your redirect rules. Your redirect rules are evaluated on each request to the app, including after a redirect. It's easy to accidentally create a loop of infinite redirects.

Original Request: `/redirect-rule/1234/5678`



The part of the expression contained within parentheses is called a *capture group*. The dot (`.`) of the expression means *match any character*. The asterisk (`*`) indicates *match the preceding character zero or more times*. Therefore, the last two path segments of the URL, `1234/5678`, are captured by capture group `(.*)`. Any value you provide in the request URL after `redirect-rule/` is captured by this single capture group.

In the replacement string, captured groups are injected into the string with the dollar sign (`$`) followed by the sequence number of the capture. The first capture group value is obtained with `$1`, the second with `$2`, and they continue in sequence for the capture groups in your regex. There's only one captured group in the redirect rule regex in the sample app, so there's only one injected group in the replacement string, which is `$1`. When the rule is applied, the URL becomes `/redirected/1234/5678`.

URL redirect to a secure endpoint

Use `AddRedirectToHttps` to redirect HTTP requests to the same host and path using HTTPS (`https://`). If the status code isn't supplied, the middleware defaults to 302 (Found). If the port isn't supplied, the middleware defaults to `null`, which means the protocol changes to `https://` and the client accesses the resource on port 443. The example shows how to set the status code to 301 (Moved Permanently) and change the port to 5001.

```
var options = new RewriteOptions()
    .AddRedirectToHttps(301, 5001);

app.UseRewriter(options);
```

Use `AddRedirectToHttpsPermanent` to redirect insecure requests to the same host and path with secure HTTPS protocol (`https://` on port 443). The middleware sets the status code to 301 (Moved Permanently).

The sample app is capable of demonstrating how to use `AddRedirectToHttps` or `AddRedirectToHttpsPermanent`. Add the extension method to the `RewriteOptions`. Make an insecure request to the app at any URL. Dismiss the browser security warning that the self-signed certificate is untrusted.

Original Request using `AddRedirectToHttps(301, 5001)`: `/secure`

localhost x +

← → ↻ | Certificate error localhost:5001/secure

Rewritten or Redirected Url: /secure

F12 DOM Explorer Console Debugger Network Performance Memory

▶ Content type Find (Ctrl+F)

Name / Path	Protocol	Method	Result / Description
secure http://localhost:5000/	HTTP	GET	301 Moved Permanently
secure https://localhost:5001/	HTTPS	GET	200 OK

Headers Body Parameters Cookies Timings

Request URL: http://localhost:5000/secure

Request Method: GET

Status Code: 301 / Moved Permanently

Request Headers

Original Request using `AddRedirectToHttpsPermanent` : /secure

localhost x +

← → ↻ | Certificate error localhost/secure

Rewritten or Redirected Url: /secure

F12 DOM Explorer Console Debugger Network Performance Memory

▶ Content type Find (Ctrl+F)

Name / Path	Protocol	Method	Result / Description
secure http://localhost:5000/	HTTP	GET	301 Moved Permanently
secure https://localhost/	HTTPS	GET	200 OK

Headers Body Parameters Cookies Timings

Request URL: http://localhost:5000/secure

Request Method: GET

Status Code: 301 / Moved Permanently

Request Headers

URL rewrite

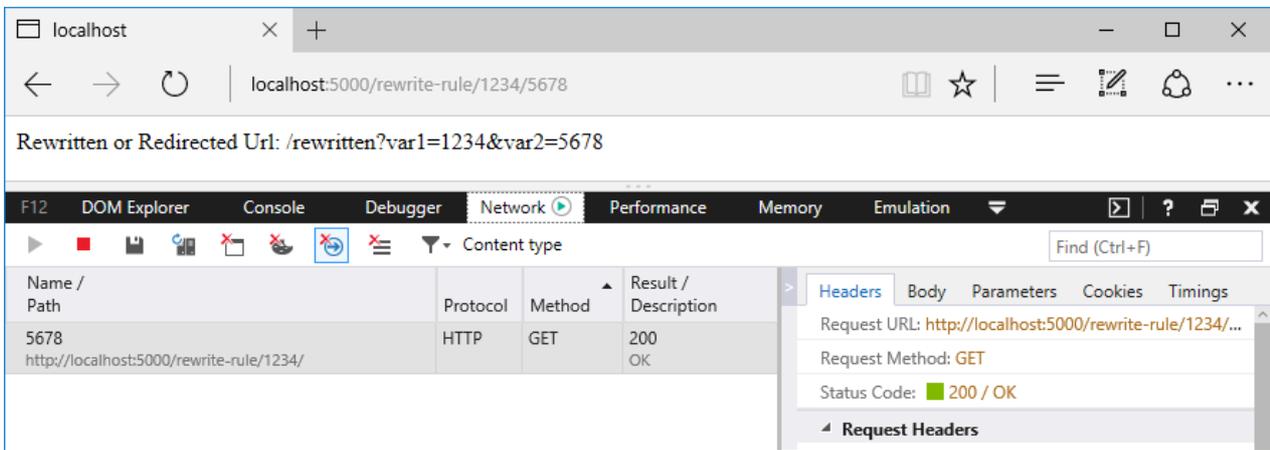
Use `AddRewrite` to create a rule for rewriting URLs. The first parameter contains your regex for matching on the incoming URL path. The second parameter is the replacement string. The third parameter, `skipRemainingRules: {true|false}`, indicates to the middleware whether or not to skip additional rewrite rules if the current rule is applied.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
using (StreamReader apacheModRewriteStreamReader = File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader = File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXMLRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}
```

Original Request: /rewrite-rule/1234/5678



The first thing you notice in the regex is the caret (`^`) at the beginning of the expression. This means that matching starts at the beginning of the URL path.

In the earlier example with the redirect rule, `redirect-rule/(.*)`, there's no caret at the start of the regex; therefore, any characters may precede `redirect-rule/` in the path for a successful match.

PATH	MATCH
<code>/redirect-rule/1234/5678</code>	Yes
<code>/my-cool-redirect-rule/1234/5678</code>	Yes
<code>/anotherredirect-rule/1234/5678</code>	Yes

The rewrite rule, `^rewrite-rule/(\d+)/(\d+)`, only matches paths if they start with `rewrite-rule/`. Notice the difference in matching between the rewrite rule below and the redirect rule above.

PATH	MATCH
<code>/rewrite-rule/1234/5678</code>	Yes
<code>/my-cool-rewrite-rule/1234/5678</code>	No
<code>/anotherrewrite-rule/1234/5678</code>	No

Following the `^rewrite-rule/` portion of the expression, there are two capture groups, `(\d+)/(\d+)`. The `\d` signifies *match a digit (number)*. The plus sign (`+`) means *match one or more of the preceding character*. Therefore, the URL must contain a number followed by a forward-slash followed by another number. These capture groups are injected into the rewritten URL as `$1` and `$2`. The rewrite rule replacement string places the captured groups into the querystring. The requested path of `/rewrite-rule/1234/5678` is rewritten to obtain the resource at `/rewritten?var1=1234&var2=5678`. If a querystring is present on the original request, it's preserved when the URL is rewritten.

There's no roundtrip to the server to obtain the resource. If the resource exists, it's fetched and returned to the client with a 200 (OK) status code. Because the client isn't redirected, the URL in the browser address bar doesn't change. As far as the client is concerned, the URL rewrite operation never occurred.

NOTE

Use `skipRemainingRules: true` whenever possible, because matching rules is an expensive process and reduces app response time. For the fastest app response:

- Order your rewrite rules from the most frequently matched rule to the least frequently matched rule.
- Skip the processing of the remaining rules when a match occurs and no additional rule processing is required.

Apache mod_rewrite

Apply Apache mod_rewrite rules with `AddApacheModRewrite`. Make sure that the rules file is deployed with the app. For more information and examples of mod_rewrite rules, see [Apache mod_rewrite](#).

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

A `StreamReader` is used to read the rules from the `ApacheModRewrite.txt` rules file.

```
using (StreamReader apacheModRewriteStreamReader = File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader = File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXMLRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}
```

The sample app redirects requests from `/apache-mod-rules-redirect/(.*)` to `/redirected?id=$1`. The response status code is 302 (Found).

```
# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

Original Request: `/apache-mod-rules-redirect/1234`

The screenshot shows a web browser window with the address bar displaying `localhost:5000/redirected?id=1234`. The page content indicates a redirect from the original URL. The Network tab is open, showing a table of requests:

Name / Path	Protocol	Method	Result / Description
1234 http://localhost:5000/apache-mod-rules-redirect/	HTTP	GET	302 Found
redirected?id=1234 http://localhost:5000/	HTTP	GET	200 OK

The right-hand pane shows the Request Headers for the 302 Found response, including the Request URL and Method.

Supported server variables

The middleware supports the following Apache mod_rewrite server variables:

- `CONN_REMOTE_ADDR`

- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_FORWARDED
- HTTP_HOST
- HTTP_REFERER
- HTTP_USER_AGENT
- HTTPS
- IPV6
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_METHOD
- REQUEST_SCHEME
- REQUEST_URI
- SCRIPT_FILENAME
- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL
- TIME
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_MON
- TIME_SEC
- TIME_WDAY
- TIME_YEAR

IIS URL Rewrite Module rules

To use rules that apply to the IIS URL Rewrite Module, use `AddIISUrlRewrite`. Make sure that the rules file is deployed with the app. Don't direct the middleware to use your *web.config* file when running on Windows Server IIS. With IIS, these rules should be stored outside of your *web.config* to avoid conflicts with the IIS Rewrite module. For more information and examples of IIS URL Rewrite Module rules, see [Using Url Rewrite Module 2.0](#) and [URL Rewrite Module Configuration Reference](#).

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

A `StreamReader` is used to read the rules from the *IISUrlRewrite.xml* rules file.

```

using (StreamReader apacheModRewriteStreamReader = File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader = File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXMLRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

```

The sample app rewrites requests from `/iis-rules-rewrite/(.*)` to `/rewritten?id=$1`. The response is sent to the client with a 200 (OK) status code.

```

<rewrite>
  <rules>
    <rule name="Rewrite segment to id querystring" stopProcessing="true">
      <match url="^iis-rules-rewrite/(.*)$" />
      <action type="Rewrite" url="rewritten?id={R:1}" appendQueryString="false"/>
    </rule>
  </rules>
</rewrite>

```

Original Request: `/iis-rules-rewrite/1234`

The screenshot shows a web browser window with the address bar displaying `localhost:5000/iis-rules-rewrite/1234`. Below the address bar, a message indicates the `Rewritten or Redirected Url: /rewritten?id=1234`. The browser's developer tools are open to the `Network` tab, showing a single request. The request table has the following data:

Name / Path	Protocol	Method	Result / Description
1234 http://localhost:5000/iis-rules-rewrite/	HTTP	GET	200 OK

The `Headers` sub-tab is selected, showing the following details:

- Request URL: `http://localhost:5000/iis-rules-rewrite/12...`
- Request Method: `GET`
- Status Code: `200 / OK`

If you have an active IIS Rewrite Module with server-level rules configured that would impact your app in undesirable ways, you can disable the IIS Rewrite Module for an app. For more information, see [Disabling IIS modules](#).

Unsupported features

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The middleware released with ASP.NET Core 2.x doesn't support the following IIS URL Rewrite Module features:

- Outbound Rules
- Custom Server Variables
- Wildcards
- LogRewrittenUrl

Supported server variables

The middleware supports the following IIS URL Rewrite Module server variables:

- CONTENT_LENGTH
- CONTENT_TYPE
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_HOST
- HTTP_REFERER
- HTTP_URL
- HTTP_USER_AGENT
- HTTPS
- LOCAL_ADDR
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_URI

NOTE

You can also obtain an `IFileProvider` via a `PhysicalFileProvider`. This approach may provide greater flexibility for the location of your rewrite rules files. Make sure that your rewrite rules files are deployed to the server at the path you provide.

```
PhysicalFileProvider fileProvider = new PhysicalFileProvider(Directory.GetCurrentDirectory());
```

Method-based rule

Use `Add(Action<RewriteContext> applyRule)` to implement your own rule logic in a method. The `RewriteContext` exposes the `HttpContext` for use in your method. The `context.Result` determines how additional pipeline processing is handled.

CONTEXT.RESULT	ACTION
<code>RuleResult.ContinueRules</code> (default)	Continue applying rules
<code>RuleResult.EndResponse</code>	Stop applying rules and send the response
<code>RuleResult.SkipRemainingRules</code>	Stop applying rules and send the context to the next middleware

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

using (StreamReader apacheModRewriteStreamReader = File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader = File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXMLRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

```

The sample app demonstrates a method that redirects requests for paths that end with *.xml*. If you make a request for `/file.xml`, it's redirected to `/xmlfiles/file.xml`. The status code is set to 301 (Moved Permanently). For a redirect, you must explicitly set the status code of the response; otherwise, a 200 (OK) status code is returned and the redirect won't occur on the client.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public static void RedirectXMLRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    // Because we're redirecting back to the same app, stop
    // processing if the request has already been redirected
    if (request.Path.StartsWithSegments(new PathString("/xmlfiles")))
    {
        return;
    }

    if (request.Path.Value.EndsWith(".xml", StringComparison.OrdinalIgnoreCase))
    {
        var response = context.HttpContext.Response;
        response.StatusCode = StatusCodes.Status301MovedPermanently;
        context.Result = RuleResult.EndResponse;
        response.Headers[HeaderNames.Location] =
            "/xmlfiles" + request.Path + request.QueryString;
    }
}

```

Original Request: `/file.xml`

The screenshot shows a browser window with the address bar displaying `localhost:5000/xmlfiles/file.xml`. Below the address bar, the browser's developer tools are open, showing the Network tab. The network tab displays two requests:

Name / Path	Protocol	Method	Result / Description
file.xml http://localhost:5000/	HTTP	GET	301 Moved Permanently
file.xml http://localhost:5000/xmlfiles/	HTTP	GET	200 OK

The 'Request Headers' panel for the 301 response is expanded, showing the following details:

- Request URL: `http://localhost:5000/file.xml`
- Request Method: `GET`
- Status Code: `301 / Moved Permanently`

IRule-based rule

Use `Add(IRule)` to implement your own rule logic in a class that derives from `IRule`. Using an `IRule` provides greater flexibility over using the method-based rule approach. Your derived class may include a constructor, where you can pass in parameters for the `ApplyRule` method.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
using (StreamReader apacheModRewriteStreamReader = File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader = File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXMLRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}
```

The values of the parameters in the sample app for the `extension` and the `newPath` are checked to meet several conditions. The `extension` must contain a value, and the value must be `.png`, `.jpg`, or `.gif`. If the `newPath` isn't valid, an `ArgumentException` is thrown. If you make a request for `image.png`, it's redirected to `/png-images/image.png`. If you make a request for `image.jpg`, it's redirected to `/jpg-images/image.jpg`. The status code is set to 301 (Moved Permanently), and the `context.Result` is set to stop processing rules and send the response.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public class RedirectImageRequests : IRule
{
    private readonly string _extension;
    private readonly PathString _newPath;

    public RedirectImageRequests(string extension, string newPath)
    {
        if (string.IsNullOrEmpty(extension))
        {
            throw new ArgumentException(nameof(extension));
        }

        if (!Regex.IsMatch(extension, @"^\.(png|jpg|gif)$"))
        {
            throw new ArgumentException("Invalid extension", nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"(/[A-Za-z0-9]+)?"))
        {
            throw new ArgumentException("Invalid path", nameof(newPath));
        }

        _extension = extension;
        _newPath = new PathString(newPath);
    }

    public void ApplyRule(RewriteContext context)
    {
        var request = context.HttpContext.Request;

        // Because we're redirecting back to the same app, stop
        // processing if the request has already been redirected
        if (request.Path.StartsWithSegments(new PathString(_newPath)))
        {
            return;
        }

        if (request.Path.Value.EndsWith(_extension, StringComparison.OrdinalIgnoreCase))
        {
            var response = context.HttpContext.Response;
            response.StatusCode = StatusCodes.Status301MovedPermanently;
            context.Result = RuleResult.EndResponse;
            response.Headers[HeaderNames.Location] =
                _newPath + request.Path + request.QueryString;
        }
    }
}

```

Original Request:

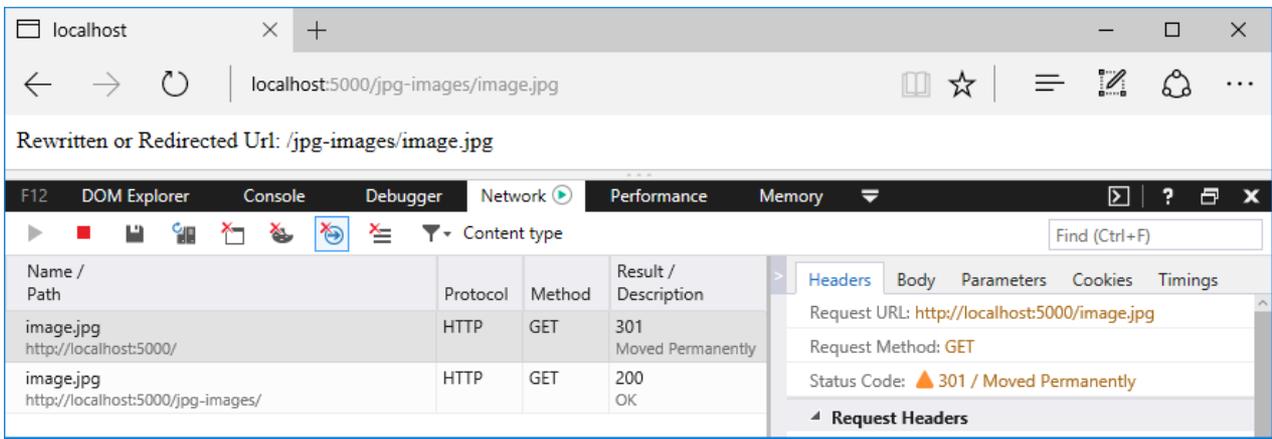
The screenshot shows a browser window with the address bar displaying `localhost:5000/png-images/image.png`. Below the address bar, it indicates the **Rewritten or Redirected Url: /png-images/image.png**. The Network tab in the developer tools is open, showing a table of requests:

Name / Path	Protocol	Method	Result / Description
image.png http://localhost:5000/	HTTP	GET	301 Moved Permanently
image.png http://localhost:5000/png-images/	HTTP	GET	200 OK

The right-hand pane of the Network tab shows the **Request Headers** for the selected request, including:

- Request URL: `http://localhost:5000/image.png`
- Request Method: `GET`
- Status Code: `301 / Moved Permanently`

Original Request:



Regex examples

GOAL	REGEX STRING & MATCH EXAMPLE	REPLACEMENT STRING & OUTPUT EXAMPLE
Rewrite path into querystring	<code>^path/(.*/)(.*)</code> /path/abc/123	<code>path?var1=\$1&var2=\$2</code> /path?var1=abc&var2=123
Strip trailing slash	<code>(.*)/\$</code> /path/	<code>\$1</code> /path
Enforce trailing slash	<code>(.*[^/])\$</code> /path	<code>\$1/</code> /path/
Avoid rewriting specific requests	<code>(.*[^(\.axd)])\$</code> Yes: /resource.htm No: /resource.axd	<code>rewritten/\$1</code> /rewritten/resource.htm /resource.axd
Rearrange URL segments	<code>path/(.*/)(.*/)(.*)</code> path/1/2/3	<code>path/\$3/\$2/\$1</code> path/3/2/1
Replace a URL segment	<code>^(.*)/segment2/(.*)</code> /segment1/segment2/segment3	<code>\$1/replaced/\$2</code> /segment1/replaced/segment3

Additional resources

- [Application Startup](#)
- [Middleware](#)
- [Regular expressions in .NET](#)
- [Regular expression language - quick reference](#)
- [Apache mod_rewrite](#)
- [Using Url Rewrite Module 2.0 \(for IIS\)](#)
- [URL Rewrite Module Configuration Reference](#)
- [IIS URL Rewrite Module Forum](#)
- [Keep a simple URL structure](#)
- [10 URL Rewriting Tips and Tricks](#)
- [To slash or not to slash](#)

Working with multiple environments

1/10/2018 • 7 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET Core provides support for controlling app behavior across multiple environments, such as development, staging, and production. Environment variables are used to indicate the runtime environment, allowing the app to be configured for that environment.

[View or download sample code \(how to download\)](#)

Development, Staging, Production

ASP.NET Core references a particular environment variable, `ASPNETCORE_ENVIRONMENT` to describe the environment the application is currently running in. This variable can be set to any value you like, but three values are used by convention: `Development`, `Staging`, and `Production`. You will find these values used in the samples and templates provided with ASP.NET Core.

The current environment setting can be detected programmatically from within your application. In addition, you can use the Environment [tag helper](#) to include certain sections in your [view](#) based on the current application environment.

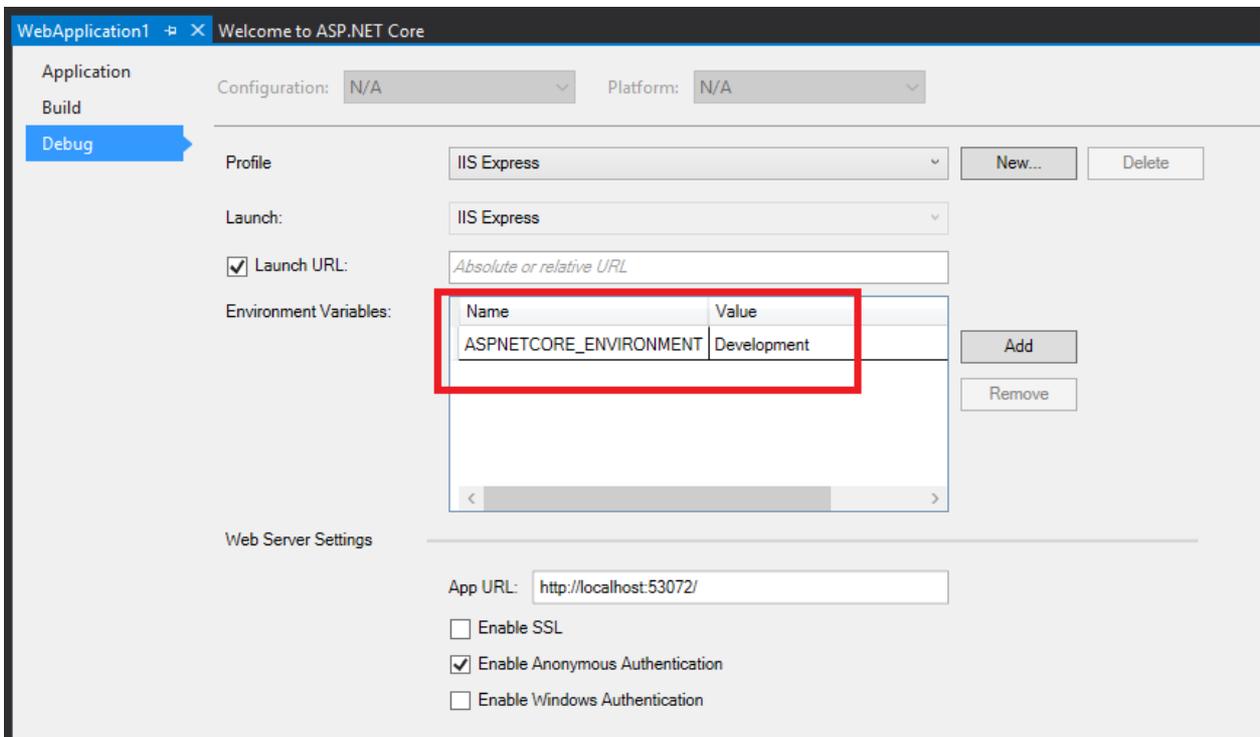
Note: On Windows and macOS, the specified environment name is case insensitive. Whether you set the variable to `Development` or `development` or `DEVELOPMENT` the results will be the same. However, Linux is a **case sensitive** OS by default. Environment variables, file names and settings require case sensitivity.

Development

This should be the environment used when developing an application. It is typically used to enable features that you wouldn't want to be available when the app runs in production, such as the [developer exception page](#).

If you're using Visual Studio, the environment can be configured in your project's debug profiles. Debug profiles specify the [server](#) to use when launching the application and any environment variables to be set. Your project can have multiple debug profiles that set environment variables differently. You manage these profiles by using the **Debug** tab of your web application project's **Properties** menu. The values you set in project properties are persisted in the `launchSettings.json` file, and you can also configure profiles by editing that file directly.

The profile for IIS Express is shown here:



Here is a `launchSettings.json` file that includes profiles for `Development` and `Staging`:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:40088/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express (Staging)": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging"
      }
    }
  }
}
```

Changes made to project profiles may not take effect until the web server used is restarted (in particular, Kestrel must be restarted before it will detect changes made to its environment).

WARNING

Environment variables stored in `launchSettings.json` are not secured in any way and will be part of the source code repository for your project, if you use one. **Never store credentials or other secret data in this file.** If you need a place to store such data, use the *Secret Manager* tool described in [Safe storage of app secrets during development](#).

Staging

By convention, a `Staging` environment is a pre-production environment used for final testing before deployment to production. Ideally, its physical characteristics should mirror that of production, so that any issues that may arise in production occur first in the staging environment, where they can be addressed without impact to users.

Production

The `Production` environment is the environment in which the application runs when it is live and being used by end users. This environment should be configured to maximize security, performance, and application robustness. Some common settings that a production environment might have that would differ from development include:

- Turn on caching
- Ensure all client-side resources are bundled, minified, and potentially served from a CDN
- Turn off diagnostic ErrorPages
- Turn on friendly error pages
- Enable production logging and monitoring (for example, [Application Insights](#))

This is by no means meant to be a complete list. It's best to avoid scattering environment checks in many parts of your application. Instead, the recommended approach is to perform such checks within the application's `Startup` class(es) wherever possible

Setting the environment

The method for setting the environment depends on the operating system.

Windows

To set the `ASPNETCORE_ENVIRONMENT` for the current session, if the app is started using `dotnet run`, the following commands are used

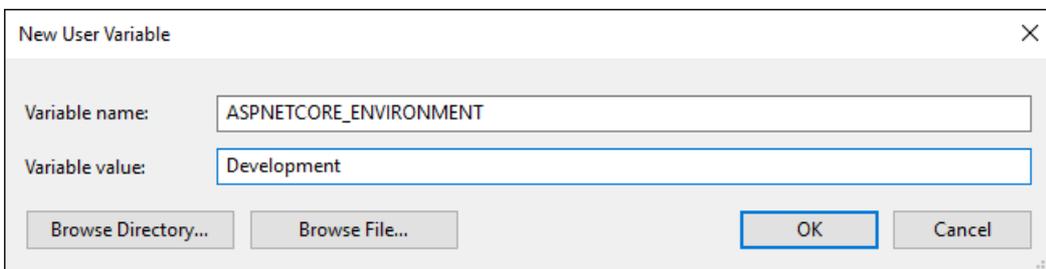
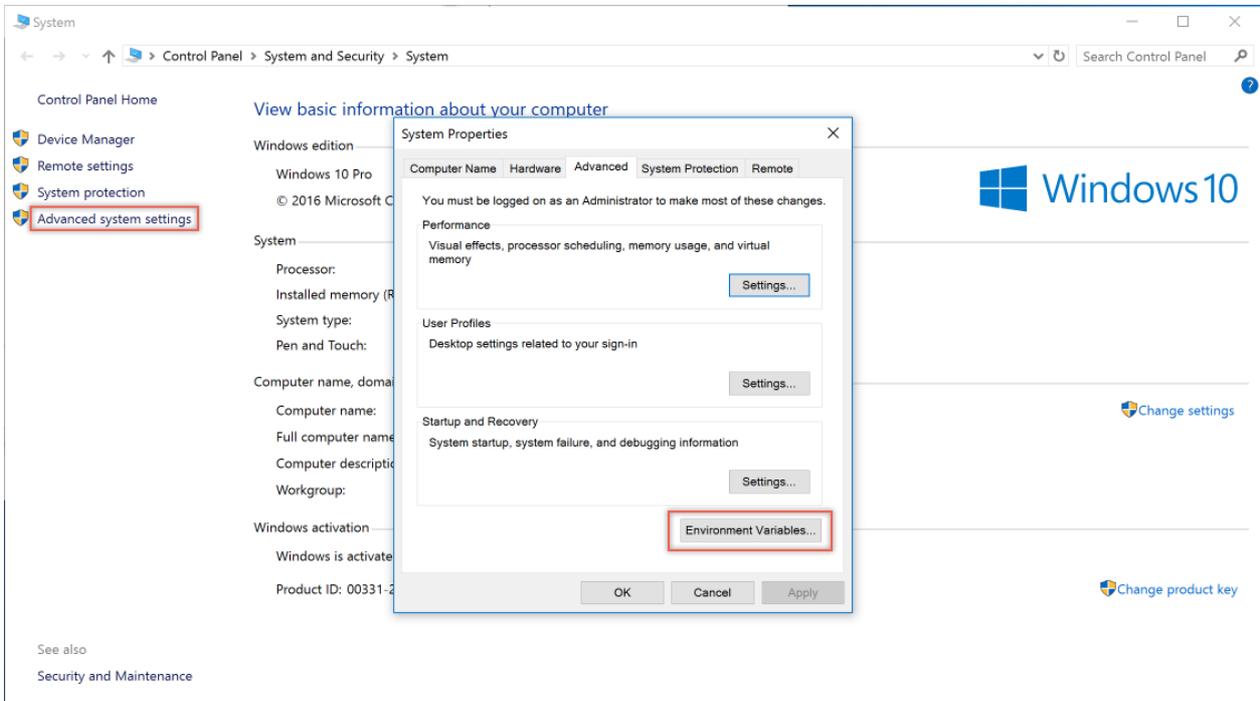
Command line

```
set ASPNETCORE_ENVIRONMENT=Development
```

PowerShell

```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

These commands take effect only for the current window. When the window is closed, the `ASPNETCORE_ENVIRONMENT` setting reverts to the default setting or machine value. In order to set the value globally on Windows open the **Control Panel** > **System** > **Advanced system settings** and add or edit the `ASPNETCORE_ENVIRONMENT` value.



web.config

See the *Setting environment variables* section of the [ASP.NET Core Module configuration reference](#) topic.

Per IIS Application Pool

If you need to set environment variables for individual apps running in isolated Application Pools (supported on IIS 10.0+), see the *AppCmd.exe command* section of the [Environment Variables <environmentVariables>](#) topic in the IIS reference documentation.

macOS

Setting the current environment for macOS can be done in-line when running the application;

```
ASPNETCORE_ENVIRONMENT=Development dotnet run
```

or using `export` to set it prior to running the app.

```
export ASPNETCORE_ENVIRONMENT=Development
```

Machine level environment variables are set in the `.bashrc` or `.bash_profile` file. Edit the file using any text editor and add the following statment.

```
export ASPNETCORE_ENVIRONMENT=Development
```

Linux

For Linux distros, use the `export` command at the command line for session based variable settings and

`bash_profile` file for machine level environment settings.

Determining the environment at runtime

The `IHostingEnvironment` service provides the core abstraction for working with environments. This service is provided by the ASP.NET hosting layer, and can be injected into your startup logic via [Dependency Injection](#). The ASP.NET Core web site template in Visual Studio uses this approach to load environment-specific configuration files (if present) and to customize the app's error handling settings. In both cases, this behavior is achieved by referring to the currently specified environment by calling `EnvironmentName` or `IsEnvironment` on the instance of `IHostingEnvironment` passed into the appropriate method.

NOTE

If you need to check whether the application is running in a particular environment, use

```
env.IsEnvironment("environmentname")
```

 since it will correctly ignore case (instead of checking if

```
env.EnvironmentName == "Development"
```

 for example).

For example, you can use the following code in your `Configure` method to setup environment specific error handling:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
}
```

If the app is running in a `Development` environment, then it enables the runtime support necessary to use the "BrowserLink" feature in Visual Studio, development-specific error pages (which typically should not be run in production) and special database error pages (which provide a way to apply migrations and should therefore only be used in development). Otherwise, if the app is not running in a development environment, a standard error handling page is configured to be displayed in response to any unhandled exceptions.

You may need to determine which content to send to the client at runtime, depending on the current environment. For example, in a development environment you generally serve non-minimized scripts and style sheets, which makes debugging easier. Production and test environments should serve the minified versions and generally from a CDN. You can do this using the Environment [tag helper](#). The Environment tag helper will only render its contents if the current environment matches one of the environments specified using the `names` attribute.

```
<environment names="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
    value="absolute" />
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

To get started with using tag helpers in your application see [Introduction to Tag Helpers](#).

Startup conventions

ASP.NET Core supports a convention-based approach to configuring an application's startup based on the current environment. You can also programmatically control how your application behaves according to which environment it is in, allowing you to create and manage your own conventions.

When an ASP.NET Core application starts, the `Startup` class is used to bootstrap the application, load its configuration settings, etc. ([learn more about ASP.NET startup](#)). However, if a class exists named `Startup{EnvironmentName}` (for example `StartupDevelopment`), and the `ASPNETCORE_ENVIRONMENT` environment variable matches that name, then that `Startup` class is used instead. Thus, you could configure `Startup` for development, but have a separate `StartupProduction` that would be used when the app is run in production. Or vice versa.

NOTE

Calling `WebHostBuilder.UseStartup<TStartup>()` overrides configuration sections.

In addition to using an entirely separate `Startup` class based on the current environment, you can also make adjustments to how the application is configured within a `Startup` class. The `Configure()` and `ConfigureServices()` methods support environment-specific versions similar to the `Startup` class itself, of the form `Configure{EnvironmentName}()` and `Configure{EnvironmentName}Services()`. If you define a method `ConfigureDevelopment()` it will be called instead of `Configure()` when the environment is set to development. Likewise, `ConfigureDevelopmentServices()` would be called instead of `ConfigureServices()` in the same environment.

Summary

ASP.NET Core provides a number of features and conventions that allow developers to easily control how their applications behave in different environments. When publishing an application from development to staging to production, environment variables set appropriately for the environment allow for optimization of the application for debugging, testing, or production use, as appropriate.

Additional Resources

- [Configuration](#)
- [Introduction to Tag Helpers](#)

Configure an ASP.NET Core App

1/10/2018 • 13 min to read • [Edit Online](#)

By [Rick Anderson](#), [Mark Michaelis](#), [Steve Smith](#), [Daniel Roth](#), and [Luke Latham](#)

The Configuration API provides a way to configure an ASP.NET Core web app based on a list of name-value pairs. Configuration is read at runtime from multiple sources. You can group these name-value pairs into a multi-level hierarchy.

There are configuration providers for:

- File formats (INI, JSON, and XML)
- Command-line arguments
- Environment variables
- In-memory .NET objects
- An encrypted user store
- [Azure Key Vault](#)
- Custom providers (installed or created)

Each configuration value maps to a string key. There's built-in binding support to deserialize settings into a custom [POCO](#) object (a simple .NET class with properties).

The options pattern uses options classes to represent groups of related settings. For more information on using the options pattern, see the [Options](#) topic.

[View or download sample code \(how to download\)](#)

JSON configuration

The following console app uses the JSON configuration provider:

```

using System;
using System.IO;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        Configuration = builder.Build();

        Console.WriteLine($"option1 = {Configuration["option1"]}");
        Console.WriteLine($"option2 = {Configuration["option2"]}");
        Console.WriteLine(
            $"subsection1 = {Configuration["subsection:suboption1"]}");
        Console.WriteLine();

        Console.WriteLine("Wizards:");
        Console.Write($"{Configuration["wizards:0:Name"]}, ");
        Console.WriteLine($"age {Configuration["wizards:0:Age"]}");
        Console.Write($"{Configuration["wizards:1:Name"]}, ");
        Console.WriteLine($"age {Configuration["wizards:1:Age"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

The app reads and displays the following configuration settings:

```

{
  "option1": "value1_from_json",
  "option2": 2,

  "subsection": {
    "suboption1": "subvalue1_from_json"
  },
  "wizards": [
    {
      "Name": "Gandalf",
      "Age": "1000"
    },
    {
      "Name": "Harry",
      "Age": "17"
    }
  ]
}

```

Configuration consists of a hierarchical list of name-value pairs in which the nodes are separated by a colon. To retrieve a value, access the `Configuration` indexer with the corresponding item's key:

```

Console.WriteLine($"option1 = {Configuration["subsection:suboption1"]}");

```

To work with arrays in JSON-formatted configuration sources, use an array index as part of the colon-separated string. The following example gets the name of the first item in the preceding `wizards` array:

```
Console.WriteLine($"{Configuration["wizards:0:Name"]}, ");
```

Name-value pairs written to the built-in `Configuration` providers are **not** persisted. However, you can create a custom provider that saves values. See [custom configuration provider](#).

The preceding sample uses the configuration indexer to read values. To access configuration outside of `Startup`, use the *options pattern*. For more information, see the [Options](#) topic.

It's typical to have different configuration settings for different environments, for example, development, testing, and production. The `CreateDefaultBuilder` extension method in an ASP.NET Core 2.x app (or using `AddJsonFile` and `AddEnvironmentVariables` directly in an ASP.NET Core 1.x app) adds configuration providers for reading JSON files and system configuration sources:

- `appsettings.json`
- `appsettings.<EnvironmentName>.json`
- Environment variables

See [AddJsonFile](#) for an explanation of the parameters. `reloadOnChange` is only supported in ASP.NET Core 1.1 and later.

Configuration sources are read in the order that they're specified. In the code above, the environment variables are read last. Any configuration values set through the environment replace those set in the two previous providers.

The environment is typically set to `Development`, `Staging`, or `Production`. See [Working with multiple environments](#) for more information.

Configuration considerations:

- `IOptionsSnapshot` can reload configuration data when it changes. See [IOptionsSnapshot](#) for more information.
- Configuration keys are case insensitive.
- Specify environment variables last so that the local environment can override settings in deployed configuration files.
- **Never** store passwords or other sensitive data in configuration provider code or in plain text configuration files. Don't use production secrets in your development or test environments. Instead, specify secrets outside of the project so that they can't be accidentally committed to your repository. Learn more about [working with multiple environments](#) and managing [safe storage of app secrets during development](#).
- If a colon (`:`) can't be used in environment variables on your system, replace the colon (`:`) with a double-underscore (`__`).

In-memory provider and binding to a POCO class

The following sample shows how to use the in-memory provider and bind to a class:

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };

        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict);

        Configuration = builder.Build();

        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        var window = new MyWindow();
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
        Console.WriteLine();

        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

```

Configuration values are returned as strings, but binding enables the construction of objects. Binding allows you to retrieve POCO objects or even entire object graphs.

GetValue

The following sample demonstrates the [GetValue<T>](#) extension method:

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };

        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict);

        Configuration = builder.Build();

        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        // Show GetValue overload and set the default value to 80
        // Requires NuGet package "Microsoft.Extensions.Configuration.Binder"
        var left = Configuration.GetValue<int>("App:MainWindow:Left", 80);
        Console.WriteLine($"Left {left}");

        var window = new MyWindow();
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

The ConfigurationBinder's `GetValue<T>` method allows you to specify a default value (80 in the sample).

`GetValue<T>` is for simple scenarios and does not bind to entire sections. `GetValue<T>` gets scalar values from `GetSection(key).Value` converted to a specific type.

Bind to an object graph

You can recursively bind to each object in a class. Consider the following `AppSettings` class:

```

public class AppSettings
{
    public Window Window { get; set; }
    public Connection Connection { get; set; }
    public Profile Profile { get; set; }
}

public class Window
{
    public int Height { get; set; }
    public int Width { get; set; }
}

public class Connection
{
    public string Value { get; set; }
}

public class Profile
{
    public string Machine { get; set; }
}

```

The following sample binds to the `AppSettings` class:

```

using System;
using System.IO;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var config = builder.Build();

        var appConfig = new AppSettings();
        config.GetSection("App").Bind(appConfig);

        Console.WriteLine($"Height {appConfig.Window.Height}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

ASP.NET Core 1.1 and higher can use `Get<T>`, which works with entire sections. `Get<T>` can be more convenient than using `Bind`. The following code shows how to use `Get<T>` with the sample above:

```

var appConfig = config.GetSection("App").Get<AppSettings>();

```

Using the following `appsettings.json` file:

```

{
  "App": {
    "Profile": {
      "Machine": "Rick"
    },
    "Connection": {
      "Value": "connectionstring"
    },
    "Window": {
      "Height": "11",
      "Width": "11"
    }
  }
}

```

The program displays `Height 11`.

The following code can be used to unit test the configuration:

```

[Fact]
public void CanBindObjectTree()
{
    var dict = new Dictionary<string, string>
    {
        {"App:Profile:Machine", "Rick"},
        {"App:Connection:Value", "connectionstring"},
        {"App:Window:Height", "11"},
        {"App:Window:Width", "11"}
    };
    var builder = new ConfigurationBuilder();
    builder.AddInMemoryCollection(dict);
    var config = builder.Build();

    var settings = new AppSettings();
    config.GetSection("App").Bind(settings);

    Assert.Equal("Rick", settings.Profile.Machine);
    Assert.Equal(11, settings.Window.Height);
    Assert.Equal(11, settings.Window.Width);
    Assert.Equal("connectionstring", settings.Connection.Value);
}

```

Create an Entity Framework custom provider

In this section, a basic configuration provider that reads name-value pairs from a database using EF is created.

Define a `ConfigurationValue` entity for storing configuration values in the database:

```

public class ConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}

```

Add a `ConfigurationContext` to store and access the configured values:

```

public class ConfigurationContext : DbContext
{
    public ConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<ConfigurationValue> Values { get; set; }
}

```

Create an class that implements [IConfigurationSource](#):

```

using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigSource : IConfigurationSource
    {
        private readonly Action<DbContextOptionsBuilder> _optionsAction;

        public EFConfigSource(Action<DbContextOptionsBuilder> optionsAction)
        {
            _optionsAction = optionsAction;
        }

        public IConfigurationProvider Build(IConfigurationBuilder builder)
        {
            return new EFConfigProvider(_optionsAction);
        }
    }
}

```

Create the custom configuration provider by inheriting from [ConfigurationProvider](#). The configuration provider initializes the database when it's empty:

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigProvider : ConfigurationProvider
    {
        public EFConfigProvider(Action<DbContextOptionsBuilder> optionsAction)
        {
            OptionsAction = optionsAction;
        }

        Action<DbContextOptionsBuilder> OptionsAction { get; }

        // Load config data from EF DB.
        public override void Load()
        {
            var builder = new DbContextOptionsBuilder<ConfigurationContext>();
            OptionsAction(builder);

            using (var dbContext = new ConfigurationContext(builder.Options))
            {
                dbContext.Database.EnsureCreated();
                Data = !dbContext.Values.Any()
                    ? CreateAndSaveDefaultValues(dbContext)
                    : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
            }
        }

        private static IDictionary<string, string> CreateAndSaveDefaultValues(
            ConfigurationContext dbContext)
        {
            var configValues = new Dictionary<string, string>
            {
                { "key1", "value_from_ef_1" },
                { "key2", "value_from_ef_2" }
            };
            dbContext.Values.AddRange(configValues
                .Select(kvp => new ConfigurationValue { Id = kvp.Key, Value = kvp.Value })
                .ToArray());
            dbContext.SaveChanges();
            return configValues;
        }
    }
}

```

The highlighted values from the database ("value_from_ef_1" and "value_from_ef_2") are displayed when the sample is run.

You can add an `EFConfigSource` extension method for adding the configuration source:

```

using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public static class EntityFrameworkExtensions
    {
        {
            public static IConfigurationBuilder AddEntityFrameworkConfig(
                this IConfigurationBuilder builder, Action<DbContextOptionsBuilder> setup)
            {
                return builder.Add(new EFConfigSource(setup));
            }
        }
    }
}

```

The following code shows how to use the custom `EFConfigProvider`:

```

using System;
using System.IO;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using CustomConfigurationProvider;

public static class Program
{
    public static void Main()
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var connectionStringConfig = builder.Build();

        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            // Add "appsettings.json" to bootstrap EF config.
            .AddJsonFile("appsettings.json")
            // Add the EF configuration provider, which will override any
            // config made with the JSON provider.
            .AddEntityFrameworkConfig(options =>
                options.UseSqlServer(connectionStringConfig.GetConnectionString(
                    "DefaultConnection"))
            )
            .Build();

        Console.WriteLine("key1={0}", config["key1"]);
        Console.WriteLine("key2={0}", config["key2"]);
        Console.WriteLine("key3={0}", config["key3"]);
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

Note the sample adds the custom `EFConfigProvider` after the JSON provider, so any settings from the database will override settings from the *appsettings.json* file.

Using the following *appsettings.json* file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\mssqllocaldb;Database=CustomConfigurationProvider;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "key1": "value_from_json_1",
  "key2": "value_from_json_2",
  "key3": "value_from_json_3"
}
```

The following is displayed:

```
key1=value_from_ef_1
key2=value_from_ef_2
key3=value_from_json_3
```

CommandLine configuration provider

The [CommandLine configuration provider](#) receives command-line argument key-value pairs for configuration at runtime.

[View or download the CommandLine configuration sample](#)

Setup and use the CommandLine configuration provider

- [Basic Configuration](#)
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

To activate command-line configuration, call the `AddCommandLine` extension method on an instance of [ConfigurationBuilder](#):

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "MairaPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args);

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

Running the code, the following output is displayed:

```

MachineName: MairaPC
Left: 1980

```

Passing argument key-value pairs on the command line changes the values of `Profile:MachineName` and `App:MainWindow:Left`:

```
dotnet run Profile:MachineName=BartPC App:MainWindow:Left=1979
```

The console window displays:

```

MachineName: BartPC
Left: 1979

```

To override configuration provided by other configuration providers with command-line configuration, call `AddCommandLine` last on `ConfigurationBuilder`:

```

var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddEnvironmentVariables()
    .AddCommandLine(args)
    .Build();

```

Arguments

Arguments passed on the command line must conform to one of two formats shown in the following table.

ARGUMENT FORMAT	EXAMPLE
Single argument: a key-value pair separated by an equals sign (=)	key1=value
Sequence of two arguments: a key-value pair separated by a space	/key1 value1

Single argument

The value must follow an equals sign (=). The value can be null (for example, mykey=).

The key may have a prefix.

KEY PREFIX	EXAMPLE
No prefix	key1=value1
Single dash (-)†	-key2=value2
Two dashes (--)	--key3=value3
Forward slash (/)	/key4=value4

†A key with a single dash prefix (-) must be provided in [switch mappings](#), described below.

Example command:

```
dotnet run key1=value1 -key2=value2 --key3=value3 /key4=value4
```

Note: If -key1 isn't present in the [switch mappings](#) given to the configuration provider, a `FormatException` is thrown.

Sequence of two arguments

The value can't be null and must follow the key separated by a space.

The key must have a prefix.

KEY PREFIX	EXAMPLE
Single dash (-)†	-key1 value1
Two dashes (--)	--key2 value2
Forward slash (/)	/key3 value3

†A key with a single dash prefix (-) must be provided in [switch mappings](#), described below.

Example command:

```
dotnet run -key1 value1 --key2 value2 /key3 value3
```

Note: If `-key1` isn't present in the [switch mappings](#) given to the configuration provider, a `FormatException` is thrown.

Duplicate keys

If duplicate keys are provided, the last key-value pair is used.

Switch mappings

When manually building configuration with `ConfigurationBuilder`, you can optionally provide a switch mappings dictionary to the `AddCommandLine` method. Switch mappings allow you to provide key name replacement logic.

When the switch mappings dictionary is used, the dictionary is checked for a key that matches the key provided by a command-line argument. If the command-line key is found in the dictionary, the dictionary value (the key replacement) is passed back to set the configuration. A switch mapping is required for any command-line key prefixed with a single dash (`-`).

Switch mappings dictionary key rules:

- Switches must start with a dash (`-`) or double-dash (`--`).
- The switch mappings dictionary must not contain duplicate keys.

In the following example, the `GetSwitchMappings` method allows your command-line arguments to use a single dash (`-`) key prefix and avoid leading subkey prefixes.

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static Dictionary<string, string> GetSwitchMappings(
        IReadOnlyDictionary<string, string> configurationStrings)
    {
        return configurationStrings.Select(item =>
            new KeyValuePair<string, string>(
                "-" + item.Key.Substring(item.Key.LastIndexOf(':') + 1),
                item.Key))
            .ToDictionary(
                item => item.Key, item => item.Value);
    }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "RickPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args, GetSwitchMappings(dict));

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

Without providing command-line arguments, the dictionary provided to `AddInMemoryCollection` sets the configuration values. Run the app with the following command:

```
dotnet run
```

The console window displays:

```
MachineName: RickPC
Left: 1980
```

Use the following to pass in configuration settings:

```
dotnet run /Profile:MachineName=DahliaPC /App:MainWindow:Left=1984
```

The console window displays:

```
MachineName: DahliaPC
Left: 1984
```

After the switch mappings dictionary is created, it contains the data shown in the following table.

KEY	VALUE
<code>-MachineName</code>	<code>Profile:MachineName</code>
<code>-Left</code>	<code>App:MainWindow:Left</code>

To demonstrate key switching using the dictionary, run the following command:

```
dotnet run -MachineName=ChadPC -Left=1988
```

The command-line keys are swapped. The console window displays the configuration values for

`Profile:MachineName` and `App:MainWindow:Left` :

```
MachineName: ChadPC
Left: 1988
```

The web.config file

A *web.config* file is required when hosting the app in IIS or IIS Express. Settings in *web.config* enable the [ASP.NET Core Module](#) to launch the app and configure other IIS settings and modules. If the *web.config* file isn't present and the project file includes `<Project Sdk="Microsoft.NET.Sdk.Web">`, publishing the project creates a *web.config* file in the published output (the *publish* folder). For more information, see [Host ASP.NET Core on Windows with IIS](#).

Additional notes

- Dependency Injection (DI) is not set up until after `ConfigureServices` is invoked.
- The configuration system is not DI aware.
- `IConfiguration` has two specializations:
 - `IConfigurationRoot` Used for the root node. Can trigger a reload.
 - `IConfigurationSection` Represents a section of configuration values. The `GetSection` and `GetChildren` methods return an `IConfigurationSection`.

Additional resources

- [Options](#)
- [Working with Multiple Environments](#)
- [Safe storage of app secrets during development](#)
- [Hosting in ASP.NET Core](#)
- [Dependency Injection](#)
- [Azure Key Vault configuration provider](#)

Configure an ASP.NET Core App

1/10/2018 • 13 min to read • [Edit Online](#)

By [Rick Anderson](#), [Mark Michaelis](#), [Steve Smith](#), [Daniel Roth](#), and [Luke Latham](#)

The Configuration API provides a way to configure an ASP.NET Core web app based on a list of name-value pairs. Configuration is read at runtime from multiple sources. You can group these name-value pairs into a multi-level hierarchy.

There are configuration providers for:

- File formats (INI, JSON, and XML)
- Command-line arguments
- Environment variables
- In-memory .NET objects
- An encrypted user store
- [Azure Key Vault](#)
- Custom providers (installed or created)

Each configuration value maps to a string key. There's built-in binding support to deserialize settings into a custom [POCO](#) object (a simple .NET class with properties).

The options pattern uses options classes to represent groups of related settings. For more information on using the options pattern, see the [Options](#) topic.

[View or download sample code](#) ([how to download](#))

JSON configuration

The following console app uses the JSON configuration provider:

```

using System;
using System.IO;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        Configuration = builder.Build();

        Console.WriteLine($"option1 = {Configuration["option1"]}");
        Console.WriteLine($"option2 = {Configuration["option2"]}");
        Console.WriteLine(
            $"subsection1 = {Configuration["subsection:suboption1"]}");
        Console.WriteLine();

        Console.WriteLine("Wizards:");
        Console.WriteLine($"{Configuration["wizards:0:Name"]}, ");
        Console.WriteLine($"age {Configuration["wizards:0:Age"]}");
        Console.WriteLine($"{Configuration["wizards:1:Name"]}, ");
        Console.WriteLine($"age {Configuration["wizards:1:Age"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

The app reads and displays the following configuration settings:

```

{
  "option1": "value1_from_json",
  "option2": 2,

  "subsection": {
    "suboption1": "subvalue1_from_json"
  },
  "wizards": [
    {
      "Name": "Gandalf",
      "Age": "1000"
    },
    {
      "Name": "Harry",
      "Age": "17"
    }
  ]
}

```

Configuration consists of a hierarchical list of name-value pairs in which the nodes are separated by a colon. To retrieve a value, access the `Configuration` indexer with the corresponding item's key:

```

Console.WriteLine($"option1 = {Configuration["subsection:suboption1"]}");

```

To work with arrays in JSON-formatted configuration sources, use an array index as part of the colon-separated string. The following example gets the name of the first item in the preceding `wizards` array:

```
Console.WriteLine($"{Configuration["wizards:0:Name"]}, ");
```

Name-value pairs written to the built-in `Configuration` providers are **not** persisted. However, you can create a custom provider that saves values. See [custom configuration provider](#).

The preceding sample uses the configuration indexer to read values. To access configuration outside of `Startup`, use the *options pattern*. For more information, see the [Options](#) topic.

It's typical to have different configuration settings for different environments, for example, development, testing, and production. The `CreateDefaultBuilder` extension method in an ASP.NET Core 2.x app (or using `AddJsonFile` and `AddEnvironmentVariables` directly in an ASP.NET Core 1.x app) adds configuration providers for reading JSON files and system configuration sources:

- `appsettings.json`
- `appsettings.<EnvironmentName>.json`
- Environment variables

See [AddJsonFile](#) for an explanation of the parameters. `reloadOnChange` is only supported in ASP.NET Core 1.1 and later.

Configuration sources are read in the order that they're specified. In the code above, the environment variables are read last. Any configuration values set through the environment replace those set in the two previous providers.

The environment is typically set to `Development`, `Staging`, or `Production`. See [Working with multiple environments](#) for more information.

Configuration considerations:

- `IOptionsSnapshot` can reload configuration data when it changes. See [IOptionsSnapshot](#) for more information.
- Configuration keys are case insensitive.
- Specify environment variables last so that the local environment can override settings in deployed configuration files.
- **Never** store passwords or other sensitive data in configuration provider code or in plain text configuration files. Don't use production secrets in your development or test environments. Instead, specify secrets outside of the project so that they can't be accidentally committed to your repository. Learn more about [working with multiple environments](#) and managing [safe storage of app secrets during development](#).
- If a colon (`:`) can't be used in environment variables on your system, replace the colon (`:`) with a double-underscore (`__`).

In-memory provider and binding to a POCO class

The following sample shows how to use the in-memory provider and bind to a class:

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };

        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict);

        Configuration = builder.Build();

        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        var window = new MyWindow();
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
        Console.WriteLine();

        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

```

Configuration values are returned as strings, but binding enables the construction of objects. Binding allows you to retrieve POCO objects or even entire object graphs.

GetValue

The following sample demonstrates the [GetValue<T>](#) extension method:

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };

        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict);

        Configuration = builder.Build();

        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        // Show GetValue overload and set the default value to 80
        // Requires NuGet package "Microsoft.Extensions.Configuration.Binder"
        var left = Configuration.GetValue<int>("App:MainWindow:Left", 80);
        Console.WriteLine($"Left {left}");

        var window = new MyWindow();
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

The ConfigurationBinder's `GetValue<T>` method allows you to specify a default value (80 in the sample). `GetValue<T>` is for simple scenarios and does not bind to entire sections. `GetValue<T>` gets scalar values from `GetSection(key).Value` converted to a specific type.

Bind to an object graph

You can recursively bind to each object in a class. Consider the following `AppSettings` class:

```

public class AppSettings
{
    public Window Window { get; set; }
    public Connection Connection { get; set; }
    public Profile Profile { get; set; }
}

public class Window
{
    public int Height { get; set; }
    public int Width { get; set; }
}

public class Connection
{
    public string Value { get; set; }
}

public class Profile
{
    public string Machine { get; set; }
}

```

The following sample binds to the `AppSettings` class:

```

using System;
using System.IO;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var config = builder.Build();

        var appConfig = new AppSettings();
        config.GetSection("App").Bind(appConfig);

        Console.WriteLine($"Height {appConfig.Window.Height}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

ASP.NET Core 1.1 and higher can use `Get<T>`, which works with entire sections. `Get<T>` can be more convenient than using `Bind`. The following code shows how to use `Get<T>` with the sample above:

```

var appConfig = config.GetSection("App").Get<AppSettings>();

```

Using the following `appsettings.json` file:

```

{
  "App": {
    "Profile": {
      "Machine": "Rick"
    },
    "Connection": {
      "Value": "connectionstring"
    },
    "Window": {
      "Height": "11",
      "Width": "11"
    }
  }
}

```

The program displays `Height 11`.

The following code can be used to unit test the configuration:

```

[Fact]
public void CanBindObjectTree()
{
    var dict = new Dictionary<string, string>
    {
        {"App:Profile:Machine", "Rick"},
        {"App:Connection:Value", "connectionstring"},
        {"App:Window:Height", "11"},
        {"App:Window:Width", "11"}
    };
    var builder = new ConfigurationBuilder();
    builder.AddInMemoryCollection(dict);
    var config = builder.Build();

    var settings = new AppSettings();
    config.GetSection("App").Bind(settings);

    Assert.Equal("Rick", settings.Profile.Machine);
    Assert.Equal(11, settings.Window.Height);
    Assert.Equal(11, settings.Window.Width);
    Assert.Equal("connectionstring", settings.Connection.Value);
}

```

Create an Entity Framework custom provider

In this section, a basic configuration provider that reads name-value pairs from a database using EF is created.

Define a `ConfigurationValue` entity for storing configuration values in the database:

```

public class ConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}

```

Add a `ConfigurationContext` to store and access the configured values:

```
public class ConfigurationContext : DbContext
{
    public ConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<ConfigurationValue> Values { get; set; }
}
```

Create an class that implements [IConfigurationSource](#):

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigSource : IConfigurationSource
    {
        private readonly Action<DbContextOptionsBuilder> _optionsAction;

        public EFConfigSource(Action<DbContextOptionsBuilder> optionsAction)
        {
            _optionsAction = optionsAction;
        }

        public IConfigurationProvider Build(IConfigurationBuilder builder)
        {
            return new EFConfigProvider(_optionsAction);
        }
    }
}
```

Create the custom configuration provider by inheriting from [ConfigurationProvider](#). The configuration provider initializes the database when it's empty:

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigProvider : ConfigurationProvider
    {
        public EFConfigProvider(Action<DbContextOptionsBuilder> optionsAction)
        {
            OptionsAction = optionsAction;
        }

        Action<DbContextOptionsBuilder> OptionsAction { get; }

        // Load config data from EF DB.
        public override void Load()
        {
            var builder = new DbContextOptionsBuilder<ConfigurationContext>();
            OptionsAction(builder);

            using (var dbContext = new ConfigurationContext(builder.Options))
            {
                dbContext.Database.EnsureCreated();
                Data = !dbContext.Values.Any()
                    ? CreateAndSaveDefaultValues(dbContext)
                    : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
            }
        }

        private static IDictionary<string, string> CreateAndSaveDefaultValues(
            ConfigurationContext dbContext)
        {
            var configValues = new Dictionary<string, string>
            {
                { "key1", "value_from_ef_1" },
                { "key2", "value_from_ef_2" }
            };
            dbContext.Values.AddRange(configValues
                .Select(kvp => new ConfigurationValue { Id = kvp.Key, Value = kvp.Value })
                .ToArray());
            dbContext.SaveChanges();
            return configValues;
        }
    }
}

```

The highlighted values from the database ("value_from_ef_1" and "value_from_ef_2") are displayed when the sample is run.

You can add an `EFConfigSource` extension method for adding the configuration source:

```

using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public static class EntityFrameworkExtensions
    {
        public static IConfigurationBuilder AddEntityFrameworkConfig(
            this IConfigurationBuilder builder, Action<DbContextOptionsBuilder> setup)
        {
            return builder.Add(new EFConfigSource(setup));
        }
    }
}

```

The following code shows how to use the custom `EFConfigProvider`:

```

using System;
using System.IO;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using CustomConfigurationProvider;

public static class Program
{
    public static void Main()
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var connectionStringConfig = builder.Build();

        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            // Add "appsettings.json" to bootstrap EF config.
            .AddJsonFile("appsettings.json")
            // Add the EF configuration provider, which will override any
            // config made with the JSON provider.
            .AddEntityFrameworkConfig(options =>
                options.UseSqlServer(connectionStringConfig.GetConnectionString(
                    "DefaultConnection")))
            .Build();

        Console.WriteLine("key1={0}", config["key1"]);
        Console.WriteLine("key2={0}", config["key2"]);
        Console.WriteLine("key3={0}", config["key3"]);
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

Note the sample adds the custom `EFConfigProvider` after the JSON provider, so any settings from the database will override settings from the `appsettings.json` file.

Using the following `appsettings.json` file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\mssqllocaldb;Database=CustomConfigurationProvider;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "key1": "value_from_json_1",
  "key2": "value_from_json_2",
  "key3": "value_from_json_3"
}
```

The following is displayed:

```
key1=value_from_ef_1
key2=value_from_ef_2
key3=value_from_json_3
```

CommandLine configuration provider

The [CommandLine configuration provider](#) receives command-line argument key-value pairs for configuration at runtime.

[View or download the CommandLine configuration sample](#)

Setup and use the CommandLine configuration provider

- [Basic Configuration](#)
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

To activate command-line configuration, call the `AddCommandLine` extension method on an instance of [ConfigurationBuilder](#):

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "MairaPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args);

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

Running the code, the following output is displayed:

```

MachineName: MairaPC
Left: 1980

```

Passing argument key-value pairs on the command line changes the values of `Profile:MachineName` and `App:MainWindow:Left`:

```
dotnet run Profile:MachineName=BartPC App:MainWindow:Left=1979
```

The console window displays:

```

MachineName: BartPC
Left: 1979

```

To override configuration provided by other configuration providers with command-line configuration, call `AddCommandLine` last on `ConfigurationBuilder`:

```

var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddEnvironmentVariables()
    .AddCommandLine(args)
    .Build();

```

Arguments

Arguments passed on the command line must conform to one of two formats shown in the following table.

ARGUMENT FORMAT	EXAMPLE
Single argument: a key-value pair separated by an equals sign (=)	key1=value
Sequence of two arguments: a key-value pair separated by a space	/key1 value1

Single argument

The value must follow an equals sign (=). The value can be null (for example, mykey=).

The key may have a prefix.

KEY PREFIX	EXAMPLE
No prefix	key1=value1
Single dash (-)†	-key2=value2
Two dashes (--)	--key3=value3
Forward slash (/)	/key4=value4

†A key with a single dash prefix (-) must be provided in [switch mappings](#), described below.

Example command:

```
dotnet run key1=value1 -key2=value2 --key3=value3 /key4=value4
```

Note: If -key1 isn't present in the [switch mappings](#) given to the configuration provider, a `FormatException` is thrown.

Sequence of two arguments

The value can't be null and must follow the key separated by a space.

The key must have a prefix.

KEY PREFIX	EXAMPLE
Single dash (-)†	-key1 value1
Two dashes (--)	--key2 value2
Forward slash (/)	/key3 value3

†A key with a single dash prefix (-) must be provided in [switch mappings](#), described below.

Example command:

```
dotnet run -key1 value1 --key2 value2 /key3 value3
```

Note: If `-key1` isn't present in the [switch mappings](#) given to the configuration provider, a `FormatException` is thrown.

Duplicate keys

If duplicate keys are provided, the last key-value pair is used.

Switch mappings

When manually building configuration with `ConfigurationBuilder`, you can optionally provide a switch mappings dictionary to the `AddCommandLine` method. Switch mappings allow you to provide key name replacement logic.

When the switch mappings dictionary is used, the dictionary is checked for a key that matches the key provided by a command-line argument. If the command-line key is found in the dictionary, the dictionary value (the key replacement) is passed back to set the configuration. A switch mapping is required for any command-line key prefixed with a single dash (`-`).

Switch mappings dictionary key rules:

- Switches must start with a dash (`-`) or double-dash (`--`).
- The switch mappings dictionary must not contain duplicate keys.

In the following example, the `GetSwitchMappings` method allows your command-line arguments to use a single dash (`-`) key prefix and avoid leading subkey prefixes.

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfigurationRoot Configuration { get; set; }

    public static Dictionary<string, string> GetSwitchMappings(
        IReadOnlyDictionary<string, string> configurationStrings)
    {
        return configurationStrings.Select(item =>
            new KeyValuePair<string, string>(
                "-" + item.Key.Substring(item.Key.LastIndexOf('.') + 1),
                item.Key))
            .ToDictionary(
                item => item.Key, item => item.Value);
    }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "RickPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args, GetSwitchMappings(dict));

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

Without providing command-line arguments, the dictionary provided to `AddInMemoryCollection` sets the configuration values. Run the app with the following command:

```
dotnet run
```

The console window displays:

```
MachineName: RickPC
Left: 1980
```

Use the following to pass in configuration settings:

```
dotnet run /Profile:MachineName=DahliaPC /App:MainWindow:Left=1984
```

The console window displays:

```
MachineName: DahliaPC
Left: 1984
```

After the switch mappings dictionary is created, it contains the data shown in the following table.

KEY	VALUE
<code>-MachineName</code>	<code>Profile:MachineName</code>
<code>-Left</code>	<code>App:MainWindow:Left</code>

To demonstrate key switching using the dictionary, run the following command:

```
dotnet run -MachineName=ChadPC -Left=1988
```

The command-line keys are swapped. The console window displays the configuration values for

`Profile:MachineName` and `App:MainWindow:Left` :

```
MachineName: ChadPC
Left: 1988
```

The web.config file

A *web.config* file is required when hosting the app in IIS or IIS Express. Settings in *web.config* enable the [ASP.NET Core Module](#) to launch the app and configure other IIS settings and modules. If the *web.config* file isn't present and the project file includes `<Project Sdk="Microsoft.NET.Sdk.Web">`, publishing the project creates a *web.config* file in the published output (the *publish* folder). For more information, see [Host ASP.NET Core on Windows with IIS](#).

Additional notes

- Dependency Injection (DI) is not set up until after `ConfigureServices` is invoked.
- The configuration system is not DI aware.
- `IConfiguration` has two specializations:
 - `IConfigurationRoot` Used for the root node. Can trigger a reload.
 - `IConfigurationSection` Represents a section of configuration values. The `GetSection` and `GetChildren` methods return an `IConfigurationSection`.

Additional resources

- [Options](#)
- [Working with Multiple Environments](#)
- [Safe storage of app secrets during development](#)
- [Hosting in ASP.NET Core](#)
- [Dependency Injection](#)
- [Azure Key Vault configuration provider](#)

Options pattern in ASP.NET Core

11/29/2017 • 9 min to read • [Edit Online](#)

By [Luke Latham](#)

The options pattern uses options classes to represent groups of related settings. When configuration settings are isolated by feature into separate options classes, the app adheres to two important software engineering principles:

- The [Interface Segregation Principle \(ISP\)](#): Features (classes) that depend on configuration settings depend only on the configuration settings that they use.
- [Separation of Concerns](#): Settings for different parts of the app aren't dependent or coupled to one another.

[View or download sample code \(how to download\)](#) This article is easier to follow with the sample app.

Basic options configuration

Basic options configuration is demonstrated as Example #1 in the [sample app](#).

An options class must be non-abstract with a public parameterless constructor. The following class, `MyOptions`, has two properties, `Option1` and `Option2`. Setting default values is optional, but the class constructor in the following example sets the default value of `Option1`. `Option2` has a default value set by initializing the property directly (*Models/MyOptions.cs*):

```
public class MyOptions
{
    public MyOptions()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

The `MyOptions` class is added to the service container with `IConfigureOptions<TOptions>` and bound to configuration:

```
// Example #1: Basic options
// Register the ConfigurationBuilder instance which MyOptions binds against.
services.Configure<MyOptions>(Configuration);
```

The following page model uses [constructor dependency injection](#) with `IOptions<TOptions>` to access the settings (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _options;
```

```

public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}

```

```

// Example #1: Simple options
var option1 = _options.Option1;
var option2 = _options.Option2;
SimpleOptions = $"option1 = {option1}, option2 = {option2}";

```

The sample's *appsettings.json* file specifies values for `option1` and `option2`:

```

{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  }
}

```

When the app is run, the page model's `OnGet` method returns a string showing the option class values:

```

option1 = value1_from_json, option2 = -1

```

Configure simple options with a delegate

Configuring simple options with a delegate is demonstrated as Example #2 in the [sample app](#).

Use a delegate to set options values. The sample app uses the `MyOptionsWithDelegateConfig` class (*Models/MyOptionsWithDelegateConfig.cs*):

```

public class MyOptionsWithDelegateConfig
{
    public MyOptionsWithDelegateConfig()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}

```

In the following code, a second `IConfigureOptions<TOptions>` service is added to the service container. It uses a delegate to configure the binding with `MyOptionsWithDelegateConfig`:

```
// Example #2: Options bound and configured by a delegate
services.Configure<MyOptionsWithDelegateConfig>(myOptions =>
{
    myOptions.Option1 = "value1_configured_by_delegate";
    myOptions.Option2 = 500;
});
```

Index.cshhtml.cs:

```
private readonly MyOptionsWithDelegateConfig _optionsWithDelegateConfig;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #2: Options configured by delegate
var delegate_config_option1 = _optionsWithDelegateConfig.Option1;
var delegate_config_option2 = _optionsWithDelegateConfig.Option2;
SimpleOptionsWithDelegateConfig =
    $"delegate_option1 = {delegate_config_option1}, " +
    $"delegate_option2 = {delegate_config_option2}";
```

You can add multiple configuration providers. Configuration providers are available in NuGet packages. They're applied in order that they're registered.

Each call to `Configure<TOptions>` adds an `IConfigureOptions<TOptions>` service to the service container. In the preceding example, the values of `Option1` and `Option2` are both specified in `appsettings.json`, but the values of `Option1` and `Option2` are overridden by the configured delegate.

When more than one configuration service is enabled, the last configuration source specified *wins* and sets the configuration value. When the app is run, the page model's `OnGet` method returns a string showing the option class values:

```
delegate_option1 = value1_configured_by_delegate, delegate_option2 = 500
```

Suboptions configuration

Suboptions configuration is demonstrated as Example #3 in the [sample app](#).

Apps should create options classes that pertain to specific feature groups (classes) in the app. Parts of the app that require configuration values should only have access to the configuration values that they use.

When binding options to configuration, each property in the options type is bound to a configuration key of the form `property[:sub-property:]`. For example, the `MyOptions.Option1` property is bound to the key `Option1`,

which is read from the `option1` property in `appsettings.json`.

In the following code, a third `IConfigureOptions<TOptions>` service is added to the service container. It binds `MySubOptions` to the section `subsection` of the `appsettings.json` file:

```
// Example #3: Sub-options
// Bind options using a sub-section of the appsettings.json file.
services.Configure<MySubOptions>(Configuration.GetSection("subsection"));
```

The `GetSection` extension method requires the `Microsoft.Extensions.Options.ConfigurationExtensions` NuGet package. If the app uses the `Microsoft.AspNetCore.All` metapackage, the package is automatically included.

The sample's `appsettings.json` file defines a `subsection` member with keys for `suboption1` and `suboption2`:

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  }
}
```

The `MySubOptions` class defines properties, `SubOption1` and `SubOption2`, to hold the sub-option values (`Models/MySubOptions.cs`):

```
public class MySubOptions
{
    public MySubOptions()
    {
        // Set default values.
        SubOption1 = "value1_from_ctor";
        SubOption2 = 5;
    }

    public string SubOption1 { get; set; }
    public int SubOption2 { get; set; }
}
```

The page model's `OnGet` method returns a string with the sub-option values (`Pages/Index.cshtml.cs`):

```
private readonly MySubOptions _subOptions;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #3: Sub-options
var subOption1 = _subOptions.SubOption1;
var subOption2 = _subOptions.SubOption2;
SubOptions = $"subOption1 = {subOption1}, subOption2 = {subOption2}";
```

When the app is run, the `OnGet` method returns a string showing the sub-option class values:

```
subOption1 = subvalue1_from_json, subOption2 = 200
```

Options provided by a view model or with direct view injection

Options provided by a view model or with direct view injection is demonstrated as Example #4 in the [sample app](#).

Options can be supplied in a view model or by injecting `IOptions<TOptions>` directly into a view (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #4: Bind options directly to the page
MyOptions = _options;
```

For direct injection, inject `IOptions<MyOptions>` with an `@inject` directive:

```
@page
@model IndexModel
@using Microsoft.Extensions.Options
@using UsingOptionsSample.Models
@inject IOptions<MyOptions> OptionsAccessor
@{
    ViewData["Title"] = "Using Options Sample";
}

<h1>@ViewData["Title"]</h1>
```

When the app is run, the option values are shown in the rendered page:

Example #4: Model and injected options

Options provided by the model

Options provided by the model: `@Model.MyOptions.Option1` and `@Model.MyOptions.Option2`

Option1: value1_from_json

Option2: -1

Options injected into the page

Options injected into the page: `@inject IOptions<MyOptions> OptionsAccessor` with `@OptionsAccessor.Value.Option1` and `@OptionsAccessor.Value.Option2`

Option1: value1_from_json

Option2: -1

Reload configuration data with IOptionsSnapshot

Reloading configuration data with `IOptionsSnapshot` is demonstrated in Example #5 in the [sample app](#).

Requires ASP.NET Core 1.1 or later.

`IOptionsSnapshot` supports reloading options with minimal processing overhead. In ASP.NET Core 1.1, `IOptionsSnapshot` is a snapshot of `IOptionsMonitor<TOptions>` and updates automatically whenever the monitor triggers changes based on the data source changing. In ASP.NET Core 2.0 and later, options are computed once per request when accessed and cached for the lifetime of the request.

The following example demonstrates how a new `IOptionsSnapshot` is created after `appsettings.json` changes (`Pages/Index.cshtml.cs`). Multiple requests to the server return constant values provided by the `appsettings.json` file until the file is changed and configuration reloads.

```
private readonly MyOptions _snapshotOptions;
```

```
public IndexModel(  
    IOptions<MyOptions> optionsAccessor,  
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,  
    IOptions<MySubOptions> subOptionsAccessor,  
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,  
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)  
{  
    _options = optionsAccessor.Value;  
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;  
    _subOptions = subOptionsAccessor.Value;  
    _snapshotOptions = snapshotOptionsAccessor.Value;  
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");  
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");  
}
```

```
// Example #5: Snapshot options  
var snapshotOption1 = _snapshotOptions.Option1;  
var snapshotOption2 = _snapshotOptions.Option2;  
SnapshotOptions =  
    $"snapshot option1 = {snapshotOption1}, " +  
    $"snapshot option2 = {snapshotOption2}";
```

The following image shows the initial `option1` and `option2` values loaded from the `appsettings.json` file:

```
snapshot option1 = value1_from_json, snapshot option2 = -1
```

Change the values in the `appsettings.json` file to `value1_from_json UPDATED` and `200`. Save the `appsettings.json` file. Refresh the browser to see that the options values are updated:

```
snapshot option1 = value1_from_json UPDATED, snapshot option2 = 200
```

Named options support with IConfigurationNamedOptions

Named options support with [IConfigurationNamedOptions](#) is demonstrated as Example #6 in the [sample app](#).

Requires ASP.NET Core 2.0 or later.

Named options support allows the app to distinguish between named options configurations. In the sample app, named options are declared with the [ConfigureNamedOptions<TOptions>.Configure](#) method:

```
// Example #6: Named options (named_options_1)
// Register the ConfigurationBuilder instance which MyOptions binds against.
// Specify that the options loaded from configuration are named
// "named_options_1".
services.Configure<MyOptions>("named_options_1", Configuration);

// Example #6: Named options (named_options_2)
// Specify that the options loaded from the MyOptions class are named
// "named_options_2".
// Use a delegate to configure option values.
services.Configure<MyOptions>("named_options_2", myOptions =>
{
    myOptions.Option1 = "named_options_2_value1_from_action";
});
```

The sample app accesses the named options with [IOptionsSnapshot<TOptions>.Get](#) (`Pages/Index.cshtml.cs`):

```
private readonly MyOptions _named_options_1;
private readonly MyOptions _named_options_2;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #6: Named options
var named_options_1 =
    $"named_options_1: option1 = {_named_options_1.Option1}, " +
    $"option2 = {_named_options_1.Option2}";
var named_options_2 =
    $"named_options_2: option1 = {_named_options_2.Option1}, " +
    $"option2 = {_named_options_2.Option2}";
NamedOptions = $"{named_options_1} {named_options_2}";
```

Running the sample app, the named options are returned:

```
named_options_1: option1 = value1_from_json, option2 = -1
named_options_2: option1 = named_options_2_value1_from_action, option2 = 5
```

`named_options_1` values are provided from configuration, which are loaded from the `appsettings.json` file.

`named_options_2` values are provided by:

- The `named_options_2` delegate in `ConfigureServices` for `Option1`.
- The default value for `Option2` provided by the `MyOptions` class.

Configure all named options instances with the `OptionsServiceCollectionExtensions.ConfigureAll` method. The following code configures `Option1` for all named configuration instances with a common value. Add the following code manually to the `Configure` method:

```
services.ConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "ConfigureAll replacement value";
});
```

Running the sample app after adding the code produces the following result:

```
named_options_1: option1 = ConfigureAll replacement value, option2 = -1
named_options_2: option1 = ConfigureAll replacement value, option2 = 5
```

NOTE

In ASP.NET Core 2.0 and later, all options are named instances. Existing `IConfigureOption` instances are treated as targeting the `Options.DefaultName` instance, which is `string.Empty`. `IConfigureNamedOptions` also implements `IConfigureOptions`. The default implementation of the `IOptionsFactory<TOptions>` ([reference source](#)) has logic to use each appropriately. The `null` named option is used to target all of the named instances instead of a specific named instance (`ConfigureAll` and `PostConfigureAll` use this convention).

IPostConfigureOptions

Requires ASP.NET Core 2.0 or later.

Set postconfiguration with `IPostConfigureOptions<TOptions>`. Postconfiguration runs after all `IConfigureOptions<TOptions>` configuration occurs:

```
services.PostConfigure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

[PostConfigure<TOptions>](#) is available to post-configure named options:

```
services.PostConfigure<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Use [PostConfigureAll<TOptions>](#) to post-configure all named configuration instances:

```
services.PostConfigureAll<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Options factory, monitoring, and cache

[IOptionsMonitor](#) is used for notifications when [TOptions](#) instances change. [IOptionsMonitor](#) supports reloadable options, change notifications, and [IPostConfigureOptions](#).

[IOptionsFactory<TOptions>](#) (ASP.NET Core 2.0 or later) is responsible for creating new options instances. It has a single [Create](#) method. The default implementation takes all registered [IConfigureOptions](#) and [IPostConfigureOptions](#) and runs all the configures first, followed by the post-configures. It distinguishes between [IConfigureNamedOptions](#) and [IConfigureOptions](#) and only calls the appropriate interface.

[IOptionsMonitorCache<TOptions>](#) (ASP.NET Core 2.0 or later) is used by [IOptionsMonitor](#) to cache [TOptions](#) instances. The [IOptionsMonitorCache](#) invalidates options instances in the monitor so that the value is recomputed ([TryRemove](#)). Values can be manually introduced as well with [TryAdd](#). The [Clear](#) method is used when all named instances should be recreated on demand.

See also

- [Configuration](#)

Introduction to logging in ASP.NET Core

1/10/2018 • 23 min to read • [Edit Online](#)

By [Steve Smith](#) and [Tom Dykstra](#)

ASP.NET Core supports a logging API that works with a variety of logging providers. Built-in providers let you send logs to one or more destinations, and you can plug in a third-party logging framework. This article shows how to use the built-in logging API and providers in your code.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

[View or download sample code](#) ([how to download](#))

How to create logs

To create logs, get an `ILogger` object from the [dependency injection](#) container:

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
}
```

Then call logging methods on that logger object:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

This example creates logs with the `TodoController` class as the *category*. Categories are explained [later in this article](#).

ASP.NET Core does not provide async logger methods because logging should be so fast that it isn't worth the cost of using async. If you're in a situation where that's not true, consider changing the way you log. If your data store is slow, write the log messages to a fast store first, then move them to a slow store later. For example, log to a message queue that is read and persisted to slow storage by another process.

How to add providers

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

A logging provider takes the messages that you create with an `ILogger` object and displays or stores them. For example, the Console provider displays messages on the console, and the Azure App Service provider can store them in Azure blob storage.

To use a provider, call the provider's `Add<ProviderName>` extension method in *Program.cs*:

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange:
true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

The default project template enables logging with the `CreateDefaultBuilder` method:

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
```

You'll find information about each [built-in logging provider](#) and links to [third-party logging providers](#) later in the article.

Sample logging output

With the sample code shown in the preceding section, you'll see logs in the console when you run from the command line. Here's an example of console output:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/0
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TODOController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TODOController[1002]
      Getting item 0
warn: TodoApi.Controllers.TODOController[4000]
      GetById(0) NOT FOUND
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TODOController.GetById (TodoApi) in 42.9286ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 148.889ms 404
```

These logs were created by going to `http://localhost:5000/api/todo/0`, which triggers execution of both `ILogger` calls shown in the preceding section.

Here's an example of the same logs as they appear in the Debug window when you run the sample application in Visual Studio:

```
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request starting HTTP/1.1 GET
http://localhost:53104/api/todo/0
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executing action method
TodoApi.Controllers.TODOController.GetById (TodoApi) with arguments (0) - ModelState is Valid
TodoApi.Controllers.TODOController:Information: Getting item 0
TodoApi.Controllers.TODOController:Warning: GetById(0) NOT FOUND
Microsoft.AspNetCore.Mvc.StatusCodeResult:Information: Executing HttpStatusCodeResult, setting HTTP status
code 404
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executed action
TodoApi.Controllers.TODOController.GetById (TodoApi) in 152.5657ms
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request finished in 316.3195ms 404
```

The logs that were created by the `ILogger` calls shown in the preceding section begin with "TodoApi.Controllers.TODOController". The logs that begin with "Microsoft" categories are from ASP.NET Core. ASP.NET Core itself and your application code are using the same logging API and the same logging providers.

The remainder of this article explains some details and options for logging.

NuGet packages

The `ILogger` and `ILoggerFactory` interfaces are in [Microsoft.Extensions.Logging.Abstractions](#), and default implementations for them are in [Microsoft.Extensions.Logging](#).

Log category

A *category* is included with each log that you create. You specify the category when you create an `ILogger` object. The category may be any string, but a convention is to use the fully qualified name of the class from which the logs are written. For example: "TodoApi.Controllers.TODOController".

You can specify the category as a string or use an extension method that derives the category from the type. To specify the category as a string, call `CreateLogger` on an `ILoggerFactory` instance, as shown below.

```

public class TodoController : Controller
{
    private readonly IToDoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(IToDoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApi.Controllers.TodoController");
    }
}

```

Most of the time, it will be easier to use `ILogger<T>`, as in the following example.

```

public class TodoController : Controller
{
    private readonly IToDoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(IToDoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
}

```

This is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

Each time you write a log, you specify its [LogLevel](#). The log level indicates the degree of severity or importance. For example, you might write an `Information` log when a method ends normally, a `Warning` log when a method returns a 404 return code, and an `Error` log when you catch an unexpected exception.

In the following code example, the names of the methods (for example, `LogWarning`) specify the log level. The first parameter is the [Log event ID](#). The second parameter is a [message template](#) with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in more detail later in this article.

```

public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}

```

Log methods that include the level in the method name are [extension methods for ILogger](#). Behind the scenes, these methods call a `Log` method that takes a `LogLevel` parameter. You can call the `Log` method directly rather than one of these extension methods, but the syntax is relatively complicated. For more information, see the [ILogger interface](#) and the [logger extensions source code](#).

ASP.NET Core defines the following [log levels](#), ordered here from least to highest severity.

- Trace = 0

For information that is valuable only to a developer debugging an issue. These messages may contain sensitive application data and so should not be enabled in a production environment. *Disabled by default.*

Example: `Credentials: {"User": "someuser", "Password": "P@ssword"}`

- Debug = 1

For information that has short-term usefulness during development and debugging. Example:

`Entering method Configure with flag set to true.` You typically would not enable `Debug` level logs in production unless you are troubleshooting, due to the high volume of logs.

- Information = 2

For tracking the general flow of the application. These logs typically have some long-term value. Example:

`Request received for path /api/todo`

- Warning = 3

For abnormal or unexpected events in the application flow. These may include errors or other conditions that do not cause the application to stop, but which may need to be investigated. Handled exceptions are a common place to use the `Warning` log level. Example: `FileNotFoundException for file quotes.txt.`

- Error = 4

For errors and exceptions that cannot be handled. These messages indicate a failure in the current activity or operation (such as the current HTTP request), not an application-wide failure. Example log message:

`Cannot insert record due to duplicate key violation.`

- Critical = 5

For failures that require immediate attention. Examples: data loss scenarios, out of disk space.

You can use the log level to control how much log output is written to a particular storage medium or display window. For example, in production you might want all logs of `Information` level and lower to go to a volume data store, and all logs of `Warning` level and higher to go to a value data store. During development, you might normally send logs of `Warning` or higher severity to the console. Then when you need to troubleshoot, you can add `Debug` level. The [Log filtering](#) section later in this article explains how to control which log levels a provider handles.

The ASP.NET Core framework writes `Debug` level logs for framework events. The log examples earlier in this article excluded logs below `Information` level, so no `Debug` level logs were shown. Here's an example of console logs if you run the sample application configured to show `Debug` and higher logs for the console provider.

```

info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:62555/api/todo/0
debug: Microsoft.AspNetCore.Routing.Tree.TreeRouter[1]
      Request successfully matched the route with name 'GetTodo' and template 'api/ToDo/{id}'.
debug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'ToDoApi.Controllers.ToDoController.Update (ToDoApi)' with id '089d59b6-92ec-472d-b552-cc613dfd625d' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
debug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'ToDoApi.Controllers.ToDoController.Delete (ToDoApi)' with id 'f3476abe-4bd9-4ad3-9261-3ead09607366' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
debug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action ToDoApi.Controllers.ToDoController.GetById (ToDoApi)
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method ToDoApi.Controllers.ToDoController.GetById (ToDoApi) with arguments (0) - ModelState is Valid
info: ToDoApi.Controllers.ToDoController[1002]
      Getting item 0
warn: ToDoApi.Controllers.ToDoController[4000]
      GetById(0) NOT FOUND
debug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action method ToDoApi.Controllers.ToDoController.GetById (ToDoApi), returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action ToDoApi.Controllers.ToDoController.GetById (ToDoApi) in 0.8788ms
debug: Microsoft.AspNetCore.Server.Kestrel[9]
      Connection id "0HL6L7NEFF2QD" completed keep alive response.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2.7286ms 404

```

Log event ID

Each time you write a log, you can specify an *event ID*. The sample app does this by using a locally-defined

`LoggingEvents` class:

```

public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}

```

```

public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}

```

An event ID is an integer value that you can use to associate a set of logged events with one another. For instance, a log for adding an item to a shopping cart could be event ID 1000 and a log for completing a purchase could be event ID 1001.

In logging output, the event ID may be stored in a field or included in the text message, depending on the provider. The Debug provider doesn't show event IDs, but the console provider shows them in brackets after the category:

```
info: TodoApi.Controllers.TODOController[1002]
      Getting item invaldid
warn: TodoApi.Controllers.TODOController[4000]
      GetById(invaldid) NOT FOUND
```

Log message template

Each time you write a log message, you provide a message template. The message template can be a string or it can contain named placeholders into which argument values are placed. The template isn't a format string, and placeholders should be named, not numbered.

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

The order of placeholders, not their names, determines which parameters are used to provide their values. If you have the following code:

```
string p1 = "parm1";
string p2 = "parm2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

The resulting log message looks like this:

```
Parameter values: parm1, parm2
```

The logging framework does message formatting in this way to make it possible for logging providers to implement [semantic logging, also known as structured logging](#). Because the arguments themselves are passed to the logging system, not just the formatted message template, logging providers can store the parameter values as fields in addition to the message template. If you're directing your log output to Azure Table Storage and your logger method call looks like this:

```
_logger.LogInformation("Getting item {ID} at {RequestTime}", id, DateTime.Now);
```

Each Azure Table entity can have `ID` and `RequestTime` properties, which simplifies queries on log data. You can find all logs within a particular `RequestTime` range without the need to parse the time out of the text message.

Logging exceptions

The logger methods have overloads that let you pass in an exception, as in the following example:

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({ID}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```

Different providers handle the exception information in different ways. Here's an example of Debug provider output from the code shown above.

```
TodoApi.Controllers.TODOController:Warning: GetById(036dd898-fb01-47e8-9a65-f92eb73cf924) NOT FOUND

System.Exception: Item not found exception.
   at TodoApi.Controllers.TODOController.GetById(String id) in
   C:\logging\sample\src\TodoApi\Controllers\TODOController.cs:line 226
```

Log filtering

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

You can specify a minimum log level for a specific provider and category or for all providers or all categories. Any logs below the minimum level aren't passed to that provider, so they don't get displayed or stored.

If you want to suppress all logs, you can specify `LogLevel.None` as the minimum log level. The integer value of `LogLevel.None` is 6, which is higher than `LogLevel.Critical` (5).

Create filter rules in configuration

The project templates create code that calls `CreateDefaultBuilder` to set up logging for the Console and Debug providers. The `CreateDefaultBuilder` method also sets up logging to look for configuration in a `Logging` section, using code like the following:

```

public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange:
true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}

```

The configuration data specifies minimum log levels by provider and category, as in the following example:

```

{
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}

```

This JSON creates six filter rules, one for the Debug provider, four for the Console provider, and one that applies to all providers. You'll see later how just one of these rules is chosen for each provider when an `ILogger` object is created.

Filter rules in code

You can register filter rules in code, as shown in the following example:

```

WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("System", LogLevel.Debug)
            .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace))
    .Build();

```

The second `AddFilter` specifies the Debug provider by using its type name. The first `AddFilter` applies to all providers because it doesn't specify a provider type.

How filtering rules are applied

The configuration data and the `AddFilter` code shown in the preceding examples create the rules shown in the following table. The first six come from the configuration example and the last two come from the code example.

NUMBER	PROVIDER	CATEGORIES THAT BEGIN WITH ...	MINIMUM LOG LEVEL
1	Debug	All categories	Information
2	Console	Microsoft.AspNetCore.Mvc. Razor.Internal	Warning
3	Console	Microsoft.AspNetCore.Mvc. Razor.Razor	Debug
4	Console	Microsoft.AspNetCore.Mvc. Razor	Error
5	Console	All categories	Information
6	All providers	All categories	Debug
7	All providers	System	Debug
8	Debug	Microsoft	Trace

When you create an `ILogger` object to write logs with, the `ILoggerFactory` object selects a single rule per provider to apply to that logger. All messages written by that `ILogger` object are filtered based on the selected rules. The most specific rule possible for each provider and category pair is selected from the available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If none are found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If none are found, select all rules that don't specify a category.
- If multiple rules are selected take the **last** one.
- If no rules are selected, use `MinimumLevel`.

For example, suppose you have the preceding list of rules and you create an `ILogger` object for category "Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine":

- For the Debug provider, rules 1, 6, and 8 apply. Rule 8 is most specific, so that's the one selected.
- For the Console provider, rules 3, 4, 5, and 6 apply. Rule 3 is most specific.

When you create logs with an `ILogger` for category "Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine", logs

of `Trace` level and above will go to the Debug provider, and logs of `Debug` level and above will go to the Console provider.

Provider aliases

You can use the type name to specify a provider in configuration, but each provider defines a shorter *alias* that is easier to use. For the built-in providers, use the following aliases:

- Console
- Debug
- EventLog
- AzureAppServices
- TraceSource
- EventSource

Default minimum level

There is a minimum level setting that takes effect only if no rules from configuration or code apply for a given provider and category. The following example shows how to set the minimum level:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning))
    .Build();
```

If you don't explicitly set the minimum level, the default value is `Information`, which means that `Trace` and `Debug` logs are ignored.

Filter functions

You can write code in a filter function to apply filtering rules. A filter function is invoked for all providers and categories that do not have rules assigned to them by configuration or code. Code in the function has access to the provider type, category, and log level to decide whether or not a message should be logged. For example:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logBuilder =>
    {
        logBuilder.AddFilter((provider, category, logLevel) =>
        {
            if (provider == "Microsoft.Extensions.Logging.Console.ConsoleLoggerProvider" &&
                category == "TodoApi.Controllers.TODOController")
            {
                return false;
            }
            return true;
        });
    })
    .Build();
```

Log scopes

You can group a set of logical operations within a *scope* in order to attach the same data to each log that is created as part of that set. For example, you might want every log created as part of processing a transaction to include the transaction ID.

A scope is an `IDisposable` type that is returned by the `ILogger.BeginScope<TState>` method and lasts until it is disposed. You use a scope by wrapping your logger calls in a `using` block, as shown here:

```

public IActionResult GetById(string id)
{
    TodoItem item;
    using (_logger.BeginScope("Message attached to logs created in the using block"))
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
        item = _todoRepository.Find(id);
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
            return NotFound();
        }
    }
    return new ObjectResult(item);
}

```

The following code enables scopes for the console provider:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

In *Program.cs*:

```

.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})

```

NOTE

Configuring the `IncludeScopes` console logger option is required to enable scope-based logging. Configuration of `IncludeScopes` using *appsettings* configuration files will be available with the release of ASP.NET Core 2.1.

Each log message includes the scoped information:

```

info: TodoApi.Controllers.TodoController[1002]
    => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
    Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
    => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
    GetById(0) NOT FOUND

```

Built-in logging providers

ASP.NET Core ships the following providers:

- [Console](#)
- [Debug](#)
- [EventSource](#)
- [EventLog](#)
- [TraceSource](#)
- [Azure App Service](#)

The console provider

The [Microsoft.Extensions.Logging.Console](#) provider package sends log output to the console.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
logging.AddConsole()
```

The Debug provider

The [Microsoft.Extensions.Logging.Debug](#) provider package writes log output by using the [System.Diagnostics.Debug](#) class (`Debug.WriteLine` method calls).

On Linux, this provider writes logs to `/var/log/message`.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
logging.AddDebug()
```

The EventSource provider

For apps that target ASP.NET Core 1.1.0 or higher, the [Microsoft.Extensions.Logging.EventSource](#) provider package can implement event tracing. On Windows, it uses [ETW](#). The provider is cross-platform, but there are no event collection and display tools yet for Linux or macOS.

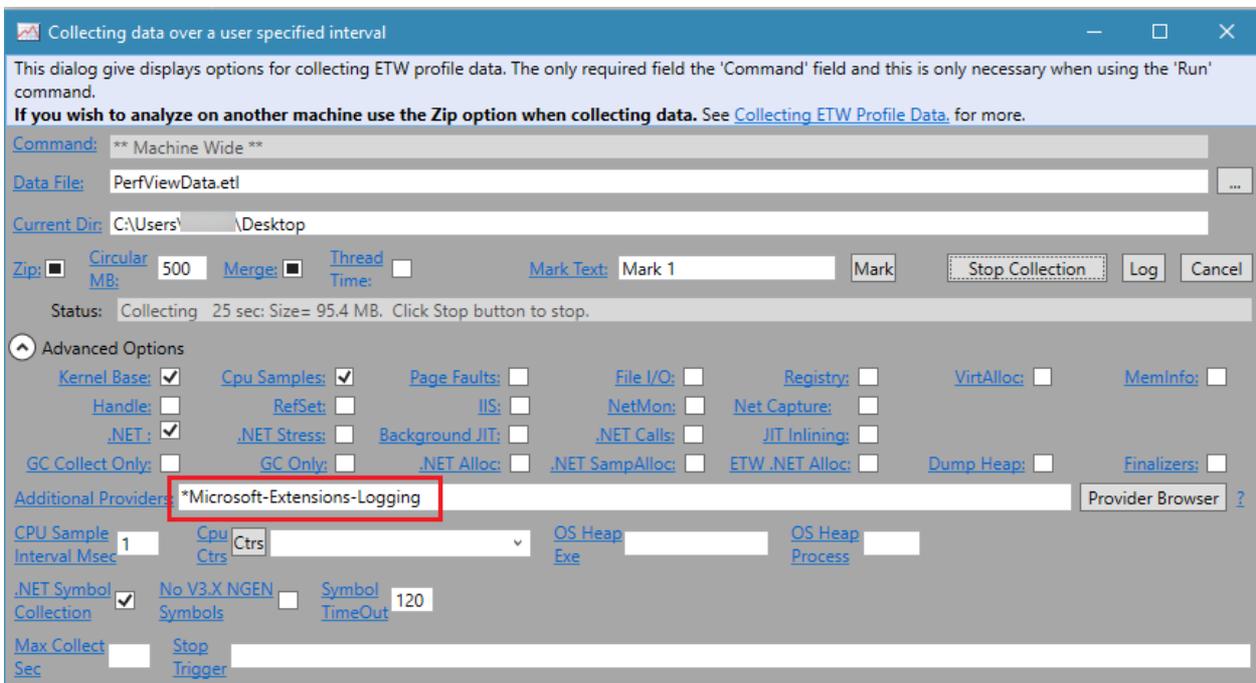
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
logging.AddEventSourceLogger()
```

A good way to collect and view logs is to use the [PerfView utility](#). There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET.

To configure PerfView for collecting events logged by this provider, add the string

`*Microsoft-Extensions-Logging` to the **Additional Providers** list. (Don't miss the asterisk at the start of the string.)



Capturing events on Nano Server requires some additional setup:

- Connect PowerShell remoting to the Nano Server:

```
Enter-PSSession [name]
```

- Create an ETW session:

```
New-EtwTraceSession -Name "MyAppTrace" -LocalFilePath C:\trace.etl
```

- Add ETW providers for CLR, ASP.NET Core, and others as needed. The ASP.NET Core provider GUID is

```
3ac73b97-af73-50e9-0822-5da4367920d0 .
```

```
Add-EtwTraceProvider -Guid "{e13c0d23-ccbc-4e12-931b-d9cc2eee27e4}" -SessionName MyAppTrace
Add-EtwTraceProvider -Guid "{3ac73b97-af73-50e9-0822-5da4367920d0}" -SessionName MyAppTrace
```

- Run the site and do whatever actions you want tracing information for.
- Stop the tracing session when you're finished:

```
Stop-EtwTraceSession -Name "MyAppTrace"
```

The resulting `C:\trace.etl` file can be analyzed with PerfView as on other editions of Windows.

The Windows EventLog provider

The `Microsoft.Extensions.Logging.EventLog` provider package sends log output to the Windows Event Log.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
logging.AddEventLog()
```

The TraceSource provider

The [Microsoft.Extensions.Logging.TraceSource](#) provider package uses the [System.Diagnostics.TraceSource](#) libraries and providers.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
logging.AddTraceSource(sourceSwitchName);
```

[AddTraceSource overloads](#) let you pass in a source switch and a trace listener.

To use this provider, an application has to run on the .NET Framework (rather than .NET Core). The provider lets you route messages to a variety of [listeners](#), such as the [TextWriterTraceListener](#) used in the sample application.

The following example configures a `TraceSource` provider that logs `Warning` and higher messages to the console window.

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddDebug();

    // add Trace Source logging
    var testSwitch = new SourceSwitch("sourceSwitch", "Logging Sample");
    testSwitch.Level = SourceLevels.Warning;
    loggerFactory.AddTraceSource(testSwitch,
        new TextWriterTraceListener(writer: Console.Out));
}
```

The Azure App Service provider

The [Microsoft.Extensions.Logging.AzureAppServices](#) provider package writes logs to text files in an Azure App Service app's file system and to [blob storage](#) in an Azure Storage account. The provider is available only for apps that target ASP.NET Core 1.1.0 or higher.

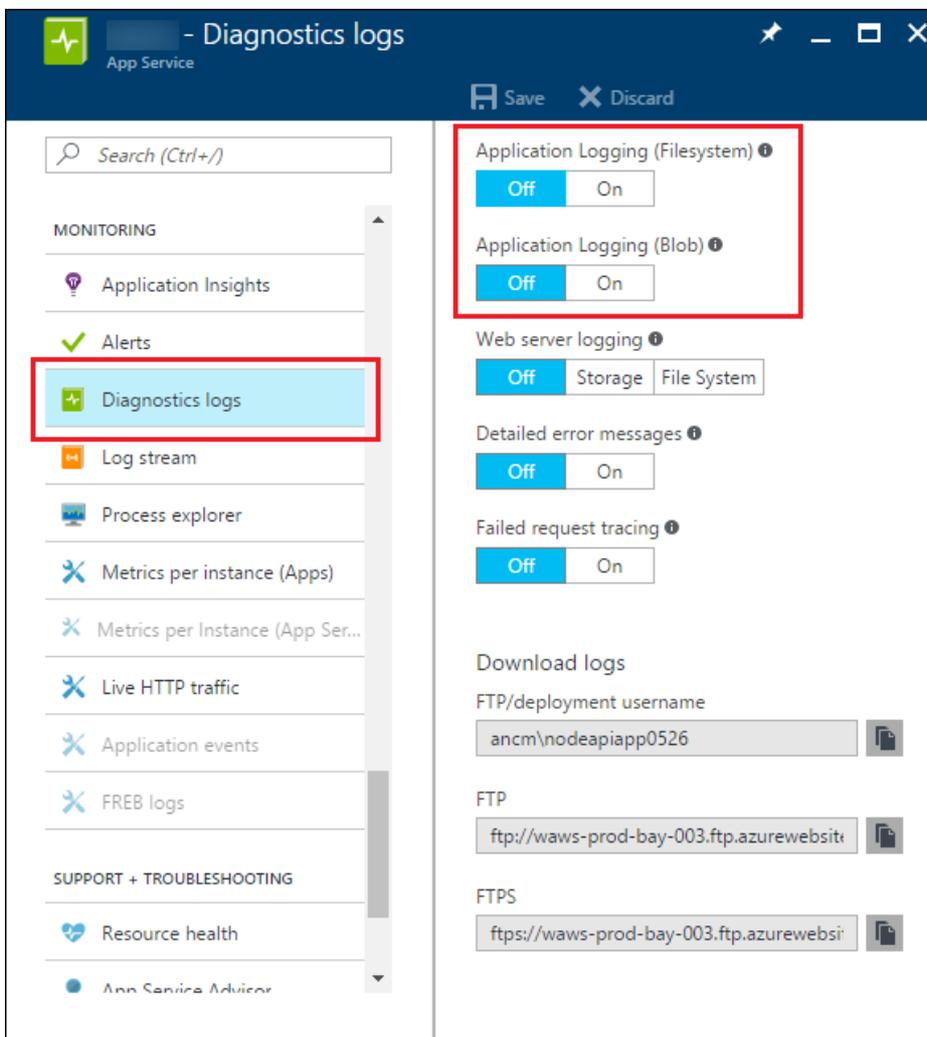
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

If targeting .NET Core, you don't have to install the provider package or explicitly call `AddAzureWebAppDiagnostics`. The provider is automatically available to your app when you deploy the app to Azure App Service.

If targeting .NET Framework, add the provider package to your project and invoke `AddAzureWebAppDiagnostics`:

```
logging.AddAzureWebAppDiagnostics();
```

When you deploy to an App Service app, your application honors the settings in the [Diagnostic Logs](#) section of the **App Service** page of the Azure portal. When you change those settings, the changes take effect immediately without requiring that you restart the app or redeploy code to it.



The default location for log files is in the `D:\home\LogFiles\Application` folder, and the default file name is `diagnostics-yyyymmdd.txt`. The default file size limit is 10 MB, and the default maximum number of files retained is 2. The default blob name is `{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt`. For more information about default behavior, see [AzureAppServicesDiagnosticsSettings](#).

The provider only works when your project runs in the Azure environment. It has no effect when you run locally — it does not write to local files or local development storage for blobs.

Third-party logging providers

Here are some third-party logging frameworks that work with ASP.NET Core:

- [elmah.io](#) - provider for the Elmah.io service
- [JSNLog](#) - logs JavaScript exceptions and other client-side events in your server-side log.
- [Loggr](#) - provider for the Loggr service
- [NLog](#) - provider for the NLog library
- [Serilog](#) - provider for the Serilog library

Some third-party frameworks can do [semantic logging](#), also known as [structured logging](#).

Using a third-party framework is similar to using one of the built-in providers: add a NuGet package to your project and call an extension method on `ILoggerFactory`. For more information, see each framework's documentation.

You can create your own custom providers as well, to support other logging frameworks or your own logging

requirements.

Azure log streaming

Azure log streaming enables you to view log activity in real time from:

- The application server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the **Diagnostics Logs** page from your application's portal page
- Set **Application Logging (Filesystem)** to on.

The screenshot shows the 'Diagnostics logs' configuration page in the Azure App Service portal. The left sidebar contains a search bar and various navigation options, with 'Diagnostics logs' highlighted in blue and a red box. The main content area has a 'Save' and 'Discard' button at the top. The configuration is organized into sections: 'Application Logging (Filesystem)' is set to 'On' with a 'Warning' level; 'Application Logging (Blob)' is set to 'Off'; 'Web server logging' is set to 'File System'; 'Detailed error messages' is set to 'On'; and 'Failed request tracing' is set to 'On'. Below these are 'Download logs' settings for FTP and FTPS, each with a text input field and a download icon.

Navigate to the **Log Streaming** page to view application messages. They are logged by application through the `ILogger` interface.

The screenshot shows the Azure App Service Log Stream interface. The left sidebar contains navigation options: Performance test, Resource explorer, Testing in production, Extensions, MOBILE (Easy tables, Easy APIs, Data connections), API (API definition, CORS), MONITORING (Application Insights, Alerts, Diagnostics logs, **Log stream**), Process explorer, SUPPORT + TROUBLESHOOTING (Resource health, App Service Advisor). The main area displays application logs with the following content:

```
Application logs

Connecting...
2017-11-16T18:39:47 Welcome, you are now connected to log-streaming service.
2017-11-16 18:39:04 ~1J GET /api/vfs/site/wwwroot/_=1510857327404&X-ARR-LOG-ID=36b4e37b-e31e-433d-82dd-
(Windows+NT+10.0;+Win64;+x64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 - https://we
ntent/WebsitesIndex?cacheability=3&region=eastus&cacheVersion=0&defaultCloudName=azure&extensionName=WebsitesExtens
uthority=portal.azure.com .scm.azurewebsites.net 200 0 0 966 1697 156
2017-11-16 18:39:06 GET / X-ARR-LOG-ID=784009e2-cecb-4815-968a-f63b9e8879a6 80 - 73.130.153.74 Mozilla
t/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145f0b07798967
websites.net 403 14 0 370 943 296
2017-11-16 18:39:11 ~1 GET /api/siteextensions X-ARR-LOG-ID=7850d378-909d-4b1d-a425-9408dab70910 443 -
ozilla/5.0+(Windows+NT+10.0;+Win64;+x64)+AppleWebKit/537.36+(KHTML,like+Gecko)+Chrome/62.0.3202.94+Safari/537.36
0 499 1190 703
2017-11-16 18:39:39 GET /ticket.html X-ARR-LOG-ID=b7e33a93-f31e-436f-8214-672dfe954c74 80 - 73.130.153
+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145
affle.azurewebsites.net 304 0 0 314 1087 696
2017-11-16 18:39:40 AZURE-RAFFLE GET /ticket.html X-ARR-LOG-ID=70f2946b-4139-4f65-a936-ba437411cdf9 80 - 73.130.153
+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145
bsites.net 304 0 0 314 1087 218
2017-11-16 18:39:40 AZURE-RAFFLE GET /ticket.html X-ARR-LOG-ID=c5f3593f-4d4b-4811-b150-500d58341622 80 - 73.130.153
+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145
affle.azurewebsites.net 304 0 0 314 1087 0
2017-11-16 18:39:54 AZURE-RAFFLE GET /ticket.html X-ARR-LOG-ID=ae4526f8-c8c7-4399-9f19-9c3fdae7cdcb 80 - 73.130.153
+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145
bsites.net 304 0 0 314 1087 15
```

See also

[High-performance logging with LoggerMessage](#)

High-performance logging with LoggerMessage in ASP.NET Core

11/7/2017 • 7 min to read • [Edit Online](#)

By [Luke Latham](#)

`LoggerMessage` features create cacheable delegates that require fewer object allocations and reduced computational overhead than [logger extension methods](#), such as `LogInformation`, `LogDebug`, and `LogError`. For high-performance logging scenarios, use the `LoggerMessage` pattern.

`LoggerMessage` provides the following performance advantages over Logger extension methods:

- Logger extension methods require "boxing" (converting) value types, such as `int`, into `object`. The `LoggerMessage` pattern avoids boxing by using static `Action` fields and extension methods with strongly-typed parameters.
- Logger extension methods must parse the message template (named format string) every time a log message is written. `LoggerMessage` only requires parsing a template once when the message is defined.

[View or download sample code \(how to download\)](#)

The sample app demonstrates `LoggerMessage` features with a basic quote tracking system. The app adds and deletes quotes using an in-memory database. As these operations occur, log messages are generated using the `LoggerMessage` pattern.

LoggerMessage.Define

`Define(LogLevel, EventId, String)` creates an `Action` delegate for logging a message. `Define` overloads permit passing up to six type parameters to a named format string (template).

LoggerMessage.DefineScope

`DefineScope(String)` creates a `Func` delegate for defining a [log scope](#). `DefineScope` overloads permit passing up to three type parameters to a named format string (template).

Message template (named format string)

The string provided to the `Define` and `DefineScope` methods is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend [Pascal casing](#) for placeholder names. For example, `{Count}`, `{FirstName}`.

Implementing LoggerMessage.Define

Each log message is an `Action` held in a static field created by `LoggerMessage.Define`. For example, the sample app creates a field to describe a log message for a GET request for the Index page (*Internal/LoggerExtensions.cs*):

```
private static readonly Action<ILogger, Exception> _indexPathRequested;
```

For the `Action`, specify:

- The log level.
- A unique event identifier ([EventId](#)) with the name of the static extension method.
- The message template (named format string).

A request for the Index page of the sample app sets the:

- Log level to `Information`.
- Event id to `1` with the name of the `IndexPageRequested` method.
- Message template (named format string) to a string.

```
_indexPageRequested = LoggerMessage.Define(
    LogLevel.Information,
    new EventId(1, nameof(IndexPageRequested)),
    "GET request for Index page");
```

Structured logging stores may use the event name when it's supplied with the event id to enrich logging. For example, [Serilog](#) uses the event name.

The `Action` is invoked through a strongly-typed extension method. The `IndexPageRequested` method logs a message for an Index page GET request in the sample app:

```
public static void IndexPageRequested(this ILogger logger)
{
    _indexPageRequested(logger, null);
}
```

`IndexPageRequested` is called on the logger in the `OnGetAsync` method in `Pages/Index.cshtml.cs`:

```
public async Task OnGetAsync()
{
    _logger.IndexPageRequested();

    Quotes = await _db.Quotes.AsNoTracking().ToListAsync();
}
```

Inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[1]
=> RequestId:0HL90M6E7PHK4:00000001 RequestPath:/ => /Index
GET request for Index page
```

To pass parameters to a log message, define up to six types when creating the static field. The sample app logs a string when adding a quote by defining a `string` type for the `Action` field:

```
private static readonly Action<ILogger, string, Exception> _quoteAdded;
```

The delegate's log message template receives its placeholder values from the types provided. The sample app defines a delegate for adding a quote where the quote parameter is a `string`:

```
_quoteAdded = LoggerMessage.Define<string>(
    LogLevel.Information,
    new EventId(2, nameof(QuoteAdded)),
    "Quote added (Quote = '{Quote}')");
```

The static extension method for adding a quote, `QuoteAdded`, receives the quote argument value and passes it to the `Action` delegate:

```
public static void QuoteAdded(this ILogger logger, string quote)
{
    _quoteAdded(logger, quote, null);
}
```

In the Index page's code-behind file (*Pages/Index.cshtml.cs*), `QuoteAdded` is called to log the message:

```
public async Task<IActionResult> OnPostAddQuoteAsync()
{
    _db.Quotes.Add(Quote);
    await _db.SaveChangesAsync();

    _logger.QuoteAdded(Quote.Text);

    return RedirectToPage();
}
```

Inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[2]
      => RequestId:0HL90M6E7PHK5:0000000A RequestPath:/ => /Index
      Quote added (Quote = 'You can avoid reality, but you cannot avoid the consequences of avoiding reality.
- Ayn Rand')
```

The sample app implements a `try - catch` pattern for quote deletion. An informational message is logged for a successful delete operation. An error message is logged for a delete operation when an exception is thrown. The log message for the unsuccessful delete operation includes the exception stack trace (*Internal/LoggerExtensions.cs*):

```
private static readonly Action<ILogger, string, int, Exception> _quoteDeleted;
private static readonly Action<ILogger, int, Exception> _quoteDeleteFailed;
```

```
_quoteDeleted = LoggerMessage.Define<string, int>(
    LogLevel.Information,
    new EventId(4, nameof(QuoteDeleted)),
    "Quote deleted (Quote = '{Quote}' Id = {Id})");

_quoteDeleteFailed = LoggerMessage.Define<int>(
    LogLevel.Error,
    new EventId(5, nameof(QuoteDeleteFailed)),
    "Quote delete failed (Id = {Id})");
```

Note how the exception is passed to the delegate in `QuoteDeleteFailed`:

```
public static void QuoteDeleted(this ILogger logger, string quote, int id)
{
    _quoteDeleted(logger, quote, id, null);
}

public static void QuoteDeleteFailed(this ILogger logger, int id, Exception ex)
{
    _quoteDeleteFailed(logger, id, ex);
}
```

In the Index page code-behind, a successful quote deletion calls the `QuoteDeleted` method on the logger. When a quote isn't found for deletion, an `ArgumentNullException` is thrown. The exception is trapped by the `try - catch` statement and logged by calling the `QuoteDeleteFailed` method on the logger in the `catch` block (*Pages/Index.cshtml.cs*):

```
public async Task<IActionResult> OnPostDeleteQuoteAsync(int id)
{
    var quote = await _db.Quotes.FindAsync(id);

    // DO NOT use this approach in production code!
    // You should check quote to see if it's null before removing
    // it and saving changes to the database. A try-catch is used
    // here for demonstration purposes of LoggerMessage features.
    try
    {
        _db.Quotes.Remove(quote);
        await _db.SaveChangesAsync();

        _logger.QuoteDeleted(quote.Text, id);
    }
    catch (ArgumentNullException ex)
    {
        _logger.QuoteDeleteFailed(id, ex);
    }

    return RedirectToPage();
}
```

When a quote is successfully deleted, inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[4]
    => RequestId:0HL90M6E7PHK5:00000016 RequestPath:/ => /Index
    Quote deleted (Quote = 'You can avoid reality, but you cannot avoid the consequences of avoiding
    reality. - Ayn Rand' Id = 1)
```

When quote deletion fails, inspect the app's console output. Note that the exception is included in the log message:

```
fail: LoggerMessageSample.Pages.IndexModel[5]
    => RequestId:0HL90M6E7PHK5:00000010 RequestPath:/ => /Index
    Quote delete failed (Id = 999)
    System.ArgumentNullException: Value cannot be null.
    Parameter name: entity
    at Microsoft.EntityFrameworkCore.Utilities.Check.NotNull[T](T value, String parameterName)
    at Microsoft.EntityFrameworkCore.DbContext.Remove[TEntity](TEntity entity)
    at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.Remove(TEntity entity)
    at LoggerMessageSample.Pages.IndexModel.<OnPostDeleteQuoteAsync>d__14.MoveNext() in
    <PATH>\sample\Pages\Index.cshtml.cs:line 87
```

Implementing `LoggerMessage.DefineScope`

Define a [log scope](#) to apply to a series of log messages using the `DefineScope(String)` method.

The sample app has a **Clear All** button for deleting all of the quotes in the database. The quotes are deleted by removing them one at a time. Each time a quote is deleted, the `QuoteDeleted` method is called on the logger. A log scope is added to these log messages.

Enable `IncludeScopes` in the console logger options:

```

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging((hostingContext, logging) =>
        {
            // Remove the Debug provider (and other providers) and filter most
            // of the remaining logging. Reducing the logging output makes it
            // easier to see the log messages produced by the LoggerMessage
            // pattern demonstrated in this sample app.
            //
            // Setting options.IncludeScopes is required in ASP.NET Core 2.0
            // apps. Setting IncludeScopes via appsettings configuration files
            // is a feature that's planned for the ASP.NET Core 2.1 release.
            // See: https://github.com/aspnet/Logging/pull/706
            logging.ClearProviders();
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddFilter("Microsoft.EntityFrameworkCore.Update", LogLevel.None);
            logging.AddFilter("Microsoft.EntityFrameworkCore.Infrastructure", LogLevel.None);
            logging.AddFilter("Microsoft.AspNetCore.Mvc.RedirectToRouteResult", LogLevel.None);
            logging.AddFilter("Microsoft.AspNetCore.Mvc.RazorPages.Internal.PageActionInvoker",
LogLevel.None);
            logging.AddFilter("Microsoft.AspNetCore.StaticFiles.StaticFileMiddleware", LogLevel.None);
            logging.AddConsole(options => options.IncludeScopes = true);
        })
        .Build();

```

Setting `IncludeScopes` is required in ASP.NET Core 2.0 apps to enable log scopes. Setting `IncludeScopes` via `appsettings` configuration files is a feature that's planned for the ASP.NET Core 2.1 release.

The sample app clears other providers and adds filters to reduce the logging output. This makes it easier to see the sample's log messages that demonstrate `LoggerMessage` features.

To create a log scope, add a field to hold a `Func` delegate for the scope. The sample app creates a field called `_allQuotesDeletedScope` (*Internal/LoggerExtensions.cs*):

```
private static Func<ILogger, int, IDisposable> _allQuotesDeletedScope;
```

Use `DefineScope` to create the delegate. Up to three types can be specified for use as template arguments when the delegate is invoked. The sample app uses a message template that includes the number of deleted quotes (an `int` type):

```
_allQuotesDeletedScope = LoggerMessage.DefineScope<int>("All quotes deleted (Count = {Count})");
```

Provide a static extension method for the log message. Include any type parameters for named properties that appear in the message template. The sample app takes in a `count` of quotes to delete and returns

`_allQuotesDeletedScope`:

```
public static IDisposable AllQuotesDeletedScope(this ILogger logger, int count)
{
    return _allQuotesDeletedScope(logger, count);
}
```

The scope wraps the logging extension calls in a `using` block:

```

public async Task<IActionResult> OnPostDeleteAllQuotesAsync()
{
    var quoteCount = await _db.Quotes.CountAsync();

    using (_logger.AllQuotesDeletedScope(quoteCount))
    {
        foreach (Quote quote in _db.Quotes)
        {
            _db.Quotes.Remove(quote);

            _logger.QuoteDeleted(quote.Text, quote.Id);
        }
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}

```

Inspect the log messages in the app's console output. The following result shows three quotes deleted with the log scope message included:

```

info: LoggerMessageSample.Pages.IndexModel[4]
    => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
    Quote deleted (Quote = 'Quote 1' Id = 2)
info: LoggerMessageSample.Pages.IndexModel[4]
    => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
    Quote deleted (Quote = 'Quote 2' Id = 3)
info: LoggerMessageSample.Pages.IndexModel[4]
    => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
    Quote deleted (Quote = 'Quote 3' Id = 4)

```

See also

- [Logging](#)

Introduction to Error Handling in ASP.NET Core

10/2/2017 • 4 min to read • [Edit Online](#)

By [Steve Smith](#) and [Tom Dykstra](#)

This article covers common approaches to handling errors in ASP.NET Core apps.

[View or download sample code](#) ([how to download](#))

The developer exception page

To configure an app to display a page that shows detailed information about exceptions, install the

`Microsoft.AspNetCore.Diagnostics` NuGet package and add a line to the [Configure method in the Startup class](#):

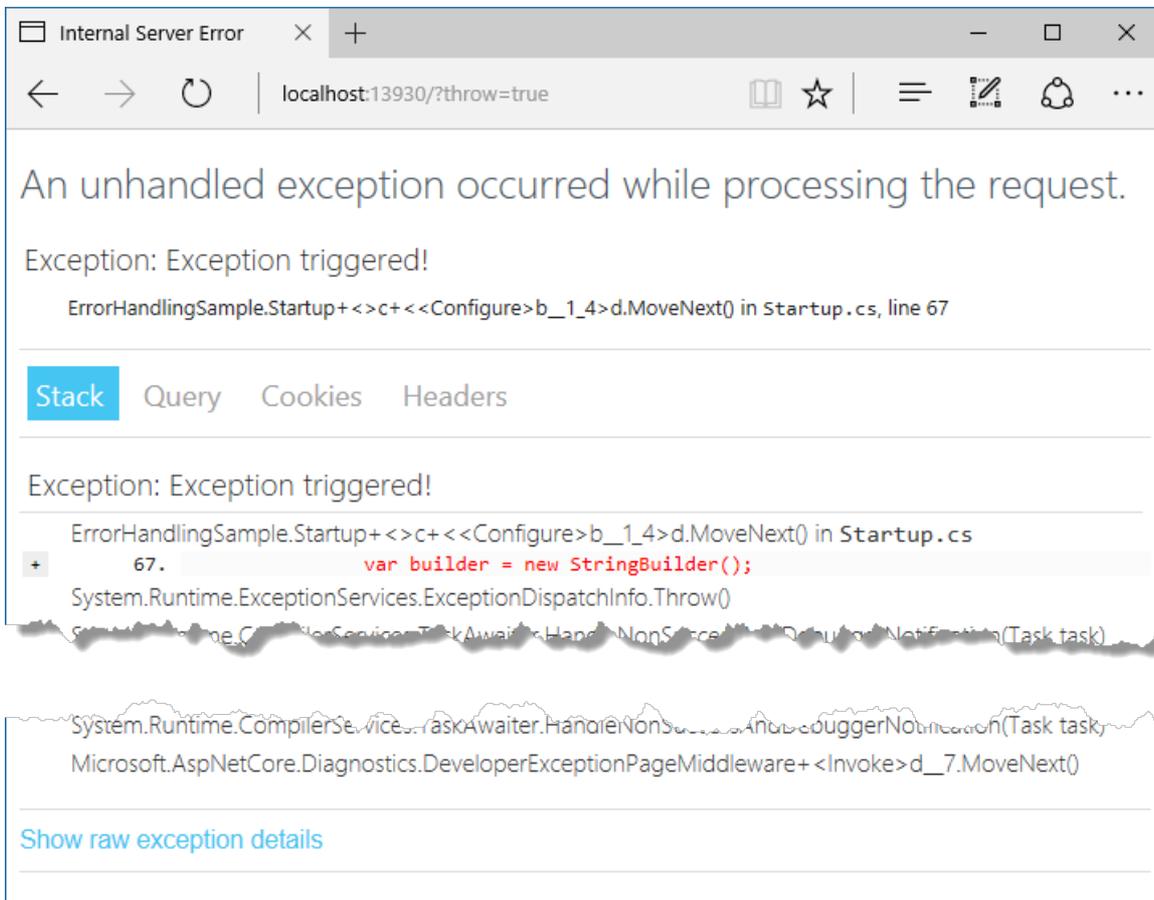
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    env.EnvironmentName = EnvironmentName.Production;
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
}
```

Put `UseDeveloperExceptionPage` before any middleware you want to catch exceptions in, such as `app.UseMvc`.

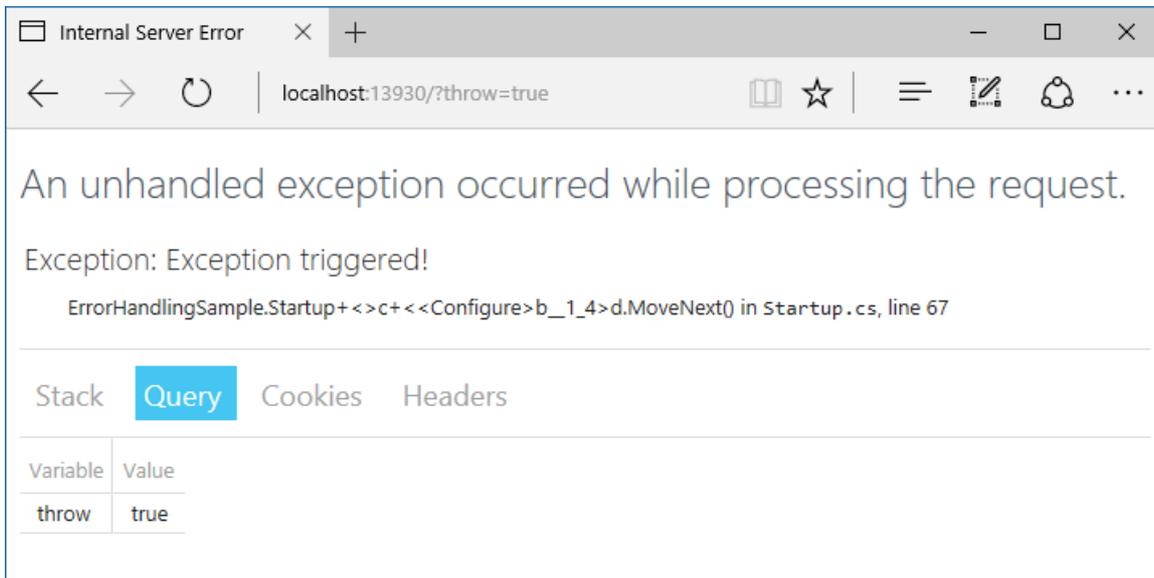
WARNING

Enable the developer exception page **only when the app is running in the Development environment**. You don't want to share detailed exception information publicly when the app runs in production. [Learn more about configuring environments](#).

To see the developer exception page, run the sample application with the environment set to `Development`, and add `?throw=true` to the base URL of the app. The page includes several tabs with information about the exception and the request. The first tab includes a stack trace.



The next tab shows the query string parameters, if any.



This request didn't have any cookies, but if it did, they would appear on the **Cookies** tab. You can see the headers that were passed in the last tab.

Internal Server Error

localhost:13930/?throw=true

An unhandled exception occurred while processing the request.

Exception: Exception triggered!

ErrorHandlingSample.Startup+<>c+<<Configure>b_1_4>d.MoveNext() in Startup.cs, line 67

Variable	Value
Accept	text/html, application/xhtml+xml, image/jxr, */*
Accept-Encoding	gzip, deflate
Accept-Language	en-US
Connection	Keep-Alive
Host	localhost:13930
MS-ASPNETCORE-TOKEN	55ddb2cf-3cf5-4734-afa9-7abcbf38839f
Referer	http://localhost:13930/
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.79 Safari/537.36 Edge/14.14393
X-Original-For	127.0.0.1:20566
X-Original-Proto	http

Configuring a custom exception handling page

It's a good idea to configure an exception handler page to use when the app is not running in the `Development` environment.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    env.EnvironmentName = EnvironmentName.Production;
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
}
```

In an MVC app, don't explicitly decorate the error handler action method with HTTP method attributes, such as `HttpGet`. Using explicit verbs could prevent some requests from reaching the method.

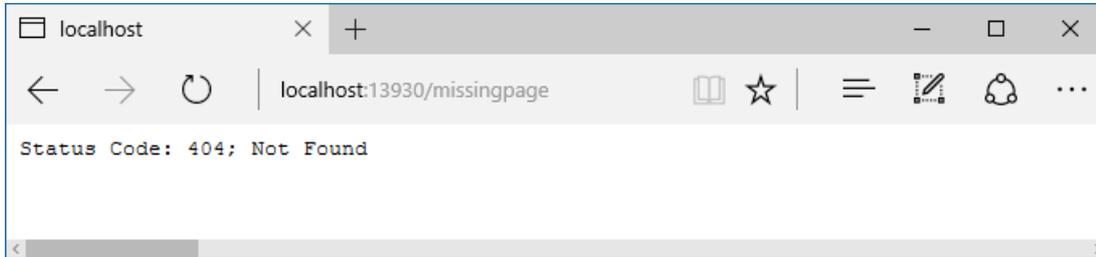
```
[Route("/Error")]
public IActionResult Index()
{
    // Handle error here
}
```

Configuring status code pages

By default, your app will not provide a rich status code page for HTTP status codes such as 500 (Internal Server Error) or 404 (Not Found). You can configure the `StatusCodePagesMiddleware` by adding a line to the `Configure` method:

```
app.UseStatusCodePages();
```

By default, this middleware adds simple, text-only handlers for common status codes, such as 404:



The middleware supports several different extension methods. One takes a lambda expression, another takes a content type and format string.

```
app.UseStatusCodePages(async context =>
{
    context.HttpContext.Response.ContentType = "text/plain";
    await context.HttpContext.Response.WriteAsync(
        "Status code page, status code: " +
        context.HttpContext.Response.StatusCode);
});
```

```
app.UseStatusCodePages("text/plain", "Status code page, status code: {0}");
```

There are also redirect extension methods. One sends a 302 status code to the client, and one returns the original status code to the client but also executes the handler for the redirect URL.

```
app.UseStatusCodePagesWithRedirects("/error/{0}");
```

```
app.UseStatusCodePagesWithReExecute("/error/{0}");
```

If you need to disable status code pages for certain requests, you can do so:

```
var statusCodePagesFeature = context.Features.Get<IStatusCodePagesFeature>();
if (statusCodePagesFeature != null)
{
    statusCodePagesFeature.Enabled = false;
}
```

Exception-handling code

Code in exception handling pages can throw exceptions. It's often a good idea for production error pages to consist of purely static content.

Also, be aware that once the headers for a response have been sent, you can't change the response's status code, nor can any exception pages or handlers run. The response must be completed or the connection aborted.

Server exception handling

In addition to the exception handling logic in your app, the [server](#) hosting your app performs some exception handling. If the server catches an exception before the headers are sent, the server sends a 500 Internal Server Error response with no body. If the server catches an exception after the headers have been sent, the server closes the connection. Requests that aren't handled by your app are handled by the server. Any exception that occurs is handled by the server's exception handling. Any configured custom error pages or exception handling middleware or filters don't affect this behavior.

Startup exception handling

Only the hosting layer can handle exceptions that take place during app startup. You can [configure how the host behaves in response to errors during startup](#) using `captureStartupErrors` and the `detailedErrors` key.

Hosting can only show an error page for a captured startup error if the error occurs after host address/port binding. If any binding fails for any reason, the hosting layer logs a critical exception, the dotnet process crashes, and no error page is displayed.

ASP.NET MVC error handling

[MVC](#) apps have some additional options for handling errors, such as configuring exception filters and performing model validation.

Exception Filters

Exception filters can be configured globally or on a per-controller or per-action basis in an MVC app. These filters handle any unhandled exception that occurs during the execution of a controller action or another filter, and are not called otherwise. Learn more about exception filters in [Filters](#).

TIP

Exception filters are good for trapping exceptions that occur within MVC actions, but they're not as flexible as error handling middleware. Prefer middleware for the general case, and use filters only where you need to do error handling *differently* based on which MVC action was chosen.

Handling Model State Errors

[Model validation](#) occurs prior to each controller action being invoked, and it is the action method's responsibility to inspect `ModelState.IsValid` and react appropriately.

Some apps will choose to follow a standard convention for dealing with model validation errors, in which case a [filter](#) may be an appropriate place to implement such a policy. You should test how your actions behave with invalid model states. Learn more in [Testing controller logic](#).

File Providers in ASP.NET Core

10/2/2017 • 6 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET Core abstracts file system access through the use of File Providers.

[View or download sample code \(how to download\)](#)

File Provider abstractions

File Providers are an abstraction over file systems. The main interface is `IFileProvider`. `IFileProvider` exposes methods to get file information (`IFileInfo`), directory information (`IDirectoryContents`), and to set up change notifications (using an `IChangeToken`).

`IFileInfo` provides methods and properties about individual files or directories. It has two boolean properties, `Exists` and `IsDirectory`, as well as properties describing the file's `Name`, `Length` (in bytes), and `LastModified` date. You can read from the file using its `CreateReadStream` method.

File Provider implementations

Three implementations of `IFileProvider` are available: Physical, Embedded, and Composite. The physical provider is used to access the actual system's files. The embedded provider is used to access files embedded in assemblies. The composite provider is used to provide combined access to files and directories from one or more other providers.

PhysicalFileProvider

The `PhysicalFileProvider` provides access to the physical file system. It wraps the `System.IO.File` type (for the physical provider), scoping all paths to a directory and its children. This scoping limits access to a certain directory and its children, preventing access to the file system outside of this boundary. When instantiating this provider, you must provide it with a directory path, which serves as the base path for all requests made to this provider (and which restricts access outside of this path). In an ASP.NET Core app, you can instantiate a `PhysicalFileProvider` provider directly, or you can request an `IFileProvider` in a Controller or service's constructor through [dependency injection](#). The latter approach will typically yield a more flexible and testable solution.

The sample below shows how to create a `PhysicalFileProvider`.

```
IFileProvider provider = new PhysicalFileProvider(applicationRoot);
IDirectoryContents contents = provider.GetDirectoryContents(""); // the applicationRoot contents
IFileInfo fileInfo = provider.GetFileInfo("wwwroot/js/site.js"); // a file under applicationRoot
```

You can iterate through its directory contents or get a specific file's information by providing a subpath.

To request a provider from a controller, specify it in the controller's constructor and assign it to a local field. Use the local instance from your action methods:

```

public class HomeController : Controller
{
    private readonly IFileProvider _fileProvider;

    public HomeController(IFileProvider fileProvider)
    {
        _fileProvider = fileProvider;
    }

    public IActionResult Index()
    {
        var contents = _fileProvider.GetDirectoryContents("");
        return View(contents);
    }
}

```

Then, create the provider in the app's `Startup` class:

```

using System.Linq;
using System.Reflection;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.FileProviders;
using Microsoft.Extensions.Logging;

namespace FileProviderSample
{
    public class Startup
    {
        private IHostingEnvironment _hostingEnvironment;
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
                .AddEnvironmentVariables();
            Configuration = builder.Build();

            _hostingEnvironment = env;
        }

        public IConfigurationRoot Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            // Add framework services.
            services.AddMvc();

            var physicalProvider = _hostingEnvironment.ContentRootFileProvider;
            var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
            var compositeProvider = new CompositeFileProvider(physicalProvider, embeddedProvider);

            // choose one provider to use for the app and register it
            //services.AddSingleton<IFileProvider>(physicalProvider);
            //services.AddSingleton<IFileProvider>(embeddedProvider);
            services.AddSingleton<IFileProvider>(compositeProvider);
        }
    }
}

```

In the `Index.cshtml` view, iterate through the `IDirectoryContents` provided:

```

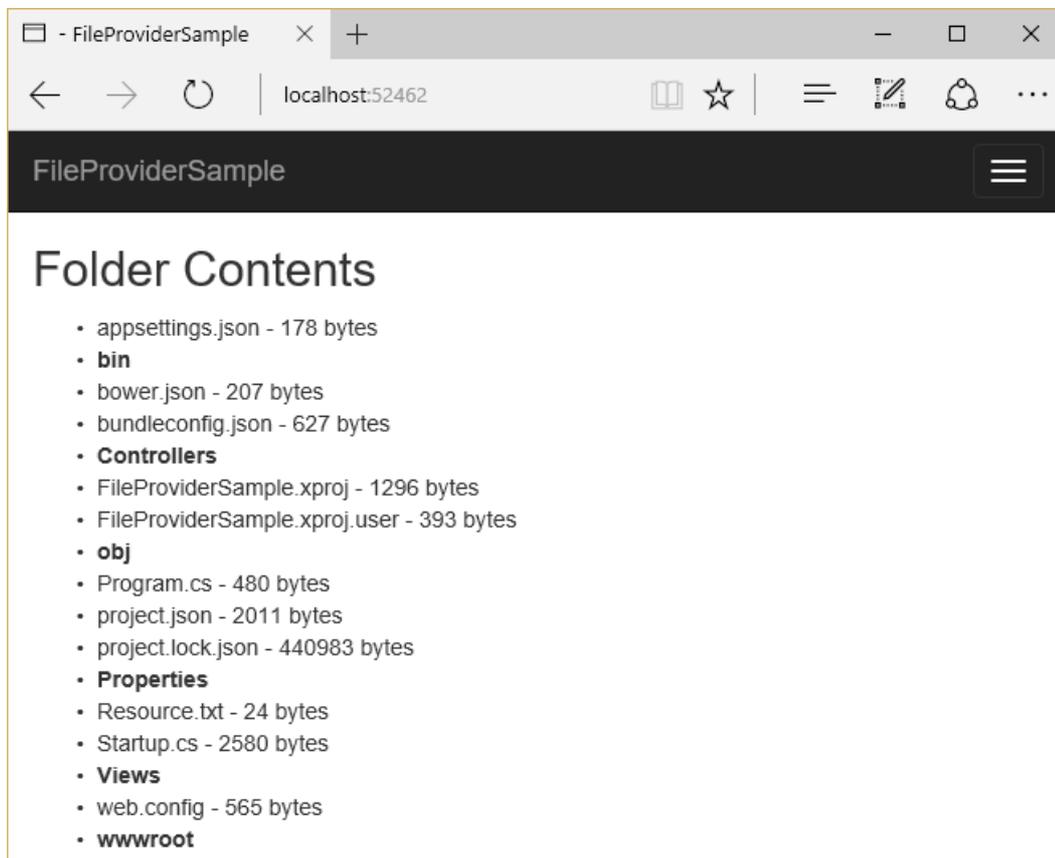
@using Microsoft.Extensions.FileProviders
@model IDirectoryContents

<h2>Folder Contents</h2>

<ul>
  @foreach (FileInfo item in Model)
  {
    if (item.IsDirectory)
    {
      <li><strong>@item.Name</strong></li>
    }
    else
    {
      <li>@item.Name - @item.Length bytes</li>
    }
  }
</ul>

```

The result:



EmbeddedFileProvider

The `EmbeddedFileProvider` is used to access files embedded in assemblies. In .NET Core, you embed files in an assembly with the `<EmbeddedResource>` element in the `.csproj` file:

```

<ItemGroup>
  <EmbeddedResource Include="Resource.txt;*\*.js"
    Exclude="bin\*;obj\*;*\*.xproj;packages\*;@(EmbeddedResource)" />
  <Content Update="wwwroot\*\*;Views\*\*;Areas\*\*\Views;appsettings.json;web.config">
    <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
  </Content>
</ItemGroup>

```

You can use [globbing patterns](#) when specifying files to embed in the assembly. These patterns can be used to

match one or more files.

NOTE

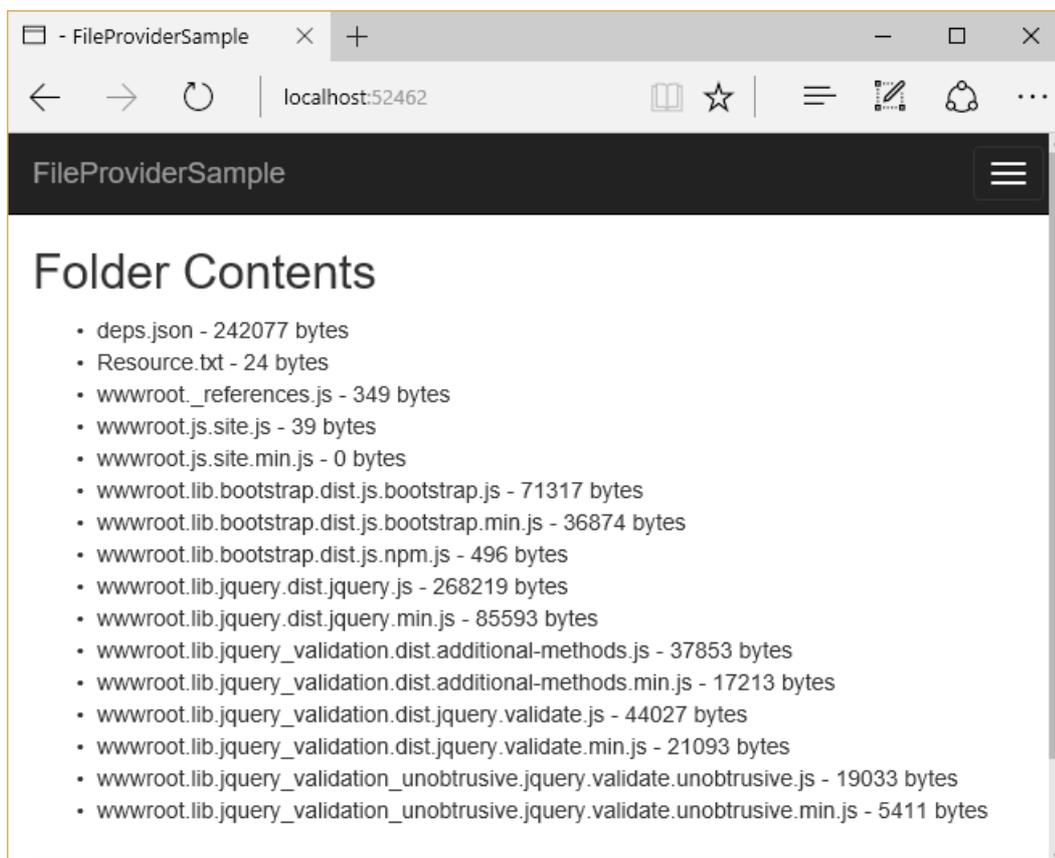
It's unlikely you would ever want to actually embed every .js file in your project in its assembly; the above sample is for demo purposes only.

When creating an `EmbeddedFileProvider`, pass the assembly it will read to its constructor.

```
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
```

The snippet above demonstrates how to create an `EmbeddedFileProvider` with access to the currently executing assembly.

Updating the sample app to use an `EmbeddedFileProvider` results in the following output:



NOTE

Embedded resources do not expose directories. Rather, the path to the resource (via its namespace) is embedded in its filename using `.` separators.

TIP

The `EmbeddedFileProvider` constructor accepts an optional `baseNamespace` parameter. Specifying this will scope calls to `GetDirectoryContents` to those resources under the provided namespace.

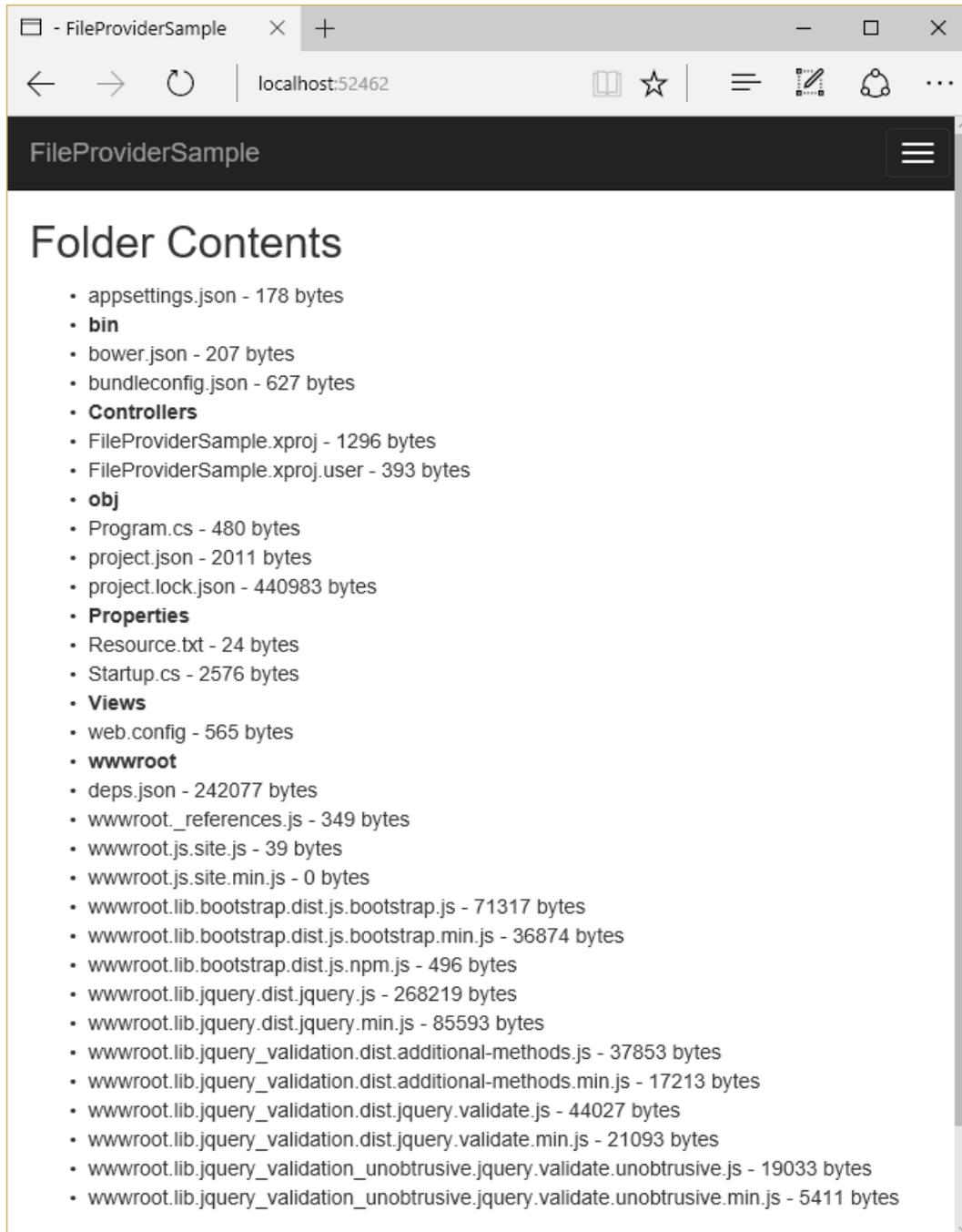
CompositeFileProvider

The `CompositeFileProvider` combines `IFileProvider` instances, exposing a single interface for working with files from multiple providers. When creating the `CompositeFileProvider`, you pass one or more `IFileProvider` instances

to its constructor:

```
var physicalProvider = _hostingEnvironment.ContentRootFileProvider;  
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());  
var compositeProvider = new CompositeFileProvider(physicalProvider, embeddedProvider);
```

Updating the sample app to use a `CompositeFileProvider` that includes both the physical and embedded providers configured previously, results in the following output:



Watching for changes

The `IFileProvider` `Watch` method provides a way to watch one or more files or directories for changes. This method accepts a path string, which can use [globbing patterns](#) to specify multiple files, and returns an `IChangeToken`. This token exposes a `HasChanged` property that can be inspected, and a `RegisterChangeCallback` method that is called when changes are detected to the specified path string. Note that each change token only calls its associated callback in response to a single change. To enable constant monitoring, you can use a `TaskCompletionSource` as shown below, or re-create `IChangeToken` instances in response to changes.

In this article's sample, a console application is configured to display a message whenever a text file is modified:

```
private static PhysicalFileProvider _fileProvider =
    new PhysicalFileProvider(Directory.GetCurrentDirectory());

public static void Main(string[] args)
{
    Console.WriteLine("Monitoring quotes.txt for changes (Ctrl-c to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
    }
}

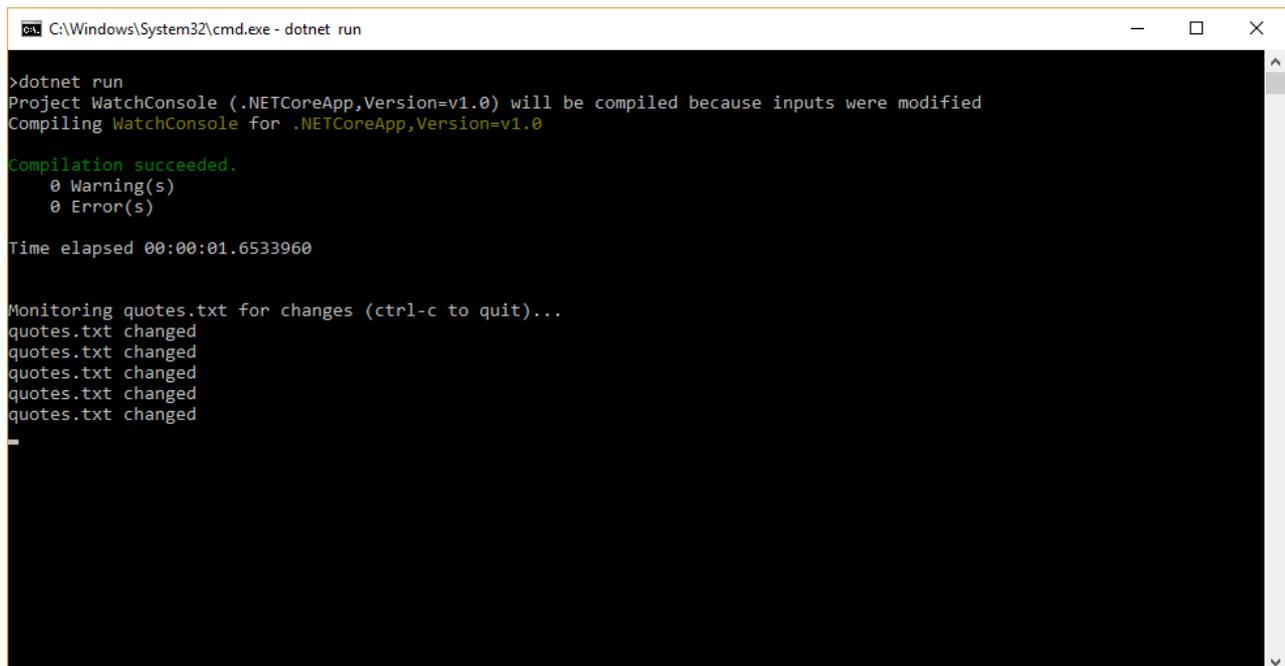
private static async Task MainAsync()
{
    IChangeToken token = _fileProvider.Watch("quotes.txt");
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("quotes.txt changed");
}
```

The result, after saving the file several times:



```
C:\Windows\System32\cmd.exe - dotnet run

>dotnet run
Project WatchConsole (.NETCoreApp,Version=v1.0) will be compiled because inputs were modified
Compiling WatchConsole for .NETCoreApp,Version=v1.0

Compilation succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:01.6533960

Monitoring quotes.txt for changes (ctrl-c to quit)...
quotes.txt changed
quotes.txt changed
quotes.txt changed
quotes.txt changed
quotes.txt changed
```

NOTE

Some file systems, such as Docker containers and network shares, may not reliably send change notifications. Set the `DOTNET_USE_POLLINGFILEWATCHER` environment variable to `1` or `true` to poll the file system for changes every 4 seconds.

Globbering patterns

File system paths use wildcard patterns called *globbing patterns*. These simple patterns can be used to specify groups of files. The two wildcard characters are `*` and `**`.

```
*
```

Matches anything at the current folder level, or any filename, or any file extension. Matches are terminated by `/` and `.` characters in the file path.

```
**
```

Matches anything across multiple directory levels. Can be used to recursively match many files within a directory hierarchy.

Globbing pattern examples

```
directory/file.txt
```

Matches a specific file in a specific directory.

```
directory/*.txt
```

Matches all files with `.txt` extension in a specific directory.

```
directory/*/bower.json
```

Matches all `bower.json` files in directories exactly one level below the `directory` directory.

```
directory/**/*.txt
```

Matches all files with `.txt` extension found anywhere under the `directory` directory.

File Provider usage in ASP.NET Core

Several parts of ASP.NET Core utilize file providers. `IHostingEnvironment` exposes the app's content root and web root as `IFileProvider` types. The static files middleware uses file providers to locate static files. Razor makes heavy use of `IFileProvider` in locating views. Dotnet's publish functionality uses file providers and globbing patterns to specify which files should be published.

Recommendations for use in apps

If your ASP.NET Core app requires file system access, you can request an instance of `IFileProvider` through dependency injection, and then use its methods to perform the access, as shown in this sample. This allows you to configure the provider once, when the app starts up, and reduces the number of implementation types your app instantiates.

Hosting in ASP.NET Core

1/10/2018 • 15 min to read • [Edit Online](#)

By [Luke Latham](#)

ASP.NET Core apps configure and launch a *host*. The host is responsible for app startup and lifetime management. At a minimum, the host configures a server and a request processing pipeline.

Setting up a host

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Create a host using an instance of [WebHostBuilder](#). This is typically performed in the app's entry point, the `Main` method. In the project templates, `Main` is located in *Program.cs*. A typical *Program.cs* calls [CreateDefaultBuilder](#) to start setting up a host:

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

`CreateDefaultBuilder` performs the following tasks:

- Configures [Kestrel](#) as the web server. For the Kestrel default options, see [the Kestrel options section of Kestrel web server implementation in ASP.NET Core](#).
- Sets the content root to the path returned by [Directory.GetCurrentDirectory](#).
- Loads optional configuration from:
 - *appsettings.json*.
 - *appsettings.{Environment}.json*.
 - [User secrets](#) when the app runs in the `Development` environment.
 - Environment variables.
 - Command-line arguments.
- Configures [logging](#) for console and debug output. Logging includes [log filtering](#) rules specified in a Logging configuration section of an *appsettings.json* or *appsettings.{Environment}.json* file.
- When running behind IIS, enables [IIS integration](#). Configures the base path and port the server listens on when using the [ASP.NET Core Module](#). The module creates a reverse proxy between IIS and Kestrel. Also configures the app to [capture startup errors](#). For the IIS default options, see [the IIS options section of Host ASP.NET Core on Windows with IIS](#).

The *content root* determines where the host searches for content files, such as MVC view files. When the app is started from the project's root folder, the project's root folder is used as the content root. This is the default used in [Visual Studio](#) and the [dotnet new templates](#).

For more information on app configuration, see [Configuration in ASP.NET Core](#).

NOTE

As an alternative to using the static `CreateDefaultBuilder` method, creating a host from `WebHostBuilder` is a supported approach with ASP.NET Core 2.x. For more information, see the ASP.NET Core 1.x tab.

When setting up a host, `Configure` and `ConfigureServices` methods can be provided. If a `Startup` class is specified, it must define a `Configure` method. For more information, see [Application Startup in ASP.NET Core](#). Multiple calls to `ConfigureServices` append to one another. Multiple calls to `Configure` or `UseStartup` on the `WebHostBuilder` replace previous settings.

Host configuration values

`WebHostBuilder` relies on the following approaches to set the host configuration values:

- Host builder configuration, which includes environment variables with the format `ASPNETCORE_{configurationKey}`. For example, `ASPNETCORE_URLS`.
- Explicit methods, such as `CaptureStartupErrors`.
- `UseSetting` and the associated key. When setting a value with `UseSetting`, the value is set as a string regardless of the type.

The host uses whichever option sets a value last. For more information, see [Overriding configuration](#) in the next section.

Capture Startup Errors

This setting controls the capture of startup errors.

Key: `captureStartupErrors`

Type: `bool` (`true` or `1`)

Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.

Set using: `CaptureStartupErrors`

Environment variable: `ASPNETCORE_CAPTURESTARTUPERRORS`

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .CaptureStartupErrors(true)
    ...
```

Content Root

This setting determines where ASP.NET Core begins searching for content files, such as MVC views.

Key: `contentRoot`

Type: `string`

Default: Defaults to the folder where the app assembly resides.

Set using: `UseContentRoot`

Environment variable: `ASPNETCORE_CONTENTROOT`

The content root is also used as the base path for the [Web Root setting](#). If the path doesn't exist, the host fails to

start.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseContentRoot("c:\\mywebsite")
    ...
```

Detailed Errors

Determines if detailed errors should be captured.

Key: detailedErrors

Type: *bool* (`true` or `1`)

Default: false

Set using: `UseSetting`

Environment variable: `ASPNETCORE_DETAILEDERRORS`

When enabled (or when the **Environment** is set to `Development`), the app captures detailed exceptions.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.DetailedErrorsKey, "true")
    ...
```

Environment

Sets the app's environment.

Key: environment

Type: *string*

Default: Production

Set using: `UseEnvironment`

Environment variable: `ASPNETCORE_ENVIRONMENT`

The environment can be set to any value. Framework-defined values include `Development` , `Staging` , and `Production` . Values aren't case sensitive. By default, the *Environment* is read from the `ASPNETCORE_ENVIRONMENT` environment variable. When using [Visual Studio](#), environment variables may be set in the *launchSettings.json* file. For more information, see [Working with Multiple Environments](#).

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseEnvironment("Development")
    ...
```

Hosting Startup Assemblies

Sets the app's hosting startup assemblies.

Key: hostingStartupAssemblies

Type: *string*

Default: Empty string

Set using: `UseSetting`

Environment variable: `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`

A semicolon-delimited string of hosting startup assemblies to load on startup. This feature is new in ASP.NET Core 2.0.

Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2")
    ...
```

Prefer Hosting URLs

Indicates whether the host should listen on the URLs configured with the `WebHostBuilder` instead of those configured with the `IServer` implementation.

Key: `preferHostingUrls`

Type: `bool` (`true` or `1`)

Default: `true`

Set using: `PreferHostingUrls`

Environment variable: `ASPNETCORE_PREFERHOSTINGURLS`

This feature is new in ASP.NET Core 2.0.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .PreferHostingUrls(false)
    ...
```

Prevent Hosting Startup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. See [Add app features from an external assembly using IHostingStartup](#) for more information.

Key: `preventHostingStartup`

Type: `bool` (`true` or `1`)

Default: `false`

Set using: `UseSetting`

Environment variable: `ASPNETCORE_PREVENTHOSTINGSTARTUP`

This feature is new in ASP.NET Core 2.0.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.PreventHostingStartupKey, "true")
    ...
```

Server URLs

Indicates the IP addresses or host addresses with ports and protocols that the server should listen on for requests.

Key: `urls`

Type: `string`

Default: `http://localhost:5000`

Set using: `UseUrls`

Environment variable: `ASPNETCORE_URLS`

Set to a semicolon-separated (;) list of URL prefixes to which the server should respond. For example, `http://localhost:123`. Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary between servers.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")
    ...
```

Kestrel has its own endpoint configuration API. For more information, see [Kestrel web server implementation in ASP.NET Core](#).

Shutdown Timeout

Specifies the amount of time to wait for the web host to shutdown.

Key: `shutdownTimeoutSeconds`

Type: `int`

Default: 5

Set using: `UseShutdownTimeout`

Environment variable: `ASPNETCORE_SHUTDOWNTIMEOUTSECONDS`

Although the key accepts an `int` with `UseSetting` (for example, `.UseSetting(WebHostDefaults.ShutdownTimeoutKey, "10")`), the `UseShutdownTimeout` extension method takes a `TimeSpan`. This feature is new in ASP.NET Core 2.0.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...
```

Startup Assembly

Determines the assembly to search for the `Startup` class.

Key: `startupAssembly`

Type: `string`

Default: The app's assembly

Set using: `UseStartup`

Environment variable: `ASPNETCORE_STARTUPASSEMBLY`

The assembly by name (`string`) or type (`TStartup`) can be referenced. If multiple `UseStartup` methods are called, the last one takes precedence.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup("StartupAssemblyName")
    ...
```

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<TStartup>()
    ...
```

Web Root

Sets the relative path to the app's static assets.

Key: `webroot`

Type: `string`

Default: If not specified, the default is `“(Content Root)/wwwroot”`, if the path exists. If the path doesn't exist, then a no-op file provider is used.

Set using: `UseWebRoot`

Environment variable: `ASPNETCORE_WEBROOT`

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .UseWebRoot("public")
    ...
```

Overriding configuration

Use [Configuration](#) to configure the host. In the following example, host configuration is optionally specified in a `hosting.json` file. Any configuration loaded from the `hosting.json` file may be overridden by command-line arguments. The built configuration (in `config`) is used to configure the host with `UseConfiguration`.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

`hosting.json`:

```
{
  urls: "http://*:5005"
}
```

Overriding the configuration provided by `UseUrls` with `hosting.json` config first, command-line argument `config` second:

```

public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("hosting.json", optional: true)
            .AddCommandLine(args)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseUrls("http://*:5000")
            .UseConfiguration(config)
            .Configure(app =>
            {
                app.Run(context =>
                    context.Response.WriteAsync("Hello, World!"));
            })
            .Build();
    }
}

```

NOTE

The `UseConfiguration` extension method isn't currently capable of parsing a configuration section returned by `GetSection` (for example, `.UseConfiguration(Configuration.GetSection("section"))`). The `GetSection` method filters the configuration keys to the section requested but leaves the section name on the keys (for example, `section:urls`, `section:environment`). The `UseConfiguration` method expects the keys to match the `WebHostBuilder` keys (for example, `urls`, `environment`). The presence of the section name on the keys prevents the section's values from configuring the host. This issue will be addressed in an upcoming release. For more information and workarounds, see [Passing configuration section into WebHostBuilder.UseConfiguration uses full keys](#).

To specify the host run on a particular URL, the desired value can be passed in from a command prompt when executing `dotnet run`. The command-line argument overrides the `urls` value from the `hosting.json` file, and the server listens on port 8080:

```
dotnet run --urls "http://*:8080"
```

Starting the host

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Run

The `Run` method starts the web app and blocks the calling thread until the host is shutdown:

```
host.Run();
```

Start

Run the host in a non-blocking manner by calling its `Start` method:

```
using (host)
{
    host.Start();
    Console.ReadLine();
}
```

If a list of URLs is passed to the `Start` method, it listens on the URLs specified:

```
var urls = new List<string>()
{
    "http://*:5000",
    "http://localhost:5001"
};

var host = new WebHostBuilder()
    .UseKestrel()
    .UseStartup<Startup>()
    .Start(urls.ToArray());

using (host)
{
    Console.ReadLine();
}
```

The app can initialize and start a new host using the pre-configured defaults of `CreateDefaultBuilder` using a static convenience method. These methods start the server without console output and with [WaitForShutdown](#) wait for a break (Ctrl-C/SIGINT or SIGTERM):

Start(RequestDelegate app)

Start with a `RequestDelegate` :

```
using (var host = WebHost.Start(app => app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!"

`WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, RequestDelegate app)

Start with a URL and `RequestDelegate` :

```
using (var host = WebHost.Start("http://localhost:8080", app => app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **Start(RequestDelegate app)**, except the app responds on `http://localhost:8080`.

Start(Action routeBuilder)

Use an instance of `IRouteBuilder` ([Microsoft.AspNetCore.Routing](#)) to use routing middleware:

```

using (var host = WebHost.Start(router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]}, {data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}

```

Use the following browser requests with the example:

REQUEST	RESPONSE
<code>http://localhost:5000/hello/Martin</code>	Hello, Martin!
<code>http://localhost:5000/buenosdias/Catrina</code>	Buenos dias, Catrina!
<code>http://localhost:5000/throw/ooops!</code>	Throws an exception with string "ooops!"
<code>http://localhost:5000/throw</code>	Throws an exception with string "Uh oh!"
<code>http://localhost:5000/Sante/Kevin</code>	Sante, Kevin!
<code>http://localhost:5000</code>	Hello World!

`WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, Action routeBuilder)

Use a URL and an instance of `IRouteBuilder`:

```

using (var host = WebHost.Start("http://localhost:8080", router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]}, {data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}

```

Produces the same result as **Start(Action routeBuilder)**, except the app responds at `http://localhost:8080`.

StartWith(Action app)

Provide a delegate to configure an `IApplicationBuilder`:

```

using (var host = WebHost.StartWith(app =>
    app.Use(next =>
        {
            return async context =>
            {
                await context.Response.WriteAsync("Hello World!");
            };
        }
    )))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}

```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!"

`WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

StartWith(string url, Action app)

Provide a URL and a delegate to configure an `IApplicationBuilder` :

```

using (var host = WebHost.StartWith("http://localhost:8080", app =>
    app.Use(next =>
        {
            return async context =>
            {
                await context.Response.WriteAsync("Hello World!");
            };
        }
    )))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}

```

Produces the same result as **StartWith(Action app)**, except the app responds on `http://localhost:8080` .

IHostingEnvironment interface

The [IHostingEnvironment interface](#) provides information about the app's web hosting environment. Use [constructor injection](#) to obtain the `IHostingEnvironment` in order to use its properties and extension methods:

```

public class CustomFileReader
{
    private readonly IHostingEnvironment _env;

    public CustomFileReader(IHostingEnvironment env)
    {
        _env = env;
    }

    public string ReadFile(string filePath)
    {
        var fileProvider = _env.WebRootFileProvider;
        // Process the file here
    }
}

```

A [convention-based approach](#) can be used to configure the app at startup based on the environment.

Alternatively, inject the `IHostingEnvironment` into the `Startup` constructor for use in `ConfigureServices` :

```

public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        HostingEnvironment = env;
    }

    public IHostingEnvironment HostingEnvironment { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }

        var contentRootPath = HostingEnvironment.ContentRootPath;
    }
}

```

NOTE

In addition to the `IsDevelopment` extension method, `IHostingEnvironment` offers `IsStaging`, `IsProduction`, and `IsEnvironment(string environmentName)` methods. See [Working with multiple environments](#) for details.

The `IHostingEnvironment` service can also be injected directly into the `Configure` method for setting up the processing pipeline:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        // In Development, use the developer exception page
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // In Staging/Production, route exceptions to /error
        app.UseExceptionHandler("/error");
    }

    var contentRootPath = env.ContentRootPath;
}

```

`IHostingEnvironment` can be injected into the `Invoke` method when creating custom [middleware](#):

```

public async Task Invoke(HttpContext context, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        // Configure middleware for Development
    }
    else
    {
        // Configure middleware for Staging/Production
    }

    var contentRootPath = env.ContentRootPath;
}

```

IApplicationLifetime interface

[IApplicationLifetime](#) allows for post-startup and shutdown activities. Three properties on the interface are cancellation tokens used to register `Action` methods that define startup and shutdown events. There's also a `StopApplication` method.

CANCELLATION TOKEN	TRIGGERED WHEN...
<code>ApplicationStarted</code>	The host has fully started.
<code>ApplicationStopping</code>	The host is performing a graceful shutdown. Requests may still be processing. Shutdown blocks until this event completes.
<code>ApplicationStopped</code>	The host is completing a graceful shutdown. All requests should be processed. Shutdown blocks until this event completes.
METHOD	ACTION
<code>StopApplication</code>	Requests termination of the current application.

```

public class Startup
{
    public void Configure(IApplicationBuilder app, IApplicationLifetime appLifetime)
    {
        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);

        Console.CancelKeyPress += (sender, eventArgs) =>
        {
            appLifetime.StopApplication();
            // Don't terminate the process immediately, wait for the Main thread to exit gracefully.
            eventArgs.Cancel = true;
        };
    }

    private void OnStarted()
    {
        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        // Perform post-stopped activities here
    }
}

```

Troubleshooting System.ArgumentException

Applies to ASP.NET Core 2.0 Only

A host may be built by injecting `IStartup` directly into the dependency injection container rather than calling `UseStartup` OR `Configure` :

```
services.AddSingleton<IStartup, Startup>();
```

If the host is built this way, the following error may occur:

```
Unhandled Exception: System.ArgumentException: A valid non-empty application name must be provided.
```

This occurs because the `applicationName(ApplicationKey)` (the current assembly) is required to scan for `HostingStartupAttributes` . If the app manually injects `IStartup` into the dependency injection container, add the following call to `WebHostBuilder` with the assembly name specified:

```

WebHost.CreateDefaultBuilder(args)
    .UseSetting("applicationName", "<Assembly Name>")
    ...

```

Alternatively, add a dummy `Configure` to the `WebHostBuilder` , which sets the `applicationName (ApplicationKey)` automatically:

```
WebHost.CreateDefaultBuilder(args)
    .Configure(_ => { })
    ...
```

NOTE: This is only required with the ASP.NET Core 2.0 release and only when the app doesn't call `UseStartup` or `Configure`.

For more information, see [Announcements: Microsoft.Extensions.PlatformAbstractions has been removed \(comment\)](#) and the [StartupInjection sample](#).

Additional resources

- [Host on Windows with IIS](#)
- [Host on Linux with Nginx](#)
- [Host on Linux with Apache](#)
- [Host in a Windows Service](#)

Introduction to session and application state in ASP.NET Core

11/29/2017 • 13 min to read • [Edit Online](#)

By [Rick Anderson](#), [Steve Smith](#), and [Diana LaRose](#)

HTTP is a stateless protocol. A web server treats each HTTP request as an independent request and does not retain user values from previous requests. This article discusses different ways to preserve application and session state between requests.

Session state

Session state is a feature in ASP.NET Core that you can use to save and store user data while the user browses your web app. Consisting of a dictionary or hash table on the server, session state persists data across requests from a browser. The session data is backed by a cache.

ASP.NET Core maintains session state by giving the client a cookie that contains the session ID, which is sent to the server with each request. The server uses the session ID to fetch the session data. Because the session cookie is specific to the browser, you cannot share sessions across browsers. Session cookies are deleted only when the browser session ends. If a cookie is received for an expired session, a new session that uses the same session cookie is created.

The server retains a session for a limited time after the last request. You can either set the session timeout or use the default value of 20 minutes. Session state is ideal for storing user data that is specific to a particular session but doesn't need to be persisted permanently. Data is deleted from the backing store either when you call `Session.Clear` or when the session expires in the data store. The server does not know when the browser is closed or when the session cookie is deleted.

WARNING

Do not store sensitive data in session. The client might not close the browser and clear the session cookie (and some browsers keep session cookies alive across windows). Also, a session might not be restricted to a single user; the next user might continue with the same session.

The in-memory session provider stores session data on the local server. If you plan to run your web app on a server farm, you must use sticky sessions to tie each session to a specific server. The Windows Azure Web Sites platform defaults to sticky sessions (Application Request Routing or ARR). However, sticky sessions can affect scalability and complicate web app updates. A better option is to use the Redis or SQL Server distributed caches, which don't require sticky sessions. For more information, see [Working with a Distributed Cache](#). For details on setting up service providers, see [Configuring Session](#) later in this article.

TempData

ASP.NET Core MVC exposes the `TempData` property on a [controller](#). This property stores data until it is read. The `Keep` and `Peek` methods can be used to examine the data without deletion. `TempData` is particularly useful for redirection, when data is needed for more than a single request. `TempData` is implemented by `TempData` providers, for example, using either cookies or session state.

TempData providers

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

In ASP.NET Core 2.0 and later, the cookie-based TempData provider is used by default to store TempData in cookies.

The cookie data is encoded with the [Base64UrlTextEncoder](#). Because the cookie is encrypted and chunked, the single cookie size limit found in ASP.NET Core 1.x does not apply. The cookie data is not compressed because compressing encrypted data can lead to security problems such as the [CRIME](#) and [BREACH](#) attacks. For more information on the cookie-based TempData provider, see [CookieTempDataProvider](#).

Choosing a TempData provider

Choosing a TempData provider involves several considerations, such as:

1. Does the application already use session state for other purposes? If so, using the session state TempData provider has no additional cost to the application (aside from the size of the data).
2. Does the application use TempData only sparingly, for relatively small amounts of data (up to 500 bytes)? If so, the cookie TempData provider will add a small cost to each request that carries TempData. If not, the session state TempData provider can be beneficial to avoid round-tripping a large amount of data in each request until the TempData is consumed.
3. Does the application run in a web farm (multiple servers)? If so, there is no additional configuration needed to use the cookie TempData provider.

NOTE

Most web clients (such as web browsers) enforce limits on the maximum size of each cookie, the total number of cookies, or both. Therefore, when using the cookie TempData provider, verify the app won't exceed these limits. Consider the total size of the data, accounting for the overheads of encryption and chunking.

Configure the TempData provider

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The cookie-based TempData provider is enabled by default. The following `Startup` class code configures the session-based TempData provider:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddSessionStateTempDataProvider();

    services.AddSession();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseSession();
    app.UseMvcWithDefaultRoute();
}
```

Ordering is critical for middleware components. In the preceding example, an exception of type

`InvalidOperationException` occurs when `UseSession` is invoked after `UseMvcWithDefaultRoute`. See [Middleware Ordering](#) for more detail.

IMPORTANT

If targeting .NET Framework and using the session-based provider, add the [Microsoft.AspNetCore.Session](#) NuGet package to your project.

Query strings

You can pass a limited amount of data from one request to another by adding it to the new request's query string. This is useful for capturing state in a persistent manner that allows links with embedded state to be shared through email or social networks. However, for this reason, you should never use query strings for sensitive data. In addition to being easily shared, including data in query strings can create opportunities for [Cross-Site Request Forgery \(CSRF\)](#) attacks, which can trick users into visiting malicious sites while authenticated. Attackers can then steal user data from your app or take malicious actions on behalf of the user. Any preserved application or session state must protect against CSRF attacks. For more information on CSRF attacks, see [Preventing Cross-Site Request Forgery \(XSRF/CSRF\) Attacks in ASP.NET Core](#).

Post data and hidden fields

Data can be saved in hidden form fields and posted back on the next request. This is common in multi-page forms. However, because the client can potentially tamper with the data, the server must always revalidate it.

Cookies

Cookies provide a way to store user-specific data in web applications. Because cookies are sent with every request, their size should be kept to a minimum. Ideally, only an identifier should be stored in a cookie with the actual data stored on the server. Most browsers restrict cookies to 4096 bytes. In addition, only a limited number of cookies are available for each domain.

Because cookies are subject to tampering, they must be validated on the server. Although the durability of the cookie on a client is subject to user intervention and expiration, they are generally the most durable form of data persistence on the client.

Cookies are often used for personalization, where content is customized for a known user. Because the user is only identified and not authenticated in most cases, you can typically secure a cookie by storing the user name, account name, or a unique user ID (such as a GUID) in the cookie. You can then use the cookie to access the user personalization infrastructure of a site.

HttpContext.Items

The `Items` collection is a good location to store data that is needed only while processing one particular request. The collection's contents are discarded after each request. The `Items` collection is best used as a way for components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters. For more information, see [Working with HttpContext.Items](#), later in this article.

Cache

Caching is an efficient way to store and retrieve data. You can control the lifetime of cached items based on time and other considerations. Learn more about [Caching](#).

Working with Session State

Configuring Session

The `Microsoft.AspNetCore.Session` package provides middleware for managing session state. To enable the session middleware, `Startup` must contain:

- Any of the `IDistributedCache` memory caches. The `IDistributedCache` implementation is used as a backing store for session.
- `AddSession` call, which requires NuGet package "Microsoft.AspNetCore.Session".
- `UseSession` call.

The following code shows how to set up the in-memory session provider.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using System;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        // Adds a default in-memory implementation of IDistributedCache.
        services.AddDistributedMemoryCache();

        services.AddSession(options =>
        {
            // Set a short timeout for easy testing.
            options.IdleTimeout = TimeSpan.FromSeconds(10);
            options.Cookie.HttpOnly = true;
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseSession();
        app.UseMvcWithDefaultRoute();
    }
}
```

You can reference `Session` from `HttpContext` once it is installed and configured.

If you try to access `Session` before `UseSession` has been called, the exception

```
InvalidOperationException: Session has not been configured for this application or request
```

 is thrown.

If you try to create a new `Session` (that is, no session cookie has been created) after you have already begun writing to the `Response` stream, the exception

```
InvalidOperationException: The session cannot be established after the response has started
```

 is thrown. The exception can be found in the web server log; it will not be displayed in the browser.

Loading Session asynchronously

The default session provider in ASP.NET Core loads the session record from the underlying `IDistributedCache` store asynchronously only if the `ISession.LoadAsync` method is explicitly called before the `TryGetValue`, `Set`, or `Remove` methods. If `LoadAsync` is not called first, the underlying session record is loaded synchronously, which could potentially impact the ability of the app to scale.

To have applications enforce this pattern, wrap the `DistributedSessionStore` and `DistributedSession` implementations with versions that throw an exception if the `LoadAsync` method is not called before `TryGetValue`, `Set`, or `Remove`. Register the wrapped versions in the services container.

Implementation Details

Session uses a cookie to track and identify requests from a single browser. By default, this cookie is named ".AspNet.Session", and it uses a path of "/". Because the cookie default does not specify a domain, it is not made available to the client-side script on the page (because `CookieHttpOnly` defaults to `true`).

To override session defaults, use `SessionOptions`:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    // Adds a default in-memory implementation of IDistributedCache.
    services.AddDistributedMemoryCache();

    services.AddSession(options =>
    {
        options.Cookie.Name = ".AdventureWorks.Session";
        options.IdleTimeout = TimeSpan.FromSeconds(10);
    });
}
```

The server uses the `IdleTimeout` property to determine how long a session can be idle before its contents are abandoned. This property is independent of the cookie expiration. Each request that passes through the Session middleware (read from or written to) resets the timeout.

Because `Session` is *non-locking*, if two requests both attempt to modify the contents of session, the last one overrides the first. `Session` is implemented as a *coherent session*, which means that all the contents are stored together. Two requests that are modifying different parts of the session (different keys) might still impact each other.

Setting and getting Session values

Session is accessed through the `Session` property on `HttpContext`. This property is an [ISession](#) implementation.

The following example shows setting and getting an int and a string:

```
public class HomeController : Controller
{
    const string SessionKeyName = "_Name";
    const string SessionKeyYearsMember = "_YearsMember";
    const string SessionKeyDate = "_Date";

    public IActionResult Index()
    {
        // Requires using Microsoft.AspNetCore.Http;
        HttpContext.Session.SetString(SessionKeyName, "Rick");
        HttpContext.Session.SetInt32(SessionKeyYearsMember, 3);
        return RedirectToAction("SessionNameYears");
    }

    public IActionResult SessionNameYears()
    {
        var name = HttpContext.Session.GetString(SessionKeyName);
        var yearsMember = HttpContext.Session.GetInt32(SessionKeyYearsMember);

        return Content($"Name: \"{name}\", Membership years: \"{yearsMember}\"");
    }
}
```

If you add the following extension methods, you can set and get serializable objects to Session:

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return value == null ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }
}
```

The following sample shows how to set and get a serializable object:

```
public IActionResult SetDate()
{
    // Requires you add the Set extension method mentioned in the article.
    HttpContext.Session.Set<DateTime>(SessionKeyDate, DateTime.Now);
    return RedirectToAction("GetDate");
}

public IActionResult GetDate()
{
    // Requires you add the Get extension method mentioned in the article.
    var date = HttpContext.Session.Get<DateTime>(SessionKeyDate);
    var sessionTime = date.TimeOfDay.ToString();
    var currentTime = DateTime.Now.TimeOfDay.ToString();

    return Content($"Current time: {currentTime} - "
        + $"session time: {sessionTime}");
}
```

Working with HttpContext.Items

The `HttpContext` abstraction provides support for a dictionary collection of type `IDictionary<object, object>`, called `Items`. This collection is available from the start of an *HttpRequest* and is discarded at the end of each request. You can access it by assigning a value to a keyed entry, or by requesting the value for a particular key.

In the sample below, [Middleware](#) adds `isVerified` to the `Items` collection.

```
app.Use(async (context, next) =>
{
    // perform some verification
    context.Items["isVerified"] = true;
    await next.Invoke();
});
```

Later in the pipeline, another middleware could access it:

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Verified request? " +
        context.Items["isVerified"]);
});
```

For middleware that will only be used by a single app, `string` keys are acceptable. However, middleware that will be shared between applications should use unique object keys to avoid any chance of key collisions. If you are developing middleware that must work across multiple applications, use a unique object key defined in your middleware class as shown below:

```
public class SampleMiddleware
{
    public static readonly object SampleKey = new Object();

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[SampleKey] = "some value";
        // additional code omitted
    }
}
```

Other code can access the value stored in `HttpContext.Items` using the key exposed by the middleware class:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        string value = HttpContext.Items[SampleMiddleware.SampleKey];
    }
}
```

This approach also has the advantage of eliminating repetition of "magic strings" in multiple places in the code.

Application state data

Use [Dependency Injection](#) to make data available to all users:

1. Define a service containing the data (for example, a class named `MyAppData`).

```
public class MyAppData
{
    // Declare properties/methods/etc.
}
```

1. Add the service class to `ConfigureServices` (for example `services.AddSingleton<MyAppData>();`).
2. Consume the data service class in each controller:

```
public class MyController : Controller
{
    public MyController(MyAppData myService)
    {
        // Do something with the service (read some data from it,
        // store it in a private field/property, etc.)
    }
}
```

Common errors when working with session

- "Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

This is usually caused by failing to configure at least one `IDistributedCache` implementation. For more information, see [Working with a Distributed Cache](#) and [In memory caching](#).

- In the event that the session middleware fails to persist a session (for example: if the database is not available), it logs the exception and swallows it. The request will then continue normally, which leads to very unpredictable behavior.

A typical example:

Someone stores a shopping basket in session. The user adds an item but the commit fails. The app doesn't know about the failure so it reports the message "The item has been added", which isn't true.

The recommended way to check for such errors is to call `await feature.Session.CommitAsync();` from app code when you're done writing to the session. Then you can do what you like with the error. It works the same way when calling `LoadAsync`.

Additional Resources

- [ASP.NET Core 1.x: Sample code used in this document](#)
- [ASP.NET Core 2.x: Sample code used in this document](#)

Web server implementations in ASP.NET Core

1/10/2018 • 4 min to read • [Edit Online](#)

By [Tom Dykstra](#), [Steve Smith](#), [Stephen Halter](#), and [Chris Ross](#)

An ASP.NET Core application runs with an in-process HTTP server implementation. The server implementation listens for HTTP requests and surfaces them to the application as sets of [request features](#) composed into an `HttpContext`.

ASP.NET Core ships two server implementations:

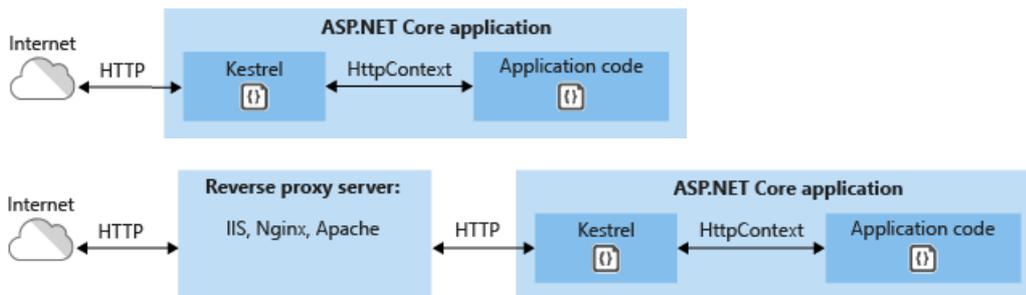
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)
- [Kestrel](#) is a cross-platform HTTP server based on [libuv](#), a cross-platform asynchronous I/O library.
- [HTTP.sys](#) is a Windows-only HTTP server based on the [Http.Sys kernel driver](#).

Kestrel

Kestrel is the web server that is included by default in ASP.NET Core new-project templates.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

You can use Kestrel by itself or with a *reverse proxy server*, such as IIS, Nginx, or Apache. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling.



Either configuration — with or without a reverse proxy server — can also be used if Kestrel is exposed only to an internal network.

For information about when to use Kestrel with a reverse proxy, see [Introduction to Kestrel](#).

You can't use IIS, Nginx, or Apache without Kestrel or a [custom server implementation](#). ASP.NET Core was designed to run in its own process so that it can behave consistently across platforms. IIS, Nginx, and Apache dictate their own startup process and environment; to use them directly, ASP.NET Core would have to adapt to the needs of each one. Using a web server implementation such as Kestrel gives ASP.NET Core control over the startup process and environment. So rather than trying to adapt ASP.NET Core to IIS, Nginx, or Apache, you just set up those web servers to proxy requests to Kestrel. This arrangement allows your `Program.Main` and `Startup` classes to be essentially the same no matter where you deploy.

IIS with Kestrel

When you use IIS or IIS Express as a reverse proxy for ASP.NET Core, the ASP.NET Core application runs in a process separate from the IIS worker process. In the IIS process, a special IIS module runs to coordinate the reverse proxy relationship. This is the *ASP.NET Core Module*. The primary functions of the ASP.NET Core Module

are to start the ASP.NET Core application, restart it when it crashes, and forward HTTP traffic to it. For more information, see [ASP.NET Core Module](#).

Ngix with Kestrel

For information about how to use Nginx on Linux as a reverse proxy server for Kestrel, see [Host on Linux with Nginx](#).

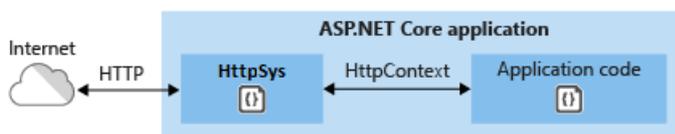
Apache with Kestrel

For information about how to use Apache on Linux as a reverse proxy server for Kestrel, see [Host on Linux with Apache](#).

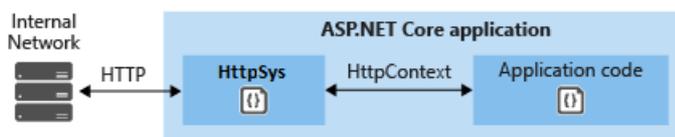
HTTP.sys

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

If you run your ASP.NET Core app on Windows, HTTP.sys is an alternative to Kestrel. You can use HTTP.sys for scenarios where you expose your app to the Internet and you need HTTP.sys features that Kestrel doesn't support.



HTTP.sys can also be used for applications that are exposed only to an internal network.



For internal network scenarios, Kestrel is generally recommended for best performance; but in some scenarios, you might want to use a feature that only HTTP.sys offers. For information about HTTP.sys features, see [HTTP.sys](#).

Notes about ASP.NET Core server infrastructure

The `IApplicationBuilder` available in the `Startup` class `Configure` method exposes the `ServerFeatures` property of type `IFeatureCollection`. Kestrel and WebListener both expose only a single feature, `IServerAddressesFeature`, but different server implementations may expose additional functionality.

`IServerAddressesFeature` can be used to find out which port the server implementation has bound to at runtime.

Custom servers

If the built-in servers don't meet your needs, you can create a custom server implementation. The [Open Web Interface for .NET \(OWIN\) guide](#) demonstrates how to write a `Nowin`-based `IServer` implementation. You're free to implement just the feature interfaces your application needs, though at a minimum you must support `IHttpRequestFeature` and `IHttpResponseFeature`.

Next steps

For more information, see the following resources:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)
- [Kestrel](#)

- [Kestrel with IIS](#)
- [Host on Linux with Nginx](#)
- [Host on Linux with Apache](#)
- [HTTP.sys](#)

Introduction to Kestrel web server implementation in ASP.NET Core

10/2/2017 • 11 min to read • [Edit Online](#)

By [Tom Dykstra](#), [Chris Ross](#), and [Stephen Halter](#)

Kestrel is a cross-platform [web server for ASP.NET Core](#) based on [libuv](#), a cross-platform asynchronous I/O library. Kestrel is the web server that is included by default in ASP.NET Core project templates.

Kestrel supports the following features:

- HTTPS
- Opaque upgrade used to enable [WebSockets](#)
- Unix sockets for high performance behind Nginx

Kestrel is supported on all platforms and versions that .NET Core supports.

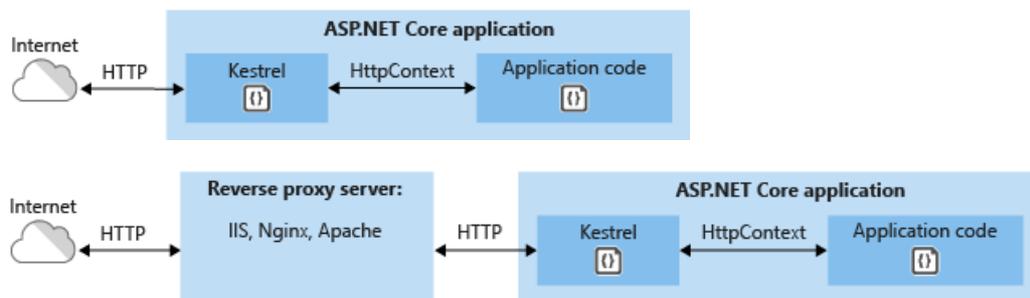
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

[View or download sample code for 2.x \(how to download\)](#)

When to use Kestrel with a reverse proxy

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

You can use Kestrel by itself or with a *reverse proxy server*, such as IIS, Nginx, or Apache. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling.



Either configuration — with or without a reverse proxy server — can also be used if Kestrel is exposed only to an internal network.

A scenario that requires a reverse proxy is when you have multiple applications that share the same IP and port running on a single server. That doesn't work with Kestrel directly because Kestrel doesn't support sharing the same IP and port between multiple processes. When you configure Kestrel to listen on a port, it handles all traffic for that port regardless of host header. A reverse proxy that can share ports must then forward to Kestrel on a unique IP and port.

Even if a reverse proxy server isn't required, using one might be a good choice for other reasons:

- It can limit your exposed surface area.
- It provides an optional additional layer of configuration and defense.
- It might integrate better with existing infrastructure.

- It simplifies load balancing and SSL set-up. Only your reverse proxy server requires an SSL certificate, and that server can communicate with your application servers on the internal network using plain HTTP.

How to use Kestrel in ASP.NET Core apps

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The [Microsoft.AspNetCore.Server.Kestrel](#) package is included in the [Microsoft.AspNetCore.All](#) metapackage.

ASP.NET Core project templates use Kestrel by default. In *Program.cs*, the template code calls

`CreateDefaultBuilder`, which calls [UseKestrel](#) behind the scenes.

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
            {
                options.Listen(IPAddress.Loopback, 5000);
                options.Listen(IPAddress.Loopback, 5001, listenOptions =>
                    {
                        listenOptions.UseHttps("testCert.pfx", "testPassword");
                    });
            })
        .Build();
```

If you need to configure Kestrel options, call `UseKestrel` in *Program.cs* as shown in the following example:

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
            {
                options.Listen(IPAddress.Loopback, 5000);
                options.Listen(IPAddress.Loopback, 5001, listenOptions =>
                    {
                        listenOptions.UseHttps("testCert.pfx", "testPassword");
                    });
            })
        .Build();
```

Kestrel options

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The Kestrel web server has constraint configuration options that are especially useful in Internet-facing deployments. Here are some of the limits you can set:

- Maximum client connections
- Maximum request body size

- Minimum request body data rate

You set these constraints and others in the `Limits` property of the `KestrelServerOptions` class. The `Limits` property holds an instance of the `KestrelServerLimits` class.

Maximum client connections

The maximum number of concurrent open TCP connections can be set for the entire application with the following code:

```
.UseKestrel(options =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
})
```

There's a separate limit for connections that have been upgraded from HTTP or HTTPS to another protocol (for example, on a WebSockets request). After a connection is upgraded, it isn't counted against the

`MaxConcurrentConnections` limit.

The maximum number of connections is unlimited (null) by default.

Maximum request body size

The default maximum request body size is 30,000,000 bytes, which is approximately 28.6MB.

The recommended way to override the limit in an ASP.NET Core MVC app is to use the `RequestSizeLimit` attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

Here's an example that shows how to configure the constraint for the entire application, every request:

```
.UseKestrel(options =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
})
```

You can override the setting on a specific request in middleware:

```

app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;
    context.Features.Get<IHttpMinRequestBodyDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    context.Features.Get<IHttpMinResponseDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
}

```

An exception is thrown if you try to configure the limit on a request after the application has started reading the request. There's an `IsReadOnly` property that tells you if the `MaxRequestBodySize` property is in read-only state, meaning it's too late to configure the limit.

Minimum request body data rate

Kestrel checks every second if data is coming in at the specified rate in bytes/second. If the rate drops below the minimum, the connection is timed out. The grace period is the amount of time that Kestrel gives the client to increase its send rate up to the minimum; the rate is not checked during that time. The grace period helps avoid dropping connections that are initially sending data at a slow rate due to TCP slow-start.

The default minimum rate is 240 bytes/second, with a 5 second grace period.

A minimum rate also applies to the response. The code to set the request limit and the response limit is the same except for having `RequestBody` or `Response` in the property and interface names.

Here's an example that shows how to configure the minimum data rates in *Program.cs*:

```

.UseKestrel(options =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
}

```

You can configure the rates per request in middleware:

```

app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;
    context.Features.Get<IHttpMinRequestBodyDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    context.Features.Get<IHttpMinResponseDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
}

```

For information about other Kestrel options, see the following classes:

- [KestrelServerOptions](#)
- [KestrelServerLimits](#)
- [ListenOptions](#)

Endpoint configuration

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

By default ASP.NET Core binds to `http://localhost:5000`. You configure URL prefixes and ports for Kestrel to listen on by calling `Listen` or `ListenUnixSocket` methods on `KestrelServerOptions`. (`UseUrls`, the `urls` command-line argument, and the `ASPNETCORE_URLS` environment variable also work but have the limitations noted [later in this article](#).)

Bind to a TCP socket

The `Listen` method binds to a TCP socket, and an options lambda lets you configure an SSL certificate:

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
            {
                options.Listen(IPAddress.Loopback, 5000);
                options.Listen(IPAddress.Loopback, 5001, listenOptions =>
                    {
                        listenOptions.UseHttps("testCert.pfx", "testPassword");
                    });
            })
        .Build();
```

Notice how this example configures SSL for a particular endpoint by using `ListenOptions`. You can use the same API to configure other Kestrel settings for particular endpoints.

For generating self-signed SSL certificates on Windows, you can use the PowerShell cmdlet [New-SelfSignedCertificate](#). For a third-party tool that makes it easier for you to generate self-signed certificates, see [SelfCert](#).

On macOS and Linux you can create a self-signed certificate using [OpenSSL](#).

Bind to a Unix socket

You can listen on a Unix socket for improved performance with Nginx, as shown in this example:

```
.UseKestrel(options =>
{
    options.ListenUnixSocket("/tmp/kestrel-test.sock");
    options.ListenUnixSocket("/tmp/kestrel-test.sock", listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx", "testpassword");
        });
})
```

Port 0

If you specify port number 0, Kestrel dynamically binds to an available port. The following example shows how to determine which port Kestrel actually bound to at runtime:

```

public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    var serverAddressesFeature = app.ServerFeatures.Get<IServerAddressesFeature>();

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/html";
        await context.Response
            .WriteAsync("<p>Hosted by Kestrel</p>");

        if (serverAddressesFeature != null)
        {
            await context.Response
                .WriteAsync("<p>Listening on the following addresses: " +
                    string.Join(", ", serverAddressesFeature.Addresses) +
                    "</p>");
        }

        await context.Response.WriteAsync($"<p>Request URL: {context.Request.GetDisplayUrl()}</p>");
    });
}

```

UseUrls limitations

You can configure endpoints by calling the `UseUrls` method or using the `urls` command-line argument or the `ASPNETCORE_URLS` environment variable. These methods are useful if you want your code to work with servers other than Kestrel. However, be aware of these limitations:

- You can't use SSL with these methods.
- If you use both the `Listen` method and `UseUrls`, the `Listen` endpoints override the `UseUrls` endpoints.

Endpoint configuration for IIS

If you use IIS, the URL bindings for IIS override any bindings that you set by calling either `Listen` or `UseUrls`. For more information, see [Introduction to ASP.NET Core Module](#).

URL prefixes

If you call `UseUrls` or use the `urls` command-line argument or `ASPNETCORE_URLS` environment variable, the URL prefixes can be in any of the following formats.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Only HTTP URL prefixes are valid; Kestrel does not support SSL when you configure URL bindings by using `UseUrls`.

- IPv4 address with port number

```
http://65.55.39.10:80/
```

0.0.0.0 is a special case that binds to all IPv4 addresses.

- IPv6 address with port number

```
http://[0:0:0:0:ffff:4137:270a]:80/
```

`:::` is the IPv6 equivalent of IPv4 0.0.0.0.

- Host name with port number

```
http://contoso.com:80/  
http://*:80/
```

Host names, *, and +, are not special. Anything that is not a recognized IP address or "localhost" will bind to all IPv4 and IPv6 IPs. If you need to bind different host names to different ASP.NET Core applications on the same port, use [HTTP.sys](#) or a reverse proxy server such as IIS, Nginx, or Apache.

- "Localhost" name with port number or loopback IP with port number

```
http://localhost:5000/  
http://127.0.0.1:5000/  
http://[::1]:5000/
```

When `localhost` is specified, Kestrel tries to bind to both IPv4 and IPv6 loopback interfaces. If the requested port is in use by another service on either loopback interface, Kestrel fails to start. If either loopback interface is unavailable for any other reason (most commonly because IPv6 is not supported), Kestrel logs a warning.

Next steps

For more information, see the following resources:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)
- [Sample app for 2.x](#)
- [Kestrel source code](#)

Introduction to ASP.NET Core Module

1/10/2018 • 6 min to read • [Edit Online](#)

By [Tom Dykstra](#), [Rick Strahl](#), and [Chris Ross](#)

ASP.NET Core Module (ANCM) lets you run ASP.NET Core applications behind IIS, using IIS for what it's good at (security, manageability, and lots more) and using [Kestrel](#) for what it's good at (being really fast), and getting the benefits from both technologies at once. **ANCM works only with Kestrel; it isn't compatible with WebListener (in ASP.NET Core 1.x) or HTTP.sys (in 2.x).**

Supported Windows versions:

- Windows 7 and Windows Server 2008 R2 and later

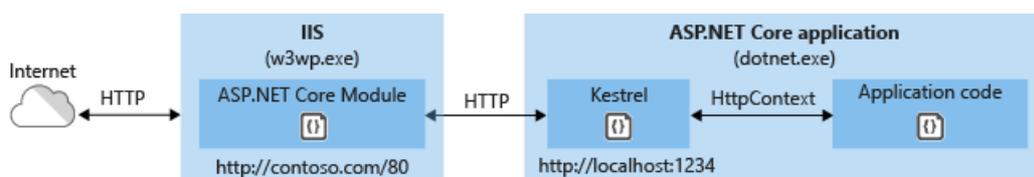
[View or download sample code](#) (how to download)

What ASP.NET Core Module does

ANCM is a native IIS module that hooks into the IIS pipeline and redirects traffic to the backend ASP.NET Core application. Most other modules, such as windows authentication, still get a chance to run. ANCM only takes control when a handler is selected for the request, and handler mapping is defined in the application *web.config* file.

Because ASP.NET Core applications run in a process separate from the IIS worker process, ANCM also does process management. ANCM starts the process for the ASP.NET Core application when the first request comes in and restarts it when it crashes. This is essentially the same behavior as classic ASP.NET applications that run in-process in IIS and are managed by WAS (Windows Activation Service).

Here's a diagram that illustrates the relationship between IIS, ANCM, and ASP.NET Core applications.



Requests come in from the Web and hit the kernel mode Http.Sys driver which routes them into IIS on the primary port (80) or SSL port (443). ANCM forwards the requests to the ASP.NET Core application on the HTTP port configured for the application, which is not port 80/443.

Kestrel listens for traffic coming from ANCM. ANCM specifies the port via environment variable at startup, and the [UseIISIntegration](#) method configures the server to listen on `http://localhost:{port}`. There are additional checks to reject requests not from ANCM. (ANCM does not support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.)

Kestrel picks up requests from ANCM and pushes them into the ASP.NET Core middleware pipeline, which then handles them and passes them on as `HttpContext` instances to application logic. The application's responses are then passed back to IIS, which pushes them back out to the HTTP client that initiated the requests.

ANCM has a few other functions as well:

- Sets environment variables.
- Logs `stdout` output to file storage.
- Forwards Windows authentication tokens.

How to use ANCM in ASP.NET Core apps

This section provides an overview of the process for setting up an IIS server and ASP.NET Core application. For detailed instructions, see [Host on Windows with IIS](#).

Install ANCM

The ASP.NET Core Module has to be installed in IIS on your servers and in IIS Express on your development machines. For servers, ANCM is included in the [.NET Core Windows Server Hosting bundle](#). For development machines, Visual Studio automatically installs ANCM in IIS Express, and in IIS if it is already installed on the machine.

Install the IISIntegration NuGet package

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The [Microsoft.AspNetCore.Server.IISIntegration](#) package is included in the ASP.NET Core metapackages ([Microsoft.AspNetCore](#) and [Microsoft.AspNetCore.All](#)). If you don't use one of the metapackages, install `Microsoft.AspNetCore.Server.IISIntegration` separately. The `IISIntegration` package is an interoperability pack that reads environment variables broadcast by ANCM to set up your app. The environment variables provide configuration information, such as the port to listen on.

Call UseIISIntegration

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The `UseIISIntegration` extension method on `WebHostBuilder` is called automatically when you run with IIS.

If you aren't using one of the ASP.NET Core metapackages and haven't installed the

`Microsoft.AspNetCore.Server.IISIntegration` package, you get a runtime error. If you call `UseIISIntegration` explicitly, you get a compile time error if the package isn't installed.

The `UseIISIntegration` method looks for environment variables that ANCM sets, and it no-ops if they aren't found. This behavior facilitates scenarios like developing and testing on macOS or Linux and deploying to a server that runs IIS. While running on macOS or Linux, Kestrel acts as the web server; but when the app is deployed to the IIS environment, it automatically uses ANCM and IIS.

ANCM port binding overrides other port bindings

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

ANCM generates a dynamic port to assign to the back-end process. The `UseIISIntegration` method picks up this dynamic port and configures Kestrel to listen on `http://localhost:{dynamicPort}/`. This overrides other URL configurations, such as calls to `UseUrls` or [Kestrel's Listen API](#). Therefore, you don't need to call `UseUrls` or Kestrel's `Listen` API when you use ANCM. If you do call `UseUrls` or `Listen`, Kestrel listens on the port you specify when you run the app without IIS.

Configure ANCM options in Web.config

Configuration for the ASP.NET Core Module is stored in the `web.config` file that is located in the application's root folder. Settings in this file point to the startup command and arguments that start your ASP.NET Core app. For sample `web.config` code and guidance on configuration options, see [ASP.NET Core Module Configuration Reference](#).

Run with IIS Express in development

IIS Express can be launched by Visual Studio using the default profile defined by the ASP.NET Core templates.

Proxy configuration uses HTTP protocol and a pairing token

The proxy created between the ANCM and Kestrel uses the HTTP protocol. Using HTTP is a performance optimization where the traffic between the ANCM and Kestrel takes place on a loopback address off of the network interface. There's no risk of eavesdropping the traffic between the ANCM and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the ANCM. The pairing token is also set into a header (`MSAspNetCoreToken`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the ANCM and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, an attacker can't submit requests that bypass the check in the IIS Middleware.

Next steps

For more information, see the following resources:

- [Sample app for this article](#)
- [ASP.NET Core Module source code](#)
- [ASP.NET Core Module Configuration Reference](#)
- [Host on Windows with IIS](#)

HTTP.sys web server implementation in ASP.NET Core

12/12/2017 • 6 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Chris Ross](#)

NOTE

This topic applies only to ASP.NET Core 2.0 and later. In earlier versions of ASP.NET Core, HTTP.sys is named [WebListener](#).

HTTP.sys is a [web server for ASP.NET Core](#) that runs only on Windows. It's built on the [Http.Sys kernel mode driver](#). HTTP.sys is an alternative to [Kestrel](#) that offers some features that Kestrel doesn't. **HTTP.sys can't be used with IIS or IIS Express, as it's incompatible with the ASP.NET Core Module.**

HTTP.sys supports the following features:

- [Windows Authentication](#)
- Port sharing
- HTTPS with SNI
- HTTP/2 over TLS (Windows 10)
- Direct file transmission
- Response caching
- WebSockets (Windows 8)

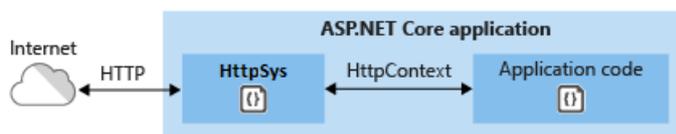
Supported Windows versions:

- Windows 7 and Windows Server 2008 R2 and later

[View or download sample code](#) ([how to download](#))

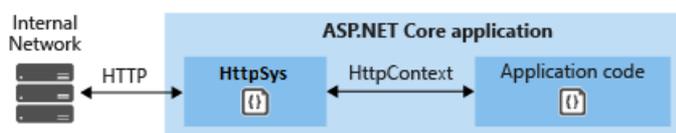
When to use HTTP.sys

HTTP.sys is useful for deployments where you need to expose the server directly to the Internet without using IIS.



Because it's built on Http.Sys, HTTP.sys doesn't require a reverse proxy server for protection against attacks. Http.Sys is mature technology that protects against many kinds of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of Http.Sys.

HTTP.sys is a good choice for internal deployments when you need a feature not available in Kestrel, such as Windows authentication.



How to use HTTP.sys

Here's an overview of setup tasks for the host OS and your ASP.NET Core application.

Configure Windows Server

- Install the version of .NET that your application requires, such as [.NET Core](#) or [.NET Framework](#).
- Preregister URL prefixes to bind to HTTP.sys, and set up SSL certificates

If you don't preregister URL prefixes in Windows, you have to run your application with administrator privileges. The only exception is if you bind to localhost using HTTP (not HTTPS) with a port number greater than 1024; in that case, administrator privileges aren't required.

For details, see [How to preregister prefixes and configure SSL](#) later in this article.

- Open firewall ports to allow traffic to reach HTTP.sys.

You can use *netsh.exe* or [PowerShell cmdlets](#).

There are also [Http.Sys registry settings](#).

Configure your ASP.NET Core application to use HTTP.sys

- No package install is needed if you use the [Microsoft.AspNetCore.All](#) metapackage. The [Microsoft.AspNetCore.Server.HttpSys](#) package is included in the metapackage.
- Call the `UseHttpSys` extension method on `WebHostBuilder` in your `Main` method, specifying any [HTTP.sys options](#) that you need, as shown in the following example:

```
public static void Main(string[] args)
{
    Console.WriteLine("Running demo with HTTP.sys.");

    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseHttpSys(options =>
        {
            options.Authentication.Schemes = AuthenticationSchemes.None;
            options.Authentication.AllowAnonymous = true;
            options.MaxConnections = 100;
            options.MaxRequestBodySize = 30000000;
            options.UrlPrefixes.Add("http://localhost:5000");
        })
        .Build();
```

Configure HTTP.sys options

Here are some of the HTTP.sys settings and limits that you can configure.

Maximum client connections

The maximum number of concurrent open TCP connections can be set for the entire application with the following code in *Program.cs*:

```

.UseHttpSys(options =>
{
    options.Authentication.Schemes = AuthenticationSchemes.None;
    options.Authentication.AllowAnonymous = true;
    options.MaxConnections = 100;
    options.MaxRequestBodySize = 30000000;
    options.UrlPrefixes.Add("http://localhost:5000");
})

```

The maximum number of connections is unlimited (null) by default.

Maximum request body size

The default maximum request body size is 30,000,000 bytes, which is approximately 28.6MB.

The recommended way to override the limit in an ASP.NET Core MVC app is to use the [RequestSizeLimit](#) attribute on an action method:

```

[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()

```

Here's an example that shows how to configure the constraint for the entire application, every request:

```

.UseHttpSys(options =>
{
    options.Authentication.Schemes = AuthenticationSchemes.None;
    options.Authentication.AllowAnonymous = true;
    options.MaxConnections = 100;
    options.MaxRequestBodySize = 30000000;
    options.UrlPrefixes.Add("http://localhost:5000");
})

```

You can override the setting on a specific request in *Startup.cs*:

```

public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    var serverAddressesFeature = app.ServerFeatures.Get<IServerAddressesFeature>();

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        context.Features.Get<IHttpMaxRequestBodySizeFeature>()
            .MaxRequestBodySize = 10 * 1024;

        context.Response.ContentType = "text/html";
        await context.Response.WriteAsync("<p>Hosted by HTTP.sys</p>");

        if (serverAddressesFeature != null)
        {
            await context.Response.WriteAsync($"<p>Listening on the following addresses: {string.Join(", ",
serverAddressesFeature.Addresses)}</p>");
        }

        await context.Response.WriteAsync($"<p>Request URL: {context.Request.GetDisplayUrl()}</p>");
    });
}

```

An exception is thrown if you try to configure the limit on a request after the application has started reading the request. There's an `IsReadOnly` property that tells you if the `MaxRequestBodySize` property is in read-only state,

meaning it's too late to configure the limit.

For information about other HTTP.sys options, see [HttpSysOptions](#).

Configure URLs and ports to listen on

By default ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, you can use the `UseUrls` extension method, the `urls` command-line argument, the `ASPNETCORE_URLS` environment variable, or the `UrlPrefixes` property on [HttpSysOptions](#). The following code example uses `UrlPrefixes`.

```
public static void Main(string[] args)
{
    Console.WriteLine("Running demo with HTTP.sys.");

    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseHttpSys(options =>
        {
            options.Authentication.Schemes = AuthenticationSchemes.None;
            options.Authentication.AllowAnonymous = true;
            options.MaxConnections = 100;
            options.MaxRequestBodySize = 30000000;
            options.UrlPrefixes.Add("http://localhost:5000");
        })
        .Build();
```

An advantage of `UrlPrefixes` is that you get an error message immediately if you try to add a prefix that is formatted wrong. An advantage of `UseUrls` (shared with `urls` and `ASPNETCORE_URLS`) is that you can more easily switch between Kestrel and HTTP.sys.

If you use both `UseUrls` (or `urls` or `ASPNETCORE_URLS`) and `UrlPrefixes`, the settings in `UrlPrefixes` override the ones in `UseUrls`. For more information, see [Hosting](#).

HTTP.sys uses the [HTTP Server API UriPrefix string formats](#).

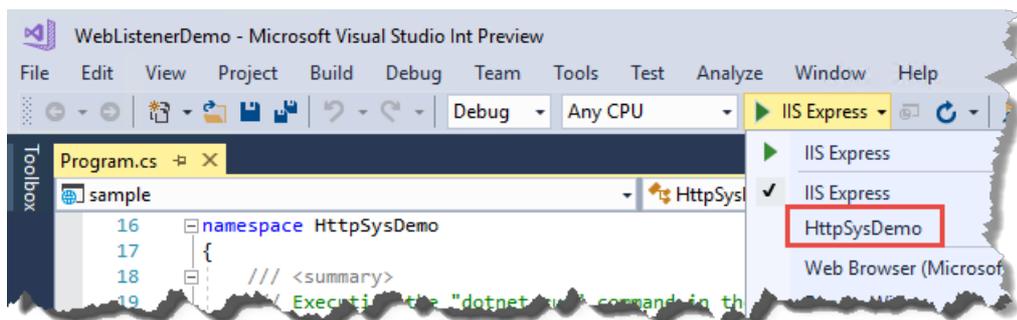
NOTE

Make sure that you specify the same prefix strings in `UseUrls` or `UrlPrefixes` that you preregister on the server.

Don't use IIS

Make sure your application isn't configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console application, manually change the selected profile, as shown in the following screen shot.



Preregister URL prefixes and configure SSL

Both IIS and HTTP.sys rely on the underlying Http.Sys kernel mode driver to listen for requests and do initial processing. In IIS, the management UI gives you a relatively easy way to configure everything. However, you need to configure Http.Sys yourself. The built-in tool for doing that is *netsh.exe*.

With *netsh.exe* you can reserve URL prefixes and assign SSL certificates. The tool requires administrative privileges.

The following example shows the minimum needed to reserve URL prefixes for ports 80 and 443:

```
netsh http add urlacl url=http://+:80/ user=Users
netsh http add urlacl url=https://+:443/ user=Users
```

The following example shows how to assign an SSL certificate:

```
netsh http add sslcert ipport=0.0.0.0:443 certhash=MyCertHash_Here appid={00000000-0000-0000-0000-000000000000}"
```

Here is the reference documentation for *netsh.exe*:

- [Netsh Commands for Hypertext Transfer Protocol \(HTTP\)](#)
- [UrlPrefix Strings](#)

The following resources provide detailed instructions for several scenarios. Articles that refer to HttpListener apply equally to HTTP.sys, as both are based on Http.Sys.

- [How to: Configure a Port with an SSL Certificate](#)
- [HTTPS Communication - HttpListener based Hosting and Client Certification](#) This is a third-party blog and is fairly old but still has useful information.
- [How To: Walkthrough Using HttpListener or Http Server unmanaged code \(C++\) as an SSL Simple Server](#) This too is an older blog with useful information.

Here are some third-party tools that can be easier to use than the *netsh.exe* command line. These are not provided by or endorsed by Microsoft. The tools run as administrator by default, since *netsh.exe* itself requires administrator privileges.

- [http.sys Manager](#) provides UI for listing and configuring SSL certificates and options, prefix reservations, and certificate trust lists.
- [HttpConfig](#) lets you list or configure SSL certificates and URL prefixes. The UI is more refined than http.sys Manager and exposes a few more configuration options, but otherwise it provides similar functionality. It cannot create a new certificate trust list (CTL), but can assign existing ones.

For generating self-signed SSL certificates on Windows, you can use the PowerShell cmdlet [New-SelfSignedCertificate](#). For a third-party tool that makes it easier for you to generate self-signed certificates, see [SelfCert](#).

On macOS and Linux you can create a self-signed certificate using [OpenSSL](#).

Next steps

For more information, see the following resources:

- [Sample app for this article](#)
- [HTTP.sys source code](#)

- [Hosting](#)

Globalization and localization in ASP.NET Core

1/4/2018 • 16 min to read • [Edit Online](#)

By [Rick Anderson](#), [Damien Bowden](#), [Bart Calixto](#), [Nadeem Afana](#), and [Hisham Bin Ateya](#)

Creating a multilingual website with ASP.NET Core will allow your site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

Internationalization involves [Globalization](#) and [Localization](#). Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas.

Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

1. Make the app's content localizable
2. Provide localized resources for the languages and cultures you support
3. Implement a strategy to select the language/culture for each request

Make the app's content localizable

Introduced in ASP.NET Core, `IStringLocalizer` and `IStringLocalizer<T>` were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the [ResourceManager](#) and [ResourceReader](#) to provide culture-specific resources at run time. The simple interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require you to store the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development. The code below shows how to wrap the string "About Title" for localization.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.StarterWeb.Controllers
{
    [Route("api/[controller]")]
    public class AboutController : Controller
    {
        private readonly IStringLocalizer<AboutController> _localizer;

        public AboutController(IStringLocalizer<AboutController> localizer)
        {
            _localizer = localizer;
        }

        [HttpGet]
        public string Get()
        {
            return _localizer["About Title"];
        }
    }
}
```

In the code above, the `IStringLocalizer<T>` implementation comes from [Dependency Injection](#). If the localized value of "About Title" is not found, then the indexer key is returned, that is, the string "About Title". You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop your app with your default language and prepare it for the localization step without first creating a default resource file. Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers the new workflow of not having a default language `.resx` file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers will prefer the traditional work flow as it can make it easier to work with longer string literals and make it easier to update localized strings.

Use the `IHtmlLocalizer<T>` implementation for resources that contain HTML. `IHtmlLocalizer` HTML encodes arguments that are formatted in the resource string, but does not HTML encode the resource string itself. In the sample highlighted below, only the value of `name` parameter is HTML encoded.

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.StarterWeb.Controllers
{
    public class BookController : Controller
    {
        private readonly IHtmlLocalizer<BookController> _localizer;

        public BookController(IHtmlLocalizer<BookController> localizer)
        {
            _localizer = localizer;
        }

        public IActionResult Hello(string name)
        {
            ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

            return View();
        }
    }
}
```

Note: You generally want to only localize text and not HTML.

At the lowest level, you can get `IStringLocalizerFactory` out of [Dependency Injection](#):

```
{
    public class TestController : Controller
    {
        private readonly IStringLocalizer _localizer;
        private readonly IStringLocalizer _localizer2;

        public TestController(IStringLocalizerFactory factory)
        {
            var type = typeof(SharedResource);
            var assemblyName = new AssemblyName(type.GetTypeInfo().Assembly.FullName);
            _localizer = factory.Create(type);
            _localizer2 = factory.Create("SharedResource", assemblyName.Name);
        }

        public IActionResult About()
        {
            ViewData["Message"] = _localizer["Your application description page."]
                + " loc 2: " + _localizer2["Your application description page."];
        }
    }
}
```

The code above demonstrates each of the two factory create methods.

You can partition your localized strings by controller, area, or have just one container. In the sample app, a dummy class named `SharedResource` is used for shared resources.

```
// Dummy class to group shared resources

namespace Localization.StarterWeb
{
    public class SharedResource
    {
    }
}
```

Some developers use the `Startup` class to contain global or shared strings. In the sample below, the `InfoController` and the `SharedResource` localizers are used:

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
        IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
            " Info resx " + _localizer["Hello!"];
        return msg;
    }
}
```

View localization

The `IViewLocalizer` service provides localized strings for a [view](#). The `ViewLocalizer` class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

```
@using Microsoft.AspNetCore.Mvc.Localization

@Inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There is no option to use a global shared resource file. `ViewLocalizer` implements the localizer using `IHtmlLocalizer`, so Razor doesn't HTML encode the localized string. You can parameterize resource strings and `IViewLocalizer` will HTML encode the parameters, but not the resource string. Consider the following Razor markup:

```
@Localizer["<i>Hello</i> <b>{0}</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following:

KEY	VALUE
<code><i>Hello</i> {0}</code>	<code><i>Bonjour</i> {0} !</code>

The rendered view would contain the HTML markup from the resource file.

Note: You generally want to only localize text and not HTML.

To use a shared resource file in a view, inject `IHtmlLocalizer<T>`:

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.StarterWeb.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

DataAnnotations error messages are localized with `IStringLocalizer<T>`. Using the option

`ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

- Resources/ViewModels.Account.RegisterViewModel.fr.resx
- Resources/ViewModels/Account/RegisterViewModel.fr.resx

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid e-mail address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

In ASP.NET Core MVC 1.1.0 and higher, non-validation attributes are localized. ASP.NET Core MVC 1.0 does **not** look up localized strings for non-validation attributes.

Using one resource string for multiple classes

The following code shows how to use one resource string for validation attributes with multiple classes:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```

In the preceding code, `SharedResource` is the class corresponding to the resx where your validation messages are stored. With this approach, `DataAnnotations` will only use `SharedResource`, rather than the resource for each class.

Provide localized resources for the languages and cultures you support

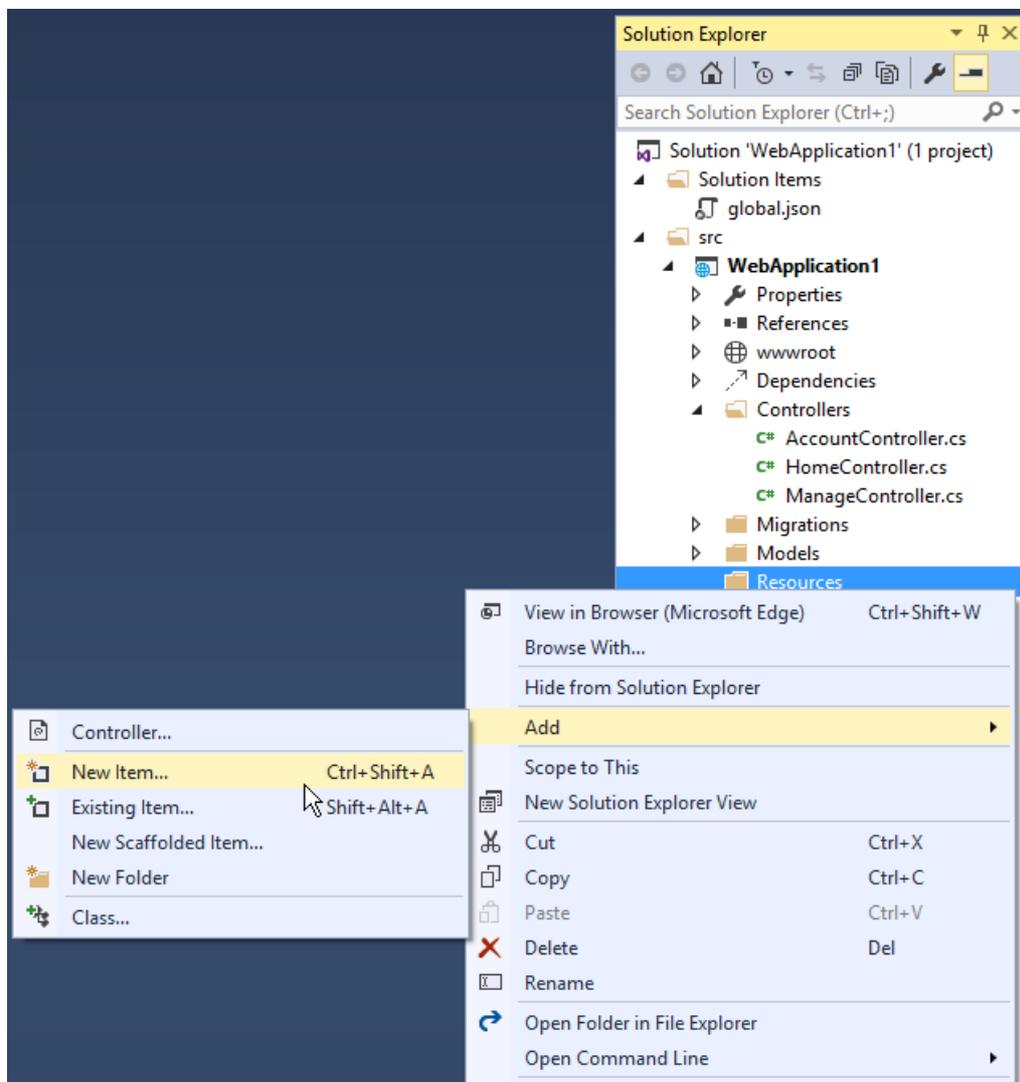
SupportedCultures and SupportedUICultures

ASP.NET Core allows you to specify two culture values, `SupportedCultures` and `SupportedUICultures`. The `CultureInfo` object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See `CultureInfo.CurrentCulture` for more info on how the server gets the Culture. The `SupportedUICultures` determines which translates strings (from `.resx` files) are looked up by the `ResourceManager`. The `ResourceManager` simply looks up culture-specific strings that is determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18, 2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

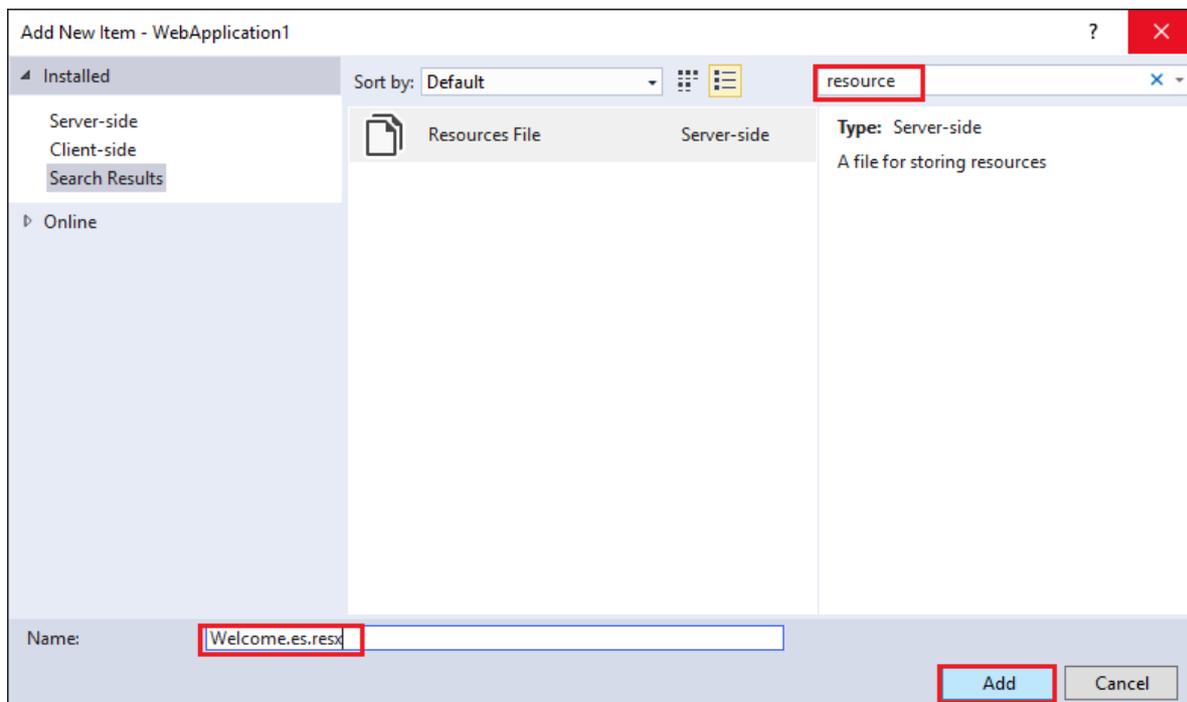
Resource files

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated `.resx` resource files. For example, you might want to create Spanish resource file named `Welcome.es.resx` containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

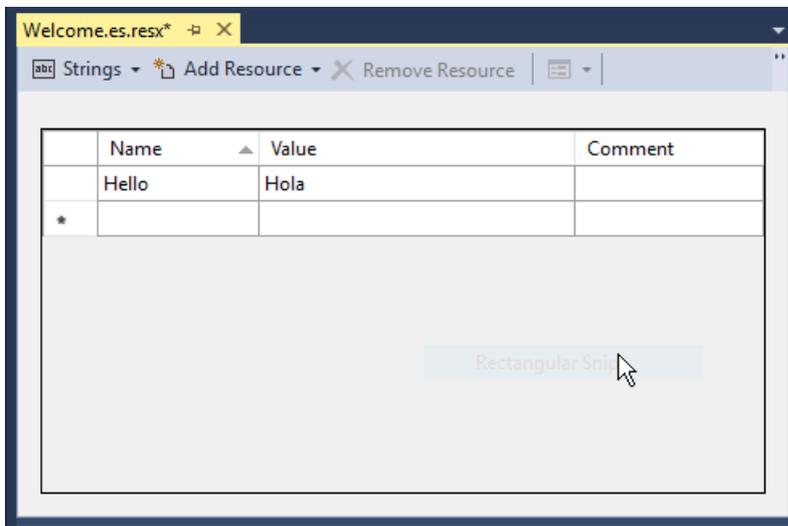
1. In **Solution Explorer**, right click on the folder which will contain the resource file > **Add** > **New Item**.



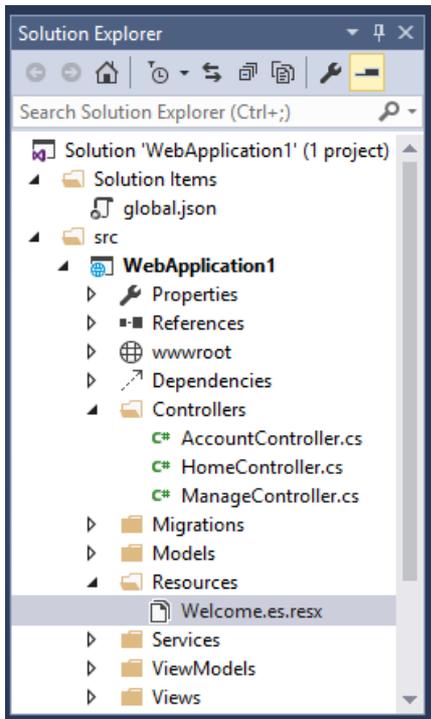
2. In the **Search installed templates** box, enter "resource" and name the file.



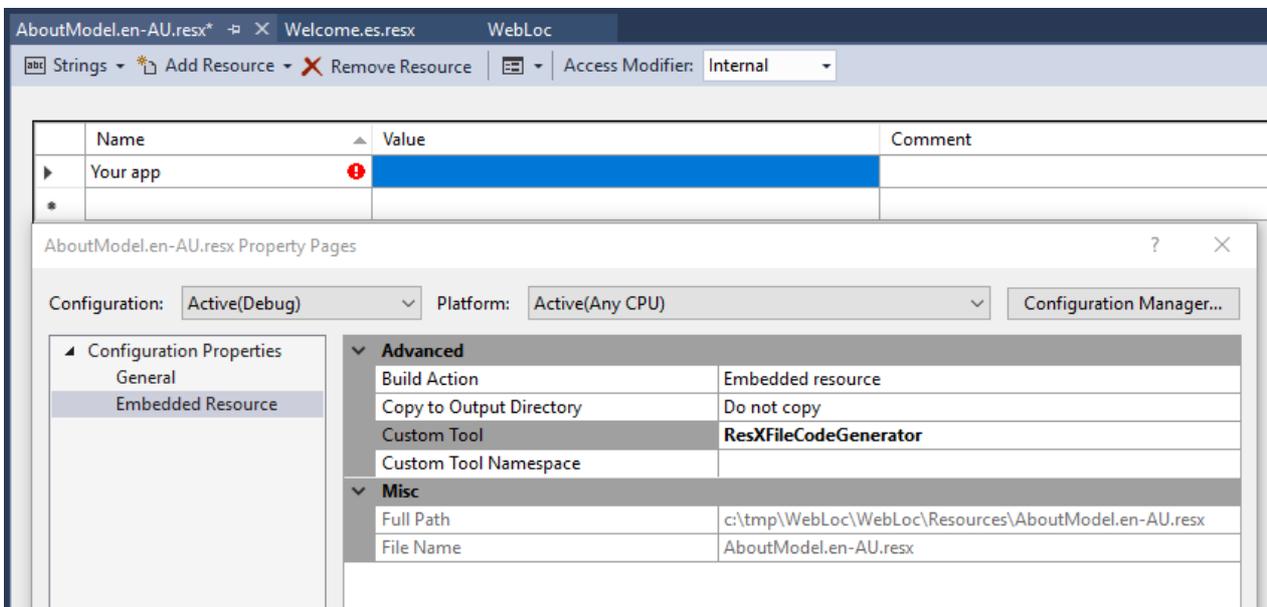
3. Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.



Visual Studio shows the *Welcome.es.resx* file.



If you are using Visual Studio 2017 Preview version 15.3, you'll get an error indicator in the resource editor. Remove the *ResXFileCodeGenerator* value from the *Custom Tool* properties grid to prevent this error message:



Alternatively, you can ignore this error. We hope to fix this in the next release.

Resource file naming

Resources are named for the full type name of their class minus the assembly name. For example, a French resource in a project whose main assembly is `LocalizationWebsite.Web.dll` for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. A resource for the class `LocalizationWebsite.Web.Controllers.HomeController` would be named *Controllers.HomeController.fr.resx*. If your targeted class's namespace is not the same as the assembly name you will need the full type name. For example, in the sample project a resource for the type `ExtraNamespace.Tools` would be named *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

RESOURCE NAME	DOT OR PATH NAMING
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the *Views/Home/About.cshtml* view could be either of the following:

- Resources/Views/Home/About.fr.resx
- Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the view.

Culture fallback behavior

As an example, if you remove the ".fr" culture designator and you have the culture set to French, the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource for when nothing meets your requested culture. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Generate resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core; you typically don't have a default *.resx* resource file (A *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio will not generate the class file. We anticipate that many developers will **not** create a default language resource file.

Add Other Cultures

Each language and culture combination (other than the default language) requires a unique resource file. You create resource files for different cultures and locales by creating new resource files in which the ISO language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These ISO codes are placed between the file name and the *.resx* file name extension, as in *Welcome.es-MX.resx* (Spanish/Mexico). To specify a culturally neutral language, remove the country code (`MX` in the preceding example). The culturally neutral Spanish resource file name is *Welcome.es.resx*.

Implement a strategy to select the language/culture for each request

Configure localization

Localization is configured in the `ConfigureServices` method:

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- `AddLocalization` Adds the localization services to the services container. The code above also sets the resources path to "Resources".
- `AddViewLocalization` Adds support for localized view files. In this sample view localization is based on the view file suffix. For example "fr" in the *Index.fr.cshtml* file.
- `AddDataAnnotationsLocalization` Adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

Localization middleware

The current culture on a request is set in the localization [Middleware](#). The localization middleware is enabled in the `Configure` method of *Program.cs* file. Note, the localization middleware must be configured before any middleware which might check the request culture (for example, `app.UseMvcWithDefaultRoute()`).

```

var supportedCultures = new[]
{
    new CultureInfo(enUSCulture),
    new CultureInfo("en-AU"),
    new CultureInfo("en-GB"),
    new CultureInfo("en"),
    new CultureInfo("es-ES"),
    new CultureInfo("es-MX"),
    new CultureInfo("es"),
    new CultureInfo("fr-FR"),
    new CultureInfo("fr"),
};

app.UseRequestLocalization(new RequestLocalizationOptions
{
    DefaultRequestCulture = new RequestCulture(enUSCulture),
    // Formatting numbers, dates, etc.
    SupportedCultures = supportedCultures,
    // UI strings that we have localized.
    SupportedUICultures = supportedCultures
});

app.UseStaticFiles();
// To configure external authentication,
// see: http://go.microsoft.com/fwlink/?LinkID=532715
app.UseAuthentication();
app.UseMvcWithDefaultRoute();

```

`UseRequestLocalization` initializes a `RequestLocalizationOptions` object. On every request the list of `RequestCultureProvider` in the `RequestLocalizationOptions` is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

1. `QueryStringRequestCultureProvider`
2. `CookieRequestCultureProvider`
3. `AcceptLanguageHeaderRequestCultureProvider`

The default list goes from most specific to least specific. Later in the article we'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the `DefaultRequestCulture` is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the [culture and UI culture](#). For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the `QueryStringRequestCultureProvider` is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

If you only pass in one of the two (`culture` or `ui-culture`), the query string provider will set both values using the one you passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

```
http://localhost:5000/?culture=es-MX
```

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the `MakeCookieValue` method to create a cookie.

The `CookieRequestCultureProvider.DefaultCookieName` returns the default cookie name used to track the user's preferred culture information. The default cookie name is ".AspNetCore.Culture".

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

```
c=en-UK|uic=en-US
```

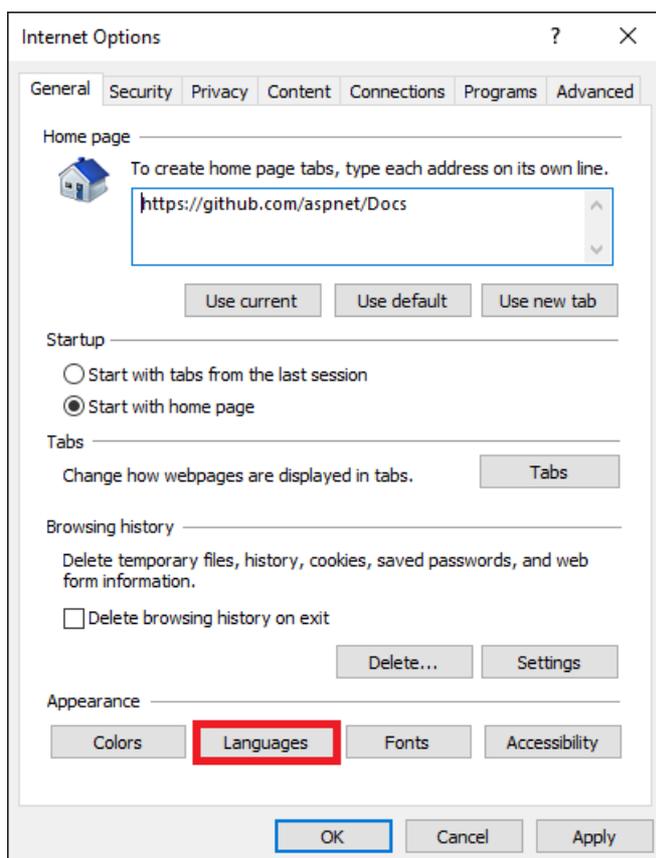
If you only specify one of culture info and UI culture, the specified culture will be used for both culture info and UI culture.

The Accept-Language HTTP header

The [Accept-Language header](#) is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request is not an infallible way to detect the user's preferred language (see [Setting language preferences in a browser](#)). A production app should include a way for a user to customize their choice of culture.

Set the Accept-Language HTTP header in IE

1. From the gear icon, tap **Internet Options**.
2. Tap **Languages**.



3. Tap **Set Language Preferences**.
4. Tap **Add a language**.
5. Add the language.
6. Tap the language, then tap **Move Up**.

Use a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

```

private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture, uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
});

```

Use `RequestLocalizationOptions` to add or remove localization providers.

Set the culture programmatically

This sample **Localization.StarterWeb** project on [GitHub](#) contains UI to set the `Culture`. The `Views/Shared/_SelectLanguagePartial.cshtml` file allows you to select the culture from the list of supported cultures:

```

@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@Inject IViewLocalizer Localizer
@Inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~//" : $"{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
        asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
        method="post" class="form-horizontal" role="form">
        <label asp-for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]</label>
        <select name="culture"
            onchange="this.form.submit();"
            asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
            </select>
        </form>
    </div>

```

The `Views/Shared/_SelectLanguagePartial.cshtml` file is added to the `footer` section of the layout file so it will be available to all views:

```

<div class="container body-content" style="margin-top:60px">
  @RenderBody()
  <hr>
  <footer>
    <div class="row">
      <div class="col-md-6">
        <p>&copy; @System.DateTime.Now.Year - Localization.StarterWeb</p>
      </div>
      <div class="col-md-6 text-right">
        @await Html.PartialAsync("_SelectLanguagePartial")
      </div>
    </div>
  </footer>
</div>

```

The `SetLanguage` method sets the culture cookie.

```

[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}

```

You can't plug in the `_SelectLanguagePartial.cshtml` to sample code for this project. The **Localization.StarterWeb** project on [GitHub](#) has code to flow the `RequestLocalizationOptions` to a Razor partial through the [Dependency Injection](#) container.

Globalization and localization terms

The process of localizing your app also requires a basic understanding of relevant character sets commonly used in modern software development and an understanding of the issues associated with them. Although all computers store text as numbers (codes), different systems store the same text using different numbers. The localization process refers to translating the app user interface (UI) for a specific culture/locale.

[Localizability](#) is an intermediate process for verifying that a globalized app is ready for localization.

The [RFC 4646](#) format for the culture name is `<languagecode2>-<country/regioncode2>`, where `<languagecode2>` is the language code and `<country/regioncode2>` is the subculture code. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). [RFC 4646](#) is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. See [Language Culture Name](#).

Internationalization is often abbreviated to "I18N". The abbreviation takes the first and last letters and the number of letters between them, so 18 stands for the number of letters between the first "I" and the last "N". The same applies to Globalization (G11N), and Localization (L10N).

Terms:

- Globalization (G11N): The process of making an app support different languages and regions.
- Localization (L10N): The process of customizing an app for a given language and region.
- Internationalization (I18N): Describes both globalization and localization.
- Culture: It is a language and, optionally, a region.

- Neutral culture: A culture that has a specified language, but not a region. (for example "en", "es")
- Specific culture: A culture that has a specified language and region. (for example "en-US", "en-GB", "es-CL")
- Parent culture: The neutral culture that contains a specific culture. (for example, "en" is the parent culture of "en-US" and "en-GB")
- Locale: A locale is the same as a culture.

Additional resources

- [Localization.StarterWeb project](#) used in the article.
- [Resource Files in Visual Studio](#)
- [Resources in .resx Files](#)

Configure portable object localization with Orchard Core

10/2/2017 • 6 min to read • [Edit Online](#)

By [Sébastien Ros](#) and [Scott Addie](#)

This article walks through the steps for using Portable Object (PO) files in an ASP.NET Core application with the [Orchard Core](#) framework.

Note: Orchard Core is not a Microsoft product. Consequently, Microsoft provides no support for this feature.

[View or download sample code](#) ([how to download](#))

What is a PO file?

PO files are distributed as text files containing the translated strings for a given language. Some advantages of using PO files instead `.resx` files include:

- PO files support pluralization; `.resx` files don't support pluralization.
- PO files aren't compiled like `.resx` files. As such, specialized tooling and build steps aren't required.
- PO files work well with collaborative online editing tools.

Example

Here is a sample PO file containing the translation for two strings in French, including one with its plural form:

fr.po

```
#: Services/EmailService.cs:29
msgid "Enter a comma separated list of email addresses."
msgstr "Entrez une liste d'emails séparés par une virgule."

#: Views/Email.cshtml:112
msgid "The email address is \"{0}\"."
msgid_plural "The email addresses are \"{0}\"."
msgstr[0] "L'adresse email est \"{0}\"."
msgstr[1] "Les adresses email sont \"{0}\""
```

This example uses the following syntax:

- `#:` : A comment indicating the context of the string to be translated. The same string might be translated differently depending on where it is being used.
- `msgid` : The untranslated string.
- `msgstr` : The translated string.

In the case of pluralization support, more entries can be defined.

- `msgid_plural` : The untranslated plural string.
- `msgstr[0]` : The translated string for the case 0.
- `msgstr[N]` : The translated string for the case N.

The PO file specification can be found [here](#).

Configuring PO file support in ASP.NET Core

This example is based on an ASP.NET Core MVC application generated from a Visual Studio 2017 project template.

Referencing the package

Add a reference to the `OrchardCore.Localization.Core` NuGet package. It is available on [MyGet](https://www.myget.org/F/orchardcore-preview/api/v3/index.json) at the following package source: <https://www.myget.org/F/orchardcore-preview/api/v3/index.json>

The `.csproj` file now contains a line similar to the following (version number may vary):

```
<PackageReference Include="OrchardCore.Localization.Core" Version="1.0.0-beta1-3187" />
```

Registering the service

Add the required services to the `ConfigureServices` method of `Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix);

    services.AddPortableObjectLocalization();

    services.Configure<RequestLocalizationOptions>(options =>
    {
        var supportedCultures = new List<CultureInfo>
        {
            new CultureInfo("en-US"),
            new CultureInfo("en"),
            new CultureInfo("fr-FR"),
            new CultureInfo("fr")
        };

        options.DefaultRequestCulture = new RequestCulture("en-US");
        options.SupportedCultures = supportedCultures;
        options.SupportedUICultures = supportedCultures;
    });
}
```

Add the required middleware to the `Configure` method of `Startup.cs`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseRequestLocalization();

    app.UseMvcWithDefaultRoute();
}
```

Add the following code to your Razor view of choice. `About.cshtml` is used in this example.

```
@using Microsoft.AspNetCore.Mvc.Localization
@Inject IViewLocalizer Localizer

<p>@Localizer["Hello world!"]</p>
```

An `IViewLocalizer` instance is injected and used to translate the text "Hello world!".

Creating a PO file

Create a file named `.po` in your application root folder. In this example, the file name is `fr.po` because the French language is used:

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

This file stores both the string to translate and the French-translated string. Translations revert to their parent culture, if necessary. In this example, the `fr.po` file is used if the requested culture is `fr-FR` or `fr-CA`.

Testing the application

Run your application, and navigate to the URL `/Home/About`. The text **Hello world!** is displayed.

Navigate to the URL `/Home/About?culture=fr-FR`. The text **Bonjour le monde!** is displayed.

Pluralization

PO files support pluralization forms, which is useful when the same string needs to be translated differently based on a cardinality. This task is made complicated by the fact that each language defines custom rules to select which string to use based on the cardinality.

The Orchard Localization package provides an API to invoke these different plural forms automatically.

Creating pluralization PO files

Add the following content to the previously mentioned `fr.po` file:

```
msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

See [What is a PO file?](#) for an explanation of what each entry in this example represents.

Adding a language using different pluralization forms

English and French strings were used in the previous example. English and French have only two pluralization forms and share the same form rules, which is that a cardinality of one is mapped to the first plural form. Any other cardinality is mapped to the second plural form.

Not all languages share the same rules. This is illustrated with the Czech language, which has three plural forms.

Create the `cs.po` file as follows, and note how the pluralization needs three different translations:

```
msgid "Hello world!"
msgstr "Ahoj světe!!"
```

```
msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Existuje jedna položka."
msgstr[1] "Existují {0} položky."
msgstr[2] "Existuje {0} položek."
```

To accept Czech localizations, add `"cs"` to the list of supported cultures in the `ConfigureServices` method:

```
var supportedCultures = new List<CultureInfo>
{
    new CultureInfo("en-US"),
    new CultureInfo("en"),
    new CultureInfo("fr-FR"),
    new CultureInfo("fr"),
    new CultureInfo("cs")
};
```

Edit the `Views/Home/About.cshtml` file to render localized, plural strings for several cardinalities:

```
<p>@Localizer.Plural(1, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(2, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(5, "There is one item.", "There are {0} items.")</p>
```

Note: In a real world scenario, a variable would be used to represent the count. Here, we repeat the same code with three different values to expose a very specific case.

Upon switching cultures, you see the following:

For `/Home/About` :

```
There is one item.
There are 2 items.
There are 5 items.
```

For `/Home/About?culture=fr` :

```
Il y a un élément.
Il y a 2 éléments.
Il y a 5 éléments.
```

For `/Home/About?culture=cs` :

```
Existuje jedna položka.
Existují 2 položky.
Existuje 5 položek.
```

Note that for the Czech culture, the three translations are different. The French and English cultures share the same construction for the two last translated strings.

Advanced tasks

Contextualizing strings

Applications often contain the strings to be translated in several places. The same string may have a different translation in certain locations within an app (Razor views or class files). A PO file supports the notion of a file context, which can be used to categorize the string being represented. Using a file context, a string can be translated differently, depending on the file context (or lack of a file context).

The PO localization services use the name of the full class or the view that is used when translating a string. This is accomplished by setting the value on the `msgctxt` entry.

Consider a minor addition to the previous *fr.po* example. A Razor view located at *Views/Home/About.cshtml* can be defined as the file context by setting the reserved `msgctxt` entry's value:

```
msgctxt "Views.Home.About"
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

With the `msgctxt` set as such, text translation occurs when navigating to `/Home/About?culture=fr-FR`. The translation won't occur when navigating to `/Home/Contact?culture=fr-FR`.

When no specific entry is matched with a given file context, Orchard Core's fallback mechanism looks for an appropriate PO file without a context. Assuming there is no specific file context defined for *Views/Home/Contact.cshtml*, navigating to `/Home/Contact?culture=fr-FR` loads a PO file such as:

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

Changing the location of PO files

The default location of PO files can be changed in `ConfigureServices`:

```
services.AddPortableObjectLocalization(options => options.ResourcesPath = "Localization");
```

In this example, the PO files are loaded from the *Localization* folder.

Implementing a custom logic for finding localization files

When more complex logic is needed to locate PO files, the `OrchardCore.Localization.PortableObject.ILocalizationFileLocationProvider` interface can be implemented and registered as a service. This is useful when PO files can be stored in varying locations or when the files have to be found within a hierarchy of folders.

Using a different default pluralized language

The package includes a `Plural` extension method that is specific to two plural forms. For languages requiring more plural forms, create an extension method. With an extension method, you won't need to provide any localization file for the default language — the original strings are already available directly in the code.

You can use the more generic `Plural(int count, string[] pluralForms, params object[] arguments)` overload which accepts a string array of translations.

Request Features in ASP.NET Core

10/2/2017 • 2 min to read • [Edit Online](#)

By [Steve Smith](#)

Web server implementation details related to HTTP requests and responses are defined in interfaces. These interfaces are used by server implementations and middleware to create and modify the application's hosting pipeline.

Feature interfaces

ASP.NET Core defines a number of HTTP feature interfaces in `Microsoft.AspNetCore.Http.Features` which are used by servers to identify the features they support. The following feature interfaces handle requests and return responses:

`IHttpRequestFeature` Defines the structure of an HTTP request, including the protocol, path, query string, headers, and body.

`IHttpResponseFeature` Defines the structure of an HTTP response, including the status code, headers, and body of the response.

`IHttpAuthenticationFeature` Defines support for identifying users based on a `ClaimsPrincipal` and specifying an authentication handler.

`IHttpUpgradeFeature` Defines support for [HTTP Upgrades](#), which allow the client to specify which additional protocols it would like to use if the server wishes to switch protocols.

`IHttpBufferingFeature` Defines methods for disabling buffering of requests and/or responses.

`IHttpConnectionFeature` Defines properties for local and remote addresses and ports.

`IHttpRequestLifetimeFeature` Defines support for aborting connections, or detecting if a request has been terminated prematurely, such as by a client disconnect.

`IHttpSendFileFeature` Defines a method for sending files asynchronously.

`IHttpWebSocketFeature` Defines an API for supporting web sockets.

`IHttpRequestIdentifierFeature` Adds a property that can be implemented to uniquely identify requests.

`ISessionFeature` Defines `ISessionFactory` and `ISession` abstractions for supporting user sessions.

`ITlsConnectionFeature` Defines an API for retrieving client certificates.

`ITlsTokenBindingFeature` Defines methods for working with TLS token binding parameters.

NOTE

`ISessionFeature` is not a server feature, but is implemented by the `SessionMiddleware` (see [Managing Application State](#)).

Feature collections

The `Features` property of `HttpContext` provides an interface for getting and setting the available HTTP features

for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features.

Middleware and request features

While servers are responsible for creating the feature collection, middleware can both add to this collection and consume features from the collection. For example, the `StaticFileMiddleware` accesses the `IHttpSendFileFeature` feature. If the feature exists, it is used to send the requested static file from its physical path. Otherwise, a slower alternative method is used to send the file. When available, the `IHttpSendFileFeature` allows the operating system to open the file and perform a direct kernel mode copy to the network card.

Additionally, middleware can add to the feature collection established by the server. Existing features can even be replaced by middleware, allowing the middleware to augment the functionality of the server. Features added to the collection are available immediately to other middleware or the underlying application itself later in the request pipeline.

By combining custom server implementations and specific middleware enhancements, the precise set of features an application requires can be constructed. This allows missing features to be added without requiring a change in server, and ensures only the minimal amount of features are exposed, thus limiting attack surface area and improving performance.

Summary

Feature interfaces define specific HTTP features that a given request may support. Servers define collections of features, and the initial set of features supported by that server, but middleware can be used to enhance these features.

Additional Resources

- [Servers](#)
- [Middleware](#)
- [Open Web Interface for .NET \(OWIN\)](#)

Primitives in ASP.NET Core

11/29/2017 • 1 min to read • [Edit Online](#)

ASP.NET Core primitives are low-level building blocks shared by framework extensions. You can use these building blocks in your own code.

[Detect changes with Change Tokens](#)

Detect changes with change tokens in ASP.NET Core

11/29/2017 • 9 min to read • [Edit Online](#)

By [Luke Latham](#)

A *change token* is a general-purpose, low-level building block used to track changes.

[View or download sample code \(how to download\)](#)

IChangeToken interface

`IChangeToken` propagates notifications that a change has occurred. `IChangeToken` resides in the `Microsoft.Extensions.Primitives` namespace. For apps that don't use the `Microsoft.AspNetCore.All` metapackage, reference the `Microsoft.Extensions.Primitives` NuGet package in the project file.

`IChangeToken` has two properties:

- `ActiveChangedCallbacks` indicate if the token proactively raises callbacks. If `ActiveChangedCallbacks` is set to `false`, a callback is never called, and the app must poll `HasChanged` for changes. It's also possible for a token to never be cancelled if no changes occur or the underlying change listener is disposed or disabled.
- `HasChanged` gets a value that indicates if a change has occurred.

The interface has one method, `RegisterChangeCallback(Action<Object>, Object)`, which registers a callback that's invoked when the token has changed. `HasChanged` must be set before the callback is invoked.

ChangeToken class

`ChangeToken` is a static class used to propagate notifications that a change has occurred. `ChangeToken` resides in the `Microsoft.Extensions.Primitives` namespace. For apps that don't use the `Microsoft.AspNetCore.All` metapackage, reference the `Microsoft.Extensions.Primitives` NuGet package in the project file.

The `ChangeToken.OnChange(Func<IChangeToken>, Action)` method registers an `Action` to call whenever the token changes:

- `Func<IChangeToken>` produces the token.
- `Action` is called when the token changes.

`ChangeToken` has an `OnChange<TState>(Func<IChangeToken>, Action<TState>, TState)` overload that takes an additional `TState` parameter that's passed into the token consumer `Action`.

`OnChange` returns an `IDisposable`. Calling `Dispose` stops the token from listening for further changes and releases the token's resources.

Example uses of change tokens in ASP.NET Core

Change tokens are used in prominent areas of ASP.NET Core monitoring changes to objects:

- For monitoring changes to files, `IFileProvider`'s `Watch` method creates an `IChangeToken` for the specified files or folder to watch.
- `IChangeToken` tokens can be added to cache entries to trigger cache evictions on change.
- For `TOptions` changes, the default `OptionsMonitor` implementation of `IOptionsMonitor` has an overload that accepts one or more `IOptionsChangeTokenSource` instances. Each instance returns an `IChangeToken` to register

a change notification callback for tracking options changes.

Monitoring for configuration changes

By default, ASP.NET Core templates use [JSON configuration files](#) (*appsettings.json*, *appsettings.Development.json*, and *appsettings.Production.json*) to load app configuration settings.

These files are configured using the [AddJsonFile\(IConfigurationBuilder, String, Boolean, Boolean\)](#) extension method on [ConfigurationBuilder](#) that accepts a `reloadOnChange` parameter (ASP.NET Core 1.1 and later).

`reloadOnChange` indicates if configuration should be reloaded on file changes. See this setting in the [WebHost](#) convenience method [CreateDefaultBuilder](#) ([reference source](#)):

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);
```

File-based configuration is represented by [FileConfigurationSource](#). `FileConfigurationSource` uses [IFileProvider](#) ([reference source](#)) to monitor files.

By default, the `IFileMonitor` is provided by a [PhysicalFileProvider](#) ([reference source](#)), which uses [FileSystemWatcher](#) to monitor for configuration file changes.

The sample app demonstrates two implementations for monitoring configuration changes. If either the *appsettings.json* file changes or the Environment version of the file changes, each implementation executes custom code. The sample app writes a message to the console.

A configuration file's `FileSystemWatcher` can trigger multiple token callbacks for a single configuration file change. The sample's implementation guards against this problem by checking file hashes on the configuration files. Checking file hashes ensures that at least one of the configuration files has changed before running the custom code. The sample uses SHA1 file hashing (*Utilities/Utilities.cs*):

```

public static byte[] ComputeHash(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fs = File.OpenRead(filePath))
                {
                    return System.Security.Cryptography.SHA1.Create().ComputeHash(fs);
                }
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return new byte[20];
}

```

A retry is implemented with an exponential back-off. The re-try is present because file locking may occur that temporarily prevents computing a new hash on one of the files.

Simple startup change token

Register a token consumer `Action` callback for change notifications to the configuration reload token (*Startup.cs*):

```

ChangeToken.OnChange(
    () => config.GetReloadToken(),
    (state) => InvokeChanged(state),
    env);

```

`config.GetReloadToken()` provides the token. The callback is the `InvokeChanged` method:

```

private void InvokeChanged(IHostingEnvironment env)
{
    byte[] appsettingsHash = ComputeHash("appSettings.json");
    byte[] appsettingsEnvHash =
        ComputeHash($"appSettings.{env.EnvironmentName}.json");

    if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
        !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
    {
        _appsettingsHash = appsettingsHash;
        _appsettingsEnvHash = appsettingsEnvHash;

        WriteConsole("Configuration changed (Simple Startup Change Token)");
    }
}

```

The `state` of the callback is used to pass in the `IHostingEnvironment`. This is useful to determine the correct `appsettings` configuration JSON file to monitor, `appsettings.<Environment>.json`. File hashes are used to prevent the `WriteConsole` statement from running multiple times due to multiple token callbacks when the configuration file has only changed once.

This system runs as long as the app is running and can't be disabled by the user.

Monitoring configuration changes as a service

The sample implements:

- Basic startup token monitoring.
- Monitoring as a service.
- A mechanism to enable and disable monitoring.

The sample establishes an `IConfigurationMonitor` interface (`Extensions/ConfigurationMonitor.cs`):

```
public interface IConfigurationMonitor
{
    bool MonitoringEnabled { get; set; }
    string CurrentState { get; set; }
}
```

The constructor of the implemented class, `ConfigurationMonitor`, registers a callback for change notifications:

```
public ConfigurationMonitor(IConfiguration config, IHostingEnvironment env)
{
    _env = env;

    ChangeToken.OnChange<string>(
        () => config.GetReloadToken(),
        (state) => InvokeChanged(state),
        "Not monitoring");
}

public bool MonitoringEnabled { get; set; } = false;
public string CurrentState { get; set; } = "Not monitoring";
```

`config.GetReloadToken()` supplies the token. `InvokeChanged` is the callback method. The `state` in this instance is a string that describes the monitoring state. Two properties are used:

- `MonitoringEnabled` indicates if the callback should run its custom code.
- `CurrentState` describes the current monitoring state for use in the UI.

The `InvokeChanged` method is similar to the earlier approach, except that it:

- Doesn't run its code unless `MonitoringEnabled` is `true`.
- Sets the `CurrentState` property string to a descriptive message that records the time that the code ran.
- Notes the current `state` in its `WriteConsole` output.

```

private void InvokeChanged(string state)
{
    if (MonitoringEnabled)
    {
        byte[] appsettingsHash = ComputeHash("appSettings.json");
        byte[] appsettingsEnvHash =
            ComputeHash($"appSettings.{{_env.EnvironmentName}}.json");

        if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
            !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
        {
            state = $"State updated at {DateTime.Now}";
            CurrentState = state;

            _appsettingsHash = appsettingsHash;
            _appsettingsEnvHash = appsettingsEnvHash;

            WriteConsole($"Configuration changed (ConfigurationMonitor Class) {state}");
        }
    }
}

```

An instance `ConfigurationMonitor` is registered as a service in `ConfigureServices` of `Startup.cs`:

```

services.AddSingleton<IConfigurationMonitor, ConfigurationMonitor>();

```

The Index page offers the user control over configuration monitoring. The instance of `IConfigurationMonitor` is injected into the `IndexModel` :

```

public IndexModel(
    IConfiguration config,
    IConfigurationMonitor monitor,
    FileService fileService)
{
    _config = config;
    _monitor = monitor;
    _fileService = fileService;
}

```

A button enables and disables monitoring:

```

<button class="btn btn-danger" asp-page-handler="StopMonitoring">Stop Monitoring</button>

```

```

public IActionResult OnPostStartMonitoring()
{
    _monitor.MonitoringEnabled = true;
    _monitor.CurrentState = string.Empty;

    return RedirectToPage();
}

public IActionResult OnPostStopMonitoring()
{
    _monitor.MonitoringEnabled = false;
    _monitor.CurrentState = "Not monitoring";

    return RedirectToPage();
}

```

When `OnPostStartMonitoring` is triggered, monitoring is enabled, and the current state is cleared. When `OnPostStopMonitoring` is triggered, monitoring is disabled, and the state is set to reflect that monitoring is not occurring.

Monitoring cached file changes

File content can be cached in-memory using `IMemoryCache`. In-memory caching is described in the [In-memory caching](#) topic. Without taking additional steps, such as the implementation described below, *stale* (outdated) data is returned from a cache if the source data changes.

Not taking into account the status of a cached source file when renewing a [sliding expiration](#) period leads to stale cache data. Each request for the data renews the sliding expiration period, but the file is never reloaded into the cache. Any app features that use the file's cached content are subject to possibly receiving stale content.

Using change tokens in a file caching scenario prevents stale file content in the cache. The sample app demonstrates an implementation of the approach.

The sample uses `GetFileContent` to:

- Return file content.
- Implement a retry algorithm with exponential back-off to cover cases where a file lock is temporarily preventing a file from being read.

Utilities/Utilities.cs:

```
public async static Task<string> GetFileContent(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fileStreamReader = File.OpenText(filePath))
                {
                    return await fileStreamReader.ReadToEndAsync();
                }
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                await Task.Delay(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return null;
}
```

A `FileService` is created to handle cached file lookups. The `GetFileContent` method call of the service attempts to obtain file content from the in-memory cache and return it to the caller (*Services/FileService.cs*).

If cached content isn't found using the cache key, the following actions are taken:

1. The file content is obtained using `GetFileContent`.
2. A change token is obtained from the file provider with `IFileProviders.Watch`. The token's callback is triggered when the file is modified.
3. The file content is cached with a [sliding expiration](#) period. The change token is attached with `MemoryCacheEntryExtensions.AddExpirationToken` to evict the cache entry if the file changes while it's cached.

```
public class FileService
{
    private readonly IMemoryCache _cache;
    private readonly IFileProvider _fileProvider;
    private List<string> _tokens = new List<string>();

    public FileService(IMemoryCache cache, IHostingEnvironment env)
    {
        _cache = cache;
        _fileProvider = env.ContentRootFileProvider;
    }

    public async Task<string> GetFileContents(string fileName)
    {
        // For the purposes of this example, files are stored
        // in the content root of the app. To obtain the physical
        // path to a file at the content root, use the
        // ContentRootFileProvider on IHostingEnvironment.
        var filePath = _fileProvider.GetFileInfo(fileName).PhysicalPath;
        string fileContent;

        // Try to obtain the file contents from the cache.
        if (_cache.TryGetValue(filePath, out fileContent))
        {
            return fileContent;
        }

        // The cache doesn't have the entry, so obtain the file
        // contents from the file itself.
        fileContent = await GetFileContent(filePath);

        if (fileContent != null)
        {
            // Obtain a change token from the file provider whose
            // callback is triggered when the file is modified.
            var changeToken = _fileProvider.Watch(fileName);

            // Configure the cache entry options for a five minute
            // sliding expiration and use the change token to
            // expire the file in the cache if the file is
            // modified.
            var cacheEntryOptions = new MemoryCacheEntryOptions()
                .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                .AddExpirationToken(changeToken);

            // Put the file contents into the cache.
            _cache.Set(filePath, fileContent, cacheEntryOptions);

            return fileContent;
        }

        return string.Empty;
    }
}
```

The `FileService` is registered in the service container along with the memory caching service (*Startup.cs*):

```
services.AddMemoryCache();
services.AddSingleton<FileService>();
```

The page model loads the file's content using the service (*Pages/Index.cshtml.cs*):

```
var fileContent = await _fileService.GetFileContents("poem.txt");
```

CompositeChangeToken class

For representing one or more `IChangeToken` instances in a single object, use the `CompositeChangeToken` class ([reference source](#)).

```
var firstCancellationTokenSource = new CancellationTokenSource();
var secondCancellationTokenSource = new CancellationTokenSource();

var firstCancellationToken = firstCancellationTokenSource.Token;
var secondCancellationToken = secondCancellationTokenSource.Token;

var firstCancellationChangeToken = new CancellationChangeToken(firstCancellationToken);
var secondCancellationChangeToken = new CancellationChangeToken(secondCancellationToken);

var compositeChangeToken =
    new CompositeChangeToken(
        new List<IChangeToken>
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken
        }
    );
```

`HasChanged` on the composite token reports `true` if any represented token `HasChanged` is `true`.

`ActiveChangeCallbacks` on the composite token reports `true` if any represented token `ActiveChangeCallbacks` is `true`. If multiple concurrent change events occur, the composite change callback is invoked exactly one time.

See also

- [In-memory caching](#)
- [Working with a distributed cache](#)
- [Detect changes with change tokens](#)
- [Response caching](#)
- [Response Caching Middleware](#)
- [Cache Tag Helper](#)
- [Distributed Cache Tag Helper](#)

Introduction to Open Web Interface for .NET (OWIN)

10/13/2017 • 5 min to read • [Edit Online](#)

By [Steve Smith](#) and [Rick Anderson](#)

ASP.NET Core supports the Open Web Interface for .NET (OWIN). OWIN allows web apps to be decoupled from web servers. It defines a standard way for middleware to be used in a pipeline to handle requests and associated responses. ASP.NET Core applications and middleware can interoperate with OWIN-based applications, servers, and middleware.

OWIN provides a decoupling layer that allows two frameworks with disparate object models to be used together. The `Microsoft.AspNetCore.Owin` package provides two adapter implementations:

- ASP.NET Core to OWIN
- OWIN to ASP.NET Core

This allows ASP.NET Core to be hosted on top of an OWIN compatible server/host, or for other OWIN compatible components to be run on top of ASP.NET Core.

Note: Using these adapters comes with a performance cost. Applications using only ASP.NET Core components should not use the Owin package or adapters.

[View or download sample code \(how to download\)](#)

Running OWIN middleware in the ASP.NET pipeline

ASP.NET Core's OWIN support is deployed as part of the `Microsoft.AspNetCore.Owin` package. You can import OWIN support into your project by installing this package.

OWIN middleware conforms to the [OWIN specification](#), which requires a `Func<IDictionary<string, object>, Task>` interface, and specific keys be set (such as `owin.ResponseBody`). The following simple OWIN middleware displays "Hello World":

```
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);

    // OWIN Environment Keys: http://owin.org/spec/spec/owin-1.0.0.html
    var responseStream = (Stream)environment["owin.ResponseBody"];
    var responseHeaders = (IDictionary<string, string[]>)environment["owin.ResponseHeaders"];

    responseHeaders["Content-Length"] = new string[] {
responseBytes.Length.ToString(CultureInfo.InvariantCulture) };
    responseHeaders["Content-Type"] = new string[] { "text/plain" };

    return responseStream.WriteAsync(responseBytes, 0, responseBytes.Length);
}
```

The sample signature returns a `Task` and accepts an `IDictionary<string, object>` as required by OWIN.

The following code shows how to add the `OwinHello` middleware (shown above) to the ASP.NET pipeline with the `UseOwin` extension method.

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

You can configure other actions to take place within the OWIN pipeline.

NOTE

Response headers should only be modified prior to the first write to the response stream.

NOTE

Multiple calls to `UseOwin` is discouraged for performance reasons. OWIN components will operate best if grouped together.

```
app.UseOwin(pipeline =>
{
    pipeline(next =>
    {
        // do something before
        return OwinHello;
        // do something after
    });
});
```

Using ASP.NET Hosting on an OWIN-based server

OWIN-based servers can host ASP.NET applications. One such server is [Nowin](#), a .NET OWIN web server. In the sample for this article, I've included a project that references Nowin and uses it to create an `IServer` capable of self-hosting ASP.NET Core.

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

namespace NowinSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseNowin()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}

```

`IServer` is an interface that requires an `Features` property and a `Start` method.

`Start` is responsible for configuring and starting the server, which in this case is done through a series of fluent API calls that set addresses parsed from the `IServerAddressesFeature`. Note that the fluent configuration of the `_builder` variable specifies that requests will be handled by the `appFunc` defined earlier in the method. This `Func` is called on each request to process incoming requests.

We'll also add an `IWebHostBuilder` extension to make it easy to add and configure the Nowin server.

```

using System;
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.Extensions.DependencyInjection;
using Nowin;
using NowinSample;

namespace Microsoft.AspNetCore.Hosting
{
    public static class NowinWebHostBuilderExtensions
    {
        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder)
        {
            return builder.ConfigureServices(services =>
            {
                services.AddSingleton<IServer, NowinServer>();
            });
        }

        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder, Action<ServerBuilder> configure)
        {
            builder.ConfigureServices(services =>
            {
                services.Configure(configure);
            });
            return builder.UseNowin();
        }
    }
}

```

With this in place, all that's required to run an ASP.NET application using this custom server to call the extension in *Program.cs*:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

namespace NowinSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseNowin()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

Learn more about ASP.NET [Servers](#).

Run ASP.NET Core on an OWIN-based server and use its WebSockets support

Another example of how OWIN-based servers' features can be leveraged by ASP.NET Core is access to features like WebSockets. The .NET OWIN web server used in the previous example has support for Web Sockets built in, which can be leveraged by an ASP.NET Core application. The example below shows a simple web app that supports Web Sockets and echoes back everything sent to the server through WebSockets.

```

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            if (context.WebSockets.IsWebSocketRequest)
            {
                WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
                await EchoWebSocket(webSocket);
            }
            else
            {
                await next();
            }
        });

        app.Run(context =>
        {
            return context.Response.WriteAsync("Hello World");
        });
    }

    private async Task EchoWebSocket(WebSocket webSocket)
    {
        byte[] buffer = new byte[1024];
        WebSocketReceiveResult received = await webSocket.ReceiveAsync(
            new ArraySegment<byte>(buffer), CancellationToken.None);

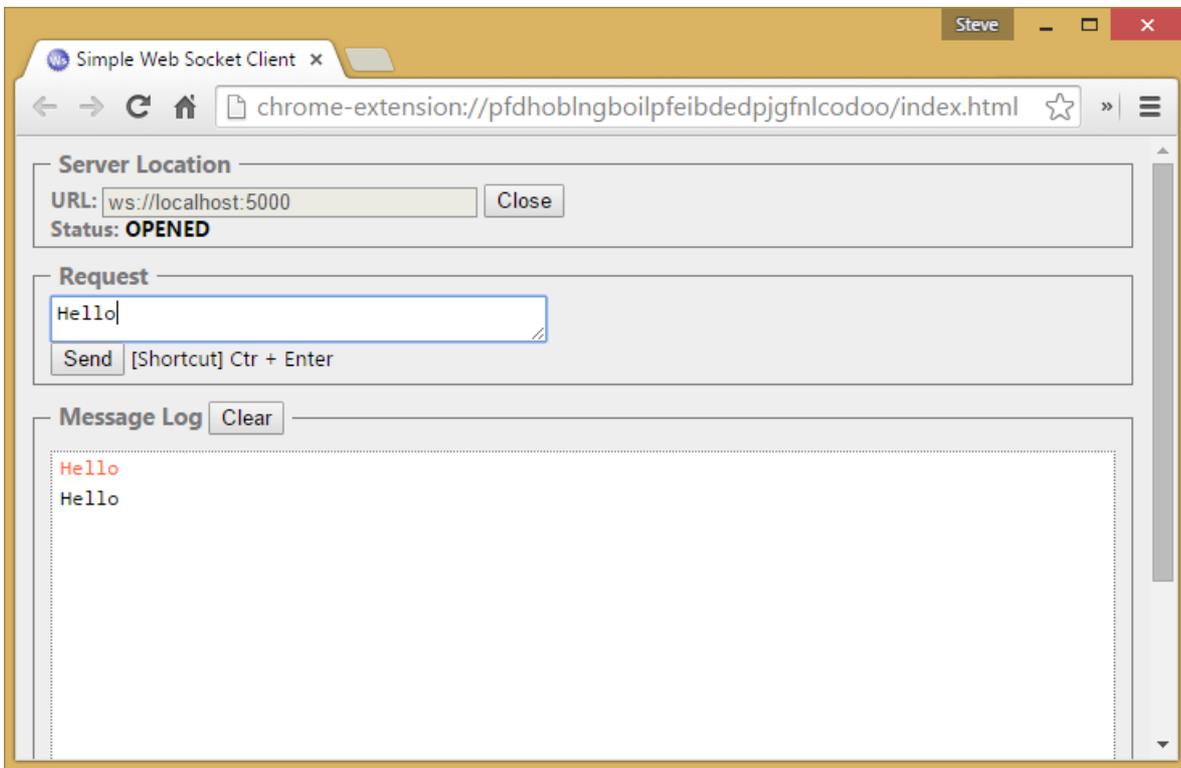
        while (!webSocket.CloseStatus.HasValue)
        {
            // Echo anything we receive
            await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, received.Count),
                received.MessageType, received.EndOfMessage, CancellationToken.None);

            received = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
                CancellationToken.None);
        }

        await webSocket.CloseAsync(webSocket.CloseStatus.Value,
            webSocket.CloseStatusDescription, CancellationToken.None);
    }
}

```

This [sample](#) is configured using the same `NowInServer` as the previous one - the only difference is in how the application is configured in its `Configure` method. A test using [a simple websocket client](#) demonstrates the application:



OWIN environment

You can construct a OWIN environment using the `HttpContext`.

```
var environment = new OwinEnvironment(HttpContext);
var features = new OwinFeatureCollection(environment);
```

OWIN keys

OWIN depends on an `IDictionary<string,object>` object to communicate information throughout an HTTP Request/Response exchange. ASP.NET Core implements the keys listed below. See the [primary specification](#), [extensions](#), and [OWIN Key Guidelines and Common Keys](#).

Request Data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestScheme	<code>String</code>	
owin.RequestMethod	<code>String</code>	
owin.RequestPathBase	<code>String</code>	
owin.RequestPath	<code>String</code>	
owin.RequestQueryString	<code>String</code>	
owin.RequestProtocol	<code>String</code>	
owin.RequestHeaders	<code>IDictionary<string,string[]></code>	

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestBody	Stream	

Request Data (OWIN v1.1.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestId	String	Optional

Response Data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.ResponseStatusCode	int	Optional
owin.ResponseReasonPhrase	String	Optional
owin.ResponseHeaders	IDictionary<string, string[]>	
owin.ResponseBody	Stream	

Other Data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.CallCancelled	CancellationToken	
owin.Version	String	

Common Keys

KEY	VALUE (TYPE)	DESCRIPTION
ssl.ClientCertificate	X509Certificate	
ssl.LoadClientCertAsync	Func<Task>	
server.RemoteIpAddress	String	
server.RemotePort	String	
server.LocalIpAddress	String	
server.LocalPort	String	
server.IsLocal	bool	
server.OnSendingHeaders	Action<Action<object>, object>	

SendFiles v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
sendfile.SendAsync	See delegate signature	Per Request

Opaque v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
opaque.Version	String	
opaque.Upgrade	OpaqueUpgrade	See delegate signature
opaque.Stream	Stream	
opaque.CallCancelled	CancellationToken	

WebSocket v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
websocket.Version	String	
websocket.Accept	WebSocketAccept	See delegate signature
websocket.AcceptAlt		Non-spec
websocket.SubProtocol	String	See RFC6455 Section 4.2.2 Step 5.5
websocket.SendAsync	WebSocketSendAsync	See delegate signature
websocket.ReceiveAsync	WebSocketReceiveAsync	See delegate signature
websocket.CloseAsync	WebSocketCloseAsync	See delegate signature
websocket.CallCancelled	CancellationToken	
websocket.ClientCloseStatus	int	Optional
websocket.ClientCloseDescription	String	Optional

Additional Resources

- [Middleware](#)
- [Servers](#)

Introduction to WebSockets in ASP.NET Core

10/2/2017 • 3 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Andrew Stanton-Nurse](#)

This article explains how to get started with WebSockets in ASP.NET Core. [WebSocket](#) is a protocol that enables two-way persistent communication channels over TCP connections. It is used for applications such as chat, stock tickers, games, anywhere you want real-time functionality in a web application.

[View or download sample code \(how to download\)](#). See the [Next Steps](#) section for more information.

Prerequisites

- ASP.NET Core 1.1 (does not run on 1.0)
- Any OS that ASP.NET Core runs on:
 - Windows 7 / Windows Server 2008 and later
 - Linux
 - macOS
- **Exception:** If your app runs on Windows with IIS, or with WebListener, you must use:
 - Windows 8 / Windows Server 2012 or later
 - IIS 8 / IIS 8 Express
 - WebSocket must be enabled in IIS
- For supported browsers, see <http://caniuse.com/#feat=websockets>.

When to use it

Use WebSockets when you need to work directly with a socket connection. For example, you might need the best possible performance for a real-time game.

[ASP.NET SignalR](#) provides a richer application model for real-time functionality, but it runs only on ASP.NET, not ASP.NET Core. A Core version of SignalR is under development; to follow its progress, see the [GitHub repository for SignalR Core](#).

If you don't want to wait for SignalR Core, you can use WebSockets directly now. But you might have to develop features that SignalR would provide, such as:

- Support for a broader range of browser versions by using automatic fallback to alternative transport methods.
- Automatic reconnection when a connection drops.
- Support for clients calling methods on the server or vice versa.
- Support for scaling to multiple servers.

How to use it

- Install the [Microsoft.AspNetCore.WebSockets](#) package.
- Configure the middleware.
- Accept WebSocket requests.
- Send and receive messages.

Configure the middleware

Add the WebSockets middleware in the `Configure` method of the `Startup` class.

```
app.UseWebSockets();
```

The following settings can be configured:

- `KeepAliveInterval` - How frequently to send "ping" frames to the client, to ensure proxies keep the connection open.
- `ReceiveBufferSize` - The size of the buffer used to receive data. Only advanced users would need to change this, for performance tuning based on the size of their data.

```
var websocketOptions = new WebSocketOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(120),
    ReceiveBufferSize = 4 * 1024
};
app.UseWebSockets(websocketOptions);
```

Accept WebSocket requests

Somewhere later in the request life cycle (later in the `Configure` method or in an MVC action, for example) check if it's a WebSocket request and accept the WebSocket request.

This example is from later in the `Configure` method.

```
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            WebSocket websocket = await context.WebSockets.AcceptWebSocketAsync();
            await Echo(context, websocket);
        }
        else
        {
            context.Response.StatusCode = 400;
        }
    }
    else
    {
        await next();
    }
});
```

A WebSocket request could come in on any URL, but this sample code only accepts requests for `/ws`.

Send and receive messages

The `AcceptWebSocketAsync` method upgrades the TCP connection to a WebSocket connection and gives you a [WebSocket](#) object. Use the `WebSocket` object to send and receive messages.

The code shown earlier that accepts the WebSocket request passes the `WebSocket` object to an `Echo` method; here's the `Echo` method. The code receives a message and immediately sends back the same message. It stays in a loop doing that until the client closes the connection.

```

private async Task Echo(HttpContext context, WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
CancellationToken.None);
    while (!result.CloseStatus.HasValue)
    {
        await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, result.Count), result.MessageType,
result.EndOfMessage, CancellationToken.None);

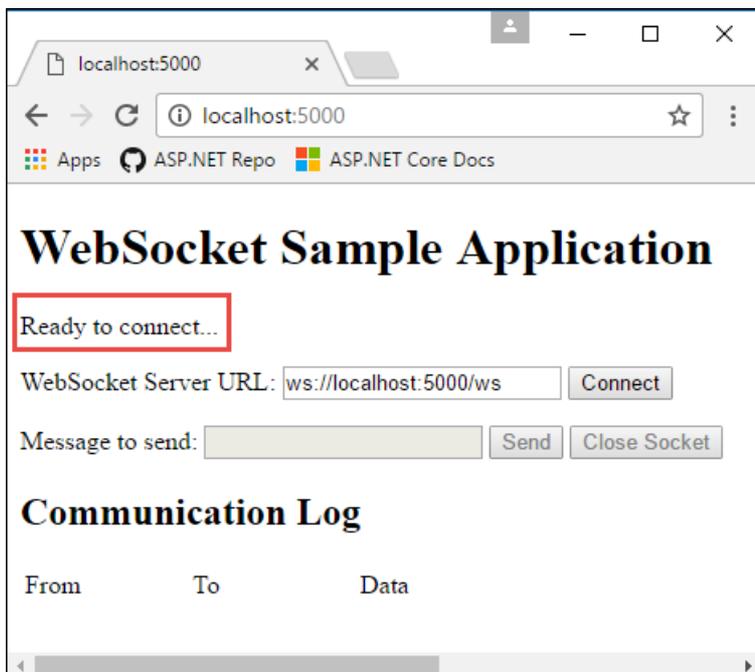
        result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);
    }
    await webSocket.CloseAsync(result.CloseStatus.Value, result.CloseStatusDescription,
CancellationToken.None);
}

```

When you accept the WebSocket before beginning this loop, the middleware pipeline ends. Upon closing the socket, the pipeline unwinds. That is, the request stops moving forward in the pipeline when you accept a WebSocket, just as it would when you hit an MVC action, for example. But when you finish this loop and close the socket, the request proceeds back up the pipeline.

Next steps

The [sample application](#) that accompanies this article is a simple echo application. It has a web page that makes WebSocket connections, and the server just resends back to the client any messages it receives. Run it from a command prompt (it's not set up to run from Visual Studio with IIS Express) and navigate to <http://localhost:5000>. The web page shows connection status at the upper left:



Select **Connect** to send a WebSocket request to the URL shown. Enter a test message and select **Send**. When done, select **Close Socket**. The **Communication Log** section reports each open, send, and close action as it happens.

localhost:5000

localhost:5000

Apps ASP.NET Repo ASP.NET Core Docs

WebSocket Sample Application

Closed

WebSocket Server URL:

Message to send:

Communication Log

From	To	Data
Connection opened		
Client	Server	first test message
Server	Client	first test message
Client	Server	second test message
Server	Client	second test message
Connection closed. Code: 1000. Reason: Closing from client		

Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.x

10/3/2017 • 1 min to read • [Edit Online](#)

This feature requires ASP.NET Core 2.x targeting .NET Core 2.x.

The `Microsoft.AspNetCore.All` metapackage for ASP.NET Core includes:

- All supported packages by the ASP.NET Core team.
- All supported packages by the Entity Framework Core.
- Internal and 3rd-party dependencies used by ASP.NET Core and Entity Framework Core.

All the features of ASP.NET Core 2.x and Entity Framework Core 2.x are included in the `Microsoft.AspNetCore.All` package. The default project templates use this package.

The version number of the `Microsoft.AspNetCore.All` metapackage represents the ASP.NET Core version and Entity Framework Core version (aligned with the .NET Core version).

Applications that use the `Microsoft.AspNetCore.All` metapackage automatically take advantage of the [.NET Core Runtime Store](#). The Runtime Store contains all the runtime assets needed to run ASP.NET Core 2.x applications. When you use the `Microsoft.AspNetCore.All` metapackage, **no** assets from the referenced ASP.NET Core NuGet packages are deployed with the application — the .NET Core Runtime Store contains these assets. The assets in the Runtime Store are precompiled to improve application startup time.

You can use the package trimming process to remove packages that you don't use. Trimmed packages are excluded in published application output.

The following `.csproj` file references the `Microsoft.AspNetCore.All` metapackage for ASP.NET Core:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

</Project>
```

Choose between ASP.NET and ASP.NET Core

10/2/2017 • 1 min to read • [Edit Online](#)

No matter the web application you are creating, ASP.NET has a solution for you: from enterprise web applications targeting Windows Server, to small microservices targeting Linux containers, and everything in between.

ASP.NET Core

ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based web applications on Windows, macOS, or Linux.

ASP.NET

ASP.NET is a mature framework that provides all the services needed to build enterprise-class, server-based web applications on Windows.

Which one is right for me?

ASP.NET CORE	ASP.NET
Build for Windows, macOS, or Linux	Build for Windows
Razor Pages is the recommended approach to create a Web UI with ASP.NET Core 2.0. See also MVC and Web API	Use Web Forms , SignalR , MVC , Web API , or Web Pages
Multiple versions per machine	One version per machine
Develop with Visual Studio, Visual Studio for Mac , or Visual Studio Code using C# or F#	Develop with Visual Studio using C#, VB, or F#
Higher performance than ASP.NET	Good performance
Choose .NET Framework or .NET Core runtime	Use .NET Framework runtime

ASP.NET Core scenarios

- [Razor Pages](#) is the recommended approach to create a Web UI with ASP.NET Core 2.0.
- [Websites](#)
- [APIs](#)

ASP.NET scenarios

- [Websites](#)
- [APIs](#)
- [Real-time](#)

Resources

- [Introduction to ASP.NET](#)

- [Introduction to ASP.NET Core](#)

Overview of ASP.NET Core MVC

1/9/2018 • 9 min to read • [Edit Online](#)

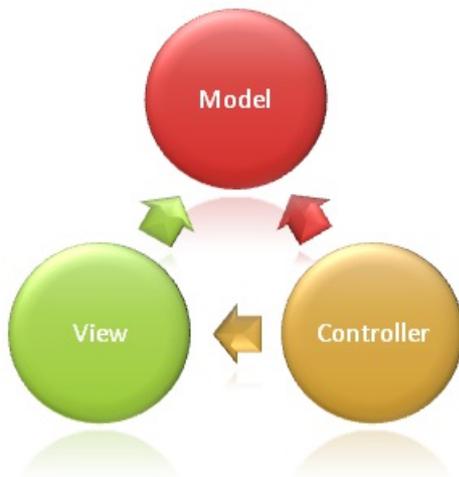
By [Steve Smith](#)

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

What is the MVC pattern?

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve [separation of concerns](#). Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job (and follows the [Single Responsibility Principle](#)). It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, you have to modify an object containing business logic every time you change the user interface. This is likely to introduce errors and require the retesting of all business logic after every minimal user interface change.

NOTE

Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views typically use ViewModel types designed to

contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

NOTE

There are many ways to organize the model in an app that uses the MVC architectural pattern. Learn more about some [different kinds of model types](#).

View Responsibilities

Views are responsible for presenting content through the user interface. They use the [Razor view engine](#) to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a [View Component](#), ViewModel, or view template to simplify the view.

Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

NOTE

Controllers should not be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, use the [Single Responsibility Principle](#) to push business logic out of the controller and into the domain model.

TIP

If you find that your controller actions frequently perform the same kinds of actions, you can follow the [Don't Repeat Yourself principle](#) by moving these common actions into [filters](#).

What is ASP.NET Core MVC

The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.

ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns. It gives you full control over markup, supports TDD-friendly development and uses the latest web standards.

Features

ASP.NET Core MVC includes the following:

- [Routing](#)
- [Model binding](#)
- [Model validation](#)
- [Dependency injection](#)
- [Filters](#)
- [Areas](#)
- [Web APIs](#)
- [Testability](#)

- [Razor view engine](#)
- [Strongly typed views](#)
- [Tag Helpers](#)
- [View Components](#)

Routing

ASP.NET Core MVC is built on top of [ASP.NET Core's routing](#), a powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs. This enables you to define your application's URL naming patterns that work well for search engine optimization (SEO) and for link generation, without regard for how the files on your web server are organized. You can define your routes using a convenient route template syntax that supports route value constraints, defaults and optional values.

Convention-based routing enables you to globally define the URL formats that your application accepts and how each of those formats maps to a specific action method on given controller. When an incoming request is received, the routing engine parses the URL and matches it to one of the defined URL formats, and then calls the associated controller's action method.

```
routes.MapRoute(name: "Default", template: "{controller=Home}/{action=Index}/{id?}");
```

Attribute routing enables you to specify routing information by decorating your controllers and actions with attributes that define your application's routes. This means that your route definitions are placed next to the controller and action with which they're associated.

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        ...
    }
}
```

Model binding

ASP.NET Core MVC [model binding](#) converts client request data (form values, route data, query string parameters, HTTP headers) into objects that the controller can handle. As a result, your controller logic doesn't have to do the work of figuring out the incoming request data; it simply has the data as parameters to its action methods.

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null) { ... }
```

Model validation

ASP.NET Core MVC supports [validation](#) by decorating your model object with data annotation validation attributes. The validation attributes are checked on the client side before values are posted to the server, as well as on the server before the controller action is called.

```

using System.ComponentModel.DataAnnotations;
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}

```

A controller action:

```

public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        // work with the model
    }
    // At this point, something failed, redisplay form
    return View(model);
}

```

The framework handles validating request data both on the client and on the server. Validation logic specified on model types is added to the rendered views as unobtrusive annotations and is enforced in the browser with [jQuery Validation](#).

Dependency injection

ASP.NET Core has built-in support for [dependency injection \(DI\)](#). In ASP.NET Core MVC, [controllers](#) can request needed services through their constructors, allowing them to follow the [Explicit Dependencies Principle](#).

Your app can also use [dependency injection in view files](#), using the `@inject` directive:

```

@inject SomeService ServiceName
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ServiceName.GetTitle</title>
</head>
<body>
    <h1>@ServiceName.GetTitle</h1>
</body>
</html>

```

Filters

[Filters](#) help developers encapsulate cross-cutting concerns, like exception handling or authorization. Filters enable running custom pre- and post-processing logic for action methods, and can be configured to run at certain points within the execution pipeline for a given request. Filters can be applied to controllers or actions as attributes (or can be run globally). Several filters (such as `Authorize`) are included in the framework.

```

[Authorize]
public class AccountController : Controller
{

```

Areas

[Areas](#) provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is an MVC structure inside an application. In an MVC project, logical components like Model, Controller, and View are kept in different folders, and MVC uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search etc. Each of these units have their own logical component views, controllers, and models.

Web APIs

In addition to being a great platform for building web sites, ASP.NET Core MVC has great support for building Web APIs. You can build services that reach a broad range of clients including browsers and mobile devices.

The framework includes support for HTTP content-negotiation with built-in support for [formatting data](#) as JSON or XML. Write [custom formatters](#) to add support for your own formats.

Use link generation to enable support for hypermedia. Easily enable support for [cross-origin resource sharing \(CORS\)](#) so that your Web APIs can be shared across multiple Web applications.

Testability

The framework's use of interfaces and dependency injection make it well-suited to unit testing, and the framework includes features (like a TestHost and InMemory provider for Entity Framework) that make [integration testing](#) quick and easy as well. Learn more about [testing controller logic](#).

Razor view engine

[ASP.NET Core MVC views](#) use the [Razor view engine](#) to render views. Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code. Razor is used to dynamically generate web content on the server. You can cleanly mix server code with client side content and code.

```
<ul>
  @for (int i = 0; i < 5; i++) {
    <li>List item @i</li>
  }
</ul>
```

Using the Razor view engine you can define [layouts](#), [partial views](#) and replaceable sections.

Strongly typed views

Razor views in MVC can be strongly typed based on your model. Controllers can pass a strongly typed model to views enabling your views to have type checking and IntelliSense support.

For example, the following view renders a model of type `IEnumerable<Product>`:

```
@model IEnumerable<Product>
<ul>
  @foreach (Product p in Model)
  {
    <li>@p.Name</li>
  }
</ul>
```

Tag Helpers

[Tag Helpers](#) enable server side code to participate in creating and rendering HTML elements in Razor files. You can use tag helpers to define custom tags (for example, `<environment>`) or to modify the behavior of existing tags (for example, `<label>`). Tag Helpers bind to specific elements based on the element name and its attributes. They provide the benefits of server-side rendering while still preserving an HTML editing experience.

There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LinkTagHelper` can be used to create a link to the `Login` action of the `AccountsController`:

```
<p>
  Thank you for confirming your email.
  Please <a asp-controller="Account" asp-action="Login">Click here to Log in</a>.
</p>
```

The `EnvironmentTagHelper` can be used to include different scripts in your views (for example, raw or minified) based on the runtime environment, such as Development, Staging, or Production:

```
<environment names="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery">
  </script>
</environment>
```

Tag Helpers provide an HTML-friendly development experience and a rich IntelliSense environment for creating HTML and Razor markup. Most of the built-in Tag Helpers target existing HTML elements and provide server-side attributes for the element.

View Components

[View Components](#) allow you to package rendering logic and reuse it throughout the application. They're similar to [partial views](#), but with associated logic.

Introduction to Razor Pages in ASP.NET Core

12/19/2017 • 16 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Ryan Nowak](#)

Razor Pages is a new feature of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

If you're looking for a tutorial that uses the Model-View-Controller approach, see [Getting started with ASP.NET Core MVC](#).

This document provides an introduction to Razor Pages. It's not a step by step tutorial. If you find some of the sections difficult to follow, see [Getting started with Razor Pages](#).

ASP.NET Core 2.0 prerequisites

Install [.NET Core 2.0.0](#) or later.

If you're using Visual Studio, install [Visual Studio 2017](#) version 15.3 or later with the following workloads:

- **ASP.NET and web development**
- **.NET Core cross-platform development**

Creating a Razor Pages project

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)

See [Getting started with Razor Pages](#) for detailed instructions on how to create a Razor Pages project using Visual Studio.

Razor Pages

Razor Pages is enabled in *Startup.cs*:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Includes support for Razor Pages and controllers.
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
    }
}
```

Consider a basic page:

```
@page
```

```
<h1>Hello, world!</h1>  
<h2>The time on the server is @DateTime.Now</h2>
```

The preceding code looks a lot like a Razor view file. What makes it different is the `@page` directive. `@page` makes the file into an MVC action - which means that it handles requests directly, without going through a controller. `@page` must be the first Razor directive on a page. `@page` affects the behavior of other Razor constructs.

A similar page, using a `PageModel` class, is shown in the following two files. The `Pages/Index2.cshtml` file:

```
@page  
@using RazorPagesIntro.Pages  
@model IndexModel2  
  
<h2>Separate page model</h2>  
<p>  
    @Model.Message  
</p>
```

The `Pages/Index2.cshtml.cs` "code-behind" file:

```
using Microsoft.AspNetCore.Mvc.RazorPages;  
using System;  
  
namespace RazorPagesIntro.Pages  
{  
    public class IndexModel2 : PageModel  
    {  
        public string Message { get; private set; } = "PageModel in C#";  
  
        public void OnGet()  
        {  
            Message += $" Server time is { DateTime.Now }";  
        }  
    }  
}
```

By convention, the `PageModel` class file has the same name as the Razor Page file with `.cs` appended. For example, the previous Razor Page is `Pages/Index2.cshtml`. The file containing the `PageModel` class is named `Pages/Index2.cshtml.cs`.

The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Page path and the matching URL:

FILE NAME AND PATH	MATCHING URL
<code>/Pages/Index.cshtml</code>	<code>/</code> or <code>/Index</code>
<code>/Pages/Contact.cshtml</code>	<code>/Contact</code>
<code>/Pages/Store/Contact.cshtml</code>	<code>/Store/Contact</code>
<code>/Pages/Store/Index.cshtml</code>	<code>/Store</code> or <code>/Store/Index</code>

Notes:

- The runtime looks for Razor Pages files in the *Pages* folder by default.
- `Index` is the default page when a URL doesn't include a page.

Writing a basic form

Razor Pages features are designed to make common patterns used with web browsers easy. [Model binding](#), [Tag Helpers](#), and HTML helpers all *just work* with the properties defined in a Razor Page class. Consider a page that implements a basic "contact us" form for the `Contact` model:

For the samples in this document, the `DbContext` is initialized in the `Startup.cs` file.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesContacts.Data;

namespace RazorPagesContacts
{
    public class Startup
    {
        public IHostingEnvironment HostingEnvironment { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(options =>
                options.UseInMemoryDatabase("name"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

The data model:

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Data
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
        public string Name { get; set; }
    }
}
```

The db context:

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions options)
            : base(options)
        {
        }

        public DbSet<Customer> Customers { get; set; }
    }
}
```

The *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

The *Pages/Create.cshtml.cs* code-behind file for the view:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class CreateModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }
    }
}

```

By convention, the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page.

The `PageModel` class allows separation of the logic of a page from its presentation. It defines page handlers for requests sent to the page and the data used to render the page. This separation allows you to manage page dependencies through [dependency injection](#) and to [unit test](#) the pages.

The page has an `OnPostAsync` *handler method*, which runs on `POST` requests (when a user posts the form). You can add handler methods for any HTTP verb. The most common handlers are:

- `OnGet` to initialize state needed for the page. [OnGet](#) sample.
- `OnPost` to handle form submissions.

The `Async` naming suffix is optional but is often used by convention for asynchronous functions. The `OnPostAsync` code in the preceding example looks similar to what you would normally write in a controller. The preceding code is typical for Razor Pages. Most of the MVC primitives like [model binding](#), [validation](#), and action results are shared.

The previous `OnPostAsync` method:

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}

```

The basic flow of `OnPostAsync`:

Check for validation errors.

- If there are no errors, save the data and redirect.
- If there are errors, show the page again with validation messages. Client-side validation is identical to traditional ASP.NET Core MVC applications. In many cases, validation errors would be detected on the client, and never submitted to the server.

When the data is entered successfully, the `OnPostAsync` handler method calls the `RedirectToPage` helper method to return an instance of `RedirectToPageResult`. `RedirectToPage` is a new action result, similar to `RedirectToAction` or `RedirectToRoute`, but customized for pages. In the preceding sample, it redirects to the root Index page (`/Index`). `RedirectToPage` is detailed in the [URL generation for Pages](#) section.

When the submitted form has validation errors (that are passed to the server), the `OnPostAsync` handler method calls the `Page` helper method. `Page` returns an instance of `PageResult`. Returning `Page` is similar to how actions in controllers return `View`. `PageResult` is the default return type for a handler method. A handler method that returns `void` renders the page.

The `Customer` property uses `[BindProperty]` attribute to opt in to model binding.

```

public class CreateModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateModel(AppDbContext db)
    {
        _db = db;
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        return RedirectToPage("/Index");
    }
}

```

Razor Pages, by default, bind properties only with non-GET verbs. Binding to properties can reduce the amount of code you have to write. Binding reduces code by using the same property to render form fields (

`<input asp-for="Customer.Name" />`) and accept the input.

The home page (*Index.cshtml*):

```
@page
@model RazorPagesContacts.Pages.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts</h1>
<form method="post">
  <table class="table">
    <thead>
      <tr>
        <th>ID</th>
        <th>Name</th>
      </tr>
    </thead>
    <tbody>
      @foreach (var contact in Model.Customers)
      {
        <tr>
          <td>@contact.Id</td>
          <td>@contact.Name</td>
          <td>
            <a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
            <button type="submit" asp-page-handler="delete"
              asp-route-id="@contact.Id">delete</button>
          </td>
        </tr>
      }
    </tbody>
  </table>

  <a asp-page="./Create">Create</a>
</form>
```

The code behind *Index.cshtml.cs* file:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Pages
{
    public class IndexModel : PageModel
    {
        private readonly AppDbContext _db;

        public IndexModel(AppDbContext db)
        {
            _db = db;
        }

        public IList<Customer> Customers { get; private set; }

        public async Task OnGetAsync()
        {
            Customers = await _db.Customers.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostDeleteAsync(int id)
        {
            var contact = await _db.Customers.FindAsync(id);

            if (contact != null)
            {
                _db.Customers.Remove(contact);
                await _db.SaveChangesAsync();
            }

            return RedirectToPage();
        }
    }
}

```

The *Index.cshtml* file contains the following markup to create an edit link for each contact:

```
<a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
```

The [Anchor Tag Helper](#) used the `asp-route-{value}` attribute to generate a link to the Edit page. The link contains route data with the contact ID. For example, `http://localhost:5000/Edit/1`.

The *Pages/Edit.cshtml* file:

```

@page "{id:int}"
@model RazorPagesContacts.Pages.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@{
    ViewData["Title"] = "Edit Customer";
}

<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name" ></span>
        </div>
    </div>
    <div>
        <button type="submit">Save</button>
    </div>
</form>

```

The first line contains the `@page "{id:int}"` directive. The routing constraint `"{id:int}"` tells the page to accept requests to the page that contain `int` route data. If a request to the page doesn't contain route data that can be converted to an `int`, the runtime returns an HTTP 404 (not found) error.

The `Pages/Edit.cshtml.cs` file:

```

using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Customer = await _db.Customers.FindAsync(id);

            if (Customer == null)
            {
                return RedirectToPage("/Index");
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Attach(Customer).State = EntityState.Modified;

            try
            {
                await _db.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                throw new Exception($"Customer {Customer.Id} not found!");
            }

            return RedirectToPage("/Index");
        }
    }
}

```

The *Index.cshtml* file also contains markup to create a delete button for each customer contact:

```

<button type="submit" asp-page-handler="delete"
        asp-route-id="@contact.Id">delete</button>

```

When the delete button is rendered in HTML, its `formaction` includes parameters for:

- The customer contact ID specified by the `asp-route-id` attribute.

- The `handler` specified by the `asp-page-handler` attribute.

Here is an example of a rendered delete button with a customer contact ID of `1`:

```
<button type="submit" formaction="/?id=1&handler=delete">delete</button>
```

When the button is selected, a form `POST` request is sent to the server. By convention, the name of the handler method is selected based the value of the `handler` parameter according to the scheme `OnPost[handler]Async`.

Because the `handler` is `delete` in this example, the `OnPostDeleteAsync` handler method is used to process the `POST` request. If the `asp-page-handler` is set to a different value, such as `remove`, a page handler method with the name `OnPostRemoveAsync` is selected.

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _db.Customers.FindAsync(id);

    if (contact != null)
    {
        _db.Customers.Remove(contact);
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

The `OnPostDeleteAsync` method:

- Accepts the `id` from the query string.
- Queries the database for the customer contact with `FindAsync`.
- If the customer contact is found, they're removed from the list of customer contacts. The database is updated.
- Calls `RedirectToPage` to redirect to the root Index page (`/Index`).

XSRF/CSRF and Razor Pages

You don't have to write any code for [antiforgery validation](#). Antiforgery token generation and validation are automatically included in Razor Pages.

Using Layouts, partials, templates, and Tag Helpers with Razor Pages

Pages work with all the features of the Razor view engine. Layouts, partials, templates, Tag Helpers, `_ViewStart.cshtml`, `_ViewImports.cshtml` work in the same way they do for conventional Razor views.

Let's declutter this page by taking advantage of some of those features.

Add a [layout page](#) to `Pages/_Layout.cshtml`:

```

<!DOCTYPE html>
<html>
<head>
  <title>Razor Pages Sample</title>
</head>
<body>
  <a asp-page="/Index">Home</a>
  @RenderBody()
  <a asp-page="/Customers/Create">Create</a> <br />
</body>
</html>

```

The [Layout](#):

- Controls the layout of each page (unless the page opts out of layout).
- Imports HTML structures such as JavaScript and stylesheets.

See [layout page](#) for more information.

The [Layout](#) property is set in *Pages/_ViewStart.cshtml*:

```

@{
    Layout = "_Layout";
}

```

Note: The layout is in the *Pages* folder. Pages look for other views (layouts, templates, partials) hierarchically, starting in the same folder as the current page. A layout in the *Pages* folder can be used from any Razor page under the *Pages* folder.

We recommend you **not** put the layout file in the *Views/Shared* folder. *Views/Shared* is an MVC views pattern. Razor Pages are meant to rely on folder hierarchy, not path conventions.

View search from a Razor Page includes the *Pages* folder. The layouts, templates, and partials you're using with MVC controllers and conventional Razor views *just work*.

Add a *Pages/_ViewImports.cshtml* file:

```

@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

```

`@namespace` is explained later in the tutorial. The `@addTagHelper` directive brings in the [built-in Tag Helpers](#) to all the pages in the *Pages* folder.

When the `@namespace` directive is used explicitly on a page:

```

@page
@namespace RazorPagesIntro.Pages.Customers

@model NameSpaceModel

<h2>Name space</h2>
<p>
  @Model.Message
</p>

```

The directive sets the namespace for the page. The `@model` directive doesn't need to include the namespace.

When the `@namespace` directive is contained in *_ViewImports.cshtml*, the specified namespace supplies the

prefix for the generated namespace in the Page that imports the `@namespace` directive. The rest of the generated namespace (the suffix portion) is the dot-separated relative path between the folder containing `_ViewImports.cshtml` and the folder containing the page.

For example, the code behind file `Pages/Customers/Edit.cshtml.cs` explicitly sets the namespace:

```
namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly ApplicationDbContext _db;

        public EditModel(ApplicationDbContext db)
        {
            _db = db;
        }

        // Code removed for brevity.
    }
}
```

The `Pages/_ViewImports.cshtml` file sets the following namespace:

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The generated namespace for the `Pages/Customers/Edit.cshtml` Razor Page is the same as the code behind file. The `@namespace` directive was designed so the C# classes added to a project and pages-generated code *just work* without having to add an `@using` directive for the code behind file.

Note: `@namespace` also works with conventional Razor views.

The original `Pages/Create.cshtml` view file:

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

The updated `Pages/Create.cshtml` view file:

```

@page
@model CreateModel

<html>
<body>
  <p>
    Enter your name.
  </p>
  <div asp-validation-summary="All"></div>
  <form method="POST">
    <div>Name: <input asp-for="Customer.Name" /></div>
    <input type="submit" />
  </form>
</body>
</html>

```

The [Razor Pages starter project](#) contains the *Pages/_ValidationScriptsPartial.cshtml*, which hooks up client-side validation.

URL generation for Pages

The `Create` page, shown previously, uses `RedirectToPage` :

```

public async Task<IActionResult> OnPostAsync()
{
  if (!ModelState.IsValid)
  {
    return Page();
  }

  _db.Customers.Add(Customer);
  await _db.SaveChangesAsync();
  return RedirectToPage("/Index");
}

```

The app has the following file/folder structure:

- */Pages*
 - *Index.cshtml*
 - */Customer*
 - *Create.cshtml*
 - *Edit.cshtml*
 - *Index.cshtml*

The *Pages/Customers/Create.cshtml* and *Pages/Customers/Edit.cshtml* pages redirect to *Pages/Index.cshtml* after success. The string `/Index` is part of the URI to access the preceding page. The string `/Index` can be used to generate URIs to the *Pages/Index.cshtml* page. For example:

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">My Index Page`
- `RedirectToPage("/Index")`

The page name is the path to the page from the root */Pages* folder (including a leading `/`, for example `/Index`). The preceding URL generation samples are much more feature rich than just hardcoding a URL. URL generation uses [routing](#) and can generate and encode parameters according to how the route is defined in the destination path.

URL generation for pages supports relative names. The following table shows which Index page is selected with different `RedirectToPage` parameters from `Pages/Customers/Create.cshtml`:

REDIRECTTOPAGE(X)	PAGE
<code>RedirectToPage("/Index")</code>	<code>Pages/Index</code>
<code>RedirectToPage("./Index");</code>	<code>Pages/Customers/Index</code>
<code>RedirectToPage("../Index")</code>	<code>Pages/Index</code>
<code>RedirectToPage("Index")</code>	<code>Pages/Customers/Index</code>

`RedirectToPage("Index")`, `RedirectToPage("./Index")`, and `RedirectToPage("../Index")` are *relative names*. The `RedirectToPage` parameter is *combined* with the path of the current page to compute the name of the destination page.

Relative name linking is useful when building sites with a complex structure. If you use relative names to link between pages in a folder, you can rename that folder. All the links still work (because they didn't include the folder name).

TempData

ASP.NET Core exposes the `TempData` property on a `controller`. This property stores data until it is read. The `Keep` and `Peek` methods can be used to examine the data without deletion. `TempData` is useful for redirection, when data is needed for more than a single request.

The `[TempData]` attribute is new in ASP.NET Core 2.0 and is supported on controllers and pages.

The following code sets the value of `Message` using `TempData`:

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

The following markup in the *Pages/Customers/Index.cshtml* file displays the value of `Message` using `TempData`.

```
<h3>Msg: @Model.Message</h3>
```

The *Pages/Customers/Index.cshtml.cs* code-behind file applies the `[TempData]` attribute to the `Message` property.

```
[TempData]  
public string Message { get; set; }
```

See [TempData](#) for more information.

Multiple handlers per page

The following page generates markup for two page handlers using the `asp-page-handler` Tag Helper:

```
@page  
@model CreateFATHModel  
  
<html>  
<body>  
  <p>  
    Enter your name.  
  </p>  
  <div asp-validation-summary="All"></div>  
  <form method="POST">  
    <div>Name: <input asp-for="Customer.Name" /></div>  
    <input type="submit" asp-page-handler="JoinList" value="Join" />  
    <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />  
  </form>  
</body>  
</html>
```

The form in the preceding example has two submit buttons, each using the `FormActionTagHelper` to submit to a different URL. The `asp-page-handler` attribute is a companion to `asp-page`. `asp-page-handler` generates URLs that submit to each of the handler methods defined by a page. `asp-page` is not specified because the sample is linking to the current page.

The code-behind file:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpper();
            return await OnPostJoinListAsync();
        }
    }
}

```

The preceding code uses *named handler methods*. Named handler methods are created by taking the text in the name after `On<HTTP Verb>` and before `Async` (if present). In the preceding example, the page methods are `OnPostJoinListAsync` and `OnPostJoinListUCAsync`. With `OnPost` and `Async` removed, the handler names are `JoinList` and `JoinListUC`.

```



```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `http://localhost:5000/Customers/CreateFATH?handler=JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `http://localhost:5000/Customers/CreateFATH?handler=JoinListUC`.

Customizing Routing

If you don't like the query string `?handler=JoinList` in the URL, you can change the route to put the handler name in the path portion of the URL. You can customize the route by adding a route template enclosed in double quotes after the `@page` directive.

```

@page "{handler?}"
@model CreateRouteModel

<html>
<body>
  <p>
    Enter your name.
  </p>
  <div asp-validation-summary="All"></div>
  <form method="POST">
    <div>Name: <input asp-for="Customer.Name" /></div>
    <input type="submit" asp-page-handler="JoinList" value="Join" />
    <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
  </form>
</body>
</html>

```

The preceding route puts the handler name in the URL path instead of the query string. The `?` following `handler` means the route parameter is optional.

You can use `@page` to add additional segments and parameters to a page's route. Whatever's there is **appended** to the default route of the page. Using an absolute or virtual path to change the page's route (like `"~/Some/Other/Path"`) is not supported.

Configuration and settings

To configure advanced options, use the extension method `AddRazorPagesOptions` on the MVC builder:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.RootDirectory = "/MyPages";
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        });
}

```

Currently you can use the `RazorPagesOptions` to set the root directory for pages, or add application model conventions for pages. We'll enable more extensibility this way in the future.

To precompile views, see [Razor view compilation](#).

[Download or view sample code.](#)

See [Getting started with Razor Pages in ASP.NET Core](#), which builds on this introduction.

Specify that Razor Pages are at the content root

By default, Razor Pages are rooted in the `/Pages` directory. Add `WithRazorPagesAtContentRoot` to `AddMvc` to specify that your Razor Pages are at the content root (`ContentRootPath`) of the app:

```

services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesAtContentRoot();

```

Specify that Razor Pages are at a custom root directory

Add [WithRazorPagesRoot](#) to [AddMvc](#) to specify that your Razor Pages are at a custom root directory in the app (provide a relative path):

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesRoot("/path/to/razor/pages");
```

See also

- [Getting started with Razor Pages](#)
- [Razor Pages authorization conventions](#)
- [Razor Pages custom route and page model providers](#)
- [Razor Pages unit and integration testing](#)

Razor syntax for ASP.NET Core

11/1/2017 • 12 min to read • [Edit Online](#)

By [Rick Anderson](#), [Luke Latham](#), [Taylor Mullen](#), and [Dan Vicarel](#)

Razor is a markup syntax for embedding server-based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a *.cshtml* file extension.

Rendering HTML

The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in *.cshtml* Razor files is rendered by the server unchanged.

Razor syntax

Razor supports C# and uses the `@` symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output.

When an `@` symbol is followed by a [Razor reserved keyword](#), it transitions into Razor-specific markup. Otherwise, it transitions into plain C#.

To escape an `@` symbol in Razor markup, use a second `@` symbol:

```
<p>@@Username</p>
```

The code is rendered in HTML with a single `@` symbol:

```
<p>@Username</p>
```

HTML attributes and content containing email addresses don't treat the `@` symbol as a transition character. The email addresses in the following example are untouched by Razor parsing:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Implicit Razor expressions

Implicit Razor expressions start with `@` followed by C# code:

```
<p>@DateTime.Now</p>  
<p>@DateTime.IsLeapYear(2016)</p>
```

With the exception of the C# `await` keyword, implicit expressions must not contain spaces. If the C# statement has a clear ending, spaces can be intermingled:

```
<p>@await DoSomething("hello", "world")</p>
```

Implicit expressions **cannot** contain C# generics, as the characters inside the brackets (`<>`) are interpreted as an HTML tag. The following code is **not** valid:

```
<p>@GenericMethod<int>(</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element was not closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?`

Generic method calls must be wrapped in an [explicit Razor expression](#) or a [Razor code block](#). This restriction doesn't apply to *.vbhtml* Razor files because Visual Basic syntax places parentheses around generic type parameters instead of brackets.

Explicit Razor expressions

Explicit Razor expressions consist of an `@` symbol with balanced parenthesis. To render last week's time, the following Razor markup is used:

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Any content within the `@()` parenthesis is evaluated and rendered to the output.

Implicit expressions, described in the previous section, generally can't contain spaces. In the following code, one week isn't subtracted from the current time:

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

The code renders the following HTML:

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

Explicit expressions can be used to concatenate text with an expression result:

```
@{  
    var joe = new Person("Joe", 33);  
}  
  
<p>Age@(joe.Age)</p>
```

Without the explicit expression, `<p>Age@joe.Age</p>` is treated as an email address, and `<p>Age@joe.Age</p>` is rendered. When written as an explicit expression, `<p>Age33</p>` is rendered.

Explicit expressions can be used to render output from generic methods in *.cshtml* files. In an implicit expression, the characters inside the brackets (`<>`) are interpreted as an HTML tag. The following markup is **not** valid Razor:

```
<p>@GenericMethod<int>(</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element was not closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?`

The following markup shows the correct way write this code. The code is written as an explicit expression:

```
<p>@(GenericMethod<int>())</p>
```

Note: this restriction doesn't apply to *.vbhtml* Razor files. With *.vbhtml* Razor files, Visual Basic syntax places parentheses around generic type parameters instead of brackets.

Expression encoding

C# expressions that evaluate to a string are HTML encoded. C# expressions that evaluate to `IHtmlContent` are rendered directly through `IHtmlContent.WriteTo`. C# expressions that don't evaluate to `IHtmlContent` are converted to a string by `ToString` and encoded before they're rendered.

```
@("<span>Hello World</span>")
```

The code renders the following HTML:

```
&lt;span&gt;Hello World&lt;/span&gt;
```

The HTML is shown in the browser as:

```
<span>Hello World</span>
```

`HtmlHelper.Raw` output isn't encoded but rendered as HTML markup.

WARNING

Using `HtmlHelper.Raw` on unsanitized user input is a security risk. User input might contain malicious JavaScript or other exploits. Sanitizing user input is difficult. Avoid using `HtmlHelper.Raw` with user input.

```
@Html.Raw("<span>Hello World</span>")
```

The code renders the following HTML:

```
<span>Hello World</span>
```

Razor code blocks

Razor code blocks start with `@` and are enclosed by `{}`. Unlike expressions, C# code inside code blocks isn't rendered. Code blocks and expressions in a view share the same scope and are defined in order:

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";
}

<p>@quote</p>
```

The code renders the following HTML:

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.</p>
```

Implicit transitions

The default language in a code block is C#, but the Razor Page can transition back to HTML:

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Explicit delimited transition

To define a subsection of a code block that should render HTML, surround the characters for rendering with the Razor **<text>** tag:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

Use this approach to render HTML that isn't surrounded by an HTML tag. Without an HTML or Razor tag, a Razor runtime error occurs.

The **<text>** tag is useful to control whitespace when rendering content:

- Only the content between the **<text>** tag is rendered.
- No whitespace before or after the **<text>** tag appears in the HTML output.

Explicit Line Transition with @:

To render the rest of an entire line as HTML inside a code block, use the **@:** syntax:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

Without the **@:** in the code, a Razor runtime error is generated.

Warning: Extra **@** characters in a Razor file can cause compiler errors at statements later in the block. These compiler errors can be difficult to understand because the actual error occurs before the reported error. This error is

common after combining multiple implicit/explicit expressions into a single code block.

Control Structures

Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures:

Conditionals @if, else if, else, and @switch

@if controls when code runs:

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

else and else if don't require the @ symbol:

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

The following markup shows how to use a switch statement:

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

Looping @for, @foreach, @while, and @do while

Templated HTML can be rendered with looping control statements. To render a list of people:

```
@{
    var people = new Person[]
    {
        new Person("Weston", 33),
        new Person("Johnathon", 41),
        ...
    };
}
```

The following looping statements are supported:

@for

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

@foreach

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

@while

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

@do while

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

Compound @using

In C#, a `using` statement is used to ensure an object is disposed. In Razor, the same mechanism is used to create HTML Helpers that contain additional content. In the following code, HTML Helpers render a form tag with the `@using` statement:

```
@using (Html.BeginForm())
{
    <div>
        email:
        <input type="email" id="Email" value="">
        <button>Register</button>
    </div>
}
```

Scope-level actions can be performed with [Tag Helpers](#).

@try, catch, finally

Exception handling is similar to C#:

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

@lock

Razor has the capability to protect critical sections with lock statements:

```
@lock (SomeLock)
{
    // Do critical section work
}
```

Comments

Razor supports C# and HTML comments:

```
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
```

The code renders the following HTML:

```
<!-- HTML comment -->
```

Razor comments are removed by the server before the webpage is rendered. Razor uses `@* *@` to delimit comments. The following code is commented out, so the server doesn't render any markup:

```
@*
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
*@
```

Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive typically changes the way a view is parsed or enables different functionality.

Understanding how Razor generates code for a view makes it easier to understand how directives work.

```
@{
    var quote = "Getting old ain't for wimps! - Anonymous";
}

<div>Quote of the Day: @quote</div>
```

The code generates a class similar to the following:

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

Later in this article, the section [Viewing the Razor C# class generated for a view](#) explains how to view this generated class.

@using

The `@using` directive adds the C# `using` directive to the generated view:

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

@model

The `@model` directive specifies the type of the model passed to a view:

```
@model TypeNameOfModel
```

In an ASP.NET Core MVC app created with individual user accounts, the `Views/Account/Login.cshtml` view contains the following model declaration:

```
@model LoginViewModel
```

The class generated inherits from `RazorPage<dynamic>`:

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor exposes a `Model` property for accessing the model passed to the view:

```
<div>The Login Email: @Model.Email</div>
```

The `@model` directive specifies the type of this property. The directive specifies the `T` in `RazorPage<T>` that the generated class that the view derives from. If the `@model` directive isn't specified, the `Model` property is of type `dynamic`. The value of the model is passed from the controller to the view. For more information, see [Strongly

typed models and the @model keyword.

@inherits

The @inherits directive provides full control of the class the view inherits:

```
@inherits TypeNameOfClassToInheritFrom
```

The following code is a custom Razor page type:

```
using Microsoft.AspNetCore.Mvc.Razor;  
  
public abstract class CustomRazorPage<TModel> : RazorPage<TModel>  
{  
    public string CustomText { get; } = "Gardylloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below."  
}
```

The CustomText is displayed in a view:

```
@inherits CustomRazorPage<TModel>  
  
<div>Custom text: @CustomText</div>
```

The code renders the following HTML:

```
<div>Custom text: Gardylloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below.</div>
```

@model and @inherits can be used in the same view. @inherits can be in a *_ViewImports.cshtml* file that the view imports:

```
@inherits CustomRazorPage<TModel>
```

The following code is an example of a strongly-typed view:

```
@inherits CustomRazorPage<TModel>  
  
<div>The Login Email: @Model.Email</div>  
<div>Custom text: @CustomText</div>
```

If "rick@contoso.com" is passed in the model, the view generates the following HTML markup:

```
<div>The Login Email: rick@contoso.com</div>  
<div>Custom text: Gardylloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below.</div>
```

@inject

The @inject directive enables the Razor Page to inject a service from the [service container](#) into a view. For more information, see [Dependency injection into views](#).

@functions

The @functions directive enables a Razor Page to add function-level content to a view:

```
@functions { // C# Code }
```

For example:

```
@functions {  
    public string GetHello()  
    {  
        return "Hello";  
    }  
}  
  
<div>From method: @GetHello()</div>
```

The code generates the following HTML markup:

```
<div>From method: Hello</div>
```

The following code is the generated Razor C# class:

```
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Mvc.Razor;  
  
public class _Views_Home_Test_cshtml : RazorPage<dynamic>  
{  
    // Functions placed between here  
    public string GetHello()  
    {  
        return "Hello";  
    }  
    // And here.  
    #pragma warning disable 1998  
    public override async Task ExecuteAsync()  
    {  
        WriteLiteral("\r\n<div>From method: ");  
        Write(GetHello());  
        WriteLiteral("</div>\r\n");  
    }  
    #pragma warning restore 1998  
}
```

@section

The `@section` directive is used in conjunction with the [layout](#) to enable views to render content in different parts of the HTML page. For more information, see [Sections](#).

Tag Helpers

There are three directives that pertain to [Tag Helpers](#).

DIRECTIVE	FUNCTION
@addTagHelper	Makes Tag Helpers available to a view.
@removeTagHelper	Removes Tag Helpers previously added from a view.
@tagHelperPrefix	Specifies a tag prefix to enable Tag Helper support and to make Tag Helper usage explicit.

Razor reserved keywords

Razor keywords

- page (Requires ASP.NET Core 2.0 and later)
- functions
- inherits
- model
- section
- helper (Not currently supported by ASP.NET Core)

Razor keywords are escaped with `@(Razor Keyword)` (for example, `@(functions)`).

C# Razor keywords

- case
- do
- default
- for
- foreach
- if
- else
- lock
- switch
- try
- catch
- finally
- using
- while

C# Razor keywords must be double-escaped with `@(@C# Razor Keyword)` (for example, `@(@case)`). The first `@` escapes the Razor parser. The second `@` escapes the C# parser.

Reserved keywords not used by Razor

- namespace
- class

Viewing the Razor C# class generated for a view

Add the following class to the ASP.NET Core MVC project:

```

using Microsoft.AspNetCore.Mvc.Razor.Extensions;
using Microsoft.AspNetCore.Mvc.Razor.Language;

public class CustomTemplateEngine : MvcRazorTemplateEngine
{
    public CustomTemplateEngine(RazorEngine engine, RazorProject project)
        : base(engine, project)
    {
    }

    public override RazorCSharpDocument GenerateCode(RazorCodeDocument codeDocument)
    {
        var csharpDocument = base.GenerateCode(codeDocument);
        var generatedCode = csharpDocument.GeneratedCode;

        // Look at generatedCode

        return csharpDocument;
    }
}

```

Override the `RazorTemplateEngine` added by MVC with the `CustomTemplateEngine` class:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSingleton<RazorTemplateEngine, CustomTemplateEngine>();
}

```

Set a break point on the `return csharpDocument` statement of `CustomTemplateEngine`. When program execution stops at the break point, view the value of `generatedCode`.

The screenshot shows the 'Text Visualizer' window with the following content:

```

Expression: generatedCode
Value:
    public override async Task ExecuteAsync()
    {
#line 1 "/Views/Home/Contact3.cshtml"
        var output = "Hello World";
#line default
#line hidden
        BeginContext(40, 15, true);
        WriteLiteral("\r\n<div>Output: ");
        EndContext();
        BeginContext(56, 6, false);
#line 5 "/Views/Home/Contact3.cshtml"
        Write(output);
#line default
#line hidden
        EndContext();
        BeginContext(62, 6, true);
        WriteLiteral("</div>");
        EndContext();
    }
#pragma warning restore 1998

```

At the bottom of the window, there is a 'Wrap' checkbox (unchecked), and 'Close' and 'Help' buttons.

View lookups and case sensitivity

The Razor view engine performs case-sensitive lookups for views. However, the actual lookup is determined by the underlying file system:

- File based source:
 - On operating systems with case insensitive file systems (for example, Windows), physical file provider lookups are case insensitive. For example, `return View("Test")` results in matches for */Views/Home/Test.cshtml*, */Views/home/test.cshtml*, and any other casing variant.
 - On case-sensitive file systems (for example, Linux, OSX, and with `EmbeddedFileProvider`), lookups are case-sensitive. For example, `return View("Test")` specifically matches */Views/Home/Test.cshtml*.
- Precompiled views: With ASP.NET Core 2.0 and later, looking up precompiled views is case insensitive on all operating systems. The behavior is identical to physical file provider's behavior on Windows. If two precompiled views differ only in case, the result of lookup is non-deterministic.

Developers are encouraged to match the casing of file and directory names to the casing of:

- * Area, controller, and action names.
- * Razor Pages.

Matching case ensures the deployments find their views regardless of the underlying file system.

Razor Pages route and app convention features in ASP.NET Core

11/3/2017 • 15 min to read • [Edit Online](#)

By [Luke Latham](#)

Learn how to use page route and app model provider convention features to control page routing, discovery, and processing in Razor Pages apps. When you need to configure custom page routes for individual pages, configure routing to pages with the [AddPageRoute convention](#) described later in this topic.

Use the [sample app \(how to download\)](#) to explore the features described in this topic.

FEATURES	THE SAMPLE DEMONSTRATES ...
Route and app model conventions Conventions.Add <ul style="list-style-type: none">• IPageRouteModelConvention• IPageApplicationModelConvention	Adding a route template and header to an app's pages.
Page route action conventions <ul style="list-style-type: none">• AddFolderRouteModelConvention• AddPageRouteModelConvention• AddPageRoute	Adding a route template to pages in a folder and to a single page.
Page model action conventions <ul style="list-style-type: none">• AddFolderApplicationModelConvention• AddPageApplicationModelConvention• ConfigureFilter (filter class, lambda expression, or filter factory)	Adding a header to pages in a folder, adding a header to a single page, and configuring a filter factory to add a header to an app's pages.
Default page app model provider	Replacing the default page model provider to change the conventions for handler naming.

Add route and app model conventions

Add a delegate for [IPageConvention](#) to add route and app model conventions that apply to Razor Pages.

Add a route model convention to all pages

Use [Conventions](#) to create and add an [IPageRouteModelConvention](#) to the collection of [IPageConvention](#) instances that are applied during route and page model construction.

The sample app adds a `{globalTemplate?}` route template to all of the pages in the app:

```

public class GlobalTemplatePageRouteModelConvention
    : IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 0,
                    Template = AttributeRouteModel.CombineTemplates(
                        selector.AttributeRouteModel.Template,
                        "{globalTemplate?}"),
                }
            });
        }
    }
}

```

NOTE

The `Order` property for the `AttributeRouteModel` is set to `0` (zero). This ensures that this template is given priority for the first route data value position when a single route value is provided. For example, the sample adds an `{aboutTemplate?}` route template later in the topic. The `{aboutTemplate?}` template is given an `Order` of `1`. When the About page is requested at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 0`) and not `RouteData.Values["aboutTemplate"]` (`Order = 1`) due to setting the `Order` property.

Startup.cs:

```
options.Conventions.Add(new GlobalTemplatePageRouteModelConvention());
```

Request the sample's About page at `localhost:5000/About/GlobalRouteValue` and inspect the result:

Add an app model convention to all pages

Use [Conventions](#) to create and add an `IPageApplicationModelConvention` to the collection of `IPageConvention`

instances that are applied during route and page model construction.

To demonstrate this and other conventions later in the topic, the sample app includes an `AddHeaderAttribute` class. The class constructor accepts a `name` string and a `values` string array. These values are used in its `OnResultExecuting` method to set a response header. The full class is shown in the [Page model action conventions](#) section later in the topic.

The sample app uses the `AddHeaderAttribute` class to add a header, `GlobalHeader`, to all of the pages in the app:

```
public class GlobalHeaderPageApplicationModelConvention
    : IPageApplicationModelConvention
{
    public void Apply(PageApplicationModel model)
    {
        model.Filters.Add(new AddHeaderAttribute(
            "GlobalHeader", new string[] { "Global Header Value" }));
    }
}
```

Startup.cs:

```
options.Conventions.Add(new GlobalHeaderPageApplicationModelConvention());
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:

```
▼ Response Headers    view source
  AboutHeader: About Header Value
  Content-Type: text/html; charset=utf-8
  Date: Thu, 19 Oct 2017 21:09:07 GMT
  FilterFactoryHeader: Filter Factory Header Value 1
  FilterFactoryHeader: Filter Factory Header Value 2
  GlobalHeader: Global Header Value
  Server: Kestrel
  Transfer-Encoding: chunked
```

Page route action conventions

The default route model provider that derives from [IPageRouteModelProvider](#) invokes conventions which are designed to provide extensibility points for configuring page routes.

Folder route model convention

Use [AddFolderRouteModelConvention](#) to create and add an [IPageRouteModelConvention](#) that invokes an action on the [PageRouteModel](#) for all of the pages under the specified folder.

The sample app uses `AddFolderRouteModelConvention` to add an `{otherPagesTemplate?}` route template to the pages in the *OtherPages* folder:

```

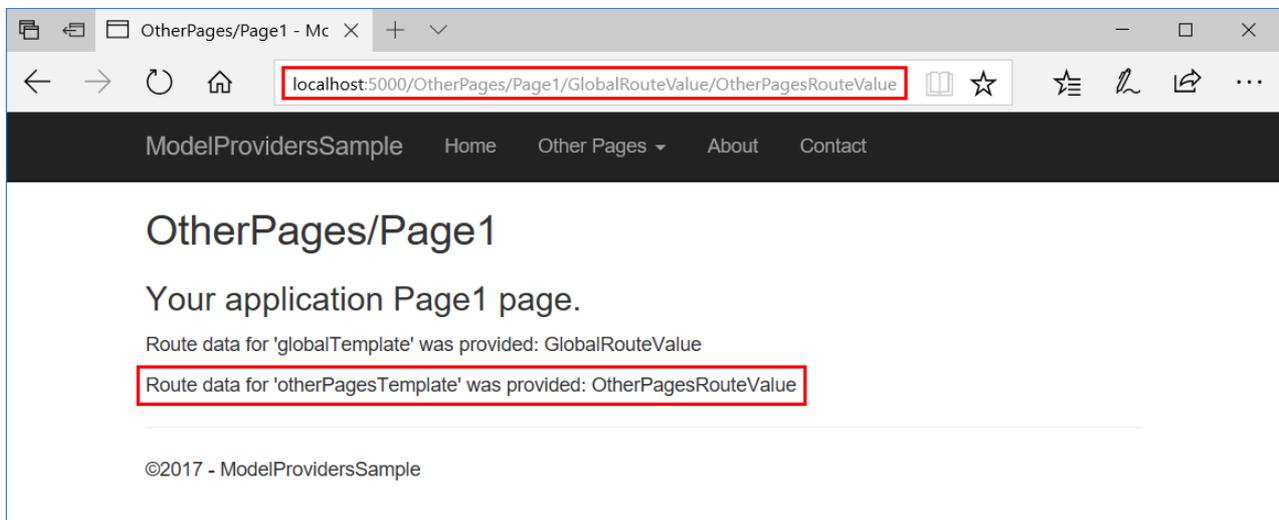
options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 1,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{otherPagesTemplate?}"),
            }
        });
    }
});
});

```

NOTE

The `Order` property for the `AttributeRouteModel` is set to `1`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic) is given priority for the first route data value position when a single route value is provided. If the Page1 page is requested at `/OtherPages/Page1/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 0`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 1`) due to setting the `Order` property.

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1/GlobalRouteValue/OtherPagesRouteValue` and inspect the result:



Page route model convention

Use `AddPageRouteModelConvention` to create and add an `IPageRouteModelConvention` that invokes an action on the `PageRouteModel` for the page with the specified name.

The sample app uses `AddPageRouteModelConvention` to add an `{aboutTemplate?}` route template to the About page:

```

options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 1,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{aboutTemplate?}"),
            }
        });
    }
});
});

```

NOTE

The `Order` property for the `AttributeRouteModel` is set to `1`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic) is given priority for the first route data value position when a single route value is provided. If the About page is requested at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 0`) and not `RouteData.Values["aboutTemplate"]` (`Order = 1`) due to setting the `Order` property.

Request the sample's About page at `localhost:5000/About/GlobalRouteValue/AboutRouteValue` and inspect the result:

Configure a page route

Use `AddPageRoute` to configure a route to a page at the specified page path. Generated links to the page use your specified route. `AddPageRoute` uses `AddPageRouteModelConvention` to establish the route.

The sample app creates a route to `/TheContactPage` for `Contact.cshtml`:

```

options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");

```

The Contact page can also be reached at `/Contact` via its default route.

The sample app's custom route to the Contact page allows for an optional `text` route segment (`{text?}`). The page also includes this optional segment in its `@page` directive in case the visitor accesses the page at its `/Contact` route:

```
@page "{text?}"
@model ContactModel
@{
    ViewData["Title"] = "Contact";
}

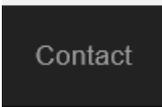
<h1>@ViewData["Title"]</h1>
<h2>@Model.Message</h2>

<address>
    One Microsoft Way<br>
    Redmond, WA 98052-6399<br>
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a href="mailto:Support@example.com">Support@example.com</a><br>
    <strong>Marketing:</strong> <a href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>

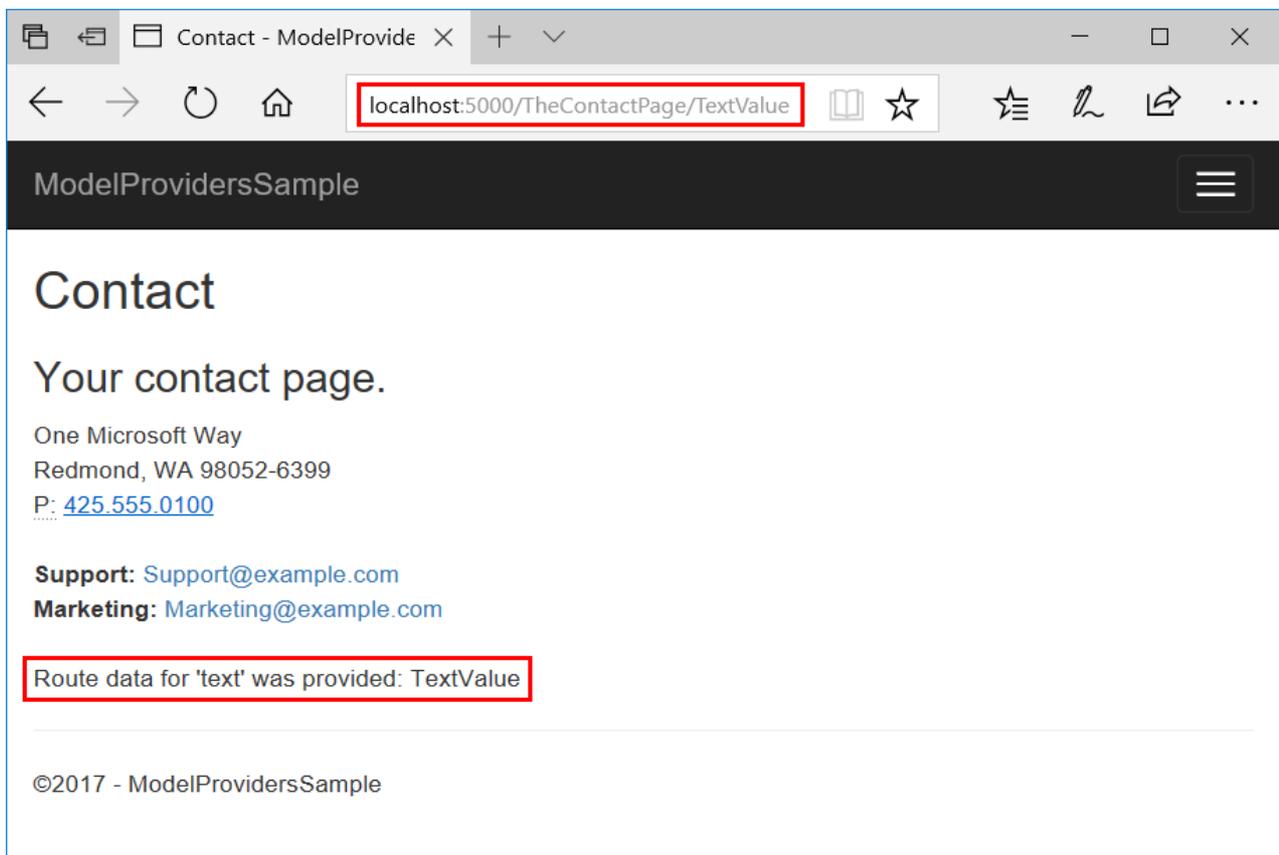
<p>@Model.RouteDataTextTemplateValue</p>
```

Note that the URL generated for the **Contact** link in the rendered page reflects the updated route:



```
<li>...</li>
<li>
    <a href="/TheContactPage">Contact</a>
</li>
::after
</ul>
```

Visit the Contact page at either its ordinary route, `/Contact`, or the custom route, `/TheContactPage`. If you supply an additional `text` route segment, the page shows the HTML-encoded segment that you provide:



Page model action conventions

The default page model provider that implements [IPageApplicationModelProvider](#) invokes conventions which are designed to provide extensibility points for configuring page models. These conventions are useful when building and modifying page discovery and processing features.

For the examples in this section, the sample app uses an `AddHeaderAttribute` class, which is a [ResultFilterAttribute](#), that applies a response header:

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string[] _values;

    public AddHeaderAttribute(string name, string[] values)
    {
        _name = name;
        _values = values;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _values);
        base.OnResultExecuting(context);
    }
}
```

Using conventions, the sample demonstrates how to apply the attribute to all of the pages in a folder and to a single page.

Folder app model convention

Use [AddFolderApplicationModelConvention](#) to create and add an [IPageApplicationModelConvention](#) that invokes an action on [PageApplicationModel](#) instances for all pages under the specified folder.

The sample demonstrates the use of `AddFolderApplicationModelConvention` by adding a header, `OtherPagesHeader`, to the pages inside the `OtherPages` folder of the app:

```
options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "OtherPagesHeader", new string[] { "OtherPages Header Value" }));
});
```

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1` and inspect the headers to view the result:

▼ **Response Headers** [view source](#)

Content-Type: text/html; charset=utf-8
Date: Sat, 21 Oct 2017 06:13:16 GMT
FilterFactoryHeader: Filter Factory Header Value 1
FilterFactoryHeader: Filter Factory Header Value 2
GlobalHeader: Global Header Value
OtherPagesHeader: OtherPages Header Value
Server: Kestrel
Transfer-Encoding: chunked

Page app model convention

Use `AddPageApplicationModelConvention` to create and add an `IPageApplicationModelConvention` that invokes an action on the `PageApplicationModel` for the page with the specified name.

The sample demonstrates the use of `AddPageApplicationModelConvention` by adding a header, `AboutHeader`, to the About page:

```
options.Conventions.AddPageApplicationModelConvention("/About", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "AboutHeader", new string[] { "About Header Value" }));
});
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:

▼ **Response Headers** [view source](#)

AboutHeader: About Header Value
Content-Type: text/html; charset=utf-8
Date: Thu, 19 Oct 2017 21:09:07 GMT
FilterFactoryHeader: Filter Factory Header Value 1
FilterFactoryHeader: Filter Factory Header Value 2
GlobalHeader: Global Header Value
Server: Kestrel
Transfer-Encoding: chunked

Configure a filter

`ConfigureFilter` configures the specified filter to apply. You can implement a filter class, but the sample app shows how to implement a filter in a lambda expression, which is implemented behind-the-scenes as a factory that returns a filter:

```
options.Conventions.ConfigureFilter(model =>
{
    if (model.RelativePath.Contains("OtherPages/Page2"))
    {
        return new AddHeaderAttribute(
            "OtherPagesPage2Header",
            new string[] { "OtherPages/Page2 Header Value" });
    }
    return new EmptyFilter();
});
```

The page app model is used to check the relative path for segments that lead to the Page2 page in the *OtherPages* folder. If the condition passes, a header is added. If not, the `EmptyFilter` is applied.

`EmptyFilter` is an [Action filter](#). Since Action filters are ignored by Razor Pages, the `EmptyFilter` no-ops as intended if the path doesn't contain `OtherPages/Page2`.

Request the sample's Page2 page at `localhost:5000/OtherPages/Page2` and inspect the headers to view the result:

▼ **Response Headers** [view source](#)

```
Content-Type: text/html; charset=utf-8
Date: Mon, 23 Oct 2017 06:06:52 GMT
FilterFactoryHeader: Filter Factory Header Value 1
FilterFactoryHeader: Filter Factory Header Value 2
GlobalHeader: Global Header Value
OtherPagesHeader: OtherPages Header Value
OtherPagesPage2Header: OtherPages/Page2 Header Value
Server: Kestrel
Transfer-Encoding: chunked
```

Configure a filter factory

`ConfigureFilter` configures the specified factory to apply [filters](#) to all Razor Pages.

The sample app provides an example of using a [filter factory](#) by adding a header, `FilterFactoryHeader`, with two values to the app's pages:

```
options.Conventions.ConfigureFilter(new AddHeaderWithFactory());
```

AddHeaderWithFactory.cs:

```

public class AddHeaderWithFactory : IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new AddHeaderFilter();
    }

    private class AddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "FilterFactoryHeader",
                new string[]
                {
                    "Filter Factory Header Value 1",
                    "Filter Factory Header Value 2"
                });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}

```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:

▼ **Response Headers** [view source](#)

AboutHeader: About Header Value
Content-Type: text/html; charset=utf-8
Date: Thu, 19 Oct 2017 21:09:07 GMT
FilterFactoryHeader: Filter Factory Header Value 1
FilterFactoryHeader: Filter Factory Header Value 2
GlobalHeader: Global Header Value
Server: Kestrel
Transfer-Encoding: chunked

Replace the default page app model provider

Razor Pages uses the `IPageApplicationModelProvider` interface to create a [DefaultPageApplicationModelProvider](#). You can inherit from the default model provider to provide your own implementation logic for handler discovery and processing. The default implementation ([reference source](#)) establishes conventions for *unnamed* and *named* handler naming, which is described below.

Default unnamed handler methods

Handler methods for HTTP verbs ("unnamed" handler methods) follow a convention: `On<HTTP verb>[Async]` (appending `Async` is optional but recommended for async methods).

UNNAMED HANDLER METHOD	OPERATION
<code>OnGet / OnGetAsync</code>	Initialize the page state.
<code>OnPost / OnPostAsync</code>	Handle POST requests.
<code>OnDelete / OnDeleteAsync</code>	Handle DELETE requests†.
<code>OnPut / OnPutAsync</code>	Handle PUT requests†.
<code>OnPatch / OnPatchAsync</code>	Handle PATCH requests†.

†Used for making API calls to the page.

Default named handler methods

Handler methods provided by the developer ("named" handler methods) follow a similar convention. The handler name appears after the HTTP verb or between the HTTP verb and `Async`: `On<HTTP verb><handler name>[Async]` (appending `Async` is optional but recommended for async methods). For example, methods that process messages might take the naming shown in the table below.

EXAMPLE NAMED HANDLER METHOD	EXAMPLE OPERATION
<code>OnGetMessage / OnGetMessageAsync</code>	Obtain a message.
<code>OnPostMessage / OnPostMessageAsync</code>	POST a message.
<code>OnDeleteMessage / OnDeleteMessageAsync</code>	DELETE a message†.
<code>OnPutMessage / OnPutMessageAsync</code>	PUT a message†.
<code>OnPatchMessage / OnPatchMessageAsync</code>	PATCH a message†.

†Used for making API calls to the page.

Customize handler method names

Assume that you prefer to change the way unnamed and named handler methods are named. An alternative naming scheme is to avoid starting the method names with "On" and use the first word segment to determine the HTTP verb. You can make other changes, such as converting the verbs for DELETE, PUT, and PATCH to POST. Such a scheme provides the method names shown in the following table.

HANDLER METHOD	OPERATION
<code>Get</code>	Initialize the page state.
<code>Post / PostAsync</code>	Handle POST requests.
<code>Delete / DeleteAsync</code>	Handle DELETE requests†.
<code>Put / PutAsync</code>	Handle PUT requests†.
<code>Patch / PatchAsync</code>	Handle PATCH requests†.

HANDLER METHOD	OPERATION
<code>GetMessage</code>	Obtain a message.
<code>PostMessage</code> / <code>PostMessageAsync</code>	POST a message.
<code>DeleteMessage</code> / <code>DeleteMessageAsync</code>	POST a message to delete.
<code>PutMessage</code> / <code>PutMessageAsync</code>	POST a message to put.
<code>PatchMessage</code> / <code>PatchMessageAsync</code>	POST a message to patch.

†Used for making API calls to the page.

To establish this scheme, inherit from the `DefaultPageApplicationModelProvider` class and override the `CreateHandlerModel` method to supply custom logic for resolving `PageModel` handler names. The sample app shows you how this is done in its `CustomPageApplicationModelProvider` class:

```
public class CustomPageApplicationModelProvider :
    DefaultPageApplicationModelProvider
{
    protected override PageHandlerModel CreateHandlerModel(MethodInfo method)
    {
        if (method == null)
        {
            throw new ArgumentNullException(nameof(method));
        }

        if (!IsHandler(method))
        {
            return null;
        }

        if (!TryParseHandlerMethod(
            method.Name, out var httpMethod, out var handlerName))
        {
            return null;
        }

        var handlerModel = new PageHandlerModel(
            method,
            method.GetCustomAttributes(inherit: true))
        {
            Name = method.Name,
            HandlerName = handlerName,
            HttpMethod = httpMethod,
        };

        var methodParameters = handlerModel.MethodInfo.GetParameters();

        for (var i = 0; i < methodParameters.Length; i++)
        {
            var parameter = methodParameters[i];
            var parameterModel = CreateParameterModel(parameter);
            parameterModel.Handler = handlerModel;

            handlerModel.Parameters.Add(parameterModel);
        }

        return handlerModel;
    }

    private static bool TryParseHandlerMethod(
```

```

private static bool TryParseHandlerMethod(
    string methodName, out string httpMethod, out string handler)
{
    httpMethod = null;
    handler = null;

    // Parse the method name according to our conventions to
    // determine the required HTTP verb and optional
    // handler name.
    //
    // Valid names look like:
    // - Get
    // - Post
    // - PostAsync
    // - GetMessage
    // - PostMessage
    // - DeleteMessage
    // - DeleteMessageAsync

    var length = methodName.Length;
    if (methodName.EndsWith("Async", StringComparison.Ordinal))
    {
        length -= "Async".Length;
    }

    if (length == 0)
    {
        // The method is named "Async". Exit processing.
        return false;
    }

    // The HTTP verb is at the start of the method name. Use
    // casing to determine where it ends.
    var handlerNameStart = 1;
    for (; handlerNameStart < length; handlerNameStart++)
    {
        if (char.IsUpper(methodName[handlerNameStart]))
        {
            break;
        }
    }

    httpMethod = methodName.Substring(0, handlerNameStart);

    // The handler name follows the HTTP verb and is optional.
    // It includes everything up to the end excluding the
    // "Async" suffix, if present.
    handler = handlerNameStart == length ? null : methodName.Substring(0, length);

    if (string.Equals(httpMethod, "GET", StringComparison.OrdinalIgnoreCase) ||
        string.Equals(httpMethod, "POST", StringComparison.OrdinalIgnoreCase))
    {
        // Do nothing. The httpMethod is correct for GET and POST.
        return true;
    }
    if (string.Equals(httpMethod, "DELETE", StringComparison.OrdinalIgnoreCase) ||
        string.Equals(httpMethod, "PUT", StringComparison.OrdinalIgnoreCase) ||
        string.Equals(httpMethod, "PATCH", StringComparison.OrdinalIgnoreCase))
    {
        // Convert HTTP verbs for DELETE, PUT, and PATCH to POST
        // For example: DeleteMessage, PutMessage, PatchMessage -> POST
        httpMethod = "POST";
        return true;
    }
    else
    {
        return false;
    }
}

```

```
}
```

Highlights of the class include:

- The class inherits from `DefaultPageApplicationModelProvider`.
- The `TryParseHandlerMethod` processes a handler to determine the HTTP verb (`httpMethod`) and named handler name (`handlerName`) when creating the `PageHandlerModel`.
 - An `Async` postfix is ignored, if present.
 - Casing is used to parse the HTTP verb from the method name.
 - When the method name (without `Async`) is equal to the HTTP verb name, there's no named handler. The `handlerName` is set to `null`, and the method name is `Get`, `Post`, `Delete`, `Put`, or `Patch`.
 - When the method name (without `Async`) is longer than the HTTP verb name, there's a named handler. The `handlerName` is set to `<method name (less 'Async', if present)>`. For example, both "GetMessage" and "GetMessageAsync" yield a handler name of "GetMessage".
 - DELETE, PUT, and PATCH HTTP verbs are converted to POST.

Register the `CustomPageApplicationModelProvider` in the `Startup` class:

```
services.AddSingleton<IPageApplicationModelProvider,  
    CustomPageApplicationModelProvider>();
```

The code-behind file `Index.cshtml.cs` shows how the ordinary handler method naming conventions are changed for pages in the app. The ordinary "On" prefix naming used with Razor Pages is removed. The method that initializes the page state is now named `Get`. You can see this convention used throughout the app if you open any code-behind file for any of the pages.

Each of the other methods start with the HTTP verb that describes its processing. The two methods that start with `Delete` would normally be treated as DELETE HTTP verbs, but the logic in `TryParseHandlerMethod` explicitly sets the verb to POST for both handlers.

Note that `Async` is optional between `DeleteAllMessages` and `DeleteMessageAsync`. They're both asynchronous methods, but you can choose to use the `Async` postfix or not; we recommend that you do. `DeleteAllMessages` is used here for demonstration purposes, but we recommend that you name such a method `DeleteAllMessagesAsync`. It doesn't affect the processing of the sample's implementation, but using the `Async` postfix calls out the fact that it's an asynchronous method.

```

public async Task Get()
{
    Messages = await _db.Messages.AsNoTracking().ToListAsync();
}

public async Task<IActionResult> PostMessageAsync()
{
    _db.Messages.Add(Message);
    await _db.SaveChangesAsync();

    Result = $"{nameof(PostMessageAsync)} handler: Message '{Message.Text}' added.";

    return RedirectToPage();
}

public async Task<IActionResult> DeleteAllMessages()
{
    foreach (Message message in _db.Messages)
    {
        _db.Messages.Remove(message);
    }
    await _db.SaveChangesAsync();

    Result = $"{nameof(DeleteAllMessages)} handler: All messages deleted.";

    return RedirectToPage();
}

public async Task<IActionResult> DeleteMessageAsync(int id)
{
    var message = await _db.Messages.FindAsync(id);

    if (message != null)
    {
        _db.Messages.Remove(message);
        await _db.SaveChangesAsync();
    }

    Result = $"{nameof(DeleteMessageAsync)} handler: Message with Id: {id} deleted.";

    return RedirectToPage();
}

```

Note the handler names provided in *Index.cshtml* match the `DeleteAllMessages` and `DeleteMessageAsync` handler methods:

```

<div class="row">
  <div class="col-md-3">
    <form method="post">
      <h2>Clear all messages</h2>
      <hr>
      <div class="form-group">
        <button type="submit" asp-page-handler="DeleteAllMessages"
          class="btn btn-danger">Clear All</button>
      </div>
    </form>
  </div>
</div>

<div class="row">
  <div class="col-md-12">
    <form method="post">
      <h2>Messages</h2>
      <hr>
      <ol>
        @foreach (var message in Model.Messages)
        {
          <li>
            @message.Text
            <button type="submit" asp-page-handler="DeleteMessage"
              asp-route-id="@message.Id">delete</button>
          </li>
        }
      </ol>
    </form>
  </div>
</div>

```

`Async` in the handler method name `DeleteMessageAsync` is factored out by the `TryParseHandlerMethod` for handler matching of POST request to method. The `asp-page-handler` name of `DeleteMessage` is matched to the handler method `DeleteMessageAsync`.

MVC Filters and the Page filter (IPageFilter)

MVC [Action filters](#) are ignored by Razor Pages, since Razor Pages use handler methods. Other types of MVC filters are available for you to use: [Authorization](#), [Exception](#), [Resource](#), and [Result](#). For more information, see the [Filters](#) topic.

The Page filter ([IPageFilter](#)) is a filter that applies to Razor Pages. It surrounds the execution of a page handler method. It allows you to process custom code at stages of page handler method execution. Here's an example from the sample app:

```
[AttributeUsage(AttributeTargets.Class)]
public class ReplaceRouteValueFilterAttribute : Attribute, IPageFilter
{
    public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
    {
        // Called after the handler method executes before the result.
    }

    public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        // Called before the handler method executes after model binding is complete.
    }

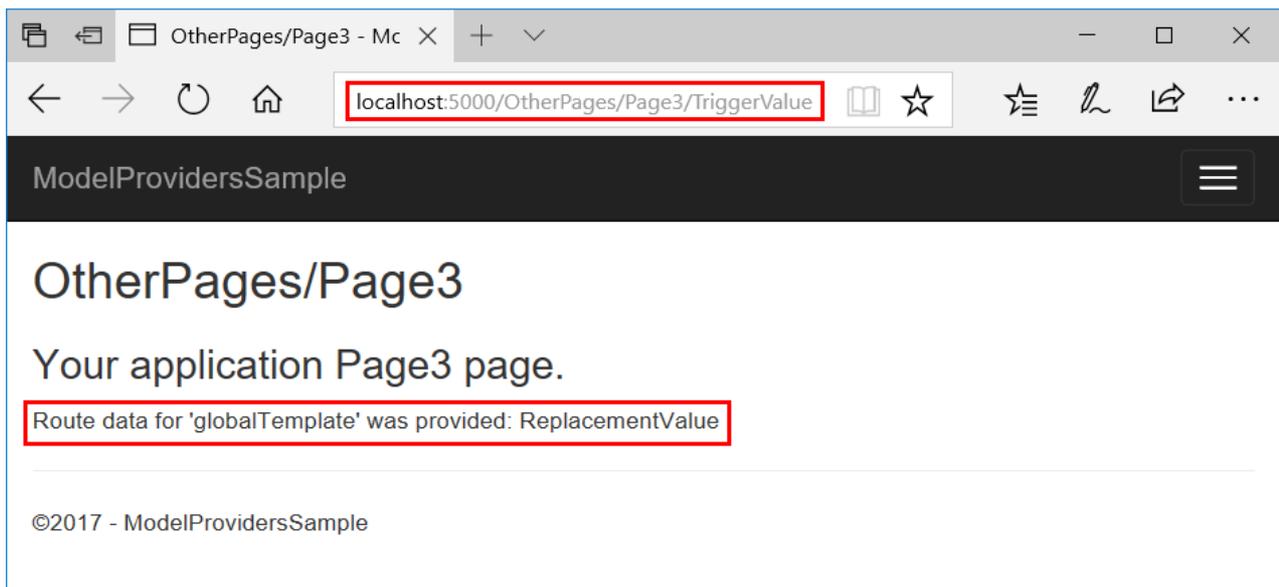
    public void OnPageHandlerSelected(PageHandlerSelectedContext context)
    {
        // Called after a handler method is selected but before model binding occurs.
        context.RouteData.Values.TryGetValue("globalTemplate",
            out var globalTemplateValue);
        if (string.Equals((string)globalTemplateValue, "TriggerValue",
            StringComparison.Ordinal))
        {
            context.RouteData.Values["globalTemplate"] = "ReplacementValue";
        }
    }
}
}
```

This filter checks for a `globalTemplate` route value of "TriggerValue" and swaps in "ReplacementValue".

The `ReplaceRouteValueFilter` attribute can be applied directly to a `PageModel` in code-behind:

```
[ReplaceRouteValueFilter]
public class Page3Model : PageModel
{
}
```

Request the Page3 page from the sample app with at `localhost:5000/OtherPages/Page3/TriggerValue`. Notice how the filter replaces the route value:



See also

- [Razor Pages authorization conventions](#)

Model Binding

11/21/2017 • 7 min to read • [Edit Online](#)

By [Rachel Appel](#)

Introduction to model binding

Model binding in ASP.NET Core MVC maps data from HTTP requests to action method parameters. The parameters may be simple types such as strings, integers, or floats, or they may be complex types. This is a great feature of MVC because mapping incoming data to a counterpart is an often repeated scenario, regardless of size or complexity of the data. MVC solves this problem by abstracting binding away so developers don't have to keep rewriting a slightly different version of that same code in every app. Writing your own text to type converter code is tedious, and error prone.

How model binding works

When MVC receives an HTTP request, it routes it to a specific action method of a controller. It determines which action method to run based on what is in the route data, then it binds values from the HTTP request to that action method's parameters. For example, consider the following URL:

```
http://contoso.com/movies/edit/2
```

Since the route template looks like this, `{controller=Home}/{action=Index}/{id?}`, `movies/edit/2` routes to the `Movies` controller, and its `Edit` action method. It also accepts an optional parameter called `id`. The code for the action method should look something like this:

```
public IActionResult Edit(int? id)
```

Note: The strings in the URL route are not case sensitive.

MVC will try to bind request data to the action parameters by name. MVC will look for values for each parameter using the parameter name and the names of its public settable properties. In the above example, the only action parameter is named `id`, which MVC binds to the value with the same name in the route values. In addition to route values MVC will bind data from various parts of the request and it does so in a set order. Below is a list of the data sources in the order that model binding looks through them:

1. `Form values`: These are form values that go in the HTTP request using the POST method. (including jQuery POST requests).
2. `Route values`: The set of route values provided by [Routing](#)
3. `Query strings`: The query string part of the URI.

Note: Form values, route data, and query strings are all stored as name-value pairs.

Since model binding asked for a key named `id` and there is nothing named `id` in the form values, it moved on to the route values looking for that key. In our example, it's a match. Binding happens, and the value is converted to the integer 2. The same request using `Edit(string id)` would convert to the string "2".

So far the example uses simple types. In MVC simple types are any .NET primitive type or type with a string type converter. If the action method's parameter were a class such as the `Movie` type, which contains both simple and complex types as properties, MVC's model binding will still handle it nicely. It uses reflection and

recursion to traverse the properties of complex types looking for matches. Model binding looks for the pattern `parameter_name.property_name` to bind values to properties. If it doesn't find matching values of this form, it will attempt to bind using just the property name. For those types such as `Collection` types, model binding looks for matches to `parameter_name[index]` or just `[index]`. Model binding treats `Dictionary` types similarly, asking for `parameter_name[key]` or just `[key]`, as long as the keys are simple types. Keys that are supported match the field names HTML and tag helpers generated for the same model type. This enables round-tripping values so that the form fields remain filled with the user's input for their convenience, for example, when bound data from a create or edit did not pass validation.

In order for binding to happen the class must have a public default constructor and member to be bound must be public writable properties. When model binding happens the class will only be instantiated using the public default constructor, then the properties can be set.

When a parameter is bound, model binding stops looking for values with that name and it moves on to bind the next parameter. Otherwise, the default model binding behavior sets parameters to their default values depending on their type:

- `T[]`: With the exception of arrays of type `byte[]`, binding sets parameters of type `T[]` to `Array.Empty<T>()`. Arrays of type `byte[]` are set to `null`.
- Reference Types: Binding creates an instance of a class with the default constructor without setting properties. However, model binding sets `string` parameters to `null`.
- Nullable Types: Nullable types are set to `null`. In the above example, model binding sets `id` to `null` since it is of type `int?`.
- Value Types: Non-nullable value types of type `T` are set to `default(T)`. For example, model binding will set a parameter `int id` to 0. Consider using model validation or nullable types rather than relying on default values.

If binding fails, MVC does not throw an error. Every action which accepts user input should check the `ModelState.IsValid` property.

Note: Each entry in the controller's `ModelState` property is a `ModelStateEntry` containing an `Errors` property. It's rarely necessary to query this collection yourself. Use `ModelState.IsValid` instead.

Additionally, there are some special data types that MVC must consider when performing model binding:

- `IFormFile`, `IEnumerable<IFormFile>`: One or more uploaded files that are part of the HTTP request.
- `CancellationToken`: Used to cancel activity in asynchronous controllers.

These types can be bound to action parameters or to properties on a class type.

Once model binding is complete, [Validation](#) occurs. Default model binding works great for the vast majority of development scenarios. It is also extensible so if you have unique needs you can customize the built-in behavior.

Customize model binding behavior with attributes

MVC contains several attributes that you can use to direct its default model binding behavior to a different source. For example, you can specify whether binding is required for a property, or if it should never happen at all by using the `[BindRequired]` or `[BindNever]` attributes. Alternatively, you can override the default data source, and specify the model binder's data source. Below is a list of model binding attributes:

- `[BindRequired]`: This attribute adds a model state error if binding cannot occur.
- `[BindNever]`: Tells the model binder to never bind to this parameter.

- `[FromHeader]`, `[FromQuery]`, `[FromRoute]`, `[FromForm]`: Use these to specify the exact binding source you want to apply.
- `[FromServices]`: This attribute uses [dependency injection](#) to bind parameters from services.
- `[FromBody]`: Use the configured formatters to bind data from the request body. The formatter is selected based on content type of the request.
- `[ModelBinder]`: Used to override the default model binder, binding source and name.

Attributes are very helpful tools when you need to override the default behavior of model binding.

Binding formatted data from the request body

Request data can come in a variety of formats including JSON, XML and many others. When you use the `[FromBody]` attribute to indicate that you want to bind a parameter to data in the request body, MVC uses a configured set of formatters to handle the request data based on its content type. By default MVC includes a `JsonInputFormatter` class for handling JSON data, but you can add additional formatters for handling XML and other custom formats.

NOTE

There can be at most one parameter per action decorated with `[FromBody]`. The ASP.NET Core MVC run-time delegates the responsibility of reading the request stream to the formatter. Once the request stream is read for a parameter, it's generally not possible to read the request stream again for binding other `[FromBody]` parameters.

NOTE

The `JsonInputFormatter` is the default formatter and is based on [Json.NET](#).

ASP.NET selects input formatters based on the [Content-Type](#) header and the type of the parameter, unless there is an attribute applied to it specifying otherwise. If you'd like to use XML or another format you must configure it in the *Startup.cs* file, but you may first have to obtain a reference to

`Microsoft.AspNetCore.Mvc.Formatters.Xml` using NuGet. Your startup code should look something like this:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddXmlSerializerFormatters();
}
```

Code in the *Startup.cs* file contains a `ConfigureServices` method with a `services` argument you can use to build up services for your ASP.NET app. In the sample, we are adding an XML formatter as a service that MVC will provide for this app. The `options` argument passed into the `AddMvc` method allows you to add and manage filters, formatters, and other system options from MVC upon app startup. Then apply the `Consumes` attribute to controller classes or action methods to work with the format you want.

Custom Model Binding

You can extend model binding by writing your own custom model binders. Learn more about [custom model binding](#).

Introduction to model validation in ASP.NET Core MVC

12/19/2017 • 15 min to read • [Edit Online](#)

By [Rachel Appel](#)

Introduction to model validation

Before an app stores data in a database, the app must validate the data. Data must be checked for potential security threats, verified that it is appropriately formatted by type and size, and it must conform to your rules. Validation is necessary although it can be redundant and tedious to implement. In MVC, validation happens on both the client and server.

Fortunately, .NET has abstracted validation into validation attributes. These attributes contain validation code, thereby reducing the amount of code you must write.

[View or download sample from GitHub.](#)

Validation Attributes

Validation attributes are a way to configure model validation so it's similar conceptually to validation on fields in database tables. This includes constraints such as assigning data types or required fields. Other types of validation include applying patterns to data to enforce business rules, such as a credit card, phone number, or email address. Validation attributes make enforcing these requirements much simpler and easier to use.

Below is an annotated `Movie` model from an app that stores information about movies and TV shows. Most of the properties are required and several string properties have length requirements. Additionally, there is a numeric range restriction in place for the `Price` property from 0 to \$999.99, along with a custom validation attribute.

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    [Required]
    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

Simply reading through the model reveals the rules about data for this app, making it easier to maintain the code. Below are several popular built-in validation attributes:

- `[CreditCard]`: Validates the property has a credit card format.
- `[Compare]`: Validates two properties in a model match.
- `[EmailAddress]`: Validates the property has an email format.
- `[Phone]`: Validates the property has a telephone format.
- `[Range]`: Validates the property value falls within the given range.
- `[RegularExpression]`: Validates that the data matches the specified regular expression.
- `[Required]`: Makes a property required.
- `[StringLength]`: Validates that a string property has at most the given maximum length.
- `[Url]`: Validates the property has a URL format.

MVC supports any attribute that derives from `ValidationAttribute` for validation purposes. Many useful validation attributes can be found in the `System.ComponentModel.DataAnnotations` namespace.

There may be instances where you need more features than built-in attributes provide. For those times, you can create custom validation attributes by deriving from `ValidationAttribute` or changing your model to implement `IValidatableObject`.

Notes on the use of the Required attribute

Non-nullable [value types](#) (such as `decimal`, `int`, `float`, and `DateTime`) are inherently required and don't need the `Required` attribute. The app performs no server-side validation checks for non-nullable types that are marked `Required`.

MVC model binding, which isn't concerned with validation and validation attributes, rejects a form field submission containing a missing value or whitespace for a non-nullable type. In the absence of a `BindRequired` attribute on the target property, model binding ignores missing data for non-nullable types, where the form field is absent from the incoming form data.

The `BindRequired` attribute (also see [Customize model binding behavior with attributes](#)) is useful to ensure form data is complete. When applied to a property, the model binding system requires a value for that property. When applied to a type, the model binding system requires values for all of the properties of that type.

When you use a [Nullable<T> type](#) (for example, `decimal?` or `System.Nullable<decimal>`) and mark it `Required`, a server-side validation check is performed as if the property were a standard nullable type (for example, a `string`).

Client-side validation requires a value for a form field that corresponds to a model property that you've marked `Required` and for a non-nullable type property that you haven't marked `Required`. `Required` can be used to control the client-side validation error message.

Model State

Model state represents validation errors in submitted HTML form values.

MVC will continue validating fields until reaches the maximum number of errors (200 by default). You can configure this number by inserting the following code into the `ConfigureServices` method in the `Startup.cs` file:

```
services.AddMvc(options => options.MaxModelValidationErrors = 50);
```

Handling Model State Errors

Model validation occurs prior to each controller action being invoked, and it is the action method's responsibility to inspect `ModelState.IsValid` and react appropriately. In many cases, the appropriate reaction is to return an error response, ideally detailing the reason why model validation failed.

Some apps will choose to follow a standard convention for dealing with model validation errors, in which case a filter may be an appropriate place to implement such a policy. You should test how your actions behave with valid and invalid model states.

Manual validation

After model binding and validation are complete, you may want to repeat parts of it. For example, a user may have entered text in a field expecting an integer, or you may need to compute a value for a model's property.

You may need to run validation manually. To do so, call the `TryValidateModel` method, as shown here:

```
TryValidateModel(movie);
```

Custom validation

Validation attributes work for most validation needs. However, some validation rules are specific to your business. Your rules might not be common data validation techniques such as ensuring a field is required or that it conforms to a range of values. For these scenarios, custom validation attributes are a great solution. Creating your own custom validation attributes in MVC is easy. Just inherit from the `ValidationAttribute`, and override the `IsValid` method. The `IsValid` method accepts two parameters, the first is an object named *value* and the second is a `ValidationContext` object named *validationContext*. *Value* refers to the actual value from the field that your custom validator is validating.

In the following sample, a business rule states that users may not set the genre to *Classic* for a movie released after 1960. The `[ClassicMovie]` attribute checks the genre first, and if it is a classic, then it checks the release date to see that it is later than 1960. If it is released after 1960, validation fails. The attribute accepts an integer parameter representing the year that you can use to validate data. You can capture the value of the parameter in the attribute's constructor, as shown here:

```

public class ClassicMovieAttribute : ValidationAttribute, IClientModelValidator
{
    private int _year;

    public ClassicMovieAttribute(int Year)
    {
        _year = Year;
    }

    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        Movie movie = (Movie)validationContext.ObjectInstance;

        if (movie.Genre == Genre.Classic && movie.ReleaseDate.Year > _year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
}

```

The `movie` variable above represents a `Movie` object that contains the data from the form submission to validate. In this case, the validation code checks the date and genre in the `IsValid` method of the `ClassicMovieAttribute` class as per the rules. Upon successful validation `IsValid` returns a `ValidationResult.Success` code, and when validation fails, a `ValidationResult` with an error message. When a user modifies the `Genre` field and submits the form, the `IsValid` method of the `ClassicMovieAttribute` will verify whether the movie is a classic. Like any built-in attribute, apply the `ClassicMovieAttribute` to a property such as `ReleaseDate` to ensure validation happens, as shown in the previous code sample. Since the example works only with `Movie` types, a better option is to use `IValidatableObject` as shown in the following paragraph.

Alternatively, this same code could be placed in the model by implementing the `Validate` method on the `IValidatableObject` interface. While custom validation attributes work well for validating individual properties, implementing `IValidatableObject` can be used to implement class-level validation as seen here.

```

public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
    {
        yield return new ValidationResult(
            $"Classic movies must have a release year earlier than {_classicYear}.",
            new[] { "ReleaseDate" });
    }
}

```

Client side validation

Client side validation is a great convenience for users. It saves time they would otherwise spend waiting for a round trip to the server. In business terms, even a few fractions of seconds multiplied hundreds of times each day adds up to be a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

You must have a view with the proper JavaScript script references in place for client side validation to work as you see here.

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"></script>
```

```

<script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js"></script>
<script
src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js
"></script>

```

The [jQuery Unobtrusive Validation](#) script is a custom Microsoft front-end library that builds on the popular [jQuery Validate](#) plugin. Without jQuery Unobtrusive Validation, you would have to code the same validation logic in two places: once in the server side validation attributes on model properties, and then again in client side scripts (the examples for jQuery Validate's `validate()` method shows how complex this could become). Instead, MVC's [Tag Helpers](#) and [HTML helpers](#) are able to use the validation attributes and type metadata from model properties to render HTML 5 [data- attributes](#) in the form elements that need validation. MVC generates the `data-` attributes for both built-in and custom attributes. jQuery Unobtrusive Validation then parses the `data-` attributes and passes the logic to jQuery Validate, effectively "copying" the server side validation logic to the client. You can display validation errors on the client using the relevant tag helpers as shown here:

```

<div class="form-group">
  <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="ReleaseDate" class="form-control" />
    <span asp-validation-for="ReleaseDate" class="text-danger"></span>
  </div>
</div>

```

The tag helpers above render the HTML below. Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `ReleaseDate` property. The `data-val-required` attribute below contains an error message to display if the user doesn't fill in the release date field. jQuery Unobtrusive Validation passes this value to the jQuery Validate `required()` method, which then displays that message in the accompanying `` element.

```

<form action="/Movies/Create" method="post">
  <div class="form-horizontal">
    <h4>Movie</h4>
    <div class="text-danger"></div>
    <div class="form-group">
      <label class="col-md-2 control-label" for="ReleaseDate">ReleaseDate</label>
      <div class="col-md-10">
        <input class="form-control" type="datetime"
          data-val="true" data-val-required="The ReleaseDate field is required."
          id="ReleaseDate" name="ReleaseDate" value="" />
        <span class="text-danger field-validation-valid"
          data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>
      </div>
    </div>
  </div>
</form>

```

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

MVC determines type attribute values based on the .NET data type of a property, possibly overridden using `[DataType]` attributes. The base `[DataType]` attribute does no real server-side validation. Browsers choose their own error messages and display those errors however they wish, however the jQuery Validation Unobtrusive package can override the messages and display them consistently with others. This happens most obviously when users apply `[DataType]` subclasses such as `[EmailAddress]`.

Add Validation to Dynamic Forms

Because jQuery Unobtrusive Validation passes validation logic and parameters to jQuery Validate when the page first loads, dynamically generated forms will not automatically exhibit validation. Instead, you must tell jQuery Unobtrusive Validation to parse the dynamic form immediately after creating it. For example, the code below shows how you might set up client side validation on a form added via AJAX.

```
$.get({
  url: "https://url/that/returns/a/form",
  dataType: "html",
  error: function(jqXHR, textStatus, errorThrown) {
    alert(textStatus + ": Could not add form. " + errorThrown);
  },
  success: function(newFormHTML) {
    var container = document.getElementById("form-container");
    container.insertAdjacentHTML("beforeend", newFormHTML);
    var forms = container.getElementsByTagName("form");
    var newForm = forms[forms.length - 1];
    $.validator.unobtrusive.parse(newForm);
  }
})
```

The `$.validator.unobtrusive.parse()` method accepts a jQuery selector for its one argument. This method tells jQuery Unobtrusive Validation to parse the `data-` attributes of forms within that selector. The values of those attributes are then passed to the jQuery Validate plugin so that the form exhibits the desired client side validation rules.

Add Validation to Dynamic Controls

You can also update the validation rules on a form when individual controls, such as `<input/>`s and `<select/>`s, are dynamically generated. You cannot pass selectors for these elements to the `parse()` method directly because the surrounding form has already been parsed and will not update. Instead, you first remove the existing validation data, then reparse the entire form, as shown below:

```
$.get({
  url: "https://url/that/returns/a/control",
  dataType: "html",
  error: function(jqXHR, textStatus, errorThrown) {
    alert(textStatus + ": Could not add form. " + errorThrown);
  },
  success: function(newInputHTML) {
    var form = document.getElementById("my-form");
    form.insertAdjacentHTML("beforeend", newInputHTML);
    form.removeData("validator") // Added by the raw jQuery Validate
      .removeData("unobtrusiveValidation"); // Added by jQuery Unobtrusive Validation
    $.validator.unobtrusive.parse(form);
  }
})
```

IClientModelValidator

You may create client side logic for your custom attribute, and [unobtrusive validation](#) will execute it on the client for you automatically as part of validation. The first step is to control what data- attributes are added by implementing the `IClientModelValidator` interface as shown here:

```

public void AddValidation(ClientModelValidationContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    MergeAttribute(context.Attributes, "data-val", "true");
    MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMessage());

    var year = _year.ToString(CultureInfo.InvariantCulture);
    MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
}

```

Attributes that implement this interface can add HTML attributes to generated fields. Examining the output for the `ReleaseDate` element reveals HTML that is similar to the previous example, except now there is a `data-val-classicmovie` attribute that was defined in the `AddValidation` method of `IClientModelValidator`.

```

<input class="form-control" type="datetime"
    data-val="true"
    data-val-classicmovie="Classic movies must have a release year earlier than 1960."
    data-val-classicmovie-year="1960"
    data-val-required="The ReleaseDate field is required."
    id="ReleaseDate" name="ReleaseDate" value="" />

```

Unobtrusive validation uses the data in the `data-` attributes to display error messages. However, jQuery doesn't know about rules or messages until you add them to jQuery's `validator` object. This is shown in the example below that adds a method named `classicmovie` containing custom client validation code to the jQuery `validator` object.

```

$(function () {
    jQuery.validator.addMethod('classicmovie',
        function (value, element, params) {
            // Get element value. Classic genre has value '0'.
            var genre = $(params[0]).val(),
                year = params[1],
                date = new Date(value);
            if (genre && genre.length > 0 && genre[0] === '0') {
                // Since this is a classic movie, invalid if release date is after given year.
                return date.getFullYear() <= year;
            }
        }

        return true;
    });

    jQuery.validator.unobtrusive.adapters.add('classicmovie',
        [ 'element', 'year' ],
        function (options) {
            var element = $(options.form).find('select#Genre')[0];
            options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
            options.messages['classicmovie'] = options.message;
        });
})(jQuery);

```

Now jQuery has the information to execute the custom JavaScript validation as well as the error message to display if that validation code returns false.

Remote validation

Remote validation is a great feature to use when you need to validate data on the client against data on the

server. For example, your app may need to verify whether an email or user name is already in use, and it must query a large amount of data to do so. Downloading large sets of data for validating one or a few fields consumes too many resources. It may also expose sensitive information. An alternative is to make a round-trip request to validate a field.

You can implement remote validation in a two step process. First, you must annotate your model with the `[Remote]` attribute. The `[Remote]` attribute accepts multiple overloads you can use to direct client side JavaScript to the appropriate code to call. The example below points to the `VerifyEmail` action method of the `Users` controller.

```
[Remote(action: "VerifyEmail", controller: "Users")]
public string Email { get; set; }
```

The second step is putting the validation code in the corresponding action method as defined in the `[Remote]` attribute. According to the jQuery Validate `remote()` method documentation:

The serverside response must be a JSON string that must be `"true"` for valid elements, and can be `"false"`, `undefined`, or `null` for invalid elements, using the default error message. If the serverside response is a string, eg. `"That name is already taken, try peter123 instead"`, this string will be displayed as a custom error message in place of the default.

The definition of the `VerifyEmail()` method follows these rules, as shown below. It returns a validation error message if the email is taken, or `true` if the email is free, and wraps the result in a `JsonResult` object. The client side can then use the returned value to proceed or display the error if needed.

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyEmail(string email)
{
    if (!_userRepository.VerifyEmail(email))
    {
        return Json($"Email {email} is already in use.");
    }

    return Json(true);
}
```

Now when users enter an email, JavaScript in the view makes a remote call to see if that email has been taken and, if so, displays the error message. Otherwise, the user can submit the form as usual.

The `AdditionalFields` property of the `[Remote]` attribute is useful for validating combinations of fields against data on the server. For example, if the `User` model from above had two additional properties called `FirstName` and `LastName`, you might want to verify that no existing users already have that pair of names. You define the new properties as shown in the following code:

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(LastName))]
public string FirstName { get; set; }
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName))]
public string LastName { get; set; }
```

`AdditionalFields` could have been set explicitly to the strings `"FirstName"` and `"LastName"`, but using the `nameof` operator like this simplifies later refactoring. The action method to perform the validation must then accept two arguments, one for the value of `FirstName` and one for the value of `LastName`.

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userRepository.VerifyName(firstName, lastName))
    {
        return Json(data: $"A user named {firstName} {lastName} already exists.");
    }

    return Json(data: true);
}
```

Now when users enter a first and last name, JavaScript:

- Makes a remote call to see if that pair of names has been taken.
- If the pair has been taken, an error message is displayed.
- If not taken, the user can submit the form.

If you need to validate two or more additional fields with the `[Remote]` attribute, you provide them as a comma-delimited list. For example, to add a `MiddleName` property to the model, set the `[Remote]` attribute as shown in the following code:

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName) + "," +
nameof(LastName))]
public string MiddleName { get; set; }
```

`AdditionalFields`, like all attribute arguments, must be a constant expression. Therefore, you must not use an [interpolated string](#) or call `string.Join()` to initialize `AdditionalFields`. For every additional field that you add to the `[Remote]` attribute, you must add another argument to the corresponding controller action method.

Views in ASP.NET Core MVC

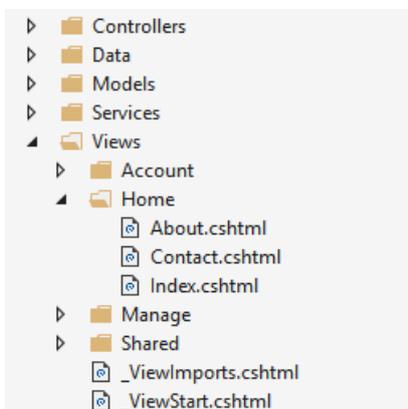
12/12/2017 • 12 min to read • [Edit Online](#)

By [Steve Smith](#) and [Luke Latham](#)

This document explains views used in ASP.NET Core MVC applications. For information on Razor Pages, see [Introduction to Razor Pages](#).

In the **Model-View-Controller** (MVC) pattern, the *view* handles the app's data presentation and user interaction. A view is an HTML template with embedded [Razor markup](#). Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.

In ASP.NET Core MVC, views are *.cshtml* files that use the [C# programming language](#) in Razor markup. Usually, view files are grouped into folders named for each of the app's [controllers](#). The folders are stored in a *Views* folder at the root of the app:



The *Home* controller is represented by a *Home* folder inside the *Views* folder. The *Home* folder contains the views for the *About*, *Contact*, and *Index* (homepage) webpages. When a user requests one of these three webpages, controller actions in the *Home* controller determine which of the three views is used to build and return a webpage to the user.

Use [layouts](#) to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

[Partial views](#) reduce code duplication by managing reusable parts of views. For example, a partial view is useful for an author biography on a blog website that appears in several views. An author biography is ordinary view content and doesn't require code to execute in order to produce the content for the webpage. Author biography content is available to the view by model binding alone, so using a partial view for this type of content is ideal.

[View components](#) are similar to partial views in that they allow you to reduce repetitive code, but they're appropriate for view content that requires code to run on the server in order to render the webpage. View components are useful when the rendered content requires database interaction, such as for a website shopping cart. View components aren't limited to model binding in order to produce webpage output.

Benefits of using views

Views help to establish a [Separation of Concerns \(SoC\) design](#) within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes your app modular, which provides several benefits:

- The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
- The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
- It's easier to test the user interface parts of the app because the views are separate units.
- Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

Creating a view

Views that are specific to a controller are created in the *Views/[ControllerName]* folder. Views that are shared among controllers are placed in the *Views/Shared* folder. To create a view, add a new file and give it the same name as its associated controller action with the *.cshtml* file extension. To create a view that corresponds with the *About* action in the *Home* controller, create an *About.cshtml* file in the *Views/Home* folder:

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Razor markup starts with the `@` symbol. Run C# statements by placing C# code within [Razor code blocks](#) set off by curly braces (`{ ... }`). For example, see the assignment of "About" to `ViewData["Title"]` shown above. You can display values within HTML by simply referencing the value with the `@` symbol. See the contents of the `<h2>` and `<h3>` elements above.

The view content shown above is only part of the entire webpage that's rendered to the user. The rest of the page's layout and other common aspects of the view are specified in other view files. To learn more, see the [Layout topic](#).

How controllers specify views

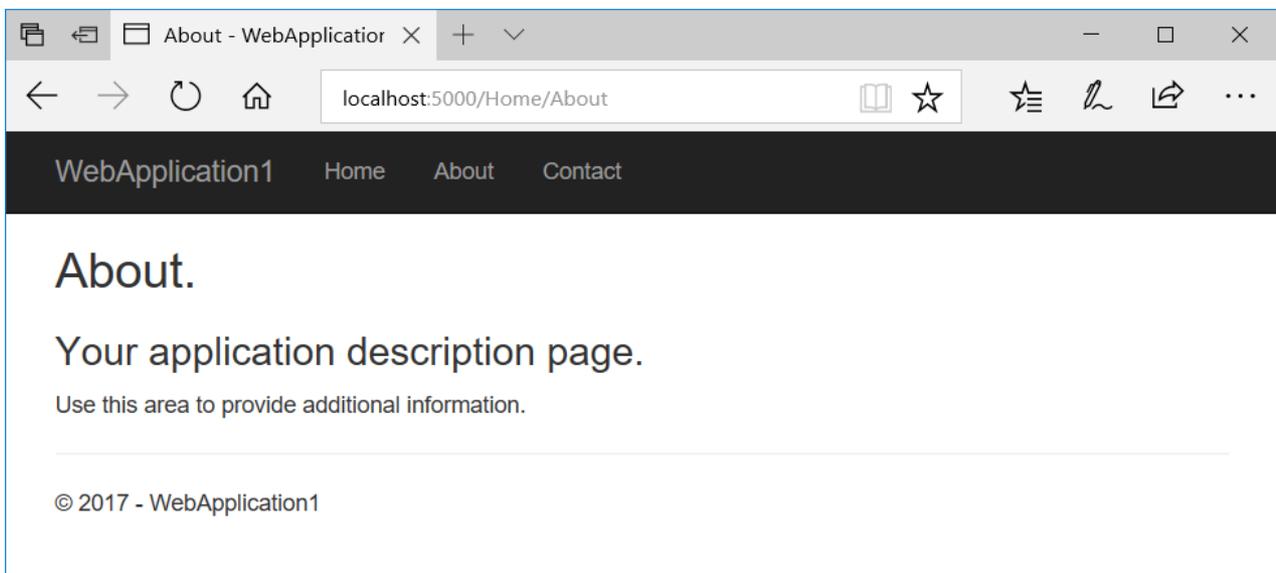
Views are typically returned from actions as a [ViewResult](#), which is a type of [ActionResult](#). Your action method can create and return a `ViewResult` directly, but that isn't commonly done. Since most controllers inherit from [Controller](#), you simply use the `View` helper method to return the `ViewResult`:

HomeController.cs

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

When this action returns, the *About.cshtml* view shown in the last section is rendered as the following webpage:



The `View` helper method has several overloads. You can optionally specify:

- An explicit view to return:

```
return View("Orders");
```

- A [model](#) to pass to the view:

```
return View(Orders);
```

- Both a view and a model:

```
return View("Orders", Orders);
```

View discovery

When an action returns a view, a process called *view discovery* takes place. This process determines which view file is used based on the view name.

The default behavior of the `View` method (`return View();`) is to return a view with the same name as the action method from which it's called. For example, the `About` `ActionResult` method name of the controller is used to search for a view file named `About.cshtml`. First, the runtime looks in the `Views/[ControllerName]` folder for the view. If it doesn't find a matching view there, it searches the `Shared` folder for the view.

It doesn't matter if you implicitly return the `ViewResult` with `return View();` or explicitly pass the view name to the `View` method with `return View("<ViewName>");`. In both cases, view discovery searches for a matching view file in this order:

1. `Views/[ControllerName]/[ViewName].cshtml`
2. `Views/Shared/[ViewName].cshtml`

A view file path can be provided instead of a view name. If using an absolute path starting at the app root (optionally starting with `/` or `~/`), the `.cshtml` extension must be specified:

```
return View("Views/Home/About.cshtml");
```

You can also use a relative path to specify views in different directories without the `.cshtml` extension. Inside the `HomeController`, you can return the `Index` view of your `Manage` views with a relative path:

```
return View("../Manage/Index");
```

Similarly, you can indicate the current controller-specific directory with the "/" prefix:

```
return View("./About");
```

[Partial views](#) and [view components](#) use similar (but not identical) discovery mechanisms.

You can customize the default convention for how views are located within the app by using a custom [ViewLocationExpander](#).

View discovery relies on finding view files by file name. If the underlying file system is case sensitive, view names are probably case sensitive. For compatibility across operating systems, match case between controller and action names and associated view folders and file names. If you encounter an error that a view file can't be found while working with a case-sensitive file system, confirm that the casing matches between the requested view file and the actual view file name.

Follow the best practice of organizing the file structure for your views to reflect the relationships among controllers, actions, and views for maintainability and clarity.

Passing data to views

You can pass data to views using several approaches. The most robust approach is to specify a [model](#) type in the view. This model is commonly referred to as a *viewmodel*. You pass an instance of the viewmodel type to the view from the action.

Using a viewmodel to pass data to a view allows the view to take advantage of *strong* type checking. *Strong typing* (or *strongly-typed*) means that every variable and constant has an explicitly defined type (for example, `string`, `int`, or `DateTime`). The validity of types used in a view is checked at compile time.

[Visual Studio](#) and [Visual Studio Code](#) list strongly-typed class members using a feature called [IntelliSense](#). When you want to see the properties of a viewmodel, type the variable name for the viewmodel followed by a period (`.`). This helps you write code faster with fewer errors.

Specify a model using the `@model` directive. Use the model with `@Model`:

```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
  @Model.Street<br>
  @Model.City, @Model.State @Model.PostalCode<br>
  <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

To provide the model to the view, the controller passes it as a parameter:

```

public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}

```

There are no restrictions on the model types that you can provide to a view. We recommend using **Plain Old CLR Object** (POCO) viewmodels with little or no behavior (methods) defined. Usually, viewmodel classes are either stored in the *Models* folder or a separate *ViewModels* folder at the root of the app. The *Address* viewmodel used in the example above is a POCO viewmodel stored in a file named *Address.cs*:

```

namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}

```

NOTE

Nothing prevents you from using the same classes for both your viewmodel types and your business model types. However, using separate models allows your views to vary independently from the business logic and data access parts of your app. Separation of models and viewmodels also offers security benefits when models use [model binding](#) and [validation](#) for data sent to the app by the user.

Weakly-typed data (ViewData and ViewBag)

Note: `ViewBag` is not available in the Razor Pages.

In addition to strongly-typed views, views have access to a *weakly-typed* (also called *loosely-typed*) collection of data. Unlike strong types, *weak types* (or *loose types*) means that you don't explicitly declare the type of data you're using. You can use the collection of weakly-typed data for passing small amounts of data in and out of controllers and views.

PASSING DATA BETWEEN A ...	EXAMPLE
Controller and a view	Populating a dropdown list with data.
View and a layout view	Setting the <title> element content in the layout view from a view file.

PASSING DATA BETWEEN A ...	EXAMPLE
Partial view and a view	A widget that displays data based on the webpage that the user requested.

This collection can be referenced through either the `ViewData` or `ViewBag` properties on controllers and views. The `ViewData` property is a dictionary of weakly-typed objects. The `ViewBag` property is a wrapper around `ViewData` that provides dynamic properties for the underlying `ViewData` collection.

`ViewData` and `ViewBag` are dynamically resolved at runtime. Since they don't offer compile-time type checking, both are generally more error-prone than using a viewmodel. For that reason, some developers prefer to minimally or never use `ViewData` and `ViewBag`.

ViewData

`ViewData` is a `ViewDataDictionary` object accessed through `string` keys. String data can be stored and used directly without the need for a cast, but you must cast other `ViewData` object values to specific types when you extract them. You can use `ViewData` to pass data from controllers to views and within views, including [partial views](#) and [layouts](#).

The following is an example that sets values for a greeting and an address using `ViewData` in an action:

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"] = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

Work with the data in a view:

```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```

ViewBag

Note: `ViewBag` is not available in the Razor Pages.

`ViewBag` is a `DynamicViewData` object that provides dynamic access to the objects stored in `ViewData`. `ViewBag` can be more convenient to work with, since it doesn't require casting. The following example shows how to use `ViewBag` with the same result as using `ViewData` above:

```

public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}

```

```

@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>

```

Using ViewData and ViewBag simultaneously

Note: `ViewBag` is not available in the Razor Pages.

Since `ViewData` and `ViewBag` refer to the same underlying `ViewData` collection, you can use both `ViewData` and `ViewBag` and mix and match between them when reading and writing values.

Set the title using `ViewBag` and the description using `ViewData` at the top of an *About.cshtml* view:

```

@{
    Layout = "/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Contoso";
    ViewData["Description"] = "Let us tell you about Contoso's philosophy and mission.";
}

```

Read the properties but reverse the use of `ViewData` and `ViewBag`. In the *_Layout.cshtml* file, obtain the title using `ViewData` and obtain the description using `ViewBag`:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"]</title>
    <meta name="description" content="@ViewBag.Description">
    ...

```

Remember that strings don't require a cast for `ViewData`. You can use `@ViewData["Title"]` without casting.

Using both `ViewData` and `ViewBag` at the same time works, as does mixing and matching reading and writing the properties. The following markup is rendered:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>About Contoso</title>
  <meta name="description" content="Let us tell you about Contoso's philosophy and mission.">
  ...

```

Summary of the differences between ViewData and ViewBag

`ViewBag` is not available in the Razor Pages.

- `ViewData`
 - Derives from `ViewDataDictionary`, so it has dictionary properties that can be useful, such as `ContainsKey`, `Add`, `Remove`, and `Clear`.
 - Keys in the dictionary are strings, so whitespace is allowed. Example:
`ViewData["Some Key With Whitespace"]`
 - Any type other than a `string` must be cast in the view to use `ViewData`.
- `ViewBag`
 - Derives from `DynamicViewData`, so it allows the creation of dynamic properties using dot notation (`@ViewBag.SomeKey = <value or object>`), and no casting is required. The syntax of `ViewBag` makes it quicker to add to controllers and views.
 - Simpler to check for null values. Example: `@ViewBag.Person?.Name`

When to use ViewData or ViewBag

Both `ViewData` and `ViewBag` are equally valid approaches for passing small amounts of data among controllers and views. The choice of which one to use is based on preference. You can mix and match `ViewData` and `ViewBag` objects, however, the code is easier to read and maintain with one approach used consistently. Both approaches are dynamically resolved at runtime and thus prone to causing runtime errors. Some development teams avoid them.

Dynamic views

Views that don't declare a model type using `@model` but that have a model instance passed to them (for example, `return View(Address);`) can reference the instance's properties dynamically:

```
<address>
  @Model.Street<br>
  @Model.City, @Model.State @Model.PostalCode<br>
  <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

This feature offers flexibility but doesn't offer compilation protection or IntelliSense. If the property doesn't exist, webpage generation fails at runtime.

More view features

[Tag Helpers](#) make it easy to add server-side behavior to existing HTML tags. Using Tag Helpers avoids the need to write custom code or helpers within your views. Tag helpers are applied as attributes to HTML elements and are ignored by editors that can't process them. This allows you to edit and render view markup in a variety of tools.

Generating custom HTML markup can be achieved with many built-in HTML Helpers. More complex user interface logic can be handled by [View Components](#). View components provide the same SoC that controllers and views offer. They can eliminate the need for actions and views that deal with data used by common user interface elements.

Like many other aspects of ASP.NET Core, views support [dependency injection](#), allowing services to be [injected into views](#).

Razor syntax for ASP.NET Core

11/1/2017 • 12 min to read • [Edit Online](#)

By [Rick Anderson](#), [Luke Latham](#), [Taylor Mullen](#), and [Dan Vicarel](#)

Razor is a markup syntax for embedding server-based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a *.cshtml* file extension.

Rendering HTML

The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in *.cshtml* Razor files is rendered by the server unchanged.

Razor syntax

Razor supports C# and uses the `@` symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output.

When an `@` symbol is followed by a [Razor reserved keyword](#), it transitions into Razor-specific markup. Otherwise, it transitions into plain C#.

To escape an `@` symbol in Razor markup, use a second `@` symbol:

```
<p>@@Username</p>
```

The code is rendered in HTML with a single `@` symbol:

```
<p>@Username</p>
```

HTML attributes and content containing email addresses don't treat the `@` symbol as a transition character. The email addresses in the following example are untouched by Razor parsing:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Implicit Razor expressions

Implicit Razor expressions start with `@` followed by C# code:

```
<p>@DateTime.Now</p>  
<p>@DateTime.IsLeapYear(2016)</p>
```

With the exception of the C# `await` keyword, implicit expressions must not contain spaces. If the C# statement has a clear ending, spaces can be intermingled:

```
<p>@await DoSomething("hello", "world")</p>
```

Implicit expressions **cannot** contain C# generics, as the characters inside the brackets (`<>`) are interpreted as an HTML tag. The following code is **not** valid:

```
<p>@GenericMethod<int>()</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element was not closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?

Generic method calls must be wrapped in an [explicit Razor expression](#) or a [Razor code block](#). This restriction doesn't apply to *.vbhtml* Razor files because Visual Basic syntax places parentheses around generic type parameters instead of brackets.

Explicit Razor expressions

Explicit Razor expressions consist of an `@` symbol with balanced parenthesis. To render last week's time, the following Razor markup is used:

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Any content within the `@()` parenthesis is evaluated and rendered to the output.

Implicit expressions, described in the previous section, generally can't contain spaces. In the following code, one week isn't subtracted from the current time:

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

The code renders the following HTML:

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

Explicit expressions can be used to concatenate text with an expression result:

```
@{
    var joe = new Person("Joe", 33);
}

<p>Age@(joe.Age)</p>
```

Without the explicit expression, `<p>Age@joe.Age</p>` is treated as an email address, and `<p>Age@joe.Age</p>` is rendered. When written as an explicit expression, `<p>Age33</p>` is rendered.

Explicit expressions can be used to render output from generic methods in *.cshtml* files. In an implicit expression, the characters inside the brackets (`<>`) are interpreted as an HTML tag. The following markup is **not** valid Razor:

```
<p>@GenericMethod<int>()</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element was not closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?

The following markup shows the correct way write this code. The code is written as an explicit expression:

```
<p>@(GenericMethod<int>())</p>
```

Note: this restriction doesn't apply to *.vbhtml* Razor files. With *.vbhtml* Razor files, Visual Basic syntax places parentheses around generic type parameters instead of brackets.

Expression encoding

C# expressions that evaluate to a string are HTML encoded. C# expressions that evaluate to `IHtmlContent` are rendered directly through `IHtmlContent.WriteTo`. C# expressions that don't evaluate to `IHtmlContent` are converted to a string by `ToString` and encoded before they're rendered.

```
@("<span>Hello World</span>")
```

The code renders the following HTML:

```
&lt;span&gt;Hello World&lt;/span&gt;
```

The HTML is shown in the browser as:

```
<span>Hello World</span>
```

`HtmlHelper.Raw` output isn't encoded but rendered as HTML markup.

WARNING

Using `HtmlHelper.Raw` on unsanitized user input is a security risk. User input might contain malicious JavaScript or other exploits. Sanitizing user input is difficult. Avoid using `HtmlHelper.Raw` with user input.

```
@Html.Raw("<span>Hello World</span>")
```

The code renders the following HTML:

```
<span>Hello World</span>
```

Razor code blocks

Razor code blocks start with `@` and are enclosed by `{ }`. Unlike expressions, C# code inside code blocks isn't rendered. Code blocks and expressions in a view share the same scope and are defined in order:

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";
}

<p>@quote</p>
```

The code renders the following HTML:

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.</p>
```

Implicit transitions

The default language in a code block is C#, but the Razor Page can transition back to HTML:

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Explicit delimited transition

To define a subsection of a code block that should render HTML, surround the characters for rendering with the Razor **<text>** tag:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

Use this approach to render HTML that isn't surrounded by an HTML tag. Without an HTML or Razor tag, a Razor runtime error occurs.

The **<text>** tag is useful to control whitespace when rendering content:

- Only the content between the **<text>** tag is rendered.
- No whitespace before or after the **<text>** tag appears in the HTML output.

Explicit Line Transition with @:

To render the rest of an entire line as HTML inside a code block, use the **@:** syntax:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

Without the **@:** in the code, a Razor runtime error is generated.

Warning: Extra **@** characters in a Razor file can cause compiler errors at statements later in the block. These compiler errors can be difficult to understand because the actual error occurs before the reported error. This error

is common after combining multiple implicit/explicit expressions into a single code block.

Control Structures

Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures:

Conditionals @if, else if, else, and @switch

`@if` controls when code runs:

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

`else` and `else if` don't require the `@` symbol:

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

The following markup shows how to use a switch statement:

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

Looping @for, @foreach, @while, and @do while

Templated HTML can be rendered with looping control statements. To render a list of people:

```
@{
    var people = new Person[]
    {
        new Person("Weston", 33),
        new Person("Johnathon", 41),
        ...
    };
}
```

The following looping statements are supported:

@for

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

@foreach

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

@while

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

@do while

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

Compound @using

In C#, a `using` statement is used to ensure an object is disposed. In Razor, the same mechanism is used to create HTML Helpers that contain additional content. In the following code, HTML Helpers render a form tag with the `@using` statement:

```
@using (Html.BeginForm())
{
    <div>
        email:
        <input type="email" id="Email" value="">
        <button>Register</button>
    </div>
}
```

Scope-level actions can be performed with [Tag Helpers](#).

@try, catch, finally

Exception handling is similar to C#:

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

@lock

Razor has the capability to protect critical sections with lock statements:

```
@lock (SomeLock)
{
    // Do critical section work
}
```

Comments

Razor supports C# and HTML comments:

```
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
```

The code renders the following HTML:

```
<!-- HTML comment -->
```

Razor comments are removed by the server before the webpage is rendered. Razor uses `@* *@` to delimit comments. The following code is commented out, so the server doesn't render any markup:

```
@*
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
*@
```

Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive typically changes the way a view is parsed or enables different functionality.

Understanding how Razor generates code for a view makes it easier to understand how directives work.

```
@{
    var quote = "Getting old ain't for wimps! - Anonymous";
}

<div>Quote of the Day: @quote</div>
```

The code generates a class similar to the following:

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

Later in this article, the section [Viewing the Razor C# class generated for a view](#) explains how to view this generated class.

@using

The `@using` directive adds the C# `using` directive to the generated view:

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

@model

The `@model` directive specifies the type of the model passed to a view:

```
@model TypeNameOfModel
```

In an ASP.NET Core MVC app created with individual user accounts, the *Views/Account/Login.cshtml* view contains the following model declaration:

```
@model LoginViewModel
```

The class generated inherits from `RazorPage<dynamic>`:

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor exposes a `Model` property for accessing the model passed to the view:

```
<div>The Login Email: @Model.Email</div>
```

The `@model` directive specifies the type of this property. The directive specifies the `T` in `RazorPage<T>` that the generated class that the view derives from. If the `@model` directive isn't specified, the `Model` property is of type `dynamic`. The value of the model is passed from the controller to the view. For more information, see [Strongly

typed models and the @model keyword.

@inherits

The @inherits directive provides full control of the class the view inherits:

```
@inherits TypeNameOfClassToInheritFrom
```

The following code is a custom Razor page type:

```
using Microsoft.AspNetCore.Mvc.Razor;  
  
public abstract class CustomRazorPage<TModel> : RazorPage<TModel>  
{  
    public string CustomText { get; } = "Gardylloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below."  
}
```

The CustomText is displayed in a view:

```
@inherits CustomRazorPage<TModel>  
  
<div>Custom text: @CustomText</div>
```

The code renders the following HTML:

```
<div>Custom text: Gardylloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below.</div>
```

@model and @inherits can be used in the same view. @inherits can be in a *_ViewImports.cshtml* file that the view imports:

```
@inherits CustomRazorPage<TModel>
```

The following code is an example of a strongly-typed view:

```
@inherits CustomRazorPage<TModel>  
  
<div>The Login Email: @Model.Email</div>  
<div>Custom text: @CustomText</div>
```

If "rick@contoso.com" is passed in the model, the view generates the following HTML markup:

```
<div>The Login Email: rick@contoso.com</div>  
<div>Custom text: Gardylloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below.</div>
```

@inject

The @inject directive enables the Razor Page to inject a service from the [service container](#) into a view. For more information, see [Dependency injection into views](#).

@functions

The @functions directive enables a Razor Page to add function-level content to a view:

```
@functions { // C# Code }
```

For example:

```
@functions {  
    public string GetHello()  
    {  
        return "Hello";  
    }  
}  
  
<div>From method: @GetHello(</div>
```

The code generates the following HTML markup:

```
<div>From method: Hello</div>
```

The following code is the generated Razor C# class:

```
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Mvc.Razor;  
  
public class _Views_Home_Test_cshtml : RazorPage<dynamic>  
{  
    // Functions placed between here  
    public string GetHello()  
    {  
        return "Hello";  
    }  
    // And here.  
    #pragma warning disable 1998  
    public override async Task ExecuteAsync()  
    {  
        WriteLiteral("\r\n<div>From method: ");  
        Write(GetHello());  
        WriteLiteral("</div>\r\n");  
    }  
    #pragma warning restore 1998  
}
```

@section

The `@section` directive is used in conjunction with the [layout](#) to enable views to render content in different parts of the HTML page. For more information, see [Sections](#).

Tag Helpers

There are three directives that pertain to [Tag Helpers](#).

DIRECTIVE	FUNCTION
@addTagHelper	Makes Tag Helpers available to a view.
@removeTagHelper	Removes Tag Helpers previously added from a view.
@tagHelperPrefix	Specifies a tag prefix to enable Tag Helper support and to make Tag Helper usage explicit.

Razor reserved keywords

Razor keywords

- page (Requires ASP.NET Core 2.0 and later)
- functions
- inherits
- model
- section
- helper (Not currently supported by ASP.NET Core)

Razor keywords are escaped with `@(Razor Keyword)` (for example, `@(functions)`).

C# Razor keywords

- case
- do
- default
- for
- foreach
- if
- else
- lock
- switch
- try
- catch
- finally
- using
- while

C# Razor keywords must be double-escaped with `@(@C# Razor Keyword)` (for example, `@(@case)`). The first `@` escapes the Razor parser. The second `@` escapes the C# parser.

Reserved keywords not used by Razor

- namespace
- class

Viewing the Razor C# class generated for a view

Add the following class to the ASP.NET Core MVC project:

```

using Microsoft.AspNetCore.Mvc.Razor.Extensions;
using Microsoft.AspNetCore.Mvc.Razor.Language;

public class CustomTemplateEngine : MvcRazorTemplateEngine
{
    public CustomTemplateEngine(RazorEngine engine, RazorProject project)
        : base(engine, project)
    {
    }

    public override RazorCSharpDocument GenerateCode(RazorCodeDocument codeDocument)
    {
        var csharpDocument = base.GenerateCode(codeDocument);
        var generatedCode = csharpDocument.GeneratedCode;

        // Look at generatedCode

        return csharpDocument;
    }
}

```

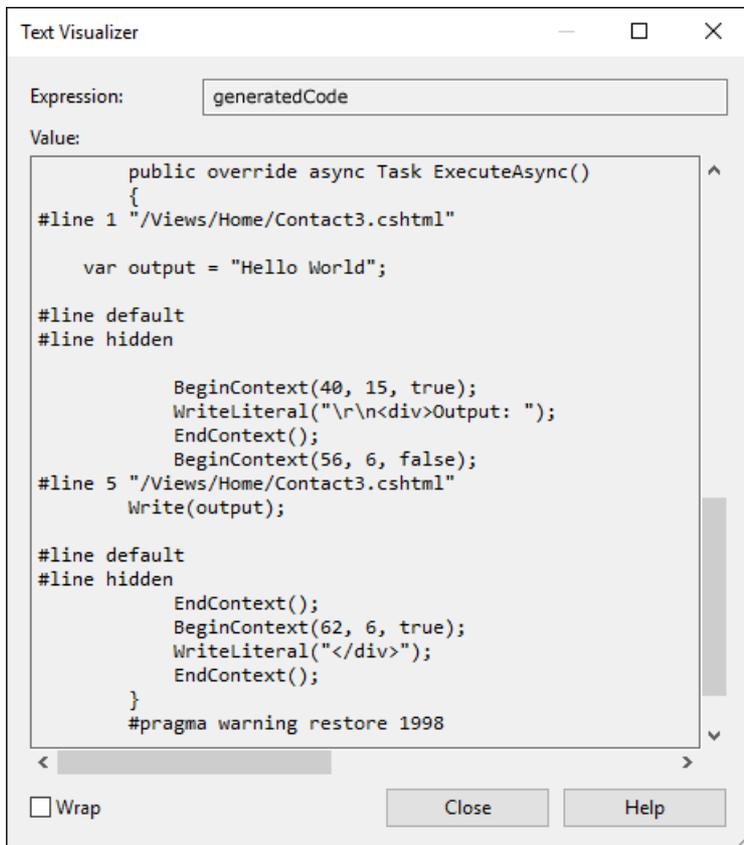
Override the `RazorTemplateEngine` added by MVC with the `CustomTemplateEngine` class:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSingleton<RazorTemplateEngine, CustomTemplateEngine>();
}

```

Set a break point on the `return csharpDocument` statement of `CustomTemplateEngine`. When program execution stops at the break point, view the value of `generatedCode`.



View lookups and case sensitivity

The Razor view engine performs case-sensitive lookups for views. However, the actual lookup is determined by the underlying file system:

- File based source:
 - On operating systems with case insensitive file systems (for example, Windows), physical file provider lookups are case insensitive. For example, `return View("Test")` results in matches for */Views/Home/Test.cshtml*, */Views/home/test.cshtml*, and any other casing variant.
 - On case-sensitive file systems (for example, Linux, OSX, and with `EmbeddedFileProvider`), lookups are case-sensitive. For example, `return View("Test")` specifically matches */Views/Home/Test.cshtml*.
- Precompiled views: With ASP.NET Core 2.0 and later, looking up precompiled views is case insensitive on all operating systems. The behavior is identical to physical file provider's behavior on Windows. If two precompiled views differ only in case, the result of lookup is non-deterministic.

Developers are encouraged to match the casing of file and directory names to the casing of:

- * Area, controller, and action names.
- * Razor Pages.

Matching case ensures the deployments find their views regardless of the underlying file system.

Razor view compilation and precompilation in ASP.NET Core

12/13/2017 • 1 min to read • [Edit Online](#)

By [Rick Anderson](#)

Razor views are compiled at runtime when the view is invoked. ASP.NET Core 1.1.0 and higher can optionally compile Razor views and deploy them with the app—a process known as precompilation. The ASP.NET Core 2.x project templates enable precompilation by default.

IMPORTANT

Razor view precompilation is currently unavailable when performing a [self-contained deployment \(SCD\)](#) in ASP.NET Core 2.0. The feature will be available for SCDs when 2.1 releases. For more information, see [View compilation fails when cross-compiling for Linux on Windows](#).

Precompilation considerations:

- Precompiling views results in a smaller published bundle and faster startup time.
- You can't edit Razor files after you precompile views. The edited views won't be present in the published bundle.

To deploy precompiled views:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

If your project targets .NET Framework, include a package reference to [Microsoft.AspNetCore.Mvc.Razor.ViewCompilation](#):

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0" PrivateAssets="All" />
```

If your project targets .NET Core, no changes are necessary.

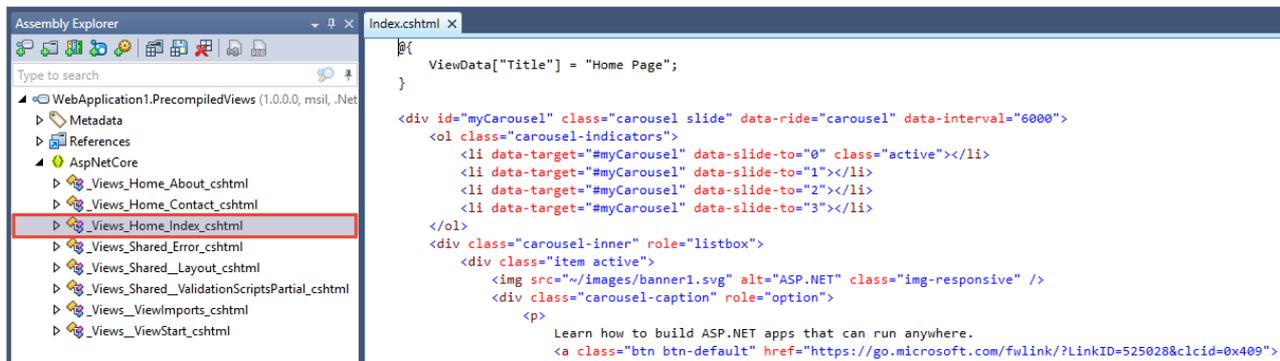
The ASP.NET Core 2.x project templates implicitly set `MvcRazorCompileOnPublish` to `true` by default, which means this node can be safely removed from the `.csproj` file. If you prefer to be explicit, there's no harm in setting the `MvcRazorCompileOnPublish` property to `true`. The following `.csproj` sample highlights this setting:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>
</Project>
```

Prepare the app for a [framework-dependent deployment](#) by executing a command such as the following at the project root:

```
dotnet publish -c Release
```

A `<project_name>.PrecompiledViews.dll` file, containing the compiled Razor views, is produced when precompilation succeeds. For example, the screenshot below depicts the contents of `Index.cshtml` inside of `WebApplication1.PrecompiledViews.dll`:



Layout

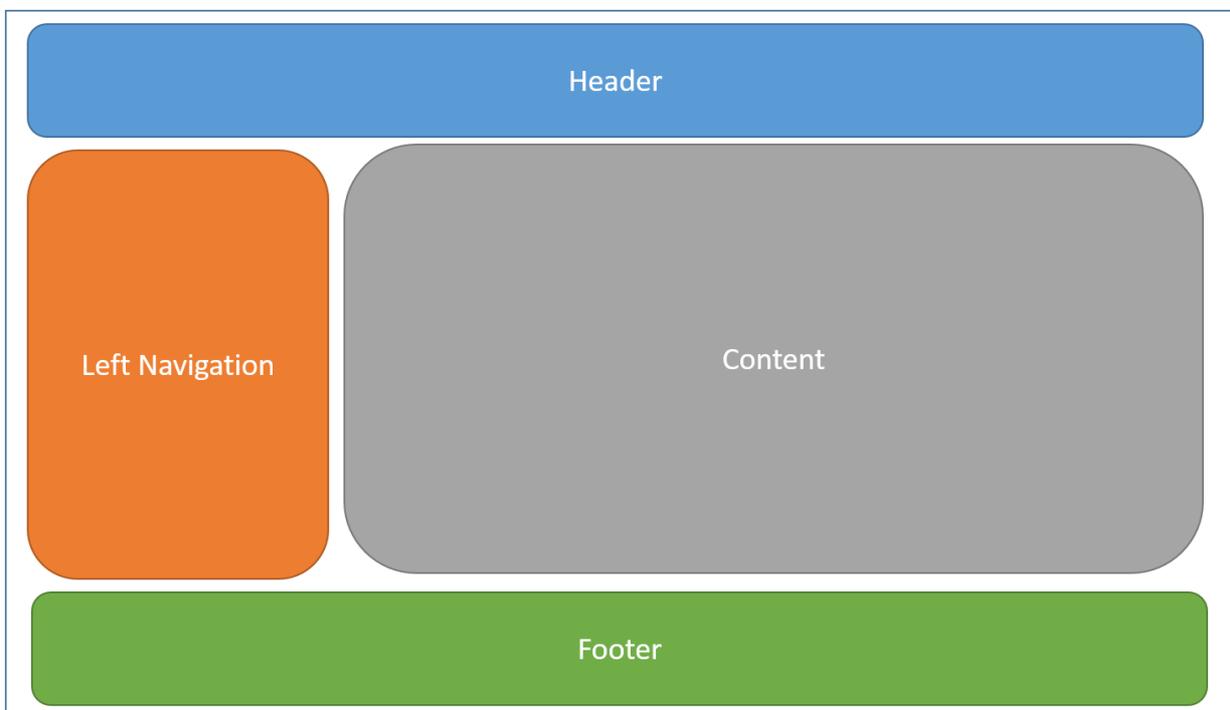
10/13/2017 • 5 min to read • [Edit Online](#)

By [Steve Smith](#)

Views frequently share visual and programmatic elements. In this article, you'll learn how to use common layouts, share directives, and run common code before rendering views in your ASP.NET app.

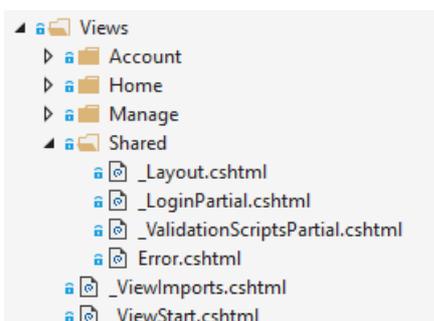
What is a Layout

Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.



Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a *layout* file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views, helping them follow the [Don't Repeat Yourself \(DRY\) principle](#).

By convention, the default layout for an ASP.NET app is named `_Layout.cshtml`. The Visual Studio ASP.NET Core MVC project template includes this layout file in the `Views/Shared` folder:



This layout defines a top level template for views in the app. Apps do not require a layout, and apps can define

more than one layout, with different views specifying different layouts.

An example `_Layout.cshtml` :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - WebApplication1</title>

  <environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment names="Staging,Production">
    <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css"
asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>
</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">

          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-
brand">WebApplication1</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
          <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
          <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
        </ul>
        @await Html.PartialAsync("_LoginPartial")
      </div>
    </div>
  </div>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; 2016 - WebApplication1</p>
    </footer>
  </div>

  <environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
  </environment>
  <environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
asp-fallback-test="window.jQuery">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js">
    </script>
  </environment>
</body>
</html>
```

```

        asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

    @RenderSection("scripts", required: false)
</body>
</html>

```

Specifying a Layout

Razor views have a `Layout` property. Individual views specify a layout by setting this property:

```

@{
    Layout = "_Layout";
}

```

The layout specified can use a full path (example: `/Views/Shared/_Layout.cshtml`) or a partial name (example: `_Layout`). When a partial name is provided, the Razor view engine will search for the layout file using its standard discovery process. The controller-associated folder is searched first, followed by the `Shared` folder. This discovery process is identical to the one used to discover [partial views](#).

By default, every layout must call `RenderBody`. Wherever the call to `RenderBody` is placed, the contents of the view will be rendered.

Sections

A layout can optionally reference one or more *sections*, by calling `RenderSection`. Sections provide a way to organize where certain page elements should be placed. Each call to `RenderSection` can specify whether that section is required or optional. If a required section is not found, an exception will be thrown. Individual views specify the content to be rendered within a section using the `@section` Razor syntax. If a view defines a section, it must be rendered (or an error will occur).

An example `@section` definition in a view:

```

@section Scripts {
    <script type="text/javascript" src="/scripts/main.js"></script>
}

```

In the code above, validation scripts are added to the `scripts` section on a view that includes a form. Other views in the same application might not require any additional scripts, and so wouldn't need to define a scripts section.

Sections defined in a view are available only in its immediate layout page. They cannot be referenced from partials, view components, or other parts of the view system.

Ignoring sections

By default, the body and all sections in a content page must all be rendered by the layout page. The Razor view engine enforces this by tracking whether the body and each section have been rendered.

To instruct the view engine to ignore the body or sections, call the `IgnoreBody` and `IgnoreSection` methods.

The body and every section in a Razor page must be either rendered or ignored.

Importing Shared Directives

Views can use Razor directives to do many things, such as importing namespaces or performing [dependency](#)

injection. Directives shared by many views may be specified in a common `_ViewImports.cshtml` file. The `_ViewImports` file supports the following directives:

- `@addTagHelper`
- `@removeTagHelper`
- `@tagHelperPrefix`
- `@using`
- `@model`
- `@inherits`
- `@inject`

The file does not support other Razor features, such as functions and section definitions.

A sample `_ViewImports.cshtml` file:

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The `_ViewImports.cshtml` file for an ASP.NET Core MVC app is typically placed in the `Views` folder. A `_ViewImports.cshtml` file can be placed within any folder, in which case it will only be applied to views within that folder and its subfolders. `_ViewImports` files are processed starting at the root level, and then for each folder leading up to the location of the view itself, so settings specified at the root level may be overridden at the folder level.

For example, if a root level `_ViewImports.cshtml` file specifies `@model` and `@addTagHelper`, and another `_ViewImports.cshtml` file in the controller-associated folder of the view specifies a different `@model` and adds another `@addTagHelper`, the view will have access to both tag helpers and will use the latter `@model`.

If multiple `_ViewImports.cshtml` files are run for a view, combined behavior of the directives included in the `ViewImports.cshtml` files will be as follows:

- `@addTagHelper`, `@removeTagHelper`: all run, in order
- `@tagHelperPrefix`: the closest one to the view overrides any others
- `@model`: the closest one to the view overrides any others
- `@inherits`: the closest one to the view overrides any others
- `@using`: all are included; duplicates are ignored
- `@inject`: for each property, the closest one to the view overrides any others with the same property name

Running Code Before Each View

If you have code you need to run before every view, this should be placed in the `_ViewStart.cshtml` file. By convention, the `_ViewStart.cshtml` file is located in the `Views` folder. The statements listed in `_ViewStart.cshtml` are run before every full view (not layouts, and not partial views). Like `ViewImports.cshtml`,

`_ViewStart.cshtml` is hierarchical. If a `_ViewStart.cshtml` file is defined in the controller-associated view folder, it will be run after the one defined in the root of the `Views` folder (if any).

A sample `_ViewStart.cshtml` file:

```
@{
    Layout = "_Layout";
}
```

The file above specifies that all views will use the `_Layout.cshtml` layout.

NOTE

Neither `_ViewStart.cshtml` nor `_ViewImports.cshtml` are typically placed in the `/Views/Shared` folder. The app-level versions of these files should be placed directly in the `/Views` folder.

Introduction to Tag Helpers in ASP.NET Core

10/31/2017 • 11 min to read • [Edit Online](#)

By [Rick Anderson](#)

What are Tag Helpers?

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. For example, the built-in `ImageTagHelper` can append a version number to the image name. Whenever the image changes, the server generates a new unique version for the image, so clients are guaranteed to get the current image (instead of a stale cached image). There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LabelTagHelper` can target the HTML `<label>` element when the `LabelTagHelper` attributes are applied. If you're familiar with [HTML Helpers](#), Tag Helpers reduce the explicit transitions between HTML and C# in Razor views. In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers do not replace HTML Helpers and there is not a Tag Helper for each HTML Helper. [Tag Helpers compared to HTML Helpers](#) explains the differences in more detail.

What Tag Helpers provide

An HTML-friendly development experience For the most part, Razor markup using Tag Helpers looks like standard HTML. Front-end designers conversant with HTML/CSS/JavaScript can edit Razor without learning C# Razor syntax.

A rich IntelliSense environment for creating HTML and Razor markup This is in sharp contrast to HTML Helpers, the previous approach to server-side creation of markup in Razor views. [Tag Helpers compared to HTML Helpers](#) explains the differences in more detail. [IntelliSense support for Tag Helpers](#) explains the IntelliSense environment. Even developers experienced with Razor C# syntax are more productive using Tag Helpers than writing C# Razor markup.

A way to make you more productive and able to produce more robust, reliable, and maintainable code using information only available on the server For example, historically the mantra on updating images was to change the name of the image when you change the image. Images should be aggressively cached for performance reasons, and unless you change the name of an image, you risk clients getting a stale copy. Historically, after an image was edited, the name had to be changed and each reference to the image in the web app needed to be updated. Not only is this very labor intensive, it's also error prone (you could miss a reference, accidentally enter the wrong string, etc.) The built-in `ImageTagHelper` can do this for you automatically. The `ImageTagHelper` can append a version number to the image name, so whenever the image changes, the server automatically generates a new unique version for the image. Clients are guaranteed to get the current image. This robustness and labor savings comes essentially free by using the `ImageTagHelper`.

Most of the built-in Tag Helpers target existing HTML elements and provide server-side attributes for the element. For example, the `<input>` element used in many of the views in the `Views/Account` folder contains the `asp-for` attribute, which extracts the name of the specified model property into the rendered HTML. The following Razor markup:

```
<label asp-for="Email"></label>
```

Generates the following HTML:

```
<label for="Email">Email</label>
```

The `asp-for` attribute is made available by the `For` property in the `LabelTagHelper`. See [Authoring Tag Helpers](#) for more information.

Managing Tag Helper scope

Tag Helpers scope is controlled by a combination of `@addTagHelper`, `@removeTagHelper`, and the "!" opt-out character.

`@addTagHelper` makes Tag Helpers available

If you create a new ASP.NET Core web app named *AuthoringTagHelpers* (with no authentication), the following *Views/_ViewImports.cshtml* file will be added to your project:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

The `@addTagHelper` directive makes Tag Helpers available to the view. In this case, the view file is *Views/_ViewImports.cshtml*, which by default is inherited by all view files in the *Views* folder and sub-directories; making Tag Helpers available. The code above uses the wildcard syntax ("*") to specify that all Tag Helpers in the specified assembly (*Microsoft.AspNetCore.Mvc.TagHelpers*) will be available to every view file in the *Views* directory or sub-directory. The first parameter after `@addTagHelper` specifies the Tag Helpers to load (we are using "*" for all Tag Helpers), and the second parameter "Microsoft.AspNetCore.Mvc.TagHelpers" specifies the assembly containing the Tag Helpers. *Microsoft.AspNetCore.Mvc.TagHelpers* is the assembly for the built-in ASP.NET Core Tag Helpers.

To expose all of the Tag Helpers in this project (which creates an assembly named *AuthoringTagHelpers*), you would use the following:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

If your project contains an `EmailTagHelper` with the default namespace (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), you can provide the fully qualified name (FQN) of the Tag Helper:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

To add a Tag Helper to a view using an FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (*AuthoringTagHelpers*). Most developers prefer to use the "*" wildcard syntax. The wildcard syntax allows you to insert the wildcard character "*" as the suffix in an FQN. For example, any of the following directives will bring in the `EmailTagHelper`:

```
@addTagHelper AuthoringTagHelpers.TagHelpers.E*, AuthoringTagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.Email*, AuthoringTagHelpers
```

As mentioned previously, adding the `@addTagHelper` directive to the `Views/_ViewImports.cshtml` file makes the Tag Helper available to all view files in the `Views` directory and sub-directories. You can use the `@addTagHelper` directive in specific view files if you want to opt-in to exposing the Tag Helper to only those views.

`@removeTagHelper` removes Tag Helpers

The `@removeTagHelper` has the same two parameters as `@addTagHelper`, and it removes a Tag Helper that was previously added. For example, `@removeTagHelper` applied to a specific view removes the specified Tag Helper from the view. Using `@removeTagHelper` in a `Views/Folder/_ViewImports.cshtml` file removes the specified Tag Helper from all of the views in `Folder`.

Controlling Tag Helper scope with the `_ViewImports.cshtml` file

You can add a `_ViewImports.cshtml` to any view folder, and the view engine applies the directives from both that file and the `Views/_ViewImports.cshtml` file. If you added an empty `Views/Home/_ViewImports.cshtml` file for the `Home` views, there would be no change because the `_ViewImports.cshtml` file is additive. Any `@addTagHelper` directives you add to the `Views/Home/_ViewImports.cshtml` file (that are not in the default `Views/_ViewImports.cshtml` file) would expose those Tag Helpers to views only in the `Home` folder.

Opting out of individual elements

You can disable a Tag Helper at the element level with the Tag Helper opt-out character ("!"). For example, `Email` validation is disabled in the `` with the Tag Helper opt-out character:

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

You must apply the Tag Helper opt-out character to the opening and closing tag. (The Visual Studio editor automatically adds the opt-out character to the closing tag when you add one to the opening tag). After you add the opt-out character, the element and Tag Helper attributes are no longer displayed in a distinctive font.

Using `@tagHelperPrefix` to make Tag Helper usage explicit

The `@tagHelperPrefix` directive allows you to specify a tag prefix string to enable Tag Helper support and to make Tag Helper usage explicit. For example, you could add the following markup to the `Views/_ViewImports.cshtml` file:

```
@tagHelperPrefix th:
```

In the code image below, the Tag Helper prefix is set to `th:`, so only those elements using the prefix `th:` support Tag Helpers (Tag Helper-enabled elements have a distinctive font). The `<label>` and `<input>` elements have the Tag Helper prefix and are Tag Helper-enabled, while the `` element does not.

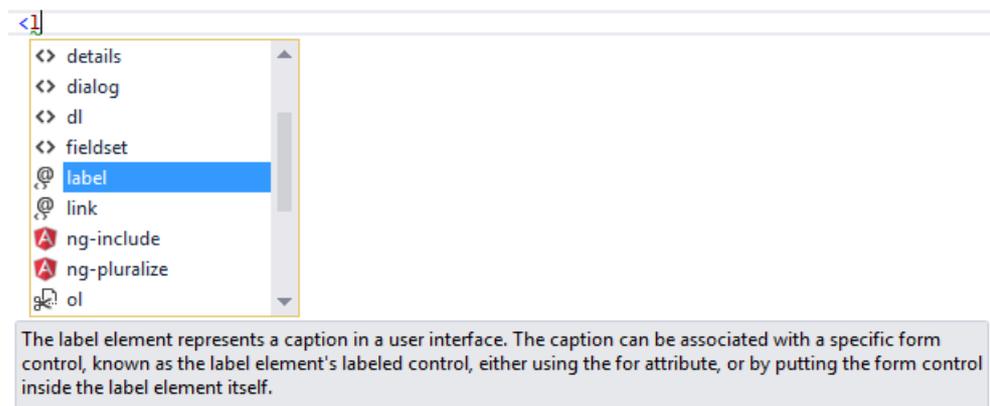
```
<div class="form-group">
  <th:label asp-for="Password" class="col-md-2"></th:label>
  <div class="col-md-10">
    <th:input asp-for="Password" class="form-control" />
    <span asp-validation-for="Password" class="text-danger"></span>
  </div>
</div>
```

The same hierarchy rules that apply to `@addTagHelper` also apply to `@tagHelperPrefix`.

IntelliSense support for Tag Helpers

When you create a new ASP.NET web app in Visual Studio, it adds the NuGet package "Microsoft.AspNetCore.Razor.Tools". This is the package that adds Tag Helper tooling.

Consider writing an HTML `<label>` element. As soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



Not only do you get HTML help, but the icon (the "@" symbol with "<>" under it).

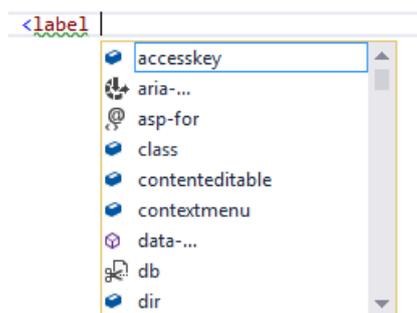


identifies the element as targeted by Tag Helpers. Pure HTML elements (such as the `fieldset`) display the "<>" icon.

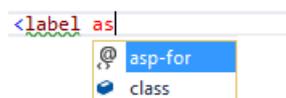
A pure HTML `<label>` tag displays the HTML tag (with the default Visual Studio color theme) in a brown font, the attributes in red, and the attribute values in blue.

```
<label class="col-md-2">Email</label>
```

After you enter `<label`, IntelliSense lists the available HTML/CSS attributes and the Tag Helper-targeted attributes:



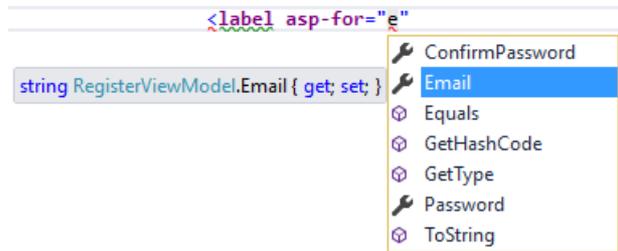
IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:



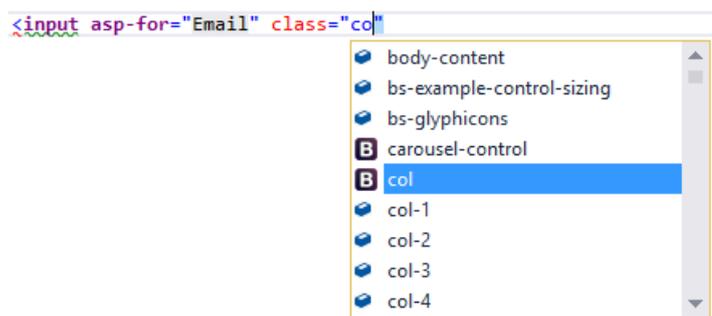
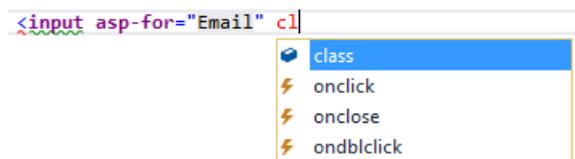
As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio "Blue" or "Light" color theme, the font is bold purple. If you're using the "Dark" theme the font is bold teal. The images in this document were taken using the default theme.

```
<label asp-for
```

You can enter the Visual Studio *CompleteWord* shortcut (Ctrl +spacebar is the [default](#) inside the double quotes ("")), and you are now in C#, just like you would be in a C# class. IntelliSense displays all the methods and properties on the page model. The methods and properties are available because the property type is `ModelExpression`. In the image below, I'm editing the `Register` view, so the `RegisterViewModel` is available.



IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:



Tag Helpers compared to HTML Helpers

Tag Helpers attach to HTML elements in Razor views, while [HTML Helpers](#) are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class "caption":

```
@Html.Label("FirstName", "First Name:", new { @class="caption" })
```

The `@` symbol tells Razor this is the start of code. The next two parameters ("FirstName" and "First Name:") are strings, so [IntelliSense](#) can't help. The last argument:

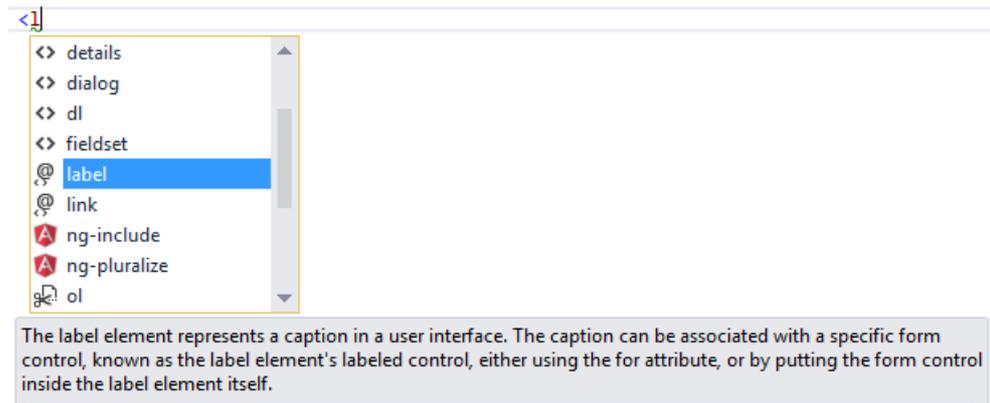
```
new { @class="caption" }
```

Is an anonymous object used to represent attributes. Because `class` is a reserved keyword in C#, you use the `@` symbol to force C# to interpret "`@class="`" as a symbol (property name). To a front-end designer (someone familiar with HTML/CSS/JavaScript and other client technologies but not familiar with C# and Razor), most of the line is foreign. The entire line must be authored with no help from IntelliSense.

Using the `LabelTagHelper`, the same markup can be written as:

```
<label class="caption" asp-for="FirstName"></label>
```

With the Tag Helper version, as soon as you enter `<1` in the Visual Studio editor, IntelliSense displays matching elements:



IntelliSense helps you write the entire line. The `LabelTagHelper` also defaults to setting the content of the `asp-for` attribute value ("FirstName") to "First Name"; It converts camel-cased properties to a sentence composed of the property name with a space where each new upper-case letter occurs. In the following markup:

```
<label class="caption" asp-for="FirstName"></label>
```

generates:

```
<label class="caption" for="FirstName">First Name</label>
```

The camel-cased to sentence-cased content is not used if you add content to the `<label>`. For example:

```
<label class="caption" asp-for="FirstName">Name First</label>
```

generates:

```
<label class="caption" for="FirstName">Name First</label>
```

The following code image shows the Form portion of the *Views/Account/Register.cshtml* Razor view generated from the legacy ASP.NET 4.5.x MVC template included with Visual Studio 2015.

```

@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}

```

The Visual Studio editor displays C# code with a grey background. For example, the `AntiForgeryToken` HTML Helper:

```
@Html.AntiForgeryToken()
```

is displayed with a grey background. Most of the markup in the Register view is C#. Compare that to the equivalent approach using Tag Helpers:

```

<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>

```

The markup is much cleaner and easier to read, edit, and maintain than the HTML Helpers approach. The C# code is reduced to the minimum that the server needs to know about. The Visual Studio editor displays markup targeted by a Tag Helper in a distinctive font.

Consider the *Email* group:

```
<div class="form-group">
  <label asp-for="Email" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="Email" class="form-control" />
    <span asp-validation-for="Email" class="text-danger"></span>
  </div>
</div>
```

Each of the "asp-" attributes has a value of "Email", but "Email" is not a string. In this context, "Email" is the C# model expression property for the `RegisterViewModel`.

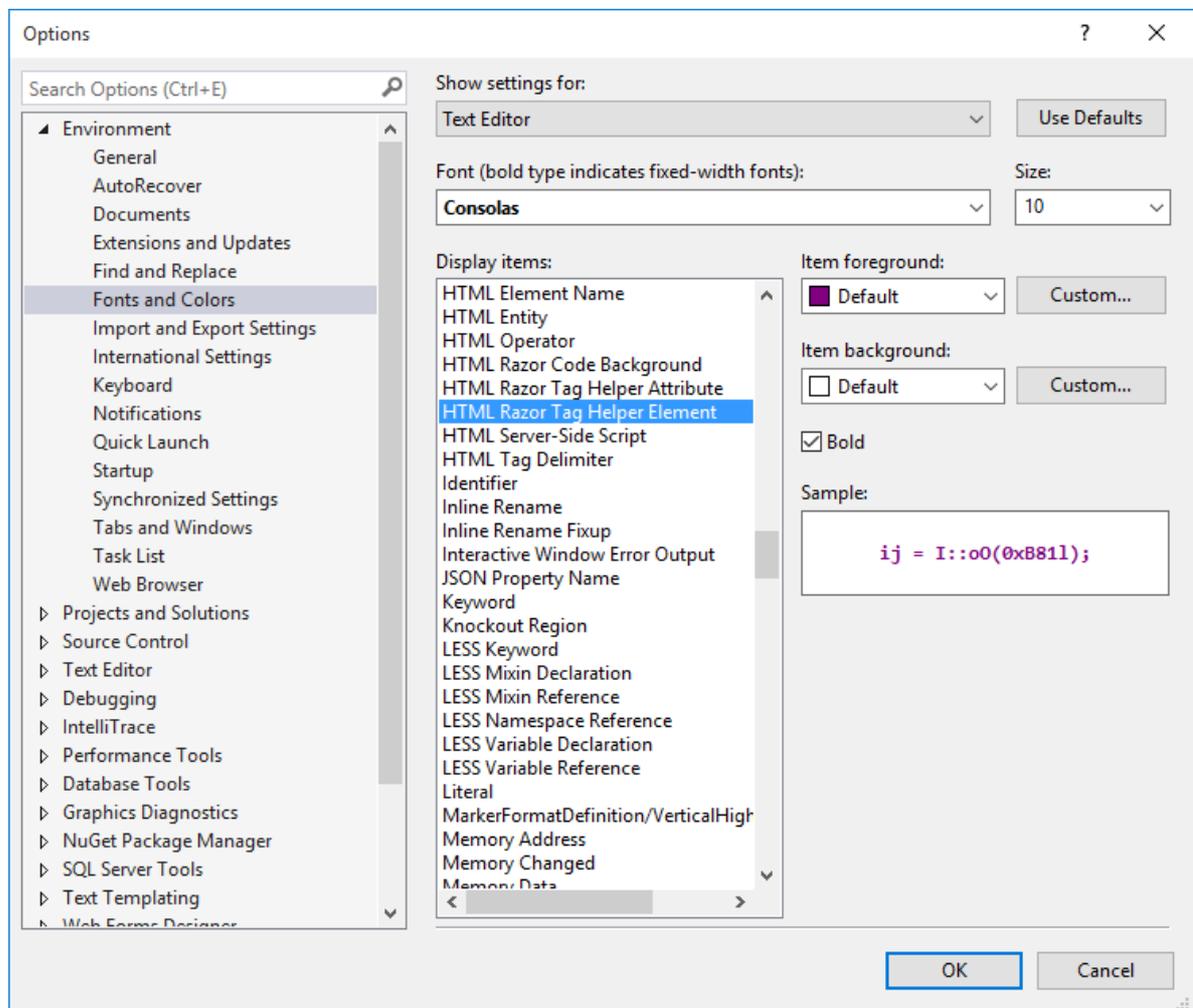
The Visual Studio editor helps you write **all** of the markup in the Tag Helper approach of the register form, while Visual Studio provides no help for most of the code in the HTML Helpers approach. [IntelliSense support for Tag Helpers](#) goes into detail on working with Tag Helpers in the Visual Studio editor.

Tag Helpers compared to Web Server Controls

- Tag Helpers don't own the element they're associated with; they simply participate in the rendering of the element and content. ASP.NET [Web Server controls](#) are declared and invoked on a page.
- [Web Server controls](#) have a non-trivial lifecycle that can make developing and debugging difficult.
- Web Server controls allow you to add functionality to the client Document Object Model (DOM) elements by using a client control. Tag Helpers have no DOM.
- Web Server controls include automatic browser detection. Tag Helpers have no knowledge of the browser.
- Multiple Tag Helpers can act on the same element (see [Avoiding Tag Helper conflicts](#)) while you typically can't compose Web Server controls.
- Tag Helpers can modify the tag and content of HTML elements that they're scoped to, but don't directly modify anything else on a page. Web Server controls have a less specific scope and can perform actions that affect other parts of your page; enabling unintended side effects.
- Web Server controls use type converters to convert strings into objects. With Tag Helpers, you work natively in C#, so you don't need to do type conversion.
- Web Server controls use [System.ComponentModel](#) to implement the run-time and design-time behavior of components and controls. `System.ComponentModel` includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components. Contrast that to Tag Helpers, which typically derive from `TagHelper`, and the `TagHelper` base class exposes only two methods, `Process` and `ProcessAsync`.

Customizing the Tag Helper element font

You can customize the font and colorization from **Tools > Options > Environment > Fonts and Colors:**



Additional Resources

- [Authoring Tag Helpers](#)
- [Working with Forms](#)
- [TagHelperSamples on GitHub](#) contains Tag Helper samples for working with [Bootstrap](#).

Authoring Tag Helpers in ASP.NET Core, a walkthrough with samples

12/19/2017 • 16 min to read • [Edit Online](#)

By [Rick Anderson](#)

[View or download sample code \(how to download\)](#)

Getting started with Tag Helpers

This tutorial provides an introduction to programming Tag Helpers. [Introduction to Tag Helpers](#) describes the benefits that Tag Helpers provide.

A tag helper is any class that implements the `ITagHelper` interface. However, when you author a tag helper, you generally derive from `TagHelper`, doing so gives you access to the `Process` method.

1. Create a new ASP.NET Core project called **AuthoringTagHelpers**. You won't need authentication for this project.
2. Create a folder to hold the Tag Helpers called *TagHelpers*. The *TagHelpers* folder is *not* required, but it is a reasonable convention. Now let's get started writing some simple tag helpers.

A minimal Tag Helper

In this section, you write a tag helper that updates an email tag. For example:

```
<email>Support</email>
```

The server will use our email tag helper to convert that markup into the following:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

That is, an anchor tag that makes this an email link. You might want to do this if you are writing a blog engine and need it to send email for marketing, support, and other contacts, all to the same domain.

1. Add the following `EmailTagHelper` class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

namespace AuthoringTagHelpers.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "a";    // Replaces <email> with <a> tag
        }
    }
}
```

Notes:

- Tag helpers use a naming convention that targets elements of the root class name (minus the *TagHelper* portion of the class name). In this example, the root name of **EmailTagHelper** is *email*, so the `<email>` tag will be targeted. This naming convention should work for most tag helpers, later on I'll show how to override it.
- The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides methods and properties for writing Tag Helpers.
- The overridden `Process` method controls what the tag helper does when executed. The `TagHelper` class also provides an asynchronous version (`ProcessAsync`) with the same parameters.
- The context parameter to `Process` (and `ProcessAsync`) contains information associated with the execution of the current HTML tag.
- The output parameter to `Process` (and `ProcessAsync`) contains a stateful HTML element representative of the original source used to generate an HTML tag and content.
- Our class name has a suffix of **TagHelper**, which is *not* required, but it's considered a best practice convention. You could declare the class as:

```
public class Email : TagHelper
```

2. To make the `EmailTagHelper` class available to all our Razor views, add the `addTagHelper` directive to the `Views/_ViewImports.cshtml` file:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

The code above uses the wildcard syntax to specify all the tag helpers in our assembly will be available. The first string after `@addTagHelper` specifies the tag helper to load (Use "*" for all tag helpers), and the second string "AuthoringTagHelpers" specifies the assembly the tag helper is in. Also, note that the second line brings in the ASP.NET Core MVC tag helpers using the wildcard syntax (those helpers are discussed in [Introduction to Tag Helpers](#).) It's the `@addTagHelper` directive that makes the tag helper available to the Razor view. Alternatively, you can provide the fully qualified name (FQN) of a tag helper as shown below:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

To add a tag helper to a view using a FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (`AuthoringTagHelpers`). Most developers will prefer to use the wildcard syntax. [Introduction to Tag Helpers](#) goes into detail on tag helper adding, removing, hierarchy, and wildcard syntax.

1. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

2. Run the app and use your favorite browser to view the HTML source so you can verify that the email tags are replaced with anchor markup (For example, `<a>Support`). *Support* and *Marketing* are rendered as links, but they don't have an `href` attribute to make them functional. We'll fix that in the next section.

Note: Like HTML tags and attributes, tags, class names and attributes in Razor, and C# are not case-sensitive.

SetAttribute and SetContent

In this section, we'll update the `EmailTagHelper` so that it will create a valid anchor tag for email. We'll update it to take information from a Razor view (in the form of a `mail-to` attribute) and use that in generating the anchor.

Update the `EmailTagHelper` class with the following:

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // Pascal case gets translated into lower-kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a"; // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + address);
        output.Content.SetContent(address);
    }
}
```

Notes:

- Pascal-cased class and property names for tag helpers are translated into their [lower kebab case](#). Therefore, to use the `MailTo` attribute, you'll use `<email mail-to="value"/>` equivalent.
- The last line sets the completed content for our minimally functional tag helper.
- The highlighted line shows the syntax for adding attributes:

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "a";    // Replaces <email> with <a> tag

    var address = MailTo + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
}
```

That approach works for the attribute "href" as long as it doesn't currently exist in the attributes collection. You can also use the `output.Attributes.Add` method to add a tag helper attribute to the end of the collection of tag attributes.

1. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

```
@{
    ViewData["Title"] = "Contact Copy";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way Copy Version <br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email mail-to="Support"></email><br />
    <strong>Marketing:</strong><email mail-to="Marketing"></email>
</address>
```

2. Run the app and verify that it generates the correct links.

NOTE

If you were to write the email tag self-closing (`<email mail-to="Rick" />`), the final output would also be self-closing. To enable the ability to write the tag with only a start tag (`<email mail-to="Rick">`) you must decorate the class with the following:

```
[HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
public class EmailVoidTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    // Code removed for brevity
}
```

With a self-closing email tag helper, the output would be ``. Self-closing anchor tags are not valid HTML, so you wouldn't want to create one, but you might want to create a tag helper that is self-closing. Tag helpers set the type of the `TagMode` property after reading a tag.

ProcessAsync

In this section, we'll write an asynchronous email helper.

1. Replace the `EmailTagHelper` class with the following code:

```

public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a"; // Replaces <email> with <a> tag
        var content = await output.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + target);
        output.Content.SetContent(target);
    }
}

```

Notes:

- This version uses the asynchronous `ProcessAsync` method. The asynchronous `GetChildContentAsync` returns a `Task` containing the `TagHelperContent`.
- Use the `output` parameter to get contents of the HTML element.

2. Make the following change to the `Views/Home/Contact.cshtml` file so the tag helper can get the target email.

```

@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

```

3. Run the app and verify that it generates valid email links.

RemoveAll, PreContent.SetHtmlContent and PostContent.SetHtmlContent

1. Add the following `BoldTagHelper` class to the `TagHelpers` folder.

```

using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = "bold")]
    public class BoldTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.Attributes.RemoveAll("bold");
            output.PreContent.SetHtmlContent("<strong>");
            output.PostContent.SetHtmlContent("</strong>");
        }
    }
}

```

Notes:

- The `[HtmlTargetElement]` attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named "bold" will match, and the `Process` override method in the class will run. In our sample, the `Process` method removes the "bold" attribute and surrounds the containing markup with ``.
- Because you don't want to replace the existing tag content, you must write the opening `` tag with the `PreContent.SetHtmlContent` method and the closing `` tag with the `PostContent.SetHtmlContent` method.

2. Modify the *About.cshtml* view to contain a `bold` attribute value. The completed code is shown below.

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>
```

3. Run the app. You can use your favorite browser to inspect the source and verify the markup.

The `[HtmlTargetElement]` attribute above only targets HTML markup that provides an attribute name of "bold". The `<bold>` element was not modified by the tag helper.

4. Comment out the `[HtmlTargetElement]` attribute line and it will default to targeting `<bold>` tags, that is, HTML markup of the form `<bold>`. Remember, the default naming convention will match the class name **BoldTagHelper** to `<bold>` tags.

5. Run the app and verify that the `<bold>` tag is processed by the tag helper.

Decorating a class with multiple `[HtmlTargetElement]` attributes results in a logical-OR of the targets. For example, using the code below, a bold tag or a bold attribute will match.

```
[HtmlTargetElement("bold")]
[HtmlTargetElement(Attributes = "bold")]
public class BoldTagHelper : TagHelper
{
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.Attributes.RemoveAll("bold");
        output.PreContent.SetHtmlContent("<strong>");
        output.PostContent.SetHtmlContent("</strong>");
    }
}
```

When multiple attributes are added to the same statement, the runtime treats them as a logical-AND. For example, in the code below, an HTML element must be named "bold" with an attribute named "bold" (`<bold bold />`) to match.

```
[HtmlTargetElement("bold", Attributes = "bold")]
```

You can also use the `[HtmlTargetElement]` to change the name of the targeted element. For example if you wanted the `BoldTagHelper` to target `<MyBold>` tags, you would use the following attribute:

```
[HtmlTargetElement("MyBold")]
```

Passing a model to a Tag Helper

1. Add a *Models* folder.
2. Add the following `WebsiteContext` class to the *Models* folder:

```
using System;

namespace AuthoringTagHelpers.Models
{
    public class WebsiteContext
    {
        public Version Version { get; set; }
        public int CopyrightYear { get; set; }
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

3. Add the following `WebsiteInformationTagHelper` class to the *TagHelpers* folder.

```
using System;
using AuthoringTagHelpers.Models;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
        public WebsiteContext Info { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "section";
            output.Content.SetHtmlContent(
                $"<ul><li><strong>Version:</strong> {Info.Version}</li>
                <li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
                <li><strong>Approved:</strong> {Info.Approved}</li>
                <li><strong>Number of tags to show:</strong> {Info.TagsToShow}</li></ul>");
            output.TagMode = TagMode.StartTagAndEndTag;
        }
    }
}
```

Notes:

- As mentioned previously, tag helpers translates Pascal-cased C# class names and properties for tag helpers into [lower kebab case](#). Therefore, to use the `WebsiteInformationTagHelper` in Razor, you'll write `<website-information />`.
- You are not explicitly identifying the target element with the `[HtmlTargetElement]` attribute, so the default of `website-information` will be targeted. If you applied the following attribute (note it's not kebab case but matches the class name):

```
[HtmlTargetElement("WebsiteInformation")]
```

The lower kebab case tag `<website-information />` would not match. If you want use the `[HtmlTargetElement]` attribute, you would use kebab case as shown below:

```
[HtmlTargetElement("Website-Information")]
```

- Elements that are self-closing have no content. For this example, the Razor markup will use a self-closing tag, but the tag helper will be creating a `section` element (which is not self-closing and you are writing content inside the `section` element). Therefore, you need to set `TagMode` to `StartTagAndEndTag` to write output. Alternatively, you can comment out the line setting `TagMode` and write markup with a closing tag. (Example markup is provided later in this tutorial.)
- The `$` (dollar sign) in the following line uses an [interpolated string](#):

```
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

4. Add the following markup to the `About.cshtml` view. The highlighted markup displays the web site information.

```
@using AuthoringTagHelpers.Models
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="new WebsiteContext {
    Version = new Version(1, 3),
    CopyrightYear = 1638,
    Approved = true,
    TagsToShow = 131 }" />
```

NOTE

In the Razor markup shown below:

```
<website-information info="new WebsiteContext {
    Version = new Version(1, 3),
    CopyrightYear = 1638,
    Approved = true,
    TagsToShow = 131 }" />
```

Razor knows the `info` attribute is a class, not a string, and you want to write C# code. Any non-string tag helper attribute should be written without the `@` character.

5. Run the app, and navigate to the About view to see the web site information.

NOTE

You can use the following markup with a closing tag and remove the line with `TagMode.StartTagAndEndTag` in the tag helper:

```
<website-information info="new WebsiteContext {
    Version = new Version(1, 3),
    CopyrightYear = 1638,
    Approved = true,
    TagsToShow = 131 }" >
</website-information>
```

Condition Tag Helper

The condition tag helper renders output when passed a true value.

1. Add the following `ConditionTagHelper` class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = nameof(Condition))]
    public class ConditionTagHelper : TagHelper
    {
        public bool Condition { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            if (!Condition)
            {
                output.SuppressOutput();
            }
        }
    }
}
```

2. Replace the contents of the *Views/Home/Index.cshtml* file with the following markup:

```
@using AuthoringTagHelpers.Models
@model WebsiteContext

@{
    ViewData["Title"] = "Home Page";
}

<div>
    <h3>Information about our website (outdated):</h3>
    <website-information info=@Model />
    <div condition="@Model.Approved">
        <p>
            This website has <strong surround="em"> @Model.Approved </strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>
```

3. Replace the `Index` method in the `Home` controller with the following code:

```

public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}

```

4. Run the app and browse to the home page. The markup in the conditional `div` will not be rendered. Append the query string `?approved=true` to the URL (for example, `http://localhost:1235/Home/Index?approved=true`). `approved` is set to true and the conditional markup will be displayed.

NOTE

Use the `nameof` operator to specify the attribute to target rather than specifying a string as you did with the bold tag helper:

```

[HtmlTargetElement(Attributes = nameof(Condition))]
// [HtmlTargetElement(Attributes = "condition")]
public class ConditionTagHelper : TagHelper
{
    public bool Condition { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        if (!Condition)
        {
            output.SuppressOutput();
        }
    }
}

```

The `nameof` operator will protect the code should it ever be refactored (we might want to change the name to `RedCondition`).

Avoiding Tag Helper conflicts

In this section, you write a pair of auto-linking tag helpers. The first will replace markup containing a URL starting with HTTP to an HTML anchor tag containing the same URL (and thus yielding a link to the URL). The second will do the same for a URL starting with WWW.

Because these two helpers are closely related and you may refactor them in the future, we'll keep them in the same file.

1. Add the following `AutoLinkerHttpTagHelper` class to the `TagHelpers` folder.

```
[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\b(?:https?://)(\S+)\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>")); // http link version
    }
}
```

NOTE

The `AutoLinkerHttpTagHelper` class targets `p` elements and uses [Regex](#) to create the anchor.

2. Add the following markup to the end of the `Views/Home/Contact.cshtml` file:

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

<p>Visit us at http://docs.asp.net or at www.microsoft.com</p>
```

3. Run the app and verify that the tag helper renders the anchor correctly.
4. Update the `AutoLinker` class to include the `AutoLinkerWwwTagHelper` which will convert `www` text to an anchor tag that also contains the original `www` text. The updated code is highlighted below:

```

[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\b(?:https?://)(\S+)\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>")); // http link version}
    }
}

[HtmlTargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\b(www\.\S+)\b",
            "<a target=\"_blank\" href=\"http://$0\">$0</a>")); // www version
    }
}
}

```

- Run the app. Notice the www text is rendered as a link but the HTTP text is not. If you put a break point in both classes, you can see that the HTTP tag helper class runs first. The problem is that the tag helper output is cached, and when the WWW tag helper is run, it overwrites the cached output from the HTTP tag helper. Later in the tutorial we'll see how to control the order that tag helpers run in. We'll fix the code with the following:

```

public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\b(?:https?://)(\S+)\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>")); // http link version)
    }
}

[HtmlTargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\b(www\.\S+)\b",
            "<a target=\"_blank\" href=\"http://$0\">$0</a>")); // www version
    }
}

```

NOTE

In the first edition of the auto-linking tag helpers, you got the content of the target with the following code:

```
var childContent = await output.GetChildContentAsync();
```

That is, you call `GetChildContentAsync` using the `TagHelperOutput` passed into the `ProcessAsync` method. As mentioned previously, because the output is cached, the last tag helper to run wins. You fixed that problem with the following code:

```
var childContent = output.Content.IsModified ? output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

The code above checks to see if the content has been modified, and if it has, it gets the content from the output buffer.

- Run the app and verify that the two links work as expected. While it might appear our auto linker tag helper is correct and complete, it has a subtle problem. If the WWW tag helper runs first, the www links will not be correct. Update the code by adding the `Order` overload to control the order that the tag runs in. The `Order` property determines the execution order relative to other tag helpers targeting the same element. The default order value is zero and instances with lower values are executed first.

```

public class AutoLinkerHttpTagHelper : TagHelper
{
    // This filter must run before the AutoLinkerWwwTagHelper as it searches and replaces http and
    // the AutoLinkerWwwTagHelper adds http to the markup.
    public override int Order
    {
        get { return int.MinValue; }
    }
}

```

The above code will guarantee that the HTTP tag helper runs before the WWW tag helper. Change `Order` to `MaxValue` and verify that the markup generated for the WWW tag is incorrect.

Inspecting and retrieving child content

The tag helpers provide several properties to retrieve content.

- The result of `GetChildContentAsync` can be appended to `output.Content`.
- You can inspect the result of `GetChildContentAsync` with `GetContent`.
- If you modify `output.Content`, the TagHelper body will not be executed or rendered unless you call `GetChildContentAsync` as in our auto-linker sample:

```

public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\"b(?:https?://)(\\S+)\"b",
            "<a target=\"_blank\" href=\"\\$0\">\\$0</a>")); // http link version
    }
}

```

- Multiple calls to `GetChildContentAsync` will return the same value and will not re-execute the `TagHelper` body unless you pass in a false parameter indicating not use the cached result.

Introduction to using tag helpers in forms in ASP.NET Core

10/13/2017 • 16 min to read • [Edit Online](#)

By [Rick Anderson](#), [Dave Paquette](#), and [Jerrie Pelsler](#)

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML `Form` element provides the primary mechanism web apps use to post back data to the server. Most of this document describes [Tag Helpers](#) and how they can help you productively create robust HTML forms. We recommend you read [Introduction to Tag Helpers](#) before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers do not replace HTML Helpers and there is not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it is mentioned.

The Form Tag Helper

The `Form` Tag Helper:

- Generates the HTML `<FORM>` `action` attribute value for a MVC controller action or named route
- Generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
  <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
  <!-- Input and Submit elements -->
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a

route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
  <!-- Input and Submit elements -->
</form>
```

Many of the views in the `Views/Account` folder (generated when you create a new web app with *Individual User Accounts*) contain the `asp-route-returnurl` attribute:

```
<form asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@ViewData["ReturnUrl"]"
      method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Input Tag Helper

The Input Tag Helper binds an HTML `<input>` element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>" />
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the [Expression names](#) section for additional information.
- Sets the HTML `type` attribute value based on the model type and [data annotation](#) attributes applied to the model property
- Will not overwrite the HTML `type` attribute value when one is specified
- Generates [HTML5](#) validation attributes from [data annotation](#) attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

An error occurred during the compilation of a resource required to process this request. Please review the following specific error details and modify your source code appropriately.

Type expected

'RegisterViewModel' does not contain a definition for 'Email' and no extension method 'Email' accepting a first argument of type 'RegisterViewModel' could be found (are you missing a using directive or an assembly reference?)

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common [data annotations](#) attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```

using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}

```

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email: <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>

```

The code above generates the following HTML:

```

<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
        data-val-email="The Email Address field is not a valid e-mail address."
        data-val-required="The Email Address field is required."
        id="Email" name="Email" value="" /> <br>
    Password:
    <input type="password" data-val="true"
        data-val-required="The Password field is required."
        id="Password" name="Password" /><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>

```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces [HTML5](#) `data-val-*` attributes (see [Model Validation](#)). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and [jQuery](#) validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc). If an error message is provided in the attribute, it is displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` will not. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper does not. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression

names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
    new { htmlAttributes = new { @class="myCssClass", style="width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
    var joe = "Joe";
}

```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe" />
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```

@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email: <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>

```

The following HTML is generated for `Address.AddressLine1` :

```

<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />

```

Expression names and Collections

Sample, a model containing an array of `Colors` :

```

public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}

```

The action method:

```

public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}

```

The following Razor shows how you access a specific `Color` element:

```

@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>

```

The `Views/Shared/EditorTemplates/String.cshtml` template:

```

@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />

```

Sample using `List<T>` :

```

public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}

```

The following Razor shows how to iterate over a collection:

```

@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>

```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```

@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
This template replaces the following Razor which evaluates the indexer three times.
<td>
    <label asp-for="@Model[i].Name"></label>
    @Html.DisplayFor(model => model[i].Name)
</td>
<td>
    <input asp-for="@Model[i].IsDone" />
</td>
*@

```

NOTE

Always use `for` (and *not* `foreach`) to iterate over a list. Evaluating an indexer in a LINQ expression can be expensive and should be minimized.

NOTE

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
        data-val-maxlength="The field Description must be a string or array type with a maximum length of
        &#x27;1024&#x27;."
        data-val-maxlength-max="1024"
        data-val-minlength="The field Description must be a string or array type with a minimum length of
        &#x27;5&#x27;."
        data-val-minlength-min="5"
        id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a element for an expression name
- HTML Helper alternative: `Html.LabelFor`

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the `span` element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, `jQuery` displays the error message in the `` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can

only be done on the server side.

- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML `span` element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
      data-valmsg-for="Email"
      data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

NOTE

You must have a view with the correct JavaScript and [jQuery](#) script references in place for client side validation. See [Model Validation](#) for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
  The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
<code>ValidationSummary.All</code>	Property and model level
<code>ValidationSummary.ModelOnly</code>	Model
<code>ValidationSummary.None</code>	None

Sample

In the following example, the data model is decorated with `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```

using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}

```

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email: <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

The generated HTML (when the model is valid):

```

<form action="/DemoReg/Register" method="post">
    <div class="validation-summary-valid" data-valmsg-summary="true">
    <ul><li style="display:none"></li></ul></div>
    Email: <input name="Email" id="Email" type="email" value=""
        data-val-required="The Email field is required."
        data-val-email="The Email field is not a valid e-mail address."
        data-val="true"> <br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Email"></span><br>
    Password: <input name="Password" id="Password" type="password"
        data-val-required="The Password field is required." data-val="true"><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Password"></span><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>

```

The Select Tag Helper

- Generates [select](#) and associated [option](#) elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the [select](#) element and `asp-items` specifies the [option](#) elements. For example:

```

<select asp-for="Country" asp-items="Model.Countries"></select>

```

Sample:

```

using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}

```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```

public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}

```

The HTTP POST `Index` method displays the selection:

```

[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}

```

The `Index` view:

```

@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>

```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
  <select id="Country" name="Country">
    <option value="MX">Mexico</option>
    <option selected="selected" value="CA">Canada</option>
    <option value="US">USA</option>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity"> />
</form>
```

NOTE

We do not recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
  <select asp-for="EnumCountry"
    asp-items="Html.GetEnumSelectList<CountryEnum>()"> >
  </select>
  <br /><button type="submit">Register</button>
</form>
```

You can decorate your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
  public enum CountryEnum
  {
    [Display(Name = "United Mexican States")]
    Mexico,
    [Display(Name = "United States of America")]
    USA,
    Canada,
    France,
    Germany,
    Spain
  }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
  <select data-val="true" data-val-required="The EnumCountry field is required."
    id="EnumCountry" name="EnumCountry">
    <option value="0">United Mexican States</option>
    <option value="1">United States of America</option>
    <option value="2">Canada</option>
    <option value="3">France</option>
    <option value="4">Germany</option>
    <option selected="selected" value="5">Spain</option>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

Option Group

The HTML `<optgroup>` element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

```

public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

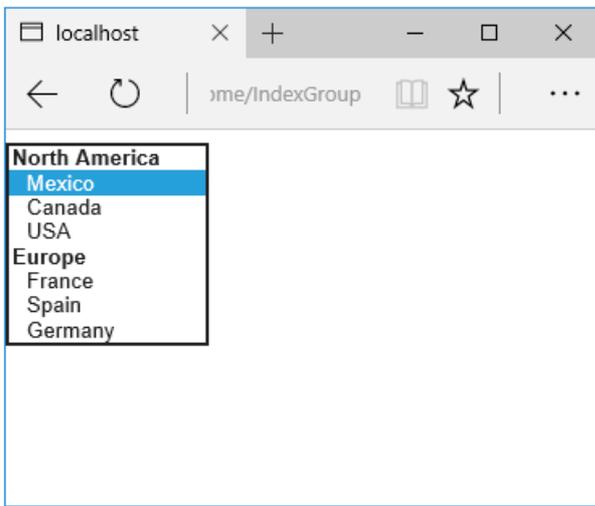
        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
}

```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
  <select id="Country" name="Country">
    <optgroup label="North America">
      <option value="MEX">Mexico</option>
      <option value="CAN">Canada</option>
      <option value="US">USA</option>
    </optgroup>
    <optgroup label="Europe">
      <option value="FR">France</option>
      <option value="ES">Spain</option>
      <option value="DE">Germany</option>
    </optgroup>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

Multiple select

The Select Tag Helper will automatically generate the `multiple = "multiple"` attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
  public class CountryViewModelIEnumerable
  {
    public IEnumerable<string> CountryCodes { get; set; }

    public List<SelectListItem> Countries { get; } = new List<SelectListItem>
    {
      new SelectListItem { Value = "MX", Text = "Mexico" },
      new SelectListItem { Value = "CA", Text = "Canada" },
      new SelectListItem { Value = "US", Text = "USA" },
      new SelectListItem { Value = "FR", Text = "France" },
      new SelectListItem { Value = "ES", Text = "Spain" },
      new SelectListItem { Value = "DE", Text = "Germany" }
    };
  }
}
```

With the following view:

```
@model CountryViewModelIEnumerableable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
  <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
  <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
  <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
  <option value="CA">Canada</option>
  <option value="US">USA</option>
  <option value="FR">France</option>
  <option value="ES">Spain</option>
  <option value="DE">Germany</option>
</select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
  @Html.EditorForModel()
  <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
  <option value="">--none--</option>
</select>
```

Adding HTML [<option>](#) elements is not limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexOption(int id)
{
  var model = new CountryViewModel();
  model.Country = "CA";
  return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
  <select asp-for="Country">
    <option value="">&lt;none&gt;</option>
    <option value="MX">Mexico</option>
    <option value="CA">Canada</option>
    <option value="US">USA</option>
  </select>
  <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
<form method="post" action="/Home/IndexEmpty">
  <select id="Country" name="Country">
    <option value="">&lt;none&gt;</option>
    <option value="MX">Mexico</option>
    <option value="CA" selected="selected">Canada</option>
    <option value="US">USA</option>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

Additional Resources

- [Tag Helpers](#)
- [HTML Form element](#)
- [Request Verification Token](#)
- [Model Binding](#)
- [Model Validation](#)
- [data annotations](#)
- [Code snippets for this document.](#)

ASP.NET Core built-in Tag Helpers

9/25/2017 • 1 min to read • [Edit Online](#)

By [Peter Kellner](#)

ASP.NET Core includes many built-in Tag Helpers to boost your productivity. This section provides an overview of the built-in Tag Helpers.

NOTE

There are built-in Tag Helpers which aren't discussed, since they're used internally by the [Razor](#) view engine. This includes a Tag Helper for the ~ character, which expands to the root path of the website.

Built-in ASP.NET Core Tag Helpers

[Anchor Tag Helper](#)

[Cache Tag Helper](#)

[Distributed Cache Tag Helper](#)

[Environment Tag Helper](#)

[Form Tag Helper](#)

[Image Tag Helper](#)

[Input Tag Helper](#)

[Label Tag Helper](#)

[Select Tag Helper](#)

[Textarea Tag Helper](#)

[Validation Message Tag Helper](#)

[Validation Summary Tag Helper](#)

Additional resources

- [Client-Side Development](#)
- [Tag Helpers](#)

Anchor Tag Helper

1/11/2018 • 6 min to read • [Edit Online](#)

By [Peter Kellner](#)

The Anchor Tag Helper enhances the HTML anchor (`<a ... >`) tag by adding new attributes. The link generated (on the `href` tag) is created using the new attributes. That URL can include an optional protocol such as https.

The speaker controller below is used in samples in this document.

SpeakerController.cs

```

using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;

namespace TagHelpersBuiltInAspNetCore.Controllers
{
    public class SpeakerController : Controller
    {
        public List<ModelData> Speakers =
            new List<ModelData>
            {
                new ModelData {SpeakerId = 10},
                new ModelData {SpeakerId = 11},
                new ModelData {SpeakerId = 12}
            };

        [Route("Speaker/{id:int}")]
        public IActionResult Detail(int id)
        {
            return View(Speakers.
                FirstOrDefault(a => a.SpeakerId == id));
        }

        [Route("/Speaker/Evaluations",
            Name = "speakerevals")]
        public IActionResult Evaluations()
        {
            return View();
        }

        [Route("/Speaker/EvaluationsCurrent",
            Name = "speakerevalscurrent")]
        public IActionResult
            EvaluationsCurrent(string speakerId,
                string currentYear)
        {
            return View();
        }

        // GET: /<controller>/
        public IActionResult Index()
        {
            return View(Speakers);
        }
    }

    public class ModelData
    {
        public int SpeakerId { get; set; }
    }
}

```

Anchor Tag Helper Attributes

asp-controller

`asp-controller` is used to associate which controller will be used to generate the URL. The controllers specified must exist in the current project. The following code lists all speakers:

```
<a asp-controller="Speaker" asp-action="Index">All Speakers</a>
```

The generated markup will be:

```
<a href="/Speaker">All Speakers</a>
```

If the `asp-controller` is specified and `asp-action` is not, the default `asp-action` will be the default controller method of the currently executing view. That is, in the above example, if `asp-action` is left out, and this Anchor Tag Helper is generated from *HomeController*'s `Index` view (**/Home**), the generated markup will be:

```
<a href="/Home">All Speakers</a>
```

asp-action

`asp-action` is the name of the action method in the controller that will be included in the generated `href`. For example, the following code set the generated `href` to point to the speaker detail page:

```
<a asp-controller="Speaker" asp-action="Detail">Speaker Detail</a>
```

The generated markup will be:

```
<a href="/Speaker/Detail">Speaker Detail</a>
```

If no `asp-controller` attribute is specified, the default controller calling the view executing the current view will be used.

If the attribute `asp-action` is `Index`, then no action is appended to the URL, leading to the default `Index` method being called. The action specified (or defaulted), must exist in the controller referenced in `asp-controller`.

asp-page

Use the `asp-page` attribute in an anchor tag to set its URL to point to a specific page. Prefixing the page name with a forward slash "/" creates the URL. The URL in the sample below points to the "Speaker" page in the current directory.

```
<a asp-page="/Speakers">All Speakers</a>
```

The `asp-page` attribute in the previous code sample renders HTML output in the view that is similar to the following snippet:

```
<a href="/items?page=%2FSpeakers">Speakers</a>
```

The `asp-page` attribute is mutually exclusive with the `asp-route`, `asp-controller`, and `asp-action` attributes. However, `asp-page` can be used with `asp-route-id` to control routing, as the following code sample demonstrates:

```
<a asp-page="/Speaker" asp-route-id="@speaker.Id">View Speaker</a>
```

The `asp-route-id` produces the following output:

```
https://localhost:44399/Speakers/Index/2?page=%2FSpeaker
```

NOTE

To use the `asp-page` attribute in Razor Pages, the URLs must be a relative path, for example `"/Speaker"`. Relative paths in the `asp-page` attribute are not available in MVC views. Use the `/` syntax for MVC views instead.

asp-route-{value}

`asp-route-` is a wild card route prefix. Any value you put after the trailing dash will be interpreted as a potential route parameter. If a default route is not found, this route prefix will be appended to the generated href as a request parameter and value. Otherwise it will be substituted in the route template.

Assuming you have a controller method defined as follows:

```
public IActionResult AnchorTagHelper(string id)
{
    var speaker = new SpeakerData()
    {
        SpeakerId = 12
    };
    return View(viewName, speaker);
}
```

And have the default route template defined in your *Startup.cs* as follows:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

The **cshtml** file that contains the Anchor Tag Helper necessary to use the **speaker** model parameter passed in from the controller to the view is as follows:

```
@model SpeakerData
<!DOCTYPE html>
<html><body>
<a asp-controller='Speaker' asp-action='Detail' asp-route-id=@Model.SpeakerId>SpeakerId: @Model.SpeakerId</a>
</body></html>
```

The generated HTML will then be as follows because **id** was found in the default route.

```
<a href='/Speaker/Detail/12'>SpeakerId: 12</a>
```

If the route prefix is not part of the routing template found, which is the case with the following **cshtml** file:

```
@model SpeakerData
<!DOCTYPE html>
<html><body>
<a asp-controller='Speaker' asp-action='Detail' asp-route-speakerid=@Model.SpeakerId>SpeakerId:
@Model.SpeakerId</a>
</body></html>
```

The generated HTML will then be as follows because **speakerid** was not found in the route matched:

```
<a href='/Speaker/Detail?speakerid=12'>SpeakerId: 12</a>
```

If either `asp-controller` or `asp-action` are not specified, then the same default processing is followed as is in the `asp-route` attribute.

asp-route

`asp-route` provides a way to create a URL that links directly to a named route. Using routing attributes, a route can be named as shown in the `SpeakerController` and used in its `Evaluations` method.

`Name = "spekerevals"` tells the Anchor Tag Helper to generate a route directly to that controller method using the URL `/Speaker/Evaluations`. If `asp-controller` or `asp-action` is specified in addition to `asp-route`, the route generated may not be what you expect. `asp-route` should not be used with either of the attributes `asp-controller` or `asp-action` to avoid a route conflict.

asp-all-route-data

`asp-all-route-data` allows creating a dictionary of key value pairs where the key is the parameter name and the value is the value associated with that key.

As the example below shows, an inline dictionary is created and the data is passed to the razor view. As an alternative, the data could also be passed in with your model.

```
@{
    var dict =
        new Dictionary<string, string>
        {
            {"speakerId", "11"},
            {"currentYear", "true"}
        };
}
<a asp-route="spekerevalscurrent"
    asp-all-route-data="dict">SpeakerEvals</a>
```

The code above generates the following URL: <http://localhost/Speaker/EvaluationsCurrent?speakerId=11&tYear=true>

When the link is clicked, the controller method `EvaluationsCurrent` is called. It is called because that controller has two string parameters that match what has been created from the `asp-all-route-data` dictionary.

If any keys in the dictionary match route parameters, those values will be substituted in the route as appropriate and the other non-matching values will be generated as request parameters.

asp-fragment

`asp-fragment` defines a URL fragment to append to the URL. The Anchor Tag Helper will add the hash character (#). If you create a tag:

```
<a asp-action="Evaluations" asp-controller="Speaker"
    asp-fragment="SpeakerEvaluations">About Speaker Evals</a>
```

The generated URL will be: <http://localhost/Speaker/Evaluations#SpeakerEvaluations>

Hash tags are useful when building client-side applications. They can be used for easy marking and searching in JavaScript, for example.

asp-area

`asp-area` sets the area name that ASP.NET Core uses to set the appropriate route. Below are examples of how the

area attribute causes a remapping of routes. Setting `asp-area` to Blogs prefixes the directory `Areas/Blogs` to the routes of the associated controllers and views for this anchor tag.

- Project name
 - wwwroot
 - Areas
 - Blogs
 - Controllers
 - HomeController.cs
 - Views
 - Home
 - Index.cshtml
 - AboutBlog.cshtml
 - Controllers

Specifying an area tag that is valid, such as `area="Blogs"` when referencing the `AboutBlog.cshtml` file will look like the following using the Anchor Tag Helper.

```
<a asp-action="AboutBlog" asp-controller="Home" asp-area="Blogs">Blogs About</a>
```

The generated HTML will include the areas segment and will be as follows:

```
<a href="/Blogs/Home/AboutBlog">Blogs About</a>
```

TIP

For MVC areas to work in a web application, the route template must include a reference to the area if it exists. That template, which is the second parameter of the `routes.MapRoute` method call, will appear as:

```
template: "{area:exists}/{controller=Home}/{action=Index}"
```

asp-protocol

The `asp-protocol` is for specifying a protocol (such as `https`) in your URL. An example Anchor Tag Helper that includes the protocol will look as follows:

```
<a asp-protocol="https" asp-action="About" asp-controller="Home">About</a>
```

and will generate HTML as follows:

```
<a href="https://localhost/Home/About">About</a>
```

The domain in the example is localhost, but the Anchor Tag Helper uses the website's public domain when generating the URL.

Additional resources

- [Areas](#)

Cache Tag Helper in ASP.NET Core MVC

12/13/2017 • 4 min to read • [Edit Online](#)

By [Peter Kellner](#)

The Cache Tag Helper provides the ability to dramatically improve the performance of your ASP.NET Core app by caching its content to the internal ASP.NET Core cache provider.

The Razor View Engine sets the default `expires-after` to twenty minutes.

The following Razor markup caches the date/time:

```
<cache>@DateTime.Now</cache>
```

The first request to the page that contains `CacheTagHelper` will display the current date/time. Additional requests will show the cached value until the cache expires (default 20 minutes) or is evicted by memory pressure.

You can set the cache duration with the following attributes:

Cache Tag Helper Attributes

enabled

ATTRIBUTE TYPE	VALID VALUES
boolean	"true" (default)
	"false"

Determines whether the content enclosed by the Cache Tag Helper is cached. The default is `true`. If set to `false` this Cache Tag Helper will have no caching effect on the rendered output.

Example:

```
<cache enabled="true">  
    Current Time Inside Cache Tag Helper: @DateTime.Now  
</cache>
```

expires-on

ATTRIBUTE TYPE	EXAMPLE VALUE
DateTimeOffset	"@new DateTime(2025,1,29,17,02,0)"

Sets an absolute expiration date. The following example will cache the contents of the Cache Tag Helper until 5:02 PM on January 29, 2025.

Example:

```
<cache expires-on="@new DateTime(2025,1,29,17,02,0)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-after

ATTRIBUTE TYPE	EXAMPLE VALUE
TimeSpan	"@TimeSpan.FromSeconds(120)"

Sets the length of time from the first request time to cache the contents.

Example:

```
<cache expires-after="@TimeSpan.FromSeconds(120)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-sliding

ATTRIBUTE TYPE	EXAMPLE VALUE
TimeSpan	"@TimeSpan.FromSeconds(60)"

Sets the time that a cache entry should be evicted if it has not been accessed.

Example:

```
<cache expires-sliding="@TimeSpan.FromSeconds(60)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-header

ATTRIBUTE TYPE	EXAMPLE VALUES
String	"User-Agent"
	"User-Agent,content-encoding"

Accepts a single header value or a comma-separated list of header values that trigger a cache refresh when they change. The following example monitors the header value `User-Agent`. The example will cache the content for every different `User-Agent` presented to the web server.

Example:

```
<cache vary-by-header="User-Agent">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-query

ATTRIBUTE TYPE	EXAMPLE VALUES
String	"Make"
	"Make,Model"

Accepts a single header value or a comma-separated list of header values that trigger a cache refresh when the header value changes. The following example looks at the values of `Make` and `Model`.

Example:

```
<cache vary-by-query="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-route

ATTRIBUTE TYPE	EXAMPLE VALUES
String	"Make"
	"Make,Model"

Accepts a single header value or a comma-separated list of header values that trigger a cache refresh when the route data parameter value(s) change. Example:

Startup.cs

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{Make?}/{Model?}");
```

Index.cshtml

```
<cache vary-by-route="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-cookie

ATTRIBUTE TYPE	EXAMPLE VALUES
String	".AspNetCore.Identity.Application"
	".AspNetCore.Identity.Application,HairColor"

Accepts a single header value or a comma-separated list of header values that trigger a cache refresh when the header values(s) change. The following example looks at the cookie associated with ASP.NET Identity. When a user is authenticated the request cookie to be set which triggers a cache refresh.

Example:

```
<cache vary-by-cookie=".AspNetCore.Identity.Application">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-user

ATTRIBUTE TYPE	EXAMPLE VALUES
Boolean	"true"
	"false" (default)

Specifies whether or not the cache should reset when the logged-in user (or Context Principal) changes. The current user is also known as the Request Context Principal and can be viewed in a Razor view by referencing `@User.Identity.Name`.

The following example looks at the current logged in user.

Example:

```
<cache vary-by-user="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

Using this attribute maintains the contents in cache through a log-in and log-out cycle. When using `vary-by-user="true"`, a log-in and log-out action invalidates the cache for the authenticated user. The cache is invalidated because a new unique cookie value is generated on login. Cache is maintained for the anonymous state when no cookie is present or has expired. This means if no user is logged in, the cache will be maintained.

vary-by

ATTRIBUTE TYPE	EXAMPLE VALUES
String	"@Model"

Allows for customization of what data gets cached. When the object referenced by the attribute's string value changes, the content of the Cache Tag Helper is updated. Often a string-concatenation of model values are assigned to this attribute. Effectively, that means an update to any of the concatenated values invalidates the cache.

The following example assumes the controller method rendering the view sums the integer value of the two route parameters, `myParam1` and `myParam2`, and returns that as the single model property. When this sum changes, the content of the Cache Tag Helper is rendered and cached again.

Example:

Action:

```
public IActionResult Index(string myParam1,string myParam2,string myParam3)
{
    int num1;
    int num2;
    int.TryParse(myParam1, out num1);
    int.TryParse(myParam2, out num2);
    return View(viewName, num1 + num2);
}
```

Index.cshtml

```
<cache vary-by="@Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

priority

ATTRIBUTE TYPE	EXAMPLE VALUES
CacheItemPriority	"High"
	"Low"
	"NeverRemove"
	"Normal"

Provides cache eviction guidance to the built-in cache provider. The web server will evict `Low` cache entries first when it's under memory pressure.

Example:

```
<cache priority="High">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

The `priority` attribute does not guarantee a specific level of cache retention. `CacheItemPriority` is only a suggestion. Setting this attribute to `NeverRemove` does not guarantee that the cache will always be retained. See [Additional Resources](#) for more information.

The Cache Tag Helper is dependent on the [memory cache service](#). The Cache Tag Helper adds the service if it has not been added.

Additional resources

- [In-memory caching in ASP.NET Core](#)
- [Introduction to Identity on ASP.NET Core](#)

Distributed Cache Tag Helper

9/25/2017 • 1 min to read • [Edit Online](#)

By [Peter Kellner](#)

The Distributed Cache Tag Helper provides the ability to dramatically improve the performance of your ASP.NET Core app by caching its content to a distributed cache source.

The Distributed Cache Tag Helper inherits from the same base class as the Cache Tag Helper. All attributes associated with the Cache Tag Helper will also work on the Distributed Tag Helper.

The Distributed Cache Tag Helper follows the **Explicit Dependencies Principle** known as **Constructor Injection**. Specifically, the `IDistributedCache` interface container is passed into the Distributed Cache Tag Helper's constructor. If no specific concrete implementation of `IDistributedCache` has been created in `ConfigureServices`, usually found in `startup.cs`, then the Distributed Cache Tag Helper will use the same in-memory provider for storing cached data as the basic Cache Tag Helper.

Distributed Cache Tag Helper Attributes

enabled expires-on expires-after expires-sliding vary-by-header vary-by-query vary-by-route vary-by-cookie vary-by-user vary-by priority

See Cache Tag Helper for definitions. Distributed Cache Tag Helper inherits from the same class as Cache Tag Helper so all these attributes are common from Cache Tag Helper.

name (required)

ATTRIBUTE TYPE	EXAMPLE VALUE
string	"my-distributed-cache-unique-key-101"

The required `name` attribute is used as a key to that cache stored for each instance of a Distributed Cache Tag Helper. Unlike the basic Cache Tag Helper that assigns a key to each Cache Tag Helper instance based on the Razor page name and location of the tag helper in the razor page, the Distributed Cache Tag Helper only bases its key on the attribute `name`.

Usage Example:

```
<distributed-cache name="my-distributed-cache-unique-key-101">
  Time Inside Cache Tag Helper: @DateTime.Now
</distributed-cache>
```

Distributed Cache Tag Helper IDistributedCache Implementations

There are two implementations of `IDistributedCache` built in to ASP.NET Core. One is based on **Sql Server** and the other is based on **Redis**. Details of these implementations can be found at the resource referenced below named "Working with a distributed cache". Both implementations involve setting an instance of `IDistributedCache` in ASP.NET Core's `startup.cs`.

There are no tag attributes specifically associated with using any specific implementation of `IDistributedCache`.

Additional resources

- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Dependency Injection in ASP.NET Core](#)
- [Working with a distributed cache in ASP.NET Core](#)
- [In-memory caching in ASP.NET Core](#)
- [Introduction to Identity on ASP.NET Core](#)

Environment Tag Helper in ASP.NET Core

11/1/2017 • 1 min to read • [Edit Online](#)

By [Peter Kellner](#) and [Hisham Bin Ateya](#)

The Environment Tag Helper conditionally renders its enclosed content based on the current hosting environment. Its single attribute `names` is a comma separated list of environment names, that if any match to the current environment, will trigger the enclosed content to be rendered.

Environment Tag Helper Attributes

names

Accepts a single hosting environment name or a comma-separated list of hosting environment names that trigger the rendering of the enclosed content.

These value(s) are compared to the current value returned from the ASP.NET Core static property

`HostingEnvironment.EnvironmentName`. This value is one of the following: **Staging**; **Development** or **Production**.

The comparison ignores case.

An example of a valid `environment` tag helper is:

```
<environment names="Staging,Production">
  <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

include and exclude attributes

ASP.NET Core 2.x adds the `include` & `exclude` attributes. These attributes control rendering the enclosed content based on the included or excluded hosting environment names.

include ASP.NET Core 2.0 and later

The `include` property has a similar behavior of the `names` attribute in ASP.NET Core 1.0.

```
<environment include="Staging,Production">
  <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

exclude ASP.NET Core 2.0 and later

In contrast, the `exclude` property lets the `EnvironmentTagHelper` render the enclosed content for all hosting environment names except the one(s) that you specified.

```
<environment exclude="Development">
  <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

Additional resources

- [Working with multiple environments in ASP.NET Core](#)
- [Dependency Injection in ASP.NET Core](#)

ImageTagHelper

9/25/2017 • 1 min to read • [Edit Online](#)

By [Peter Kellner](#)

The Image Tag Helper enhances the `img` (``) tag. It requires a `src` tag as well as the `boolean` attribute `asp-append-version`.

If the image source (`src`) is a static file on the host web server, a unique cache busting string is appended as a query parameter to the image source. This ensures that if the file on the host web server changes, a unique request URL is generated that includes the updated request parameter. The cache busting string is a unique value representing the hash of the static image file.

If the image source (`src`) isn't a static file (for example a remote URL or the file doesn't exist on the server), the `` tag's `src` attribute is generated with no cache busting query string parameter.

Image Tag Helper Attributes

asp-append-version

When specified along with a `src` attribute, the Image Tag Helper is invoked.

An example of a valid `img` tag helper is:

```

```

If the static file exists in the directory `..wwwroot/images/asplogo.png` the generated html is similar to the following (the hash will be different):

```

```

The value assigned to the parameter `v` is the hash value of the file on disk. If the web server is unable to obtain read access to the static file referenced, no `v` parameters is added to the `src` attribute.

src

To activate the Image Tag Helper, the `src` attribute is required on the `` element.

NOTE

The Image Tag Helper uses the `cache` provider on the local web server to store the calculated `Sha512` of a given file. If the file is requested again the `Sha512` does not need to be recalculated. The Cache is invalidated by a file watcher that is attached to the file when the file's `Sha512` is calculated.

Additional resources

- [In-memory caching in ASP.NET Core](#)

Partial Views

10/2/2017 • 4 min to read • [Edit Online](#)

By [Steve Smith](#), [Maher JENDOUBI](#), and [Rick Anderson](#)

ASP.NET Core MVC supports partial views, which are useful when you have reusable parts of web pages you want to share between different views.

[View or download sample code](#) ([how to download](#))

What are Partial Views?

A partial view is a view that is rendered within another view. The HTML output generated by executing the partial view is rendered into the calling (or parent) view. Like views, partial views use the `.cshtml` file extension.

When Should I Use Partial Views?

Partial views are an effective way of breaking up large views into smaller components. They can reduce duplication of view content and allow view elements to be reused. Common layout elements should be specified in `_Layout.cshtml`. Non-layout reusable content can be encapsulated into partial views.

If you have a complex page made up of several logical pieces, it can be helpful to work with each piece as its own partial view. Each piece of the page can be viewed in isolation from the rest of the page, and the view for the page itself becomes much simpler since it only contains the overall page structure and calls to render the partial views.

Tip: Follow the [Don't Repeat Yourself Principle](#) in your views.

Declaring Partial Views

Partial views are created like any other view: you create a `.cshtml` file within the `Views` folder. There is no semantic difference between a partial view and a regular view - they are just rendered differently. You can have a view that is returned directly from a controller's `ViewResult`, and the same view can be used as a partial view. The main difference between how a view and a partial view are rendered is that partial views do not run `_ViewStart.cshtml` (while views do - learn more about `_ViewStart.cshtml` in [Layout](#)).

Referencing a Partial View

From within a view page, there are several ways in which you can render a partial view. The simplest is to use `Html.Partial`, which returns an `IHtmlString` and can be referenced by prefixing the call with `@`:

```
@Html.Partial("AuthorPartial")
```

The `PartialAsync` method is available for partial views containing asynchronous code (although code in views is generally discouraged):

```
@await Html.PartialAsync("AuthorPartial")
```

You can render a partial view with `RenderPartial`. This method doesn't return a result; it streams the rendered output directly to the response. Because it doesn't return a result, it must be called within a Razor code block (you can also call `RenderPartialAsync` if necessary):

```
@{
    Html.RenderPartial("AuthorPartial");
}
```

Because it streams the result directly, `RenderPartial` and `RenderPartialAsync` may perform better in some scenarios. However, in most cases it's recommended you use `Partial` and `PartialAsync`.

NOTE

If your views need to execute code, the recommended pattern is to use a [view component](#) instead of a partial view.

Partial View Discovery

When referencing a partial view, you can refer to its location in several ways:

```
// Uses a view in current folder with this name
// If none is found, searches the Shared folder
@Html.Partial("ViewName")

// A view with this name must be in the same folder
@Html.Partial("ViewName.cshtml")

// Locate the view based on the application root
// Paths that start with "/" or "~/\" refer to the application root
@Html.Partial("~/Views/Folder/ViewName.cshtml")
@Html.Partial("/Views/Folder/ViewName.cshtml")

// Locate the view using relative paths
@Html.Partial("../Account/LoginPartial.cshtml")
```

You can have different partial views with the same name in different view folders. When referencing the views by name (without file extension), views in each folder will use the partial view in the same folder with them. You can also specify a default partial view to use, placing it in the *Shared* folder. The shared partial view will be used by any views that don't have their own version of the partial view. You can have a default partial view (in *Shared*), which is overridden by a partial view with the same name in the same folder as the parent view.

Partial views can be *chained*. That is, a partial view can call another partial view (as long as you don't create a loop). Within each view or partial view, relative paths are always relative to that view, not the root or parent view.

NOTE

If you declare a [Razor section](#) in a partial view, it will not be visible to its parent(s); it will be limited to the partial view.

Accessing Data From Partial Views

When a partial view is instantiated, it gets a copy of the parent view's `ViewData` dictionary. Updates made to the data within the partial view are not persisted to the parent view. `ViewData` changed in a partial view is lost when the partial view returns.

You can pass an instance of `ViewDataDictionary` to the partial view:

```
@Html.Partial("PartialName", customViewData)
```

You can also pass a model into a partial view. This can be the page's view model, or some portion of it, or a custom object. You can pass a model to `Partial`, `PartialAsync`, `RenderPartial`, or `RenderPartialAsync`:

```
@Html.Partial("PartialName", viewModel)
```

You can pass an instance of `ViewDataDictionary` and a view model to a partial view:

```
@Html.Partial("ArticleSection", section,  
    new ViewDataDictionary(this.ViewData) { { "index", index } })
```

The markup below shows the `Views/Articles/Read.cshtml` view which contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. You can pass new `ViewData` dictionary while retaining the existing `ViewData` if you use the constructor overload of the `ViewDataDictionary` highlighted below:

```
@using Microsoft.AspNetCore.Mvc.ViewFeatures  
@using PartialViewSample.ViewModels  
@model Article  
  
<h2>@Model.Title</h2>  
@*Pass the authors name to Views\Shared\AuthorPartial.cshtml*@  
@Html.Partial("AuthorPartial", Model.AuthorName)  
@Model.PublicationDate  
  
@*Loop over the Sections and pass in a section and additional ViewData  
to the strongly typed Views\Articles\ArticleSection.cshtml partial view.*@  
@{ var index = 0;  
    @foreach (var section in Model.Sections)  
    {  
        @Html.Partial("ArticleSection", section,  
            new ViewDataDictionary(this.ViewData) { { "index", index } })  
        index++;  
    }  
}
```

Views/Shared/AuthorPartial:

```
@model string  
<div>  
    <h3>@Model</h3>  
    This partial view came from /Views/Shared/AuthorPartial.cshtml.<br />  
</div>
```

The *ArticleSection* partial:

```
@using PartialViewSample.ViewModels  
@model ArticleSection  
  
<h3>@Model.Title Index: @ViewData["index"] </h3>  
<div>  
    @Model.Content  
</div>
```

At runtime, the partials are rendered into the parent view, which itself is rendered within the shared `_Layout.cshtml`

PartialViewsSample × + - □ ×

localhost:3501/articles, 📖 ☆ | ☰ ...

PartialViewsSample ☰

The Gettysburg Address

Abraham Lincoln

This partial view came from /Views/Shared/AuthorPartial.cshtml.
11/19/1863 12:00:00 AM

Section One Index: 0

Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

Section Two Index: 1

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

Section Three Index: 2

But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallow -- this ground. The brave men, living and dead, who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to

Dependency injection into views

10/2/2017 • 4 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET Core supports [dependency injection](#) into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. You should try to maintain [separation of concerns](#) between your controllers and views. Most of the data your views display should be passed in from the controller.

[View or download sample code \(how to download\)](#)

A Simple Example

You can inject a service into a view using the `@inject` directive. You can think of `@inject` as adding a property to your view, and populating the property using DI.

The syntax for `@inject` is: `@inject <type> <name>`

An example of `@inject` in action:

```
@using System.Threading.Tasks
@using ViewInjectSample.Model
@using ViewInjectSample.Model.Services
@model IEnumerable<ToDoItem>
@inject StatisticsService StatsService
<!DOCTYPE html>
<html>
<head>
<title>To Do Items</title>
</head>
<body>
<div>
<h1>To Do Items</h1>
<ul>
<li>Total Items: @StatsService.GetCount()/li>
<li>Completed: @StatsService.GetCompletedCount()/li>
<li>Avg. Priority: @StatsService.GetAveragePriority()/li>
</ul>
<table>
<tr>
<th>Name</th>
<th>Priority</th>
<th>Is Done?</th>
</tr>
@foreach (var item in Model)
{
<tr>
<td>@item.Name</td>
<td>@item.Priority</td>
<td>@item.IsDone</td>
</tr>
}
</table>
</div>
</body>
</html>
```

This view displays a list of `ToDoItem` instances, along with a summary showing overall statistics. The summary is

populated from the injected `StatisticsService`. This service is registered for dependency injection in `ConfigureServices` in `Startup.cs`:

```
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
    services.AddTransient<StatisticsService>();
    services.AddTransient<ProfileOptionsService>();
}
```

The `StatisticsService` performs some calculations on the set of `ToDoItem` instances, which it accesses via a repository:

```
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _todoItemRepository;

        public StatisticsService(IToDoItemRepository todoItemRepository)
        {
            _todoItemRepository = todoItemRepository;
        }

        public int GetCount()
        {
            return _todoItemRepository.List().Count();
        }

        public int GetCompletedCount()
        {
            return _todoItemRepository.List().Count(x => x.IsDone);
        }

        public double GetAveragePriority()
        {
            if (_todoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _todoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

The sample repository uses an in-memory collection. The implementation shown above (which operates on all of the data in memory) is not recommended for large, remotely accessed data sets.

The sample displays data from the model bound to the view and the service injected into the view:

To Do Items

- Total Items: 50
- Completed: 17
- Avg. Priority: 3

Name	Priority	Is Done?
Task 1	1	True
Task 2	2	False
Task 3	3	False
Task 4	4	True
Task 5	5	False

Populating Lookup Data

View injection can be useful to populate options in UI elements, such as dropdown lists. Consider a user profile form that includes options for specifying gender, state, and other preferences. Rendering such a form using a standard MVC approach would require the controller to request data access services for each of these sets of options, and then populate a model or `ViewBag` with each set of options to be bound.

An alternative approach injects services directly into the view to obtain the options. This minimizes the amount of code required by the controller, moving this view element construction logic into the view itself. The controller action to display a profile editing form only needs to pass the form the profile instance:

```
using Microsoft.AspNetCore.Mvc;
using ViewInjectSample.Model;

namespace ViewInjectSample.Controllers
{
    public class ProfileController : Controller
    {
        [Route("Profile")]
        public IActionResult Index()
        {
            // TODO: look up profile based on logged-in user
            var profile = new Profile()
            {
                Name = "Steve",
                FavColor = "Blue",
                Gender = "Male",
                State = new State("Ohio", "OH")
            };
            return View(profile);
        }
    }
}
```

The HTML form used to update these preferences includes dropdown lists for three of the properties:

Update Profile

Name:

Gender:

State:

Fav. Color:

These lists are populated by a service that has been injected into the view:

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
  <title>Update Profile</title>
</head>
<body>
<div>
  <h1>Update Profile</h1>
  Name: @Html.TextBoxFor(m => m.Name)
  <br/>
  Gender: @Html.DropDownList("Gender",
    Options.ListGenders().Select(g =>
      new SelectListItem() { Text = g, Value = g }))
  <br/>
  State: @Html.DropDownListFor(m => m.State.Code,
    Options.ListStates().Select(s =>
      new SelectListItem() { Text = s.Name, Value = s.Code}))
  <br />
  Fav. Color: @Html.DropDownList("FavColor",
    Options.ListColors().Select(c =>
      new SelectListItem() { Text = c, Value = c }))
</div>
</body>
</html>
```

The `ProfileOptionsService` is a UI-level service designed to provide just the data needed for this form:

```

using System.Collections.Generic;

namespace ViewInjectSample.Model.Services
{
    public class ProfileOptionsService
    {
        public List<string> ListGenders()
        {
            // keeping this simple
            return new List<string>() {"Female", "Male"};
        }

        public List<State> ListStates()
        {
            // a few states from USA
            return new List<State>()
            {
                new State("Alabama", "AL"),
                new State("Alaska", "AK"),
                new State("Ohio", "OH")
            };
        }

        public List<string> ListColors()
        {
            return new List<string>() { "Blue","Green","Red","Yellow" };
        }
    }
}

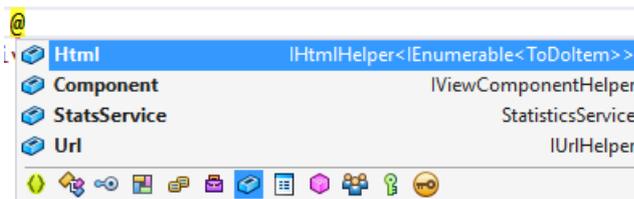
```

TIP

Don't forget to register types you will request through dependency injection in the `ConfigureServices` method in `Startup.cs`.

Overriding Services

In addition to injecting new services, this technique can also be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:



As you can see, the default fields include `Html`, `Component`, and `Url` (as well as the `StatsService` that we injected). If for instance you wanted to replace the default HTML Helpers with your own, you could easily do so using `@inject`:

```
@using System.Threading.Tasks
@using ViewInjectSample.Helpers
@inject MyHtmlHelper Html
<!DOCTYPE html>
<html>
<head>
  <title>My Helper</title>
</head>
<body>
  <div>
    Test: @Html.Value
  </div>
</body>
</html>
```

If you want to extend existing services, you can simply use this technique while inheriting from or wrapping the existing implementation with your own.

See Also

- Simon Timms Blog: [Getting Lookup Data Into Your View](#)

View components

10/31/2017 • 9 min to read • [Edit Online](#)

By [Rick Anderson](#)

[View or download sample code \(how to download\)](#)

Introducing view components

New to ASP.NET Core MVC, view components are similar to partial views, but they are much more powerful. View components don't use model binding, and only depend on the data you provide when calling into it. A view component:

- Renders a chunk rather than a whole response
- Includes the same separation-of-concerns and testability benefits found between a controller and view
- Can have parameters and business logic
- Is typically invoked from a layout page

View components are intended anywhere you have reusable rendering logic that is too complex for a partial view, such as:

- Dynamic navigation menus
- Tag cloud (where it queries the database)
- Login panel
- Shopping cart
- Recently published articles
- Sidebar content on a typical blog
- A login panel that would be rendered on every page and show either the links to log out or log in, depending on the log in state of the user

A view component consists of two parts: the class (typically derived from [ViewComponent](#)) and the result it returns (typically a view). Like controllers, a view component can be a POCO, but most developers will want to take advantage of the methods and properties available by deriving from `ViewComponent`.

Creating a view component

This section contains the high-level requirements to create a view component. Later in the article, we'll examine each step in detail and create a view component.

The view component class

A view component class can be created by any of the following:

- Deriving from *ViewComponent*
- Decorating a class with the `[ViewComponent]` attribute, or deriving from a class with the `[ViewComponent]` attribute
- Creating a class where the name ends with the suffix *ViewComponent*

Like controllers, view components must be public, non-nested, and non-abstract classes. The view component name is the class name with the "ViewComponent" suffix removed. It can also be explicitly specified using the `ViewComponentAttribute.Name` property.

A view component class:

- Fully supports constructor [dependency injection](#)
- Does not take part in the controller lifecycle, which means you can't use [filters](#) in a view component

View component methods

A view component defines its logic in an `InvokeAsync` method that returns an `IViewComponentResult`. Parameters come directly from invocation of the view component, not from model binding. A view component never directly handles a request. Typically, a view component initializes a model and passes it to a view by calling the `View` method. In summary, view component methods:

- Define an `InvokeAsync` method that returns an `IViewComponentResult`
- Typically initializes a model and passes it to a view by calling the `ViewComponent` `View` method
- Parameters come from the calling method, not HTTP, there is no model binding
- Are not reachable directly as an HTTP endpoint, they are invoked from your code (usually in a view). A view component never handles a request
- Are overloaded on the signature rather than any details from the current HTTP request

View search path

The runtime searches for the view in the following paths:

- Views/<controller_name>/Components/<view_component_name>/<view_name>
- Views/Shared/Components/<view_component_name>/<view_name>

The default view name for a view component is *Default*, which means your view file will typically be named *Default.cshtml*. You can specify a different view name when creating the view component result or when calling the `View` method.

We recommend you name the view file *Default.cshtml* and use the *Views/Shared/Components/<view_component_name>/<view_name>* path. The `PriorityList` view component used in this sample uses *Views/Shared/Components/PriorityList/Default.cshtml* for the view component view.

Invoking a view component

To use the view component, call the following inside a view:

```
@Component.InvokeAsync("Name of view component", <anonymous type containing parameters>)
```

The parameters will be passed to the `InvokeAsync` method. The `PriorityList` view component developed in the article is invoked from the *Views/ToDo/Index.cshtml* view file. In the following, the `InvokeAsync` method is called with two parameters:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

Invoking a view component as a Tag Helper

For ASP.NET Core 1.1 and higher, you can invoke a view component as a [Tag Helper](#):

```
<vc:priority-list max-priority="2" is-done="false">  
</vc:priority-list>
```

Pascal-cased class and method parameters for Tag Helpers are translated into their [lower kebab case](#). The Tag

Helper to invoke a view component uses the `<vc></vc>` element. The view component is specified as follows:

```
<vc:[view-component-name]
  parameter1="parameter1 value"
  parameter2="parameter2 value">
</vc:[view-component-name]>
```

Note: In order to use a View Component as a Tag Helper, you must register the assembly containing the View Component using the `@addTagHelper` directive. For example, if your View Component is in an assembly called "MyWebApp", add the following directive to the `_ViewImports.cshtml` file:

```
@addTagHelper *, MyWebApp
```

You can register a View Component as a Tag Helper to any file that references the View Component. See [Managing Tag Helper Scope](#) for more information on how to register Tag Helpers.

The `InvokeAsync` method used in this tutorial:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

In Tag Helper markup:

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

In the sample above, the `PriorityList` view component becomes `priority-list`. The parameters to the view component are passed as attributes in lower kebab case.

Invoking a view component directly from a controller

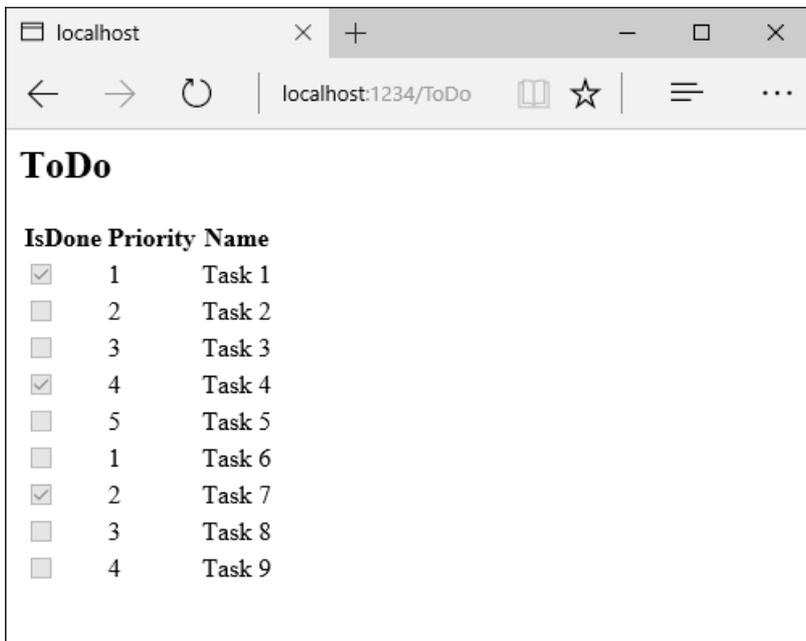
View components are typically invoked from a view, but you can invoke them directly from a controller method. While view components do not define endpoints like controllers, you can easily implement a controller action that returns the content of a `ViewComponentResult`.

In this example, the view component is called directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

Walkthrough: Creating a simple view component

[Download](#), build and test the starter code. It's a simple project with a `Todo` controller that displays a list of *Todo* items.



Add a ViewComponent class

Create a *ViewComponents* folder and add the following `PriorityListViewComponent` class:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityListViewComponent : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityListViewComponent(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
            int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }

        private Task<List<ToDoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

Notes on the code:

- View component classes can be contained in **any** folder in the project.
- Because the class name `PriorityListViewComponent` ends with the suffix **ViewComponent**, the runtime will use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.
- The `[ViewComponent]` attribute can change the name used to reference a view component. For example, we

could have named the class `XYZ` and applied the `ViewComponent` attribute:

```
[ViewComponent(Name = "PriorityList")]
public class XYZ : ViewComponent
```

- The `ViewComponent` attribute above tells the view component selector to use the name `PriorityList` when looking for the views associated with the component, and to use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.
- The component uses [dependency injection](#) to make the data context available.
- `InvokeAsync` exposes a method which can be called from a view, and it can take an arbitrary number of arguments.
- The `InvokeAsync` method returns the set of `ToDo` items that satisfy the `isDone` and `maxPriority` parameters.

Create the view component Razor view

- Create the `Views/Shared/Components` folder. This folder **must** be named `Components`.
- Create the `Views/Shared/Components/PriorityList` folder. This folder name must match the name of the view component class, or the name of the class minus the suffix (if we followed convention and used the `ViewComponent` suffix in the class name). If you used the `ViewComponent` attribute, the class name would need to match the attribute designation.
- Create a `Views/Shared/Components/PriorityList/Default.cshtml` Razor view:

```
@model IEnumerable<ViewComponentSample.Models.ToDoItem>

<h3>Priority Items</h3>
<ul>
  @foreach (var todo in Model)
  {
    <li>@todo.Name</li>
  }
</ul>
```

The Razor view takes a list of `ToDoItem` and displays them. If the view component `InvokeAsync` method doesn't pass the name of the view (as in our sample), `Default` is used for the view name by convention. Later in the tutorial, I'll show you how to pass the name of the view. To override the default styling for a specific controller, add a view to the controller-specific view folder (for example `Views/ToDo/Components/PriorityList/Default.cshtml`).

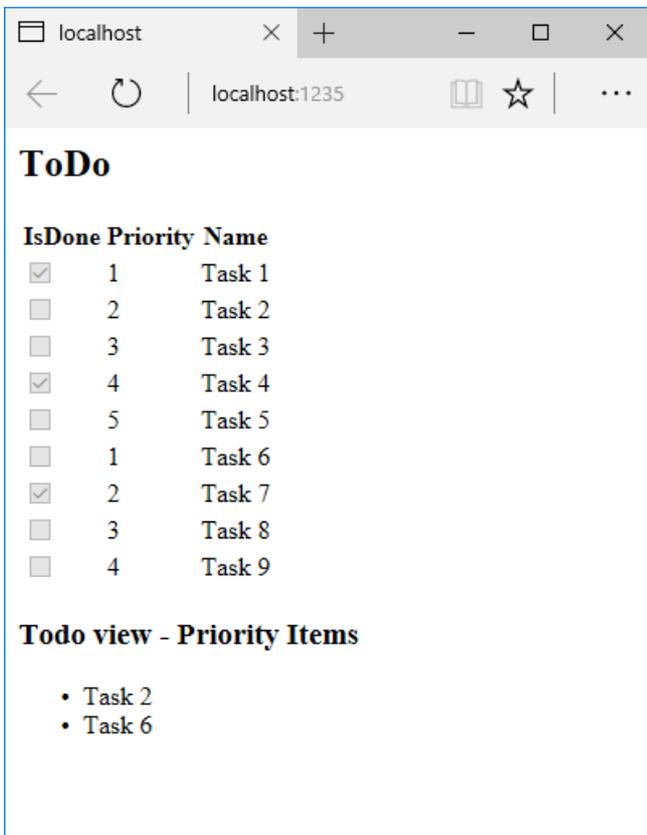
If the view component is controller-specific, you can add it to the controller-specific folder (`Views/ToDo/Components/PriorityList/Default.cshtml`).

- Add a `div` containing a call to the priority list component to the bottom of the `Views/ToDo/index.cshtml` file:

```
</table>
<div>
  @await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })
</div>
```

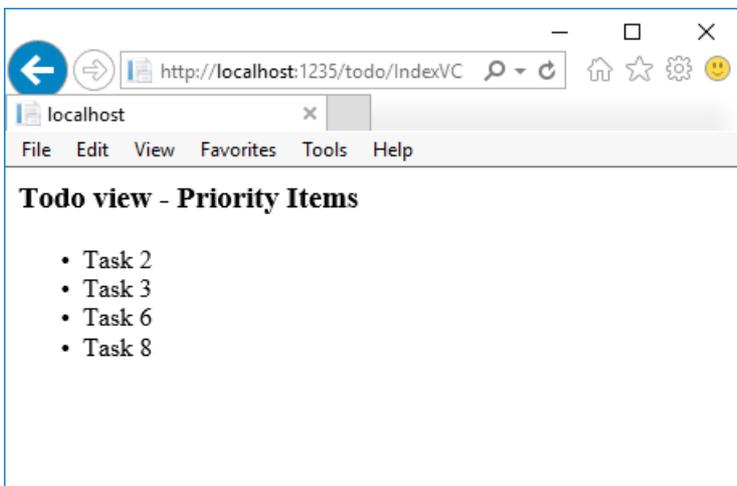
The markup `@await Component.InvokeAsync` shows the syntax for calling view components. The first argument is the name of the component we want to invoke or call. Subsequent parameters are passed to the component. `InvokeAsync` can take an arbitrary number of arguments.

Test the app. The following image shows the `ToDo` list and the priority items:



You can also call the view component directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```



Specifying a view name

A complex view component might need to specify a non-default view under some conditions. The following code shows how to specify the "PVC" view from the `InvokeAsync` method. Update the `InvokeAsync` method in the `PriorityListViewComponent` class.

```
public async Task<IViewComponentResult> InvokeAsync(
    int maxPriority, bool isDone)
{
    string MyView = "Default";
    // If asking for all completed tasks, render with the "PVC" view.
    if (maxPriority > 3 && isDone == true)
    {
        MyView = "PVC";
    }
    var items = await GetItemsAsync(maxPriority, isDone);
    return View(MyView, items);
}
```

Copy the *Views/Shared/Components/PriorityList/Default.cshtml* file to a view named *Views/Shared/Components/PriorityList/PVC.cshtml*. Add a heading to indicate the PVC view is being used.

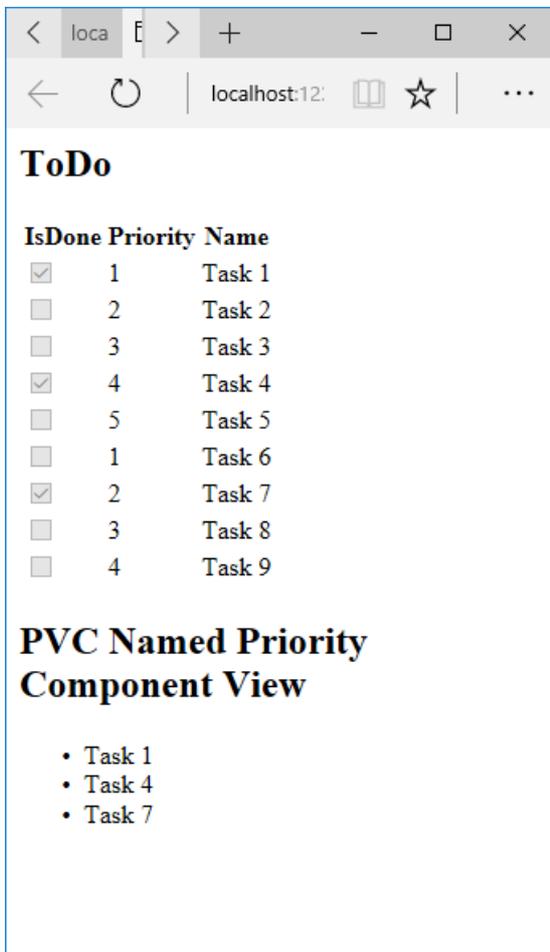
```
@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h2> PVC Named Priority Component View</h2>
<h4>@ViewBag.PriorityMessage</h4>
<ul>
    @foreach (var todo in Model)
    {
        <li>@todo.Name</li>
    }
</ul>
```

Update *Views/ToDoList/Index.cshtml*:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

Run the app and verify PVC view.



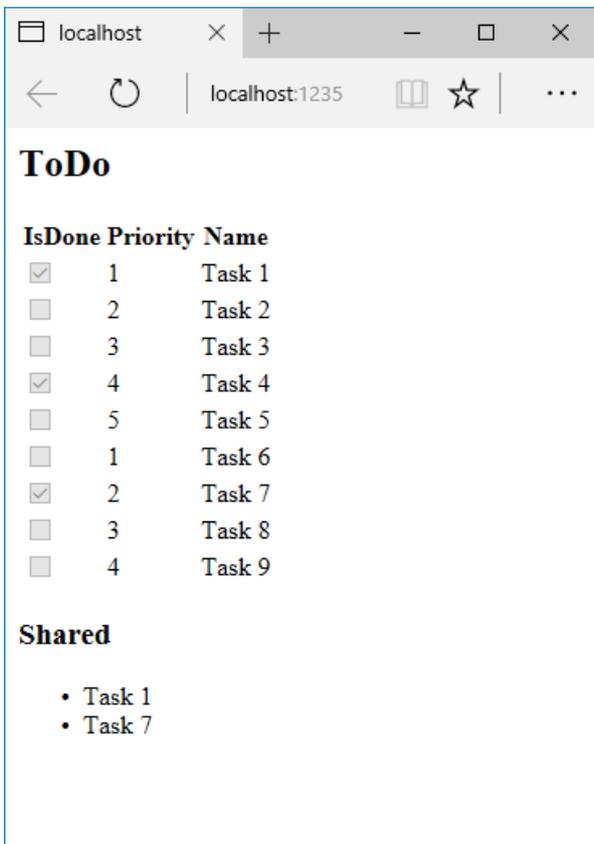
If the PVC view is not rendered, verify you are calling the view component with a priority of 4 or higher.

Examine the view path

- Change the priority parameter to three or less so the priority view is not returned.
- Temporarily rename the *Views/ToDo/Components/PriorityList/Default.cshtml* to *1Default.cshtml*.
- Test the app, you'll get the following error:

```
An unhandled exception occurred while processing the request.
InvalidOperationException: The view 'Components/PriorityList/Default' was not found. The following
locations were searched:
/Views/ToDo/Components/PriorityList/Default.cshtml
/Views/Shared/Components/PriorityList/Default.cshtml
EnsureSuccessful
```

- Copy *Views/ToDo/Components/PriorityList/1Default.cshtml* to *Views/Shared/Components/PriorityList/Default.cshtml*.
- Add some markup to the *Shared* *ToDo* view component view to indicate the view is from the *Shared* folder.
- Test the **Shared** component view.



Avoiding magic strings

If you want compile time safety, you can replace the hard-coded view component name with the class name. Create the view component without the "ViewComponent" suffix:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityList : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityList(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
            int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }

        private Task<List<ToDoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

Add a `using` statement to your Razor view file, and use the `nameof` operator:

```
@using ViewComponentSample.Models
@using ViewComponentSample.ViewComponents
@model IEnumerable<TodoItem>

<h2>ToDo nameof</h2>
<!-- Markup removed for brevity. -->
    }
</table>

<div>

    @await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })
</div>
```

Additional Resources

- [Dependency injection into views](#)

Handling requests with controllers in ASP.NET Core MVC

9/12/2017 • 5 min to read • [Edit Online](#)

By [Steve Smith](#) and [Scott Addie](#)

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET Core MVC.

What is a Controller?

A controller is used to define and group a set of actions. An action (or *action method*) is a method on a controller which handles requests. Controllers logically group similar actions together. This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively. Requests are mapped to actions through [routing](#).

By convention, controller classes:

- Reside in the project's root-level *Controllers* folder
- Inherit from `Microsoft.AspNetCore.Mvc.Controller`

A controller is an instantiable class in which at least one of the following conditions is true:

- The class name is suffixed with "Controller"
- The class inherits from a class whose name is suffixed with "Controller"
- The class is decorated with the `[Controller]` attribute

A controller class must not have an associated `[NonController]` attribute.

Controllers should follow the [Explicit Dependencies Principle](#). There are a couple approaches to implementing this principle. If multiple controller actions require the same service, consider using [constructor injection](#) to request those dependencies. If the service is needed by only a single action method, consider using [Action Injection](#) to request the dependency.

Within the **Model-View-Controller** pattern, a controller is responsible for the initial processing of the request and instantiation of the model. Generally, business decisions should be performed within the model.

The controller takes the result of the model's processing (if any) and returns either the proper view and its associated view data or the result of the API call. Learn more at [Overview of ASP.NET Core MVC](#) and [Getting started with ASP.NET Core MVC and Visual Studio](#).

The controller is a *UI-level* abstraction. Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned. In well-factored apps, it does not directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.

Defining Actions

Public methods on a controller, except those decorated with the `[NonAction]` attribute, are actions. Parameters on actions are bound to request data and are validated using [model binding](#). Model validation occurs for everything that's model-bound. The `ModelState.IsValid` property value indicates whether model binding and validation succeeded.

Action methods should contain logic for mapping a request to a business concern. Business concerns should typically be represented as services that the controller accesses through [dependency injection](#). Actions then map the result of the business action to an application state.

Actions can return anything, but frequently return an instance of `IActionResult` (or `Task<IActionResult>` for async methods) that produces a response. The action method is responsible for choosing *what kind of response*. The action result *does the responding*.

Controller Helper Methods

Controllers usually inherit from [Controller](#), although this is not required. Deriving from `Controller` provides access to three categories of helper methods:

1. Methods resulting in an empty response body

No `Content-Type` HTTP response header is included, since the response body lacks content to describe.

There are two result types within this category: Redirect and HTTP Status Code.

• HTTP Status Code

This type returns an HTTP status code. A couple helper methods of this type are `BadRequest`, `NotFound`, and `Ok`. For example, `return BadRequest();` produces a 400 status code when executed. When methods such as `BadRequest`, `NotFound`, and `Ok` are overloaded, they no longer qualify as HTTP Status Code responders, since content negotiation is taking place.

• Redirect

This type returns a redirect to an action or destination (using `Redirect`, `LocalRedirect`, `RedirectToAction`, or `RedirectToRoute`). For example, `return RedirectToAction("Complete", new {id = 123});` redirects to `Complete`, passing an anonymous object.

The Redirect result type differs from the HTTP Status Code type primarily in the addition of a `Location` HTTP response header.

2. Methods resulting in a non-empty response body with a predefined content type

Most helper methods in this category include a `ContentType` property, allowing you to set the `Content-Type` response header to describe the response body.

There are two result types within this category: [View](#) and [Formatted Response](#).

• View

This type returns a view which uses a model to render HTML. For example, `return View(customer);` passes a model to the view for data-binding.

• Formatted Response

This type returns JSON or a similar data exchange format to represent an object in a specific manner. For example, `return Json(customer);` serializes the provided object into JSON format.

Other common methods of this type include `File`, `PhysicalFile`, and `VirtualFile`. For example, `return PhysicalFile(customerFilePath, "text/xml");` returns an XML file described by a `Content-Type` response header value of "text/xml".

3. Methods resulting in a non-empty response body formatted in a content type negotiated with the client

This category is better known as **Content Negotiation**. [Content negotiation](#) applies whenever an action returns an `ObjectResult` type or something other than an `IActionResult` implementation. An action that returns a non-`IActionResult` implementation (for example, `object`) also returns a Formatted Response.

Some helper methods of this type include `BadRequest`, `CreatedAtRoute`, and `Ok`. Examples of these methods

include `return BadRequest(modelState);`, `return CreatedAtRoute("routename", values, newobject);`, and `return Ok(value);`, respectively. Note that `BadRequest` and `Ok` perform content negotiation only when passed a value; without being passed a value, they instead serve as HTTP Status Code result types. The `CreatedAtRoute` method, on the other hand, always performs content negotiation since its overloads all require that a value be passed.

Cross-Cutting Concerns

Applications typically share parts of their workflow. Examples include an app that requires authentication to access the shopping cart, or an app that caches data on some pages. To perform logic before or after an action method, use a *filter*. Using [Filters](#) on cross-cutting concerns can reduce duplication, allowing them to follow the [Don't Repeat Yourself \(DRY\) principle](#).

Most filter attributes, such as `[Authorize]`, can be applied at the controller or action level depending upon the desired level of granularity.

Error handling and response caching are often cross-cutting concerns:

- [Error handling](#)
- [Response Caching](#)

Many cross-cutting concerns can be handled using filters or custom [middleware](#).

Routing to Controller Actions

11/28/2017 • 30 min to read • [Edit Online](#)

By [Ryan Nowak](#) and [Rick Anderson](#)

ASP.NET Core MVC uses the Routing [middleware](#) to match the URLs of incoming requests and map them to actions. Routes are defined in startup code or attributes. Routes describe how URL paths should be matched to actions. Routes are also used to generate URLs (for links) sent out in responses.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. See [Mixed routing](#) for more information.

This document will explain the interactions between MVC and routing, and how typical MVC apps make use of routing features. See [Routing](#) for details on advanced routing.

Setting up Routing Middleware

In your *Configure* method you may see code similar to:

```
app.UseMvc(routes =>
{
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

Inside the call to `UseMvc`, `MapRoute` is used to create a single route, which we'll refer to as the `default` route. Most MVC apps will use a route with a template similar to the `default` route.

The route template `"{controller=Home}/{action=Index}/{id?}"` can match a URL path like `/Products/Details/5` and will extract the route values `{ controller = Products, action = Details, id = 5 }` by tokenizing the path. MVC will attempt to locate a controller named `ProductsController` and run the action `Details`:

```
public class ProductsController : Controller
{
    public IActionResult Details(int id) { ... }
}
```

Note that in this example, model binding would use the value of `id = 5` to set the `id` parameter to `5` when invoking this action. See the [Model Binding](#) for more details.

Using the `default` route:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

The route template:

- `{controller=Home}` defines `Home` as the default `controller`
- `{action=Index}` defines `Index` as the default `action`
- `{id?}` defines `id` as optional

Default and optional route parameters do not need to be present in the URL path for a match. See [Route](#)

[Template Reference](#) for a detailed description of route template syntax.

"{controller=Home}/{action=Index}/{id?}" can match the URL path `/` and will produce the route values `{ controller = Home, action = Index }`. The values for `controller` and `action` make use of the default values, `id` does not produce a value since there is no corresponding segment in the URL path. MVC would use these route values to select the `HomeController` and `Index` action:

```
public class HomeController : Controller
{
    public IActionResult Index() { ... }
}
```

Using this controller definition and route template, the `HomeController.Index` action would be executed for any of the following URL paths:

- `/Home/Index/17`
- `/Home/Index`
- `/Home`
- `/`

The convenience method `UseMvcWithDefaultRoute`:

```
app.UseMvcWithDefaultRoute();
```

Can be used to replace:

```
app.UseMvc(routes =>
{
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

`UseMvc` and `UseMvcWithDefaultRoute` add an instance of `RouterMiddleware` to the middleware pipeline. MVC doesn't interact directly with middleware, and uses routing to handle requests. MVC is connected to the routes through an instance of `MvcRouteHandler`. The code inside of `UseMvc` is similar to the following:

```
var routes = new RouteBuilder(app);

// Add connection to MVC, will be hooked up by calls to MapRoute.
routes.DefaultHandler = new MvcRouteHandler(...);

// Execute callback to register routes.
// routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");

// Create route collection and add the middleware.
app.UseRouter(routes.Build());
```

`UseMvc` does not directly define any routes, it adds a placeholder to the route collection for the `attribute` route. The overload `UseMvc(Action<IRouteBuilder>)` lets you add your own routes and also supports attribute routing. `UseMvc` and all of its variations adds a placeholder for the attribute route - attribute routing is always available regardless of how you configure `UseMvc`. `UseMvcWithDefaultRoute` defines a default route and supports attribute routing. The [Attribute Routing](#) section includes more details on attribute routing.

Conventional routing

The `default` route:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

is an example of a *conventional routing*. We call this style *conventional routing* because it establishes a *convention* for URL paths:

- the first path segment maps to the controller name
- the second maps to the action name.
- the third segment is used for an optional `id` used to map to a model entity

Using this `default` route, the URL path `/Products/List` maps to the `ProductsController.List` action, and `/Blog/Article/17` maps to `BlogController.Article`. This mapping is based on the controller and action names **only** and is not based on namespaces, source file locations, or method parameters.

TIP

Using conventional routing with the default route allows you to build the application quickly without having to come up with a new URL pattern for each action you define. For an application with CRUD style actions, having consistency for the URLs across your controllers can help simplify your code and make your UI more predictable.

WARNING

The `id` is defined as optional by the route template, meaning that your actions can execute without the ID provided as part of the URL. Usually what will happen if `id` is omitted from the URL is that it will be set to `0` by model binding, and as a result no entity will be found in the database matching `id == 0`. Attribute routing can give you fine-grained control to make the ID required for some actions and not for others. By convention the documentation will include optional parameters like `id` when they are likely to appear in correct usage.

Multiple routes

You can add multiple routes inside `UseMvc` by adding more calls to `MapRoute`. Doing so allows you to define multiple conventions, or to add conventional routes that are dedicated to a specific action, such as:

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

The `blog` route here is a *dedicated conventional route*, meaning that it uses the conventional routing system, but is dedicated to a specific action. Since `controller` and `action` don't appear in the route template as parameters, they can only have the default values, and thus this route will always map to the action `BlogController.Article`.

Routes in the route collection are ordered, and will be processed in the order they are added. So in this example, the `blog` route will be tried before the `default` route.

NOTE

Dedicated conventional routes often use catch-all route parameters like `{*article}` to capture the remaining portion of the URL path. This can make a route 'too greedy' meaning that it matches URLs that you intended to be matched by other routes. Put the 'greedy' routes later in the route table to solve this.

Fallback

As part of request processing, MVC will verify that the route values can be used to find a controller and action in your application. If the route values don't match an action then the route is not considered a match, and the next route will be tried. This is called *fallback*, and it's intended to simplify cases where conventional routes overlap.

Disambiguating actions

When two actions match through routing, MVC must disambiguate to choose the 'best' candidate or else throw an exception. For example:

```
public class ProductsController : Controller
{
    public IActionResult Edit(int id) { ... }

    [HttpPost]
    public IActionResult Edit(int id, Product product) { ... }
}
```

This controller defines two actions that would match the URL path `/Products/Edit/17` and route data `{ controller = Products, action = Edit, id = 17 }`. This is a typical pattern for MVC controllers where `Edit(int)` shows a form to edit a product, and `Edit(int, Product)` processes the posted form. To make this possible MVC would need to choose `Edit(int, Product)` when the request is an HTTP `POST` and `Edit(int)` when the HTTP verb is anything else.

The `HttpPostAttribute` (`[HttpPost]`) is an implementation of `IActionConstraint` that will only allow the action to be selected when the HTTP verb is `POST`. The presence of an `IActionConstraint` makes the `Edit(int, Product)` a 'better' match than `Edit(int)`, so `Edit(int, Product)` will be tried first.

You will only need to write custom `IActionConstraint` implementations in specialized scenarios, but it's important to understand the role of attributes like `HttpPostAttribute` - similar attributes are defined for other HTTP verbs. In conventional routing it's common for actions to use the same action name when they are part of a `show form -> submit form` workflow. The convenience of this pattern will become more apparent after reviewing the [Understanding IActionConstraint](#) section.

If multiple routes match, and MVC can't find a 'best' route, it will throw an `AmbiguousActionException`.

Route names

The strings `"blog"` and `"default"` in the following examples are route names:

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

The route names give the route a logical name so that the named route can be used for URL generation. This greatly simplifies URL creation when the ordering of routes could make URL generation complicated. Route names must be unique application-wide.

Route names have no impact on URL matching or handling of requests; they are used only for URL generation. [Routing](#) has more detailed information on URL generation including URL generation in MVC-specific helpers.

Attribute routing

Attribute routing uses a set of attributes to map actions directly to route templates. In the following example, `app.UseMvc();` is used in the `Configure` method and no route is passed. The `HomeController` will match a set of URLs similar to what the default route `{controller=Home}/{action=Index}/{id?}` would match:

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    public IActionResult Index()
    {
        return View();
    }
    [Route("Home/About")]
    public IActionResult About()
    {
        return View();
    }
    [Route("Home/Contact")]
    public IActionResult Contact()
    {
        return View();
    }
}
```

The `HomeController.Index()` action will be executed for any of the URL paths `/`, `/Home`, or `/Home/Index`.

NOTE

This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route; the conventional default route handles routes more succinctly. However, attribute routing allows (and requires) precise control of which route templates apply to each action.

With attribute routing the controller name and action names play **no** role in which action is selected. This example will match the same URLs as the previous example.

```

public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    public IActionResult MyIndex()
    {
        return View("Index");
    }
    [Route("Home/About")]
    public IActionResult MyAbout()
    {
        return View("About");
    }
    [Route("Home/Contact")]
    public IActionResult MyContact()
    {
        return View("Contact");
    }
}

```

NOTE

The route templates above don't define route parameters for `action`, `area`, and `controller`. In fact, these route parameters are not allowed in attribute routes. Since the route template is already associated with an action, it wouldn't make sense to parse the action name from the URL.

Attribute routing with `Http[Verb]` attributes

Attribute routing can also make use of the `Http[Verb]` attributes such as `HttpPostAttribute`. All of these attributes can accept a route template. This example shows two actions that match the same route template:

```

[HttpGet("/products")]
public IActionResult ListProducts()
{
    // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
    // ...
}

```

For a URL path like `/products` the `ProductsApi.ListProducts` action will be executed when the HTTP verb is `GET` and `ProductsApi.CreateProduct` will be executed when the HTTP verb is `POST`. Attribute routing first matches the URL against the set of route templates defined by route attributes. Once a route template matches, `IActionConstraint` constraints are applied to determine which actions can be executed.

TIP

When building a REST API, it's rare that you will want to use `[Route(...)]` on an action method. It's better to use the more specific `Http*Verb*Attributes` to be precise about what your API supports. Clients of REST APIs are expected to know what paths and HTTP verbs map to specific logical operations.

Since an attribute route applies to a specific action, it's easy to make parameters required as part of the route template definition. In this example, `id` is required as part of the URL path.

```
public class ProductsApiController : Controller
{
    [HttpGet("/products/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id) { ... }
}
```

The `ProductsApiController.GetProduct(int)` action will be executed for a URL path like `/products/3` but not for a URL path like `/products`. See [Routing](#) for a full description of route templates and related options.

Route Name

The following code defines a *route name* of `Products_List`:

```
public class ProductsApiController : Controller
{
    [HttpGet("/products/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id) { ... }
}
```

Route names can be used to generate a URL based on a specific route. Route names have no impact on the URL matching behavior of routing and are only used for URL generation. Route names must be unique application-wide.

NOTE

Contrast this with the conventional *default route*, which defines the `id` parameter as optional (`{id?}`). This ability to precisely specify APIs has advantages, such as allowing `/products` and `/products/5` to be dispatched to different actions.

Combining routes

To make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual actions. Any route templates defined on the controller are prepended to route templates on the actions. Placing a route attribute on the controller makes **all** actions in the controller use attribute routing.

```
[Route("products")]
public class ProductsApiController : Controller
{
    [HttpGet]
    public IActionResult ListProducts() { ... }

    [HttpGet("{id}")]
    public IActionResult GetProduct(int id) { ... }
}
```

In this example the URL path `/products` can match `ProductsApiController.ListProducts`, and the URL path `/products/5` can match `ProductsApiController.GetProduct(int)`. Both of these actions only match HTTP `GET` because they are decorated with the `HttpGetAttribute`.

Route templates applied to an action that begin with a `/` do not get combined with route templates applied to the controller. This example matches a set of URL paths similar to the *default route*.

```

[Route("Home")]
public class HomeController : Controller
{
    [Route("")] // Combines to define the route template "Home"
    [Route("Index")] // Combines to define the route template "Home/Index"
    [Route("/")] // Does not combine, defines the route template ""
    public IActionResult Index()
    {
        ViewData["Message"] = "Home index";
        var url = Url.Action("Index", "Home");
        ViewData["Message"] = "Home index" + "var url = Url.Action; = " + url;
        return View();
    }

    [Route("About")] // Combines to define the route template "Home/About"
    public IActionResult About()
    {
        return View();
    }
}

```

Ordering attribute routes

In contrast to conventional routes which execute in a defined order, attribute routing builds a tree and matches all routes simultaneously. This behaves as-if the route entries were placed in an ideal ordering; the most specific routes have a chance to execute before the more general routes.

For example, a route like `blog/search/{topic}` is more specific than a route like `blog/{*article}`. Logically speaking the `blog/search/{topic}` route 'runs' first, by default, because that's the only sensible ordering. Using conventional routing, the developer is responsible for placing routes in the desired order.

Attribute routes can configure an order, using the `Order` property of all of the framework provided route attributes. Routes are processed according to an ascending sort of the `Order` property. The default order is `0`. Setting a route using `Order = -1` will run before routes that don't set an order. Setting a route using `Order = 1` will run after default route ordering.

TIP

Avoid depending on `Order`. If your URL-space requires explicit order values to route correctly, then it's likely confusing to clients as well. In general attribute routing will select the correct route with URL matching. If the default order used for URL generation isn't working, using route name as an override is usually simpler than applying the `Order` property.

Token replacement in route templates (`[controller]`, `[action]`, `[area]`)

For convenience, attribute routes support *token replacement* by enclosing a token in square-braces (`[]`). The tokens `[action]`, `[area]`, and `[controller]` will be replaced with the values of the action name, area name, and controller name from the action where the route is defined. In this example the actions can match URL paths as described in the comments:

```

[Route("[controller]/[action]")]
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}

```

Token replacement occurs as the last step of building the attribute routes. The above example will behave the same as the following code:

```

public class ProductsController : Controller
{
    [HttpGet("[controller]/[action]")] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("[controller]/[action]/{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}

```

Attribute routes can also be combined with inheritance. This is particularly powerful combined with token replacement.

```

[Route("api/[controller]")]
public abstract class MyBaseController : Controller { ... }

public class ProductsController : MyBaseController
{
    [HttpGet] // Matches '/api/Products'
    public IActionResult List() { ... }

    [HttpPost("{id}")] // Matches '/api/Products/{id}'
    public IActionResult Edit(int id) { ... }
}

```

Token replacement also applies to route names defined by attribute routes.

`[Route("[controller]/[action]", Name="[controller]_[action]")]` will generate a unique route name for each action.

To match the literal token replacement delimiter `[` or `]`, escape it by repeating the character (`[[` or `]]`).

Multiple Routes

Attribute routing supports defining multiple routes that reach the same action. The most common usage of this is to mimic the behavior of the *default conventional route* as shown in the following example:

```
[Route("[controller]")]
public class ProductsController : Controller
{
    [Route("")] // Matches 'Products'
    [Route("Index")] // Matches 'Products/Index'
    public IActionResult Index()
}

```

Putting multiple route attributes on the controller means that each one will combine with each of the route attributes on the action methods.

```
[Route("Store")]
[Route("[controller]")]
public class ProductsController : Controller
{
    [HttpPost("Buy")] // Matches 'Products/Buy' and 'Store/Buy'
    [HttpPost("Checkout")] // Matches 'Products/Checkout' and 'Store/Checkout'
    public IActionResult Buy()
}

```

When multiple route attributes (that implement `IActionConstraint`) are placed on an action, then each action constraint combines with the route template from the attribute that defined it.

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpPut("Buy")] // Matches PUT 'api/Products/Buy'
    [HttpPost("Checkout")] // Matches POST 'api/Products/Checkout'
    public IActionResult Buy()
}

```

TIP

While using multiple routes on actions can seem powerful, it's better to keep your application's URL space simple and well-defined. Use multiple routes on actions only where needed, for example to support existing clients.

Specifying attribute route optional parameters, default values, and constraints

Attribute routes support the same inline syntax as conventional routes to specify optional parameters, default values, and constraints.

```
[HttpPost("product/{id:int}")]
public IActionResult ShowProduct(int id)
{
    // ...
}

```

See [Route Template Reference](#) for a detailed description of route template syntax.

Custom route attributes using `IRouteTemplateProvider`

All of the route attributes provided in the framework (`[Route(...)]`, `[HttpGet(...)]`, etc.) implement the `IRouteTemplateProvider` interface. MVC looks for attributes on controller classes and action methods when the app starts and uses the ones that implement `IRouteTemplateProvider` to build the initial set of routes.

You can implement `IRouteTemplateProvider` to define your own route attributes. Each `IRouteTemplateProvider` allows you to define a single route with a custom route template, order, and name:

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
    public string Template => "api/[controller]";

    public int? Order { get; set; }

    public string Name { get; set; }
}
```

The attribute from the above example automatically sets the `Template` to `"api/[controller]"` when `[MyApiController]` is applied.

Using Application Model to customize attribute routes

The *application model* is an object model created at startup with all of the metadata used by MVC to route and execute your actions. The *application model* includes all of the data gathered from route attributes (through `IRouteTemplateProvider`). You can write *conventions* to modify the application model at startup time to customize how routing behaves. This section shows a simple example of customizing routing using application model.

```

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;
using System.Text;
public class NamespaceRoutingConvention : IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
            selector.AttributeRouteModel != null);

        if (hasRouteAttributes)
        {
            // This controller manually defined some routes, so treat this
            // as an override and not apply the convention here.
            return;
        }

        // Use the namespace and controller name to infer a route for the controller.
        //
        // Example:
        //
        // controller.ControllerTypeInfo -> "My.Application.Admin.UsersController"
        // baseNamespace -> "My.Application"
        //
        // template => "Admin/[controller]"
        //
        // This makes your routes roughly line up with the folder structure of your project.
        //
        var namespace = controller.ControllerType.Namespace;

        var template = new StringBuilder();
        template.Append(namespace, _baseNamespace.Length + 1,
            namespace.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
        template.Append("/[controller]");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
        }
    }
}

```

Mixed routing: Attribute routing vs conventional routing

MVC applications can mix the use of conventional routing and attribute routing. It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. Actions that define attribute routes cannot be reached through the conventional routes and vice-versa. **Any** route attribute on the controller makes all actions in the controller attribute routed.

NOTE

What distinguishes the two types of routing systems is the process applied after a URL matches a route template. In conventional routing, the route values from the match are used to choose the action and controller from a lookup table of all conventional routed actions. In attribute routing, each template is already associated with an action, and no further lookup is needed.

URL Generation

MVC applications can use routing's URL generation features to generate URL links to actions. Generating URLs eliminates hardcoding URLs, making your code more robust and maintainable. This section focuses on the URL generation features provided by MVC and will only cover basics of how URL generation works. See [Routing](#) for a detailed description of URL generation.

The `IUrlHelper` interface is the underlying piece of infrastructure between MVC and routing for URL generation. You'll find an instance of `IUrlHelper` available through the `Url` property in controllers, views, and view components.

In this example, the `IUrlHelper` interface is used through the `Controller.Url` property to generate a URL to another action.

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return Content($"Go check out {url}, it's really great.");
    }

    public IActionResult Destination()
    {
        return View();
    }
}
```

If the application is using the default conventional route, the value of the `url` variable will be the URL path string `/UrlGeneration/Destination`. This URL path is created by routing by combining the route values from the current request (ambient values), with the values passed to `Url.Action` and substituting those values into the route template:

```
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action = "Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

Each route parameter in the route template has its value substituted by matching names with the values and ambient values. A route parameter that does not have a value can use a default value if it has one, or be skipped if it is optional (as in the case of `id` in this example). URL generation will fail if any required route parameter doesn't have a corresponding value. If URL generation fails for a route, the next route is tried until all routes have been tried or a match is found.

The example of `Url.Action` above assumes conventional routing, but URL generation works similarly with

attribute routing, though the concepts are different. With conventional routing, the route values are used to expand a template, and the route values for `controller` and `action` usually appear in that template - this works because the URLs matched by routing adhere to a *convention*. In attribute routing, the route values for `controller` and `action` are not allowed to appear in the template - they are instead used to look up which template to use.

This example uses attribute routing:

```
// In Startup class
public void Configure(IApplicationBuilder app)
{
    app.UseMvc();
}
```

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination"); // Generates /custom/url/to/destination
        return Content($"Go check out {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination() {
        return View();
    }
}
```

MVC builds a lookup table of all attribute routed actions and will match the `controller` and `action` values to select the route template to use for URL generation. In the sample above, `custom/url/to/destination` is generated.

Generating URLs by action name

`Url.Action (IUrlHelper . Action)` and all related overloads all are based on that idea that you want to specify what you're linking to by specifying a controller name and action name.

NOTE

When using `Url.Action`, the current route values for `controller` and `action` are specified for you - the value of `controller` and `action` are part of both *ambient values* and *values*. The method `Url.Action`, always uses the current values of `action` and `controller` and will generate a URL path that routes to the current action.

Routing attempts to use the values in ambient values to fill in information that you didn't provide when generating a URL. Using a route like `{a}/{b}/{c}/{d}` and ambient values

`{ a = Alice, b = Bob, c = Carol, d = David }`, routing has enough information to generate a URL without any additional values - since all route parameters have a value. If you added the value `{ d = Donovan }`, the value `{ d = David }` would be ignored, and the generated URL path would be `Alice/Bob/Carol/Donovan`.

WARNING

URL paths are hierarchical. In the example above, if you added the value `{ c = Cheryl }`, both of the values `{ c = Carol, d = David }` would be ignored. In this case we no longer have a value for `d` and URL generation will fail. You would need to specify the desired value of `c` and `d`. You might expect to hit this problem with the default route (`{controller}/{action}/{id?}`) - but you will rarely encounter this behavior in practice as `Url.Action` will always explicitly specify a `controller` and `action` value.

Longer overloads of `Url.Action` also take an additional *route values* object to provide values for route parameters other than `controller` and `action`. You will most commonly see this used with `id` like `Url.Action("Buy", "Products", new { id = 17 })`. By convention the *route values* object is usually an object of anonymous type, but it can also be an `IDictionary<>` or a *plain old .NET object*. Any additional route values that don't match route parameters are put in the query string.

```
using Microsoft.AspNetCore.Mvc;

public class TestController : Controller
{
    public IActionResult Index()
    {
        // Generates /Products/Buy/17?color=red
        var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
        return Content(url);
    }
}
```

TIP

To create an absolute URL, use an overload that accepts a `protocol` :

```
Url.Action("Buy", "Products", new { id = 17 }, protocol: Request.Scheme)
```

Generating URLs by route

The code above demonstrated generating a URL by passing in the controller and action name. `UrlHelper` also provides the `Url.RouteUrl` family of methods. These methods are similar to `Url.Action`, but they do not copy the current values of `action` and `controller` to the route values. The most common usage is to specify a route name to use a specific route to generate the URL, generally *without* specifying a controller or action name.

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route"); // Generates /custom/url/to/destination
        return Content($"See {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination", Name = "Destination_Route")]
    public IActionResult Destination() {
        return View();
    }
}
```

Generating URLs in HTML

`IHtmlHelper` provides the `HtmlHelper` methods `Html.BeginForm` and `Html.ActionLink` to generate `<form>` and `<a>` elements respectively. These methods use the `Url.Action` method to generate a URL and they accept similar arguments. The `Url.RouteUrl` companions for `HtmlHelper` are `Html.BeginRouteForm` and `Html.RouteLink` which have similar functionality.

TagHelpers generate URLs through the `form` TagHelper and the `<a>` TagHelper. Both of these use `IUrlHelper` for their implementation. See [Working with Forms](#) for more information.

Inside views, the `IUrlHelper` is available through the `Url` property for any ad-hoc URL generation not covered by the above.

Generating URLs in Action Results

The examples above have shown using `IUrlHelper` in a controller, while the most common usage in a controller is to generate a URL as part of an action result.

The `ControllerBase` and `Controller` base classes provide convenience methods for action results that reference another action. One typical usage is to redirect after accepting user input.

```
public Task<IActionResult> Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        return RedirectToAction("Index");
    }
}
```

The action results factory methods follow a similar pattern to the methods on `IUrlHelper`.

Special case for dedicated conventional routes

Conventional routing can use a special kind of route definition called a *dedicated conventional route*. In the example below, the route named `blog` is a dedicated conventional route.

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

Using these route definitions, `Url.Action("Index", "Home")` will generate the URL path `/` with the `default` route, but why? You might guess the route values `{ controller = Home, action = Index }` would be enough to generate a URL using `blog`, and the result would be `/blog?action=Index&controller=Home`.

Dedicated conventional routes rely on a special behavior of default values that don't have a corresponding route parameter that prevents the route from being "too greedy" with URL generation. In this case the default values are `{ controller = Blog, action = Article }`, and neither `controller` nor `action` appears as a route parameter. When routing performs URL generation, the values provided must match the default values. URL generation using `blog` will fail because the values `{ controller = Home, action = Index }` don't match `{ controller = Blog, action = Article }`. Routing then falls back to try `default`, which succeeds.

Areas

[Areas](#) are an MVC feature used to organize related functionality into a group as a separate routing-namespace (for controller actions) and folder structure (for views). Using areas allows an application to have multiple controllers with the same name - as long as they have different *areas*. Using areas creates a hierarchy for the

purpose of routing by adding another route parameter, `area` to `controller` and `action`. This section will discuss how routing interacts with areas - see [Areas](#) for details about how areas are used with views.

The following example configures MVC to use the default conventional route and an *area route* for an area named `Blog`:

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

When matching a URL path like `/Manage/Users/AddUser`, the first route will produce the route values `{ area = Blog, controller = Users, action = AddUser }`. The `area` route value is produced by a default value for `area`, in fact the route created by `MapAreaRoute` is equivalent to the following:

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog_route", "Manage/{controller}/{action}/{id?}",
        defaults: new { area = "Blog" }, constraints: new { area = "Blog" });
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

`MapAreaRoute` creates a route using both a default value and constraint for `area` using the provided area name, in this case `Blog`. The default value ensures that the route always produces `{ area = Blog, ... }`, the constraint requires the value `{ area = Blog, ... }` for URL generation.

TIP

Conventional routing is order-dependent. In general, routes with areas should be placed earlier in the route table as they are more specific than routes without an area.

Using the above example, the route values would match the following action:

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

The `AreaAttribute` is what denotes a controller as part of an area, we say that this controller is in the `Blog` area. Controllers without an `[Area]` attribute are not members of any area, and will **not** match when the `area` route value is provided by routing. In the following example, only the first controller listed can match the route values `{ area = Blog, controller = Users, action = AddUser }`.

```

using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}

```

```

using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}

```

```

using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}

```

NOTE

The namespace of each controller is shown here for completeness - otherwise the controllers would have a naming conflict and generate a compiler error. Class namespaces have no effect on MVC's routing.

The first two controllers are members of areas, and only match when their respective area name is provided by the `area` route value. The third controller is not a member of any area, and can only match when no value for `area` is provided by routing.

NOTE

In terms of matching *no value*, the absence of the `area` value is the same as if the value for `area` were null or the empty string.

When executing an action inside an area, the route value for `area` will be available as an *ambient value* for routing to use for URL generation. This means that by default areas act *sticky* for URL generation as demonstrated by the following sample.

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("duck_route", "Duck",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default", "Manage/{controller=Home}/{action=Index}/{id?}");
});
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
    public class UsersController : Controller
    {
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area
            var url = Url.Action("Index", "Home");
            // returns /Manage
            return Content(url);
        }

        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area
            var url = Url.Action("Index", "Home", new { area = "" });
            // returns /Manage/Home/Index
            return Content(url);
        }
    }
}
```

Understanding IActionResult

NOTE

This section is a deep-dive on framework internals and how MVC chooses an action to execute. A typical application won't need a custom `IActionConstraint`

You have likely already used `IActionConstraint` even if you're not familiar with the interface. The `[HttpGet]` Attribute and similar `[Http-VERB]` attributes implement `IActionConstraint` in order to limit the execution of an action method.

```

public class ProductsController : Controller
{
    [HttpGet]
    public IActionResult Edit() { }

    public IActionResult Edit(...) { }
}

```

Assuming the default conventional route, the URL path `/Products/Edit` would produce the values `{ controller = Products, action = Edit }`, which would match **both** of the actions shown here. In `IActionConstraint` terminology we would say that both of these actions are considered candidates - as they both match the route data.

When the `HttpGetAttribute` executes, it will say that `Edit()` is a match for `GET` and is not a match for any other HTTP verb. The `Edit(...)` action doesn't have any constraints defined, and so will match any HTTP verb. So assuming a `POST` - only `Edit(...)` matches. But, for a `GET` both actions can still match - however, an action with an `IActionConstraint` is always considered *better* than an action without. So because `Edit()` has `[HttpGet]` it is considered more specific, and will be selected if both actions can match.

Conceptually, `IActionConstraint` is a form of *overloading*, but instead of overloading methods with the same name, it is overloading between actions that match the same URL. Attribute routing also uses `IActionConstraint` and can result in actions from different controllers both being considered candidates.

Implementing IActionConstraint

The simplest way to implement an `IActionConstraint` is to create a class derived from `System.Attribute` and place it on your actions and controllers. MVC will automatically discover any `IActionConstraint` that are applied as attributes. You can use the application model to apply constraints, and this is probably the most flexible approach as it allows you to metaprogram how they are applied.

In the following example a constraint chooses an action based on a *country code* from the route data. The [full sample on GitHub](#).

```

public class CountrySpecificAttribute : Attribute, IActionConstraint
{
    private readonly string _countryCode;

    public CountrySpecificAttribute(string countryCode)
    {
        _countryCode = countryCode;
    }

    public int Order
    {
        get
        {
            return 0;
        }
    }

    public bool Accept(ActionConstraintContext context)
    {
        return string.Equals(
            context.RouteContext.RouteData.Values["country"].ToString(),
            _countryCode,
            StringComparison.OrdinalIgnoreCase);
    }
}

```

You are responsible for implementing the `Accept` method and choosing an 'Order' for the constraint to execute.

In this case, the `Accept` method returns `true` to denote the action is a match when the `country` route value matches. This is different from a `RouteValueAttribute` in that it allows fallback to a non-attributed action. The sample shows that if you define an `en-US` action then a country code like `fr-FR` will fall back to a more generic controller that does not have `[CountrySpecific(...)]` applied.

The `order` property decides which *stage* the constraint is part of. Action constraints run in groups based on the `order`. For example, all of the framework provided HTTP method attributes use the same `order` value so that they run in the same stage. You can have as many stages as you need to implement your desired policies.

TIP

To decide on a value for `order` think about whether or not your constraint should be applied before HTTP methods. Lower numbers run first.

File uploads in ASP.NET Core

9/22/2017 • 8 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET MVC actions support uploading of one or more files using simple model binding for smaller files or streaming for larger files.

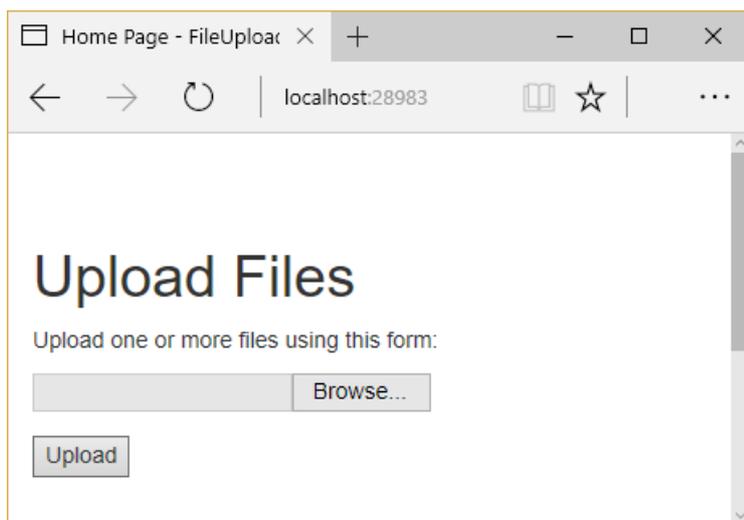
[View or download sample from GitHub](#)

Uploading small files with model binding

To upload small files, you can use a multi-part HTML form or construct a POST request using JavaScript. An example form using Razor, which supports multiple uploaded files, is shown below:

```
<form method="post" enctype="multipart/form-data" asp-controller="UploadFiles" asp-action="Index">
  <div class="form-group">
    <div class="col-md-10">
      <p>Upload one or more files using this form:</p>
      <input type="file" name="files" multiple />
    </div>
  </div>
  <div class="form-group">
    <div class="col-md-10">
      <input type="submit" value="Upload" />
    </div>
  </div>
</form>
```

In order to support file uploads, HTML forms must specify an `enctype` of `multipart/form-data`. The `files` input element shown above supports uploading multiple files. Omit the `multiple` attribute on this input element to allow just a single file to be uploaded. The above markup renders in a browser as:



The individual files uploaded to the server can be accessed through [Model Binding](#) using the [IFormFile](#) interface.

`IFormFile` has the following structure:

```

public interface IFormFile
{
    string ContentType { get; }
    string ContentDisposition { get; }
    IDictionary Headers { get; }
    long Length { get; }
    string Name { get; }
    string FileName { get; }
    Stream OpenReadStream();
    void CopyTo(Stream target);
    Task CopyToAsync(Stream target, CancellationToken cancellationToken = null);
}

```

WARNING

Don't rely on or trust the `FileName` property without validation. The `FileName` property should only be used for display purposes.

When uploading files using model binding and the `IFormFile` interface, the action method can accept either a single `IFormFile` or an `IEnumerable<IFormFile>` (or `List<IFormFile>`) representing several files. The following example loops through one or more uploaded files, saves them to the local file system, and returns the total number and size of files uploaded.

Warning: The following code uses `GetTempFileName`, which throws an `IOException` if more than 65535 files are created without deleting previous temporary files. A real app should either delete temporary files or use `GetTempPath` and `GetRandomFileName` to create temporary file names. The 65535 files limit is per server, so another app on the server can use up all 65535 files.

```

[HttpPost("UploadFiles")]
public async Task<IActionResult> Post(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    // full path to file in temp location
    var filePath = Path.GetTempFileName();

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            using (var stream = new FileStream(filePath, FileMode.Create))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size, filePath });
}

```

Files uploaded using the `IFormFile` technique are buffered in memory or on disk on the web server before being processed. Inside the action method, the `IFormFile` contents are accessible as a stream. In addition to the local file system, files can be streamed to [Azure Blob storage](#) or [Entity Framework](#).

To store binary file data in a database using Entity Framework, define a property of type `byte[]` on the entity:

```
public class ApplicationUser : IdentityUser
{
    public byte[] AvatarImage { get; set; }
}
```

Specify a viewmodel property of type `IFormFile` :

```
public class RegisterViewModel
{
    // other properties omitted

    public IFormFile AvatarImage { get; set; }
}
```

NOTE

`IFormFile` can be used directly as an action method parameter or as a viewmodel property, as shown above.

Copy the `IFormFile` to a stream and save it to the byte array:

```
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser {
            UserName = model.Email,
            Email = model.Email
        };
        using (var memoryStream = new MemoryStream())
        {
            await model.AvatarImage.CopyToAsync(memoryStream);
            user.AvatarImage = memoryStream.ToArray();
        }
        // additional logic omitted

        // Don't rely on or trust the model.AvatarImage.FileName property
        // without validation.
    }
}
```

NOTE

Use caution when storing binary data in relational databases, as it can adversely impact performance.

Uploading large files with streaming

If the size or frequency of file uploads is causing resource problems for the app, consider streaming the file upload rather than buffering it in its entirety, as the model binding approach shown above does. While using `IFormFile` and model binding is a much simpler solution, streaming requires a number of steps to implement properly.

NOTE

Any single buffered file exceeding 64KB will be moved from RAM to a temp file on disk on the server. The resources (disk, RAM) used by file uploads depend on the number and size of concurrent file uploads. Streaming is not so much about perf, it's about scale. If you try to buffer too many uploads, your site will crash when it runs out of memory or disk space.

The following example demonstrates using JavaScript/Angular to stream to a controller action. The file's antiforgery token is generated using a custom filter attribute and passed in HTTP headers instead of in the request body. Because the action method processes the uploaded data directly, model binding is disabled by another filter. Within the action, the form's contents are read using a `MultipartReader`, which reads each individual `MultipartSection`, processing the file or storing the contents as appropriate. Once all sections have been read, the action performs its own model binding.

The initial action loads the form and saves an antiforgery token in a cookie (via the `GenerateAntiforgeryTokenCookieForAjax` attribute):

```
[HttpGet]
[GenerateAntiforgeryTokenCookieForAjax]
public IActionResult Index()
{
    return View();
}
```

The attribute uses ASP.NET Core's built-in [Antiforgery](#) support to set a cookie with a request token:

```
public class GenerateAntiforgeryTokenCookieForAjaxAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext context)
    {
        var antiforgery = context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // We can send the request token as a JavaScript-readable cookie,
        // and Angular will use it by default.
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);
        context.HttpContext.Response.Cookies.Append(
            "XSRF-TOKEN",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }
}
```

Angular automatically passes an antiforgery token in a request header named `X-XSRF-TOKEN`. The ASP.NET Core MVC app is configured to refer to this header in its configuration in `Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    // Angular's default header name for sending the XSRF token.
    services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");

    services.AddMvc();
}
```

The `DisableFormValueModelBinding` attribute, shown below, is used to disable model binding for the `Upload` action method.

```

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}

```

Since model binding is disabled, the `Upload` action method doesn't accept parameters. It works directly with the `Request` property of `ControllerBase`. A `MultipartReader` is used to read each section. The file is saved with a GUID filename and the key/value data is stored in a `KeyValueAccumulator`. Once all sections have been read, the contents of the `KeyValueAccumulator` are used to bind the form data to a model type.

The complete `Upload` method is shown below:

Warning: The following code uses `GetTempFileName`, which throws an `IOException` if more than 65535 files are created without deleting previous temporary files. A real app should either delete temporary files or use `GetTempPath` and `GetRandomFileName` to create temporary file names. The 65535 files limit is per server, so another app on the server can use up all 65535 files.

```

// 1. Disable the form value model binding here to take control of handling
// potentially large files.
// 2. Typically antiforgery tokens are sent in request body, but since we
// do not want to read the request body early, the tokens are made to be
// sent via headers. The antiforgery token filter first looks for tokens
// in the request header and then falls back to reading the body.
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Upload()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        return BadRequest($"Expected a multipart request, but got {Request.ContentType}");
    }

    // Used to accumulate all the form url encoded key value pairs in the
    // request.
    var formAccumulator = new KeyValueAccumulator();
    string targetFilePath = null;

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();
    while (section != null)
    {
        ContentDispositionHeaderValue contentDisposition;
        var hasContentDispositionHeader = ContentDispositionHeaderValue.TryParse(section.ContentDisposition,
            out contentDisposition);

        if (hasContentDispositionHeader)
        {
            if (MultipartRequestHelper.HasFileContentDisposition(contentDisposition))
            {

```

```

        targetFilePath = Path.GetTempFileName();
        using (var targetStream = System.IO.File.Create(targetFilePath))
        {
            await section.Body.CopyToAsync(targetStream);

            _logger.LogInformation($"Copied the uploaded file '{targetFilePath}'");
        }
    }
}
else if (MultipartRequestHelper.HasFormDataContentDisposition(contentDisposition))
{
    // Content-Disposition: form-data; name="key"
    //
    // value

    // Do not limit the key name length here because the
    // multipart headers length limit is already in effect.
    var key = HeaderUtilities.RemoveQuotes(contentDisposition.Name);
    var encoding = GetEncoding(section);
    using (var streamReader = new StreamReader(
        section.Body,
        encoding,
        detectEncodingFromByteOrderMarks: true,
        bufferSize: 1024,
        leaveOpen: true))
    {
        // The value length limit is enforced by MultipartBodyLengthLimit
        var value = await streamReader.ReadToEndAsync();
        if (String.Equals(value, "undefined", StringComparison.OrdinalIgnoreCase))
        {
            value = String.Empty;
        }
        formAccumulator.Append(key, value);

        if (formAccumulator.ValueCount > _defaultFormOptions.ValueCountLimit)
        {
            throw new InvalidDataException($"Form key count limit
{_defaultFormOptions.ValueCountLimit} exceeded.");
        }
    }
}
}

// Drains any remaining section body that has not been consumed and
// reads the headers for the next section.
section = await reader.ReadNextSectionAsync();
}

// Bind form data to a model
var user = new User();
var formValueProvider = new FormValueProvider(
    BindingSource.Form,
    new FormCollection(formAccumulator.GetResults()),
    CultureInfo.CurrentCulture);

var bindingSuccessful = await TryUpdateModelAsync(user, prefix: "",
    valueProvider: formValueProvider);
if (!bindingSuccessful)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
}

var uploadedData = new UploadedData()
{
    Name = user.Name,
    Age = user.Age,
    Zipcode = user.Zipcode
};

```

```
        zipcode = user.Zipcode,
        FilePath = targetFilePath
    };
    return Json(uploadedData);
}
```

Troubleshooting

Below are some common problems encountered when working with uploading files and their possible solutions.

Unexpected Not Found error with IIS

The following error indicates your file upload exceeds the server's configured `maxAllowedContentLength`:

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds the request content length.
```

The default setting is `30000000`, which is approximately 28.6MB. The value can be customized by editing `web.config`:

```
<system.webServer>
  <security>
    <requestFiltering>
      <!-- This will handle requests up to 50MB -->
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

This setting only applies to IIS. The behavior doesn't occur by default when hosting on Kestrel. For more information, see [Request Limits <requestLimits>](#).

Null Reference Exception with IFormFile

If your controller is accepting uploaded files using `IFormFile` but you find that the value is always null, confirm that your HTML form is specifying an `enctype` value of `multipart/form-data`. If this attribute is not set on the `<form>` element, the file upload will not occur and any bound `IFormFile` arguments will be null.

Dependency injection into controllers

11/29/2017 • 5 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET Core MVC controllers should request their dependencies explicitly via their constructors. In some instances, individual controller actions may require a service, and it may not make sense to request at the controller level. In this case, you can also choose to inject a service as a parameter on the action method.

[View or download sample code](#) ([how to download](#))

Dependency Injection

Dependency injection is a technique that follows the [Dependency Inversion Principle](#), allowing for applications to be composed of loosely coupled modules. ASP.NET Core has built-in support for [dependency injection](#), which makes applications easier to test and maintain.

Constructor Injection

ASP.NET Core's built-in support for constructor-based dependency injection extends to MVC controllers. By simply adding a service type to your controller as a constructor parameter, ASP.NET Core will attempt to resolve that type using its built in service container. Services are typically, but not always, defined using interfaces. For example, if your application has business logic that depends on the current time, you can inject a service that retrieves the time (rather than hard-coding it), which would allow your tests to pass in implementations that use a set time.

```
using System;

namespace ControllerDI.Interfaces
{
    public interface IDateTime
    {
        DateTime Now { get; }
    }
}
```

Implementing an interface like this one so that it uses the system clock at runtime is trivial:

```
using System;
using ControllerDI.Interfaces;

namespace ControllerDI.Services
{
    public class SystemDateTime : IDateTime
    {
        public DateTime Now
        {
            get { return DateTime.Now; }
        }
    }
}
```

With this in place, we can use the service in our controller. In this case, we have added some logic to the

[HomeController](#)

[Index](#)

method to display a greeting to the user based on the time of day.

```

using ControllerDI.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace ControllerDI.Controllers
{
    public class HomeController : Controller
    {
        private readonly IDateTime _dateTime;

        public HomeController(IDateTime dateTime)
        {
            _dateTime = dateTime;
        }

        public IActionResult Index()
        {
            var serverTime = _dateTime.Now;
            if (serverTime.Hour < 12)
            {
                ViewData["Message"] = "It's morning here - Good Morning!";
            }
            else if (serverTime.Hour < 17)
            {
                ViewData["Message"] = "It's afternoon here - Good Afternoon!";
            }
            else
            {
                ViewData["Message"] = "It's evening here - Good Evening!";
            }
            return View();
        }
    }
}

```

If we run the application now, we will most likely encounter an error:

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'ControllerDI.Interfaces.IDateTime' while attempting to activate 'ControllerDI.Controllers.HomeController'.
 Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type requiredBy, Boolean isDefaultParameterRequired)

This error occurs when we have not configured a service in the `ConfigureServices` method in our `Startup` class. To specify that requests for `IDateTime` should be resolved using an instance of `SystemDateTime`, add the highlighted line in the listing below to your `ConfigureServices` method:

```

public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IDateTime, SystemDateTime>();
}

```

NOTE

This particular service could be implemented using any of several different lifetime options (`Transient`, `Scoped`, or `Singleton`). See [Dependency Injection](#) to understand how each of these scope options will affect the behavior of your service.

Once the service has been configured, running the application and navigating to the home page should display the time-based message as expected:



A Message From The Server

It's afternoon here - Good Afternoon!

TIP

See [Testing Controller Logic](http://deviq.com/explicit-dependencies-principle/) to learn how to explicitly request dependencies <http://deviq.com/explicit-dependencies-principle/> in controllers makes code easier to test.

ASP.NET Core's built-in dependency injection supports having only a single constructor for classes requesting services. If you have more than one constructor, you may get an exception stating:

```
An unhandled exception occurred while processing the request.
```

```
InvalidOperationException: Multiple constructors accepting all given argument types have been found in type 'ControllerDI.Controllers.HomeController'. There should only be one applicable constructor.
Microsoft.Extensions.DependencyInjection.ActivatorUtilities.FindApplicableConstructor(Type instanceType,
Type[] argumentTypes, ConstructorInfo& matchingConstructor, Nullable`1[]& parameterMap)
```

As the error message states, you can correct this problem having just a single constructor. You can also [replace the default dependency injection support with a third party implementation](#), many of which support multiple constructors.

Action Injection with FromServices

Sometimes you don't need a service for more than one action within your controller. In this case, it may make sense to inject the service as a parameter to the action method. This is done by marking the parameter with the attribute `[FromServices]` as shown here:

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = "Currently on the server the time is " + dateTime.Now;

    return View();
}
```

Accessing Settings from a Controller

Accessing application or configuration settings from within a controller is a common pattern. This access should use the Options pattern described in [configuration](#). You generally should not request settings directly from your controller using dependency injection. A better approach is to request an `IOptions<T>` instance, where `T` is the configuration class you need.

To work with the options pattern, you need to create a class that represents the options, such as this one:

```

namespace ControllerDI.Model
{
    public class SampleWebSettings
    {
        public string Title { get; set; }
        public int Updates { get; set; }
    }
}

```

Then you need to configure the application to use the options model and add your configuration class to the services collection in `ConfigureServices` :

```

public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("samplewebsettings.json");
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; set; }

// This method gets called by the runtime. Use this method to add services to the container.
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    // Required to use the Options<T> pattern
    services.AddOptions();

    // Add settings from configuration
    services.Configure<SampleWebSettings>(Configuration);

    // Uncomment to add settings from code
    //services.Configure<SampleWebSettings>(settings =>
    //{
    //    settings.Updates = 17;
    //});

    services.AddMvc();

    // Add application services.
    services.AddTransient<IDateTime, SystemDateTime>();
}

```

NOTE

In the above listing, we are configuring the application to read the settings from a JSON-formatted file. You can also configure the settings entirely in code, as is shown in the commented code above. See [Configuration](#) for further configuration options.

Once you've specified a strongly-typed configuration object (in this case, `SampleWebSettings`) and added it to the services collection, you can request it from any Controller or Action method by requesting an instance of `IOptions<T>` (in this case, `IOptions<SampleWebSettings>`). The following code shows how one would request the settings from a controller:

```
public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}
```

Following the Options pattern allows settings and configuration to be decoupled from one another, and ensures the controller is following [separation of concerns](#), since it doesn't need to know how or where to find the settings information. It also makes the controller easier to unit test [Testing Controller Logic](#), since there is no [static cling](#) or direct instantiation of settings classes within the controller class.

Testing controller logic in ASP.NET Core

10/13/2017 • 17 min to read • [Edit Online](#)

By [Steve Smith](#)

Controllers in ASP.NET MVC apps should be small and focused on user-interface concerns. Large controllers that deal with non-UI concerns are more difficult to test and maintain.

[View or download sample from GitHub](#)

Testing controllers

Controllers are a central part of any ASP.NET Core MVC application. As such, you should have confidence they behave as intended for your app. Automated tests can provide you with this confidence and can detect errors before they reach production. It's important to avoid placing unnecessary responsibilities within your controllers and ensure your tests focus only on controller responsibilities.

Controller logic should be minimal and not be focused on business logic or infrastructure concerns (for example, data access). Test controller logic, not the framework. Test how the controller *behaves* based on valid or invalid inputs. Test controller responses based on the result of the business operation it performs.

Typical controller responsibilities:

- Verify `ModelState.IsValid`.
- Return an error response if `ModelState` is invalid.
- Retrieve a business entity from persistence.
- Perform an action on the business entity.
- Save the business entity to persistence.
- Return an appropriate `ActionResult`.

Unit testing

[Unit testing](#) involves testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action is tested, not the behavior of its dependencies or of the framework itself. As you unit test your controller actions, make sure you focus only on its behavior. A controller unit test avoids things like [filters](#), [routing](#), or [model binding](#). By focusing on testing just one thing, unit tests are generally simple to write and quick to run. A well-written set of unit tests can be run frequently without much overhead. However, unit tests do not detect issues in the interaction between components, which is the purpose of [integration testing](#).

If you're writing custom filters, routes, etc, you should unit test them, but not as part of your tests on a particular controller action. They should be tested in isolation.

TIP

[Create and run unit tests with Visual Studio.](#)

To demonstrate unit testing, review the following controller. It displays a list of brainstorming sessions and allows new brainstorming sessions to be created with a POST:

```

using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class HomeController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public HomeController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index()
        {
            var sessionList = await _sessionRepository.ListAsync();

            var model = sessionList.Select(session => new StormSessionViewModel()
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
}

```

The controller is following the [explicit dependencies principle](#), expecting dependency injection to provide it with an instance of `IBrainstormSessionRepository`. This makes it fairly easy to test using a mock object framework, like [Moq](#). The `HTTP GET Index` method has no looping or branching and only calls one method. To test this `Index` method,

we need to verify that a `ViewResult` is returned, with a `ViewModel` from the repository's `List` method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class HomeControllerTests
    {
        [Fact]
        public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
        {
            // Arrange
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
            var controller = new HomeController(mockRepo.Object);

            // Act
            var result = await controller.Index();

            // Assert
            var viewResult = Assert.IsType<ViewResult>(result);
            var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
                viewResult.ViewData.Model);
            Assert.Equal(2, model.Count());
        }

        private List<BrainstormSession> GetTestSessions()
        {
            var sessions = new List<BrainstormSession>();
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 2),
                Id = 1,
                Name = "Test One"
            });
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 1),
                Id = 2,
                Name = "Test Two"
            });
            return sessions;
        }
    }
}
```

The `HomeController` `HTTP POST Index` method (shown above) should verify:

- The action method returns a Bad Request `ViewResult` with the appropriate data when `ModelState.IsValid` is `false`
- The `Add` method on the repository is called and a `RedirectToActionResult` is returned with the correct arguments when `ModelState.IsValid` is true.

Invalid model state can be tested by adding errors using `AddModelError` as shown in the first test below.

```

[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}

```

The first test confirms when `ModelState` is not valid, the same `ViewResult` is returned as for a `GET` request. Note that the test doesn't attempt to pass in an invalid model. That wouldn't work anyway since model binding isn't running (though an [integration test](#) would use exercise model binding). In this case, model binding is not being tested. These unit tests are only testing what the code in the action method does.

The second test verifies that when `ModelState` is valid, a new `BrainstormSession` is added (via the repository), and the method returns a `RedirectToActionResult` with the expected properties. Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows it to be verified in the test. This is done with the call to `mockRepo.Verify`, which will fail the test if the expected method was not called.

NOTE

The Moq library used in this sample makes it easy to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about [customizing Mock behavior with Moq](#).

Another controller in the app displays information related to a particular brainstorming session. It includes some logic to deal with invalid id values:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class SessionController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public SessionController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index(int? id)
        {
            if (!id.HasValue)
            {
                return RedirectToAction(actionName: nameof(Index), controllerName: "Home");
            }

            var session = await _sessionRepository.GetByIdAsync(id.Value);
            if (session == null)
            {
                return Content("Session not found.");
            }

            var viewModel = new StormSessionViewModel()
            {
                DateCreated = session.DateCreated,
                Name = session.Name,
                Id = session.Id
            };

            return View(viewModel);
        }
    }
}

```

The controller action has three cases to test, one for each `return` statement:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class SessionControllerTests
    {
        [Fact]
        public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
        {
            // Arrange
            var controller = new SessionController(sessionRepository: null);

            // Act

```

```

        var result = await controller.Index(id: null);

        // Assert
        var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
        Assert.Equal("Home", redirectToActionResult.ControllerName);
        Assert.Equal("Index", redirectToActionResult.ActionName);
    }

    [Fact]
    public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult((BrainstormSession)null));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var contentResult = Assert.IsType<ContentResult>(result);
        Assert.Equal("Session not found.", contentResult.Content);
    }

    [Fact]
    public async Task IndexReturnsViewResultWithStormSessionViewModel()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult(GetTestSessions().FirstOrDefault(s => s.Id == testSessionId)));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var viewResult = Assert.IsType<ViewResult>(result);
        var model = Assert.IsType<StormSessionViewModel>(viewResult.ViewData.Model);
        Assert.Equal("Test One", model.Name);
        Assert.Equal(2, model.DateCreated.Day);
        Assert.Equal(testSessionId, model.Id);
    }

    private List<BrainstormSession> GetTestSessions()
    {
        var sessions = new List<BrainstormSession>();
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 2),
            Id = 1,
            Name = "Test One"
        });
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 1),
            Id = 2,
            Name = "Test Two"
        });
        return sessions;
    }
}
}
}

```

The app exposes functionality as a web API (a list of ideas associated with a brainstorming session and a method for adding new ideas to a session):

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;

namespace TestingControllersSample.Api
{
    [Route("api/ideas")]
    public class IdeasController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public IdeasController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        [HttpGet("forsession/{sessionId}")]
        public async Task<IActionResult> ForSession(int sessionId)
        {
            var session = await _sessionRepository.GetByIdAsync(sessionId);
            if (session == null)
            {
                return NotFound(sessionId);
            }

            var result = session.Ideas.Select(idea => new IdeaDTO()
            {
                Id = idea.Id,
                Name = idea.Name,
                Description = idea.Description,
                DateCreated = idea.DateCreated
            }).ToList();

            return Ok(result);
        }

        [HttpPost("create")]
        public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            var session = await _sessionRepository.GetByIdAsync(model.SessionId);
            if (session == null)
            {
                return NotFound(model.SessionId);
            }

            var idea = new Idea()
            {
                DateCreated = DateTimeOffset.Now,
                Description = model.Description,
                Name = model.Name
            };
            session.AddIdea(idea);

            await _sessionRepository.UpdateAsync(session);

            return Ok(result);
        }
    }
}
```

```

        return OK(session);
    }
}
}

```

The `ForSession` method returns a list of `IdeaDTO` types. Avoid returning your business domain entities directly via API calls, since frequently they include more data than the API client requires, and they unnecessarily couple your app's internal domain model with the API you expose externally. Mapping between domain entities and the types you will return over the wire can be done manually (using a LINQ `Select` as shown here) or using a library like [AutoMapper](#)

The unit tests for the `Create` and `ForSession` API methods:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Api;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class ApiIdeasControllerTests
    {
        [Fact]
        public async Task Create_ReturnsBadRequest_GivenInvalidModel()
        {
            // Arrange & Act
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            var controller = new IdeasController(mockRepo.Object);
            controller.ModelState.AddModelError("error", "some error");

            // Act
            var result = await controller.Create(model: null);

            // Assert
            Assert.IsType<BadRequestObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
        {
            // Arrange
            int testSessionId = 123;
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
                .Returns(Task.FromResult((BrainstormSession)null));
            var controller = new IdeasController(mockRepo.Object);

            // Act
            var result = await controller.Create(new NewIdeaModel());

            // Assert
            Assert.IsType<NotFoundObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsNewlyCreatedIdeaForSession()
        {
            // Arrange
            int testSessionId = 123;

```

```

string testName = "test name";
string testDescription = "test description";
var testSession = GetTestSession();
var mockRepo = new Mock<IBrainstormSessionRepository>();
mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
    .Returns(Task.FromResult(testSession));
var controller = new IdeasController(mockRepo.Object);

var newIdea = new NewIdeaModel()
{
    Description = testDescription,
    Name = testName,
    SessionId = testSessionId
};
mockRepo.Setup(repo => repo.UpdateAsync(testSession))
    .Returns(Task.CompletedTask)
    .Verifiable();

// Act
var result = await controller.Create(newIdea);

// Assert
var okResult = Assert.IsType<OkObjectResult>(result);
var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
mockRepo.Verify();
Assert.Equal(2, returnSession.Ideas.Count());
Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}

private BrainstormSession GetTestSession()
{
    var session = new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    };

    var idea = new Idea() { Name = "One" };
    session.AddIdea(idea);
    return session;
}
}
}

```

As stated previously, to test the behavior of the method when `ModelState` is invalid, add a model error to the controller as part of the test. Don't try to test model validation or model binding in your unit tests - just test your action method's behavior when confronted with a particular `ModelState` value.

The second test depends on the repository returning null, so the mock repository is configured to return null. There's no need to create a test database (in memory or otherwise) and construct a query that will return this result - it can be done in a single statement as shown.

The last test verifies that the repository's `Update` method is called. As we did previously, the mock is called with `Verifiable` and then the mocked repository's `Verify` method is called to confirm the verifiable method was executed. It's not a unit test responsibility to ensure that the `Update` method saved the data; that can be done with an integration test.

Integration testing

[Integration testing](#) is done to ensure separate modules within your app work correctly together. Generally, anything you can test with a unit test, you can also test with an integration test, but the reverse isn't true. However,

integration tests tend to be much slower than unit tests. Thus, it's best to test whatever you can with unit tests, and use integration tests for scenarios that involve multiple collaborators.

Although they may still be useful, mock objects are rarely used in integration tests. In unit testing, mock objects are an effective way to control how collaborators outside of the unit being tested should behave for the purposes of the test. In an integration test, real collaborators are used to confirm the whole subsystem works together correctly.

Application state

One important consideration when performing integration testing is how to set your app's state. Tests need to run independent of one another, and so each test should start with the app in a known state. If your app doesn't use a database or have any persistence, this may not be an issue. However, most real-world apps persist their state to some kind of data store, so any modifications made by one test could impact another test unless the data store is reset. Using the built-in `TestServer`, it's very straightforward to host ASP.NET Core apps within our integration tests, but that doesn't necessarily grant access to the data it will use. If you're using an actual database, one approach is to have the app connect to a test database, which your tests can access and ensure is reset to a known state before each test executes.

In this sample application, I'm using Entity Framework Core's `InMemoryDatabase` support, so I can't just connect to it from my test project. Instead, I expose an `InitializeDatabase` method from the app's `Startup` class, which I call when the app starts up if it's in the `Development` environment. My integration tests automatically benefit from this as long as they set the environment to `Development`. I don't have to worry about resetting the database, since the `InMemoryDatabase` is reset each time the app restarts.

The `Startup` class:

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.Infrastructure;

namespace TestingControllersSample
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(
                optionsBuilder => optionsBuilder.UseInMemoryDatabase("InMemoryDb"));

            services.AddMvc();

            services.AddScoped<IBrainstormSessionRepository,
                EFStormSessionRepository>();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env,
            ILoggerFactory loggerFactory)
        {
            if (env.IsDevelopment())
            {
                var repository = app.ApplicationServices.GetService<IBrainstormSessionRepository>();
                InitializeDatabaseAsync(repository).Wait();
            }
        }
    }
}
```

```

        app.UseStaticFiles();

        app.UseMvcWithDefaultRoute();
    }

    public async Task InitializeDatabaseAsync(IBrainstormSessionRepository repo)
    {
        var sessionList = await repo.ListAsync();
        if (!sessionList.Any())
        {
            await repo.AddAsync(GetTestSession());
        }
    }

    public static BrainstormSession GetTestSession()
    {
        var session = new BrainstormSession()
        {
            Name = "Test Session 1",
            DateCreated = new DateTime(2016, 8, 1)
        };
        var idea = new Idea()
        {
            DateCreated = new DateTime(2016, 8, 1),
            Description = "Totally awesome idea",
            Name = "Awesome idea"
        };
        session.AddIdea(idea);
        return session;
    }
}

```

You'll see the `GetTestSession` method used frequently in the integration tests below.

Accessing views

Each integration test class configures the `TestServer` that will run the ASP.NET Core app. By default, `TestServer` hosts the web app in the folder where it's running - in this case, the test project folder. Thus, when you attempt to test controller actions that return `ViewResult`, you may see this error:

```

The view 'Index' was not found. The following locations were searched:
(list of locations)

```

To correct this issue, you need to configure the server's content root, so it can locate the views for the project being tested. This is done by a call to `UseContentRoot` in the `TestFixture` class, shown below:

```

using System;
using System.IO;
using System.Net.Http;
using System.Reflection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.TestHost;
using Microsoft.Extensions.DependencyInjection;

namespace TestingControllersSample.Tests.IntegrationTests
{
    /// <summary>
    /// A test fixture which hosts the target project (project we wish to test) in an in-memory server.
    /// </summary>
    /// <typeparam name="TStartup">Target project's startup type</typeparam>
    public class TestFixture<TStartup> : IDisposable

```

```

{
    private readonly TestServer _server;

    public TestFixture()
        : this(Path.Combine("src"))
    {
    }

    protected TestFixture(string relativeTargetProjectParentDir)
    {
        var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;
        var contentRoot = GetProjectPath(relativeTargetProjectParentDir, startupAssembly);

        var builder = new WebHostBuilder()
            .UseContentRoot(contentRoot)
            .ConfigureServices(InitializeServices)
            .UseEnvironment("Development")
            .UseStartup(typeof(TStartup));

        _server = new TestServer(builder);

        Client = _server.CreateClient();
        Client.BaseAddress = new Uri("http://localhost");
    }

    public HttpClient Client { get; }

    public void Dispose()
    {
        Client.Dispose();
        _server.Dispose();
    }

    protected virtual void InitializeServices(IServiceCollection services)
    {
        var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;

        // Inject a custom application part manager.
        // Overrides AddMvcCore() because it uses TryAdd().
        var manager = new ApplicationPartManager();
        manager.ApplicationParts.Add(new AssemblyPart(startupAssembly));
        manager.FeatureProviders.Add(new ControllerFeatureProvider());
        manager.FeatureProviders.Add(new ViewComponentFeatureProvider());

        services.AddSingleton(manager);
    }

    /// <summary>
    /// Gets the full path to the target project that we wish to test
    /// </summary>
    /// <param name="projectRelativePath">
    /// The parent directory of the target project.
    /// e.g. src, samples, test, or test/Websites
    /// </param>
    /// <param name="startupAssembly">The target project's assembly.</param>
    /// <returns>The full path to the target project.</returns>
    private static string GetProjectPath(string projectRelativePath, Assembly startupAssembly)
    {
        // Get name of the target project which we want to test
        var projectName = startupAssembly.GetName().Name;

        // Get currently executing test project path
        var applicationBasePath = System.AppContext.BaseDirectory;

        // Find the path to the target project
        var directoryInfo = new DirectoryInfo(applicationBasePath);
        do
        {
            directoryInfo = directoryInfo.Parent:

```

```

        var projectDirectoryInfo = new DirectoryInfo(Path.Combine(directoryInfo.FullName,
projectRelativePath));
        if (projectDirectoryInfo.Exists)
        {
            var projectFileInfo = new FileInfo(Path.Combine(projectDirectoryInfo.FullName, projectName,
"${projectName}.csproj"));
            if (projectFileInfo.Exists)
            {
                return Path.Combine(projectDirectoryInfo.FullName, projectName);
            }
        }
    }
    while (directoryInfo.Parent != null);

    throw new Exception($"Project root could not be located using the application root
{applicationBasePath}.");
}
}
}
}

```

The `TestFixture` class is responsible for configuring and creating the `TestServer`, setting up an `HttpClient` to communicate with the `TestServer`. Each of the integration tests uses the `Client` property to connect to the test server and make a request.

```

using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class HomeControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        private readonly HttpClient _client;

        public HomeControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task ReturnsInitialListOfBrainstormSessions()
        {
            // Arrange - get a session known to exist
            var testSession = Startup.GetTestSession();

            // Act
            var response = await _client.GetAsync("/");

            // Assert
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains(testSession.Name, responseString);
        }

        [Fact]
        public async Task PostAddsNewBrainstormSession()
        {
            // Arrange
            string testSessionName = Guid.NewGuid().ToString();
            var data = new Dictionary<string, string>();
            data.Add("SessionName", testSessionName);
            var content = new FormUrlEncodedContent(data);

            // Act
            var response = await _client.PostAsync("/", content);

            // Assert
            Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
            Assert.Equal("/", response.Headers.Location.ToString());
        }
    }
}

```

In the first test above, the `responseString` holds the actual rendered HTML from the View, which can be inspected to confirm it contains expected results.

The second test constructs a form POST with a unique session name and POSTs it to the app, then verifies that the expected redirect is returned.

API methods

If your app exposes web APIs, it's a good idea to have automated tests confirm they execute as expected. The built-in `TestServer` makes it easy to test web APIs. If your API methods are using model binding, you should always check `ModelState.IsValid`, and integration tests are the right place to confirm that your model validation is working properly.

The following set of tests target the `Create` method in the `IdeasController` class shown above:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Newtonsoft.Json;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class ApiIdeasControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        internal class NewIdeaDto
        {
            public NewIdeaDto(string name, string description, int sessionId)
            {
                Name = name;
                Description = description;
                SessionId = sessionId;
            }

            public string Name { get; set; }
            public string Description { get; set; }
            public int SessionId { get; set; }
        }

        private readonly HttpClient _client;

        public ApiIdeasControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingNameValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("", "Description", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingDescriptionValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForSessionIdValueTooSmall()
        {
            // Arrange
```

```

// Arrange
var newIdea = new NewIdeaDto("Name", "Description", 0);

// Act
var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

// Assert
Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsBadRequestForSessionIdValueTooLarge()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 1000001);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsNotFoundForInvalidSession()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 123);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsCreatedIdeaWithCorrectInputs()
{
    // Arrange
    var testIdeaName = Guid.NewGuid().ToString();
    var newIdea = new NewIdeaDto(testIdeaName, "Description", 1);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    response.EnsureSuccessStatusCode();
    var returnedSession = await response.Content.ReadAsJsonAsync<BrainstormSession>();
    Assert.Equal(2, returnedSession.Ideas.Count);
    Assert.Contains(testIdeaName, returnedSession.Ideas.Select(i => i.Name).ToList());
}

[Fact]
public async Task ForSessionReturnsNotFoundForBadSessionId()
{
    // Arrange & Act
    var response = await _client.GetAsync("/api/ideas/forsession/500");

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task ForSessionReturnsIdeasForValidSessionId()
{
    // Arrange
    var testSession = Startup.GetTestSession();

    // Act

```

```
// Act
var response = await _client.GetAsync("/api/ideas/forsession/1");

// Assert
response.EnsureSuccessStatusCode();
var idealist = JsonConvert.DeserializeObject<List<IdeaDTO>>(
    await response.Content.ReadAsStringAsync());
var firstIdea = idealist.First();
Assert.Equal(testSession.Ideas.First().Name, firstIdea.Name);
    }
}
}
```

Unlike integration tests of actions that returns HTML views, web API methods that return results can usually be deserialized as strongly typed objects, as the last test above shows. In this case, the test deserializes the result to a `BrainstormSession` instance, and confirms that the idea was correctly added to its collection of ideas.

You'll find additional examples of integration tests in this article's [sample project](#).

Advanced topics for ASP.NET Core MVC

7/5/2017 • 1 min to read • [Edit Online](#)

- [Working with the Application Model](#)
- [Filters](#)
- [Areas](#)
- [Application parts](#)
- [Custom Model Binding](#)
- [Custom formatters](#)

Working with the Application Model

9/22/2017 • 10 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET Core MVC defines an *application model* representing the components of an MVC app. You can read and manipulate this model to modify how MVC elements behave. By default, MVC follows certain conventions to determine which classes are considered to be controllers, which methods on those classes are actions, and how parameters and routing behave. You can customize this behavior to suit your app's needs by creating your own conventions and applying them globally or as attributes.

Models and Providers

The ASP.NET Core MVC application model include both abstract interfaces and concrete implementation classes that describe an MVC application. This model is the result of MVC discovering the app's controllers, actions, action parameters, routes, and filters according to default conventions. By working with the application model, you can modify your app to follow different conventions from the default MVC behavior. The parameters, names, routes, and filters are all used as configuration data for actions and controllers.

The ASP.NET Core MVC Application Model has the following structure:

- ApplicationModel
 - Controllers (ControllerModel)
 - Actions (ActionModel)
 - Parameters (ParameterModel)

Each level of the model has access to a common `Properties` collection, and lower levels can access and overwrite property values set by higher levels in the hierarchy. The properties are persisted to the `ActionDescriptor.Properties` when the actions are created. Then when a request is being handled, any properties a convention added or modified can be accessed through `ActionContext.ActionDescriptor.Properties`. Using properties is a great way to configure your filters, model binders, etc. on a per-action basis.

NOTE

The `ActionDescriptor.Properties` collection is not thread safe (for writes) once app startup has finished. Conventions are the best way to safely add data to this collection.

IApplicationModelProvider

ASP.NET Core MVC loads the application model using a provider pattern, defined by the [IApplicationModelProvider](#) interface. This section covers some of the internal implementation details of how this provider functions. This is an advanced topic - most apps that leverage the application model should do so by working with conventions.

Implementations of the `IApplicationModelProvider` interface "wrap" one another, with each implementation calling `OnProvidersExecuting` in ascending order based on its `Order` property. The `OnProvidersExecuted` method is then called in reverse order. The framework defines several providers:

First (`Order=-1000`):

- [DefaultApplicationModelProvider](#)

Then (`Order=-990`):

- `AuthorizationApplicationModelProvider`
- `CorsApplicationModelProvider`

NOTE

The order in which two providers with the same value for `Order` are called is undefined, and therefore should not be relied upon.

NOTE

`IApplicationModelProvider` is an advanced concept for framework authors to extend. In general, apps should use conventions and frameworks should use providers. The key distinction is that providers always run before conventions.

The `DefaultApplicationModelProvider` establishes many of the default behaviors used by ASP.NET Core MVC. Its responsibilities include:

- Adding global filters to the context
- Adding controllers to the context
- Adding public controller methods as actions
- Adding action method parameters to the context
- Applying route and other attributes

Some built-in behaviors are implemented by the `DefaultApplicationModelProvider`. This provider is responsible for constructing the `ControllerModel`, which in turn references `ActionModel`, `PropertyModel`, and `ParameterModel` instances. The `DefaultApplicationModelProvider` class is an internal framework implementation detail that can and will change in the future.

The `AuthorizationApplicationModelProvider` is responsible for applying the behavior associated with the `AuthorizeFilter` and `AllowAnonymousFilter` attributes. [Learn more about these attributes.](#)

The `CorsApplicationModelProvider` implements behavior associated with the `AllowAnonymousAttribute` and `IDisableCorsAttribute`, and the `DisableCorsAuthorizationFilter`. [Learn more about CORS.](#)

Conventions

The application model defines convention abstractions that provide a simpler way to customize the behavior of the models than overriding the entire model or provider. These abstractions are the recommended way to modify your app's behavior. Conventions provide a way for you to write code that will dynamically apply customizations. While [filters](#) provide a means of modifying the framework's behavior, customizations let you control how the whole app is wired together.

The following conventions are available:

- `IApplicationModelConvention`
- `IControllerModelConvention`
- `IActionModelConvention`
- `IParameterModelConvention`

Conventions are applied by adding them to MVC options or by implementing `Attributes` and applying them to controllers, actions, or action parameters (similar to [Filters](#)). Unlike filters, conventions are only executed when the app is starting, not as part of each request.

Sample: Modifying the ApplicationModel

The following convention is used to add a property to the application model.

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ApplicationDescription : IApplicationModelConvention
    {
        private readonly string _description;

        public ApplicationDescription(string description)
        {
            _description = description;
        }

        public void Apply(ApplicationModel application)
        {
            application.Properties["description"] = _description;
        }
    }
}
```

Application model conventions are applied as options when MVC is added in `ConfigureServices` in `Startup`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}
```

Properties are accessible from the `ActionDescriptor` properties collection within controller actions:

```
public class AppModelController : Controller
{
    public string Description()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

Sample: Modifying the ControllerModel Description

As in the previous example, the controller model can also be modified to include custom properties. These will override existing properties with the same name specified in the application model. The following convention attribute adds a description at the controller level:

```

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ControllerDescriptionAttribute : Attribute, IControllerModelConvention
    {
        private readonly string _description;

        public ControllerDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ControllerModel controllerModel)
        {
            controllerModel.Properties["description"] = _description;
        }
    }
}

```

This convention is applied as an attribute on a controller.

```

[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}

```

The "description" property is accessed in the same manner as in previous examples.

Sample: Modifying the ActionModel Description

A separate attribute convention can be applied to individual actions, overriding behavior already applied at the application or controller level.

```

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ActionDescriptionAttribute : Attribute, IActionModelConvention
    {
        private readonly string _description;

        public ActionDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ActionModel actionModel)
        {
            actionModel.Properties["description"] = _description;
        }
    }
}

```

Applying this to an action within the previous example's controller demonstrates how it overrides the controller-level convention:

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }

    [ActionDescription("Action Description")]
    public string UseActionDescriptionAttribute()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

Sample: Modifying the ParameterModel

The following convention can be applied to action parameters to modify their `BindingInfo`. The following convention requires that the parameter be a route parameter; other potential binding sources (such as query string values) are ignored.

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace AppModelSample.Conventions
{
    public class MustBeInRouteParameterModelConvention : Attribute, IParameterModelConvention
    {
        public void Apply(ParameterModel model)
        {
            if (model.BindingInfo == null)
            {
                model.BindingInfo = new BindingInfo();
            }
            model.BindingInfo.BindingSource = BindingSource.Path;
        }
    }
}
```

The attribute may be applied to any action parameter:

```
public class ParameterModelController : Controller
{
    // Will bind: /ParameterModel/GetById/123
    // WON'T bind: /ParameterModel/GetById?id=123
    public string GetById([MustBeInRouteParameterModelConvention]int id)
    {
        return $"Bound to id: {id}";
    }
}
```

Sample: Modifying the ActionModel Name

The following convention modifies the `ActionModel` to update the *name* of the action to which it is applied. The new name is provided as a parameter to the attribute. This new name is used by routing, so it will affect the route used to reach this action method.

```

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class CustomActionNameAttribute : Attribute, IActionModelConvention
    {
        private readonly string _actionName;

        public CustomActionNameAttribute(string actionName)
        {
            _actionName = actionName;
        }

        public void Apply(ActionModel actionModel)
        {
            // this name will be used by routing
            actionModel.ActionName = _actionName;
        }
    }
}

```

This attribute is applied to an action method in the `HomeController`:

```

// Route: /Home/MyCoolAction
[CustomActionName("MyCoolAction")]
public string SomeName()
{
    return ControllerContext.ActionDescriptor.ActionName;
}

```

Even though the method name is `SomeName`, the attribute overrides the MVC convention of using the method name and replaces the action name with `MyCoolAction`. Thus, the route used to reach this action is `/Home/MyCoolAction`.

NOTE

This example is essentially the same as using the built-in [ActionName](#) attribute.

Sample: Custom Routing Convention

You can use an `IApplicationModelConvention` to customize how routing works. For example, the following convention will incorporate Controllers' namespaces into their routes, replacing `.` in the namespace with `/` in the route:

```

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace AppModelSample.Conventions
{
    public class NamespaceRoutingConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            foreach (var controller in application.Controllers)
            {
                var hasAttributeRouteModels = controller.Selectors
                    .Any(selector => selector.AttributeRouteModel != null);

                if (!hasAttributeRouteModels
                    && controller.ControllerName.Contains("Namespace")) // affect one controller in this
sample
                {
                    // Replace the . in the namespace with a / to create the attribute route
                    // Ex: MySite.Admin namespace will correspond to MySite/Admin attribute route
                    // Then attach [controller], [action] and optional {id?} token.
                    // [Controller] and [action] is replaced with the controller and action
                    // name to generate the final template
                    controller.Selectors[0].AttributeRouteModel = new AttributeRouteModel()
                    {
                        Template = controller.ControllerType.Namespace.Replace('.', '/') +
"/[controller]/[action]/{id?}"
                    };
                }
            }

            // You can continue to put attribute route templates for the controller actions depending on the
            way you want them to behave
        }
    }
}

```

The convention is added as an option in Startup.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}

```

TIP

You can add conventions to your [middleware](#) by accessing `MvcOptions` using `services.Configure<MvcOptions>(c => c.Conventions.Add(YOURCONVENTION));`

This sample applies this convention to routes that are not using attribute routing where the controller has "Namespace" in its name. The following controller demonstrates this convention:

```
using Microsoft.AspNetCore.Mvc;

namespace AppModelSample.Controllers
{
    public class NamespaceRoutingController : Controller
    {
        // using NamespaceRoutingConvention
        // route: /AppModelSample/Controllers/NamespaceRouting/Index
        public string Index()
        {
            return "This demonstrates namespace routing.";
        }
    }
}
```

Application Model Usage in WebApiCompatShim

ASP.NET Core MVC uses a different set of conventions from ASP.NET Web API 2. Using custom conventions, you can modify an ASP.NET Core MVC app's behavior to be consistent with that of a Web API app. Microsoft ships the [WebApiCompatShim](#) specifically for this purpose.

NOTE

Learn more about [migrating from ASP.NET Web API](#).

To use the Web API Compatibility Shim, you need to add the package to your project and then add the conventions to MVC by calling `AddWebApiConventions` in `Startup`:

```
services.AddMvc().AddWebApiConventions();
```

The conventions provided by the shim are only applied to parts of the app that have had certain attributes applied to them. The following four attributes are used to control which controllers should have their conventions modified by the shim's conventions:

- [UseWebApiActionConventionsAttribute](#)
- [UseWebApiOverloadingAttribute](#)
- [UseWebApiParameterConventionsAttribute](#)
- [UseWebApiRoutesAttribute](#)

Action Conventions

The `UseWebApiActionConventionsAttribute` is used to map the HTTP method to actions based on their name (for instance, `Get` would map to `HttpGet`). It only applies to actions that do not use attribute routing.

Overloading

The `UseWebApiOverloadingAttribute` is used to apply the `WebApiOverloadingApplicationModelConvention` convention. This convention adds an `OverloadActionConstraint` to the action selection process, which limits candidate actions to those for which the request satisfies all non-optional parameters.

Parameter Conventions

The `UseWebApiParameterConventionsAttribute` is used to apply the `WebApiParameterConventionsApplicationModelConvention` action convention. This convention specifies that simple types used as action parameters are bound from the URI by default, while complex types are bound from the request body.

Routes

The `UseWebApiRoutesAttribute` controls whether the `WebApiApplicationModelConvention` controller convention is applied. When enabled, this convention is used to add support for [areas](#) to the route.

In addition to a set of conventions, the compatibility package includes a `System.Web.Http.ApiController` base class that replaces the one provided by Web API. This allows your controllers written for Web API and inheriting from its `ApiController` to work as they were designed, while running on ASP.NET Core MVC. This base controller class is decorated with all of the `UseWebApi*` attributes listed above. The `ApiController` exposes properties, methods, and result types that are compatible with those found in Web API.

Using ApiExplorer to Document Your App

The application model exposes an `ApiExplorer` property at each level that can be used to traverse the app's structure. This can be used to [generate help pages for your Web APIs using tools like Swagger](#). The `ApiExplorer` property exposes an `IsVisible` property that can be set to specify which parts of your app's model should be exposed. You can configure this setting using a convention:

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class EnableApiExplorerApplicationConvention : IApplicationModelConvention
    {
        {
            public void Apply(ApplicationModel application)
            {
                application.ApiExplorer.IsVisible = true;
            }
        }
    }
}
```

Using this approach (and additional conventions if required), you can enable or disable API visibility at any level within your app.

Filters

11/7/2017 • 19 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Steve Smith](#)

Filters in ASP.NET Core MVC allow you to run code before or after certain stages in the request processing pipeline.

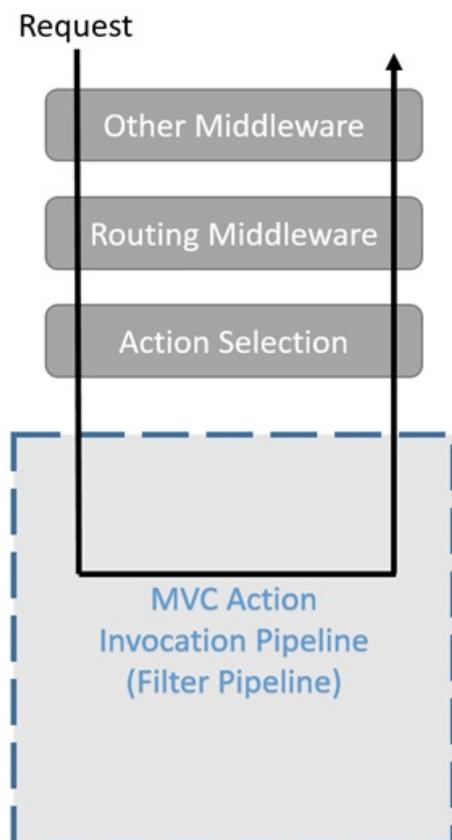
Built-in filters handle tasks such as authorization (preventing access to resources a user isn't authorized for), ensuring that all requests use HTTPS, and response caching (short-circuiting the request pipeline to return a cached response).

You can create custom filters to handle cross-cutting concerns for your application. Anytime you want to avoid duplicating code across actions, filters are the solution. For example, you can consolidate error handling code in an exception filter.

[View or download sample from GitHub.](#)

How do filters work?

Filters run within the *MVC action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after MVC selects the action to execute.



Filter types

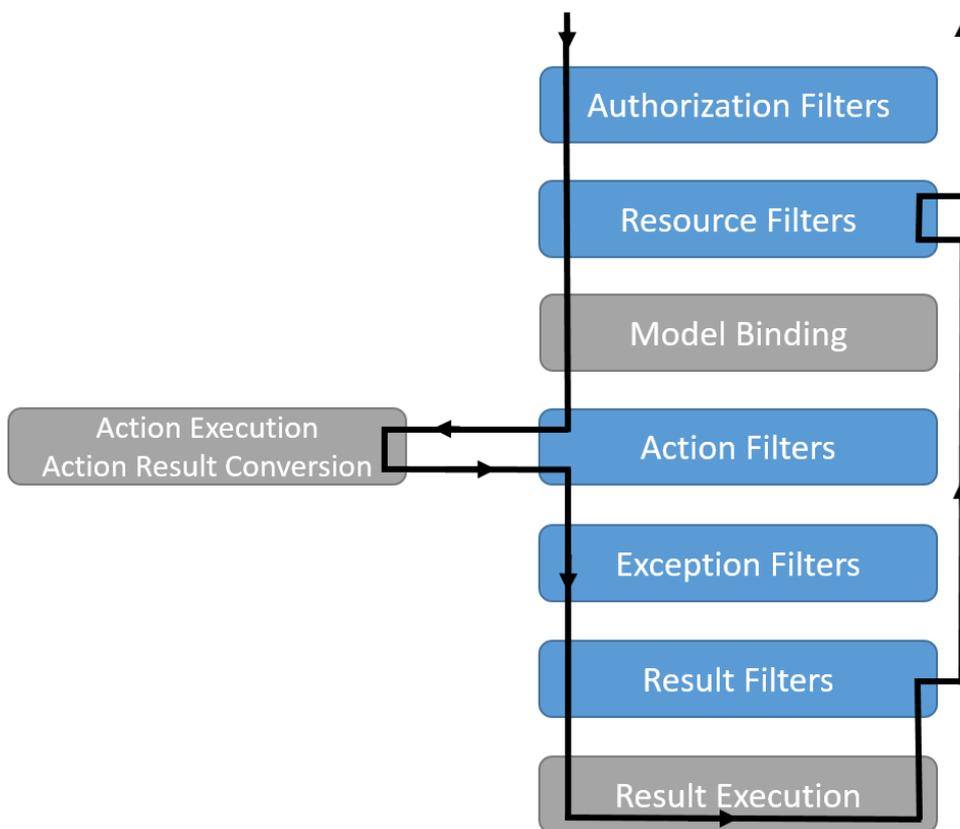
Each filter type is executed at a different stage in the filter pipeline.

- [Authorization filters](#) run first and are used to determine whether the current user is authorized for the

current request. They can short-circuit the pipeline if a request is unauthorized.

- **Resource filters** are the first to handle a request after authorization. They can run code before the rest of the filter pipeline, and after the rest of the pipeline has completed. They're useful to implement caching or otherwise short-circuit the filter pipeline for performance reasons. Since they run before model binding, they're useful for anything that needs to influence model binding.
- **Action filters** can run code immediately before and after an individual action method is called. They can be used to manipulate the arguments passed into an action and the result returned from the action.
- **Exception filters** are used to apply global policies to unhandled exceptions that occur before anything has been written to the response body.
- **Result filters** can run code immediately before and after the execution of individual action results. They run only when the action method has executed successfully and are useful for logic that must surround view or formatter execution.

The following diagram shows how these filter types interact in the filter pipeline.



Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions. Choose either the sync or async variant depending on the kind of task you need to perform.

Synchronous filters that can run code both before and after their pipeline stage define `OnStageExecuting` and `OnStageExecuted` methods. For example, `OnActionExecuting` is called before the action method is called, and `OnActionExecuted` is called after the action method returns.

```

using FiltersSample.Helper;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    #region snippet_ActionFilter
    public class SampleActionFilter : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            // do something before the action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // do something after the action executes
        }
    }
    #endregion
}

```

Asynchronous filters define a single `OnStageExecutionAsync` method. This method takes a `FilterTypeExecutionDelegate` delegate which executes the filter's pipeline stage. For example, `ActionExecutionDelegate` calls the action method, and you can execute code before and after you call it.

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class SampleAsyncActionFilter : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(
            ActionExecutingContext context,
            ActionExecutionDelegate next)
        {
            // do something before the action executes
            var resultContext = await next();
            // do something after the action executes; resultContext.Result will be set
        }
    }
}

```

You can implement interfaces for multiple filter stages in a single class. For example, the [ActionFilterAttribute](#) abstract class implements both `IActionFilter` and `IResultFilter`, as well as their async equivalents.

NOTE

Implement **either** the synchronous or the async version of a filter interface, not both. The framework checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If you were to implement both interfaces on one class, only the async method would be called. When using abstract classes like [ActionFilterAttribute](#) you would override only the synchronous methods or the async method for each filter type.

IFilterFactory

`IFilterFactory` implements `IFilter`. Therefore, an `IFilterFactory` instance can be used as an `IFilter` instance anywhere in the filter pipeline. When the framework prepares to invoke the filter, it attempts to cast it to an `IFilterFactory`. If that cast succeeds, the `CreateInstance` method is called to create the `IFilter` instance that will be invoked. This provides a very flexible design, since the precise filter pipeline does not need

to be set explicitly when the application starts.

You can implement `IFilterFactory` on your own attribute implementations as another approach to creating filters:

```
public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new InternalAddHeaderFilter();
    }

    private class InternalAddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "Internal", new string[] { "Header Added" });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Built-in filter attributes

The framework includes built-in attribute-based filters that you can subclass and customize. For example, the following Result filter adds a header to the response.

```
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class AddHeaderAttribute : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute(string name, string value)
        {
            _name = name;
            _value = value;
        }

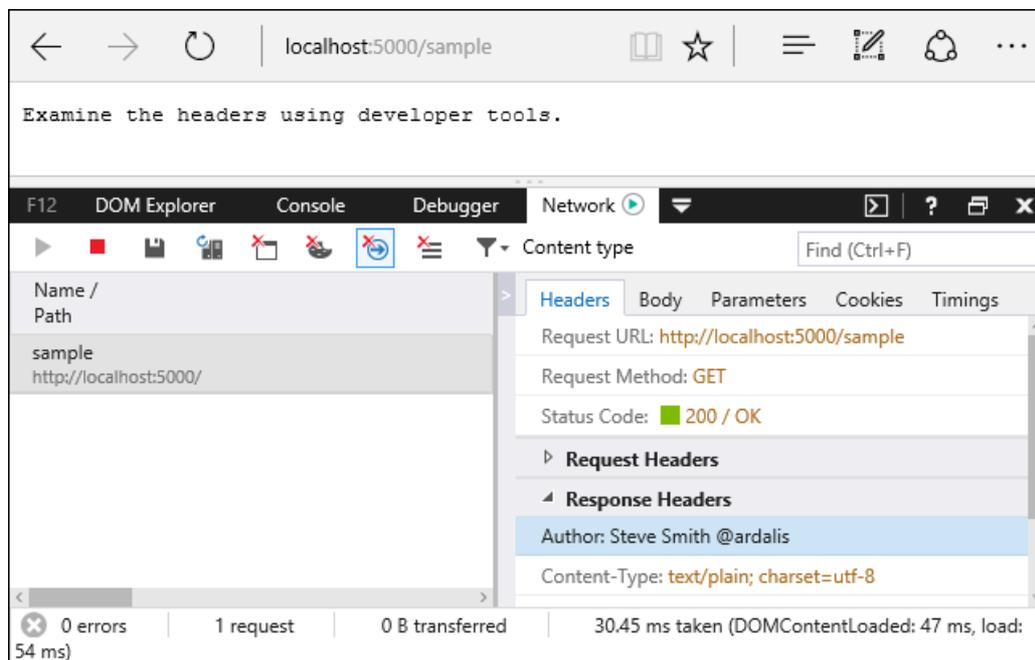
        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                _name, new string[] { _value });
            base.OnResultExecuting(context);
        }
    }
}
```

Attributes allow filters to accept arguments, as shown in the example above. You would add this attribute to a controller or action method and specify the name and value of the HTTP header:

```
[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header should be set.");
    }
}
```

The result of the `Index` action is shown below - the response headers are displayed on the bottom right.



Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

- `ActionFilterAttribute`
- `ExceptionHandlerAttribute`
- `ResultFilterAttribute`
- `FormatFilterAttribute`
- `ServiceFilterAttribute`
- `TypeFilterAttribute`

`TypeFilterAttribute` and `ServiceFilterAttribute` are explained [later in this article](#).

Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*. You can add a filter to a particular action method or to a controller class by using an attribute. Or you can register a filter globally (for all controllers and actions) by adding it to the `MvcOptions.Filters` collection in the `ConfigureServices` method in the `Startup` class:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters")); // an instance
        options.Filters.Add(typeof(SampleActionFilter)); // by type
        options.Filters.Add(new SampleGlobalActionFilter()); // an instance
    });

    services.AddScoped<AddHeaderFilterWithDi>();
}

```

Default order of execution

When there are multiple filters for a particular stage of the pipeline, scope determines the default order of filter execution. Global filters surround class filters, which in turn surround method filters. This is sometimes referred to as "Russian doll" nesting, as each increase in scope is wrapped around the previous scope, like a [nesting doll](#). You generally get the desired overriding behavior without having to explicitly determine ordering.

As a result of this nesting, the *after* code of filters runs in the reverse order of the *before* code. The sequence looks like this:

- The *before* code of filters applied globally
 - The *before* code of filters applied to controllers
 - The *before* code of filters applied to action methods
 - The *after* code of filters applied to action methods
 - The *after* code of filters applied to controllers
- The *after* code of filters applied globally

Here's an example that illustrates the order in which filter methods are called for synchronous Action filters.

SEQUENCE	FILTER SCOPE	FILTER METHOD
1	Global	OnActionExecuting
2	Controller	OnActionExecuting
3	Method	OnActionExecuting
4	Method	OnActionExecuted
5	Controller	OnActionExecuted
6	Global	OnActionExecuted

This sequence shows that the method filter is nested within the controller filter, and the controller filter is nested within the global filter. To put it another way, if you're inside an async filter's *OnStageExecutionAsync* method, all of the filters with a tighter scope run while your code is on the stack.

NOTE

Every controller that inherits from the `Controller` base class includes `OnActionExecuting` and `OnActionExecuted` methods. These methods wrap the filters that run for a given action: `OnActionExecuting` is called before any of the filters, and `OnActionExecuted` is called after all of the filters.

Overriding the default order

You can override the default sequence of execution by implementing `IOrderedFilter`. This interface exposes an `Order` property that takes precedence over scope to determine the order of execution. A filter with a lower `Order` value will have its *before* code executed before that of a filter with a higher value of `Order`. A filter with a lower `Order` value will have its *after* code executed after that of a filter with a higher `Order` value. You can set the `Order` property by using a constructor parameter:

```
[MyFilter(Name = "Controller Level Attribute", Order=1)]
```

If you have the same 3 Action filters shown in the preceding example but set the `Order` property of the controller and global filters to 1 and 2 respectively, the order of execution would be reversed.

SEQUENCE	FILTER SCOPE	ORDER PROPERTY	FILTER METHOD
1	Method	0	OnActionExecuting
2	Controller	1	OnActionExecuting
3	Global	2	OnActionExecuting
4	Global	2	OnActionExecuted
5	Controller	1	OnActionExecuted
6	Method	0	OnActionExecuted

The `Order` property trumps scope when determining the order in which filters will run. Filters are sorted first by order, then scope is used to break ties. All of the built-in filters implement `IOrderedFilter` and set the default `Order` value to 0, so scope determines order unless you set `Order` to a non-zero value.

Cancellation and short circuiting

You can short-circuit the filter pipeline at any point by setting the `Result` property on the `context` parameter provided to the filter method. For instance, the following Resource filter prevents the rest of the pipeline from executing.

```

using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class ShortCircuitingResourceFilterAttribute : Attribute,
        IResourceFilter
    {
        {
            public void OnResourceExecuting(ResourceExecutingContext context)
            {
                {
                    context.Result = new ContentResult()
                    {
                        Content = "Resource unavailable - header should not be set"
                    };
                }
            }

            public void OnResourceExecuted(ResourceExecutedContext context)
            {
                {
                }
            }
        }
    }
}

```

In the following code, both the `ShortCircuitingResourceFilter` and the `AddHeader` filter target the `SomeResource` action method. However, because the `ShortCircuitingResourceFilter` runs first (because it is a Resource Filter and `AddHeader` is an Action Filter) and short-circuits the rest of the pipeline, the `AddHeader` filter never runs for the `SomeResource` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilter` ran first (because of its filter type, or explicit use of `Order` property, for instance).

```

[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header should be set.");
    }
}

```

Dependency injection

Filters can be added by type or by instance. If you add an instance, that instance will be used for every request. If you add a type, it will be type-activated, meaning an instance will be created for each request and any constructor dependencies will be populated by [dependency injection](#) (DI). Adding a filter by type is equivalent to `filters.Add(new TypeFilterAttribute(typeof(MyFilter)))`.

Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by [dependency injection](#) (DI). This is because attributes must have their constructor parameters supplied where they are applied. This is a limitation of how attributes work.

If your filters have dependencies that you need to access from DI, there are several supported approaches. You can apply your filter to a class or action method using one of the following:

- `ServiceFilterAttribute`

- `TypeFilterAttribute`
- `IFilterFactory` implemented on your attribute

NOTE

One dependency you might want to get from DI is a logger. However, avoid creating and using filters purely for logging purposes, since the [built-in framework logging features](#) may already provide what you need. If you're going to add logging to your filters, it should focus on business domain concerns or behavior specific to your filter, rather than MVC actions or other framework events.

ServiceFilterAttribute

A `ServiceFilter` retrieves an instance of the filter from DI. You add the filter to the container in `ConfigureServices`, and reference it in a `ServiceFilter` attribute

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters")); // an instance
        options.Filters.Add(typeof(SampleActionFilter)); // by type
        options.Filters.Add(new SampleGlobalActionFilter()); // an instance
    });

    services.AddScoped<AddHeaderFilterWithDi>();
}
```

```
[ServiceFilter(typeof(AddHeaderFilterWithDi))]
public IActionResult Index()
{
    return View();
}
```

Using `ServiceFilter` without registering the filter type results in an exception:

```
System.InvalidOperationException: No service for type
'FiltersSample.Filters.AddHeaderFilterWithDI' has been registered.
```

`ServiceFilterAttribute` implements `IFilterFactory`, which exposes a single method for creating an `IFilter` instance. In the case of `ServiceFilterAttribute`, the `IFilterFactory` interface's `CreateInstance` method is implemented to load the specified type from the services container (DI).

TypeFilterAttribute

`TypeFilterAttribute` is very similar to `ServiceFilterAttribute` (and also implements `IFilterFactory`), but its type is not resolved directly from the DI container. Instead, it instantiates the type by using `Microsoft.Extensions.DependencyInjection.ObjectFactory`.

Because of this difference, types that are referenced using the `TypeFilterAttribute` do not need to be registered with the container first (but they will still have their dependencies fulfilled by the container). Also, `TypeFilterAttribute` can optionally accept constructor arguments for the type in question. The following example demonstrates how to pass arguments to a type using `TypeFilterAttribute`:

```
[TypeFilter(typeof(AddHeaderAttribute),
    Arguments = new object[] { "Author", "Steve Smith (@ardalis)" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

If you have a filter that doesn't require any arguments, but which has constructor dependencies that need to be filled by DI, you can use your own named attribute on classes and methods instead of

`[TypeFilter(typeof(FilterType))]`). The following filter shows how this can be implemented:

```
public class SampleActionFilterAttribute : TypeFilterAttribute
{
    public SampleActionFilterAttribute():base(typeof(SampleActionFilterImpl))
    {
    }

    private class SampleActionFilterImpl : IActionFilter
    {
        private readonly ILogger _logger;
        public SampleActionFilterImpl(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation("Business action starting...");
            // perform some business logic work
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // perform some business logic work
            _logger.LogInformation("Business action completed.");
        }
    }
}
```

This filter can be applied to classes or methods using the `[SampleActionFilter]` syntax, instead of having to use `[TypeFilter]` or `[ServiceFilter]`.

Authorization filters

Authorization filters control access to action methods and are the first filters to be executed within the filter pipeline. They have only a before method, unlike most filters that support before and after methods. You should only write a custom authorization filter if you are writing your own authorization framework. Prefer configuring your authorization policies or writing a custom authorization policy over writing a custom filter. The built-in filter implementation is just responsible for calling the authorization system.

Note that you should not throw exceptions within authorization filters, since nothing will handle the exception (exception filters won't handle them). Instead, issue a challenge or find another way.

Learn more about [Authorization](#).

Resource filters

Resource filters implement either the `IResourceFilter` or `IAsyncResourceFilter` interface, and their execution

wraps most of the filter pipeline. (Only [Authorization filters](#) run before them.) Resource filters are especially useful if you need to short-circuit most of the work a request is doing. For example, a caching filter can avoid the rest of the pipeline if the response is already in the cache.

The [short circuiting resource filter](#) shown earlier is one example of a resource filter. Another example is [DisableFormValueModelBindingAttribute](#), which prevents model binding from accessing the form data. It's useful for cases where you know that you're going to receive large file uploads and want to prevent the form from being read into memory.

Action filters

Action filters implement either the `IActionFilter` or `IAsyncActionFilter` interface, and their execution surrounds the execution of action methods.

Here's a sample action filter:

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // do something before the action executes
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // do something after the action executes
    }
}
```

The [ActionExecutingContext](#) provides the following properties:

- `ActionArguments` - lets you manipulate the inputs to the action.
- `Controller` - lets you manipulate the controller instance.
- `Result` - setting this short-circuits execution of the action method and subsequent action filters. Throwing an exception also prevents execution of the action method and subsequent filters, but is treated as a failure instead of a successful result.

The [ActionExecutedContext](#) provides `Controller` and `Result` plus the following properties:

- `Canceled` - will be true if the action execution was short-circuited by another filter.
- `Exception` - will be non-null if the action or a subsequent action filter threw an exception. Setting this property to null effectively 'handles' an exception, and `Result` will be executed as if it were returned from the action method normally.

For an `IAsyncActionFilter`, a call to the `ActionExecutionDelegate` executes any subsequent action filters and the action method, returning an `ActionExecutedContext`. To short-circuit, assign `ActionExecutingContext.Result` to some result instance and do not call the `ActionExecutionDelegate`.

The framework provides an abstract `ActionFilterAttribute` that you can subclass.

You can use an action filter to automatically validate model state and return any errors if the state is invalid:

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class ValidateModelAttribute : ActionFilterAttribute
    {
        public override void OnActionExecuting(ActionExecutingContext context)
        {
            if (!context.ModelState.IsValid)
            {
                context.Result = new BadRequestObjectResult(context.ModelState);
            }
        }
    }
}

```

The `OnActionExecuted` method runs after the action method and can see and manipulate the results of the action through the `ActionExecutedContext.Result` property. `ActionExecutedContext.Canceled` will be set to true if the action execution was short-circuited by another filter. `ActionExecutedContext.Exception` will be set to a non-null value if the action or a subsequent action filter threw an exception. Setting `ActionExecutedContext.Exception` to null effectively 'handles' an exception, and `ActionExecutedContext.Result` will then be executed as if it were returned from the action method normally.

Exception filters

Exception filters implement either the `IExceptionHandler` or `IAsyncExceptionHandler` interface. They can be used to implement common error handling policies for an app.

The following sample exception filter uses a custom developer error view to display details about exceptions that occur when the application is in development:

```

public class CustomExceptionHandlerAttribute : ExceptionFilterAttribute
{
    private readonly IHostingEnvironment _hostingEnvironment;
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionHandlerAttribute(
        IHostingEnvironment hostingEnvironment,
        IModelMetadataProvider modelMetadataProvider)
    {
        _hostingEnvironment = hostingEnvironment;
        _modelMetadataProvider = modelMetadataProvider;
    }

    public override void OnException(ExceptionContext context)
    {
        if (!_hostingEnvironment.IsDevelopment())
        {
            // do nothing
            return;
        }
        var result = new ViewResult {ViewName = "CustomError"};
        result.ViewData = new ViewDataDictionary(_modelMetadataProvider, context.ModelState);
        result.ViewData.Add("Exception", context.Exception);
        // TODO: Pass additional detailed data via ViewData
        context.Result = result;
    }
}

```

Exception filters do not have two events (for before and after) - they only implement `OnException` (or

`OnExceptionAsync`).

Exception filters handle unhandled exceptions that occur in controller creation, [model binding](#), action filters, or action methods. They won't catch exceptions that occur in Resource filters, Result filters, or MVC Result execution.

To handle an exception, set the `ExceptionContext.ExceptionHandled` property to true or write a response. This stops propagation of the exception. Note that an Exception filter can't turn an exception into a "success". Only an Action filter can do that.

NOTE

In ASP.NET 1.1, the response is not sent if you set `ExceptionHandled` to true **and** write a response. In that scenario, ASP.NET Core 1.0 does send the response, and ASP.NET Core 1.1.2 will return to the 1.0 behavior. For more information, see [issue #5594](#) in the GitHub repository.

Exception filters are good for trapping exceptions that occur within MVC actions, but they're not as flexible as error handling middleware. Prefer middleware for the general case, and use filters only where you need to do error handling *differently* based on which MVC action was chosen. For example, your app might have action methods for both API endpoints and for views/HTML. The API endpoints could return error information as JSON, while the view-based actions could return an error page as HTML.

The framework provides an abstract `ExceptionHandlerAttribute` that you can subclass.

Result filters

Result filters implement either the `IResultFilter` or `IAsyncResultFilter` interface, and their execution surrounds the execution of action results.

Here's an example of a Result filter that adds an HTTP header.

```
public class AddHeaderFilterWithDi : IResultFilter
{
    private ILogger _logger;
    public AddHeaderFilterWithDi(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderFilterWithDi>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation($"Header added: {headerName}");
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has already begun.
    }
}
```

The kind of result being executed depends on the action in question. An MVC action returning a view would include all razor processing as part of the `ViewResult` being executed. An API method might perform some serialization as part of the execution of the result. Learn more about [action results](#)

Result filters are only executed for successful results - when the action or action filters produce an action result. Result filters are not executed when exception filters handle an exception.

The `OnResultExecuting` method can short-circuit execution of the action result and subsequent result filters by setting `ResultExecutingContext.Cancel` to true. You should generally write to the response object when short-circuiting to avoid generating an empty response. Throwing an exception will also prevent execution of the action result and subsequent filters, but will be treated as a failure instead of a successful result.

When the `OnResultExecuted` method runs, the response has likely been sent to the client and cannot be changed further (unless an exception was thrown). `ResultExecutedContext.Canceled` will be set to true if the action result execution was short-circuited by another filter.

`ResultExecutedContext.Exception` will be set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `Exception` to null effectively 'handles' an exception and prevents the exception from being rethrown by MVC later in the pipeline. When you're handling an exception in a result filter, you might not be able to write any data to the response. If the action result throws partway through its execution, and the headers have already been flushed to the client, there's no reliable mechanism to send a failure code.

For an `IAsyncResultFilter` a call to `await next()` on the `ResultExecutionDelegate` executes any subsequent result filters and the action result. To short-circuit, set `ResultExecutingContext.Cancel` to true and do not call the `ResultExecutionDelegate`.

The framework provides an abstract `ResultFilterAttribute` that you can subclass. The [AddHeaderAttribute](#) class shown earlier is an example of a result filter attribute.

Using middleware in the filter pipeline

Resource filters work like [middleware](#) in that they surround the execution of everything that comes later in the pipeline. But filters differ from middleware in that they are part of MVC, which means that they have access to MVC context and constructs.

In ASP.NET Core 1.1, you can use middleware in the filter pipeline. You might want to do that if you have a middleware component that needs access to MVC route data, or one that should run only for certain controllers or actions.

To use middleware as a filter, create a type with a `Configure` method that specifies the middleware that you want to inject into the filter pipeline. Here's an example that uses the localization middleware to establish the current culture for a request:

```

public class LocalizationPipeline
{
    public void Configure(IApplicationBuilder applicationBuilder)
    {
        var supportedCultures = new[]
        {
            new CultureInfo("en-US"),
            new CultureInfo("fr")
        };

        var options = new RequestLocalizationOptions
        {

            DefaultRequestCulture = new RequestCulture(culture: "en-US", uiCulture: "en-US"),
            SupportedCultures = supportedCultures,
            SupportedUICultures = supportedCultures
        };
        options.RequestCultureProviders = new[]
        { new RouteDataRequestCultureProvider() { Options = options } };

        applicationBuilder.UseRequestLocalization(options);
    }
}

```

You can then use the `MiddlewareFilterAttribute` to run the middleware for a selected controller or action or globally:

```

[Route("{culture}/[controller]/[action]")]
[MiddlewareFilter(typeof(LocalizationPipeline))]
public IActionResult CultureFromRouteData()
{
    return Content($"CurrentCulture:{CultureInfo.CurrentCulture.Name},"
        + "CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");
}

```

Middleware filters run at the same stage of the filter pipeline as Resource filters, before model binding and after the rest of the pipeline.

Next actions

To experiment with filters, [download, test and modify the sample](#).

Areas

10/26/2017 • 4 min to read • [Edit Online](#)

By [Dhananjay Kumar](#) and [Rick Anderson](#)

Areas are an ASP.NET MVC feature used to organize related functionality into a group as a separate namespace (for routing) and folder structure (for views). Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area`, to `controller` and `action`.

Areas provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is effectively an MVC structure inside an application. In an MVC project, logical components like Model, Controller, and View are kept in different folders, and MVC uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search etc. Each of these units have their own logical component views, controllers, and models. In this scenario, you can use Areas to physically partition the business components in the same project.

An area can be defined as smaller functional units in an ASP.NET Core MVC project with its own set of controllers, views, and models.

Consider using Areas in an MVC project when:

- Your application is made of multiple high-level functional components that should be logically separated
- You want to partition your MVC project so that each functional area can be worked on independently

Area features:

- An ASP.NET Core MVC app can have any number of areas
- Each area has its own controllers, models, and views
- Allows you to organize large MVC projects into multiple high-level components that can be worked on independently
- Supports multiple controllers with the same name - as long as they have different *areas*

Let's take a look at an example to illustrate how Areas are created and used. Let's say you have a store app that has two distinct groupings of controllers and views: Products and Services. A typical folder structure for that using MVC areas looks like below:

- Project name
 - Areas
 - Products
 - HomeController.cs
 - ManageController.cs
 - Views
 - Home

- Index.cshtml
 - Manage
 - Index.cshtml
- Services
 - Controllers
 - HomeController.cs
 - Views
 - Home
 - Index.cshtml

When MVC tries to render a view in an Area, by default, it tries to look in the following locations:

```
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml
/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml
/Views/Shared/<Action-Name>.cshtml
```

These are the default locations which can be changed via the `AreaViewLocationFormats` on the `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions`.

For example, in the below code instead of having the folder name as 'Areas', it has been changed to 'Categories'.

```
services.Configure<RazorViewEngineOptions>(options =>
{
    optionsAreaViewLocationFormats.Clear();
    optionsAreaViewLocationFormats.Add("/Categories/{2}/Views/{1}/{0}.cshtml");
    optionsAreaViewLocationFormats.Add("/Categories/{2}/Views/Shared/{0}.cshtml");
    optionsAreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
});
```

One thing to note is that the structure of the *Views* folder is the only one which is considered important here and the content of the rest of the folders like *Controllers* and *Models* does **not** matter. For example, you need not have a *Controllers* and *Models* folder at all. This works because the content of *Controllers* and *Models* is just code which gets compiled into a .dll where as the content of the *Views* is not until a request to that view has been made.

Once you've defined the folder hierarchy, you need to tell MVC that each controller is associated with an area. You do that by decorating the controller name with the `[Area]` attribute.

```

...
namespace MyStore.Areas.Products.Controllers
{
    [Area("Products")]
    public class HomeController : Controller
    {
        // GET: /Products/Home/Index
        public IActionResult Index()
        {
            return View();
        }

        // GET: /Products/Home/Create
        public IActionResult Create()
        {
            return View();
        }
    }
}

```

Set up a route definition that works with your newly created areas. The [Routing to Controller Actions](#) article goes into detail about how to create route definitions, including using conventional routes versus attribute routes. In this example, we'll use a conventional route. To do so, open the *Startup.cs* file and modify it by adding the `areaRoute` named route definition below.

```

...
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "areaRoute",
        template: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

```

Browsing to `http://<yourApp>/products`, the `Index` action method of the `HomeController` in the `Products` area will be invoked.

Link Generation

- Generating links from an action within an area based controller to another action within the same controller.

Let's say the current request's path is like `/Products/Home/Create`

HtmlHelper syntax: `@Html.ActionLink("Go to Product's Home Page", "Index")`

TagHelper syntax: `<a asp-action="Index">Go to Product's Home Page`

Note that we need not supply the 'area' and 'controller' values here as they are already available in the context of the current request. These kind of values are called `ambient` values.

- Generating links from an action within an area based controller to another action on a different controller

Let's say the current request's path is like `/Products/Home/Create`

HtmlHelper syntax: `@Html.ActionLink("Go to Manage Products' Home Page", "Index", "Manage")`

TagHelper syntax: `<a asp-controller="Manage" asp-action="Index">Go to Manage Products' Home Page`

Note that here the ambient value of an 'area' is used but the 'controller' value is specified explicitly above.

- Generating links from an action within an area based controller to another action on a different controller and a different area.

Let's say the current request's path is like `/Products/Home/Create`

HtmlHelper syntax:

```
@Html.ActionLink("Go to Services' Home Page", "Index", "Home", new { area = "Services" })
```

TagHelper syntax:

```
<a asp-area="Services" asp-controller="Home" asp-action="Index">Go to Services' Home Page</a>
```

Note that here no ambient values are used.

- Generating links from an action within an area based controller to another action on a different controller and **not** in an area.

HtmlHelper syntax:

```
@Html.ActionLink("Go to Manage Products' Home Page", "Index", "Home", new { area = "" })
```

TagHelper syntax:

```
<a asp-area="" asp-controller="Manage" asp-action="Index">Go to Manage Products' Home Page</a>
```

Since we want to generate links to a non-area based controller action, we empty the ambient value for 'area' here.

Publishing Areas

All `*.cshtml` and `wwwroot/**` files are published to output when `<Project Sdk="Microsoft.NET.Sdk.Web">` is included in the `.csproj` file.

Application Parts in ASP.NET Core

10/2/2017 • 4 min to read • [Edit Online](#)

[View or download sample code \(how to download\)](#)

An *Application Part* is an abstraction over the resources of an application, from which MVC features like controllers, view components, or tag helpers may be discovered. One example of an application part is an *AssemblyPart*, which encapsulates an assembly reference and exposes types and compilation references. *Feature providers* work with application parts to populate the features of an ASP.NET Core MVC app. The main use case for application parts is to allow you to configure your app to discover (or avoid loading) MVC features from an assembly.

Introducing Application Parts

MVC apps load their features from [application parts](#). In particular, the [AssemblyPart](#) class represents an application part that is backed by an assembly. You can use these classes to discover and load MVC features, such as controllers, view components, tag helpers, and razor compilation sources. The [ApplicationPartManager](#) is responsible for tracking the application parts and feature providers available to the MVC app. You can interact with the `ApplicationPartManager` in `Startup` when you configure MVC:

```
// create an assembly part from a class's assembly
var assembly = typeof(Startup).GetTypeInfo().Assembly;
services.AddMvc()
    .AddApplicationPart(assembly);

// OR
var assembly = typeof(Startup).GetTypeInfo().Assembly;
var part = new AssemblyPart(assembly);
services.AddMvc()
    .ConfigureApplicationPartManager(apm => p.ApplicationParts.Add(part));
```

By default MVC will search the dependency tree and find controllers (even in other assemblies). To load an arbitrary assembly (for instance, from a plugin that isn't referenced at compile time), you can use an application part.

You can use application parts to *avoid* looking for controllers in a particular assembly or location. You can control which parts (or assemblies) are available to the app by modifying the `ApplicationParts` collection of the `ApplicationPartManager`. The order of the entries in the `ApplicationParts` collection is not important. It is important to fully configure the `ApplicationPartManager` before using it to configure services in the container. For example, you should fully configure the `ApplicationPartManager` before invoking `AddControllersAsServices`. Failing to do so, will mean that controllers in application parts added after that method call will not be affected (will not get registered as services) which might result in incorrect behavior of your application.

If you have an assembly that contains controllers you do not want to be used, remove it from the `ApplicationPartManager`:

```

services.AddMvc()
    .ConfigureApplicationPartManager(p =>
    {
        var dependentLibrary = p.ApplicationParts
            .FirstOrDefault(part => part.Name == "DependentLibrary");

        if (dependentLibrary != null)
        {
            p.ApplicationParts.Remove(dependentLibrary);
        }
    })

```

In addition to your project's assembly and its dependent assemblies, the `ApplicationPartManager` will include parts for `Microsoft.AspNetCore.Mvc.TagHelpers` and `Microsoft.AspNetCore.Mvc.Razor` by default.

Application Feature Providers

Application Feature Providers examine application parts and provide features for those parts. There are built-in feature providers for the following MVC features:

- [Controllers](#)
- [Metadata Reference](#)
- [Tag Helpers](#)
- [View Components](#)

Feature providers inherit from `IApplicationFeatureProvider<T>`, where `T` is the type of the feature. You can implement your own feature providers for any of MVC's feature types listed above. The order of feature providers in the `ApplicationPartManager.FeatureProviders` collection can be important, since later providers can react to actions taken by previous providers.

Sample: Generic controller feature

By default, ASP.NET Core MVC ignores generic controllers (for example, `SomeController<T>`). This sample uses a controller feature provider that runs after the default provider and adds generic controller instances for a specified list of types (defined in `EntityTypes.Types`):

```

public class GenericControllerFeatureProvider : IApplicationFeatureProvider<ControllerFeature>
{
    public void PopulateFeature(IEnumerable<ApplicationPart> parts, ControllerFeature feature)
    {
        // This is designed to run after the default ControllerTypeProvider,
        // so the list of 'real' controllers has already been populated.
        foreach (var entityType in EntityTypes.Types)
        {
            var typeName = entityType.Name + "Controller";
            if (!feature.Controllers.Any(t => t.Name == typeName))
            {
                // There's no 'real' controller for this entity, so add the generic version.
                var controllerType = typeof(GenericController<>)
                    .MakeGenericType(entityType.AsType()).GetTypeInfo();
                feature.Controllers.Add(controllerType);
            }
        }
    }
}

```

The entity types:

```

public static class EntityTypes
{
    public static IReadOnlyList<TypeInfo> Types => new List<TypeInfo>()
    {
        typeof(Sprocket).GetTypeInfo(),
        typeof(Widget).GetTypeInfo(),
    };

    public class Sprocket { }
    public class Widget { }
}

```

The feature provider is added in `Startup`:

```

services.AddMvc()
    .ConfigureApplicationPartManager(p =>
        p.FeatureProviders.Add(new GenericControllerFeatureProvider()));

```

By default, the generic controller names used for routing would be of the form *GenericController`1[Widget]* instead of *Widget*. The following attribute is used to modify the name to correspond to the generic type used by the controller:

```

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System;

namespace AppPartsSample
{
    // Used to set the controller name for routing purposes. Without this convention the
    // names would be like 'GenericController`1[Widget]' instead of 'Widget'.
    //
    // Conventions can be applied as attributes or added to MvcOptions.Conventions.
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = true)]
    public class GenericControllerNameConvention : Attribute, IControllerModelConvention
    {
        public void Apply(ControllerModel controller)
        {
            if (controller.ControllerType.GetGenericTypeDefinition() !=
                typeof(GenericController<>))
            {
                // Not a GenericController, ignore.
                return;
            }

            var entityType = controller.ControllerType.GenericTypeArguments[0];
            controller.ControllerName = entityType.Name;
        }
    }
}

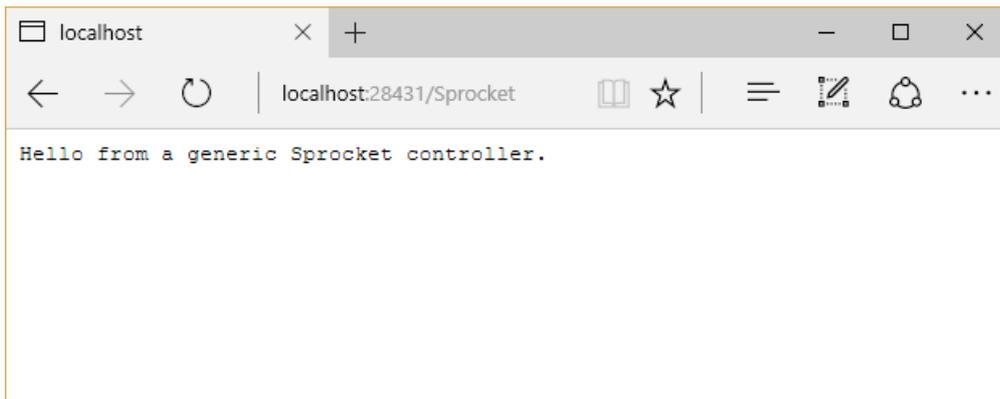
```

The `GenericController` class:

```
using Microsoft.AspNetCore.Mvc;

namespace AppPartsSample
{
    [GenericControllerNameConvention] // Sets the controller name based on typeof(T).Name
    public class GenericController<T> : Controller
    {
        public IActionResult Index()
        {
            return Content($"Hello from a generic {typeof(T).Name} controller.");
        }
    }
}
```

The result, when a matching route is requested:



Sample: Display available features

You can iterate through the populated features available to your app by requesting an `ApplicationPartManager` through [dependency injection](#) and using it to populate instances of the appropriate features:

```

using AppPartsSample.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AppPartsSample.Controllers
{
    public class FeaturesController : Controller
    {
        private readonly ApplicationPartManager _partManager;

        public FeaturesController(ApplicationPartManager partManager)
        {
            _partManager = partManager;
        }

        public IActionResult Index()
        {
            var viewModel = new FeaturesViewModel();

            var controllerFeature = new ControllerFeature();
            _partManager.PopulateFeature(controllerFeature);
            viewModel.Controllers = controllerFeature.Controllers.ToList();

            var metaDataReferenceFeature = new MetadataReferenceFeature();
            _partManager.PopulateFeature(metaDataReferenceFeature);
            viewModel.MetadataReferences = metaDataReferenceFeature.MetadataReferences
                .ToList();

            var tagHelperFeature = new TagHelperFeature();
            _partManager.PopulateFeature(tagHelperFeature);
            viewModel.TagHelpers = tagHelperFeature.TagHelpers.ToList();

            var viewComponentFeature = new ViewComponentFeature();
            _partManager.PopulateFeature(viewComponentFeature);
            viewModel.ViewComponents = viewComponentFeature.ViewComponents.ToList();

            return View(viewModel);
        }
    }
}

```

Example output:

Home Page - AppPartS: × +

localhost:28431/Features

Features

Controllers:

- FeaturesController
- HomeController
- HelloController
- GenericController`1
- GenericController`1

Metadata References:

- C:\dev\ssmith\github.com\aspnet-docs\aspnetcore\mvc\extensibility\app-parts\sample\src\AppPartSample\bin\Debug\netcoreapp1.0\AppPartSample.dll
- C:\Users\steve_000\nuget\packages\Microsoft.AspNetCore.Antiforgery\1.0.1\lib\netstandard1.3\Microsoft.AspNetCore.Antiforgery.dll
- C:\Users\steve_000\nuget\packages\Microsoft.AspNetCore.Authorization\1.0.0

Tag Helpers:

- AnchorTagHelper
- CacheTagHelper
- DistributedCacheTagHelper
- EnvironmentTagHelper
- FormTagHelper

View Components:

- AnchorTagHelper
- CacheTagHelper
- DistributedCacheTagHelper
- EnvironmentTagHelper
- FormTagHelper

[Home](#)

Custom Model Binding

9/22/2017 • 6 min to read • [Edit Online](#)

By [Steve Smith](#)

Model binding allows controller actions to work directly with model types (passed in as method arguments), rather than HTTP requests. Mapping between incoming request data and application models is handled by model binders. Developers can extend the built-in model binding functionality by implementing custom model binders (though typically, you don't need to write your own provider).

[View or download sample from GitHub](#)

Default model binder limitations

The default model binders support most of the common .NET Core data types and should meet most developers' needs. They expect to bind text-based input from the request directly to model types. You might need to transform the input prior to binding it. For example, when you have a key that can be used to look up model data. You can use a custom model binder to fetch data based on the key.

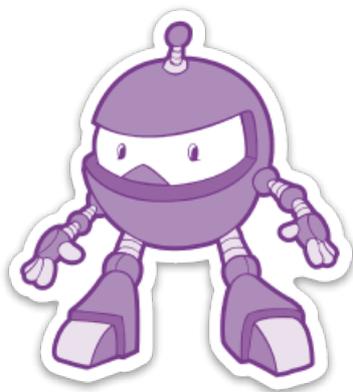
Model binding review

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string in the input. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter`. We recommend you create a type converter if you have a simple `string` -> `SomeType` mapping that doesn't require external resources.

Before creating your own custom model binder, it's worth reviewing how existing model binders are implemented. Consider the [ByteArrayModelBinder](#) which can be used to convert base64-encoded strings into byte arrays. The byte arrays are often stored as files or database BLOB fields.

Working with the ByteArrayModelBinder

Base64-encoded strings can be used to represent binary data. For example, the following image can be encoded as a string.



A small portion of the encoded string is shown in the following image:

```
Base64 Encoded Dotnet Bot.txt - Notepad
File Edit Format View Help
iVBORw0KGgoAAAANSUHEUgAAAQAAAEFCAYAAADq1vKRAAAAAXNSR0IArs4c6QAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsMAAA7DAcdvqQAAAF4vSURBVHhe7d0L0G51dSf4zLXnVjM105WSsiiLoqAwioUU1mU5Z2
acDCbT1s1IVzLV0t3V1ZZKTabSyaqTRukkTroNalqaCJmIoLYhyIgjSKLQKkcNGhAvNCThEjGo5BQXD9KcN
B4oQcyZ/dvu9bn0/p537/1+7+W7Patq1fn0d9nv3s+z1v/5r8vz7B
+rUqVK1SpVq1SpUqVK1SpVq1SpUqVK1SpVq1SpUqVK1SpVViP/wYhWqVJ1j0s4+3/Y6X/U6X/c6X+Svo6f
+b0KEFwQ7DEJIAgA4Pz/aaN/o9H/rNH/oqB+RgMoAhwqQFSpsos1nBgYBBAAgf+80f+y0f+q0f+m0f
+20f8uqf//14360YDwNwE0r1WBoUqVXSYYBBMEKgEAAAKf/8UZPaPSF11577d+6+eab39jX5mcn
+nn3u/6mDxA1tKhSZdIZgboP0Cw6rdgcPHFF7/64YcfvVTo0aN3HBsRv3Pfffdd/PM//OnNX/7NxsFKsD
BNbG0CgxVquxw4ZyCNLMDocALrP5TgKakzz//F8Bkg4cMAfXxBoAQ
+Qbq1SpssMkAIGTBjv4cY585MiRwzr/Xkiee
+65h4UbzXWFHoBBXkI44TMrW6hSZQdJhAxWbis4hz2BA3PkzqeXI//vefuvP009/SXF+
+IUIKIFQZQ5Uq00QyIARDeMH111//U2h/58tL13vvvfeFNJ
+DMQAGYURkGcPjqfJ1GyVCBjkEDKENGTCeVQICwRi6UAJjkmQES06jsouUqVbZJAhCszhwSjf
+bqgsctvPd5FK9o88e
+/aDjx979pnnuu9ME6HJ0eccc1bzuRgDtqDaUd1C1SrbJAEIHBEGcMyTnnrqa90PjtT/voHf33sz2978Nh
1//ymY7/++t8+9tb//t0b+s5z33fsk
+/9/LGjTz7d/faw3H//Vf43Eb7YUSVK1XWkDmPoAKgTHjS3Xff/Uudr86UP/3Mnx979//+/u0AoS/
+j//5rE7b76n+6vZIkxJbAE4RXNTZQtVqxr0JxsP5ZgdeaQp45VGu761L3HLvoFLjn0+f3/4r/93mPv+N
+u2MQa/OyBL32r++vZcvtvt/9y8/lyC71/oYJC1SprFKEDx4vy40nKhJ2PFuW+Wx84DhAAwQf/ye8fu
+FffvrYxy65ZUN/79f/zbHf
+JmrNn7v137q8mOPP/REd5WyHD58+HPuodEIIYBVBYUqVdYouUHpBY00Nig99Z2jbTgQjn7J3/3AcUBQ0nf
9nSs3fv8jv/aJ7kp1efbZ7Z7/b3MPJ3b24JyEE4KrAUKXKGoSjidn1ErCEf772ta99deefRfnYuz694eBYQJ
8d1PT33/nJDWaBVYxJcx+nNhp5BYBVqx8VqxxJ0JrQAU1H1190zTXXvKnzU2i3Pgrr7m0de5/9j/
+q20//45PFkGgpP/Pay9r/+5t/8t7uqvNluY+7IuQV6igUKXKmoWjRRmyrTooC3a
+uUnkEoI1/KvzP1h0/1kqn
+DvhB5j0txHgAL2UkGhSpU1CSctq4vZ0Z8Y/pS77rnrng51vbpLPfPD2DVB4/wW/V3T+Wtpn
+PDiRnMFOoJC1SprEE7G2azEWovF8KcNgcKn3/eFDVD43bfewHT+ksoPxN
```

Follow the instructions in the [sample's README](#) to convert the base64-encoded string into a file.

ASP.NET Core MVC can take a base64-encoded strings and use a `ByteArrayModelBinder` to convert it into a byte array. The `ByteArrayModelBinderProvider` which implements `IModelBinderProvider` maps `byte[]` arguments to `ByteArrayModelBinder`:

```
public IModelBinder GetBinder(ModelBinderProviderContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (context.Metadata.ModelType == typeof(byte[]))
    {
        return new ByteArrayModelBinder();
    }

    return null;
}
```

When creating your own custom model binder, you can implement your own `IModelBinderProvider` type, or use the `ModelBinderAttribute`.

The following example shows how to use `ByteArrayModelBinder` to convert a base64-encoded string to a `byte[]` and save the result to a file:

```
// POST: api/image
[HttpPost]
public void Post(byte[] file, string filename)
{
    string filePath = Path.Combine(_env.ContentRootPath, "wwwroot/images/upload", filename);
    if (System.IO.File.Exists(filePath)) return;
    System.IO.File.WriteAllBytes(filePath, file);
}
}
```

You can POST a base64-encoded string to this api method using a tool like [Postman](#):

The screenshot shows the Postman interface. The URL is `http://localhost:64513/api/image`. The method is `POST`. The body is set to `form-data`. The parameters are:

Key	Value
file	iVBORw0KGgoAAAANSU... (base64 encoded image data)
filename	bot.png

The response status is `200 OK` and the time taken is `38 ms`. The response body is currently empty.

As long as the binder can bind request data to appropriately named properties or arguments, model binding will succeed. The following example shows how to use `ByteArrayModelBinder` with a view model:

```
[HttpPost("Profile")]
public void SaveProfile(ProfileViewModel model)
{
    string filePath = Path.Combine(_env.ContentRootPath, "wwwroot/images/upload", model.FileName);
    if (System.IO.File.Exists(model.FileName)) return;
    System.IO.File.WriteAllBytes(filePath, model.File);
}

public class ProfileViewModel
{
    public byte[] File { get; set; }
    public string FileName { get; set; }
}
```

Custom model binder sample

In this section we'll implement a custom model binder that:

- Converts incoming request data into strongly typed key arguments.
- Uses Entity Framework Core to fetch the associated entity.
- Passes the associated entity as an argument to the action method.

The following sample uses the `ModelBinder` attribute on the `Author` model:

```
using CustomModelBindingSample.Binders;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace CustomModelBindingSample.Data
{
    [ModelBinder(BinderType = typeof(AuthorEntityBinder))]
    public class Author
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string GitHub { get; set; }
        public string Twitter { get; set; }
        public string BlogUrl { get; set; }
    }
}
```

In the preceding code, the `ModelBinder` attribute specifies the type of `IModelBinder` that should be used to bind `Author` action parameters.

The `AuthorEntityBinder` is used to bind an `Author` parameter by fetching the entity from a data source using Entity Framework Core and an `authorId`:

```

public class AuthorEntityBinder : IModelBinder
{
    private readonly AppDbContext _db;
    public AuthorEntityBinder(AppDbContext db)
    {
        _db = db;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException(nameof(bindingContext));
        }

        // Specify a default argument name if none is set by ModelBinderAttribute
        var modelName = bindingContext.BinderModelName;
        if (string.IsNullOrEmpty(modelName))
        {
            modelName = "authorId";
        }

        // Try to fetch the value of the argument by name
        var valueProviderResult =
            bindingContext.ValueProvider.GetValue(modelName);

        if (valueProviderResult == ValueProviderResult.None)
        {
            return Task.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName,
            valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return Task.CompletedTask;
        }

        int id = 0;
        if (!int.TryParse(value, out id))
        {
            // Non-integer arguments result in model state errors
            bindingContext.ModelState.TryAddModelError(
                bindingContext.ModelName,
                "Author Id must be an integer.");
            return Task.CompletedTask;
        }

        // Model will be null if not found, including for
        // out of range id values (0, -3, etc.)
        var model = _db.Authors.Find(id);
        bindingContext.Result = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}

```

The following code shows how to use the `AuthorEntityBinder` in an action method:

```
[HttpGet("get/{authorId}")]
public IActionResult Get(Author author)
{
    return Ok(author);
}
```

The `ModelBinder` attribute can be used to apply the `AuthorEntityBinder` to parameters that do not use default conventions:

```
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")]Author author)
{
    if (author == null)
    {
        return NotFound();
    }
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    return Ok(author);
}
```

In this example, since the name of the argument is not the default `authorId`, it's specified on the parameter using `ModelBinder` attribute. Note that both the controller and action method are simplified compared to looking up the entity in the action method. The logic to fetch the author using Entity Framework Core is moved to the model binder. This can be considerable simplification when you have several methods that bind to the author model, and can help you to follow the [DRY principle](#).

You can apply the `ModelBinder` attribute to individual model properties (such as on a viewmodel) or to action method parameters to specify a certain model binder or model name for just that type or action.

Implementing a ModelBinderProvider

Instead of applying an attribute, you can implement `IModelBinderProvider`. This is how the built-in framework binders are implemented. When you specify the type your binder operates on, you specify the type of argument it produces, **not** the input your binder accepts. The following binder provider works with the `AuthorEntityBinder`. When it's added to MVC's collection of providers, you don't need to use the `ModelBinder` attribute on `Author` or `Author` typed parameters.

```

using CustomModelBindingSample.Data;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using System;

namespace CustomModelBindingSample.Binders
{
    public class AuthorEntityBinderProvider : IModelBinderProvider
    {
        {
            public IModelBinder GetBinder(ModelBinderProviderContext context)
            {
                {
                    if (context == null)
                    {
                        throw new ArgumentNullException(nameof(context));
                    }

                    if (context.Metadata.ModelType == typeof(Author))
                    {
                        return new BinderTypeModelBinder(typeof(AuthorEntityBinder));
                    }

                    return null;
                }
            }
        }
    }
}

```

Note: The preceding code returns a `BinderTypeModelBinder`. `BinderTypeModelBinder` acts as a factory for model binders and provides dependency injection (DI). The `AuthorEntityBinder` requires DI to access EF Core. Use `BinderTypeModelBinder` if your model binder requires services from DI.

To use a custom model binder provider, add it in `ConfigureServices`:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase());

    services.AddMvc(options =>
    {
        // add custom binder to beginning of collection
        options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
    });
}

```

When evaluating model binders, the collection of providers is examined in order. The first provider that returns a binder is used.

The following image shows the default model binders from the debugger.

Autos	
Name	Value
options.ModelBindi	Count = 14
[0]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.BinderTypeModelBinderProvider}
[1]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ServicesModelBinderProvider}
[2]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.BodyModelBinderProvider}
[3]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.HeaderModelBinderProvider}
[4]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.SimpleTypeModelBinderProvider}
[5]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.CancellationTokenModelBinderProvider}
[6]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ByteArrayModelBinderProvider}
[7]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormFileModelBinderProvider}
[8]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormCollectionModelBinderProvider}
[9]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.KeyValuePairModelBinderProvider}
[10]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DictionaryModelBinderProvider}
[11]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ArrayModelBinderProvider}
[12]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.CollectionModelBinderProvider}
[13]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ComplexTypeModelBinderProvider}

Adding your provider to the end of the collection may result in a built-in model binder being called before your custom binder has a chance. In this example, the custom provider is added to the beginning of the collection to ensure it is used for `Author` action arguments.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase());

    services.AddMvc(options =>
    {
        // add custom binder to beginning of collection
        options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
    });
}
```

Recommendations and best practices

Custom model binders:

- Should not attempt to set status codes or return results (for example, 404 Not Found). If model binding fails, an [action filter](#) or logic within the action method itself should handle the failure.
- Are most useful for eliminating repetitive code and cross-cutting concerns from action methods.
- Typically should not be used to convert a string into a custom type, a [TypeConverter](#) is usually a better option.

Custom formatters in ASP.NET Core MVC web APIs

9/22/2017 • 4 min to read • [Edit Online](#)

By [Tom Dykstra](#)

ASP.NET Core MVC has built-in support for data exchange in web APIs by using JSON, XML, or plain text formats. This article shows how to add support for additional formats by creating custom formatters.

[View or download sample from GitHub.](#)

When to use custom formatters

Use a custom formatter when you want the [content negotiation](#) process to support a content type that isn't supported by the built-in formatters (JSON, XML, and plain text).

For example, if some of the clients for your web API can handle the [Protobuf](#) format, you might want to use Protobuf with those clients because it's more efficient. Or you might want your web API to send contact names and addresses in [vCard](#) format, a commonly used format for exchanging contact data. The sample app provided with this article implements a simple vCard formatter.

Overview of how to use a custom formatter

Here are the steps to create and use a custom formatter:

- Create an output formatter class if you want to serialize data to send to the client.
- Create an input formatter class if you want to deserialize data received from the client.
- Add instances of your formatters to the `InputFormatters` and `OutputFormatters` collections in [MvcOptions](#).

The following sections provide guidance and code examples for each of these steps.

How to create a custom formatter class

To create a formatter:

- Derive the class from the appropriate base class.
- Specify valid media types and encodings in the constructor.
- Override `CanReadType` / `CanWriteType` methods
- Override `ReadRequestBodyAsync` / `WriteResponseBodyAsync` methods

Derive from the appropriate base class

For text media types (for example, vCard), derive from the [TextInputFormatter](#) or [TextOutputFormatter](#) base class.

```
public class VcardOutputFormatter : TextOutputFormatter
```

For binary types, derive from the [InputFormatter](#) or [OutputFormatter](#) base class.

Specify valid media types and encodings

In the constructor, specify valid media types and encodings by adding to the `SupportedMediaTypes` and `SupportedEncodings` collections.

```
public VcardOutputFormatter()
{
    SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
```

NOTE

You can't do constructor dependency injection in a formatter class. For example, you can't get a logger by adding a logger parameter to the constructor. To access services, you have to use the context object that gets passed in to your methods. A code example [below](#) shows how to do this.

Override CanReadType/CanWriteType

Specify the type you can deserialize into or serialize from by overriding the `CanReadType` or `CanWriteType` methods. For example, you might only be able to create vCard text from a `Contact` type and vice versa.

```
protected override bool CanWriteType(Type type)
{
    if (typeof(Contact).IsAssignableFrom(type)
        || typeof(IEnumerable<Contact>).IsAssignableFrom(type))
    {
        return base.CanWriteType(type);
    }
    return false;
}
```

The CanWriteResult method

In some scenarios you have to override `CanWriteResult` instead of `CanWriteType`. Use `CanWriteResult` if the following conditions are true:

- Your action method returns a model class.
- There are derived classes which might be returned at runtime.
- You need to know at runtime which derived class was returned by the action.

For example, suppose your action method signature returns a `Person` type, but it may return a `Student` or `Instructor` type that derives from `Person`. If you want your formatter to handle only `Student` objects, check the type of `Object` in the context object provided to the `CanWriteResult` method. Note that it's not necessary to use `CanWriteResult` when the action method returns `ActionResult`; in that case, the `CanWriteType` method receives the runtime type.

Override ReadRequestBodyAsync/WriteResponseBodyAsync

You do the actual work of deserializing or serializing in `ReadRequestBodyAsync` or `WriteResponseBodyAsync`. The highlighted lines in the following example show how to get services from the dependency injection container (you can't get them from constructor parameters).

```

public override Task WriteResponseBodyAsync(OutputFormatterWriteContext context, Encoding selectedEncoding)
{
    IServiceProvider serviceProvider = context.HttpContext.RequestServices;
    var logger = serviceProvider.GetService(typeof(ILogger<VcardOutputFormatter>)) as ILogger;

    var response = context.HttpContext.Response;

    var buffer = new StringBuilder();
    if (context.Object is IEnumerable<Contact>)
    {
        foreach (Contact contact in context.Object as IEnumerable<Contact>)
        {
            FormatVcard(buffer, contact, logger);
        }
    }
    else
    {
        var contact = context.Object as Contact;
        FormatVcard(buffer, contact, logger);
    }
    return response.WriteAsync(buffer.ToString());
}

private static void FormatVcard(StringBuilder buffer, Contact contact, ILogger logger)
{
    buffer.AppendLine("BEGIN:VCARD");
    buffer.AppendLine("VERSION:2.1");
    buffer.AppendFormat($"N:{contact.LastName};{contact.FirstName}\r\n");
    buffer.AppendFormat($"FN:{contact.FirstName} {contact.LastName}\r\n");
    buffer.AppendFormat($"UID:{contact.ID}\r\n");
    buffer.AppendLine("END:VCARD");
    logger.LogInformation($"Writing {contact.FirstName} {contact.LastName}");
}

```

How to configure MVC to use a custom formatter

To use a custom formatter, add an instance of the formatter class to the `InputFormatters` Or `OutputFormatters` collection.

```

services.AddMvc(options =>
{
    options.InputFormatters.Insert(0, new VcardInputFormatter());
    options.OutputFormatters.Insert(0, new VcardOutputFormatter());
});

```

Formatters are evaluated in the order you insert them. The first one takes precedence.

Next steps

See the [sample application](#), which implements simple vCard input and output formatters. The application reads and writes vCards that look like the following example:

```

BEGIN:VCARD
VERSION:2.1
N:Davolio;Nancy
FN:Nancy Davolio
UID:20293482-9240-4d68-b475-325df4a83728
END:VCARD

```

To see vCard output, run the application and send a Get request with Accept header "text/vcard" to

`http://localhost:63313/api/contacts/` (when running from Visual Studio) or `http://localhost:5000/api/contacts/` (when running from the command line).

To add a vCard to the in-memory collection of contacts, send a Post request to the same URL, with Content-Type header "text/vcard" and with vCard text in the body, formatted like the example above.

Introduction to formatting response data in ASP.NET Core MVC

11/19/2017 • 8 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET Core MVC has built-in support for formatting response data, using fixed formats or in response to client specifications.

[View or download sample from GitHub.](#)

Format-Specific Action Results

Some action result types are specific to a particular format, such as `JsonResult` and `ContentResult`. Actions can return specific results that are always formatted in a particular manner. For example, returning a `JsonResult` will return JSON-formatted data, regardless of client preferences. Likewise, returning a `ContentResult` will return plain-text-formatted string data (as will simply returning a string).

NOTE

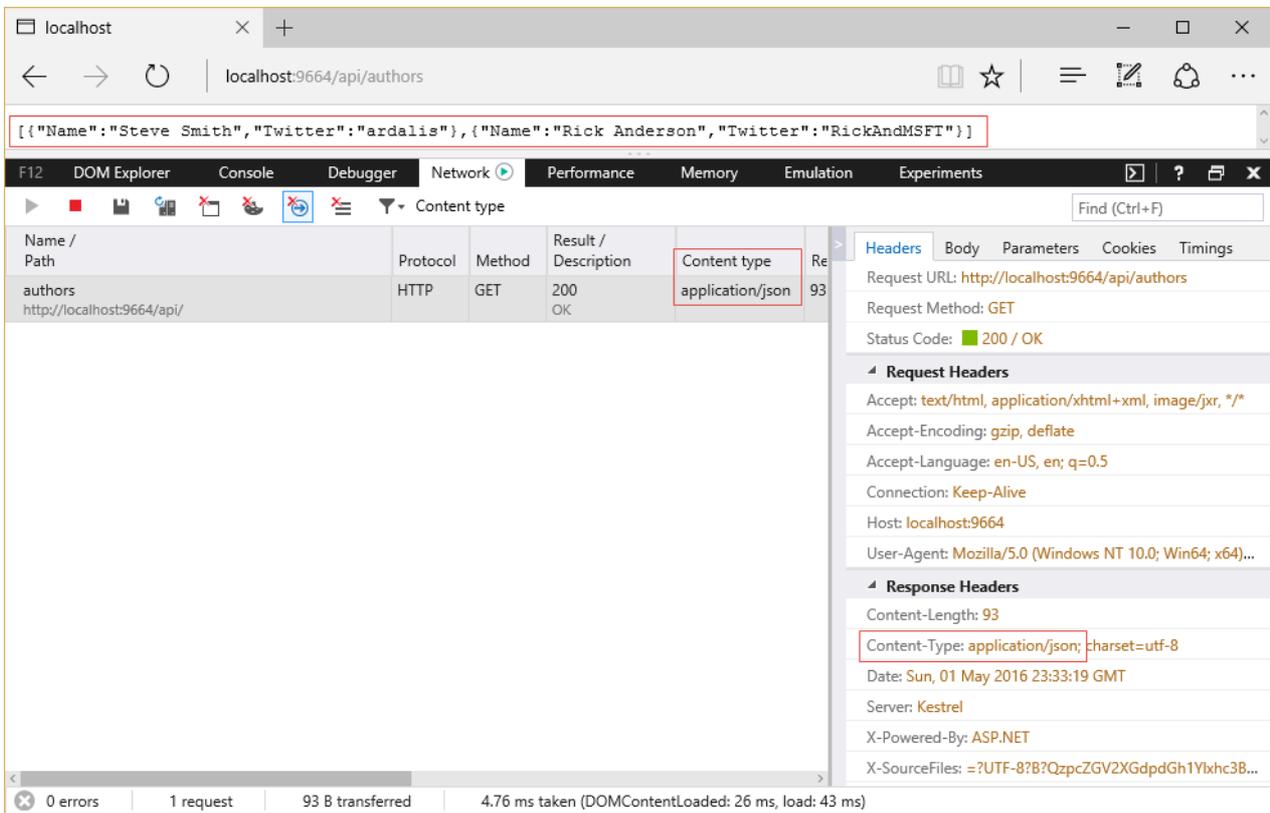
An action isn't required to return any particular type; MVC supports any object return value. If an action returns an `IActionResult` implementation and the controller inherits from `Controller`, developers have many helper methods corresponding to many of the choices. Results from actions that return objects that are not `IActionResult` types will be serialized using the appropriate `IOutputFormatter` implementation.

To return data in a specific format from a controller that inherits from the `Controller` base class, use the built-in helper method `Json` to return JSON and `Content` for plain text. Your action method should return either the specific result type (for instance, `JsonResult`) or `IActionResult`.

Returning JSON-formatted data:

```
// GET: api/authors
[HttpGet]
public JsonResult Get()
{
    return Json(_authorRepository.List());
}
```

Sample response from this action:



Note that the content type of the response is `application/json`, shown both in the list of network requests and in the Response Headers section. Also note the list of options presented by the browser (in this case, Microsoft Edge) in the Accept header in the Request Headers section. The current technique is ignoring this header; obeying it is discussed below.

To return plain text formatted data, use `ContentResult` and the `Content` helper:

```
// GET api/authors/about
[HttpGet("About")]
public ContentResult About()
{
    return Content("An API listing authors of docs.asp.net.");
}
```

A response from this action:

The screenshot shows a browser window at localhost:9664/api/authors/about. The developer tools network tab is open, displaying a single request. The request details are as follows:

Name / Path	Protocol	Method	Result / Description	Content type	Re
about http://localhost:9664/api/authors/	HTTP	GET	200 OK	text/plain	15

The response headers are:

- Request URL: http://localhost:9664/api/authors/about
- Request Method: GET
- Status Code: 200 / OK
- Request Headers:
 - Accept: text/html, application/xhtml+xml, image/jxr, */*
 - Accept-Encoding: gzip, deflate
 - Accept-Language: en-US, en; q=0.5
 - Connection: Keep-Alive
 - Host: localhost:9664
 - User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64...
- Response Headers:
 - Content-Encoding: gzip
 - Content-Length: 152
 - Content-Type: text/plain; charset=utf-8
 - Date: Sun, 01 May 2016 23:40:02 GMT
 - Server: Kestrel
 - Vary: Accept-Encoding

At the bottom of the developer tools, the status bar shows: 0 errors, 1 request, 152 B transferred, 1.19 s taken (DOMContentLoaded: 1.21 s, load: 1.22 s).

Note in this case the `Content-Type` returned is `text/plain`. You can also achieve this same behavior using just a string response type:

```
// GET api/authors/version
[HttpGet("version")]
public string Version()
{
    return "Version 1.0.0";
}
```

TIP

For non-trivial actions with multiple return types or options (for example, different HTTP status codes based on the result of operations performed), prefer `ActionResult` as the return type.

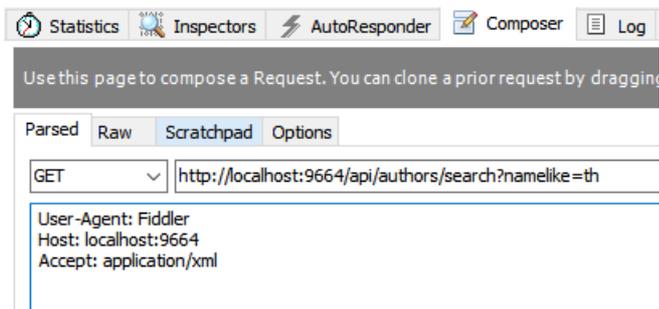
Content Negotiation

Content negotiation (*conneg* for short) occurs when the client specifies an `Accept` header. The default format used by ASP.NET Core MVC is JSON. Content negotiation is implemented by `ObjectResult`. It is also built into the status code specific action results returned from the helper methods (which are all based on `ObjectResult`). You can also return a model type (a class you've defined as your data transfer type) and the framework will automatically wrap it in an `ObjectResult` for you.

The following action method uses the `Ok` and `NotFound` helper methods:

```
// GET: api/authors/search?namelike=th
[HttpGet("Search")]
public IActionResult Search(string namelike)
{
    var result = _authorRepository.GetByNameSubstring(namelike);
    if (!result.Any())
    {
        return NotFound(namelike);
    }
    return Ok(result);
}
```

A JSON-formatted response will be returned unless another format was requested and the server can return the requested format. You can use a tool like [Fiddler](#) to create a request that includes an Accept header and specify another format. In that case, if the server has a *formatter* that can produce a response in the requested format, the result will be returned in the client-preferred format.



In the above screenshot, the Fiddler Composer has been used to generate a request, specifying `Accept: application/xml`. By default, ASP.NET Core MVC only supports JSON, so even when another format is specified, the result returned is still JSON-formatted. You'll see how to add additional formatters in the next section.

Controller actions can return POCOs (Plain Old CLR Objects), in which case ASP.NET MVC will automatically create an `ObjectResult` for you that wraps the object. The client will get the formatted serialized object (JSON format is the default; you can configure XML or other formats). If the object being returned is `null`, then the framework will return a `204 No Content` response.

Returning an object type:

```
// GET api/authors/ardalis
[HttpGet("{alias}")]
public Author Get(string alias)
{
    return _authorRepository.GetByAlias(alias);
}
```

In the sample, a request for a valid author alias will receive a 200 OK response with the author's data. A request for an invalid alias will receive a 204 No Content response. Screenshots showing the response in XML and JSON formats are shown below.

Content Negotiation Process

Content *negotiation* only takes place if an `Accept` header appears in the request. When a request contains an accept header, the framework will enumerate the media types in the accept header in preference order and will try to find a formatter that can produce a response in one of the formats specified by the accept header. In case no formatter is found that can satisfy the client's request, the framework will try to find the first formatter that can produce a response (unless the developer has configured the option on `MvcOptions` to return 406 Not Acceptable instead). If the request specifies XML, but the XML formatter has not been configured, then the JSON formatter

will be used. More generally, if no formatter is configured that can provide the requested format, then the first formatter that can format the object is used. If no header is given, the first formatter that can handle the object to be returned will be used to serialize the response. In this case, there isn't any negotiation taking place - the server is determining what format it will use.

NOTE

If the Accept header contains `*/*`, the Header will be ignored unless `RespectBrowserAcceptHeader` is set to true on `MvcOptions`.

Browsers and Content Negotiation

Unlike typical API clients, web browsers tend to supply `Accept` headers that include a wide array of formats, including wildcards. By default, when the framework detects that the request is coming from a browser, it will ignore the `Accept` header and instead return the content in the application's configured default format (JSON unless otherwise configured). This provides a more consistent experience when using different browsers to consume APIs.

If you would prefer your application honor browser accept headers, you can configure this as part of MVC's configuration by setting `RespectBrowserAcceptHeader` to `true` in the `ConfigureServices` method in `Startup.cs`.

```
services.AddMvc(options =>
{
    options.RespectBrowserAcceptHeader = true; // false by default
});
```

Configuring Formatters

If your application needs to support additional formats beyond the default of JSON, you can add NuGet packages and configure MVC to support them. There are separate formatters for input and output. Input formatters are used by [Model Binding](#); output formatters are used to format responses. You can also configure [Custom Formatters](#).

Adding XML Format Support

To add support for XML formatting, install the `Microsoft.AspNetCore.Mvc.Formatters.Xml` NuGet package.

Add the `XmlSerializerFormatters` to MVC's configuration in `Startup.cs`:

```
services.AddMvc()
    .AddXmlSerializerFormatters();
```

Alternately, you can add just the output formatter:

```
services.AddMvc(options =>
{
    options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
});
```

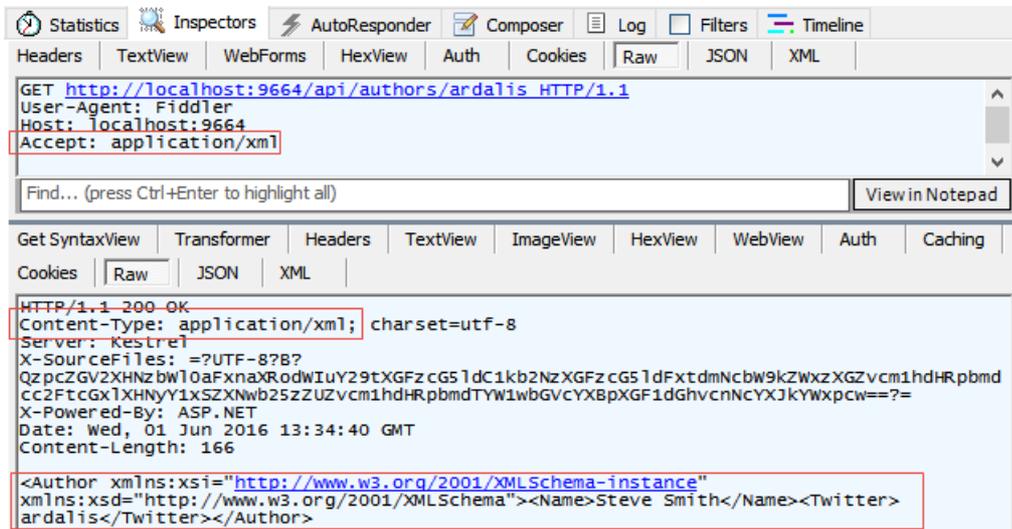
These two approaches will serialize results using `System.Xml.Serialization.XmlSerializer`. If you prefer, you can use the `System.Runtime.Serialization.DataContractSerializer` by adding its associated formatter:

```

services.AddMvc(options =>
{
    options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
});

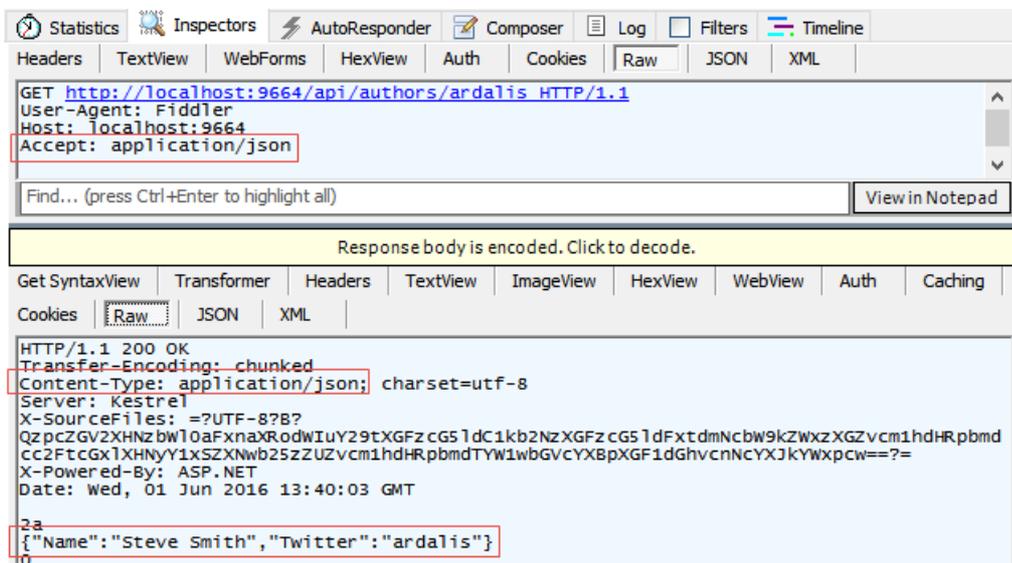
```

Once you've added support for XML formatting, your controller methods should return the appropriate format based on the request's `Accept` header, as this Fiddler example demonstrates:



You can see in the Inspectors tab that the Raw GET request was made with an `Accept: application/xml` header set. The response pane shows the `Content-Type: application/xml` header, and the `Author` object has been serialized to XML.

Use the Composer tab to modify the request to specify `application/json` in the `Accept` header. Execute the request, and the response will be formatted as JSON:



In this screenshot, you can see the request sets a header of `Accept: application/json` and the response specifies the same as its `Content-Type`. The `Author` object is shown in the body of the response, in JSON format.

Forcing a Particular Format

If you would like to restrict the response formats for a specific action you can, you can apply the `[Produces]` filter. The `[Produces]` filter specifies the response formats for a specific action (or controller). Like most `Filters`, this can be applied at the action, controller, or global scope.

```
[Produces("application/json")]
public class AuthorsController
```

The `[Produces]` filter will force all actions within the `AuthorsController` to return JSON-formatted responses, even if other formatters were configured for the application and the client provided an `Accept` header requesting a different, available format. See [Filters](#) to learn more, including how to apply filters globally.

Special Case Formatters

Some special cases are implemented using built-in formatters. By default, `string` return types will be formatted as `text/plain` (`text/html` if requested via `Accept` header). This behavior can be removed by removing the `TextOutputFormatter`. You remove formatters in the `Configure` method in `Startup.cs` (shown below). Actions that have a model object return type will return a 204 No Content response when returning `null`. This behavior can be removed by removing the `HttpNoContentOutputFormatter`. The following code removes the `TextOutputFormatter` and `HttpNoContentOutputFormatter`.

```
services.AddMvc(options =>
{
    options.OutputFormatters.RemoveType<TextOutputFormatter>();
    options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
});
```

Without the `TextOutputFormatter`, `string` return types return 406 Not Acceptable, for example. Note that if an XML formatter exists, it will format `string` return types if the `TextOutputFormatter` is removed.

Without the `HttpNoContentOutputFormatter`, null objects are formatted using the configured formatter. For example, the JSON formatter will simply return a response with a body of `null`, while the XML formatter will return an empty XML element with the attribute `xsi:nil="true"` set.

Response Format URL Mappings

Clients can request a particular format as part of the URL, such as in the query string or part of the path, or by using a format-specific file extension such as `.xml` or `.json`. The mapping from request path should be specified in the route the API is using. For example:

```
[FormatFilter]
public class ProductsController
{
    [Route("[controller]/[action]/{id}.{format?}")]
    public Product GetById(int id)
```

This route would allow the requested format to be specified as an optional file extension. The `[FormatFilter]` attribute checks for the existence of the format value in the `RouteData` and will map the response format to the appropriate formatter when the response is created.

ROUTE	FORMATTER
<code>/products/GetById/5</code>	The default output formatter
<code>/products/GetById/5.json</code>	The JSON formatter (if configured)
<code>/products/GetById/5.xml</code>	The XML formatter (if configured)

Test and debug in ASP.NET Core

1/10/2018 • 1 min to read • [Edit Online](#)

- [Unit testing](#)
- [Integration testing](#)
- [Razor Pages unit and integration testing](#)
- [Test controllers](#)
- [Debug ASP.NET Core 2.x source](#)
- [Remote debugging](#)
- [Snapshot debugging](#)

Integration testing in ASP.NET Core

10/2/2017 • 8 min to read • [Edit Online](#)

By [Steve Smith](#)

Integration testing ensures that an application's components function correctly when assembled together. ASP.NET Core supports integration testing using unit test frameworks and a built-in test web host that can be used to handle requests without network overhead.

[View or download sample code \(how to download\)](#)

Introduction to integration testing

Integration tests verify that different parts of an application work correctly together. Unlike [Unit testing](#), integration tests frequently involve application infrastructure concerns, such as a database, file system, network resources, or web requests and responses. Unit tests use fakes or mock objects in place of these concerns, but the purpose of integration tests is to confirm that the system works as expected with these systems.

Integration tests, because they exercise larger segments of code and because they rely on infrastructure elements, tend to be orders of magnitude slower than unit tests. Thus, it's a good idea to limit how many integration tests you write, especially if you can test the same behavior with a unit test.

NOTE

If some behavior can be tested using either a unit test or an integration test, prefer the unit test, since it will be almost always be faster. You might have dozens or hundreds of unit tests with many different inputs but just a handful of integration tests covering the most important scenarios.

Testing the logic within your own methods is usually the domain of unit tests. Testing how your application works within its framework, for example with ASP.NET Core, or with a database is where integration tests come into play. It doesn't take too many integration tests to confirm that you're able to write a row to the database and read it back. You don't need to test every possible permutation of your data access code - you only need to test enough to give you confidence that your application is working properly.

Integration testing ASP.NET Core

To get set up to run integration tests, you'll need to create a test project, add a reference to your ASP.NET Core web project, and install a test runner. This process is described in the [Unit testing](#) documentation, along with more detailed instructions on running tests and recommendations for naming your tests and test classes.

NOTE

Separate your unit tests and your integration tests using different projects. This helps ensure you don't accidentally introduce infrastructure concerns into your unit tests and lets you easily choose which set of tests to run.

The Test Host

ASP.NET Core includes a test host that can be added to integration test projects and used to host ASP.NET Core applications, serving test requests without the need for a real web host. The provided sample includes an integration test project which has been configured to use [xUnit](#) and the Test Host. It uses the

`Microsoft.AspNetCore.TestHost` NuGet package.

Once the `Microsoft.AspNetCore.TestHost` package is included in the project, you'll be able to create and configure a `TestServer` in your tests. The following test shows how to verify that a request made to the root of a site returns "Hello World!" and should run successfully against the default ASP.NET Core Empty Web template created by Visual Studio.

```
public class PrimeWebDefaultRequestShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;
    public PrimeWebDefaultRequestShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }

    [Fact]
    public async Task ReturnHelloWorld()
    {
        // Act
        var response = await _client.GetAsync("/");
        response.EnsureSuccessStatusCode();

        var responseString = await response.Content.ReadAsStringAsync();

        // Assert
        Assert.Equal("Hello World!",
            responseString);
    }
}
```

This test is using the Arrange-Act-Assert pattern. The Arrange step is done in the constructor, which creates an instance of `TestServer`. A configured `WebHostBuilder` will be used to create a `TestHost`; in this example, the `Configure` method from the system under test (SUT)'s `Startup` class is passed to the `WebHostBuilder`. This method will be used to configure the request pipeline of the `TestServer` identically to how the SUT server would be configured.

In the Act portion of the test, a request is made to the `TestServer` instance for the "/" path, and the response is read back into a string. This string is compared with the expected string of "Hello World!". If they match, the test passes; otherwise, it fails.

Now you can add a few additional integration tests to confirm that the prime checking functionality works via the web application:

```

public class PrimeWebCheckPrimeShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;
    public PrimeWebCheckPrimeShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }

    private async Task<string> GetCheckPrimeResponseString(
        string querystring = "")
    {
        var request = "/checkprime";
        if(!string.IsNullOrEmpty(querystring))
        {
            request += "?" + querystring;
        }
        var response = await _client.GetAsync(request);
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsStringAsync();
    }

    [Fact]
    public async Task ReturnInstructionsGivenEmptyQueryString()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString();

        // Assert
        Assert.Equal("Pass in a number to check in the form /checkprime?5",
            responseString);
    }

    [Fact]
    public async Task ReturnPrimeGiven5()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString("5");

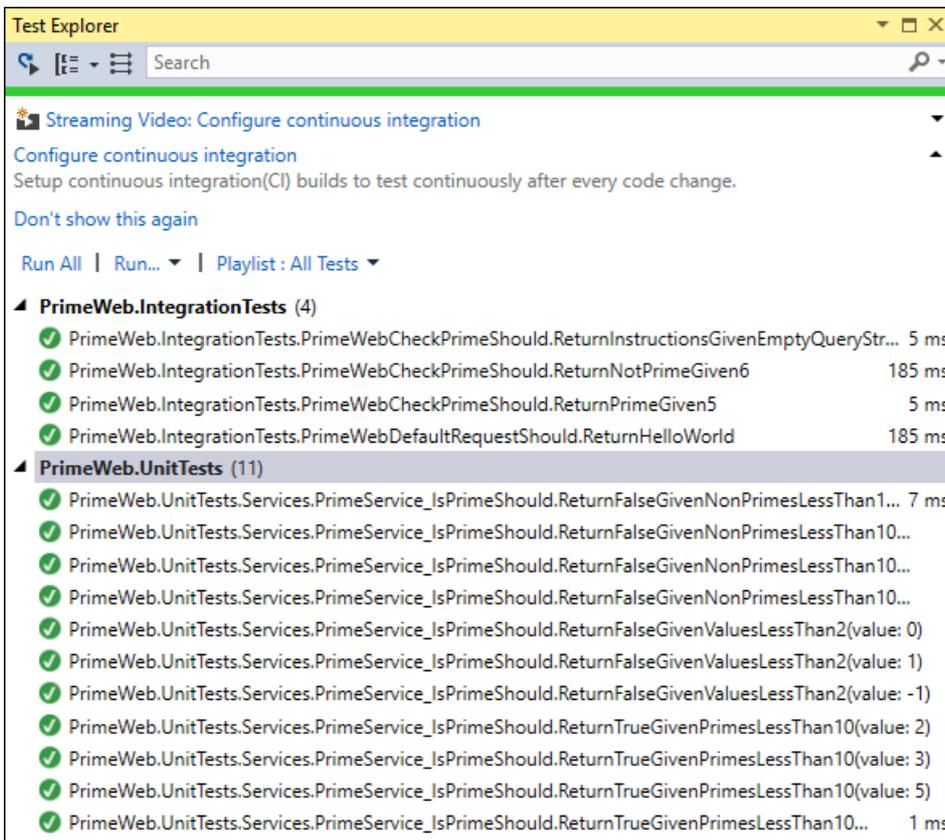
        // Assert
        Assert.Equal("5 is prime!",
            responseString);
    }

    [Fact]
    public async Task ReturnNotPrimeGiven6()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString("6");

        // Assert
        Assert.Equal("6 is NOT prime!",
            responseString);
    }
}

```

Note that you're not really trying to test the correctness of the prime number checker with these tests but rather that the web application is doing what you expect. You already have unit test coverage that gives you confidence in `PrimeService`, as you can see here:



You can learn more about the unit tests in the [Unit testing](#) article.

Integration testing Mvc/Razor

Test projects that contain Razor views require `<PreserveCompilationContext>` be set to true in the `.csproj` file:

```
<PreserveCompilationContext>true</PreserveCompilationContext>
```

Projects missing this element will generate an error similar to the following:

```
Microsoft.AspNetCore.Mvc.Razor.Compilation.CompilationFailedException: 'One or more compilation failures occurred:  
oebhccx.1bd(4,62): error CS0012: The type 'Attribute' is defined in an assembly that is not referenced. You must add a reference to assembly 'netstandard, Version=2.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51'.
```

Refactoring to use middleware

Refactoring is the process of changing an application's code to improve its design without changing its behavior. It should ideally be done when there is a suite of passing tests, since these help ensure the system's behavior remains the same before and after the changes. Looking at the way in which the prime checking logic is implemented in the web application's `Configure` method, you see:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
        if (context.Request.Path.Value.Contains("checkprime"))
        {
            int numberToCheck;
            try
            {
                numberToCheck = int.Parse(context.Request.QueryString.Value.Replace("?", ""));
                var primeService = new PrimeService();
                if (primeService.IsPrime(numberToCheck))
                {
                    await context.Response.WriteAsync($"{numberToCheck} is prime!");
                }
                else
                {
                    await context.Response.WriteAsync($"{numberToCheck} is NOT prime!");
                }
            }
            catch
            {
                await context.Response.WriteAsync("Pass in a number to check in the form /checkprime?5");
            }
        }
        else
        {
            await context.Response.WriteAsync("Hello World!");
        }
    });
}

```

This code works, but it's far from how you would like to implement this kind of functionality in an ASP.NET Core application, even as simple as this is. Imagine what the `Configure` method would look like if you needed to add this much code to it every time you add another URL endpoint!

One option to consider is adding [MVC](#) to the application and creating a controller to handle the prime checking. However, assuming you don't currently need any other MVC functionality, that's a bit overkill.

You can, however, take advantage of ASP.NET Core [middleware](#), which will help us encapsulate the prime checking logic in its own class and achieve better [separation of concerns](#) in the `Configure` method.

You want to allow the path the middleware uses to be specified as a parameter, so the middleware class expects a `RequestDelegate` and a `PrimeCheckerOptions` instance in its constructor. If the path of the request doesn't match what this middleware is configured to expect, you simply call the next middleware in the chain and do nothing further. The rest of the implementation code that was in `Configure` is now in the `Invoke` method.

NOTE

Since the middleware depends on the `PrimeService` service, you're also requesting an instance of this service with the constructor. The framework will provide this service via [Dependency Injection](#), assuming it has been configured, for example in `ConfigureServices`.

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using PrimeWeb.Services;

```

```

using System;
using System.Threading.Tasks;

namespace PrimeWeb.Middleware
{
    public class PrimeCheckerMiddleware
    {
        private readonly RequestDelegate _next;
        private readonly PrimeCheckerOptions _options;
        private readonly PrimeService _primeService;

        public PrimeCheckerMiddleware(RequestDelegate next,
            PrimeCheckerOptions options,
            PrimeService primeService)
        {
            if (next == null)
            {
                throw new ArgumentNullException(nameof(next));
            }
            if (options == null)
            {
                throw new ArgumentNullException(nameof(options));
            }
            if (primeService == null)
            {
                throw new ArgumentNullException(nameof(primeService));
            }

            _next = next;
            _options = options;
            _primeService = primeService;
        }

        public async Task Invoke(HttpContext context)
        {
            var request = context.Request;
            if (!request.Path.HasValue ||
                request.Path != _options.Path)
            {
                await _next.Invoke(context);
            }
            else
            {
                int numberToCheck;
                if (int.TryParse(request.QueryString.Value.Replace("?", ""), out numberToCheck))
                {
                    if (_primeService.IsPrime(numberToCheck))
                    {
                        await context.Response.WriteAsync($"{numberToCheck} is prime!");
                    }
                    else
                    {
                        await context.Response.WriteAsync($"{numberToCheck} is NOT prime!");
                    }
                }
                else
                {
                    await context.Response.WriteAsync($"Pass in a number to check in the form {_options.Path}?
5");
                }
            }
        }
    }
}

```

Since this middleware acts as an endpoint in the request delegate chain when its path matches, there is no call to `_next.Invoke` when this middleware handles the request.

With this middleware in place and some helpful extension methods created to make configuring it easier, the refactored `Configure` method looks like this:

```
IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UsePrimeChecker();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

Following this refactoring, you're confident that the web application still works as before, since your integration tests are all passing.

NOTE

It's a good idea to commit your changes to source control after you complete a refactoring and your tests pass. If you're practicing Test Driven Development, [consider adding Commit to your Red-Green-Refactor cycle](#).

Resources

- [Unit testing](#)
- [Middleware](#)
- [Testing controllers](#)

Razor Pages unit and integration testing in ASP.NET Core

1/3/2018 • 12 min to read • [Edit Online](#)

By [Luke Latham](#)

ASP.NET Core supports unit and integration testing of Razor Pages apps. Testing the data access layer (DAL), page models, and integrated page components helps ensure:

- Parts of a Razor Pages app work independently and together as a unit during app construction.
- Classes and methods have limited scopes of responsibility.
- Additional documentation exists on how the app should behave.
- Regressions, which are errors brought about by updates to the code, are found during automated building and deployment.

This topic assumes that you have a basic understanding of Razor Pages apps, unit testing, and integration testing. If you're unfamiliar with Razor Pages apps or testing concepts, see the following topics:

- [Introduction to Razor Pages](#)
- [Getting started with Razor Pages](#)
- [Unit testing C# in .NET Core using dotnet test and xUnit](#)
- [Integration testing](#)

[View or download sample code \(how to download\)](#)

The sample project is composed of two apps:

APP	PROJECT FOLDER	DESCRIPTION
Message app	<i>src/RazorPagesTestingSample</i>	Allows a user to add, delete one, delete all, and analyze messages.
Test app	<i>tests/RazorPagesTestingSample.Tests</i>	Used to test the message app. <ul style="list-style-type: none">• Unit tests: Data access layer (DAL), Index page model• Integration tests: Index page

The tests can be run using the built-in testing features of an IDE, such as [Visual Studio](#). If using [Visual Studio Code](#) or the command line, execute the following command at a command prompt in the *tests/RazorPagesTestingSample.Tests* folder:

```
dotnet test
```

Message app organization

The message app is a simple Razor Pages message system with the following characteristics:

- The Index page of the app (*Pages/Index.cshtml* and *Pages/Index.cshtml.cs*) provides a UI and page model methods to control the addition, deletion, and analysis of messages (average words per message).

- A message is described by the `Message` class (*Data/Message.cs*) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using [Entity Framework's in-memory database](#)†.
- The app contains a data access layer (DAL) in its database context class, `AppDbContext` (*Data/AppDbContext.cs*). The DAL methods are marked `virtual`, which allows mocking the methods for use in the tests.
- If the database is empty on app startup, the message store is initialized with three messages. These *seeded messages* are also used in testing.

†The EF topic, [Testing with InMemory](#), explains how to use an in-memory database for testing with MSTest. This topic uses the [xUnit](#) testing framework. Testing concepts and test implementations across different testing frameworks are similar but not identical.

Although the app doesn't use the [repository pattern](#) and isn't an effective example of the [Unit of Work \(UoW\) pattern](#), Razor Pages supports these patterns of development. For more information, see [Designing the infrastructure persistence layer](#), [Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application](#), and [Testing controller logic](#) (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the *tests/RazorPagesTestingSample.Tests* folder:

TEST APP FOLDER	DESCRIPTION
<i>IntegrationTests</i>	<ul style="list-style-type: none"> • <i>IndexPageTest.cs</i> contains the integration tests for the Index page. • <i>TestFixture.cs</i> creates the test host to test the message app.
<i>UnitTests</i>	<ul style="list-style-type: none"> • <i>DataAccessLayerTest.cs</i> contains the unit tests for the DAL. • <i>IndexPageTest.cs</i> contains the unit tests for the Index page model.
<i>Utilities</i>	<p><i>Utilities.cs</i> contains the:</p> <ul style="list-style-type: none"> • <code>TestingDbContextOptions</code> method used to create new database context options for each DAL unit test so that the database is reset to its baseline condition for each test. • <code>GetRequestContentAsync</code> method used to prepare the <code>HttpClient</code> and content for requests that are sent to the message app during integration testing.

The test framework is [xUnit](#). The object mocking framework is [Moq](#). Integration tests are conducted using the [ASP.NET Core Test Host](#).

Unit testing the data access layer (DAL)

The message app has a DAL with four methods contained in the `AppDbContext` class (*src/RazorPagesTestingSample/Data/AppDbContext.cs*). Each method has one or two unit tests in the test app.

DAL METHOD	FUNCTION
------------	----------

DAL METHOD	FUNCTION
<code>GetMessagesAsync</code>	Obtains a <code>List<Message></code> from the database sorted by the <code>Text</code> property.
<code>AddMessageAsync</code>	Adds a <code>Message</code> to the database.
<code>DeleteAllMessagesAsync</code>	Deletes all <code>Message</code> entries from the database.
<code>DeleteMessageAsync</code>	Deletes a single <code>Message</code> from the database by <code>Id</code> .

Unit tests of the DAL require `DbContextOptions` when creating a new `AppDbContext` for each test. One approach to creating the `DbContextOptions` for each test is to use a `DbContextOptionsBuilder`:

```
var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase("InMemoryDb");

using (var db = new AppDbContext(optionsBuilder.Options))
{
    // Use the db here in the unit test.
}
```

The problem with this approach is that each test receives the database in whatever state the previous test left it. This can be problematic when trying to write atomic unit tests that don't interfere with each other. To force the `AppDbContext` to use a new database context for each test, supply a `DbContextOptions` instance that's based on a new service provider. The test app shows how to do this using its `Utilities` class method `TestingDbContextOptions` (*tests/RazorPagesTestingSample.Tests/Utilities/Utilities.cs*):

```
public static DbContextOptions<AppDbContext> TestingDbContextOptions()
{
    // Create a new service provider to create a new in-memory database.
    var serviceProvider = new ServiceCollection()
        .AddEntityFrameworkInMemoryDatabase()
        .BuildServiceProvider();

    // Create a new options instance using an in-memory database and
    // IServiceProvider that the context should resolve all of its
    // services from.
    var builder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb")
        .UseInternalServiceProvider(serviceProvider);

    return builder.Options;
}
```

Using the `DbContextOptions` in the DAL unit tests allows each test to run atomically with a fresh database instance:

```
using (var db = new AppDbContext(Utilities.TestingDbContextOptions()))
{
    // Use the db here in the unit test.
}
```

Each test method in the `DataAccessLayerTest` class (*UnitTests/DataAccessLayerTest.cs*) follows a similar Arrange-Act-Assert pattern:

1. Arrange: The database is configured for the test and/or the expected outcome is defined.
2. Act: The test is executed.
3. Assert: Assertions are made to determine if the test result is a success.

For example, the `DeleteMessageAsync` method is responsible for removing a single message identified by its `Id` (`src/RazorPagesTestingSample/Data/AppDbContext.cs`):

```
public async virtual Task DeleteMessageAsync(int id)
{
    var message = await Messages.FindAsync(id);

    if (message != null)
    {
        Messages.Remove(message);
        await SaveChangesAsync();
    }
}
```

There are two tests for this method. One test checks that the method deletes a message when the message is present in the database. The other method tests that the database doesn't change if the message `Id` for deletion doesn't exist. The `DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound` method is shown below:

```
[Fact]
public async Task DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound()
{
    using (var db = new AppDbContext(Utilities.TestingDbContextOptions()))
    {
        // Arrange
        var seedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(seedMessages);
        await db.SaveChangesAsync();
        var recId = 1;
        var expectedMessages =
            seedMessages.Where(message => message.Id != recId).ToList();

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(x => x.Id),
            actualMessages.OrderBy(x => x.Id),
            new Utilities.MessageComparer());
    }
}
```

First, the method performs the Arrange step, where preparation for the Act step takes place. The seeding messages are obtained and held in `seedMessages`. The seeding messages are saved into the database. The message with an `Id` of `1` is set for deletion. When the `DeleteMessageAsync` method is executed, the expected messages should have all of the messages except for the one with an `Id` of `1`. The `expectedMessages` variable represents this expected outcome.

```
// Arrange
var seedMessages = AppDbContext.GetSeedingMessages();
await db.AddRangeAsync(seedMessages);
await db.SaveChangesAsync();
var recId = 1;
var expectedMessages =
    seedMessages.Where(message => message.Id != recId).ToList();
```

The method acts: The `DeleteMessageAsync` method is executed passing in the `recId` of `1`:

```
// Act
await db.DeleteMessageAsync(recId);
```

Finally, the method obtains the `Messages` from the context and compares it to the `expectedMessages` asserting that the two are equal:

```
// Assert
var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

In order to compare that the two `List<Message>` are the same:

- The messages are ordered by `Id`.
- Message pairs are compared on the `Text` property.

A similar test method, `DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound` checks the result of attempting to delete a message that doesn't exist. In this case, the expected messages in the database should be equal to the actual messages after the `DeleteMessageAsync` method is executed. There should be no change to the database's content:

```
[Fact]
public async Task DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound()
{
    using (var db = new AppDbContext(Utilities.TestingDbContextOptions()))
    {
        // Arrange
        var expectedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(expectedMessages);
        await db.SaveChangesAsync();
        var recId = 4;

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

Unit testing the page model methods

Another set of unit tests is responsible for testing page model methods. In the message app, the Index page

models are found in the `IndexModel` class in `src/RazorPagesTestingSample/Pages/Index.cshtml.cs`.

PAGE MODEL METHOD	FUNCTION
<code>OnGetAsync</code>	Obtains the messages from the DAL for the UI using the <code>GetMessagesAsync</code> method.
<code>OnPostAddMessageAsync</code>	If the <code>ModelState</code> is valid, calls <code>AddMessageAsync</code> to add a message to the database.
<code>OnPostDeleteAllMessagesAsync</code>	Calls <code>DeleteAllMessagesAsync</code> to delete all of the messages in the database.
<code>OnPostDeleteMessageAsync</code>	Executes <code>DeleteMessageAsync</code> to delete a message with the <code>Id</code> specified.
<code>OnPostAnalyzeMessagesAsync</code>	If one or more messages are in the database, calculates the average number of words per message.

The page model methods are tested using seven tests in the `IndexPageTest` class (`tests/RazorPagesTestingSample.Tests/UnitTests/IndexPageTest.cs`). The tests use the familiar Arrange-Assert-Act pattern. These tests focus on:

- Determining if the methods follow the correct behavior when the `ModelState` is invalid.
- Confirming the methods produce the correct `ActionResult`.
- Checking that property value assignments are made correctly.

This group of tests often mock the methods of the DAL to produce expected data for the Act step where a page model method is executed. For example, the `GetMessagesAsync` method of the `AppDbContext` is mocked to produce output. When a page model method executes this method, the mock returns the result. The data doesn't come from the database. This creates predictable, reliable test conditions for using the DAL in the page model tests.

The `OnGetAsync_PopulatesThePageModel_WithAListOfMessages` test shows how the `GetMessagesAsync` method is mocked for the page model:

```
var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
var expectedMessages = AppDbContext.GetSeedingMessages();
mockAppDbContext.Setup(
    db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
var pageModel = new IndexModel(mockAppDbContext.Object);
```

When the `OnGetAsync` method is executed in the Act step, it calls the page model's `GetMessagesAsync` method.

Unit test Act step (`tests/RazorPagesTestingSample.Tests/UnitTests/IndexPageTest.cs`):

```
// Act
await pageModel.OnGetAsync();
```

`IndexPage` page model's `OnGetAsync` method (`src/RazorPagesTestingSample/Pages/Index.cshtml.cs`):

```
public async Task OnGetAsync()
{
    Messages = await _db.GetMessagesAsync();
}
```

The `GetMessagesAsync` method in the DAL doesn't return the result for this method call. The mocked version of the method returns the result.

In the `Assert` step, the actual messages (`actualMessages`) are assigned from the `Messages` property of the page model. A type check is also performed when the messages are assigned. The expected and actual messages are compared by their `Text` properties. The test asserts that the two `List<Message>` instances contain the same messages.

```
// Assert
var actualMessages = Assert.IsAssignableFrom<List<Message>>(pageModel.Messages);
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

Other tests in this group create page model objects that include the `DefaultHttpContext`, the `ModelStateDictionary`, an `ActionContext` to establish the `PageContext`, a `ViewDataDictionary`, and a `PageContext`. These are useful in conducting tests. For example, the message app establishes a `ModelState` error with `AddModelError` to check that a valid `PageResult` is returned when `OnPostAddMessageAsync` is executed:

```
[Fact]
public async Task OnPostAddMessageAsync_ReturnsAPageResult_WhenModelStateIsInvalid()
{
    // Arrange
    var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb");
    var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
    var expectedMessages = AppDbContext.GetSeedingMessages();
    mockAppDbContext.Setup(db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
    var httpContext = new DefaultHttpContext();
    var modelState = new ModelStateDictionary();
    var actionContext = new ActionContext(httpContext, new RouteData(), new PageActionDescriptor(),
    modelState);
    var modelMetadataProvider = new EmptyModelMetadataProvider();
    var viewData = new ViewDataDictionary(modelMetadataProvider, modelState);
    var tempData = new TempDataDictionary(httpContext, Mock.Of<ITempDataProvider>());
    var pageContext = new PageContext(actionContext)
    {
        ViewData = viewData
    };
    var pageModel = new IndexModel(mockAppDbContext.Object)
    {
        PageContext = pageContext,
        TempData = tempData,
        Url = new UrlHelper(actionContext)
    };
    pageModel.ModelState.AddModelError("Message.Text", "The Text field is required.");

    // Act
    var result = await pageModel.OnPostAddMessageAsync();

    // Assert
    Assert.IsType<PageResult>(result);
}
```

Integration testing the app

The integration tests focus on testing that the app's components work together. Integration tests are conducted using the [ASP.NET Core Test Host](#). Full request-response lifecycle processing is tested. These tests assert that the page produces the correct status code and `Location` header, if set.

An integration testing example from the sample checks the result of requesting the Index page of the message app (*tests/RazorPagesTestingSample.Tests/IntegrationTests/IndexPageTest.cs*):

```
[Fact]
public async Task Request_ReturnsSuccess()
{
    // Act
    var response = await _client.GetAsync("/");

    // Assert
    response.EnsureSuccessStatusCode();
}
```

There's no Arrange step. The `GetAsync` method is called on the `HttpClient` to send a GET request to the endpoint. The test asserts that the result is a 200-OK status code.

Any POST request to the message app must satisfy the antiforgery check that's automatically made by the app's [data protection antiforgery system](#). In order to arrange for a test's POST request, the test app must:

1. Make a request for the page.
2. Parse the antiforgery cookie and request validation token from the response.
3. Make the POST request with the antiforgery cookie and request validation token in place.

The `Post_AddMessageHandler_ReturnsRedirectToRoot` test method:

- Prepares a message and the `HttpClient`.
- Makes a POST request to the app.
- Checks the response is a redirect back to the Index page.

`Post_AddMessageHandler_ReturnsRedirectToRoot` method

(*tests/RazorPagesTestingSample.Tests/IntegrationTests/IndexPageTest.cs*):

```
[Fact]
public async Task Post_AddMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var data = new Dictionary<string, string>()
    {
        { "Message.Text", "Test message to add." }
    };
    var content = await Utilities.GetRequestContentAsync(_client, "/", data);

    // Act
    var response = await _client.PostAsync("?handler=AddMessage", content);

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

The `GetRequestContentAsync` utility method manages preparing the client with the antiforgery cookie and request verification token. Note how the method receives an `IDictionary` that permits the calling test method to pass in data for the request to encode along with the request verification token

(*tests/RazorPagesTestingSample.Tests/Utilities/Utilities.cs*):

```

public static async Task<FormUrlEncodedContent> GetRequestContentAsync(
    HttpClient _client, string path, IDictionary<string, string> data)
{
    // Make a request for the resource.
    var getResponse = await _client.GetAsync(path);

    // Set the response's antiforgery cookie on the HttpClient.
    _client.DefaultRequestHeaders.Add("Cookie",
        getResponse.Headers.GetValues("Set-Cookie"));

    // Obtain the request verification token from the response.
    // Any <form> element in the response contains a token, and
    // they're all the same within a single response.
    //
    // This method uses Regex to parse the element and its value
    // from the response markup. A better approach in a production
    // app would be to use an HTML parser (for example,
    // HtmlAgilityPack: http://html-agility-pack.net/).
    var responseMarkup = await getResponse.Content.ReadAsStringAsync();
    var regExp_RequestVerificationToken = new Regex(
        "<input name=\"__RequestVerificationToken\" type=\"hidden\" value=\"(.*)\" \\/>",
        RegexOptions.Compiled);
    var matches = regExp_RequestVerificationToken.Matches(responseMarkup);
    // Group[1] represents the captured characters, represented
    // by (.*) in the Regex pattern string.
    var token = matches?.FirstOrDefault().Groups[1].Value;

    // Add the token to the form data for the request.
    data.Add("__RequestVerificationToken", token);

    return new FormUrlEncodedContent(data);
}

```

Integration tests can also pass bad data to the app to test the app's response behavior. The message app limits message length to 200 characters (*src/RazorPagesTestingSample/Data/Message.cs*):

```

public class Message
{
    public int Id { get; set; }

    [Required]
    [DataType(DataType.Text)]
    [StringLength(200, ErrorMessage = "There's a 200 character limit on messages. Please shorten your message.")]
    public string Text { get; set; }
}

```

The `Post_AddMessageHandler_ReturnsSuccess_WhenMessageTextTooLong` test `Message` explicitly passes in text with 201 "X" characters. This results in a `ModelState` error. The POST doesn't redirect back to the Index page. It returns a 200-OK with a `null` `Location` header (*tests/RazorPagesTestingSample.Tests/IntegrationTests/IndexPageTest.cs*):

```
[Fact]
public async Task Post_AddMessageHandler_ReturnsSuccess_WhenMessageTextTooLong()
{
    // Arrange
    var data = new Dictionary<string, string>()
    {
        { "Message.Text", new string('X', 201) }
    };
    var content = await Utilities.GetRequestContentAsync(_client, "/", data);

    // Act
    var response = await _client.PostAsync("?handler=AddMessage", content);

    // Assert
    // A ModelState failure returns to Page (200-OK) and doesn't redirect.
    response.EnsureSuccessStatusCode();
    Assert.Null(response.Headers.Location?.OriginalString);
}
```

See also

- [Unit testing C# in .NET Core using dotnet test and xUnit](#)
- [Integration testing](#)
- [Testing controllers](#)
- [Unit Test Your Code \(Visual Studio\)](#)
- [xUnit.net](#)
- [Getting started with xUnit.net \(.NET Core/ASP.NET Core\)](#)
- [Moq](#)
- [Moq Quickstart](#)

Testing controller logic in ASP.NET Core

10/13/2017 • 17 min to read • [Edit Online](#)

By [Steve Smith](#)

Controllers in ASP.NET MVC apps should be small and focused on user-interface concerns. Large controllers that deal with non-UI concerns are more difficult to test and maintain.

[View or download sample from GitHub](#)

Testing controllers

Controllers are a central part of any ASP.NET Core MVC application. As such, you should have confidence they behave as intended for your app. Automated tests can provide you with this confidence and can detect errors before they reach production. It's important to avoid placing unnecessary responsibilities within your controllers and ensure your tests focus only on controller responsibilities.

Controller logic should be minimal and not be focused on business logic or infrastructure concerns (for example, data access). Test controller logic, not the framework. Test how the controller *behaves* based on valid or invalid inputs. Test controller responses based on the result of the business operation it performs.

Typical controller responsibilities:

- Verify `ModelState.IsValid`.
- Return an error response if `ModelState` is invalid.
- Retrieve a business entity from persistence.
- Perform an action on the business entity.
- Save the business entity to persistence.
- Return an appropriate `ActionResult`.

Unit testing

[Unit testing](#) involves testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action is tested, not the behavior of its dependencies or of the framework itself. As you unit test your controller actions, make sure you focus only on its behavior. A controller unit test avoids things like [filters](#), [routing](#), or [model binding](#). By focusing on testing just one thing, unit tests are generally simple to write and quick to run. A well-written set of unit tests can be run frequently without much overhead. However, unit tests do not detect issues in the interaction between components, which is the purpose of [integration testing](#).

If you're writing custom filters, routes, etc, you should unit test them, but not as part of your tests on a particular controller action. They should be tested in isolation.

TIP

[Create and run unit tests with Visual Studio.](#)

To demonstrate unit testing, review the following controller. It displays a list of brainstorming sessions and allows new brainstorming sessions to be created with a POST:

```

using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class HomeController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public HomeController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index()
        {
            var sessionList = await _sessionRepository.ListAsync();

            var model = sessionList.Select(session => new StormSessionViewModel()
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
}

```

The controller is following the [explicit dependencies principle](#), expecting dependency injection to provide it with an instance of `IBrainstormSessionRepository`. This makes it fairly easy to test using a mock object framework, like [Moq](#). The `HTTP GET Index` method has no looping or branching and only calls one method. To test this `Index`

method, we need to verify that a `ViewResult` is returned, with a `ViewModel` from the repository's `List` method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class HomeControllerTests
    {
        [Fact]
        public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
        {
            // Arrange
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
            var controller = new HomeController(mockRepo.Object);

            // Act
            var result = await controller.Index();

            // Assert
            var viewResult = Assert.IsType<ViewResult>(result);
            var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
                viewResult.ViewData.Model);
            Assert.Equal(2, model.Count());
        }

        private List<BrainstormSession> GetTestSessions()
        {
            var sessions = new List<BrainstormSession>();
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 2),
                Id = 1,
                Name = "Test One"
            });
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 1),
                Id = 2,
                Name = "Test Two"
            });
            return sessions;
        }
    }
}
```

The `HomeController` `HTTP POST Index` method (shown above) should verify:

- The action method returns a Bad Request `ViewResult` with the appropriate data when `ModelState.IsValid` is `false`
- The `Add` method on the repository is called and a `RedirectToActionResult` is returned with the correct arguments when `ModelState.IsValid` is true.

Invalid model state can be tested by adding errors using `AddModelError` as shown in the first test below.

```

[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectActionResult.ControllerName);
    Assert.Equal("Index", redirectActionResult.ActionName);
    mockRepo.Verify();
}

```

The first test confirms when `ModelState` is not valid, the same `ViewResult` is returned as for a `GET` request. Note that the test doesn't attempt to pass in an invalid model. That wouldn't work anyway since model binding isn't running (though an [integration test](#) would use exercise model binding). In this case, model binding is not being tested. These unit tests are only testing what the code in the action method does.

The second test verifies that when `ModelState` is valid, a new `BrainstormSession` is added (via the repository), and the method returns a `RedirectToActionResult` with the expected properties. Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows it to be verified in the test. This is done with the call to `mockRepo.Verify`, which will fail the test if the expected method was not called.

NOTE

The Moq library used in this sample makes it easy to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about [customizing Mock behavior with Moq](#).

Another controller in the app displays information related to a particular brainstorming session. It includes some logic to deal with invalid id values:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class SessionController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public SessionController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index(int? id)
        {
            if (!id.HasValue)
            {
                return RedirectToAction(actionName: nameof(Index), controllerName: "Home");
            }

            var session = await _sessionRepository.GetByIdAsync(id.Value);
            if (session == null)
            {
                return Content("Session not found.");
            }

            var viewModel = new StormSessionViewModel()
            {
                DateCreated = session.DateCreated,
                Name = session.Name,
                Id = session.Id
            };

            return View(viewModel);
        }
    }
}

```

The controller action has three cases to test, one for each `return` statement:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class SessionControllerTests
    {
        [Fact]
        public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
        {
            // Arrange
            var controller = new SessionController(sessionRepository: null);

            // Act

```

```

        var result = await controller.Index(id: null);

        // Assert
        var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
        Assert.Equal("Home", redirectToActionResult.ControllerName);
        Assert.Equal("Index", redirectToActionResult.ActionName);
    }

    [Fact]
    public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId)
            .Returns(Task.FromResult((BrainstormSession)null));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var contentResult = Assert.IsType<ContentResult>(result);
        Assert.Equal("Session not found.", contentResult.Content);
    }

    [Fact]
    public async Task IndexReturnsViewResultWithStormSessionViewModel()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId)
            .Returns(Task.FromResult(GetTestSessions().FirstOrDefault(s => s.Id == testSessionId)));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var viewResult = Assert.IsType<ViewResult>(result);
        var model = Assert.IsType<StormSessionViewModel>(viewResult.ViewData.Model);
        Assert.Equal("Test One", model.Name);
        Assert.Equal(2, model.DateCreated.Day);
        Assert.Equal(testSessionId, model.Id);
    }

    private List<BrainstormSession> GetTestSessions()
    {
        var sessions = new List<BrainstormSession>();
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 2),
            Id = 1,
            Name = "Test One"
        });
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 1),
            Id = 2,
            Name = "Test Two"
        });
        return sessions;
    }
}
}
}

```

The app exposes functionality as a web API (a list of ideas associated with a brainstorming session and a method for adding new ideas to a session):

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;

namespace TestingControllersSample.Api
{
    [Route("api/ideas")]
    public class IdeasController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public IdeasController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        [HttpGet("forsession/{sessionId}")]
        public async Task<ActionResult> ForSession(int sessionId)
        {
            var session = await _sessionRepository.GetByIdAsync(sessionId);
            if (session == null)
            {
                return NotFound(sessionId);
            }

            var result = session.Ideas.Select(idea => new IdeaDTO()
            {
                Id = idea.Id,
                Name = idea.Name,
                Description = idea.Description,
                DateCreated = idea.DateCreated
            }).ToList();

            return Ok(result);
        }

        [HttpPost("create")]
        public async Task<ActionResult> Create([FromBody]NewIdeaModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            var session = await _sessionRepository.GetByIdAsync(model.SessionId);
            if (session == null)
            {
                return NotFound(model.SessionId);
            }

            var idea = new Idea()
            {
                DateCreated = DateTimeOffset.Now,
                Description = model.Description,
                Name = model.Name
            };
            session.AddIdea(idea);

            await _sessionRepository.UpdateAsync(session);

            return Ok(session);
        }
    }
}
```

```

        return UK(session);
    }
}
}

```

The `ForSession` method returns a list of `IdeaDTO` types. Avoid returning your business domain entities directly via API calls, since frequently they include more data than the API client requires, and they unnecessarily couple your app's internal domain model with the API you expose externally. Mapping between domain entities and the types you will return over the wire can be done manually (using a LINQ `Select` as shown here) or using a library like [AutoMapper](#)

The unit tests for the `Create` and `ForSession` API methods:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Api;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class ApiIdeasControllerTests
    {
        [Fact]
        public async Task Create_ReturnsBadRequest_GivenInvalidModel()
        {
            // Arrange & Act
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            var controller = new IdeasController(mockRepo.Object);
            controller.ModelState.AddModelError("error", "some error");

            // Act
            var result = await controller.Create(model: null);

            // Assert
            Assert.IsType<BadRequestObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
        {
            // Arrange
            int testSessionId = 123;
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
                .Returns(Task.FromResult((BrainstormSession)null));
            var controller = new IdeasController(mockRepo.Object);

            // Act
            var result = await controller.Create(new NewIdeaModel());

            // Assert
            Assert.IsType<NotFoundObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsNewlyCreatedIdeaForSession()
        {
            // Arrange
            int testSessionId = 123;

```

```

string testName = "test name";
string testDescription = "test description";
var testSession = GetTestSession();
var mockRepo = new Mock<IBrainstormSessionRepository>();
mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
    .Returns(Task.FromResult(testSession));
var controller = new IdeasController(mockRepo.Object);

var newIdea = new NewIdeaModel()
{
    Description = testDescription,
    Name = testName,
    SessionId = testSessionId
};
mockRepo.Setup(repo => repo.UpdateAsync(testSession))
    .Returns(Task.CompletedTask)
    .Verifiable();

// Act
var result = await controller.Create(newIdea);

// Assert
var okResult = Assert.IsType<OkObjectResult>(result);
var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
mockRepo.Verify();
Assert.Equal(2, returnSession.Ideas.Count());
Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}

private BrainstormSession GetTestSession()
{
    var session = new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    };

    var idea = new Idea() { Name = "One" };
    session.AddIdea(idea);
    return session;
}
}
}

```

As stated previously, to test the behavior of the method when `ModelState` is invalid, add a model error to the controller as part of the test. Don't try to test model validation or model binding in your unit tests - just test your action method's behavior when confronted with a particular `ModelState` value.

The second test depends on the repository returning null, so the mock repository is configured to return null. There's no need to create a test database (in memory or otherwise) and construct a query that will return this result - it can be done in a single statement as shown.

The last test verifies that the repository's `Update` method is called. As we did previously, the mock is called with `Verifiable` and then the mocked repository's `Verify` method is called to confirm the verifiable method was executed. It's not a unit test responsibility to ensure that the `Update` method saved the data; that can be done with an integration test.

Integration testing

[Integration testing](#) is done to ensure separate modules within your app work correctly together. Generally, anything you can test with a unit test, you can also test with an integration test, but the reverse isn't true.

However, integration tests tend to be much slower than unit tests. Thus, it's best to test whatever you can with unit tests, and use integration tests for scenarios that involve multiple collaborators.

Although they may still be useful, mock objects are rarely used in integration tests. In unit testing, mock objects are an effective way to control how collaborators outside of the unit being tested should behave for the purposes of the test. In an integration test, real collaborators are used to confirm the whole subsystem works together correctly.

Application state

One important consideration when performing integration testing is how to set your app's state. Tests need to run independent of one another, and so each test should start with the app in a known state. If your app doesn't use a database or have any persistence, this may not be an issue. However, most real-world apps persist their state to some kind of data store, so any modifications made by one test could impact another test unless the data store is reset. Using the built-in `TestServer`, it's very straightforward to host ASP.NET Core apps within our integration tests, but that doesn't necessarily grant access to the data it will use. If you're using an actual database, one approach is to have the app connect to a test database, which your tests can access and ensure is reset to a known state before each test executes.

In this sample application, I'm using Entity Framework Core's `InMemoryDatabase` support, so I can't just connect to it from my test project. Instead, I expose an `InitializeDatabase` method from the app's `Startup` class, which I call when the app starts up if it's in the `Development` environment. My integration tests automatically benefit from this as long as they set the environment to `Development`. I don't have to worry about resetting the database, since the `InMemoryDatabase` is reset each time the app restarts.

The `Startup` class:

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.Infrastructure;

namespace TestingControllersSample
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(
                optionsBuilder => optionsBuilder.UseInMemoryDatabase("InMemoryDb"));

            services.AddMvc();

            services.AddScoped<IBrainstormSessionRepository,
                EFStormSessionRepository>();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env,
            ILoggerFactory loggerFactory)
        {
            if (env.IsDevelopment())
            {
                var repository = app.ApplicationServices.GetService<IBrainstormSessionRepository>();
                InitializeDatabaseAsync(repository).Wait();
            }
        }
    }
}
```

```

        app.UseStaticFiles();

        app.UseMvcWithDefaultRoute();
    }

    public async Task InitializeDatabaseAsync(IBrainstormSessionRepository repo)
    {
        var sessionList = await repo.ListAsync();
        if (!sessionList.Any())
        {
            await repo.AddAsync(GetTestSession());
        }
    }

    public static BrainstormSession GetTestSession()
    {
        var session = new BrainstormSession()
        {
            Name = "Test Session 1",
            DateCreated = new DateTime(2016, 8, 1)
        };
        var idea = new Idea()
        {
            DateCreated = new DateTime(2016, 8, 1),
            Description = "Totally awesome idea",
            Name = "Awesome idea"
        };
        session.AddIdea(idea);
        return session;
    }
}
}
}

```

You'll see the `GetTestSession` method used frequently in the integration tests below.

Accessing views

Each integration test class configures the `TestServer` that will run the ASP.NET Core app. By default, `TestServer` hosts the web app in the folder where it's running - in this case, the test project folder. Thus, when you attempt to test controller actions that return `ViewResult`, you may see this error:

```

The view 'Index' was not found. The following locations were searched:
(list of locations)

```

To correct this issue, you need to configure the server's content root, so it can locate the views for the project being tested. This is done by a call to `UseContentRoot` in the `TestFixture` class, shown below:

```

using System;
using System.IO;
using System.Net.Http;
using System.Reflection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.TestHost;
using Microsoft.Extensions.DependencyInjection;

namespace TestingControllersSample.Tests.IntegrationTests
{
    /// <summary>
    /// A test fixture which hosts the target project (project we wish to test) in an in-memory server.
    /// </summary>
    /// <typeparam name="TStartup">Target project's startup type.</typeparam>

```

```

/// <typeparam name="TStartup" />target project's startup type/>typeparam
public class TestFixture<TStartup> : IDisposable
{
    private readonly TestServer _server;

    public TestFixture()
        : this(Path.Combine("src"))
    {
    }

    protected TestFixture(string relativeTargetProjectParentDir)
    {
        var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;
        var contentRoot = GetProjectPath(relativeTargetProjectParentDir, startupAssembly);

        var builder = new WebHostBuilder()
            .UseContentRoot(contentRoot)
            .ConfigureServices(InitializeServices)
            .UseEnvironment("Development")
            .UseStartup(typeof(TStartup));

        _server = new TestServer(builder);

        Client = _server.CreateClient();
        Client.BaseAddress = new Uri("http://localhost");
    }

    public HttpClient Client { get; }

    public void Dispose()
    {
        Client.Dispose();
        _server.Dispose();
    }

    protected virtual void InitializeServices(IServiceCollection services)
    {
        var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;

        // Inject a custom application part manager.
        // Overrides AddMvcCore() because it uses TryAdd().
        var manager = new ApplicationPartManager();
        manager.ApplicationParts.Add(new AssemblyPart(startupAssembly));
        manager.FeatureProviders.Add(new ControllerFeatureProvider());
        manager.FeatureProviders.Add(new ViewComponentFeatureProvider());

        services.AddSingleton(manager);
    }

    /// <summary>
    /// Gets the full path to the target project that we wish to test
    /// </summary>
    /// <param name="projectRelativePath">
    /// The parent directory of the target project.
    /// e.g. src, samples, test, or test/websites
    /// </param>
    /// <param name="startupAssembly">The target project's assembly.</param>
    /// <returns>The full path to the target project.</returns>
    private static string GetProjectPath(string projectRelativePath, Assembly startupAssembly)
    {
        // Get name of the target project which we want to test
        var projectName = startupAssembly.GetName().Name;

        // Get currently executing test project path
        var applicationBasePath = System.AppContext.BaseDirectory;

        // Find the path to the target project
        var directoryInfo = new DirectoryInfo(applicationBasePath);
        do
        {

```

```

        {
            directoryInfo = directoryInfo.Parent;

            var projectDirectoryInfo = new DirectoryInfo(Path.Combine(directoryInfo.FullName,
projectRelativePath));
            if (projectDirectoryInfo.Exists)
            {
                var projectFileInfo = new FileInfo(Path.Combine(projectDirectoryInfo.FullName,
projectName, $"{projectName}.csproj"));
                if (projectFileInfo.Exists)
                {
                    return Path.Combine(projectDirectoryInfo.FullName, projectName);
                }
            }
        }
        while (directoryInfo.Parent != null);

        throw new Exception($"Project root could not be located using the application root
{applicationBasePath}.");
    }
}
}

```

The `TestFixture` class is responsible for configuring and creating the `TestServer`, setting up an `HttpClient` to communicate with the `TestServer`. Each of the integration tests uses the `Client` property to connect to the test server and make a request.

```

using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class HomeControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        private readonly HttpClient _client;

        public HomeControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task ReturnsInitialListOfBrainstormSessions()
        {
            // Arrange - get a session known to exist
            var testSession = Startup.GetTestSession();

            // Act
            var response = await _client.GetAsync("/");

            // Assert
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains(testSession.Name, responseString);
        }

        [Fact]
        public async Task PostAddsNewBrainstormSession()
        {
            // Arrange
            string testSessionName = Guid.NewGuid().ToString();
            var data = new Dictionary<string, string>();
            data.Add("SessionName", testSessionName);
            var content = new FormUrlEncodedContent(data);

            // Act
            var response = await _client.PostAsync("/", content);

            // Assert
            Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
            Assert.Equal("/", response.Headers.Location.ToString());
        }
    }
}

```

In the first test above, the `responseString` holds the actual rendered HTML from the View, which can be inspected to confirm it contains expected results.

The second test constructs a form POST with a unique session name and POSTs it to the app, then verifies that the expected redirect is returned.

API methods

If your app exposes web APIs, it's a good idea to have automated tests confirm they execute as expected. The built-in `TestServer` makes it easy to test web APIs. If your API methods are using model binding, you should always check `ModelState.IsValid`, and integration tests are the right place to confirm that your model validation is working properly.

The following set of tests target the `Create` method in the `IdeasController` class shown above:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Newtonsoft.Json;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class ApiIdeasControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        internal class NewIdeaDto
        {
            public NewIdeaDto(string name, string description, int sessionId)
            {
                Name = name;
                Description = description;
                SessionId = sessionId;
            }

            public string Name { get; set; }
            public string Description { get; set; }
            public int SessionId { get; set; }
        }

        private readonly HttpClient _client;

        public ApiIdeasControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingNameValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("", "Description", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingDescriptionValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForSessionIdValueTooSmall()
        {
            // Arrange
```

```

// Arrange
var newIdea = new NewIdeaDto("Name", "Description", 0);

// Act
var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

// Assert
Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsBadRequestForSessionIdValueTooLarge()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 1000001);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsNotFoundForInvalidSession()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 123);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsCreatedIdeaWithCorrectInputs()
{
    // Arrange
    var testIdeaName = Guid.NewGuid().ToString();
    var newIdea = new NewIdeaDto(testIdeaName, "Description", 1);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    response.EnsureSuccessStatusCode();
    var returnedSession = await response.Content.ReadAsJsonAsync<BrainstormSession>();
    Assert.Equal(2, returnedSession.Ideas.Count);
    Assert.Contains(testIdeaName, returnedSession.Ideas.Select(i => i.Name).ToList());
}

[Fact]
public async Task ForSessionReturnsNotFoundForBadSessionId()
{
    // Arrange & Act
    var response = await _client.GetAsync("/api/ideas/forsession/500");

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task ForSessionReturnsIdeasForValidSessionId()
{
    // Arrange
    var testSession = Startup.GetTestSession();

    // Act

```

```
// Act
var response = await _client.GetAsync("/api/ideas/forsession/1");

// Assert
response.EnsureSuccessStatusCode();
var ideaList = JsonConvert.DeserializeObject<List<IdeaDTO>>(
    await response.Content.ReadAsStringAsync());
var firstIdea = ideaList.First();
Assert.Equal(testSession.Ideas.First().Name, firstIdea.Name);
    }
}
}
```

Unlike integration tests of actions that returns HTML views, web API methods that return results can usually be deserialized as strongly typed objects, as the last test above shows. In this case, the test deserializes the result to a `BrainstormSession` instance, and confirms that the idea was correctly added to its collection of ideas.

You'll find additional examples of integration tests in this article's [sample project](#).

Working with Data in ASP.NET Core

1/10/2018 • 1 min to read • [Edit Online](#)

- [Get started with Razor Pages and Entity Framework Core using Visual Studio](#)
 - [Get started with Razor Pages and EF](#)
 - [Create, Read, Update, and Delete operations](#)
 - [Sort, filter, page, and group](#)
 - [Migrations](#)
 - [Create a complex data model](#)
 - [Read related data](#)
 - [Update related data](#)
 - [Handle concurrency conflicts](#)
- [Get started with ASP.NET Core MVC and Entity Framework Core using Visual Studio](#)
 - [Get started](#)
 - [Create, Read, Update, and Delete operations](#)
 - [Sort, filter, page, and group](#)
 - [Migrations](#)
 - [Create a complex data model](#)
 - [Read related data](#)
 - [Update related data](#)
 - [Handle concurrency conflicts](#)
 - [Inheritance](#)
 - [Advanced topics](#)
- [ASP.NET Core with EF Core - new database](#) (Entity Framework Core documentation site)
- [ASP.NET Core with EF Core - existing database](#) (Entity Framework Core documentation site)
- [Get started with ASP.NET Core and Entity Framework 6](#)
- [Azure Storage](#)
 - [Add Azure Storage by using Visual Studio Connected Services](#)
 - [Get started with Azure Blob storage and Visual Studio Connected Services](#)
 - [Get started with Queue Storage and Visual Studio Connected Services](#)
 - [Get started with Azure Table Storage and Visual Studio Connected Services](#)

Getting started with Razor Pages and Entity Framework Core using Visual Studio (1 of 8)

1/8/2018 • 17 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web app demonstrates how to create ASP.NET Core 2.0 MVC web applications using Entity Framework (EF) Core 2.0 and Visual Studio 2017.

The sample app is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This page is the first in a series of tutorials that explain how to build the Contoso University sample app.

[Download or view the completed app.](#) [Download instructions.](#)

Prerequisites

Install the following:

- [.NET Core 2.0.0 SDK](#) or later.
- [Visual Studio 2017](#) version 15.3 or later with the **ASP.NET and web development** workload.

Familiarity with [Razor Pages](#). New programmers should complete [Get started with Razor Pages](#) before starting this series.

Troubleshooting

If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the [completed stage](#) or [completed project](#). For a list of common errors and how to solve them, see [the Troubleshooting section of the last tutorial in the series](#). If you don't find what you need there, you can post a question to StackOverflow.com for [ASP.NET Core](#) or [EF Core](#).

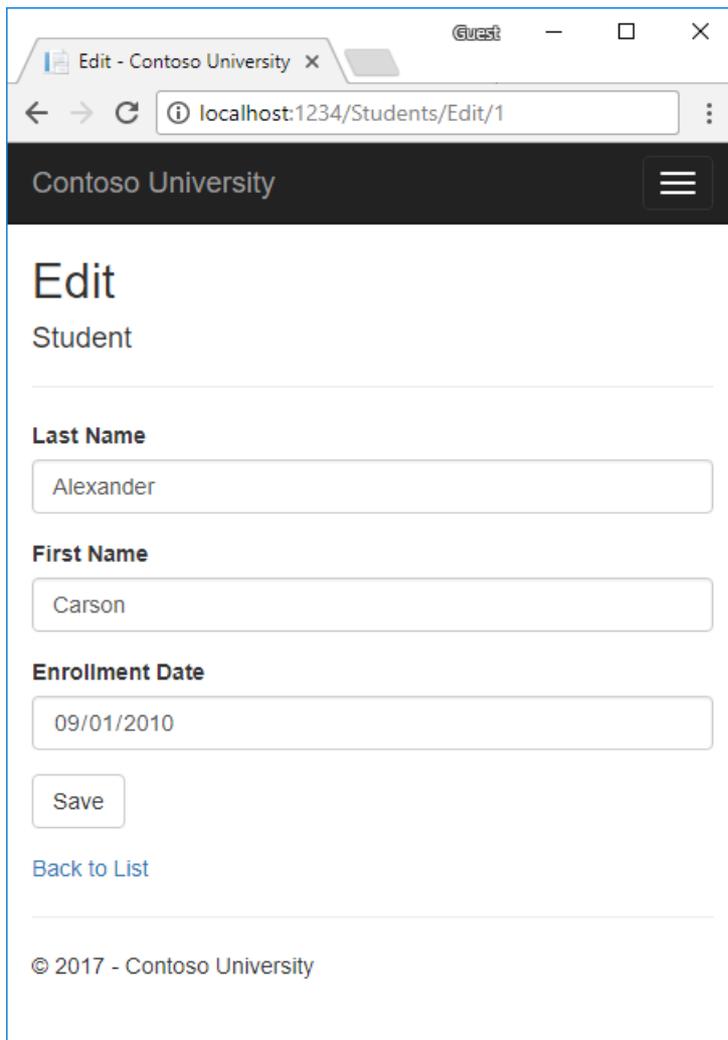
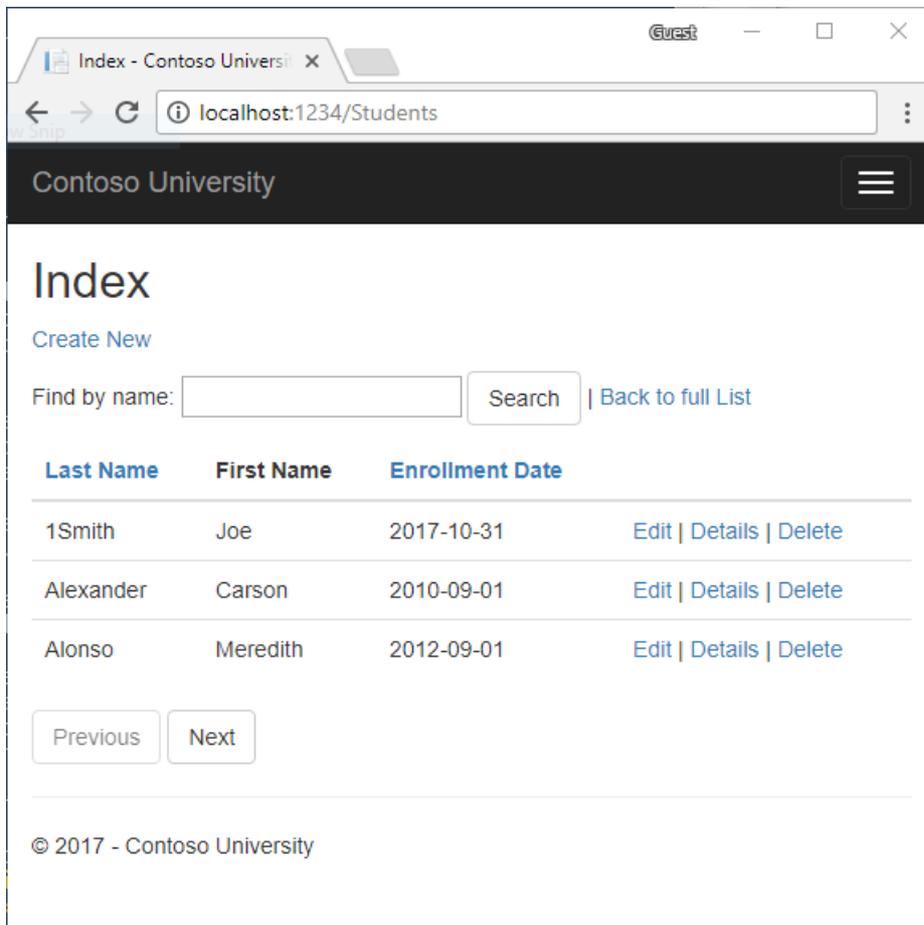
TIP

This series of tutorials builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. If you run into problems, you can start over from the previous tutorial instead of going back to the beginning. Alternatively, you can download a [completed stage](#) and start over using the completed stage.

The Contoso University web app

The app built in these tutorials is a basic university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.

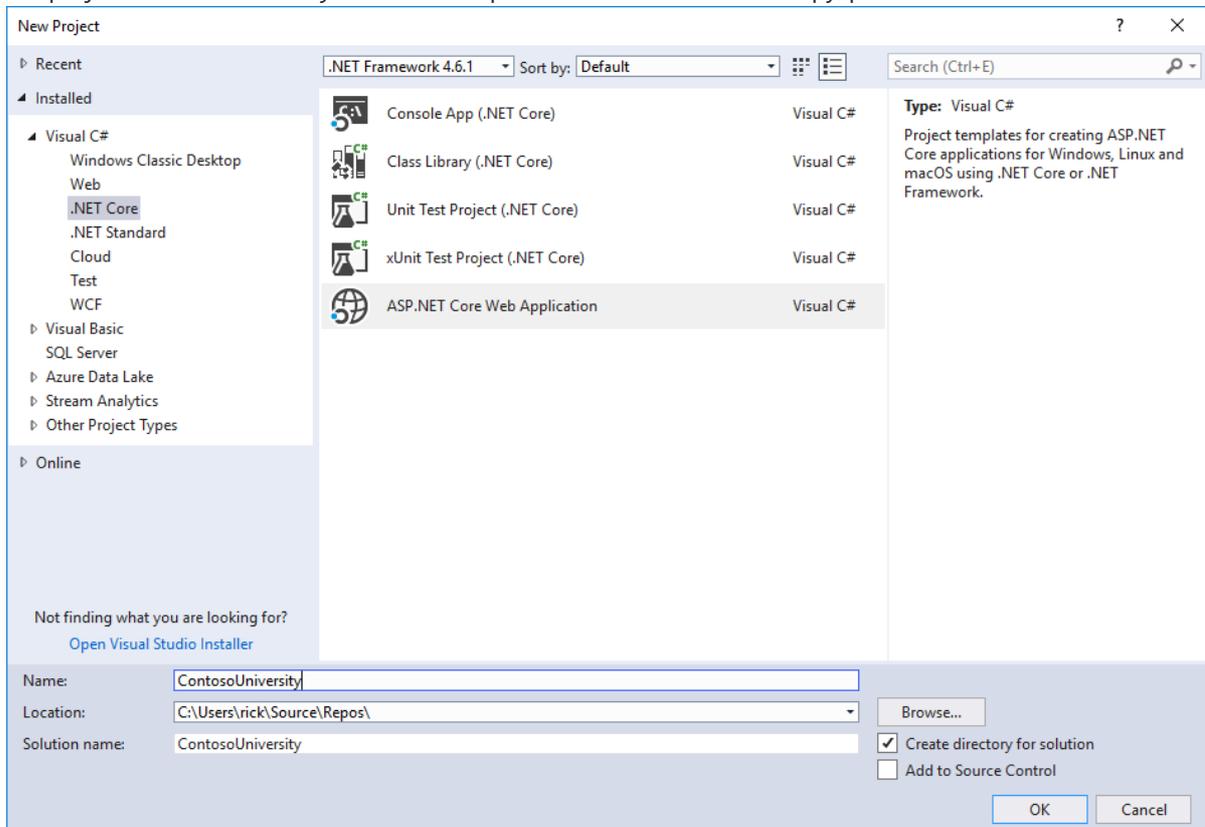


The UI style of this site is close to what's generated by the built-in templates. The tutorial focus is on EF Core

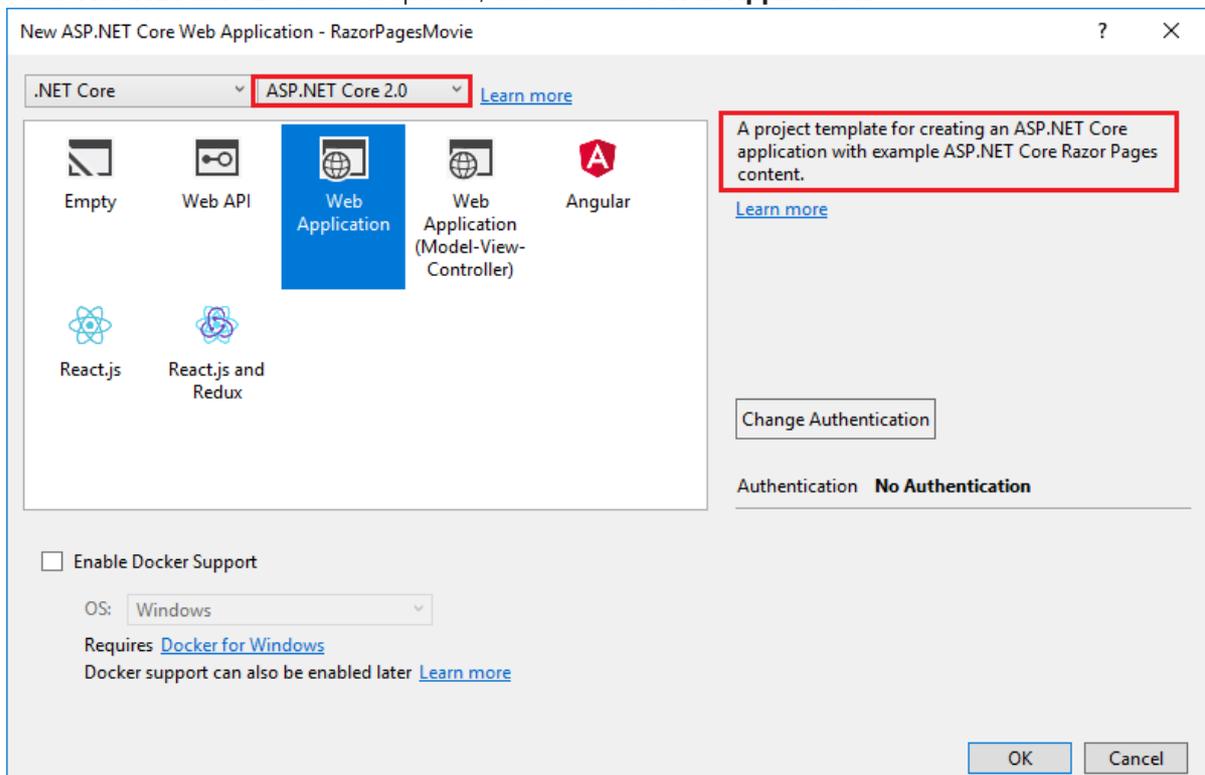
with Razor Pages, not the UI.

Create a Razor Pages web app

- From the Visual Studio **File** menu, select **New > Project**.
- Create a new ASP.NET Core Web Application. Name the project **ContosoUniversity**. It's important to name the project *ContosoUniversity* so the namespaces match when code is copy/pasted.



- Select **ASP.NET Core 2.0** in the dropdown, and then select **Web Application**.



Press **F5** to run the app in debug mode or **Ctrl-F5** to run without attaching the debugger

Set up the site style

A few changes set up the site menu, layout, and home page.

Open *Pages/_Layout.cshtml* and make the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University." There are three occurrences.
- Add menu entries for **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Contact** menu entry.

The changes are highlighted. (All the markup is *not* displayed.)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - Contoso University</title>

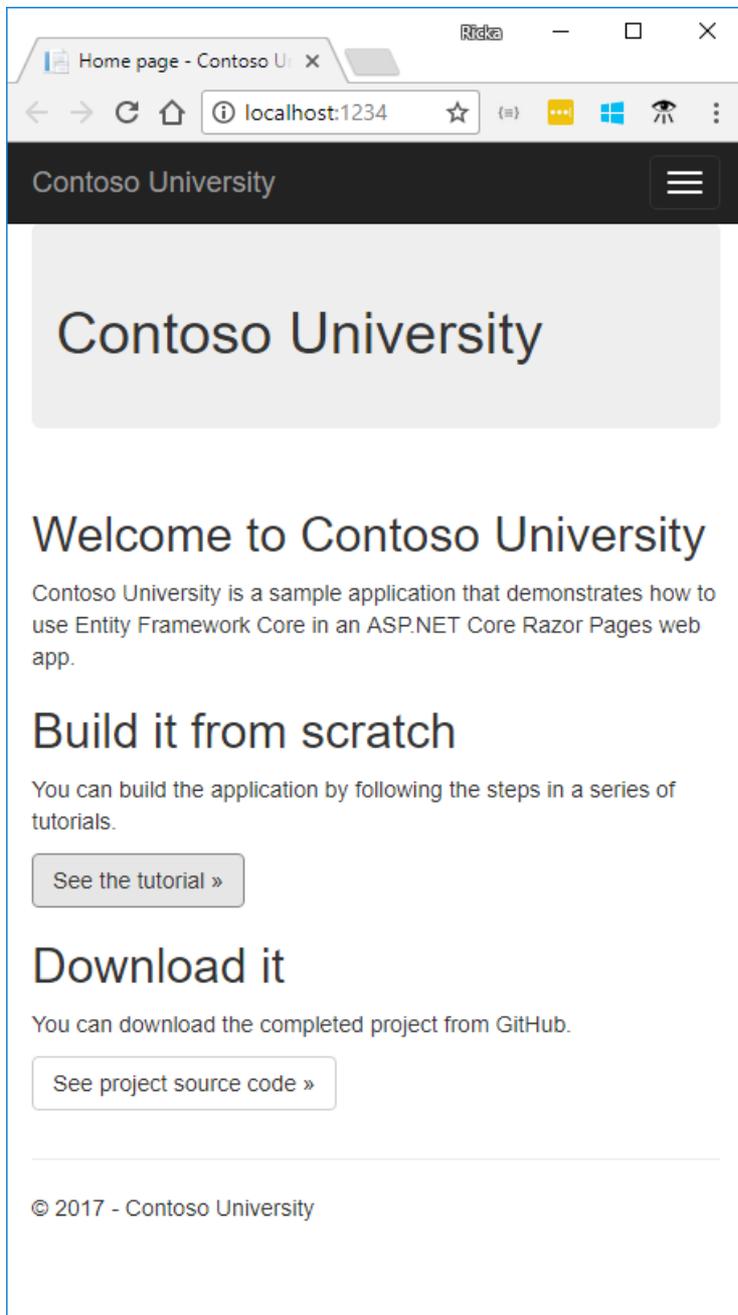
  <environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment exclude="Development">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>
</head>
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-page="/Index" class="navbar-brand">Contoso University</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-page="/Index">Home</a></li>
          <li><a asp-page="/About">About</a></li>
          <li><a asp-page="/Students/Index">Students</a></li>
          <li><a asp-page="/Courses/Index">Courses</a></li>
          <li><a asp-page="/Instructors/Index">Instructors</a></li>
          <li><a asp-page="/Departments/Index">Departments</a></li>
        </ul>
      </div>
    </div>
  </nav>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; 2017 - Contoso University</p>
    </footer>
  </div>
```

In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this app:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

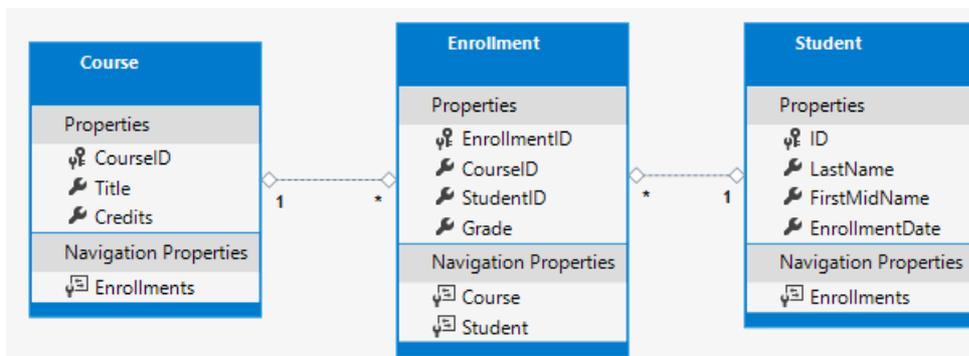
<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core Razor Pages web app.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default"
            href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro">
            See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
            href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-rp/intro/samples/cu-
            final">
            See project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project. The home page is displayed with tabs created in the following tutorials:



Create the data model

Create entity classes for the Contoso University app. Start with the following three entities:



There's a one-to-many relationship between `Student` and `Enrollment` entities. There's a one-to-many relationship between `Course` and `Enrollment` entities. A student can enroll in any number of courses. A course can have any number of students enrolled in it.

In the following sections, a class for each one of these entities is created.

The Student entity

Student
Properties
PK ID
LastName
FirstMidName
EnrollmentDate
Navigation Properties
Enrollments

Create a *Models* folder. In the *Models* folder, create a class file named *Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property becomes the primary key column of the database (DB) table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key.

The `Enrollments` property is a navigation property. Navigation properties link to other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that `Student`. For example, if a `Student` row in the DB has two related `Enrollment` rows, the `Enrollments` navigation property contains those two `Enrollment` entities. A related `Enrollment` row is a row that contains that student's primary key value in the `StudentID` column. For example, suppose the student with `ID=1` has two rows in the `Enrollment` table. The `Enrollment` table has two rows with `StudentID = 1`. `StudentID` is a foreign key in the `Enrollment` table that specifies the student in the `Student` table.

If a navigation property can hold multiple entities, the navigation property must be a list type, such as `ICollection<T>`. `ICollection<T>` can be specified, or a type such as `List<T>` or `HashSet<T>`. When `ICollection<T>` is used, EF Core creates a `HashSet<T>` collection by default. Navigation properties that hold multiple entities come from many-to-many and one-to-many relationships.

The Enrollment entity

Enrollment
Properties
PK EnrollmentID
CourseID
StudentID
Grade
Navigation Properties
Course
Student

In the *Models* folder, create *Enrollment.cs* with the following code:

```

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}

```

The `EnrollmentID` property is the primary key. This entity uses the `classnameID` pattern instead of `ID` like the `student` entity. Typically developers choose one pattern and use it throughout the data model. In a later tutorial, using `ID` without `classname` is shown to make it easier to implement inheritance in the data model.

The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity. The `Student` entity differs from the `Student.Enrollments` navigation property, which contains multiple `Enrollment` entities.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity

Course	
Properties	
🔑	CourseID
🔗	Title
🔗	Credits
Navigation Properties	
🔗	Enrollments

In the `Models` folder, create `Course.cs` with the following code:

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the DB generate it.

Create the SchoolContext DB context

The main class that coordinates EF Core functionality for a given data model is the DB context class. The data context is derived from `Microsoft.EntityFrameworkCore.DbContext`. The data context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named *Data*.

In the *Data* folder create *SchoolContext.cs* with the following code:

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}

```

This code creates a `DbSet` property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a DB table.
- An entity corresponds to a row in the table.

`DbSet<Enrollment>` and `DbSet<Course>` can be omitted. EF Core includes them implicitly because the `Student` entity references the `Enrollment` entity, and the `Enrollment` entity references the `Course` entity. For this tutorial, keep `DbSet<Enrollment>` and `DbSet<Course>` in the `SchoolContext`.

When the DB is created, EF Core creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural (Students rather than Student). Developers disagree about

whether table names should be plural. For these tutorials, the default behavior is overridden by specifying singular table names in the DbContext. To specify singular table names, add the following highlighted code:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Register the context with dependency injection

ASP.NET Core includes [dependency injection](#). Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a db context instance is shown later in the tutorial.

To register `SchoolContext` as a service, open `Startup.cs`, and add the highlighted lines to the `ConfigureServices` method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the [ASP.NET Core configuration system](#) reads the connection string from the `appsettings.json` file.

Add `using` statements for `ContosoUniversity.Data` and `Microsoft.EntityFrameworkCore` namespaces. Build the project.

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

Open the `appsettings.json` file and add a connection string as shown in the following code:

```

{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\mssqllocaldb;Database=ContosoUniversity1;ConnectRetryCount=0;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}

```

The preceding connection string uses `ConnectRetryCount=0` to prevent [SQLClient](#) from hanging.

SQL Server Express LocalDB

The connection string specifies a SQL Server LocalDB DB. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB creates *.mdf* DB files in the `C:/Users/<user>` directory.

Add code to initialize the DB with test data

EF Core creates an empty DB. In this section, a *Seed* method is written to populate it with test data.

In the *Data* folder, create a new class file named *DbInitializer.cs* and add the following code:

```

using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return; // DB has been seeded
            }

            var students = new Student[]
            {
                new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
            }

```



```

// Unused usings removed
using System;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred while seeding the database.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

The first time the app is run, the DB is created and seeded with test data. When the data model is updated:

- Delete the DB.
- Update the seed method.
- Run the app and a new seeded DB is created.

In later tutorials, the DB is updated when the data model changes, without deleting and re-creating the DB.

Add scaffold tooling

In this section, the Package Manager Console (PMC) is used to add the Visual Studio web code generation package. This package is required to run the scaffolding engine.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.

In the Package Manager Console (PMC), enter the following commands:

```

Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design
Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Utils

```

The previous command adds the NuGet packages to the *.csproj file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Utils" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>
```

Scaffold the model

- Open a command window in the project directory (The directory that contains the *Program.cs*, *Startup.cs*, and *.csproj* files).
- Run the following commands:

```
dotnet restore
dotnet aspnet-codegenerator razorpage -m Student -dc SchoolContext -udl -outDir Pages\Students --
referenceScriptLibraries
```

If the following error is generated:

```
Unhandled Exception: System.IO.FileNotFoundException:
Could not load file or assembly
'Microsoft.VisualStudio.Web.CodeGeneration.Utils,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60'.
The system cannot find the file specified.
```

Run the command again and leave a comment at the bottom of the page.

If you get the error:

```
No executable found matching command "dotnet-aspnet-codegenerator"
```

Open a command window in the project directory (The directory that contains the *Program.cs*, *Startup.cs*, and *.csproj* files).

Build the project. The build generates errors like the following:

```
1>Pages\Students\Index.cshhtml.cs(26,38,26,45): error CS1061: 'SchoolContext' does not contain a definition
for 'Student'
```

Globally change `_context.Student` to `_context.Students` (that is, add an "s" to `Student`). 7 occurrences are found and updated. We hope to fix [this bug](#) in the next release.

The following table details the ASP.NET Core code generators` parameters:

PARAMETER	DESCRIPTION
-m	The name of the model.
-dc	The data context.

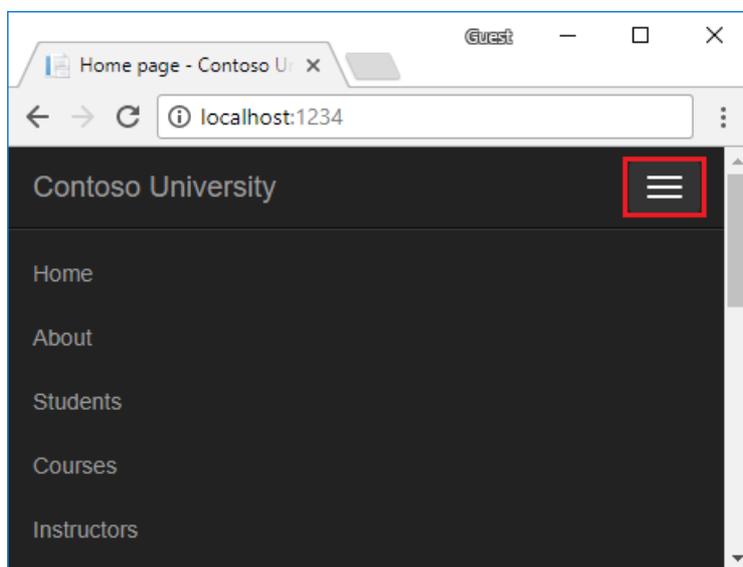
PARAMETER	DESCRIPTION
-udl	Use the default layout.
-outDir	The relative output folder path to create the views.
--referenceScriptLibraries	Adds <code>_ValidationScriptsPartial</code> to Edit and Create pages

Use the `-h` switch to get help on the `aspnet-codegenerator razorpage` command:

```
dotnet aspnet-codegenerator razorpage -h
```

Test the app

Run the app and select the **Students** link. Depending on the browser width, the **Students** link appears at the top of the page. If the **Students** link is not visible, click the navigation icon in the upper right corner.



Test the **Create**, **Edit**, and **Details** links.

View the DB

When the app is started, `DbInitializer.Initialize` calls `EnsureCreated`. `EnsureCreated` detects if the DB exists, and creates one if necessary. If there are no Students in the DB, the `Initialize` method adds students.

Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio. In SSOX, click **(localdb)\MSSQLLocalDB > Databases > ContosoUniversity1**.

Expand the **Tables** node.

Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.

The `.mdf` and `.ldf` DB files are in the `C:\Users\` folder.

`EnsureCreated` is called on app start, which allows the following work flow:

- Delete the DB.
- Change the DB schema (for example, add an `EmailAddress` field).
- Run the app.

`EnsureCreated` creates a DB with the `EmailAddress` column.

Conventions

The amount of code written in order for EF Core to create a complete DB is minimal because of the use of conventions, or assumptions that EF Core makes.

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or `classNameID` are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can be named (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

Conventional behavior can be overridden. For example, the table names can be explicitly specified, as shown earlier in this tutorial. The column names can be explicitly set. Primary keys and foreign keys can be explicitly set.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the `async` keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Student = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
 - Generate callbacks for parts of the method body.
 - Automatically create the `Task` object that is returned. For more information, see [Task Return Type](#).
- The implicit return type `Task` represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the DB are executed asynchronously. That includes, `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It does not include statements that just change an `IQueryable`, such as

```
var students = context.Students.Where(s => s.LastName == "Davolio");
```
- An EF Core context is not threaded safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the DB.

For more information about asynchronous programming in .NET, see [Async Overview](#).

In the next tutorial, basic CRUD (create, read, update, delete) operations are examined.

NEXT

Getting started with ASP.NET Core MVC and Entity Framework Core using Visual Studio

12/13/2017 • 1 min to read • [Edit Online](#)

Note: A Razor Pages version of this tutorial is available [here](#). The Razor Pages version is easier to follow and covers more EF features.

This series of tutorials teaches you how to create ASP.NET Core MVC web applications that use Entity Framework Core for data access. The tutorials require Visual Studio 2017.

1. [Getting started](#)
2. [Create, Read, Update, and Delete operations](#)
3. [Sorting, filtering, paging, and grouping](#)
4. [Migrations](#)
5. [Creating a complex data model](#)
6. [Reading related data](#)
7. [Updating related data](#)
8. [Handling concurrency conflicts](#)
9. [Inheritance](#)
10. [Advanced topics](#)

Getting started with ASP.NET Core and Entity Framework 6

9/22/2017 • 3 min to read • [Edit Online](#)

By [Paweł Grudzień](#), [Damien Pontifex](#), and [Tom Dykstra](#)

This article shows how to use Entity Framework 6 in an ASP.NET Core application.

Overview

To use Entity Framework 6, your project has to compile against .NET Framework, as Entity Framework 6 does not support .NET Core. If you need cross-platform features you will need to upgrade to [Entity Framework Core](#).

The recommended way to use Entity Framework 6 in an ASP.NET Core application is to put the EF6 context and model classes in a class library project that targets the full framework. Add a reference to the class library from the ASP.NET Core project. See the sample [Visual Studio solution with EF6 and ASP.NET Core projects](#).

You can't put an EF6 context in an ASP.NET Core project because .NET Core projects don't support all of the functionality that EF6 commands such as *Enable-Migrations* require.

Regardless of project type in which you locate your EF6 context, only EF6 command-line tools work with an EF6 context. For example, `Scaffold-DbContext` is only available in Entity Framework Core. If you need to do reverse engineering of a database into an EF6 model, see [Code First to an Existing Database](#).

Reference full framework and EF6 in the ASP.NET Core project

Your ASP.NET Core project needs to reference .NET framework and EF6. For example, the `.csproj` file of your ASP.NET Core project will look similar to the following example (only relevant parts of the file are shown).

```
<PropertyGroup>
  <TargetFramework>net452</TargetFramework>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>MVCCore</AssemblyName>
  <OutputType>Exe</OutputType>
  <PackageId>MVCCore</PackageId>
</PropertyGroup>
```

If you're creating a new project, use the **ASP.NET Core Web Application (.NET Framework)** template.

Handle connection strings

The EF6 command-line tools that you'll use in the EF6 class library project require a default constructor so they can instantiate the context. But you'll probably want to specify the connection string to use in the ASP.NET Core project, in which case your context constructor must have a parameter that lets you pass in the connection string. Here's an example.

```
public class SchoolContext : DbContext
{
    public SchoolContext(string connString) : base(connString)
    {
    }
}
```

Since your EF6 context doesn't have a parameterless constructor, your EF6 project has to provide an implementation of `IDbContextFactory`. The EF6 command-line tools will find and use that implementation so they can instantiate the context. Here's an example.

```
public class SchoolContextFactory : IDbContextFactory<SchoolContext>
{
    public SchoolContext Create()
    {
        return new EF6.SchoolContext("Server=
(localdb)\\mssqllocaldb;Database=EF6MVCCore;Trusted_Connection=True;MultipleActiveResultSets=true");
    }
}
```

In this sample code, the `IDbContextFactory` implementation passes in a hard-coded connection string. This is the connection string that the command-line tools will use. You'll want to implement a strategy to ensure that the class library uses the same connection string that the calling application uses. For example, you could get the value from an environment variable in both projects.

Set up dependency injection in the ASP.NET Core project

In the Core project's `Startup.cs` file, set up the EF6 context for dependency injection (DI) in `ConfigureServices`. EF context objects should be scoped for a per-request lifetime.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
    services.AddScoped<SchoolContext>(_ => new
SchoolContext(Configuration.GetConnectionString("DefaultConnection")));
}
```

You can then get an instance of the context in your controllers by using DI. The code is similar to what you'd write for an EF Core context:

```
public class StudentsController : Controller
{
    private readonly SchoolContext _context;

    public StudentsController(SchoolContext context)
    {
        _context = context;
    }
}
```

Sample application

For a working sample application, see the [sample Visual Studio solution](#) that accompanies this article.

This sample can be created from scratch by the following steps in Visual Studio:

- Create a solution.
- **Add New Project > Web > ASP.NET Core Web Application (.NET Framework)**
- **Add New Project > Windows Classic Desktop > Class Library (.NET Framework)**
- In **Package Manager Console** (PMC) for both projects, run the command `Install-Package Entityframework`.

- In the class library project, create data model classes and a context class, and an implementation of `IDbContextFactory`.
- In PMC for the class library project, run the commands `Enable-Migrations` and `Add-Migration Initial`. If you have set the ASP.NET Core project as the startup project, add `-StartupProjectName EF6` to these commands.
- In the Core project, add a project reference to the class library project.
- In the Core project, in `Startup.cs`, register the context for DI.
- In the Core project, in `appsettings.json`, add the connection string.
- In the Core project, add a controller and view(s) to verify that you can read and write data. (Note that ASP.NET Core MVC scaffolding won't work with the EF6 context referenced from the class library.)

Summary

This article has provided basic guidance for using Entity Framework 6 in an ASP.NET Core application.

Additional Resources

- [Entity Framework - Code-Based Configuration](#)

Azure Storage in ASP.NET Core

9/12/2017 • 1 min to read • [Edit Online](#)

- [Adding Azure Storage by using Visual Studio Connected Services](#)
- [Get Started with Blob storage and Visual Studio Connected Services](#)
- [Get Started with Queue Storage and Visual Studio Connected Services](#)
- [Get Started with Table Storage and Visual Studio Connected Services](#)

Client-side development in ASP.NET Core

1/10/2018 • 1 min to read • [Edit Online](#)

- [Use Gulp](#)
- [Use Grunt](#)
- [Manage client-side packages with Bower](#)
- [Build responsive sites with Bootstrap](#)
- [Style apps with LESS, Sass, and Font Awesome](#)
- [Bundle and minify](#)
- [TypeScript](#)
- [Use Browser Link](#)
- [Use JavaScriptServices for SPAs](#)
- [Use the SPA project templates \(RC\)](#)
 - [Angular project template](#)
 - [React project template](#)
 - [React with Redux project template](#)

Introduction to using Gulp in ASP.NET Core

11/10/2017 • 9 min to read • [Edit Online](#)

By [Erik Reitan](#), [Scott Addie](#), [Daniel Roth](#), and [Shayne Boyer](#)

In a typical modern web app, the build process might:

- Bundle and minify JavaScript and CSS files.
- Run tools to call the bundling and minification tasks before each build.
- Compile LESS or SASS files to CSS.
- Compile CoffeeScript or TypeScript files to JavaScript.

A *task runner* is a tool which automates these routine development tasks and more. Visual Studio provides built-in support for two popular JavaScript-based task runners: [Gulp](#) and [Grunt](#).

Gulp

Gulp is a JavaScript-based streaming build toolkit for client-side code. It is commonly used to stream client-side files through a series of processes when a specific event is triggered in a build environment. For instance, Gulp can be used to automate [bundling and minification](#) or the cleansing of a development environment before a new build.

A set of Gulp tasks is defined in *gulpfile.js*. The following JavaScript includes Gulp modules and specifies file paths to be referenced within the forthcoming tasks:

```
/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var paths = {
  webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";
```

The above code specifies which Node modules are required. The `require` function imports each module so that the dependent tasks can utilize their features. Each of the imported modules is assigned to a variable. The modules can be located either by name or path. In this example, the modules named `gulp`, `rimraf`, `gulp-concat`, `gulp-cssmin`, and `gulp-uglify` are retrieved by name. Additionally, a series of paths are created so that the locations of CSS and JavaScript files can be reused and referenced within the tasks. The following table provides descriptions of the modules included in *gulpfile.js*.

MODULE NAME	DESCRIPTION
gulp	The Gulp streaming build system. For more information, see gulp .
rimraf	A Node deletion module. For more information, see rimraf .
gulp-concat	A module that concatenates files based on the operating system's newline character. For more information, see gulp-concat .
gulp-cssmin	A module that minifies CSS files. For more information, see gulp-cssmin .
gulp-uglify	A module that minifies <i>.js</i> files. For more information, see gulp-uglify .

Once the requisite modules are imported, the tasks can be specified. Here there are six tasks registered, represented by the following code:

```

gulp.task("clean:js", function (cb) {
  rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
  rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);

```

The following table provides an explanation of the tasks specified in the code above:

TASK NAME	DESCRIPTION
clean:js	A task that uses the rimraf Node deletion module to remove the minified version of the site.js file.
clean:css	A task that uses the rimraf Node deletion module to remove the minified version of the site.css file.
clean	A task that calls the <code>clean:js</code> task, followed by the <code>clean:css</code> task.

TASK NAME	DESCRIPTION
min:js	A task that minifies and concatenates all .js files within the js folder. The .min.js files are excluded.
min:css	A task that minifies and concatenates all .css files within the css folder. The .min.css files are excluded.
min	A task that calls the <code>min:js</code> task, followed by the <code>min:css</code> task.

Running default tasks

If you haven't already created a new Web app, create a new ASP.NET Web Application project in Visual Studio.

1. Add a new JavaScript file to your project and name it *gulpfile.js*, then copy the following code.

```

/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var paths = {
  webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";

gulp.task("clean:js", function (cb) {
  rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
  rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});

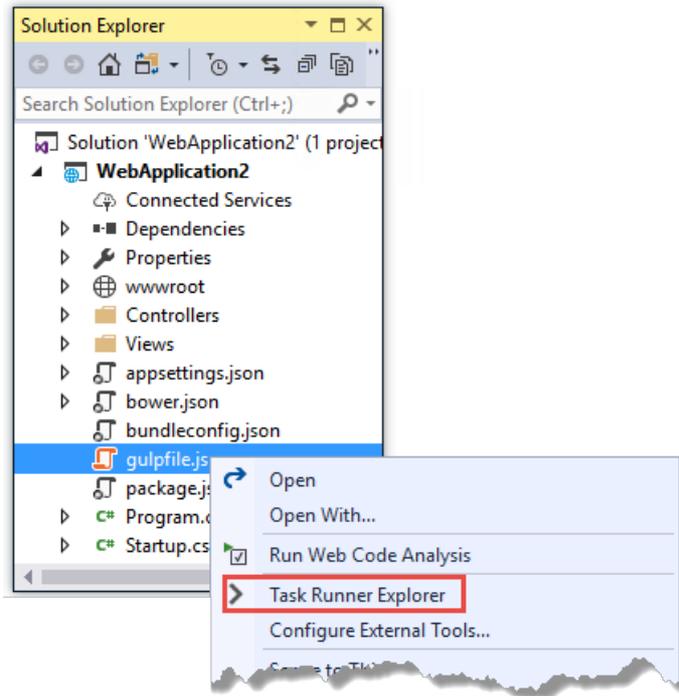
gulp.task("min", ["min:js", "min:css"]);

```

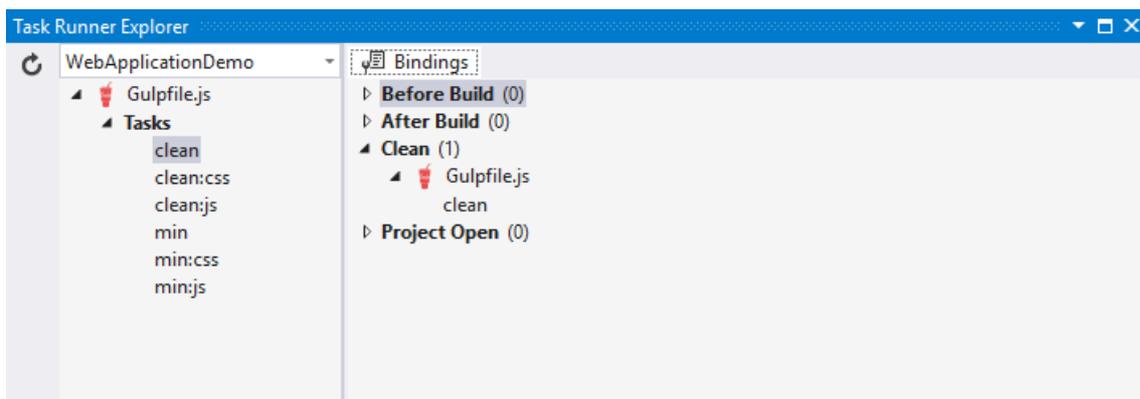
2. Open the *package.json* file (add if not there) and add the following.

```
{
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-concat": "2.6.1",
    "gulp-cssmin": "0.1.7",
    "gulp-uglify": "2.0.1",
    "rimraf": "2.6.1"
  }
}
```

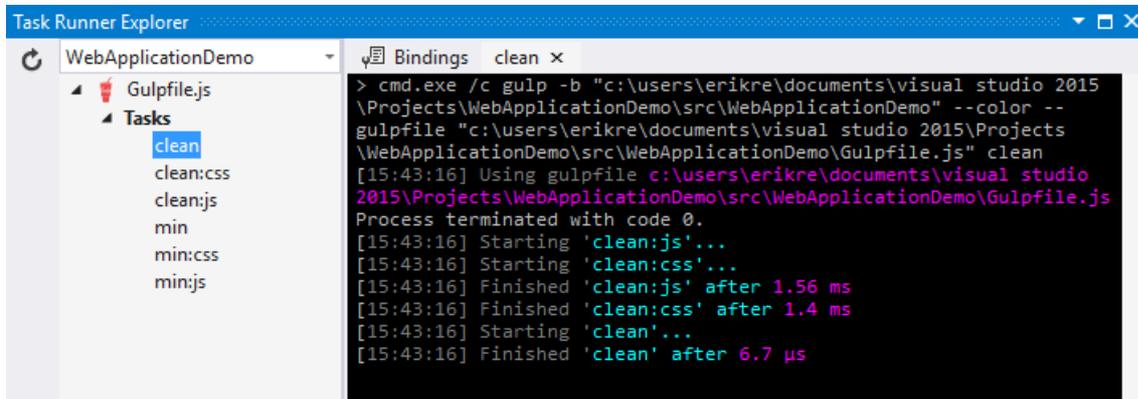
3. In **Solution Explorer**, right-click *gulpfile.js*, and select **Task Runner Explorer**.



Task Runner Explorer shows the list of Gulp tasks. (You might have to click the **Refresh** button that appears to the left of the project name.)

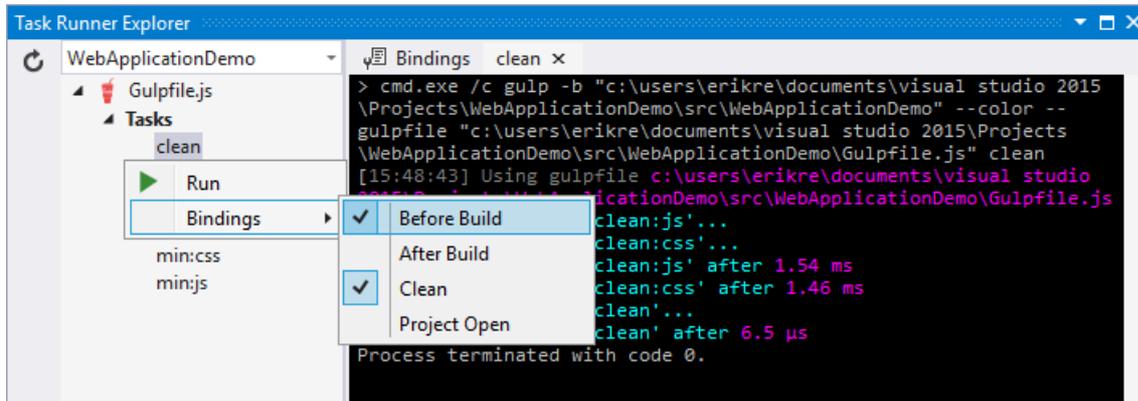


4. Underneath **Tasks** in **Task Runner Explorer**, right-click **clean**, and select **Run** from the pop-up menu.



Task Runner Explorer will create a new tab named **clean** and execute the clean task as it is defined in *gulpfile.js*.

5. Right-click the **clean** task, then select **Bindings > Before Build**.



The **Before Build** binding configures the clean task to run automatically before each build of the project.

The bindings you set up with **Task Runner Explorer** are stored in the form of a comment at the top of your *gulpfile.js* and are effective only in Visual Studio. An alternative that doesn't require Visual Studio is to configure automatic execution of gulp tasks in your *.csproj* file. For example, put this in your *.csproj* file:

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp clean" />
</Target>
```

Now the clean task is executed when you run the project in Visual Studio or from a command prompt using the `dotnet run` command (run `npm install` first).

Defining and running a new task

To define a new Gulp task, modify *gulpfile.js*.

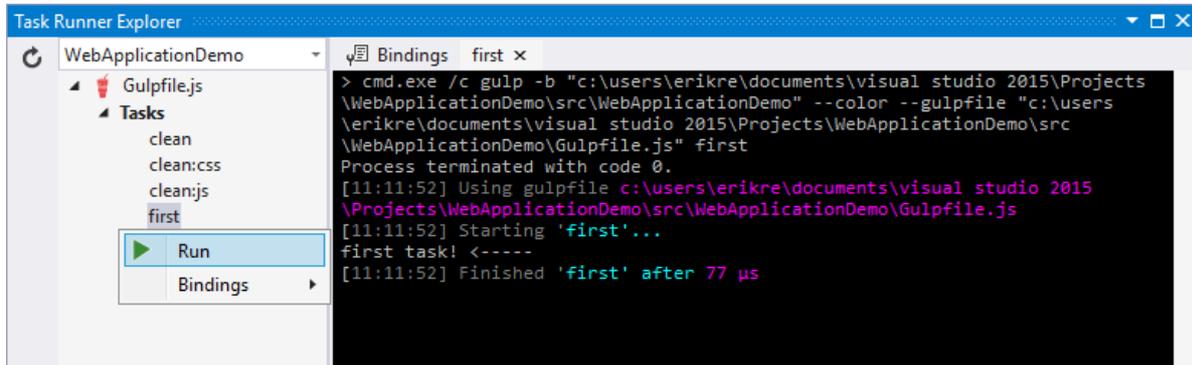
1. Add the following JavaScript to the end of *gulpfile.js*:

```
gulp.task("first", function () {
  console.log('first task! <----->');
});
```

This task is named `first`, and it simply displays a string.

2. Save *gulpfile.js*.
3. In **Solution Explorer**, right-click *gulpfile.js*, and select *Task Runner Explorer*.

4. In **Task Runner Explorer**, right-click **first**, and select **Run**.



You'll see that the output text is displayed. If you are interested in examples based on a common scenario, see [Gulp Recipes](#).

Defining and running tasks in a series

When you run multiple tasks, the tasks run concurrently by default. However, if you need to run tasks in a specific order, you must specify when each task is complete, as well as which tasks depend on the completion of another task.

1. To define a series of tasks to run in order, replace the `first` task that you added above in `gulpfile.js` with the following:

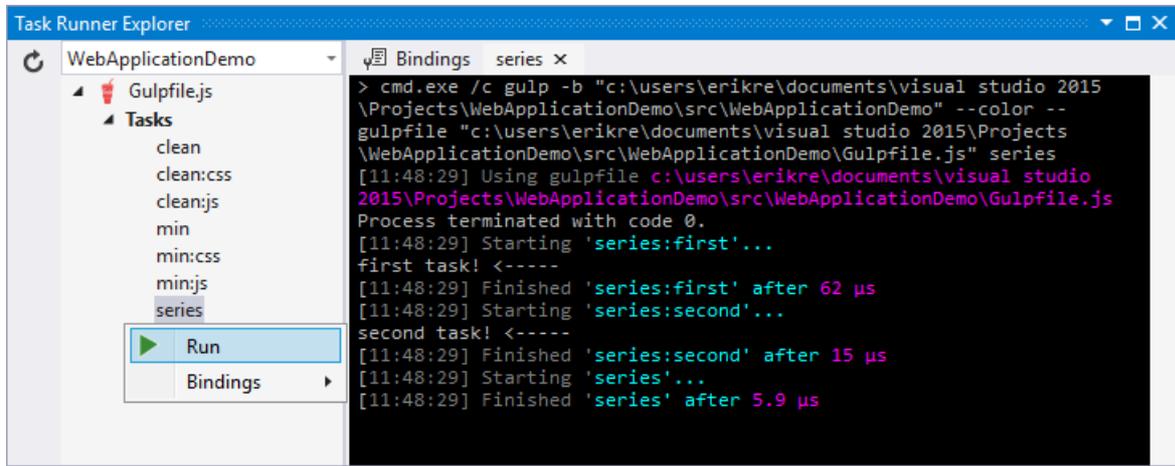
```
gulp.task("series:first", function () {
  console.log('first task! <-----');
});

gulp.task("series:second", ["series:first"], function () {
  console.log('second task! <-----');
});

gulp.task("series", ["series:first", "series:second"], function () {});
```

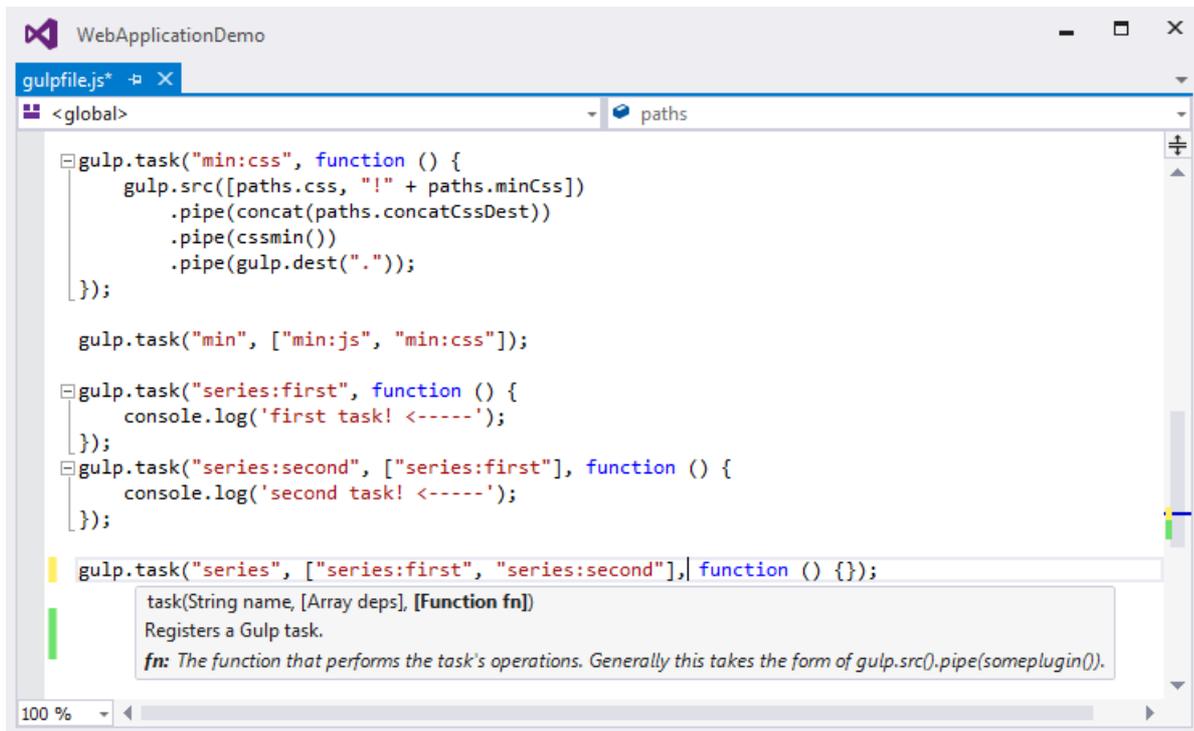
You now have three tasks: `series:first`, `series:second`, and `series`. The `series:second` task includes a second parameter which specifies an array of tasks to be run and completed before the `series:second` task will run. As specified in the code above, only the `series:first` task must be completed before the `series:second` task will run.

2. Save `gulpfile.js`.
3. In **Solution Explorer**, right-click `gulpfile.js` and select **Task Runner Explorer** if it isn't already open.
4. In **Task Runner Explorer**, right-click `series` and select **Run**.



IntelliSense

IntelliSense provides code completion, parameter descriptions, and other features to boost productivity and to decrease errors. Gulp tasks are written in JavaScript; therefore, IntelliSense can provide assistance while developing. As you work with JavaScript, IntelliSense lists the objects, functions, properties, and parameters that are available based on your current context. Select a coding option from the pop-up list provided by IntelliSense to complete the code.



For more information about IntelliSense, see [JavaScript IntelliSense](#).

Development, staging, and production environments

When Gulp is used to optimize client-side files for staging and production, the processed files are saved to a local staging and production location. The `_Layout.cshtml` file uses the **environment** tag helper to provide two different versions of CSS files. One version of CSS files is for development and the other version is optimized for both staging and production. In Visual Studio 2017, when you change the **ASPNETCORE_ENVIRONMENT** environment variable to `Production`, Visual Studio will build the Web app and link to the minimized CSS files. The following markup shows the **environment** tag helpers containing link tags to the `DeveIopment` CSS files and the minified `Staging, Production` CSS files.

```

<environment names="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery"
    crossorigin="anonymous"
    integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
    asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
    crossorigin="anonymous"
    integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIPg9mGCD8wGNIcPD7Txa">
  </script>
  <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

```

Switching between environments

To switch between compiling for different environments, modify the **ASPNETCORE_ENVIRONMENT** environment variable's value.

1. In **Task Runner Explorer**, verify that the **min** task has been set to run **Before Build**.
2. In **Solution Explorer**, right-click the project name and select **Properties**.

The property sheet for the Web app is displayed.

3. Click the **Debug** tab.
4. Set the value of the **Hosting:Environment** environment variable to .
5. Press **F5** to run the application in a browser.
6. In the browser window, right-click the page and select **View Source** to view the HTML for the page.

Notice that the stylesheet links point to the minified CSS files.

7. Close the browser to stop the Web app.
8. In Visual Studio, return to the property sheet for the Web app and change the **Hosting:Environment** environment variable back to .
9. Press **F5** to run the application in a browser again.

10. In the browser window, right-click the page and select **View Source** to see the HTML for the page.

Notice that the stylesheet links point to the unminified versions of the CSS files.

For more information related to environments in ASP.NET Core, see [Working with Multiple Environments](#).

Task and module details

A Gulp task is registered with a function name. You can specify dependencies if other tasks must run before the current task. Additional functions allow you to run and watch the Gulp tasks, as well as set the source (*src*) and destination (*dest*) of the files being modified. The following are the primary Gulp API functions:

GULP FUNCTION	SYNTAX	DESCRIPTION
task	<code>gulp.task(name[, deps], fn) { }</code>	The <code>task</code> function creates a task. The <code>name</code> parameter defines the name of the task. The <code>deps</code> parameter contains an array of tasks to be completed before this task runs. The <code>fn</code> parameter represents a callback function which performs the operations of the task.
watch	<code>gulp.watch(glob [, opts], tasks) { }</code>	The <code>watch</code> function monitors files and runs tasks when a file change occurs. The <code>glob</code> parameter is a <code>string</code> or <code>array</code> that determines which files to watch. The <code>opts</code> parameter provides additional file watching options.
src	<code>gulp.src(globs[, options]) { }</code>	The <code>src</code> function provides files that match the glob value(s). The <code>glob</code> parameter is a <code>string</code> or <code>array</code> that determines which files to read. The <code>options</code> parameter provides additional file options.
dest	<code>gulp.dest(path[, options]) { }</code>	The <code>dest</code> function defines a location to which files can be written. The <code>path</code> parameter is a string or function that determines the destination folder. The <code>options</code> parameter is an object that specifies output folder options.

For additional Gulp API reference information, see [Gulp Docs API](#).

Gulp recipes

The Gulp community provides Gulp [recipes](#). These recipes consist of Gulp tasks to address common scenarios.

Additional resources

- [Gulp documentation](#)
- [Bundling and minification in ASP.NET Core](#)
- [Using Grunt in ASP.NET Core](#)

Using Grunt in ASP.NET Core

9/12/2017 • 8 min to read • [Edit Online](#)

By [Noel Rice](#)

Grunt is a JavaScript task runner that automates script minification, TypeScript compilation, code quality "lint" tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is fully supported in Visual Studio, though the ASP.NET project templates use Gulp by default (see [Using Gulp](#)).

This example uses an empty ASP.NET Core project as its starting point, to show how to automate the client build process from scratch.

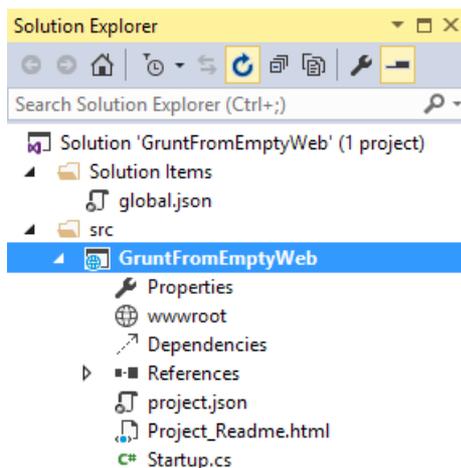
The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

- **grunt**: The Grunt task runner package.
- **grunt-contrib-clean**: A plugin that removes files or directories.
- **grunt-contrib-jshint**: A plugin that reviews JavaScript code quality.
- **grunt-contrib-concat**: A plugin that joins files into a single file.
- **grunt-contrib-uglify**: A plugin that minifies JavaScript to reduce size.
- **grunt-contrib-watch**: A plugin that watches file activity.

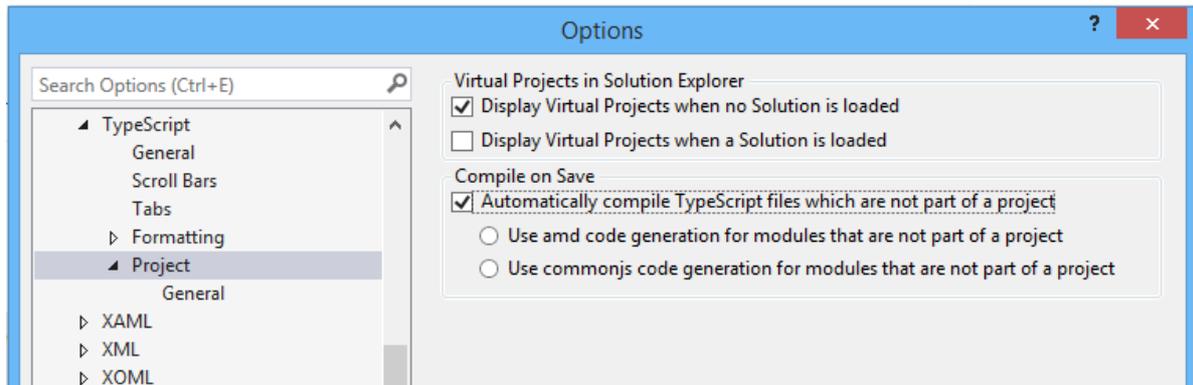
Preparing the application

To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio settings and will be our raw material to process using Grunt.

1. In Visual Studio, create a new `ASP.NET Web Application`.
2. In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template and click the OK button.
3. In the Solution Explorer, review the project structure. The `\src` folder includes empty `wwwroot` and `Dependencies` nodes.



4. Add a new folder named `TypeScript` to your project directory.
5. Before adding any files, let's make sure that Visual Studio has the option 'compile on save' for TypeScript files checked. *Tools > Options > Text Editor > Typescript > Project*



6. Right-click the `TypeScript` directory and select **Add > New Item** from the context menu. Select the **JavaScript file** item and name the file `Tastes.ts` (note the `.ts` extension). Copy the line of TypeScript code below into the file (when you save, a new `Tastes.js` file will appear with the JavaScript source).

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. Add a second file to the **TypeScript** directory and name it `Food.ts`. Copy the code below into the file.

```
class Food {
  constructor(name: string, calories: number) {
    this._name = name;
    this._calories = calories;
  }

  private _name: string;
  get Name() {
    return this._name;
  }

  private _calories: number;
  get Calories() {
    return this._calories;
  }

  private _taste: Tastes;
  get Taste(): Tastes { return this._taste }
  set Taste(value: Tastes) {
    this._taste = value;
  }
}
```

Configuring NPM

Next, configure NPM to download grunt and grunt-tasks.

1. In the Solution Explorer, right-click the project and select **Add > New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, `package.json`, and click the **Add** button.
2. In the `package.json` file, inside the `devDependencies` object braces, enter "grunt". Select `grunt` from the Intellisense list and press the Enter key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press `Ctrl+Space` if Intellisense does not appear).

```
"devDependencies": {
  "grunt": "0.4.5"
}
```

NOTE

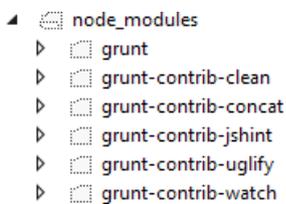
NPM uses [semantic versioning](#) to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme ... Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (0.4.5 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version. See the [NPM semver version parser reference](#) as a guide to the full expressivity that SemVer provides.

- 3. Add more dependencies to load grunt-contrib-* packages for *clean*, *jshint*, *concat*, *uglify*, and *watch* as shown in the example below. The versions do not need to match the example.

```
"devDependencies": {
  "grunt": "0.4.5",
  "grunt-contrib-clean": "0.6.0",
  "grunt-contrib-jshint": "0.11.0",
  "grunt-contrib-concat": "0.5.1",
  "grunt-contrib-uglify": "0.8.0",
  "grunt-contrib-watch": "0.6.1"
}
```

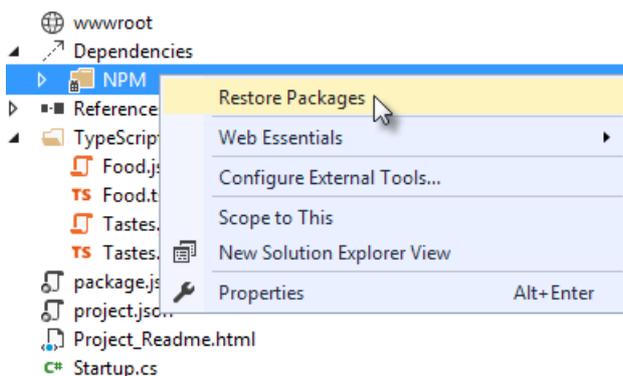
- 4. Save the *package.json* file.

The packages for each devDependencies item will download, along with any files that each package requires. You can find the package files in the `node_modules` directory by enabling the **Show All Files** button in the Solution Explorer.



NOTE

If you need to, you can manually restore dependencies in Solution Explorer by right-clicking on `Dependencies\NPM` and selecting the **Restore Packages** menu option.



Configuring Grunt

Grunt is configured using a manifest named *Gruntfile.js* that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add > New Item**. Select the **Grunt Configuration file** option, leave the default name, *Gruntfile.js*, and click the **Add** button.

The initial code includes a module definition and the `grunt.initConfig()` method. The `initConfig()` is used to set options for each package, and the remainder of the module will load and register tasks.

```
module.exports = function (grunt) {
  grunt.initConfig({
  });
};
```

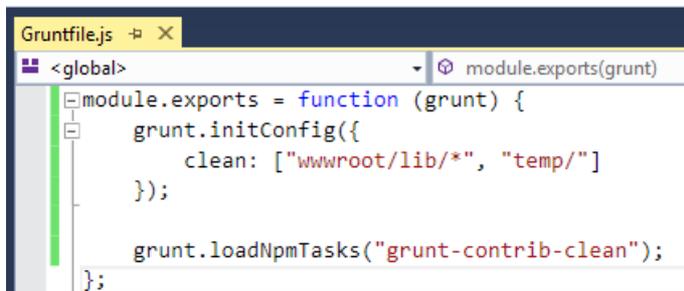
2. Inside the `initConfig()` method, add options for the `clean` task as shown in the example *Gruntfile.js* below. The clean task accepts an array of directory strings. This task removes files from `wwwroot/lib` and removes the entire `/temp` directory.

```
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"],
  });
};
```

3. Below the `initConfig()` method, add a call to `grunt.loadNpmTasks()`. This will make the task runnable from Visual Studio.

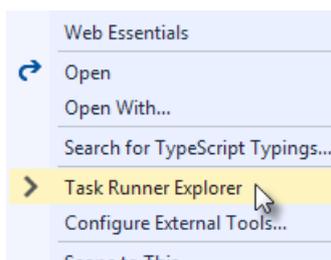
```
grunt.loadNpmTasks("grunt-contrib-clean");
```

4. Save *Gruntfile.js*. The file should look something like the screenshot below.

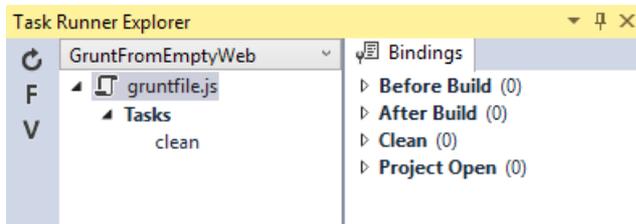


```
Gruntfile.js
<global> module.exports(grunt)
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"]
  });
  grunt.loadNpmTasks("grunt-contrib-clean");
};
```

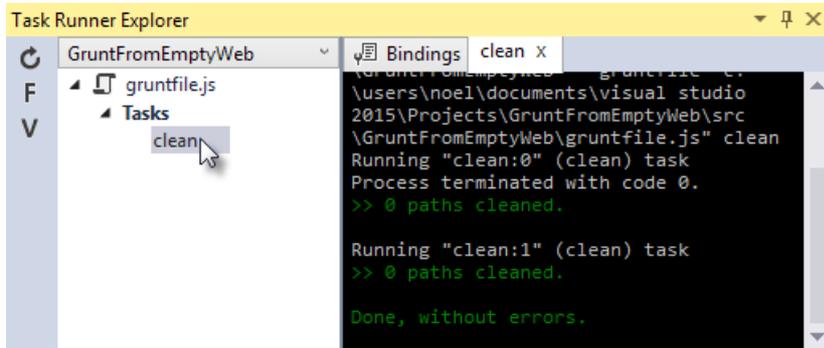
5. Right-click *Gruntfile.js* and select **Task Runner Explorer** from the context menu. The Task Runner Explorer window will open.



6. Verify that `clean` shows under **Tasks** in the Task Runner Explorer.



- Right-click the clean task and select **Run** from the context menu. A command window displays progress of the task.



NOTE

There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

- In the `initConfig()` method, add an entry for `concat` using the code below.

The `src` property array lists files to combine, in the order that they should be combined. The `dest` property assigns the path to the combined file that is produced.

```
concat: {
  all: {
    src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
    dest: 'temp/combined.js'
  }
},
```

NOTE

The `all` property in the code above is the name of a target. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using Intellisense or assign your own.

- Add the `jshint` task using the code below.

The jshint code-quality utility is run against every JavaScript file found in the temp directory.

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

NOTE

The option "-W069" is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, i.e. `Tastes["Sweet"]` instead of `Tastes.Sweet`. The option turns off the warning to allow the rest of the process to continue.

10. Add the `uglify` task using the code below.

The task minifies the `combined.js` file found in the temp directory and creates the result file in `wwwroot/lib` following the standard naming convention `<file name>.min.js`.

```
uglify: {
  all: {
    src: ['temp/combined.js'],
    dest: 'wwwroot/lib/combined.min.js'
  }
},
```

11. Under the call `grunt.loadNpmTasks()` that loads `grunt-contrib-clean`, include the same call for `jshint`, `concat` and `uglify` using the code below.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
```

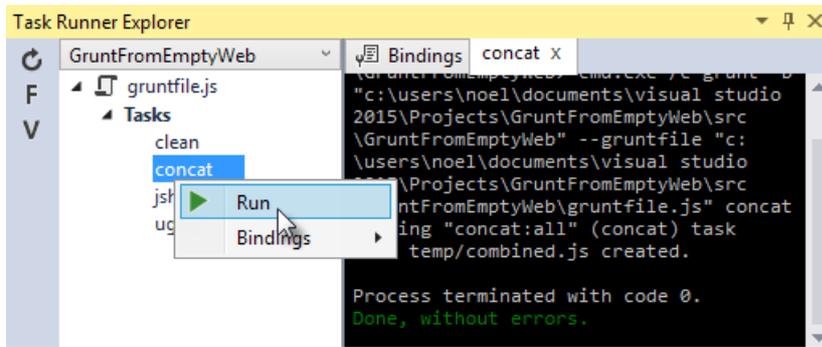
12. Save `Gruntfile.js`. The file should look something like the example below.



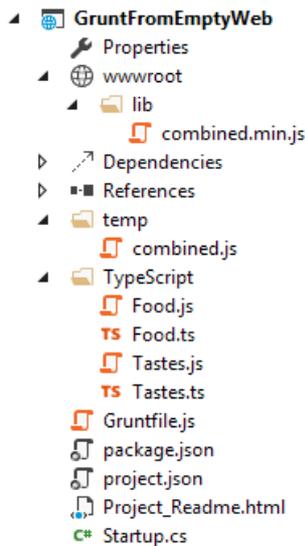
```
Gruntfile.js  X
({} module exports(grunt)
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"],
    concat: {
      all: {
        src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
        dest: 'temp/combined.js'
      }
    },
    jshint: { files: ['temp/*.js'], options: { '-W069': false, } },
    uglify: {
      all: {
        src: ['temp/combined.js'],
        dest: 'wwwroot/lib/combined.min.js'
      }
    }
  });
};

grunt.loadNpmTasks('grunt-contrib-clean');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
```

13. Notice that the Task Runner Explorer Tasks list includes `clean`, `concat`, `jshint` and `uglify` tasks. Run each task in order and observe the results in Solution Explorer. Each task should run without errors.



The concat task creates a new *combined.js* file and places it into the temp directory. The jshint task simply runs and doesn't produce output. The uglify task creates a new *combined.min.js* file and places it into *wwwroot/lib*. On completion, the solution should look something like the screenshot below:



NOTE

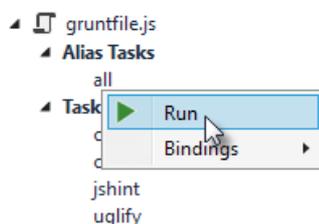
For more information on the options for each package, visit <https://www.npmjs.com/> and lookup the package name in the search box on the main page. For example, you can look up the `grunt-contrib-clean` package to get a documentation link that explains all of its parameters.

All together now

Use the Grunt `registerTask()` method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order `clean -> concat -> jshint -> uglify`, add the code below to the module. The code should be added to the same level as the `loadNpmTasks()` calls, outside `initConfig`.

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The `all` task will run `clean`, `concat`, `jshint` and `uglify`, in order.



Watching for changes

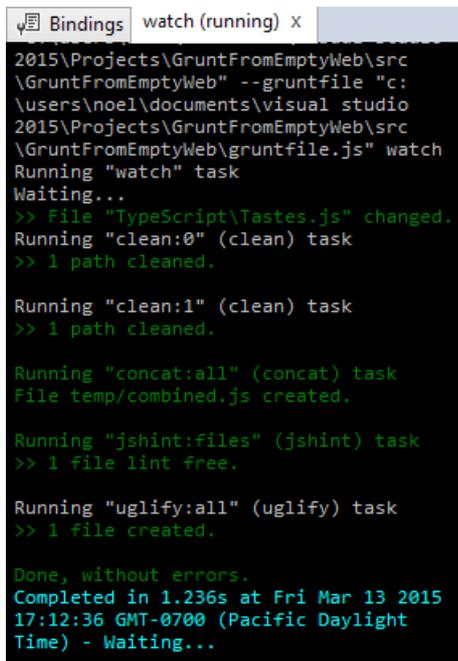
A `watch` task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to `initConfig` to watch for changes to `*.js` files in the TypeScript directory. If a JavaScript file is changed, `watch` will run the `all` task.

```
watch: {
  files: ["TypeScript/*.js"],
  tasks: ["all"]
}
```

Add a call to `loadNpmTasks()` to show the `watch` task in Task Runner Explorer.

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

Right-click the watch task in Task Runner Explorer and select Run from the context menu. The command window that shows the watch task running will display a "Waiting..." message. Open one of the TypeScript files, add a space, and then save the file. This will trigger the watch task and trigger the other tasks to run in order. The screenshot below shows a sample run.



```
ψ Bindings watch (running) x
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb" --gruntfile "c:
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" watch
Running "watch" task
Waiting...
>> File "TypeScript\Tastes.js" changed.
Running "clean:0" (clean) task
>> 1 path cleaned.

Running "clean:1" (clean) task
>> 1 path cleaned.

Running "concat:all" (concat) task
File temp/combined.js created.

Running "jshint:files" (jshint) task
>> 1 file lint free.

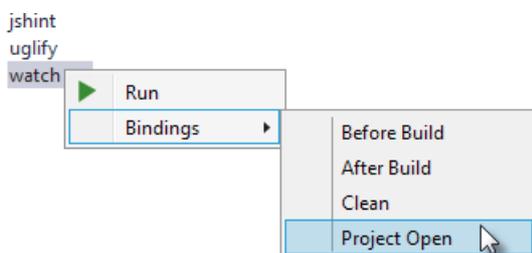
Running "uglify:all" (uglify) task
>> 1 file created.

Done, without errors.
Completed in 1.236s at Fri Mar 13 2015
17:12:36 GMT-0700 (Pacific Daylight
Time) - Waiting...
```

Binding to Visual Studio events

Unless you want to manually start your tasks every time you work in Visual Studio, you can bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Let's bind `watch` so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the watch task and select **Bindings > Project Open** from the context menu.



Unload and reload the project. When the project loads again, the watch task will start running automatically.

Summary

Grunt is a powerful task runner that can be used to automate most client-build tasks. Grunt leverages NPM to deliver its packages, and features tooling integration with Visual Studio. Visual Studio's Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

Additional resources

- [Using Gulp](#)

Manage client-side packages with Bower in ASP.NET Core

12/22/2017 • 5 min to read • [Edit Online](#)

By [Rick Anderson](#), [Noel Rice](#), and [Scott Addie](#)

IMPORTANT

While Bower is maintained, they recommend using a different solution. Yarn with Webpack is one popular alternative for which [migration instructions](#) are available.

[Bower](#) calls itself "A package manager for the web". Within the .NET ecosystem, it fills the void left by NuGet's inability to deliver static content files. For ASP.NET Core projects, these static files are inherent to client-side libraries like [jQuery](#) and [Bootstrap](#). For .NET libraries, you still use [NuGet](#) package manager.

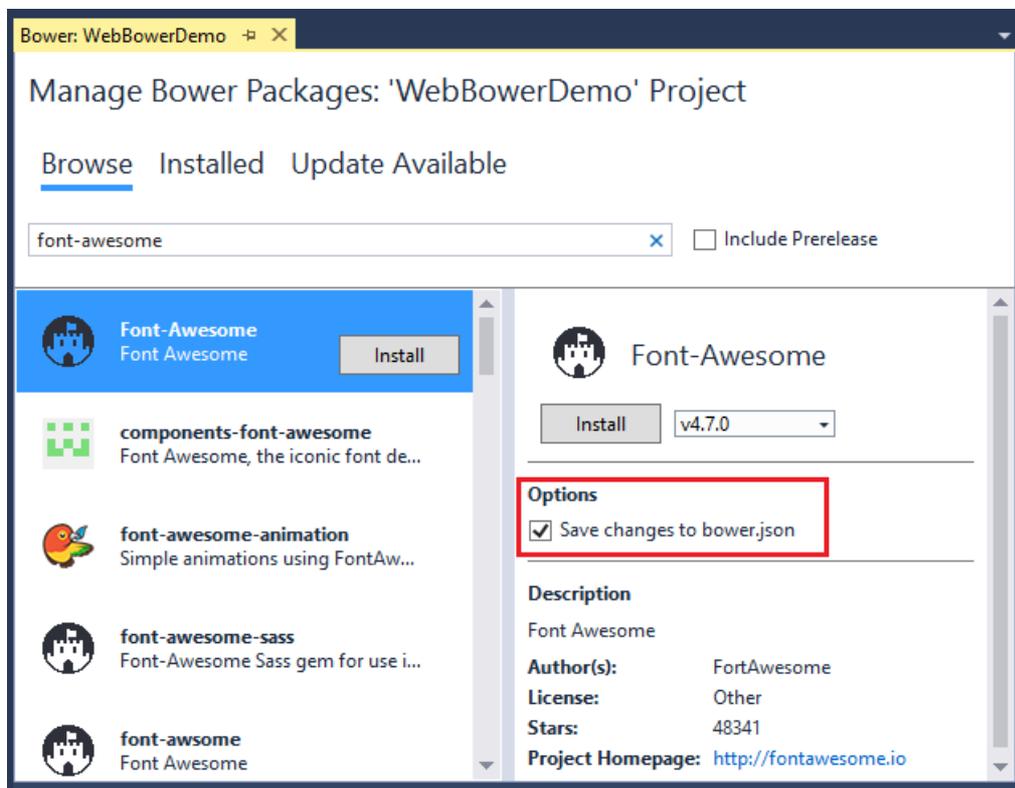
New projects created with the ASP.NET Core project templates set up the client-side build process. [jQuery](#) and [Bootstrap](#) are installed, and Bower is supported.

Client-side packages are listed in the *bower.json* file. The ASP.NET Core project templates configures *bower.json* with jQuery, jQuery validation, and Bootstrap.

In this tutorial, we'll add support for [Font Awesome](#). Bower packages can be installed with the **Manage Bower Packages** UI or manually in the *bower.json* file.

Installation via Manage Bower Packages UI

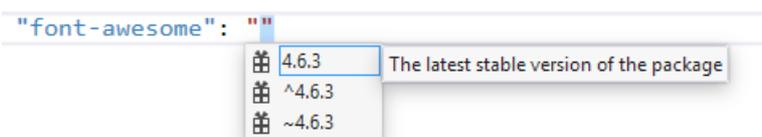
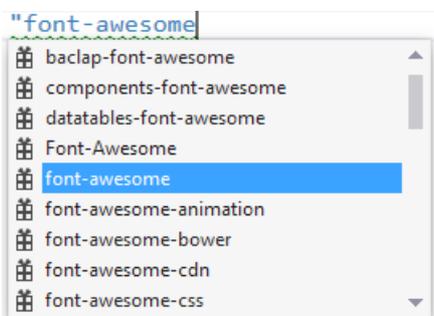
- Create a new ASP.NET Core Web app with the **ASP.NET Core Web Application (.NET Core)** template. Select **Web Application** and **No Authentication**.
- Right-click the project in Solution Explorer and select **Manage Bower Packages** (alternatively from the main menu, **Project** > **Manage Bower Packages**).
- In the **Bower: <project name>** window, click the "Browse" tab, and then filter the packages list by entering in the search box:



- Confirm that the "Save changes to *bower.json*" checkbox is checked. Select a version from the drop-down list and click the **Install** button. The **Output** window shows the installation details.

Manual installation in *bower.json*

Open the *bower.json* file and add "font-awesome" to the dependencies. IntelliSense shows the available packages. When a package is selected, the available versions are displayed. The images below are older and will not match what you see.



Bower uses [semantic versioning](#) to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme `<major>.<minor>.<patch>`. IntelliSense simplifies semantic versioning by showing only a few common choices. The top item in the IntelliSense list (4.6.3 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version.

Save the *bower.json* file. Visual Studio watches the *bower.json* file for changes. Upon saving, the *bower install* command is executed. See the Output window's **Bower/npm** view for the exact command executed.

Open the *bowerrc* file under *bower.json*. The `directory` property is set to *wwwroot/lib* which indicates the location Bower will install the package assets.

```
{
  "directory": "wwwroot/lib"
}
```

You can use the search box in Solution Explorer to find and display the font-awesome package.

Open the *Views\Shared_Layout.cshtml* file and add the font-awesome CSS file to the environment [Tag Helper](#) for `Development`. From Solution Explorer, drag and drop *font-awesome.css* inside the `<environment names="Development">` element.

```
<environment names="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
  <link href="~/lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</environment>
```

In a production app you would add *font-awesome.min.css* to the environment tag helper for `Staging, Production`.

Replace the contents of the *Views\Home>About.cshtml* Razor file with the following markup:

```
@{
    ViewData["Title"] = "About";
}

<div class="list-group">
  <a class="list-group-item" href="#"><i class="fa fa-home fa-fw" aria-hidden="true"></i>&nbsp; Home</a>
  <a class="list-group-item" href="#"><i class="fa fa-book fa-fw" aria-hidden="true"></i>&nbsp; Library</a>
  <a class="list-group-item" href="#"><i class="fa fa-pencil fa-fw" aria-hidden="true"></i>&nbsp;
  Applications</a>
  <a class="list-group-item" href="#"><i class="fa fa-cog fa-fw" aria-hidden="true"></i>&nbsp; Settings</a>
</div>
```

Run the app and navigate to the About view to verify the font-awesome package works.

Exploring the client-side build process

Most ASP.NET Core project templates are already configured to use Bower. This next walkthrough starts with an empty ASP.NET Core project and adds each piece manually, so you can get a feel for how Bower is used in a project. You can see what happens to the project structure and the runtime output as each configuration change is made.

The general steps to use the client-side build process with Bower are:

- Define packages used in your project.
- Reference packages from your web pages.

Define packages

Once you list packages in the *bower.json* file, Visual Studio will download them. The following example uses Bower to load jQuery and Bootstrap to the *wwwroot* folder.

- Create a new ASP.NET Core Web app with the **ASP.NET Core Web Application (.NET Core)** template. Select the **Empty** project template and click **OK**.
- In Solution Explorer, right-click the project > **Add New Item** and select **Bower Configuration File**. Note: A *.bowerrc* file is also added.
- Open *bower.json*, and add jquery and bootstrap to the `dependencies` section. The resulting *bower.json* file

will look like the following example. The versions will change over time and may not match the image below.

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "jquery": "3.1.1",
    "bootstrap": "3.3.7"
  }
}
```

- Save the *bower.json* file.

Verify the project includes the *bootstrap* and *jQuery* directories in *wwwroot/lib*. Bower uses the *.bowerrc* file to install the assets in *wwwroot/lib*.

Note: The "Manage Bower Packages" UI provides an alternative to manual file editing.

Enable static files

- Add the `Microsoft.AspNetCore.StaticFiles` NuGet package to the project.
- Enable static files to be served with the [Static file middleware](#). Add a call to `UseStaticFiles` to the `Configure` method of `Startup`.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseStaticFiles();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

Reference packages

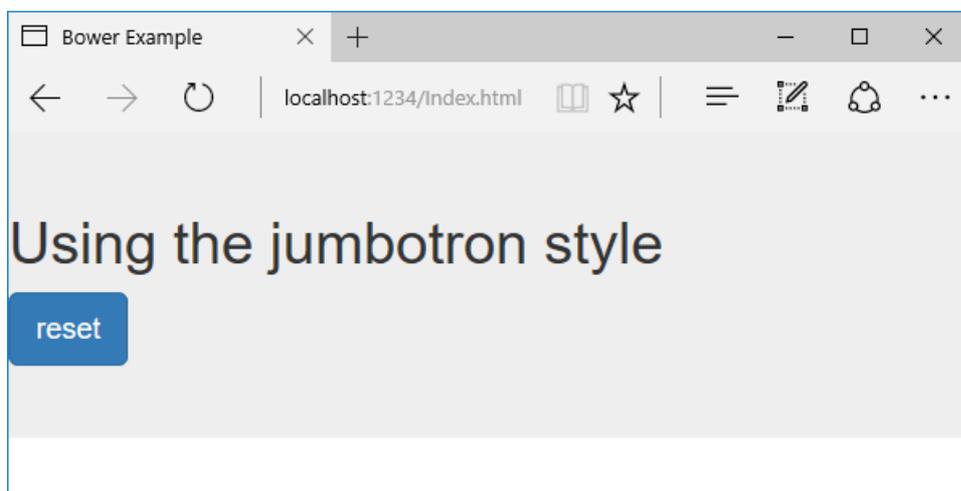
In this section, you will create an HTML page to verify it can access the deployed packages.

- Add a new HTML page named *Index.html* to the *wwwroot* folder. Note: You must add the HTML file to the *wwwroot* folder. By default, static content cannot be served outside *wwwroot*. See [Working with static files](#) for more information.

Replace the contents of *Index.html* with the following markup:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Bower Example</title>
  <link href="lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
  <div class="jumbotron">
    <h1>Using the jumbotron style</h1>
    <p>
      <a class="btn btn-primary btn-lg" role="button">Stateful button</a>
    </p>
  </div>
  <script src="lib/jquery/dist/jquery.js"></script>
  <script src="lib/bootstrap/dist/js/bootstrap.js"></script>
  <script>
    $(".btn").click(function () {
      $(this).text('loading')
        .delay(1000)
        .queue(function () {
          $(this).text('reset');
          $(this).dequeue();
        });
    });
  </script>
</body>
</html>
```

- Run the app and navigate to `http://localhost:<port>/Index.html`. Alternatively, with `Index.html` opened, press `Ctrl+Shift+W`. Verify that the jumbotron styling is applied, the jQuery code responds when the button is clicked, and that the Bootstrap button changes state.



Building beautiful, responsive sites with Bootstrap

10/13/2017 • 11 min to read • [Edit Online](#)

By [Steve Smith](#)

Bootstrap is currently the most popular web framework for developing responsive web applications. It offers a number of features and benefits that can improve your users' experience with your web site, whether you're a novice at front-end design and development or an expert. Bootstrap is deployed as a set of CSS and JavaScript files, and is designed to help your website or application scale efficiently from phones to tablets to desktops.

Getting started

There are several ways to get started with Bootstrap. If you're starting a new web application in Visual Studio, you can choose the default starter template for ASP.NET Core, in which case Bootstrap will come pre-installed:



Adding Bootstrap to an ASP.NET Core project is simply a matter of adding it to *bower.json* as a dependency:

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

This is the recommended way to add Bootstrap to an ASP.NET Core project.

You can also install bootstrap using one of several package managers, such as Bower, npm, or NuGet. In each case, the process is essentially the same:

Bower

```
bower install bootstrap
```

npm

```
npm install bootstrap
```

NuGet

```
Install-Package bootstrap
```

NOTE

The recommended way to install client-side dependencies like Bootstrap in ASP.NET Core is via Bower (using *bower.json*, as shown above). The use of npm/NuGet are shown to demonstrate how easily Bootstrap can be added to other kinds of web applications, including earlier versions of ASP.NET.

If you're referencing your own local versions of Bootstrap, you'll need to reference them in any pages that will use it. In production you should reference bootstrap using a CDN. In the default ASP.NET site template, the `_Layout.cshtml` file does so like this:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - WebApplication1</title>

  <environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment names="Staging,Production">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
      value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>
</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-
brand">WebApplication1</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
          <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
          <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
        </ul>
        @await Html.PartialAsync("~/Views/Shared/_LoginPartial")
      </div>
    </div>
  </div>
</body>
</html>
```

```

        </div>
    </div>
</div>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; 2016 - WebApplication1</p>
    </footer>
</div>

<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
        asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

    @RenderSection("scripts", required: false)
</body>
</html>

```

NOTE

If you're going to be using any of Bootstrap's jQuery plugins, you will also need to reference jQuery.

Basic templates and features

The most basic Bootstrap template looks very much like the *_Layout.cshtml* file shown above, and simply includes a basic menu for navigation and a place to render the rest of the page.

Basic navigation

The default template uses a set of `<div>` elements to render a top navbar and the main body of the page. If you're using HTML5, you can replace the first `<div>` tag with a `<nav>` tag to get the same effect, but with more precise semantics. Within this first `<div>` you can see there are several others. First, a `<div>` with a class of "container", and then within that, two more `<div>` elements: "navbar-header" and "navbar-collapse". The navbar-header div includes a button that will appear when the screen is below a certain minimum width, showing 3 horizontal lines (a so-called "hamburger icon"). The icon is rendered using pure HTML and CSS; no image is required. This is the code that displays the icon, with each of the tags rendering one of the white bars:

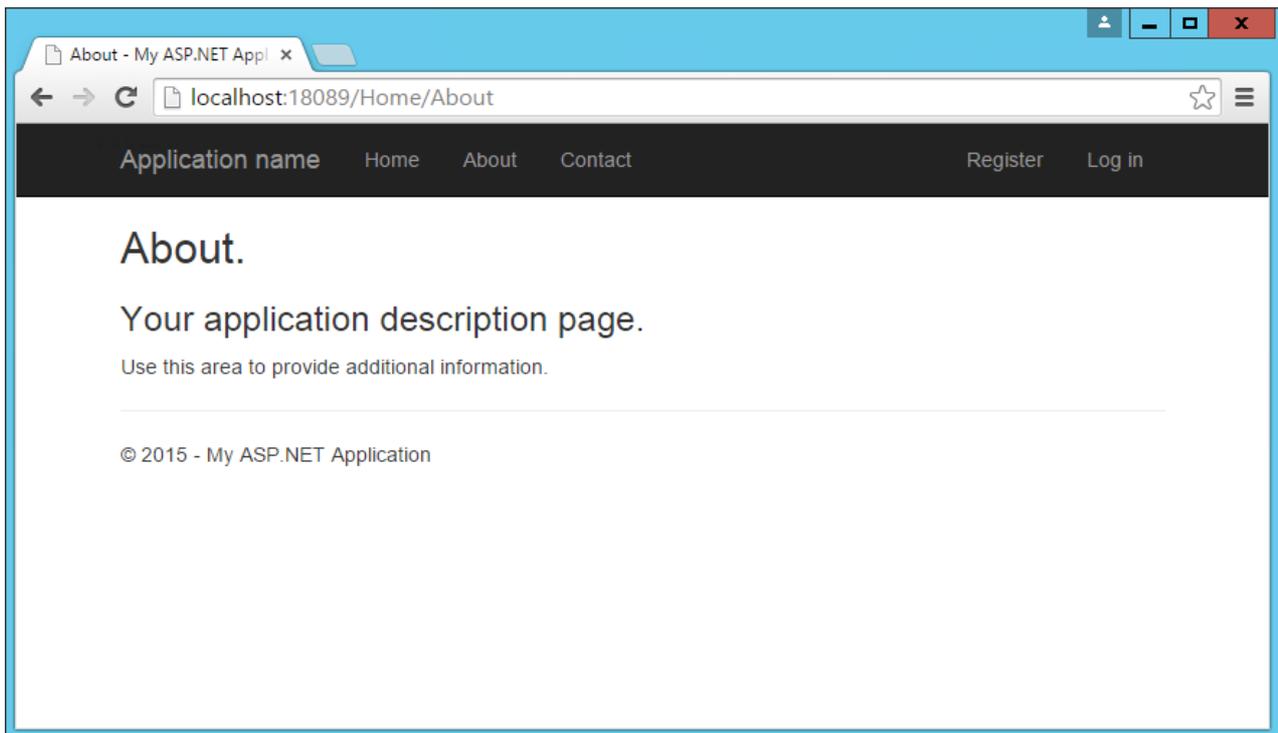
```

<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</button>

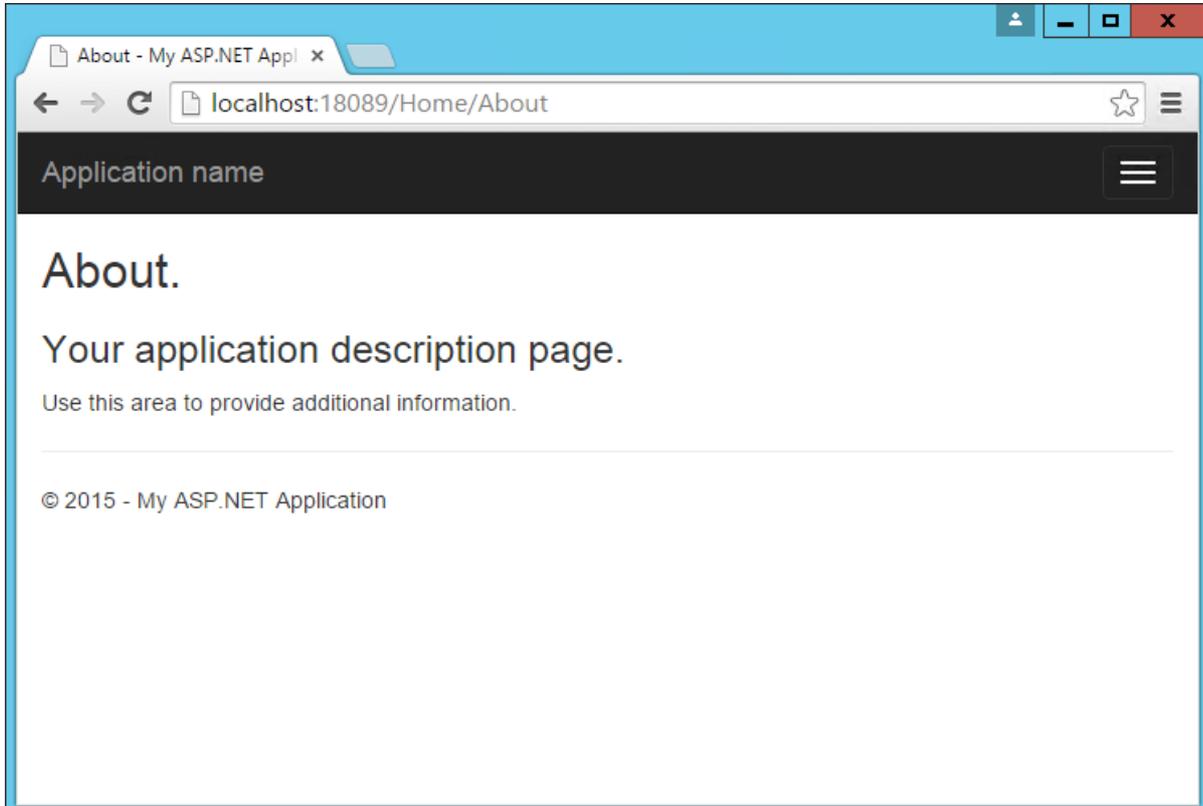
```

It also includes the application name, which appears in the top left. The main navigation menu is rendered by the `` element within the second div, and includes links to Home, About, and Contact. Additional links for Register and Login are added by the `_LoginPartial` line on line 29. Below the navigation, the main body of each page is

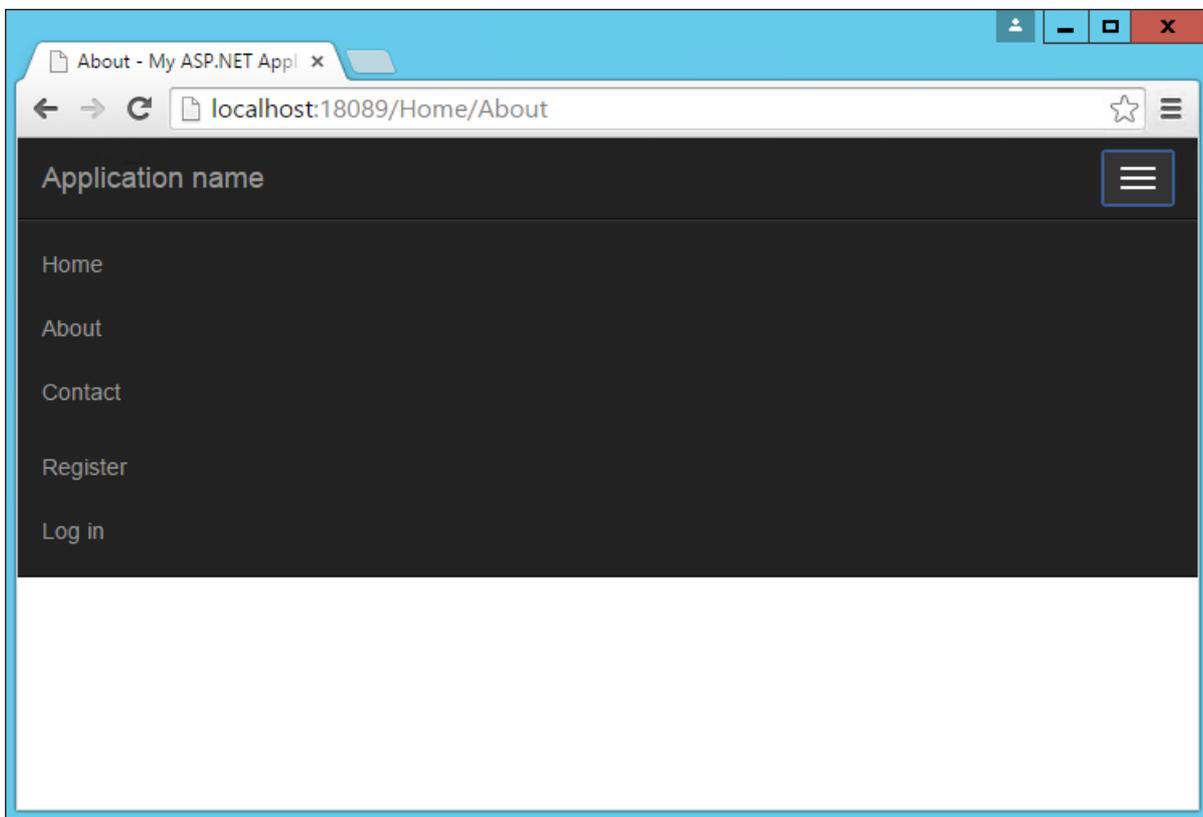
rendered in another `<div>`, marked with the "container" and "body-content" classes. In the simple default `_Layout` file shown here, the contents of the page are rendered by the specific View associated with the page, and then a simple `<footer>` is added to the end of the `<div>` element. You can see how the built-in About page appears using this template:



The collapsed navbar, with "hamburger" button in the top right, appears when the window drops below a certain width:



Clicking the icon reveals the menu items in a vertical drawer that slides down from the top of the page:



Typography and links

Bootstrap sets up the site's basic typography, colors, and link formatting in its CSS file. This CSS file includes default styles for tables, buttons, form elements, images, and more ([learn more](#)). One particularly useful feature is the grid layout system, covered next.

Grids

One of the most popular features of Bootstrap is its grid layout system. Modern web applications should avoid using the `<table>` tag for layout, instead restricting the use of this element to actual tabular data. Instead, columns and rows can be laid out using a series of `<div>` elements and the appropriate CSS classes. There are several advantages to this approach, including the ability to adjust the layout of grids to display vertically on narrow screens, such as on phones.

[Bootstrap's grid layout system](#) is based on twelve columns. This number was chosen because it can be divided evenly into 1, 2, 3, or 4 columns, and column widths can vary to within 1/12th of the vertical width of the screen. To start using the grid layout system, you should begin with a container `<div>` and then add a row `<div>`, as shown here:

```
<div class="container">
  <div class="row">
    ...
  </div>
</div>
```

Next, add additional `<div>` elements for each column, and specify the number of columns that `<div>` should occupy (out of 12) as part of a CSS class starting with "col-md-". For instance, if you want to simply have two columns of equal size, you would use a class of "col-md-6" for each one. In this case "md" is short for "medium" and refers to standard-sized desktop computer display sizes. There are four different options you can choose from, and each will be used for higher widths unless overridden (so if you want the layout to be fixed regardless of screen width, you can just specify xs classes).

CSS CLASS PREFIX	DEVICE TIER	WIDTH
col-xs-	Phones	< 768px
col-sm-	Tablets	>= 768px
col-md-	Desktops	>= 992px
col-lg-	Larger Desktop Displays	>= 1200px

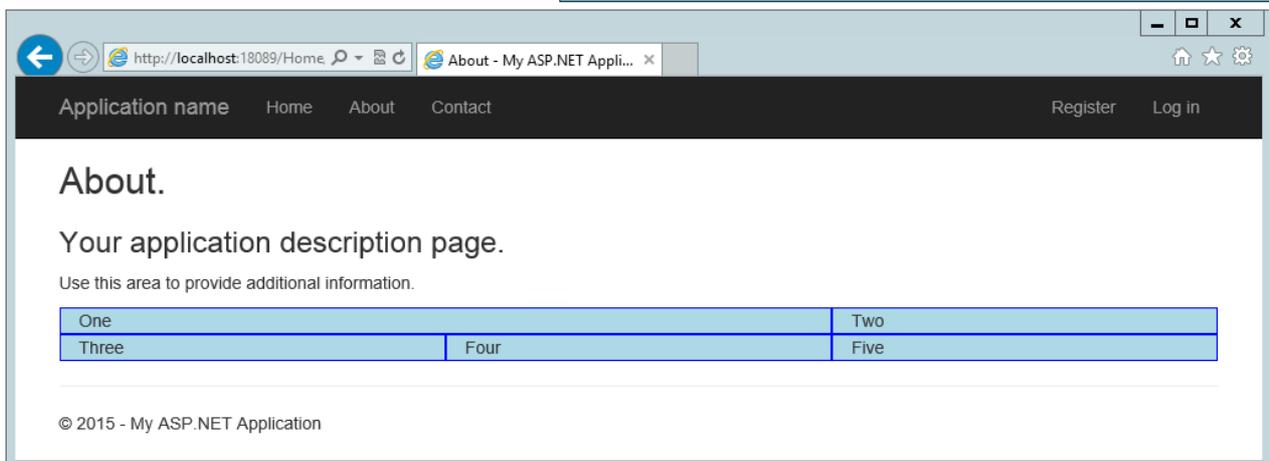
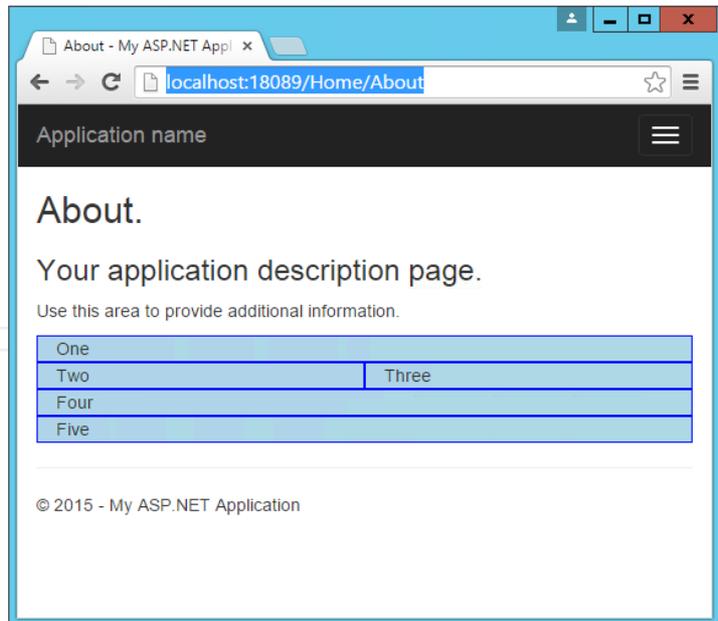
When specifying two columns both with "col-md-6" the resulting layout will be two columns at desktop resolutions, but these two columns will stack vertically when rendered on smaller devices (or a narrower browser window on a desktop), allowing users to easily view content without the need to scroll horizontally.

Bootstrap will always default to a single-column layout, so you only need to specify columns when you want more than one column. The only time you would want to explicitly specify that a `<div>` take up all 12 columns would be to override the behavior of a larger device tier. When specifying multiple device tier classes, you may need to reset the column rendering at certain points. Adding a clearfix div that is only visible within a certain viewport can achieve this, as shown here:

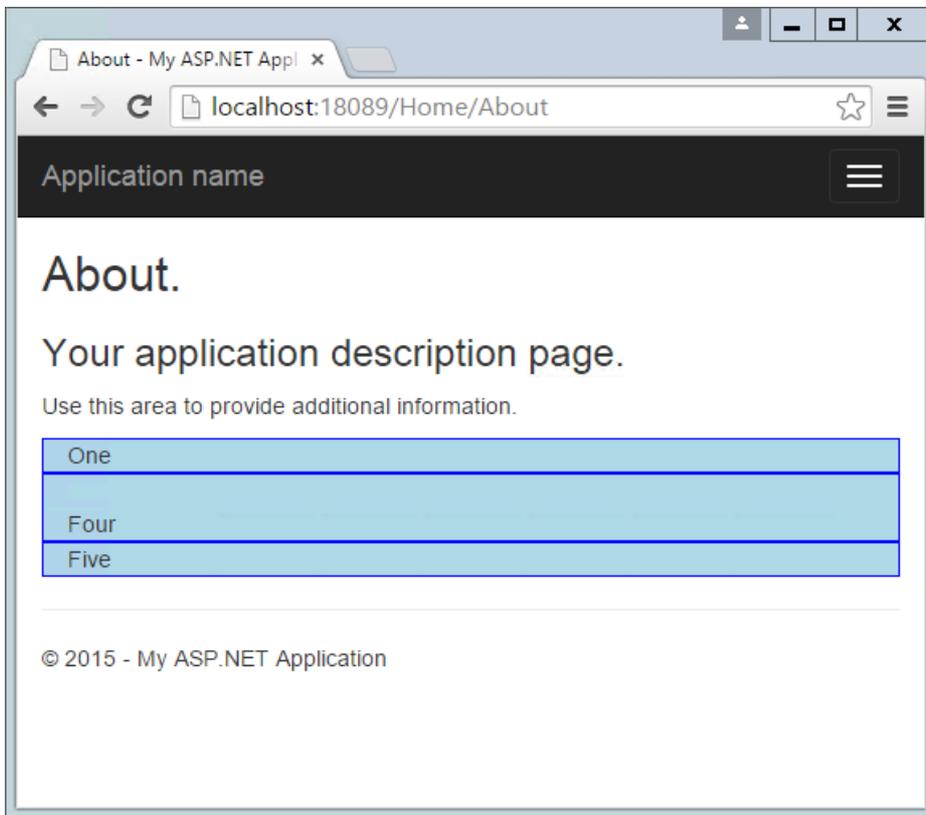
```

<p>Use this area to provide additional information.</p>
<style>
  [class*="col-"] {
    background-color: lightblue;
    border: 1px solid blue;
  }
</style>
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-md-8">
      One
    </div>
    <div class="col-xs-6 col-md-4">
      Two
    </div>
    <div class="col-xs-6 col-md-4">
      Three
    </div>
    <div class="clearfix visible-xs"></div>
    <div class="col-xs-12 col-md-4">
      Four
    </div>
    <div class="col-xs-12 col-md-4">
      Five
    </div>
  </div>
</div>

```



In the above example, One and Two share a row in the "md" layout, while Two and Three share a row in the "xs" layout. Without the clearfix `<div>`, Two and Three are not shown correctly in the "xs" view (note that only One, Four, and Five are shown):



In this example, only a single row `<div>` was used, and Bootstrap still mostly did the right thing with regard to the layout and stacking of the columns. Typically, you should specify a row `<div>` for each horizontal row your layout requires, and of course you can nest Bootstrap grids within one another. When you do, each nested grid will occupy 100% of the width of the element in which it is placed, which can then be subdivided using column classes.

Jumbotron

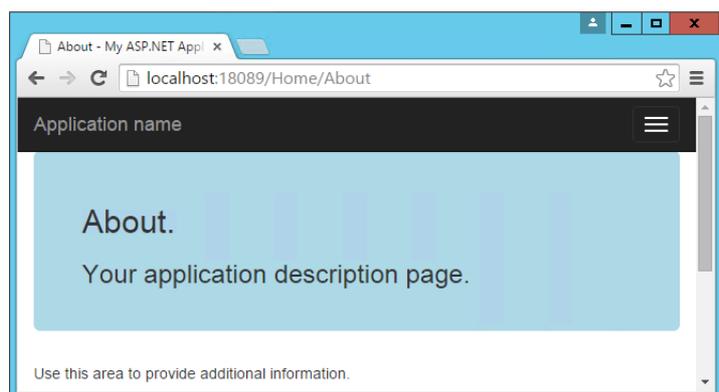
If you've used the default ASP.NET MVC templates in Visual Studio 2012 or 2013, you've probably seen the Jumbotron in action. It refers to a large full-width section of a page that can be used to display a large background image, a call to action, a rotator, or similar elements. To add a jumbotron to a page, simply add a `<div>` and give it a class of "jumbotron", then place a container `<div>` inside and add your content. We can easily adjust the standard About page to use a jumbotron for the main headings it displays:

```

<style>
  .jumbotron {
    background-color: lightblue;
  }
</style>

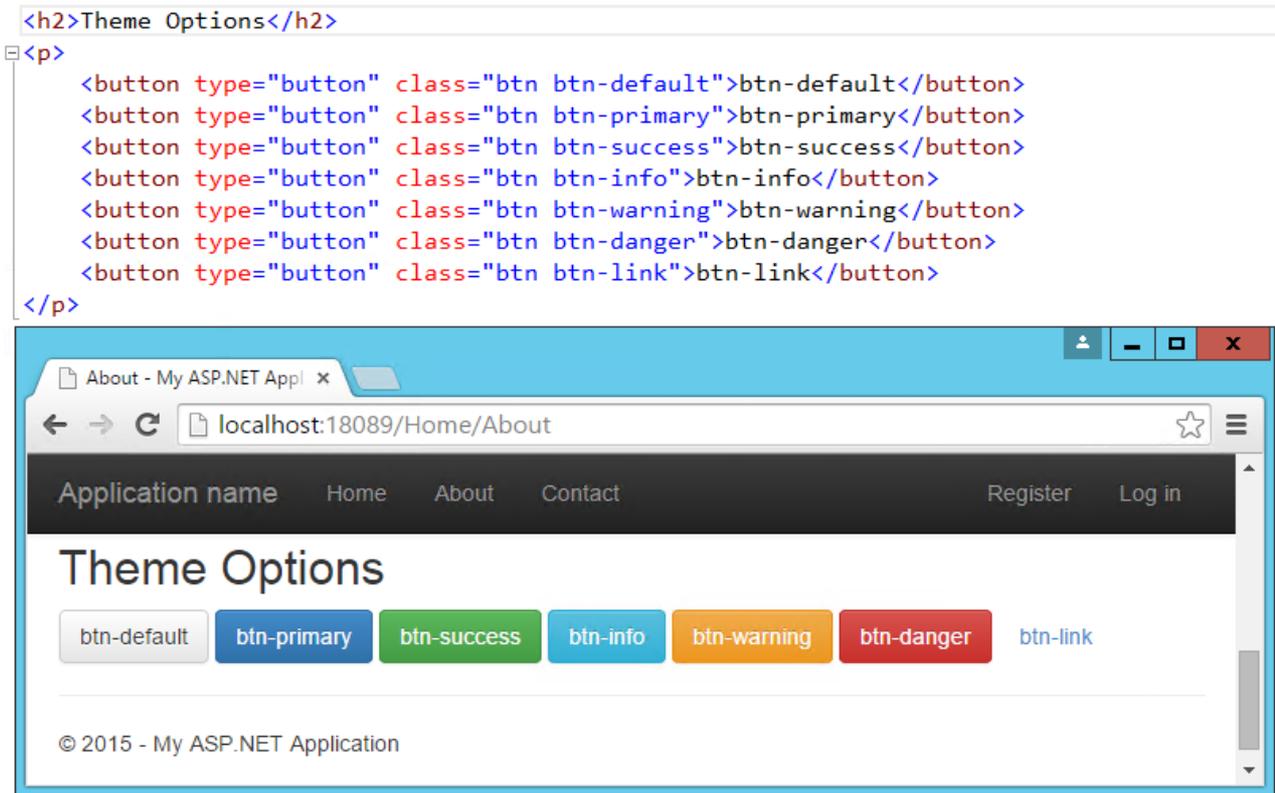
<div class="jumbotron">
  <div class="container">
    <h2>@ViewBag.Title.</h2>
    <h3>@ViewBag.Message</h3>
  </div>
</div>
<p>Use this area to provide additional information.</p>

```



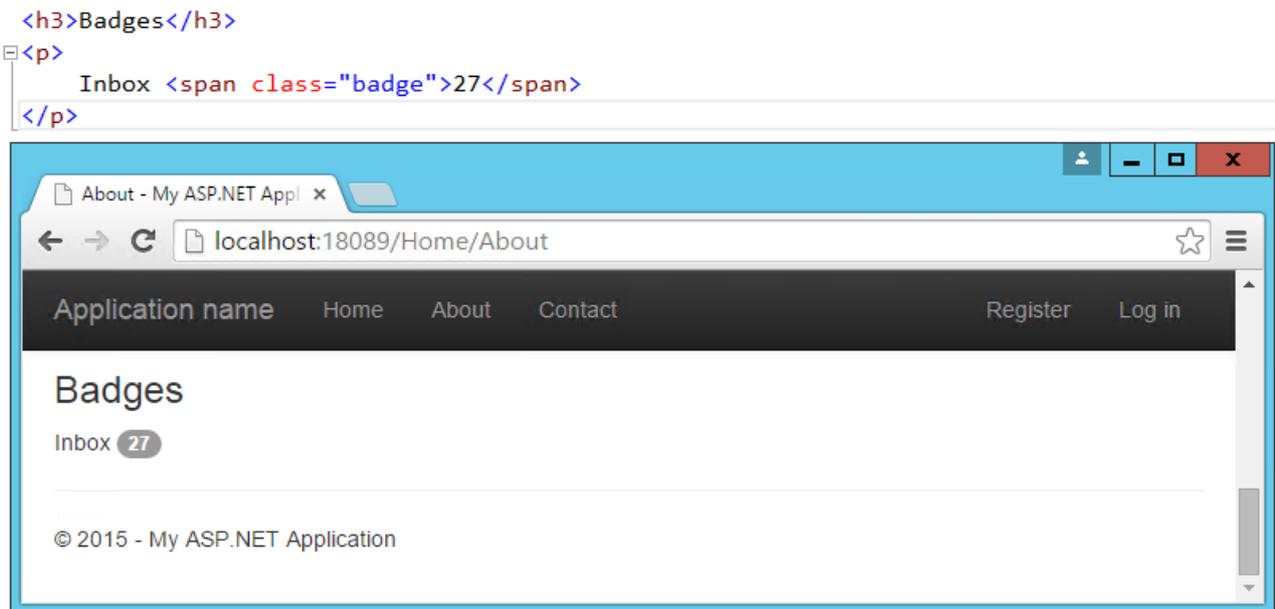
Buttons

The default button classes and their colors are shown in the figure below.



Badges

Badges refer to small, usually numeric callouts next to a navigation item. They can indicate a number of messages or notifications waiting, or the presence of updates. Specifying such badges is as simple as adding a containing the text, with a class of "badge":



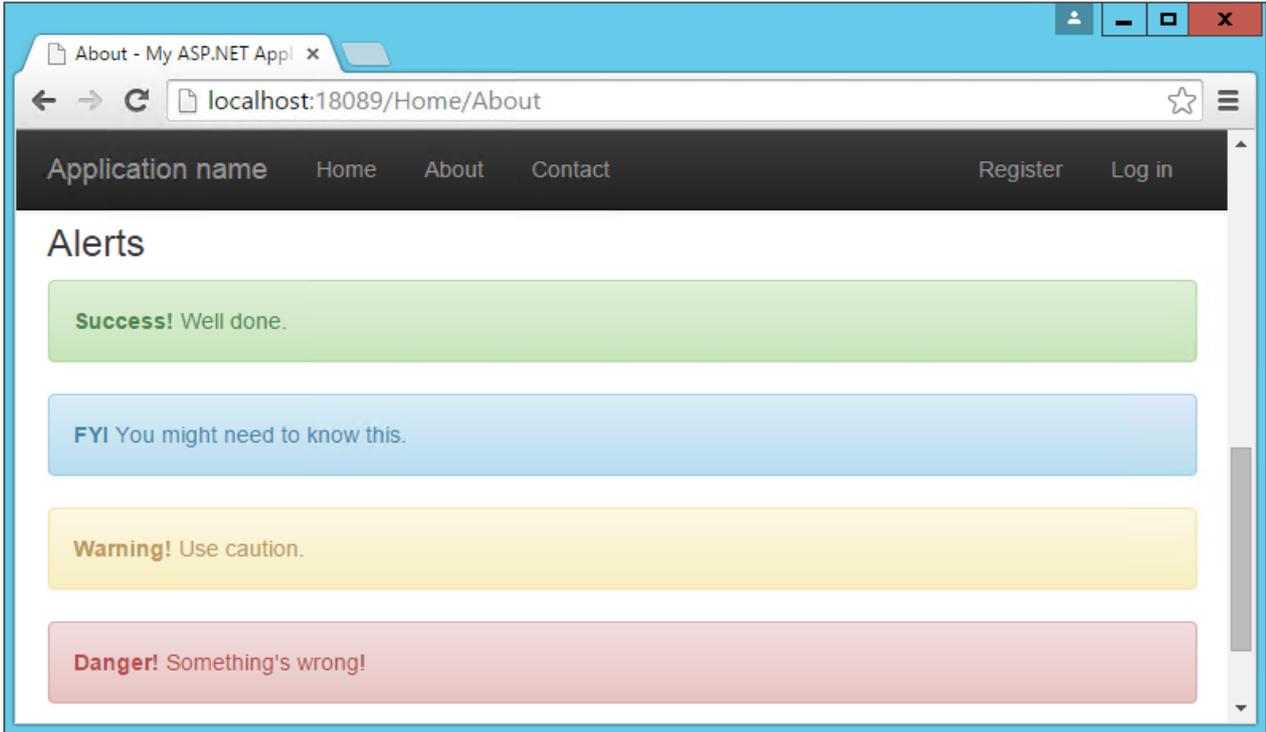
Alerts

You may need to display some kind of notification, alert, or error message to your application's users. That's where the standard alert classes are useful. There are four different severity levels with associated color schemes:

```

<h3>Alerts</h3>
<div class="alert alert-success">
  <strong>Success!</strong> Well done.
</div>
<div class="alert alert-info">
  <strong>FYI</strong> You might need to know this.
</div>
<div class="alert alert-warning">
  <strong>Warning!</strong> Use caution.
</div>
<div class="alert alert-danger">
  <strong>Danger!</strong> Something's wrong!
</div>

```



Navbars and menus

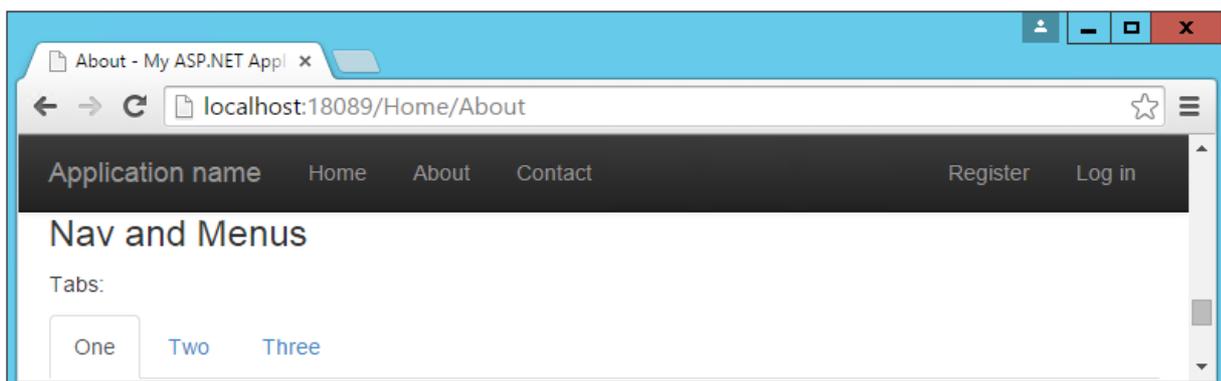
Our layout already includes a standard navbar, but the Bootstrap theme supports additional styling options. We can also easily opt to display the navbar vertically rather than horizontally if that's preferred, as well as adding sub-navigation items in flyout menus. Simple navigation menus, like tab strips, are built on top of

elements. These can be created very simply by just providing them with the CSS classes "nav" and "nav-tabs":

```

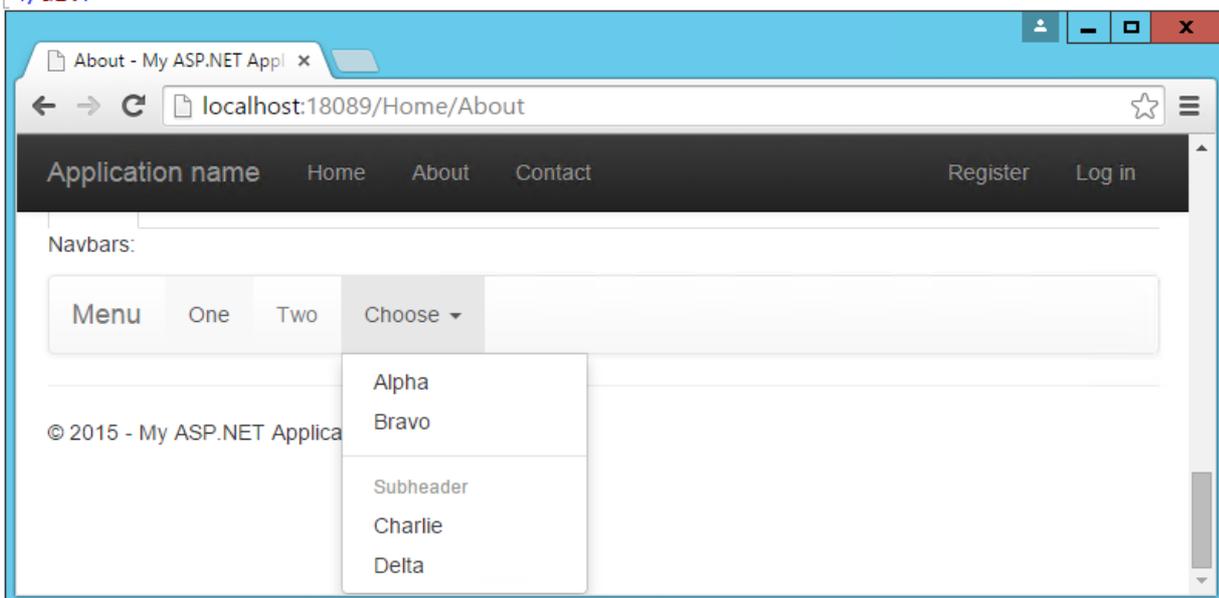
<h3>Nav and Menu</h3>
<p>Tabs:</p>
<ul class="nav nav-tabs">
  <li class="active"><a href="#">One</a></li>
  <li><a href="#">Two</a></li>
  <li><a href="#">Three</a></li>
</ul>

```



Navbars are built similarly, but are a bit more complex. They start with a `<nav>` or `<div>` with a class of "navbar", within which a container div holds the rest of the elements. Our page includes a navbar in its header already – the one shown below simply expands on this, adding support for a dropdown menu:

```
<p>Navbars:</p>
<div class="navbar navbar-default">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse"
        data-target=".navbar-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">Menu</a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li class="active"><a href="#">One</a></li>
        <li><a href="#two">Two</a></li>
        <li class="dropdown">
          <a href="#" class="dropdown-toggle" data-toggle="dropdown"
            role="button" aria-expanded="false">Choose <span class="caret"></span></a>
          <ul class="dropdown-menu" role="menu">
            <li><a href="#">Alpha</a></li>
            <li><a href="#">Bravo</a></li>
            <li class="divider"></li>
            <li class="dropdown-header">Subheader</li>
            <li><a href="#">Charlie</a></li>
            <li><a href="#">Delta</a></li>
          </ul>
        </li>
      </ul>
    </div>
  </div>
</div>
```

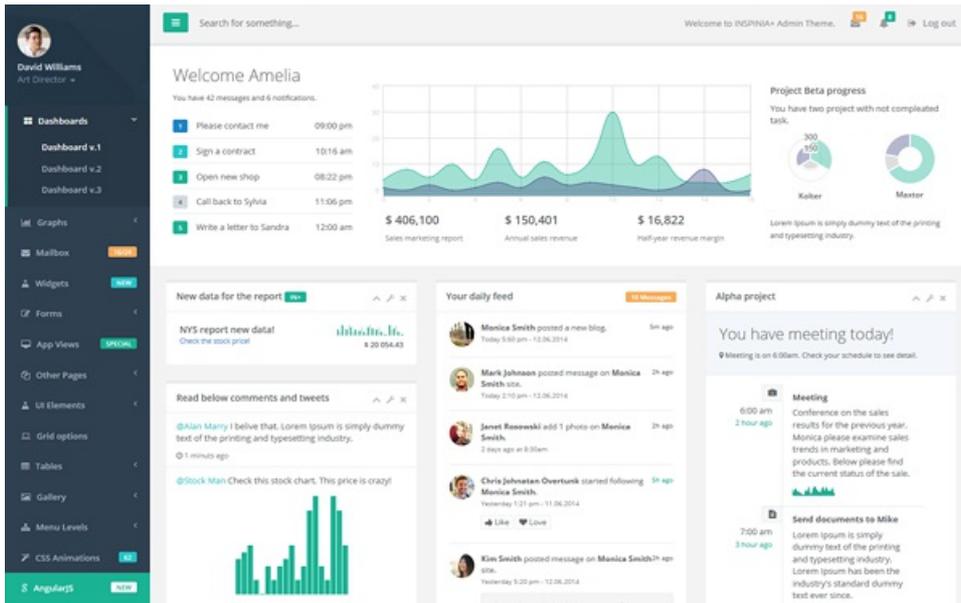


Additional elements

The default theme can also be used to present HTML tables in a nicely formatted style, including support for striped views. There are labels with styles that are similar to those of the buttons. You can create custom Dropdown menus that support additional styling options beyond the standard HTML `<select>` element, along with Navbars like the one our default starter site is already using. If you need a progress bar, there are several styles to choose from, as well as List Groups and panels that include a title and content. Explore additional options within the standard Bootstrap Theme here:

More themes

You can extend the standard Bootstrap theme by overriding some or all of its CSS, adjusting the colors and styles to suit your own application's needs. If you'd like to start from a ready-made theme, there are several theme galleries available online that specialize in Bootstrap themes, such as WrapBootstrap.com (which has a variety of commercial themes) and Bootswatch.com (which offers free themes). Some of the paid templates available provide a great deal of functionality on top of the basic Bootstrap theme, such as rich support for administrative menus, and dashboards with rich charts and gauges. An example of a popular paid template is Inspinia, currently for sale for \$18, which includes an ASP.NET MVC5 template in addition to AngularJS and static HTML versions. A sample screenshot is shown below.



If you want to change your Bootstrap theme, put the *bootstrap.css* file for the theme you want in the **wwwroot/css** folder and change the references in *_Layout.cshtml* to point it. Change the links for all environments:

```
<environment names="Development">
  <link rel="stylesheet" href="~/css/bootstrap.css" />
```

```
<environment names="Staging,Production">
  <link rel="stylesheet" href="~/css/bootstrap.min.css" />
```

If you want to build your own dashboard, you can start from the free example available here: <http://getbootstrap.com/examples/dashboard/>.

Components

In addition to those elements already discussed, Bootstrap includes support for a variety of **built-in UI components**.

Glyphicons

Bootstrap includes icon sets from Glyphicons (<http://glyphicons.com>), with over 200 icons freely available for use within your Bootstrap-enabled web application. Here's just a small sample:

glyphicon indent-right	glyphicon facetime-video	glyphicon picture	glyphicon map-marker	glyphicon adjust	glyphicon play	glyphicon stop	glyphicon share
							
glyphicon glyphicon- check	glyphicon glyphicon- move	glyphicon glyphicon- step-backward	glyphicon glyphicon-fast- backward	glyphicon glyphicon- backward	glyphicon glyphicon-play	glyphicon glyphicon- pause	glyphicon glyphicon- stop
							
glyphicon glyphicon- forward	glyphicon glyphicon-fast- forward	glyphicon glyphicon- step-forward	glyphicon glyphicon-eject	glyphicon glyphicon- chevron-left	glyphicon glyphicon- chevron-right	glyphicon glyphicon- plus-sign	glyphicon glyphicon- minus-sign
							
glyphicon glyphicon- remove-sign	glyphicon glyphicon-ok- sign	glyphicon glyphicon- question-sign	glyphicon glyphicon-info- sign	glyphicon glyphicon- screenshot	glyphicon glyphicon- remove-circle	glyphicon glyphicon-ok- circle	glyphicon glyphicon-ban- circle
							
glyphicon glyphicon- arrow-left	glyphicon glyphicon- arrow-right	glyphicon glyphicon- arrow-up	glyphicon glyphicon- arrow-down	glyphicon glyphicon- share-alt	glyphicon glyphicon- resize-full	glyphicon glyphicon- resize-small	glyphicon glyphicon- exclamation- sign
							
glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon

Input groups

Input groups allow bundling of additional text or buttons with an input element, providing the user with a more intuitive experience:

@example.com

Breadcrumbs

Breadcrumbs are a common UI component used to show a user their recent history or depth within a site's navigation hierarchy. Add them easily by applying the "breadcrumb" class to any `` list element. Include built-in support for pagination by using the "pagination" class on a `` element within a `<nav>`. Add responsive embedded slideshows and video by using `<iframe>`, `<embed>`, `<video>`, or `<object>` elements, which Bootstrap will style automatically. Specify a particular aspect ratio by using specific classes like "embed-responsive-16by9".

JavaScript support

Bootstrap's JavaScript library includes API support for the included components, allowing you to control their behavior programmatically within your application. In addition, *bootstrap.js* includes over a dozen custom jQuery plugins, providing additional features like transitions, modal dialogs, scroll detection (updating styles based on where the user has scrolled in the document), collapse behavior, carousels, and affixing menus to the window so they do not scroll off the screen. There's not sufficient room to cover all of the JavaScript add-ons built into Bootstrap – to learn more please visit <http://getbootstrap.com/javascript/>.

Summary

Bootstrap provides a web framework that can be used to quickly and productively lay out and style a wide variety of websites and applications. Its basic typography and styles provide a pleasant look and feel that can easily be manipulated through custom theme support, which can be hand-crafted or purchased commercially. It supports a host of web components that in the past would have required expensive third-party controls to accomplish, while supporting modern and open web standards.

Introduction to styling applications with Less, Sass, and Font Awesome in ASP.NET Core

9/22/2017 • 12 min to read • [Edit Online](#)

By [Steve Smith](#)

Users of web applications have increasingly high expectations when it comes to style and overall experience. Modern web applications frequently leverage rich tools and frameworks for defining and managing their look and feel in a consistent manner. Frameworks like [Bootstrap](#) can go a long way toward defining a common set of styles and layout options for web sites. However, most non-trivial sites also benefit from being able to effectively define and maintain styles and cascading style sheet (CSS) files, as well as having easy access to non-image icons that help make the site's interface more intuitive. That's where languages and tools that support [Less](#) and [Sass](#), and libraries like [Font Awesome](#), come in.

CSS preprocessor languages

Languages that are compiled into other languages, in order to improve the experience of working with the underlying language, are referred to as preprocessors. There are two popular preprocessors for CSS: Less and Sass. These preprocessors add features to CSS, such as support for variables and nested rules, which improve the maintainability of large, complex stylesheets. CSS as a language is very basic, lacking support even for something as simple as variables, and this tends to make CSS files repetitive and bloated. Adding real programming language features via preprocessors can help reduce duplication and provide better organization of styling rules. Visual Studio provides built-in support for both Less and Sass, as well as extensions that can further improve the development experience when working with these languages.

As a quick example of how preprocessors can improve readability and maintainability of style information, consider this CSS:

```
.header {
  color: black;
  font-weight: bold;
  font-size: 18px;
  font-family: Helvetica, Arial, sans-serif;
}

.small-header {
  color: black;
  font-weight: bold;
  font-size: 14px;
  font-family: Helvetica, Arial, sans-serif;
}
```

Using Less, this can be rewritten to eliminate all of the duplication, using a *mixin* (so named because it allows you to "mix in" properties from one class or rule-set into another):

```
.header {
  color: black;
  font-weight: bold;
  font-size: 18px;
  font-family: Helvetica, Arial, sans-serif;
}

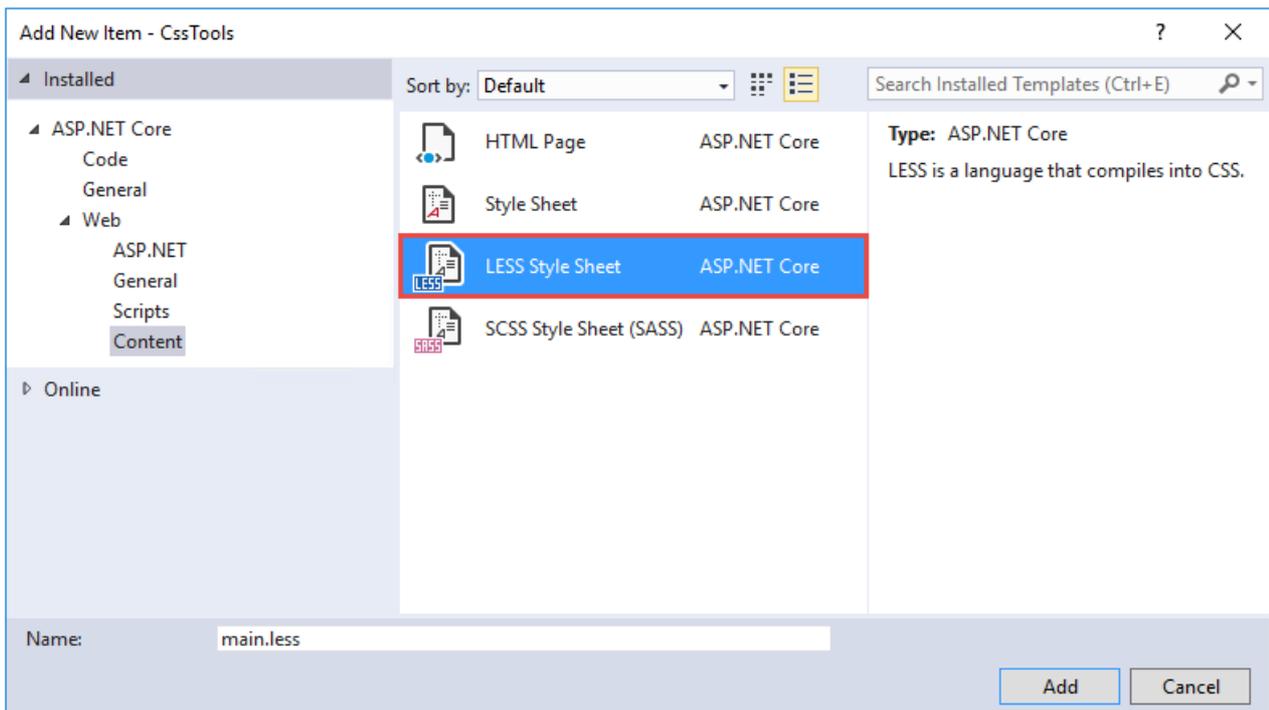
.small-header {
  .header;
  font-size: 14px;
}
```

Less

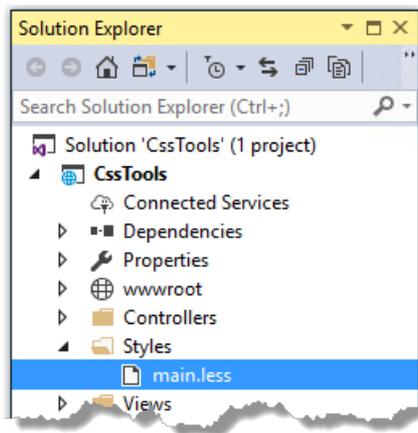
The Less CSS preprocessor runs using Node.js. To install Less, use Node Package Manager (npm) from a command prompt (-g means "global"):

```
npm install -g less
```

If you're using Visual Studio, you can get started with Less by adding one or more Less files to your project, and then configuring Gulp (or Grunt) to process them at compile-time. Add a *Styles* folder to your project, and then add a new Less file named *main.less* to this folder.



Once added, your folder structure should look something like this:



Now you can add some basic styling to the file, which will be compiled into CSS and deployed to the wwwroot folder by Gulp.

Modify *main.less* to include the following content, which creates a simple color palette from a single base color.

```
@base: #663333;
@background: spin(@base, 180);
@lighter: lighten(spin(@base, 5), 10%);
@lighter2: lighten(spin(@base, 10), 20%);
@darker: darken(spin(@base, -5), 10%);
@darker2: darken(spin(@base, -10), 20%);

body {
  background-color:@background;
}

.baseColor {color:@base}
.bgLight {color:@lighter}
.bgLight2 {color:@lighter2}
.bgDark {color:@darker}
.bgDark2 {color:@darker2}
```

`@base` and the other `@`-prefixed items are variables. Each of them represents a color. Except for `@base`, they are set using color functions: `lighten`, `darken`, and `spin`. `Lighten` and `darken` do pretty much what you would expect; `spin` adjusts the hue of a color by a number of degrees (around the color wheel). The Less processor is smart enough to ignore variables that aren't used, so to demonstrate how these variables work, we need to use them somewhere. The classes `.baseColor`, etc. will demonstrate the calculated values of each of the variables in the CSS file that is produced.

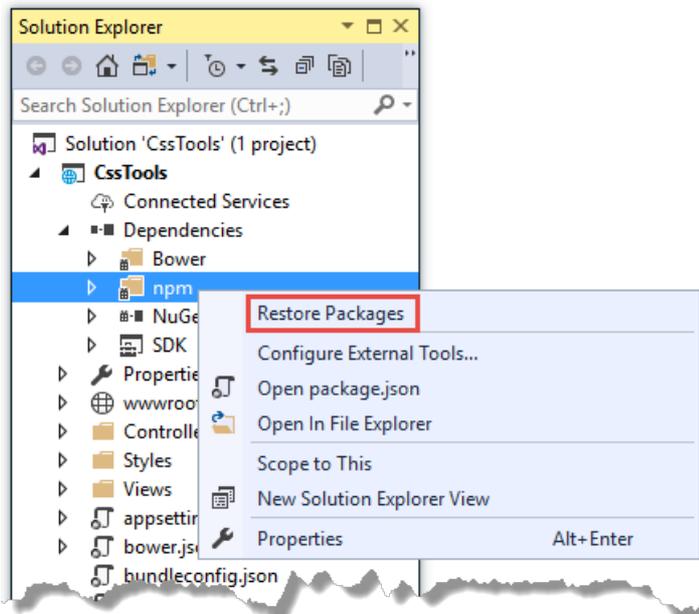
Getting started

Create an **npm Configuration File** (*package.json*) in your project folder and edit it to reference `gulp` and `gulp-less`:

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-less": "3.3.0"
  }
}
```

Install the dependencies either at a command prompt in your project folder, or in Visual Studio **Solution Explorer** (**Dependencies** > **npm** > **Restore packages**).

```
npm install
```



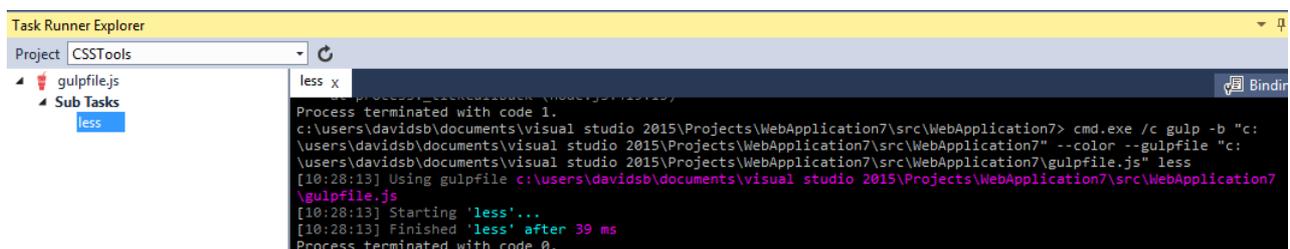
In the project folder, create a **Gulp Configuration File** (*gulpfile.js*) to define the automated process. Add a variable at the top of the file to represent Less, and a task to run Less:

```
var gulp = require("gulp"),
    fs = require("fs"),
    less = require("gulp-less");

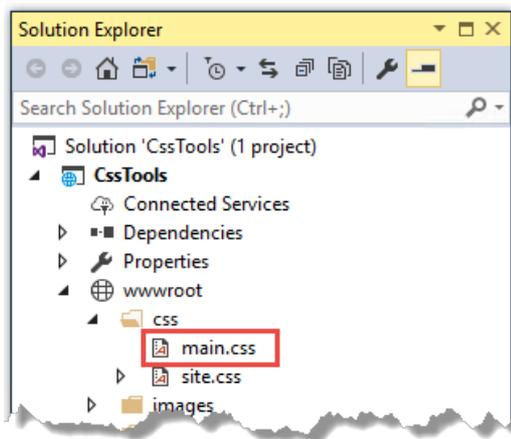
gulp.task("less", function () {
    return gulp.src('Styles/main.less')
        .pipe(less())
        .pipe(gulp.dest('wwwroot/css'));
});
```

Open the **Task Runner Explorer** (**View > Other Windows > Task Runner Explorer**). Among the tasks, you should see a new task named `less`. You might have to refresh the window.

Run the `less` task, and you see output similar to what is shown here:



The `wwwroot/css` folder now contains a new file, *main.css*:



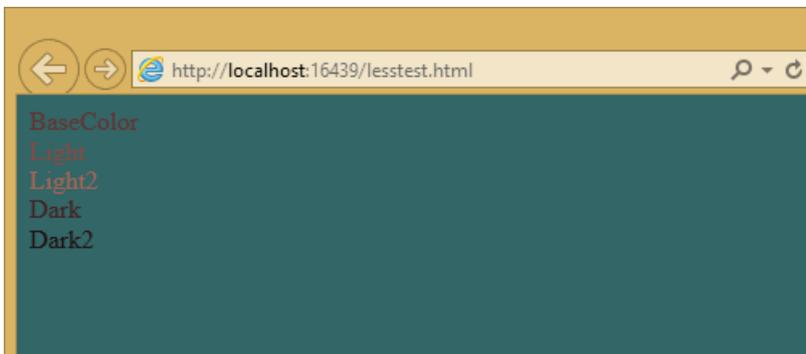
Open *main.css* and you see something like the following:

```
body {
  background-color: #336666;
}
.baseColor {
  color: #663333;
}
.bgLight {
  color: #884a44;
}
.bgLight2 {
  color: #aa6355;
}
.bgDark {
  color: #442225;
}
.bgDark2 {
  color: #221114;
}
```

Add a simple HTML page to the *wwwroot* folder, and reference *main.css* to see the color palette in action.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <link href="css/main.css" rel="stylesheet" />
  <title></title>
</head>
<body>
  <div>
    <div class="baseColor">BaseColor</div>
    <div class="bgLight">Light</div>
    <div class="bgLight2">Light2</div>
    <div class="bgDark">Dark</div>
    <div class="bgDark2">Dark2</div>
  </div>
</body>
</html>
```

You can see that the 180 degree spin on `@base` used to produce `@background` resulted in the color wheel opposing color of `@base`:



Less also provides support for nested rules, as well as nested media queries. For example, defining nested hierarchies like menus can result in verbose CSS rules like these:

```
nav {
  height: 40px;
  width: 100%;
}
nav li {
  height: 38px;
  width: 100px;
}
nav li a:link {
  color: #000;
  text-decoration: none;
}
nav li a:visited {
  text-decoration: none;
  color: #CC3333;
}
nav li a:hover {
  text-decoration: underline;
  font-weight: bold;
}
nav li a:active {
  text-decoration: underline;
}
```

Ideally all of the related style rules will be placed together within the CSS file, but in practice there is nothing enforcing this rule except convention and perhaps block comments.

Defining these same rules using Less looks like this:

```
nav {
  height: 40px;
  width: 100%;
  li {
    height: 38px;
    width: 100px;
    a {
      color: #000;
      &:link { text-decoration:none}
      &:visited { color: #CC3333; text-decoration:none}
      &:hover { text-decoration:underline; font-weight:bold}
      &:active {text-decoration:underline}
    }
  }
}
```

Note that in this case, all of the subordinate elements of `nav` are contained within its scope. There is no longer any repetition of parent elements (`nav`, `li`, `a`), and the total line count has dropped as well (though some of that is a result of putting values on the same lines in the second example). It can be very helpful, organizationally, to see all

of the rules for a given UI element within an explicitly bounded scope, in this case set off from the rest of the file by curly braces.

The `&` syntax is a Less selector feature, with `&` representing the current selector parent. So, within the `a {...}` block, `&` represents an `a` tag, and thus `&:link` is equivalent to `a:link`.

Media queries, extremely useful in creating responsive designs, can also contribute heavily to repetition and complexity in CSS. Less allows media queries to be nested within classes, so that the entire class definition doesn't need to be repeated within different top-level `@media` elements. For example, here is CSS for a responsive menu:

```
.navigation {
  margin-top: 30%;
  width: 100%;
}
@media screen and (min-width: 40em) {
  .navigation {
    margin: 0;
  }
}
@media screen and (min-width: 62em) {
  .navigation {
    width: 960px;
    margin: 0;
  }
}
```

This can be better defined in Less as:

```
.navigation {
  margin-top: 30%;
  width: 100%;
  @media screen and (min-width: 40em) {
    margin: 0;
  }
  @media screen and (min-width: 62em) {
    width: 960px;
    margin: 0;
  }
}
```

Another feature of Less that we have already seen is its support for mathematical operations, allowing style attributes to be constructed from pre-defined variables. This makes updating related styles much easier, since the base variable can be modified and all dependent values change automatically.

CSS files, especially for large sites (and especially if media queries are being used), tend to get quite large over time, making working with them unwieldy. Less files can be defined separately, then pulled together using `@import` directives. Less can also be used to import individual CSS files, as well, if desired.

Mixins can accept parameters, and Less supports conditional logic in the form of mixin guards, which provide a declarative way to define when certain mixins take effect. A common use for mixin guards is to adjust colors based on how light or dark the source color is. Given a mixin that accepts a parameter for color, a mixin guard can be used to modify the mixin based on that color:

```
.box (@color) when (lightness(@color) >= 50%) {
  background-color: #000;
}
.box (@color) when (lightness(@color) < 50%) {
  background-color: #FFF;
}
.box (@color) {
  color: @color;
}

.feature {
  .box (@base);
}
```

Given our current `@base` value of `#663333`, this Less script will produce the following CSS:

```
.feature {
  background-color: #FFF;
  color: #663333;
}
```

Less provides a number of additional features, but this should give you some idea of the power of this preprocessing language.

Sass

Sass is similar to Less, providing support for many of the same features, but with slightly different syntax. It is built using Ruby, rather than JavaScript, and so has different setup requirements. The original Sass language did not use curly braces or semicolons, but instead defined scope using white space and indentation. In version 3 of Sass, a new syntax was introduced, **SCSS** ("Sassy CSS"). SCSS is similar to CSS in that it ignores indentation levels and whitespace, and instead uses semicolons and curly braces.

To install Sass, typically you would first install Ruby (pre-installed on Mac), and then run:

```
gem install sass
```

However, if you're running Visual Studio, you can get started with Sass in much the same way as you would with Less. Open *package.json* and add the "gulp-sass" package to `devDependencies`:

```
"devDependencies": {
  "gulp": "3.9.1",
  "gulp-less": "3.3.0",
  "gulp-sass": "3.1.0"
}
```

Next, modify *gulpfile.js* to add a sass variable and a task to compile your Sass files and place the results in the `wwwroot` folder:

```

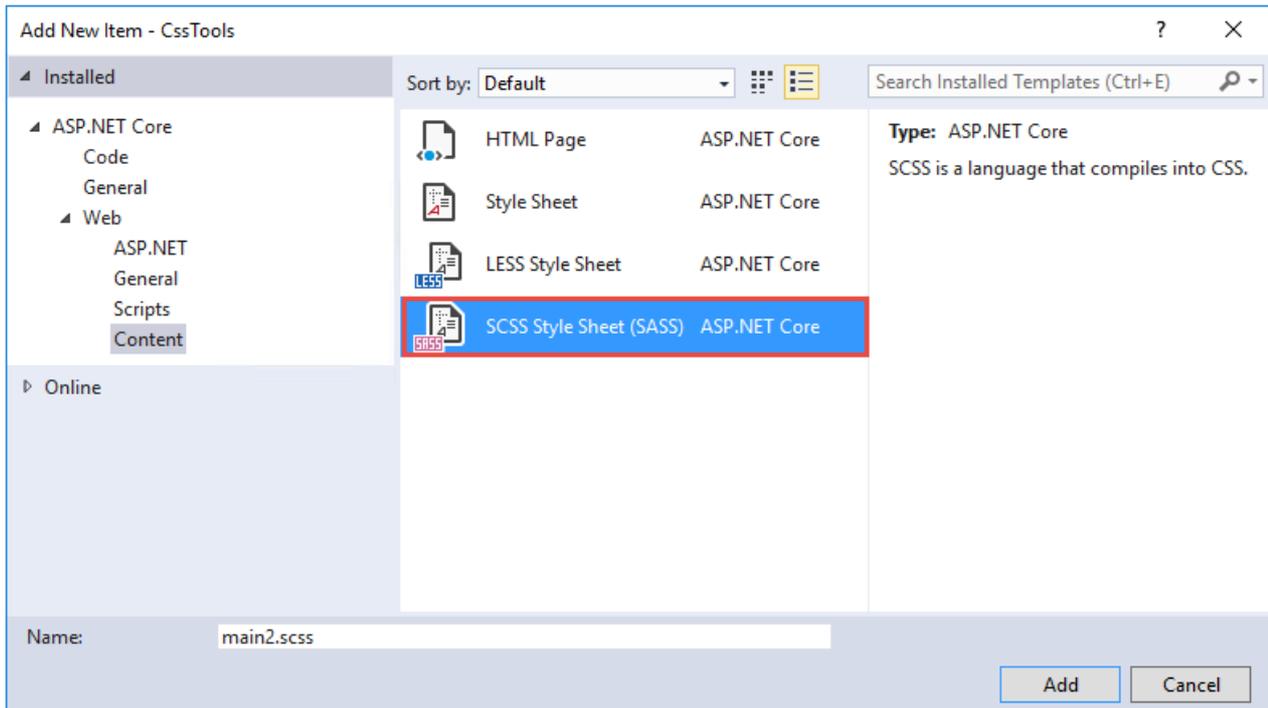
var gulp = require("gulp"),
    fs = require("fs"),
    less = require("gulp-less"),
    sass = require("gulp-sass");

// other content removed

gulp.task("sass", function () {
    return gulp.src('Styles/main2.scss')
        .pipe(sass())
        .pipe(gulp.dest('wwwroot/css'));
});

```

Now you can add the Sass file *main2.scss* to the *Styles* folder in the root of the project:



Open *main2.scss* and add the following:

```

$base: #CC0000;
body {
    background-color: $base;
}

```

Save all of your files. Now when you refresh **Task Runner Explorer**, you see a `sass` task. Run it, and look in the */wwwroot/css* folder. There is now a *main2.css* file, with these contents:

```

body {
    background-color: #CC0000;
}

```

Sass supports nesting in much the same way that Less does, providing similar benefits. Files can be split up by function and included using the `@import` directive:

```
@import 'anotherfile';
```

Sass supports mixins as well, using the `@mixin` keyword to define them and `@include` to include them, as in this

example from sass-lang.com:

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  -ms-border-radius: $radius;
  border-radius: $radius;
}

.box { @include border-radius(10px); }
```

In addition to mixins, Sass also supports the concept of inheritance, allowing one class to extend another. It's conceptually similar to a mixin, but results in less CSS code. It's accomplished using the `@extend` keyword. To try out mixins, add the following to your *main2.scss* file:

```
@mixin alert {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
}

.success {
  @include alert;
  border-color: green;
}

.error {
  @include alert;
  color: red;
  border-color: red;
  font-weight: bold;
}
```

Examine the output in *main2.css* after running the `sass` task in **Task Runner Explorer**:

```
.success {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
  border-color: green;
}

.error {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
  color: red;
  border-color: red;
  font-weight: bold;
}
```

Notice that all of the common properties of the alert mixin are repeated in each class. The mixin did a good job of helping eliminate duplication at development time, but it's still creating CSS with a lot of duplication in it, resulting in larger than necessary CSS files - a potential performance issue.

Now replace the alert mixin with a `.alert` class, and change `@include` to `@extend` (remembering to extend `.alert`, not `alert`):

```
.alert {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
}

.success {
  @extend .alert;
  border-color: green;
}

.error {
  @extend .alert;
  color: red;
  border-color: red;
  font-weight:bold;
}
```

Run Sass once more, and examine the resulting CSS:

```
.alert, .success, .error {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
}

.success {
  border-color: green;
}

.error {
  color: red;
  border-color: red;
  font-weight: bold;
}
```

Now the properties are defined only as many times as needed, and better CSS is generated.

Sass also includes functions and conditional logic operations, similar to Less. In fact, the two languages' capabilities are very similar.

Less or Sass?

There is still no consensus as to whether it's generally better to use Less or Sass (or even whether to prefer the original Sass or the newer SCSS syntax within Sass). Probably the most important decision is to **use one of these tools**, as opposed to just hand-coding your CSS files. Once you've made that decision, both Less and Sass are good choices.

Font Awesome

In addition to CSS preprocessors, another great resource for styling modern web applications is Font Awesome. Font Awesome is a toolkit that provides over 500 scalable vector icons that can be freely used in your web applications. It was originally designed to work with Bootstrap, but it has no dependency on that framework or on any JavaScript libraries.

The easiest way to get started with Font Awesome is to add a reference to it, using its public content delivery network (CDN) location:

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css">
```

You can also add it to your Visual Studio project by adding it to the "dependencies" in *bower.json*:

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "bootstrap": "3.0.0",
    "jquery": "1.10.2",
    "jquery-validation": "1.11.1",
    "jquery-validation-unobtrusive": "3.2.2",
    "hammer.js": "2.0.4",
    "bootstrap-touch-carousel": "0.8.0",
    "Font-Awesome": "4.3.0"
  }
}
```

Once you have a reference to Font Awesome on a page, you can add icons to your application by applying Font Awesome classes, typically prefixed with "fa-", to your inline HTML elements (such as `` or `<i>`). For example, you can add icons to simple lists and menus using code like this:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
  <link href="lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</head>
<body>
  <ul class="fa-ul">
    <li><i class="fa fa-li fa-home"></i> Home</li>
    <li><i class="fa fa-li fa-cog"></i> Settings</li>
  </ul>
</body>
</html>
```

This produces the following in the browser - note the icon beside each item:



You can view a complete list of the available icons here:

<http://fontawesome.io/icons/>

Summary

Modern web applications increasingly demand responsive, fluid designs that are clean, intuitive, and easy to use from a variety of devices. Managing the complexity of the CSS stylesheets required to achieve these goals is best done using a preprocessor like Less or Sass. In addition, toolkits like Font Awesome quickly provide well-known icons to textual navigation menus and buttons, improving the overall user experience of your application.

Bundling and minification

1/10/2018 • 10 min to read • [Edit Online](#)

By [Scott Addie](#)

This article explains the benefits of applying bundling and minification, including how these features can be used with ASP.NET Core web apps.

What is bundling and minification?

Bundling and minification are two distinct performance optimizations you can apply in a web app. Used together, bundling and minification improve performance by reducing the number of server requests and reducing the size of the requested static assets.

Bundling and minification primarily improve the first page request load time. Once a web page has been requested, the browser caches the static assets (JavaScript, CSS, and images). Consequently, bundling and minification don't improve performance when requesting the same page, or pages, on the same site requesting the same assets. If you don't set the expires header correctly on your assets, and if you don't use bundling and minification, the browser's freshness heuristics mark the assets stale after a few days. Additionally, the browser requires a validation request for each asset. In this case, bundling and minification provide a performance improvement even after the first page request.

Bundling

Bundling combines multiple files into a single file. Bundling reduces the number of server requests which are necessary to render a web asset, such as a web page. You can create any number of individual bundles specifically for CSS, JavaScript, etc. Fewer files means fewer HTTP requests from the browser to the server or from the service providing your application. This results in improved first page load performance.

Minification

Minification removes unnecessary characters from code without altering functionality. The result is a significant size reduction in requested assets (such as CSS, images, and JavaScript files). Common side effects of minification include shortening variable names to one character and removing comments and unnecessary whitespace.

Consider the following JavaScript function:

```
AddAltToImg = function (imageTagAndImageID, imageContext) {
    ///
```

Minification reduces the function to the following:

```
AddAltToImg=function(n,t){var i=$(n,t);i.attr("alt",i.attr("id").replace(/ID/,""))};
```

In addition to removing the comments and unnecessary whitespace, the following parameter and variable names were renamed as follows:

ORIGINAL	RENAMED
imageTagAndImageID	t
imageContext	a
imageElement	r

Impact of bundling and minification

The following table outlines differences between individually loading assets and using bundling and minification:

ACTION	WITH B/M	WITHOUT B/M	CHANGE
File Requests	7	18	157%
KB Transferred	156	264.68	70%
Load Time (ms)	885	2360	167%

Browsers are fairly verbose with regard to HTTP request headers. The total bytes sent metric saw a significant reduction when bundling. The load time shows a significant improvement, however this example ran locally. Greater performance gains are realized when using bundling and minification with assets transferred over a network.

Choose a bundling and minification strategy

The MVC and Razor Pages project templates provide an out-of-the-box solution for bundling and minification consisting of a JSON configuration file. Third-party tools, such as the [Gulp](#) and [Grunt](#) task runners, accomplish the same tasks with a bit more complexity. A third-party tool is a great fit when your development workflow requires processing beyond bundling and minification—such as linting and image optimization. By using design-time bundling and minification, the minified files are created prior to the app's deployment. Bundling and minifying before deployment provides the advantage of reduced server load. However, it's important to recognize that design-time bundling and minification increases build complexity and only works with static files.

Configure bundling and minification

The MVC and Razor Pages project templates provide a *bundleconfig.json* configuration file which defines the options for each bundle. By default, a single bundle configuration is defined for the custom JavaScript (*wwwroot/js/site.js*) and stylesheet (*wwwroot/css/site.css*) files:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

Configuration options include:

- `outputFileName`: The name of the bundle file to output. Can contain a relative path from the `bundleconfig.json` file. **required**
- `inputFiles`: An array of files to bundle together. These are relative paths to the configuration file. **optional**, *an empty value results in an empty output file. [globbing](#) patterns are supported.
- `minify`: The minification options for the output type. **optional**, *default - `minify: { enabled: true }`*
 - Configuration options are available per output file type.
 - [CSS Minifier](#)
 - [JavaScript Minifier](#)
 - [HTML Minifier](#)
- `includeInProject`: Flag indicating whether to add generated files to project file. **optional**, *default - false*
- `sourceMap`: Flag indicating whether to generate a source map for the bundled file. **optional**, *default - false*
- `sourceMapRootPath`: The root path for storing the generated source map file.

Build-time execution of bundling and minification

The [BuildBundlerMinifier](#) NuGet package enables the execution of bundling and minification at build time. The package injects [MSBuild Targets](#) which run at build and clean time. The `bundleconfig.json` file is analyzed by the build process to produce the output files based on the defined configuration.

NOTE

BuildBundlerMinifier belongs to a community-driven project on GitHub for which Microsoft provides no support. Issues should be filed [here](#).

- [Visual Studio](#)
- [.NET Core CLI](#)

Add the `BuildBundlerMinifier` package to your project.

Build the project. The following appears in the Output window:

```
1>----- Build started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU -----
1>
1>Bundler: Begin processing bundleconfig.json
1> Minified wwwroot/css/site.min.css
1> Minified wwwroot/js/site.min.js
1>Bundler: Done processing bundleconfig.json
1>BuildBundlerMinifierApp -> C:\BuildBundlerMinifierApp\bin\Debug\netcoreapp2.0\BuildBundlerMinifierApp.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

Clean the project. The following appears in the Output window:

```
1>----- Clean started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU -----
1>
1>Bundler: Cleaning output from bundleconfig.json
1>Bundler: Done cleaning output file from bundleconfig.json
===== Clean: 1 succeeded, 0 failed, 0 skipped =====
```

Ad hoc execution of bundling and minification

It's possible to run the bundling and minification tasks on an ad hoc basis, without building the project. Add the [BundlerMinifier.Core](#) NuGet package to your project:

```
<DotNetCliToolReference Include="BundlerMinifier.Core" Version="2.6.362" />
```

NOTE

BundlerMinifier.Core belongs to a community-driven project on GitHub for which Microsoft provides no support. Issues should be filed [here](#).

This package extends the .NET Core CLI to include the *dotnet-bundle* tool. The following command can be executed in the Package Manager Console (PMC) window or in a command shell:

```
dotnet bundle
```

IMPORTANT

NuGet Package Manager adds dependencies to the *.csproj file as `<PackageReference />` nodes. The `dotnet bundle` command is registered with the .NET Core CLI only when a `<DotNetCliToolReference />` node is used. Modify the *.csproj file accordingly.

Add files to workflow

Consider an example in which an additional *custom.css* file is added resembling the following:

```
.about, [role=main], [role=complementary] {
    margin-top: 60px;
}

footer {
    margin-top: 10px;
}
```

To minify *custom.css* and bundle it with *site.css* into a *site.min.css* file, add the relative path to *bundleconfig.json*:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css",
      "wwwroot/css/custom.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

NOTE

Alternatively, the following globbing pattern could be used:

```
"inputFiles": ["wwwroot/**/*.css!(*.min.css)"]
```

This globbing pattern matches all CSS files and excludes the minified file pattern.

Build the application. Open *site.min.css* and notice the content of *custom.css* is appended to the end of the file.

Environment-based bundling and minification

As a best practice, the bundled and minified files of your app should be used in a production environment. During development, the original files make for easier debugging of the app.

Specify which files to include in your pages by using the [Environment Tag Helper](#) in your views. The Environment Tag Helper only renders its contents when running in specific [environments](#).

The following `environment` tag renders the unprocessed CSS files when running in the `Development` environment:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

The following `environment` tag renders the bundled and minified CSS files when running in an environment other than `Development`. For example, running in `Production` or `Staging` triggers the rendering of these stylesheets:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
<environment exclude="Development">
  <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

Consume bundleconfig.json from Gulp

There are cases in which an app's bundling and minification workflow requires additional processing. Examples include image optimization, cache busting, and CDN asset processing. To satisfy these requirements, you can convert the bundling and minification workflow to use Gulp.

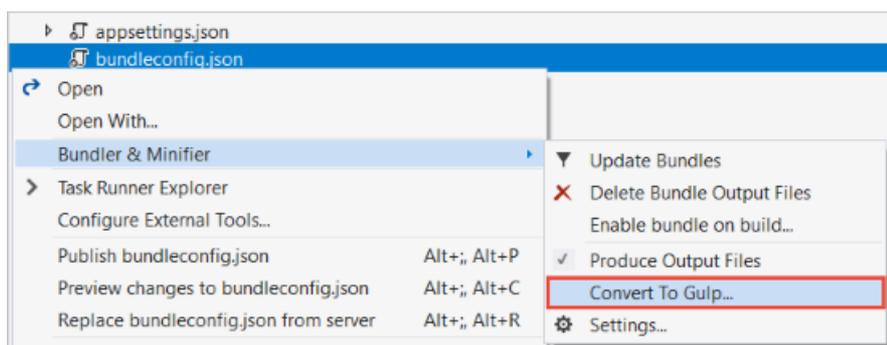
Use the Bundler & Minifier extension

The Visual Studio [Bundler & Minifier](#) extension handles the conversion to Gulp.

NOTE

The Bundler & Minifier extension belongs to a community-driven project on GitHub for which Microsoft provides no support. Issues should be filed [here](#).

Right-click the *bundleconfig.json* file in Solution Explorer and select **Bundler & Minifier > Convert To Gulp...**:



The *gulpfile.js* and *package.json* files are added to the project. The supporting [npm](#) packages listed in the *package.json* file's `devDependencies` section are installed.

Run the following command in the PMC window to install the Gulp CLI as a global dependency:

```
npm i -g gulp-cli
```

The *gulpfile.js* file reads the *bundleconfig.json* file for the inputs, outputs, and settings.

```
"use strict";

var gulp = require("gulp"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    htmlmin = require("gulp-htmlmin"),
    uglify = require("gulp-uglify"),
    merge = require("merge-stream"),
    del = require("del"),
    bundleconfig = require("./bundleconfig.json");

// Code omitted for brevity
```

Convert manually

If Visual Studio and/or the Bundler & Minifier extension aren't available, convert manually.

Add a *package.json* file, with the following `devDependencies`, to the project root:

```
"devDependencies": {
  "del": "^3.0.0",
  "gulp": "^3.9.1",
  "gulp-concat": "^2.6.1",
  "gulp-cssmin": "^0.2.0",
  "gulp-htmlmin": "^3.0.0",
  "gulp-uglify": "^3.0.0",
  "merge-stream": "^1.0.1"
}
```

Install the dependencies by running the following command at the same level as *package.json*:

```
npm i
```

Install the Gulp CLI as a global dependency:

```
npm i -g gulp-cli
```

Copy the *gulpfile.js* file below to the project root:

```
"use strict";

var gulp = require("gulp"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    htmlmin = require("gulp-htmlmin"),
    uglify = require("gulp-uglify"),
    merge = require("merge-stream"),
    del = require("del"),
    bundleconfig = require("./bundleconfig.json");

var regex = {
  css: /\.css$/,
  html: /\.html|htm$/,
  js: /\.js$/
};

gulp.task("min", ["min:js", "min:css", "min:html"]);

gulp.task("min:js", function () {
  var tasks = getBundles(regex.js).map(function (bundle) {
    return gulp.src(bundle.inputFiles, { base: "." })
      .pipe(concat(bundle.outputFileName))
      .pipe(uglify())
      .pipe(gulp.dest("."));
  });
  return merge(tasks);
});

gulp.task("min:css", function () {
  var tasks = getBundles(regex.css).map(function (bundle) {
    return gulp.src(bundle.inputFiles, { base: "." })
      .pipe(concat(bundle.outputFileName))
      .pipe(cssmin())
      .pipe(gulp.dest("."));
  });
  return merge(tasks);
});
```

```

});

gulp.task("min:html", function () {
  var tasks = getBundles(regex.html).map(function (bundle) {
    return gulp.src(bundle.inputFiles, { base: "." })
      .pipe(concat(bundle.outputFileName))
      .pipe(htmlmin({ collapseWhitespace: true, minifyCSS: true, minifyJS: true }))
      .pipe(gulp.dest("."));
  });
  return merge(tasks);
});

gulp.task("clean", function () {
  var files = bundleconfig.map(function (bundle) {
    return bundle.outputFileName;
  });

  return del(files);
});

gulp.task("watch", function () {
  getBundles(regex.js).forEach(function (bundle) {
    gulp.watch(bundle.inputFiles, ["min:js"]);
  });

  getBundles(regex.css).forEach(function (bundle) {
    gulp.watch(bundle.inputFiles, ["min:css"]);
  });

  getBundles(regex.html).forEach(function (bundle) {
    gulp.watch(bundle.inputFiles, ["min:html"]);
  });
});

function getBundles(regexPattern) {
  return bundleconfig.filter(function (bundle) {
    return regexPattern.test(bundle.outputFileName);
  });
}

```

Run Gulp tasks

To trigger the Gulp minification task before the project builds in Visual Studio, add the following [MSBuild Target](#) to the *.csproj file:

```

<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp min" />
</Target>

```

In this example, any tasks defined within the `MyPreCompileTarget` target run before the predefined `Build` target. Output similar to the following appears in Visual Studio's Output window:

```

1>----- Build started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU -----
1>BuildBundlerMinifierApp -> C:\BuildBundlerMinifierApp\bin\Debug\netcoreapp2.0\BuildBundlerMinifierApp.dll
1>[14:17:49] Using gulpfile C:\BuildBundlerMinifierApp\gulpfile.js
1>[14:17:49] Starting 'min:js'...
1>[14:17:49] Starting 'min:css'...
1>[14:17:49] Starting 'min:html'...
1>[14:17:49] Finished 'min:js' after 83 ms
1>[14:17:49] Finished 'min:css' after 88 ms
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====

```

Alternatively, Visual Studio's Task Runner Explorer may be used to bind Gulp tasks to specific Visual Studio events.

See [Running default tasks](#) for instructions on doing that.

Additional resources

- [Using Gulp](#)
- [Using Grunt](#)
- [Working with Multiple Environments](#)
- [Tag Helpers](#)

Browser Link in ASP.NET Core

11/3/2017 • 3 min to read • [Edit Online](#)

By [Nicolò Carandini](#), [Mike Wasson](#), and [Tom Dykstra](#)

Browser Link is a feature in Visual Studio that creates a communication channel between the development environment and one or more web browsers. You can use Browser Link to refresh your web application in several browsers at once, which is useful for cross-browser testing.

Browser Link setup

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The ASP.NET Core 2.x **Web Application**, **Empty**, and **Web API** template projects use the [Microsoft.AspNetCore.All](#) meta-package, which contains a package reference for [Microsoft.VisualStudio.Web.BrowserLink](#). Therefore, using the `Microsoft.AspNetCore.All` meta-package requires no further action to make Browser Link available for use.

Configuration

In the `Configure` method of the `Startup.cs` file:

```
app.UseBrowserLink();
```

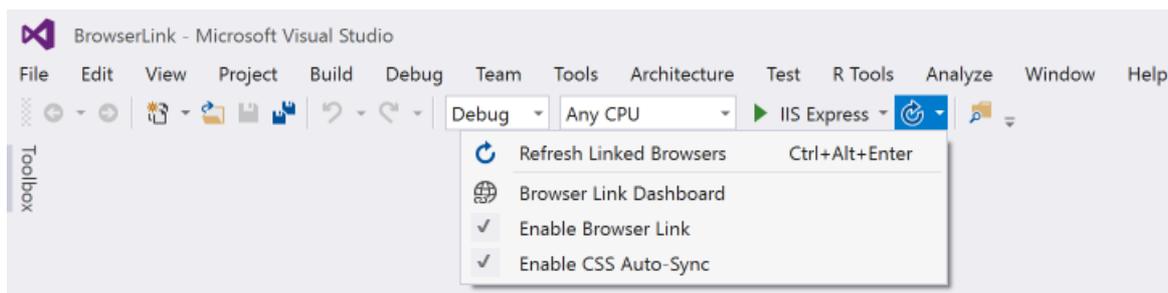
Usually the code is inside an `if` block that only enables Browser Link in the Development environment, as shown here:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseBrowserLink();
}
```

For more information, see [Working with Multiple Environments](#).

How to use Browser Link

When you have an ASP.NET Core project open, Visual Studio shows the Browser Link toolbar control next to the **Debug Target** toolbar control:



From the Browser Link toolbar control, you can:

- Refresh the web application in several browsers at once.

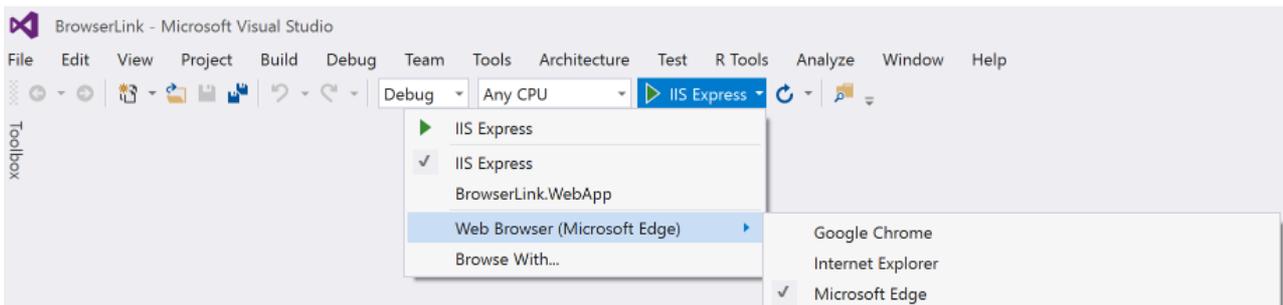
- Open the **Browser Link Dashboard**.
- Enable or disable **Browser Link**. Note: Browser Link is disabled by default in Visual Studio 2017 (15.3).
- Enable or disable [CSS Auto-Sync](#).

NOTE

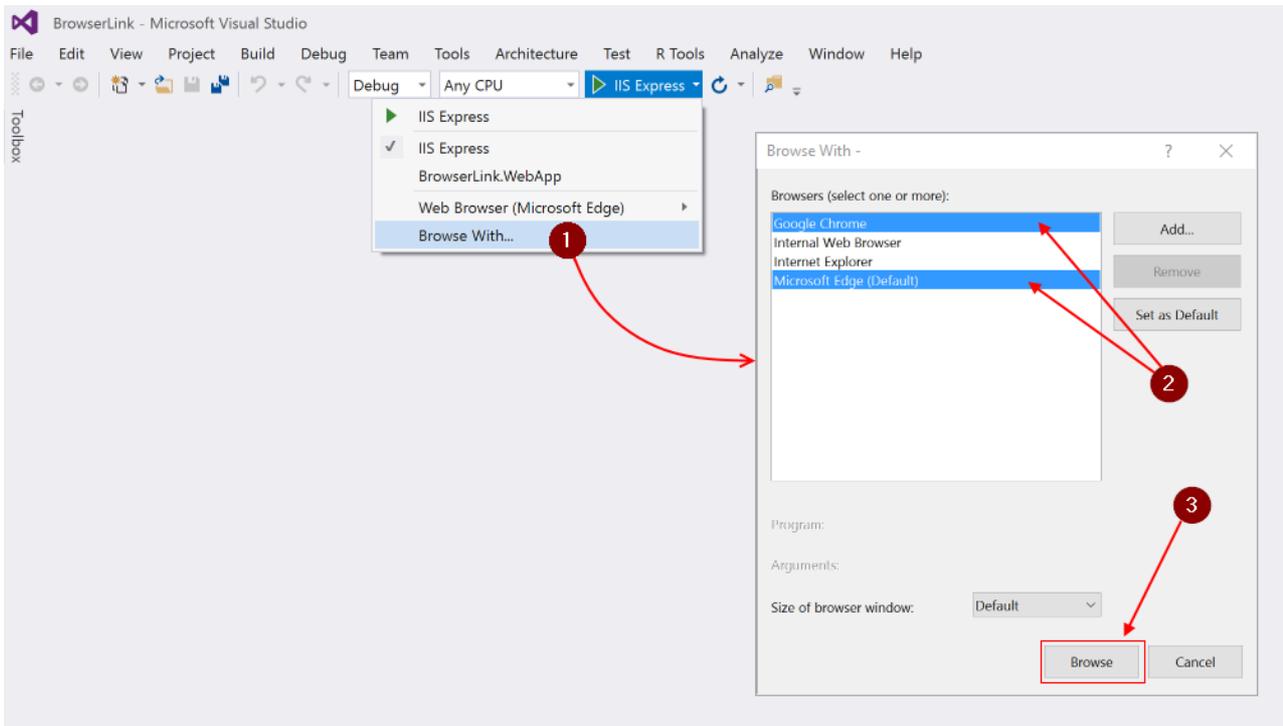
Some Visual Studio plug-ins, most notably *Web Extension Pack 2015* and *Web Extension Pack 2017*, offer extended functionality for Browser Link, but some of the additional features don't work with ASP.NET Core projects.

Refresh the web application in several browsers at once

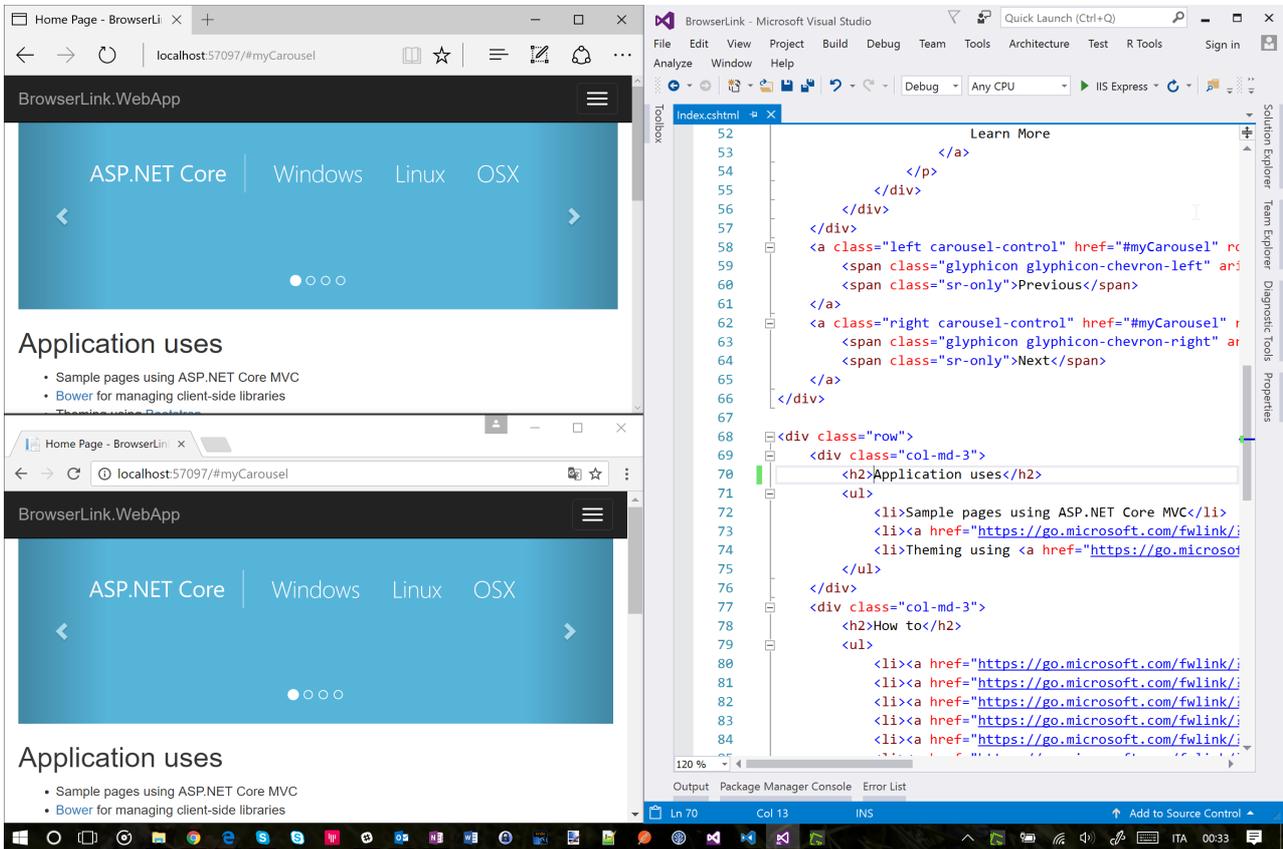
To choose a single web browser to launch when starting the project, use the drop-down menu in the **Debug Target** toolbar control:



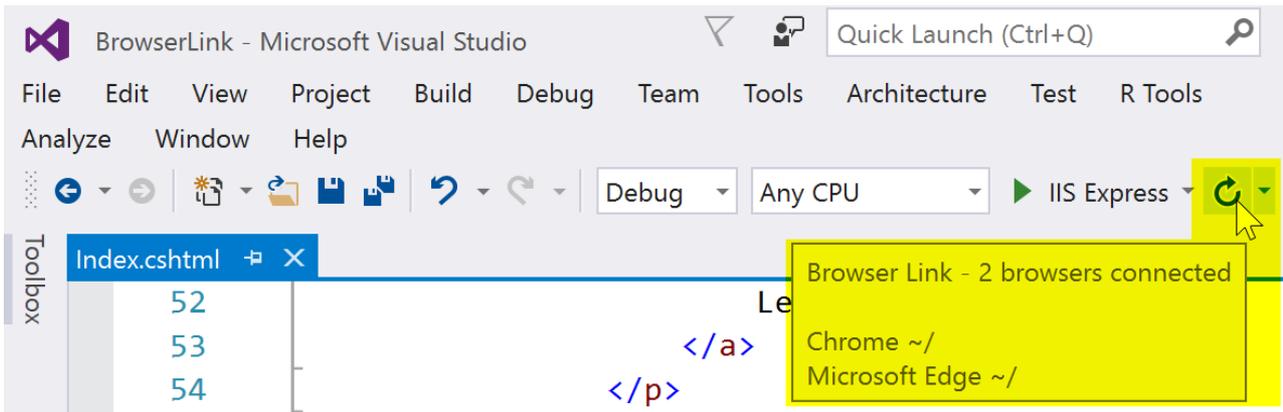
To open multiple browsers at once, choose **Browse with...** from the same drop-down. Hold down the CTRL key to select the browsers you want, and then click **Browse**:



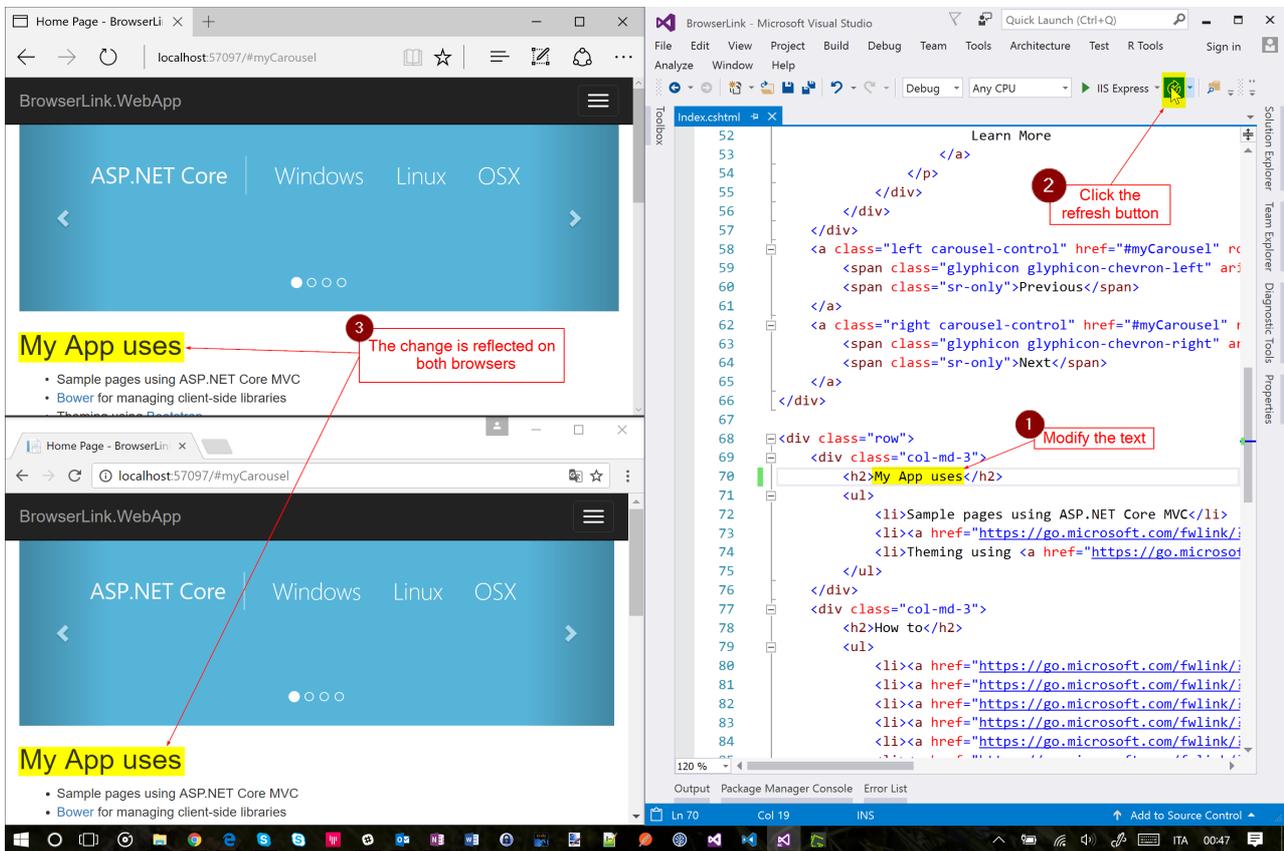
Here's a screenshot showing Visual Studio with the Index view open and two open browsers:



Hover over the Browser Link toolbar control to see the browsers that are connected to the project:



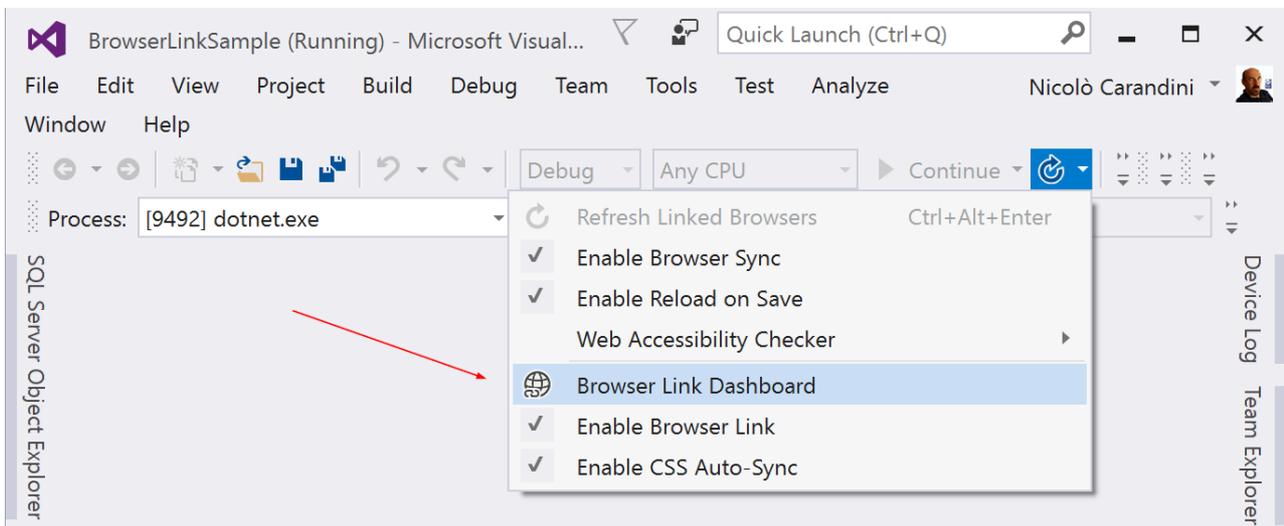
Change the Index view, and all connected browsers are updated when you click the Browser Link refresh button:



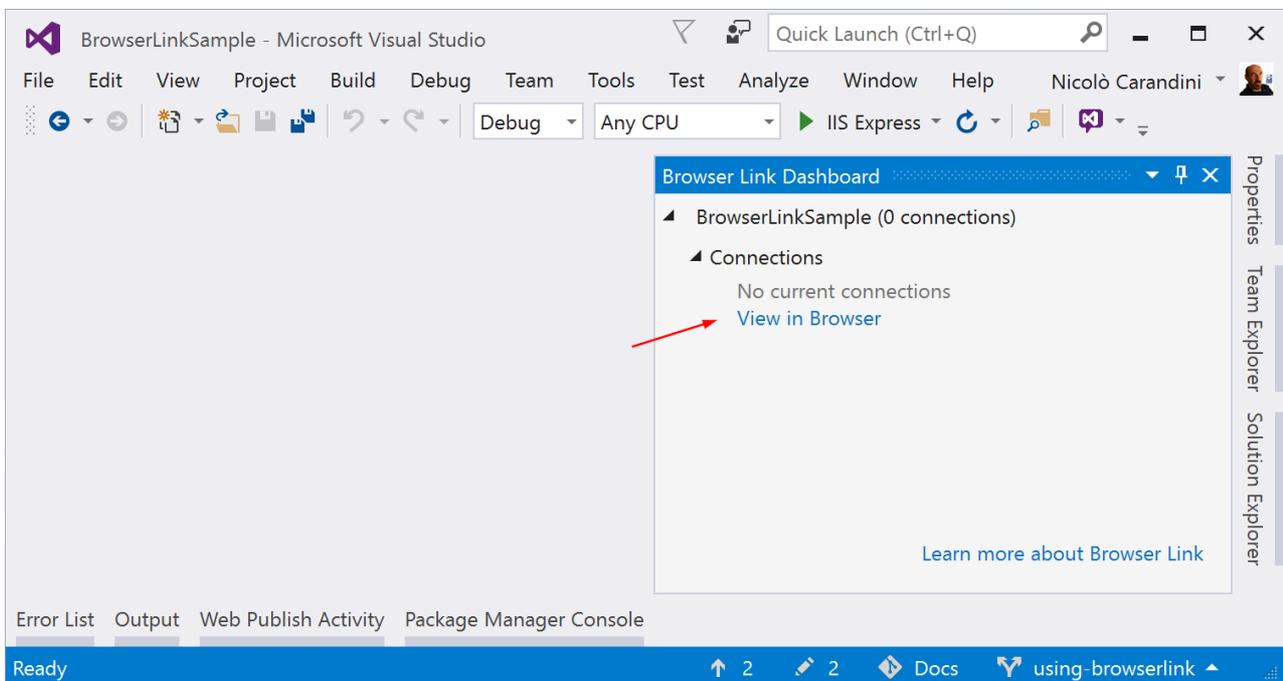
Browser Link also works with browsers that you launch from outside Visual Studio and navigate to the application URL.

The Browser Link Dashboard

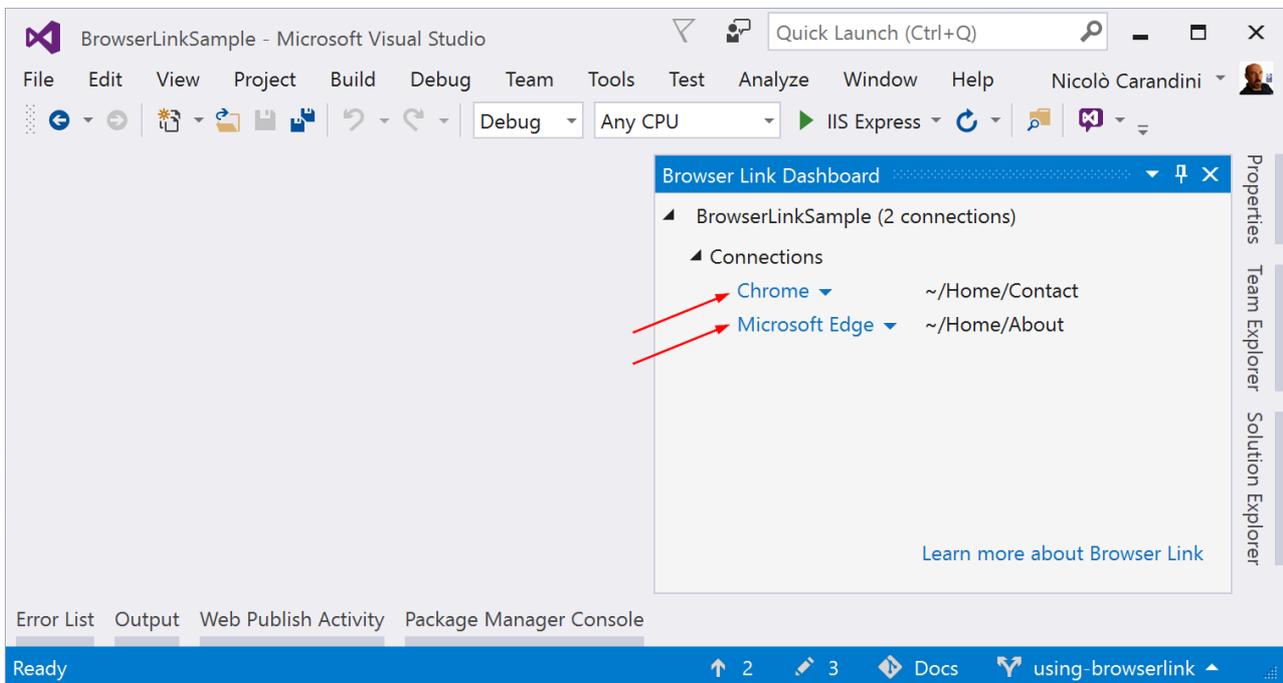
Open the Browser Link Dashboard from the Browser Link drop down menu to manage the connection with open browsers:



If no browser is connected, you can start a non-debugging session by selecting the *View in Browser* link:



Otherwise, the connected browsers are shown with the path to the page that each browser is showing:



If you like, you can click on a listed browser name to refresh that single browser.

Enable or disable Browser Link

When you re-enable Browser Link after disabling it, you must refresh the browsers to reconnect them.

Enable or disable CSS Auto-Sync

When CSS Auto-Sync is enabled, connected browsers are automatically refreshed when you make any change to CSS files.

How does it work?

Browser Link uses SignalR to create a communication channel between Visual Studio and the browser. When Browser Link is enabled, Visual Studio acts as a SignalR server that multiple clients (browsers) can connect to. Browser Link also registers a middleware component in the ASP.NET request pipeline. This component injects special `<script>` references into every page request from the server. You can see the script references by selecting

View source in the browser and scrolling to the end of the `<body>` tag content:

```
<!-- Visual Studio Browser Link -->
<script type="application/json" id="__browserLink_initializationData">
  {"requestId":"a717d5a07c1741949a7cefd6fa2bad08","requestMappingFromServer":false}
</script>
<script type="text/javascript" src="http://localhost:54139/b6e36e429d034f578ebccd6a79bf19bf/browserLink"
async="async"></script>
<!-- End Browser Link -->
</body>
```

Your source files aren't modified. The middleware component injects the script references dynamically.

Because the browser-side code is all JavaScript, it works on all browsers that SignalR supports without requiring a browser plug-in.

Using JavaScriptServices for Creating Single Page Applications with ASP.NET Core

11/21/2017 • 11 min to read • [Edit Online](#)

By [Scott Addie](#) and [Fiyaz Hasan](#)

A Single Page Application (SPA) is a popular type of web application due to its inherent rich user experience. Integrating client-side SPA frameworks or libraries, such as [Angular](#) or [React](#), with server-side frameworks like ASP.NET Core can be difficult. [JavaScriptServices](#) was developed to reduce friction in the integration process. It enables seamless operation between the different client and server technology stacks.

[View or download sample code \(how to download\)](#)

What is JavaScriptServices?

JavaScriptServices is a collection of client-side technologies for ASP.NET Core. Its goal is to position ASP.NET Core as developers' preferred server-side platform for building SPAs.

JavaScriptServices consists of three distinct NuGet packages:

- [Microsoft.AspNetCore.NodeServices](#) (NodeServices)
- [Microsoft.AspNetCore.SpaServices](#) (SpaServices)
- [Microsoft.AspNetCore.SpaTemplates](#) (SpaTemplates)

These packages are useful if you:

- Run JavaScript on the server
- Use a SPA framework or library
- Build client-side assets with Webpack

Much of the focus in this article is placed on using the SpaServices package.

What is SpaServices?

SpaServices was created to position ASP.NET Core as developers' preferred server-side platform for building SPAs. SpaServices is not required to develop SPAs with ASP.NET Core, and it doesn't lock you into a particular client framework.

SpaServices provides useful infrastructure such as:

- [Server-side prerendering](#)
- [Webpack Dev Middleware](#)
- [Hot Module Replacement](#)
- [Routing helpers](#)

Collectively, these infrastructure components enhance both the development workflow and the runtime experience. The components can be adopted individually.

Prerequisites for using SpaServices

To work with SpaServices, install the following:

- [Node.js](#) (version 6 or later) with npm
 - To verify these components are installed and can be found, run the following from the command line:

```
node -v && npm -v
```

Note: If you're deploying to an Azure web site, you don't need to do anything here — Node.js is installed and available in the server environments.

- [.NET Core SDK 1.0](#) (or later)
 - If you're on Windows, this can be installed by selecting Visual Studio 2017's **.NET Core cross-platform development** workload.
- [Microsoft.AspNetCore.SpaServices](#) NuGet package

Server-side prerendering

A universal (also known as isomorphic) application is a JavaScript application capable of running both on the server and the client. Angular, React, and other popular frameworks provide a universal platform for this application development style. The idea is to first render the framework components on the server via Node.js, and then delegate further execution to the client.

ASP.NET Core [Tag Helpers](#) provided by SpaServices simplify the implementation of server-side prerendering by invoking the JavaScript functions on the server.

Prerequisites

Install the following:

- [aspnet-prerendering](#) npm package:

```
npm i -S aspnet-prerendering
```

Configuration

The Tag Helpers are made discoverable via namespace registration in the project's `_ViewImports.cshtml` file:

```
@using SpaServicesSampleApp
@addTagHelper "*", Microsoft.AspNetCore.Mvc.TagHelpers"
@addTagHelper "*", Microsoft.AspNetCore.SpaServices"
```

These Tag Helpers abstract away the intricacies of communicating directly with low-level APIs by leveraging an HTML-like syntax inside the Razor view:

```
<app asp-prerender-module="ClientApp/dist/main-server">Loading...</app>
```

The `asp-prerender-module` Tag Helper

The `asp-prerender-module` Tag Helper, used in the preceding code example, executes `ClientApp/dist/main-server.js` on the server via Node.js. For clarity's sake, `main-server.js` file is an artifact of the TypeScript-to-JavaScript transpilation task in the [Webpack](#) build process. Webpack defines an entry point alias of `main-server`; and, traversal of the dependency graph for this alias begins at the `ClientApp/boot-server.ts` file:

```
entry: { 'main-server': './ClientApp/boot-server.ts' },
```

In the following Angular example, the `ClientApp/boot-server.ts` file utilizes the `createServerRenderer` function and `RenderResult` type of the `aspnet-prerendering` npm package to configure server rendering via Node.js. The HTML markup destined for server-side rendering is passed to a resolve function call, which is wrapped in a strongly-typed JavaScript `Promise` object. The `Promise` object's significance is that it asynchronously supplies the HTML markup to the page for injection in the DOM's placeholder element.

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
  const providers = [
    { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
    { provide: 'ORIGIN_URL', useValue: params.origin }
  ];

  return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
    const appRef = moduleRef.injector.get(ApplicationRef);
    const state = moduleRef.injector.get(PlatformState);
    const zone = moduleRef.injector.get(NgZone);

    return new Promise<RenderResult>((resolve, reject) => {
      zone.onError.subscribe(errorInfo => reject(errorInfo));
      appRef.isStable.first(isStable => isStable).subscribe(() => {
        // Because 'onStable' fires before 'onError', we have to delay slightly before
        // completing the request in case there's an error to report
        setImmediate(() => {
          resolve({
            html: state.renderToString()
          });
          moduleRef.destroy();
        });
      });
    });
  });
});
```

The `asp-prerender-data` Tag Helper

When coupled with the `asp-prerender-module` Tag Helper, the `asp-prerender-data` Tag Helper can be used to pass contextual information from the Razor view to the server-side JavaScript. For example, the following markup passes user data to the `main-server` module:

```
<app asp-prerender-module="ClientApp/dist/main-server"
  asp-prerender-data='new {
    UserName = "John Doe"
  }'>Loading...</app>
```

The received `UserName` argument is serialized using the built-in JSON serializer and is stored in the `params.data` object. In the following Angular example, the data is used to construct a personalized greeting within an `h1` element:

```

import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
  const providers = [
    { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
    { provide: 'ORIGIN_URL', useValue: params.origin }
  ];

  return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
    const appRef = moduleRef.injector.get(ApplicationRef);
    const state = moduleRef.injector.get(PlatformState);
    const zone = moduleRef.injector.get(NgZone);

    return new Promise<RenderResult>((resolve, reject) => {
      const result = `

# Hello, ${params.data.userName}</h1>`; zone.onError.subscribe(errorInfo => reject(errorInfo)); appRef.isStable.first(isStable => isStable).subscribe(() => { // Because 'onStable' fires before 'onError', we have to delay slightly before // completing the request in case there's an error to report setImmediate(() => { resolve({ html: result }); moduleRef.destroy(); }); }); }); }); });


```

Note: Property names passed in Tag Helpers are represented with **PascalCase** notation. Contrast that to JavaScript, where the same property names are represented with **camelCase**. The default JSON serialization configuration is responsible for this difference.

To expand upon the preceding code example, data can be passed from the server to the view by hydrating the `globals` property provided to the `resolve` function:

```

import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
  const providers = [
    { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
    { provide: 'ORIGIN_URL', useValue: params.origin }
  ];

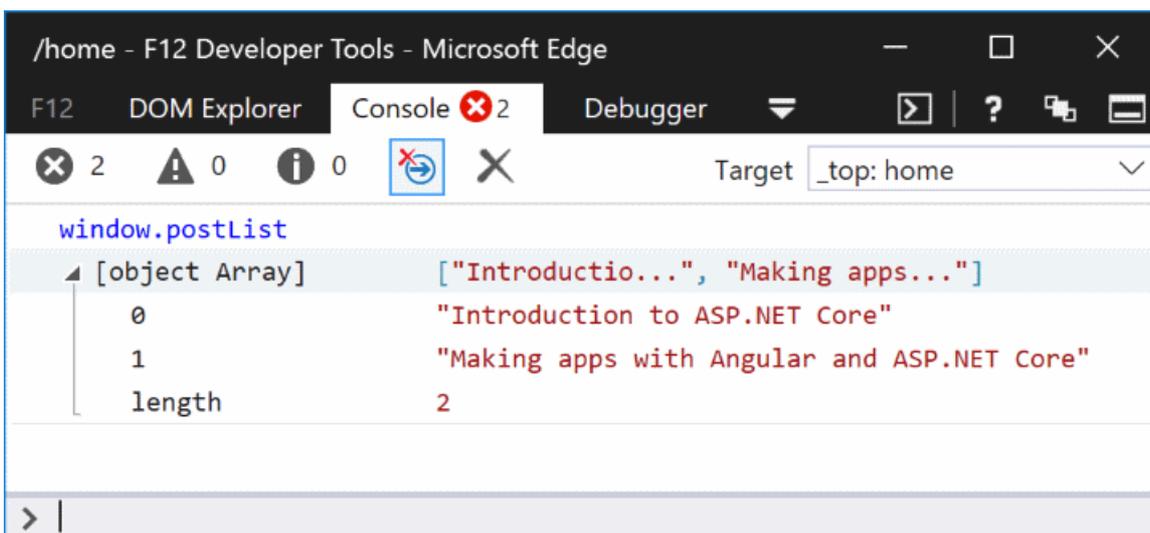
  return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
    const appRef = moduleRef.injector.get(ApplicationRef);
    const state = moduleRef.injector.get(PlatformState);
    const zone = moduleRef.injector.get(NgZone);

    return new Promise<RenderResult>((resolve, reject) => {
      const result = `

# Hello, ${params.data.userName}</h1>`; zone.onError.subscribe(errorInfo => reject(errorInfo)); appRef.isStable.first(isStable => isStable).subscribe(() => { // Because 'onStable' fires before 'onError', we have to delay slightly before // completing the request in case there's an error to report setImmediate(() => { resolve({ html: result, globals: { postList: [ 'Introduction to ASP.NET Core', 'Making apps with Angular and ASP.NET Core' ] } }); moduleRef.destroy(); }); }); }); }); }); }); }); }); });


```

The `postList` array defined inside the `globals` object is attached to the browser's global `window` object. This variable hoisting to global scope eliminates duplication of effort, particularly as it pertains to loading the same data once on the server and again on the client.



Webpack Dev Middleware

[Webpack Dev Middleware](#) introduces a streamlined development workflow whereby Webpack builds resources on demand. The middleware automatically compiles and serves client-side resources when a page is reloaded in the browser. The alternate approach is to manually invoke Webpack via the project's npm build script when a third-

party dependency or the custom code changes. An npm build script in the *package.json* file is shown in the following example:

```
"build": "npm run build:vendor && npm run build:custom",
```

Prerequisites

Install the following:

- [aspnet-webpack](#) npm package:

```
npm i -D aspnet-webpack
```

Configuration

Webpack Dev Middleware is registered into the HTTP request pipeline via the following code in the *Startup.cs* file's `Configure` method:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseWebpackDevMiddleware();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}

// Call UseWebpackDevMiddleware before UseStaticFiles
app.UseStaticFiles();
```

The `UseWebpackDevMiddleware` extension method must be called before [registering static file hosting](#) via the `UseStaticFiles` extension method. For security reasons, register the middleware only when the app runs in development mode.

The *webpack.config.js* file's `output.publicPath` property tells the middleware to watch the `dist` folder for changes:

```
module.exports = (env) => {
    output: {
        filename: '[name].js',
        publicPath: '/dist/' // Webpack dev middleware, if enabled, handles requests for this URL prefix
    },
};
```

Hot Module Replacement

Think of Webpack's [Hot Module Replacement](#) (HMR) feature as an evolution of [Webpack Dev Middleware](#). HMR introduces all the same benefits, but it further streamlines the development workflow by automatically updating page content after compiling the changes. Don't confuse this with a refresh of the browser, which would interfere with the current in-memory state and debugging session of the SPA. There is a live link between the Webpack Dev Middleware service and the browser, which means changes are pushed to the browser.

Prerequisites

Install the following:

- [webpack-hot-middleware](#) npm package:

```
npm i -D webpack-hot-middleware
```

Configuration

The HMR component must be registered into MVC's HTTP request pipeline in the `Configure` method:

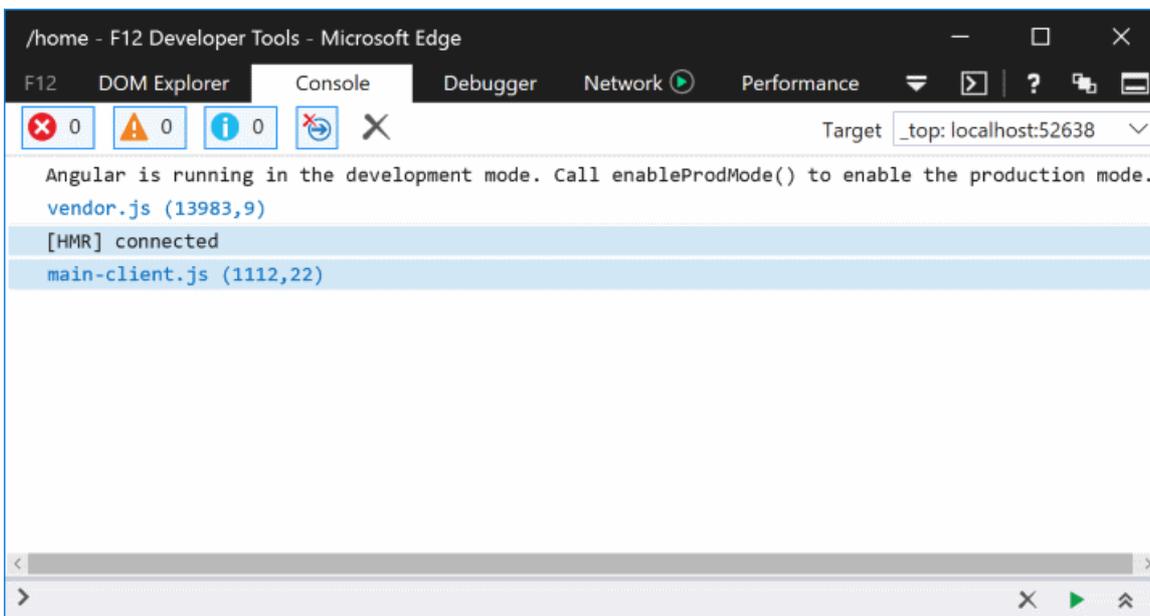
```
app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {  
    HotModuleReplacement = true  
});
```

As was true with [Webpack Dev Middleware](#), the `UseWebpackDevMiddleware` extension method must be called before the `UseStaticFiles` extension method. For security reasons, register the middleware only when the app runs in development mode.

The `webpack.config.js` file must define a `plugins` array, even if it's left empty:

```
module.exports = (env) => {  
    plugins: [new CheckerPlugin()]  
};
```

After loading the app in the browser, the developer tools' Console tab provides confirmation of HMR activation:



Routing helpers

In most ASP.NET Core-based SPAs, you'll want client-side routing in addition to server-side routing. The SPA and MVC routing systems can work independently without interference. There is, however, one edge case posing challenges: identifying 404 HTTP responses.

Consider the scenario in which an extensionless route of `/some/page` is used. Assume the request doesn't pattern-match a server-side route, but its pattern does match a client-side route. Now consider an incoming request for `/images/user-512.png`, which generally expects to find an image file on the server. If that requested resource path doesn't match any server-side route or static file, it's unlikely that the client-side application would handle it — you generally want to return a 404 HTTP status code.

Prerequisites

Install the following:

- The client-side routing npm package. Using Angular as an example:

```
npm i -S @angular/router
```

Configuration

An extension method named `MapSpaFallbackRoute` is used in the `Configure` method:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");

    routes.MapSpaFallbackRoute(
        name: "spa-fallback",
        defaults: new { controller = "Home", action = "Index" });
});
```

Tip: Routes are evaluated in the order in which they're configured. Consequently, the `default` route in the preceding code example is used first for pattern matching.

Creating a new project

JavaScriptServices provides pre-configured application templates. SpaServices is used in these templates, in conjunction with different frameworks and libraries such as Angular, Aurelia, Knockout, React, and Vue.

These templates can be installed via the .NET Core CLI by running the following command:

```
dotnet new --install Microsoft.AspNetCore.SpaTemplates::*
```

A list of available SPA templates is displayed:

TEMPLATES	SHORT NAME	LANGUAGE	TAGS
MVC ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
MVC ASP.NET Core with Aurelia	aurelia	[C#]	Web/MVC/SPA
MVC ASP.NET Core with Knockout.js	knockout	[C#]	Web/MVC/SPA
MVC ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA
MVC ASP.NET Core with React.js and Redux	reactredux	[C#]	Web/MVC/SPA
MVC ASP.NET Core with Vue.js	vue	[C#]	Web/MVC/SPA

To create a new project using one of the SPA templates, include the **Short Name** of the template in the `dotnet new` command. The following command creates an Angular application with ASP.NET Core MVC configured for the server side:

```
dotnet new angular
```

Set the runtime configuration mode

Two primary runtime configuration modes exist:

- **Development:**
 - Includes source maps to ease debugging.
 - Doesn't optimize the client-side code for performance.
- **Production:**
 - Excludes source maps.
 - Optimizes the client-side code via bundling & minification.

ASP.NET Core uses an environment variable named `ASPNETCORE_ENVIRONMENT` to store the configuration mode. See [Setting the environment](#) for more information.

Running with .NET Core CLI

Restore the required NuGet and npm packages by running the following command at the project root:

```
dotnet restore && npm i
```

Build and run the application:

```
dotnet run
```

The application starts on localhost according to the [runtime configuration mode](#). Navigating to `http://localhost:5000` in the browser displays the landing page.

Running with Visual Studio 2017

Open the `.csproj` file generated by the `dotnet new` command. The required NuGet and npm packages are restored automatically upon project open. This restoration process may take up to a few minutes, and the application is ready to run when it completes. Click the green run button or press `Ctrl + F5`, and the browser opens to the application's landing page. The application runs on localhost according to the [runtime configuration mode](#).

Testing the app

SpaServices templates are pre-configured to run client-side tests using [Karma](#) and [Jasmine](#). Jasmine is a popular unit testing framework for JavaScript, whereas Karma is a test runner for those tests. Karma is configured to work with the [Webpack Dev Middleware](#) such that you don't have to stop and run the test every time changes are made. Whether it's the code running against the test case or the test case itself, the test runs automatically.

Using the Angular application as an example, two Jasmine test cases are already provided for the `CounterComponent` in the `counter.component.spec.ts` file:

```

it('should display a title', async(() => {
    const titleText = fixture.nativeElement.querySelector('h1').textContent;
    expect(titleText).toEqual('Counter');
}));

it('should start with count 0, then increments by 1 when clicked', async(() => {
    const countElement = fixture.nativeElement.querySelector('strong');
    expect(countElement.textContent).toEqual('0');

    const incrementButton = fixture.nativeElement.querySelector('button');
    incrementButton.click();
    fixture.detectChanges();
    expect(countElement.textContent).toEqual('1');
}));

```

Open the command prompt at the project root, and run the following command:

```
npm test
```

The script launches the Karma test runner, which reads the settings defined in the *karma.conf.js* file. Among other settings, the *karma.conf.js* identifies the test files to be executed via its `files` array:

```

module.exports = function (config) {
    config.set({
        files: [
            '../wwwroot/dist/vendor.js',
            './boot-tests.ts'
        ],
    });
};

```

Publishing the application

Combining the generated client-side assets and the published ASP.NET Core artifacts into a ready-to-deploy package can be cumbersome. Thankfully, SpaServices orchestrates that entire publication process with a custom MSBuild target named `RunWebpack`:

```

<Target Name="RunWebpack" AfterTargets="ComputeFilesToPublish">
  <!-- As part of publishing, ensure the JS resources are freshly built in production mode -->
  <Exec Command="npm install" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --config webpack.config.vendor.js --env.prod" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --env.prod" />

  <!-- Include the newly-built files in the publish output -->
  <ItemGroup>
    <DistFiles Include="wwwroot\dist\**; ClientApp\dist\*" />
    <ResolvedFileToPublish Include="@{(DistFiles->'%(FullPath)')}" Exclude="@{(ResolvedFileToPublish)}">
      <RelativePath>%(DistFiles.Identity)</RelativePath>
      <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
    </ResolvedFileToPublish>
  </ItemGroup>
</Target>

```

The MSBuild target has the following responsibilities:

1. Restore the npm packages
2. Create a production-grade build of the third-party, client-side assets
3. Create a production-grade build of the custom client-side assets
4. Copy the Webpack-generated assets to the publish folder

The MSBuild target is invoked when running:

```
dotnet publish -c Release
```

Additional resources

- [Angular Docs](#)

Use the Single-Page Application templates (release candidate)

1/5/2018 • 1 min to read • [Edit Online](#)

NOTE

The released .NET Core 2.0.x SDK includes project templates for Angular, React, and React with Redux. **This documentation is not about those released project templates.** This documentation is for the next version of the Angular, React, and React with Redux templates, which we hope to ship in early 2018.

Prerequisites

- [.NET Core SDK](#), version 2.0.0 or later
- [Node.js](#), version 6 or later

Installation

Run the following command to install the **release candidate** of the ASP.NET Core templates for Angular, React, and React with Redux:

```
dotnet new --install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0-rc1-final
```

Use the templates

- [Use the Angular project template](#)
- [Use the React project template](#)
- [Use the React with Redux project template](#)

Use the Angular project template (release candidate)

1/10/2018 • 10 min to read • [Edit Online](#)

NOTE

This documentation is not about the released Angular project template. **This documentation is about the release candidate of the Angular template.** We hope to ship the released version in early 2018.

The updated Angular project template provides a convenient starting point for ASP.NET Core apps using Angular 5 and the Angular CLI to implement a rich, client-side user interface (UI).

The template is equivalent to creating an ASP.NET Core project to act as an API backend and an Angular CLI project to act as a UI. The template offers the convenience of hosting both project types in a single app project. Consequently, the app project can be built and published as a single unit.

Create a new app

To get started, ensure you've [installed the updated Angular project template](#). These instructions don't apply to the previous Angular project template included in the .NET Core 2.0.x SDK.

Create a new project from a command prompt using the command `dotnet new angular` in an empty directory. For example, the following commands create the app in a *my-new-app* directory and switch to that directory:

```
dotnet new angular -o my-new-app
cd my-new-app
```

Run the app from either Visual Studio or the .NET Core CLI:

- [Visual Studio](#)
- [.NET Core CLI](#)

Open the generated *.csproj* file, and run the app as normal from there.

The build process restores npm dependencies on the first run, which can take several minutes. Subsequent builds are much faster.

The project template creates an ASP.NET Core app and an Angular app. The ASP.NET Core app is intended to be used for data access, authorization, and other server-side concerns. The Angular app, residing in the *ClientApp* subdirectory, is intended to be used for all UI concerns.

Add pages, images, styles, modules, etc.

The *ClientApp* directory contains a standard Angular CLI app. See the official [Angular documentation](#) for more information.

There are slight differences between the Angular app created by this template and the one created by Angular CLI itself (via `ng new`); however, the app's capabilities are unchanged. The app created by the template contains a [Bootstrap](#)-based layout and a basic routing example.

Run ng commands

In a command prompt, switch to the *ClientApp* subdirectory:

```
cd ClientApp
```

If you have the `ng` tool installed globally, you can run any of its commands. For example, you can run `ng lint`, `ng test`, or any of the other [Angular CLI commands](#). There's no need to run `ng serve` though, because your ASP.NET Core app deals with serving both server-side and client-side parts of your app. Internally, it uses `ng serve` in development.

If you don't have the `ng` tool installed, run `npm run ng` instead. For example, you can run `npm run ng lint` or `npm run ng test`.

Install npm packages

To install third-party npm packages, use a command prompt in the *ClientApp* subdirectory. For example:

```
cd ClientApp
npm install --save <package_name>
```

Publish and deploy

In development, the app runs in a mode optimized for developer convenience. For example, JavaScript bundles include source maps (so that when debugging, you can see your original TypeScript code). The app watches for TypeScript, HTML, and CSS file changes on disk and automatically recompiles and reloads when it sees those files change.

In production, serve a version of your app that is optimized for performance. This is configured to happen automatically. When you publish, the build configuration emits a minified, ahead-of-time (AoT) compiled build of your client-side code. Unlike the development build, the production build doesn't require Node.js to be installed on the server (unless you have enabled [server-side prerendering](#)).

You can use standard [ASP.NET Core hosting and deployment methods](#).

Run "ng serve" independently

The project is configured to start its own instance of the Angular CLI server in the background when the ASP.NET Core app starts in development mode. This is convenient because you don't have to run a separate server manually.

There is a drawback to this default setup. Each time you modify your C# code and your ASP.NET Core app needs to restart, the Angular CLI server restarts. Around 10 seconds is required to start back up. If you're making frequent C# code edits and don't want to wait for Angular CLI to restart, run the Angular CLI server externally, independently of the ASP.NET Core process. To do so:

1. In a command prompt, switch to the *ClientApp* subdirectory, and launch the Angular CLI development server:

```
cd ClientApp
npm start
```

IMPORTANT

Use `npm start` to launch the Angular CLI development server, not `ng serve`, so that the configuration in `package.json` is respected. To pass additional parameters to the Angular CLI server, add them to the relevant `scripts` line in your `package.json` file.

2. Modify your ASP.NET Core app to use the external Angular CLI instance instead of launching one of its own. In your `Startup` class, replace the `spa.UseAngularCliServer` invocation with the following:

```
spa.UseProxyToSpaDevelopmentServer("http://localhost:4200");
```

When you start your ASP.NET Core app, it won't launch an Angular CLI server. The instance you started manually is used instead. This enables it to start and restart faster. It's no longer waiting for Angular CLI to rebuild your client app each time.

Server-side rendering

As a performance feature, you can choose to pre-render your Angular app on the server as well as running it on the client. This means that browsers receive HTML markup representing your app's initial UI, so they display it even before downloading and executing your JavaScript bundles. Most of the implementation of this comes from an Angular feature called [Angular Universal](#).

TIP

Enabling server-side rendering (SSR) introduces a number of extra complications both during development and deployment. Read [drawbacks of SSR](#) to determine if SSR is a good fit for your requirements.

To enable SSR, you need to make a number of additions to your project.

In the `Startup` class, *after* the line that configures `spa.Options.SourcePath`, and *before* the call to `UseAngularCliServer` OR `UseProxyToSpaDevelopmentServer`, add the following:

```
app.UseSpa(spa =>
{
    spa.Options.SourcePath = "ClientApp";

    spa.UseSpaPrerendering(options =>
    {
        options.BootModulePath = $"{spa.Options.SourcePath}/dist-server/main.bundle.js";
        options.BootModuleBuilder = env.IsDevelopment()
            ? new AngularCliBuilder(npmScript: "build:ssr")
            : null;
        options.ExcludeUrls = new[] { "/sockjs-node" };
    });

    if (env.IsDevelopment())
    {
        spa.UseAngularCliServer(npmScript: "start");
    }
});
```

In development mode, this code attempts to build the SSR bundle by running the script `build:ssr`, which is defined in `ClientApp/package.json`. This builds an Angular app named `ssr`, which is not yet defined.

At the end of the `apps` array in `ClientApp/angular-cli.json`, define an extra app with name `ssr`. Use the following

options:

```
{
  "name": "ssr",
  "root": "src",
  "outDir": "dist-server",
  "assets": [
    "assets"
  ],
  "main": "main.server.ts",
  "tsconfig": "tsconfig.server.json",
  "prefix": "app",
  "scripts": [],
  "environmentSource": "environments/environment.ts",
  "environments": {
    "dev": "environments/environment.ts",
    "prod": "environments/environment.prod.ts"
  },
  "platform": "server"
}
```

This new SSR-enabled app configuration requires two further files: *tsconfig.server.json* and *main.server.ts*. The *tsconfig.server.json* file specifies TypeScript compilation options. The *main.server.ts* file serves as the code entry point during SSR.

Add a new file called *tsconfig.server.json* inside *ClientApp/src* (alongside the existing *tsconfig.app.json*), containing the following:

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "baseUrl": ".",
    "module": "commonjs"
  },
  "angularCompilerOptions": {
    "entryModule": "app/app.server.module#AppServerModule"
  }
}
```

This file configures Angular's AoT compiler to look for a module called `app.server.module`. Add this by creating a new file at *ClientApp/src/app/app.server.module.ts* (alongside the existing *app.module.ts*) containing the following:

```
import { NgModule } from '@angular/core';
import { ServerModule } from '@angular/platform-server';
import { ModuleMapLoaderModule } from '@nguniversal/module-map-ngfactory-loader';
import { AppComponent } from './app.component';
import { AppModule } from './app.module';

@NgModule({
  imports: [AppModule, ServerModule, ModuleMapLoaderModule],
  bootstrap: [AppComponent]
})
export class AppServerModule { }
```

This module inherits from your client-side `app.module` and defines which extra Angular modules are available during SSR.

Recall that the new `ssr` entry in *.angular-cli.json* referenced an entry point file called *main.server.ts*. You haven't yet added that file, and now is time to do so. Create a new file at *ClientApp/src/main.server.ts* (alongside the existing *main.ts*), containing the following:

```

import 'zone.js/dist/zone-node';
import 'reflect-metadata';
import { renderModule, renderModuleFactory } from '@angular/platform-server';
import { APP_BASE_HREF } from '@angular/common';
import { enableProdMode } from '@angular/core';
import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';
import { createServerRenderer } from 'aspnet-prerendering';
export { AppServerModule } from './app/app.server.module';

enableProdMode();

export default createServerRenderer(params => {
  const { AppServerModule, AppServerModuleNgFactory, LAZY_MODULE_MAP } = (module as any).exports;

  const options = {
    document: params.data.originalHtml,
    url: params.url,
    extraProviders: [
      provideModuleMap(LAZY_MODULE_MAP),
      { provide: APP_BASE_HREF, useValue: params.baseUrl },
      { provide: 'BASE_URL', useValue: params.origin + params.baseUrl }
    ]
  };

  const renderPromise = AppServerModuleNgFactory
    ? /* AoT */ renderModuleFactory(AppServerModuleNgFactory, options)
    : /* dev */ renderModule(AppServerModule, options);

  return renderPromise.then(html => ({ html }));
});

```

This file's code is what ASP.NET Core executes for each request when it runs the `UseSpaPrerendering` middleware that you added to the `Startup` class. It deals with receiving `params` from the .NET code (such as the URL being requested), and making calls to Angular SSR APIs to get the resulting HTML.

Strictly-speaking, this is sufficient to enable SSR in development mode. It's essential to make one final change so that your app works correctly when published. In your app's main `.csproj` file, set the `BuildServerSideRenderer` property value to `true`:

```

<!-- Set this to true if you enable server-side prerendering -->
<BuildServerSideRenderer>true</BuildServerSideRenderer>

```

This configures the build process to run `build:ssr` during publishing and deploy the SSR files to the server. If you don't enable this, SSR fails in production.

When your app runs in either development or production mode, the Angular code pre-renders as HTML on the server. The client-side code executes as normal.

Pass data from .NET code into TypeScript code

During SSR, you might want to pass per-request data from your ASP.NET Core app into your Angular app. For example, you could pass cookie information or something read from a database. To do this, edit your `Startup` class. In the callback for `UseSpaPrerendering`, set a value for `options.SupplyData` such as the following:

```

options.SupplyData = (context, data) =>
{
  // Creates a new value called isHttpRequest that is passed to TypeScript code
  data["isHttpRequest"] = context.Request.IsHttps;
};

```

The `SupplyData` callback lets you pass arbitrary, per-request, JSON-serializable data (for example, strings, booleans, or numbers). Your `main.server.ts` code receives this as `params.data`. For example, the preceding code sample passes a boolean value as `params.data.isHttpRequest` into the `createServerRenderer` callback. You can pass this to other parts of your app in any way supported by Angular. For example, see how `main.server.ts` passes the `BASE_URL` value to any component whose constructor is declared to receive it.

Drawbacks of SSR

Not all apps benefit from SSR. The primary benefit is perceived performance. Visitors reaching your app over a slow network connection or on slow mobile devices see the initial UI quickly, even if it takes a while to fetch or parse the JavaScript bundles. However, many SPAs are mainly used over fast, internal company networks on fast computers where the app appears almost instantly.

At the same time, there are significant drawbacks to enabling SSR. It adds complexity to your development process. Your code must run in two different environments: client-side and server-side (in a Node.js environment invoked from ASP.NET Core). Here are some things to bear in mind:

- SSR requires a Node.js installation on your production servers. This is automatically the case for some deployment scenarios, such as Azure App Services, but not for others, such as Azure Service Fabric.
- Enabling the `BuildServerSideRenderer` build flag causes your `node_modules` directory to publish. This folder contains 20,000+ files, which increases deployment time.
- To run your code in a Node.js environment, it can't rely on the existence of browser-specific JavaScript APIs such as `window` or `localStorage`. If your code (or some third-party library you reference) tries to use these APIs, you'll get an error during SSR. For example, don't use jQuery because it references browser-specific APIs in many places. To prevent errors, you must either avoid SSR or avoid browser-specific APIs or libraries. You can wrap any calls to such APIs in checks to ensure they aren't invoked during SSR. For example, use a check such as the following in JavaScript or TypeScript code:

```
if (typeof window !== 'undefined') {  
    // Call browser-specific APIs here  
}
```

Use the React project template (release candidate)

1/10/2018 • 3 min to read • [Edit Online](#)

NOTE

This documentation is not about the released React project template. **This documentation is about the release candidate of the React template.** We hope to ship the released version in early 2018.

The updated React project template provides a convenient starting point for ASP.NET Core apps using React and [create-react-app](#) (CRA) conventions to implement a rich, client-side user interface (UI).

The template is equivalent to creating both an ASP.NET Core project to act as an API backend, and a standard CRA React project to act as a UI, but with the convenience of hosting both in a single app project that can be built and published as a single unit.

Create a new app

To get started, ensure you've [installed the updated React project template](#). These instructions don't apply to the previous React project template included in the .NET Core 2.0.x SDK.

Create a new project from a command prompt using the command `dotnet new react` in an empty directory. For example, the following commands create the app in a *my-new-app* directory and switch to that directory:

```
dotnet new -o my-new-app
cd my-new-app
```

Run the app from either Visual Studio or the .NET Core CLI:

- [Visual Studio](#)
- [.NET Core CLI](#)

Open the generated *.csproj* file, and run the app as normal from there.

The build process restores npm dependencies on the first run, which can take several minutes. Subsequent builds are much faster.

The project template creates an ASP.NET Core app and a React app. The ASP.NET Core app is intended to be used for data access, authorization, and other server-side concerns. The React app, residing in the *ClientApp* subdirectory, is intended to be used for all UI concerns.

Add pages, images, styles, modules, etc.

The *ClientApp* directory is a standard CRA React app. See the official [CRA documentation](#) for more information.

There are slight differences between the React app created by this template and the one created by CRA itself; however, the app's capabilities are unchanged. The app created by the template contains a [Bootstrap](#)-based layout and a basic routing example.

Install npm packages

To install third-party npm packages, use a command prompt in the *ClientApp* subdirectory. For example:

```
cd ClientApp
npm install --save <package_name>
```

Publish and deploy

In development, the app runs in a mode optimized for developer convenience. For example, JavaScript bundles include source maps (so that when debugging, you can see your original source code). The app watches JavaScript, HTML, and CSS file changes on disk and automatically recompiles and reloads when it sees those files change.

In production, serve a version of your app that is optimized for performance. This is configured to happen automatically. When you publish, the build configuration emits a minified, transpiled build of your client-side code. Unlike the development build, the production build doesn't require Node.js to be installed on the server.

You can use standard [ASP.NET Core hosting and deployment methods](#).

Run the CRA server independently

The project is configured to start its own instance of the CRA development server in the background when the ASP.NET Core app starts in development mode. This is convenient because it means you don't have to run a separate server manually.

There is a drawback to this default setup. Each time you modify your C# code and your ASP.NET Core app needs to restart, the CRA server restarts. A few seconds are required to start back up. If you're making frequent C# code edits and don't want to wait for the CRA server to restart, run the CRA server externally, independently of the ASP.NET Core process. To do so:

1. In a command prompt, switch to the *ClientApp* subdirectory, and launch the CRA development server:

```
cd ClientApp
npm start
```

2. Modify your ASP.NET Core app to use the external CRA server instance instead of launching one of its own. In your *Startup* class, replace the `spa.UseReactDevelopmentServer` invocation with the following:

```
spa.UseProxyToSpaDevelopmentServer("http://localhost:3000");
```

When you start your ASP.NET Core app, it won't launch a CRA server. The instance you started manually is used instead. This enables it to start and restart faster. It's no longer waiting for your React app to rebuild each time.

Use the React-with-Redux project template (release candidate)

1/5/2018 • 1 min to read • [Edit Online](#)

NOTE

This documentation is not about the released React-with-Redux project template. **This documentation is about the release candidate of the React-with-Redux template.** We hope to ship the released version in early 2018.

The updated React-with-Redux project template provides a convenient starting point for ASP.NET Core apps using React, Redux, and [create-react-app](#) (CRA) conventions to implement a rich, client-side user interface (UI).

With the exception of the project creation command, all information about the React-with-Redux template is the same as the React template. To create this project type, run `dotnet new reactredux` instead of `dotnet new react`. For more information about the functionality common to both React-based templates, see [React template documentation](#).

Mobile

1/10/2018 • 1 min to read • [Edit Online](#)

- [Create backend services for native mobile apps](#)

Creating Backend Services for Native Mobile Applications

9/30/2017 • 8 min to read • [Edit Online](#)

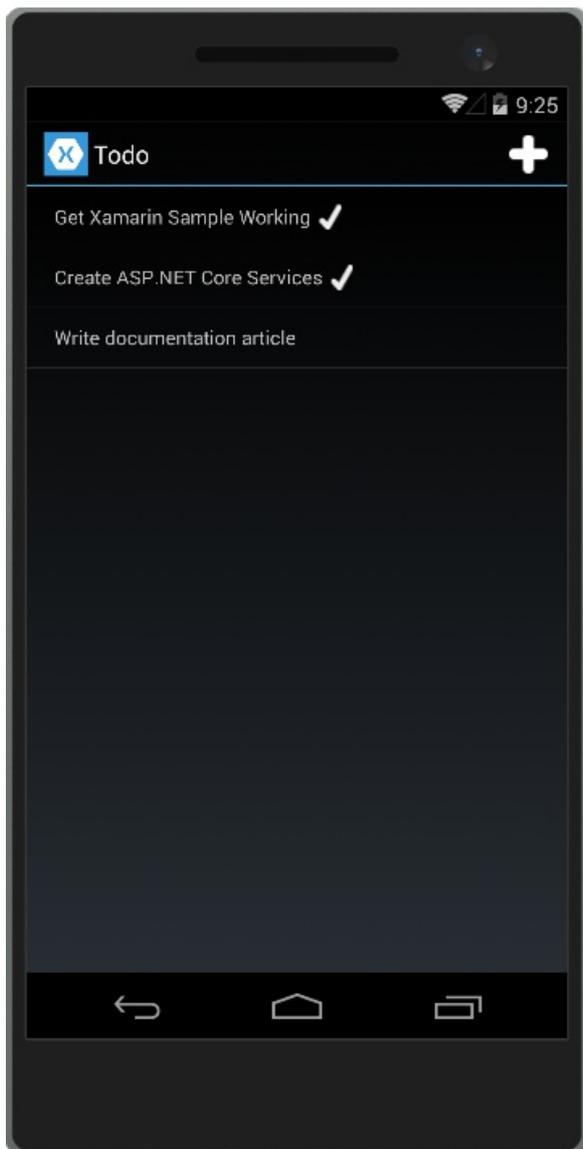
By [Steve Smith](#)

Mobile apps can easily communicate with ASP.NET Core backend services.

[View or download sample backend services code](#)

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core MVC to support native mobile apps. It uses the [Xamarin Forms ToDoRest app](#) as its native client, which includes separate native clients for Android, iOS, Windows Universal, and Window Phone devices. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), as well as download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Web API 2 services project, which this article's ASP.NET Core app replaces (with no changes required by the client).



Features

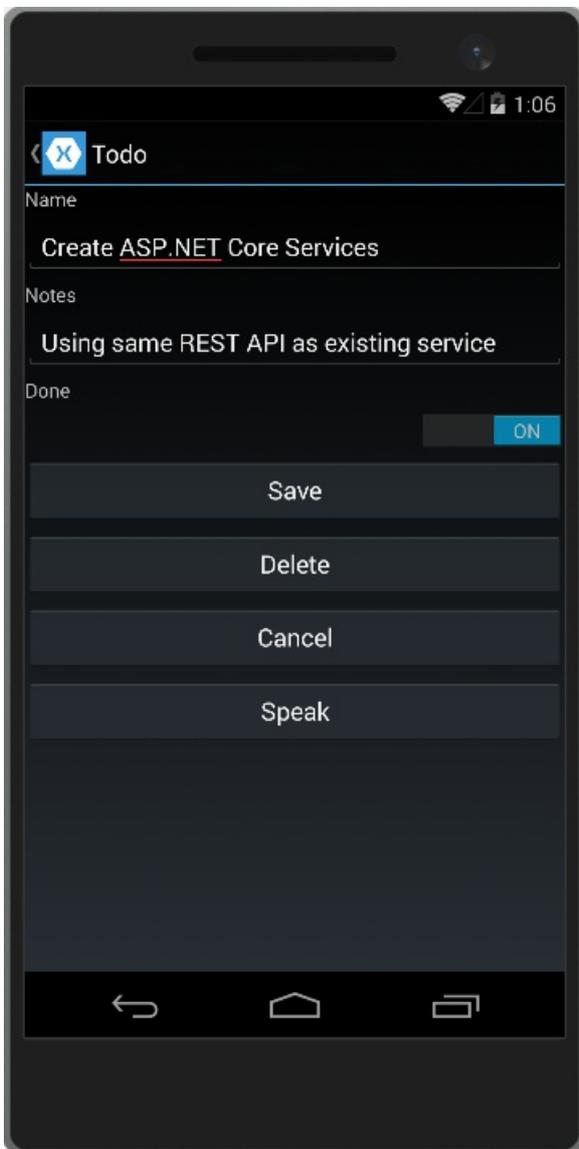
The ToDoRest app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

The main view of the items, as shown above, lists each item's name and indicates if it is done with a checkmark.

Tapping the  icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:



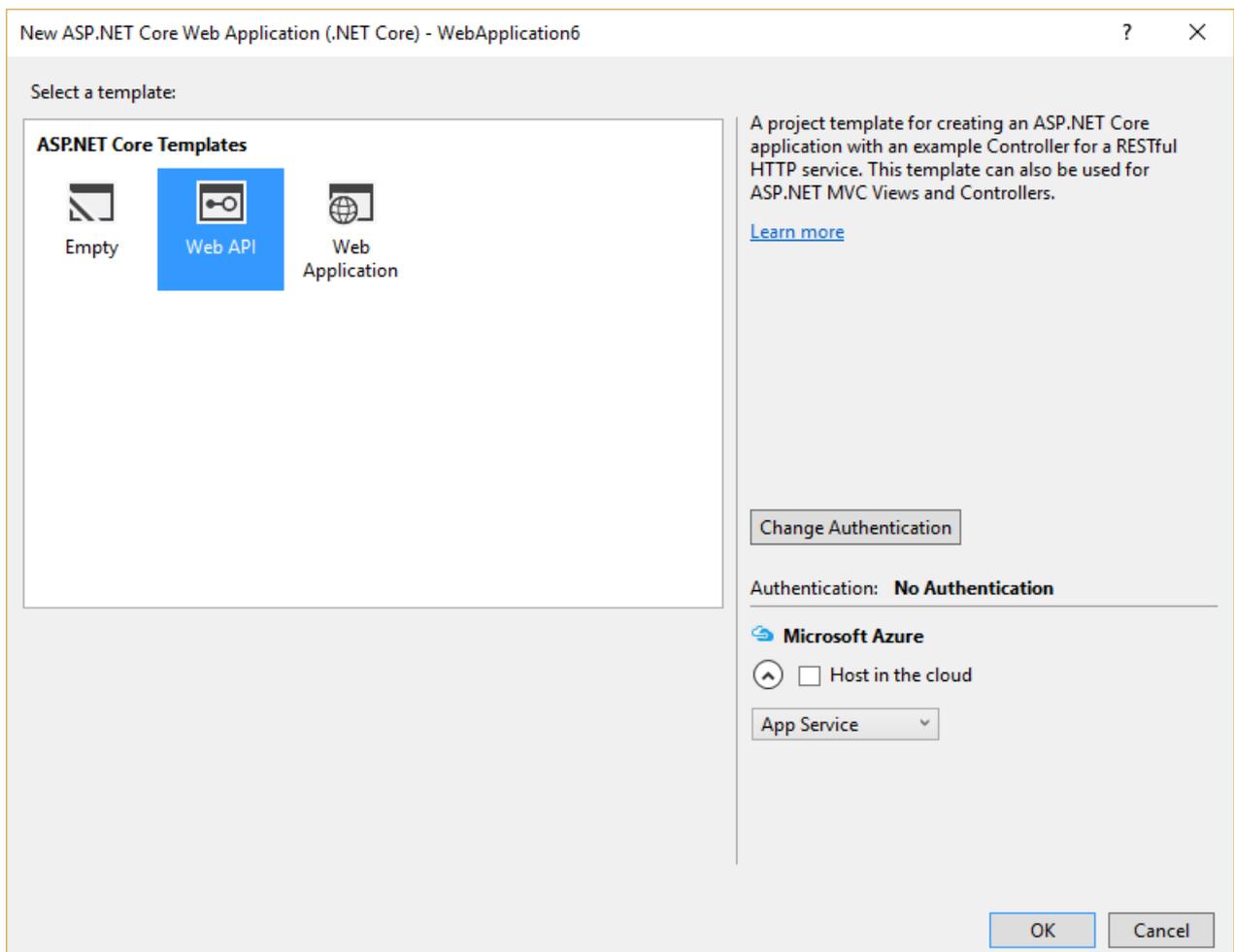
This sample is configured by default to use backend services hosted at `developer.xamarin.com`, which allow read-only operations. To test it out yourself against the ASP.NET Core app created in the next section running on your computer, you'll need to update the app's `RestUrl` constant. Navigate to the `ToDoREST` project and open the `Constants.cs` file. Replace the `RestUrl` with a URL that includes your machine's IP address (not localhost or 127.0.0.1, since this address is used from the device emulator, not from your machine). Include the port number as well (5000). In order to test that your services work with a device, ensure you don't have an active firewall blocking access to this port.

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems/{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template and No Authentication. Name the project `ToDoApi`.



The application should respond to all requests made to port 5000. Update *Program.cs* to include

`.UseUrls("http://*:5000")` to achieve this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

NOTE

Make sure you run the application directly, rather than behind IIS Express, which ignores non-local requests by default. Run `dotnet run` from a command prompt, or choose the application name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields using the `[Required]` attribute:

```

using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        public string Notes { get; set; }

        public bool Done { get; set; }
    }
}

```

The API methods require some way to work with data. Use the same `IToDoRepository` interface the original Xamarin sample uses:

```

using System.Collections.Generic;
using ToDoApi.Models;

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}

```

For this sample, the implementation just uses a private collection of items:

```

using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _toDoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _toDoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _toDoList.Any(item => item.ID == id);
        }
    }
}

```

```

    }

    public TodoItem Find(string id)
    {
        return _todoList.FirstOrDefault(item => item.ID == id);
    }

    public void Insert(TodoItem item)
    {
        _todoList.Add(item);
    }

    public void Update(TodoItem item)
    {
        var todoItem = this.Find(item.ID);
        var index = _todoList.IndexOf(todoItem);
        _todoList.RemoveAt(index);
        _todoList.Insert(index, item);
    }

    public void Delete(string id)
    {
        _todoList.Remove(this.Find(id));
    }

    private void InitializeData()
    {
        _todoList = new List<TodoItem>();

        var todoItem1 = new TodoItem
        {
            ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
            Name = "Learn app development",
            Notes = "Attend Xamarin University",
            Done = true
        };

        var todoItem2 = new TodoItem
        {
            ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
            Name = "Develop apps",
            Notes = "Use Xamarin Studio/Visual Studio",
            Done = false
        };

        var todoItem3 = new TodoItem
        {
            ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
            Name = "Publish apps",
            Notes = "All app stores",
            Done = false,
        };

        _todoList.Add(todoItem1);
        _todoList.Add(todoItem2);
        _todoList.Add(todoItem3);
    }
}
}

```

Configure the implementation in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<ITodoRepository, TodoRepository>();
}
```

At this point, you're ready to create the *ToDoItemsController*.

TIP

Learn more about creating web APIs in [Building Your First Web API with ASP.NET Core MVC and Visual Studio](#).

Creating the Controller

Add a new controller to the project, *ToDoItemsController*. It should inherit from `Microsoft.AspNetCore.Mvc.Controller`. Add a `Route` attribute to indicate that the controller will handle requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about [routing](#).

The controller requires an `ITodoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance will be provided using the framework's support for [dependency injection](#).

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly ITodoRepository _todoRepository;

        public ToDoItemsController(ITodoRepository todoRepository)
        {
            _todoRepository = todoRepository;
        }
    }
}
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP GET request.

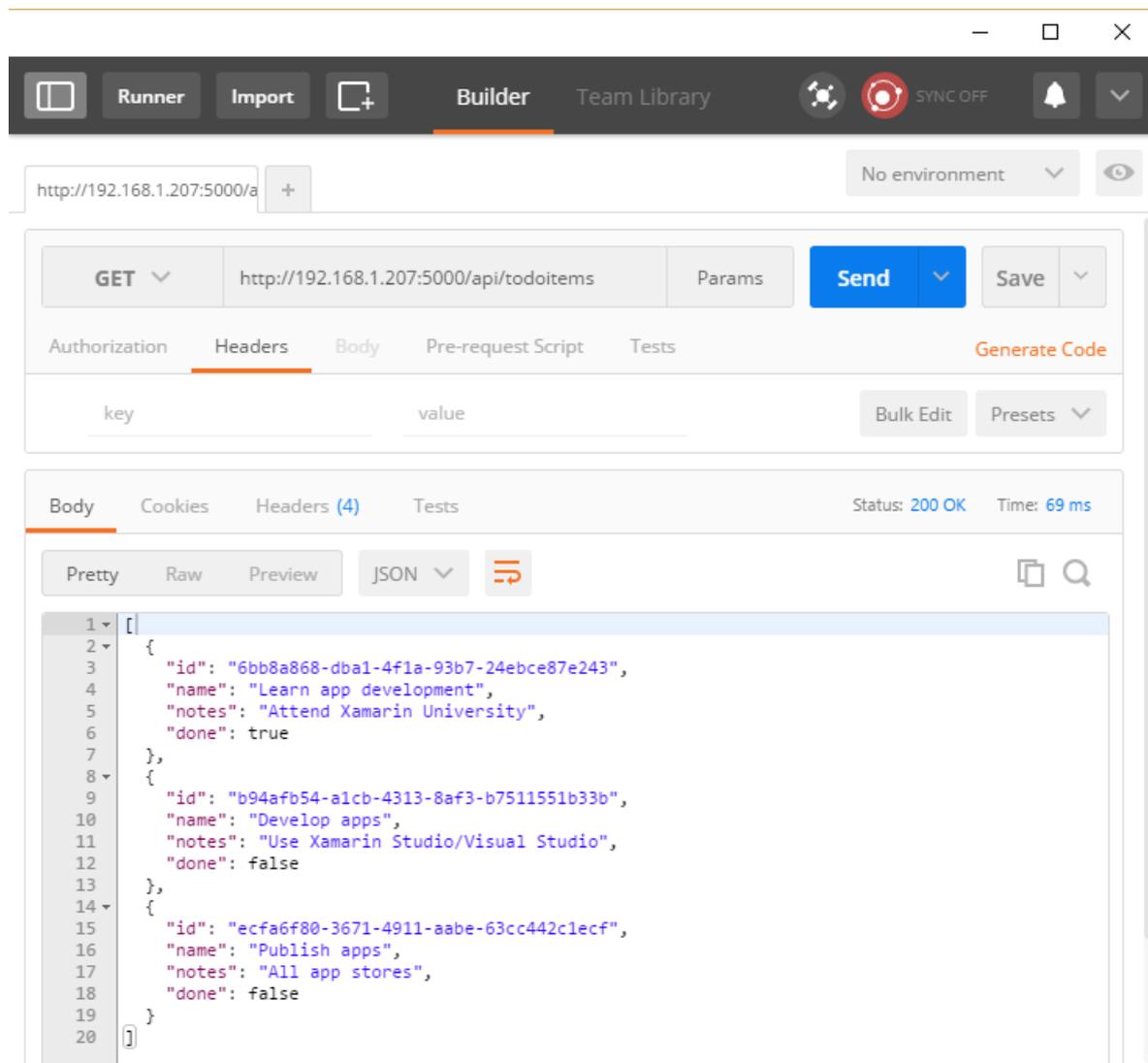
Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

```
[HttpGet]
public IActionResult List()
{
    return Ok(_todoRepository.All);
}
```

The `List` method returns a 200 OK response code and all of the `ToDo` items, serialized as JSON.

You can test your new API method using a variety of tools, such as [Postman](#), shown here:



Creating Items

By convention, creating new data items is mapped to the HTTP POST verb. The `Create` method has an `[HttpPost]` attribute applied to it, and accepts a `ToDoItem` instance. Since the `item` argument will be passed in the body of the POST, this parameter is decorated with the `[FromBody]` attribute.

Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it is added using the repository. Checking `ModelState.IsValid` performs [model validation](#), and should be done in every API method that accepts user input.

```

[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TODOItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _todoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TODOItemIDInUse.ToString());
        }
        _todoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}

```

The sample uses an enum containing error codes that are passed to the mobile client:

```

public enum ErrorCode
{
    TODOItemNameAndNotesRequired,
    TODOItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}

```

Test adding new items using Postman by choosing the POST verb providing the new object in JSON format in the Body of the request. You should also add a request header specifying a `Content-Type` of `application/json`.

The screenshot shows a REST client interface with a dark top bar containing 'Runner', 'Import', 'Builder', and 'Team Library' tabs. Below the top bar, there's a URL input field with 'http://192.168.1.207:5000/a' and a 'No environment' dropdown. The main area is divided into two sections. The top section is for the request, with a 'POST' method selected and the URL 'http://192.168.1.207:5000/api/todoitems'. The 'Body' tab is active, showing a JSON payload:

```
{
  "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",
  "Name": "A Test Item",
  "Notes": "asdf",
  "Done": false
}
```

. The bottom section shows the response, with a status of '200 OK' and a time of '227 ms'. The 'Body' tab is active, showing the response JSON:

```
{
  "id": "6bb8b868-dba1-4f1a-93b7-24ebce87243",
  "name": "A Test Item",
  "notes": "asdf",
  "done": false
}
```

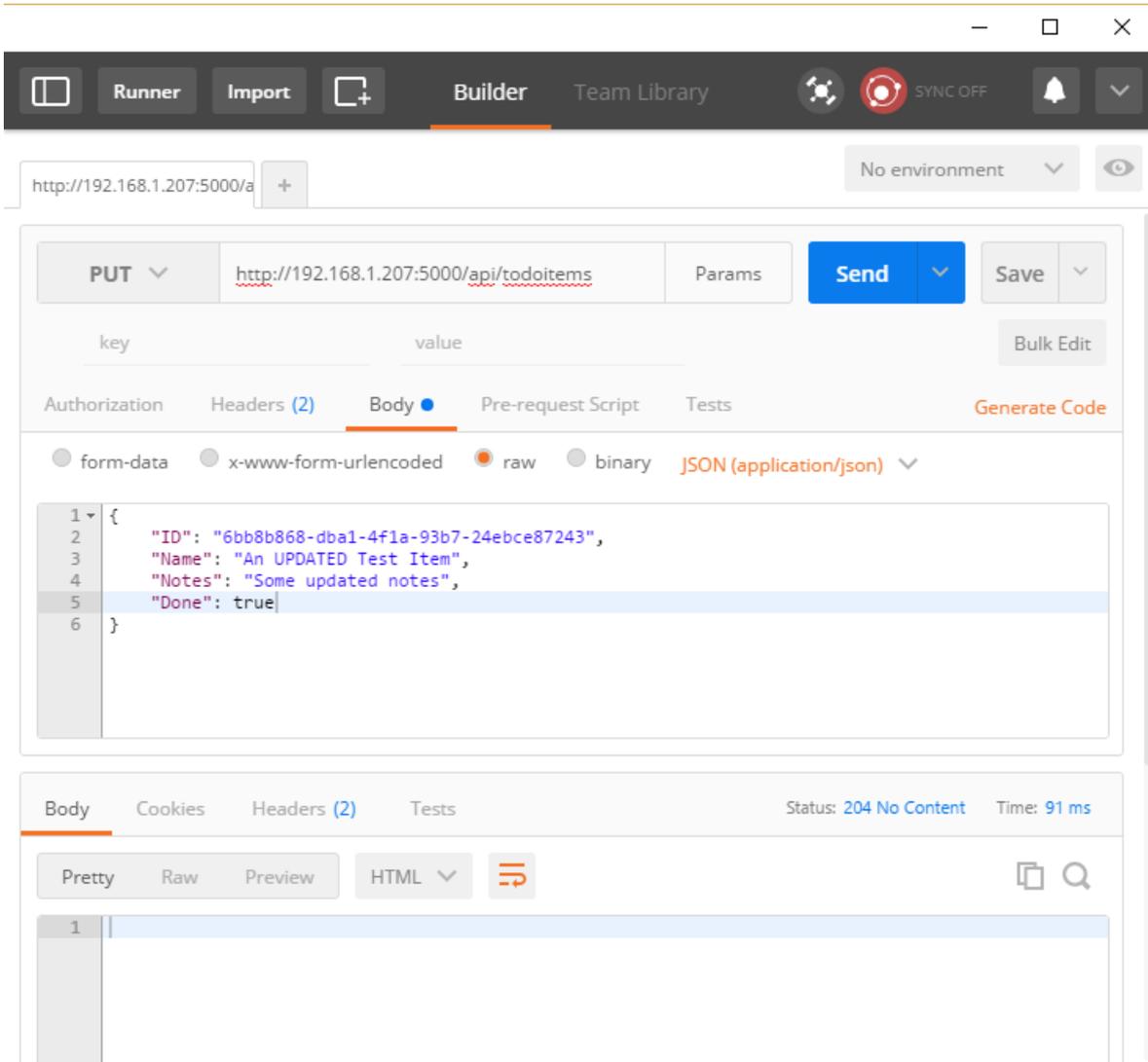
The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP PUT requests. Other than this change, the `Edit` method is almost identical to `Create`. Note that if the record isn't found, the `Edit` action will return a `NotFound` (404) response.

```
[HttpPut]
public IActionResult Edit([FromBody] TodoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TODOItemNameAndNotesRequired.ToString());
        }
        var existingItem = _todoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with Postman, change the verb to PUT. Specify the updated object data in the Body of the request.



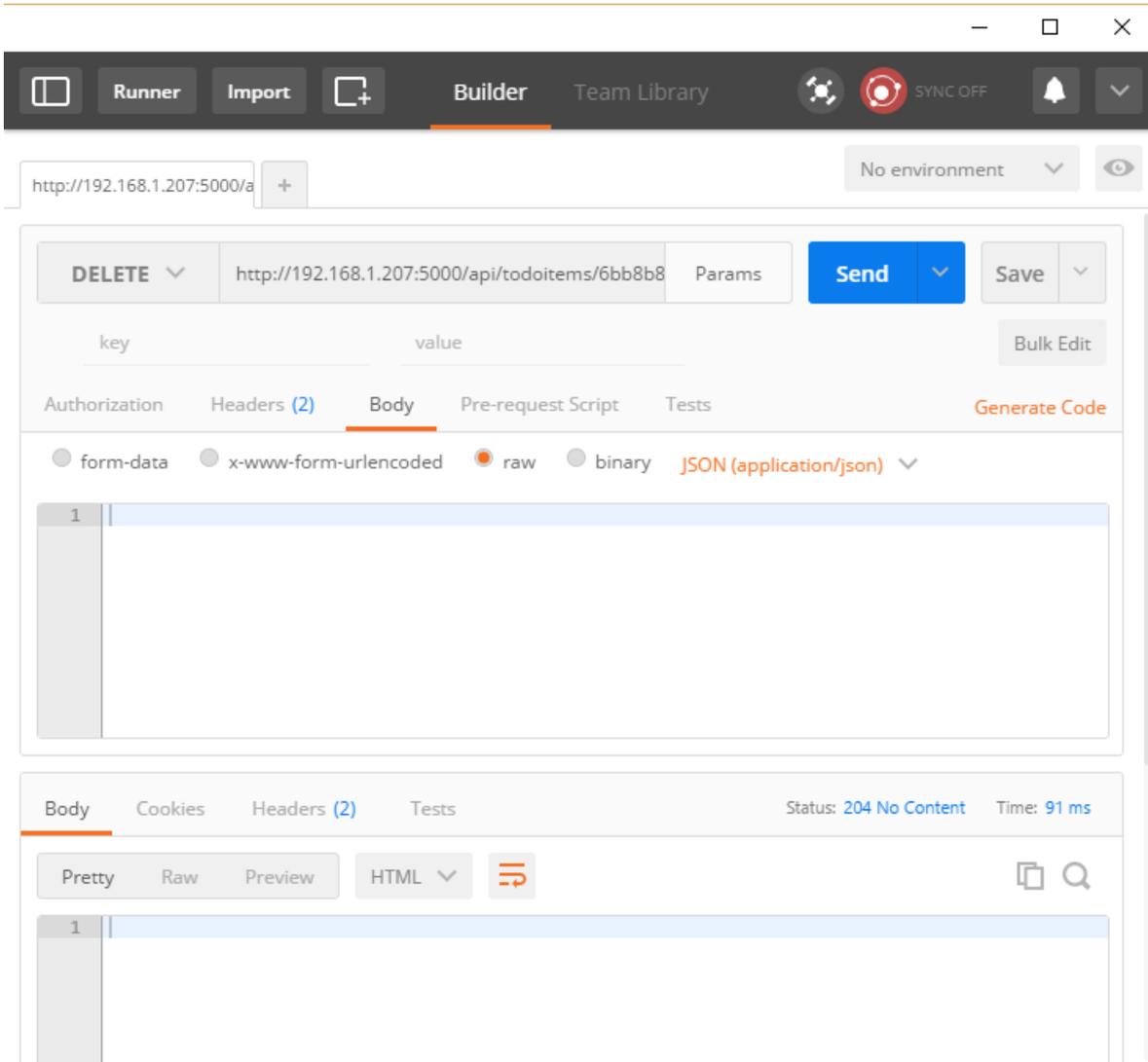
This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making DELETE requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist will receive `NotFound` responses. Otherwise, a successful request will get a `NoContent` (204) response.

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _todoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

Note that when testing the delete functionality, nothing is required in the Body of the request.



Common Web API Conventions

As you develop the backend services for your app, you will want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown above, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a [filter](#). Learn more about [how to encapsulate common API policies in ASP.NET Core MVC applications](#).

Host and deploy ASP.NET Core

1/10/2018 • 2 min to read • [Edit Online](#)

In general, to deploy an ASP.NET Core app to a hosting environment:

- Publish the app to a folder on the hosting server.
- Set up a process manager that starts the app when requests arrive and restarts the app after it crashes or the server reboots.
- Set up a reverse proxy that forwards requests to the app.

Publish to a folder

The `dotnet publish` CLI command compiles app code and copies the files needed to run the app into a *publish* folder. When deploying from Visual Studio, the `dotnet publish` step happens automatically before the files are copied to the deployment destination.

Folder contents

The *publish* folder contains *.exe* and *.dll* files for the app, its dependencies, and optionally the .NET runtime.

A .NET Core app can be published as *self-contained* or *framework-dependent* app. If the app is self-contained, the *.dll* files that contain the .NET runtime are included in the *publish* folder. If the app is framework-dependent, the .NET runtime files aren't included because the app has a reference to a version of .NET that is installed on the server. The default deployment model is framework-dependent. For more information, see [.NET Core application deployment](#).

In addition to *.exe* and *.dll* files, the *publish* folder for an ASP.NET Core app typically contains configuration files, static assets, and MVC views. For more information, see [Directory structure](#).

Set up a process manager

An ASP.NET Core app is a console app that must be started when a server boots and restarted if it crashes. To automate starts and restarts, a process manager is required. The most common process managers for ASP.NET Core are:

- Linux
 - [nginx](#)
 - [Apache](#)
- Windows
 - [IIS](#)
 - [Windows Service](#)

Set up a reverse proxy

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

If the app uses the [Kestrel](#) web server, [nginx](#), [Apache](#), or [IIS](#) can be used as a reverse proxy server. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling. For more information, see [When to use Kestrel with a reverse proxy](#).

Using Visual Studio and MSBuild to automate deployment

Deployment often requires additional tasks besides copying the output from `dotnet publish` to a server. For example, extra files might be required or excluded from the *publish* folder. Visual Studio uses MSBuild for web deployment, and MSBuild can be customized to do many other tasks during deployment. For more information, see [Publish profiles in Visual Studio](#) and the [Using MSBuild and Team Foundation Build](#) book.

By using [the Publish Web feature](#) or [built-in Git support](#), apps can be deployed directly from Visual Studio to the Azure App Service. Visual Studio Team Services supports [continuous deployment to Azure App Service](#).

Publishing to Azure

See [Publish an ASP.NET Core web app to Azure App Service using Visual Studio](#) for instructions on how to publish an app to Azure using Visual Studio. The app can also be published to Azure from the [command line](#).

Additional resources

For information on using Docker as a hosting environment, see [Host ASP.NET Core apps in Docker](#).

Host ASP.NET Core on Azure App Service

1/10/2018 • 1 min to read • [Edit Online](#)

The following articles are available for learning about hosting ASP.NET Core apps in Azure App Service:

[Publish to Azure with Visual Studio](#)

Learn how to publish an ASP.NET Core app to Azure App Service using Visual Studio.

[Publish to Azure with CLI tools](#)

Learn how to publish an ASP.NET Core app to Azure App Service using the Git command line client.

[Continuous deployment to Azure with Visual Studio and Git](#)

Learn how to create an ASP.NET Core web app using Visual Studio and deploy it to Azure App Service using Git for continuous deployment.

[Continuous deployment to Azure with VSTS](#)

Set up a CI build for an ASP.NET Core app, then create a continuous deployment release to Azure App Service.

[Troubleshoot ASP.NET Core on Azure App Service](#) *(Topic under development)*

Learn how to diagnose problems with ASP.NET Core Azure App Service deployments.

/en-us

Publish an ASP.NET Core web app to Azure App Service using Visual Studio

By [Rick Anderson](#), [Cesar Blum Silveira](#), and [Rachel Appel](#)

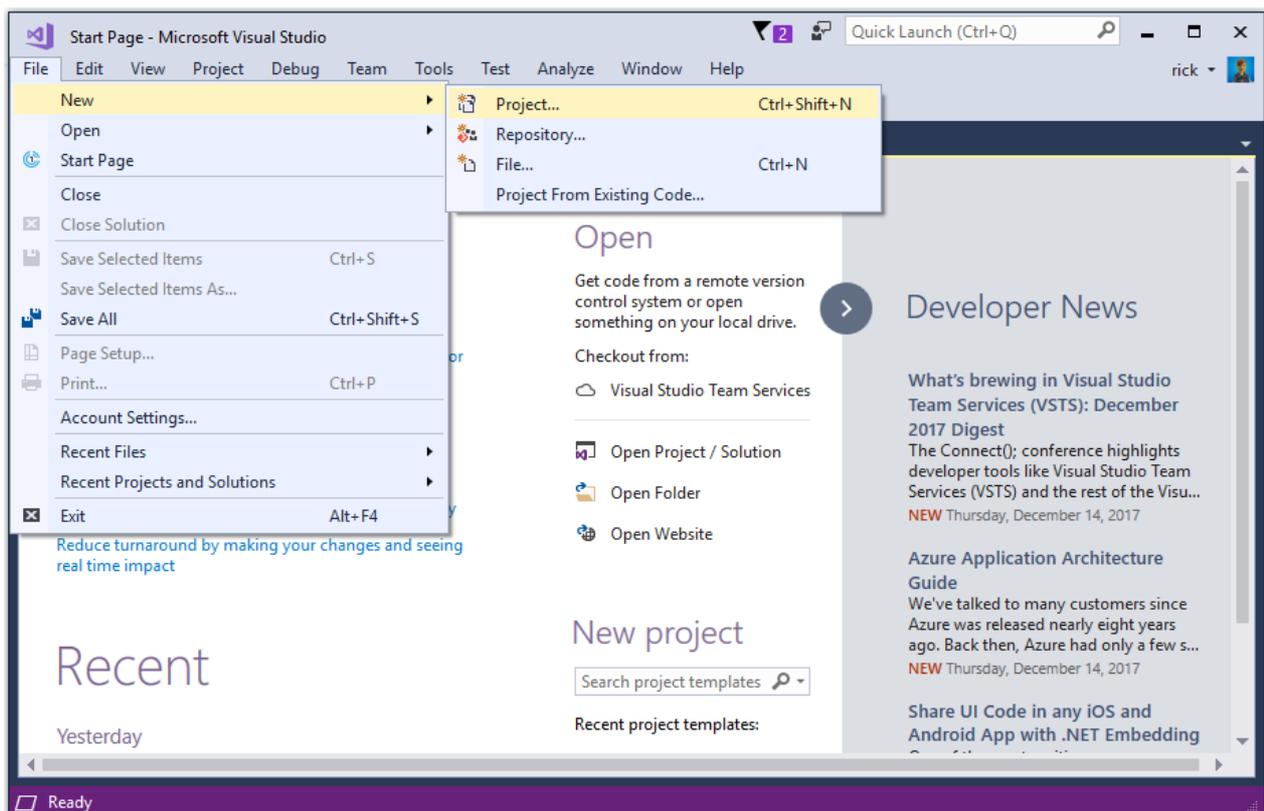
See [Publish to Azure from Visual Studio for Mac](#) if you are working on a Mac.

Set up

- Open a [free Azure account](#) if you do not have one.

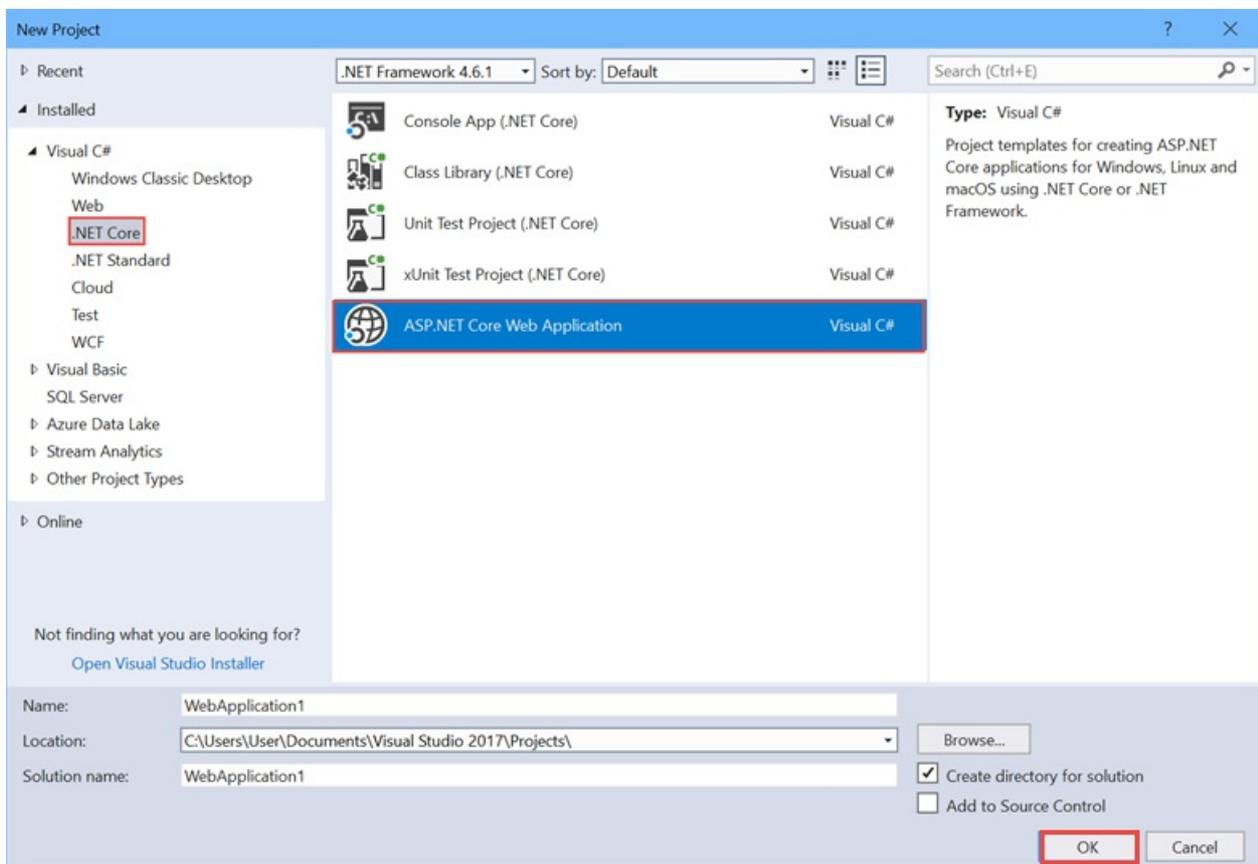
Create a web app

In the Visual Studio Start Page, select **File > New > Project...**



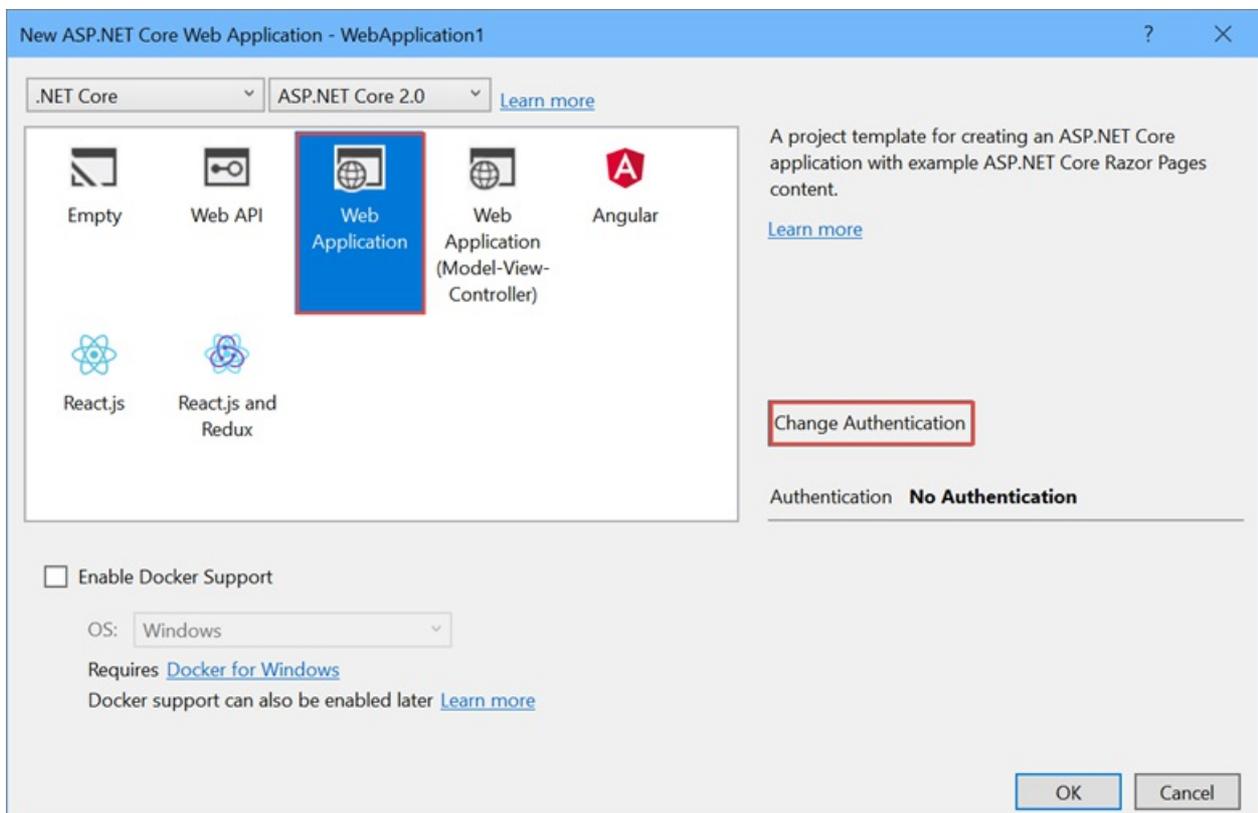
Complete the **New Project** dialog:

- In the left pane, select **.NET Core**.
- In the center pane, select **ASP.NET Core Web Application**.
- Select **OK**.



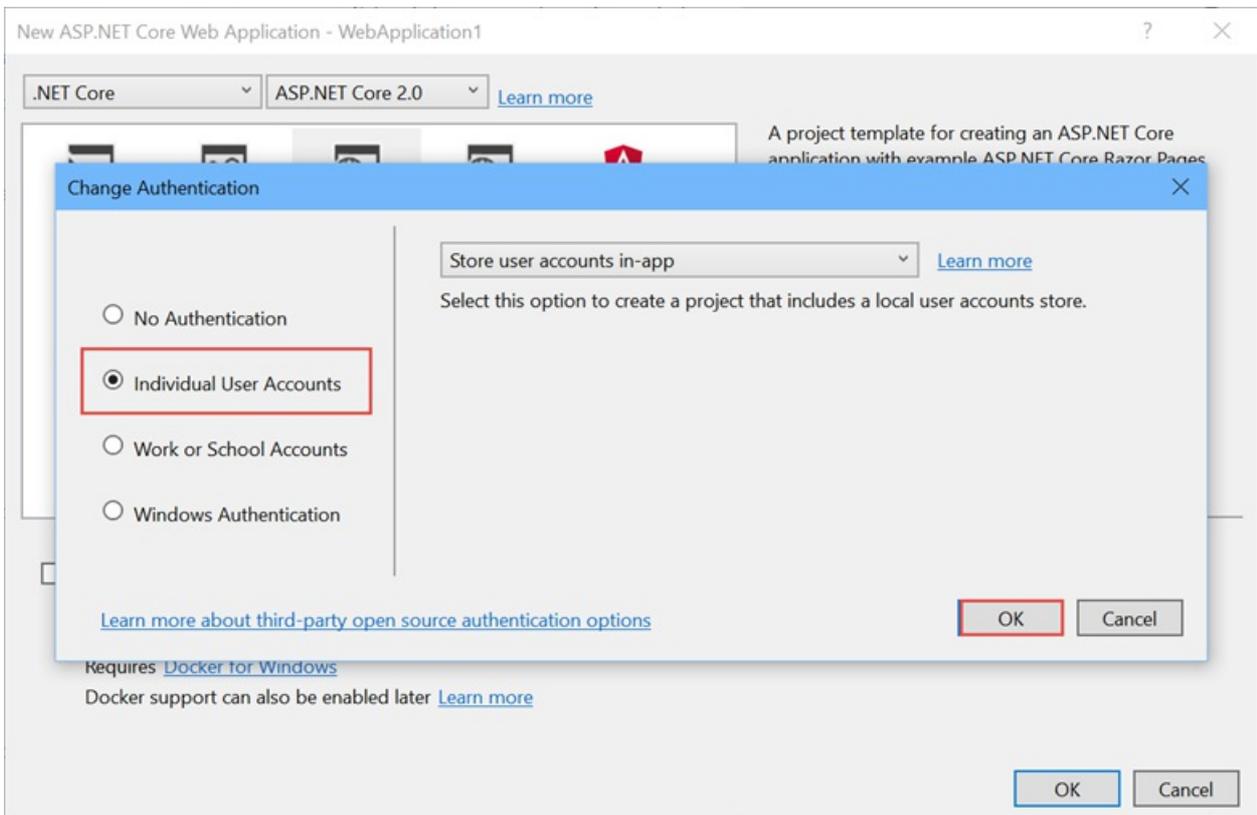
In the **New ASP.NET Core Web Application** dialog:

- Select **Web Application**.
- Select **Change Authentication**.



The **Change Authentication** dialog appears.

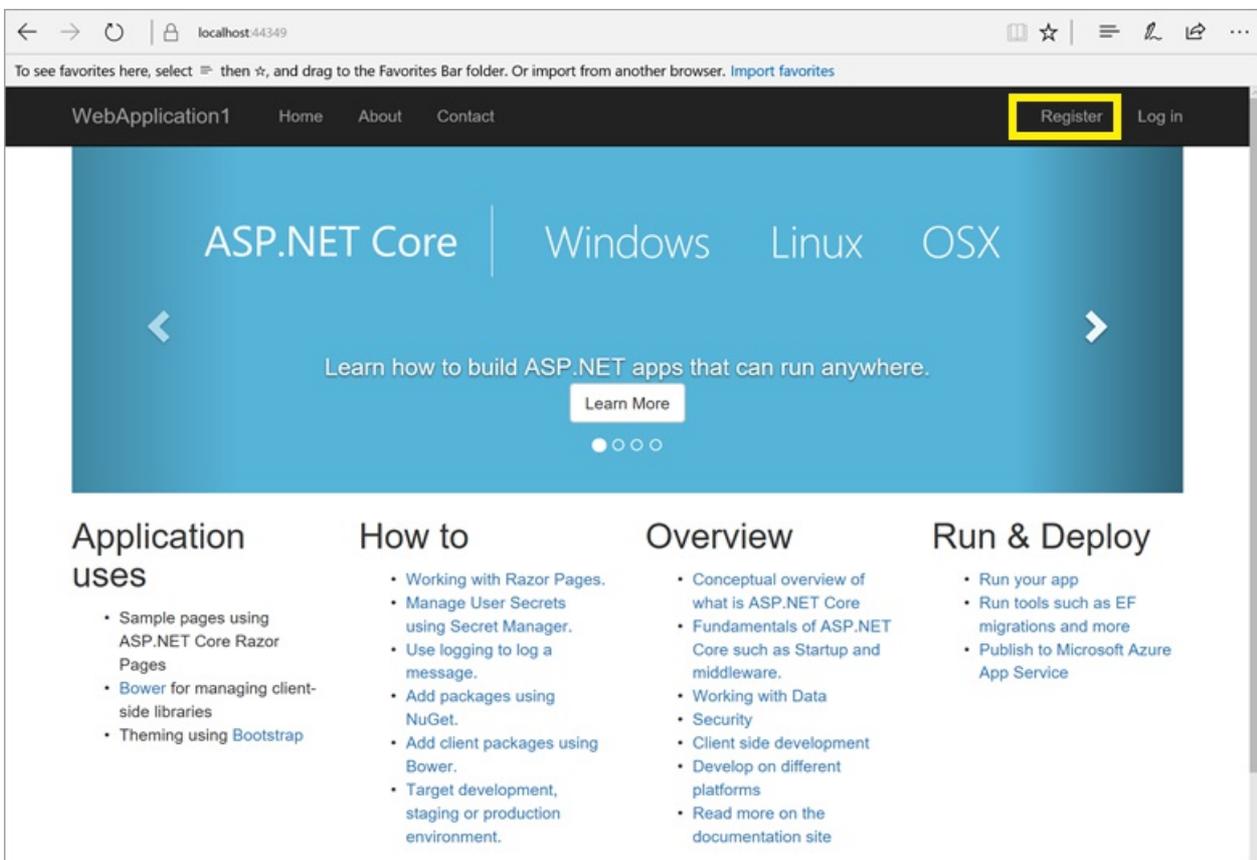
- Select **Individual User Accounts**.
- Select **OK** to return to the **New ASP.NET Core Web Application**, then select **OK** again.



Visual Studio creates the solution.

Run the app

- Press CTRL+F5 to run the project.
- Test the **About** and **Contact** links.

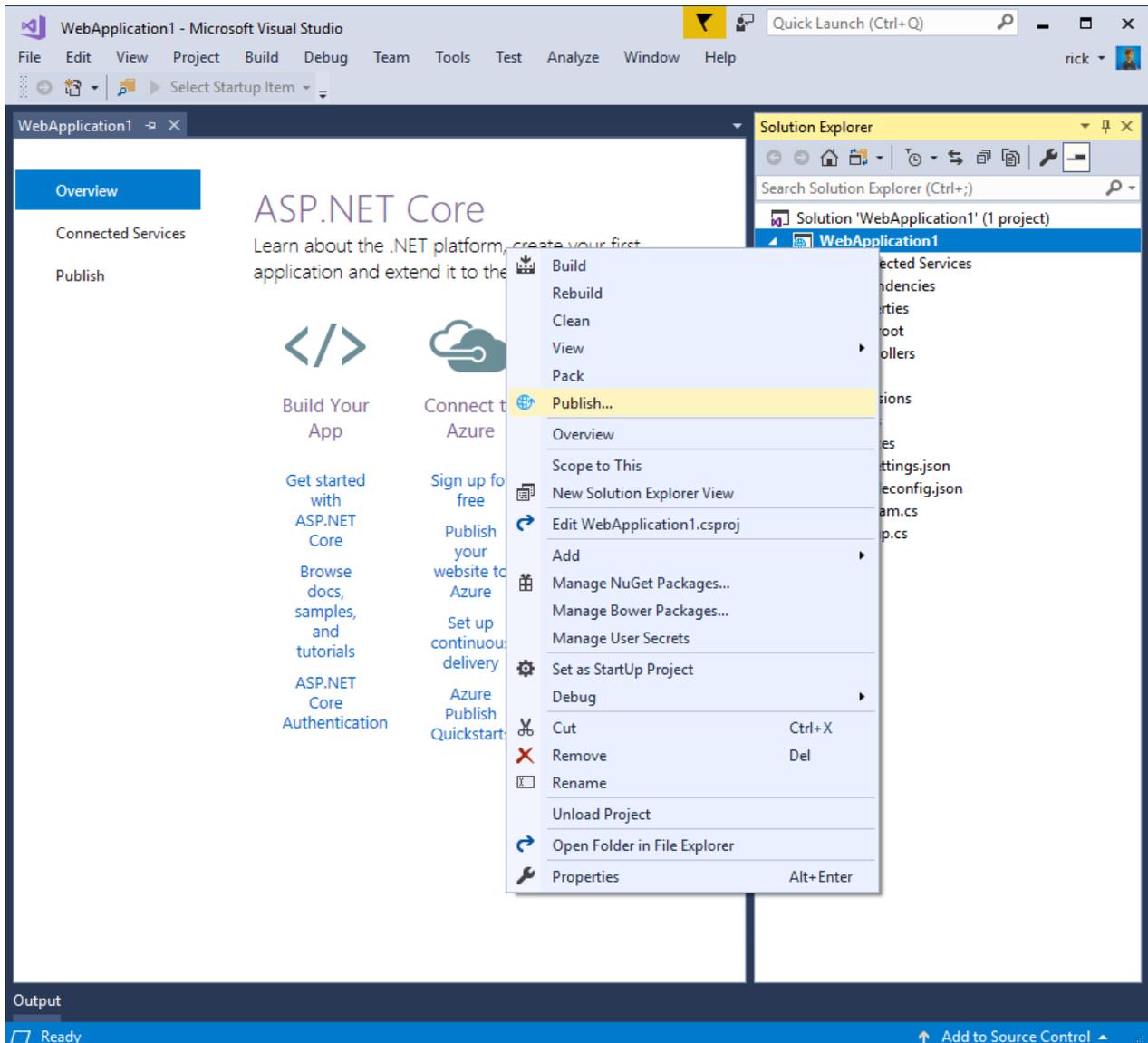


Register a user

- Select **Register** and register a new user. You can use a fictitious email address. When you submit, the page

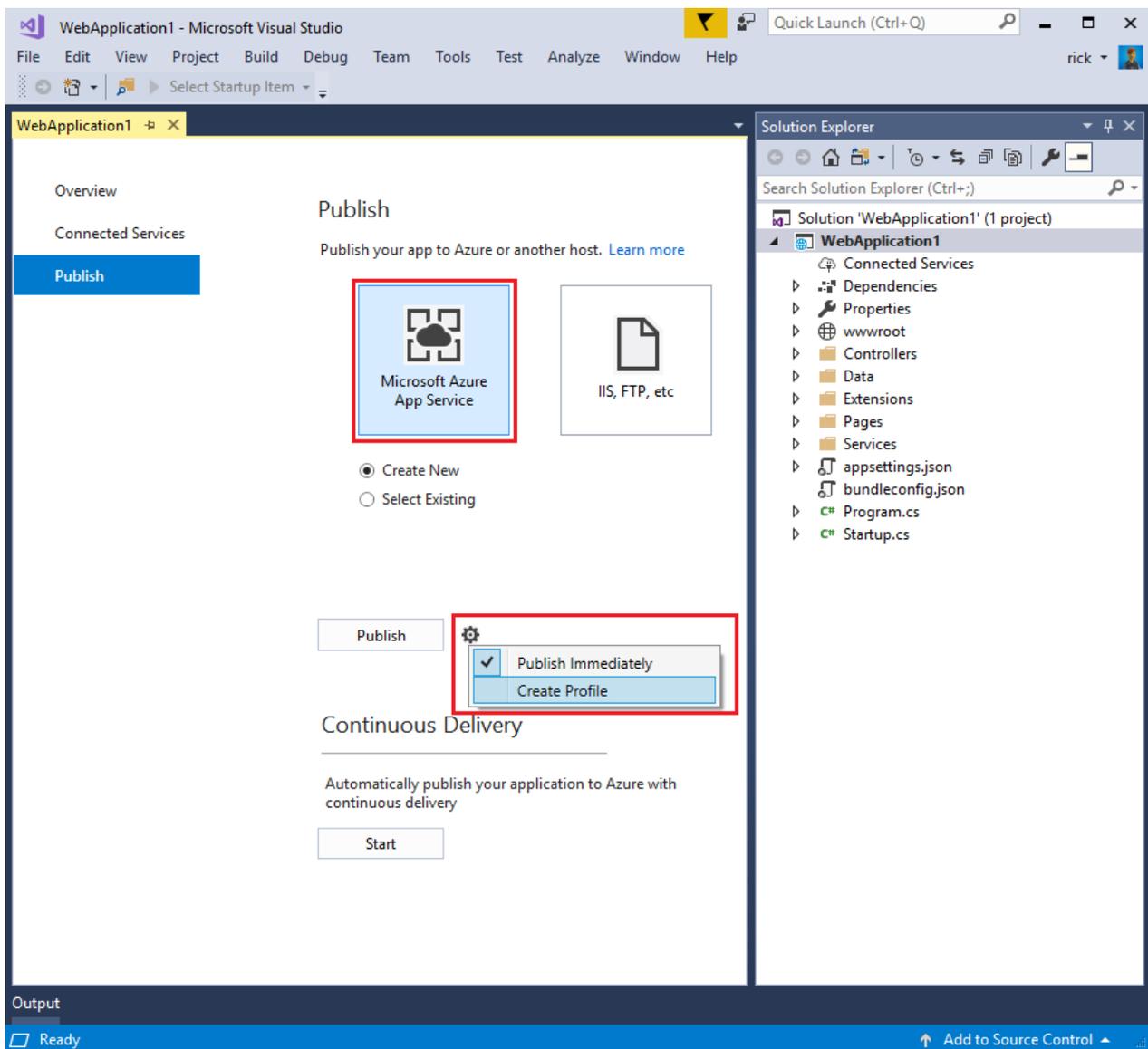
Deploy the app to Azure

Right-click on the project in Solution Explorer and select **Publish....**



In the **Publish** dialog:

- Select **Microsoft Azure App Service**.
- Select the gear icon and then select **Create Profile**.
- Select **Create Profile**.



Create Azure resources

The **Create App Service** dialog appears:

- Enter your subscription.
- The **App Name**, **Resource Group**, and **App Service Plan** entry fields are populated. You can keep these names or change them.

Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account
a.ricka00@gmail.com

Hosting ⓘ
Services

App Name Change Type ▾
WebApplication120171215025005

Subscription
MSDN ▾

Resource Group
WebApplication120171215025005ResourceGroup* New... ⓘ

App Service Plan
WebApplication120171215025005Plan* New...

Clicking the Create button will create the following Azure resources
[Explore additional Azure services](#)
App Service - WebApplication120171215025005
App Service Plan - WebApplication120171215025005Plan

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

Export... Create Cancel

- Select the **Services** tab to create a new database.
- Select the green + icon to create a new SQL Database

Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account
a.ricka00@gmail.com

Hosting ⓘ
Services

Select any additional Azure resources your app will need Show: Recommended ▾

Resource Type

 **SQL Database**
Scalable and managed database service for modern business-class apps + (highlighted)

Resources you've selected and configured

Resource Type

 **WebApplication120171215025005Plan**
App Service Plan ⚙️ ✕

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

Export... Create Cancel

- Select **New...** on the **Configure SQL Database** dialog to create a new database.

Configure SQL Database

Create a SQL Database in your subscription for storing data used by your application.

SQL Server

webapplication1dbserver*

Administrator Username

sqladmin

Administrator Password

.....

Database Name

WebApplication1-123_db

Connection String Name

DefaultConnection

The **Configure SQL Server** dialog appears.

- Enter an administrator user name and password, and then select **OK**. You can keep the default **Server Name**.

NOTE

"admin" is not allowed as the administrator user name.

Configure SQL Server
Create a SQL Database in your subscription for storing data used by your application.

Server Name
webapplication120160701041926dbserver

Administrator Username
adminuser

Administrator Password
●●●●●●●●

Administrator Password (confirm)
●●●●●●●●

OK Cancel

- Select **OK**.

Visual Studio returns to the **Create App Service** dialog.

- Select **Create** on the **Create App Service** dialog.

Create App Service
Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account user@microsoft.com

Hosting Services

Select any additional Azure resources your app will need Show: Recommended

Resource Type

- SQL Database Scalable and managed database service for modern business-class apps +

Resources you've selected and configured

Resource Type

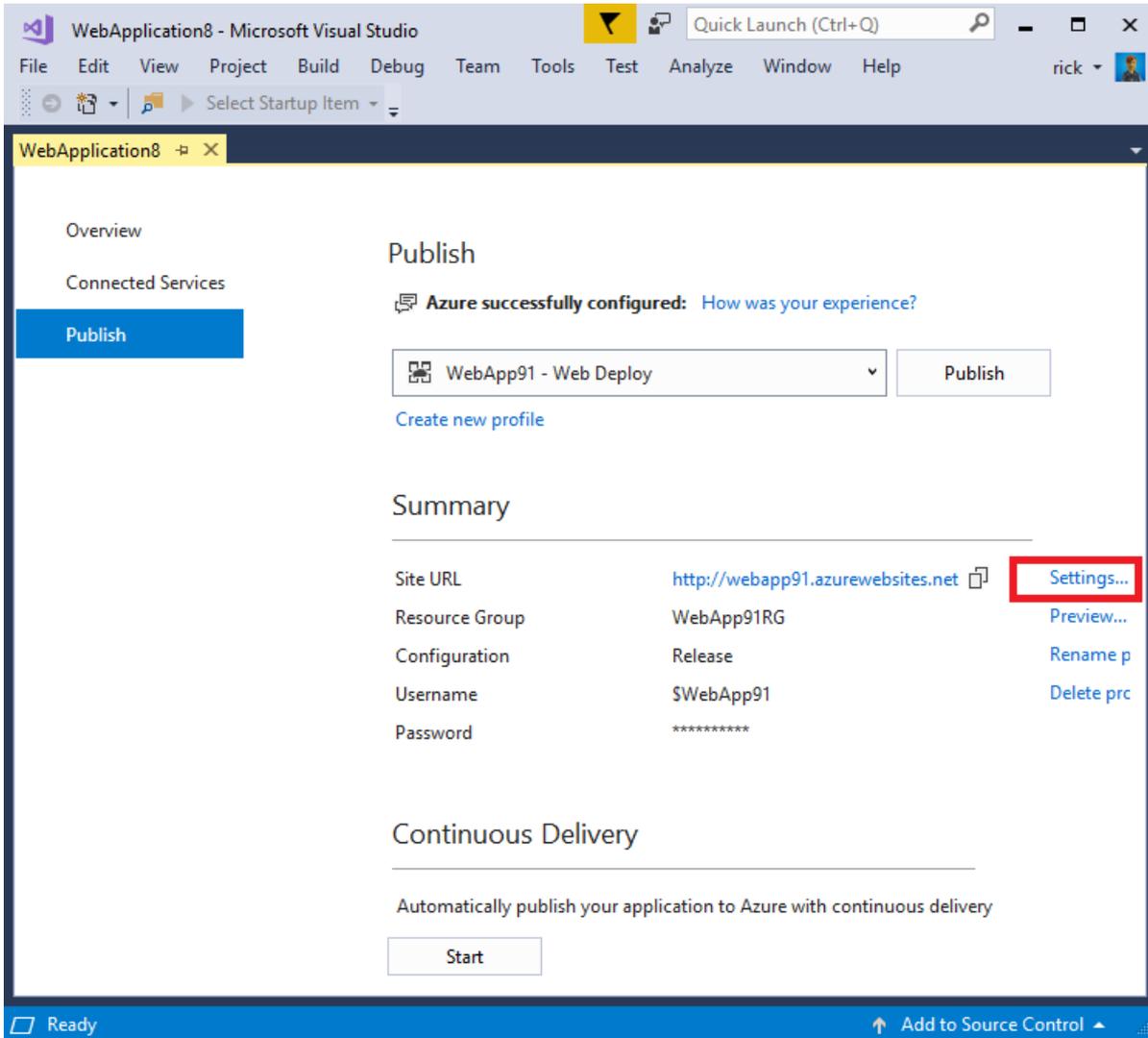
- WebApplication1Plan App Service Plan ⚙️ ✖️
- webapplication1dbserver SQL Server ⚙️ ✖️
- WebApplication1_db SQL Database ⚙️ ✖️

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources. [Learn More](#)

Export... Create Cancel

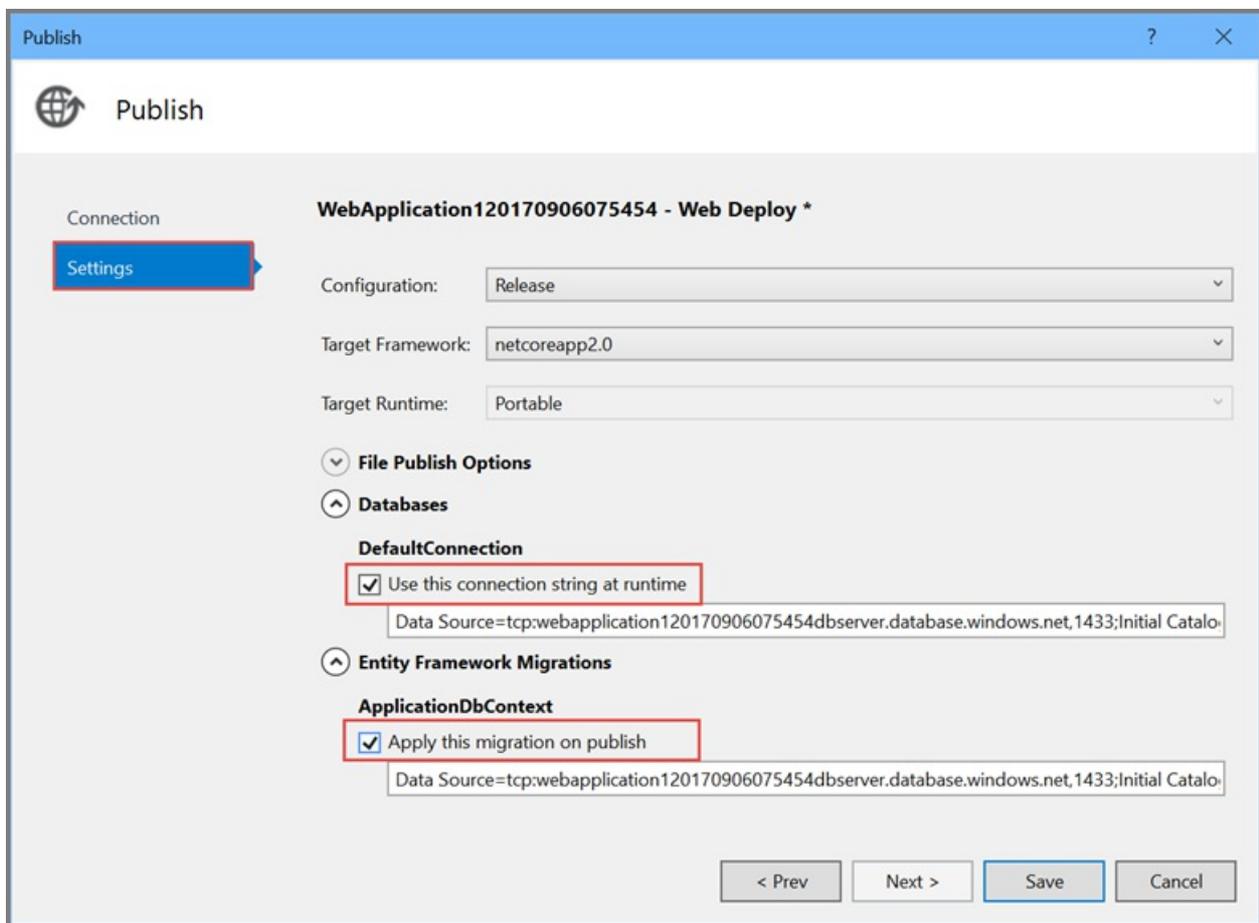
Visual Studio creates the Web app and SQL Server on Azure. This step can take a few minutes. For information on the resources created, see [Additional resources](#).

When deployment completes, select **Settings**:



On the **Settings** page of the **Publish** dialog:

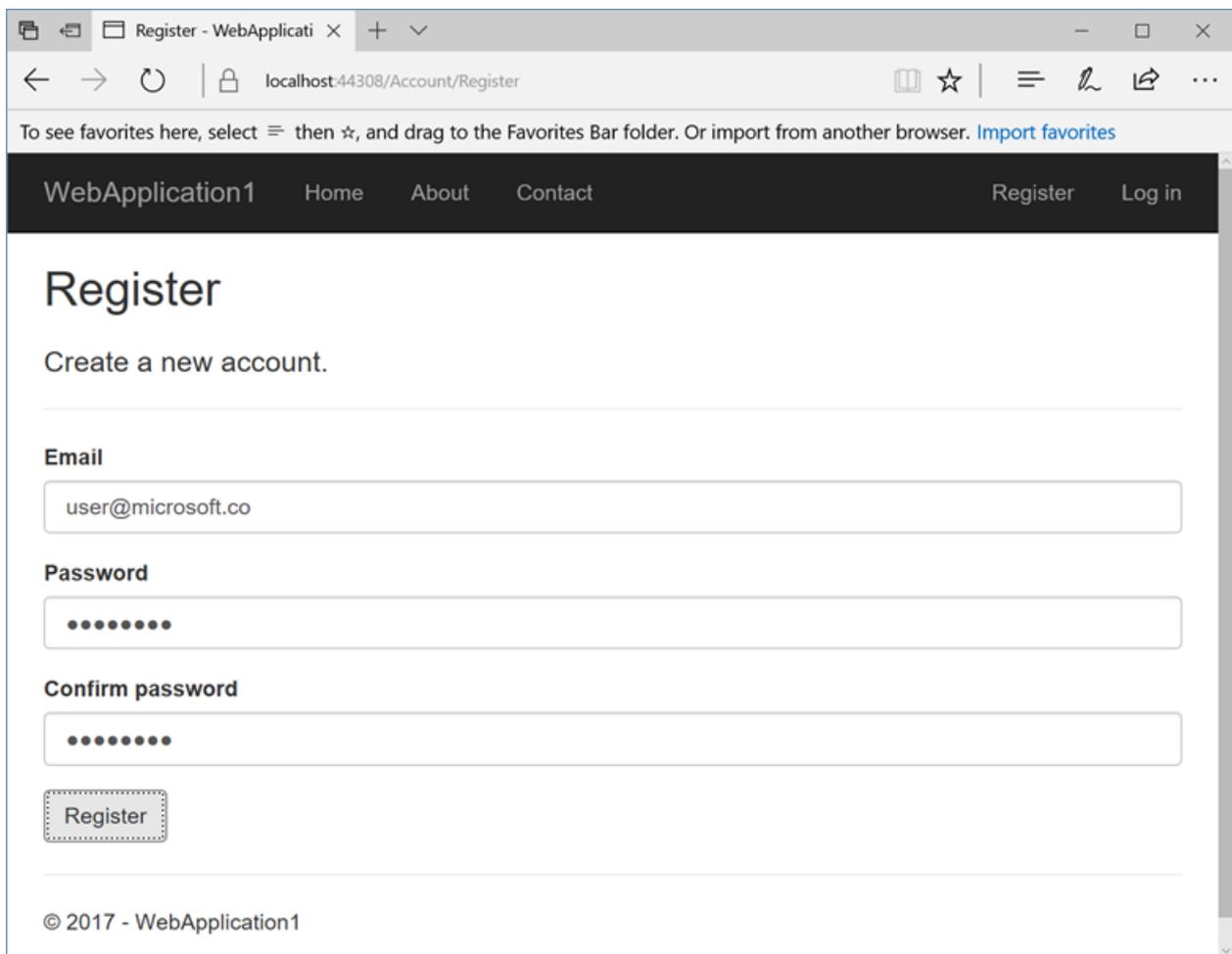
- Expand **Databases** and check **Use this connection string at runtime**.
- Expand **Entity Framework Migrations** and check **Apply this migration on publish**.
- Select **Save**. Visual Studio returns to the **Publish** dialog.



Click **Publish**. Visual Studio publishes your app to Azure. When the deployment completes, the app is opened in a browser.

Test your app in Azure

- Test the **About** and **Contact** links
- Register a new user



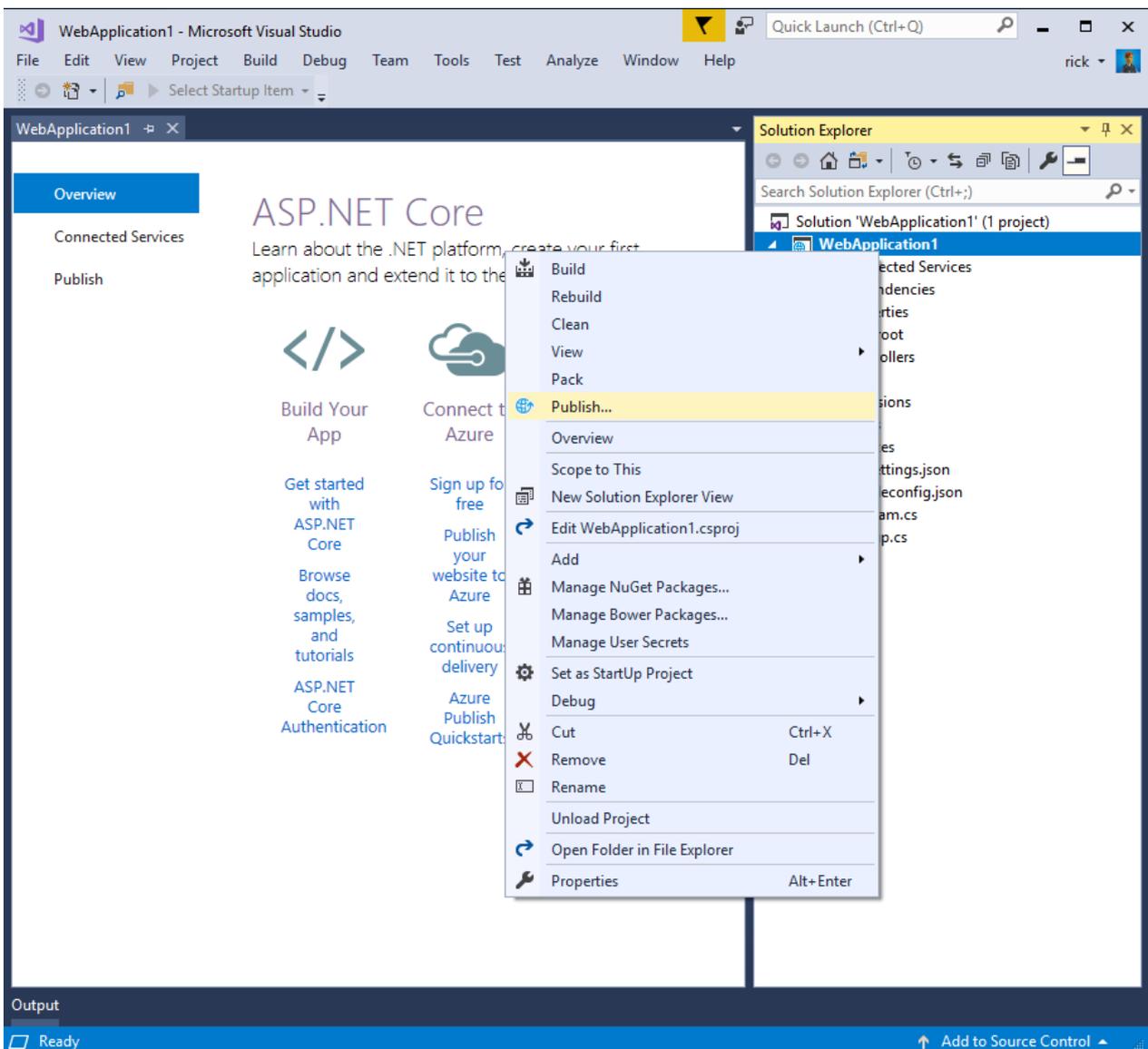
Update the app

- Edit the `Pages/About.cshtml` Razor page and change its contents. For example, you can modify the paragraph to say "Hello ASP.NET Core!":

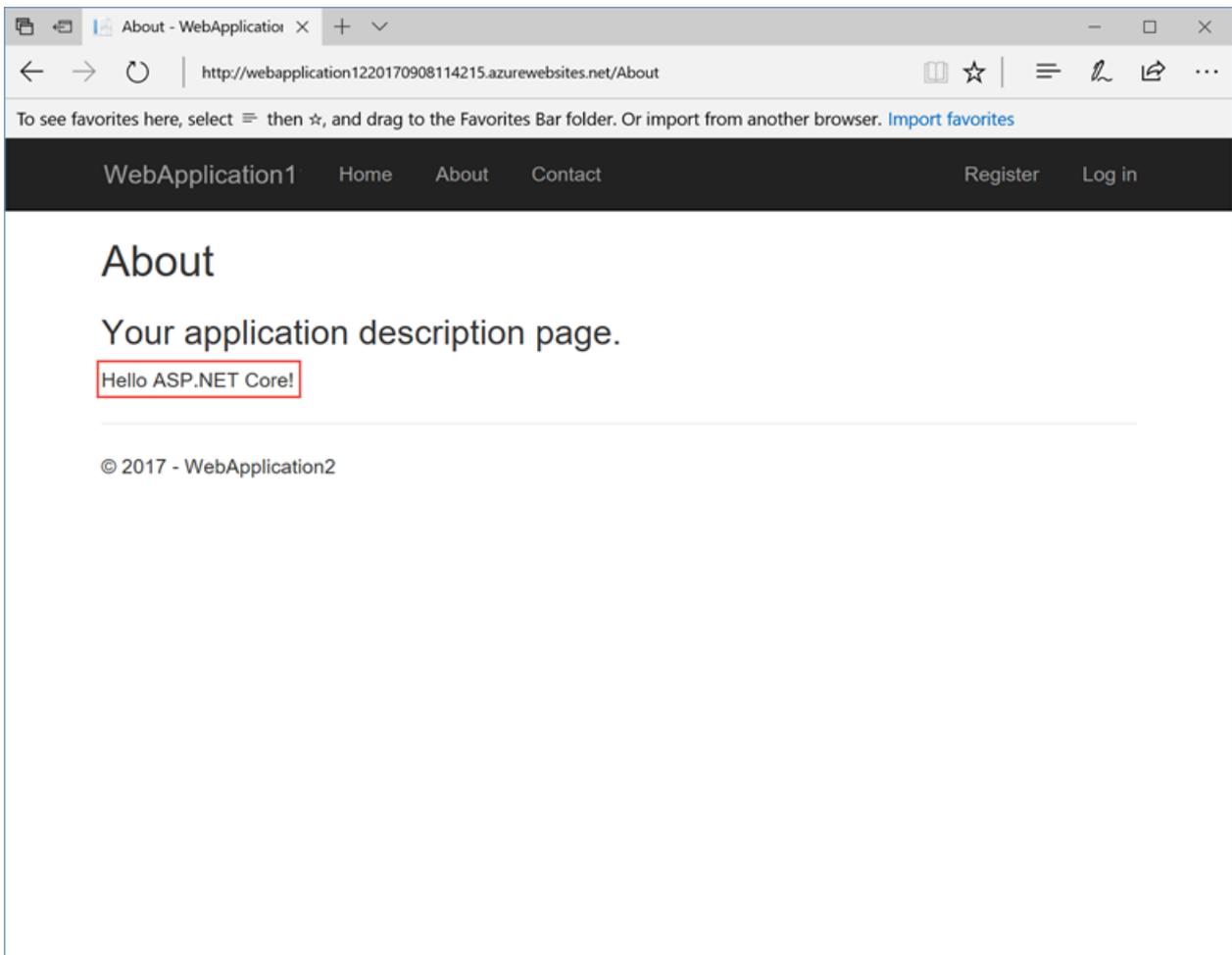
```
@page
@model AboutModel
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"]</h2>
<h3>@Model.Message</h3>

<p>Hello ASP.NET Core!</p>
```

- Right-click on the project and select **Publish...** again.



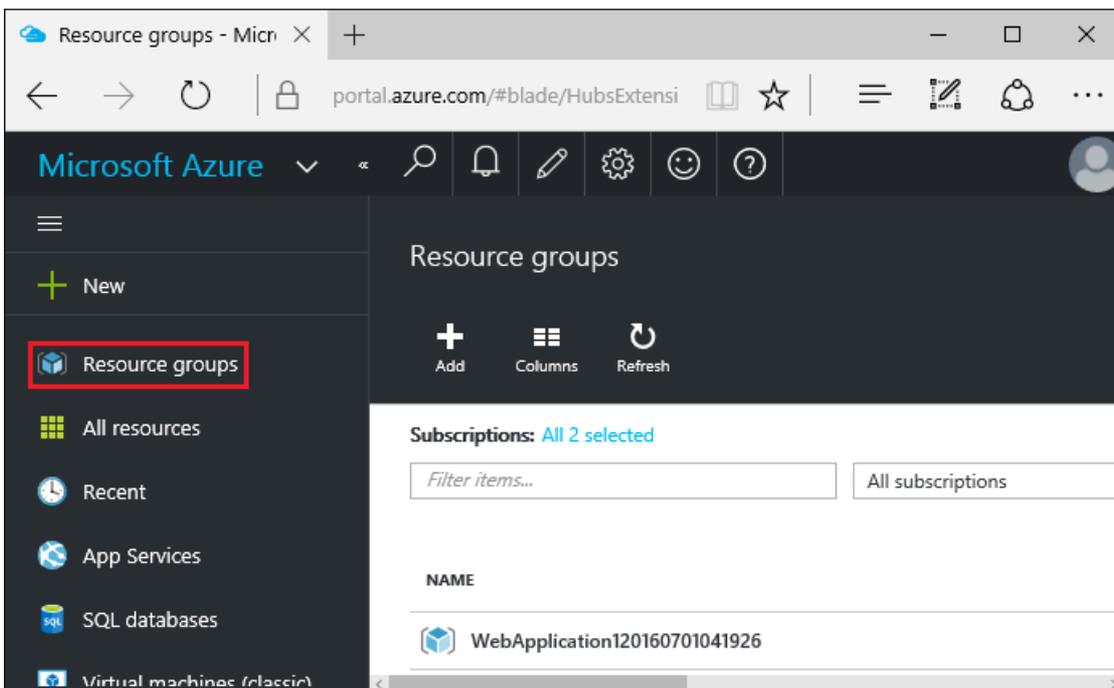
- After the app is published, verify the changes you made are available on Azure.



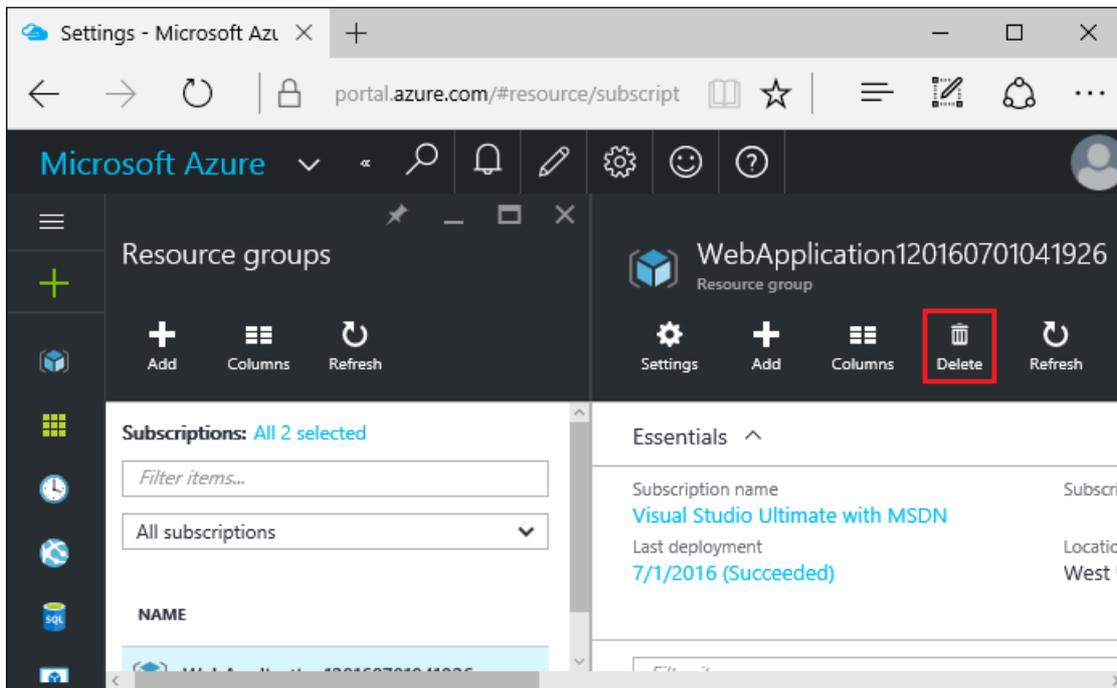
Clean up

When you have finished testing the app, go to the [Azure portal](#) and delete the app.

- Select **Resource groups**, then select the resource group you created.



- In the **Resource groups** page, select **Delete**.



- Enter the name of the resource group and select **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure.

Next steps

- [Continuous Deployment to Azure with Visual Studio and Git](#)

Additional resources

- [Azure App Service](#)
- [Azure resource groups](#)
- [Azure SQL Database](#)

Deploy an ASP.NET Core application to Azure App Service from the command line

1/10/2018 • 3 min to read • [Edit Online](#)

By [Cam Soper](#)

This tutorial will show you how to build and deploy an ASP.NET Core application to Microsoft Azure App Service using command line tools. When finished, you'll have a web application built in ASP.NET MVC Core hosted as an Azure App Service Web App. This tutorial is written using Windows command line tools, but can be applied to macOS and Linux environments, as well.

In this tutorial, you learn how to:

- Create an Azure App Service website using Azure CLI
- Deploy an ASP.NET Core application to Azure App Service using the Git command line tool

Prerequisites

To complete this tutorial, you'll need:

- A [Microsoft Azure subscription](#)
- [.NET Core](#)
- [Git](#) command line client

Create a web application

Create a new directory for the web application, create a new ASP.NET Core MVC application, and then run the website locally.

- [Windows](#)
- [Other](#)

```
REM Create a new ASP.NET Core MVC application
dotnet new razor -o MyApplication
```

```
REM Change to the new directory that was just created
cd MyApplication
```

```
REM Run the application
dotnet run
```

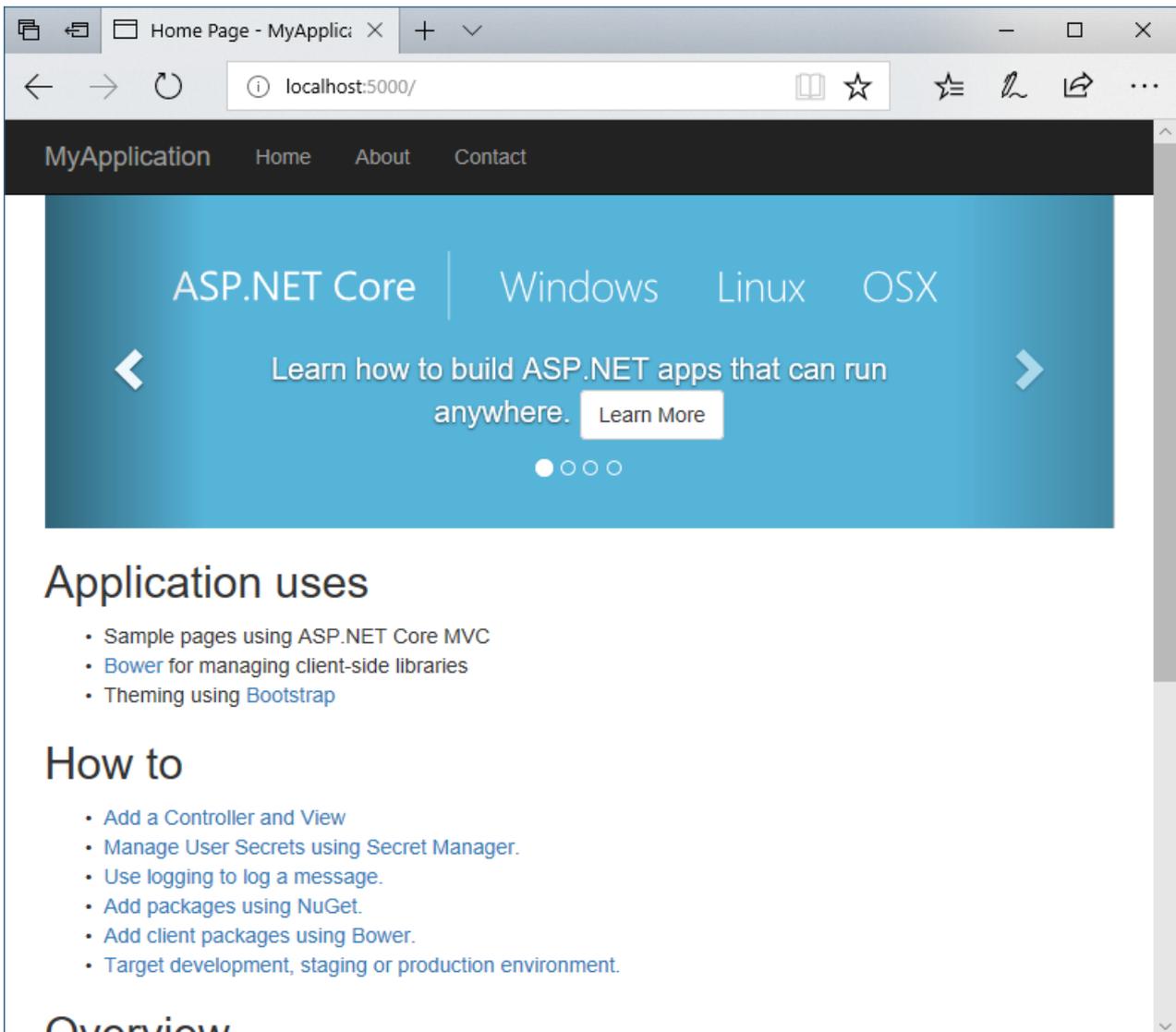
```
Command Prompt - dotnet run
C:\>dotnet new razor -o MyApplication
The template "ASP.NET Core Web App" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/template-3pn for details.
Processing post-creation actions...
Running 'dotnet restore' on MyApplication\MyApplication.csproj...
  Restoring packages for C:\MyApplication\MyApplication.csproj...
  Restore completed in 42.37 ms for C:\MyApplication\MyApplication.csproj.
  Generating MSBuild file C:\MyApplication\obj\MyApplication.csproj.nuget.g.props.
  Generating MSBuild file C:\MyApplication\obj\MyApplication.csproj.nuget.g.targets.
  Restore completed in 1.5 sec for C:\MyApplication\MyApplication.csproj.

Restore succeeded.

C:\>cd MyApplication

C:\MyApplication>dotnet run
Hosting environment: Production
Content root path: C:\MyApplication
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Test the application by browsing to <http://localhost:5000>.



Create the Azure App Service instance

Using the [Azure Cloud Shell](#), create a resource group, App Service plan, and an App Service web app.

```
# Generate a unique Web App name
let randomNum=$RANDOM*$RANDOM
webappname=tutorialApp$randomNum

# Create the DotNetAzureTutorial resource group
az group create --name DotNetAzureTutorial --location EastUS

# Create an App Service plan.
az appservice plan create --name $webappname --resource-group DotNetAzureTutorial --sku FREE

# Create the Web App
az webapp create --name $webappname --resource-group DotNetAzureTutorial --plan $webappname
```

Before deployment, set the account-level deployment credentials using the following command:

```
az webapp deployment user set --user-name <desired user name> --password <desired password>
```

A deployment URL is needed to deploy the application using Git. Retrieve the URL like this.

```
az webapp deployment source config-local-git -n $webappname -g DotNetAzureTutorial --query [url] -o tsv
```

Note the displayed URL ending in `.git`. It's used in the next step.

Deploy the application using Git

You're ready to deploy from your local machine using Git.

NOTE

It's safe to ignore any warnings from Git about line endings.

- [Windows](#)
- [Other](#)

```
REM Initialize the local Git repository
git init

REM Add the contents of the working directory to the repo
git add --all

REM Commit the changes to the local repo
git commit -a -m "Initial commit"

REM Add the URL as a Git remote repository
git remote add azure <THE GIT URL YOU NOTED EARLIER>

REM Push the local repository to the remote
git push azure master
```

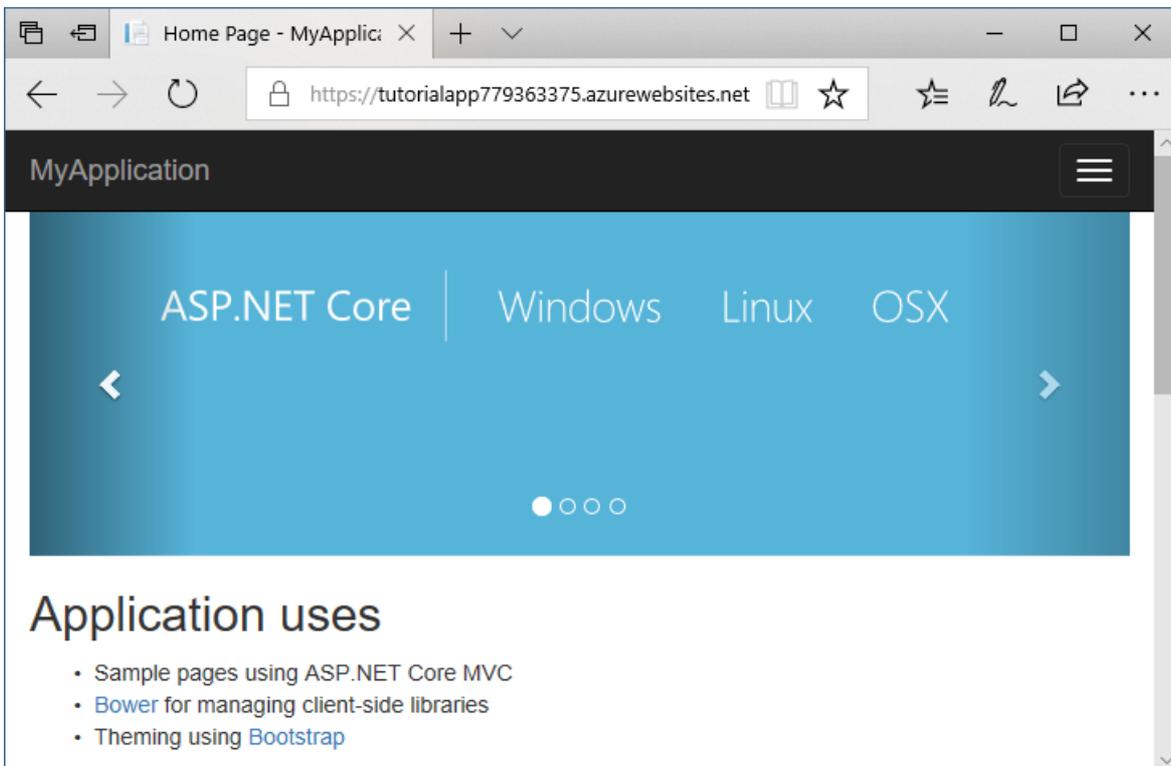
Git will prompt for the deployment credentials that were set earlier. After authenticating, the application will be pushed to the remote location, built, and deployed.

```
Command Prompt
remote: Copying file: 'wwwroot\lib\bootstrap\dist\css\bootstrap.css.map'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\css\bootstrap.min.css'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\css\bootstrap.min.css.map'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicons-halflings-regular.eot'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicons-halflings-regular.svg'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicons-halflings-regular.ttf'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicons-halflings-regular.woff'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicons-halflings-regular.woff2'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\js\bootstrap.js'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\js\bootstrap.min.js'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\js\npm.js'
remote: Copying file: 'wwwroot\lib\jquery\bower.json'
remote: Copying file: 'wwwroot\lib\jquery\LICENSE.txt'
remote: Copying file: 'wwwroot\lib\jquery\dist\jquery.js'
remote: Copying file: 'wwwroot\lib\jquery\dist\jquery.min.js'
remote: Copying file: 'wwwroot\lib\jquery\dist\jquery.min.map'
remote: Copying file: 'wwwroot\lib\jquery-validation\bower.json'
remote: Copying file: 'wwwroot\lib\jquery-validation\LICENSE.md'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\additional-methods.js'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\additional-methods.min.js'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\jquery.validate.js'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\jquery.validate.min.js'
remote: Omitting next output lines...
remote: Finished successfully.
remote: Running post deployment command(s)...
remote: Deployment successful.
To https://tutorialapp779363375.scm.azurewebsites.net/tutorialApp779363375.git
* [new branch]      master -> master
C:\MyApplication>
```

Test the application

Test the application by browsing to `https://<web app name>.azurewebsites.net`. To display the address in the Cloud Shell (or Azure CLI), use the following:

```
az webapp show -n $webappname -g DotNetAzureTutorial --query defaultHostName -o tsv
```



Clean up

When finished testing the app and inspecting the code and resources, delete the web app and plan by deleting the resource group.

```
az group delete -n DotNetAzureTutorial
```

Next steps

In this tutorial, you learned how to:

- Create an Azure App Service website using Azure CLI
- Deploy an ASP.NET Core application to Azure App Service using the Git command line tool

Next, learn to use the command line to deploy an existing web app that uses CosmosDB.

[Deploy to Azure from the command line with .NET Core](#)

Continuous deployment to Azure for ASP.NET Core with Visual Studio and Git

1/10/2018 • 6 min to read • [Edit Online](#)

By [Erik Reitan](#)

This tutorial shows how to create an ASP.NET Core web app using Visual Studio and deploy it from Visual Studio to Azure App Service using continuous deployment.

See also [Use VSTS to Build and Publish to an Azure Web App with Continuous Deployment](#), which shows how to configure a continuous delivery (CD) workflow for [Azure App Service](#) using Visual Studio Team Services. Azure Continuous Delivery in Team Services simplifies setting up a robust deployment pipeline to publish updates for apps hosted in Azure App Service. The pipeline can be configured from the Azure portal to build, run tests, deploy to a staging slot, and then deploy to production.

NOTE

To complete this tutorial, a Microsoft Azure account is required. To obtain an account, [activate MSDN subscriber benefits](#) or [sign up for a free trial](#).

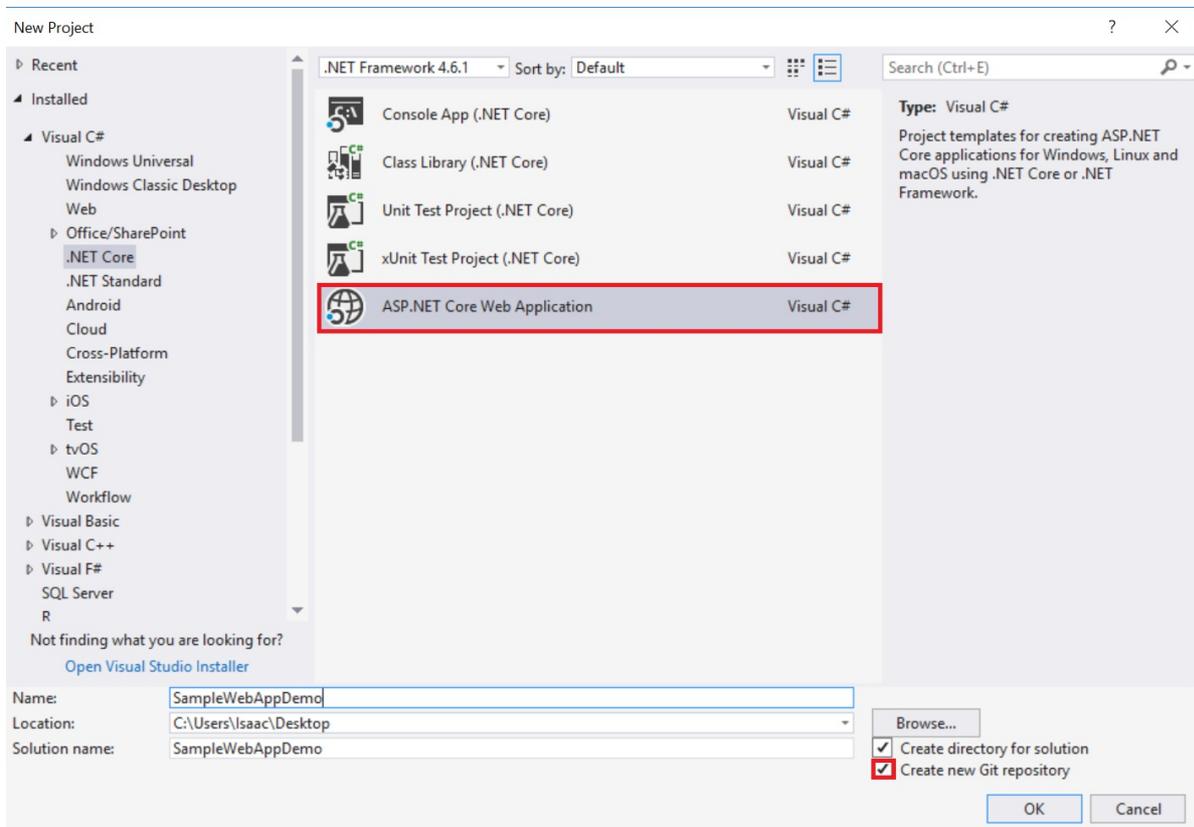
Prerequisites

This tutorial assumes the following software is installed:

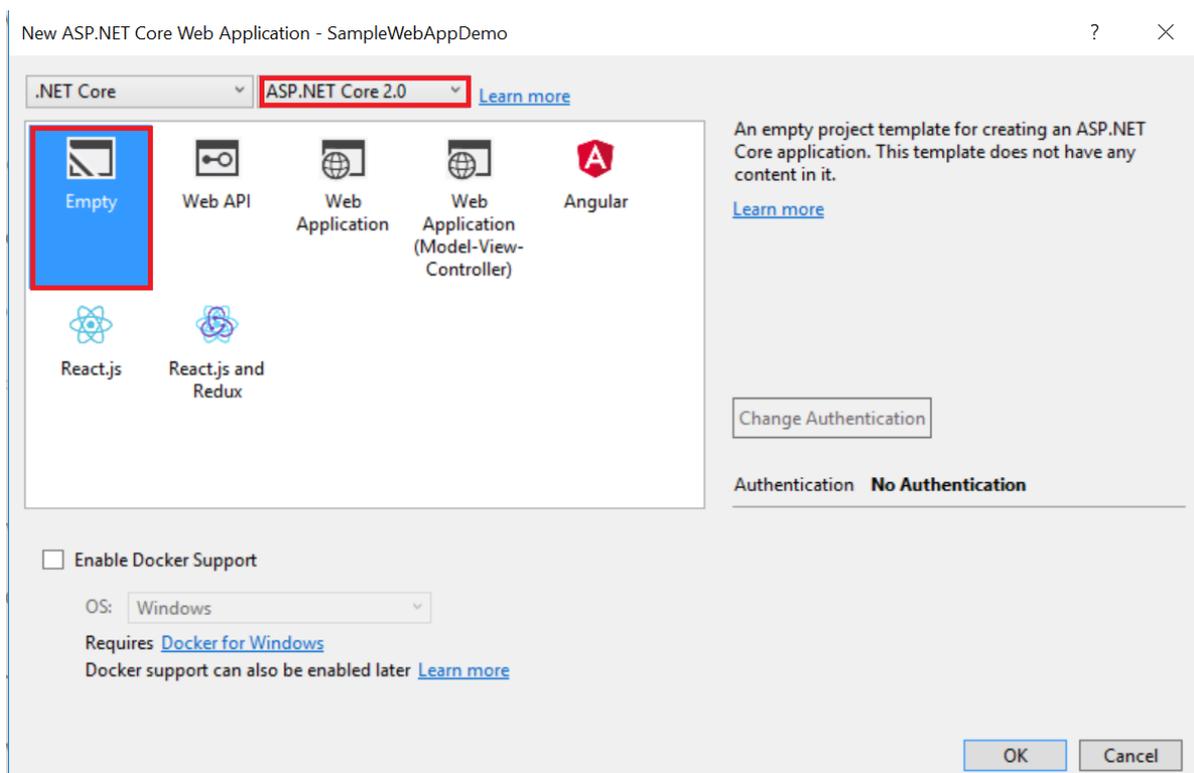
- [Visual Studio](#)
- [.NET Core SDK](#) (runtime and tooling)
- [Git](#) for Windows

Create an ASP.NET Core web app

1. Start Visual Studio.
2. From the **File** menu, select **New > Project**.
3. Select the **ASP.NET Core Web Application** project template. It appears under **Installed > Templates > Visual C# > .NET Core**. Name the project `SampleWebAppDemo`. Select the **Create new Git repository** option and click **OK**.



4. In the **New ASP.NET Core Project** dialog, select the ASP.NET Core **Empty** template, then click **OK**.



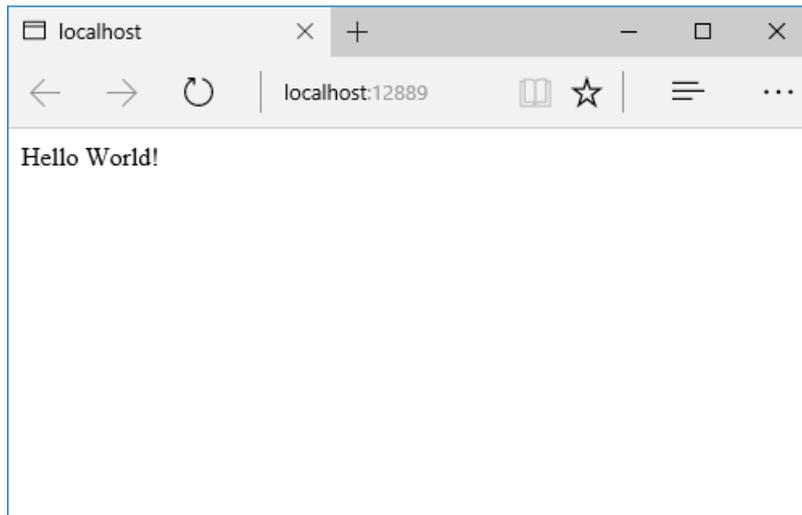
NOTE

The most recent release of .NET Core is 2.0.

Running the web app locally

1. Once Visual Studio finishes creating the app, run the app by selecting **Debug > Start Debugging**. As an alternative, press **F5**.

It may take time to initialize Visual Studio and the new app. Once it's complete, the browser shows the running app.

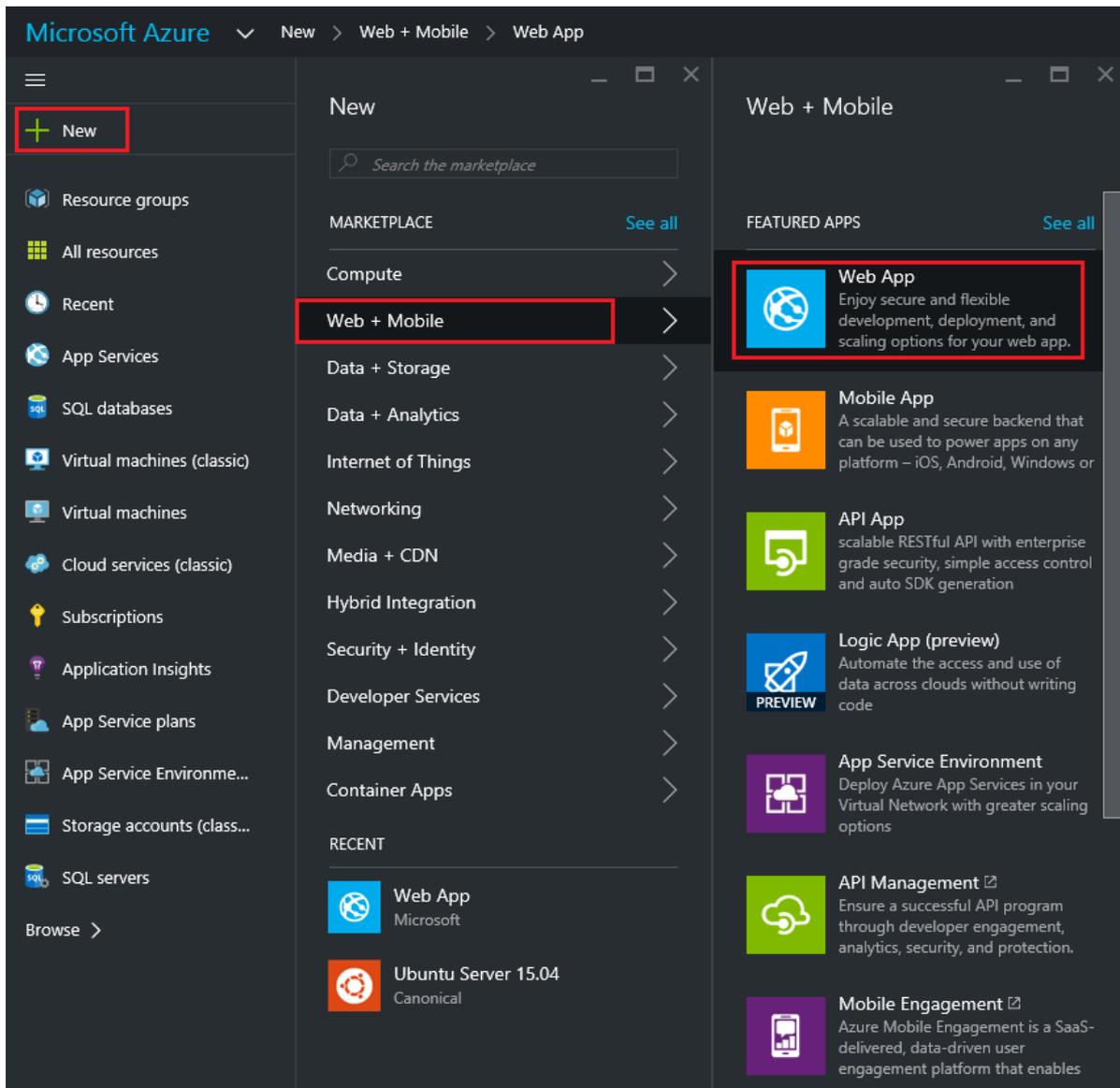


2. After reviewing the running Web app, close the browser and select the "Stop Debugging" icon in the toolbar of Visual Studio to stop the app.

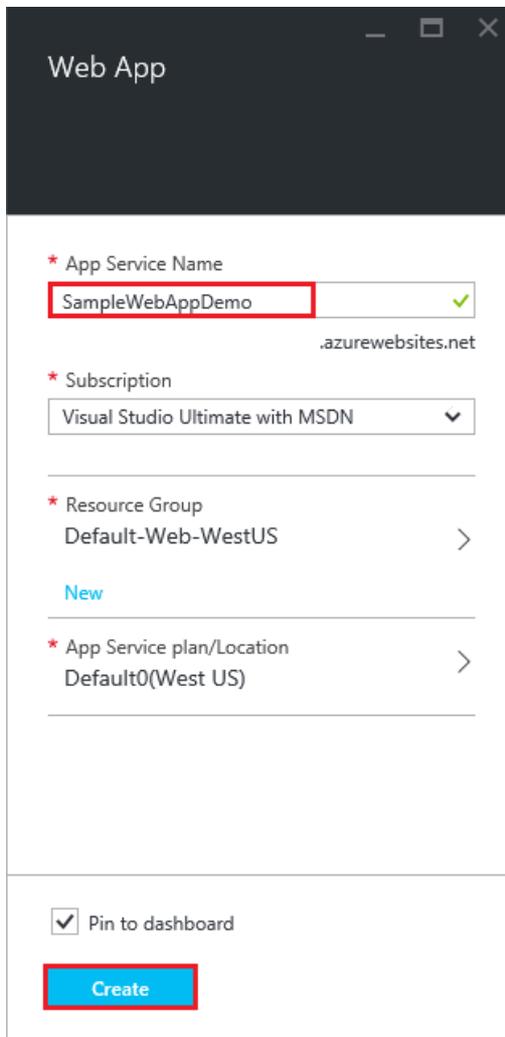
Create a web app in the Azure Portal

The following steps create a web app in the Azure Portal:

1. Log in to the [Azure Portal](#).
2. Select **NEW** at the top left of the portal interface.
3. Select **Web + Mobile > Web App**.



4. In the **Web App** blade, enter a unique value for the **App Service Name**.



Web App

* App Service Name
SampleWebAppDemo ✓
.azurewebsites.net

* Subscription
Visual Studio Ultimate with MSDN

* Resource Group
Default-Web-WestUS >
New

* App Service plan/Location
Default0(West US) >

Pin to dashboard

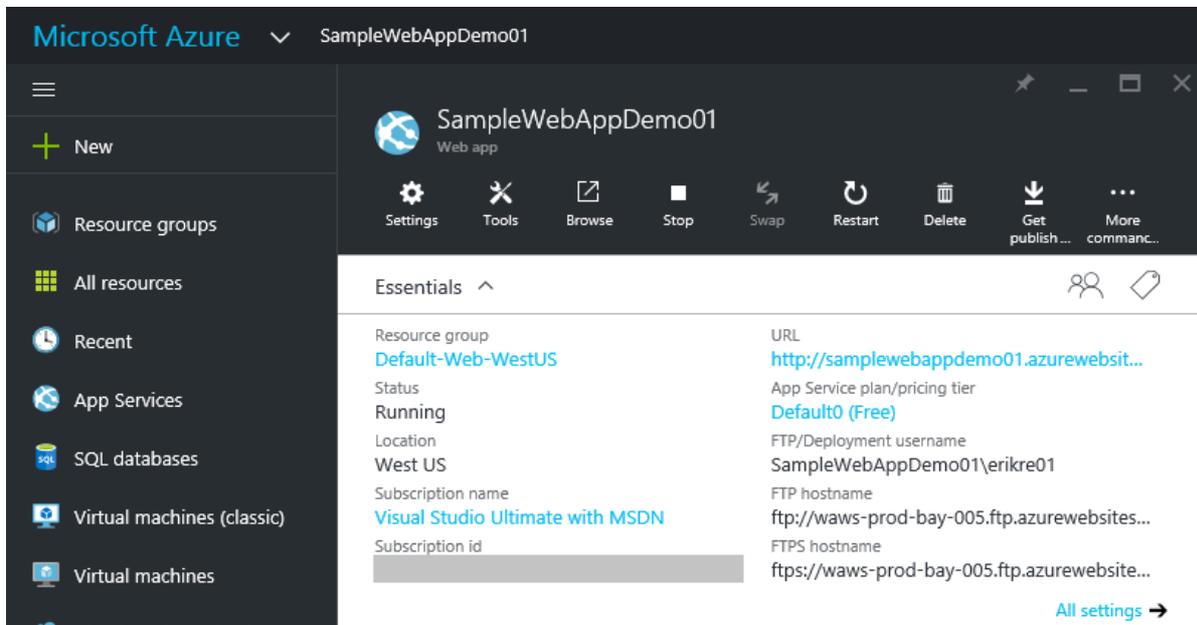
Create

NOTE

The **App Service Name** name must be unique. The portal enforces this rule when the name is provided. If providing a different value, substitute that value for each occurrence of **SampleWebAppDemo** in this tutorial.

Also in the **Web App** blade, select an existing **App Service Plan/Location** or create a new one. If creating a new plan, select the pricing tier, location, and other options. For more information on App Service plans, see [Azure App Service plans in-depth overview](#).

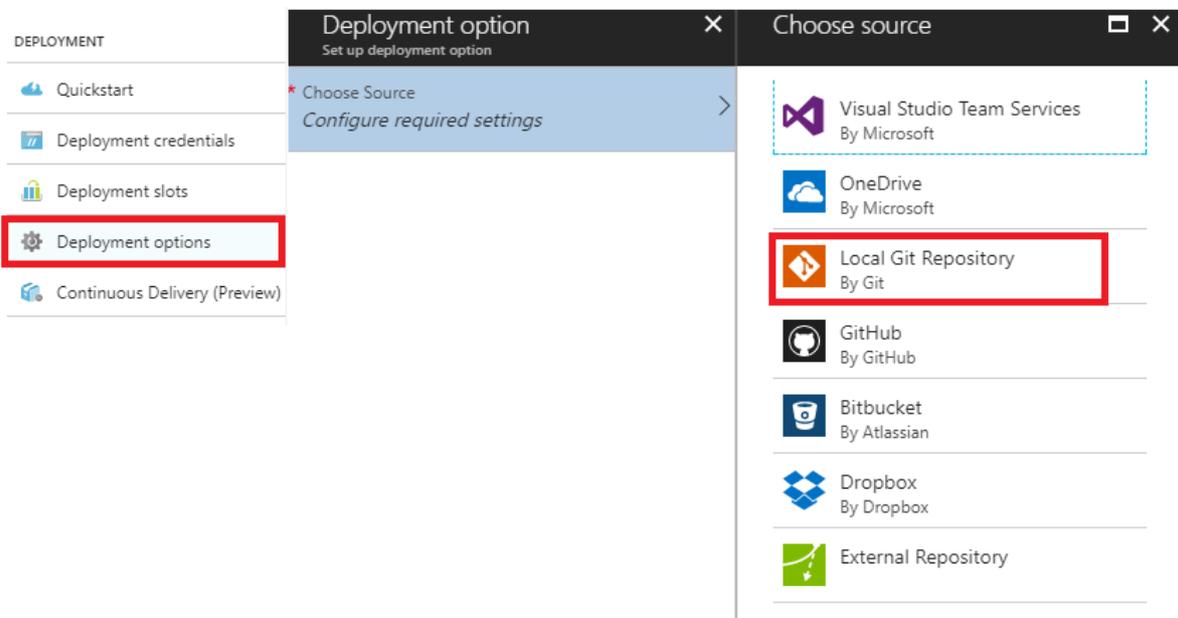
5. Select **Create**. Azure will provision and start the web app.



Enable Git publishing for the new web app

Git is a distributed version control system that can be used to deploy an Azure App Service web app. Web app code is stored in a local Git repository, and the code is deployed to Azure by pushing to a remote repository.

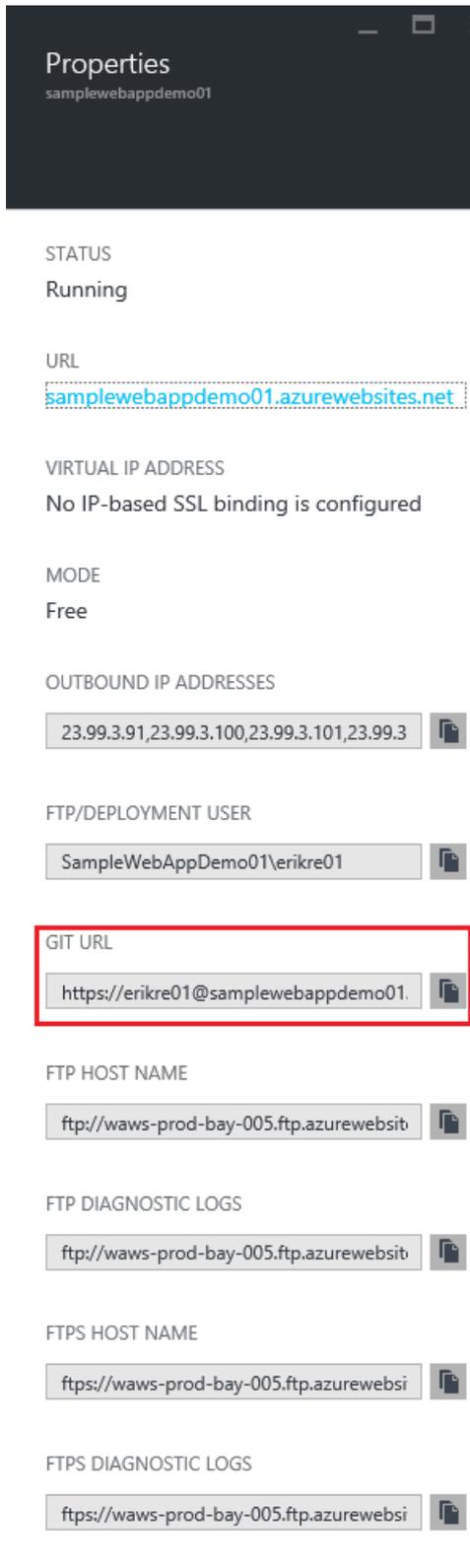
1. Log into the [Azure Portal](#).
2. Select **App Services** to view a list of the app services associated with the Azure subscription.
3. Select the web app created in the previous section of this tutorial.
4. In the **Deployment** blade, select **Deployment options** > **Choose Source** > **Local Git Repository**.



5. Select **OK**.
6. If deployment credentials for publishing a web app or other App Service app haven't previously been set up, set them up now:
 - Select **Settings** > **Deployment credentials**. The **Set deployment credentials** blade is displayed.
 - Create a user name and password. Save the password for later use when setting up Git.
 - Select **Save**.
7. In the **Web App** blade, select **Settings** > **Properties**. The URL of the remote Git repository to deploy to is

shown under **GIT URL**.

8. Copy the **GIT URL** value for later use in the tutorial.

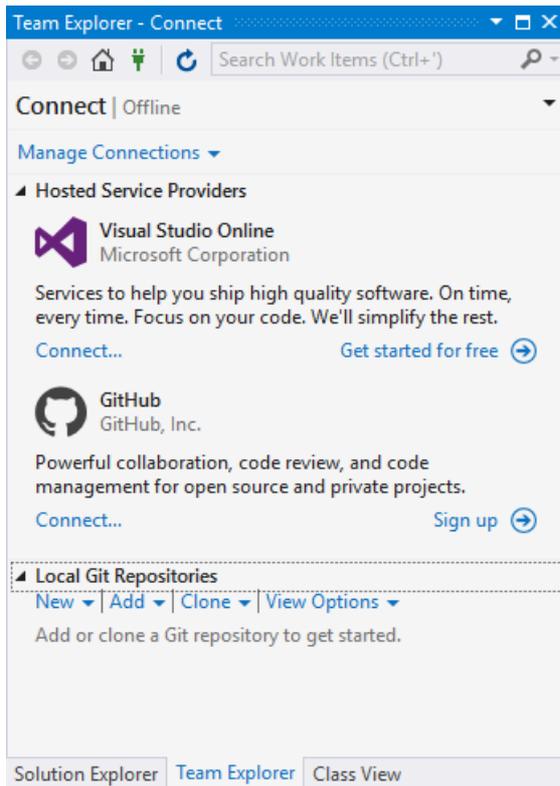


Publish the web app to Azure App Service

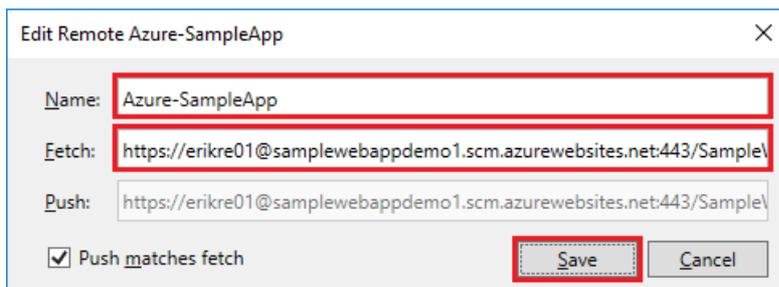
In this section, create a local Git repository using Visual Studio and push from that repository to Azure to deploy the web app. The steps involved include the following:

- Add the remote repository setting using the GIT URL value, so the local repository can be deployed to Azure.
- Commit project changes.
- Push project changes from the local repository to the remote repository on Azure.

1. In **Solution Explorer** right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** is displayed.



2. In **Team Explorer**, select the **Home** (home icon) > **Settings** > **Repository Settings**.
3. In the **Remotes** section of the **Repository Settings**, select **Add**. The **Add Remote** dialog box is displayed.
4. Set the **Name** of the remote to **Azure-SampleApp**.
5. Set the value for **Fetch** to the **Git URL** that copied from Azure earlier in this tutorial. Note that this is the URL that ends with **.git**.

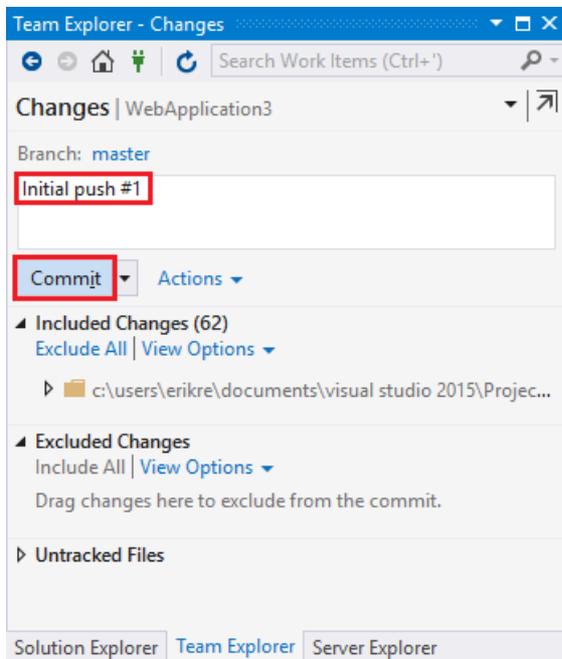


NOTE

As an alternative, specify the remote repository from the **Command Window** by opening the **Command Window**, changing to the project directory, and entering the command. Example:

```
git remote add Azure-SampleApp https://me@sampleapp.scm.azurewebsites.net:443/SampleApp.git
```

6. Select the **Home** (home icon) > **Settings** > **Global Settings**. Confirm that the name and email address are set. Select **Update** if required.
7. Select **Home** > **Changes** to return to the **Changes** view.
8. Enter a commit message, such as **Initial Push #1** and select **Commit**. This action creates a *commit* locally.



NOTE

As an alternative, commit changes from the **Command Window** by opening the **Command Window**, changing to the project directory, and entering the git commands. Example:

```
git add .
```

```
git commit -am "Initial Push #1"
```

9. Select **Home > Sync > Actions > Open Command Prompt**. The command prompt opens to the project directory.
10. Enter the following command in the command window:

```
git push -u Azure-SampleApp master
```

11. Enter the Azure **deployment credentials** password created earlier in Azure.

This command starts the process of pushing the local project files to Azure. The output from the above command ends with a message that the deployment was successful.

```
remote: Finished successfully.
remote: Running post deployment command(s)...
remote: Deployment successful.
To https://username@samplewebappdemo01.scm.azurewebsites.net:443/SampleWebAppDemo01.git
* [new branch]      master -> master
Branch master set up to track remote branch master from Azure-SampleApp.
```

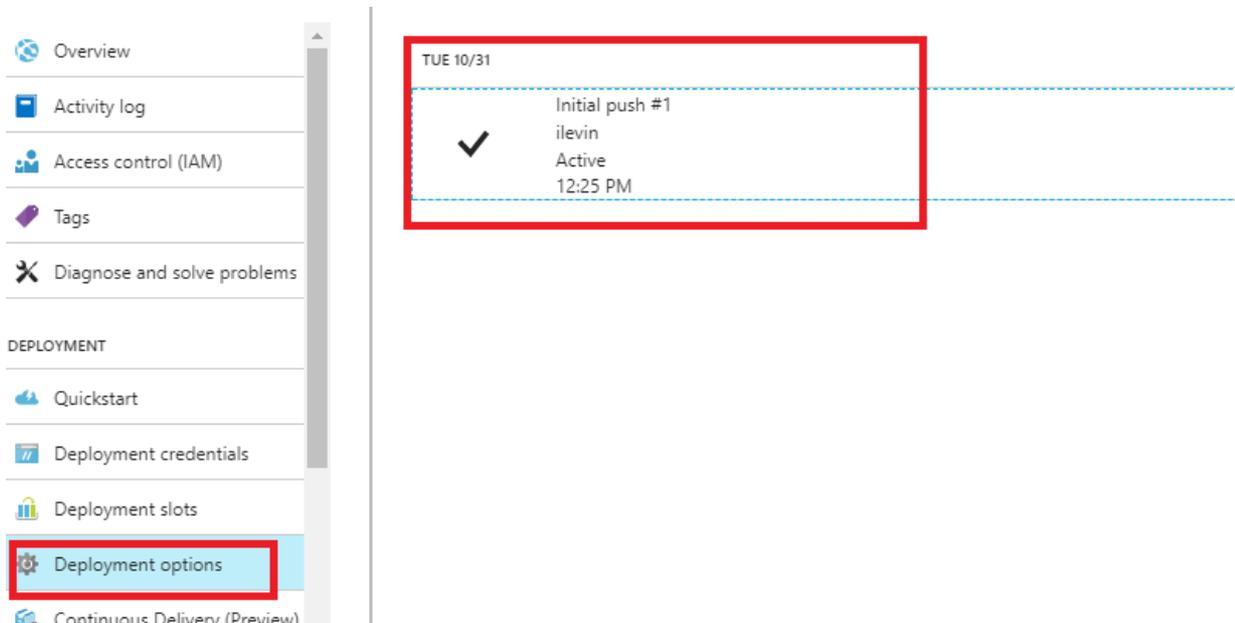
NOTE

If collaboration on the project is required, consider pushing to [GitHub](#) before pushing to Azure.

Verify the Active Deployment

Verify that the web app transfer from the local environment to Azure is successful.

In the [Azure Portal](#), select the web app. Select **Deployment > Deployment options**.



Run the app in Azure

Now that the web app is deployed to Azure, run the app.

This can be accomplished in two ways:

- In the Azure Portal, locate the web app blade for the web app. Select **Browse** to view the app in the default browser.
- Open a browser and enter the URL for the web app. Example: `http://SampleWebAppDemo.azurewebsites.net`

Update the web app and republish

After making changes to the local code, republish:

1. In **Solution Explorer** of Visual Studio, open the *Startup.cs* file.
2. In the `Configure` method, modify the `Response.WriteAsync` method so that it appears as follows:

```
await context.Response.WriteAsync("Hello World! Deploy to Azure.");
```

3. Save the changes to *Startup.cs*.
4. In **Solution Explorer**, right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** is displayed.
5. Enter a commit message, such as `Update #2`.
6. Press the **Commit** button to commit the project changes.
7. Select **Home > Sync > Actions > Push**.

NOTE

As an alternative, push the changes from the **Command Window** by opening the **Command Window**, changing to the project directory, and entering a git command. Example:

```
git push -u Azure-SampleApp master
```

View the updated web app in Azure

View the updated web app by selecting **Browse** from the web app blade in the Azure Portal or by opening a browser and entering the URL for the web app. Example: `http://SampleWebAppDemo.azurewebsites.net`

Additional resources

- [Use VSTS to Build and Publish to an Azure Web App with Continuous Deployment](#)
- [Project Kudu](#)

Troubleshoot ASP.NET Core on Azure App Service

1/10/2018 • 1 min to read • [Edit Online](#)

This topic is under development the week of January 8, 2018 and will appear soon.

Host ASP.NET Core on Windows with IIS

1/10/2018 • 12 min to read • [Edit Online](#)

By [Luke Latham](#) and [Rick Anderson](#)

Supported operating systems

The following operating systems are supported:

- Windows 7 and newer
- Windows Server 2008 R2 and newer†

†Conceptually, the IIS configuration described in this document also applies to hosting ASP.NET Core apps on Nano Server IIS, but refer to [ASP.NET Core with IIS on Nano Server](#) for specific instructions.

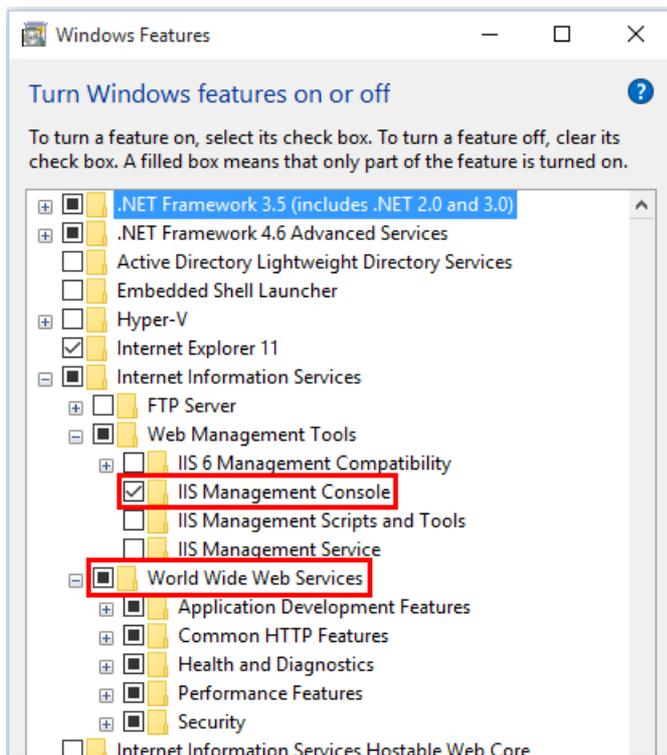
[HTTP.sys server](#) (formerly called [WebListener](#)) won't work in a reverse proxy configuration with IIS. Use the [Kestrel server](#).

IIS configuration

Enable the **Web Server (IIS)** role and establish role services.

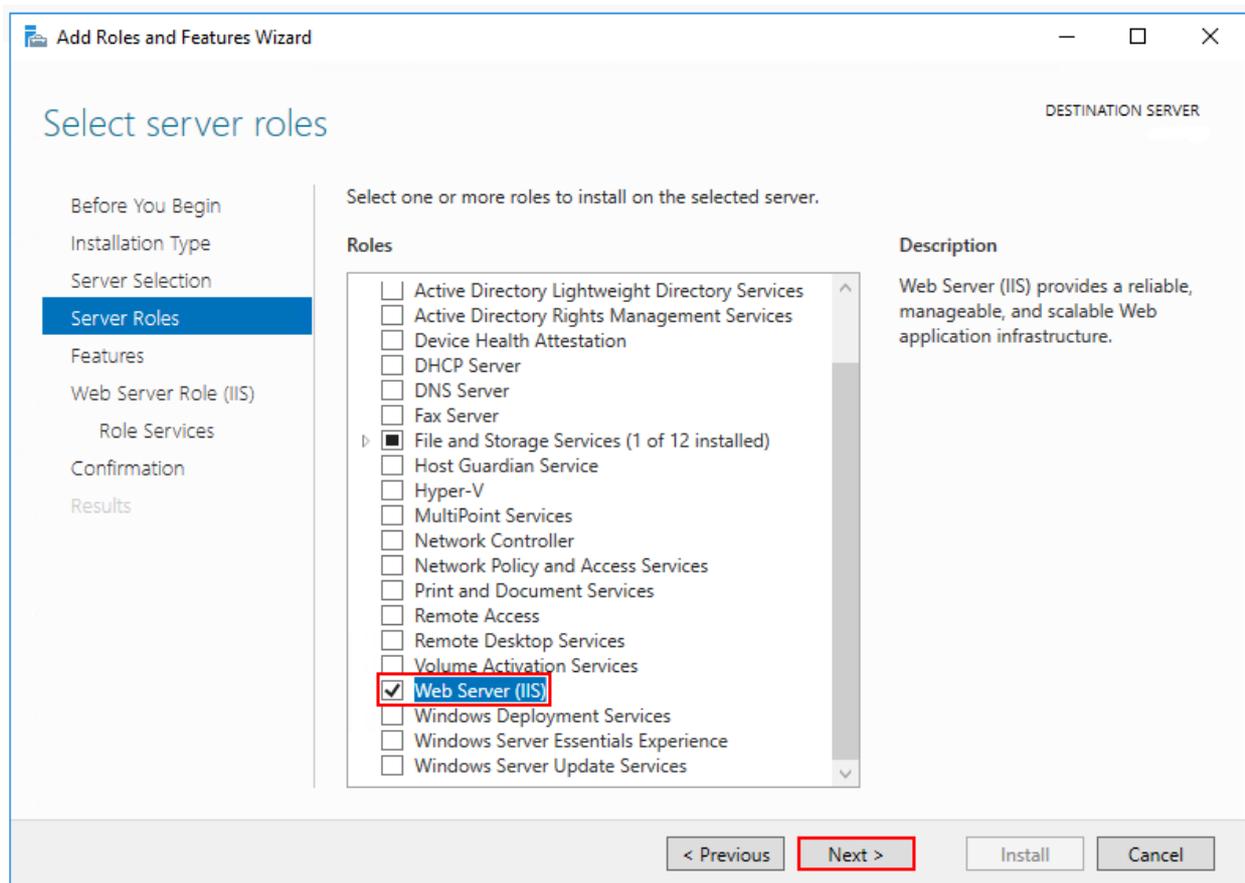
Windows desktop operating systems

Navigate to **Control Panel > Programs > Programs and Features > Turn Windows features on or off** (left side of the screen). Open the group for **Internet Information Services** and **Web Management Tools**. Check the box for **IIS Management Console**. Check the box for **World Wide Web Services**. Accept the default features for **World Wide Web Services** or customize the IIS features.

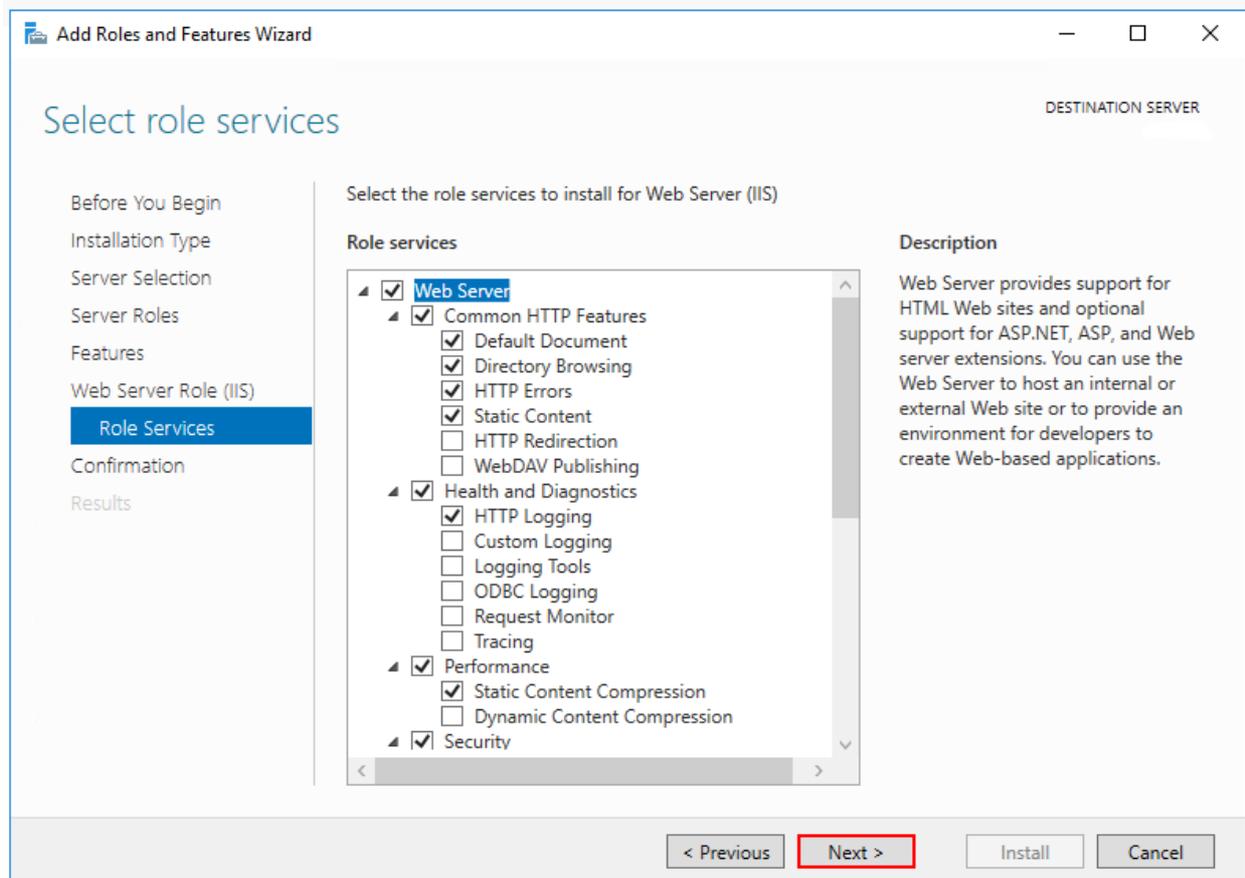


Windows Server operating systems

For server operating systems, use the **Add Roles and Features** wizard via the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



On the **Role services** step, select the IIS role services desired or accept the default role services provided.



Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart isn't required after installing the Web Server (IIS) role.

Install the .NET Core Windows Server Hosting bundle

1. Install the [.NET Core Windows Server Hosting bundle](#) on the hosting system. The bundle installs the .NET Core Runtime, .NET Core Library, and the [ASP.NET Core Module](#). The module creates the reverse proxy between IIS and the Kestrel server. If the system doesn't have an Internet connection, obtain and install the [Microsoft Visual C++ 2015 Redistributable](#) before installing the .NET Core Windows Server Hosting bundle.
2. Restart the system or execute **net stop was /y** followed by **net start w3svc** from a command prompt to pick up a change to the system PATH.

NOTE

For information on IIS Shared Configuration, see [ASP.NET Core Module with IIS Shared Configuration](#).

Install Web Deploy when publishing with Visual Studio

When deploying apps to servers with [Web Deploy](#), install the latest version of Web Deploy on the server. To install Web Deploy, use the [Web Platform Installer \(WebPI\)](#) or obtain an installer directly from the [Microsoft Download Center](#). The preferred method is to use WebPI. WebPI offers a standalone setup and a configuration for hosting providers.

Application configuration

Enabling the IISIntegration components

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

A typical *Program.cs* calls [CreateDefaultBuilder](#) to begin setting up a host. `CreateDefaultBuilder` configures [Kestrel](#) as the web server and enables IIS integration by configuring the base path and port for the [ASP.NET Core Module](#):

```
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        ...
```

For more information on hosting, see [Hosting in ASP.NET Core](#).

IIS options

To configure IIS options, include a service configuration for `IISOptions` in `ConfigureServices`:

```
services.Configure<IISOptions>(options =>
{
    ...
});
```

OPTION	DEFAULT	SETTING
--------	---------	---------

OPTION	DEFAULT	SETTING
<code>AutomaticAuthentication</code>	<code>true</code>	If <code>true</code> , the authentication middleware sets the <code>HttpContext.User</code> and responds to generic challenges. If <code>false</code> , the authentication middleware only provides an identity (<code>HttpContext.User</code>) and responds to challenges when explicitly requested by the <code>AuthenticationScheme</code> . Windows Authentication must be enabled in IIS for <code>AutomaticAuthentication</code> to function.
<code>AuthenticationDisplayName</code>	<code>null</code>	Sets the display name shown to users on login pages.
<code>ForwardClientCertificate</code>	<code>true</code>	If <code>true</code> and the <code>MS-ASPNETCORE-CLIENTCERT</code> request header is present, the <code>HttpContext.Connection.ClientCertificate</code> is populated.

web.config

The *web.config* file's primary purpose is to configure the [ASP.NET Core Module](#). It may optionally provide additional IIS configuration settings. Creating, transforming, and publishing *web.config* is handled by the .NET Core Web SDK (`Microsoft.NET.Sdk.Web`). The SDK is set at the top of the project file,

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

To prevent the SDK from transforming the *web.config* file, add the **<IsTransformWebConfigDisabled>** property to the project file with a setting of `true`:

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

If a *web.config* file is in the project, it's transformed with the correct *processPath* and *arguments* to configure the [ASP.NET Core Module](#) and moved to [published output](#). The transformation doesn't modify IIS configuration settings in the file.

web.config location

.NET Core apps are hosted via a reverse proxy between IIS and the Kestrel server. In order to create the reverse proxy, the *web.config* file must be present at the content root path (typically the app base path) of the deployed app, which is the website physical path provided to IIS. The *web.config* file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.

Sensitive files exist on the app's physical path, including subfolders, such as `<assembly_name>.runtimeconfig.json`, `<assembly_name>.xml` (XML Documentation comments), and `<assembly_name>.deps.json`. When the *web.config* file is present and configures the site, IIS prevents these sensitive files from being served. **Therefore, it's important that the *web.config* file isn't accidentally renamed or removed from the deployment.**

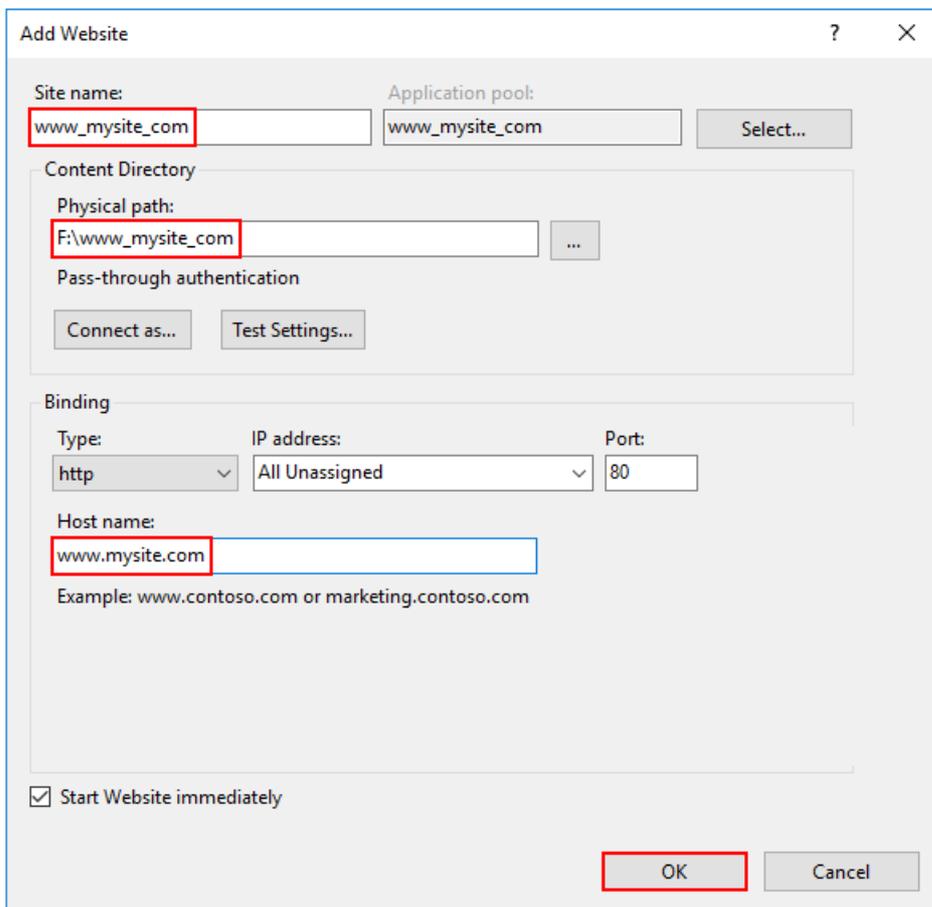
Create the IIS Website

1. On the target IIS system, create a folder to contain the app's published folders and files, which are described in [Directory Structure](#).

2. Within the folder, create a *logs* folder to hold stdout logs when stdout logging is enabled. If the app is deployed with a *logs* folder in the payload, skip this step. For instructions on how to make MSBuild create the *logs* folder, see the [Directory structure](#) topic.
3. In **IIS Manager**, create a new website. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website.
4. Set the application pool to **No Managed Code**. ASP.NET Core runs in a separate process and manages the runtime.
5. Open the **Add Website** window.



6. Configure the website.

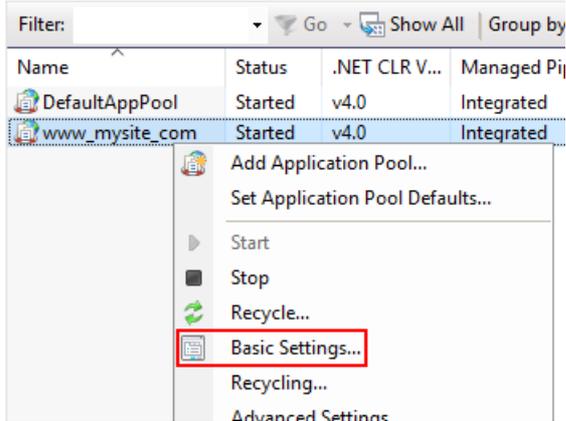


7. In the **Application Pools** panel, open the **Edit Application Pool** window by right-clicking on the website's app pool and selecting **Basic Settings...** from the popup menu.

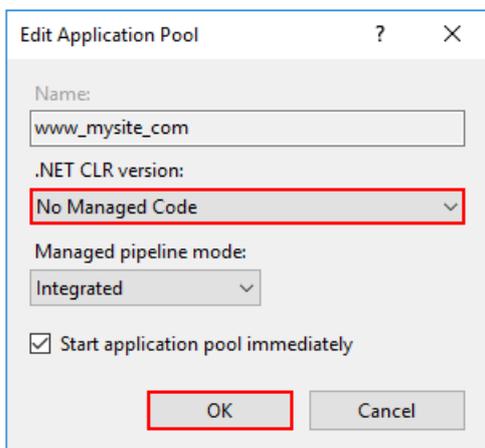


Application Pools

This page lets you view and manage the list of application pools c one or more applications, and provide isolation among different e



8. Set the **.NET CLR version** to **No Managed Code**.



Note: Setting the **.NET CLR version** to **No Managed Code** is optional. ASP.NET Core doesn't rely on loading the desktop CLR.

9. Confirm the process model identity has the proper permissions.

If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources.

Deploy the app

Deploy the app to the folder created on the target IIS system. [Web Deploy](#) is the recommended mechanism for deployment.

Confirm that the published app for deployment isn't running. Files in the *publish* folder are locked when the app is running. Locked files can't be overwritten. To release locked files in a deployment, stop the app pool:

- Manually in the IIS Manager on the server.
- Using Web Deploy and referencing `Microsoft.NET.Sdk.Web` in the project file. An *app_offline.htm* file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the *app_offline.htm* file during the deployment. For more information, see the [ASP.NET Core Module configuration reference](#).
- Use PowerShell to stop and restart the app pool (requires PowerShell 5 or later):

```

$webAppPoolName = 'APP_POOL_NAME'

# Stop the AppPool
if((Get-WebAppPoolState $webAppPoolName).Value -ne 'Stopped') {
    Stop-WebAppPool -Name $webAppPoolName
    while((Get-WebAppPoolState $webAppPoolName).Value -ne 'Stopped') {
        Start-Sleep -s 1
    }
    Write-Host `~AppPool Stopped
}

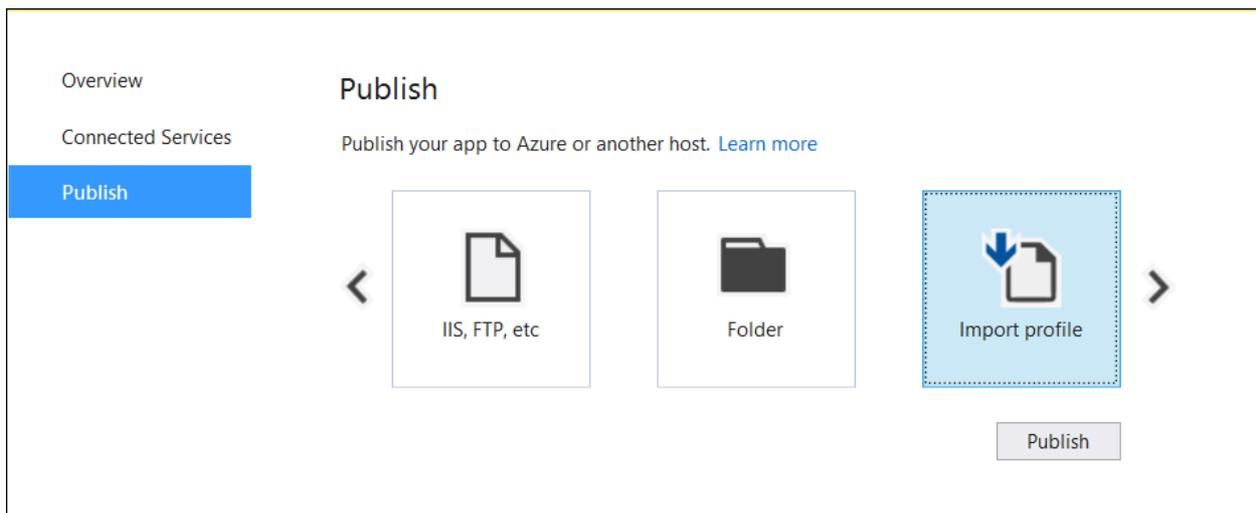
# Provide script commands here to deploy the app

# Restart the AppPool
if((Get-WebAppPoolState $webAppPoolName).Value -ne 'Started') {
    Start-WebAppPool -Name $webAppPoolName
    while((Get-WebAppPoolState $webAppPoolName).Value -ne 'Started') {
        Start-Sleep -s 1
    }
    Write-Host `~AppPool Started
}

```

Web Deploy with Visual Studio

See the [Visual Studio publish profiles for ASP.NET Core app deployment](#) topic to learn how to create a publish profile for use with Web Deploy. If the hosting provider supplies a Publish Profile or support for creating one, download their profile and import it using the Visual Studio **Publish** dialog.



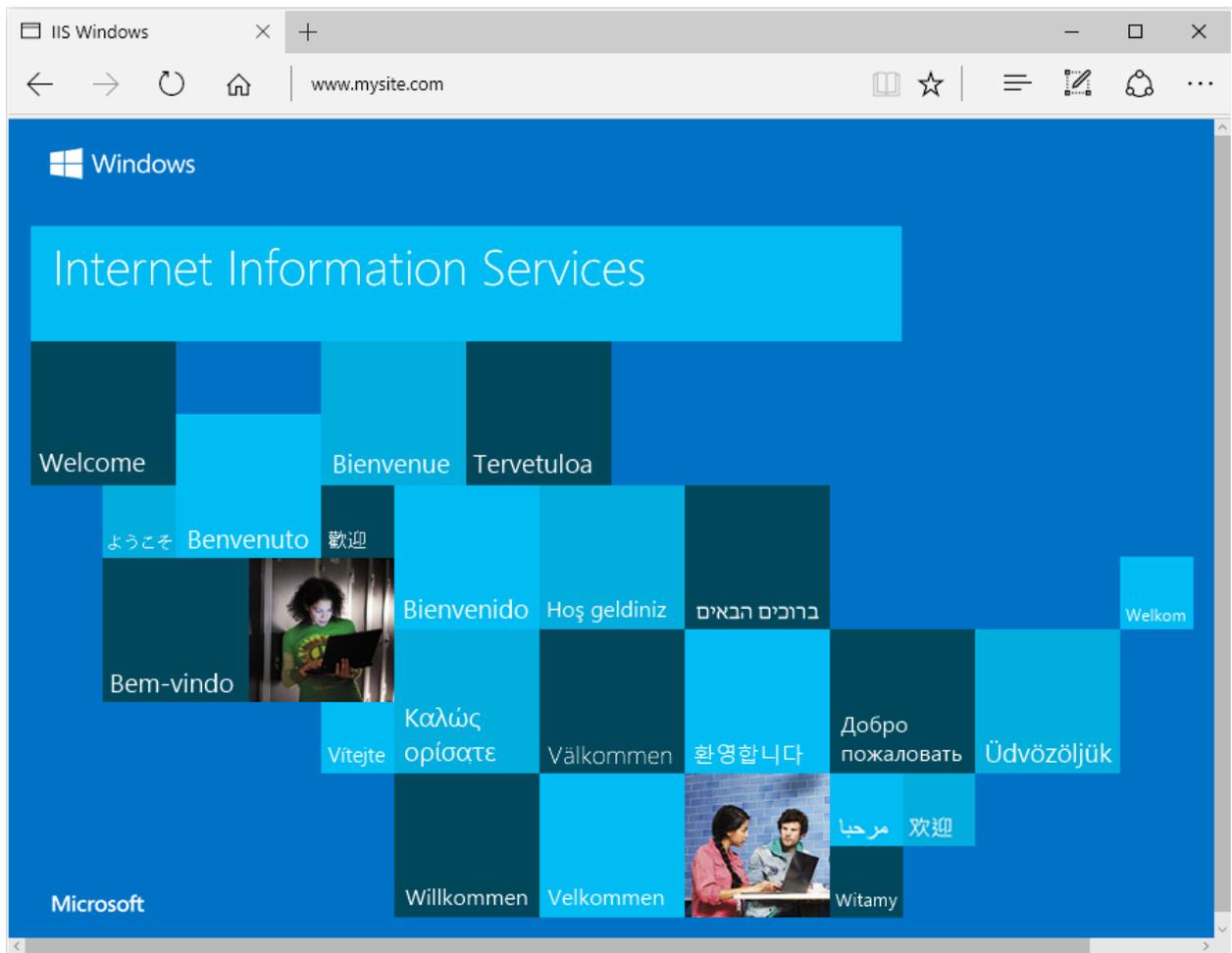
Web Deploy outside of Visual Studio

[Web Deploy](#) can also be used outside of Visual Studio from the command line. For more information, see [Web Deployment Tool](#).

Alternatives to Web Deploy

Use any of several methods to move the app to the hosting system, such as Xcopy, Robocopy, or PowerShell. Visual Studio users may use the [Publish Samples](#).

Browse the website



Data protection

Data Protection is used by several ASP.NET middlewares, including those used in authentication. Even if Data Protection APIs aren't called from user's code, Data Protection should be configured with a deployment script or in user code to create a persistent key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the keyring is stored in memory when the app restarts:

- All forms authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the keyring can no longer be decrypted.

To configure Data Protection under IIS, use **one** of the following approaches:

Create a Data Protection Registry Hive

Data Protection keys used by ASP.NET apps are stored in registry hives external to the apps. To persist the keys for a given app, create a registry hive for the app pool.

For standalone, non-webfarm IIS installations, the [Data Protection Provision-AutoGenKeys.ps1 PowerShell script](#) can be used for each app pool used with an ASP.NET Core app. This script creates a special registry key in the HKLM registry that is ACLed only to the worker process account. Keys are encrypted at rest using DPAPI with a machine-wide key.

In web farm scenarios, an app can be configured to use a UNC path to store its data protection keyring. By default, the data protection keys are not encrypted. Ensure that the file permissions for such a share are limited to the Windows account the app runs as. In addition, an X509 certificate can be used to protect keys at rest. Consider a mechanism to allow users to upload certificates: Place certificates into the user's trusted certificate store and ensure they're available on all machines where the user's app runs. See [Configuring Data Protection](#)

for details.

Configure the IIS Application Pool to load the user profile

This setting is in the **Process Model** section under the **Advanced Settings** for the app pool. Set Load User Profile to True. This stores keys under the user profile directory and protects them using DPAPI with a key specific to the user account used for the app pool.

Use the file system as a key ring store

Adjust the app code to [use the file system as a key ring store](#). Use an X509 certificate to protect the keyring and ensure the certificate is a trusted certificate. If it's a self signed certificate, place the certificate in the Trusted Root store.

When using IIS in a web farm:

- Use a file share that all machines can access.
- Deploy an X509 certificate to each machine. Configure [data protection in code](#).

Set a machine-wide policy for data protection

The data protection system has limited support for setting a default [machine-wide policy](#) for all apps that consume the Data Protection APIs. See the [data protection](#) documentation for more details.

Configuration of sub-applications

Sub-apps added under the root app shouldn't include the ASP.NET Core Module as a handler. If the module is added as a handler in a sub-app's *web.config* file, a 500.19 (Internal Server Error) referencing the faulty config file is received when attempting to browse the sub-app. The following example shows the contents of a published *web.config* file for an ASP.NET Core sub-app:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <aspNetCore processPath="dotnet"
      arguments=".\<<assembly_name>.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

When hosting a non-ASP.NET Core sub-app underneath an ASP.NET Core app, explicitly remove the inherited handler in the sub-app *web.config* file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore"/>
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\<<assembly_name>.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

For more information on configuring the ASP.NET Core Module, see the [Introduction to ASP.NET Core Module](#) topic and the [ASP.NET Core Module configuration reference](#).

Configuration of IIS with web.config

IIS configuration is influenced by the **<system.webServer>** section of *web.config* for those IIS features that apply to a reverse proxy configuration. If IIS is configured at the system level to use dynamic compression, that setting can be disabled for an app with the **<urlCompression>** element in the app's *web.config* file. For more information, see the [configuration reference for <system.webServer>](#), [ASP.NET Core Module Configuration Reference](#) and [Using IIS Modules with ASP.NET Core](#). If there's a need to set environment variables for individual apps running in isolated app pools (supported on IIS 10.0+), see the *AppCmd.exe command* section of the [Environment Variables <environmentVariables>](#) topic in the IIS reference documentation.

Configuration sections of web.config

Configuration sections of ASP.NET Framework apps in *web.config* aren't used by ASP.NET Core apps for configuration:

- **<system.web>**
- **<appSettings>**
- **<connectionStrings>**
- **<location>**

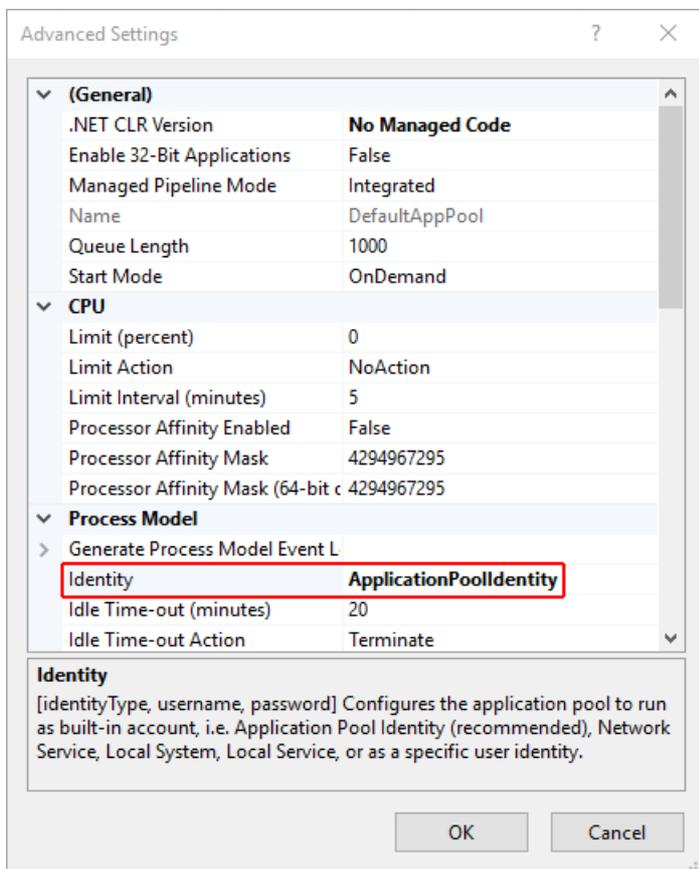
ASP.NET Core apps are configured using other configuration providers. For more information, see [Configuration](#).

Application Pools

When hosting multiple websites on a single system, isolate the apps from each other by running each app in its own app pool. The IIS **Add Website** dialog defaults to this configuration. When **Site name** is provided, the text is automatically transferred to the **Application pool** textbox. A new app pool is created using the site name when the website is added.

Application Pool Identity

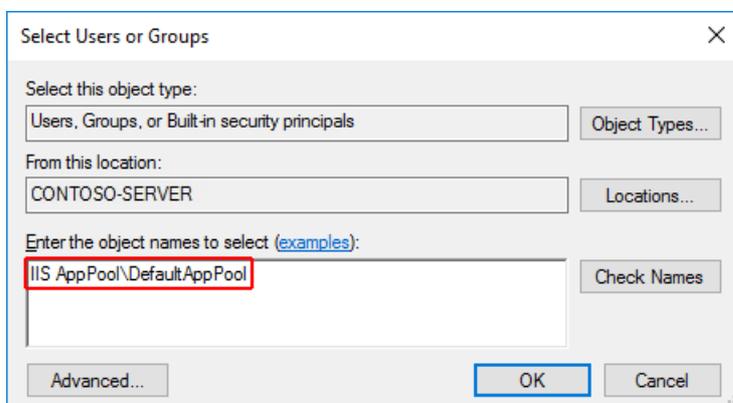
An app pool identity account allows an app to run under a unique account without having to create and manage domains or local accounts. On IIS 8.0+, the IIS Admin Worker Process (WAS) creates a virtual account with the name of the new app pool and runs the app pool's worker processes under this account by default. In the IIS Management Console under **Advanced Settings** for the app pool, ensure that the **Identity** is set to use **ApplicationPoolIdentity**:



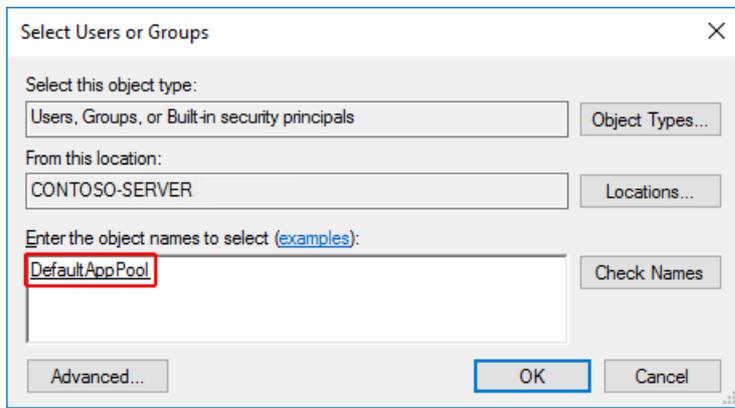
The IIS management process creates a secure identifier with the name of the app pool in the Windows Security System. Resources can be secured by using this identity; however, this identity isn't a real user account and won't show up in the Windows User Management Console.

If the IIS worker process requires elevated access to the app, modify the Access Control List (ACL) for the directory containing the app:

1. Open Windows Explorer and navigate to the directory.
2. Right-click on the directory and select **Properties**.
3. Under the **Security** tab, select the **Edit** button and then the **Add** button.
4. Select the **Locations** button and make sure the system is selected.
5. Enter **IIS AppPool\DefaultAppPool** in **Enter the object names to select** textbox.



6. Select the **Check Names** button. Select **OK**.



This can also be accomplished via a command prompt using the **ICACLS** tool:

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool":F
```

Additional resources

- [Troubleshoot ASP.NET Core on IIS](#)
- [Common errors reference for Azure App Service and IIS with ASP.NET Core](#)
- [Introduction to ASP.NET Core Module](#)
- [ASP.NET Core Module configuration reference](#)
- [Using IIS Modules with ASP.NET Core](#)
- [Introduction to ASP.NET Core](#)
- [The Official Microsoft IIS Site](#)
- [Microsoft TechNet Library: Windows Server](#)

Troubleshoot ASP.NET Core on IIS

1/10/2018 • 3 min to read • [Edit Online](#)

By [Luke Latham](#)

To diagnose issues with IIS deployments:

- Study browser output.
- Examine the system's **Application** log through **Event Viewer**.
- Enable `stdout` logging. The **ASP.NET Core Module** log is found on the path provided in the `stdoutLogFile` attribute of the `<aspNetCore>` element in `web.config`. Any folders on the path provided in the attribute value must exist in the deployment. Set `stdoutLogEnabled` to `true`. Apps that use the `Microsoft.NET.Sdk.Web` SDK to create the `web.config` file default the `stdoutLogEnabled` setting to `false`, so manually provide the `web.config` file or modify the file in order to enable `stdout` logging.

Use information from those three sources with the [common errors reference topic](#) to determine the problem. Follow the troubleshooting advice provided to resolve the issue.

Several of the common errors don't appear in the browser, Application Log, and ASP.NET Core Module Log until the module `startupTimeLimit` (default: 120 seconds) and `startupRetryCount` (default: 2) have passed. Therefore, wait a full six minutes before deducing that the module has failed to start a process for the app.

One quick way to determine if the app is working properly is to run the app directly on Kestrel. If the app was published as a [framework-dependent deployment](#), execute `dotnet <assembly_name>.dll` in the deployment folder, which is the IIS physical path to the app. If the app was published as a [self-contained deployment](#), run the app's executable directly from a command prompt, `<assembly_name>.exe`, in the deployment folder. If Kestrel is listening on default port 5000, the app should be available at `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the reverse proxy configuration and less likely within the app.

One way to determine if the reverse proxy is working properly is to perform a simple static file request for a stylesheet, script, or image from the app's static files in `wwwroot` using [Static File Middleware](#). If the app can serve static files but MVC Views and other endpoints are failing, the problem is less likely related to the reverse proxy configuration and more likely within the app (for example, MVC routing or 500 Internal Server Error).

When Kestrel starts normally behind IIS but the app won't run on the system after successfully running locally, an environment variable can be temporarily added to `web.config` to set the `ASPNETCORE_ENVIRONMENT` to `Development`. As long as the environment isn't overridden in app startup, setting the environment variable allows the [developer exception page](#) to appear when the app is run. Setting the environment variable for `ASPNETCORE_ENVIRONMENT` in this way is only recommended for staging/testing servers that aren't exposed to the Internet. Be sure to remove the environment variable from the `web.config` file when finished. For information on setting environment variables via `web.config`, see [environmentVariables child element of aspNetCore](#).

In most cases, enabling application logging assists in troubleshooting problems with the app or the reverse proxy. See [Logging](#) for more information.

The last troubleshooting tip pertains to apps that fail to run after upgrading either the .NET Core SDK on the development machine or package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by:

- Deleting the `bin` and `obj` folders in the project.
- Clearing package caches at `%UserProfile%\nuget\packages\` and `%LocalAppData%\Nuget\v3-cache`.

- Restoring and rebuilding the project.
- Confirming that the prior deployment on the server has been completely deleted prior to re-deploying the app.

TIP

A convenient way to clear package caches is to execute `dotnet nuget locals all --clear` from a command prompt.

Clearing package caches can also be accomplished by using the [nuget.exe](#) tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with Windows 10 and must be obtained separately from the NuGet website.

Additional resources

- [Common errors reference for Azure App Service and IIS with ASP.NET Core](#)
- [ASP.NET Core Module configuration reference](#)

ASP.NET Core Module configuration reference

1/10/2018 • 6 min to read • [Edit Online](#)

By [Luke Latham](#), [Rick Anderson](#), and [Sourabh Shirhatti](#)

This document provides details on how to configure the ASP.NET Core Module for hosting ASP.NET Core applications. For an introduction to the ASP.NET Core Module and installation instructions, see the [ASP.NET Core Module overview](#).

Configuration via web.config

The ASP.NET Core Module is configured via a site or application *web.config* file and has its own `aspNetCore` configuration section within `system.webServer`. Here's an example *web.config* file that the `Microsoft.NET.Sdk.Web` SDK will provide when the project is published for a [framework-dependent deployment](#) with placeholders for the `processPath` and `arguments`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%"
      arguments="%LAUNCHER_ARGS%"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

The *web.config* example below is for a [self-contained deployment](#) to the [Azure App Service](#). For more information, see [Host on Windows with IIS](#). See [Configuration of sub-applications](#) for an important note pertaining to the configuration of *web.config* files in sub-applications.

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath=".\MyApp.exe"
      stdoutLogEnabled="false"
      stdoutLogFile="\\?\%home%\LogFiles\stdout" />
  </system.webServer>
</configuration>
```

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION
-----------	-------------

ATTRIBUTE	DESCRIPTION
processPath	<p>Required string attribute.</p> <p>Path to the executable that will launch a process listening for HTTP requests. Relative paths are supported. If the path begins with '.', the path is considered to be relative to the site root.</p> <p>There is no default value.</p>
arguments	<p>Optional string attribute.</p> <p>Arguments to the executable specified in processPath.</p> <p>The default value is an empty string.</p>
startupTimeLimit	<p>Optional integer attribute.</p> <p>Duration in seconds that the module will wait for the executable to start a process listening on the port. If this time limit is exceeded, the module will kill the process. The module will attempt to launch the process again when it receives a new request and will continue to attempt to restart the process on subsequent incoming requests unless the application fails to start rapidFailsPerMinute number of times in the last rolling minute.</p> <p>The default value is 120.</p>
shutdownTimeLimit	<p>Optional integer attribute.</p> <p>Duration in seconds for which the module will wait for the executable to gracefully shutdown when the <i>app_offline.htm</i> file is detected.</p> <p>The default value is 10.</p>
rapidFailsPerMinute	<p>Optional integer attribute.</p> <p>Specifies the number of times the process specified in processPath is allowed to crash per minute. If this limit is exceeded, the module will stop launching the process for the remainder of the minute.</p> <p>The default value is 10.</p>
requestTimeout	<p>Optional timespan attribute.</p> <p>Specifies the duration for which the ASP.NET Core Module will wait for a response from the process listening on %ASPNETCORE_PORT%.</p> <p>The default value is "00:02:00".</p> <p>The <code>requestTimeout</code> must be specified in whole minutes only, otherwise it defaults to 2 minutes.</p>

ATTRIBUTE	DESCRIPTION
stdoutLogEnabled	<p>Optional Boolean attribute.</p> <p>If true, stdout and stderr for the process specified in processPath will be redirected to the file specified in stdoutLogFile.</p> <p>The default value is false.</p>
stdoutLogFile	<p>Optional string attribute.</p> <p>Specifies the relative or absolute file path for which stdout and stderr from the process specified in processPath will be logged. Relative paths are relative to the root of the site. Any path starting with '.' will be relative to the site root and all other paths will be treated as absolute paths. Any folders provided in the path must exist in order for the module to create the log file. The process ID, timestamp (<i>yyyyMMddhms</i>), and file extension (<i>.log</i>) with underscore delimiters are added to the last segment of the stdoutLogFile provided.</p> <p>The default value is <code>aspnetcore-stdout</code>.</p>
forwardWindowsAuthToken	<p>true or false.</p> <p>If true, the token will be forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It is the responsibility of that process to call CloseHandle on this token per request.</p> <p>The default value is true.</p>
disableStartupErrorPage	<p>true or false.</p> <p>If true, the 502.5 - Process Failure page will be suppressed, and the 502 status code page configured in your <i>web.config</i> will take precedence.</p> <p>The default value is false.</p>

Setting environment variables

The ASP.NET Core Module allows you specify environment variables for the process specified in the `processPath` attribute by specifying them in one or more `environmentVariable` child elements of an `environmentVariables` collection element under the `aspNetCore` element. Environment variables set in this section take precedence over system environment variables for the process.

The example below sets two environment variables. `ASPNETCORE_ENVIRONMENT` will configure the application's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the [developer exception page](#) to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that will read the value on startup to form a path in order to load the app's configuration file.

```
<aspNetCore processPath="dotnet"
  arguments=".\\MyApp.dll"
  stdoutLogEnabled="false"
  stdoutLogFile="\\?\%home%\LogFiles\stdout">
<environmentVariables>
  <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
</environmentVariables>
</aspNetCore>
```

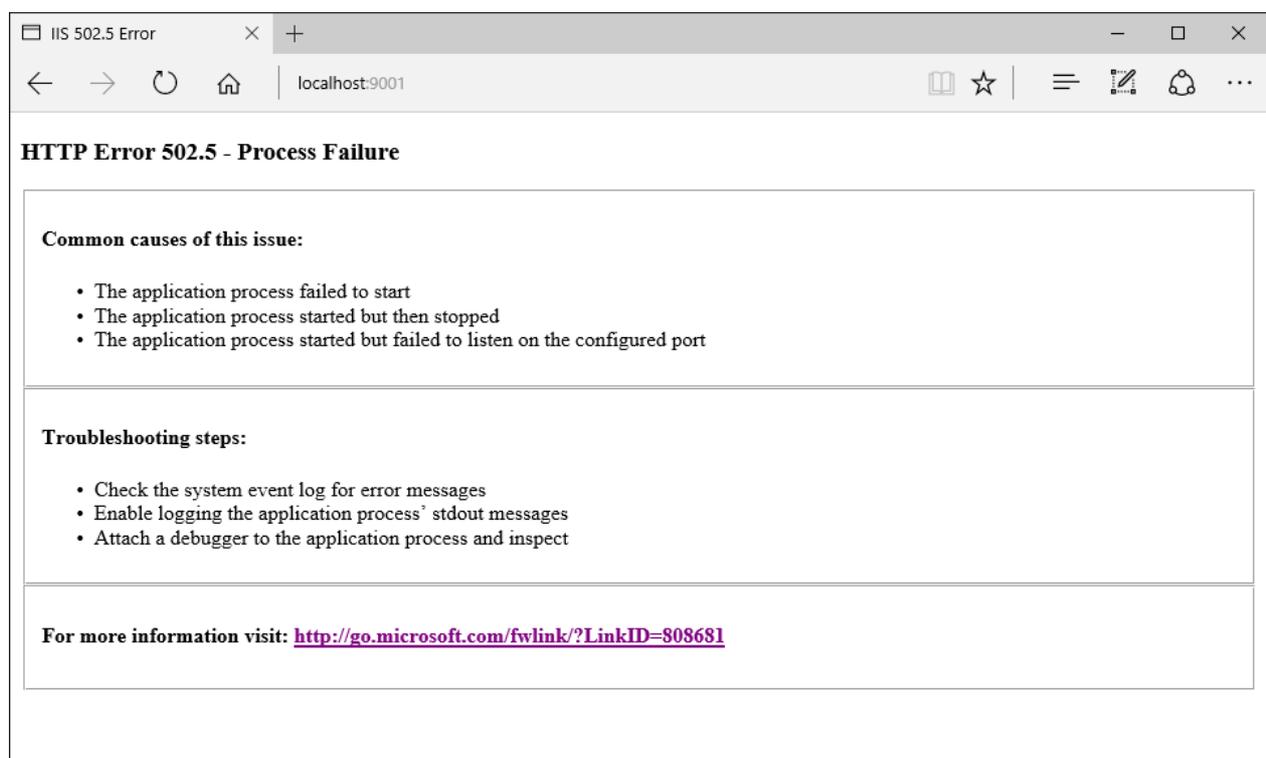
app_offline.htm

If you place a file with the name *app_offline.htm* at the root of a web application directory, the ASP.NET Core Module will attempt to gracefully shutdown the app and stop processing incoming requests. If the app is still running after `shutdownTimeLimit` number of seconds, the ASP.NET Core Module will kill the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module will respond to requests by sending back the contents of the *app_offline.htm* file. Once the *app_offline.htm* file is removed, the next request loads the application, which then responds to requests.

Start-up error page

If the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, you will see an HTTP 502.5 status code page. To suppress this page and revert to the default IIS 502 status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see [HTTP Errors](#) `<httpErrors>`.



The screenshot shows a browser window with the title "IIS 502.5 Error" and the address bar showing "localhost:9001". The main content of the page is titled "HTTP Error 502.5 - Process Failure". It contains three sections: "Common causes of this issue:" with a bulleted list of three items, "Troubleshooting steps:" with a bulleted list of three items, and "For more information visit:" with a link to "http://go.microsoft.com/fwlink/?LinkID=808681".

Log creation and redirection

The ASP.NET Core Module redirects `stdout` and `stderr` logs to disk if you set the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element. Any folders in the `stdoutLogFile` path must exist in order for the module to create the log file. A timestamp and file extension will be added automatically when the log file is created. Logs are not rotated, unless process recycling/restart occurs. It is the responsibility of the hoster to

limit the disk space the logs consume. Using the `stdout` log is only recommended for troubleshooting application startup issues and not for general application logging purposes.

The log file name is composed by appending the process ID (PID), timestamp (*yyyyMMddhms*), and file extension (*.log*) to the last segment of the `stdoutLogFile` path (typically *stdout*) delimited by underscores. For example if the `stdoutLogFile` path ends with *stdout*, a log for an app with a PID of 10652 created on 8/10/2017 at 12:05:02 has the file name *stdout_10652_20178101252.log*.

Here's a sample `aspNetCore` element that configures `stdout` logging. The `stdoutLogFile` path shown in the example is appropriate for the Azure App Service. A local path or network share path is acceptable for local logging. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
  arguments=". MyApp.dll"
  stdoutLogEnabled="true"
  stdoutLogFile="\\?\%home%\LogFiles\stdout">
</aspNetCore>
```

See [Configuration via web.config](#) for an example of the `aspNetCore` element in the *web.config* file.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **SYSTEM** account. Because the local system account does not have modify permission for the share path which is used by the IIS Shared Configuration, the installer will hit an access denied error when attempting to configure the module settings in *applicationHost.config* on the share.

The unsupported workaround is to disable the IIS Shared Configuration, run the installer, export the updated *applicationHost.config* file to the share, and re-enable the IIS Shared Configuration.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll

IIS Express (x86/amd64):

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll

Schema

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml

Configuration

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- `.vs\config\applicationHost.config`

You can search for `aspnetcore.dll` in the `applicationHost.config` file. For IIS Express, the `applicationHost.config` file won't exist by default. The file is created at `{application root}\.vs\config` when you start any web application project in the Visual Studio solution.

Development-time IIS support in Visual Studio for ASP.NET Core

1/10/2018 • 1 min to read • [Edit Online](#)

By: [Sourabh Shirhatti](#)

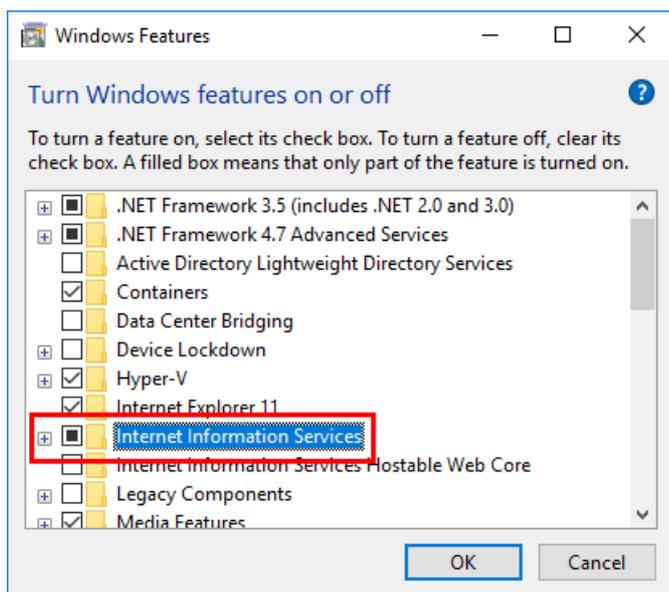
This article describes [Visual Studio](#) support for debugging ASP.NET Core applications running behind IIS on Windows Server. This topic walks through enabling this feature and setting up a project.

Prerequisites

- Visual Studio (2017/version 15.3 or later)
- ASP.NET and web development workload *OR* the .NET Core cross-platform development workload

Enable IIS

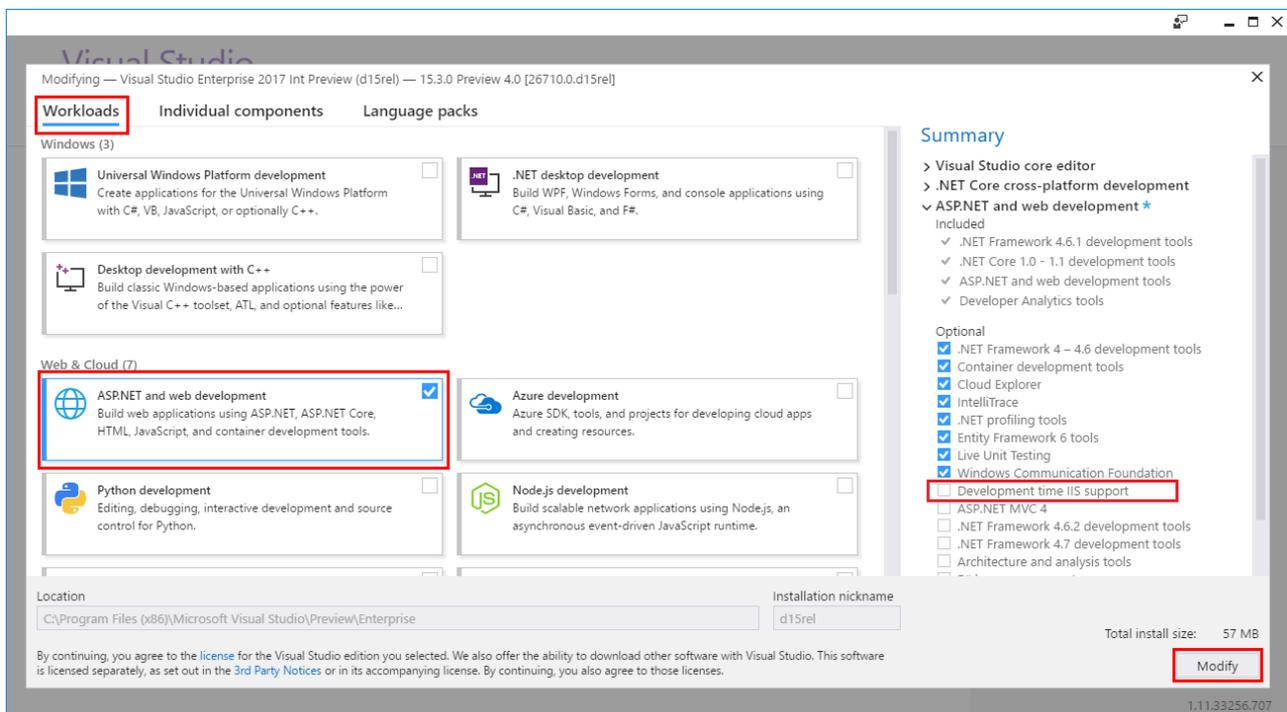
Enable IIS. Navigate to **Control Panel > Programs > Programs and Features > Turn Windows features on or off** (left side of the screen). Select the **Internet Information Services** checkbox.



If the IIS installation requires a reboot, reboot the system.

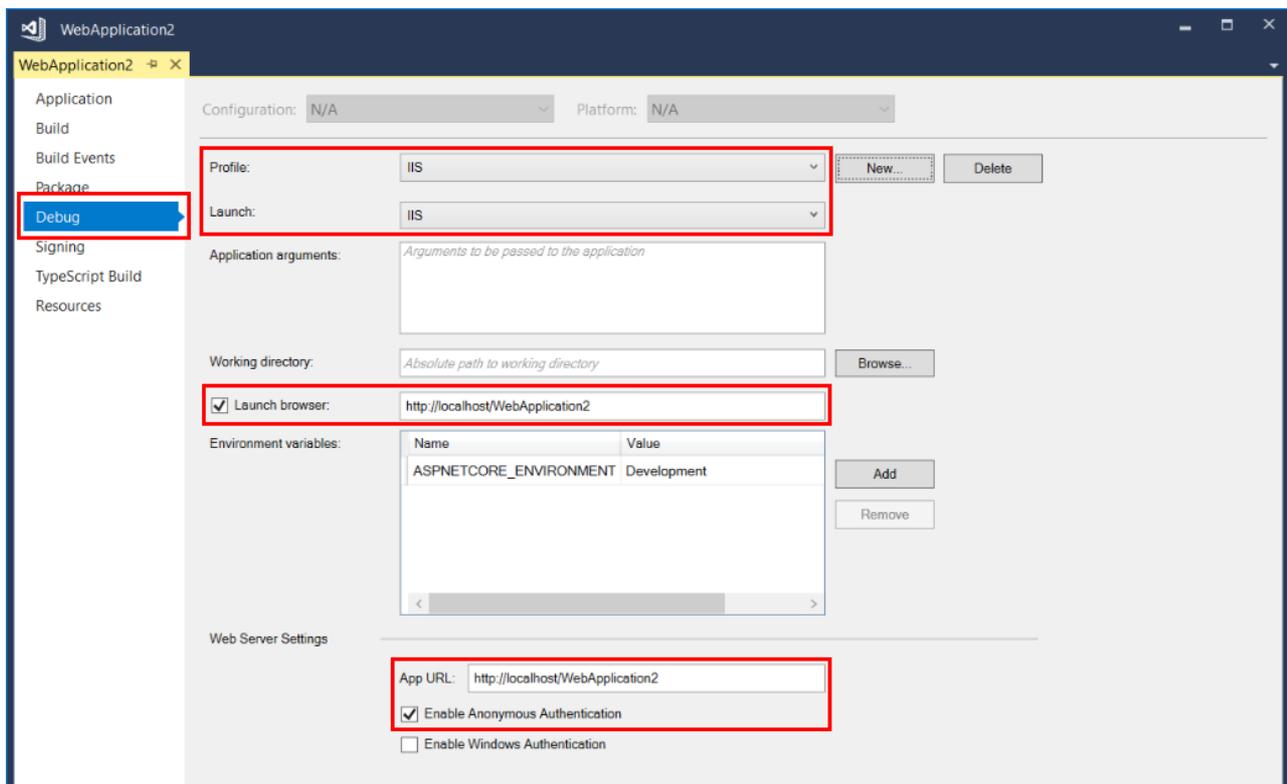
Enable development-time IIS support

Once IIS is installed, launch the Visual Studio installer to modify the existing Visual Studio installation. In the installer, select the **Development time IIS support** component. The component is listed as an optional component in the **Summary** panel for the **ASP.NET and web development** workload. This installs the [ASP.NET Core Module](#), which is a native IIS module required to run ASP.NET Core applications.



Configure the project

Create a new launch profile to add development-time IIS support. In Visual Studio's **Solution Explorer**, right-click the project and select **Properties**. Select the **Debug** tab. Select **IIS** from the **Launch** dropdown. Confirm that the **Launch browser** feature is enabled with the correct URL.



Alternatively, manually add a launch profile to the [launchSettings.json](#) file in the app:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iis": {
      "applicationUrl": "http://localhost/WebApplication2",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS": {
      "commandName": "IIS",
      "launchBrowser": "true",
      "launchUrl": "http://localhost/WebApplication2",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Visual Studio may prompt a restart if not running as an administrator. If prompted, restart Visual Studio.

Congratulations! At this point, the project is configured for development-time IIS support.

Additional resources

- [Host ASP.NET Core on Windows with IIS](#)
- [Introduction to ASP.NET Core Module](#)
- [ASP.NET Core Module configuration reference](#)

Using IIS Modules with ASP.NET Core

1/10/2018 • 4 min to read • [Edit Online](#)

By [Luke Latham](#)

ASP.NET Core applications are hosted by IIS in a reverse proxy configuration. Some of the native IIS modules and all of the IIS managed modules aren't available to process requests for ASP.NET Core apps. In many cases, ASP.NET Core offers an alternative to the features of IIS native and managed modules.

Native Modules

MODULE	.NET CORE ACTIVE	ASP.NET CORE OPTION
Anonymous Authentication <code>AnonymousAuthenticationModule</code>	Yes	
Basic Authentication <code>BasicAuthenticationModule</code>	Yes	
Client Certification Mapping Authentication <code>CertificateMappingAuthenticationModule</code>	Yes	
CGI <code>CgiModule</code>	No	
Configuration Validation <code>ConfigurationValidationModule</code>	Yes	
HTTP Errors <code>CustomErrorModule</code>	No	Status Code Pages Middleware
Custom Logging <code>CustomLoggingModule</code>	Yes	
Default Document <code>DefaultDocumentModule</code>	No	Default Files Middleware
Digest Authentication <code>DigestAuthenticationModule</code>	Yes	
Directory Browsing <code>DirectoryListingModule</code>	No	Directory Browsing Middleware
Dynamic Compression <code>DynamicCompressionModule</code>	Yes	Response Compression Middleware
Tracing <code>FailedRequestsTracingModule</code>	Yes	ASP.NET Core Logging

MODULE	.NET CORE ACTIVE	ASP.NET CORE OPTION
File Caching FileCacheModule	No	Response Caching Middleware
HTTP Caching HttpCacheModule	No	Response Caching Middleware
HTTP Logging HttpLoggingModule	Yes	ASP.NET Core Logging Implementations: elmah.io , Loggr , NLog , Serilog
HTTP Redirection HttpRedirectionModule	Yes	URL Rewriting Middleware
IIS Client Certificate Mapping Authentication IISCertificateMappingAuthenticationModule	Yes	
IP and Domain Restrictions IpRestrictionModule	Yes	
ISAPI Filters IsapiFilterModule	Yes	Middleware
ISAPI IsapiModule	Yes	Middleware
Protocol Support ProtocolSupportModule	Yes	
Request Filtering RequestFilteringModule	Yes	URL Rewriting Middleware IRule
Request Monitor RequestMonitorModule	Yes	
URL Rewriting RewriteModule	Yes†	URL Rewriting Middleware
Server Side Includes ServerSideIncludeModule	No	
Static Compression StaticCompressionModule	No	Response Compression Middleware
Static Content StaticFileModule	No	Static File Middleware
Token Caching TokenCacheModule	Yes	
URI Caching UriCacheModule	Yes	

MODULE	.NET CORE ACTIVE	ASP.NET CORE OPTION
URL Authorization <code>UrlAuthorizationModule</code>	Yes	ASP.NET Core Identity
Windows Authentication <code>WindowsAuthenticationModule</code>	Yes	

†The URL Rewrite Module's `isFile` and `isDirectory` don't work with ASP.NET Core apps due to the changes in [directory structure](#).

Managed Modules

MODULE	.NET CORE ACTIVE	ASP.NET CORE OPTION
AnonmousIdentification	No	
DefaultAuthentication	No	
FileAuthorization	No	
FormsAuthentication	No	Cookie Authentication Middleware
OutputCache	No	Response Caching Middleware
Profile	No	
RoleManager	No	
ScriptModule-4.0	No	
Session	No	Session Middleware
UrlAuthorization	No	
UrlMappingsModule	No	URL Rewriting Middleware
UrlRoutingModule-4.0	No	ASP.NET Core Identity
WindowsAuthentication	No	

IIS Manager application changes

Using IIS Manager to configure settings, the `web.config` file of the app is changed. If deploying an app and including `web.config`, any changes made with IIS Manger are overwritten by the deployed `web.config` file. If changes are made to the server's `web.config` file, copy the updated `web.config` file to the local project immediately.

Disabling IIS modules

If an IIS module is configured at the server level that must be disabled for an app, an addition to the app's `web.config` file can disable the module. Either leave the module in place and deactivate it using a configuration setting (if available) or remove the module from the app.

Module deactivation

Many modules offer a configuration setting that allows them to be disabled without removing them from the app. This is the simplest and quickest way to deactivate a module. For example if wishing to disable the IIS URL Rewrite Module, use the `<httpRedirect>` element as shown below. For more information on disabling modules with configuration settings, follow the links in the *Child Elements* section of IIS `<system.webServer>`.

```
<configuration>
  <system.webServer>
    <httpRedirect enabled="false" />
  </system.webServer>
</configuration>
```

Module removal

If opting to remove a module with a setting in *web.config*, unlock the module and unlock the `<modules>` section of *web.config* first. The steps are outlined below:

1. Unlock the module at the server level. Click on the IIS server in the IIS Manager **Connections** sidebar. Open the **Modules** in the **IIS** area. Click on the module in the list. In the **Actions** sidebar on the right, click **Unlock**. Unlock as many modules as are planned to remove from *web.config* later.
2. Deploy the app without a `<modules>` section in *web.config*. If an app is deployed with a *web.config* containing the `<modules>` section without having unlocked the section first in the IIS Manager, the Configuration Manager throws an exception when attempting to unlock the section. Therefore, deploy the app without a `<modules>` section.
3. Unlock the `<modules>` section of *web.config*. In the **Connections** sidebar, click the website in **Sites**. In the **Management** area, open the **Configuration Editor**. Use the navigation controls to select the `system.webServer/modules` section. In the **Actions** sidebar on the right, click to **Unlock** the section.
4. At this point, a `<modules>` section can be added to the *web.config* file with a `<remove>` element to remove the module from the app. Multiple `<remove>` elements can be added to remove multiple modules. Don't forget that if *web.config* changes are made on the server to make them immediately in the project locally. Removing a module this way won't affect the use of the module with other apps on the server.

```
<configuration>
  <system.webServer>
    <modules>
      <remove name="MODULE_NAME" />
    </modules>
  </system.webServer>
</configuration>
```

For an IIS installation with the default modules installed, use the following `<module>` section to remove the default modules.

```
<modules>
  <remove name="CustomErrorModule" />
  <remove name="DefaultDocumentModule" />
  <remove name="DirectoryListingModule" />
  <remove name="HttpCacheModule" />
  <remove name="HttpLoggingModule" />
  <remove name="ProtocolSupportModule" />
  <remove name="RequestFilteringModule" />
  <remove name="StaticCompressionModule" />
  <remove name="StaticFileModule" />
</modules>
```

An IIS module can also be removed with *Appcmd.exe*. Provide the `MODULE_NAME` and `APPLICATION_NAME` in the command shown below:

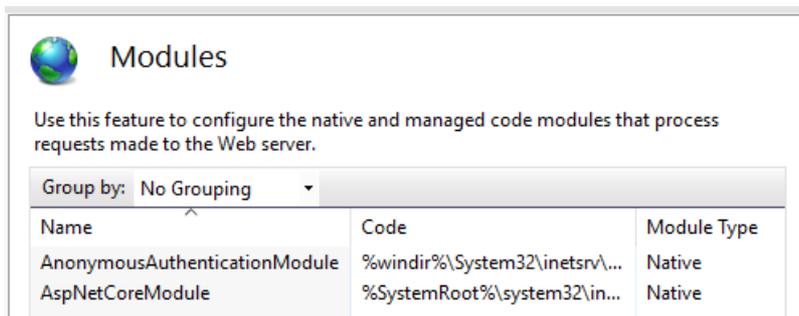
```
Appcmd.exe delete module MODULE_NAME /app.name:APPLICATION_NAME
```

Here's how to remove the `DynamicCompressionModule` from the Default Web Site:

```
%windir%\system32\inetsrv\appcmd.exe delete module DynamicCompressionModule /app.name:"Default Web Site"
```

Minimum module configuration

The only modules required to run an ASP.NET Core app are the Anonymous Authentication Module and the ASP.NET Core Module.



Use this feature to configure the native and managed code modules that process requests made to the Web server.

Group by: No Grouping

Name	Code	Module Type
AnonymousAuthenticationModule	%windir%\System32\inetsrv\...	Native
AspNetCoreModule	%SystemRoot%\system32\in...	Native

Additional resources

- [Host on Windows with IIS](#)
- [IIS Modules Overview](#)
- [Customizing IIS 7.0 Roles and Modules](#)
- [IIS <system.webServer>](#)

Host an ASP.NET Core app in a Windows Service

1/10/2018 • 4 min to read • [Edit Online](#)

By [Tom Dykstra](#)

The recommended way to host an ASP.NET Core app on Windows without using IIS is to run it in a [Windows Service](#). That way it can automatically start after reboots and crashes, without waiting for someone to log in.

[View or download sample code \(how to download\)](#). See the [Next Steps](#) section for instructions on how to run it.

Prerequisites

- The app must run on the .NET Framework runtime. In the `.csproj` file, specify appropriate values for [TargetFramework](#) and [RuntimeIdentifier](#). Here's an example:

```
<PropertyGroup>
  <TargetFramework>net452</TargetFramework>
  <RuntimeIdentifier>win7-x86</RuntimeIdentifier>
</PropertyGroup>
```

When creating a project in Visual Studio, use the **ASP.NET Core Application (.NET Framework)** template.

- If the app receives requests from the Internet (not just from an internal network), it must use the [WebListener](#) web server rather than [Kestrel](#). Kestrel must be used with IIS for edge deployments. For more information, see [When to use Kestrel with a reverse proxy](#).

Getting started

This section explains the minimum changes required to set up an existing ASP.NET Core project to run in a service.

- Install the NuGet package [Microsoft.AspNetCore.Hosting.WindowsServices](#).
- Make the following changes in `Program.Main`:
 - Call `host.RunAsService` instead of `host.Run`.
 - If the code calls `UseContentRoot`, use a path to the publish location instead of `Directory.GetCurrentDirectory()`

```
public static void Main(string[] args)
{
    var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
    var pathToContentRoot = Path.GetDirectoryName(pathToExe);

    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(pathToContentRoot)
        .UseIISIntegration()
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    host.RunAsService();
}
```

- Publish the application to a folder.

Use [dotnet publish](#) or a [Visual Studio publish profile](#) that publishes to a folder.

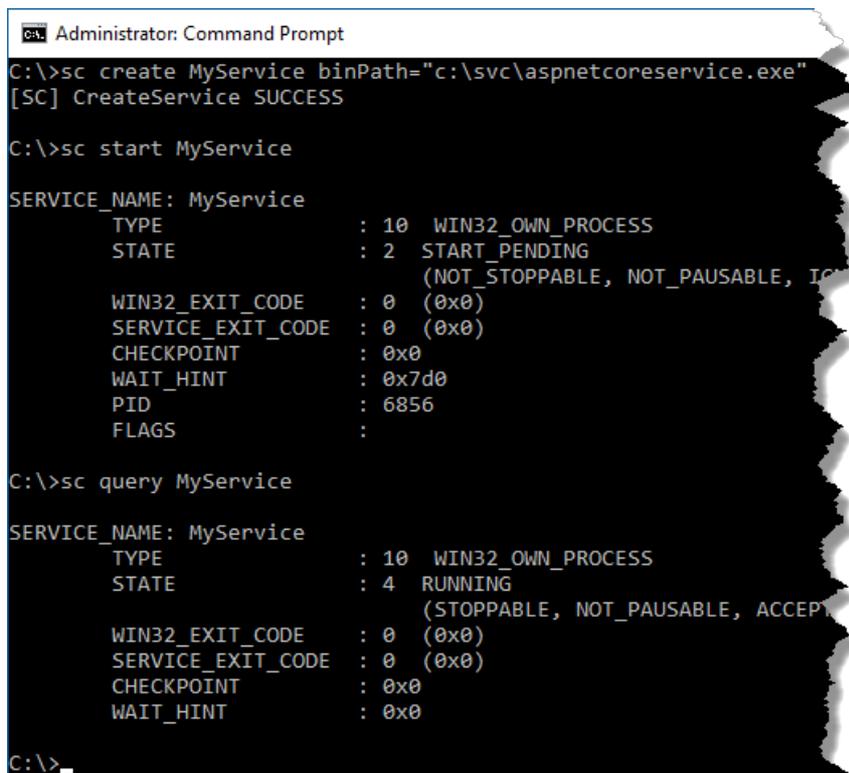
- Test by creating and starting the service.

Open an administrator command prompt window to use the [sc.exe](#) command-line tool to create and start a service.

If the service is named MyService, publish the app to `c:\svc`, and the app itself is named `AspNetCoreService`, the commands would look like this:

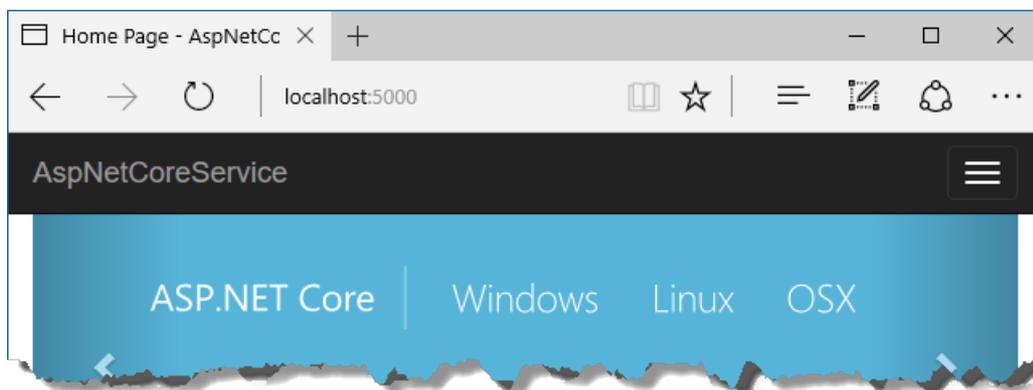
```
sc create MyService binPath="C:\Svc\AspNetCoreService.exe"  
sc start MyService
```

The `binPath` value is the path to the app's executable, including the executable filename itself.



```
Administrator: Command Prompt  
C:\>sc create MyService binPath="c:\svc\aspnetcoreservice.exe"  
[SC] CreateService SUCCESS  
  
C:\>sc start MyService  
  
SERVICE_NAME: MyService  
        TYPE               : 10  WIN32_OWN_PROCESS  
        STATE                : 2   START_PENDING  
                (NOT_STOPPABLE, NOT_PAUSABLE, IGNORE_SILENCE)  
        WIN32_EXIT_CODE       : 0    (0x0)  
        SERVICE_EXIT_CODE    : 0    (0x0)  
        CHECKPOINT           : 0x0  
        WAIT_HINT            : 0x7d0  
        PID                 : 6856  
        FLAGS                 :  
  
C:\>sc query MyService  
  
SERVICE_NAME: MyService  
        TYPE               : 10  WIN32_OWN_PROCESS  
        STATE                : 4   RUNNING  
                (STOPPABLE, NOT_PAUSABLE, ACCEPTS_SHUTDOWN)  
        WIN32_EXIT_CODE       : 0    (0x0)  
        SERVICE_EXIT_CODE    : 0    (0x0)  
        CHECKPOINT           : 0x0  
        WAIT_HINT            : 0x0  
  
C:\>
```

When these commands finish, browse to the same path as when running as a console app (by default, `http://localhost:5000`)



Provide a way to run outside of a service

It's easier to test and debug when running outside of a service, so it's customary to add code that calls

`host.RunAsService` only under certain conditions. For example, the app can run as a console app with a `--console` command-line argument or if the debugger is attached.

```
public static void Main(string[] args)
{
    bool isService = true;
    if (Debugger.IsAttached || args.Contains("--console"))
    {
        isService = false;
    }

    var pathToContentRoot = Directory.GetCurrentDirectory();
    if (isService)
    {
        var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
        pathToContentRoot = Path.GetDirectoryName(pathToExe);
    }

    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(pathToContentRoot)
        .UseIISIntegration()
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    if (isService)
    {
        host.RunAsService();
    }
    else
    {
        host.Run();
    }
}
```

Handle stopping and starting events

To handle `OnStarting`, `OnStarted`, and `OnStopping` events, make the following additional changes:

- Create a class that derives from `WebHostService`.

```
internal class CustomWebHostService : WebHostService
{
    public CustomWebHostService(IWebHost host) : base(host)
    {
    }

    protected override void OnStarting(string[] args)
    {
        base.OnStarting(args);
    }

    protected override void OnStarted()
    {
        base.OnStarted();
    }

    protected override void OnStopping()
    {
        base.OnStopping();
    }
}
```

- Create an extension method for `IWebHost` that passes the custom `WebHostService` to `ServiceBase.Run`.

```
public static class WebHostServiceExtensions
{
    public static void RunAsCustomService(this IWebHost host)
    {
        var webHostService = new CustomWebHostService(host);
        ServiceBase.Run(webHostService);
    }
}
```

- In `Program.Main` change call the new extension method instead of `host.RunAsService`.

```
public static void Main(string[] args)
{
    bool isService = true;
    if (Debugger.IsAttached || args.Contains("--console"))
    {
        isService = false;
    }

    var pathToContentRoot = Directory.GetCurrentDirectory();
    if (isService)
    {
        var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
        pathToContentRoot = Path.GetDirectoryName(pathToExe);
    }

    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(pathToContentRoot)
        .UseIISIntegration()
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    if (isService)
    {
        host.RunAsCustomService();
    }
    else
    {
        host.Run();
    }
}
```

If the custom `WebHostService` code needs to get a service from dependency injection (such as a logger), get it from the `Services` property of `IWebHost`.

```
internal class CustomWebHostService : WebHostService
{
    private ILogger _logger;

    public CustomWebHostService(IWebHost host) : base(host)
    {
        _logger = host.Services.GetRequiredService<ILogger<CustomWebHostService>>();
    }

    protected override void OnStarting(string[] args)
    {
        _logger.LogDebug("OnStarting method called.");
        base.OnStarting(args);
    }

    protected override void OnStarted()
    {
        _logger.LogDebug("OnStarted method called.");
        base.OnStarted();
    }

    protected override void OnStopping()
    {
        _logger.LogDebug("OnStopping method called.");
        base.OnStopping();
    }
}
```

Next steps

The [sample application](#) that accompanies this article is a simple MVC web app that has been modified as shown in preceding code examples. To run it in a service, do the following steps:

- Publish to c:\svc.
- Open an administrator window.
- Enter the following commands:

```
sc create MyService binPath="c:\svc\aspnetcoreservice.exe"
sc start MyService
```

- In a browser, go to <http://localhost:5000> to verify that it's running.

If the app doesn't start up as expected when running in a service, a quick way to make error messages accessible is to add a logging provider such as the [Windows EventLog provider](#).

Acknowledgments

This article was written with the help of sources that were already published. The earliest and most useful of them were these:

- [Hosting ASP.NET Core as Windows service](#)
- [How to host your ASP.NET Core in a Windows Service](#)

Host ASP.NET Core on Linux with nginx

1/10/2018 • 8 min to read • [Edit Online](#)

By [Sourabh Shirhatti](#)

This guide explains setting up a production-ready ASP.NET Core environment on an Ubuntu 16.04 Server.

Note: For Ubuntu 14.04, *supervisord* is recommended as a solution for monitoring the Kestrel process. *systemd* isn't available on Ubuntu 14.04. [See previous version of this document](#)

This guide:

- Places an existing ASP.NET Core app behind a reverse proxy server.
- Sets up the reverse proxy server to forward requests to the Kestrel web server.
- Ensures the web app runs on startup as a daemon.
- Configures a process management tool to help restart the web app.

Prerequisites

1. Access to an Ubuntu 16.04 Server with a standard user account with sudo privilege
2. An existing ASP.NET Core app

Copy over the app

Run `dotnet publish` from the dev environment to package an app into a self-contained directory that can run on the server.

Copy the ASP.NET Core app to the server using whatever tool integrates into the organization's workflow (for example, SCP, FTP). Test the app, for example:

- From the command line, run `dotnet <app_assembly>.dll`.
- In a browser, navigate to `http://<serveraddress>:<port>` to verify the app works on Linux.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web apps. A reverse proxy terminates the HTTP request and forwards it to the ASP.NET Core app.

Why use a reverse proxy server?

Kestrel is great for serving dynamic content from ASP.NET Core; however, the web serving parts aren't as feature rich as servers like IIS, Apache, or nginx. A reverse proxy server can offload work like serving static content, caching requests, compressing requests, and SSL termination from the HTTP server. A reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, a single instance of nginx is used. It runs on the same server, alongside the HTTP server. Based on requirements, a different setup may be chosen.

Because requests are forwarded by reverse proxy, use the `ForwardedHeaders` middleware from the `Microsoft.AspNetCore.HttpOverrides` package. This middleware updates `Request.Scheme`, using the `X-Forwarded-Proto` header, so that redirect URIs and other security policies work correctly.

When setting up a reverse proxy server, the authentication middleware needs `UseForwardedHeaders` to run first.

This ordering ensures that the authentication middleware can consume the affected values and generate correct redirect URIs.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Invoke the `UseForwardedHeaders` method (in the `Configure` method of `Startup.cs`) before calling `UseAuthentication` or similar authentication scheme middleware:

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

Install nginx

```
sudo apt-get install nginx
```

NOTE

If optional nginx modules will be installed, building nginx from source might be required.

Use `apt-get` to install nginx. The installer creates a System V init script that runs nginx as daemon on system startup. Since nginx was installed for the first time, explicitly start it by running:

```
sudo service nginx start
```

Verify a browser displays the default landing page for nginx.

Configure nginx

To configure nginx as a reverse proxy to forward requests to our ASP.NET Core app, modify `/etc/nginx/sites-available/default`. Open it in a text editor, and replace the contents with the following:

```
server {
    listen 80;
    location / {
        proxy_pass http://localhost:5000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection keep-alive;
        proxy_set_header Host $http_host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

This nginx configuration file forwards incoming public traffic from port `80` to port `5000`.

Once the nginx configuration is established, run `sudo nginx -t` to verify the syntax of the configuration files. If the configuration file test is successful, force nginx to pick up the changes by running `sudo nginx -s reload`.

Monitoring the app

The server is setup to forward requests made to `http://<serveraddress>:80` on to the ASP.NET Core app running

on Kestrel at `http://127.0.0.1:5000`. However, nginx is not set up to manage the Kestrel process. `systemd` can be used to create a service file to start and monitor the underlying web app. `systemd` is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

```
sudo nano /etc/systemd/system/kestrel-hellomvc.service
```

The following is an example service file for the app:

```
[Unit]
Description=Example .NET Web API App running on Ubuntu

[Service]
WorkingDirectory=/var/aspnetcore/hellomvc
ExecStart=/usr/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
Restart=always
RestartSec=10 # Restart service after 10 seconds if dotnet service crashes
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_PRINT_TELEMETRY_MESSAGE=false

[Install]
WantedBy=multi-user.target
```

Note: If the user `www-data` is not used by the configuration, the user defined here must be created first and given proper ownership for files.

Save the file and enable the service.

```
systemctl enable kestrel-hellomvc.service
```

Start the service and verify that it is running.

```
systemctl start kestrel-hellomvc.service
systemctl status kestrel-hellomvc.service
```

- kestrel-hellomvc.service - Example .NET Web API App running on Ubuntu
 - Loaded: loaded (/etc/systemd/system/kestrel-hellomvc.service; enabled)
 - Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
 - Main PID: 9021 (dotnet)
 - CGroup: /system.slice/kestrel-hellomvc.service
 - └─9021 /usr/local/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll

With the reverse proxy configured and Kestrel managed through `systemd`, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. It is also accessible from a remote machine, barring any firewall that might be blocking. Inspecting the response headers, the `Server` header shows the ASP.NET Core app being served by Kestrel.

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

Viewing logs

Since the web app using Kestrel is managed using `systemd`, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by `systemd`. To view the `kestrel-helloMvc.service`-specific items, use the following command:

```
sudo journalctl -fu kestrel-helloMvc.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago` or a combination of these can reduce the amount of entries returned.

```
sudo journalctl -fu kestrel-helloMvc.service --since "2016-10-18" --until "2016-10-18 04:00"
```

Securing the app

Enable AppArmor

Linux Security Modules (LSM) is a framework that is part of the Linux kernel since Linux 2.6. LSM supports different implementations of security modules. [AppArmor](#) is a LSM that implements a Mandatory Access Control system which allows confining the program to a limited set of resources. Ensure AppArmor is enabled and properly configured.

Configuring the firewall

Close off all external ports that are not in use. Uncomplicated firewall (ufw) provides a front end for `iptables` by providing a command line interface for configuring the firewall. Verify that `ufw` is configured to allow traffic on any ports needed.

```
sudo apt-get install ufw
sudo ufw enable

sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
```

Securing nginx

The default distribution of nginx doesn't enable SSL. To enable additional security features, build from source.

Download the source and install the build dependencies

```
# Install the build dependencies
sudo apt-get update
sudo apt-get install build-essential zlib1g-dev libpcre3-dev libssl-dev libxslt1-dev libxml2-dev libgd2-xpm-dev libgeoip-dev libgoogle-perftools-dev libperl-dev

# Download nginx 1.10.0 or latest
wget http://www.nginx.org/download/nginx-1.10.0.tar.gz
tar zxf nginx-1.10.0.tar.gz
```

Change the nginx response name

Edit `src/http/nginx_http_header_filter_module.c`:

```
static char ngx_http_server_string[] = "Server: Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Web Server" CRLF;
```

Configure the options and build

The PCRE library is required for regular expressions. Regular expressions are used in the location directive for the `ngx_http_rewrite_module`. The `http_ssl_module` adds HTTPS protocol support.

Consider using a web app firewall like *ModSecurity* to harden the app.

```
./configure
--with-pcre=../pcre-8.38
--with-zlib=../zlib-1.2.8
--with-http_ssl_module
--with-stream
--with-mail=dynamic
```

Configure SSL

- Configure the server to listen to HTTPS traffic on port `443` by specifying a valid certificate issued by a trusted Certificate Authority (CA).
- Harden the security by employing some of the practices depicted in the following `/etc/nginx/nginx.conf` file. Examples include choosing a stronger cipher and redirecting all traffic over HTTP to HTTPS.
- Adding an `HTTP Strict-Transport-Security` (HSTS) header ensures all subsequent requests made by the client are over HTTPS only.
- Do not add the Strict-Transport-Security header or chose an appropriate `max-age` if SSL will be disabled in the future.

Add the `/etc/nginx/proxy.conf` configuration file:

```
proxy_redirect    off;
proxy_set_header  Host      $host;
proxy_set_header  X-Real-IP  $remote_addr;
proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header  X-Forwarded-Proto $scheme;
client_max_body_size 10m;
client_body_buffer_size 128k;
proxy_connect_timeout 90;
proxy_send_timeout 90;
proxy_read_timeout 90;
proxy_buffers 32 4k;
```

Edit the `/etc/nginx/nginx.conf` configuration file. The example contains both `http` and `server` sections in one configuration file.

```

http {
    include    /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens off;

    sendfile on;
    keepalive_timeout 29; # Adjust to the lowest possible value that makes sense for your use case.
    client_body_timeout 10; client_header_timeout 10; send_timeout 10;

    upstream hellomvc{
        server localhost:5000;
    }

    server {
        listen *:80;
        add_header Strict-Transport-Security max-age=15768000;
        return 301 https://$host$request_uri;
    }

    server {
        listen *:443    ssl;
        server_name     example.com;
        ssl_certificate /etc/ssl/certs/testCert.crt;
        ssl_certificate_key /etc/ssl/certs/testCert.key;
        ssl_protocols  TLSv1.1 TLSv1.2;
        ssl_prefer_server_ciphers on;
        ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
        ssl_ecdh_curve secp384r1;
        ssl_session_cache shared:SSL:10m;
        ssl_session_tickets off;
        ssl_stapling on; #ensure your cert is capable
        ssl_stapling_verify on; #ensure your cert is capable

        add_header Strict-Transport-Security "max-age=63072000; includeSubdomains; preload";
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        #Redirects all traffic
        location / {
            proxy_pass http://hellomvc;
            limit_req zone=one burst=10;
        }
    }
}

```

Secure nginx from clickjacking

Clickjacking is a malicious technique to collect an infected user's clicks. Clickjacking tricks the victim (visitor) into clicking on an infected site. Use X-FRAME-OPTIONS to secure the site.

Edit the *nginx.conf* file:

```
sudo nano /etc/nginx/nginx.conf
```

Add the line `add_header X-Frame-Options "SAMEORIGIN";` and save the file, then restart nginx.

MIME-type sniffing

This header prevents most browsers from MIME-sniffing a response away from the declared content type, as the header instructs the browser not to override the response content type. With the `nosniff` option, if the server says the content is "text/html", the browser renders it as "text/html".

Edit the *nginx.conf* file:

```
sudo nano /etc/nginx/nginx.conf
```

Add the line `add_header X-Content-Type-Options "nosniff";` and save the file, then restart nginx.

Host ASP.NET Core on Linux with Apache

1/10/2018 • 7 min to read • [Edit Online](#)

By [Shayne Boyer](#)

Using this guide, learn how to set up [Apache](#) as a reverse proxy server on [CentOS 7](#) to redirect HTTP traffic to an ASP.NET Core web app running on [Kestrel](#). The [mod_proxy extension](#) and related modules create the server's reverse proxy.

Prerequisites

1. Server running CentOS 7 with a standard user account with sudo privilege
2. ASP.NET Core app

Publish the app

Publish the app as a [self-contained deployment](#) in Release configuration for the CentOS 7 runtime (`centos.7-x64`). Copy the contents of the `bin/Release/netcoreapp2.0/centos.7-x64/publish` folder to the server using SCP, FTP, or other file transfer method.

NOTE

Under a production deployment scenario, a continuous integration workflow does the work of publishing the app and copying the assets to the server.

Configure a proxy server

A reverse proxy is a common setup for serving dynamic web apps. The reverse proxy terminates the HTTP request and forwards it to the ASP.NET app.

A proxy server is one which forwards client requests to another server instead of fulfilling them itself. A reverse proxy forwards to a fixed destination, typically on behalf of arbitrary clients. In this guide, Apache is configured as the reverse proxy running on the same server that Kestrel is serving the ASP.NET Core app.

Install Apache

Update CentOS packages to their latest stable versions:

```
sudo yum update -y
```

Install the Apache web server on CentOS with a single `yum` command:

```
sudo yum -y install httpd mod_ssl
```

Sample output after running the command:

```
Downloading packages:
httpd-2.4.6-40.el7.centos.4.x86_64.rpm | 2.7 MB 00:00:01
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Installing : httpd-2.4.6-40.el7.centos.4.x86_64 1/1
Verifying : httpd-2.4.6-40.el7.centos.4.x86_64 1/1

Installed:
httpd.x86_64 0:2.4.6-40.el7.centos.4

Complete!
```

NOTE

In this example, the output reflects `httpd.x86_64` since the CentOS 7 version is 64 bit. To verify where Apache is installed, run `whereis httpd` from a command prompt.

Configure Apache for reverse proxy

Configuration files for Apache are located within the `/etc/httpd/conf.d/` directory. Any file with the `.conf` extension is processed in alphabetical order in addition to the module configuration files in `/etc/httpd/conf.modules.d/`, which contains any configuration files necessary to load modules.

Create a configuration file for the app named `hellomvc.conf`:

```
<VirtualHost *:80>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ErrorLog /var/log/httpd/hellomvc-error.log
    CustomLog /var/log/httpd/hellomvc-access.log common
</VirtualHost>
```

The **VirtualHost** node can appear multiple times in one or more files on a server. **VirtualHost** is set to listen on any IP address using port 80. The next two lines are set to proxy requests at the root to the server at 127.0.0.1 on port 5000. For bi-directional communication, *ProxyPass* and *ProxyPassReverse* are required.

Logging can be configured per **VirtualHost** using **ErrorLog** and **CustomLog** directives. **ErrorLog** is the location where the server logs errors, and **CustomLog** sets the filename and format of log file. In this case, this is where request information is logged. There is one line for each request.

Save the file and test the configuration. If everything passes, the response should be `Syntax [OK]`.

```
sudo service httpd configtest
```

Restart Apache:

```
sudo systemctl restart httpd
sudo systemctl enable httpd
```

Monitoring the app

Apache is now setup to forward requests made to `http://localhost:80` to the ASP.NET Core app running on

Kestrel at `http://127.0.0.1:5000`. However, Apache isn't set up to manage the Kestrel process. Use *systemd* and create a service file to start and monitor the underlying web app. *systemd* is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

```
sudo nano /etc/systemd/system/kestrel-hellovc.service
```

An example service file for the app:

```
[Unit]
Description=Example .NET Web API App running on CentOS 7

[Service]
WorkingDirectory=/var/aspnetcore/hellovc
ExecStart=/usr/local/bin/dotnet /var/aspnetcore/hellovc/hellovc.dll
Restart=always
# Restart service after 10 seconds if dotnet service crashes
RestartSec=10
SyslogIdentifier=dotnet-example
User=apache
Environment=ASPNETCORE_ENVIRONMENT=Production

[Install]
WantedBy=multi-user.target
```

NOTE

User — If the user *apache* isn't used by the configuration, the user must be created first and given proper ownership for files.

Save the file and enable the service:

```
systemctl enable kestrel-hellovc.service
```

Start the service and verify that it's running:

```
systemctl start kestrel-hellovc.service
systemctl status kestrel-hellovc.service
```

- kestrel-hellovc.service - Example .NET Web API App running on CentOS 7
 - Loaded: loaded (/etc/systemd/system/kestrel-hellovc.service; enabled)
 - Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
 - Main PID: 9021 (dotnet)
 - CGroup: /system.slice/kestrel-hellovc.service
 - └─9021 /usr/local/bin/dotnet /var/aspnetcore/hellovc/hellovc.dll

With the reverse proxy configured and Kestrel managed through *systemd*, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. Inspecting the response headers, the

Server header indicates that the ASP.NET Core app is served by Kestrel:

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

Viewing logs

Since the web app using Kestrel is managed using *systemd*, events and processes are logged to a centralized journal. However, this journal includes entries for all of the services and processes managed by *systemd*. To view the `kestrel-helloMvc.service`-specific items, use the following command:

```
sudo journalctl -fu kestrel-helloMvc.service
```

For time filtering, specify time options with the command. For example, use `--since today` to filter for the current day or `--until 1 hour ago` to see the previous hour's entries. For more information, see the [man page for journalctl](#).

```
sudo journalctl -fu kestrel-helloMvc.service --since "2016-10-18" --until "2016-10-18 04:00"
```

Securing the app

Configure firewall

Firewalld is a dynamic daemon to manage the firewall with support for network zones. Ports and packet filtering can still be managed by *iptables*. *Firewalld* should be installed by default. `yum` can be used to install the package or verify it's installed.

```
sudo yum install firewalld -y
```

Use `firewalld` to open only the ports needed for the app. In this case, port 80 and 443 are used. The following commands permanently set ports 80 and 443 to open:

```
sudo firewall-cmd --add-port=80/tcp --permanent
sudo firewall-cmd --add-port=443/tcp --permanent
```

Reload the firewall settings. Check the available services and ports in the default zone. Options are available by inspecting `firewall-cmd -h`.

```
sudo firewall-cmd --reload
sudo firewall-cmd --list-all
```

```
public (default, active)
interfaces: eth0
sources:
services: dhcpv6-client
ports: 443/tcp 80/tcp
masquerade: no
forward-ports:
icmp-blocks:
rich rules:
```

SSL configuration

To configure Apache for SSL, the `mod_ssl` module is used. When the `httpd` module was installed, the `mod_ssl` module was also installed. If it wasn't installed, use `yum` to add it to the configuration.

```
sudo yum install mod_ssl
```

To enforce SSL, install the `mod_rewrite` module to enable URL rewriting:

```
sudo yum install mod_rewrite
```

Modify the `hellomvc.conf` file to enable URL rewriting and secure communication on port 443:

```
<VirtualHost *:80>
    RewriteEngine On
    RewriteCond %{HTTPS} !=on
    RewriteRule ^/?(.*) https://%{SERVER_NAME}/ [R,L]
</VirtualHost>

<VirtualHost *:443>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ErrorLog /var/log/httpd/hellomvc-error.log
    CustomLog /var/log/httpd/hellomvc-access.log common
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
</VirtualHost>
```

NOTE

This example is using a locally-generated certificate. **SSLCertificateFile** should be the primary certificate file for the domain name. **SSLCertificateKeyFile** should be the key file generated when CSR is created. **SSLCertificateChainFile** should be the intermediate certificate file (if any) that was supplied by the certificate authority.

Save the file and test the configuration:

```
sudo service httpd configtest
```

Restart Apache:

```
sudo systemctl restart httpd
```

Additional Apache suggestions

Additional headers

In order to secure against malicious attacks, there are a few headers that should either be modified or added.

Ensure that the `mod_headers` module is installed:

```
sudo yum install mod_headers
```

Secure Apache from clickjacking attacks

[Clickjacking](#), also known as a *UI redress attack*, is a malicious attack where a website visitor is tricked into clicking a link or button on a different page than they're currently visiting. Use `X-FRAME-OPTIONS` to secure the site.

Edit the `httpd.conf` file:

```
sudo nano /etc/httpd/conf/httpd.conf
```

Add the line `Header append X-FRAME-OPTIONS "SAMEORIGIN"`. Save the file. Restart Apache.

MIME-type sniffing

The `X-Content-Type-Options` header prevents Internet Explorer from *MIME-sniffing* (determining a file's `Content-Type` from the file's content). If the server sets the `Content-Type` header to `text/html` with the `nosniff` option set, Internet Explorer renders the content as `text/html` regardless of the file's content.

Edit the `httpd.conf` file:

```
sudo nano /etc/httpd/conf/httpd.conf
```

Add the line `Header set X-Content-Type-Options "nosniff"`. Save the file. Restart Apache.

Load Balancing

This example shows how to setup and configure Apache on CentOS 7 and Kestrel on the same instance machine. In order to not have a single point of failure; using `mod_proxy_balancer` and modifying the **VirtualHost** would allow for managing multiple instances of the web apps behind the Apache proxy server.

```
sudo yum install mod_proxy_balancer
```

In the configuration file shown below, an additional instance of the `hellomvc` app is setup to run on port 5001. The *Proxy* section is set with a balancer configuration with two members to load balance *byrequests*.

```

<VirtualHost *:80>
    RewriteEngine On
    RewriteCond %{HTTPS} !=on
    RewriteRule ^/?(.*) https://%{SERVER_NAME}/ [R,L]
</VirtualHost>

<VirtualHost *:443>
    ProxyPass / balancer://mycluster/

    ProxyPassReverse / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5001/

    <Proxy balancer://mycluster>
        BalancerMember http://127.0.0.1:5000
        BalancerMember http://127.0.0.1:5001
        ProxySet lbmethod=byrequests
    </Proxy>

    <Location />
        SetHandler balancer
    </Location>
    ErrorLog /var/log/httpd/hellomvc-error.log
    CustomLog /var/log/httpd/hellomvc-access.log common
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
</VirtualHost>

```

Rate Limits

Using *mod_ratelimit*, which is included in the *httpd* module, the bandwidth of clients can be limited:

```
sudo nano /etc/httpd/conf.d/ratelimit.conf
```

The example file limits bandwidth as 600 KB/sec under the root location:

```

<IfModule mod_ratelimit.c>
    <Location />
        SetOutputFilter RATE_LIMIT
        SetEnv rate-limit 600
    </Location>
</IfModule>

```

Host ASP.NET Core in Docker containers

1/10/2018 • 1 min to read • [Edit Online](#)

The following articles are available for learning about hosting ASP.NET Core apps in Docker:

[Introduction to Containers and Docker](#)

See how containerization is an approach to software development in which an application or service, its dependencies, and its configuration are packaged together as a container image. The image can be tested and then deployed to a host.

[What is Docker](#)

Discover how Docker is an open-source project for automating the deployment of apps as portable, self-sufficient containers that can run on the cloud or on-premises.

[Docker Terminology](#)

Learn terms and definitions for Docker technology.

[Docker containers, images, and registries](#)

Find out how Docker container images are stored in an image registry for consistent deployment across environments.

[Building Docker Images for .NET Core Applications](#)

Learn how to build and dockerize an ASP.NET Core app. Explore Docker images maintained by Microsoft and examine use cases.

[Visual Studio Tools for Docker](#)

Discover how Visual Studio 2017 supports building, debugging, and running ASP.NET Core apps targeting either .NET Framework or .NET Core on Docker for Windows. Both Windows and Linux containers are supported.

[Publish to a Docker Image](#)

Find out how to use the Visual Studio Tools for Docker extension to deploy an ASP.NET Core app to a Docker host on Azure using PowerShell.

Visual Studio Tools for Docker with ASP.NET Core

1/10/2018 • 6 min to read • [Edit Online](#)

Visual Studio 2017 supports building, debugging, and running ASP.NET Core apps targeting either .NET Framework or .NET Core. Both Windows and Linux containers are supported.

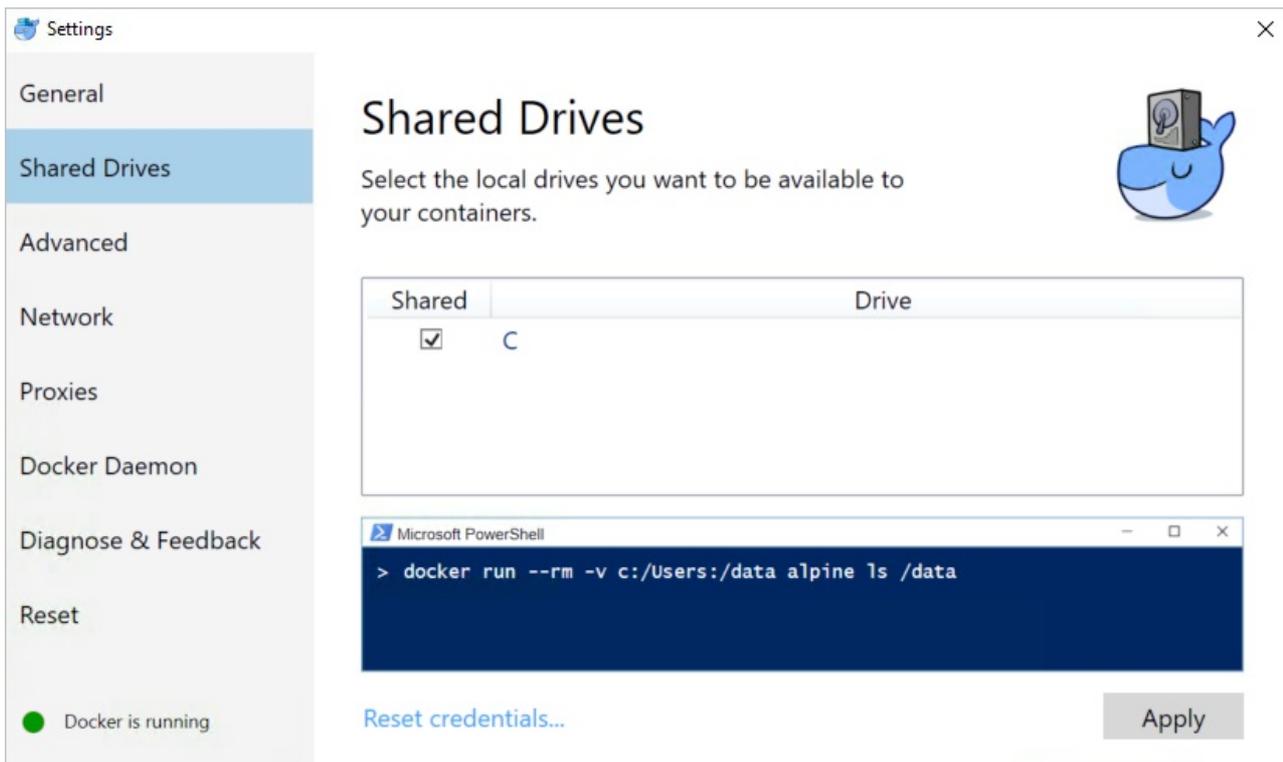
Prerequisites

- [Visual Studio 2017](#) with the **.NET Core cross-platform development** workload
- [Docker for Windows](#)

Installation and setup

For Docker installation, review the information at [Docker for Windows: What to know before you install](#) and install [Docker For Windows](#).

Shared Drives in Docker for Windows must be configured to support volume mapping and debugging. Right-click the System Tray's Docker icon, select **Settings...**, and select **Shared Drives**. Select the drive where Docker stores files. Select **Apply**.



Settings

General

Shared Drives

Advanced

Network

Proxies

Docker Daemon

Diagnose & Feedback

Reset

Docker is running

Shared Drives

Select the local drives you want to be available to your containers.

Shared	Drive
<input checked="" type="checkbox"/>	C

```
Microsoft PowerShell
> docker run --rm -v c:/Users:/data alpine ls /data
```

[Reset credentials...](#) Apply

TIP

Visual Studio 2017 versions 15.6 and later prompt when **Shared Drives** aren't configured.

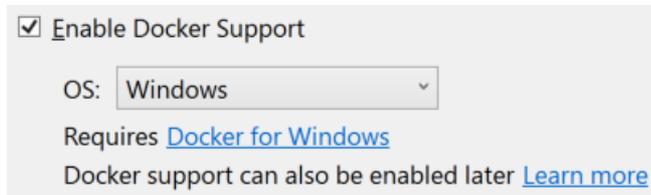
Add Docker support to an app

The ASP.NET Core project's target framework determines the supported container types. Projects targeting .NET Core support both Linux and Windows containers. Projects targeting .NET Framework only support Windows containers.

When adding Docker support to a project, choose either a Windows or a Linux container. The Docker host must be running the same container type. To change the container type in the running Docker instance, right-click the System Tray's Docker icon and choose **Switch to Windows containers...** or **Switch to Linux containers...**

New app

When creating a new app with the **ASP.NET Core Web Application** project templates, select the **Enable Docker Support** checkbox:



If the target framework is .NET Core, the **OS** drop-down allows for the selection of a container type.

Existing app

The Visual Studio Tools for Docker don't support adding Docker to an existing ASP.NET Core project targeting .NET Framework. For ASP.NET Core projects targeting .NET Core, there are two options for adding Docker support via the tooling. Open the project in Visual Studio, and choose one of the following options:

- Select **Docker Support** from the **Project** menu.
- Right-click the project in Solution Explorer and select **Add > Docker Support**.

Docker assets overview

The Visual Studio Tools for Docker add a *docker-compose* project to the solution, containing the following:

- *.dockerignore*: Contains a list of file and directory patterns to exclude when generating a build context.
- *docker-compose.yml*: The base [Docker Compose](#) file used to define the collection of images to be built and run with `docker-compose build` and `docker-compose run`, respectively.
- *docker-compose.override.yml*: An optional file, read by Docker Compose, containing configuration overrides for services. Visual Studio executes `docker-compose -f "docker-compose.yml" -f "docker-compose.override.yml"` to merge these files.

A *Dockerfile*, the recipe for creating a final Docker image, is added to the project root. Refer to [Dockerfile reference](#) for an understanding of the commands within it. This particular *Dockerfile* uses a [multi-stage build](#) containing four distinct, named build stages:

```

FROM microsoft/aspnetcore:2.0-nanoserver-1709 AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/aspnetcore-build:2.0-nanoserver-1709 AS build
WORKDIR /src
COPY *.sln ./
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore
COPY . .
WORKDIR /src/HelloDockerTools
RUN dotnet build -c Release -o /app

FROM build AS publish
RUN dotnet publish -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]

```

The *Dockerfile* is based on the [microsoft/aspnetcore](#) image. This base image includes the ASP.NET Core NuGet packages, which have been pre-jitted to improve startup performance.

The *docker-compose.yml* file contains the name of the image that is created when the project runs:

```

version: '3'

services:
  helldockertools:
    image: helldockertools
    build:
      context: .
      dockerfile: HelloDockerTools\Dockerfile

```

In the preceding example, `image: helldockertools` generates the image `helldockertools:dev` when the app runs in **Debug** mode. The `helldockertools:latest` image is generated when the app runs in **Release** mode.

Prefix the image name with the [Docker Hub](#) username (for example, `dockerhubusername/helldockertools`) if the image will be pushed to the registry. Alternatively, change the image name to include the private registry URL (for example, `privateregistry.domain.com/helldockertools`) depending on the configuration.

Debug

Select **Docker** from the debug drop-down in the toolbar, and start debugging the app. The **Docker** view of the **Output** window shows the following actions taking place:

- The *microsoft/aspnetcore* runtime image is acquired (if not already in the cache).
- The *microsoft/aspnetcore-build* compile/publish image is acquired (if not already in the cache).
- The `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development` within the container.
- Port 80 is exposed and mapped to a dynamically-assigned port for localhost. The port is determined by the Docker host and can be queried with the `docker ps` command.
- The app is copied to the container.
- The default browser is launched with the debugger attached to the container using the dynamically-assigned port.

The resulting Docker image is the *dev* image of the app with the *microsoft/aspnetcore* images as the base image. Run the `docker images` command in the **Package Manager Console** (PMC) window. The images on the machine

are displayed:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hellodockertools	latest	f8f9d6c923e2	About an hour ago	391MB
hellodockertools	dev	85c5ffee5258	About an hour ago	389MB
microsoft/aspnetcore-build	2.0-nanoserver-1709	d7cce94e3eb0	15 hours ago	1.86GB
microsoft/aspnetcore	2.0-nanoserver-1709	8872347d7e5d	40 hours ago	389MB

NOTE

The dev image lacks the app contents, as **Debug** configurations use volume mounting to provide the iterative experience. To push an image, use the **Release** configuration.

Run the `docker ps` command in PMC. Notice the app is running using the container:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
baf9a678c88d	hellodockertools:dev	"C:\\remote_debugge..."	21 seconds ago	Up 19 seconds
0.0.0.0:37630->80/tcp	dockercompose4642749010770307127_hellodockertools_1			

Edit and continue

Changes to static files and Razor views are automatically updated without the need for a compilation step. Make the change, save, and refresh the browser to view the update.

Modifications to code files requires compiling and a restart of Kestrel within the container. After making the change, use CTRL + F5 to perform the process and start the app within the container. The Docker container isn't rebuilt or stopped. Run the `docker ps` command in PMC. Notice the original container is still running as of 10 minutes ago:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
baf9a678c88d	hellodockertools:dev	"C:\\remote_debugge..."	10 minutes ago	Up 10 minutes
0.0.0.0:37630->80/tcp	dockercompose4642749010770307127_hellodockertools_1			

Publish Docker images

Once the develop and debug cycle of the app is completed, the Visual Studio Tools for Docker assist in creating the production image of the app. Change the configuration drop-down to **Release** and build the app. The tooling produces the image with the *latest* tag, which can be pushed to the private registry or Docker Hub.

Run the `docker images` command in PMC to see the list of images:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hellodockertools	latest	4cb1fca533f0	19 seconds ago	391MB
hellodockertools	dev	85c5ffee5258	About an hour ago	389MB
microsoft/aspnetcore-build	2.0-nanoserver-1709	d7cce94e3eb0	16 hours ago	1.86GB
microsoft/aspnetcore	2.0-nanoserver-1709	8872347d7e5d	40 hours ago	389MB

NOTE

The `docker images` command returns intermediary images with repository names and tags identified as `<none>` (not listed above). These unnamed images are produced by the [multi-stage build Dockerfile](#). They improve the efficiency of building the final image—only the necessary layers are rebuilt when changes occur. When the intermediary images are no longer needed, delete them using the `docker rmi` command.

There may be an expectation for the production or release image to be smaller in size by comparison to the *dev* image. Because of the volume mapping, the debugger and app were running from the local machine and not within the container. The *latest* image has packaged the necessary app code to run the app on a host machine. Therefore, the delta is the size of the app code.

Visual Studio publish profiles for ASP.NET Core app deployment

1/10/2018 • 11 min to read • [Edit Online](#)

By [Sayed Ibrahim Hashimi](#) and [Rick Anderson](#)

This article focuses on using Visual Studio 2017 to create publish profiles. The publish profiles created with Visual Studio can be run from MSBuild and Visual Studio 2017. The article provides details of the publishing process. See [Publish an ASP.NET Core web app to Azure App Service using Visual Studio](#) for instructions on publishing to Azure.

The following `.csproj` file was created with the command `dotnet new mvc`:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>

</Project>
```

The `Sdk` attribute in the `<Project>` element (in the first line) of the markup above does the following:

- Imports the properties file from `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.Props` at the beginning.
- Imports the targets file from `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.targets` at the end.

The default location for `MSBuildSDKsPath` (with Visual Studio 2017 Enterprise) is the `%programfiles(x86)%\Microsoft Visual Studio\2017\Enterprise\MSBuild\Sdks` folder.

`Microsoft.NET.Sdk.Web` depends on:

- `Microsoft.NET.Sdk.Web.ProjectSystem`
- `Microsoft.NET.Sdk.Publish`

Which causes the following properties and targets to be imported:

- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web.ProjectSystem\Sdk\Sdk.Props`
- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web.ProjectSystem\Sdk\Sdk.targets`
- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Publish\Sdk\Sdk.Props`
- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Publish\Sdk\Sdk.targets`

Publish targets import the right set of targets based on the publish method used.

When MSBuild or Visual Studio loads a project, the following high level actions are performed:

- Build project
- Compute files to publish
- Publish files to destination

Compute project items

When the project is loaded, the project items (files) are computed. The `item type` attribute determines how the file is processed. By default, `.cs` files are included in the `Compile` item list. Files in the `Compile` item list are compiled.

The `Content` item list contains files that are published in addition to the build outputs. By default, files matching the pattern `wwwroot/**` are included in the `Content` item. `wwwroot/**` is a [globbing pattern](#) that specifies all files in the `wwwroot` folder **and** subfolders. To explicitly add a file to the publish list, add the file directly in the `.csproj` file as shown in [Including Files](#).

When selecting the **Publish** button in Visual Studio or when publishing from the command line:

- The properties/items are computed (the files that are needed to build).
- Visual Studio only: NuGet packages are restored. (Restore needs to be explicit by the user on the CLI.)
- The project builds.
- The publish items are computed (the files that are needed to publish).
- The project is published. (The computed files are copied to the publish destination.)

When an ASP.NET Core project references `Microsoft.NET.Sdk.Web` in the project file, an `app_offline.htm` file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the `app_offline.htm` file during the deployment. For more information, see the [ASP.NET Core Module configuration reference](#).

Basic command-line publishing

Command-line publishing works on all .NET Core supported platforms and doesn't require Visual Studio. In the samples below, the `dotnet publish` command is run from the project directory (which contains the `.csproj` file). If not in the project folder, explicitly pass in the project file path. For example:

```
dotnet publish c:/webs/web1
```

Run the following commands to create and publish a web app:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
dotnet new mvc
dotnet publish
```

The `dotnet publish` produces output similar to the following:

```
C:\Webs\Web1>dotnet publish
Microsoft (R) Build Engine version 15.3.409.57025 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Web1 -> C:\Webs\Web1\bin\Debug\netcoreapp2.0\Web1.dll
Web1 -> C:\Webs\Web1\bin\Debug\netcoreapp2.0\publish\
```

The default publish folder is `bin\$(Configuration)\netcoreapp<version>\publish`. The default for `$(Configuration)`

is Debug. In the sample above, the `<TargetFramework>` is `netcoreapp2.0`.

`dotnet publish -h` displays help information for publish.

The following command specifies a `Release` build and the publishing directory:

```
dotnet publish -c Release -o C:/MyWebs/test
```

The `dotnet publish` command calls MSBuild which invokes the `Publish` target. Any parameters passed to `dotnet publish` are passed to MSBuild. The `-c` parameter maps to the `Configuration` MSBuild property. The `-o` parameter maps to `OutputPath`.

MSBuild properties can be passed using either of the following formats:

- `p:<NAME>=<VALUE>`
- `/p:<NAME>=<VALUE>`

The following command publishes a `Release` build to a network share:

```
dotnet publish -c Release /p:PublishDir=//r8/release/AdminWeb
```

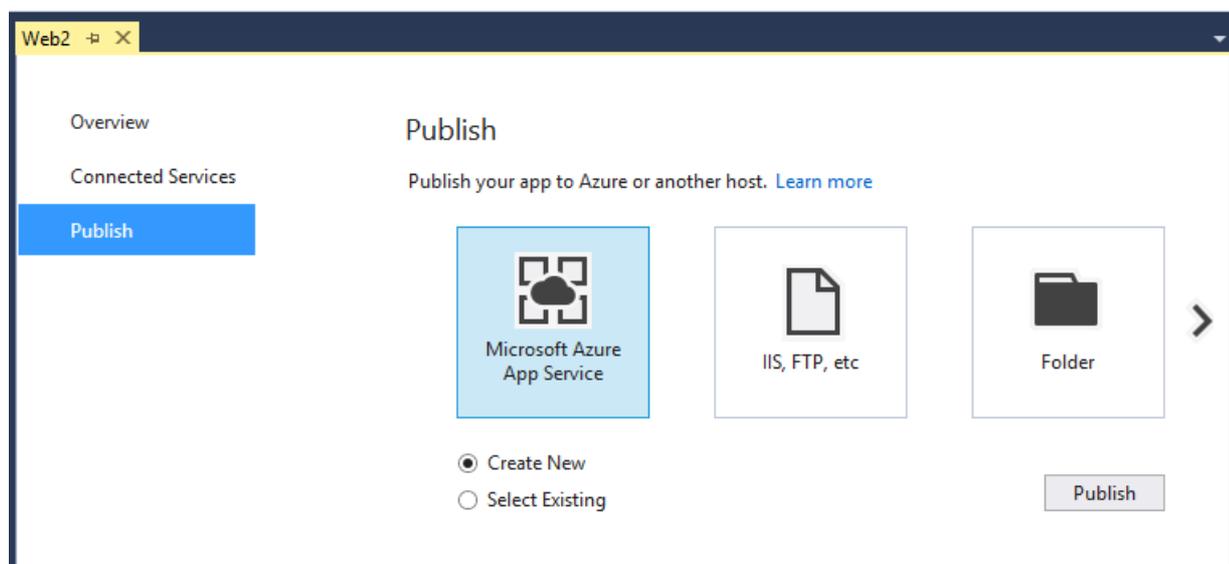
The network share is specified with forward slashes (`//r8/`) and works on all .NET Core supported platforms.

Confirm that the published app for deployment isn't running. Files in the `publish` folder are locked when the app is running. Deployment can't occur because locked files can't be copied.

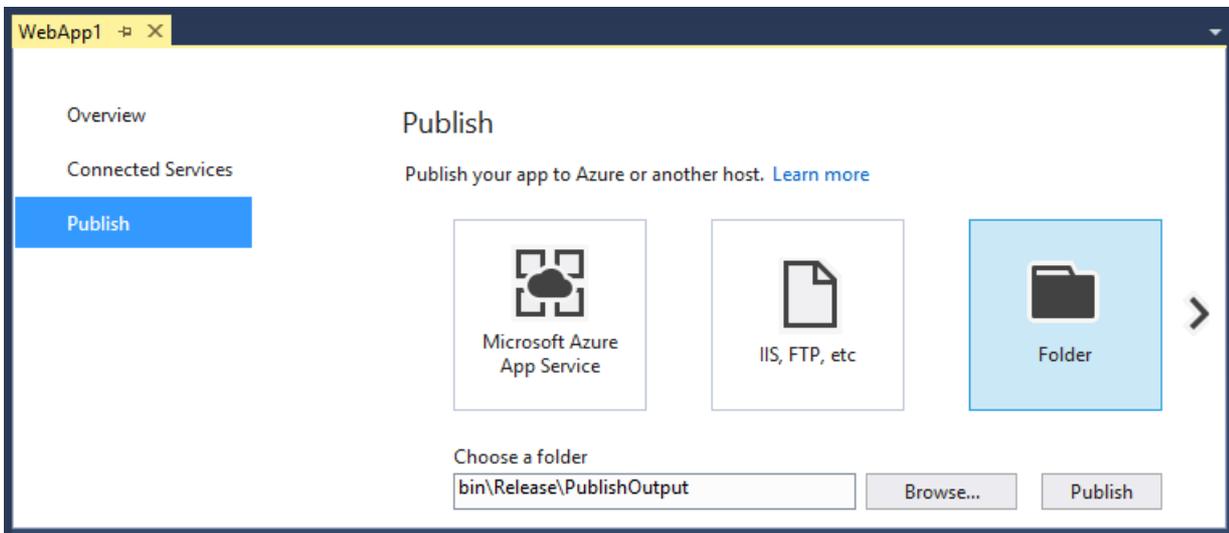
Publish profiles

This section uses Visual Studio 2017 and higher to create publishing profiles. Once created, publishing from Visual Studio or the command line is available.

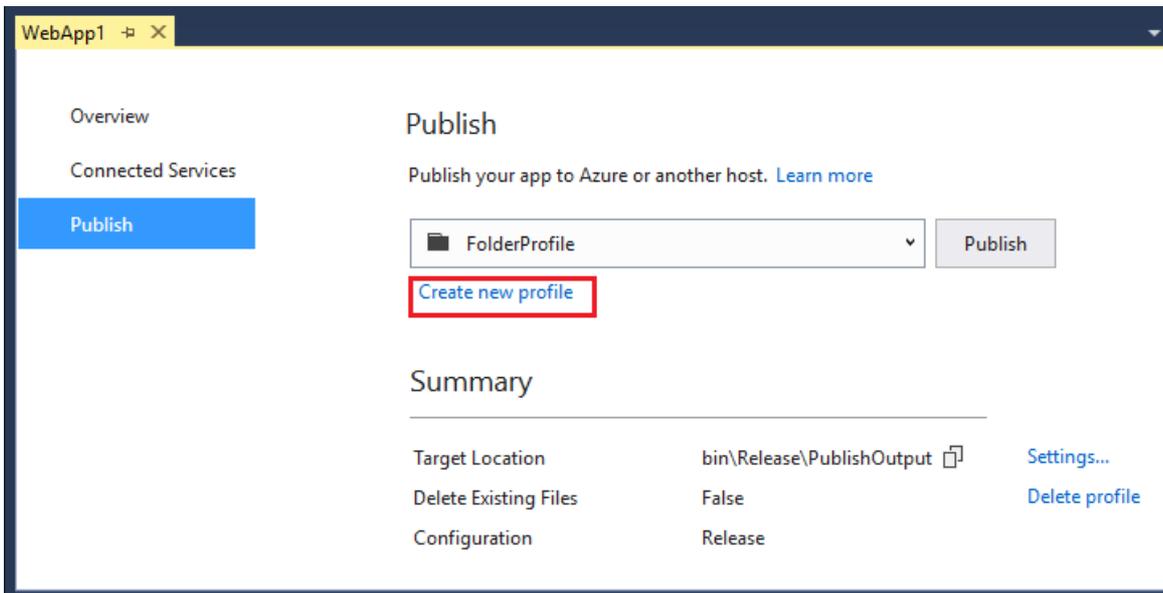
Publish profiles can simplify the publishing process. Multiple publish profiles can exist. To create a publish profile in Visual Studio, right-click on the project in Solution Explorer and select **Publish**. Alternatively, select **Publish <project name>** from the build menu. The **Publish** tab of the application capacities page is displayed. If the project doesn't contain a publish profile, the following page is displayed:



When **Folder** is selected, the **Publish** button creates a folder publish profile and publishes.



Once a publish profile is created, the **Publish** tab changes, and select **Create new profile** to create a new profile.



The Publish wizard supports the following publish targets:

- Microsoft Azure App Service
- IIS, FTP, etc (for any web server)
- Folder
- Import profile (allows profile import).
- Microsoft Azure Virtual Machines

See [What publishing options are right for me?](#) for more information.

When creating a publish profile with Visual Studio, a *Properties/PublishProfiles/<publish name>.pubxml* MSBuild file is created. This *.pubxml* file is a MSBuild file and contains publish configuration settings. This file can be changed to customize the build and publish process. This file is read by the publishing process.

`<LastUsedBuildConfiguration>` is special because it's a global property and shouldn't be in any file that's imported in the build. See [MSBuild: how to set the configuration property](#) for more info. The *.pubxml* file shouldn't be checked into source control because it depends on the *.user* file. The *.user* file should never be checked into source control because it can contain sensitive information and it's only valid for one user and machine.

Sensitive information (like the publish password) is encrypted on a per user/machine level and stored in the *Properties/PublishProfiles/<publish name>.pubxml.user* file. Because this file can contain sensitive information, it should **not** be checked into source control.

For an overview of how to publish a web app on ASP.NET Core see [Host and deploy](#). [Host and deploy](#) is an open source project at <https://github.com/aspnet/websdk>.

`dotnet publish` can use folder, MSDeploy, and KUDU publish profiles:

Folder (works cross-platform): `dotnet publish WebApplication.csproj /p:PublishProfile=<FolderProfileName>`

MSDeploy (currently this only works in windows since MSDeploy isn't cross-platform):

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployProfileName> /p:Password=<DeploymentPassword>
```

MSDeploy package(currently this only works in windows since MSDeploy isn't cross-platform):

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployPackageProfileName>
```

In the preceding samples, **don't** pass `deployonbuild` to `dotnet publish`.

For more information, see [Microsoft.NET.Sdk.Publish](#).

`dotnet publish` supports KUDU apis to publish to Azure from any platform. Visual Studio publish does support the KUDU APIs but it is supported by websdk for cross plat publish to Azure.

Add a publish profile to *Properties/PublishProfiles* folder with the following content:

```
<Project>
  <PropertyGroup>
    <PublishProtocol>Kudu</PublishProtocol>
    <PublishSiteName>nodewebapp</PublishSiteName>
    <UserName>username</UserName>
    <Password>password</Password>
  </PropertyGroup>
</Project>
```

Running the following command zips up the publish contents and publish it to Azure using the KUDU APIs:

```
dotnet publish /p:PublishProfile=Azure /p:Configuration=Release
```

Set the following MSBuild properties when using a publish profile:

- `DeployOnBuild=true`
- `PublishProfile=<Publish profile name>`

When publishing with a profile named *FolderProfile*, either of the commands below can be executed:

- `dotnet build /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`
- `msbuild /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`

When invoking `dotnet build`, it calls `msbuild` to run the build and publish process. Calling `dotnet build` or `msbuild` is essentially equivalent when passing in a folder profile. When calling MSBuild directly on Windows, the .NET Framework version of MSBuild is used. MSDeploy is currently limited to Windows machines for publishing. Calling `dotnet build` on a non-folder profile invokes MSBuild, and MSBuild uses MSDeploy on non-folder profiles. Calling `dotnet build` on a non-folder profile invokes MSBuild (using MSDeploy) and results in a failure (even when running on a Windows platform). To publish with a non-folder profile, call MSBuild directly.

The following folder publish profile was created with Visual Studio and publishes to a network share:

```

<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project.
You can customize the behavior of this process by editing this
MSBuild file.
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework>netcoreapp1.1</PublishFramework>
    <ProjectGuid>c30c453c-312e-40c4-aec9-394a145dee0b</ProjectGuid>
    <publishUrl>\\r8\Release\AdminWeb</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
</Project>

```

Note `<LastUsedBuildConfiguration>` is set to `Release`. When publishing from Visual Studio, the `<LastUsedBuildConfiguration>` configuration property value is set using the value when the publish process is started. The `<LastUsedBuildConfiguration>` configuration property is special and shouldn't be overridden in an imported MSBuild file. This property can be overridden from the command line. For example:

```
dotnet build -c Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

Using MSBuild:

```
msbuild /p:Configuration=Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

Publish to an MSDeploy endpoint from the command line

As previously mentioned, publishing can be accomplished using `dotnet publish` or the `msbuild` command. `dotnet publish` runs in the context of .NET Core. The `msbuild` command requires .NET framework, and is therefore limited to Windows environments.

The easiest way to publish with MSDeploy is to first create a publish profile in Visual Studio 2017 and use the profile from the command line.

In the following sample, an ASP.NET Core web app is created (using `dotnet new mvc`), and an Azure publish profile is added with Visual Studio.

Run `msbuild` from a **Developer Command Prompt for VS 2017**. The Developer Command Prompt has the correct `msbuild.exe` in its path with some MSBuild variables set.

MSBuild uses the following syntax:

```
msbuild <path-to-project-file> /p:DeployOnBuild=true /p:PublishProfile=<Publish Profile> /p:Username=<USERNAME> /p>Password=<PASSWORD>
```

Get the `Password` from the `<Publish name>.PublishSettings` file. Download the `.PublishSettings` file from either:

- Solution Explorer: Right-click on the Web App and select **Download Publish Profile**.
- The Azure Management Portal: Select **Get publish profile** from the Web App blade.

`Username` can be found in the publish profile.

The following sample uses the "Web11112 - Web Deploy" publish profile:

```
msbuild "C:\Webs\Web1\Web1.csproj" /p:DeployOnBuild=true
/p:PublishProfile="Web11112 - Web Deploy" /p:Username="$Web11112"
/p:Password="<password removed>"
```

Excluding files

When publishing ASP.NET Core web apps, the build artifacts and contents of the *wwwroot* folder are included.

`msbuild` supports [globbing patterns](#). For example, the following `<Content>` element markup excludes all text (*.txt*) files from the *wwwroot/content* folder and all its subfolders.

```
<ItemGroup>
  <Content Update="wwwroot/content/**/*.txt" CopyToPublishDirectory="Never" />
</ItemGroup>
```

The markup above can be added to a publish profile or the *.csproj* file. When added to the *.csproj* file, the rule is added to all publish profiles in the project.

The following `<MsDeploySkipRules>` element markup excludes all files from the *wwwroot/content* folder:

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFolder">
    <ObjectName>dirPath</ObjectName>
    <AbsolutePath>wwwroot\content</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

`<MsDeploySkipRules>` won't delete the *skip* targets from the deployment site. `<Content>` targeted files and folders are deleted from the deployment site. For example, suppose a deployed web app had the following files:

- *Views/Home/About1.cshtml*
- *Views/Home/About2.cshtml*
- *Views/Home/About3.cshtml*

If the following `<MsDeploySkipRules>` markup is added, those files wouldn't be deleted on the deployment site.

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\Home\About1.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\Home\About2.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\Home\About3.cshtml</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

The `<MsDeploySkipRules>` markup shown above prevents the *skipped* files from being deployed but won't delete those files once they are deployed.

The following `<Content>` markup deletes the targeted files at the deployment site:

```
<ItemGroup>
  <Content Update="Views/Home/About?.cshtml" CopyToPublishDirectory="Never" />
</ItemGroup>
```

Using command-line deployment with the `<Content>` markup above results in output similar to the following:

```
MSDeployPublish:
  Starting Web deployment task from source:
manifest(C:\Webs\Web1\obj\Release\netcoreapp1.1\PubTmp\Web1.SourceManifest.
xml) to Destination: auto().
Deleting file (Web11112\Views\Home>About1.cshtml).
Deleting file (Web11112\Views\Home>About2.cshtml).
Deleting file (Web11112\Views\Home>About3.cshtml).
Updating file (Web11112\web.config).
Updating file (Web11112\Web1.deps.json).
Updating file (Web11112\Web1.dll).
Updating file (Web11112\Web1.pdb).
Updating file (Web11112\Web1.runtimeconfig.json).
Successfully executed Web deployment task.
Publish Succeeded.
Done Building Project "C:\Webs\Web1\Web1.csproj" (default targets).
```

Including files

The following markup includes an *images* folder outside the project directory to the *wwwroot/images* folder of the publish site:

```
<ItemGroup>
  <_CustomFiles Include="$(MSBuildProjectDirectory)/../images/**/*" />
  <DotnetPublishFiles Include="@(_CustomFiles)">
    <DestinationRelativePath>wwwroot/images/%(RecursiveDir)%(Filename)%(Extension)</DestinationRelativePath>
  </DotnetPublishFiles>
</ItemGroup>
```

The markup can be added to the *.csproj* file or the publish profile. If it's added to the *.csproj* file, it's included in each publish profile in the project.

The following highlighted markup shows how to:

- Copy a file from outside the project into the *wwwroot* folder.
- Exclude the *wwwroot\Content* folder.
- Exclude *Views\Home>About2.cshtml*.

```

<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project.
You can customize the behavior of this process by editing this
MSBuild file.
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework />
    <ProjectGuid>afa9f185-7ce0-4935-9da1-ab676229d68a</ProjectGuid>
    <publishUrl>bin\Release\PublishOutput</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
  <ItemGroup>
    <ResolvedFileToPublish Include="..\ReadMe2.MD">
      <RelativePath>wwwroot\ReadMe2.MD</RelativePath>
    </ResolvedFileToPublish>

    <Content Update="wwwroot\Content\**\*" CopyToPublishDirectory="Never" />
    <Content Update="Views\Home\About2.cshtml" CopyToPublishDirectory="Never" />

  </ItemGroup>
</Project>

```

See the [WebSDK Readme](#) for more deployment samples.

Run a target before or after publishing

The built-in `BeforePublish` and `AfterPublish` targets can be used to execute a target before or after the publish target. The following markup can be added to the publish profile to log messages to the console output before and after publishing:

```

<Target Name="CustomActionsBeforePublish" BeforeTargets="BeforePublish">
  <Message Text="Inside BeforePublish" Importance="high" />
</Target>
<Target Name="CustomActionsAfterPublish" AfterTargets="AfterPublish">
  <Message Text="Inside AfterPublish" Importance="high" />
</Target>

```

The Kudu service

To view the files in an Azure App Service web app deployment, use the [Kudu service](#). Append the `scm` token to the name of the web app. For example:

URL	RESULT
<code>http://mysite.azurewebsites.net/</code>	Web App
<code>http://mysite.scm.azurewebsites.net/</code>	Kudu service

Select the [Debug Console](#) menu item to view/edit/delete/add files.

Additional resources

- [Web Deploy](#) (MSDeploy) simplifies deployment of web apps and websites to IIS servers.
- <https://github.com/aspnet/websdk>: File issues and request features for deployment.

Directory structure of published ASP.NET Core apps

1/10/2018 • 1 min to read • [Edit Online](#)

By [Luke Latham](#)

In ASP.NET Core, the application directory, *publish*, is comprised of application files, config files, static assets, packages, and the runtime (for self-contained apps).

APP TYPE	DIRECTORY STRUCTURE
Framework-dependent Deployment	<ul style="list-style-type: none">• publish*<ul style="list-style-type: none">◦ logs* (if included in publishOptions)◦ refs*◦ runtimes*◦ Views* (if included in publishOptions)◦ wwwroot* (if included in publishOptions)◦ .dll files◦ myapp.deps.json◦ myapp.dll◦ myapp.pdb◦ myapp.PrecompiledViews.dll (if precompiling Razor Views)◦ myapp.PrecompiledViews.pdb (if precompiling Razor Views)◦ myapp.runtimeconfig.json◦ web.config (if included in publishOptions)
Self-contained Deployment	<ul style="list-style-type: none">• publish*<ul style="list-style-type: none">◦ logs* (if included in publishOptions)◦ refs*◦ Views* (if included in publishOptions)◦ wwwroot* (if included in publishOptions)◦ .dll files◦ myapp.deps.json◦ myapp.exe◦ myapp.pdb◦ myapp.PrecompiledViews.dll (if precompiling Razor Views)◦ myapp.PrecompiledViews.pdb (if precompiling Razor Views)◦ myapp.runtimeconfig.json◦ web.config (if included in publishOptions)

* Indicates a directory

The contents of the *publish* directory represents the *content root path*, also called the *application base path*, of the deployment. Whatever name is given to the *publish* directory in the deployment, its location serves as the server's physical path to the hosted application. The *wwwroot* directory, if present, only contains static assets. The *logs* directory may be included in the deployment by creating it in the project and adding the `<Target>` element shown below to your *.csproj* file or by physically creating the directory on the server.

```
<Target Name="CreateLogsFolder" AfterTargets="Publish">
  <MakeDir Directories="$(PublishDir)Logs"
    Condition="!Exists('$(PublishDir)Logs')" />
  <WriteLinesToFile File="$(PublishDir)Logs\.log"
    Lines="Generated file"
    Overwrite="True"
    Condition="!Exists('$(PublishDir)Logs\.log')" />
</Target>
```

The `<MakeDir>` element creates an empty *Logs* folder in the published output. The element uses the `PublishDir` property to determine the target location for creating the folder. Several deployment methods, such as Web Deploy, skip empty folders during deployment. The `<WriteLinesToFile>` element generates a file in the *Logs* folder, which guarantees deployment of the folder to the server. Note that folder creation may still fail if the worker process doesn't have write access to the target folder.

The deployment directory requires Read/Execute permissions, while the *Logs* directory requires Read/Write permissions. Additional directories where assets will be written require Read/Write permissions.

Common errors reference for Azure App Service and IIS with ASP.NET Core

1/12/2018 • 8 min to read • [Edit Online](#)

By [Luke Latham](#)

The following isn't a complete list of errors. If you encounter an error not listed here, [open a new issue](#) with detailed instructions to reproduce the error.

Installer unable to obtain VC++ Redistributable

- **Installer Exception:** 0x80072efd or 0x80072f76 - Unspecified error
- **Installer Log Exception†:** Error 0x80072efd or 0x80072f76: Failed to execute EXE package

†The log is located at C:\Users\
{USER}\AppData\Local\Temp\dd_DotNetCoreWinSvrHosting_{timestamp}.log.

Troubleshooting:

- If the system doesn't have Internet access while installing the server hosting bundle, this exception occurs when the installer is prevented from obtaining the *Microsoft Visual C++ 2015 Redistributable*. Obtain an installer from the [Microsoft Download Center](#). If the installer fails, the server may not receive the .NET Core runtime required to host a framework-dependent deployment (FDD). If hosting an FDD, confirm that the runtime is installed in Programs & Features. If needed, obtain a runtime installer from [.NET Downloads](#). After installing the runtime, restart the system or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

OS upgrade removed the 32-bit ASP.NET Core Module

- **Application Log:** The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

- Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any AppPool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See [Install the .NET Core Windows Server Hosting bundle](#). Select **Repair** when the installer is run.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{assembly}.exe|dll"', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{assembly}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the

app. For more information, see [Troubleshooting](#).

- Confirm that the `<PlatformTarget>` in the `.csproj` doesn't conflict with the RID. For example, don't specify a `<PlatformTarget>` of `x86` and publish with an RID of `win10-x64`, either by using `dotnet publish -c Release -r win10-x64` or by setting the `<RuntimeIdentifiers>` in the `.csproj` to `win10-x64`. The project publishes without warning or error but fails with the above logged exceptions on the system.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED
- **Application Log:** No entry
- **ASP.NET Core Module Log:** Log file not created

Troubleshooting:

- Confirm the correct URI endpoint for the app is being used. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

- **OS Exception:** The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

- Confirm that the proper role and features are enabled. See [IIS Configuration](#).

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied --OR-- 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module Log:** Log file not created

Troubleshooting:

- Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid.
- **Application Log:** No entry
- **ASP.NET Core Module Log:** Log file not created

Troubleshooting:

- Confirm that the proper role is enabled. See [IIS Configuration](#).

- Check **Programs & Features** and confirm that the **Microsoft ASP.NET Core Module** has been installed. If the **Microsoft ASP.NET Core Module** isn't present in the list of installed programs, install the module. See [Install the .NET Core Windows Server Hosting bundle](#).
- Make sure that the **Application Pool > Process Model > Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.

Incorrect processPath, missing PATH variable, hosting bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:\{PATH}' failed to start process with commandline '"{assembly}.exe" ', ErrorCode = '0x80070002 : 0.
- **ASP.NET Core Module Log:** Log file created but empty

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see [Troubleshooting](#).
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's *dotnet* for a framework-dependent deployment (FDD) or *.{assembly}.exe* for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that `*C:\Program Files\dotnet*` exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the Application Pool. Confirm that the AppPool user identity has access to the `C:\Program Files\dotnet` directory. Confirm that there are no deny rules configured for the AppPool user identity on the `C:\Program Files\dotnet` and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Windows Server Hosting bundle installer** on the system. See [Install the .NET Core Windows Server Hosting bundle](#). If attempting to install the .NET Core runtime on a system without an Internet connection, obtain the runtime from [.NET Downloads](#) and run the hosting bundle installer to install the ASP.NET Core Module. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed and the *Microsoft Visual C++ 2015 Redistributable (x64)* isn't installed on the system. Obtain an installer from the [Microsoft Download Center](#).

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:\{PATH}' failed to start process with commandline '"dotnet" .{assembly}.dll', ErrorCode = '0x80004005 : 80008081.
- **ASP.NET Core Module Log:** The application to execute does not exist: 'PATH{assembly}.dll'

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see [Troubleshooting](#).
- Examine the `arguments` attribute on the `<aspNetCore>` element in `web.config` to confirm that it is either (a) `{assembly}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="arg1, arg2, ..."`) for a self-contained deployment (SCD).

Missing .NET Framework version

- **Browser:** 502.3 Bad Gateway - There was a connection error while trying to route the request.
- **Application Log:** ErrorCode = Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:\{PATH}' failed to start process with commandline '"dotnet" {assembly}.dll', ErrorCode = '0x80004005 : 80008081.
- **ASP.NET Core Module Log:** Missing method, file, or assembly exception. The method, file, or assembly specified in the exception is a .NET Framework method, file, or assembly.

Troubleshooting:

- Install the .NET Framework version missing from the system.
- For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system. If the project is upgraded from 1.1 to 2.0, deployed to the hosting system, and this exception results, ensure that the 2.0 framework is on the hosting system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module Log:** Log file not created

Troubleshooting

- Confirm that the Application Pool isn't in the *Stopped* state.

IIS Integration middleware not implemented

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:\{PATH}' created process with commandline '"C:\{PATH}{assembly}.exe|dll"' but either crashed or did not reponse or did not listen on the given port '{PORT}', ErrorCode = '0x800705b4'
- **ASP.NET Core Module Log:** Log file created and shows normal operation.

Troubleshooting

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see [Troubleshooting](#).
- Confirm that either:
 - The IIS Integration middleware is referenced by calling the `UseIISIntegration` method on the app's `WebHostBuilder` (ASP.NET Core 1.x)
 - The app uses the `CreateDefaultBuilder` method (ASP.NET Core 2.x).

See [Hosting in ASP.NET Core](#) for details.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module Log:** Log file created and shows normal operation for the root app. Log file not created for the sub-app.

Troubleshooting

- Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section.

Application configuration general issue

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:\{PATH}' created process with commandline '"C:\{PATH}{assembly}.{exe|dll}" ' but either crashed or did not reponse or did not listen on the given port '{PORT}', ErrorCode = '0x800705b4'
- **ASP.NET Core Module Log:** Log file created but empty

Troubleshooting

- This general exception indicates that the process failed to start, most likely due to an app configuration issue. Referring to [Directory Structure](#), confirm that the app's deployed files and folders are appropriate and that the app's configuration files are present and contain the correct settings for the app and environment. For more information, see [Troubleshooting](#).

Add app features from an external assembly using IHostingStartup in ASP.NET Core

1/10/2018 • 5 min to read • [Edit Online](#)

By [Luke Latham](#)

An `IHostingStartup` implementation allows adding features to an app at startup from outside of the app's `Startup` class. For example, an external tooling library can use an `IHostingStartup` implementation to provide additional configuration providers or services to an app. `IHostingStartup` is available in ASP.NET Core 2.0 and later.

[View or download sample code \(how to download\)](#)

Discover loaded hosting startup assemblies

To discover hosting startup assemblies loaded by the app or by libraries, enable logging and check the application logs. Errors that occur when loading assemblies are logged. Loaded hosting startup assemblies are logged at the Debug level, and all errors are logged.

The sample app reads the the `HostingStartupAssembliesKey` into a `string` array and displays the result in the app's Index page:

```
public class IndexModel : PageModel
{
    private readonly IConfiguration _config;

    public IndexModel(IConfiguration config)
    {
        _config = config;
    }

    public string[] LoadedHostingStartupAssemblies { get; private set; }

    public void OnGet()
    {
        LoadedHostingStartupAssemblies =
            _config[WebHostDefaults.HostingStartupAssembliesKey]
                .Split(new[] { ';' }, StringSplitOptions.RemoveEmptyEntries) ?? new string[0];
    }
}
```

Disable automatic loading of hosting startup assemblies

There are two ways to disable the automatic loading of hosting startup assemblies:

- Set the [Prevent Hosting Startup](#) host configuration setting.
- Set the `ASPNETCORE_preventHostingStartup` environment variable.

When either the host setting or the environment variable is set to `true` or `1`, hosting startup assemblies aren't automatically loaded. If both are set, the host setting controls the behavior.

Disabling hosting startup assemblies using the host setting or environment variable disables them globally and may disable several features of an app. It isn't currently possible to selectively disable a hosting startup assembly added by a library unless the library offers its own configuration option. A future release will offer the ability to

selectively disable hosting startup assemblies (see [GitHub issue aspnet/Hosting #1243](#)).

Implement IHostingStartup features

Create the assembly

An `IHostingStartup` feature is deployed as an assembly based on a console app without an entry point. The assembly references the `Microsoft.AspNetCore.Hosting.Abstractions` package:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Hosting.Abstractions"
      Version="2.0.0" />
  </ItemGroup>

</Project>
```

A `HostingStartup` attribute identifies a class as an implementation of `IHostingStartup` for loading and execution when building the `IWebHost`. In the following example, the namespace is `StartupFeature`, and the class is `StartupFeatureHostingStartup`:

```
[assembly: HostingStartup(typeof(StartupFeature.StartupFeatureHostingStartup))]
```

A class implements `IHostingStartup`. The class's `Configure` method uses an `IWebHostBuilder` to add features to an app:

```
namespace StartupFeature
{
  public class StartupFeatureHostingStartup : IHostingStartup
  {
    public void Configure(IWebHostBuilder builder)
    {
      // Use the IWebHostBuilder to add app features.
    }
  }
}
```

When building an `IHostingStartup` project, the dependencies file (`*.deps.json`) sets the `runtime` location of the assembly to the `bin` folder:

```
"targets": {
  ".NETCoreApp,Version=v2.0": {
    "StartupFeature/1.0.0": {
      "dependencies": {
        "Microsoft.AspNetCore.Hosting.Abstractions": "2.0.0"
      },
      "runtime": {
        "StartupFeature.dll": {}
      }
    }
  }
}
```

Only part of the file is shown. The assembly name in the example is `StartupFeature`.

Update the dependencies file

The runtime location is specified in the `*.deps.json` file. To activate the feature, the `runtime` element must specify the location of the feature's runtime assembly. Prefix the `runtime` location with `lib/netcoreapp2.0/`:

```
"targets": {
  ".NETCoreApp,Version=v2.0": {
    "StartupFeature/1.0.0": {
      "dependencies": {
        "Microsoft.AspNetCore.Hosting.Abstractions": "2.0.0"
      },
      "runtime": {
        "lib/netcoreapp2.0/StartupFeature.dll": {}
      }
    }
  }
}
```

In the sample app, modification of the `*.deps.json` file is performed by a [PowerShell](#) script. The PowerShell script is automatically triggered by a build target in the project file.

Feature activation

Place the assembly file

The `IHostingStartup` implementation's assembly file must be *bin*-deployed in the app or placed in the [runtime store](#):

For per-user use, place the assembly in the user profile's runtime store at:

```
<DRIVE>\Users\<USER>\.dotnet\store\x64\netcoreapp2.0\<FEATURE_ASSEMBLY_NAME>\
<FEATURE_VERSION>\lib\netcoreapp2.0\
```

For global use, place the assembly in the .NET Core installation's runtime store:

```
<DRIVE>\Program Files\dotnet\store\x64\netcoreapp2.0\<FEATURE_ASSEMBLY_NAME>\
<FEATURE_VERSION>\lib\netcoreapp2.0\
```

When deploying the assembly to the runtime store, the symbols file may be deployed as well but isn't required for the feature to work.

Place the dependencies file

The implementation's `*.deps.json` file must be in an accessible location.

For per-user use, place the file in the `additionalDeps` folder of the user profile's `.dotnet` settings:

```
<DRIVE>\Users\<USER>\.dotnet\x64\additionalDeps\<FEATURE_ASSEMBLY_NAME>\shared\Microsoft.NETCore.App\2.0.0\
```

For global use, place the file in the `additionalDeps` folder of the .NET Core installation:

```
<DRIVE>\Program Files\dotnet\additionalDeps\<FEATURE_ASSEMBLY_NAME>\shared\Microsoft.NETCore.App\2.0.0\
```

Note the version, `2.0.0`, reflects the version of the shared runtime that the target app uses. The shared runtime is shown in the `*.runtimeconfig.json` file. In the sample app, the shared runtime is specified in the

HostingStartupSample.runtimeconfig.json file.

Set environment variables

Set the following environment variables in the context of the app that uses the feature.

ASPNETCORE_HOSTINGSTARTUPASSEMBLIES

Only hosting startup assemblies are scanned for the `HostingStartupAttribute`. The assembly name of the implementation is provided in this environment variable. The sample app sets this value to `StartupDiagnostics`.

The value can also be set using the [Hosting Startup Assemblies](#) host configuration setting.

DOTNET_ADDITIONAL_DEPS

The location of the implementation's **.deps.json* file.

If the file is placed in the user profile's *.dotnet* folder for per-user use:

```
<DRIVE>\Users\<USER>\.dotnet\x64\additionalDeps\
```

If the file is placed in the .NET Core installation for global use, provide the full path to the file:

```
<DRIVE>\Program Files\dotnet\additionalDeps\<FEATURE_ASSEMBLY_NAME>\shared\Microsoft.NETCore.App\2.0.0\  
<FEATURE_ASSEMBLY_NAME>.deps.json
```

The sample app sets this value to:

```
%UserProfile%\dotnet\x64\additionalDeps\StartupDiagnostics\
```

For examples of how to set environment variables for various operating systems, see [Working with multiple environments](#).

Sample app

The [sample app \(how to download\)](#) uses `IHostingStartup` to create a diagnostics tool. The tool adds two middlewares to the app at startup that provide diagnostic information:

- Registered services
- Address: scheme, host, path base, path, query string
- Connection: remote IP, remote port, local IP, local port, client certificate
- Request headers
- Environment variables

To run the sample:

1. The Startup Diagnostic project uses [PowerShell](#) to modify its *StartupDiagnostics.deps.json* file. PowerShell is installed by default on Windows OS starting with Windows 7 SP1 and Windows Server 2008 R2 SP1. To obtain PowerShell on other platforms, see [Installing Windows PowerShell](#).
2. Build the Startup Diagnostic project. A build target in the project file:
 - Moves the assembly and symbols files to the user profile's runtime store.
 - Triggers the PowerShell script to modify the *StartupDiagnostics.deps.json* file.
 - Moves the *StartupDiagnostics.deps.json* file to the user profile's `additionalDeps` folder.
3. Set the environment variables:
 - `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` : `StartupDiagnostics`

- `DOTNET_ADDITIONAL_DEPS` : `%UserProfile%\dotnet\x64\additionalDeps\StartupDiagnostics\`

4. Run the sample app.

5. Request the `/services` endpoint to see the app's registered services. Request the `/diag` endpoint to see the diagnostic information.

ASP.NET Core Security Overview

1/10/2018 • 2 min to read • [Edit Online](#)

ASP.NET Core enables developers to easily configure and manage security for their apps. ASP.NET Core contains features for managing authentication, authorization, data protection, SSL enforcement, app secrets, anti-request forgery protection, and CORS management. These security features allow you to build robust yet secure ASP.NET Core apps.

ASP.NET Core security features

ASP.NET Core provides many tools and libraries to secure your apps including built-in Identity providers but you can use 3rd party identity services such as Facebook, Twitter, or LinkedIn. With ASP.NET Core, you can easily manage app secrets, which are a way to store and use confidential information without having to expose it in the code.

Authentication vs. Authorization

Authentication is a process in which a user provides credentials that are then compared to those stored in an operating system, database, app or resource. If they match, users authenticate successfully, and can then perform actions that they are authorized for, during an authorization process. The authorization refers to the process that determines what a user is allowed to do.

Another way to think of authentication is to consider it as a way to enter a space, such as a server, database, app or resource, while authorization is which actions the user can perform to which objects inside that space (server, database, or app).

Common Vulnerabilities in software

ASP.NET Core and EF contain features that help you secure your apps and prevent security breaches. The following list of links takes you to documentation detailing techniques to avoid the most common security vulnerabilities in web apps:

- [Cross-site scripting attacks](#)
- [SQL injection attacks](#)
- [Cross-Site Request Forgery \(CSRF\)](#)
- [Open redirect attacks](#)

There are more vulnerabilities that you should be aware of. For more information, see the section in this document on *ASP.NET Security Documentation*.

ASP.NET Security Documentation

- [Authentication](#)
 - [Introduction to Identity](#)
 - [Enable authentication using Facebook, Google, and other external providers](#)
 - [Configure Windows Authentication](#)
 - [Account confirmation and password recovery](#)
 - [Two-factor authentication with SMS](#)
 - [Use cookie authentication without Identity](#)

- Azure Active Directory
 - Integrate Azure AD into an ASP.NET Core web app
 - Call an ASP.NET Core Web API from a WPF app using Azure AD
 - Call a Web API in an ASP.NET Core web app using Azure AD
 - An ASP.NET Core web app with Azure AD B2C
- Secure ASP.NET Core apps with IdentityServer4
- Authorization
 - Introduction
 - Create an app with user data protected by authorization
 - Simple authorization
 - Role-based authorization
 - Claims-based authorization
 - Policy-based authorization
 - Dependency injection in requirement handlers
 - Resource-based authorization
 - View-based authorization
 - Limit identity by scheme
- Data protection
 - Introduction to data protection
 - Get started with the Data Protection APIs
 - Consumer APIs
 - Consumer APIs Overview
 - Purpose strings
 - Purpose hierarchy and multi-tenancy
 - Password hashing
 - Limit the lifetime of protected payloads
 - Unprotect payloads whose keys have been revoked
 - Configuration
 - Configure data protection
 - Default settings
 - Machine-wide policy
 - Non DI-aware scenarios
 - Extensibility APIs
 - Core cryptography extensibility
 - Key management extensibility
 - Miscellaneous APIs
 - Implementation
 - Authenticated encryption details
 - Subkey derivation and authenticated encryption
 - Context headers
 - Key management
 - Key storage providers
 - Key encryption at rest
 - Key immutability and changing settings
 - Key storage format
 - Ephemeral data protection providers

- Compatibility
 - Share cookies between apps
 - Replace in ASP.NET
- Create an app with user data protected by authorization
- Safe storage of app secrets during development
- Azure Key Vault configuration provider
- Enforce SSL
- Anti-Request Forgery
- Prevent open redirect attacks
- Prevent Cross-Site Scripting
- Enable Cross-Origin Requests (CORS)

Authentication

1/10/2018 • 1 min to read • [Edit Online](#)

- [Community OSS authentication options](#)
- [Introduction to Identity](#)
- [Enable authentication using Facebook, Google, and other external providers](#)
- [Enable QR code generation in Identity](#)
- [Configure Windows Authentication](#)
- [Account confirmation and password recovery](#)
- [Two-factor authentication with SMS](#)
- [Use cookie authentication without Identity](#)
- [Azure Active Directory](#)
 - [Integrate Azure AD into an ASP.NET Core web app](#)
 - [Call an ASP.NET Core Web API from a WPF app using Azure AD](#)
 - [Call a Web API in an ASP.NET Core web app using Azure AD](#)
- [Secure ASP.NET Core apps with IdentityServer4](#)
- [Secure ASP.NET Core apps with Azure App Service Authentication \(Easy Auth\)](#)
- [Articles based on projects created with individual user accounts](#)

Community OSS authentication options

1/4/2018 • 1 min to read • [Edit Online](#)

This page contains community-provided, open source authentication options for ASP.NET Core. This page will be periodically updated as new providers become available.

OSS Authentication Providers

The list below is sorted alphabetically.

NAME	DESCRIPTION
AspNet.Security.OpenIdConnect.Server (ASOS)	Low-level/protocol-first OpenID Connect server framework for ASP.NET Core and OWIN/Katana
IdentityServer4	OpenID Connect and OAuth 2.0 framework for ASP.NET Core - officially certified by the OpenID Foundation and under governance of the .NET Foundation
OpenIddict	Easy-to-use OpenID Connect server for ASP.NET Core
Cierge	Passwordless, drop-in OpenID Connect authentication

To get your provider added here [edit this page](#).

Introduction to Identity on ASP.NET Core

1/8/2018 • 9 min to read • [Edit Online](#)

By [Pranav Rastogi](#), [Rick Anderson](#), [Tom Dykstra](#), [Jon Galloway](#), [Erik Reitan](#), and [Steve Smith](#)

ASP.NET Core Identity is a membership system which allows you to add login functionality to your application. Users can create an account and login with a user name and password or they can use an external login provider such as Facebook, Google, Microsoft Account, Twitter or others.

You can configure ASP.NET Core Identity to use a SQL Server database to store user names, passwords, and profile data. Alternatively, you can use your own persistent store, for example, an Azure Table Storage. This document contains instructions for Visual Studio and for using the CLI.

[View or download the sample code.](#) ([How to download](#))

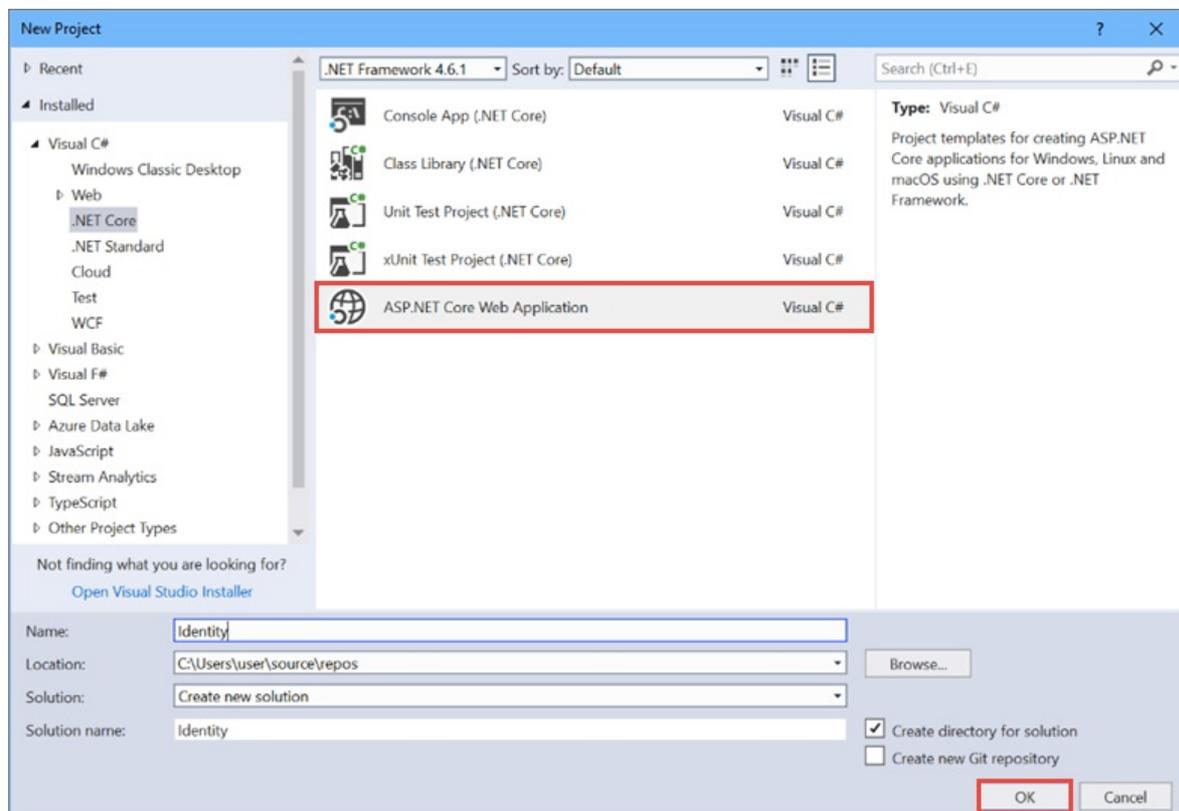
Overview of Identity

In this topic, you'll learn how to use ASP.NET Core Identity to add functionality to register, log in, and log out a user. For more detailed instructions about creating apps using ASP.NET Core Identity, see the Next Steps section at the end of this article.

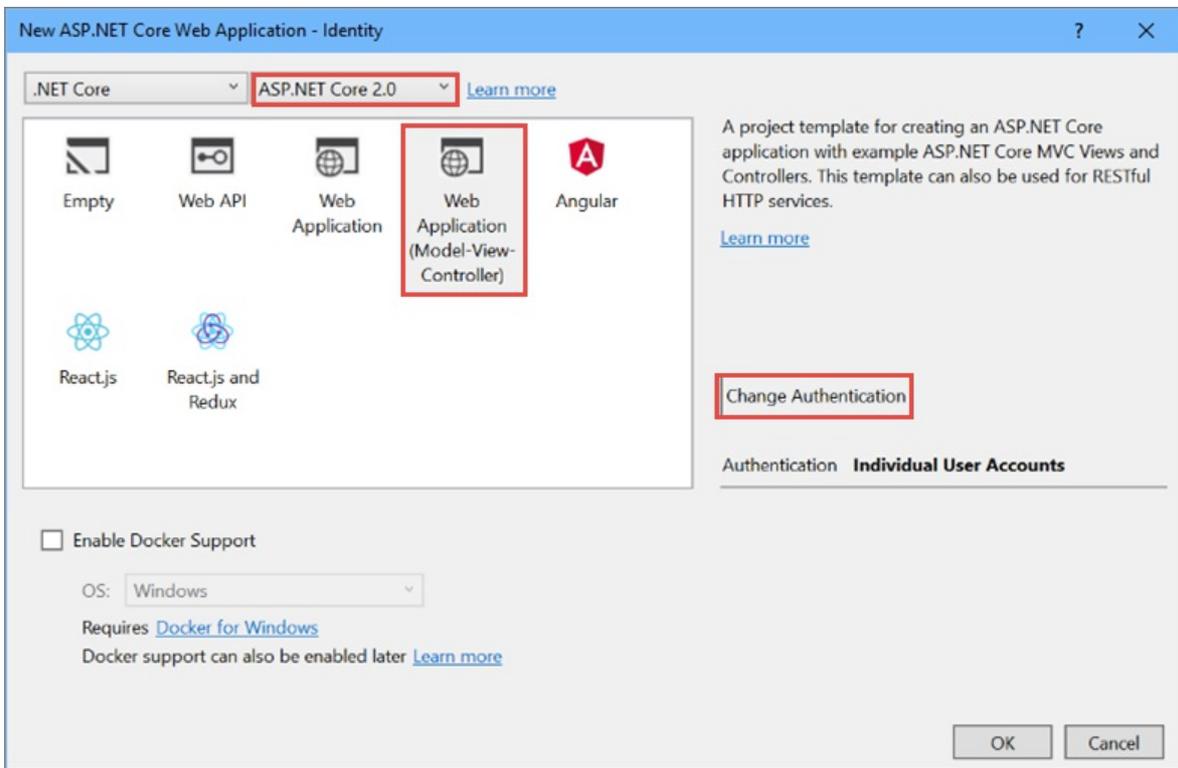
1. Create an ASP.NET Core Web Application project with Individual User Accounts.

- [Visual Studio](#)
- [.NET Core CLI](#)

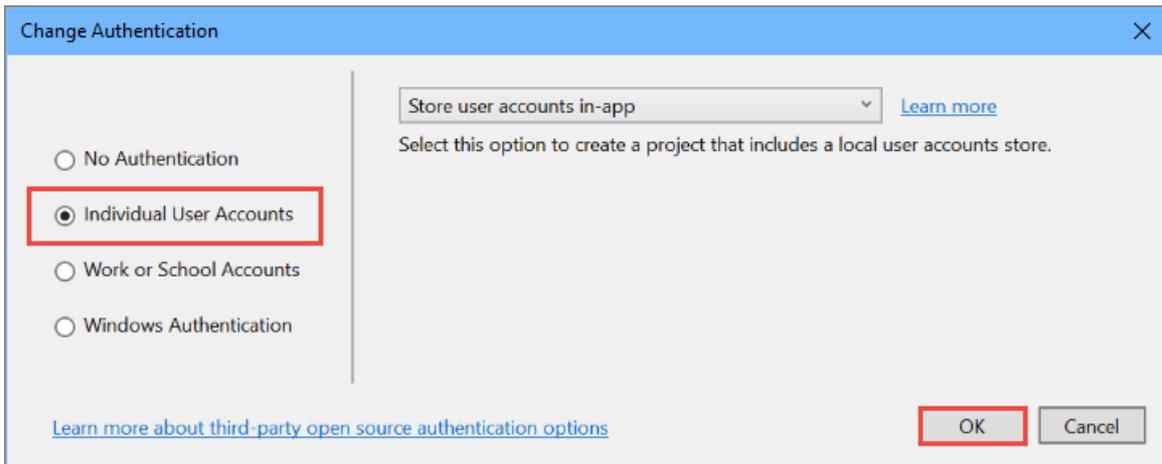
In Visual Studio, select **File** > **New** > **Project**. Select **ASP.NET Core Web Application** and click **OK**.



Select an ASP.NET Core **Web Application (Model-View-Controller)** for ASP.NET Core 2.x, then select **Change Authentication**.



A dialog appears offering authentication choices. Select **Individual User Accounts** and click **OK** to return to the previous dialog.



Selecting **Individual User Accounts** directs Visual Studio to create Models, ViewModels, Views, Controllers, and other assets required for authentication as part of the project template.

2. Configure Identity services and add middleware in `Startup`.

The Identity services are added to the application in the `ConfigureServices` method in the `Startup` class:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = false;
        options.Password.RequiredUniqueChars = 6;

        // Lockout settings
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.AllowedForNewUsers = true;

        // User settings
        options.User.RequireUniqueEmail = true;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.Cookie.Expiration = TimeSpan.FromDays(150);
        options.LoginPath = "/Account/Login"; // If the LoginPath is not set here, ASP.NET Core will
        default to /Account/Login
        options.LogoutPath = "/Account/Logout"; // If the LogoutPath is not set here, ASP.NET Core
        will default to /Account/Logout
        options.AccessDeniedPath = "/Account/AccessDenied"; // If the AccessDeniedPath is not set
        here, ASP.NET Core will default to /Account/AccessDenied
        options.SlidingExpiration = true;
    });

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();
}

```

These services are made available to the application through [dependency injection](#).

Identity is enabled for the application by calling `UseAuthentication` in the `Configure` method. `UseAuthentication` adds authentication [middleware](#) to the request pipeline.

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseAuthentication();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

For more information about the application start up process, see [Application Startup](#).

3. Create a user.

Launch the application and then click on the **Register** link.

If this is the first time you're performing this action, you may be required to run migrations. The application prompts you to **Apply Migrations**. Refresh the page if needed.

A database operation failed while processing the request.

SqlException: Cannot open database "aspnet-IdentityDemo-e5ecfa96-0c92-4175-a514-21049ad9d8b3" requested by the login. The login failed. Login failed for user 'OSIRIS\steve_000'.

Applying existing migrations for ApplicationDbContext may resolve this issue

There are migrations for ApplicationDbContext that have not been applied to the database

- 00000000000000000000_CreateIdentitySchema

Apply Migrations

In Visual Studio, you can use the Package Manager Console to apply pending migrations to the database:

PM> Update-Database

Alternatively, you can apply pending migrations from a command prompt at your project directory:

> dotnet ef database update

Alternately, you can test using ASP.NET Core Identity with your app without a persistent database by using an in-memory database. To use an in-memory database, add the

`Microsoft.EntityFrameworkCore.InMemory` package to your app and modify your app's call to `AddDbContext` in `ConfigureServices` as follows:

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseInMemoryDatabase(Guid.NewGuid().ToString()));
```

When the user clicks the **Register** link, the `Register` action is invoked on `AccountController`. The `Register` action creates the user by calling `CreateAsync` on the `_userManager` object (provided to `AccountController` by dependency injection):

```

//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // For more information on how to enable account confirmation and password reset please
            visit http://go.microsoft.com/fwlink/?LinkID=532713
            // Send an email with this link
            //var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            //var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code =
            code }, protocol: HttpContext.Request.Scheme);
            //await _emailSender.SendEmailAsync(model.Email, "Confirm your account",
            //    "Please confirm your account by clicking this link: <a href=\"" + callbackUrl +
            "\">link</a>");
            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "User created a new account with password.");
            return RedirectToAction(nameof(HomeController.Index), "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

If the user was created successfully, the user is logged in by the call to `_signInManager.SignInAsync`.

Note: See [account confirmation](#) for steps to prevent immediate login at registration.

4. Log in.

Users can sign in by clicking the **Log in** link at the top of the site, or they may be navigated to the Login page if they attempt to access a part of the site that requires authorization. When the user submits the form on the Login page, the `AccountController.Login` action is called.

The `Login` action calls `PasswordSignInAsync` on the `_signInManager` object (provided to `AccountController` by dependency injection).

```

//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email,
            model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe =
model.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

The base `Controller` class exposes a `User` property that you can access from controller methods. For instance, you can enumerate `User.Claims` and make authorization decisions. For more information, see [Authorization](#).

5. Log out.

Clicking the **Log out** link calls the `Logout` action.

```

//
// POST: /Account/Logout
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    _logger.LogInformation(4, "User logged out.");
    return RedirectToAction(nameof(HomeController.Index), "Home");
}

```

The preceding code above calls the `_signInManager.SignOutAsync` method. The `SignOutAsync` method clears the user's claims stored in a cookie.

6. Configuration.

Identity has some default behaviors that you can override in your application's startup class. You do not need to configure `IdentityOptions` if you are using the default behaviors.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = false;
        options.Password.RequiredUniqueChars = 6;

        // Lockout settings
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.AllowedForNewUsers = true;

        // User settings
        options.User.RequireUniqueEmail = true;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.Cookie.Expiration = TimeSpan.FromDays(150);
        options.LoginPath = "/Account/Login"; // If the LoginPath is not set here, ASP.NET Core will
        // default to /Account/Login
        options.LogoutPath = "/Account/Logout"; // If the LogoutPath is not set here, ASP.NET Core
        // will default to /Account/Logout
        options.AccessDeniedPath = "/Account/AccessDenied"; // If the AccessDeniedPath is not set
        // here, ASP.NET Core will default to /Account/AccessDenied
        options.SlidingExpiration = true;
    });

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();
}
```

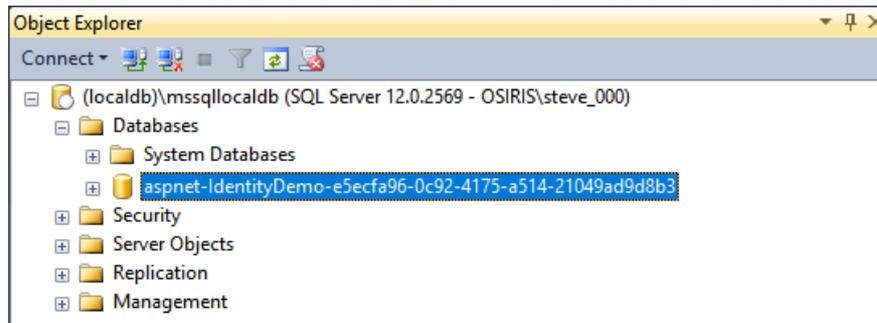
For more information about how to configure Identity, see [Configure Identity](#).

You also can configure the data type of the primary key, see [Configure Identity primary keys data type](#).

7. View the database.

If your app is using a SQL Server database (the default on Windows and for Visual Studio users), you can view the database the app created. You can use **SQL Server Management Studio**. Alternatively, from Visual Studio, select **View > SQL Server Object Explorer**. Connect to **(localdb)\MSSQLLocalDB**. The

database with a name matching **aspnet-*<name of your project>*-*<date string>*** is displayed.



Expand the database and its **Tables**, then right-click the **dbo.AspNetUsers** table and select **View Data**.

8. Verify Identity works

The default *ASP.NET Core Web Application* project template allows users to access any action in the application without having to login. To verify that ASP.NET Identity works, add an `[Authorize]` attribute to the `About` action of the `HomeController`.

```
[Authorize]
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";
    return View();
}
```

- [Visual Studio](#)
- [.NET Core CLI](#)

Run the project using **Ctrl + F5** and navigate to the **About** page. Only authenticated users may access the **About** page now, so ASP.NET redirects you to the login page to login or register.

Identity Components

The primary reference assembly for the Identity system is `Microsoft.AspNetCore.Identity`. This package contains the core set of interfaces for ASP.NET Core Identity, and is included by

```
Microsoft.AspNetCore.Identity.EntityFrameworkCore .
```

These dependencies are needed to use the Identity system in ASP.NET Core applications:

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore` - Contains the required types to use Identity with Entity Framework Core.
- `Microsoft.EntityFrameworkCore.SqlServer` - Entity Framework Core is Microsoft's recommended data access technology for relational databases like SQL Server. For testing, you can use `Microsoft.EntityFrameworkCore.InMemory`.
- `Microsoft.AspNetCore.Authentication.Cookies` - Middleware that enables an app to use cookie-based authentication.

Migrating to ASP.NET Core Identity

For additional information and guidance on migrating your existing Identity store see [Migrating Authentication and Identity](#).

Next Steps

- [Migrating Authentication and Identity](#)
- [Account Confirmation and Password Recovery](#)
- [Two-factor authentication with SMS](#)
- [Enabling authentication using Facebook, Google and other external providers](#)

Configure Identity

1/10/2018 • 3 min to read • [Edit Online](#)

ASP.NET Core Identity has common behaviors in applications such as password policy, lockout time, and cookie settings that you can override easily in your application's `Startup` class.

Passwords policy

By default, Identity requires that passwords contain an uppercase character, lowercase character, a digit, and a non-alphanumeric character. There are also some other restrictions. To simplify password restrictions, modify the

`ConfigureServices` method of the `Startup` class of your application.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

ASP.NET Core 2.0 added the `RequiredUniqueChars` property. Otherwise, the options are the same from ASP.NET Core 1.x.

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // Password settings
    options.Password.RequireDigit = true;
    options.Password.RequiredLength = 8;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequireLowercase = true;
    options.Password.RequiredUniqueChars = 2;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders();
```

`IdentityOptions.Password` has the following properties:

PROPERTY	DESCRIPTION	DEFAULT
<code>RequireDigit</code>	Requires a number between 0-9 in the password.	true
<code>RequiredLength</code>	The minimum length of the password.	6
<code>RequireNonAlphanumeric</code>	Requires a non-alphanumeric character in the password.	true
<code>RequireUppercase</code>	Requires an upper case character in the password.	true
<code>RequireLowercase</code>	Requires a lower case character in the password.	true
<code>RequiredUniqueChars</code>	Requires the number of distinct characters in the password.	1

User's lockout

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // Lockout settings
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders();
```

`IdentityOptions.Lockout` has the following properties:

PROPERTY	DESCRIPTION	DEFAULT
<code>DefaultLockoutTimeSpan</code>	The amount of time a user is locked out when a lockout occurs.	5 minutes
<code>MaxFailedAccessAttempts</code>	The number of failed access attempts until a user is locked out, if lockout is enabled.	5
<code>AllowedForNewUsers</code>	Determines if a new user can be locked out.	true

Sign in settings

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // Signin settings
    options.SignIn.RequireConfirmedEmail = true;
    options.SignIn.RequireConfirmedPhoneNumber = false;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders();
```

`IdentityOptions.SignIn` has the following properties:

PROPERTY	DESCRIPTION	DEFAULT
<code>RequireConfirmedEmail</code>	Requires a confirmed email to sign in.	false
<code>RequireConfirmedPhoneNumber</code>	Requires a confirmed phone number to sign in.	false

User validation settings

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // User settings
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders();
```

`IdentityOptions.User` has the following properties:

PROPERTY	DESCRIPTION	DEFAULT
<code>RequireUniqueEmail</code>	Requires each User to have a unique email.	false
<code>AllowedUserNameCharacters</code>	Allowed characters in the username.	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+

Application's cookie settings

Like the passwords policy, all the settings of the application's cookie can be changed in the `Startup` class.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Under `ConfigureServices` in the `Startup` class, you can configure the application's cookie.

```
services.ConfigureApplicationCookie(options =>
{
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Account/Login";
    options.LogoutPath = "/Account/Logout";
    options.AccessDeniedPath = "/Account/AccessDenied";
    options.SlidingExpiration = true;
    // Requires `using Microsoft.AspNetCore.Authentication.Cookies;`
    options.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;
});
```

`CookieAuthenticationOptions` has the following properties:

PROPERTY	DESCRIPTION	DEFAULT
<code>Cookie.Name</code>	The name of the cookie.	<code>.AspNetCore.Cookies</code> .
<code>Cookie.HttpOnly</code>	When true, the cookie is not accessible from client-side scripts.	true
<code>ExpireTimeSpan</code>	Controls how much time the authentication ticket stored in the cookie will remain valid from the point it is created.	14 days
<code>LoginPath</code>	When a user is unauthorized, they will be redirected to this path to login.	<code>/Account/Login</code>
<code>LogoutPath</code>	When a user is logged out, they will be redirected to this path.	<code>/Account/Logout</code>
<code>AccessDeniedPath</code>	When a user fails an authorization check, they will be redirected to this path.	

PROPERTY	DESCRIPTION	DEFAULT
<code>SlidingExpiration</code>	When true, a new cookie will be issued with a new expiration time when the current cookie is more than halfway through the expiration window.	<code>/Account/AccessDenied</code>
<code>ReturnUrlParameter</code>	Determines the name of the query string parameter which is appended by the middleware when a 401 Unauthorized status code is changed to a 302 redirect onto the login path.	<code>true</code>
<code>AuthenticationScheme</code>	This is only relevant for ASP.NET Core 1.x. The logical name for a particular authentication scheme.	
<code>AutomaticAuthenticate</code>	This flag is only relevant for ASP.NET Core 1.x. When true, cookie authentication should run on every request and attempt to validate and reconstruct any serialized principal it created.	

Configure Windows authentication in an ASP.NET Core app

1/10/2018 • 4 min to read • [Edit Online](#)

By [Steve Smith](#) and [Scott Addie](#)

Windows authentication can be configured for ASP.NET Core apps hosted with IIS, [HTTP.sys](#), or [WebListener](#).

What is Windows authentication?

Windows authentication relies on the operating system to authenticate users of ASP.NET Core apps. You can use Windows authentication when your server runs on a corporate network using Active Directory domain identities or other Windows accounts to identify users. Windows authentication is best suited to intranet environments in which users, client applications, and web servers belong to the same Windows domain.

[Learn more about Windows authentication and installing it for IIS.](#)

Enable Windows authentication in an ASP.NET Core app

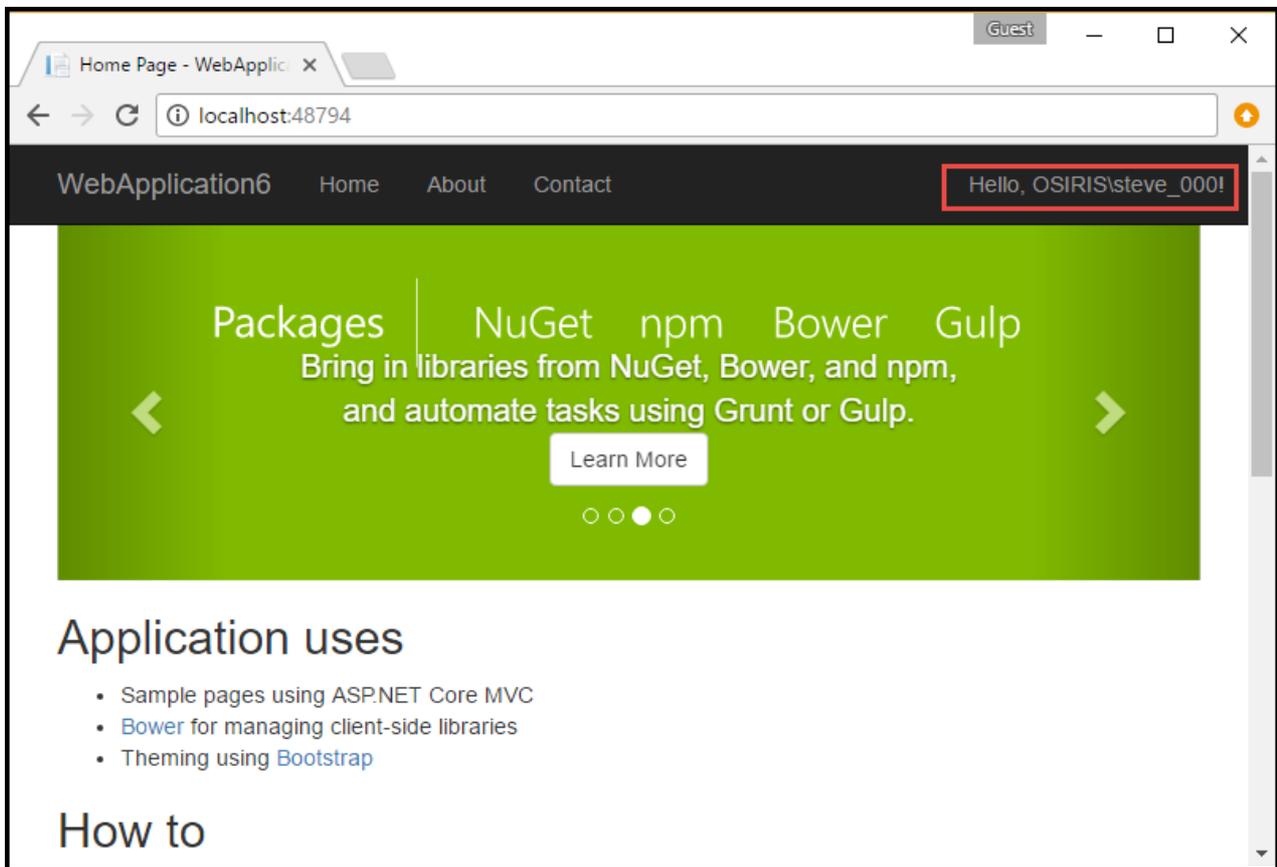
The Visual Studio Web Application template can be configured to support Windows authentication.

Use the Windows authentication app template

In Visual Studio:

1. Create a new ASP.NET Core Web Application.
2. Select Web Application from the list of templates.
3. Select the **Change Authentication** button and select **Windows Authentication**.

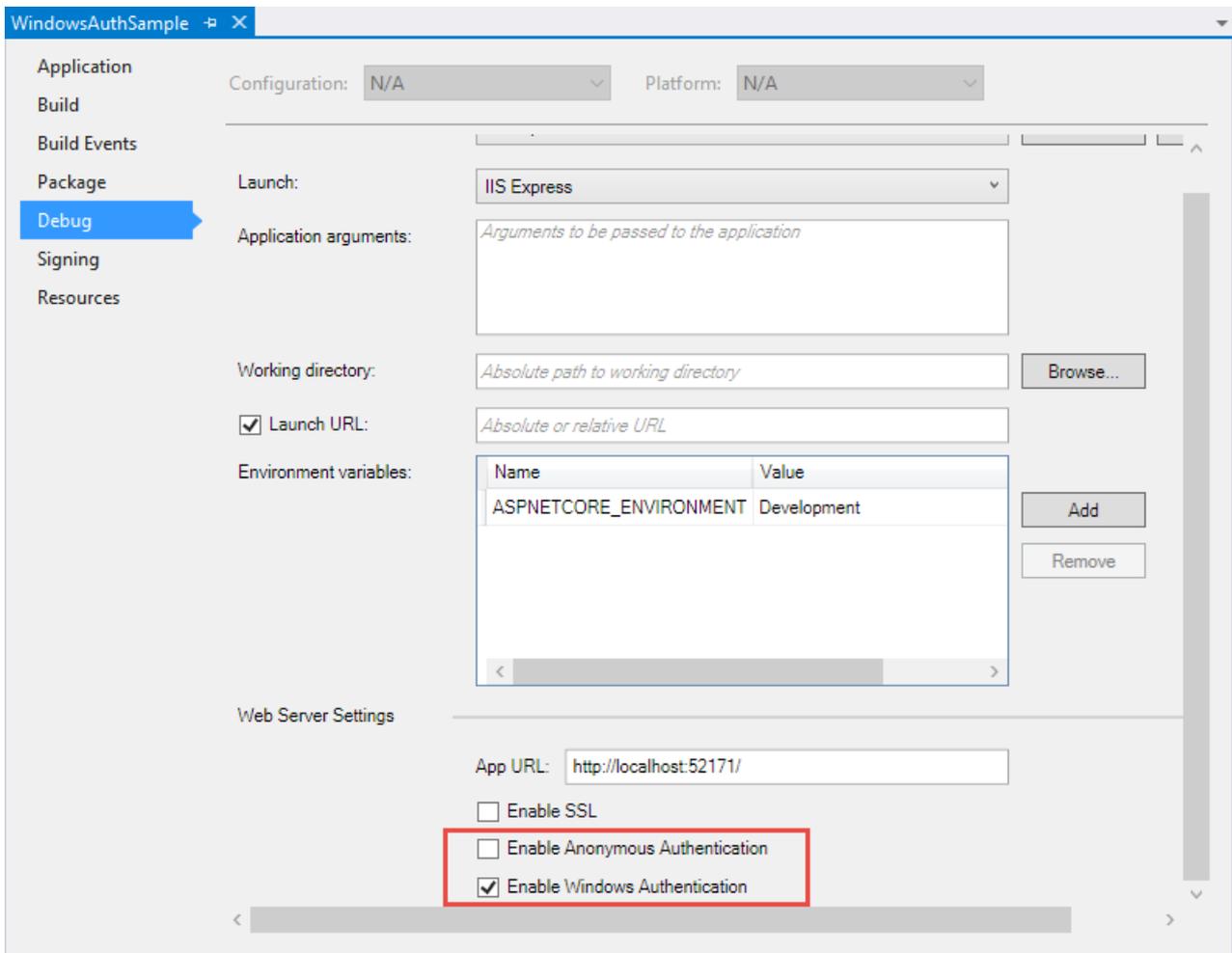
Run the app. The username appears in the top right of the app.



For development work using IIS Express, the template provides all the configuration necessary to use Windows authentication. The following section shows how to manually configure an ASP.NET Core app for Windows authentication.

Visual Studio settings for Windows and anonymous authentication

The Visual Studio project **Properties** page's **Debug** tab provides check boxes for Windows authentication and anonymous authentication.



Alternatively, these two properties can be configured in the *launchSettings.json* file:

```
{
  "iisSettings": {
    "windowsAuthentication": true,
    "anonymousAuthentication": false,
    "iisExpress": {
      "applicationUrl": "http://localhost:52171/",
      "sslPort": 0
    }
  } // additional options trimmed
}
```

Enable Windows authentication with IIS

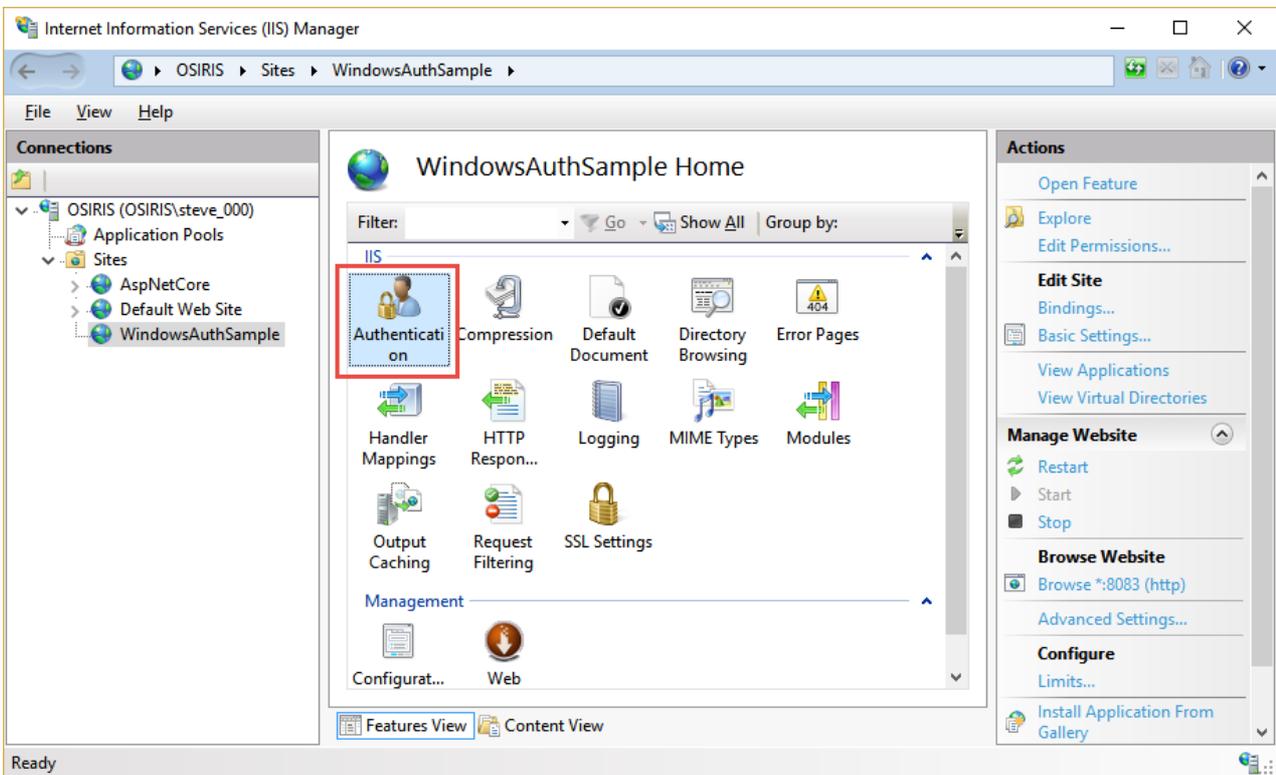
IIS uses the [ASP.NET Core Module](#) (ANCM) to host ASP.NET Core apps. The ANCM flows Windows authentication to IIS by default. Configuration of Windows authentication is done within IIS, not the application project. The following sections show how to use IIS Manager to configure an ASP.NET Core app to use Windows authentication.

Create a new IIS site

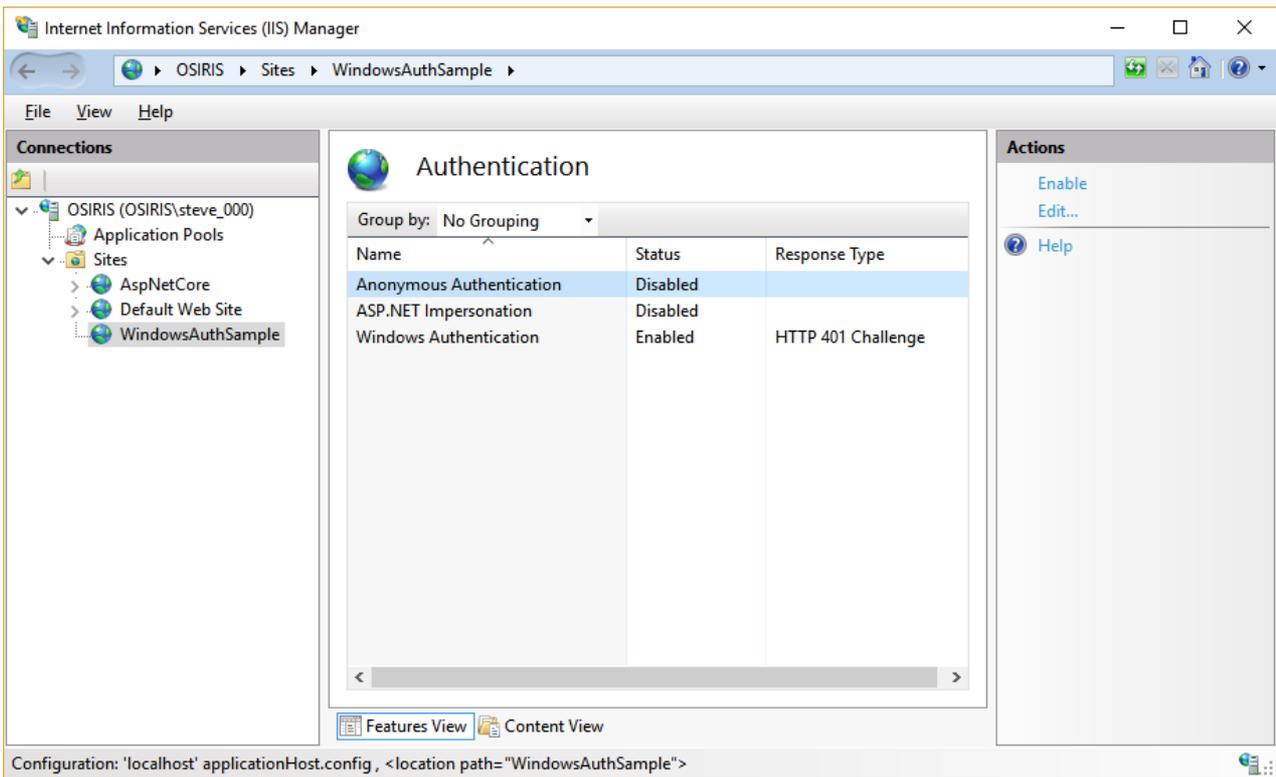
Specify a name and folder and allow it to create a new application pool.

Customize authentication

Open the Authentication menu for the site.

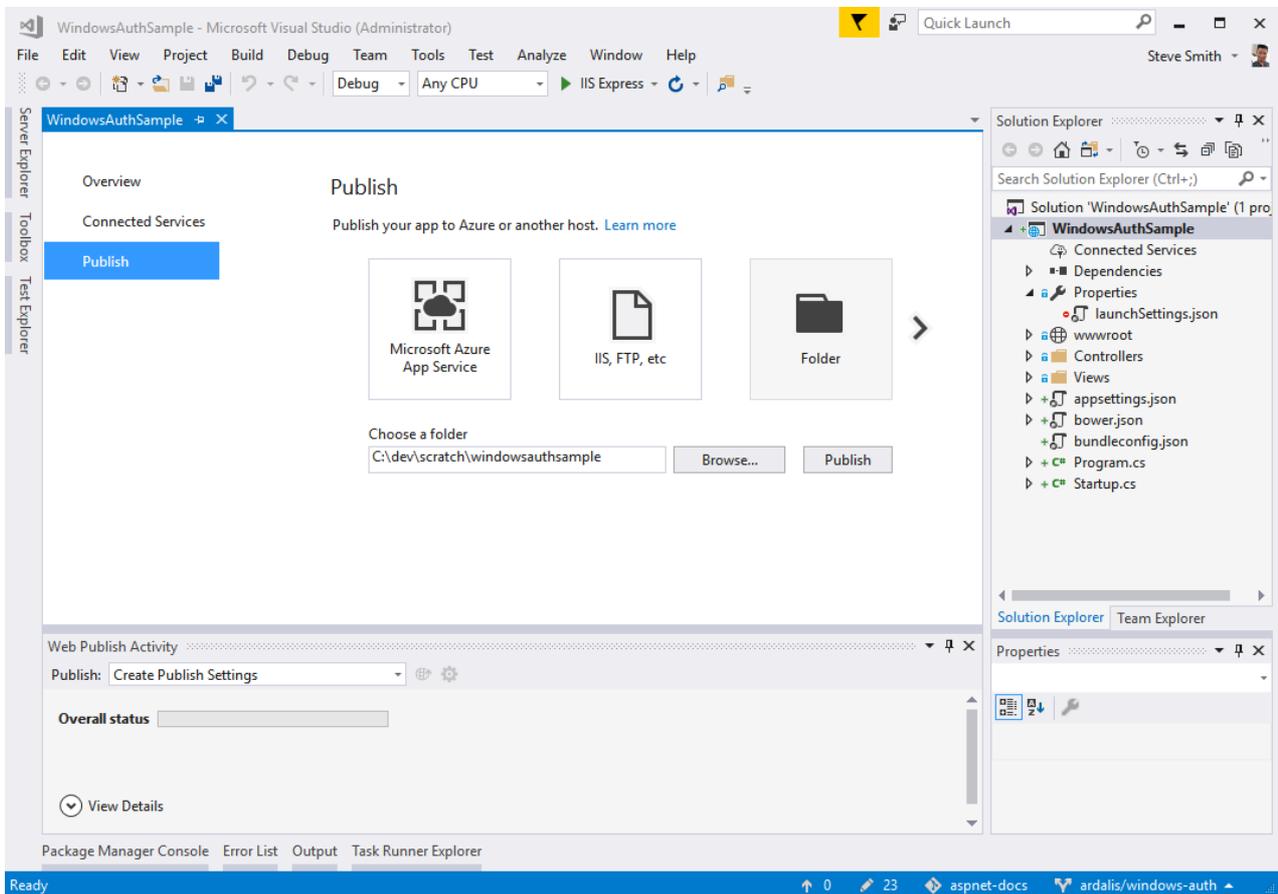


Disable Anonymous Authentication and enable Windows Authentication.



Publish your project to the IIS site folder

Using Visual Studio or the .NET Core CLI, publish the app to the destination folder.



Learn more about [publishing to IIS](#).

Launch the app to verify Windows authentication is working.

Enable Windows authentication with HTTP.sys or WebListener

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Although Kestrel doesn't support Windows authentication, you can use [HTTP.sys](#) to support self-hosted scenarios on Windows. The following example configures the app's web host to use HTTP.sys with Windows authentication:

```
public class Program
{
    public static void Main(string[] args) =>
        BuildWebHost(args).Run();

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .UseHttpSys(options =>
            {
                options.Authentication.Schemes =
                    AuthenticationSchemes.NTLM | AuthenticationSchemes.Negotiate;
                options.Authentication.AllowAnonymous = false;
            })
            .Build();
}
```

Work with Windows authentication

The configuration state of anonymous access determines the way in which the `[Authorize]` and `[AllowAnonymous]` attributes are used in the app. The following two sections explain how to handle the disallowed and allowed

configuration states of anonymous access.

Disallow anonymous access

When Windows authentication is enabled and anonymous access is disabled, the `[Authorize]` and `[AllowAnonymous]` attributes have no effect. If the IIS site (or HTTP.sys or WebListener server) is configured to disallow anonymous access, the request never reaches your app. For this reason, the `[AllowAnonymous]` attribute isn't applicable.

Allow anonymous access

When both Windows authentication and anonymous access are enabled, use the `[Authorize]` and `[AllowAnonymous]` attributes. The `[Authorize]` attribute allows you to secure pieces of the app which truly do require Windows authentication. The `[AllowAnonymous]` attribute overrides `[Authorize]` attribute usage within apps which allow anonymous access. See [Simple Authorization](#) for attribute usage details.

In ASP.NET Core 2.x, the `[Authorize]` attribute requires additional configuration in *Startup.cs* to challenge anonymous requests for Windows authentication. The recommended configuration varies slightly based on the web server being used.

IIS

If using IIS, add the following to the `ConfigureServices` method:

```
// IISDefaults requires the following import:  
// using Microsoft.AspNetCore.Server.IISIntegration;  
services.AddAuthentication(IISDefaults.AuthenticationScheme);
```

HTTP.sys

If using HTTP.sys, add the following to the `ConfigureServices` method:

```
// HttpSysDefaults requires the following import:  
// using Microsoft.AspNetCore.Server.HttpSys;  
services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
```

Impersonation

ASP.NET Core doesn't implement impersonation. Apps run with the application identity for all requests, using app pool or process identity. If you need to explicitly perform an action on behalf of a user, use

`WindowsIdentity.RunImpersonated`. Run a single action in this context and then close the context.

```
app.Run(async (context) =>
{
    try
    {
        var user = (WindowsIdentity)context.User.Identity;

        await context.Response
            .WriteAsync($"User: {user.Name}\tState: {user.ImpersonationLevel}\n");

        WindowsIdentity.RunImpersonated(user.AccessToken, () =>
        {
            var impersonatedUser = WindowsIdentity.GetCurrent();
            var message =
                $"User: {impersonatedUser.Name}\tState: {impersonatedUser.ImpersonationLevel}";

            var bytes = Encoding.UTF8.GetBytes(message);
            context.Response.Body.Write(bytes, 0, bytes.Length);
        });
    }
    catch (Exception e)
    {
        await context.Response.WriteAsync(e.ToString());
    }
});
```

Note that `RunImpersonated` doesn't support asynchronous operations and shouldn't be used for complex scenarios. For example, wrapping entire requests or middleware chains isn't supported or recommended.

Configure the ASP.NET Core Identity primary key data type

9/30/2017 • 2 min to read • [Edit Online](#)

ASP.NET Core Identity allows you to configure the data type used to represent a primary key. Identity uses the `string` data type by default. You can override this behavior.

Customize the primary key data type

1. Create a custom implementation of the `IdentityUser` class. It represents the type to be used for creating user objects. In the following example, the default `string` type is replaced with `Guid`.

```
namespace webapptemplate.Models
{
    // Add profile data for application users by adding properties to the ApplicationUser class
    public class ApplicationUser : IdentityUser<Guid>
    {
    }
}
```

2. Create a custom implementation of the `IdentityRole` class. It represents the type to be used for creating role objects. In the following example, the default `string` type is replaced with `Guid`.

```
namespace webapptemplate.Models
{
    public class ApplicationRole : IdentityRole<Guid>
    {
    }
}
```

3. Create a custom database context class. It inherits from the Entity Framework database context class used for Identity. The `TUser` and `TRole` arguments reference the custom user and role classes created in the previous step, respectively. The `Guid` data type is defined for the primary key.

```
namespace webapptemplate.Data
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            // Customize the ASP.NET Identity model and override the defaults if needed.
            // For example, you can rename the ASP.NET Identity table names and more.
            // Add your customizations after calling base.OnModelCreating(builder);
        }
    }
}
```

4. Register the custom database context class when adding the Identity service in the app's startup class.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The `AddEntityFrameworkStores` method doesn't accept a `TKey` argument as it did in ASP.NET Core 1.x. The primary key's data type is inferred by analyzing the `DbContext` object.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();
}
```

Test the changes

Upon completion of the configuration changes, the property representing the primary key reflects the new data type. The following example demonstrates accessing the property in an MVC controller.

```
[HttpGet]
[AllowAnonymous]
public async Task<Guid> GetCurrentUserId()
{
    ApplicationUser user = await _userManager.GetUserAsync(HttpContext.User);
    return user.Id; // No need to cast here because user.Id is already a Guid, and not a string
}
```

Custom storage providers for ASP.NET Core Identity

10/30/2017 • 9 min to read • [Edit Online](#)

By [Steve Smith](#)

ASP.NET Core Identity is an extensible system which enables you to create a custom storage provider and connect it to your app. This topic describes how to create a customized storage provider for ASP.NET Core Identity. It covers the important concepts for creating your own storage provider, but is not a step-by-step walkthrough.

[View or download sample from GitHub.](#)

Introduction

By default, the ASP.NET Core Identity system stores user information in a SQL Server database using Entity Framework Core. For many apps, this approach works well. However, you may prefer to use a different persistence mechanism or data schema. For example:

- You use [Azure Table Storage](#) or another data store.
- Your database tables have a different structure.
- You may wish to use a different data access approach, such as [Dapper](#).

In each of these cases, you can write a customized provider for your storage mechanism and plug that provider into your app.

ASP.NET Core Identity is included in project templates in Visual Studio with the "Individual User Accounts" option.

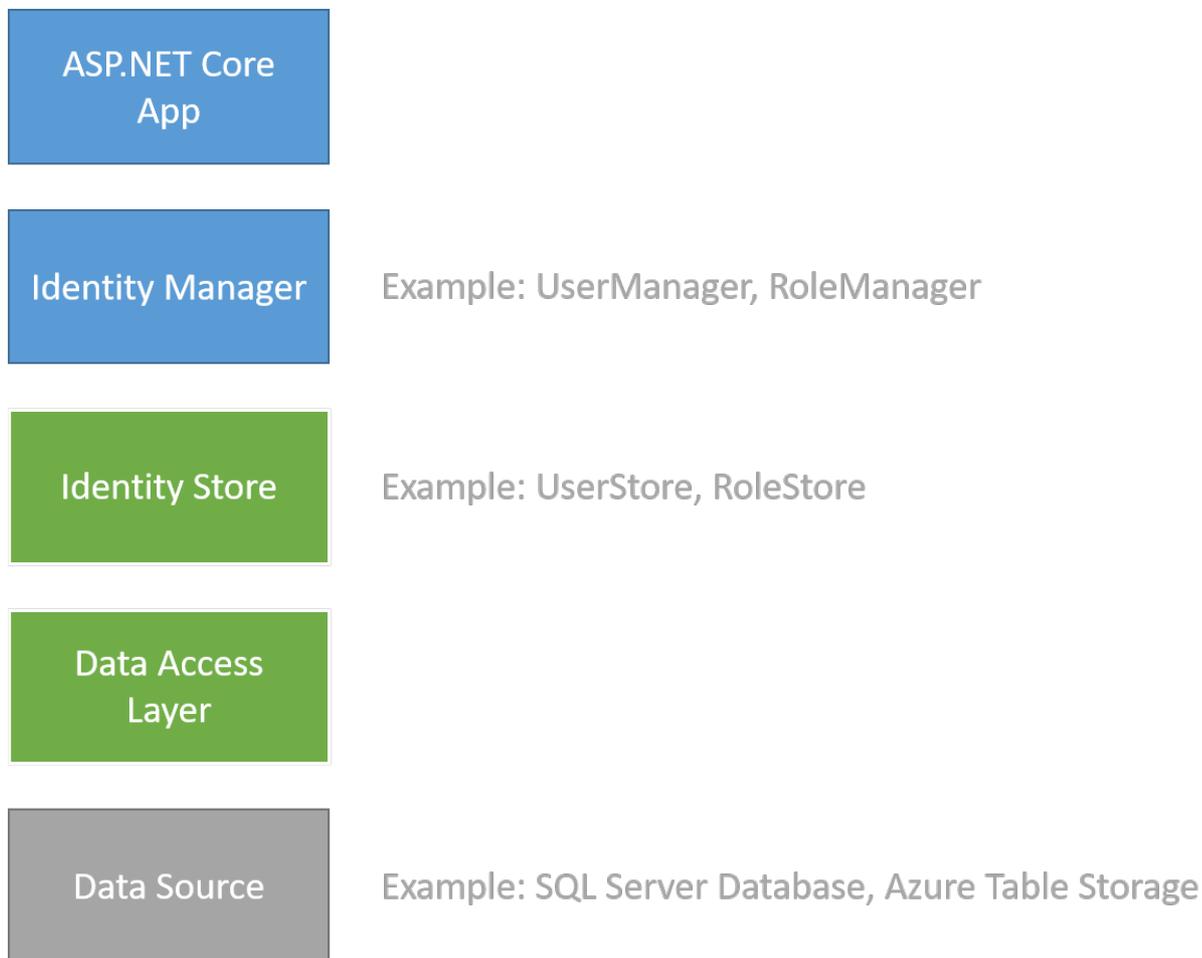
When using the .NET Core CLI, add `-au Individual`:

```
dotnet new mvc -au Individual
dotnet new webapi -au Individual
```

The ASP.NET Core Identity architecture

ASP.NET Core Identity consists of classes called managers and stores. *Managers* are high-level classes which an app developer uses to perform operations, such as creating an Identity user. *Stores* are lower-level classes that specify how entities, such as users and roles, are persisted. Stores follow the [repository pattern](#) and are closely coupled with the persistence mechanism. Managers are decoupled from stores, which means you can replace the persistence mechanism without changing your application code (except for configuration).

The following diagram shows how a web app interacts with the managers, while stores interact with the data access layer.



To create a custom storage provider, create the data source, the data access layer, and the store classes that interact with this data access layer (the green and grey boxes in the diagram above). You don't need to customize the managers or your app code that interacts with them (the blue boxes above).

When creating a new instance of `userManager` or `RoleManager` you provide the type of the user class and pass an instance of the store class as an argument. This approach enables you to plug your customized classes into ASP.NET Core.

[Reconfigure app to use new storage provider](#) shows how to instantiate `userManager` and `RoleManager` with a customized store.

ASP.NET Core Identity stores data types

[ASP.NET Core Identity](#) data types are detailed in the following sections:

Users

Registered users of your web site. The `IdentityUser` type may be extended or used as an example for your own custom type. You do not need to inherit from a particular type to implement your own custom identity storage solution.

User Claims

A set of statements (or `Claims`) about the user that represent the user's identity. Can enable greater expression of the user's identity than can be achieved through roles.

User Logins

Information about the external authentication provider (like Facebook or a Microsoft account) to use when logging in a user. [Example](#)

Roles

Authorization groups for your site. Includes the role Id and role name (like "Admin" or "Employee"). [Example](#)

The data access layer

This topic assumes you are familiar with the persistence mechanism that you are going to use and how to create entities for that mechanism. This topic does not provide details about how to create the repositories or data access classes; it provides some suggestions about design decisions when working with ASP.NET Core Identity.

You have a lot of freedom when designing the data access layer for a customized store provider. You only need to create persistence mechanisms for features that you intend to use in your app. For example, if you are not using roles in your app, you do not need to create storage for roles or user role associations. Your technology and existing infrastructure may require a structure that is very different from the default implementation of ASP.NET Core Identity. In your data access layer, you provide the logic to work with the structure of your storage implementation.

The data access layer provides the logic to save the data from ASP.NET Core Identity to a data source. The data access layer for your customized storage provider might include the following classes to store user and role information.

Context class

Encapsulates the information to connect to your persistence mechanism and execute queries. Several data classes require an instance of this class, typically provided through dependency injection. [Example](#).

User Storage

Stores and retrieves user information (such as user name and password hash). [Example](#)

Role Storage

Stores and retrieves role information (such as the role name). [Example](#)

UserClaims Storage

Stores and retrieves user claim information (such as the claim type and value). [Example](#)

UserLogins Storage

Stores and retrieves user login information (such as an external authentication provider). [Example](#)

UserRole Storage

Stores and retrieves which roles are assigned to which users. [Example](#)

TIP: Only implement the classes you intend to use in your app.

In the data access classes, provide code to perform data operations for your persistence mechanism. For example, within a custom provider, you might have the following code to create a new user in the *store* class:

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user,
    CancellationToken cancellationToken = default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(user));

    return await _usersTable.CreateAsync(user);
}
```

The implementation logic for creating the user is in the `_usersTable.CreateAsync` method, shown below.

Customize the user class

When implementing a storage provider, create a user class which is equivalent to the `IdentityUser` class.

At a minimum, your user class must include an `Id` and a `UserName` property.

The `IdentityUser` class defines the properties that the `UserManager` calls when performing requested operations. The default type of the `Id` property is a string, but you can inherit from `IdentityUser<TKey, TUserClaim, TUserRole, TUserLogin, TUserToken>` and specify a different type. The framework expects the storage implementation to handle data type conversions.

Customize the user store

Create a `UserStore` class that provides the methods for all data operations on the user. This class is equivalent to the `UserStore` class. In your `UserStore` class, implement `IUserStore<TUser>` and the optional interfaces required. You select which optional interfaces to implement based on the functionality provided in your app.

Optional interfaces

- `IUserRoleStore` <https://docs.microsoft.com/aspnet/core/api/microsoft.aspnetcore.identity.iuserrolestore-1>
- `IUserClaimStore` <https://docs.microsoft.com/aspnet/core/api/microsoft.aspnetcore.identity.iuserclaimstore-1>
- `IUserPasswordStore`
<https://docs.microsoft.com/aspnet/core/api/microsoft.aspnetcore.identity.iuserpasswordstore-1>
- `IUserSecurityStampStore`
- `IUserEmailStore`
- `IPhoneNumberStore`
- `IQueryableUserStore`
- `IUserLoginStore`
- `IUserTwoFactorStore`
- `IUserLockoutStore`

The optional interfaces inherit from `IUserStore`. You can see a partially implemented sample user store [here](#).

Within the `UserStore` class, you use the data access classes that you created to perform operations. These are passed in using dependency injection. For example, in the SQL Server with Dapper implementation, the `UserStore` class has the `CreateAsync` method which uses an instance of `DapperUsersTable` to insert a new record:

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user)
{
    string sql = "INSERT INTO dbo.CustomUser " +
        "VALUES (@id, @Email, @EmailConfirmed, @PasswordHash, @UserName)";

    int rows = await _connection.ExecuteAsync(sql, new { user.Id, user.Email, user.EmailConfirmed,
        user.PasswordHash, user.UserName });

    if(rows > 0)
    {
        return IdentityResult.Success;
    }
    return IdentityResult.Failed(new IdentityError { Description = $"Could not insert user {user.Email}." });
}
```

Interfaces to implement when customizing user store

- **`IUserStore`**
The `IUserStore<TUser>` interface is the only interface you must implement in the user store. It defines methods for creating, updating, deleting, and retrieving users.
- **`IUserClaimStore`**
The `IUserClaimStore<TUser>` interface defines the methods you implement to enable user claims. It contains methods for adding, removing and retrieving user claims.

- **IUserLoginStore**

The [IUserLoginStore<TUser>](#) defines the methods you implement to enable external authentication providers. It contains methods for adding, removing and retrieving user logins, and a method for retrieving a user based on the login information.

- **IUserRoleStore**

The [IUserRoleStore<TUser>](#) interface defines the methods you implement to map a user to a role. It contains methods to add, remove, and retrieve a user's roles, and a method to check if a user is assigned to a role.

- **IUserPasswordStore**

The [IUserPasswordStore<TUser>](#) interface defines the methods you implement to persist hashed passwords. It contains methods for getting and setting the hashed password, and a method that indicates whether the user has set a password.

- **IUserSecurityStampStore**

The [IUserSecurityStampStore<TUser>](#) interface defines the methods you implement to use a security stamp for indicating whether the user's account information has changed. This stamp is updated when a user changes the password, or adds or removes logins. It contains methods for getting and setting the security stamp.

- **IUserTwoFactorStore**

The [IUserTwoFactorStore<TUser>](#) interface defines the methods you implement to support two factor authentication. It contains methods for getting and setting whether two factor authentication is enabled for a user.

- **IUserPhoneNumberStore**

The [IUserPhoneNumberStore<TUser>](#) interface defines the methods you implement to store user phone numbers. It contains methods for getting and setting the phone number and whether the phone number is confirmed.

- **IUserEmailStore**

The [IUserEmailStore<TUser>](#) interface defines the methods you implement to store user email addresses. It contains methods for getting and setting the email address and whether the email is confirmed.

- **IUserLockoutStore**

The [IUserLockoutStore<TUser>](#) interface defines the methods you implement to store information about locking an account. It contains methods for tracking failed access attempts and lockouts.

- **IQueryableUserStore**

The [IQueryableUserStore<TUser>](#) interface defines the members implement to provide a queryable user store.

You implement only the interfaces that are needed in your app. For example:

```
public class UserStore : IUserStore<IdentityUser>,
    IUserClaimStore<IdentityUser>,
    IUserLoginStore<IdentityUser>,
    IUserRoleStore<IdentityUser>,
    IUserPasswordStore<IdentityUser>,
    IUserSecurityStampStore<IdentityUser>
{
    // interface implementations not shown
}
```

IdentityUserClaim, IdentityUserLogin, and IdentityUserRole

The `Microsoft.AspNet.Identity.EntityFramework` namespace contains implementations of the [IdentityUserClaim](#), [IdentityUserLogin](#), and [IdentityUserRole](#) classes. If you are using these features, you may want to create your own versions of these classes and define the properties for your app. However, sometimes it is more efficient to not load these entities into memory when performing basic operations (such as adding or removing a user's claim). Instead, the backend store classes can execute these operations directly on the data source. For example, the `UserStore.GetClaimsAsync` method can call the `userClaimTable.FindByUserId(user.Id)` method to execute a query on that table directly and return a list of claims.

Customize the role class

When implementing a role storage provider, you can create a custom role type. It need not implement a particular interface, but it must have an `Id` and typically it will have a `Name` property.

The following is an example role class:

```
using System;

namespace CustomIdentityProviderSample.CustomProvider
{
    public class ApplicationRole
    {
        public Guid Id { get; set; } = Guid.NewGuid();
        public string Name { get; set; }
    }
}
```

Customize the role store

You can create a `RoleStore` class that provides the methods for all data operations on roles. This class is equivalent to the `RoleStore` class. In the `RoleStore` class, you implement the `IRoleStore<TRole>` and optionally the `IQueryableRoleStore<TRole>` interface.

- **IRoleStore<TRole>**

The `IRoleStore` interface defines the methods to implement in the role store class. It contains methods for creating, updating, deleting and retrieving roles.

- **RoleStore<TRole>**

To customize `RoleStore`, create a class that implements the `IRoleStore` interface.

Reconfigure app to use new storage provider

Once you have implemented a storage provider, you configure your app to use it. If your app used the default provider, replace it with your custom provider.

1. Remove the `Microsoft.AspNetCore.EntityFrameworkCore.Identity` NuGet package.
2. If the storage provider resides in a separate project or package, add a reference to it.
3. Replace all references to `Microsoft.AspNetCore.EntityFrameworkCore.Identity` with a using statement for the namespace of your storage provider.
4. In the `ConfigureServices` method, change the `AddIdentity` method to use your custom types. You can create your own extension methods for this purpose. See [IdentityServiceCollectionExtensions](#) for an example.
5. If you are using Roles, update the `RoleManager` to use your `RoleStore` class.
6. Update the connection string and credentials to your app's configuration.

Example:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add identity types
    services.AddIdentity<ApplicationUser, ApplicationRole>()
        .AddDefaultTokenProviders();

    // Identity Services
    services.AddTransient<IUserStore<ApplicationUser>, CustomUserStore>();
    services.AddTransient<IRoleStore<ApplicationRole>, CustomRoleStore>();
    string connectionString = Configuration.GetConnectionString("DefaultConnection");
    services.AddTransient<SqlConnection>(e => new SqlConnection(connectionString));
    services.AddTransient<DapperUsersTable>();

    // additional configuration
}
```

References

- [Custom Storage Providers for ASP.NET Identity](#)
- [ASP.NET Core Identity](#) - This repository includes links to community maintained store providers.

Enabling authentication using Facebook, Google, and other external providers

12/13/2017 • 3 min to read • [Edit Online](#)

By [Valeriy Novytsky](#) and [Rick Anderson](#)

This tutorial demonstrates how to build an ASP.NET Core 2.x app that enables users to log in using OAuth 2.0 with credentials from external authentication providers.

[Facebook](#), [Twitter](#), [Google](#), and [Microsoft](#) providers are covered in the following sections. Other providers are available in third-party packages such as [AspNet.Security.OAuth.Providers](#) and [AspNet.Security.OpenId.Providers](#).

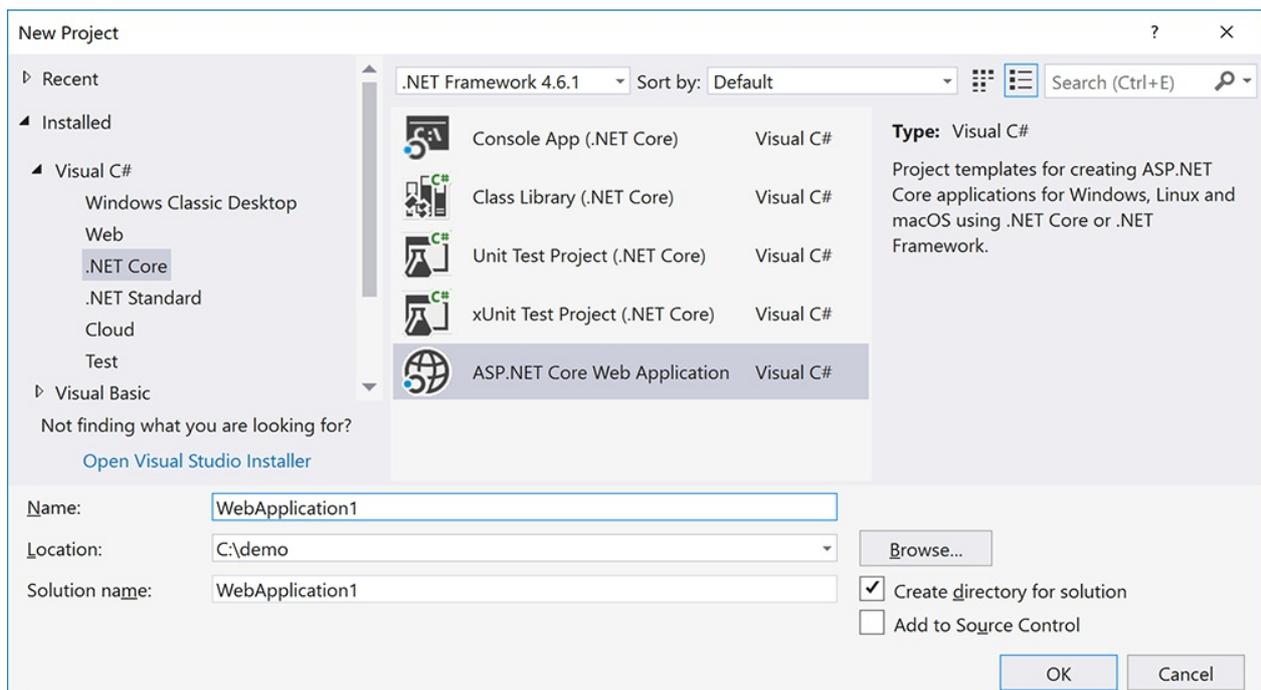


Enabling users to sign in with their existing credentials is convenient for the users and shifts many of the complexities of managing the sign-in process onto a third party. For examples of how social logins can drive traffic and customer conversions, see case studies by [Facebook](#) and [Twitter](#).

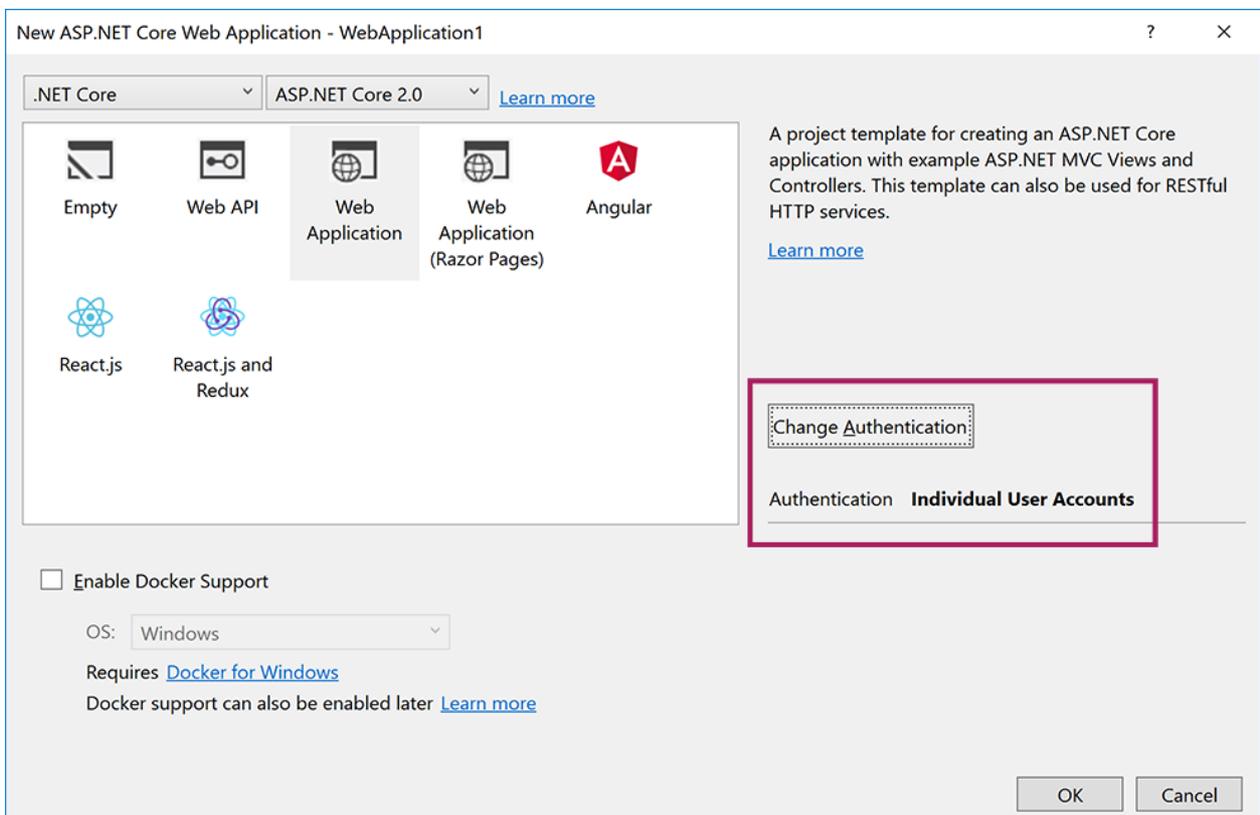
Note: Packages presented here abstract a great deal of complexity of the OAuth authentication flow, but understanding the details may become necessary when troubleshooting. Many resources are available; for example, see [Introduction to OAuth 2](#) or [Understanding OAuth 2](#). Some issues can be resolved by looking at the [ASP.NET Core source code for the provider packages](#).

Create a New ASP.NET Core Project

- In Visual Studio 2017, create a new project from the Start Page, or via **File > New > Project**.
- Select the **ASP.NET Core Web Application** template available in **Visual C# > .NET Core** category:



- Tap **Web Application** and verify **Authentication** is set to **Individual User Accounts**:



Note: This tutorial applies to ASP.NET Core 2.0 SDK version which can be selected at the top of the wizard.

Apply migrations

- Run the app and select the **Log in** link.
- Select the **Register as a new user** link.
- Enter the email and password for the new account, and then select **Register**.
- Follow the instructions to apply migrations.

Require SSL

OAuth 2.0 requires the use of SSL for authentication over the HTTPS protocol.

Note: Projects created using **Web Application** or **Web API** project templates for ASP.NET Core 2.x are automatically configured to enable SSL and launch with https URL if the **Individual User Accounts** option was selected on **Change Authentication dialog** in the project wizard as shown above.

- Require SSL on your site by following the steps in [Enforcing SSL in an ASP.NET Core app](#) topic.

Use SecretManager to store tokens assigned by login providers

Social login providers assign **Application Id** and **Application Secret** tokens during the registration process (exact naming varies by provider).

These values are effectively the *user name* and *password* your application uses to access their API, and constitute the "secrets" that can be linked to your application configuration with the help of **Secret Manager** instead of storing them in configuration files directly or hard-coding them.

Follow the steps in [Safe storage of app secrets during development in ASP.NET Core](#) topic so that you can store tokens assigned by each login provider below.

Setup login providers required by your application

Use the following topics to configure your application to use the respective providers:

- [Facebook](#) instructions
- [Twitter](#) instructions
- [Google](#) instructions
- [Microsoft](#) instructions
- [Other provider](#) instructions

Optionally set password

When you register with an external login provider, you do not have a password registered with the app. This alleviates you from creating and remembering a password for the site, but it also makes you dependent on the external login provider. If the external login provider is unavailable, you won't be able to log in to the web site.

To create a password and sign in using your email that you set during the sign in process with external providers:

- Tap the **Hello** link at the top right corner to navigate to the **Manage** view.

WebApplication224 Home About Contact Hello raspranav@gmail.com! Log off

Manage your account.

Change your account settings

Password: [Create]

External Logins: 1 [Manage]

Phone Number: Phone Numbers can be used as a second factor of verification in two-factor authentication. See [this article](#) for details on setting up this ASP.NET application to support two-factor authentication using SMS.

Two-Factor Authentic... There are no two-factor authentication providers configured. See [this article](#) for setting up this application to support two-factor authentication.

- Tap **Create**

WebApplication224 Home About Contact

You do not have a local username/password for this site. Add a local account so you can log in without an external login.

Set your password

New password [Masked]

Confirm new password [Masked]

Set password

© 2015 - WebApplication224

- Set a valid password and you can use this to sign in with your email.

Next steps

- This article introduced external authentication and explained the prerequisites required to add external logins to your ASP.NET Core app.
- Reference provider-specific pages to configure logins for the providers required by your app.

Configuring Facebook authentication

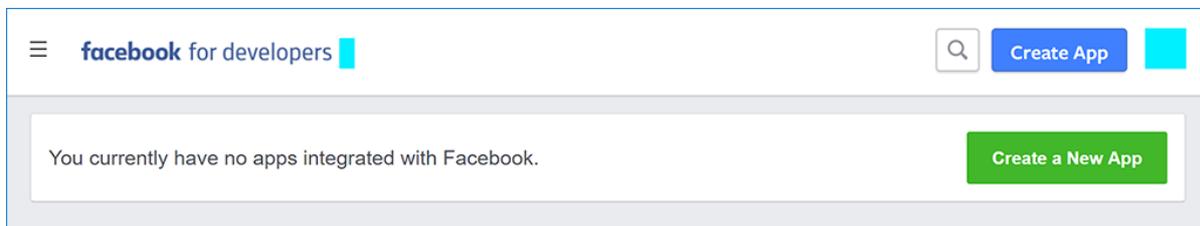
12/13/2017 • 3 min to read • [Edit Online](#)

By [Valeriy Novytsky](#) and [Rick Anderson](#)

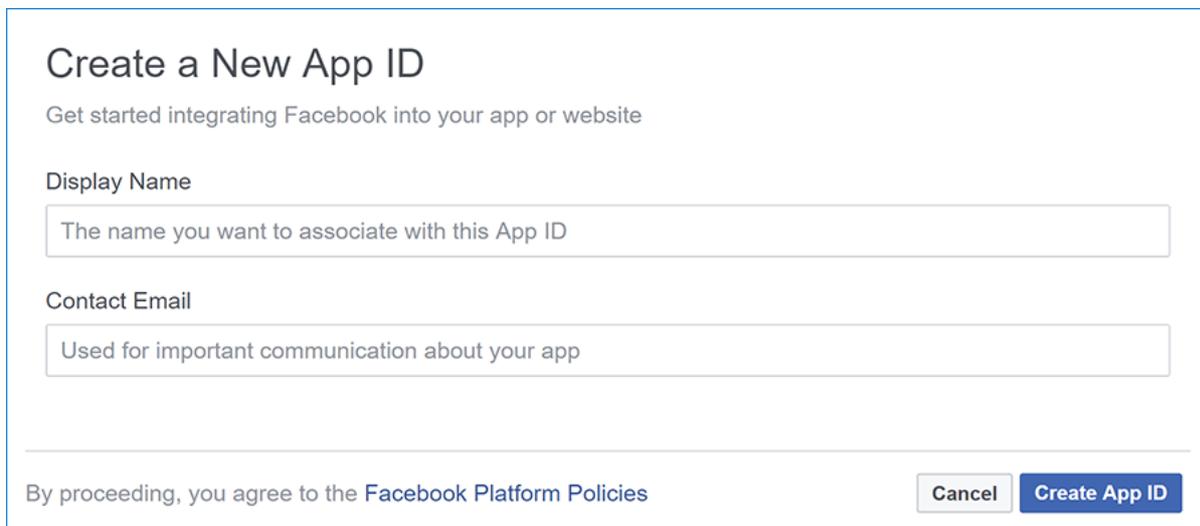
This tutorial shows you how to enable your users to sign in with their Facebook account using a sample ASP.NET Core 2.0 project created on the [previous page](#). We start by creating a Facebook App ID by following the [official steps](#).

Create the app in Facebook

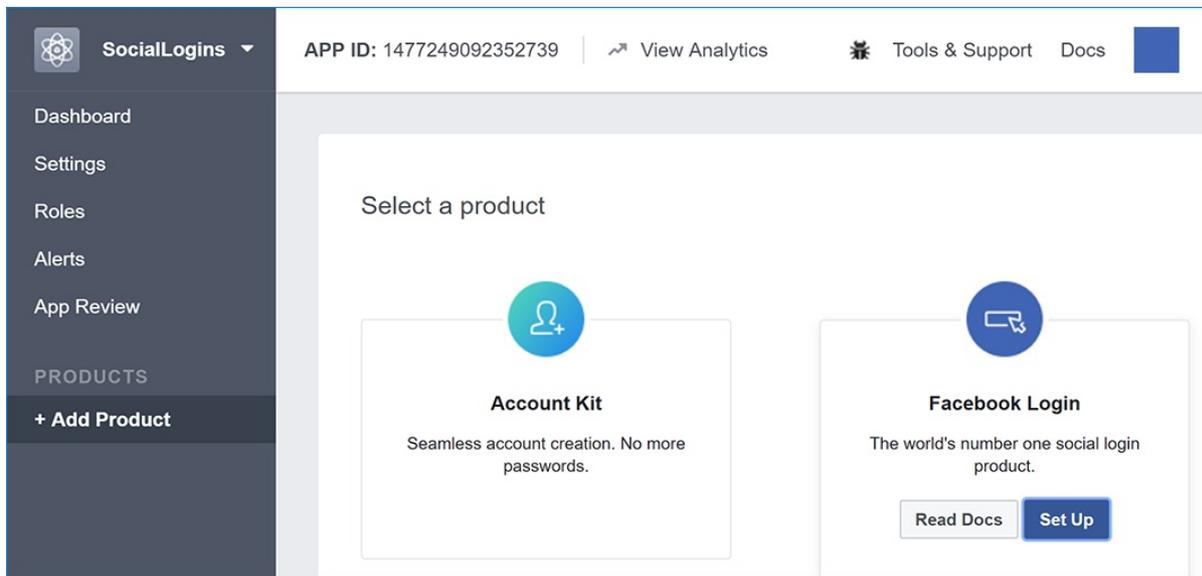
- Navigate to the [Facebook Developers app](#) page and sign in. If you don't already have a Facebook account, use the **Sign up for Facebook** link on the login page to create one.
- Tap the **Add a New App** button in the upper right corner to create a new App ID.



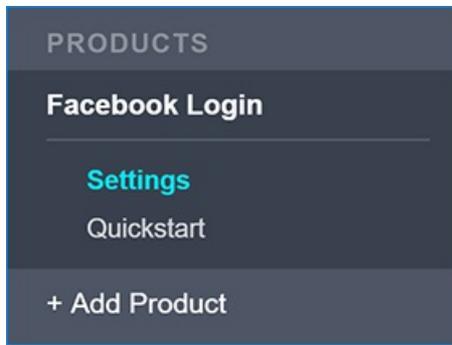
- Fill out the form and tap the **Create App ID** button.

A screenshot of the 'Create a New App ID' form. The title is 'Create a New App ID' with the subtitle 'Get started integrating Facebook into your app or website'. There are two input fields: 'Display Name' with the placeholder text 'The name you want to associate with this App ID' and 'Contact Email' with the placeholder text 'Used for important communication about your app'. At the bottom, there is a line of text: 'By proceeding, you agree to the Facebook Platform Policies' and two buttons: 'Cancel' and 'Create App ID'.

- On the **Select a product** page, click **Set Up** on the **Facebook Login** card.



- The **Quickstart** wizard will launch with **Choose a Platform** as the first page. Bypass the wizard for now by clicking the **Settings** link in the menu on the left:



- You are presented with the **Client OAuth Settings** page:

Client OAuth Settings

Yes

Client OAuth Login
Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URIs are allowed with the options below. Disable globally if not used. [?]

Yes No

Web OAuth Login
Enables web based OAuth client login for building custom login flows. [?]

No No

Force Web OAuth Reauthentication
When on, prompts people to enter their Facebook password in order to log in on the web. [?]

No No

Embedded Browser OAuth Login
Enables browser control redirect uri for OAuth client login. [?]

Valid OAuth redirect URIs

No No

Login from Devices
Enables the OAuth client login flow for devices like a smart TV [?]

Deauthorize

- Enter your development URI with `/signin-facebook` appended into the **Valid OAuth Redirect URIs** field (for example: `https://localhost:44320/signin-facebook`). The Facebook authentication configured later in this tutorial will automatically handle requests at `/signin-facebook` route to implement the OAuth flow.
- Click **Save Changes**.
- Click the **Dashboard** link in the left navigation.

On this page, make a note of your `App ID` and your `App Secret`. You will add both into your ASP.NET Core application in the next section:



SocialLogins o

This app is in development mode and can only be used by app admins, developers and testers [?]

API Version [?] `v2.10` App ID `1477249092352739`

App Secret `.....`

- When deploying the site you need to revisit the **Facebook Login** setup page and register a new public URI.

Store Facebook App ID and App Secret

Link sensitive settings like Facebook `App ID` and `App Secret` to your application configuration using the [Secret Manager](#). For the purposes of this tutorial, name the tokens `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret`.

Execute the following commands to securely store `App ID` and `App Secret` using Secret Manager:

```
dotnet user-secrets set Authentication:Facebook:AppId <app-id>
dotnet user-secrets set Authentication:Facebook:AppSecret <app-secret>
```

Configure Facebook Authentication

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Add the Facebook service in the `ConfigureServices` method in the `Startup.cs` file:

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddFacebook(facebookOptions =>
{
    facebookOptions.AppId = Configuration["Authentication:Facebook:AppId"];
    facebookOptions.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
});
```

Note: The call to `AddIdentity` configures the default scheme settings. The `AddAuthentication(string defaultScheme)` overload sets the `DefaultScheme` property; and, the `AddAuthentication(Action<AuthenticationOptions> configureOptions)` overload sets only the properties you explicitly set. Either of these overloads should only be called once when adding multiple authentication providers. Subsequent calls to it have the potential of overriding any previously configured [AuthenticationOptions](#) properties.

See the [FacebookOptions](#) API reference for more information on configuration options supported by Facebook authentication. Configuration options can be used to:

- Request different information about the user.
- Add query string arguments to customize the login experience.

Sign in with Facebook

Run your application and click **Log in**. You see an option to sign in with Facebook.

Log in.

Use a local account to log in.

Use another service to log in.

Email

Password

Remember me?

Log in

[Register as a new user?](#)

[Forgot your password?](#)

© 2016 - WebApplication3

Facebook

When you click on **Facebook**, you are redirected to Facebook for authentication:

facebook [Sign Up](#)

Log into Facebook

aspdemo@outlook.com

Password

Log In

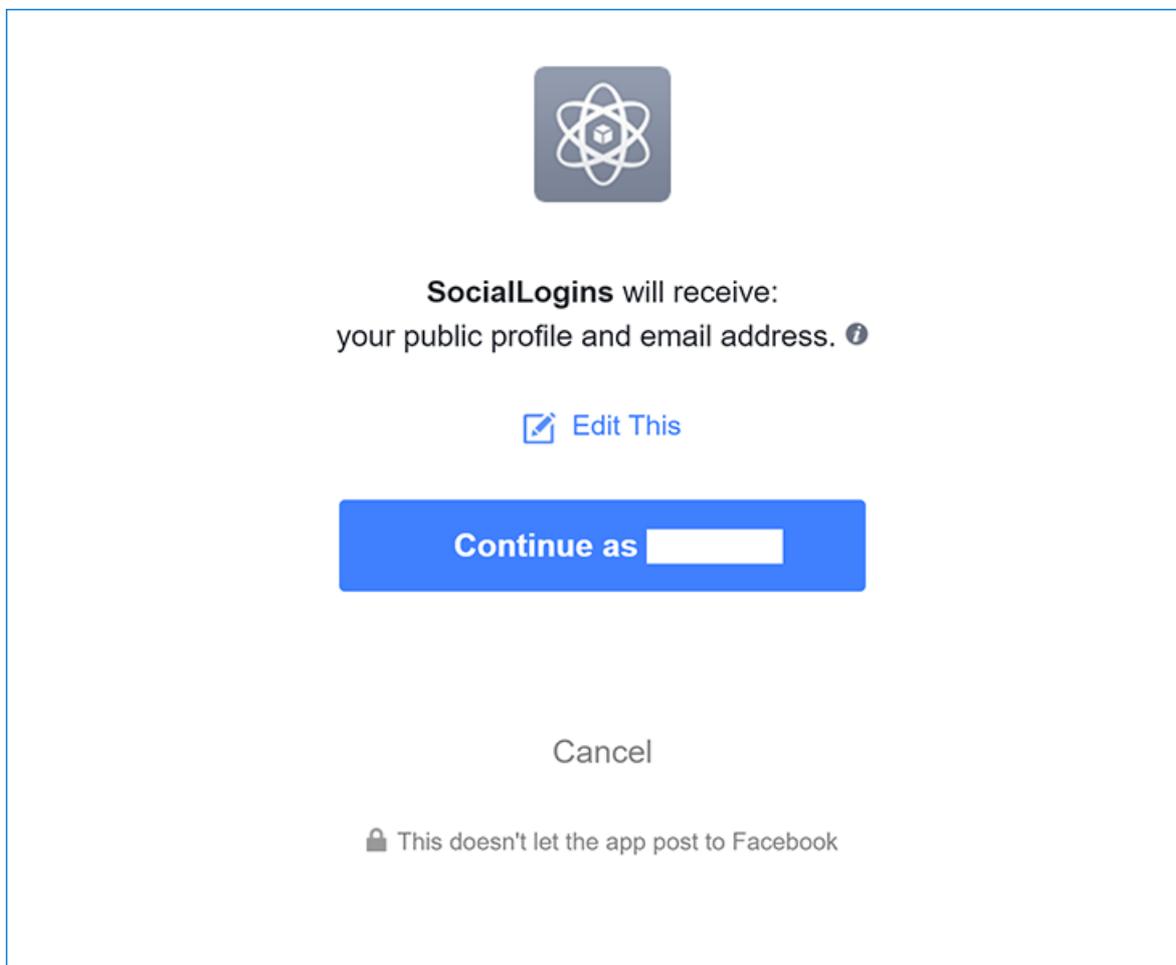
or

[Create New Account](#)

[Forgot account?](#)

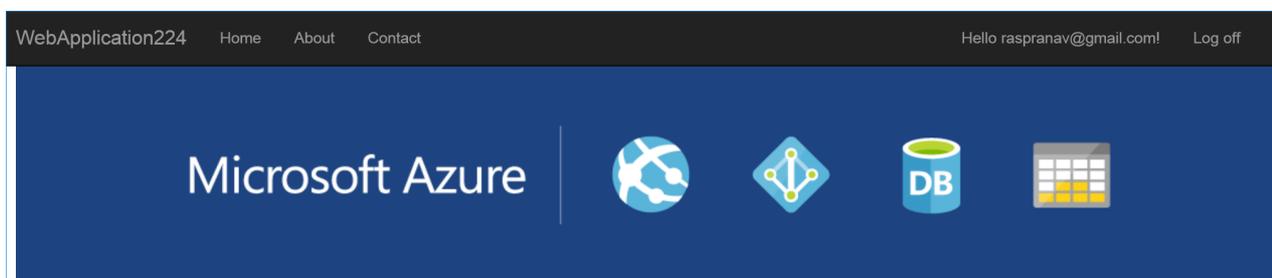
[Not now](#)

Facebook authentication requests public profile and email address by default:



Once you enter your Facebook credentials you are redirected back to your site where you can set your email.

You are now logged in using your Facebook credentials:



Troubleshooting

- **ASP.NET Core 2.x only:** If Identity is not configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Facebook. You can follow a similar approach to authenticate with other providers listed on the [previous page](#).
- Once you publish your web site to Azure web app, you should reset the `AppSecret` in the Facebook developer portal.

- Set the `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Configuring Twitter authentication

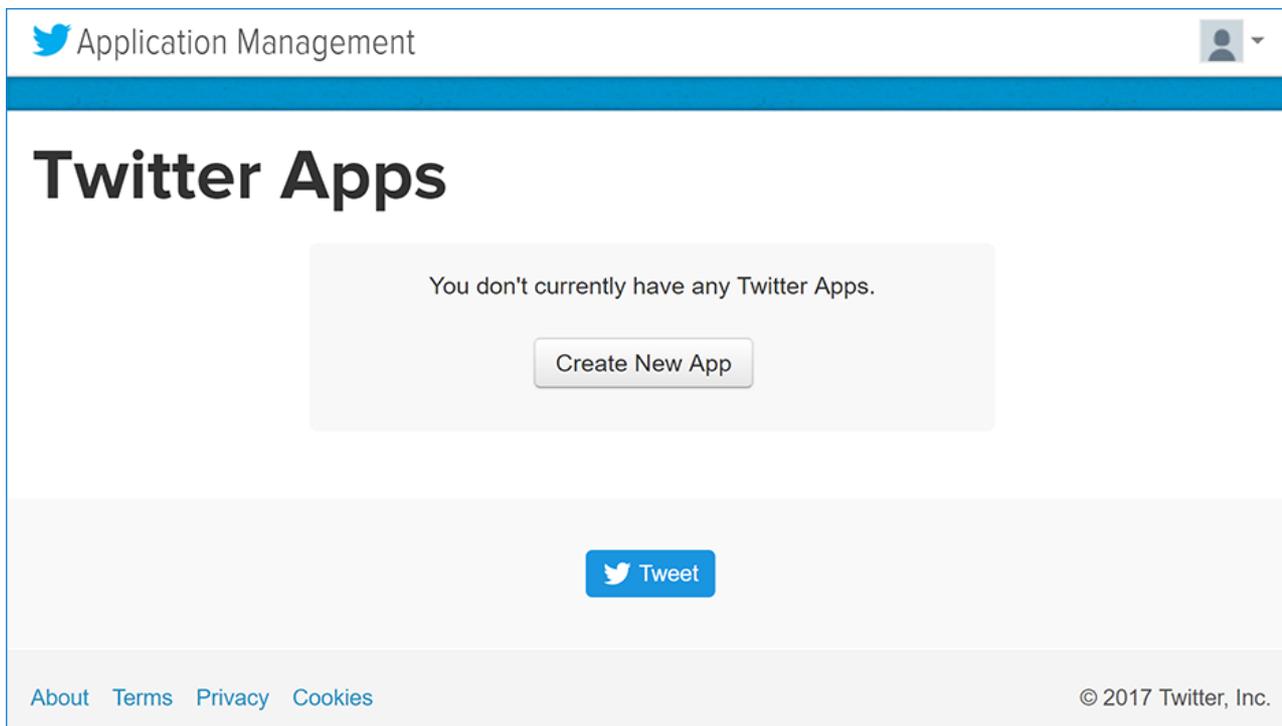
11/1/2017 • 3 min to read • [Edit Online](#)

By [Valeriy Novytsky](#) and [Rick Anderson](#)

This tutorial shows you how to enable your users to [sign in with their Twitter account](#) using a sample ASP.NET Core 2.0 project created on the [previous page](#).

Create the app in Twitter

- Navigate to <https://apps.twitter.com/> and sign in. If you don't already have a Twitter account, use the [Sign up now](#) link to create one. After signing in, the **Application Management** page is shown:



- Tap **Create New App** and fill out the application **Name**, **Description** and public **Website** URI (this can be temporary until you register the domain name):



Create an application

Application Details

Name *

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? **OAuth 1.0a** applications should explicitly specify their `oauth_callback` URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Developer Agreement

Yes, I have read and agree to the [Twitter Developer Agreement](#).

Create your Twitter application

- Enter your development URI with `/signin-twitter` appended into the **Valid OAuth Redirect URIs** field (for example: `https://localhost:44320/signin-twitter`). The Twitter authentication scheme configured later in this tutorial will automatically handle requests at `/signin-twitter` route to implement the OAuth flow.
- Fill out the rest of the form and tap **Create your Twitter application**. New application details are displayed:

Application Management [User Profile]

Your application has been created. Please take a moment to review and adjust your application's settings.

SocialLogins

Test OAuth

Details Settings Keys and Access Tokens Permissions

 Social login demo
<https://www.mysite.com/>

Organization

Information about the organization or company associated with your application. This information is optional.

Organization	None
Organization website	None

Application Settings

Your application's Consumer Key and Secret are used to **authenticate** requests to the Twitter Platform.

Access level	Read and write (modify app permissions)
Consumer Key (API Key)	(manage keys and access)

- When deploying the site you'll need to revisit the **Application Management** page and register a new public URI.

Storing Twitter ConsumerKey and ConsumerSecret

Link sensitive settings like Twitter `Consumer Key` and `Consumer Secret` to your application configuration using the [Secret Manager](#). For the purposes of this tutorial, name the tokens `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret`.

These tokens can be found on the **Keys and Access Tokens** tab after creating your new Twitter application:

Application Management 

SocialLogins

Test OAuth

Details Settings **Keys and Access Tokens** Permissions

Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key)

Consumer Secret (API Secret)

Access Level Read and write ([modify app permissions](#))

Owner

Owner ID 890753103795859456

Application Actions

[Regenerate Consumer Key and Secret](#) [Change App Permissions](#)

Configure Twitter Authentication

The project template used in this tutorial ensures that `Microsoft.AspNetCore.Authentication.Twitter` package is already installed.

- To install this package with Visual Studio 2017, right-click on the project and select **Manage NuGet Packages**.
- To install with .NET Core CLI, execute the following in your project directory:

```
dotnet add package Microsoft.AspNetCore.Authentication.Twitter
```

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Add the Twitter service in the `ConfigureServices` method in `Startup.cs` file:

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddTwitter(twitterOptions =>
{
    twitterOptions.ConsumerKey = Configuration["Authentication:Twitter:ConsumerKey"];
    twitterOptions.ConsumerSecret = Configuration["Authentication:Twitter:ConsumerSecret"];
});
```

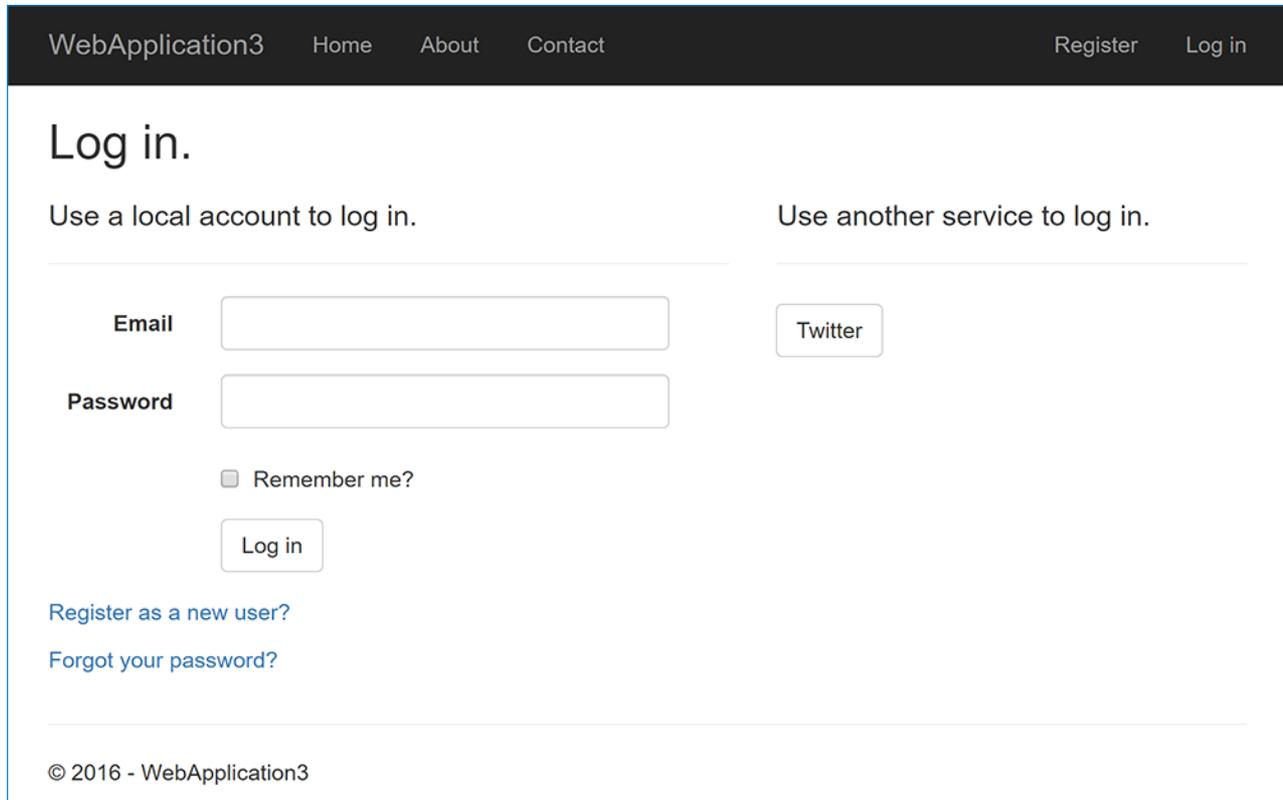
Note: The call to `AddIdentity` configures the default scheme settings. The `AddAuthentication(string defaultScheme)` overload sets the `DefaultScheme` property; and, the

`AddAuthentication(Action<AuthenticationOptions> configureOptions)` overload sets only the properties you explicitly set. Either of these overloads should only be called once when adding multiple authentication providers. Subsequent calls to it have the potential of overriding any previously configured [AuthenticationOptions](#) properties.

See the [TwitterOptions](#) API reference for more information on configuration options supported by Twitter authentication. This can be used to request different information about the user.

Sign in with Twitter

Run your application and click **Log in**. An option to sign in with Twitter appears:



The screenshot shows a web application interface for logging in. At the top, there is a navigation bar with links for 'WebApplication3', 'Home', 'About', 'Contact', 'Register', and 'Log in'. The main content area is titled 'Log in.' and is divided into two sections: 'Use a local account to log in.' and 'Use another service to log in.' The local login section contains input fields for 'Email' and 'Password', a 'Remember me?' checkbox, and a 'Log in' button. Below these fields are links for 'Register as a new user?' and 'Forgot your password?'. The social login section features a 'Twitter' button. At the bottom of the page, there is a copyright notice: '© 2016 - WebApplication3'.

Clicking on **Twitter** redirects to Twitter for authentication:



Sign up for Twitter ›

Authorize SocialLogins to use your account?

Remember me · [Forgot password?](#)

This application will be able to:

- Read Tweets from your timeline.
- See who you follow.

Will not be able to:

- Follow new people.
- Update your profile.
- Post Tweets for you.
- Access your direct messages.
- See your email address.
- See your Twitter password.



SocialLogins
www.mysite.com
Social login demo

After entering your Twitter credentials, you are redirected back to the web site where you can set your email.

You are now logged in using your Twitter credentials:

WebApplication224 Home About Contact Hello raspranav@gmail.com! [Log off](#)








Troubleshooting

- **ASP.NET Core 2.x only:** If Identity is not configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided.* The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Twitter. You can follow a similar approach to authenticate with other providers listed on the [previous page](#).
- Once you publish your web site to Azure web app, you should reset the `ConsumerSecret` in the Twitter developer portal.
- Set the `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Configuring Google authentication in ASP.NET Core

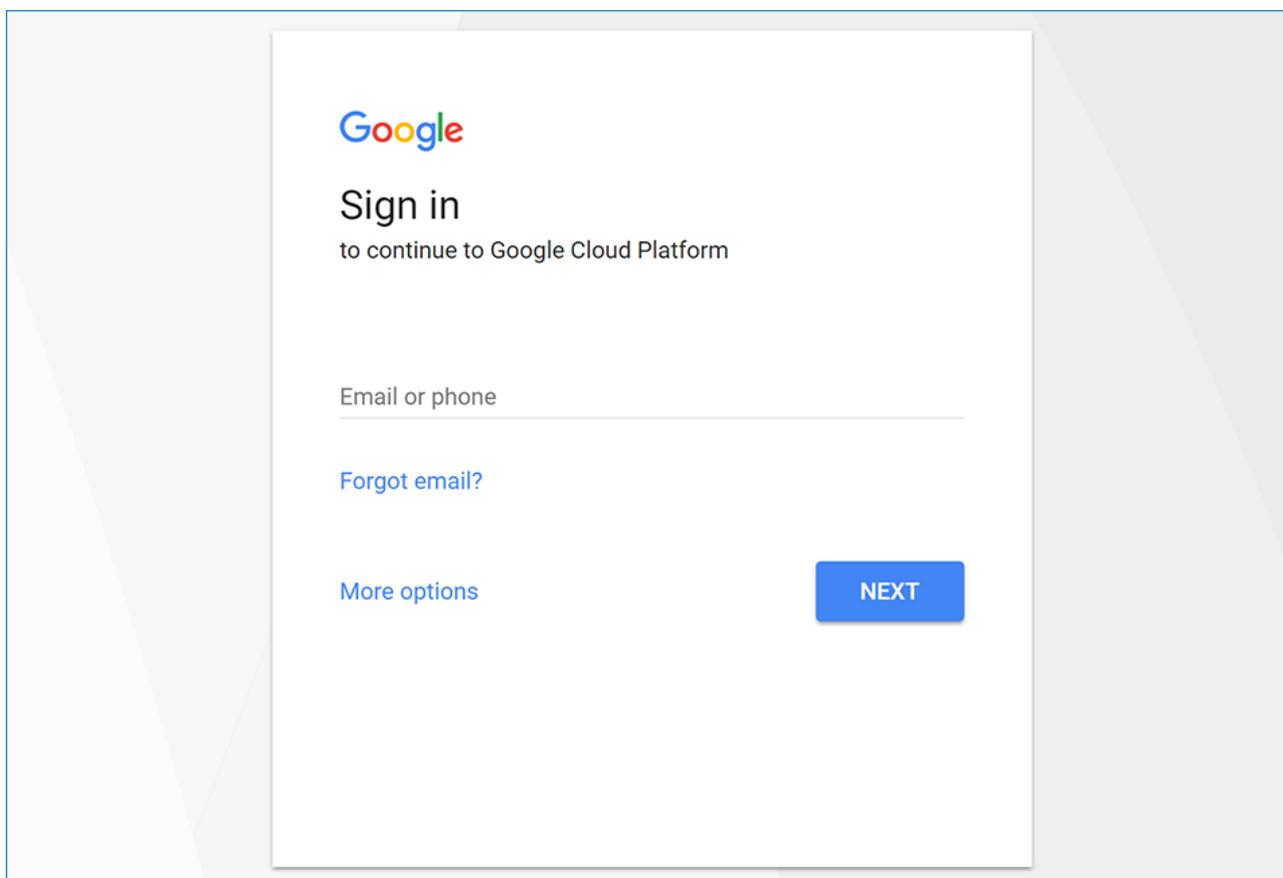
11/16/2017 • 4 min to read • [Edit Online](#)

By [Valeriy Novytsky](#) and [Rick Anderson](#)

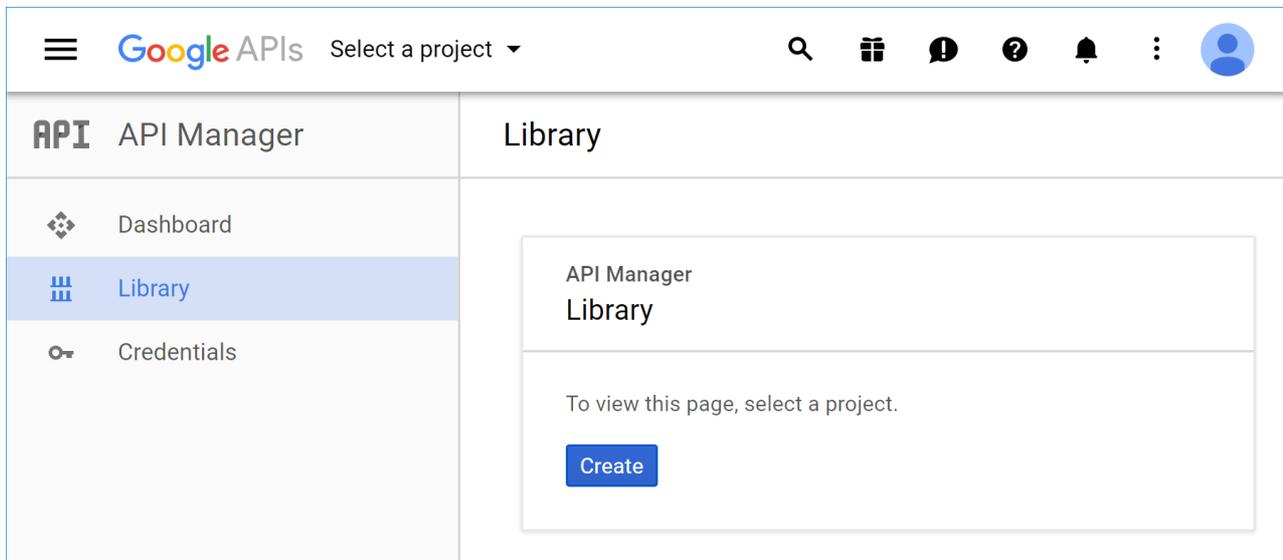
This tutorial shows you how to enable your users to sign in with their Google+ account using a sample ASP.NET Core 2.0 project created on the [previous page](#). We start by following the [official steps](#) to create a new app in Google API Console.

Create the app in Google API Console

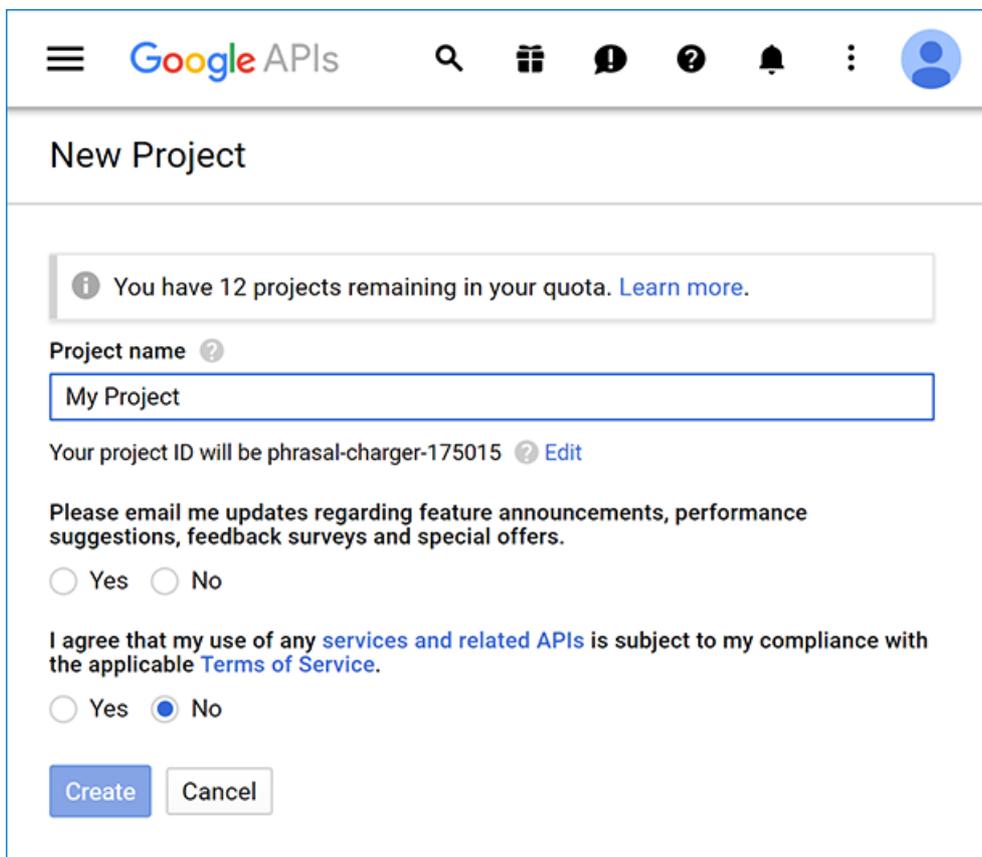
- Navigate to <https://console.developers.google.com/projectselector/apis/library> and sign in. If you don't already have a Google account, use **More options** > **Create account** link to create one:



- You are redirected to **API Manager Library** page:



- Tap **Create** and enter your **Project name**:



- After accepting the dialog, you are redirected back to the Library page allowing you to choose features for your new app. Find **Google+ API** in the list and click on its link to add the API feature:

The screenshot shows the Google APIs Library interface. On the left is a navigation menu with 'API Manager', 'Dashboard', 'Library' (selected), and 'Credentials'. The main content area is titled 'Library' and contains a search bar with 'Google+ API' entered. Below the search bar is a 'Back to popular APIs' link. A table lists search results:

Name	Description
Google+ API	The Google+ API enables developers to build on top of the Google+ platform.
Google+ Domains API	The Google+ Domains API enables developers to build on top of the Google+ platform for Google Apps

- The page for the newly added API is displayed. Tap **Enable** to add Google+ sign in feature to your app:

The screenshot shows the configuration page for the 'Google+ API'. At the top left is a back arrow and the text 'Google+ API'. At the top right is an 'ENABLE' button. The main content area is titled 'About this API' and includes links for 'Documentation' and 'Try this API in APIs Explorer'. It contains two sections with diagrams:

Using credentials with this API
Accessing user data with OAuth 2.0
 You can access user data with this API. On the Credentials page, create an OAuth 2.0 client ID. A client ID requests user consent so that your app can access user data. Include that client ID when making your API call to Google. [Learn more](#)

The diagram for 'Accessing user data with OAuth 2.0' shows a flow: 'Your app' (with an OAuth 2.0 icon) → 'User consent' (with a laptop and smartphone icon) → 'User data' (with a folder icon).

Server-to-server interaction
 You can use this API to perform server-to-server interaction, for example between a web application and a Google service. You'll need a service account, which enables app-level authentication. You'll also need a service account key, which is used to authorize your API call to Google. [Learn more](#)

The diagram for 'Server-to-server interaction' shows a flow: 'Your service' (with server rack icons) → 'Authorization' (with a document icon) → 'Google service' (with server rack icons).

- After enabling the API, tap **Create credentials** to configure the secrets:

The screenshot shows the configuration page for the 'Google+ API' after it has been enabled. The top right now shows a 'DISABLE' button. A yellow warning banner at the top of the main content area reads: 'To use this API, you may need credentials. Click "Create credentials" to get started.' Below the banner are links for 'Overview' and 'Quotas', and a prominent 'Create credentials' button.

- Choose:
 - **Google+ API**
 - **Web server (e.g. node.js, Tomcat), and**
 - **User data:**

Google APIs SocialLogins

API Manager Credentials

Dashboard
Library
Credentials

Add credentials to your project

- Find out what kind of credentials you need

We'll help you set up the correct credentials
If you wish you can skip this step and create an [API key](#), [client ID](#), or [service account](#)

Which API are you using?
Determines what kind of credentials you need.

Google+ API

Where will you be calling the API from?
Determines which settings you'll need to configure.

Web server (e.g. node.js, Tomcat)

What data will you be accessing?

User data
Access data belonging to a Google user, with their permission

Application data
Access data belonging to your own application

[What credentials do I need?](#)
- Get your credentials

< | Cancel

- Tap **What credentials do I need?** which takes you to the second step of app configuration, **Create an OAuth 2.0 client ID:**

Google APIs SocialLogins

API Manager

Dashboard

Library

Credentials

Credentials

Add credentials to your project

- Find out what kind of credentials you need
Calling Google+ API from a web server
- Create an OAuth 2.0 client ID

Name

Restrictions
Enter JavaScript origins, redirect URIs, or both

Authorized JavaScript origins
For use with requests from a browser. This is the origin URI of the client application. It can't contain a wildcard (`http://*.example.com`) or a path (`http://example.com/subdir`). If you're using a nonstandard port, you must include it in the origin URI.

Authorized redirect URIs
For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorization code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

Create client ID

- Because we are creating a Google+ project with just one feature (sign in), we can enter the same **Name** for the OAuth 2.0 client ID as the one we used for the project.
- Enter your development URI with `/signin-google` appended into the **Authorized redirect URIs** field (for example: `https://localhost:44320/signin-google`). The Google authentication configured later in this tutorial will automatically handle requests at `/signin-google` route to implement the OAuth flow.
- Press TAB to add the **Authorized redirect URIs** entry.
- Tap **Create client ID**, which takes you to the third step, **Set up the OAuth 2.0 consent screen**:

Google APIs SocialLogins

API Manager

Credentials

Add credentials to your project

- ✓ Find out what kind of credentials you need
Calling Google+ API from a web server
- ✓ Create an OAuth 2.0 client ID
Created OAuth client 'Web client 1'

3 Set up the OAuth 2.0 consent screen

Email address [?]

Product name shown to users [?]

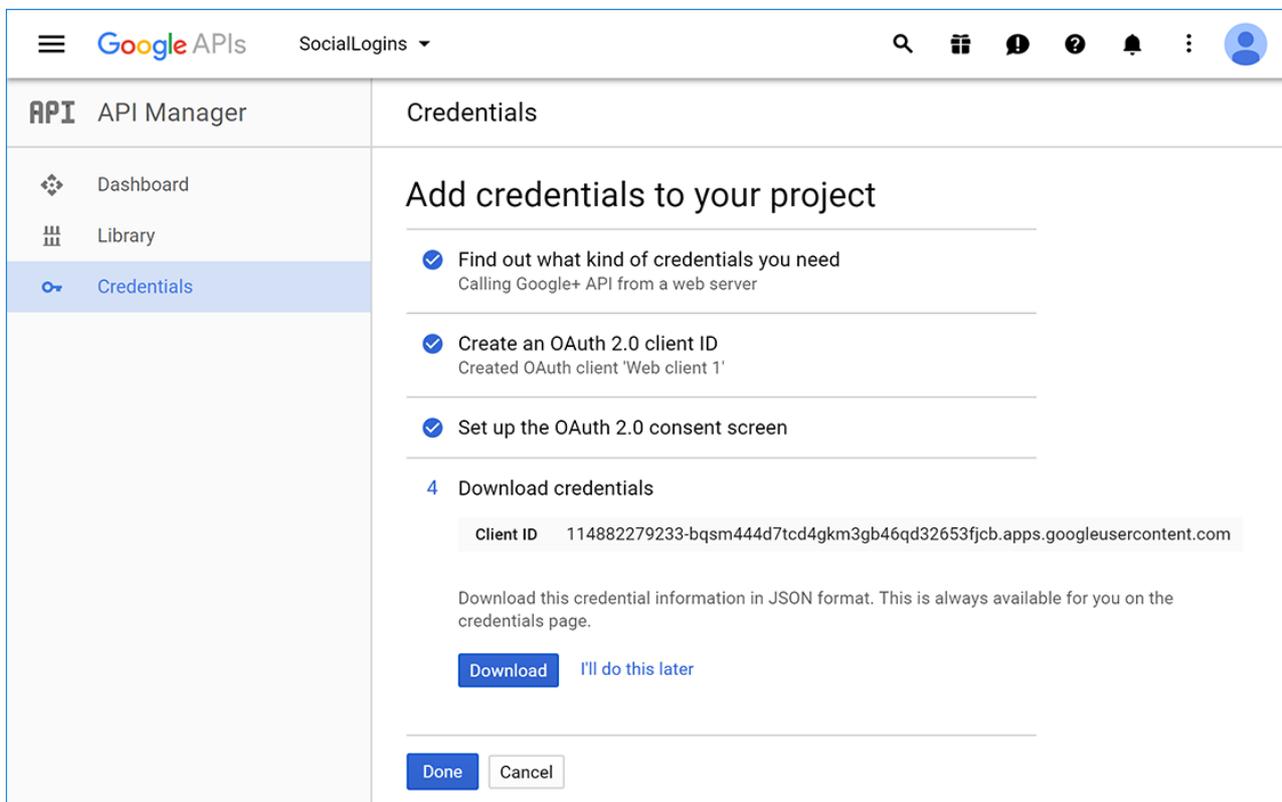
[More customization options](#)

[Continue](#)



The consent screen will be shown to users whenever you request access to their private data using your client ID. It will be shown for all applications registered in this project.

- Enter your public facing **Email address** and the **Product name** shown for your app when Google+ prompts the user to sign in. Additional options are available under **More customization options**.
- Tap **Continue** to proceed to the last step, **Download credentials**:



- Tap **Download** to save a JSON file with application secrets, and **Done** to complete creation of the new app.
- When deploying the site you'll need to revisit the **Google Console** and register a new public url.

Store Google ClientID and ClientSecret

Link sensitive settings like Google `Client ID` and `Client Secret` to your application configuration using the [Secret Manager](#). For the purposes of this tutorial, name the tokens `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret`.

The values for these tokens can be found in the JSON file downloaded in the previous step under `web.client_id` and `web.client_secret`.

Configure Google Authentication

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Add the Google service in the `ConfigureServices` method in `Startup.cs` file:

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId = Configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
});
```

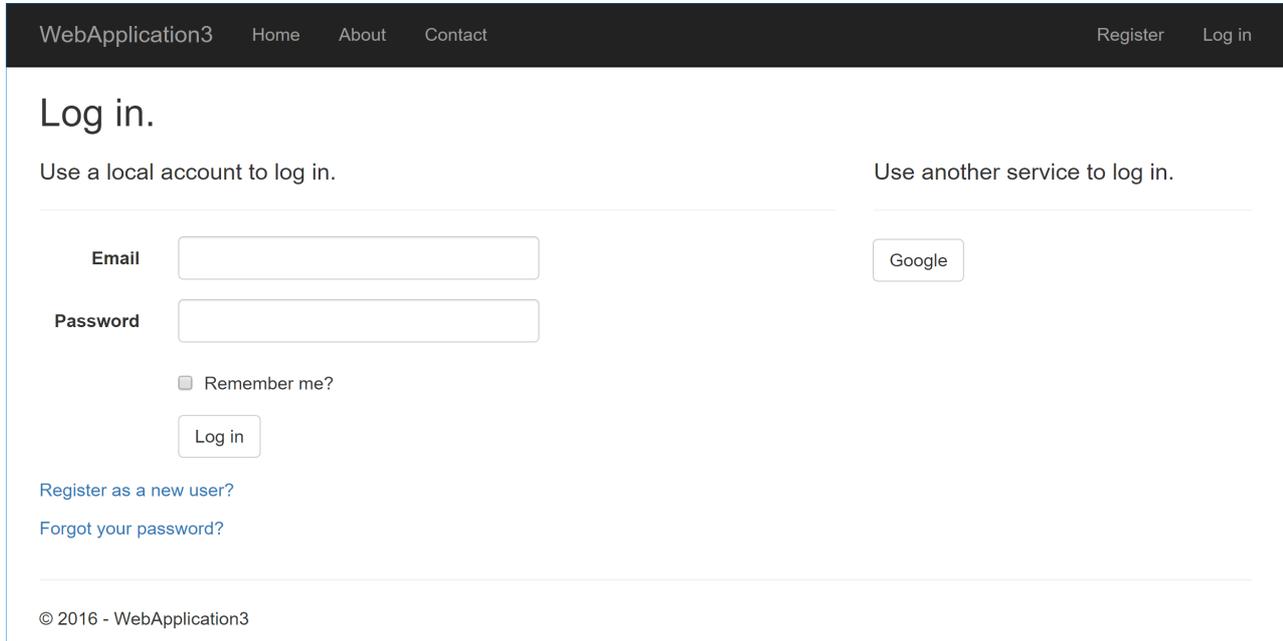
Note: The call to `AddIdentity` configures the default scheme settings. The `AddAuthentication(string defaultScheme)` overload sets the `DefaultScheme` property; and, the `AddAuthentication(Action<AuthenticationOptions> configureOptions)` overload sets only the properties you explicitly set. Either of these overloads should only be called once when adding multiple authentication providers.

Subsequent calls to it have the potential of overriding any previously configured [AuthenticationOptions](#) properties.

See the [GoogleOptions](#) API reference for more information on configuration options supported by Google authentication. This can be used to request different information about the user.

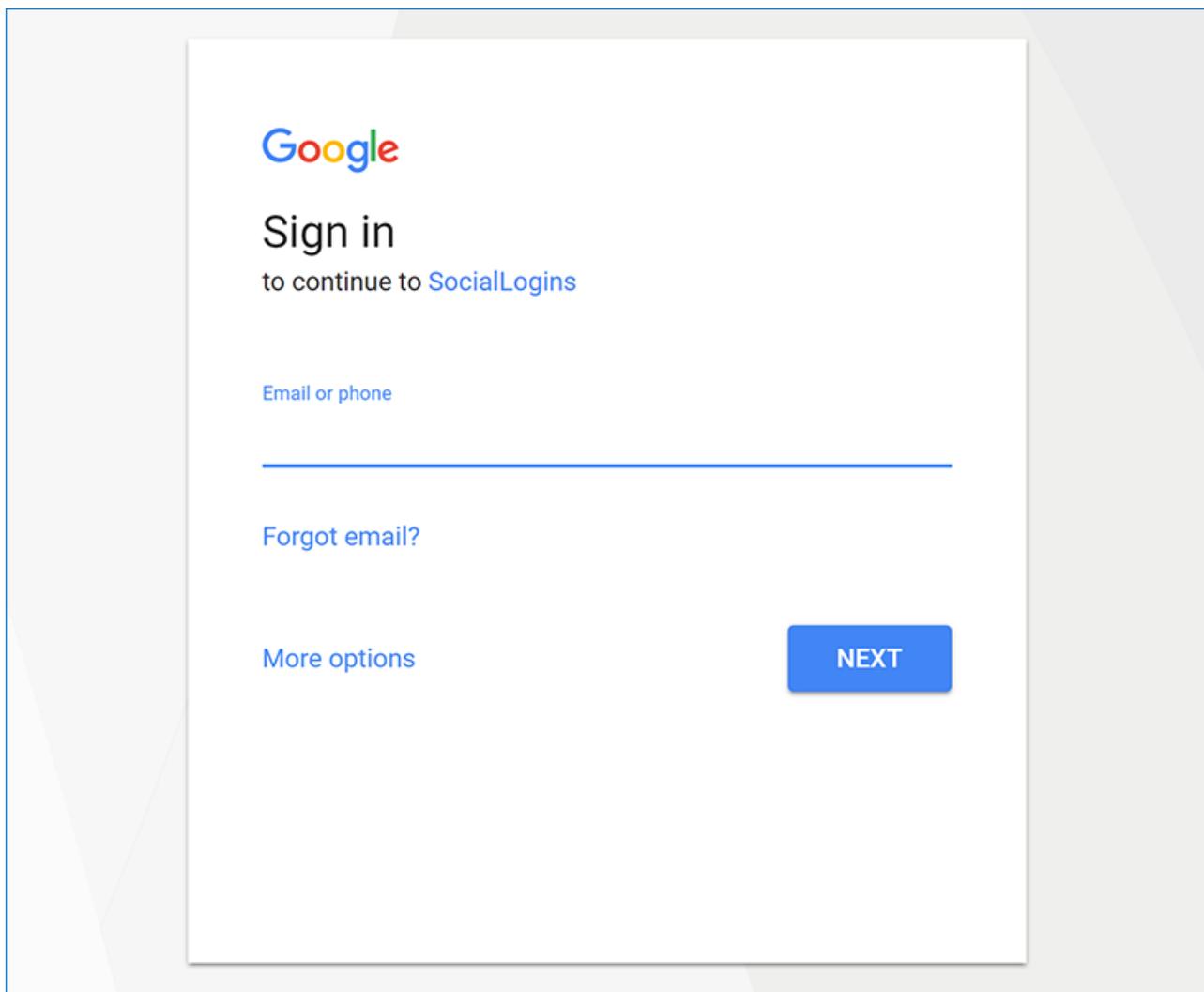
Sign in with Google

Run your application and click **Log in**. An option to sign in with Google appears:



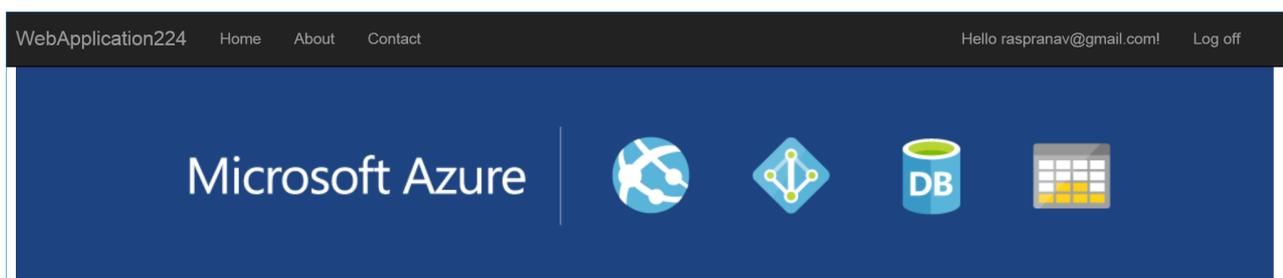
The screenshot shows a web application interface with a dark navigation bar at the top containing 'WebApplication3', 'Home', 'About', 'Contact', 'Register', and 'Log in'. The main content area is titled 'Log in.' and is split into two columns. The left column, 'Use a local account to log in.', contains input fields for 'Email' and 'Password', a 'Remember me?' checkbox, and a 'Log in' button. Below these are links for 'Register as a new user?' and 'Forgot your password?'. The right column, 'Use another service to log in.', features a 'Google' button. A footer at the bottom left reads '© 2016 - WebApplication3'.

When you click on Google, you are redirected to Google for authentication:



After entering your Google credentials, then you are redirected back to the web site where you can set your email.

You are now logged in using your Google credentials:



Troubleshooting

- If you receive a `403 (Forbidden)` error page from your own app when running in development mode (or break into the debugger with the same error), ensure that **Google+ API** has been enabled in the **API Manager Library** by following the steps listed [earlier on this page](#). If the sign in doesn't work and you aren't getting any errors, switch to development mode to make the issue easier to debug.
- **ASP.NET Core 2.x only:** If Identity is not configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Google. You can follow a similar approach to authenticate with other providers listed on the [previous page](#).
- Once you publish your web site to Azure web app, you should reset the `ClientSecret` in the Google API Console.
- Set the `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Configuring Microsoft Account authentication

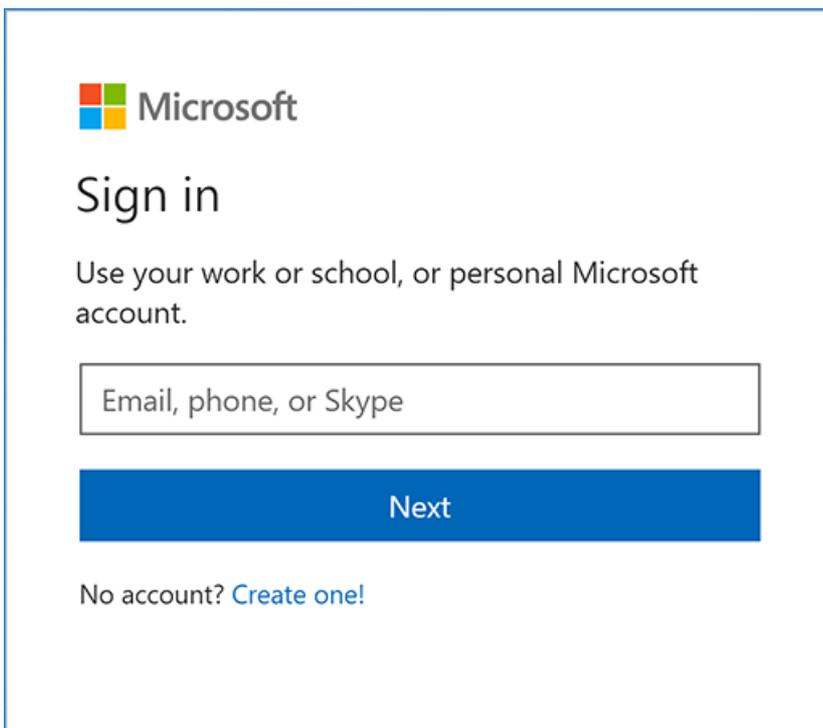
11/1/2017 • 4 min to read • [Edit Online](#)

By [Valeriy Novytsky](#) and [Rick Anderson](#)

This tutorial shows you how to enable your users to sign in with their Microsoft account using a sample ASP.NET Core 2.0 project created on the [previous page](#).

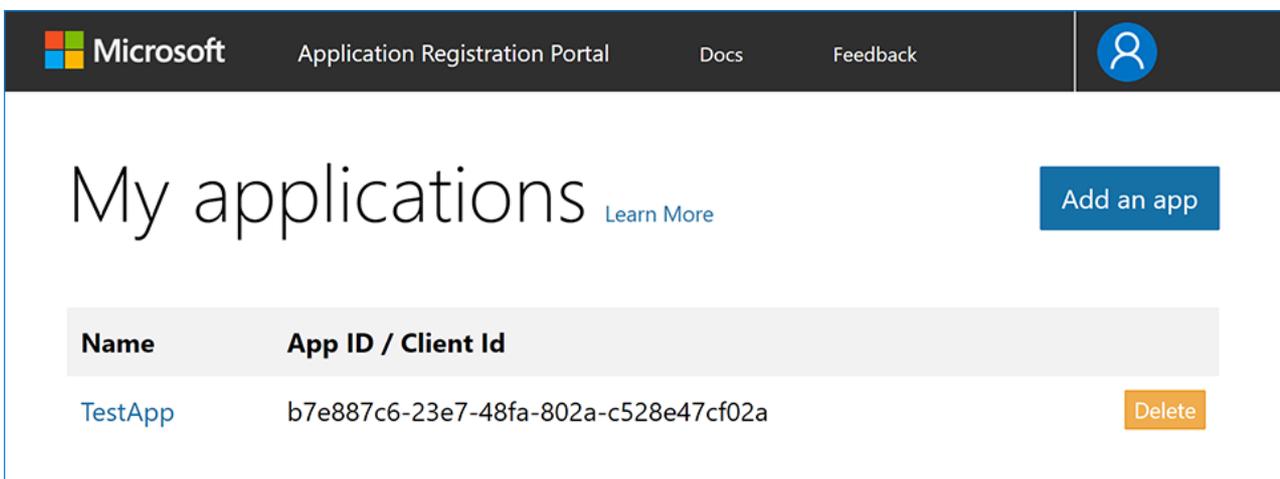
Create the app in Microsoft Developer Portal

- Navigate to <https://apps.dev.microsoft.com> and create or sign into a Microsoft account:



The screenshot shows the Microsoft sign-in interface. At the top left is the Microsoft logo. Below it, the text "Sign in" is displayed in a large font. Underneath, a message reads "Use your work or school, or personal Microsoft account." There is a text input field containing the placeholder text "Email, phone, or Skype". Below the input field is a blue button labeled "Next". At the bottom of the form, there is a link that says "No account? [Create one!](#)".

If you don't already have a Microsoft account, tap [Create one!](#) After signing in you are redirected to **My applications** page:



The screenshot shows the "My applications" page in the Microsoft Application Registration Portal. The top navigation bar includes the Microsoft logo, "Application Registration Portal", "Docs", "Feedback", and a user profile icon. The main heading is "My applications" with a "Learn More" link. In the top right corner, there is a blue button labeled "Add an app". Below this is a table listing applications:

Name	App ID / Client Id	
TestApp	b7e887c6-23e7-48fa-802a-c528e47cf02a	Delete

- Tap **Add an app** in the upper right corner and enter your **Application Name** and **Contact Email**:



Register your application

Application Name

Contact Email

Used for important communications about your application

Guided Setup

Let us help you get started

By proceeding, you agree to the [Microsoft Platform Policies](#)

Create

- For the purposes of this tutorial, clear the **Guided Setup** check box.
- Tap **Create** to continue to the **Registration** page. Provide a **Name** and note the value of the **Application Id**, which you use as `clientId` later in the tutorial:



My applications / SocialLogins

SocialLogins Registration

[Click here for help integrating your application with Microsoft.](#)

Properties

Name

SocialLogins

Application Id

4d519acd-6a8c-4df4-a246-fe5ea741c9db

Application Secrets

[Generate New Password](#)

[Generate New Key Pair](#)

[Upload Public Key](#)

Platforms

[Add Platform](#)

- Tap **Add Platform** in the **Platforms** section and select the **Web** platform:

Add Platform



Native Application



Web API

Cancel

- In the new **Web** platform section, enter your development URL with `/signin-microsoft` appended into the **Redirect URLs** field (for example: `https://localhost:44320/signin-microsoft`). The Microsoft authentication scheme configured later in this tutorial will automatically handle requests at `/signin-microsoft` route to implement the OAuth flow:

Platforms

Add Platform

Web Delete

Allow Implicit Flow

Redirect URLs ⓘ **Add URL**

`https://localhost:44320/signin-microsoft`

Logout URL ⓘ

e.g. `https://myapp.com/end-session`

- Tap **Add URL** to ensure the URL was added.
- Fill out any other application settings if necessary and tap **Save** at the bottom of the page to save changes to app configuration.
- When deploying the site you'll need to revisit the **Registration** page and set a new public URL.

Store Microsoft Application Id and Password

- Note the `Application Id` displayed on the **Registration** page.
- Tap **Generate New Password** in the **Application Secrets** section. This displays a box where you can copy the application password:

New password generated

This is the only time when it will be displayed. Please store it securely.

bae

Ok

Link sensitive settings like Microsoft `Application ID` and `Password` to your application configuration using the [Secret Manager](#). For the purposes of this tutorial, name the tokens `Authentication:Microsoft:ApplicationId` and `Authentication:Microsoft:Password`.

Configure Microsoft Account Authentication

The project template used in this tutorial ensures that [Microsoft.AspNetCore.Authentication.MicrosoftAccount](#) package is already installed.

- To install this package with Visual Studio 2017, right-click on the project and select **Manage NuGet Packages**.
- To install with .NET Core CLI, execute the following in your project directory:

```
dotnet add package Microsoft.AspNetCore.Authentication.MicrosoftAccount
```

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Add the Microsoft Account service in the `ConfigureServices` method in *Startup.cs* file:

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddMicrosoftAccount(microsoftOptions =>
{
    microsoftOptions.ClientId = Configuration["Authentication:Microsoft:ApplicationId"];
    microsoftOptions.ClientSecret = Configuration["Authentication:Microsoft:Password"];
});
```

Note: The call to `AddIdentity` configures the default scheme settings. The `AddAuthentication(string defaultScheme)` overload sets the `DefaultScheme` property; and, the `AddAuthentication(Action<AuthenticationOptions> configureOptions)` overload sets only the properties you explicitly set. Either of these overloads should only be called once when adding multiple authentication providers. Subsequent calls to it have the potential of overriding any previously configured [AuthenticationOptions](#) properties.

Although the terminology used on Microsoft Developer Portal names these tokens `ApplicationId` and `Password`, they are exposed as `ClientId` and `ClientSecret` to the configuration API.

See the [MicrosoftAccountOptions](#) API reference for more information on configuration options supported by Microsoft Account authentication. This can be used to request different information about the user.

Sign in with Microsoft Account

Run your application and click **Log in**. An option to sign in with Microsoft appears:

Log in.

Use a local account to log in.

Use another service to log in.

Email

Password

Remember me?

Log in

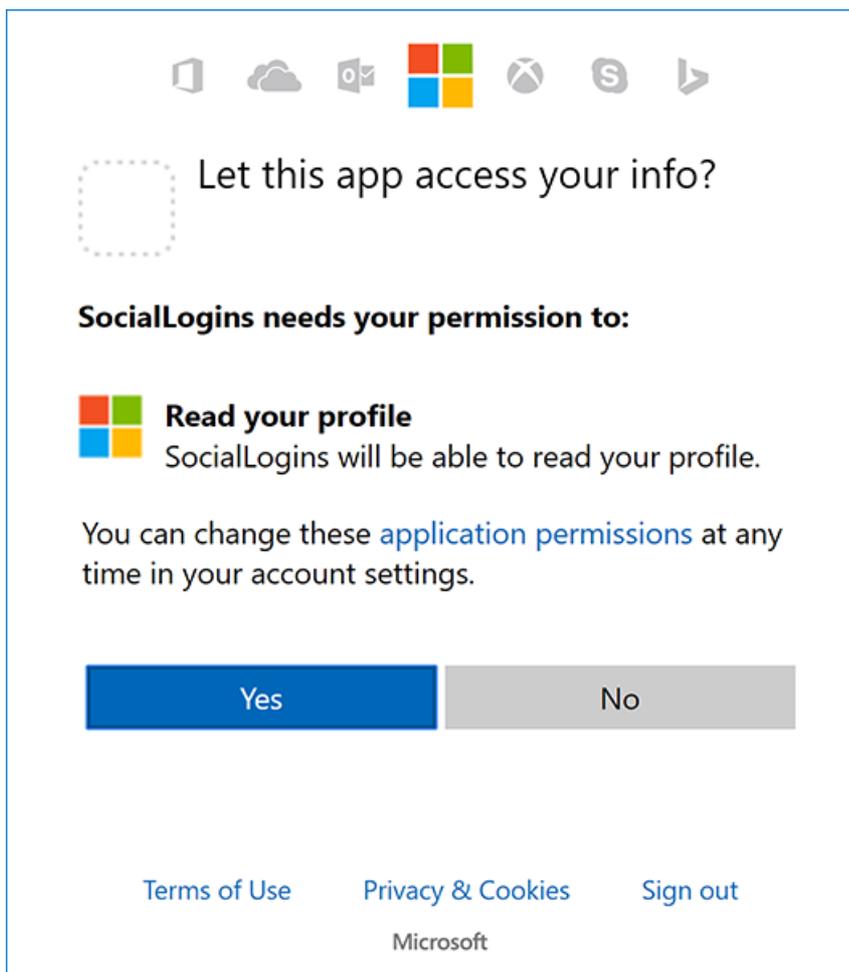
[Register as a new user?](#)

[Forgot your password?](#)

© 2016 - WebApplication3

Microsoft

When you click on Microsoft, you are redirected to Microsoft for authentication. After signing in with your Microsoft Account (if not already signed in) you will be prompted to let the app access your info:



Let this app access your info?

SocialLogins needs your permission to:

 **Read your profile**
SocialLogins will be able to read your profile.

You can change these [application permissions](#) at any time in your account settings.

[Terms of Use](#) [Privacy & Cookies](#) [Sign out](#)

Microsoft

Tap **Yes** and you will be redirected back to the web site where you can set your email.

You are now logged in using your Microsoft credentials:

Microsoft Azure



Troubleshooting

- If the Microsoft Account provider redirects you to a sign in error page, note the error title and description query string parameters directly following the `#` (hashtag) in the Uri.

Although the error message seems to indicate a problem with Microsoft authentication, the most common cause is your application Uri not matching any of the **Redirect URIs** specified for the **Web** platform.

- **ASP.NET Core 2.x only:** If Identity is not configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Microsoft. You can follow a similar approach to authenticate with other providers listed on the [previous page](#).
- Once you publish your web site to Azure web app, you should create a new `Password` in the Microsoft Developer Portal.
- Set the `Authentication:Microsoft:ApplicationId` and `Authentication:Microsoft:Password` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Short survey of other authentication providers

10/13/2017 • 1 min to read • [Edit Online](#)

By [Rick Anderson](#), [Pranav Rastogi](#), and [Valeriy Novytskyy](#)

Here are set up instructions for some other common OAuth providers. Third-party NuGet packages such as the ones maintained by [aspnet-contrib](#) can be used to complement authentication providers implemented by the ASP.NET Core team.

- Set up **LinkedIn** sign in: <https://www.linkedin.com/developer/apps>. See [official steps](#).
- Set up **Instagram** sign in: <https://www.instagram.com/developer/register/>. See [official steps](#).
- Set up **Reddit** sign in: <https://www.reddit.com/login?dest=https%3A%2F%2Fwww.reddit.com%2Fprefs%2Fapps>. See [official steps](#).
- Set up **Github** sign in: https://github.com/login?return_to=https%3A%2F%2Fgithub.com%2Fsettings%2Fapplications%2Fnew. See [official steps](#).
- Set up **Yahoo** sign in: <https://login.yahoo.com/config/login?src=devnet&.done=http%3A%2F%2Fdeveloper.yahoo.com%2Fapps%2Fcreate%2F>. See [official steps](#).
- Set up **Tumblr** sign in: <https://www.tumblr.com/oauth/apps>. See [official steps](#).
- Set up **Pinterest** sign in: <https://www.pinterest.com/login/?next=http%3A%2F%2Fdevsite%2Fapps%2F>. See [official steps](#).
- Set up **Pocket** sign in: <https://getpocket.com/developer/apps/new>. See [official steps](#).
- Set up **Flickr** sign in: <https://www.flickr.com/services/apps/create>. See [official steps](#).
- Set up **Dribbble** sign in: <https://dribbble.com/signup>. See [official steps](#).
- Set up **Vimeo** sign in: <https://vimeo.com/join>. See [official steps](#).
- Set up **SoundCloud** sign in: <https://soundcloud.com/you/apps/new>. See [official steps](#).
- Set up **VK** sign in: <https://vk.com/apps?act=manage>. See [official steps](#).

Account confirmation and password recovery in ASP.NET Core

12/2/2017 • 12 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Joe Audette](#)

This tutorial shows you how to build an ASP.NET Core app with email confirmation and password reset.

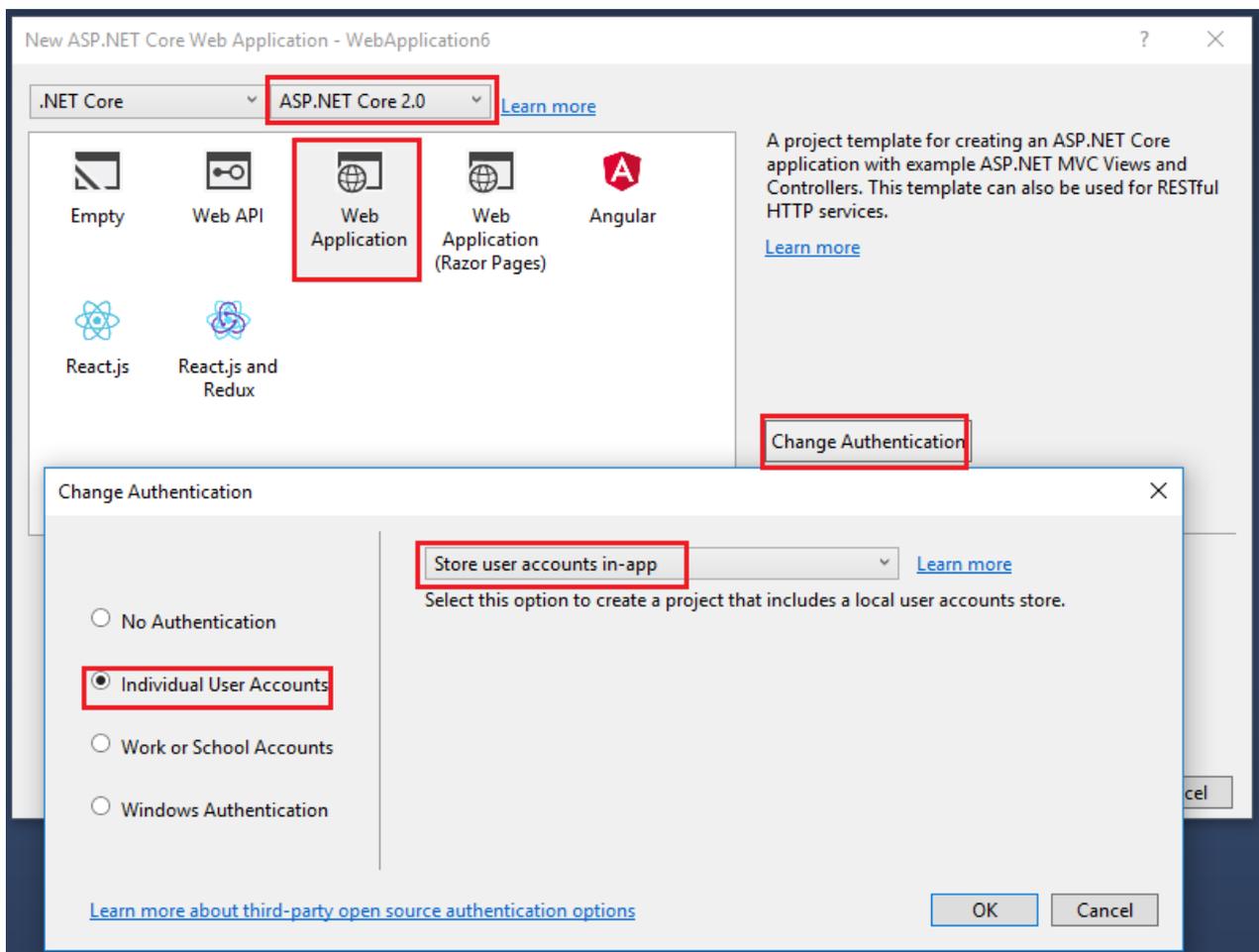
Create a New ASP.NET Core Project

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

This step applies to Visual Studio on Windows. See the next section for CLI instructions.

The tutorial requires Visual Studio 2017 Preview 2 or later.

- In Visual Studio, create a New Web Application Project.
- Select **ASP.NET Core 2.0**. The following image show **.NET Core** selected, but you can select **.NET Framework**.
- Select **Change Authentication** and set to **Individual User Accounts**.
- Keep the default **Store user accounts in-app**.



.NET Core CLI project creation for macOS and Linux

If you're using the CLI or SQLite, run the following in a command window:

```
dotnet new mvc --auth Individual
```

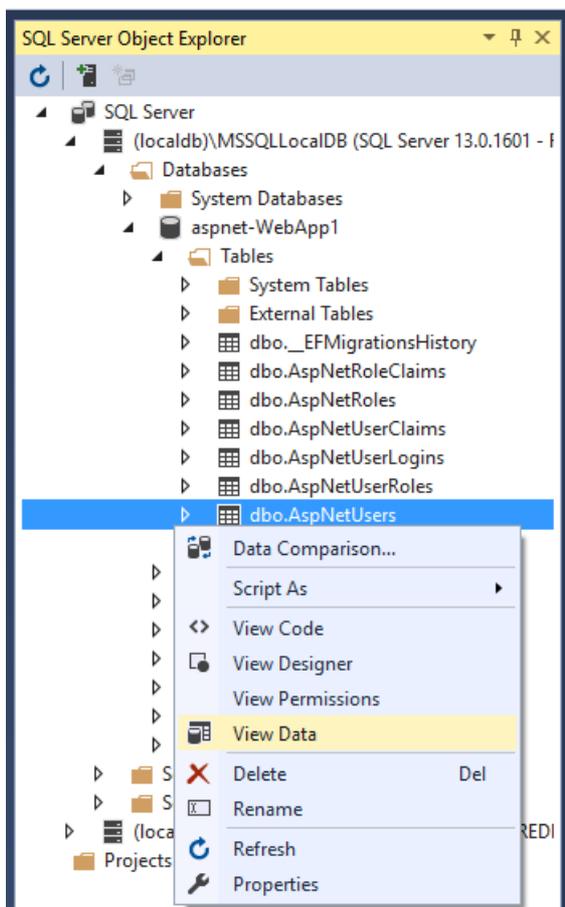
- `--auth Individual` specifies the Individual User Accounts template.
- On Windows, add the `-uId` option. The `-uId` option creates a LocalDB connection string rather than a SQLite DB.
- Run `new mvc --help` to get help on this command.

Test new user registration

Run the app, select the **Register** link, and register a user. Follow the instructions to run Entity Framework Core migrations. At this point, the only validation on the email is with the `[EmailAddress]` attribute. After you submit the registration, you are logged into the app. Later in the tutorial, we'll change this so new users cannot log in until their email has been validated.

View the Identity database

- [SQL Server](#)
- [SQLite](#)
- From the **View** menu, select **SQL Server Object Explorer** (SSOX).
- Navigate to **(localdb)\MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



Note the `EmailConfirmed` field is `False`.

You might want to use this email again in the next step when the app sends a confirmation email. Right-click on the row and select **Delete**. Deleting the email alias now will make it easier in the following steps.

Require SSL and setup IIS Express for SSL

See [Enforcing SSL](#).

Require email confirmation

It's a best practice to confirm the email of a new user registration to verify they are not impersonating someone else (that is, they haven't registered with someone else's email). Suppose you had a discussion forum, and you wanted to prevent "yli@example.com" from registering as "nolivetto@contoso.com." Without email confirmation, "nolivetto@contoso.com" could get unwanted email from your app. Suppose the user accidentally registered as "ylo@example.com" and hadn't noticed the misspelling of "yli," they wouldn't be able to use password recovery because the app doesn't have their correct email. Email confirmation provides only limited protection from bots and doesn't provide protection from determined spammers who have many working email aliases they can use to register.

You generally want to prevent new users from posting any data to your web site before they have a confirmed email.

Update `ConfigureServices` to require a confirmed email:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>(config =>
        {
            config.SignIn.RequireConfirmedEmail = true;
        })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();

    services.Configure<AuthMessageSenderOptions>(Configuration);
}
```

```
config.SignIn.RequireConfirmedEmail = true;
```

The preceding line prevents registered users from being logged in until their email is confirmed. However, that line does not prevent new users from being logged in after they register. The default code logs in a user after they register. Once they log out, they won't be able to log in again until they register. Later in the tutorial we'll change the code so newly registered user are **not** logged in.

Configure email provider

In this tutorial, SendGrid is used to send email. You need a SendGrid account and key to send email. You can use other email providers. ASP.NET Core 2.x includes `System.Net.Mail`, which allows you to send email from your app. We recommend you use SendGrid or another email service to send email.

The [Options pattern](#) is used to access the user account and key settings. For more information, see [configuration](#).

Create a class to fetch the secure email key. For this sample, the `AuthMessageSenderOptions` class is created in the `Services/AuthMessageSenderOptions.cs` file.

```
public class AuthMessageSenderOptions
{
    public string SendGridUser { get; set; }
    public string SendGridKey { get; set; }
}
```

Set the `SendGridUser` and `SendGridKey` with the [secret-manager tool](#). For example:

```
C:\WebApp1\src\WebApp1>dotnet user-secrets set SendGridUser RickAndMSFT
info: Successfully saved SendGridUser = RickAndMSFT to the secret store.
```

On Windows, Secret Manager stores your keys/value pairs in a `secrets.json` file in the `%APPDATA%/Microsoft/UserSecrets/` directory.

The contents of the `secrets.json` file are not encrypted. The `secrets.json` file is shown below (the `SendGridKey` value has been removed.)

```
{
  "SendGridUser": "RickAndMSFT",
  "SendGridKey": "<key removed>"
}
```

Configure startup to use `AuthMessageSenderOptions`

Add `AuthMessageSenderOptions` to the service container at the end of the `ConfigureServices` method in the `Startup.cs` file:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>(config =>
        {
            config.SignIn.RequireConfirmedEmail = true;
        })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();

    services.Configure<AuthMessageSenderOptions>(Configuration);
}
```

Configure the `AuthMessageSender` class

This tutorial shows how to add email notifications through [SendGrid](#), but you can send email using SMTP and other mechanisms.

- Install the `SendGrid` NuGet package. From the Package Manager Console, enter the following the following command:

```
Install-Package SendGrid
```

- See [Get Started with SendGrid for Free](#) to register for a free SendGrid account.

Configure SendGrid

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)
- Add code in *Services/EmailSender.cs* similar to the following to configure SendGrid:

```
using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;
using System.Threading.Tasks;

namespace WebPW.Services
{
    public class EmailSender : IEmailSender
    {
        public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public AuthMessageSenderOptions Options { get; } //set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            return Execute(Options.SendGridKey, subject, message, email);
        }

        public Task Execute(string apiKey, string subject, string message, string email)
        {
            var client = new SendGridClient(apiKey);
            var msg = new SendGridMessage()
            {
                From = new EmailAddress("Joe@contoso.com", "Joe Smith"),
                Subject = subject,
                PlainTextContent = message,
                HtmlContent = message
            };
            msg.AddTo(new EmailAddress(email));
            return client.SendEmailAsync(msg);
        }
    }
}
```

Enable account confirmation and password recovery

The template has the code for account confirmation and password recovery. Find the `[HttpPost] Register` method in the *AccountController.cs* file.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Prevent newly registered users from being automatically logged on by commenting out the following line:

```
await _signInManager.SignInAsync(user, isPersistent: false);
```

The complete method is shown with the changed line highlighted:

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.EmailConfirmationLink(user.Id, code, Request.Scheme);
            await _emailSender.SendEmailConfirmationAsync(model.Email, callbackUrl);

            // await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation("User created a new account with password.");
            return RedirectToLocal(returnUrl);
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

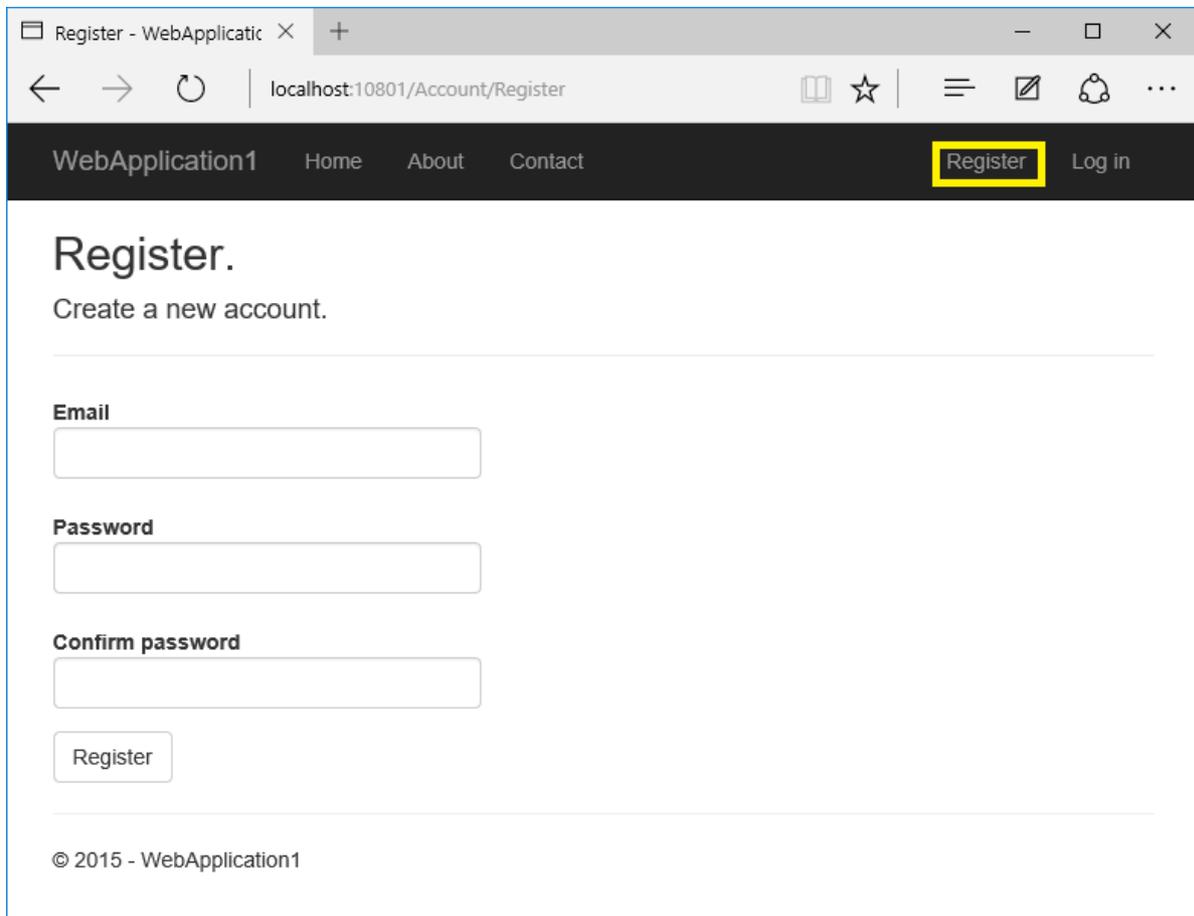
```

Note: The previous code will fail if you implement `IEmailSender` and send a plain text email. See [this issue](#) for more information and a workaround.

Register, confirm email, and reset password

Run the web app, and test the account confirmation and password recovery flow.

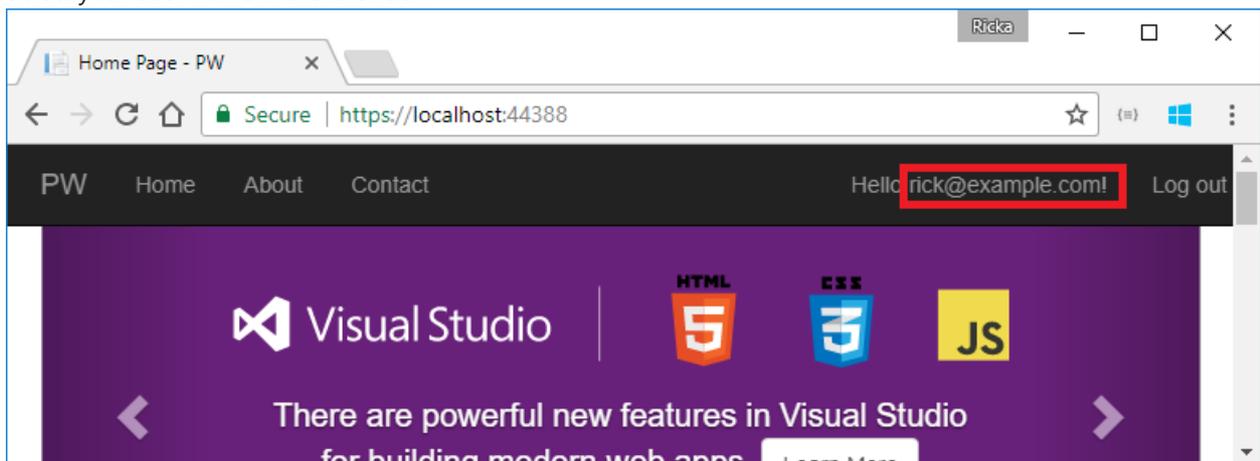
- Run the app and register a new user



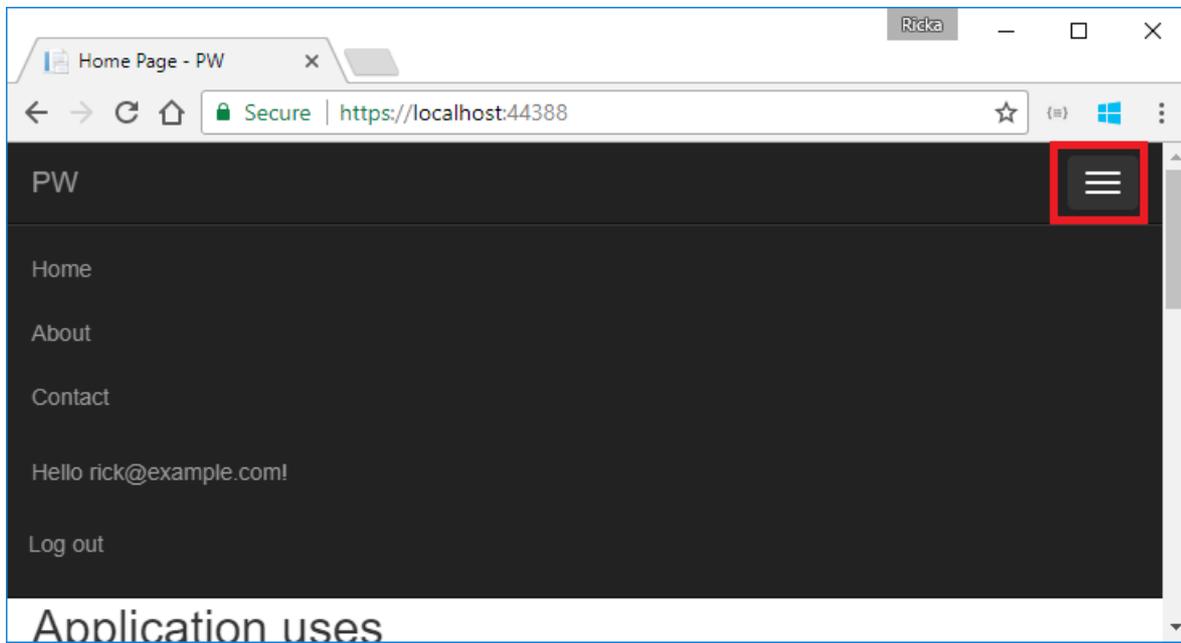
- Check your email for the account confirmation link. See [Debug email](#) if you don't get the email.
- Click the link to confirm your email.
- Log in with your email and password.
- Log off.

View the manage page

Select your user name in the browser:

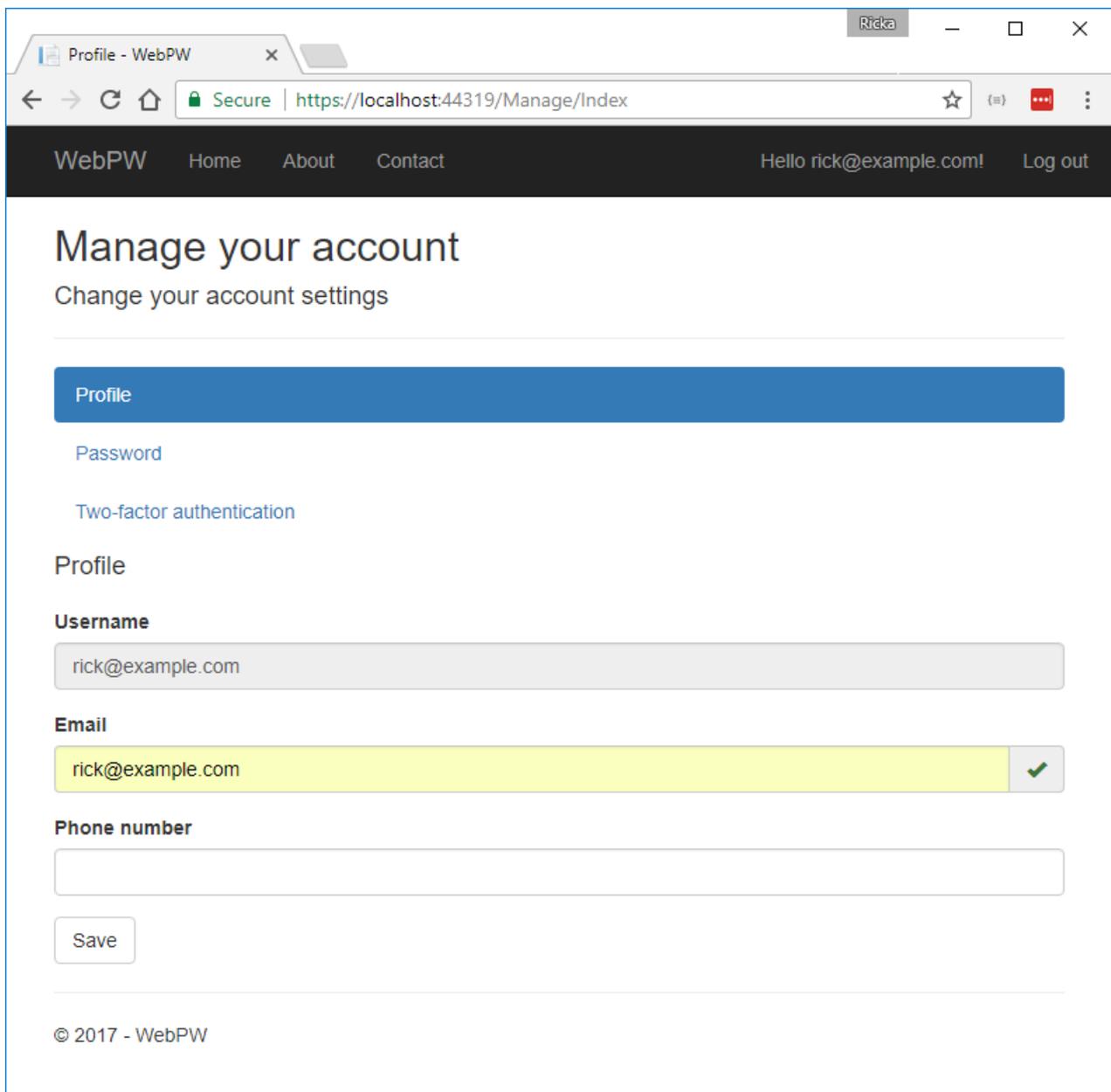


You might need to expand the navbar to see user name.



- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The manage page is displayed with the **Profile** tab selected. The **Email** shows a check box indicating the email has been confirmed.



Test password reset

- If you're logged in, select **Logout**.
- Select the **Log in** link and select the **Forgot your password?** link.
- Enter the email you used to register the account.
- An email with a link to reset your password will be sent. Check your email and click the link to reset your password. After your password has been successfully reset, you can login with your email and new password.

Debug email

If you can't get email working:

- Review the [Email Activity](#) page.
- Check your spam folder.
- Try another email alias on a different email provider (Microsoft, Yahoo, Gmail, etc.)
- Create a [console app to send email](#).
- Try sending to different email accounts.

Note: A security best practice is to not use production secrets in test and development. If you publish the app to Azure, you can set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

Prevent login at registration

With the current templates, once a user completes the registration form, they are logged in (authenticated). You generally want to confirm their email before logging them in. In the section below, we will modify the code to require new users have a confirmed email before they are logged in. Update the `[HttpPost] Login` action in the `AccountController.cs` file with the following highlighted changes.

```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // Require the user to have a confirmed email before they can log on.
        var user = await _userManager.FindByEmailAsync(model.Email);
        if (user != null)
        {
            if (!await _userManager.IsEmailConfirmedAsync(user))
            {
                ModelState.AddModelError(string.Empty,
                    "You must have a confirmed email to log in.");
                return View(model);
            }
        }
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email,
            model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode),
                new { ReturnUrl = returnUrl, RememberMe = model.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

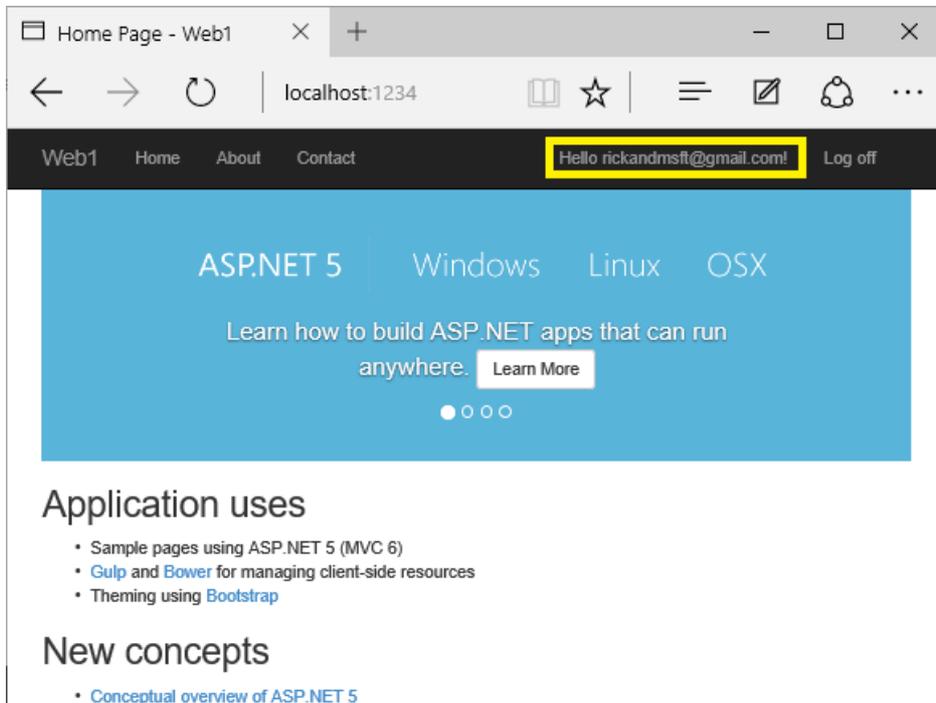
Note: A security best practice is to not use production secrets in test and development. If you publish the app to Azure, you can set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

Combine social and local login accounts

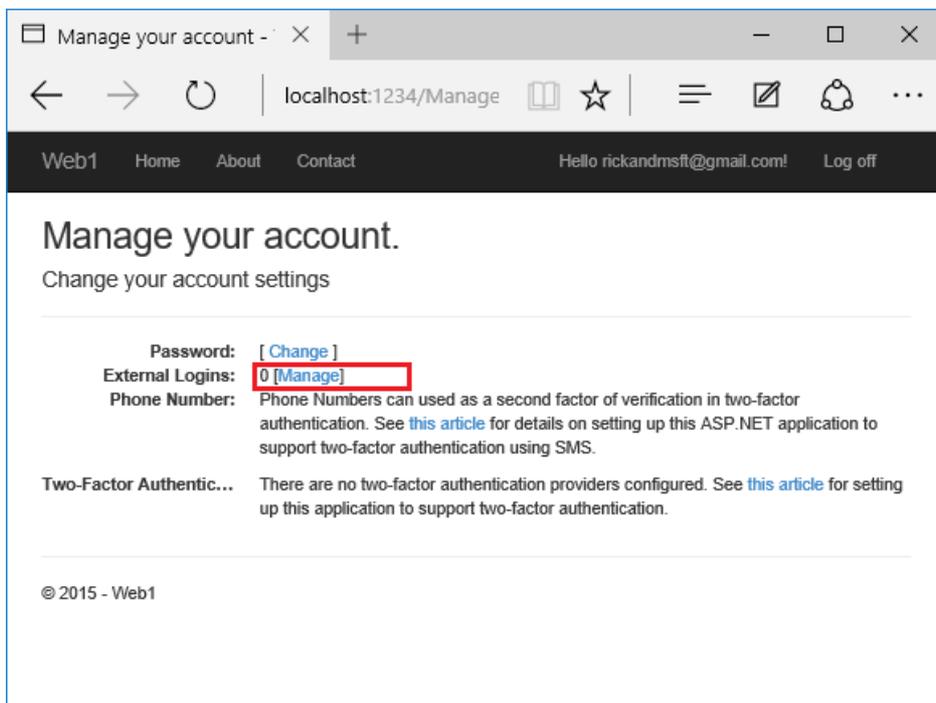
Note: This section applies only to ASP.NET Core 1.x. For ASP.NET Core 2.x, see [this](#) issue.

To complete this section, you must first enable an external authentication provider. See [Enabling authentication using Facebook, Google and other external providers](#).

You can combine local and social accounts by clicking on your email link. In the following sequence, "RickAndMSFT@gmail.com" is first created as a local login; however, you can create the account as a social login first, then add a local login.



Click on the **Manage** link. Note the 0 external (social logins) associated with this account.



Click the link to another login service and accept the app requests. In the image below, Facebook is the external authentication provider:

Manage your external logins.

Registered Logins

Facebook

© 2017 - WebApplication2

The two accounts have been combined. You will be able to log on with either account. You might want your users to add local accounts in case their social log in authentication service is down, or more likely they have lost access to their social account.

Enabling QR Code generation for authenticator apps in ASP.NET Core

10/13/2017 • 2 min to read • [Edit Online](#)

Note: This topic applies to ASP.NET Core 2.x

ASP.NET Core ships with support for authenticator applications for individual authentication. Two factor authentication (2FA) authenticator apps, using a Time-based One-time Password Algorithm (TOTP), are the industry recommended approach for 2FA. 2FA using TOTP is preferred to SMS 2FA. An authenticator app provides a 6 to 8 digit code which users must enter after confirming their username and password. Typically an authenticator app is installed on a smart phone.

The ASP.NET Core web app templates support authenticators, but do not provide support for QRCode generation. QRCode generators ease the setup of 2FA. This document will guide you through adding [QR Code](#) generation to the 2FA configuration page.

Adding QR Codes to the 2FA configuration page

These instructions use *qrcode.js* from the <https://davidshimjs.github.io/qrcodejs/> repo.

- Download the [qrcode.js javascript library](#) to the `wwwroot\lib` folder in your project.
- In *Pages\Account\Manage\EnableAuthenticator.cshtml* (Razor Pages) or *Views\Manage\EnableAuthenticator.cshtml* (MVC), locate the `Scripts` section at the end of the file:

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

- Update the `Scripts` section to add a reference to the `qrcodejs` library you added and a call to generate the QR Code. It should look as follows:

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")

    <script type="text/javascript" src="~/lib/qrcode.js"></script>
    <script type="text/javascript">
        new QRCode(document.getElementById("qrCode"),
            {
                text: "@Html.Raw(Model.AuthenticatorUri)",
                width: 150,
                height: 150
            });
    </script>
}
```

- Delete the paragraph which links you to these instructions.

Run your app and ensure that you can scan the QR code and validate the code the authenticator proves.

Change the site name in the QR Code

The site name in the QR Code is taken from the project name you choose when initially creating your project. You can change it by looking for the `GenerateQrCodeUri(string email, string unformattedKey)` method in the `Pages\Account\Manage\EnableAuthenticator.cshtml.cs` (Razor Pages) file or the `Controllers\ManageController.cs` (MVC) file.

The default code from the template looks as follows:

```
private string GenerateQrCodeUri(string email, string unformattedKey)
{
    return string.Format(
        AuthenticatorUriFormat,
        _urlEncoder.Encode("Razor Pages"),
        _urlEncoder.Encode(email),
        unformattedKey);
}
```

The second parameter in the call to `string.Format` is your site name, taken from your solution name. It can be changed to any value, but it must always be URL encoded.

Using a different QR Code library

You can replace the QR Code library with your preferred library. The HTML contains a `qrCode` element into which you can place a QR Code by whatever mechanism your library provides.

The correctly formatted URL for the QR Code is available in the:

- `AuthenticatorUri` property of the model.
- `data-url` property in the `qrCodeData` element.

Use `@Html.Raw` to access the model property in a view (otherwise the ampersands in the url will be double encoded and the label parameter of the QR Code will be ignored).

TOTP client and server time skew

TOTP authentication depends on both the server and authenticator device having an accurate time. Tokens only last for 30 seconds. If TOTP 2FA logins are failing, check that the server time is accurate, and preferably synchronized to an accurate NTP service.

Two-factor authentication with SMS

11/29/2017 • 5 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Swiss-Devs](#)

This tutorial applies to ASP.NET Core 1.x only. See [Enabling QR Code generation for authenticator apps in ASP.NET Core](#) for ASP.NET Core 2.0 and later.

This tutorial shows how to set up two-factor authentication (2FA) using SMS. Instructions are given for [twilio](#) and [ASPSMS](#), but you can use any other SMS provider. We recommend you complete [Account Confirmation and Password Recovery](#) before starting this tutorial.

View the [completed sample](#). [How to download](#).

Create a new ASP.NET Core project

Create a new ASP.NET Core web app named `Web2FA` with individual user accounts. Follow the instructions in [Enforcing SSL in an ASP.NET Core app](#) to set up and require SSL.

Create an SMS account

Create an SMS account, for example, from [twilio](#) or [ASPSMS](#). Record the authentication credentials (for twilio: accountSid and authToken, for ASPSMS: Userkey and Password).

Figuring out SMS Provider credentials

Twilio:

From the Dashboard tab of your Twilio account, copy the **Account SID** and **Auth token**.

ASPSMS:

From your account settings, navigate to **Userkey** and copy it together with your **Password**.

We will later store these values in with the secret-manager tool within the keys `SMSAccountIdentification` and `SMSAccountPassword`.

Specifying SenderID / Originator

Twilio:

From the Numbers tab, copy your Twilio **phone number**.

ASPSMS:

Within the Unlock Originators Menu, unlock one or more Originators or choose an alphanumeric Originator (Not supported by all networks).

We will later store this value with the secret-manager tool within the key `SMSAccountFrom`.

Provide credentials for the SMS service

We'll use the [Options pattern](#) to access the user account and key settings.

- Create a class to fetch the secure SMS key. For this sample, the `SMSOptions` class is created in the `Services/SMSOptions.cs` file.

```
namespace Web2FA.Services
{
    public class SMSOptions
    {
        public string SMSAccountIdentification { get; set; }
        public string SMSAccountPassword { get; set; }
        public string SMSAccountFrom { get; set; }
    }
}
```

Set the `SMSAccountIdentification`, `SMSAccountPassword` and `SMSAccountFrom` with the [secret-manager tool](#). For example:

```
C:/Web2FA/src/WebApp1>dotnet user-secrets set SMSAccountIdentification 12345
info: Successfully saved SMSAccountIdentification = 12345 to the secret store.
```

- Add the NuGet package for the SMS provider. From the Package Manager Console (PMC) run:

Twilio:

```
Install-Package Twilio
```

ASPSMS:

```
Install-Package ASPSMS
```

- Add code in the `Services/MessageServices.cs` file to enable SMS. Use either the Twilio or the ASPSMS section:

Twilio:

```

using Microsoft.Extensions.Options;
using System.Threading.Tasks;
using Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

namespace Web2FA.Services
{
    // This class is used by the application to send Email and SMS
    // when you turn on two-factor authentication in ASP.NET Identity.
    // For more details see this link https://go.microsoft.com/fwlink/?LinkID=532713
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public AuthMessageSender(IOptions<SMSOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public SMSOptions Options { get; } // set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            // Plug in your email service here to send an email.
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            // Plug in your SMS service here to send a text message.
            // Your Account SID from twilio.com/console
            var accountSid = Options.SMSAccountIdentification;
            // Your Auth Token from twilio.com/console
            var authToken = Options.SMSAccountPassword;

            TwilioClient.Init(accountSid, authToken);

            return MessageResource.CreateAsync(
                to: new PhoneNumber(number),
                from: new PhoneNumber(Options.SMSAccountFrom),
                body: message);
        }
    }
}

```

ASPSMS:

```

using Microsoft.Extensions.Options;
using System.Threading.Tasks;

namespace Web2FA.Services
{
    // This class is used by the application to send Email and SMS
    // when you turn on two-factor authentication in ASP.NET Identity.
    // For more details see this link https://go.microsoft.com/fwlink/?LinkID=532713
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public AuthMessageSender(IOptions<SMSOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public SMSOptions Options { get; } // set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            // Plug in your email service here to send an email.
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            ASPSMS.SMS SMSSender = new ASPSMS.SMS();

            SMSSender.Userkey = Options.SMSAccountIdentification;
            SMSSender.Password = Options.SMSAccountPassword;
            SMSSender.Originator = Options.SMSAccountFrom;

            SMSSender.AddRecipient(number);
            SMSSender.MessageData = message;

            SMSSender.SendTextSMS();

            return Task.FromResult(0);
        }
    }
}

```

Configure startup to use `SMSOptions`

Add `SMSOptions` to the service container in the `ConfigureServices` method in the *Startup.cs*:

```

// Add application services.
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
services.Configure<SMSOptions>(Configuration);
}

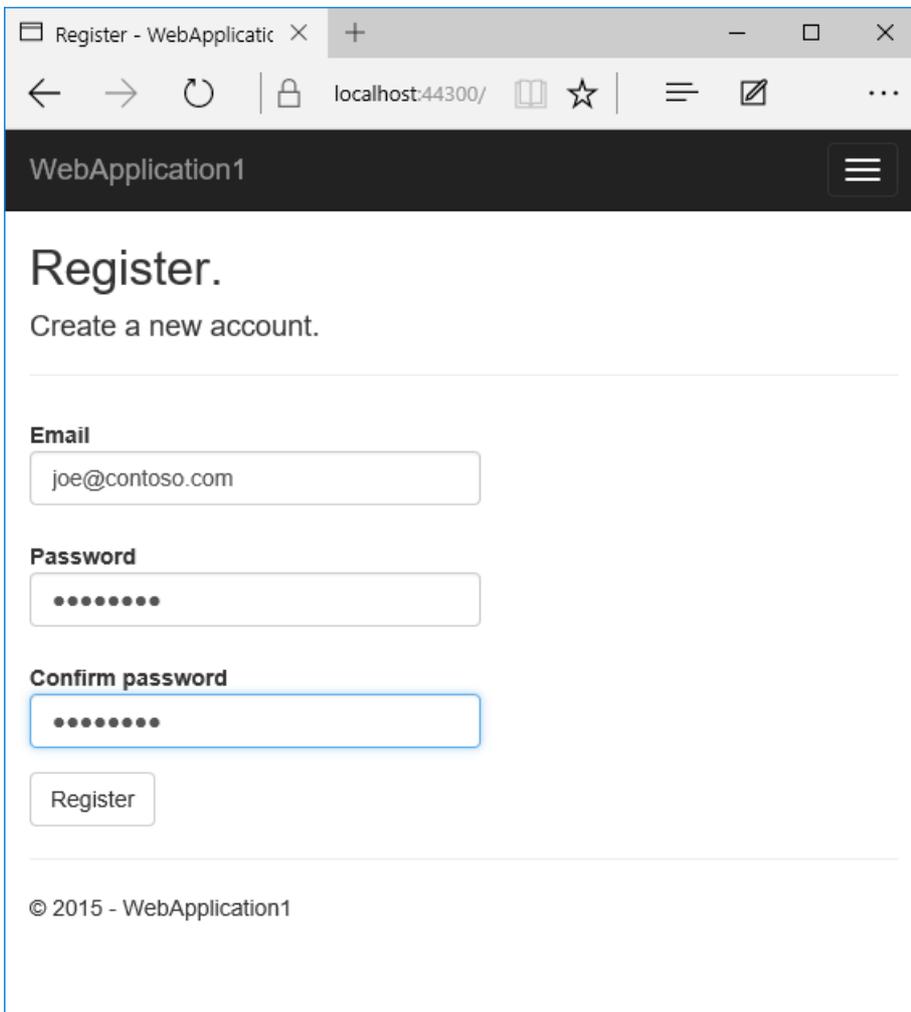
```

Enable two-factor authentication

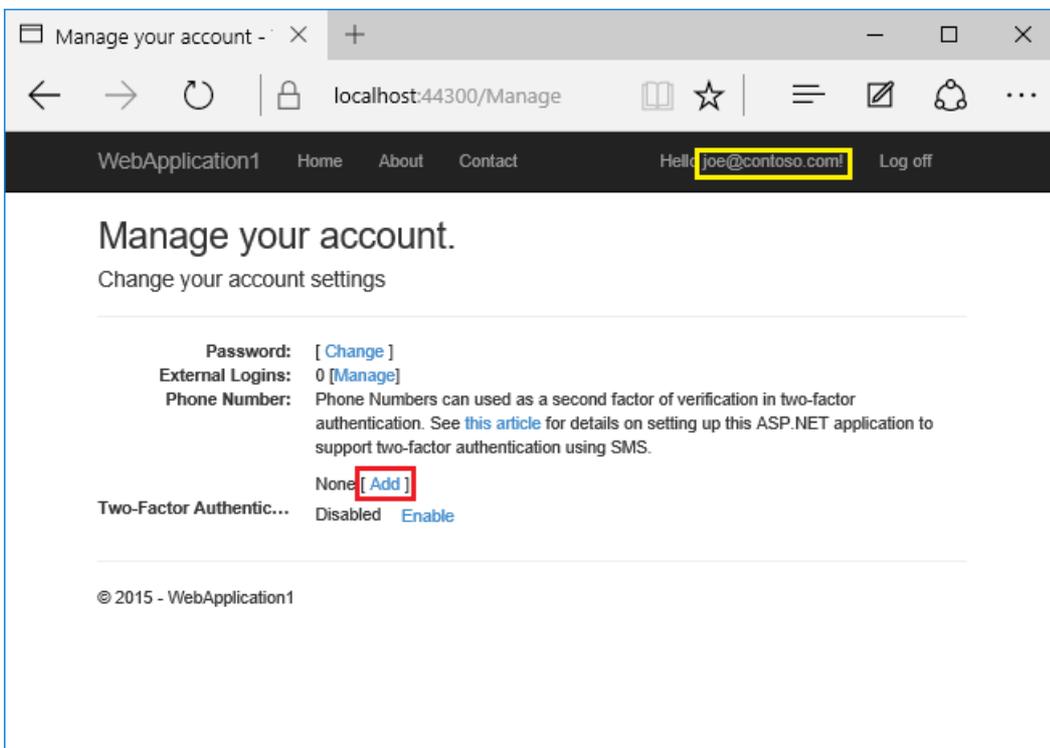
Open the *Views/Manage/Index.cshtml* Razor view file and remove the comment characters (so no markup is commented out).

Log in with two-factor authentication

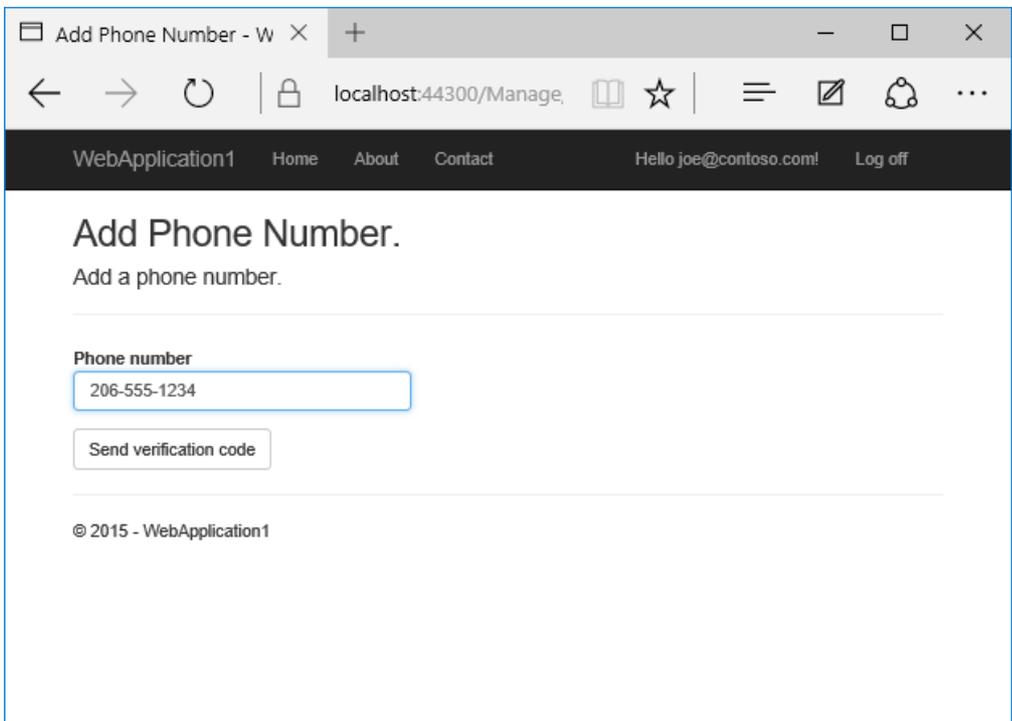
- Run the app and register a new user



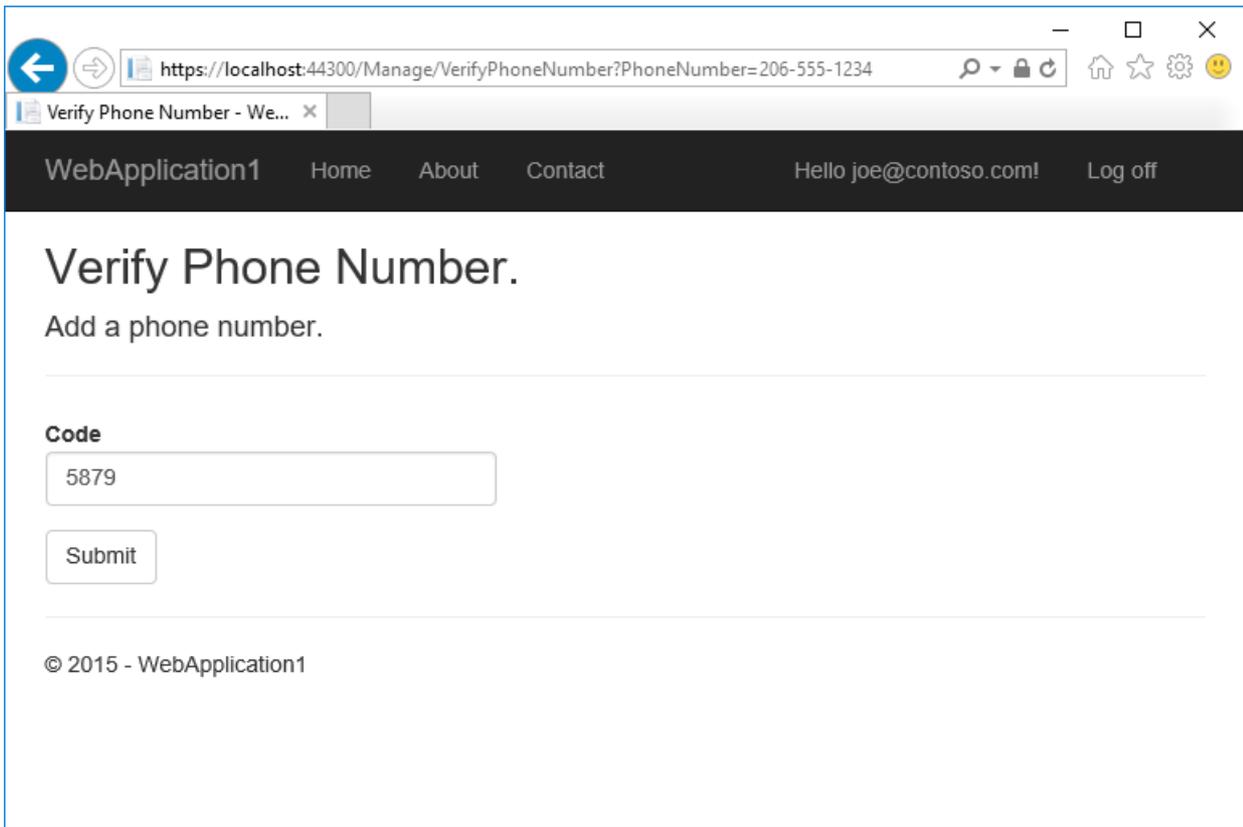
- Tap on your user name, which activates the `Index` action method in Manage controller. Then tap the phone number **Add** link.



- Add a phone number that will receive the verification code, and tap **Send verification code**.



- You will get a text message with the verification code. Enter it and tap **Submit**



If you don't get a text message, see twilio log page.

- The Manage view shows your phone number was added successfully.

WebApplication1 Home About Contact Hello joe@contoso.com! Log off

Manage your account.

Your phone number was added.

Change your account settings

Password: [[Change](#)]

External Logins: 0 [[Manage](#)]

Phone Number: Phone Numbers can be used as a second factor of verification in two-factor authentication. See [this article](#) for details on setting up this ASP.NET application to support two-factor authentication using SMS.

206-555-1234 [[Change](#) | [Remove](#)]

Two-Factor Authentic... Disabled [Enable](#)

© 2015 - WebApplication1

- Tap **Enable** to enable two-factor authentication.

WebApplication1 Home About Contact Hello joe@contoso.com! Log off

Manage your account.

Your phone number was added.

Change your account settings

Password: [[Change](#)]

External Logins: 0 [[Manage](#)]

Phone Number: Phone Numbers can be used as a second factor of verification in two-factor authentication. See [this article](#) for details on setting up this ASP.NET application to support two-factor authentication using SMS.

206-555-1234 [[Change](#) | [Remove](#)]

Two-Factor Authentic... Disabled [Enable](#)

© 2015 - WebApplication1

Test two-factor authentication

- Log off.
- Log in.
- The user account has enabled two-factor authentication, so you have to provide the second factor of authentication. In this tutorial you have enabled phone verification. The built in templates also allow you to set up email as the second factor. You can set up additional second factors for authentication such as QR

codes. Tap **Submit**.

WebApplication1 Home About Contact Register Log in

Send Verification Code.

Select Two-Factor Authentication Provider:

© 2015 - WebApplication1

- Enter the code you get in the SMS message.
- Clicking on the **Remember this browser** check box will exempt you from needing to use 2FA to log on when using the same device and browser. Enabling 2FA and clicking on **Remember this browser** will provide you with strong 2FA protection from malicious users trying to access your account, as long as they don't have access to your device. You can do this on any private device you regularly use. By setting **Remember this browser**, you get the added security of 2FA from devices you don't regularly use, and you get the convenience on not having to go through 2FA on your own devices.

WebApplication1 Home About Contact Register Log in

Verify.

Code

Remember this browser?

© 2015 - WebApplication1

Account lockout for protecting against brute force attacks

We recommend you use account lockout with 2FA. Once a user logs in (through a local account or social account), each failed attempt at 2FA is stored, and if the maximum attempts (default is 5) is reached, the user is locked out for five minutes (you can set the lock out time with `DefaultAccountLockoutTimeSpan`). The following configures Account to be locked out for 10 minutes after 10 failed attempts.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    services.Configure<IdentityOptions>(options =>
    {
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);
        options.Lockout.MaxFailedAccessAttempts = 10;
    });

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSSptions>(Configuration);
}
```

Using Cookie Authentication without ASP.NET Core Identity

1/8/2018 • 14 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Luke Latham](#)

As you've seen in the earlier authentication topics, [ASP.NET Core Identity](#) is a complete, full-featured authentication provider for creating and maintaining logins. However, you may want to use your own custom authentication logic with cookie-based authentication at times. You can use cookie-based authentication as a standalone authentication provider without ASP.NET Core Identity.

[View or download sample code \(how to download\)](#)

For information on migrating cookie-based authentication from ASP.NET Core 1.x to 2.0, see [Migrating Authentication and Identity to ASP.NET Core 2.0 topic \(Cookie-based Authentication\)](#).

Configuration

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

If you aren't using the [Microsoft.AspNetCore.All metapackage](#), install version 2.0+ of the [Microsoft.AspNetCore.Authentication.Cookies](#) NuGet package.

In the `ConfigureServices` method, create the Authentication Middleware service with the `AddAuthentication` and `AddCookie` methods:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

`AuthenticationScheme` passed to `AddAuthentication` sets the default authentication scheme for the app. `AuthenticationScheme` is useful when there are multiple instances of cookie authentication and you want to [authorize with a specific scheme](#). Setting the `AuthenticationScheme` to `CookieAuthenticationDefaults.AuthenticationScheme` provides a value of "Cookies" for the scheme. You can supply any string value that distinguishes the scheme.

In the `Configure` method, use the `UseAuthentication` method to invoke the Authentication Middleware that sets the `HttpContext.User` property. Call the `UseAuthentication` method before calling `UseMvcWithDefaultRoute` or `UseMvc` :

```
app.UseAuthentication();
```

AddCookie Options

The [CookieAuthenticationOptions](#) class is used to configure the authentication provider options.

OPTION	DESCRIPTION
--------	-------------

OPTION	DESCRIPTION
AccessDeniedPath	Provides the path to supply with a 302 Found (URL redirect) when triggered by <code>HttpContext.ForbidAsync</code> . The default value is <code>/Account/AccessDenied</code> .
ClaimsIssuer	The issuer to use for the <code>Issuer</code> property on any claims created by the cookie authentication service.
Cookie.Domain	The domain name where the cookie is served. By default, this is the host name of the request. The browser only sends the cookie in requests to a matching host name. You may wish to adjust this to have cookies available to any host in your domain. For example, setting the cookie domain to <code>.contoso.com</code> makes it available to <code>contoso.com</code> , <code>www.contoso.com</code> , and <code>staging.www.contoso.com</code> .
Cookie.Expiration	Gets or sets the lifespan of a cookie. Currently, this option no-ops and will become obsolete in ASP.NET Core 2.1+. Use the <code>ExpireTimeSpan</code> option to set cookie expiration. For more information, see Clarify behavior of CookieAuthenticationOptions.Cookie.Expiration .
Cookie.HttpOnly	A flag indicating if the cookie should be accessible only to servers. Changing this value to <code>false</code> permits client-side scripts to access the cookie and may open your app to cookie theft should your app have a Cross-site scripting (XSS) vulnerability. The default value is <code>true</code> .
Cookie.Name	Sets the name of the cookie.
Cookie.Path	Used to isolate apps running on the same host name. If you have an app running at <code>/app1</code> and want to restrict cookies to that app, set the <code>CookiePath</code> property to <code>/app1</code> . By doing so, the cookie is only available on requests to <code>/app1</code> and any app underneath it.
Cookie.SameSite	Indicates whether the browser should allow the cookie to be attached to same-site requests only (<code>SameSiteMode.Strict</code>) or cross-site requests using safe HTTP methods and same-site requests (<code>SameSiteMode.Lax</code>). When set to <code>SameSiteMode.None</code> , the cookie header value isn't set. Note that Cookie Policy Middleware might overwrite the value that you provide. To support OAuth authentication, the default value is <code>SameSiteMode.Lax</code> . For more information, see OAuth authentication broken due to SameSite cookie policy .
Cookie.SecurePolicy	A flag indicating if the cookie created should be limited to HTTPS (<code>CookieSecurePolicy.Always</code>), HTTP or HTTPS (<code>CookieSecurePolicy.None</code>), or the same protocol as the request (<code>CookieSecurePolicy.SameAsRequest</code>). The default value is <code>CookieSecurePolicy.SameAsRequest</code> .

OPTION	DESCRIPTION
DataProtectionProvider	Sets the <code>DataProtectionProvider</code> that's used to create the default <code>TicketDataFormat</code> . If the <code>TicketDataFormat</code> property is set, the <code>DataProtectionProvider</code> option isn't used. If not provided, the app's default data protection provider is used.
Events	The handler calls methods on the provider that give the app control at certain processing points. If <code>Events</code> aren't provided, a default instance is supplied that does nothing when the methods are called.
EventsType	Used as the service type to get the <code>Events</code> instance instead of the property.
ExpireTimeSpan	The <code>TimeSpan</code> after which the authentication ticket stored inside the cookie expires. <code>ExpireTimeSpan</code> is added to the current time to create the expiration time for the ticket. The <code>ExpiredTimeSpan</code> value always goes into the encrypted AuthTicket verified by the server. It may also go into the Set-Cookie header, but only if <code>IsPersistent</code> is set. To set <code>IsPersistent</code> to <code>true</code> , configure the AuthenticationProperties passed to <code>SignInAsync</code> . The default value of <code>ExpireTimeSpan</code> is 14 days.
LoginPath	Provides the path to supply with a 302 Found (URL redirect) when triggered by <code>HttpContext.ChallengeAsync</code> . The current URL that generated the 401 is added to the <code>LoginPath</code> as a query string parameter named by the <code>ReturnUrlParameter</code> . Once a request to the <code>LoginPath</code> grants a new sign-in identity, the <code>ReturnUrlParameter</code> value is used to redirect the browser back to the URL that caused the original unauthorized status code. The default value is <code>/Account/Login</code> .
LogoutPath	If the <code>LogoutPath</code> is provided to the handler, then a request to that path redirects based on the value of the <code>ReturnUrlParameter</code> . The default value is <code>/Account/Logout</code> .
ReturnUrlParameter	Determines the name of the query string parameter that's appended by the handler for a 302 Found (URL redirect) response. <code>ReturnUrlParameter</code> is used when a request arrives on the <code>LoginPath</code> or <code>LogoutPath</code> to return the browser to the original URL after the login or logout action is performed. The default value is <code>ReturnUrl</code> .
SessionStore	An optional container used to store identity across requests. When used, only a session identifier is sent to the client. <code>SessionStore</code> can be used to mitigate potential problems with large identities.

OPTION	DESCRIPTION
SlidingExpiration	A flag indicating if a new cookie with an updated expiration time should be issued dynamically. This can happen on any request where the current cookie expiration period is more than 50% expired. The new expiration date is moved forward to be the current date plus the <code>ExpireTimespan</code> . An absolute cookie expiration time can be set by using the <code>AuthenticationProperties</code> class when calling <code>SignInAsync</code> . An absolute expiration time can improve the security of your app by limiting the amount of time that the authentication cookie is valid. The default value is <code>true</code> .
TicketDataFormat	The <code>TicketDataFormat</code> is used to protect and unprotect the identity and other properties that are stored in the cookie value. If not provided, a <code>TicketDataFormat</code> is created using the DataProtectionProvider .
Validate	Method that checks that the options are valid.

Set `CookieAuthenticationOptions` in the service configuration for authentication in the `ConfigureServices` method:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        ...
    });
```

Cookie Policy Middleware

[Cookie Policy Middleware](#) enables cookie policy capabilities in an app. Adding the middleware to the app processing pipeline is order sensitive; it only affects components registered after it in the pipeline.

```
app.UseCookiePolicy(cookiePolicyOptions);
```

The `CookiePolicyOptions` provided to the Cookie Policy Middleware allow you to control global characteristics of cookie processing and hook into cookie processing handlers when cookies are appended or deleted.

PROPERTY	DESCRIPTION
HttpOnly	Affects whether cookies must be HttpOnly, which is a flag indicating if the cookie should be accessible only to servers. The default value is <code>HttpOnlyPolicy.None</code> .
MinimumSameSitePolicy	Affects the cookie's same-site attribute (see below). The default value is <code>SameSiteMode.Lax</code> . This option is available for ASP.NET Core 2.0+.
OnAppendCookie	Called when a cookie is appended.
OnDeleteCookie	Called when a cookie is deleted.
Secure	Affects whether cookies must be Secure. The default value is <code>CookieSecurePolicy.None</code> .

MinimumSameSitePolicy (ASP.NET Core 2.0+ only)

The default `MinimumSameSitePolicy` value is `SameSiteMode.Lax` to permit OAuth2 authentication. To strictly enforce a same-site policy of `SameSiteMode.Strict`, set the `MinimumSameSitePolicy`. Although this setting breaks OAuth2 and other cross-origin authentication schemes, it elevates the level of cookie security for other types of apps that don't rely on cross-origin request processing.

```
var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

The Cookie Policy Middleware setting for `MinimumSameSitePolicy` can affect your setting of `Cookie.SameSite` in `CookieAuthenticationOptions` settings according to the matrix below.

MINIMUMSAMESITEPOLICY	COOKIE.SAMESITE	RESULTANT COOKIE.SAMESITE SETTING
SameSiteMode.None	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Lax SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Strict SameSiteMode.Strict SameSiteMode.Strict

Creating an authentication cookie

To create a cookie holding user information, you must construct a [ClaimsPrincipal](#). The user information is serialized and stored in the cookie.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Call [SignInAsync](#) to sign in the user:

```
await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

`SignInAsync` creates an encrypted cookie and adds it to the current response. If you don't specify an `AuthenticationScheme`, the default scheme is used.

Under the covers, the encryption used is ASP.NET Core's [Data Protection](#) system. If you're hosting app on multiple machines, load balancing across apps, or using a web farm, then you must [configure data protection](#) to use the same key ring and app identifier.

Signing out

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

To sign out the current user and delete their cookie, call [SignInAsync](#):

```
await HttpContext.SignOutAsync(
    CookieAuthenticationDefaults.AuthenticationScheme);
```

If you aren't using `CookieAuthenticationDefaults.AuthenticationScheme` (or "Cookies") as the scheme (for example, "ContosoCookie"), supply the scheme you used when configuring the authentication provider. Otherwise, the default scheme is used.

Reacting to back-end changes

Once a cookie is created, it becomes the single source of identity. Even if you disable a user in your back-end systems, the cookie authentication system has no knowledge of this, and a user stays logged in as long as their cookie is valid.

The [ValidatePrincipal](#) event in ASP.NET Core 2.x or the [ValidateAsync](#) method in ASP.NET Core 1.x can be used to intercept and override validation of the cookie identity. This approach mitigates the risk of revoked users accessing the app.

One approach to cookie validation is based on keeping track of when the user database has been changed. If the database hasn't been changed since the user's cookie was issued, there is no need to re-authenticate the user if their cookie is still valid. To implement this scenario, the database, which is implemented in `IUserRepository` for this example, stores a `LastChanged` value. When any user is updated in the database, the `LastChanged` value is set to the current time.

In order to invalidate a cookie when the database changes based on the `LastChanged` value, create the cookie with a `LastChanged` claim containing the current `LastChanged` value from the database:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};

var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

To implement an override for the `ValidatePrincipal` event, write a method with the following signature in a class that you derive from [CookieAuthenticationEvents](#):

```
ValidatePrincipal(CookieValidatePrincipalContext)
```

An example looks like the following:

```

using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents
{
    private readonly IUserRepository _userRepository;

    public CustomCookieAuthenticationEvents(IUserRepository userRepository)
    {
        // Get the database from registered DI services.
        _userRepository = userRepository;
    }

    public override async Task ValidatePrincipal(CookieValidatePrincipalContext context)
    {
        var userPrincipal = context.Principal;

        // Look for the LastChanged claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !_userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}

```

Register the events instance during cookie service registration in the `ConfigureServices` method. Provide a scoped service registration for your `CustomCookieAuthenticationEvents` class:

```

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.EventsType = typeof(CustomCookieAuthenticationEvents);
    });

services.AddScoped<CustomCookieAuthenticationEvents>();

```

Consider a situation in which the user's name is updated — a decision that doesn't affect security in any way. If you want to non-destructively update the user principal, call `context.ReplacePrincipal` and set the `context.ShouldRenew` property to `true`.

WARNING

The approach described here is triggered on every request. This can result in a large performance penalty for the app.

Persistent cookies

You may want the cookie to persist across browser sessions. This persistence should only be enabled with explicit user consent with a "Remember Me" checkbox on login or a similar mechanism.

The following code snippet creates an identity and corresponding cookie that survives through browser closures.

Any sliding expiration settings previously configured are honored. If the cookie expires while the browser is closed, the browser clears the cookie once it's restarted.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
await HttpContext.SignInAsync(  
    CookieAuthenticationDefaults.AuthenticationScheme,  
    new ClaimsPrincipal(claimsIdentity),  
    new AuthenticationProperties  
    {  
        IsPersistent = true  
    }  
);
```

The `AuthenticationProperties` class resides in the `Microsoft.AspNetCore.Authentication` namespace.

Absolute cookie expiration

You can set an absolute expiration time with `ExpiresUtc`. You must also set `IsPersistent`; otherwise, `ExpiresUtc` is ignored and a single-session cookie is created. When `ExpiresUtc` is set on `SignInAsync`, it overrides the value of the `ExpireTimeSpan` option of `CookieAuthenticationOptions`, if set.

The following code snippet creates an identity and corresponding cookie that lasts for 20 minutes. This ignores any sliding expiration settings previously configured.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
await HttpContext.SignInAsync(  
    CookieAuthenticationDefaults.AuthenticationScheme,  
    new ClaimsPrincipal(claimsIdentity),  
    new AuthenticationProperties  
    {  
        IsPersistent = true,  
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)  
    }  
);
```

See also

- [Auth 2.0 Changes / Migration Announcement](#)
- [Limiting identity by scheme](#)

Azure Active Directory

11/1/2017 • 1 min to read • [Edit Online](#)

- [Integrating Azure AD Into an ASP.NET Core Web App](#)
- [Calling a ASP.NET Core Web API From a WPF Application Using Azure AD](#)
- [Calling a Web API in an ASP.NET Core Web Application Using Azure AD](#)
- [An ASP.NET Core web API with Azure AD B2C](#)

Authorization in ASP.NET Core: Simple, role, claims-based, and custom

1/10/2018 • 1 min to read • [Edit Online](#)

- [Introduction](#)
- [Razor Pages authorization](#)
- [Simple authorization](#)
- [Role-based authorization](#)
- [Claims-based authorization](#)
- [Policy-based authorization](#)
- [Dependency injection in requirement handlers](#)
- [Resource-based authorization](#)
- [View-based authorization](#)
- [Limiting identity by scheme](#)

Introduction

11/1/2017 • 1 min to read • [Edit Online](#)

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create a document library, add documents, edit documents, and delete them. A non-administrative user working with the library is only authorized to read the documents.

Authorization is orthogonal and independent from authentication, which is the process of ascertaining who a user is. Authentication may create one or more identities for the current user.

Authorization Types

ASP.NET Core authorization provides a simple declarative [role](#) and a [rich policy based](#) model. Authorization is expressed in requirements, and handlers evaluate a user's claims against requirements. Imperative checks can be based on simple policies or policies which evaluate both the user identity and properties of the resource that the user is attempting to access.

Namespaces

Authorization components, including the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes are found in the `Microsoft.AspNetCore.Authorization` namespace.

Create an ASP.NET Core app with user data protected by authorization

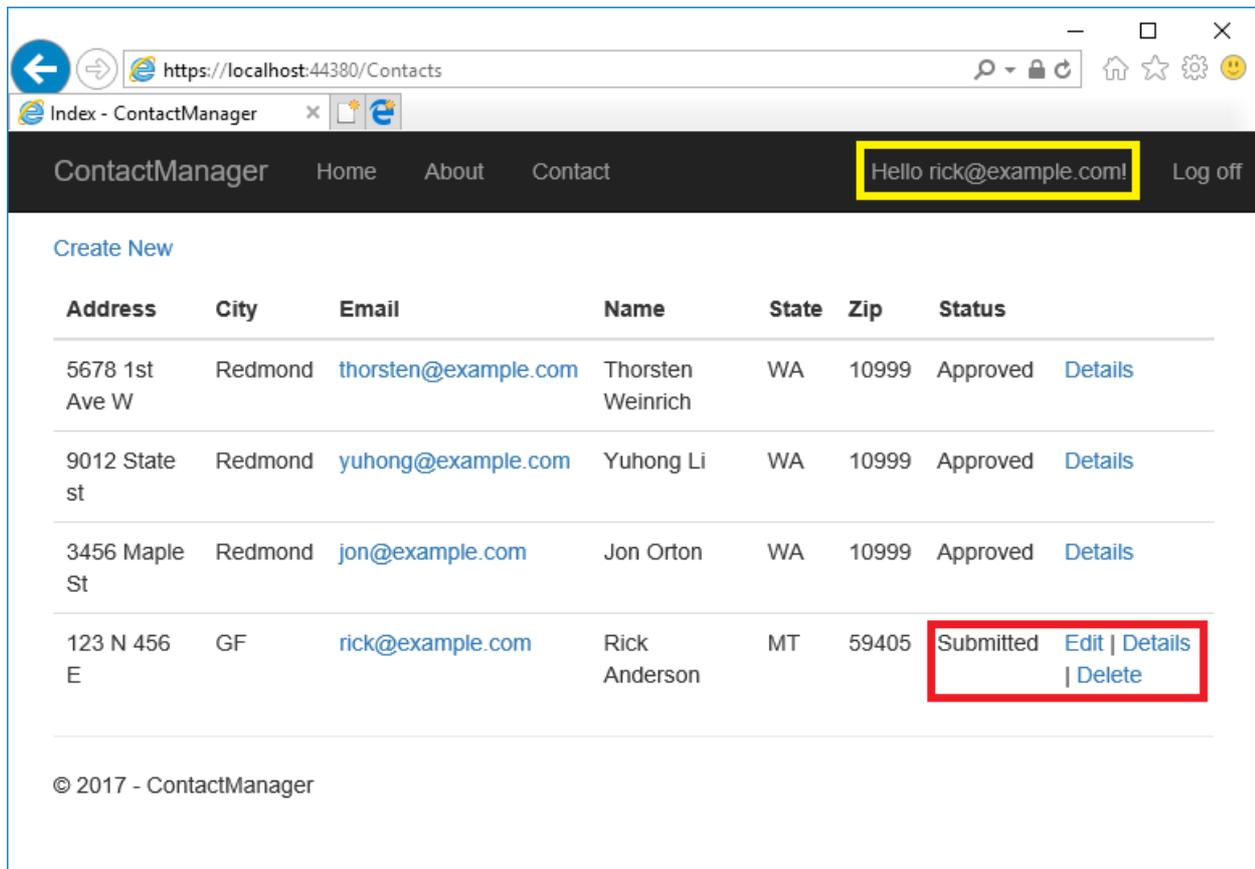
12/15/2017 • 18 min to read • [Edit Online](#)

By [Rick Anderson](#) and [Joe Audette](#)

This tutorial shows how to create a web app with user data protected by authorization. It displays a list of contacts that authenticated (registered) users have created. There are three security groups:

- Registered users can view all the approved contact data.
- Registered users can edit/delete their own data.
- Managers can approve or reject contact data. Only approved contacts are visible to users.
- Administrators can approve/reject and edit/delete any data.

In the following image, user Rick (rick@example.com) is signed in. User Rick can only view approved contacts and edit/delete his contacts. Only the last record, created by Rick, displays edit and delete links

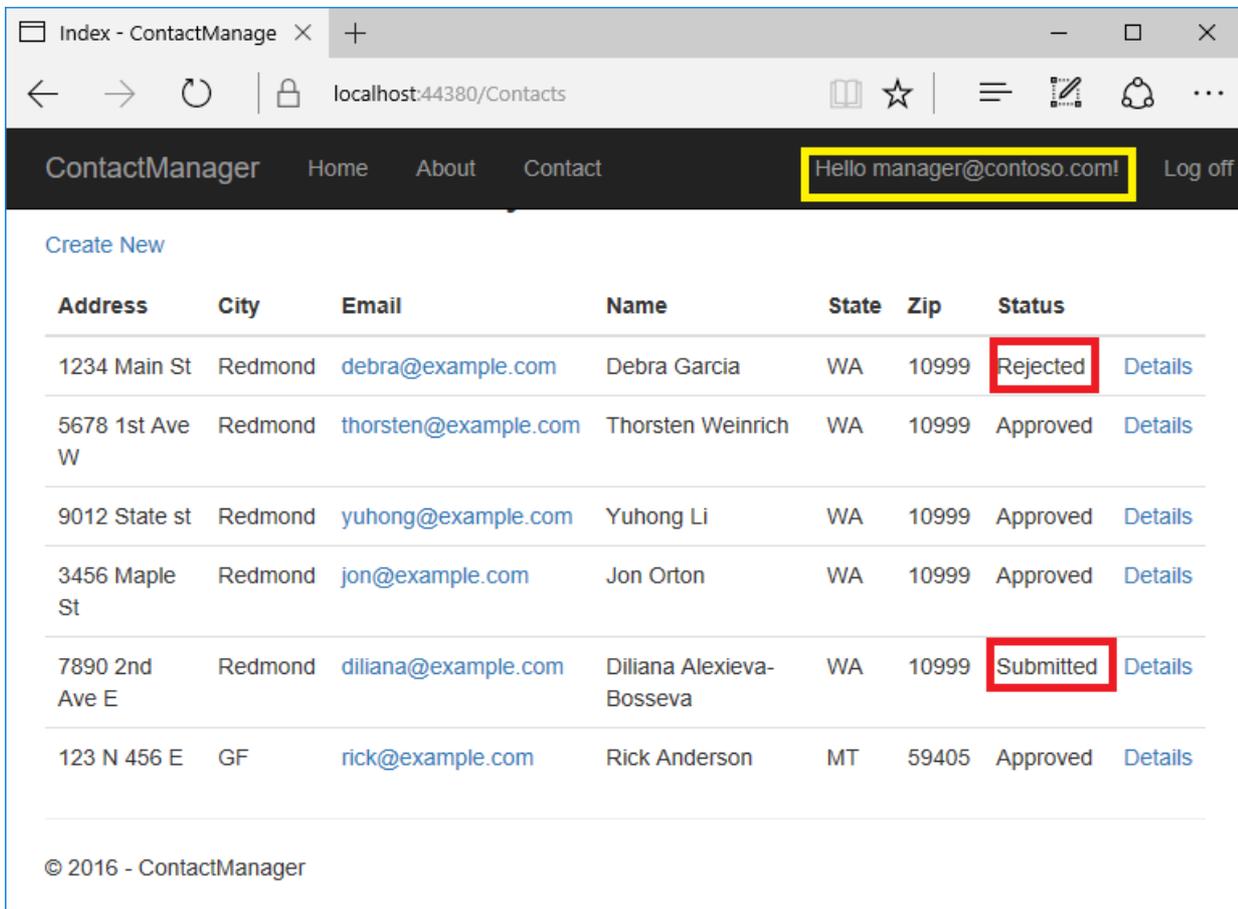


The screenshot shows a web browser window with the URL <https://localhost:44380/Contacts>. The page title is "ContactManager" and the user is logged in as "Hello rick@example.com!". The page displays a list of contacts with the following columns: Address, City, Email, Name, State, Zip, Status, and Details. The last contact, Rick Anderson, is in a "Submitted" status and has "Edit | Details | Delete" links next to it.

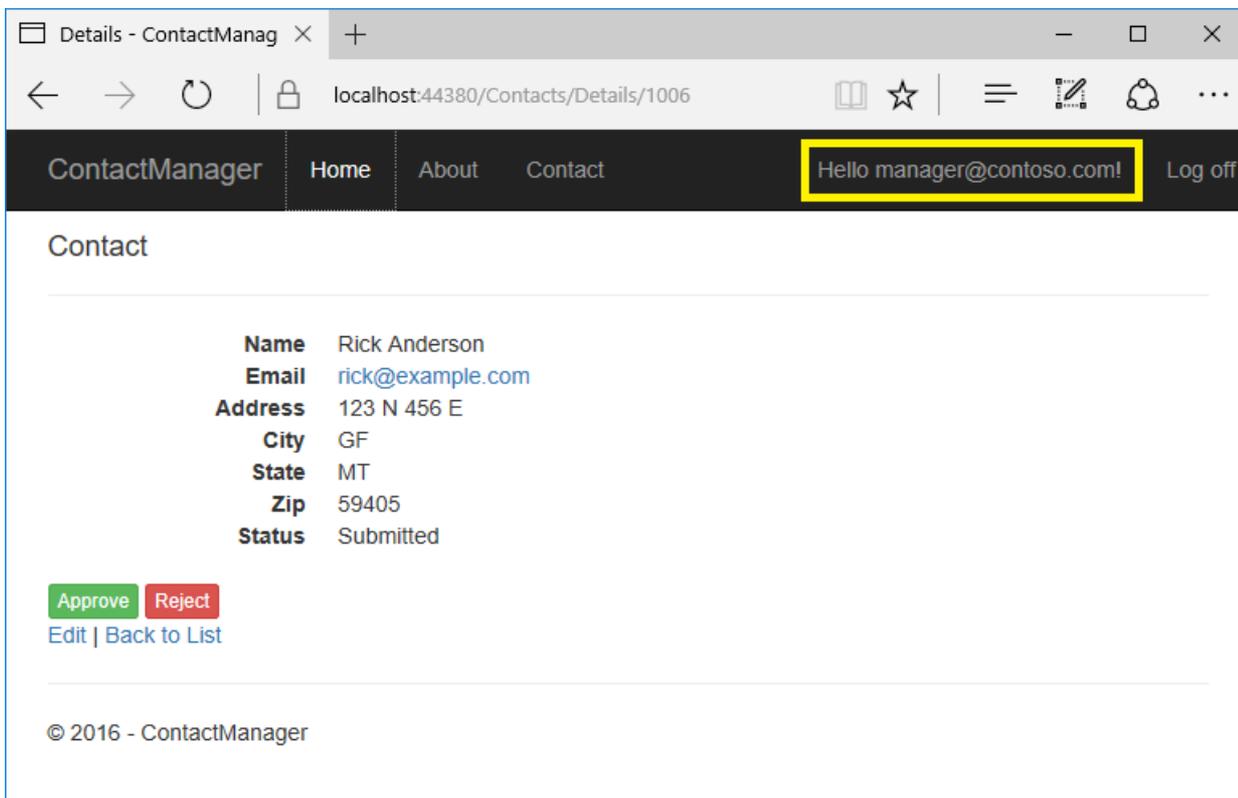
Address	City	Email	Name	State	Zip	Status	Details
5678 1st Ave W	Redmond	thorsten@example.com	Thorsten Weinrich	WA	10999	Approved	Details
9012 State st	Redmond	yuhong@example.com	Yuhong Li	WA	10999	Approved	Details
3456 Maple St	Redmond	jon@example.com	Jon Orton	WA	10999	Approved	Details
123 N 456 E	GF	rick@example.com	Rick Anderson	MT	59405	Submitted	Edit Details Delete

© 2017 - ContactManager

In the following image, manager@contoso.com is signed in and in the managers role.

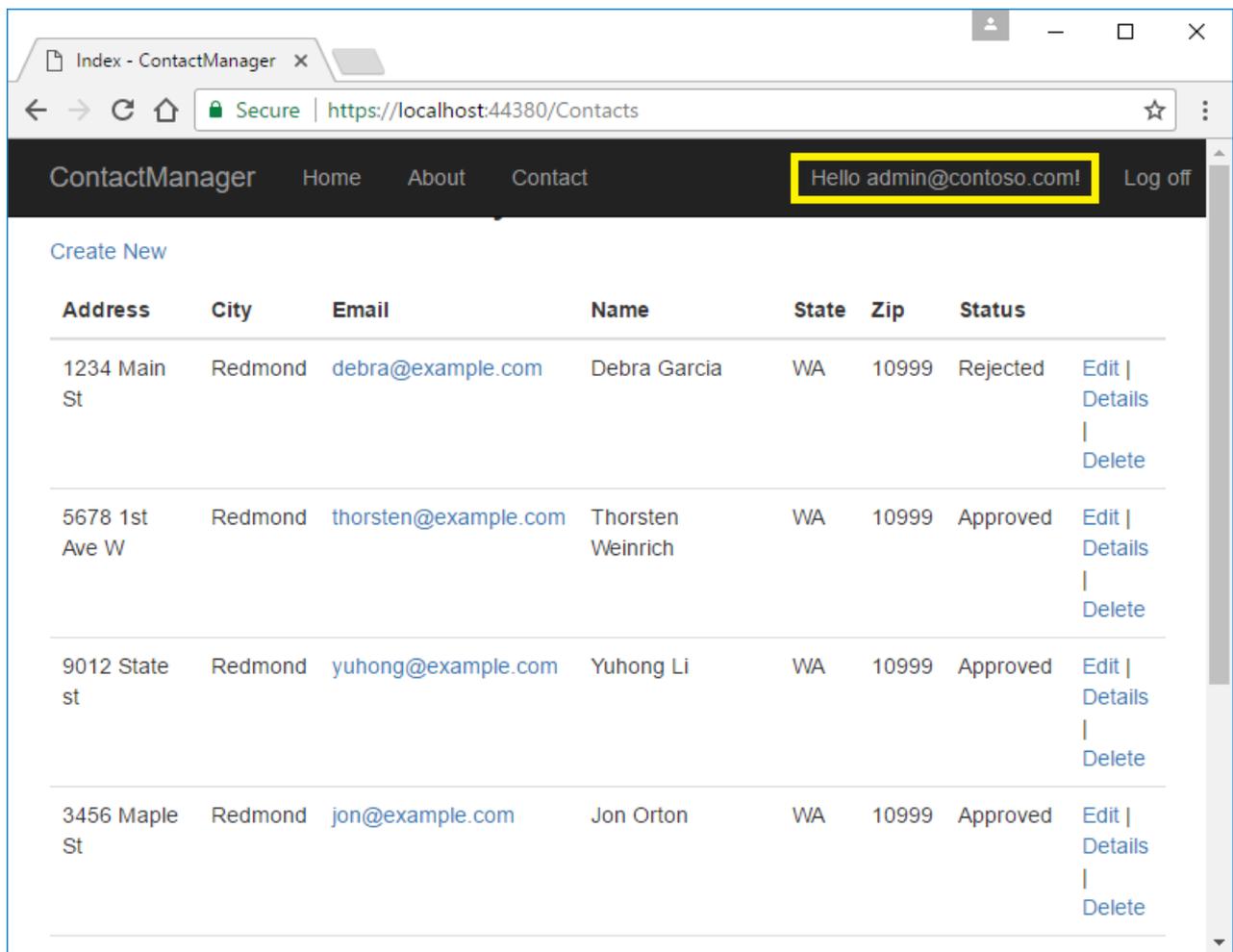


The following image shows the managers details view of a contact.



Only managers and administrators have the approve and reject buttons.

In the following image, `admin@contoso.com` is signed in and in the administrator's role.



The administrator has all privileges. She can read/edit/delete any contact and change the status of contacts.

The app was created by [scaffolding](#) the following `Contact` model:

```
public class Contact
{
    public int ContactId { get; set; }

    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    public string Email { get; set; }
}
```

A `ContactIsOwnerAuthorizationHandler` authorization handler ensures that a user can only edit their data. A `ContactManagerAuthorizationHandler` authorization handler allows managers to approve or reject contacts. A `ContactAdministratorsAuthorizationHandler` authorization handler allows administrators to approve or reject contacts and to edit/delete contacts.

Prerequisites

This is not a beginning tutorial. You should be familiar with:

- [ASP.NET Core MVC](#)
- [Entity Framework Core](#)

The starter and completed app

[Download](#) the [completed](#) app. [Test](#) the completed app so you become familiar with its security features.

The starter app

It's helpful to compare your code with the completed sample.

[Download](#) the [starter](#) app.

See [Create the starter app](#) if you'd like to create it from scratch.

Update the database:

```
dotnet ef database update
```

Run the app, tap the **ContactManager** link, and verify you can create, edit, and delete a contact.

This tutorial has all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Modify the app to secure user data

The following sections have all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Tie the contact data to the user

Use the ASP.NET [Identity](#) user ID to ensure users can edit their data, but not other users data. Add `OwnerID` to the `Contact` model:

```
public class Contact
{
    public int ContactId { get; set; }

    // user ID from AspNetUser table
    public string OwnerID { get; set; }

    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

`OwnerID` is the user's ID from the `AspNetUser` table in the [Identity](#) database. The `Status` field determines if a contact is viewable by general users.

Scaffold a new migration and update the database:

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

Require SSL and authenticated users

In the `ConfigureServices` method of the `Startup.cs` file, add the `RequireHttpsAttribute` authorization filter:

```
var skipSSL = Configuration.GetValue<bool>("LocalTest:skipSSL");
// requires using Microsoft.AspNetCore.Mvc;
services.Configure<MvcOptions>(options =>
{
// Set LocalTest:skipSSL to true to skip SSL requirement in
// debug mode. This is useful when not using Visual Studio.
if (!_hostingEnv.IsDevelopment() && !skipSSL)
{
options.Filters.Add(new RequireHttpsAttribute());
}
});
```

To redirect HTTP requests to HTTPS, see [URL Rewriting Middleware](#). If you are using Visual Studio Code or testing on local platform that doesn't include a test certificate for SSL:

- Set `"LocalTest:skipSSL": true` in the `appsettings.json` file.

Require authenticated users

Set the default authentication policy to require users to be authenticated. You can opt out of authentication at the controller or action method with the `[AllowAnonymous]` attribute. With this approach, any new controllers added will automatically require authentication, which is safer than relying on new controllers to include the `[Authorize]` attribute. Add the following to the `ConfigureServices` method of the `Startup.cs` file:

```
// requires: using Microsoft.AspNetCore.Authorization;
//           using Microsoft.AspNetCore.Mvc.Authorization;
services.AddMvc(config =>
{
var policy = new AuthorizationPolicyBuilder()
.RequireAuthenticatedUser()
.Build();
config.Filters.Add(new AuthorizeFilter(policy));
});
```

Add `[AllowAnonymous]` to the home controller so anonymous users can get information about the site before they register.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace ContactManager.Controllers
{
[AllowAnonymous]
public class HomeController : Controller
{
public IActionResult Index()
{
return View();
}
}
```

Configure the test account

The `SeedData` class creates two accounts, administrator and manager. Use the [Secret Manager tool](#) to set a password for these accounts. Do this from the project directory (the directory containing `Program.cs`).

```
dotnet user-secrets set SeedUserPW <PW>
```

Update `Configure` to use the test password:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

    app.UseMvcWithDefaultRoute();

    // Set password with the Secret Manager tool.
    // dotnet user-secrets set SeedUserPW <pw>
    var testUserPw = Configuration["SeedUserPW"];

    if (String.IsNullOrEmpty(testUserPw))
    {
        throw new System.Exception("Use secrets manager to set SeedUserPW \n" +
            "dotnet user-secrets set SeedUserPW <pw>");
    }

    try
    {
        SeedData.Initialize(app.ApplicationServices, testUserPw).Wait();
    }
    catch
    {
        throw new System.Exception("You need to update the DB "
            + "\nPM > Update-Database " + "\n or \n" +
            "> dotnet ef database update"
            + "\nIf that doesn't work, comment out SeedData and "
            + "register a new user");
    }
}
```

Add the administrator user ID and `Status = ContactStatus.Approved` to the contacts. Only one contact is shown, add the user ID to all contacts:

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
    {
        return; // DB has been seeded
    }

    context.Contact.AddRange(
        new Contact
        {
            Name = "Debra Garcia",
            Address = "1234 Main St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "debra@example.com",
            Status = ContactStatus.Approved,
            OwnerID = adminID
        },
    },
```

Create owner, manager, and administrator authorization handlers

Create a `ContactIsOwnerAuthorizationHandler` class in the *Authorization* folder. The

`ContactIsOwnerAuthorizationHandler` will verify the user acting on the resource owns the resource.

```

using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
        : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        UserManager<ApplicationUser> _userManager;

        public ContactIsOwnerAuthorizationHandler(UserManager<ApplicationUser>
            userManager)
        {
            _userManager = userManager;
        }

        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                OperationAuthorizationRequirement requirement,
                Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.FromResult(0);
            }

            // If we're not asking for CRUD permission, return.

            if (requirement.Name != Constants.CreateOperationName &&
                requirement.Name != Constants.ReadOperationName &&
                requirement.Name != Constants.UpdateOperationName &&
                requirement.Name != Constants.DeleteOperationName )
            {
                return Task.FromResult(0);
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.FromResult(0);
        }
    }
}

```

The `ContactIsOwnerAuthorizationHandler` calls `context.Succeed` if the current authenticated user is the contact owner. Authorization handlers generally return `context.Succeed` when the requirements are met. They return `Task.FromResult(0)` when requirements are not met. `Task.FromResult(0)` is neither success or failure, it allows other authorization handler to run. If you need to explicitly fail, return `context.Fail()`.

We allow contact owners to edit/delete their own data, so we don't need to check the operation passed in the requirement parameter.

Create a manager authorization handler

Create a `ContactManagerAuthorizationHandler` class in the *Authorization* folder. The `ContactManagerAuthorizationHandler` will verify the user acting on the resource is a manager. Only managers can approve or reject content changes (new or changed).

```

using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                OperationAuthorizationRequirement requirement,
                Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.FromResult(0);
            }

            // If not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.FromResult(0);
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
                context.Succeed(requirement);
            }

            return Task.FromResult(0);
        }
    }
}

```

Create an administrator authorization handler

Create a `ContactAdministratorsAuthorizationHandler` class in the *Authorization* folder. The `ContactAdministratorsAuthorizationHandler` will verify the user acting on the resource is an administrator. Administrator can do all operations.

```

using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
        : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
            AuthorizationHandlerContext context,
            OperationAuthorizationRequirement requirement,
            Contact resource)
        {
            if (context.User == null)
            {
                return Task.FromResult(0);
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.FromResult(0);
        }
    }
}

```

Register the authorization handlers

Services using Entity Framework Core must be registered for [dependency injection](#) using [AddScoped](#). The `ContactIsOwnerAuthorizationHandler` uses ASP.NET Core [Identity](#), which is built on Entity Framework Core. Register the handlers with the service collection so they will be available to the `ContactsController` through [dependency injection](#). Add the following code to the end of `ConfigureServices`:

```

// Authorization handlers.
services.AddScoped<IAuthorizationHandler,
    ContactIsOwnerAuthorizationHandler>();

services.AddSingleton<IAuthorizationHandler,
    ContactAdministratorsAuthorizationHandler>();

services.AddSingleton<IAuthorizationHandler,
    ContactManagerAuthorizationHandler>();

```

`ContactAdministratorsAuthorizationHandler` and `ContactManagerAuthorizationHandler` are added as singletons. They are singletons because they don't use EF and all the information needed is in the `Context` parameter of the `HandleRequirementAsync` method.

The complete `ConfigureServices`:

```

public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();

    var skipSSL = Configuration.GetValue<bool>("LocalTest:skipSSL");
    // requires using Microsoft.AspNetCore.Mvc;
    services.Configure<MvcOptions>(options =>
    {
        // Set LocalTest:skipSSL to true to skip SSL requirement in
        // debug mode. This is useful when not using Visual Studio.
        if (_hostingEnv.IsDevelopment() && !skipSSL)
        {
            options.Filters.Add(new RequireHttpsAttribute());
        }
    });

    // requires: using Microsoft.AspNetCore.Authorization;
    //             using Microsoft.AspNetCore.Mvc.Authorization;
    services.AddMvc(config =>
    {
        var policy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    });

    // Authorization handlers.
    services.AddScoped<IAuthorizationHandler,
        ContactIsOwnerAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
        ContactAdministratorsAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
        ContactManagerAuthorizationHandler>();
}

```

Update the code to support authorization

In this section, you update the controller and views and add an operations requirements class.

Update the Contacts controller

Update the `ContactsController` constructor:

- Add the `IAuthorizationService` service to access to the authorization handlers.
- Add the `Identity` `userManager` service:

```

public class ContactsController : Controller
{
    private readonly ApplicationDbContext _context;
    private readonly IAuthorizationService _authorizationService;
    private readonly UserManager<ApplicationUser> _userManager;

    public ContactsController(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<ApplicationUser> userManager)
    {
        _context = context;
        _userManager = userManager;
        _authorizationService = authorizationService;
    }
}

```

Add a contact operations requirements class

Add the `ContactOperations` class to the *Authorization* folder. This class contain the requirements our app supports:

```

using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
            new OperationAuthorizationRequirement {Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
            new OperationAuthorizationRequirement {Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
            new OperationAuthorizationRequirement {Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
            new OperationAuthorizationRequirement {Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
            new OperationAuthorizationRequirement {Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
            new OperationAuthorizationRequirement {Name=Constants.RejectOperationName};
    }

    public class Constants
    {
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =
            "ContactAdministrators";
        public static readonly string ContactManagersRole = "ContactManagers";
    }
}

```

Update Create

Update the `HTTP POST Create` method to:

- Add the user ID to the `Contact` model.
- Call the authorization handler to verify the user owns the contact.

```

// POST: Contacts/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(ContactEditViewModel editModel)
{
    if (!ModelState.IsValid)
    {
        return View(editModel);
    }

    var contact = ViewModel_to_model(new Contact(), editModel);

    contact.OwnerID = _userManager.GetUserId(User);

    var isAuthorized = await _authorizationService.AuthorizeAsync(
        User, contact,
        ContactOperations.Create);

    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    _context.Add(contact);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}

```

Update Edit

Update both `Edit` methods to use the authorization handler to verify the user owns the contact. Because we are performing resource authorization we cannot use the `[Authorize]` attribute. We don't have access to the resource when attributes are evaluated. Resource based authorization must be imperative. Checks must be performed once we have access to the resource, either by loading it in our controller, or by loading it within the handler itself. Frequently you will access the resource by passing in the resource key.

```

public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var contact = await _context.Contact.SingleOrDefaultAsync(
        m => m.ContactId == id);

    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await _authorizationService.AuthorizeAsync(
        User, contact,
        ContactOperations.Update);

    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    var editModel = Model_to_viewModel(contact);

    return View(editModel);
}

// POST: Contacts/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]

```

```

public async Task<IActionResult> Edit(int id, ContactEditViewModel editModel)
{
    if (!ModelState.IsValid)
    {
        return View(editModel);
    }

    // Fetch Contact from DB to get OwnerID.
    var contact = await _context.Contact.SingleOrDefaultAsync(m => m.ContactId == id);
    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await _authorizationService.AuthorizeAsync(User, contact,
                                                         ContactOperations.Update);

    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    contact = ViewModel_to_model(contact, editModel);

    if (contact.Status == ContactStatus.Approved)
    {
        // If the contact is updated after approval,
        // and the user cannot approve set the status back to submitted
        var canApprove = await _authorizationService.AuthorizeAsync(User, contact,
                                                                    ContactOperations.Approve);

        if (!canApprove) contact.Status = ContactStatus.Submitted;
    }

    _context.Update(contact);
    await _context.SaveChangesAsync();

    return RedirectToAction("Index");
}

```

Update the Delete method

Update both `Delete` methods to use the authorization handler to verify the user owns the contact.

```

public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var contact = await _context.Contact.SingleOrDefaultAsync(m => m.ContactId == id);
    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await _authorizationService.AuthorizeAsync(User, contact,
        ContactOperations.Delete);

    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    return View(contact);
}

// POST: Contacts/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var contact = await _context.Contact.SingleOrDefaultAsync(m => m.ContactId == id);

    var isAuthorized = await _authorizationService.AuthorizeAsync(User, contact,
        ContactOperations.Delete);

    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    _context.Contact.Remove(contact);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}

```

Inject the authorization service into the views

Currently the UI shows edit and delete links for data the user cannot modify. We'll fix that by applying the authorization handler to the views.

Inject the authorization service in the *Views/_ViewImports.cshtml* file so it will be available to all views:

```

@using ContactManager
@using ContactManager.Models
@using ContactManager.Models.AccountViewModels
@using ContactManager.Models.ManageViewModels
@using ContactManager.Authorization
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

```

Update the *Views/Contacts/Index.cshtml* Razor view to only display the edit and delete links for users who can edit/delete the contact.

Add `@using ContactManager.Authorization;`

Update the `Edit` and `Delete` links so they are only rendered for users with permission to edit and delete the contact.

```
</td>
<td>
    @Html.DisplayFor(modelItem => item.Zip)
</td>
<td>
    @Html.DisplayFor(modelItem => item.Status)
</td>
<td>
    @if (await AuthorizationService.AuthorizeAsync(User,
                                                item, ContactOperations.Update))
    {
        <a asp-action="Edit" asp-route-id="@item.ContactId">Edit</a><text> | </text>
    }
    <a asp-action="Details" asp-route-id="@item.ContactId">Details</a>
    @if (await AuthorizationService.AuthorizeAsync(User,
                                                item, ContactOperations.Delete))
    {
        <text> | </text>
        <a asp-action="Delete" asp-route-id="@item.ContactId">Delete</a>
    }
</td>
</tr>
```

Warning: Hiding links from users that do not have permission to edit or delete data does not secure the app. Hiding links makes the app more user friendly by displaying only valid links. Users can hack the generated URLs to invoke edit and delete operations on data they don't own. The controller must repeat the access checks to be secure.

Update the Details view

Update the details view so managers can approve or reject contacts:

```

        <dt>
            @Html.DisplayNameFor(model => model.Zip)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Zip)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Status)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Status)
        </dd>
    </dl>
</div>
@if (Model.Status != ContactStatus.Approved)
{
    @if (await AuthorizationService.AuthorizeAsync(User, Model, ContactOperations.Approve))
    {
        <form asp-action="SetStatus" asp-controller="Contacts" style="display:inline;">
            <input type="hidden" name="id" value="@Model.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-success">Approve</button>
        </form>
    }
}
@if (Model.Status != ContactStatus.Rejected)
{
    @if (await AuthorizationService.AuthorizeAsync(User, Model, ContactOperations.Reject))
    {
        <form asp-action="SetStatus" asp-controller="Contacts" style="display:inline;">
            <input type="hidden" name="id" value="@Model.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-danger">Reject</button>
        </form>
    }
}
<div>
    @* Uncomment to perform authorization check. A real app would hide the edit link from users
       uses who don't have edit access. A user without edit access can click the link but will get denied
       access in the controller.
    @if(await AuthorizationService.AuthorizeAsync(User, Model, ContactOperations.Update))
    {
    *@
        <a asp-action="Edit" asp-route-id="@Model.ContactId">Edit</a> <text>|</text>
    @*
    }
    *@
    <a asp-action="Index">Back to List</a>
</div>

```

Test the completed app

If you are using Visual Studio Code or testing on local platform that doesn't include a test certificate for SSL:

- Set `"LocalTest:skipSSL": true` in the `appsettings.json` file.

If you have run the app and have contacts, delete all the records in the `Contact` table and restart the app to seed the database. If you are using Visual Studio, you need to exit and restart IIS Express to seed the database.

Register a user to browse the contacts.

An easy way to test the completed app is to launch three different browsers (or incognito/InPrivate versions). In one browser, register a new user, for example, `test@example.com`. Sign in to each browser with a different user. Verify the following:

- Registered users can view all the approved contact data.
- Registered users can edit/delete their own data.
- Managers can approve or reject contact data. The `Details` view shows **Approve** and **Reject** buttons.
- Administrators can approve/reject and edit/delete any data.

USER	OPTIONS
test@example.com	Can edit/delete own data
manager@contoso.com	Can approve/reject and edit/delete own data
admin@contoso.com	Can edit/delete and approve/reject all data

Create a contact in the administrators browser. Copy the URL for delete and edit from the administrator contact. Paste these links into the test user's browser to verify the test user cannot perform these operations.

Create the starter app

Follow these instructions to create the starter app.

- Create an **ASP.NET Core Web Application** using [Visual Studio 2017](#) named "ContactManager"
 - Create the app with **Individual User Accounts**.
 - Name it "ContactManager" so your namespace will match the namespace use in the sample.
- Add the following `Contact` model:

```
public class Contact
{
    public int ContactId { get; set; }

    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    public string Email { get; set; }
}
```

- Scaffold the `Contact` model using Entity Framework Core and the `ApplicationDbContext` data context. Accept all the scaffolding defaults. Using `ApplicationDbContext` for the data context class puts the contact table in the [Identity](#) database. See [Adding a model](#) for more information.
- Update the **ContactManager** anchor in the `Views/Shared/_Layout.cshtml` file from `asp-controller="Home"` to `asp-controller="Contacts"` so tapping the **ContactManager** link will invoke the Contacts controller. The original markup:

```
<a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">ContactManager</a>
```

The updated markup:

```
<a asp-area="" asp-controller="Contacts" asp-action="Index" class="navbar-brand">ContactManager</a>
```

- Scaffold the initial migration and update the database

```
dotnet ef migrations add initial
dotnet ef database update
```

- Test the app by creating, editing and deleting a contact

Seed the database

Add the `SeedData` class to the `Data` folder. If you've downloaded the sample, you can copy the `SeedData.cs` file to the `Data` folder of the starter project.

```
using ContactManager.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ContactManager.Data
{
    public static class SeedData
    {
        public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
        {
            using (var context = new ApplicationDbContext(
                serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
            {
                var uid = await CreateTestUser(serviceProvider, testUserPw);
                SeedDB(context, uid);
            }
        }

        private static async Task<string> CreateTestUser(IServiceProvider serviceProvider, string testUserPw)
        {
            if (String.IsNullOrEmpty(testUserPw))
                return "";

            const string SeedUserName = "test@example.com";

            var userManager = serviceProvider.GetService<userManager<ApplicationUser>>();

            var user = await userManager.FindByNameAsync(SeedUserName);
            if (user == null)
            {
                user = new ApplicationUser { UserName = SeedUserName };
                await userManager.CreateAsync(user, testUserPw);
            }

            return user.Id;
        }

        public static void SeedDB(ApplicationDbContext context, string uid)
        {
            if (context.Contact.Any())
            {
                return; // DB has been seeded
            }

            context.Contact.AddRange(
                new Contact
                {
                    Name = "Debra Garcia",
                    Address = "1234 Main St",
                    City = "Redmond",
                }
            );
        }
    }
}
```

```

        State = "WA",
        Zip = "10999",
        Email = "debra@example.com"
    },
    new Contact
    {
        Name = "Thorsten Weinrich",
        Address = "5678 1st Ave W",
        City = "Redmond",
        State = "WA",
        Zip = "10999",
        Email = "thorsten@example.com"
    },
    new Contact
    {
        Name = "Yuhong Li",
        Address = "9012 State st",
        City = "Redmond",
        State = "WA",
        Zip = "10999",
        Email = "yuhong@example.com"
    },
    new Contact
    {
        Name = "Jon Orton",
        Address = "3456 Maple St",
        City = "Redmond",
        State = "WA",
        Zip = "10999",
        Email = "jon@example.com"
    },
    new Contact
    {
        Name = "Diliana Alexieva-Bosseva",
        Address = "7890 2nd Ave E",
        City = "Redmond",
        State = "WA",
        Zip = "10999",
        Email = "diliana@example.com"
    }
    );
context.SaveChanges();
}
}
}

```

Add the highlighted code to the end of the `Configure` method in the *Startup.cs* file:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });

    try
    {
        SeedData.Initialize(app.ApplicationServices, "").Wait();
    }
    catch
    {
        throw new System.Exception("You need to update the DB "
            + "\nPM > Update-Database " + "\n or \n" +
            "> dotnet ef database update"
            + "\nIf that doesn't work, comment out SeedData and register a new user");
    }
}
}

```

Test that the app seeded the database. The seed method does not run if there are any rows in the contact DB.

Create a class used in the tutorial

- Create a folder named *Authorization*.
- Copy the *Authorization\ContactOperations.cs* file from the completed project download, or copy the following code:

```

using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
            new OperationAuthorizationRequirement {Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
            new OperationAuthorizationRequirement {Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
            new OperationAuthorizationRequirement {Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
            new OperationAuthorizationRequirement {Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
            new OperationAuthorizationRequirement {Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
            new OperationAuthorizationRequirement {Name=Constants.RejectOperationName};
    }

    public class Constants
    {
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =
            "ContactAdministrators";
        public static readonly string ContactManagersRole = "ContactManagers";
    }
}

```

Additional resources

- [ASP.NET Core Authorization Lab](#). This lab goes into more detail on the security features introduced in this tutorial.
- [Authorization in ASP.NET Core : Simple, role, claims-based and custom](#)
- [Custom policy-based authorization](#)

Razor Pages authorization conventions in ASP.NET Core

11/3/2017 • 2 min to read • [Edit Online](#)

By [Luke Latham](#)

One way to control access in your Razor Pages app is to use authorization conventions at startup. These conventions allow you to authorize users and allow anonymous users to access individual pages or folders of pages. The conventions described in this topic automatically apply [authorization filters](#) to control access.

[View or download sample code \(how to download\)](#)

Require authorization to access a page

Use the [AuthorizePage](#) convention via [AddRazorPagesOptions](#) to add an [AuthorizeFilter](#) to the page at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    });
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

An [AuthorizePage overload](#) is available if you need to specify an authorization policy.

Require authorization to access a folder of pages

Use the [AuthorizeFolder](#) convention via [AddRazorPagesOptions](#) to add an [AuthorizeFilter](#) to all of the pages in a folder at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    });
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

An [AuthorizeFolder overload](#) is available if you need to specify an authorization policy.

Allow anonymous access to a page

Use the [AllowAnonymousToPage](#) convention via [AddRazorPagesOptions](#) to add an [AllowAnonymousFilter](#) to a page at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    });
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

Allow anonymous access to a folder of pages

Use the [AllowAnonymousToFolder](#) convention via [AddRazorPagesOptions](#) to add an [AllowAnonymousFilter](#) to all of the pages in a folder at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    });
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

Note on combining authorized and anonymous access

It's perfectly valid to specify that a folder of pages require authorization and specify that a page within that folder allows anonymous access:

```
// This works.
.AuthorizeFolder("/Private").AllowAnonymousToPage("/Private/Public")
```

The reverse, however, isn't true. You can't declare a folder of pages for anonymous access and specify a page within for authorization:

```
// This doesn't work!
.AllowAnonymousToFolder("/Public").AuthorizePage("/Public/Private")
```

Requiring authorization on the Private page won't work because when both the [AllowAnonymousFilter](#) and [AuthorizeFilter](#) filters are applied to the page, the [AllowAnonymousFilter](#) wins and controls access.

See also

- [Razor Pages custom route and page model providers](#)
- [PageConventionCollection](#) class

Simple Authorization

12/6/2017 • 1 min to read • [Edit Online](#)

Authorization in MVC is controlled through the `AuthorizeAttribute` attribute and its various parameters. At its simplest, applying the `AuthorizeAttribute` attribute to a controller or action limits access to the controller or action to any authenticated user.

For example, the following code limits access to the `AccountController` to any authenticated user.

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

If you want to apply authorization to an action rather than the controller, apply the `AuthorizeAttribute` attribute to the action itself:

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

Now only authenticated users can access the `Logout` function.

You can also use the `AllowAnonymousAttribute` attribute to allow access by non-authenticated users to individual actions. For example:

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

This would allow only authenticated users to the `AccountController`, except for the `Login` action, which is

accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

WARNING

`[AllowAnonymous]` bypasses all authorization statements. If you apply combine `[AllowAnonymous]` and any `[Authorize]` attribute then the Authorize attributes will always be ignored. For example if you apply `[AllowAnonymous]` at the controller level any `[Authorize]` attributes on the same controller, or on any action within it will be ignored.

Role based Authorization

1/9/2018 • 2 min to read • [Edit Online](#)

When an identity is created it may belong to one or more roles. For example, Tracy may belong to the Administrator and User roles whilst Scott may only belong to the User role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the `IsInRole` method on the `ClaimsPrincipal` class.

Adding role checks

Role-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying roles which the current user must be a member of to access the requested resource.

For example, the following code limits access to any actions on the `AdministrationController` to users who are a member of the `Administrator` role:

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
}
```

You can specify multiple roles as a comma separated list:

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
```

This controller would be only accessible by users who are members of the `HRManager` role or the `Finance` role.

If you apply multiple attributes then an accessing user must be a member of all the roles specified; the following sample requires that a user must be a member of both the `PowerUser` and `ControlPanelUser` role.

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

You can further limit access by applying additional role authorization attributes at the action level:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

In the previous code snippet members of the `Administrator` role or the `PowerUser` role can access the controller and the `SetTime` action, but only members of the `Administrator` role can access the `ShutDown` action.

You can also lock down a controller but allow anonymous, unauthenticated access to individual actions.

```
[Authorize]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [AllowAnonymous]
    public ActionResult Login()
    {
    }
}
```

Policy based role checks

Role requirements can also be expressed using the new Policy syntax, where a developer registers a policy at startup as part of the Authorization service configuration. This normally occurs in `ConfigureServices()` in your `Startup.cs` file.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole", policy => policy.RequireRole("Administrator"));
    });
}
```

Policies are applied using the `Policy` property on the `AuthorizeAttribute` attribute:

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

If you want to specify multiple allowed roles in a requirement then you can specify them as parameters to the `RequireRole` method:

```
options.AddPolicy("ElevatedRights", policy =>
    policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));
```

This example authorizes users who belong to the `Administrator`, `PowerUser` or `BackupAdministrator` roles.

Claims-Based Authorization

11/1/2017 • 3 min to read • [Edit Online](#)

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is name value pair that represents what the subject is, not what the subject can do. For example, you may have a driver's license, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be `DateOfBirth`, the claim value would be your date of birth, for example `8th June 1970` and the issuer would be the driving license authority. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example if you want access to a night club the authorization process might be:

The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

Claim based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying claims which the current user must possess, and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based, the developer must build and register a policy expressing the claims requirements.

The simplest type of claim policy looks for the presence of a claim and does not check the value.

First you need to build and register the policy. This takes place as part of the Authorization service configuration, which normally takes place in `ConfigureServices()` in your `Startup.cs` file.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));
    });
}
```

In this case the `EmployeeOnly` policy checks for the presence of an `EmployeeNumber` claim on the current identity.

You then apply the policy using the `Policy` property on the `AuthorizeAttribute` attribute to specify the policy name;

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

The `AuthorizeAttribute` attribute can be applied to an entire controller, in this instance only identities matching the policy will be allowed access to any Action on the controller.

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }
}
```

If you have a controller that is protected by the `AuthorizeAttribute` attribute, but want to allow anonymous access to particular actions you apply the `AllowAnonymousAttribute` attribute.

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
    }
}
```

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4 or 5.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Founders", policy =>
            policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));
    });
}
```

Multiple Policy Evaluation

If you apply multiple policies to a controller or action, then all policies must pass before access is granted. For example:

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public ActionResult Payslip()
    {
    }

    [Authorize(Policy = "HumanResources")]
    public ActionResult UpdateSalary()
    {
    }
}
```

In the above example any identity which fulfills the `EmployeeOnly` policy can access the `Payslip` action as that policy is enforced on the controller. However in order to call the `UpdateSalary` action the identity must fulfill *both*

the `EmployeeOnly` policy and the `HumanResources` policy.

If you want more complicated policies, such as taking a date of birth claim, calculating an age from it then checking the age is 21 or older then you need to write [custom policy handlers](#).

Custom policy-based authorization

12/15/2017 • 5 min to read • [Edit Online](#)

Underneath the covers, [role-based authorization](#) and [claims-based authorization](#) use a requirement, a requirement handler, and a pre-configured policy. These building blocks support the expression of authorization evaluations in code. The result is a richer, reusable, testable authorization structure.

An authorization policy consists of one or more requirements. It's registered as part of the authorization service configuration, in the `ConfigureServices` method of the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
}
```

In the preceding example, an "AtLeast21" policy is created. It has a single requirement, that of a minimum age, which is supplied as a parameter to the requirement.

Policies are applied by using the `[Authorize]` attribute with the policy name. For example:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseController : Controller
{
    public IActionResult Login() => View();

    public IActionResult Logout() => View();
}
```

Requirements

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user principal. In our "AtLeast21" policy, the requirement is a single parameter—the minimum age. A requirement implements `IAuthorizationRequirement`, which is an empty marker interface. A parameterized minimum age requirement could be implemented as follows:

```
using Microsoft.AspNetCore.Authorization;

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; private set; }

    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
}
```

NOTE

A requirement doesn't need to have data or properties.

Authorization handlers

An authorization handler is responsible for the evaluation of a requirement's properties. The authorization handler evaluates the requirements against a provided `AuthorizationHandlerContext` to determine if access is allowed. A requirement can have [multiple handlers](#). Handlers inherit `AuthorizationHandler<T>`, where `T` is the requirement to be handled.

The minimum age handler might look like this:

```

using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System;
using System.Security.Claims;
using System.Threading.Tasks;

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        MinimumAgeRequirement requirement)
    {
        if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&
            c.Issuer == "http://contoso.com"))
        {
            //TODO: Use the following if targeting a version of
            // .NET Framework older than 4.6:
            //     return Task.FromResult(0);
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(
            context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth &&
                c.Issuer == "http://contoso.com").Value);

        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        // .NET Framework older than 4.6:
        //     return Task.FromResult(0);
        return Task.CompletedTask;
    }
}

```

The preceding code determines if the current user principal has a date of birth claim which has been issued by a known and trusted Issuer. Authorization can't occur when the claim is missing, in which case a completed task is returned. When a claim is present, the user's age is calculated. If the user meets the minimum age defined by the requirement, authorization is deemed successful. When authorization is successful, `context.Succeed` is invoked with the satisfied requirement as a parameter.

Handler registration

Handlers are registered in the services collection during configuration. For example:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}

```

Each handler is added to the services collection by invoking

```
services.AddSingleton<IAuthorizationHandler, YourHandlerClass>();
```

What should a handler return?

Note that the `Handle` method in the [handler example](#) returns no value. How is a status of either success or failure indicated?

- A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.
- A handler does not need to handle failures generally, as other handlers for the same requirement may succeed.
- To guarantee failure, even if other requirement handlers succeed, call `context.Fail()`.

Regardless of what you call inside your handler, all handlers for a requirement will be called when a policy requires the requirement. This allows requirements to have side effects, such as logging, which will always take place even if `context.Fail()` has been called in another handler.

Why would I want multiple handlers for a requirement?

In cases where you want evaluation to be on an **OR** basis, implement multiple handlers for a single requirement. For example, Microsoft has doors which only open with key cards. If you leave your key card at home, the receptionist prints a temporary sticker and opens the door for you. In this scenario, you'd have a single requirement, *BuildingEntry*, but multiple handlers, each one examining a single requirement.

BuildingEntryRequirement.cs

```
using Microsoft.AspNetCore.Authorization;

public class BuildingEntryRequirement : IAuthorizationRequirement
{
}
```

BadgeEntryHandler.cs

```
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System.Security.Claims;
using System.Threading.Tasks;

public class BadgeEntryHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.BadgeId &&
                                    c.Issuer == "http://microsoftsecurity"))
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        // .NET Framework older than 4.6:
        // return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

```
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System.Security.Claims;
using System.Threading.Tasks;

public class TemporaryStickerHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.TemporaryBadgeId &&
                                    c.Issuer == "https://microsoftsecurity"))
        {
            // We'd also check the expiration date on the sticker.
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        // .NET Framework older than 4.6:
        //     return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

Ensure that both handlers are [registered](#). If either handler succeeds when a policy evaluates the `BuildingEntryRequirement`, the policy evaluation succeeds.

Using a func to fulfill a policy

There may be situations in which fulfilling a policy is simple to express in code. It's possible to supply a `Func<AuthorizationHandlerContext, bool>` when configuring your policy with the `RequireAssertion` policy builder.

For example, the previous `BadgeEntryHandler` could be rewritten as follows:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context =>
            context.User.HasClaim(c =>
                (c.Type == ClaimTypes.BadgeId ||
                 c.Type == ClaimTypes.TemporaryBadgeId) &&
                 c.Issuer == "https://microsoftsecurity"))));
});
```

Accessing MVC request context in handlers

The `HandleRequirementAsync` method you implement in an authorization handler has two parameters: an `AuthorizationHandlerContext` and the `TRequirement` you are handling. Frameworks such as MVC or Jabbr are free to add any object to the `Resource` property on the `AuthorizationHandlerContext` to pass extra information.

For example, MVC passes an instance of `AuthorizationFilterContext` in the `Resource` property. This property provides access to `HttpContext`, `RouteData`, and everything else provided by MVC and Razor Pages.

The use of the `Resource` property is framework specific. Using information in the `Resource` property limits your authorization policies to particular frameworks. You should cast the `Resource` property using the `as` keyword, and then confirm the cast has succeeded to ensure your code doesn't crash with an `InvalidCastException` when run on other frameworks:

```
// Requires the following import:  
//     using Microsoft.AspNetCore.Mvc.Filters;  
if (context.Resource is AuthorizationFilterContext mvcContext)  
{  
    // Examine MVC-specific things like routing data.  
}
```

Dependency Injection in requirement handlers

11/1/2017 • 1 min to read • [Edit Online](#)

Authorization handlers must be registered in the service collection during configuration (using [dependency injection](#)).

Suppose you had a repository of rules you wanted to evaluate inside an authorization handler and that repository was registered in the service collection. Authorization will resolve and inject that into your constructor.

For example, if you wanted to use ASP.NET's logging infrastructure you would want to inject `ILoggerFactory` into your handler. Such a handler might look like:

```
public class LoggingAuthorizationHandler : AuthorizationHandler<MyRequirement>
{
    ILogger _logger;

    public LoggingAuthorizationHandler(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger(this.GetType().FullName);
    }

    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, MyRequirement requirement)
    {
        _logger.LogInformation("Inside my handler");
        // Check if the requirement is fulfilled.
        return Task.CompletedTask;
    }
}
```

You would register the handler with `services.AddSingleton()`:

```
services.AddSingleton<IAuthorizationHandler, LoggingAuthorizationHandler>();
```

An instance of the handler will be created when your application starts, and DI will inject the registered `ILoggerFactory` into your constructor.

NOTE

Handlers that use Entity Framework shouldn't be registered as singletons.

Resource-based authorization

11/9/2017 • 5 min to read • [Edit Online](#)

By [Scott Addie](#)

Authorization strategy depends upon the resource being accessed. Consider a document which has an author property. Only the author is allowed to update the document. Consequently, the document must be retrieved from the data store before authorization evaluation can occur.

Attribute evaluation occurs before data binding and before execution of the page handler or action which loads the document. For these reasons, declarative authorization with an `[Authorize]` attribute won't suffice. Instead, you can invoke a custom authorization method—a style known as imperative authorization.

Use the [sample apps \(how to download\)](#) to explore the features described in this topic.

Use imperative authorization

Authorization is implemented as an `IAuthorizationService` service and is registered in the service collection within the `Startup` class. The service is made available via [dependency injection](#) to page handlers or actions.

```
public class DocumentController : Controller
{
    private readonly IAuthorizationService _authorizationService;
    private readonly IDocumentRepository _documentRepository;

    public DocumentController(IAuthorizationService authorizationService,
                             IDocumentRepository documentRepository)
    {
        _authorizationService = authorizationService;
        _documentRepository = documentRepository;
    }
}
```

`IAuthorizationService` has two `AuthorizeAsync` method overloads: one accepting the resource and the policy name and the other accepting the resource and a list of requirements to evaluate.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                                         object resource,
                                         IEnumerable<IAuthorizationRequirement> requirements);
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                                         object resource,
                                         string policyName);
```

In the following example, the resource to be secured is loaded into a custom `Document` object. An `AuthorizeAsync` overload is invoked to determine whether the current user is allowed to edit the provided document. A custom "EditPolicy" authorization policy is factored into the decision. See [Custom policy-based authorization](#) for more on creating authorization policies.

NOTE

The following code samples assume authentication has run and set the `User` property.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
        .AuthorizeAsync(User, Document, "EditPolicy");

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

Write a resource-based handler

Writing a handler for resource-based authorization isn't much different than [writing a plain requirements handler](#). Create a custom requirement class, and implement a requirement handler class. The handler class specifies both the requirement and resource type. For example, a handler utilizing a `SameAuthorRequirement` requirement and a `Document` resource looks as follows:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        SameAuthorRequirement requirement,
        Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }

```

Register the requirement and handler in the `Startup.ConfigureServices` method:

```

services.AddMvc();

services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationCrudHandler>();
services.AddScoped<IDocumentRepository, DocumentRepository>();

```

Operational requirements

If you're making decisions based on the outcomes of CRUD (**C**reate, **R**ead, **U**ppdate, **D**ele) operations, use the [OperationAuthorizationRequirement](#) helper class. This class enables you to write a single handler instead of an individual class for each operation type. To use it, provide some operation names:

```

public static class Operations
{
    public static OperationAuthorizationRequirement Create =
        new OperationAuthorizationRequirement { Name = nameof(Create) };
    public static OperationAuthorizationRequirement Read =
        new OperationAuthorizationRequirement { Name = nameof(Read) };
    public static OperationAuthorizationRequirement Update =
        new OperationAuthorizationRequirement { Name = nameof(Update) };
    public static OperationAuthorizationRequirement Delete =
        new OperationAuthorizationRequirement { Name = nameof(Delete) };
}

```

The handler is implemented as follows, using an `OperationAuthorizationRequirement` requirement and a `Document` resource:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    OperationAuthorizationRequirement requirement,
                                                    Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

```

The preceding handler validates the operation using the resource, the user's identity, and the requirement's `Name` property.

To call an operational resource handler, specify the operation when invoking `AuthorizeAsync` in your page handler or action. The following example determines whether the authenticated user is permitted to view the provided document.

NOTE

The following code samples assume authentication has run and set the `User` property.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
        .AuthorizeAsync(User, Document, Operations.Read);

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}

```

If authorization succeeds, the page for viewing the document is returned. If authorization fails but the user is authenticated, returning `ForbidResult` informs any authentication middleware that authorization failed. A `ChallengeResult` is returned when authentication must be performed. For interactive browser clients, it may be

appropriate to redirect the user to a login page.

View-based authorization

11/9/2017 • 1 min to read • [Edit Online](#)

A developer often wants to show, hide, or otherwise modify a UI based on the current user identity. You can access the authorization service within MVC views via [dependency injection](#). To inject the authorization service into a Razor view, use the `@inject` directive:

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
```

If you want the authorization service in every view, place the `@inject` directive into the `_ViewImports.cshtml` file of the `Views` directory. For more information, see [Dependency injection into views](#).

Use the injected authorization service to invoke `AuthorizeAsync` in exactly the same way you would check during [resource-based authorization](#):

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
@if ((await AuthorizationService.AuthorizeAsync(User, "PolicyName")).Succeeded)
{
    <p>This paragraph is displayed because you fulfilled PolicyName.</p>
}
```

In some cases, the resource will be your view model. Invoke `AuthorizeAsync` in exactly the same way you would check during [resource-based authorization](#):

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
@if ((await AuthorizationService.AuthorizeAsync(User, Model, Operations.Edit)).Succeeded)
{
    <p><a class="btn btn-default" role="button"
        href="@Url.Action("Edit", "Document", new { id = Model.Id })">Edit</a></p>
}
```

In the preceding code, the model is passed as a resource the policy evaluation should take into consideration.

WARNING

Don't rely on toggling visibility of your app's UI elements as the sole authorization check. Hiding a UI element may not completely prevent access to its associated controller action. For example, consider the button in the preceding code snippet. A user can invoke the `Edit` action method if he or she knows the relative resource URL is `/Document/Edit/1`. For this reason, the `Edit` action method should perform its own authorization check.

Authorize with a specific scheme

10/30/2017 • 2 min to read • [Edit Online](#)

In some scenarios, such as Single Page Applications (SPAs), it's common to use multiple authentication methods. For example, the app may use cookie-based authentication to log in and JWT bearer authentication for JavaScript requests. In some cases, the app may have multiple instances of an authentication handler. For example, two cookie handlers where one contains a basic identity and one is created when a multi-factor authentication (MFA) has been triggered. MFA may be triggered because the user requested an operation that requires extra security.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

An authentication scheme is named when the authentication service is configured during authentication. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthentication()
        .AddCookie(options => {
            options.LoginPath = "/Account/Unauthorized/";
            options.AccessDeniedPath = "/Account/Forbidden/";
        })
        .AddJwtBearer(options => {
            options.Audience = "http://localhost:5001/";
            options.Authority = "http://localhost:5000/";
        });
}
```

In the preceding code, two authentication handlers have been added: one for cookies and one for bearer.

NOTE

Specifying the default scheme results in the `HttpContext.User` property being set to that identity. If that behavior isn't desired, disable it by invoking the parameterless form of `AddAuthentication`.

Selecting the scheme with the Authorize attribute

At the point of authorization, the app indicates the handler to be used. Select the handler with which the app will authorize by passing a comma-delimited list of authentication schemes to `[Authorize]`. The `[Authorize]` attribute specifies the authentication scheme or schemes to use regardless of whether a default is configured. For example:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
[Authorize(AuthenticationSchemes = AuthSchemes)]
public class MixedController : Controller
    // Requires the following imports:
    // using Microsoft.AspNetCore.Authentication.Cookies;
    // using Microsoft.AspNetCore.Authentication.JwtBearer;
    private const string AuthSchemes =
        CookieAuthenticationDefaults.AuthenticationScheme + "," +
        JwtBearerDefaults.AuthenticationScheme;
```

In the preceding example, both the cookie and bearer handlers run and have a chance to create and append an identity for the current user. By specifying a single scheme only, the corresponding handler runs.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
[Authorize(AuthenticationSchemes =
    JwtBearerDefaults.AuthenticationScheme)]
public class MixedController : Controller
```

In the preceding code, only the handler with the "Bearer" scheme runs. Any cookie-based identities are ignored.

Selecting the scheme with policies

If you prefer to specify the desired schemes in `policy`, you can set the `AuthenticationSchemes` collection when adding your policy:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy =>
    {
        policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new MinimumAgeRequirement());
    });
});
```

In the preceding example, the "Over18" policy only runs against the identity created by the "Bearer" handler. Use the policy by setting the `[Authorize]` attribute's `Policy` property:

```
[Authorize(Policy = "Over18")]
public class RegistrationController : Controller
```

Data Protection in ASP.NET Core: Consumer APIs, configuration, extensibility APIs and implementation

1/10/2018 • 1 min to read • [Edit Online](#)

- Introduction to data protection
- Get started with the Data Protection APIs
- Consumer APIs
 - Consumer APIs overview
 - Purpose strings
 - Purpose hierarchy and multi-tenancy
 - Password hashing
 - Limiting the lifetime of protected payloads
 - Unprotecting payloads whose keys have been revoked
- Configuration
 - Configuring data protection
 - Default settings
 - Machine-wide policy
 - Non DI-aware scenarios
- Extensibility APIs
 - Core cryptography extensibility
 - Key management extensibility
 - Miscellaneous APIs
- Implementation
 - Authenticated encryption details
 - Subkey derivation and authenticated encryption
 - Context headers
 - Key management
 - Key storage providers
 - Key encryption at rest
 - Key immutability and changing settings
 - Key storage format
 - Ephemeral data protection providers

- [Compatibility](#)
 - [Sharing cookies between apps](#)
 - [Replacing in ASP.NET](#)

Introduction to Data Protection

11/1/2017 • 5 min to read • [Edit Online](#)

Web applications often need to store security-sensitive data. Windows provides DPAPI for desktop applications but this is unsuitable for web applications. The ASP.NET Core data protection stack provide a simple, easy to use cryptographic API a developer can use to protect data, including key management and rotation.

The ASP.NET Core data protection stack is designed to serve as the long-term replacement for the element in ASP.NET 1.x - 4.x. It was designed to address many of the shortcomings of the old cryptographic stack while providing an out-of-the-box solution for the majority of use cases modern applications are likely to encounter.

Problem statement

The overall problem statement can be succinctly stated in a single sentence: I need to persist trusted information for later retrieval, but I do not trust the persistence mechanism. In web terms, this might be written as "I need to round-trip trusted state via an untrusted client."

The canonical example of this is an authentication cookie or bearer token. The server generates an "I am Groot and have xyz permissions" token and hands it to the client. At some future date the client will present that token back to the server, but the server needs some kind of assurance that the client hasn't forged the token. Thus the first requirement: authenticity (a.k.a. integrity, tamper-proofing).

Since the persisted state is trusted by the server, we anticipate that this state might contain information that is specific to the operating environment. This could be in the form of a file path, a permission, a handle or other indirect reference, or some other piece of server-specific data. Such information should generally not be disclosed to an untrusted client. Thus the second requirement: confidentiality.

Finally, since modern applications are componentized, what we've seen is that individual components will want to take advantage of this system without regard to other components in the system. For instance, if a bearer token component is using this stack, it should operate without interference from an anti-CSRF mechanism that might also be using the same stack. Thus the final requirement: isolation.

We can provide further constraints in order to narrow the scope of our requirements. We assume that all services operating within the cryptosystem are equally trusted and that the data does not need to be generated or consumed outside of the services under our direct control. Furthermore, we require that operations are as fast as possible since each request to the web service might go through the cryptosystem one or more times. This makes symmetric cryptography ideal for our scenario, and we can discount asymmetric cryptography until such a time that it is needed.

Design philosophy

We started by identifying problems with the existing stack. Once we had that, we surveyed the landscape of existing solutions and concluded that no existing solution quite had the capabilities we sought. We then engineered a solution based on several guiding principles.

- The system should offer simplicity of configuration. Ideally the system would be zero-configuration and developers could hit the ground running. In situations where developers need to configure a specific aspect (such as the key repository), consideration should be given to making those specific configurations simple.
- Offer a simple consumer-facing API. The APIs should be easy to use correctly and difficult to use incorrectly.
- Developers should not learn key management principles. The system should handle algorithm selection and

key lifetime on the developer's behalf. Ideally the developer should never even have access to the raw key material.

- Keys should be protected at rest when possible. The system should figure out an appropriate default protection mechanism and apply it automatically.

With these principles in mind we developed a simple, [easy to use](#) data protection stack.

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like [Windows CNG DPAPI](#) and [Azure Rights Management](#) are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data.

Audience

The data protection system is divided into five main packages. Various aspects of these APIs target three main audiences;

1. The [Consumer APIs Overview](#) target application and framework developers.

"I don't want to learn about how the stack operates or about how it is configured. I simply want to perform some operation in as simple a manner as possible with high probability of using the APIs successfully."

2. The [configuration APIs](#) target application developers and system administrators.

"I need to tell the data protection system that my environment requires non-default paths or settings."

3. The extensibility APIs target developers in charge of implementing custom policy. Usage of these APIs would be limited to rare situations and experienced, security aware developers.

"I need to replace an entire component within the system because I have truly unique behavioral requirements. I am willing to learn uncommonly-used parts of the API surface in order to build a plugin that fulfills my requirements."

Package Layout

The data protection stack consists of five packages.

- `Microsoft.AspNetCore.DataProtection.Abstractions` contains the basic `IDataProtectionProvider` and `IDataProtector` interfaces. It also contains useful extension methods that can assist working with these types (e.g., overloads of `IDataProtector.Protect`). See the consumer interfaces section for more information. If somebody else is responsible for instantiating the data protection system and you are simply consuming the APIs, you'll want to reference `Microsoft.AspNetCore.DataProtection.Abstractions`.
- `Microsoft.AspNetCore.DataProtection` contains the core implementation of the data protection system, including the core cryptographic operations, key management, configuration, and extensibility. If you're responsible for instantiating the data protection system (e.g., adding it to an `IServiceCollection`) or modifying or extending its behavior, you'll want to reference `Microsoft.AspNetCore.DataProtection`.
- `Microsoft.AspNetCore.DataProtection.Extensions` contains additional APIs which developers might find useful but which don't belong in the core package. For instance, this package contains a simple "instantiate the system pointing at a specific key storage directory with no dependency injection setup" API (more info). It also contains extension methods for limiting the lifetime of protected payloads (more info).
- `Microsoft.AspNetCore.DataProtection.SystemWeb` can be installed into an existing ASP.NET 4.x application to redirect its operations to instead use the new data protection stack. See [compatibility](#) for more information.

- Microsoft.AspNetCore.Cryptography.KeyDerivation provides an implementation of the PBKDF2 password hashing routine and can be used by systems which need to handle user passwords securely. See [Password Hashing](#) for more information.

Getting Started with the Data Protection APIs

11/1/2017 • 1 min to read • [Edit Online](#)

At its simplest, protecting data consists of the following steps:

1. Create a data protector from a data protection provider.
2. Call the `Protect` method with the data you want to protect.
3. Call the `Unprotect` method with the data you want to turn back into plain text.

Most frameworks and app models, such as ASP.NET or SignalR, already configure the data protection system and add it to a service container you access via dependency injection. The following sample demonstrates configuring a service container for dependency injection and registering the data protection stack, receiving the data protection provider via DI, creating a protector and protecting then unprotecting data

```

using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZh1ALTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */

```

When you create a protector you must provide one or more [Purpose Strings](#). A purpose string provides isolation between consumers. For example, a protector created with a purpose string of "green" would not be able to unprotect data provided by a protector with a purpose of "purple".

TIP

Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It is intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`.

A call to `Unprotect` will throw `CryptographicException` if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch `CryptographicException` instead of swallowing all exceptions.

Consumer APIs

11/1/2017 • 1 min to read • [Edit Online](#)

- [Consumer APIs Overview](#)
- [Purpose Strings](#)
- [Purpose hierarchy and multi-tenancy](#)
- [Password Hashing](#)
- [Limiting the lifetime of protected payloads](#)
- [Unprotecting payloads whose keys have been revoked](#)

Consumer APIs overview

11/1/2017 • 3 min to read • [Edit Online](#)

The `IDataProtectionProvider` and `IDataProtector` interfaces are the basic interfaces through which consumers use the data protection system. They are located in the [Microsoft.AspNetCore.DataProtection.Abstractions](#) package.

IDataProtectionProvider

The provider interface represents the root of the data protection system. It cannot directly be used to protect or unprotect data. Instead, the consumer must get a reference to an `IDataProtector` by calling `IDataProtectionProvider.CreateProtector(purpose)`, where `purpose` is a string that describes the intended consumer use case. See [Purpose Strings](#) for much more information on the intent of this parameter and how to choose an appropriate value.

IDataProtector

The protector interface is returned by a call to `CreateProtector`, and it is this interface which consumers can use to perform protect and unprotect operations.

To protect a piece of data, pass the data to the `Protect` method. The basic interface defines a method which converts `byte[]` -> `byte[]`, but there is also an overload (provided as an extension method) which converts `string` -> `string`. The security offered by the two methods is identical; the developer should choose whichever overload is most convenient for their use case. Irrespective of the overload chosen, the value returned by the `Protect` method is now protected (enciphered and tamper-proofed), and the application can send it to an untrusted client.

To unprotect a previously-protected piece of data, pass the protected data to the `Unprotect` method. (There are `byte[]`-based and `string`-based overloads for developer convenience.) If the protected payload was generated by an earlier call to `Protect` on this same `IDataProtector`, the `Unprotect` method will return the original unprotected payload. If the protected payload has been tampered with or was produced by a different `IDataProtector`, the `Unprotect` method will throw `CryptographicException`.

The concept of same vs. different `IDataProtector` ties back to the concept of purpose. If two `IDataProtector` instances were generated from the same root `IDataProtectionProvider` but via different purpose strings in the call to `IDataProtectionProvider.CreateProtector`, then they are considered [different protectors](#), and one will not be able to unprotect payloads generated by the other.

Consuming these interfaces

For a DI-aware component, the intended usage is that the component take an `IDataProtectionProvider` parameter in its constructor and that the DI system automatically provides this service when the component is instantiated.

NOTE

Some applications (such as console applications or ASP.NET 4.x applications) might not be DI-aware so cannot use the mechanism described here. For these scenarios consult the [Non DI Aware Scenarios](#) document for more information on getting an instance of an `IDataProtection` provider without going through DI.

The following sample demonstrates three concepts:

1. Adding the data protection system to the service container,
2. Using DI to receive an instance of an `IDataProtectionProvider`, and
3. Creating an `IDataProtector` from an `IDataProtectionProvider` and using it to protect and unprotect data.

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfdJ8ICcgQwZZh1AlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

The package `Microsoft.AspNetCore.DataProtection.Abstractions` contains an extension method `IServiceProvider.GetDataProtector` as a developer convenience. It encapsulates as a single operation both retrieving an `IDataProtectionProvider` from the service provider and calling `IDataProtectionProvider.CreateProtector`. The following sample demonstrates its usage.

```

using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // get an IDataProtector from the IServiceProvider
        var protector = services.GetDataProtector("Contoso.Example.v2");
        Console.WriteLine("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
    }
}

```

TIP

Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It is intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`. A call to `Unprotect` will throw `CryptographicException` if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch `CryptographicException` instead of swallowing all exceptions.

Purpose Strings

11/1/2017 • 3 min to read • [Edit Online](#)

Components which consume `IDataProtectionProvider` must pass a unique *purposes* parameter to the `CreateProtector` method. The purposes *parameter* is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root cryptographic keys are the same.

When a consumer specifies a purpose, the purpose string is used along with the root cryptographic keys to derive cryptographic subkeys unique to that consumer. This isolates the consumer from all other cryptographic consumers in the application: no other component can read its payloads, and it cannot read any other component's payloads. This isolation also renders infeasible entire categories of attack against the component.



In the diagram above, `IDataProtector` instances A and B **cannot** read each other's payloads, only their own.

The purpose string doesn't have to be secret. It should simply be unique in the sense that no other well-behaved component will ever provide the same purpose string.

TIP

Using the namespace and type name of the component consuming the data protection APIs is a good rule of thumb, as in practice this information will never conflict.

A Contoso-authored component which is responsible for minting bearer tokens might use `Contoso.Security.BearerToken` as its purpose string. Or - even better - it might use `Contoso.Security.BearerToken.v1` as its purpose string. Appending the version number allows a future version to use `Contoso.Security.BearerToken.v2` as its purpose, and the different versions would be completely isolated from one another as far as payloads go.

Since the purposes parameter to `CreateProtector` is a string array, the above could have been instead specified as `["Contoso.Security.BearerToken", "v1"]`. This allows establishing a hierarchy of purposes and opens up the possibility of multi-tenancy scenarios with the data protection system.

WARNING

Components should not allow untrusted user input to be the sole source of input for the purposes chain.

For example, consider a component `Contoso.Messaging.SecureMessage` which is responsible for storing secure messages. If the secure messaging component were to call `CreateProtector([username])`, then a malicious user might create an account with username `"Contoso.Security.BearerToken"` in an attempt to get the component to call `CreateProtector(["Contoso.Security.BearerToken"])`, thus inadvertently causing the secure messaging system to mint payloads that could be perceived as authentication tokens.

A better purposes chain for the messaging component would be

```
CreateProtector([ "Contoso.Messaging.SecureMessage", "User: username" ])
```

, which provides proper isolation.

The isolation provided by and behaviors of `IDataProtectionProvider`, `IDataProtector`, and purposes are as follows:

- For a given `IDataProtectionProvider` object, the `CreateProtector` method will create an `IDataProtector` object uniquely tied to both the `IDataProtectionProvider` object which created it and the purposes parameter which was passed into the method.
- The purpose parameter must not be null. (If purposes is specified as an array, this means that the array must not be of zero length and all elements of the array must be non-null.) An empty string purpose is technically allowed but is discouraged.
- Two purposes arguments are equivalent if and only if they contain the same strings (using an ordinal comparer) in the same order. A single purpose argument is equivalent to the corresponding single-element purposes array.
- Two `IDataProtector` objects are equivalent if and only if they are created from equivalent `IDataProtectionProvider` objects with equivalent purposes parameters.
- For a given `IDataProtector` object, a call to `Unprotect(protectedData)` will return the original `unprotectedData` if and only if `protectedData := Protect(unprotectedData)` for an equivalent `IDataProtector` object.

NOTE

We're not considering the case where some component intentionally chooses a purpose string which is known to conflict with another component. Such a component would essentially be considered malicious, and this system is not intended to provide security guarantees in the event that malicious code is already running inside of the worker process.

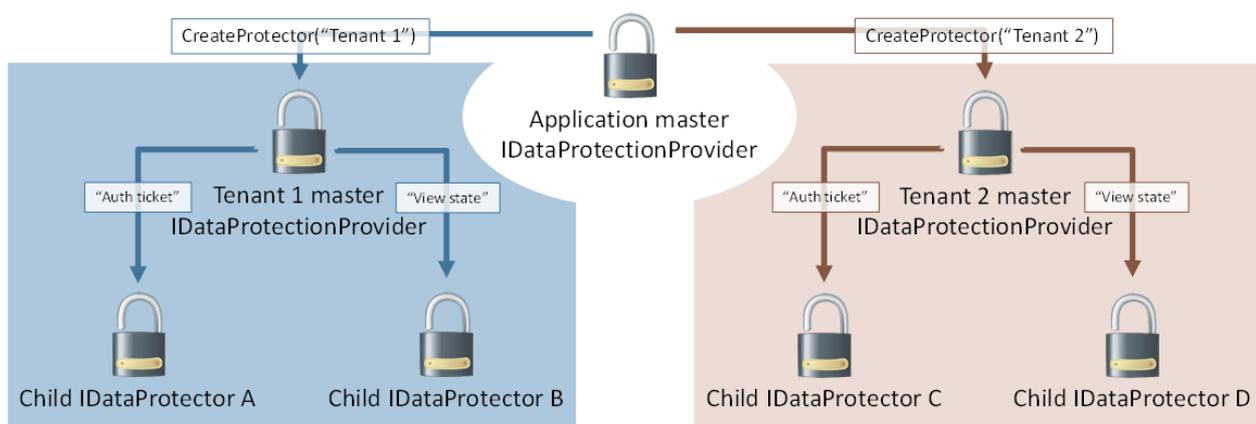
Purpose hierarchy and multi-tenancy in ASP.NET Core

11/1/2017 • 1 min to read • [Edit Online](#)

Since an `IDataProtector` is also implicitly an `IDataProtectionProvider`, purposes can be chained together. In this sense, `provider.CreateProtector(["purpose1", "purpose2"])` is equivalent to `provider.CreateProtector("purpose1").CreateProtector("purpose2")`.

This allows for some interesting hierarchical relationships through the data protection system. In the earlier example of [Contoso.Messaging.SecureMessage](#), the `SecureMessage` component can call `provider.CreateProtector("Contoso.Messaging.SecureMessage")` once up-front and cache the result into a private `_myProvide` field. Future protectors can then be created via calls to `_myProvider.CreateProtector("User: username")`, and these protectors would be used for securing the individual messages.

This can also be flipped. Consider a single logical application which hosts multiple tenants (a CMS seems reasonable), and each tenant can be configured with its own authentication and state management system. The umbrella application has a single master provider, and it calls `provider.CreateProtector("Tenant 1")` and `provider.CreateProtector("Tenant 2")` to give each tenant its own isolated slice of the data protection system. The tenants could then derive their own individual protectors based on their own needs, but no matter how hard they try they cannot create protectors which collide with any other tenant in the system. Graphically, this is represented as below.



WARNING

This assumes the umbrella application controls which APIs are available to individual tenants and that tenants cannot execute arbitrary code on the server. If a tenant can execute arbitrary code, they could perform private reflection to break the isolation guarantees, or they could just read the master keying material directly and derive whatever subkeys they desire.

The data protection system actually uses a sort of multi-tenancy in its default out-of-the-box configuration. By default master keying material is stored in the worker process account's user profile folder (or the registry, for IIS application pool identities). But it is actually fairly common to use a single account to run multiple applications, and thus all these applications would end up sharing the master keying material. To solve this, the data protection system automatically inserts a unique-per-application identifier as the first element in the overall purpose chain. This implicit purpose serves to [isolate individual applications](#) from one another by effectively treating each application as a unique tenant within the system, and the protector creation process looks identical to the image above.

Password Hashing

11/1/2017 • 1 min to read • [Edit Online](#)

The data protection code base includes a package *Microsoft.AspNetCore.Cryptography.KeyDerivation* which contains cryptographic key derivation functions. This package is a standalone component and has no dependencies on the rest of the data protection system. It can be used completely independently. The source exists alongside the data protection code base as a convenience.

The package currently offers a method `KeyDerivation.Pbkdf2` which allows hashing a password using the [PBKDF2 algorithm](#). This API is very similar to the .NET Framework's existing [Rfc2898DeriveBytes type](#), but there are three important distinctions:

1. The `KeyDerivation.Pbkdf2` method supports consuming multiple PRFs (currently `HMACSHA1`, `HMACSHA256`, and `HMACSHA512`), whereas the `Rfc2898DeriveBytes` type only supports `HMACSHA1`.
2. The `KeyDerivation.Pbkdf2` method detects the current operating system and attempts to choose the most optimized implementation of the routine, providing much better performance in certain cases. (On Windows 8, it offers around 10x the throughput of `Rfc2898DeriveBytes`.)
3. The `KeyDerivation.Pbkdf2` method requires the caller to specify all parameters (salt, PRF, and iteration count). The `Rfc2898DeriveBytes` type provides default values for these.

```

using System;
using System.Security.Cryptography;
using Microsoft.AspNetCore.Cryptography.KeyDerivation;

public class Program
{
    public static void Main(string[] args)
    {
        Console.Write("Enter a password: ");
        string password = Console.ReadLine();

        // generate a 128-bit salt using a secure PRNG
        byte[] salt = new byte[128 / 8];
        using (var rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(salt);
        }
        Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");

        // derive a 256-bit subkey (use HMACSHA1 with 10,000 iterations)
        string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
            password: password,
            salt: salt,
            prf: KeyDerivationPrf.HMACSHA1,
            iterationCount: 10000,
            numBytesRequested: 256 / 8));
        Console.WriteLine($"Hashed: {hashed}");
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter a password: Xtw9NMgx
 * Salt: NZsP6NnmfBuYeJrrAKNuVQ==
 * Hashed: /00o0er10+tGwTRDTrQSoeCxVTFr6dtYly7d0cPxIak=
 */

```

See the source code for ASP.NET Core Identity's `PasswordHasher` type for a real-world use case.

Limiting the lifetime of protected payloads

11/1/2017 • 2 min to read • [Edit Online](#)

There are scenarios where the application developer wants to create a protected payload that expires after a set period of time. For instance, the protected payload might represent a password reset token that should only be valid for one hour. It is certainly possible for the developer to create their own payload format that contains an embedded expiration date, and advanced developers may wish to do this anyway, but for the majority of developers managing these expirations can grow tedious.

To make this easier for our developer audience, the package [Microsoft.AspNetCore.DataProtection.Extensions](#) contains utility APIs for creating payloads that automatically expire after a set period of time. These APIs hang off of the `ITimeLimitedDataProtector` type.

API usage

The `ITimeLimitedDataProtector` interface is the core interface for protecting and unprotecting time-limited / self-expiring payloads. To create an instance of an `ITimeLimitedDataProtector`, you'll first need an instance of a regular `IDataProtector` constructed with a specific purpose. Once the `IDataProtector` instance is available, call the `IDataProtector.ToTimeLimitedDataProtector` extension method to get back a protector with built-in expiration capabilities.

`ITimeLimitedDataProtector` exposes the following API surface and extension methods:

- `CreateProtector(string purpose) : ITimeLimitedDataProtector` - This API is similar to the existing `IDataProtectionProvider.CreateProtector` in that it can be used to create [purpose chains](#) from a root time-limited protector.
- `Protect(byte[] plaintext, DateTimeOffset expiration) : byte[]`
- `Protect(byte[] plaintext, TimeSpan lifetime) : byte[]`
- `Protect(byte[] plaintext) : byte[]`
- `Protect(string plaintext, DateTimeOffset expiration) : string`
- `Protect(string plaintext, TimeSpan lifetime) : string`
- `Protect(string plaintext) : string`

In addition to the core `Protect` methods which take only the plaintext, there are new overloads which allow specifying the payload's expiration date. The expiration date can be specified as an absolute date (via a `DateTimeOffset`) or as a relative time (from the current system time, via a `TimeSpan`). If an overload which doesn't take an expiration is called, the payload is assumed never to expire.

- `Unprotect(byte[] protectedData, out DateTimeOffset expiration) : byte[]`
- `Unprotect(byte[] protectedData) : byte[]`
- `Unprotect(string protectedData, out DateTimeOffset expiration) : string`
- `Unprotect(string protectedData) : string`

The `Unprotect` methods return the original unprotected data. If the payload hasn't yet expired, the absolute expiration is returned as an optional out parameter along with the original unprotected data. If the payload is expired, all overloads of the `Unprotect` method will throw `CryptographicException`.

WARNING

It is not advised to use these APIs to protect payloads which require long-term or indefinite persistence. "Can I afford for the protected payloads to be permanently unrecoverable after a month?" can serve as a good rule of thumb; if the answer is no then developers should consider alternative APIs.

The sample below uses the [non-DI code paths](#) for instantiating the data protection system. To run this sample, ensure that you have first added a reference to the `Microsoft.AspNetCore.DataProtection.Extensions` package.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // create a protector for my application

        var provider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\myapp-keys\"));
        var baseProtector = provider.CreateProtector("Contoso.TimeLimitedSample");

        // convert the normal protector into a time-limited protector
        var timeLimitedProtector = baseProtector.ToTimeLimitedDataProtector();

        // get some input and protect it for five seconds
        Console.WriteLine("Enter input: ");
        string input = Console.ReadLine();
        string protectedData = timeLimitedProtector.Protect(input, lifetime: TimeSpan.FromSeconds(5));
        Console.WriteLine($"Protected data: {protectedData}");

        // unprotect it to demonstrate that round-tripping works properly
        string roundtripped = timeLimitedProtector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped data: {roundtripped}");

        // wait 6 seconds and perform another unprotect, demonstrating that the payload self-expires
        Console.WriteLine("Waiting 6 seconds...");
        Thread.Sleep(6000);
        timeLimitedProtector.Unprotect(protectedData);
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protected data: CfDJ8Hu5z0zwxn...nLk70k
 * Round-tripped data: Hello!
 * Waiting 6 seconds...
 * <<throws CryptographicException with message 'The payload expired at ...'>>
 */
```

Unprotecting payloads whose keys have been revoked

11/1/2017 • 3 min to read • [Edit Online](#)

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like [Windows CNG DPAPI](#) and [Azure Rights Management](#) are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data. Keys are never removed from the key ring, so `IDataProtector.Unprotect` can always recover existing payloads as long as the keys are available and valid.

However, an issue arises when the developer tries to unprotect data that has been protected with a revoked key, as `IDataProtector.Unprotect` will throw an exception in this case. This might be fine for short-lived or transient payloads (like authentication tokens), as these kinds of payloads can easily be recreated by the system, and at worst the site visitor might be required to log in again. But for persisted payloads, having `Unprotect` throw could lead to unacceptable data loss.

IPersistedDataProtector

To support the scenario of allowing payloads to be unprotected even in the face of revoked keys, the data protection system contains an `IPersistedDataProtector` type. To get an instance of `IPersistedDataProtector`, simply get an instance of `IDataProtector` in the normal fashion and try casting the `IDataProtector` to `IPersistedDataProtector`.

NOTE

Not all `IDataProtector` instances can be cast to `IPersistedDataProtector`. Developers should use the `C#` `as` operator or similar to avoid runtime exceptions caused by invalid casts, and they should be prepared to handle the failure case appropriately.

`IPersistedDataProtector` exposes the following API surface:

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,  
    out bool requiresMigration, out bool wasRevoked) : byte[]
```

This API takes the protected payload (as a byte array) and returns the unprotected payload. There is no string-based overload. The two out parameters are as follows.

- `requiresMigration`: will be set to true if the key used to protect this payload is no longer the active default key, e.g., the key used to protect this payload is old and a key rolling operation has since taken place. The caller may wish to consider reprotecting the payload depending on their business needs.
- `wasRevoked`: will be set to true if the key used to protect this payload was revoked.

WARNING

Exercise extreme caution when passing `ignoreRevocationErrors: true` to the `DangerousUnprotect` method. If after calling this method the `wasRevoked` value is true, then the key used to protect this payload was revoked, and the payload's authenticity should be treated as suspect. In this case, only continue operating on the unprotected payload if you have some separate assurance that it is authentic, e.g. that it's coming from a secure database rather than being sent by an untrusted web client.

```
using System;
using System.IO;
using System.Text;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // get a protector and perform a protect operation
        var protector = services.GetDataProtector("Sample.DangerousUnprotect");
        Console.WriteLine("Input: ");
        byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
        var protectedData = protector.Protect(input);
        Console.WriteLine($"Protected payload: {Convert.ToBase64String(protectedData)}");

        // demonstrate that the payload round-trips properly
        var roundTripped = protector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped payload: {Encoding.UTF8.GetString(roundTripped)}");

        // get a reference to the key manager and revoke all keys in the key ring
        var keyManager = services.GetService<IKeyManager>();
        Console.WriteLine("Revoking all keys in the key ring..");
        keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");

        // try calling Protect - this should throw
        Console.WriteLine("Calling Unprotect...");
        try
        {
            var unprotectedPayload = protector.Unprotect(protectedData);
            Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }

        // try calling DangerousUnprotect
        Console.WriteLine("Calling DangerousUnprotect...");
        try
        {
            IPersistedDataProtector persistedProtector = protector as IPersistedDataProtector;
            if (persistedProtector == null)
            {
                throw new Exception("Can't call DangerousUnprotect.");
            }

            bool requiresMigration, wasRevoked;
```

```
var unprotectedPayload = persistedProtector.DangerousUnprotect(
    protectedData: protectedData,
    ignoreRevocationErrors: true,
    requiresMigration: out requiresMigration,
    wasRevoked: out wasRevoked);
Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
Console.WriteLine($"Requires migration = {requiresMigration}, was revoked = {wasRevoked}");
}
catch (Exception ex)
{
    Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
}
}
}

/*
* SAMPLE OUTPUT
*
* Input: Hello!
* Protected payload: CfDJ8LHIzUCX1ZVBn2BZ...
* Round-tripped payload: Hello!
* Revoking all keys in the key ring...
* Calling Unprotect...
* CryptographicException: The key {...} has been revoked.
* Calling DangerousUnprotect...
* Unprotected payload: Hello!
* Requires migration = True, was revoked = True
*/
```

Data Protection configuration in ASP.NET Core

10/20/2017 • 1 min to read • [Edit Online](#)

Visit these topics to learn about Data Protection configuration in ASP.NET Core:

- [Configuring Data Protection](#)
An overview on configuring ASP.NET Core Data Protection.
- [Data Protection key management and lifetime](#)
Information on Data Protection key management and lifetime.
- [Data Protection machine-wide policy support](#)
Details on setting a default machine-wide policy for all apps that use Data Protection.
- [Non-DI aware scenarios for Data Protection in ASP.NET Core](#)
How to use the [DataProtectionProvider](#) concrete type to use Data Protection without going through DI-specific code paths.

Configuring Data Protection in ASP.NET Core

1/3/2018 • 7 min to read • [Edit Online](#)

By [Rick Anderson](#)

When the Data Protection system is initialized, it applies [default settings](#) based on the operational environment. These settings are generally appropriate for apps running on a single machine. There are cases where a developer may want to change the default settings, perhaps because their app is spread across multiple machines or for compliance reasons. For these scenarios, the Data Protection system offers a rich configuration API.

There's an extension method [AddDataProtection](#) that returns an [IDataProtectionBuilder](#). `IDataProtectionBuilder` exposes extension methods that you can chain together to configure Data Protection options.

PersistKeysToFileSystem

To store keys on a UNC share instead of at the `%LOCALAPPDATA%` default location, configure the system with [PersistKeysToFileSystem](#):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"));
}
```

WARNING

If you change the key persistence location, the system no longer automatically encrypts keys at rest, since it doesn't know whether DPAPI is an appropriate encryption mechanism.

ProtectKeysWith*

You can configure the system to protect keys at rest by calling any of the [ProtectKeysWith*](#) configuration APIs. Consider the example below, which stores keys on a UNC share and encrypts those keys at rest with a specific X.509 certificate:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
        .ProtectKeysWithCertificate("thumbprint");
}
```

See [Key Encryption At Rest](#) for more examples and discussion on the built-in key encryption mechanisms.

SetDefaultKeyLifetime

To configure the system to use a key lifetime of 14 days instead of the default 90 days, use [SetDefaultKeyLifetime](#):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
}
```

SetApplicationName

By default, the Data Protection system isolates apps from one another, even if they're sharing the same physical key repository. This prevents the apps from understanding each other's protected payloads. To share protected payloads between two apps, use [SetApplicationName](#) with the same value for each app:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetApplicationName("shared app name");
}
```

DisableAutomaticKeyGeneration

You may have a scenario where you don't want an app to automatically roll keys (create new keys) as they approach expiration. One example of this might be apps set up in a primary/secondary relationship, where only the primary app is responsible for key management concerns and secondary apps simply have a read-only view of the key ring. The secondary apps can be configured to treat the key ring as read-only by configuring the system with [DisableAutomaticKeyGeneration](#):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .DisableAutomaticKeyGeneration();
}
```

Per-application isolation

When the Data Protection system is provided by an ASP.NET Core host, it automatically isolates apps from one another, even if those apps are running under the same worker process account and are using the same master keying material. This is somewhat similar to the `IsolateApps` modifier from System.Web's `<machineKey>` element.

The isolation mechanism works by considering each app on the local machine as a unique tenant, thus the [IDataProtector](#) rooted for any given app automatically includes the app ID as a discriminator. The app's unique ID comes from one of two places:

1. If the app is hosted in IIS, the unique identifier is the app's configuration path. If an app is deployed in a web farm environment, this value should be stable assuming that the IIS environments are configured similarly across all machines in the web farm.
2. If the app isn't hosted in IIS, the unique identifier is the physical path of the app.

The unique identifier is designed to survive resets — both of the individual app and of the machine itself.

This isolation mechanism assumes that the apps are not malicious. A malicious app can always impact any other app running under the same worker process account. In a shared hosting environment where apps are mutually untrusted, the hosting provider should take steps to ensure OS-level isolation between apps, including

separating the apps' underlying key repositories.

If the Data Protection system isn't provided by an ASP.NET Core host (for example, if you instantiate it via the `DataProtectionProvider` concrete type) app isolation is disabled by default. When app isolation is disabled, all apps backed by the same keying material can share payloads as long as they provide the appropriate [purposes](#). To provide app isolation in this environment, call the [SetApplicationName](#) method on the configuration object and provide a unique name for each app.

Changing algorithms with `UseCryptographicAlgorithms`

The Data Protection stack allows you to change the default algorithm used by newly-generated keys. The simplest way to do this is to call [UseCryptographicAlgorithms](#) from the configuration callback:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.AddDataProtection()
    .UseCryptographicAlgorithms(
        new AuthenticatedEncryptorConfiguration()
        {
            EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
            ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
        });
```

The default `EncryptionAlgorithm` is AES-256-CBC, and the default `ValidationAlgorithm` is HMACSHA256. The default policy can be set by a system administrator via a [machine-wide policy](#), but an explicit call to `UseCryptographicAlgorithms` overrides the default policy.

Calling `UseCryptographicAlgorithms` allows you to specify the desired algorithm from a predefined built-in list. You don't need to worry about the implementation of the algorithm. In the scenario above, the Data Protection system attempts to use the CNG implementation of AES if running on Windows. Otherwise, it falls back to the managed [System.Security.Cryptography.Aes](#) class.

You can manually specify an implementation via a call to [UseCustomCryptographicAlgorithms](#).

TIP

Changing algorithms doesn't affect existing keys in the key ring. It only affects newly-generated keys.

Specifying custom managed algorithms

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

To specify custom managed algorithms, create a [ManagedAuthenticatedEncryptorConfiguration](#) instance that points to the implementation types:

```

serviceCollection.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new ManagedAuthenticatedEncryptorConfiguration()
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });

```

Generally the *Type properties must point to concrete, instantiable (via a public parameterless ctor) implementations of [SymmetricAlgorithm](#) and [KeyedHashAlgorithm](#), though the system special-cases some values like `typeof(Aes)` for convenience.

NOTE

The SymmetricAlgorithm must have a key length of ≥ 128 bits and a block size of ≥ 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The KeyedHashAlgorithm must have a digest size of ≥ 128 bits, and it must support keys of length equal to the hash algorithm's digest length. The KeyedHashAlgorithm is not strictly required to be HMAC.

Specifying custom Windows CNG algorithms

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

To specify a custom Windows CNG algorithm using CBC-mode encryption with HMAC validation, create a [CngCbcAuthenticatedEncryptorConfiguration](#) instance that contains the algorithmic information:

```

services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngCbcAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });

```

NOTE

The symmetric block cipher algorithm must have a key length of ≥ 128 bits, a block size of ≥ 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The hash algorithm must have a digest size of ≥ 128 bits and must support being opened with the BCRYPT_ALG_HANDLE_HMAC_FLAG flag. The *Provider properties can be set to null to use the default provider for the specified algorithm. See the [BCryptOpenAlgorithmProvider](#) documentation for more information.

- [ASP.NET Core 2.x](#)

- [ASP.NET Core 1.x](#)

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption with validation, create a [CngGcmAuthenticatedEncryptorConfiguration](#) instance that contains the algorithmic information:

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngGcmAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256
    });
```

NOTE

The symmetric block cipher algorithm must have a key length of ≥ 128 bits, a block size of exactly 128 bits, and it must support GCM encryption. You can set the [EncryptionAlgorithmProvider](#) property to null to use the default provider for the specified algorithm. See the [BCryptOpenAlgorithmProvider](#) documentation for more information.

Specifying other custom algorithms

Though not exposed as a first-class API, the Data Protection system is extensible enough to allow specifying almost any kind of algorithm. For example, it's possible to keep all keys contained within a Hardware Security Module (HSM) and to provide a custom implementation of the core encryption and decryption routines. See [IAuthenticatedEncryptor](#) in [Core cryptography extensibility](#) for more information.

Persisting keys when hosting in a Docker container

When hosting in a [Docker](#) container, keys should be maintained in either:

- A folder that's a Docker volume that persists beyond the container's lifetime, such as a shared volume or a host-mounted volume.
- An external provider, such as [Azure Key Vault](#) or [Redis](#).

See also

- [Non DI Aware Scenarios](#)
- [Machine Wide Policy](#)

Data Protection key management and lifetime in ASP.NET Core

10/20/2017 • 2 min to read • [Edit Online](#)

By [Rick Anderson](#)

Key management

The app attempts to detect its operational environment and handle key configuration on its own.

1. If the app is hosted in [Azure Apps](#), keys are persisted to the `%HOME%\ASP.NET\DataProtection-Keys` folder. This folder is backed by network storage and is synchronized across all machines hosting the app.
 - Keys aren't protected at rest.
 - The `DataProtection-Keys` folder supplies the key ring to all instances of an app in a single deployment slot.
 - Separate deployment slots, such as Staging and Production, don't share a key ring. When you swap between deployment slots, for example swapping Staging to Production or using A/B testing, any app using Data Protection won't be able to decrypt stored data using the key ring inside the previous slot. This leads to users being logged out of an app that uses the standard ASP.NET Core cookie authentication, as it uses Data Protection to protect its cookies. If you desire slot-independent key rings, use an external key ring provider, such as Azure Blob Storage, Azure Key Vault, a SQL store, or Redis cache.
2. If the user profile is available, keys are persisted to the `%LOCALAPPDATA%\ASP.NET\DataProtection-Keys` folder. If the operating system is Windows, the keys are encrypted at rest using DPAPI.
3. If the app is hosted in IIS, keys are persisted to the HKLM registry in a special registry key that is ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.
4. If none of these conditions match, keys aren't persisted outside of the current process. When the process shuts down, all generated keys are lost.

The developer is always in full control and can override how and where keys are stored. The first three options above should provide good defaults for most apps similar to how the ASP.NET **<machineKey>** auto-generation routines worked in the past. The final, fallback option is the only scenario that requires the developer to specify [configuration](#) upfront if they want key persistence, but this fallback only occurs in rare situations.

When hosting in a Docker container, keys should be persisted in a folder that's a Docker volume (a shared volume or a host-mounted volume that persists beyond the container's lifetime) or in an external provider, such as [Azure Key Vault](#) or [Redis](#). An external provider is also useful in web farm scenarios if apps can't access a shared network volume (see [PersistKeysToFileSystem](#) for more information).

WARNING

If the developer overrides the rules outlined above and points the Data Protection system at a specific key repository, automatic encryption of keys at rest is disabled. At-rest protection can be re-enabled via [configuration](#).

Key lifetime

Keys have a 90-day lifetime by default. When a key expires, the app automatically generates a new key and sets

the new key as the active key. As long as retired keys remain on the system, your app can decrypt any data protected with them. See [key management](#) for more information.

Default algorithms

The default payload protection algorithm used is AES-256-CBC for confidentiality and HMACSHA256 for authenticity. A 512-bit master key, changed every 90 days, is used to derive the two sub-keys used for these algorithms on a per-payload basis. See [subkey derivation](#) for more information.

See also

- [Key management extensibility](#)

Data Protection machine-wide policy support in ASP.NET Core

10/20/2017 • 3 min to read • [Edit Online](#)

By [Rick Anderson](#)

When running on Windows, the Data Protection system has limited support for setting a default machine-wide policy for all apps that consume ASP.NET Core Data Protection. The general idea is that an administrator might wish to change a default setting, such as the algorithms used or key lifetime, without the need to manually update every app on the machine.

WARNING

The system administrator can set default policy, but they can't enforce it. The app developer can always override any value with one of their own choosing. The default policy only affects apps where the developer hasn't specified an explicit value for a setting.

Setting default policy

To set default policy, an administrator can set known values in the system registry under the following registry key:

HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection

If you're on a 64-bit operating system and want to affect the behavior of 32-bit apps, remember to configure the Wow6432Node equivalent of the above key.

The supported values are shown below.

VALUE	TYPE	DESCRIPTION
EncryptionType	string	Specifies which algorithms should be used for data protection. The value must be CNG-CBC, CNG-GCM, or Managed and is described in more detail below.
DefaultKeyLifetime	DWORD	Specifies the lifetime for newly-generated keys. The value is specified in days and must be ≥ 7 .
KeyEscrowSinks	string	Specifies the types that are used for key escrow. The value is a semicolon-delimited list of key escrow sinks, where each element in the list is the assembly-qualified name of a type that implements IKeyEscrowSink .

Encryption types

If EncryptionType is CNG-CBC, the system is configured to use a CBC-mode symmetric block cipher for

confidentiality and HMAC for authenticity with services provided by Windows CNG (see [Specifying custom Windows CNG algorithms](#) for more details). The following additional values are supported, each of which corresponds to a property on the `CngCbcAuthenticatedEncryptionSettings` type.

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in CBC mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm <code>EncryptionAlgorithm</code> .
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.
HashAlgorithm	string	The name of a hash algorithm understood by CNG. This algorithm is opened in HMAC mode.
HashAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm <code>HashAlgorithm</code> .

If `EncryptionType` is `CNG-GCM`, the system is configured to use a Galois/Counter Mode symmetric block cipher for confidentiality and authenticity with services provided by Windows CNG (see [Specifying custom Windows CNG algorithms](#) for more details). The following additional values are supported, each of which corresponds to a property on the `CngGcmAuthenticatedEncryptionSettings` type.

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in Galois/Counter Mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm <code>EncryptionAlgorithm</code> .
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.

If `EncryptionType` is `Managed`, the system is configured to use a managed `SymmetricAlgorithm` for confidentiality and `KeyedHashAlgorithm` for authenticity (see [Specifying custom managed algorithms](#) for more details). The following additional values are supported, each of which corresponds to a property on the `ManagedAuthenticatedEncryptionSettings` type.

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithmType	string	The assembly-qualified name of a type that implements <code>SymmetricAlgorithm</code> .

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric encryption algorithm.
ValidationAlgorithmType	string	The assembly-qualified name of a type that implements KeyedHashAlgorithm.

If EncryptionType has any other value other than null or empty, the Data Protection system throws an exception at startup.

WARNING

When configuring a default policy setting that involves type names (EncryptionAlgorithmType, ValidationAlgorithmType, KeyEscrowSinks), the types must be available to the app. This means that for apps running on Desktop CLR, the assemblies that contain these types should be present in the Global Assembly Cache (GAC). For ASP.NET Core apps running on [.NET Core](#), the packages that contain these types should be installed.

Non-DI aware scenarios for Data Protection in ASP.NET Core

10/20/2017 • 2 min to read • [Edit Online](#)

By [Rick Anderson](#)

The ASP.NET Core Data Protection system is normally [added to a service container](#) and consumed by dependent components via dependency injection (DI). However, there are cases where this isn't feasible or desired, especially when importing the system into an existing app.

To support these scenarios, the [Microsoft.AspNetCore.DataProtection.Extensions](#) package provides a concrete type, [DataProtectionProvider](#), which offers a simple way to use Data Protection without relying on DI. The `DataProtectionProvider` type implements [IDataProtectionProvider](#). Constructing `DataProtectionProvider` only requires providing a [DirectoryInfo](#) instance to indicate where the provider's cryptographic keys should be stored, as seen in the following code sample:

```

using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder));

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.WriteLine("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfdJ8FWbAn6...ch3hAPm1NJA
 * Unprotect returned: Hello world!
 *
 * Press any key...
 */

```

By default, the `DataProtectionProvider` concrete type doesn't encrypt raw key material before persisting it to the file system. This is to support scenarios where the developer points to a network share and the Data Protection system can't automatically deduce an appropriate at-rest key encryption mechanism.

Additionally, the `DataProtectionProvider` concrete type doesn't [isolate apps](#) by default. All apps using the same key directory can share payloads as long as their [purpose parameters](#) match.

The `DataProtectionProvider` constructor accepts an optional configuration callback that can be used to adjust the behaviors of the system. The sample below demonstrates restoring isolation with an explicit call to [SetApplicationName](#). The sample also demonstrates configuring the system to automatically encrypt persisted keys using Windows DPAPI. If the directory points to a UNC share, you may wish to distribute a shared certificate across all relevant machines and to configure the system to use certificate-based encryption with a call to [ProtectKeysWithCertificate](#).

```

using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder),
            configuration =>
            {
                configuration.SetApplicationName("my app name");
                configuration.ProtectKeysWithDpapi();
            });

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.WriteLine("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

```

TIP

Instances of the `DataProtectionProvider` concrete type are expensive to create. If an app maintains multiple instances of this type and if they're all using the same key storage directory, app performance might degrade. If you use the `DataProtectionProvider` type, we recommend that you create this type once and reuse it as much as possible. The `DataProtectionProvider` type and all `IDataProtector` instances created from it are thread-safe for multiple callers.

Extensibility APIs

11/1/2017 • 1 min to read • [Edit Online](#)

- [Core cryptography extensibility](#)
- [Key management extensibility](#)
- [Miscellaneous APIs](#)

Core cryptography extensibility

11/1/2017 • 5 min to read • [Edit Online](#)

WARNING

Types that implement any of the following interfaces should be thread-safe for multiple callers.

IAuthenticatedEncryptor

The **IAuthenticatedEncryptor** interface is the basic building block of the cryptographic subsystem. There is generally one IAuthenticatedEncryptor per key, and the IAuthenticatedEncryptor instance wraps all cryptographic key material and algorithmic information necessary to perform cryptographic operations.

As its name suggests, the type is responsible for providing authenticated encryption and decryption services. It exposes the following two APIs.

- Decrypt(ArraySegment ciphertext, ArraySegment additionalAuthenticatedData) : byte[]
- Encrypt(ArraySegment plaintext, ArraySegment additionalAuthenticatedData) : byte[]

The Encrypt method returns a blob that includes the enciphered plaintext and an authentication tag. The authentication tag must encompass the additional authenticated data (AAD), though the AAD itself need not be recoverable from the final payload. The Decrypt method validates the authentication tag and returns the deciphered payload. All failures (except ArgumentException and similar) should be homogenized to CryptographicException.

NOTE

The IAuthenticatedEncryptor instance itself doesn't actually need to contain the key material. For example, the implementation could delegate to an HSM for all operations.

How to create an IAuthenticatedEncryptor

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The **IAuthenticatedEncryptorFactory** interface represents a type that knows how to create an [IAuthenticatedEncryptor](#) instance. Its API is as follows.

- CreateEncryptorInstance(IKey key) : IAuthenticatedEncryptor

For any given IKey instance, any authenticated encryptors created by its CreateEncryptorInstance method should be considered equivalent, as in the below code sample.

```
// we have an IAuthenticatedEncryptorFactory instance and an IKey instance
IAuthenticatedEncryptorFactory factory = ...;
IKey key = ...;

// get an encryptor instance and perform an authenticated encryption operation
ArraySegment<byte> plaintext = new ArraySegment<byte>(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = factory.CreateEncryptorInstance(key);
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated decryption operation
var encryptor2 = factory.CreateEncryptorInstance(key);
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>(ciphertext), aad);

// the 'roundTripped' and 'plaintext' buffers should be equivalent
```

IAuthenticatedEncryptorDescriptor (ASP.NET Core 2.x only)

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The **IAuthenticatedEncryptorDescriptor** interface represents a type that knows how to export itself to XML. Its API is as follows.

- `ExportToXml(): XmlSerializedDescriptorInfo`

XML Serialization

The primary difference between `IAuthenticatedEncryptor` and `IAuthenticatedEncryptorDescriptor` is that the descriptor knows how to create the encryptor and supply it with valid arguments. Consider an `IAuthenticatedEncryptor` whose implementation relies on `SymmetricAlgorithm` and `KeyedHashAlgorithm`. The encryptor's job is to consume these types, but it doesn't necessarily know where these types came from, so it can't really write out a proper description of how to recreate itself if the application restarts. The descriptor acts as a higher level on top of this. Since the descriptor knows how to create the encryptor instance (e.g., it knows how to create the required algorithms), it can serialize that knowledge in XML form so that the encryptor instance can be recreated after an application reset.

The descriptor can be serialized via its `ExportToXml` routine. This routine returns an `XmlSerializedDescriptorInfo` which contains two properties: the `XElement` representation of the descriptor and the `Type` which represents an [IAuthenticatedEncryptorDescriptorDeserializer](#) which can be used to resurrect this descriptor given the corresponding `XElement`.

The serialized descriptor may contain sensitive information such as cryptographic key material. The data protection system has built-in support for encrypting information before it's persisted to storage. To take advantage of this, the descriptor should mark the element which contains sensitive information with the attribute name `requiresEncryption` (xmlns `http://schemas.asp.net/2015/03/dataProtection`), value `true`.

TIP

There's a helper API for setting this attribute. Call the extension method `XElement.MarkAsRequiresEncryption()` located in namespace `Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel`.

There can also be cases where the serialized descriptor doesn't contain sensitive information. Consider again the case of a cryptographic key stored in an HSM. The descriptor cannot write out the key material when serializing itself since the HSM will not expose the material in plaintext form. Instead, the descriptor might write out the key-

wrapped version of the key (if the HSM allows export in this fashion) or the HSM's own unique identifier for the key.

IAuthenticatedEncryptorDescriptorDeserializer

The **IAuthenticatedEncryptorDescriptorDeserializer** interface represents a type that knows how to deserialize an `IAuthenticatedEncryptorDescriptor` instance from an `XElement`. It exposes a single method:

- `ImportFromXml(XElement element) : IAuthenticatedEncryptorDescriptor`

The `ImportFromXml` method takes the `XElement` that was returned by [IAuthenticatedEncryptorDescriptor.ExportToXml](#) and creates an equivalent of the original `IAuthenticatedEncryptorDescriptor`.

Types which implement `IAuthenticatedEncryptorDescriptorDeserializer` should have one of the following two public constructors:

- `.ctor(IServiceProvider)`
- `.ctor()`

NOTE

The `IServiceProvider` passed to the constructor may be null.

The top-level factory

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

The **AlgorithmConfiguration** class represents a type which knows how to create [IAuthenticatedEncryptorDescriptor](#) instances. It exposes a single API.

- `CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor`

Think of `AlgorithmConfiguration` as the top-level factory. The configuration serves as a template. It wraps algorithmic information (e.g., this configuration produces descriptors with an AES-128-GCM master key), but it is not yet associated with a specific key.

When `CreateNewDescriptor` is called, fresh key material is created solely for this call, and a new `IAuthenticatedEncryptorDescriptor` is produced which wraps this key material and the algorithmic information required to consume the material. The key material could be created in software (and held in memory), it could be created and held within an HSM, and so on. The crucial point is that any two calls to `CreateNewDescriptor` should never create equivalent `IAuthenticatedEncryptorDescriptor` instances.

The `AlgorithmConfiguration` type serves as the entry point for key creation routines such as [automatic key rolling](#). To change the implementation for all future keys, set the `AuthenticatedEncryptorConfiguration` property in `KeyManagementOptions`.

Key management extensibility

11/23/2017 • 7 min to read • [Edit Online](#)

TIP

Read the [key management](#) section before reading this section, as it explains some of the fundamental concepts behind these APIs.

WARNING

Types that implement any of the following interfaces should be thread-safe for multiple callers.

Key

The `IKey` interface is the basic representation of a key in cryptosystem. The term key is used here in the abstract sense, not in the literal sense of "cryptographic key material". A key has the following properties:

- Activation, creation, and expiration dates
- Revocation status
- Key identifier (a GUID)
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Additionally, `IKey` exposes a `CreateEncryptor` method which can be used to create an [IAuthenticatedEncryptor](#) instance tied to this key.

NOTE

There is no API to retrieve the raw cryptographic material from an `IKey` instance.

IKeyManager

The `IKeyManager` interface represents an object responsible for general key storage, retrieval, and manipulation. It exposes three high-level operations:

- Create a new key and persist it to storage.
- Get all keys from storage.
- Revoke one or more keys and persist the revocation information to storage.

WARNING

Writing an `IKeyManager` is a very advanced task, and the majority of developers should not attempt it. Instead, most developers should take advantage of the facilities offered by the [XmlKeyManager](#) class.

XmlKeyManager

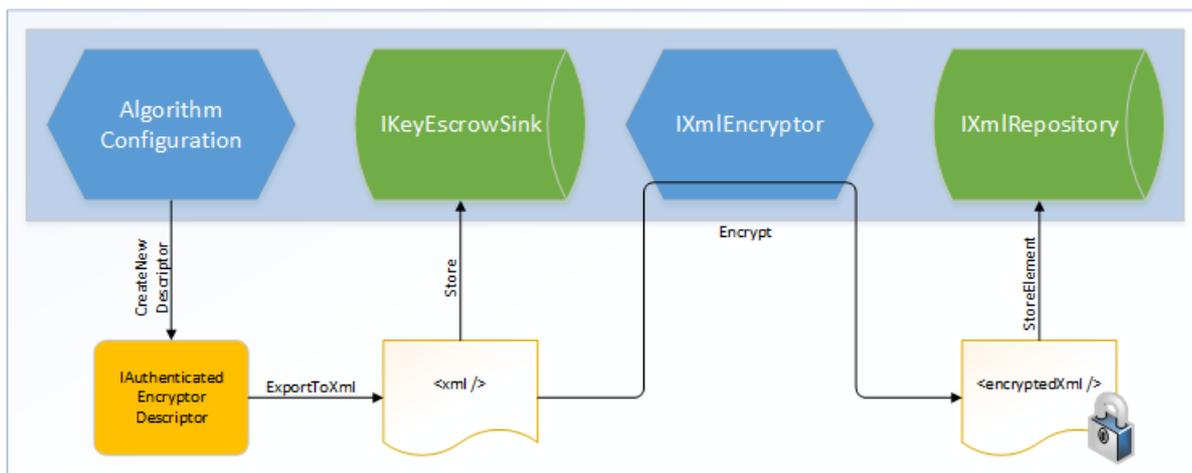
The `XmlKeyManager` type is the in-box concrete implementation of `IKeyManager`. It provides several useful facilities, including key escrow and encryption of keys at rest. Keys in this system are represented as XML elements (specifically, `XElement`).

`XmlKeyManager` depends on several other components in the course of fulfilling its tasks:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)
- `AlgorithmConfiguration`, which dictates the algorithms used by new keys.
- `IXmlRepository`, which controls where keys are persisted in storage.
- `IXmlEncryptor` [optional], which allows encrypting keys at rest.
- `IKeyEscrowSink` [optional], which provides key escrow services.

Below are high-level diagrams which indicate how these components are wired together within `XmlKeyManager`.

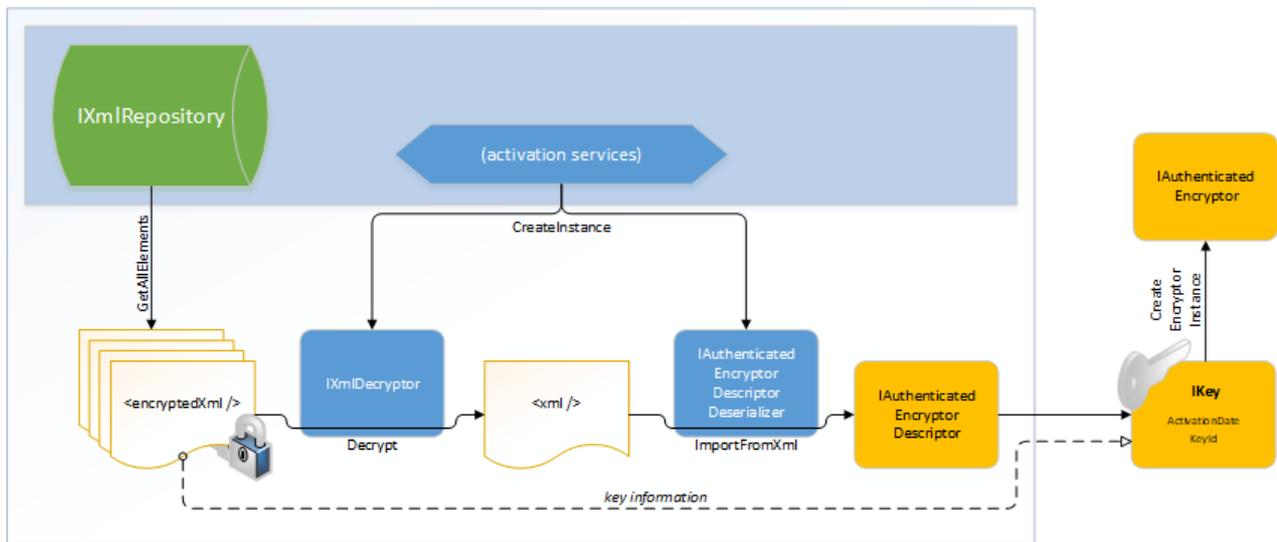
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)



Key Creation / CreateNewKey

In the implementation of `CreateNewKey`, the `AlgorithmConfiguration` component is used to create a unique `IAuthenticatedEncryptorDescriptor`, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an `IXmlEncryptor` (if required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the `IXmlRepository`. (If no `IXmlEncryptor` is configured, the unencrypted document is persisted in the `IXmlRepository`.)

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)



Key Retrieval / GetAllKeys

In the implementation of `GetAllKeys`, the XML documents representing keys and revocations are read from the underlying `IXmlRepository`. If these documents are encrypted, the system will automatically decrypt them. `XmlKeyManager` creates the appropriate `IAuthenticatedEncryptorDescriptorDeserializer` instances to deserialize the documents back into `IAuthenticatedEncryptorDescriptor` instances, which are then wrapped in individual `IKey` instances. This collection of `IKey` instances is returned to the caller.

Further information on the particular XML elements can be found in the [key storage format document](#).

IXmlRepository

The `IXmlRepository` interface represents a type that can persist XML to and retrieve XML from a backing store. It exposes two APIs:

- `GetAllElements() : IReadOnlyCollection`
- `StoreElement(XElement element, string friendlyName)`

Implementations of `IXmlRepository` don't need to parse the XML passing through them. They should treat the XML documents as opaque and let higher layers worry about generating and parsing the documents.

There are two built-in concrete types which implement `IXmlRepository`: `FileSystemXmlRepository` and `RegistryXmlRepository`. See the [key storage providers document](#) for more information. Registering a custom `IXmlRepository` would be the appropriate manner to use a different backing store, e.g., Azure Blob Storage.

To change the default repository application-wide, register a custom `IXmlRepository` instance:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.Configure<KeyManagementOptions>(options => options.XmlRepository = new MyCustomXmlRepository());
```

IXmlEncryptor

The `IXmlEncryptor` interface represents a type that can encrypt a plaintext XML element. It exposes a single API:

- `Encrypt(XElement plaintextElement) : EncryptedXmlInfo`

If a serialized `IAuthenticatedEncryptorDescriptor` contains any elements marked as "requires encryption", then `XmlKeyManager` will run those elements through the configured `IXmlEncryptor`'s `Encrypt` method, and it will

persist the enciphered element rather than the plaintext element to the `IXmlRepository`. The output of the `Encrypt` method is an `EncryptedXmlInfo` object. This object is a wrapper which contains both the resultant enciphered `XElement` and the Type which represents an `IXmlDecryptor` which can be used to decipher the corresponding element.

There are four built-in concrete types which implement `IXmlEncryptor`:

- `CertificateXmlEncryptor`
- `DpapiNGXmlEncryptor`
- `DpapiXmlEncryptor`
- `NullXmlEncryptor`

See the [key encryption at rest document](#) for more information.

To change the default key-encryption-at-rest mechanism application-wide, register a custom `IXmlEncryptor` instance:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.Configure<KeyManagementOptions>(options => options.XmlEncryptor = new MyCustomXmlEncryptor());
```

IXmlDecryptor

The `IXmlDecryptor` interface represents a type that knows how to decrypt an `XElement` that was enciphered via an `IXmlEncryptor`. It exposes a single API:

- `Decrypt(XElement encryptedElement) : XElement`

The `Decrypt` method undoes the encryption performed by `IXmlEncryptor.Encrypt`. Generally, each concrete `IXmlEncryptor` implementation will have a corresponding concrete `IXmlDecryptor` implementation.

Types which implement `IXmlDecryptor` should have one of the following two public constructors:

- `.ctor(IServiceProvider)`
- `.ctor()`

NOTE

The `IServiceProvider` passed to the constructor may be null.

IKeyEscrowSink

The `IKeyEscrowSink` interface represents a type that can perform escrow of sensitive information. Recall that serialized descriptors might contain sensitive information (such as cryptographic material), and this is what led to the introduction of the `IXmlEncryptor` type in the first place. However, accidents happen, and key rings can be deleted or become corrupted.

The escrow interface provides an emergency escape hatch, allowing access to the raw serialized XML before it is transformed by any configured `IXmlEncryptor`. The interface exposes a single API:

- `Store(Guid keyId, XElement element)`

It is up to the `IKeyEscrowSink` implementation to handle the provided element in a secure manner consistent with business policy. One possible implementation could be for the escrow sink to encrypt the XML element using a

known corporate X.509 certificate where the certificate's private key has been escrowed; the `CertificateXmlEncryptor` type can assist with this. The `IKeyEscrowSink` implementation is also responsible for persisting the provided element appropriately.

By default no escrow mechanism is enabled, though server administrators can [configure this globally](#). It can also be configured programmatically via the `IDataProtectionBuilder.AddKeyEscrowSink` method as shown in the sample below. The `AddKeyEscrowSink` method overloads mirror the `IServiceCollection.AddSingleton` and `IServiceCollection.AddInstance` overloads, as `IKeyEscrowSink` instances are intended to be singletons. If multiple `IKeyEscrowSink` instances are registered, each one will be called during key generation, so keys can be escrowed to multiple mechanisms simultaneously.

There is no API to read material from an `IKeyEscrowSink` instance. This is consistent with the design theory of the escrow mechanism: it's intended to make the key material accessible to a trusted authority, and since the application is itself not a trusted authority, it shouldn't have access to its own escrowed material.

The following sample code demonstrates creating and registering an `IKeyEscrowSink` where keys are escrowed such that only members of "CONTOSO\Domain Admins" can recover them.

NOTE

To run this sample, you must be on a domain-joined Windows 8 / Windows Server 2012 machine, and the domain controller must be Windows Server 2012 or later.

```
using System;
using System.IO;
using System.Xml.Linq;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.XmlEncryption;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi()
            .AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
        var services = serviceCollection.BuildServiceProvider();

        // get a reference to the key manager and force a new key to be generated
        Console.WriteLine("Generating new key...");
        var keyManager = services.GetService<IKeyManager>();
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddDays(7));
    }

    // A key escrow sink where keys are escrowed such that they
    // can be read by members of the CONTOSO\Domain Admins group.
    private class MyKeyEscrowSink : IKeyEscrowSink
    {
        private readonly IXmlEncryptor _escrowEncryptor;

        public MyKeyEscrowSink(IServiceProvider services)
        {
            // Assuming I'm on a machine that's a member of the CONTOSO
            // domain, I can use the Domain Admins SID to generate an
            // encrypted payload that only they can read. Sample SID from
            // https://technet.microsoft.com/library/cc778824(v=ws.10).aspx.
```

```

// ...
_escrowEncryptor = new DpapiNGXmlEncryptor(
    "SID=S-1-5-21-1004336348-1177238915-682003330-512",
    DpapiNGProtectionDescriptorFlags.None,
    services);
}

public void Store(Guid keyId, XElement element)
{
    // Encrypt the key element to the escrow encryptor.
    var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);

    // A real implementation would save the escrowed key to a
    // write-only file share or some other stable storage, but
    // in this sample we'll just write it out to the console.
    Console.WriteLine($"Escrowing key {keyId}");
    Console.WriteLine(encryptedXmlInfo.EncryptedElement);

    // Note: We cannot read the escrowed key material ourselves.
    // We need to get a member of CONTOSO\Domain Admins to read
    // it for us in the event we need to recover it.
}
}
}

/*
* SAMPLE OUTPUT
*
* Generating new key...
* Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
* <encryptedKey>
* <!-- This key is encrypted with Windows DPAPI-NG. -->
* <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
* <value>MIIIfAYJKoZIhvcNAQcDoIIbTCCGkCAQ...T5rA4g==</value>
* </encryptedKey>
*/

```

Miscellaneous APIs

11/1/2017 • 1 min to read • [Edit Online](#)

WARNING

Types that implement any of the following interfaces should be thread-safe for multiple callers.

ISecret

The `ISecret` interface represents a secret value, such as cryptographic key material. It contains the following API surface:

- `Length` : `int`
- `Dispose()` : `void`
- `WriteSecretIntoBuffer(ArraySegment<byte> buffer)` : `void`

The `WriteSecretIntoBuffer` method populates the supplied buffer with the raw secret value. The reason this API takes the buffer as a parameter rather than returning a `byte[]` directly is that this gives the caller the opportunity to pin the buffer object, limiting secret exposure to the managed garbage collector.

The `Secret` type is a concrete implementation of `ISecret` where the secret value is stored in in-process memory. On Windows platforms, the secret value is encrypted via [CryptProtectMemory](#).

Implementation

11/1/2017 • 1 min to read • [Edit Online](#)

- [Authenticated encryption details](#)
- [Subkey Derivation and Authenticated Encryption](#)
- [Context headers](#)
- [Key Management](#)
- [Key Storage Providers](#)
- [Key Encryption At Rest](#)
- [Key Immutability and Changing Settings](#)
- [Key Storage Format](#)
- [Ephemeral data protection providers](#)

Authenticated encryption details

11/1/2017 • 2 min to read • [Edit Online](#)

Calls to `IDataProtector.Protect` are authenticated encryption operations. The `Protect` method offers both confidentiality and authenticity, and it is tied to the purpose chain that was used to derive this particular `IDataProtector` instance from its root `IDataProtectionProvider`.

`IDataProtector.Protect` takes a `byte[]` plaintext parameter and produces a `byte[]` protected payload, whose format is described below. (There is also an extension method overload which takes a string plaintext parameter and returns a string protected payload. If this API is used the protected payload format will still have the below structure, but it will be [base64url-encoded](#).)

Protected payload format

The protected payload format consists of three primary components:

- A 32-bit magic header that identifies the version of the data protection system.
- A 128-bit key id that identifies the key used to protect this particular payload.
- The remainder of the protected payload is [specific to the encryptor encapsulated by this key](#). In the example below the key represents an AES-256-CBC + HMACSHA256 encryptor, and the payload is further subdivided as follows: * A 128-bit key modifier. * A 128-bit initialization vector. * 48 bytes of AES-256-CBC output. * An HMACSHA256 authentication tag.

A sample protected payload is illustrated below.

```
09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
52 C9 74 A0
```

From the payload format above the first 32 bits, or 4 bytes are the magic header identifying the version (09 F0 C9 F0)

The next 128 bits, or 16 bytes is the key identifier (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

The remainder contains the payload and is specific to the format used.

WARNING

All payloads protected to a given key will begin with the same 20-byte (magic value, key id) header. Administrators can use this fact for diagnostic purposes to approximate when a payload was generated. For example, the payload above corresponds to key {0c819c80-6619-4019-9536-53f8aaffee57}. If after checking the key repository you find that this specific key's activation date was 2015-01-01 and its expiration date was 2015-03-01, then it is reasonable to assume that the payload (if not tampered with) was generated within that window, give or take a small fudge factor on either side.

Subkey Derivation and Authenticated Encryption

11/1/2017 • 3 min to read • [Edit Online](#)

Most keys in the key ring will contain some form of entropy and will have algorithmic information stating "CBC-mode encryption + HMAC validation" or "GCM encryption + validation". In these cases, we refer to the embedded entropy as the master keying material (or KM) for this key, and we perform a key derivation function to derive the keys that will be used for the actual cryptographic operations.

NOTE

Keys are abstract, and a custom implementation might not behave as below. If the key provides its own implementation of `IAuthenticatedEncryptor` rather than using one of our built-in factories, the mechanism described in this section no longer applies.

Additional authenticated data and subkey derivation

The `IAuthenticatedEncryptor` interface serves as the core interface for all authenticated encryption operations. Its `Encrypt` method takes two buffers: plaintext and `additionalAuthenticatedData` (AAD). The plaintext contents flow unchanged the call to `IDataProtector.Protect`, but the AAD is generated by the system and consists of three components:

1. The 32-bit magic header 09 F0 C9 F0 that identifies this version of the data protection system.
2. The 128-bit key id.
3. A variable-length string formed from the purpose chain that created the `IDataProtector` that is performing this operation.

Because the AAD is unique for the tuple of all three components, we can use it to derive new keys from KM instead of using KM itself in all of our cryptographic operations. For every call to `IAuthenticatedEncryptor.Encrypt`, the following key derivation process takes place:

$$(K_E, K_H) = \text{SP800_108_CTR_HMACSHA512}(K_M, \text{AAD}, \text{contextHeader} \parallel \text{keyModifier})$$

Here, we're calling the NIST SP800-108 KDF in Counter Mode (see [NIST SP800-108](#), Sec. 5.1) with the following parameters:

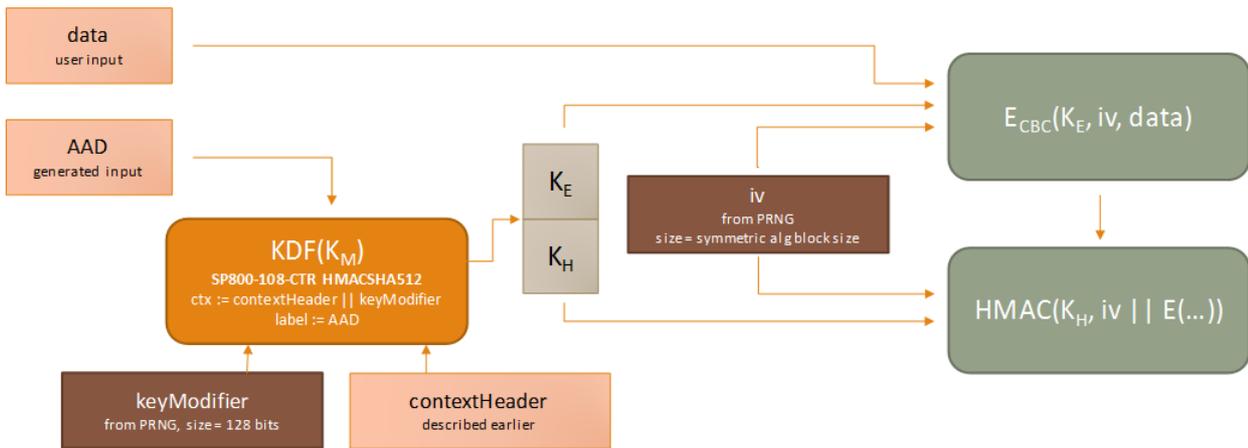
- Key derivation key (KDK) = `K_M`
- PRF = HMACSHA512
- label = `additionalAuthenticatedData`
- context = `contextHeader` \parallel `keyModifier`

The context header is of variable length and essentially serves as a thumbprint of the algorithms for which we're deriving `K_E` and `K_H`. The key modifier is a 128-bit string randomly generated for each call to `Encrypt` and serves to ensure with overwhelming probability that KE and KH are unique for this specific authentication encryption operation, even if all other input to the KDF is constant.

For CBC-mode encryption + HMAC validation operations, $|K_E|$ is the length of the symmetric block cipher key, and $|K_H|$ is the digest size of the HMAC routine. For GCM encryption + validation operations, $|K_H| = 0$.

CBC-mode encryption + HMAC validation

Once K_E is generated via the above mechanism, we generate a random initialization vector and run the symmetric block cipher algorithm to encipher the plaintext. The initialization vector and ciphertext are then run through the HMAC routine initialized with the key K_H to produce the MAC. This process and the return value is represented graphically below.



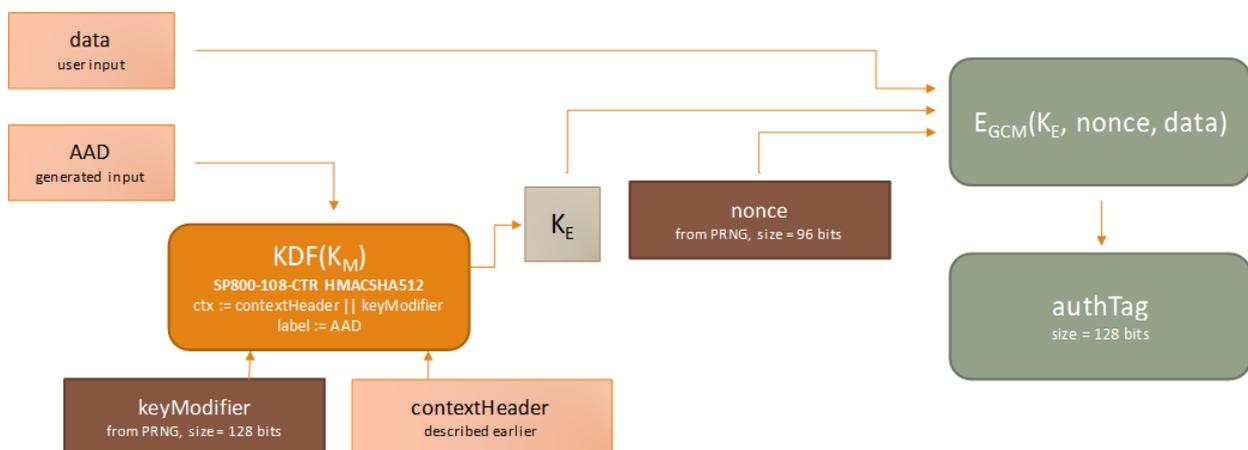
$output := keyModifier || iv || E_{cbc}(K_E, iv, data) || HMAC(K_H, iv || E_{cbc}(K_E, iv, data))$

NOTE

The `IDataProtector.Protect` implementation will [prepend the magic header and key id](#) to output before returning it to the caller. Because the magic header and key id are implicitly part of AAD, and because the key modifier is fed as input to the KDF, this means that every single byte of the final returned payload is authenticated by the MAC.

Galois/Counter Mode encryption + validation

Once K_E is generated via the above mechanism, we generate a random 96-bit nonce and run the symmetric block cipher algorithm to encipher the plaintext and produce the 128-bit authentication tag.



$output := keyModifier || nonce || E_{gcm}(K_E, nonce, data) || authTag$

NOTE

Even though GCM natively supports the concept of AAD, we're still feeding AAD only to the original KDF, opting to pass an empty string into GCM for its AAD parameter. The reason for this is two-fold. First, [to support agility](#) we never want to use `K_M` directly as the encryption key. Additionally, GCM imposes very strict uniqueness requirements on its inputs. The probability that the GCM encryption routine is ever invoked on two or more distinct sets of input data with the same (key, nonce) pair must not exceed 2^{-32} . If we fix `K_E` we cannot perform more than 2^{32} encryption operations before we run afoul of the 2^{-32} limit. This might seem like a very large number of operations, but a high-traffic web server can go through 4 billion requests in mere days, well within the normal lifetime for these keys. To stay compliant of the 2^{-32} probability limit, we continue to use a 128-bit key modifier and 96-bit nonce, which radically extends the usable operation count for any given `K_M`. For simplicity of design we share the KDF code path between CBC and GCM operations, and since AAD is already considered in the KDF there is no need to forward it to the GCM routine.

Context headers

11/1/2017 • 8 min to read • [Edit Online](#)

Background and theory

In the data protection system, a "key" means an object that can provide authenticated encryption services. Each key is identified by a unique id (a GUID), and it carries with it algorithmic information and entropic material. It is intended that each key carry unique entropy, but the system cannot enforce that, and we also need to account for developers who might change the key ring manually by modifying the algorithmic information of an existing key in the key ring. To achieve our security requirements given these cases the data protection system has a concept of [cryptographic agility](#), which allows securely using a single entropic value across multiple cryptographic algorithms.

Most systems which support cryptographic agility do so by including some identifying information about the algorithm inside the payload. The algorithm's OID is generally a good candidate for this. However, one problem that we ran into is that there are multiple ways to specify the same algorithm: "AES" (CNG) and the managed Aes, AesManaged, AesCryptoServiceProvider, AesCng, and RijndaelManaged (given specific parameters) classes are all actually the same thing, and we'd need to maintain a mapping of all of these to the correct OID. If a developer wanted to provide a custom algorithm (or even another implementation of AES!), they'd have to tell us its OID. This extra registration step makes system configuration particularly painful.

Stepping back, we decided that we were approaching the problem from the wrong direction. An OID tells you what the algorithm is, but we don't actually care about this. If we need to use a single entropic value securely in two different algorithms, it's not necessary for us to know what the algorithms actually are. What we actually care about is how they behave. Any decent symmetric block cipher algorithm is also a strong pseudorandom permutation (PRP): fix the inputs (key, chaining mode, IV, plaintext) and the ciphertext output will with overwhelming probability be distinct from any other symmetric block cipher algorithm given the same inputs. Similarly, any decent keyed hash function is also a strong pseudorandom function (PRF), and given a fixed input set its output will overwhelmingly be distinct from any other keyed hash function.

We use this concept of strong PRPs and PRFs to build up a context header. This context header essentially acts as a stable thumbprint over the algorithms in use for any given operation, and it provides the cryptographic agility needed by the data protection system. This header is reproducible and is used later as part of the [subkey derivation process](#). There are two different ways to build the context header depending on the modes of operation of the underlying algorithms.

CBC-mode encryption + HMAC authentication

The context header consists of the following components:

- [16 bits] The value 00 00, which is a marker meaning "CBC encryption + HMAC authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The key length (in bytes, big-endian) of the HMAC algorithm. (Currently the key size always matches the digest size.)
- [32 bits] The digest size (in bytes, big-endian) of the HMAC algorithm.
- EncCBC(K_E, IV, ""), which is the output of the symmetric block cipher algorithm given an empty string input and where IV is an all-zero vector. The construction of K_E is described below.

- $MAC(K_H, "")$, which is the output of the HMAC algorithm given an empty string input. The construction of K_H is described below.

Ideally, we could pass all-zero vectors for K_E and K_H . However, we want to avoid the situation where the underlying algorithm checks for the existence of weak keys before performing any operations (notably DES and 3DES), which precludes using a simple or repeatable pattern like an all-zero vector.

Instead, we use the NIST SP800-108 KDF in Counter Mode (see [NIST SP800-108](#), Sec. 5.1) with a zero-length key, label, and context and HMACSHA512 as the underlying PRF. We derive $|K_E| + |K_H|$ bytes of output, then decompose the result into K_E and K_H themselves. Mathematically, this is represented as follows.

$(K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")$

Example: AES-192-CBC + HMACSHA256

As an example, consider the case where the symmetric block cipher algorithm is AES-192-CBC and the validation algorithm is HMACSHA256. The system would generate the context header using the following steps.

First, let $(K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")$, where $|K_E| = 192$ bits and $|K_H| = 256$ bits per the specified algorithms. This leads to $K_E = 5BB6..21DD$ and $K_H = A04A..00A9$ in the example below:

```
5B B6 C9 83 13 78 22 1D 8E 10 73 CA CF 65 8E B0
61 62 42 71 CB 83 21 DD A0 4A 05 00 5B AB C0 A2
49 6F A5 61 E3 E2 49 87 AA 63 55 CD 74 0A DA C4
B7 92 3D BF 59 90 00 A9
```

Next, compute $Enc_CBC(K_E, IV, "")$ for AES-192-CBC given $IV = 0^*$ and K_E as above.

result := F474B1872B3B53E4721DE19C0841DB6F

Next, compute $MAC(K_H, "")$ for HMACSHA256 given K_H as above.

result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C

This produces the full context header below:

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36
22 0C
```

This context header is the thumbprint of the authenticated encryption algorithm pair (AES-192-CBC encryption + HMACSHA256 validation). The components, as described [above](#) are:

- the marker (00 00)
- the block cipher key length (00 00 00 18)
- the block cipher block size (00 00 00 10)
- the HMAC key length (00 00 00 20)
- the HMAC digest size (00 00 00 20)
- the block cipher PRP output (F4 74 - DB 6F) and
- the HMAC PRF output (D4 79 - end).

NOTE

The CBC-mode encryption + HMAC authentication context header is built the same way regardless of whether the algorithms implementations are provided by Windows CNG or by managed SymmetricAlgorithm and KeyedHashAlgorithm types. This allows applications running on different operating systems to reliably produce the same context header even though the implementations of the algorithms differ between OSes. (In practice, the KeyedHashAlgorithm doesn't have to be a proper HMAC. It can be any keyed hash algorithm type.)

Example: 3DES-192-CBC + HMACSHA1

First, let $(K_E || K_H) = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = "", \text{label} = "", \text{context} = "")$, where $|K_E| = 192$ bits and $|K_H| = 160$ bits per the specified algorithms. This leads to $K_E = \text{A219..E2BB}$ and $K_H = \text{DC4A..B464}$ in the example below:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

Next, compute $\text{Enc_CBC}(K_E, \text{IV}, "")$ for 3DES-192-CBC given $\text{IV} = 0^*$ and K_E as above.

result := ABB100F81E53E10E

Next, compute $\text{MAC}(K_H, "")$ for HMACSHA1 given K_H as above.

result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555

This produces the full context header which is a thumbprint of the authenticated encryption algorithm pair (3DES-192-CBC encryption + HMACSHA1 validation), shown below:

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

The components break down as follows:

- the marker (00 00)
- the block cipher key length (00 00 00 18)
- the block cipher block size (00 00 00 08)
- the HMAC key length (00 00 00 14)
- the HMAC digest size (00 00 00 14)
- the block cipher PRP output (AB B1 - E1 0E) and
- the HMAC PRF output (76 EB - end).

Galois/Counter Mode encryption + authentication

The context header consists of the following components:

- [16 bits] The value 00 01, which is a marker meaning "GCM encryption + authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The nonce size (in bytes, big-endian) used during authenticated encryption operations. (For our system, this is fixed at nonce size = 96 bits.)

- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm. (For GCM, this is fixed at block size = 128 bits.)
- [32 bits] The authentication tag size (in bytes, big-endian) produced by the authenticated encryption function. (For our system, this is fixed at tag size = 128 bits.)
- [128 bits] The tag of Enc_GCM (K_E, nonce, ""), which is the output of the symmetric block cipher algorithm given an empty string input and where nonce is a 96-bit all-zero vector.

K_E is derived using the same mechanism as in the CBC encryption + HMAC authentication scenario. However, since there is no K_H in play here, we essentially have $|K_H| = 0$, and the algorithm collapses to the below form.

$K_E = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = "", \text{label} = "", \text{context} = "")$

Example: AES-256-GCM

First, let $K_E = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = "", \text{label} = "", \text{context} = "")$, where $|K_E| = 256$ bits.

$K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8$

Next, compute the authentication tag of Enc_GCM (K_E, nonce, "") for AES-256-GCM given nonce = 096 and K_E as above.

result := E7DCCE66DF855A323A6BB7BD7A59BE45

This produces the full context header below:

```
00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59
BE 45
```

The components break down as follows:

- the marker (00 01)
- the block cipher key length (00 00 00 20)
- the nonce size (00 00 00 0C)
- the block cipher block size (00 00 00 10)
- the authentication tag size (00 00 00 10) and
- the authentication tag from running the block cipher (E7 DC - end).

Key Management

11/1/2017 • 6 min to read • [Edit Online](#)

The data protection system automatically manages the lifetime of master keys used to protect and unprotect payloads. Each key can exist in one of four stages:

- Created - the key exists in the key ring but has not yet been activated. The key shouldn't be used for new Protect operations until sufficient time has elapsed that the key has had a chance to propagate to all machines that are consuming this key ring.
- Active - the key exists in the key ring and should be used for all new Protect operations.
- Expired - the key has run its natural lifetime and should no longer be used for new Protect operations.
- Revoked - the key is compromised and must not be used for new Protect operations.

Created, active, and expired keys may all be used to unprotect incoming payloads. Revoked keys by default may not be used to unprotect payloads, but the application developer can [override this behavior](#) if necessary.

WARNING

The developer might be tempted to delete a key from the key ring (e.g., by deleting the corresponding file from the file system). At that point, all data protected by the key is permanently undecipherable, and there is no emergency override like there is with revoked keys. Deleting a key is truly destructive behavior, and consequently the data protection system exposes no first-class API for performing this operation.

Default key selection

When the data protection system reads the key ring from the backing repository, it will attempt to locate a "default" key from the key ring. The default key is used for new Protect operations.

The general heuristic is that the data protection system chooses the key with the most recent activation date as the default key. (There's a small fudge factor to allow for server-to-server clock skew.) If the key is expired or revoked, and if the application has not disabled automatic key generation, then a new key will be generated with immediate activation per the [key expiration and rolling](#) policy below.

The reason the data protection system generates a new key immediately rather than falling back to a different key is that new key generation should be treated as an implicit expiration of all keys that were activated prior to the new key. The general idea is that new keys may have been configured with different algorithms or encryption-at-rest mechanisms than old keys, and the system should prefer the current configuration over falling back.

There is an exception. If the application developer has [disabled automatic key generation](#), then the data protection system must choose something as the default key. In this fallback scenario, the system will choose the non-revoked key with the most recent activation date, with preference given to keys that have had time to propagate to other machines in the cluster. The fallback system may end up choosing an expired default key as a result. The fallback system will never choose a revoked key as the default key, and if the key ring is empty or every key has been revoked then the system will produce an error upon initialization.

Key expiration and rolling

When a key is created, it is automatically given an activation date of { now + 2 days } and an expiration date of { now + 90 days }. The 2-day delay before activation gives the key time to propagate through the system. That is, it

allows other applications pointing at the backing store to observe the key at their next auto-refresh period, thus maximizing the chances that when the key ring does become active it has propagated to all applications that might need to use it.

If the default key will expire within 2 days and if the key ring does not already have a key that will be active upon expiration of the default key, then the data protection system will automatically persist a new key to the key ring. This new key has an activation date of { default key's expiration date } and an expiration date of { now + 90 days }. This allows the system to automatically roll keys on a regular basis with no interruption of service.

There might be circumstances where a key will be created with immediate activation. One example would be when the application hasn't run for a time and all keys in the key ring are expired. When this happens, the key is given an activation date of { now } without the normal 2-day activation delay.

The default key lifetime is 90 days, though this is configurable as in the following example.

```
services.AddDataProtection()  
    // use 14-day lifetime instead of 90-day lifetime  
    .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

An administrator can also change the default system-wide, though an explicit call to `SetDefaultKeyLifetime` will override any system-wide policy. The default key lifetime cannot be shorter than 7 days.

Automatic key ring refresh

When the data protection system initializes, it reads the key ring from the underlying repository and caches it in memory. This cache allows Protect and Unprotect operations to proceed without hitting the backing store. The system will automatically check the backing store for changes approximately every 24 hours or when the current default key expires, whichever comes first.

WARNING

Developers should very rarely (if ever) need to use the key management APIs directly. The data protection system will perform automatic key management as described above.

The data protection system exposes an interface `IKeyManager` that can be used to inspect and make changes to the key ring. The DI system that provided the instance of `IDataProtectionProvider` can also provide an instance of `IKeyManager` for your consumption. Alternatively, you can pull the `IKeyManager` straight from the `IServiceProvider` as in the example below.

Any operation which modifies the key ring (creating a new key explicitly or performing a revocation) will invalidate the in-memory cache. The next call to `Protect` or `Unprotect` will cause the data protection system to reread the key ring and recreate the cache.

The sample below demonstrates using the `IKeyManager` interface to inspect and manipulate the key ring, including revoking existing keys and generating a new key manually.

```
using System;  
using System.IO;  
using System.Threading;  
using Microsoft.AspNetCore.DataProtection;  
using Microsoft.AspNetCore.DataProtection.KeyManagement;  
using Microsoft.Extensions.DependencyInjection;  
  
public class Program  
{  
    public static void Main(string[] args)  
    {  
    }  
}
```

```

var serviceCollection = new ServiceCollection();
serviceCollection.AddDataProtection()
    // point at a specific folder and use DPAPI to encrypt keys
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
    .ProtectKeysWithDpapi();
var services = serviceCollection.BuildServiceProvider();

// perform a protect operation to force the system to put at least
// one key in the key ring
services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
Console.WriteLine("Performed a protect operation.");
Thread.Sleep(2000);

// get a reference to the key manager
var keyManager = services.GetService<IKeyManager>();

// list all keys in the key ring
var allKeys = keyManager.GetAllKeys();
Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
foreach (var key in allKeys)
{
    Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
}

// revoke all keys in the key ring
keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation reason here.");
Console.WriteLine("Revoked all existing keys.");

// add a new key to the key ring with immediate activation and a 1-month expiration
keyManager.CreateNewKey(
    activationDate: DateTimeOffset.Now,
    expirationDate: DateTimeOffset.Now.AddMonths(1));
Console.WriteLine("Added a new key.");

// list all keys in the key ring
allKeys = keyManager.GetAllKeys();
Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
foreach (var key in allKeys)
{
    Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
}
}

/*
* SAMPLE OUTPUT
*
* Performed a protect operation.
* The key ring contains 1 key(s).
* Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = False
* Revoked all existing keys.
* Added a new key.
* The key ring contains 2 key(s).
* Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = True
* Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18 22:20:51Z, IsRevoked = False
*/

```

Key storage

The data protection system has a heuristic whereby it tries to deduce an appropriate key storage location and encryption at rest mechanism automatically. This is also configurable by the app developer. The following documents discuss the in-box implementations of these mechanisms:

- In-box key storage providers
- In-box key encryption at rest providers

Key storage providers

10/20/2017 • 2 min to read • [Edit Online](#)

By default the data protection system [employs a heuristic](#) to determine where cryptographic key material should be persisted. The developer can override the heuristic and manually specify the location.

NOTE

If you specify an explicit key persistence location, the data protection system will deregister the default key encryption at rest mechanism that the heuristic provided, so keys will no longer be encrypted at rest. It is recommended that you additionally [specify an explicit key encryption mechanism](#) for production applications.

The data protection system ships with several in-box key storage providers.

File system

We anticipate that many apps will use a file system-based key repository. To configure this, call the [PersistKeysToFileSystem](#) configuration routine as shown below. Provide a `DirectoryInfo` pointing to the repository where keys should be stored.

```
sc.AddDataProtection()  
    // persist keys to a specific directory  
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
```

The `DirectoryInfo` can point to a directory on the local machine, or it can point to a folder on a network share. If pointing to a directory on the local machine (and the scenario is that only applications on the local machine will need to use this repository), consider using [Windows DPAPI](#) to encrypt the keys at rest. Otherwise consider using an [X.509 certificate](#) to encrypt keys at rest.

Azure and Redis

The `Microsoft.AspNetCore.DataProtection.AzureStorage` and `Microsoft.AspNetCore.DataProtection.Redis` packages allow storing your data protection keys in Azure Storage or a Redis cache. Keys can be shared across several instances of a web app. Your ASP.NET Core app can share authentication cookies or CSRF protection across multiple servers. To configure on Azure, call one of the [PersistKeysToAzureBlobStorage](#) overloads as shown below.

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDataProtection()  
        .PersistKeysToAzureBlobStorage(new Uri("<blob URI including SAS token>"));  
  
    services.AddMvc();  
}
```

See also the [Azure test code](#).

To configure on Redis, call one of the [PersistKeysToRedis](#) overloads as shown below.

```
public void ConfigureServices(IServiceCollection services)
{
    // Connect to Redis database.
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToRedis(redis, "DataProtection-Keys");

    services.AddMvc();
}
```

See the following for more information:

- [StackExchange.Redis ConnectionMultiplexer](#)
- [Azure Redis Cache](#)
- [Redis test code.](#)

Registry

Sometimes the app might not have write access to the file system. Consider a scenario where an app is running as a virtual service account (such as w3wp.exe's app pool identity). In these cases, the administrator may have provisioned a registry key that is appropriate ACLed for the service account identity. Call the [PersistKeysToRegistry](#) configuration routine as shown below. Provide a `RegistryKey` pointing to the location where cryptographic keys/values should be stored.

```
sc.AddDataProtection()
    // persist keys to a specific location in the system registry
    .PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys"));
```

If you use the system registry as a persistence mechanism, consider using [Windows DPAPI](#) to encrypt the keys at rest.

Custom key repository

If the in-box mechanisms are not appropriate, the developer can specify their own key persistence mechanism by providing a custom `IXmlRepository`.

Key Encryption At Rest

11/1/2017 • 3 min to read • [Edit Online](#)

By default, the data protection system [employs a heuristic](#) to determine how cryptographic key material should be encrypted at rest. The developer can override the heuristic and manually specify how keys should be encrypted at rest.

NOTE

If you specify an explicit key encryption at rest mechanism, the data protection system will deregister the default key storage mechanism that the heuristic provided. You must [specify an explicit key storage mechanism](#), otherwise the data protection system will fail to start.

The data protection system ships with three in-box key encryption mechanisms.

Windows DPAPI

This mechanism is available only on Windows.

When Windows DPAPI is used, key material will be encrypted via [CryptProtectData](#) before being persisted to storage. DPAPI is an appropriate encryption mechanism for data that will never be read outside of the current machine (though it is possible to back these keys up to Active Directory; see [DPAPI and Roaming Profiles](#)). For example to configure DPAPI key-at-rest encryption.

```
sc.AddDataProtection()  
    // only the local user account can decrypt the keys  
    .ProtectKeysWithDpapi();
```

If `ProtectKeysWithDpapi` is called with no parameters, only the current Windows user account can decipher the persisted key material. You can optionally specify that any user account on the machine (not just the current user account) should be able to decipher the key material, as shown in the below example.

```
sc.AddDataProtection()  
    // all user accounts on the machine can decrypt the keys  
    .ProtectKeysWithDpapi(protectToLocalMachine: true);
```

X.509 certificate

This mechanism is not available on `.NET Core 1.0` or `1.1`.

If your application is spread across multiple machines, it may be convenient to distribute a shared X.509 certificate across the machines and to configure applications to use this certificate for encryption of keys at rest. See below for an example.

```
sc.AddDataProtection()  
    // searches the cert store for the cert with this thumbprint  
    .ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
```

Due to .NET Framework limitations only certificates with CAPI private keys are supported. See [Certificate-based](#)

encryption with [Windows DPAPI-NG](#) below for possible workarounds to these limitations.

Windows DPAPI-NG

This mechanism is available only on Windows 8 / Windows Server 2012 and later.

Beginning with Windows 8, the operating system supports DPAPI-NG (also called CNG DPAPI). Microsoft lays out its usage scenario as follows.

Cloud computing, however, often requires that content encrypted on one computer be decrypted on another. Therefore, beginning with Windows 8, Microsoft extended the idea of using a relatively straightforward API to encompass cloud scenarios. This new API, called DPAPI-NG, enables you to securely share secrets (keys, passwords, key material) and messages by protecting them to a set of principals that can be used to unprotect them on different computers after proper authentication and authorization.

From [About CNG DPAPI](#)

The principal is encoded as a protection descriptor rule. Consider the below example, which encrypts key material such that only the domain-joined user with the specified SID can decrypt the key material.

```
sc.AddDataProtection()  
    // uses the descriptor rule "SID=S-1-5-21-..."  
    .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",  
        flags: DpapiNGProtectionDescriptorFlags.None);
```

There is also a parameterless overload of `ProtectKeysWithDpapiNG`. This is a convenience method for specifying the rule "SID=mine", where mine is the SID of the current Windows user account.

```
sc.AddDataProtection()  
    // uses the descriptor rule "SID={current account SID}"  
    .ProtectKeysWithDpapiNG();
```

In this scenario, the AD domain controller is responsible for distributing the encryption keys used by the DPAPI-NG operations. The target user will be able to decipher the encrypted payload from any domain-joined machine (provided that the process is running under their identity).

Certificate-based encryption with Windows DPAPI-NG

If you're running on Windows 8.1 / Windows Server 2012 R2 or later, you can use Windows DPAPI-NG to perform certificate-based encryption, even if the application is running on [.NET Core](#). To take advantage of this, use the rule descriptor string "CERTIFICATE=HashId:thumbprint", where thumbprint is the hex-encoded SHA1 thumbprint of the certificate to use. See below for an example.

```
sc.AddDataProtection()  
    // searches the cert store for the cert with this thumbprint  
    .ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0",  
        flags: DpapiNGProtectionDescriptorFlags.None);
```

Any application which is pointed at this repository must be running on Windows 8.1 / Windows Server 2012 R2 or later to be able to decipher this key.

Custom key encryption

If the in-box mechanisms are not appropriate, the developer can specify their own key encryption mechanism by providing a custom `IXmlEncryptor`.

Key Immutability and changing settings

11/1/2017 • 1 min to read • [Edit Online](#)

Once an object is persisted to the backing store, its representation is forever fixed. New data can be added to the backing store, but existing data can never be mutated. The primary purpose of this behavior is to prevent data corruption.

One consequence of this behavior is that once a key is written to the backing store, it is immutable. Its creation, activation, and expiration dates can never be changed, though it can be revoked by using `IKeyManager`. Additionally, its underlying algorithmic information, master keying material, and encryption at rest properties are also immutable.

If the developer changes any setting that affects key persistence, those changes will not go into effect until the next time a key is generated, either via an explicit call to `IKeyManager.CreateNewKey` or via the data protection system's own [automatic key generation](#) behavior. The settings that affect key persistence are as follows:

- [The default key lifetime](#)
- [The key encryption at rest mechanism](#)
- [The algorithmic information contained within the key](#)

If you need these settings to kick in earlier than the next automatic key rolling time, consider making an explicit call to `IKeyManager.CreateNewKey` to force the creation of a new key. Remember to provide an explicit activation date (`{ now + 2 days }` is a good rule of thumb to allow time for the change to propagate) and expiration date in the call.

TIP

All applications touching the repository should specify the same settings with the `IDataProtectionBuilder` extension methods. Otherwise, the properties of the persisted key will be dependent on the particular application that invoked the key generation routines.

Key Storage Format

11/1/2017 • 2 min to read • [Edit Online](#)

Objects are stored at rest in XML representation. The default directory for key storage is %LOCALAPPDATA%\ASP.NET\DataProtection-Keys.

The <key> element

Keys exist as top-level objects in the key repository. By convention keys have the filename **key-{guid}.xml**, where {guid} is the id of the key. Each such file contains a single key. The format of the file is as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<key id="80732141-ec8f-4b80-af9c-c4d2d1ff8901" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor deserializerType="{deserializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

The <key> element contains the following attributes and child elements:

- The key id. This value is treated as authoritative; the filename is simply a nicety for human readability.
- The version of the <key> element, currently fixed at 1.
- The key's creation, activation, and expiration dates.
- A <descriptor> element, which contains information on the authenticated encryption implementation contained within this key.

In the above example, the key's id is {80732141-ec8f-4b80-af9c-c4d2d1ff8901}, it was created and activated on March 19, 2015, and it has a lifetime of 90 days. (Occasionally the activation date might be slightly before the creation date as in this example. This is due to a nit in how the APIs work and is harmless in practice.)

The <descriptor> element

The outer <descriptor> element contains an attribute `deserializerType`, which is the assembly-qualified name of a type which implements `IAuthenticatedEncryptorDescriptorDeserializer`. This type is responsible for reading the inner <descriptor> element and for parsing the information contained within.

The particular format of the <descriptor> element depends on the authenticated encryptor implementation encapsulated by the key, and each deserializer type expects a slightly different format for this. In general, though, this element will contain algorithmic information (names, types, OIDs, or similar) and secret key material. In the above example, the descriptor specifies that this key wraps AES-256-CBC encryption + HMACSHA256 validation.

The <encryptedSecret> element

An element which contains the encrypted form of the secret key material may be present if [encryption of secrets at rest is enabled](#). The attribute decryptorType will be the assembly-qualified name of a type which implements `IXmlDecryptor`. This type is responsible for reading the inner element and decrypting it to recover the original plaintext.

As with <descriptor>, the particular format of the element depends on the at-rest encryption mechanism in use. In the above example, the master key is encrypted using Windows DPAPI per the comment.

The <revocation> element

Revocations exist as top-level objects in the key repository. By convention revocations have the filename **revocation-{timestamp}.xml** (for revoking all keys before a specific date) or **revocation-{guid}.xml** (for revoking a specific key). Each file contains a single <revocation> element.

For revocations of individual keys, the file contents will be as below.

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="eb4fc299-8808-409d-8a34-23fc83d026c9" />
  <reason>human-readable reason</reason>
</revocation>
```

In this case, only the specified key is revoked. If the key id is "*", however, as in the below example, all keys whose creation date is prior to the specified revocation date are revoked.

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

The <reason> element is never read by the system. It is simply a convenient place to store a human-readable reason for revocation.

Ephemeral data protection providers

11/1/2017 • 1 min to read • [Edit Online](#)

There are scenarios where an application needs a throwaway `IDataProtectionProvider`. For example, the developer might just be experimenting in a one-off console application, or the application itself is transient (it's scripted or a unit test project). To support these scenarios the `Microsoft.AspNetCore.DataProtection` package includes a type `EphemeralDataProtectionProvider`. This type provides a basic implementation of `IDataProtectionProvider` whose key repository is held solely in-memory and isn't written out to any backing store.

Each instance of `EphemeralDataProtectionProvider` uses its own unique master key. Therefore, if an `IDataProtector` rooted at an `EphemeralDataProtectionProvider` generates a protected payload, that payload can only be unprotected by an equivalent `IDataProtector` (given the same `purpose` chain) rooted at the same `EphemeralDataProtectionProvider` instance.

The following sample demonstrates instantiating an `EphemeralDataProtectionProvider` and using it to protect and unprotect data.

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";

        // create an ephemeral provider and demonstrate that it can round-trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); // THROWS
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protect returned: CfDJ8AAAAAAAAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
 * Unprotect returned: Hello!
 * << throws CryptographicException >>
 */
```

Compatibility in ASP.NET Core

11/1/2017 • 1 min to read • [Edit Online](#)

- [Sharing cookies between applications](#)
- [Replacing <machineKey> in ASP.NET](#)

Sharing cookies between applications

11/1/2017 • 3 min to read • [Edit Online](#)

Web sites commonly consist of many individual web applications, all working together harmoniously. If an application developer wants to provide a good single-sign-on experience, they'll often need all of the different web applications within the site to share authentication tickets between each other.

To support this scenario, the data protection stack allows sharing Katana cookie authentication and ASP.NET Core cookie authentication tickets.

Sharing authentication cookies between applications

To share authentication cookies between two different ASP.NET Core applications, configure each application that should share cookies as follows.

In your configure method, use the `CookieAuthenticationOptions` to set up the data protection service for cookies and the `AuthenticationScheme` to match ASP.NET 4.x.

If you're using identity:

```
app.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    options.Cookies.ApplicationCookie.AuthenticationScheme = "ApplicationCookie";
    var protectionProvider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\shared-auth-ticket-keys\"));
    options.Cookies.ApplicationCookie.DataProtectionProvider = protectionProvider;
    options.Cookies.ApplicationCookie.TicketDataFormat = new
TicketDataFormat(protectionProvider.CreateProtector("Microsoft.AspNetCore.Authentication.Cookies.CookieAuthent
icationMiddleware", "Cookies", "v2"));
});
```

If you're using cookies directly:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    DataProtectionProvider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\shared-auth-ticket-keys\"))
});
```

The `DataProtectionProvider` requires the `Microsoft.AspNetCore.DataProtection.Extensions` NuGet package.

When used in this manner, the `DirectoryInfo` should point to a key storage location specifically set aside for authentication cookies. The cookie authentication middleware will use the explicitly provided implementation of the `DataProtectionProvider`, which is now isolated from the data protection system used by other parts of the application. The application name is ignored (intentionally so, since you're trying to get multiple applications to share payloads).

WARNING

You should consider configuring the `DataProtectionProvider` such that keys are encrypted at rest, as in the below example.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    DataProtectionProvider = DataProtectionProvider.Create(
        new DirectoryInfo(@"c:\shared-auth-ticket-keys\"),
        configure =>
        {
            configure.ProtectKeysWithCertificate("thumbprint");
        }
    });
```

Sharing authentication cookies between ASP.NET 4.x and ASP.NET Core applications

ASP.NET 4.x applications which use Katana cookie authentication middleware can be configured to generate authentication cookies which are compatible with the ASP.NET Core cookie authentication middleware. This allows upgrading a large site's individual applications piecemeal while still providing a smooth single sign on experience across the site.

TIP

You can tell if your existing application uses Katana cookie authentication middleware by the existence of a call to `UseCookieAuthentication` in your project's `Startup.Auth.cs`. ASP.NET 4.x web application projects created with Visual Studio 2013 and later use the Katana cookie authentication middleware by default.

NOTE

Your ASP.NET 4.x application must target .NET Framework 4.5.1 or higher, otherwise the necessary NuGet packages will fail to install.

To share authentication cookies between your ASP.NET 4.x applications and your ASP.NET Core applications, configure the ASP.NET Core application as stated above, then configure your ASP.NET 4.x applications by following the steps below.

1. Install the package `Microsoft.Owin.Security.Interop` into each of your ASP.NET 4.x applications.
2. In `Startup.Auth.cs`, locate the call to `UseCookieAuthentication`, which will generally look like the following.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    // ...
});
```

3. Modify the call to `UseCookieAuthentication` as follows, changing the `CookieName` to match the name used by the ASP.NET Core cookie authentication middleware, providing an instance of a `DataProtectionProvider` that has been initialized to a key storage location, and set `CookieManager` to `interop ChunkingCookieManager` so the chunking format is compatible.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
    CookieName = ".AspNetCore.Cookies",
    // CookieName = ".AspNetCore.ApplicationCookie", (if you're using identity)
    // CookiePath = "...", (if necessary)
    // ...
    TicketDataFormat = new AspNetTicketDataFormat(
        new DataProtectorShim(
            DataProtectionProvider.Create(new DirectoryInfo(@"c:\shared-auth-ticket-keys\"))

        .CreateProtector("Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddleware",
            "Cookies", "v2")),
        CookieManager = new ChunkingCookieManager()
    });
```

The DirectoryInfo has to point to the same storage location that you pointed your ASP.NET Core application to and should be configured using the same settings.

The ASP.NET 4.x and ASP.NET Core applications are now configured to share authentication cookies.

NOTE

You'll need to make sure that the identity system for each application is pointed at the same user database. Otherwise the identity system will produce failures at runtime when it tries to match the information in the authentication cookie against the information in its database.

Replacing `<machineKey>` in ASP.NET

10/20/2017 • 2 min to read • [Edit Online](#)

The implementation of the `<machineKey>` element in ASP.NET is [replaceable](#). This allows most calls to ASP.NET cryptographic routines to be routed through a replacement data protection mechanism, including the new data protection system.

Package installation

NOTE

The new data protection system can only be installed into an existing ASP.NET application targeting .NET 4.5.1 or higher. Installation will fail if the application targets .NET 4.5 or lower.

To install the new data protection system into an existing ASP.NET 4.5.1+ project, install the package `Microsoft.AspNetCore.DataProtection.SystemWeb`. This will instantiate the data protection system using the [default configuration](#) settings.

When you install the package, it inserts a line into `Web.config` that tells ASP.NET to use it for [most cryptographic operations](#), including forms authentication, view state, and calls to `MachineKey.Protect`. The line that's inserted reads as follows.

```
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

TIP

You can tell if the new data protection system is active by inspecting fields like `__VIEWSTATE`, which should begin with "CfDJ8" as in the example below. "CfDJ8" is the base64 representation of the magic "09 F0 C9 F0" header that identifies a payload protected by the data protection system.

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="CfDJ8AWPr2EQPTBGs3L2GCZ0pk..." />
```

Package configuration

The data protection system is instantiated with a default zero-setup configuration. However, since by default keys are persisted to the local file system, this won't work for applications which are deployed in a farm. To resolve this, you can provide configuration by creating a type which subclasses `DataProtectionStartup` and overrides its `ConfigureServices` method.

Below is an example of a custom data protection startup type which configured both where keys are persisted and how they're encrypted at rest. It also overrides the default app isolation policy by providing its own application name.

```

using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.SystemWeb;
using Microsoft.Extensions.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
        public override void ConfigureServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("my-app")
                .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\myapp-keys\"))
                .ProtectKeysWithCertificate("thumbprint");
        }
    }
}

```

TIP

You can also use `<machineKey applicationName="my-app" ... />` in place of an explicit call to `SetApplicationName`. This is a convenience mechanism to avoid forcing the developer to create a `DataProtectionStartup`-derived type if all they wanted to configure was setting the application name.

To enable this custom configuration, go back to `Web.config` and look for the `<appSettings>` element that the package install added to the config file. It will look like the following markup:

```

<appSettings>
  <!--
  If you want to customize the behavior of the ASP.NET Core Data Protection stack, set the
  "aspnet:dataProtectionStartupType" switch below to be the fully-qualified name of a
  type which subclasses Microsoft.AspNetCore.DataProtection.SystemWeb.DataProtectionStartup.
  -->
  <add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>

```

Fill in the blank value with the assembly-qualified name of the `DataProtectionStartup`-derived type you just created. If the name of the application is `DataProtectionDemo`, this would look like the below.

```

<add key="aspnet:dataProtectionStartupType"
  value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo" />

```

The newly-configured data protection system is now ready for use inside the application.

Enforcing SSL in an ASP.NET Core app

9/25/2017 • 1 min to read • [Edit Online](#)

By [Rick Anderson](#)

This document shows how to:

- Require SSL for all requests (HTTPS requests only).
- Redirect all HTTP requests to HTTPS.

Require SSL

The [RequireHttpsAttribute](#) is used to require SSL. You can decorate controllers or methods with this attribute or you can apply it globally as shown below:

Add the following code to `ConfigureServices` in `Startup`:

```
// Requires using Microsoft.AspNetCore.Mvc;
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new RequireHttpsAttribute());
    });
}
```

The highlighted code above requires all requests use `HTTPS`, therefore HTTP requests are ignored. The following highlighted code redirects all HTTP requests to HTTPS:

```
// Requires using Microsoft.AspNetCore.Rewrite;
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    var options = new RewriteOptions()
        .AddRedirectToHttps();

    app.UseRewriter(options);
}
```

See [URL Rewriting Middleware](#) for more information.

Requiring HTTPS globally (`options.Filters.Add(new RequireHttpsAttribute());`) is a security best practice. Applying the `[RequireHttps]` attribute to all controller is not considered as secure as requiring HTTPS globally. You can't guarantee new controllers added to your app will remember to apply the `[RequireHttps]` attribute.

Safe storage of app secrets during development in ASP.NET Core

11/29/2017 • 5 min to read • [Edit Online](#)

By [Rick Anderson](#), [Daniel Roth](#), and [Scott Addie](#)

This document shows how you can use the Secret Manager tool in development to keep secrets out of your code. The most important point is you should never store passwords or other sensitive data in source code, and you shouldn't use production secrets in development and test mode. You can instead use the [configuration](#) system to read these values from environment variables or from values stored using the Secret Manager tool. The Secret Manager tool helps prevent sensitive data from being checked into source control. The [configuration](#) system can read secrets stored with the Secret Manager tool described in this article.

The Secret Manager tool is used only in development. You can safeguard Azure test and production secrets with the [Microsoft Azure Key Vault](#) configuration provider. See [Azure Key Vault configuration provider](#) for more information.

Environment variables

To avoid storing app secrets in code or in local configuration files, you store secrets in environment variables. You can setup the [configuration](#) framework to read values from environment variables by calling `AddEnvironmentVariables`. You can then use environment variables to override configuration values for all previously specified configuration sources.

For example, if you create a new ASP.NET Core web app with individual user accounts, it will add a default connection string to the `appsettings.json` file in the project with the key `DefaultConnection`. The default connection string is setup to use LocalDB, which runs in user mode and doesn't require a password. When you deploy your application to a test or production server, you can override the `DefaultConnection` key value with an environment variable setting that contains the connection string (potentially with sensitive credentials) for a test or production database server.

WARNING

Environment variables are generally stored in plain text and are not encrypted. If the machine or process is compromised, then environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may still be required.

Secret Manager

The Secret Manager tool stores sensitive data for development work outside of your project tree. The Secret Manager tool is a project tool that can be used to store secrets for a [.NET Core](#) project during development. With the Secret Manager tool, you can associate app secrets with a specific project and share them across multiple projects.

WARNING

The Secret Manager tool does not encrypt the stored secrets and should not be treated as a trusted store. It is for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

Installing the Secret Manager tool

- [Visual Studio](#)
- [Visual Studio Code](#)

Right-click the project in Solution Explorer, and select **Edit <project_name>.csproj** from the context menu. Add the highlighted line to the *.csproj* file, and save to restore the associated NuGet package:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>
```

Right-click the project in Solution Explorer again, and select **Manage User Secrets** from the context menu. This gesture adds a new `UserSecretsId` node within a `PropertyGroup` of the *.csproj* file, as highlighted in the following sample:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <UserSecretsId>User-Secret-ID</UserSecretsId>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>
```

Saving the modified *.csproj* file also opens a `secrets.json` file in the text editor. Replace the contents of the `secrets.json` file with the following code:

```
{
  "MySecret": "ValueOfMySecret"
}
```

Accessing user secrets via configuration

You access Secret Manager secrets through the configuration system. Add the

`Microsoft.Extensions.Configuration.UserSecrets` package and run `dotnet restore`.

Add the user secrets configuration source to the `Startup` method:

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace UserSecrets
{
    public class Startup
    {
        string _testSecret = null;
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder();

            if (env.IsDevelopment())
            {
                builder.AddUserSecrets<Startup>();
            }

            Configuration = builder.Build();
        }

        public IConfigurationRoot Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            _testSecret = Configuration["MySecret"];
        }

        public void Configure(IApplicationBuilder app)
        {
            var result = string.IsNullOrEmpty(_testSecret) ? "Null" : "Not Null";
            app.Run(async (context) =>
            {
                await context.Response.WriteAsync($"Secret is {result}");
            });
        }
    }
}

```

You can access user secrets via the configuration API:

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace UserSecrets
{
    public class Startup
    {
        string _testSecret = null;
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder();

            if (env.IsDevelopment())
            {
                builder.AddUserSecrets<Startup>();
            }

            Configuration = builder.Build();
        }

        public IConfigurationRoot Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            _testSecret = Configuration["MySecret"];
        }

        public void Configure(IApplicationBuilder app)
        {
            var result = string.IsNullOrEmpty(_testSecret) ? "Null" : "Not Null";
            app.Run(async (context) =>
            {
                await context.Response.WriteAsync($"Secret is {result}");
            });
        }
    }
}

```

How the Secret Manager tool works

The Secret Manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. In the current version, the values are stored in a [JSON](#) configuration file in the user profile directory:

- Windows: `%APPDATA%\microsoft\UserSecrets\\secrets.json`
- Linux: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`
- Mac: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`

The value of `userSecretsId` comes from the value specified in `.csproj` file.

You should not write code that depends on the location or format of the data saved with the Secret Manager tool, as these implementation details might change. For example, the secret values are currently *not* encrypted today, but could be someday.

Additional Resources

- [Configuration](#)

Azure Key Vault configuration provider

12/11/2017 • 8 min to read • [Edit Online](#)

By [Luke Latham](#) and [Andrew Stanton-Nurse](#)

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

View or download sample code for 2.x:

- [Basic sample \(how to download\)](#) - Reads secret values into an app.
- [Key name prefix sample \(how to download\)](#) - Reads secret values using a key name prefix that represents the version of an app, which allows you to load a different set of secret values for each app version.

This document explains how to use the [Microsoft Azure Key Vault](#) configuration provider to load application configuration values from Azure Key Vault secrets. Azure Key Vault is a cloud-based service that helps you safeguard cryptographic keys and secrets used by apps and services. Common scenarios include controlling access to sensitive configuration data and meeting the requirement for FIPS 140-2 Level 2 validated Hardware Security Modules (HSM's) when storing configuration data. This feature is available for applications that target ASP.NET Core 1.1 or higher.

Package

To use the provider, add a reference to the [Microsoft.Extensions.Configuration.AzureKeyVault](#) package.

Application configuration

You can explore the provider with the [sample apps](#). Once you establish a key vault and create secrets in the vault, the sample apps securely load the secret values into their configurations and display them in webpages.

The provider is added to the `ConfigurationBuilder` with the `AddAzureKeyVault` extension. In the sample apps, the extension uses three configuration values loaded from the `appsettings.json` file.

APP SETTING	DESCRIPTION	EXAMPLE
<code>Vault</code>	Azure Key Vault name	contosovault
<code>ClientId</code>	Azure Active Directory App Id	627e911e-43cc-61d4-992e-12db9c81b413
<code>ClientSecret</code>	Azure Active Directory App Key	g58K3dtg59o1Pa+e59v2Tx829w6VxTB2yv9sv/101di=

```

config.SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", optional: false)
    .AddEnvironmentVariables();

var builtConfig = config.Build();

config.AddAzureKeyVault(
    $"https://{builtConfig["Vault"]}.vault.azure.net/",
    builtConfig["ClientId"],
    builtConfig["ClientSecret"]);

```

Creating key vault secrets and loading configuration values (basic-sample)

1. Create a key vault and set up Azure Active Directory (Azure AD) for the application following the guidance in [Get started with Azure Key Vault](#).

- Add secrets to the key vault using the [AzureRM Key Vault PowerShell Module](#) available from the [PowerShell Gallery](#), the [Azure Key Vault REST API](#), or the [Azure Portal](#). Secrets are created as either *Manual* or *Certificate* secrets. *Certificate* secrets are certificates for use by apps and services but are not supported by the configuration provider. You should use the *Manual* option to create name-value pair secrets for use with the configuration provider.

- Simple secrets are created as name-value pairs. Azure Key Vault secret names are limited to alphanumeric characters and dashes.
- Hierarchical values (configuration sections) use `--` (two dashes) as a separator in the sample. Colons, which are normally used to delimit a section from a subkey in [ASP.NET Core configuration](#), aren't allowed in secret names. Therefore, two dashes are used and swapped for a colon when the secrets are loaded into the app's configuration.
- Create two *Manual* secrets with the following name-value pairs. The first secret is a simple name and value, and the second secret creates a secret value with a section and subkey in the secret name:

- `SecretName : secret_value_1`
- `Section--SecretName : secret_value_2`

- Register the sample app with Azure Active Directory.
- Authorize the app to access the key vault. When you use the `Set-AzureRmKeyVaultAccessPolicy` PowerShell cmdlet to authorize the app to access the key vault, provide `List` and `Get` access to secrets with `-PermissionsToSecrets list,get`.

2. Update the app's `appsettings.json` file with the values of `Vault`, `ClientId`, and `ClientSecret`.

3. Run the sample app, which obtains its configuration values from `IConfigurationRoot` with the same name as the secret name.

- Non-hierarchical values: The value for `SecretName` is obtained with `config["SecretName"]`.
- Hierarchical values (sections): Use `:` (colon) notation or the `GetSection` extension method. Use either of these approaches to obtain the configuration value:
 - `config["Section:SecretName"]`
 - `config.GetSection("Section")["SecretName"]`

When you run the app, a webpage shows the loaded secret values:

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
SecretName	SecretName	Configuration["SecretName"]	secret_value_1
Section:SecretName	Section--SecretName	Configuration["Section:SecretName"]	secret_value_2
		Configuration.GetSection("Section")["SecretName"]	secret_value_2

Creating prefixed key vault secrets and loading configuration values (key-name-prefix-sample)

`AddAzureKeyVault` also provides an overload that accepts an implementation of `IKeyVaultSecretManager`, which allows you to control how key vault secrets are converted into configuration keys. For example, you can implement the interface to load secret values based on a prefix value you provide at app startup. This allows you, for example, to load secrets based on the version of the app.

WARNING

Don't use prefixes on key vault secrets to place secrets for multiple apps into the same key vault or to place environmental secrets (for example, *development* versus *production* secrets) into the same vault. We recommend that different apps and development/production environments use separate key vaults to isolate app environments for the highest level of security.

Using the second sample app, you create a secret in the key vault for `5000-AppSecret` (periods aren't allowed in key vault secret names) representing an app secret for version 5.0.0.0 of your app. For another version, 5.1.0.0, you create a secret for `5100-AppSecret`. Each app version loads its own secret value into its configuration as `AppSecret`, stripping off the version as it loads the secret. The sample's implementation is shown below:

```
// The appVersion obtains the app version (5.0.0.0), which
// is set in the project file and obtained from the entry
// assembly. The versionPrefix holds the version without
// dot notation for the PrefixKeyVaultSecretManager.
var appVersion = Assembly.GetEntryAssembly().GetName().Version.ToString();
var versionPrefix = appVersion.Replace(".", string.Empty);

config.AddAzureKeyVault(
    $"https://{builtConfig["Vault"]}.vault.azure.net/",
    builtConfig["ClientId"],
    builtConfig["ClientSecret"],
    new PrefixKeyVaultSecretManager(versionPrefix));
```

```

public class PrefixKeyVaultSecretManager : IKeyVaultSecretManager
{
    private readonly string _prefix;

    public PrefixKeyVaultSecretManager(string prefix)
    {
        _prefix = $"{prefix}-";
    }

    public bool Load(SecretItem secret)
    {
        // Load a vault secret when its secret name starts with the
        // prefix. Other secrets won't be loaded.
        return secret.Identifier.Name.StartsWith(_prefix);
    }

    public string GetKey(SecretBundle secret)
    {
        // Remove the prefix from the secret name and replace two
        // dashes in any name with the KeyDelimiter, which is the
        // delimiter used in configuration (usually a colon). Azure
        // Key Vault doesn't allow a colon in secret names.
        return secret.SecretIdentifier.Name
            .Substring(_prefix.Length)
            .Replace("--", ConfigurationPath.KeyDelimiter);
    }
}

```

The `Load` method is called by a provider algorithm that iterates through the vault secrets to find the ones that have the version prefix. When a version prefix is found with `Load`, the algorithm uses the `GetKey` method to return the configuration name of the secret name. It strips off the version prefix from the secret's name and returns the rest of the secret name for loading into the app's configuration name-value pairs.

When you implement this approach:

1. The key vault secrets are loaded.
2. The string secret for `5000-AppSecret` is matched.
3. The version, `5000` (with the dash), is stripped off of the key name leaving `AppSecret` to load with the secret value into the app's configuration.

NOTE

You can also provide your own `KeyVaultClient` implementation to `AddAzureKeyVault`. Supplying a custom client allows you to share a single instance of the client between the configuration provider and other parts of your app.

1. Create a key vault and set up Azure Active Directory (Azure AD) for the application following the guidance in [Get started with Azure Key Vault](#).
 - Add secrets to the key vault using the [AzureRM Key Vault PowerShell Module](#) available from the [PowerShell Gallery](#), the [Azure Key Vault REST API](#), or the [Azure Portal](#). Secrets are created as either *Manual* or *Certificate* secrets. *Certificate* secrets are certificates for use by apps and services but are not supported by the configuration provider. You should use the *Manual* option to create name-value pair secrets for use with the configuration provider.
 - Hierarchical values (configuration sections) use `--` (two dashes) as a separator.
 - Create two *Manual* secrets with the following name-value pairs:
 - `5000-AppSecret : 5.0.0.0_secret_value`
 - `5100-AppSecret : 5.1.0.0_secret_value`
 - Register the sample app with Azure Active Directory.

- Authorize the app to access the key vault. When you use the `Set-AzureRmKeyVaultAccessPolicy` PowerShell cmdlet to authorize the app to access the key vault, provide `List` and `Get` access to secrets with `-PermissionsToSecrets list,get`.
2. Update the app's `appsettings.json` file with the values of `Vault`, `ClientId`, and `ClientSecret`.
 3. Run the sample app, which obtains its configuration values from `IConfigurationRoot` with the same name as the prefixed secret name. In this sample, the prefix is the app's version, which you provided to the `PrefixKeyVaultSecretManager` when you added the Azure Key Vault configuration provider. The value for `AppSecret` is obtained with `config["AppSecret"]`. The webpage generated by the app shows the loaded value:

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
AppSecret	5000-AppSecret	Configuration["AppSecret"]	5.0.0.0_secret_value

4. Change the version of the app assembly in the project file from `5.0.0.0` to `5.1.0.0` and run the app again. This time, the secret value returned is `5.1.0.0_secret_value`. The webpage generated by the app shows the loaded value:

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
AppSecret	5100-AppSecret	Configuration["AppSecret"]	5.1.0.0_secret_value

Controlling access to the ClientSecret

Use the [Secret Manager tool](#) to maintain the `ClientSecret` outside of your project source tree. With Secret Manager, you associate app secrets with a specific project and share them across multiple projects.

When developing a .NET Framework app in an environment that supports certificates, you can authenticate to Azure Key Vault with an X.509 certificate. The X.509 certificate's private key is managed by the OS. For more information, see [Authenticate with a Certificate instead of a Client Secret](#). Use the `AddAzureKeyVault` overload that accepts an `X509Certificate2`.

```
var store = new X509Store(StoreLocation.CurrentUser);
store.Open(OpenFlags.ReadOnly);
var cert = store.Certificates.Find(X509FindType.FindByThumbprint, config["CertificateThumbprint"], false);

builder.AddAzureKeyVault(
    config["Vault"],
    config["ClientId"],
    cert.OfType<X509Certificate2>().Single(),
    new EnvironmentSecretManager(env.ApplicationName));
store.Close();

Configuration = builder.Build();
```

Reloading secrets

Secrets are cached until `IConfigurationRoot.Reload()` is called. Expired, disabled, and updated secrets in the key

vault are not respected by the application until `Reload` is executed.

```
Configuration.Reload();
```

Disabled and expired secrets

Disabled and expired secrets throw a `KeyVaultClientException`. To prevent your app from throwing, replace your app or update the disabled/expired secret.

Troubleshooting

When the application fails to load configuration using the provider, an error message is written to the [ASP.NET Logging infrastructure](#). The following conditions will prevent configuration from loading:

- The app isn't configured correctly in Azure Active Directory.
- The key vault doesn't exist in Azure Key Vault.
- The app isn't authorized to access the key vault.
- The access policy doesn't include `Get` and `List` permissions.
- In the key vault, the configuration data (name-value pair) is incorrectly named, missing, disabled, or expired.
- The app has the wrong key vault name (`vault`), Azure AD App Id (`ClientId`), or Azure AD Key (`ClientSecret`).
- The Azure AD Key (`ClientSecret`) is expired.
- The configuration key (name) is incorrect in the app for the value you're trying to load.

Additional resources

- [Configuration](#)
- [Microsoft Azure: Key Vault](#)
- [Microsoft Azure: Key Vault Documentation](#)
- [How to generate and transfer HSM-protected keys for Azure Key Vault](#)
- [KeyVaultClient Class](#)

Preventing Cross-Site Request Forgery (XSRF/CSRF) Attacks in ASP.NET Core

9/25/2017 • 13 min to read • [Edit Online](#)

Steve Smith, Fiyaz Hasan, and Rick Anderson

What attack does anti-forgery prevent?

Cross-site request forgery (also known as XSRF or CSRF, pronounced *see-surf*) is an attack against web-hosted applications whereby a malicious web site can influence the interaction between a client browser and a web site that trusts that browser. These attacks are made possible because web browsers send some types of authentication tokens automatically with every request to a web site. This form of exploit is also known as a *one-click attack* or as *session riding*, because the attack takes advantage of the user's previously authenticated session.

An example of a CSRF attack:

1. A user logs into `www.example.com`, using forms authentication.
2. The server authenticates the user and issues a response that includes an authentication cookie.
3. The user visits a malicious site.

The malicious site contains an HTML form similar to the following:

```
<h1>You Are a Winner!</h1>
<form action="http://example.com/api/account" method="post">
  <input type="hidden" name="Transaction" value="withdraw" />
  <input type="hidden" name="Amount" value="1000000" />
  <input type="submit" value="Click Me"/>
</form>
```

Notice that the form action posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.

1. The user clicks the submit button. The browser automatically includes the authentication cookie for the requested domain (the vulnerable site in this case) with the request.
2. The request runs on the server with the user's authentication context and can do anything that an authenticated user is allowed to do.

This example requires the user to click the form button. The malicious page could:

- Run a script that automatically submits the form.
- Sends a form submission as an AJAX request.
- Use a hidden form with CSS.

Using SSL does not prevent a CSRF attack, the malicious site can send an `https://` request.

Some attacks target site endpoints that respond to `GET` requests, in which case an image tag can be used to perform the action (this form of attack is common on forum sites that permit images but block JavaScript). Applications that change state with `GET` requests are vulnerable from malicious attacks.

CSRF attacks are possible against web sites that use cookies for authentication, because browsers send all relevant cookies to the destination web site. However, CSRF attacks are not limited to exploiting cookies. For

example, Basic and Digest authentication are also vulnerable. After a user logs in with Basic or Digest authentication, the browser automatically sends the credentials until the session ends.

Note: In this context, *session* refers to the client-side session during which the user is authenticated. It is unrelated to server-side sessions or [session middleware](#).

Users can guard against CSRF vulnerabilities by:

- Logging off of web sites when they have finished using them.
- Clearing their browser's cookies periodically.

However, CSRF vulnerabilities are fundamentally a problem with the web app, not the end user.

How does ASP.NET Core MVC address CSRF?

WARNING

ASP.NET Core implements anti-request-forgery using the [ASP.NET Core data protection stack](#). ASP.NET Core data protection must be configured to work in a server farm. See [Configuring data protection](#) for more information.

ASP.NET Core anti-request-forgery default data protection configuration

In ASP.NET Core MVC 2.0 the [FormTagHelper](#) injects anti-forgery tokens for HTML form elements. For example, the following markup in a Razor file will automatically generate anti-forgery tokens:

```
<form method="post">
  <!-- form markup -->
</form>
```

The automatic generation of anti-forgery tokens for HTML form elements happens when:

- The `form` tag contains the `method="post"` attribute AND
 - The action attribute is empty. (`action=""`) OR
 - The action attribute is not supplied. (`<form method="post">`)

You can disable automatic generation of anti-forgery tokens for HTML form elements by:

- Explicitly disabling `asp-antiforgery`. For example

```
<form method="post" asp-antiforgery="false">
</form>
```

- Opt the form element out of Tag Helpers by using the Tag Helper [! opt-out symbol](#).

```
<!form method="post">
</!form>
```

- Remove the `FormTagHelper` from the view. You can remove the `FormTagHelper` from a view by adding the following directive to the Razor view:

```
@removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper,
Microsoft.AspNetCore.Mvc.TagHelpers
```

NOTE

Razor Pages are automatically protected from XSRF/CSRF. You don't have to write any additional code. See [XSRF/CSRF and Razor Pages](#) for more information.

The most common approach to defending against CSRF attacks is the synchronizer token pattern (STP). STP is a technique used when the user requests a page with form data. The server sends a token associated with the current user's identity to the client. The client sends back the token to the server for verification. If the server receives a token that doesn't match the authenticated user's identity, the request is rejected. The token is unique and unpredictable. The token can also be used to ensure proper sequencing of a series of requests (ensuring page 1 precedes page 2 which precedes page 3). All the forms in ASP.NET Core MVC templates generate antiforgery tokens. The following two examples of view logic generate antiforgery tokens:

```
<form asp-controller="Manage" asp-action="ChangePassword" method="post">

</form>

@using (Html.BeginForm("ChangePassword", "Manage"))
{

}
```

You can explicitly add an antiforgery token to a `<form>` element without using tag helpers with the HTML helper `@Html.AntiForgeryToken()`:

```
<form action="/" method="post">
    @Html.AntiForgeryToken()
</form>
```

In each of the preceding cases, ASP.NET Core will add a hidden form field similar to the following:

```
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8NrAks1dwD9CpLRyOtm6FiJB1Jr_F3FQJQDvh1HoLNJJrLA6zaMUmhjMsisu2D2tFkAiYgywQawJk9vNm36sYP1esHOtamBE
PvSk1_x--Sg8Ey2a-d9CV2zHVWvIN9MVhvKHOSyKqdZF1YDVd69XYx-rOWPw3i1HGLN6K0Km-1p83jZzF0E4WU5OGg5ns2-m9Yw" />
```

ASP.NET Core includes three [filters](#) for working with antiforgery tokens: `ValidateAntiForgeryToken`, `AutoValidateAntiforgeryToken`, and `IgnoreAntiforgeryToken`.

ValidateAntiForgeryToken

The `ValidateAntiForgeryToken` is an action filter that can be applied to an individual action, a controller, or globally. Requests made to actions that have this filter applied will be blocked unless the request includes a valid antiforgery token.

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> RemoveLogin(RemoveLoginViewModel account)
{
    ManageMessageId? message = ManageMessageId.Error;
    var user = await GetCurrentUserAsync();
    if (user != null)
    {
        var result = await _userManager.RemoveLoginAsync(user, account.LoginProvider, account.ProviderKey);
        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            message = ManageMessageId.RemoveLoginSuccess;
        }
    }
    return RedirectToAction(nameof(ManageLogins), new { Message = message });
}

```

The `ValidateAntiForgeryToken` attribute requires a token for requests to action methods it decorates, including HTTP GET requests. If you apply it broadly, you can override it with the `IgnoreAntiforgeryToken` attribute.

AutoValidateAntiforgeryToken

ASP.NET Core apps generally do not generate antiforgery tokens for HTTP safe methods (GET, HEAD, OPTIONS, and TRACE). Instead of broadly applying the `ValidateAntiForgeryToken` attribute and then overriding it with `IgnoreAntiforgeryToken` attributes, you can use the `AutoValidateAntiforgeryToken` attribute. This attribute works identically to the `ValidateAntiForgeryToken` attribute, except that it doesn't require tokens for requests made using the following HTTP methods:

- GET
- HEAD
- OPTIONS
- TRACE

We recommend you use `AutoValidateAntiforgeryToken` broadly for non-API scenarios. This ensures your POST actions are protected by default. The alternative is to ignore antiforgery tokens by default, unless `ValidateAntiForgeryToken` is applied to the individual action method. It's more likely in this scenario for a POST action method to be left unprotected, leaving your app vulnerable to CSRF attacks. Even anonymous POSTS should send the antiforgery token.

Note: APIs don't have an automatic mechanism for sending the non-cookie part of the token; your implementation will likely depend on your client code implementation. Some examples are shown below.

Example (class level):

```

[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{

```

Example (global):

```

services.AddMvc(options =>
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));

```

IgnoreAntiforgeryToken

The `IgnoreAntiforgeryToken` filter is used to eliminate the need for an antiforgery token to be present for a

given action (or controller). When applied, this filter will override `ValidateAntiForgeryToken` and/or `AutoValidateAntiForgeryToken` filters specified at a higher level (globally or on a controller).

```
[Authorize]
[AutoValidateAntiForgeryToken]
public class ManageController : Controller
{
    [HttpPost]
    [IgnoreAntiForgeryToken]
    public async Task<IActionResult> DoSomethingSafe(SomeViewModel model)
    {
        // no antiforgery token required
    }
}
```

JavaScript, AJAX, and SPAs

In traditional HTML-based applications, antiforgery tokens are passed to the server using hidden form fields. In modern JavaScript-based apps and single page applications (SPAs), many requests are made programmatically. These AJAX requests may use other techniques (such as request headers or cookies) to send the token. If cookies are used to store authentication tokens and to authenticate API requests on the server, then CSRF will be a potential problem. However, if local storage is used to store the token, CSRF vulnerability may be mitigated, since values from local storage are not sent automatically to the server with every new request. Thus, using local storage to store the antiforgery token on the client and sending the token as a request header is a recommended approach.

AngularJS

AngularJS uses a convention to address CSRF. If the server sends a cookie with the name `XSRF-TOKEN`, the Angular `$http` service will add the value from this cookie to a header when it sends a request to this server. This process is automatic; you don't need to set the header explicitly. The header name is `X-XSRF-TOKEN`. The server should detect this header and validate its contents.

For ASP.NET Core API work with this convention:

- Configure your app to provide a token in a cookie called `XSRF-TOKEN`
- Configure the antiforgery service to look for a header named `X-XSRF-TOKEN`

```
services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");
```

[View sample.](#)

JavaScript

Using JavaScript with views, you can create the token using a service from within your view. To do so, you inject the `Microsoft.AspNetCore.Antiforgery.IAntiforgery` service into the view and call `GetAndStoreTokens`, as shown:

```

@{
    ViewData["Title"] = "AJAX Demo";
}
@Inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Xsrf
@functions{
    public string GetAntiXsrfRequestToken()
    {
        return Xsrf.GetAndStoreTokens(Context).RequestToken;
    }
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<div class="row">
    <input type="button" id="antiforgery" value="Antiforgery" />
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"></script>
    <script>
        $("#antiforgery").click(function () {
            $.ajax({
                type: "post",
                dataType: "html",
                headers:
                {
                    "RequestVerificationToken": '@GetAntiXsrfRequestToken()'
                },
                url: '@Url.Action("Antiforgery", "Home")',
                success: function (result) {
                    alert(result);
                },
                error: function (err, scnd) {
                    alert(err.statusText);
                }
            });
        });
    </script>
</div>

```

This approach eliminates the need to deal directly with setting cookies from the server or reading them from the client.

JavaScript can also access tokens provided in cookies, and then use the cookie's contents to create a header with the token's value, as shown below.

```

context.Response.Cookies.Append("CSRF-TOKEN", tokens.RequestToken,
    new Microsoft.AspNetCore.Http.CookieOptions { HttpOnly = false });

```

Then, assuming you construct your script requests to send the token in a header called `X-CSRF-TOKEN`, configure the antiforgery service to look for the `X-CSRF-TOKEN` header:

```

services.AddAntiforgery(options => options.HeaderName = "X-CSRF-TOKEN");

```

The following example uses jQuery to make an AJAX request with the appropriate header:

```

var csrfToken = $.cookie("CSRF-TOKEN");

$.ajax({
  url: "/api/password/changepassword",
  contentType: "application/json",
  data: JSON.stringify({ "newPassword": "ReallySecurePassword999$$$" }),
  type: "POST",
  headers: {
    "X-CSRF-TOKEN": csrfToken
  }
});

```

Configuring Antiforgery

`IAntiforgery` provides the API to configure the antiforgery system. It can be requested in the `Configure` method of the `Startup` class. The following example uses middleware from the app's home page to generate an antiforgery token and send it in the response as a cookie (using the default Angular naming convention described above):

```

public void Configure(IApplicationBuilder app,
  IAntiforgery antiforgery)
{
  app.Use(next => context =>
  {
    string path = context.Request.Path.Value;
    if (
      string.Equals(path, "/", StringComparison.OrdinalIgnoreCase) ||
      string.Equals(path, "/index.html", StringComparison.OrdinalIgnoreCase))
    {
      // We can send the request token as a JavaScript-readable cookie,
      // and Angular will use it by default.
      var tokens = antiforgery.GetAndStoreTokens(context);
      context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken,
        new CookieOptions() { HttpOnly = false });
    }

    return next(context);
  });
  //
}

```

Options

You can customize [antiforgery options](#) in `ConfigureServices`:

```

services.AddAntiforgery(options =>
{
  options.CookieDomain = "mydomain.com";
  options.CookieName = "X-CSRF-TOKEN-COOKIENAME";
  options.CookiePath = "Path";
  options.FormFieldName = "AntiforgeryFieldname";
  options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
  options.RequireSsl = false;
  options.SuppressXFrameOptionsHeader = false;
});

```

OPTION	DESCRIPTION
CookieDomain	The domain of the cookie. Defaults to <code>null</code> .

OPTION	DESCRIPTION
CookieName	The name of the cookie. If not set, the system will generate a unique name beginning with the <code>DefaultCookiePrefix</code> ("AspNetCore.Antiforgery").
CookiePath	The path set on the cookie.
FormFieldName	The name of the hidden form field used by the antiforgery system to render antiforgery tokens in views.
HeaderName	The name of the header used by the antiforgery system. If <code>null</code> , the system will consider only form data.
RequireSsl	Specifies whether SSL is required by the antiforgery system. Defaults to <code>false</code> . If <code>true</code> , non-SSL requests will fail.
SuppressXFrameOptionsHeader	Specifies whether to suppress generation of the <code>X-Frame-Options</code> header. By default, the header is generated with a value of "SAMEORIGIN". Defaults to <code>false</code> .

See <https://docs.microsoft.com/aspnet/core/api/microsoft.aspnetcore.builder.cookieauthenticationoptions> for more info.

Extending Antiforgery

The `IAntiForgeryAdditionalDataProvider` type allows developers to extend the behavior of the anti-XSRF system by round-tripping additional data in each token. The `GetAdditionalData` method is called each time a field token is generated, and the return value is embedded within the generated token. An implementer could return a timestamp, a nonce, or any other value and then call `ValidateAdditionalData` to validate this data when the token is validated. The client's username is already embedded in the generated tokens, so there is no need to include this information. If a token includes supplemental data but no `IAntiForgeryAdditionalDataProvider` has been configured, the supplemental data is not validated.

Fundamentals

CSRF attacks rely on the default browser behavior of sending cookies associated with a domain with every request made to that domain. These cookies are stored within the browser. They frequently include session cookies for authenticated users. Cookie-based authentication is a popular form of authentication. Token-based authentication systems have been growing in popularity, especially for SPAs and other "smart client" scenarios.

Cookie-based authentication

Once a user has authenticated using their username and password, they are issued a token that can be used to identify them and validate that they have been authenticated. The token is stored as a cookie that accompanies every request the client makes. Generating and validating this cookie is done by the cookie authentication middleware. ASP.NET Core provides cookie [middleware](#) which serializes a user principal into an encrypted cookie and then, on subsequent requests, validates the cookie, recreates the principal and assigns it to the `User` property on `HttpContext`.

When a cookie is used, The authentication cookie is just a container for the forms authentication ticket. The ticket is passed as the value of the forms authentication cookie with each request and is used by forms authentication, on the server, to identify an authenticated user.

When a user is logged in to a system, a user session is created on the server-side and is stored in a database or some other persistent store. The system generates a session key that points to the actual session in the data

store and it is sent as a client side cookie. The web server will check this session key any time a user requests a resource that requires authorization. The system checks whether the associated user session has the privilege to access the requested resource. If so, the request continues. Otherwise, the request returns as not authorized. In this approach, cookies are used to make the application appear to be stateful, since it is able to "remember" that the user has previously authenticated with the server.

User tokens

Token-based authentication doesn't store session on the server. Instead, when a user is logged in they are issued a token (not an antiforgery token). This token holds all the data that is required to validate the token. It also contains user information, in the form of [claims](#). When a user wants to access a server resource requiring authentication, the token is sent to the server with an additional authorization header in form of Bearer {token}. This makes the application stateless since in each subsequent request the token is passed in the request for server-side validation. This token is not *encrypted*; rather it is *encoded*. On the server-side the token can be decoded to access the raw information within the token. To send the token in subsequent requests, you can either store it in browser's local storage or in a cookie. You don't have to worry about XSRF vulnerability if your token is stored in the local storage, but it is an issue if the token is stored in a cookie.

Multiple applications are hosted in one domain

Even though `example1.cloudapp.net` and `example2.cloudapp.net` are different hosts, there is an implicit trust relationship between all hosts under the `*.cloudapp.net` domain. This implicit trust relationship allows potentially untrusted hosts to affect each other's cookies (the same-origin policies that govern AJAX requests do not necessarily apply to HTTP cookies). The ASP.NET Core runtime provides some mitigation in that the username is embedded into the field token, so even if a malicious subdomain is able to overwrite a session token it will be unable to generate a valid field token for the user. However, when hosted in such an environment the built-in anti-XSRF routines still cannot defend against session hijacking or login CSRF attacks. Shared hosting environments are vulnerable to session hijacking, login CSRF, and other attacks.

Additional Resources

- [XSRF on Open Web Application Security Project \(OWASP\)](#).

Preventing Open Redirect Attacks in an ASP.NET Core app

9/30/2017 • 3 min to read • [Edit Online](#)

A web app that redirects to a URL that is specified via the request such as the querystring or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack.

Whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn't been tampered with. ASP.NET Core has built-in functionality to help protect apps from open redirect (also known as open redirection) attacks.

What is an open redirect attack?

Web applications frequently redirect users to a login page when they access resources that require authentication. The redirection typically includes a `returnUrl` querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in. After the user authenticates, they are redirected to the URL they had originally requested.

Because the destination URL is specified in the querystring of the request, a malicious user could tamper with the querystring. A tampered querystring could allow the site to redirect the user to an external, malicious site. This technique is called an open redirect (or redirection) attack.

An example attack

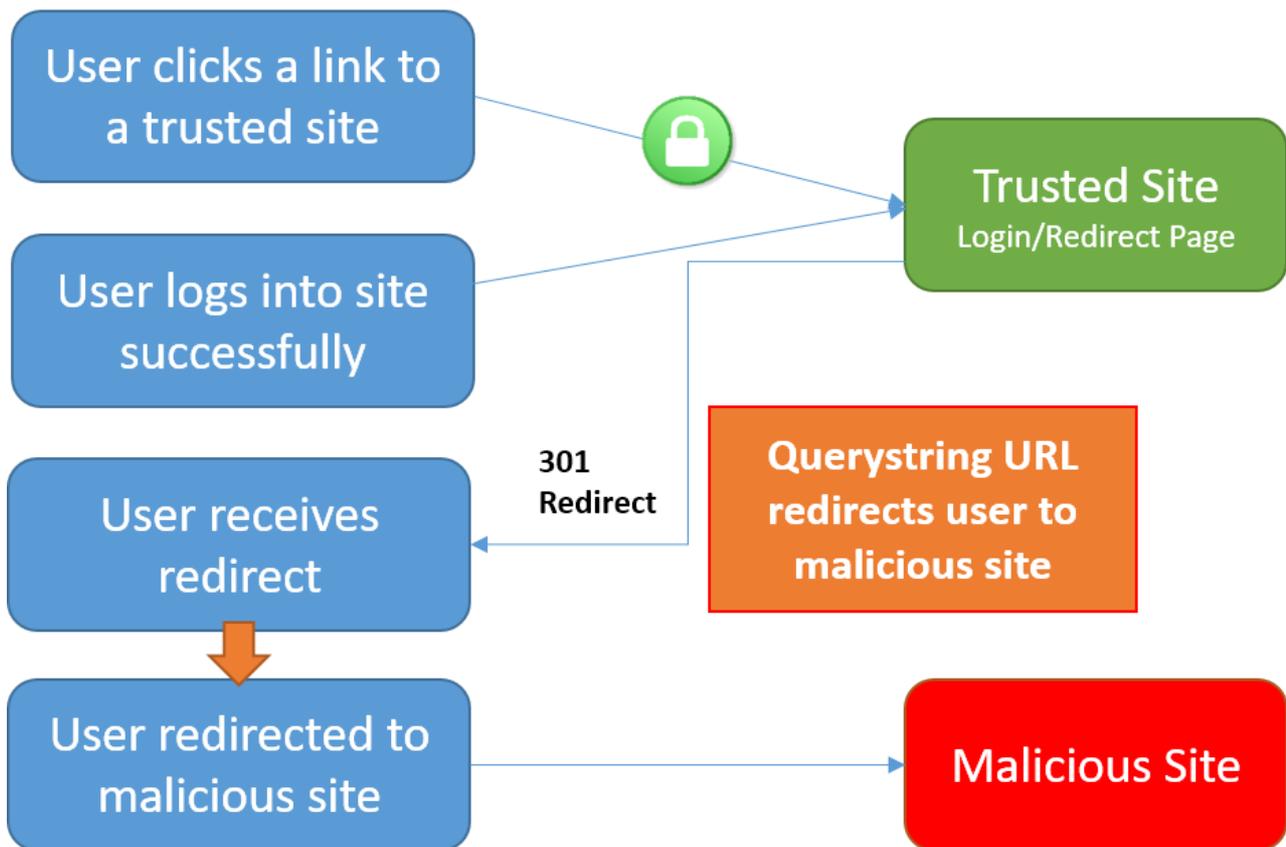
A malicious user could develop an attack intended to allow the malicious user access to a user's credentials or sensitive information on your app. To begin the attack, they convince the user to click a link to your site's login page, with a `returnUrl` querystring value added to the URL. For example, the [NerdDinner.com](#) sample application (written for ASP.NET MVC) includes such a login page here:

`http://nerddinner.com/Account/LogOn?returnUrl=/Home/About`. The attack then follows these steps:

1. User clicks a link to `http://nerddinner.com/Account/LogOn?returnUrl=http://nerddinner.com/Account/LogOn` (note, second URL is `nerddinner`, not `nerddinner`).
2. The user logs in successfully.
3. The user is redirected (by the site) to `http://nerddinner.com/Account/LogOn` (malicious site that looks like real site).
4. The user logs in again (giving malicious site their credentials) and is redirected back to the real site.

The user will likely believe their first attempt to log in failed, and their second one was successful. They'll most likely remain unaware their credentials have been compromised.

Open Redirection Attack Process



In addition to login pages, some sites provide redirect pages or endpoints. Imagine your app has a page with an open redirect, `/Home/Redirect`. An attacker could create, for example, a link in an email that goes to `[yoursite]/Home/Redirect?url=http://phishingsite.com/Home/Login`. A typical user will look at the URL and see it begins with your site name. Trusting that, they will click the link. The open redirect would then send the user to the phishing site, which looks identical to yours, and the user would likely login to what they believe is your site.

Protecting against open redirect attacks

When developing web applications, treat all user-provided data as untrustworthy. If your application has functionality that redirects the user based on the contents of the URL, ensure that such redirects are only done locally within your app (or to a known URL, not any URL that may be supplied in the querystring).

LocalRedirect

Use the `LocalRedirect` helper method from the base `Controller` class:

```
public IActionResult SomeAction(string redirectUrl)
{
    return LocalRedirect(redirectUrl);
}
```

`LocalRedirect` will throw an exception if a non-local URL is specified. Otherwise, it behaves just like the `Redirect` method.

IsLocalUrl

Use the `IsLocalUrl` method to test URLs before redirecting:

The following example shows how to check whether a URL is local before redirecting.

```
private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Home");
    }
}
```

The `IsLocalUrl` method protects users from being inadvertently redirected to a malicious site. You can log the details of the URL that was provided when a non-local URL is supplied in a situation where you expected a local URL. Logging redirect URLs may help in diagnosing redirection attacks.

Preventing Cross-Site Scripting

11/1/2017 • 6 min to read • [Edit Online](#)

By [Rick Anderson](#)

Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages the attackers scripts will run, enabling the attacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it in a page without validating, encoding or escaping it.

Protecting your application against XSS

At a basic level XSS works by tricking your application into inserting a `<script>` tag into your rendered page, or by inserting an `on*` event into an element. Developers should use the following prevention steps to avoid introducing XSS into their application.

1. Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by an attacker, HTML form inputs, query strings, HTTP headers, even data sourced from a database as an attacker may be able to breach your database even if they cannot breach your application.
2. Before putting untrusted data inside an HTML element ensure it is HTML encoded. HTML encoding takes characters such as `<` and changes them into a safe form like `<`
3. Before putting untrusted data into an HTML attribute ensure it is HTML attribute encoded. HTML attribute encoding is a superset of HTML encoding and encodes additional characters such as `"` and `'`.
4. Before putting untrusted data into JavaScript place the data in an HTML element whose contents you retrieve at runtime. If this is not possible then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example `<` would be encoded as `\u003C`.
5. Before putting untrusted data into a URL query string ensure it is URL encoded.

HTML Encoding using Razor

The Razor engine used in MVC automatically encodes all output sourced from variables, unless you work really hard to prevent it doing so. It uses HTML Attribute encoding rules whenever you use the `@` directive. As HTML attribute encoding is a superset of HTML encoding this means you don't have to concern yourself with whether you should use HTML encoding or HTML attribute encoding. You must ensure that you only use `@` in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Tag helpers will also encode input you use in tag parameters.

Take the following Razor view;

```
@{
    var untrustedInput = "<\u0027123\u0027>";
}

@untrustedInput
```

This view outputs the contents of the *untrustedInput* variable. This variable includes some characters which are used in XSS attacks, namely <, " and >. Examining the source shows the rendered output encoded as:

```
&lt;&quot;123&quot;&gt;
```

WARNING

ASP.NET Core MVC provides an `HtmlString` class which is not automatically encoded upon output. This should never be used in combination with untrusted input as this will expose an XSS vulnerability.

Javascript Encoding using Razor

There may be times you want to insert a value into JavaScript to process in your view. There are two ways to do this. The safest way to insert simple values is to place the value in a data attribute of a tag and retrieve it in your JavaScript. For example:

```
@{
    var untrustedInput = "<\\"123\>";
}

<div
    id="injectedData"
    data-untrustedinput="@untrustedInput" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    document.write(clientSideUntrustedInputOldStyle);
    document.write("<br />")
    document.write(clientSideUntrustedInputHtml5);
</script>
```

This will produce the following HTML

```
<div
    id="injectedData"
    data-untrustedinput="&lt;&quot;123&quot;&gt;" />

<script>
    var injectedData = document.getElementById("injectedData");

    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    document.write(clientSideUntrustedInputOldStyle);
    document.write("<br />")
    document.write(clientSideUntrustedInputHtml5);
</script>
```

Which, when it runs, will render the following;

```
<"123">
  <"123">
```

You can also call the JavaScript encoder directly,

```
@using System.Text.Encodings.Web;
@Inject JavaScriptEncoder encoder;

@{
    var untrustedInput = "<\\"123\\">";
}

<script>
    document.write("@encoder.Encode(untrustedInput)");
</script>
```

This will render in the browser as follows;

```
<script>
    document.write("\u003C\u0022123\u0022\u003E");
</script>
```

WARNING

Do not concatenate untrusted input in JavaScript to create DOM elements. You should use `createElement()` and assign property values appropriately such as `node.TextContent=`, or use `element.SetAttribute()` / `element[attribute]=` otherwise you expose yourself to DOM-based XSS.

Accessing encoders in code

The HTML, JavaScript and URL encoders are available to your code in two ways, you can inject them via [dependency injection](#) or you can use the default encoders contained in the `System.Text.Encodings.Web` namespace. If you use the default encoders then any you applied to character ranges to be treated as safe will not take effect - the default encoders use the safest encoding rules possible.

To use the configurable encoders via DI your constructors should take an *HtmlEncoder*, *JavaScriptEncoder* and *UrlEncoder* parameter as appropriate. For example;

```
public class HomeController : Controller
{
    HtmlEncoder _htmlEncoder;
    JavaScriptEncoder _javascriptEncoder;
    UrlEncoder _urlEncoder;

    public HomeController(HtmlEncoder htmlEncoder,
        JavaScriptEncoder javascriptEncoder,
        UrlEncoder urlEncoder)
    {
        _htmlEncoder = htmlEncoder;
        _javascriptEncoder = javascriptEncoder;
        _urlEncoder = urlEncoder;
    }
}
```

Encoding URL Parameters

If you want to build a URL query string with untrusted input as a value use the `UrlEncoder` to encode the value. For example,

```
var example = "\"Quoted Value with spaces and &\"";
var encodedValue = _urlEncoder.Encode(example);
```

After encoding the `encodedValue` variable will contain `%22Quoted%20Value%20with%20spaces%20and%20%26%22`. Spaces, quotes, punctuation and other unsafe characters will be percent encoded to their hexadecimal value, for example a space character will become `%20`.

WARNING

Do not use untrusted input as part of a URL path. Always pass untrusted input as a query string value.

Customizing the Encoders

By default encoders use a safe list limited to the Basic Latin Unicode range and encode all characters outside of that range as their character code equivalents. This behavior also affects Razor TagHelper and HtmlHelper rendering as it will use the encoders to output your strings.

The reasoning behind this is to protect against unknown or future browser bugs (previous browser bugs have tripped up parsing based on the processing of non-English characters). If your web site makes heavy use of non-Latin characters, such as Chinese, Cyrillic or others this is probably not the behavior you want.

You can customize the encoder safe lists to include Unicode ranges appropriate to your application during startup, in `ConfigureServices()`.

For example, using the default configuration you might use a Razor HtmlHelper like so;

```
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

When you view the source of the web page you will see it has been rendered as follows, with the Chinese text encoded;

```
<p>This link text is in Chinese: <a href="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

To widen the characters treated as safe by the encoder you would insert the following line into the

`ConfigureServices()` method in `startup.cs`;

```
services.AddSingleton<HtmlEncoder>(
    HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,
        UnicodeRanges.CjkUnifiedIdeographs }));
```

This example widens the safe list to include the Unicode Range `CjkUnifiedIdeographs`. The rendered output would now become

```
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

Safe list ranges are specified as Unicode code charts, not languages. The [Unicode standard](#) has a list of [code charts](#) you can use to find the chart containing your characters. Each encoder, Html, JavaScript and Url, must be

configured separately.

NOTE

Customization of the safe list only affects encoders sourced via DI. If you directly access an encoder via `System.Text.Encodings.Web.*Encoder.Default` then the default, Basic Latin only safelist will be used.

Where should encoding take place?

The general accepted practice is that encoding takes place at the point of output and encoded values should never be stored in a database. Encoding at the point of output allows you to change the use of data, for example, from HTML to a query string value. It also enables you to easily search your data without having to encode values before searching and allows you to take advantage of any changes or bug fixes made to encoders.

Validation as an XSS prevention technique

Validation can be a useful tool in limiting XSS attacks. For example, a simple numeric string containing only the characters 0-9 will not trigger an XSS attack. Validation becomes more complicated should you wish to accept HTML in user input - parsing HTML input is difficult, if not impossible. Markdown and other text formats would be a safer option for rich input. You should never rely on validation alone. Always encode untrusted input before output, no matter what validation you have performed.

Enabling Cross-Origin Requests (CORS)

11/29/2017 • 8 min to read • [Edit Online](#)

By [Mike Wasson](#), [Shayne Boyer](#), and [Tom Dykstra](#)

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the *same-origin policy*, and prevents a malicious site from reading sensitive data from another site. However, sometimes you might want to let other sites make cross-origin requests to your web API.

[Cross Origin Resource Sharing](#) (CORS) is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others. CORS is safer and more flexible than earlier techniques such as [JSONP](#). This topic shows how to enable CORS in an ASP.NET Core application.

What is "same origin"?

Two URLs have the same origin if they have identical schemes, hosts, and ports. ([RFC 6454](#))

These two URLs have the same origin:

- `http://example.com/foo.html`
- `http://example.com/bar.html`

These URLs have different origins than the previous two:

- `http://example.net` - Different domain
- `http://www.example.com/foo.html` - Different subdomain
- `https://example.com/foo.html` - Different scheme
- `http://example.com:9000/foo.html` - Different port

NOTE

Internet Explorer does not consider the port when comparing origins.

Setting up CORS

To set up CORS for your application add the `Microsoft.AspNetCore.Cors` package to your project.

Add the CORS services in Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors();
}
```

Enabling CORS with middleware

To enable CORS for your entire application add the CORS middleware to your request pipeline using the `UseCors` extension method. Note that the CORS middleware must precede any defined endpoints in your app that you want

to support cross-origin requests (ex. before any call to `UseMvc`).

You can specify a cross-origin policy when adding the CORS middleware using the `CorsPolicyBuilder` class. There are two ways to do this. The first is to call `UseCors` with a lambda:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Shows UseCors with CorsPolicyBuilder.
    app.UseCors(builder =>
        builder.WithOrigins("http://example.com"));

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

Note: The URL must be specified without a trailing slash (`/`). If the URL terminates with `/`, the comparison will return `false` and no header will be returned.

The lambda takes a `CorsPolicyBuilder` object. You'll find a list of the [configuration options](#) later in this topic. In this example, the policy allows cross-origin requests from `http://example.com` and no other origins.

Note that `CorsPolicyBuilder` has a fluent API, so you can chain method calls:

```
app.UseCors(builder =>
    builder.WithOrigins("http://example.com")
        .AllowAnyHeader()
);
```

The second approach is to define one or more named CORS policies, and then select the policy by name at run time.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy("AllowSpecificOrigin",
            builder => builder.WithOrigins("http://example.com"));
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Shows UseCors with named policy.
    app.UseCors("AllowSpecificOrigin");
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}

```

This example adds a CORS policy named "AllowSpecificOrigin". To select the policy, pass the name to `UseCors`.

Enabling CORS in MVC

You can alternatively use MVC to apply specific CORS per action, per controller, or globally for all controllers. When using MVC to enable CORS the same CORS services are used, but the CORS middleware is not.

Per action

To specify a CORS policy for a specific action add the `[EnableCors]` attribute to the action. Specify the policy name.

```

[HttpGet]
[EnableCors("AllowSpecificOrigin")]
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}

```

Per controller

To specify the CORS policy for a specific controller add the `[EnableCors]` attribute to the controller class. Specify the policy name.

```

[Route("api/[controller]")]
[EnableCors("AllowSpecificOrigin")]
public class ValuesController : Controller

```

Globally

You can enable CORS globally for all controllers by adding the `CorsAuthorizationFilterFactory` filter to the global filter collection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new CorsAuthorizationFilterFactory("AllowSpecificOrigin"));
    });
}
```

The precedence order is: Action, controller, global. Action-level policies take precedence over controller-level policies, and controller-level policies take precedence over global policies.

Disable CORS

To disable CORS for a controller or action, use the `[DisableCors]` attribute.

```
[HttpGet("{id}")]
[DisableCors]
public string Get(int id)
{
    return "value";
}
```

CORS policy options

This section describes the various options that you can set in a CORS policy.

- [Set the allowed origins](#)
- [Set the allowed HTTP methods](#)
- [Set the allowed request headers](#)
- [Set the exposed response headers](#)
- [Credentials in cross-origin requests](#)
- [Set the preflight expiration time](#)

For some options it may be helpful to read [How CORS works](#) first.

Set the allowed origins

To allow one or more specific origins:

```
options.AddPolicy("AllowSpecificOrigins",
builder =>
{
    builder.WithOrigins("http://example.com", "http://www.contoso.com");
});
```

To allow all origins:

```
options.AddPolicy("AllowAllOrigins",
builder =>
{
    builder.AllowAnyOrigin();
});
```

Consider carefully before allowing requests from any origin. It means that literally any website can make AJAX

calls to your API.

Set the allowed HTTP methods

To allow all HTTP methods:

```
options.AddPolicy("AllowAllMethods",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .AllowAnyMethod();
    });
```

This affects pre-flight requests and Access-Control-Allow-Methods header.

Set the allowed request headers

A CORS preflight request might include an Access-Control-Request-Headers header, listing the HTTP headers set by the application (the so-called "author request headers").

To whitelist specific headers:

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .WithHeaders("accept", "content-type", "origin", "x-custom-header");
    });
```

To allow all author request headers:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .AllowAnyHeader();
    });
```

Browsers are not entirely consistent in how they set Access-Control-Request-Headers. If you set headers to anything other than "*", you should include at least "accept", "content-type", and "origin", plus any custom headers that you want to support.

Set the exposed response headers

By default, the browser does not expose all of the response headers to the application. (See <http://www.w3.org/TR/cors/#simple-response-header>.) The response headers that are available by default are:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

The CORS spec calls these *simple response headers*. To make other headers available to the application:

```
options.AddPolicy("ExposeResponseHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .WithExposedHeaders("x-custom-header");
    });
```

Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser does not send any credentials with a cross-origin request. Credentials include cookies as well as HTTP authentication schemes. To send credentials with a cross-origin request, the client must set XMLHttpRequest.withCredentials to true.

Using XMLHttpRequest directly:

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'http://www.example.com/api/test');
xhr.withCredentials = true;
```

In jQuery:

```
$.ajax({
    type: 'get',
    url: 'http://www.example.com/home',
    xhrFields: {
        withCredentials: true
    }
});
```

In addition, the server must allow the credentials. To allow cross-origin credentials:

```
options.AddPolicy("AllowCredentials",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .AllowCredentials();
    });
```

Now the HTTP response will include an Access-Control-Allow-Credentials header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials, but the response does not include a valid Access-Control-Allow-Credentials header, the browser will not expose the response to the application, and the AJAX request fails.

Be very careful about allowing cross-origin credentials, because it means a website at another domain can send a logged-in user's credentials to your app on the user's behalf, without the user being aware. The CORS spec also states that setting origins to "*" (all origins) is invalid if the Access-Control-Allow-Credentials header is present.

Set the preflight expiration time

The Access-Control-Max-Age header specifies how long the response to the preflight request can be cached. To set this header:

```
options.AddPolicy("SetPreflightExpiration",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
    });
```

How CORS works

This section describes what happens in a CORS request, at the level of the HTTP messages. It's important to understand how CORS works, so that you can configure your CORS policy correctly, and troubleshoot if things don't work as you expect.

The CORS specification introduces several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests; you don't need to do anything special in your JavaScript code.

Here is an example of a cross-origin request. The "Origin" header gives the domain of the site that is making the request:

```
GET http://myservice.azurewebsites.net/api/test HTTP/1.1
Referer: http://myclient.azurewebsites.net/
Accept: */*
Accept-Language: en-US
Origin: http://myclient.azurewebsites.net
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
```

If the server allows the request, it sets the Access-Control-Allow-Origin header in the response. The value of this header either matches the Origin header from the request, or is the wildcard value "*", meaning that any origin is allowed:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Date: Wed, 20 May 2015 06:27:30 GMT
Content-Length: 12

Test message
```

If the response does not include the Access-Control-Allow-Origin header, the AJAX request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser does not make the response available to the client application.

Preflight Requests

For some CORS requests, the browser sends an additional request, called a "preflight request", before it sends the actual request for the resource. The browser can skip the preflight request if the following conditions are true:

- The request method is GET, HEAD, or POST, and
- The application does not set any request headers other than Accept, Accept-Language, Content-Language, Content-Type, or Last-Event-ID, and
- The Content-Type header (if set) is one of the following:

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

The rule about request headers applies to headers that the application sets by calling `setRequestHeader` on the `XMLHttpRequest` object. (The CORS specification calls these "author request headers".) The rule does not apply to headers the browser can set, such as `User-Agent`, `Host`, or `Content-Length`.

Here is an example of a preflight request:

```
OPTIONS http://myservice.azurewebsites.net/api/test HTTP/1.1
Accept: */*
Origin: http://myclient.azurewebsites.net
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: accept, x-my-custom-header
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
Content-Length: 0
```

The pre-flight request uses the HTTP `OPTIONS` method. It includes two special headers:

- `Access-Control-Request-Method`: The HTTP method that will be used for the actual request.
- `Access-Control-Request-Headers`: A list of request headers that the application set on the actual request. (Again, this does not include headers that the browser sets.)

Here is an example response, assuming that the server allows the request:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 0
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Access-Control-Allow-Headers: x-my-custom-header
Access-Control-Allow-Methods: PUT
Date: Wed, 20 May 2015 06:33:22 GMT
```

The response includes an `Access-Control-Allow-Methods` header that lists the allowed methods, and optionally an `Access-Control-Allow-Headers` header, which lists the allowed headers. If the preflight request succeeds, the browser sends the actual request, as described earlier.

Performance

1/10/2018 • 1 min to read • [Edit Online](#)

- Caching
 - In-memory caching
 - Work with a distributed cache
 - Response caching
- Response compression middleware

Caching

11/29/2017 • 1 min to read • [Edit Online](#)

- [In-memory caching](#)
- [Working with a distributed cache](#)
- [Detect changes with change tokens](#)
- [Response caching](#)
- [Response Caching Middleware](#)
- [Cache Tag Helper](#)
- [Distributed Cache Tag Helper](#)

In-memory caching in ASP.NET Core

11/29/2017 • 5 min to read • [Edit Online](#)

By [Rick Anderson](#), [John Luo](#), and [Steve Smith](#)

[View or download sample code \(how to download\)](#)

Caching basics

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently. Caching makes a copy of data that can be returned much faster than from the original source. You should write and test your app to never depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the `IMemoryCache`, which represents a cache stored in the memory of the web server. Apps which run on a server farm of multiple servers should ensure that sessions are sticky when using the in-memory cache. Sticky sessions ensure that subsequent requests from a client all go to the same server. For example, Azure Web apps use [Application Request Routing \(ARR\)](#) to route all subsequent requests to the same server.

Non-sticky sessions in a web farm require a [distributed cache](#) to avoid cache consistency problems. For some apps, a distributed cache can support higher scale out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

The `IMemoryCache` cache will evict cache entries under memory pressure unless the [cache priority](#) is set to `CacheItemPriority.NeverRemove`. You can set the `CacheItemPriority` to adjust the priority the cache evicts items under memory pressure.

The in-memory cache can store any object; the distributed cache interface is limited to `byte[]`.

Using IMemoryCache

In-memory caching is a *service* that is referenced from your app using [Dependency Injection](#). Call

`AddMemoryCache` in `ConfigureServices`:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMemoryCache();
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvcWithDefaultRoute();
    }
}
```

Request the `IMemoryCache` instance in the constructor:

```

public class HomeController : Controller
{
    private IMemoryCache _cache;

    public HomeController(IMemoryCache memoryCache)
    {
        _cache = memoryCache;
    }
}

```

`IMemoryCache` requires NuGet package "Microsoft.Extensions.Caching.Memory".

The following code uses `TryGetValue` to check if the current time is in the cache. If the item is not cached, a new entry is created and added to the cache with `Set`.

```

public IActionResult CacheTryGetValueSet()
{
    DateTime cacheEntry;

    // Look for cache key.
    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now;

        // Set cache options.
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Save data in cache.
        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }

    return View("Cache", cacheEntry);
}

```

The current time and the cached time is displayed:

```

@model DateTime?

<div>
    <h2>Actions</h2>
    <ul>
        <li><a asp-controller="Home" asp-action="CacheTryGetValueSet">TryGetValue and Set</a></li>
        <li><a asp-controller="Home" asp-action="CacheGet">Get</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreate">GetOrCreate</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAsync">GetOrCreateAsync</a></li>
        <li><a asp-controller="Home" asp-action="CacheRemove">Remove</a></li>
    </ul>
</div>

<h3>Current Time: @DateTime.Now.TimeOfDay.ToString()</h3>
<h3>Cached Time: @(Model == null ? "No cached entry found" : Model.Value.TimeOfDay.ToString())</h3>

```

The cached `DateTime` value will remain in the cache while there are requests within the timeout period (and no eviction due to memory pressure). The image below shows the current time and an older time retrieved from cache:

localhost × +

localhost:1234/Home/CacheTryGetValueSet

Scenarios

- [Basic cache operations](#)
- [Cache entry with eviction callback](#)
- [Dependent cache entries](#)

Actions

- [TryGetValue and Set](#)
- [Get](#)
- [GetOrCreate](#)
- [GetOrCreateAsync](#)
- [Remove](#)

Current Time: 17:04:01.1913080

Cached Time: 17:03:39.9454218

The following code uses [GetOrCreate](#) and [GetOrCreateAsync](#) to cache data.

```
public IActionResult CacheGetOrCreate()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}

public async Task<IActionResult> CacheGetOrCreateAsync()
{
    var cacheEntry = await
        _cache.GetOrCreateAsync(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return Task.FromResult(DateTime.Now);
    });

    return View("Cache", cacheEntry);
}
```

The following code calls [Get](#) to fetch the cached time:

```
public IActionResult CacheGet()
{
    var cacheEntry = _cache.Get<DateTime?>(CacheKeys.Entry);
    return View("Cache", cacheEntry);
}
```

See [IMemoryCache methods](#) and [CacheExtensions methods](#) for a description of the cache methods.

Using MemoryCacheEntryOptions

The following sample:

- Sets the absolute expiration time. This is the maximum time the entry can be cached and prevents the item from becoming too stale when the sliding expiration is continuously renewed.
- Sets a sliding expiration time. Requests that access this cached item will reset the sliding expiration clock.
- Sets the cache priority to `CacheItemPriority.NeverRemove`.
- Sets a [PostEvictionDelegate](#) that will be called after the entry is evicted from the cache. The callback is run on a different thread from the code that removes the item from the cache.

```

public IActionResult CreateCallbackEntry()
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        // Pin to cache.
        .SetPriority(CacheItemPriority.NeverRemove)
        // Add eviction callback
        .RegisterPostEvictionCallback(callback: EvictionCallback, state: this);

    _cache.Set(CacheKeys.CallbackEntry, DateTime.Now, cacheEntryOptions);

    return RedirectToAction("GetCallbackEntry");
}

public IActionResult GetCallbackEntry()
{
    return View("Callback", new CallbackViewModel
    {
        CachedTime = _cache.Get<DateTime?>(CacheKeys.CallbackEntry),
        Message = _cache.Get<string>(CacheKeys.CallbackMessage)
    });
}

public IActionResult RemoveCallbackEntry()
{
    _cache.Remove(CacheKeys.CallbackEntry);
    return RedirectToAction("GetCallbackEntry");
}

private static void EvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.CallbackMessage, message);
}

```

Cache dependencies

The following sample shows how to expire a cache entry if a dependent entry expires. A

`CancellationChangeToken` is added to the cached item. When `Cancel` is called on the `CancellationTokenSource`, both cache entries are evicted.

```

public IActionResult CreateDependentEntries()
{
    var cts = new CancellationTokenSource();
    _cache.Set(CacheKeys.DependentCTS, cts);

    using (var entry = _cache.CreateEntry(CacheKeys.Parent))
    {
        // expire this entry if the dependant entry expires.
        entry.Value = DateTime.Now;
        entry.RegisterPostEvictionCallback(DependentEvictionCallback, this);

        _cache.Set(CacheKeys.Child,
            DateTime.Now,
            new CancellationChangeToken(cts.Token));
    }

    return RedirectToAction("GetDependentEntries");
}

public IActionResult GetDependentEntries()
{
    return View("Dependent", new DependentViewModel
    {
        ParentCachedTime = _cache.Get<DateTime?>(CacheKeys.Parent),
        ChildCachedTime = _cache.Get<DateTime?>(CacheKeys.Child),
        Message = _cache.Get<string>(CacheKeys.DependentMessage)
    });
}

public IActionResult RemoveChildEntry()
{
    _cache.Get<CancellationTokenSource>(CacheKeys.DependentCTS).Cancel();
    return RedirectToAction("GetDependentEntries");
}

private static void DependentEvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Parent entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.DependentMessage, message);
}

```

Using a `CancellationTokenSource` allows multiple cache entries to be evicted as a group. With the `using` pattern in the code above, cache entries created inside the `using` block will inherit triggers and expiration settings.

Additional notes

- When using a callback to repopulate a cache item:
 - Multiple requests can find the cached key value empty because the callback hasn't completed.
 - This can result in several threads repopulating the cached item.
- When one cache entry is used to create another, the child copies the parent entry's expiration tokens and time-based expiration settings. The child is not expired by manual removal or updating of the parent entry.

Additional resources

- [Working with a distributed cache](#)
- [Detect changes with change tokens](#)
- [Response caching](#)
- [Response Caching Middleware](#)

- [Cache Tag Helper](#)
- [Distributed Cache Tag Helper](#)

Working with a distributed cache in ASP.NET Core

11/29/2017 • 6 min to read • [Edit Online](#)

By [Steve Smith](#)

Distributed caches can improve the performance and scalability of ASP.NET Core apps, especially when hosted in a cloud or server farm environment. This article explains how to work with ASP.NET Core's built-in distributed cache abstractions and implementations.

[View or download sample code \(how to download\)](#)

What is a distributed cache

A distributed cache is shared by multiple app servers (see [Caching Basics](#)). The information in the cache is not stored in the memory of individual web servers, and the cached data is available to all of the app's servers. This provides several advantages:

1. Cached data is coherent on all web servers. Users don't see different results depending on which web server handles their request
2. Cached data survives web server restarts and deployments. Individual web servers can be removed or added without impacting the cache.
3. The source data store has fewer requests made to it (than with multiple in-memory caches or no cache at all).

NOTE

If using a SQL Server Distributed Cache, some of these advantages are only true if a separate database instance is used for the cache than for the app's source data.

Like any cache, a distributed cache can dramatically improve an app's responsiveness, since typically data can be retrieved from the cache much faster than from a relational database (or web service).

Cache configuration is implementation specific. This article describes how to configure both Redis and SQL Server distributed caches. Regardless of which implementation is selected, the app interacts with the cache using a common `IDistributedCache` interface.

The IDistributedCache Interface

The `IDistributedCache` interface includes synchronous and asynchronous methods. The interface allows items to be added, retrieved, and removed from the distributed cache implementation. The `IDistributedCache` interface includes the following methods:

Get, GetAsync

Takes a string key and retrieves a cached item as a `byte[]` if found in the cache.

Set, SetAsync

Adds an item (as `byte[]`) to the cache using a string key.

Refresh, RefreshAsync

Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).

Remove, RemoveAsync

Removes a cache entry based on its key.

To use the `IDistributedCache` interface:

1. Add the required NuGet packages to your project file.
2. Configure the specific implementation of `IDistributedCache` in your `Startup` class's `ConfigureServices` method, and add it to the container there.
3. From the app's [Middleware](#) or MVC controller classes, request an instance of `IDistributedCache` from the constructor. The instance will be provided by [Dependency Injection](#) (DI).

NOTE

There is no need to use a Singleton or Scoped lifetime for `IDistributedCache` instances (at least for the built-in implementations). You can also create an instance wherever you might need one (instead of using [Dependency Injection](#)), but this can make your code harder to test, and violates the [Explicit Dependencies Principle](#).

The following example shows how to use an instance of `IDistributedCache` in a simple middleware component:

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Caching.Distributed;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DistCacheSample
{
    public class StartTimeHeader
    {
        private readonly RequestDelegate _next;
        private readonly IDistributedCache _cache;

        public StartTimeHeader(RequestDelegate next,
            IDistributedCache cache)
        {
            _next = next;
            _cache = cache;
        }

        public async Task Invoke(HttpContext httpContext)
        {
            string startTimeString = "Not found.";
            var value = await _cache.GetAsync("lastServerStartTime");
            if (value != null)
            {
                startTimeString = Encoding.UTF8.GetString(value);
            }

            httpContext.Response.Headers.Append("Last-Server-Start-Time", startTimeString);

            await _next.Invoke(httpContext);
        }
    }

    // Extension method used to add the middleware to the HTTP request pipeline.
    public static class StartTimeHeaderExtensions
    {
        public static IApplicationBuilder UseStartTimeHeader(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<StartTimeHeader>();
        }
    }
}

```

In the code above, the cached value is read, but never written. In this sample, the value is only set when a server starts up, and doesn't change. In a multi-server scenario, the most recent server to start will overwrite any previous values that were set by other servers. The `Get` and `Set` methods use the `byte[]` type. Therefore, the string value must be converted using `Encoding.UTF8.GetString` (for `Get`) and `Encoding.UTF8.GetBytes` (for `Set`).

The following code from *Startup.cs* shows the value being set:

```

public void Configure(IApplicationBuilder app,
    IDistributedCache cache)
{
    var serverStartTimeString = DateTime.Now.ToString();
    byte[] val = Encoding.UTF8.GetBytes(serverStartTimeString);
    var cacheEntryOptions = new DistributedCacheEntryOptions()
        .SetSlidingExpiration(TimeSpan.FromSeconds(30));
    cache.Set("lastServerStartTime", val, cacheEntryOptions);
}

```

NOTE

Since `IDistributedCache` is configured in the `ConfigureServices` method, it is available to the `Configure` method as a parameter. Adding it as a parameter will allow the configured instance to be provided through DI.

Using a Redis distributed cache

[Redis](#) is an open source in-memory data store, which is often used as a distributed cache. You can use it locally, and you can configure an [Azure Redis Cache](#) for your Azure-hosted ASP.NET Core apps. Your ASP.NET Core app configures the cache implementation using a `RedisDistributedCache` instance.

You configure the Redis implementation in `ConfigureServices` and access it in your app code by requesting an instance of `IDistributedCache` (see the code above).

In the sample code, a `RedisCache` implementation is used when the server is configured for a `Staging` environment. Thus the `ConfigureStagingServices` method configures the `RedisCache`:

```

/// <summary>
/// Use Redis Cache in Staging
/// </summary>
/// <param name="services"></param>
public void ConfigureStagingServices(IServiceCollection services)
{
    services.AddDistributedRedisCache(options =>
    {
        options.Configuration = "localhost";
        options.InstanceName = "SampleInstance";
    });
}

```

NOTE

To install Redis on your local machine, install the chocolatey package <https://chocolatey.org/packages/redis-64/> and run `redis-server` from a command prompt.

Using a SQL Server distributed cache

The `SqlServerCache` implementation allows the distributed cache to use a SQL Server database as its backing store. To create SQL Server table you can use `sql-cache` tool, the tool creates a table with the name and schema you specify.

To use the `sql-cache` tool, add `SqlConfig.Tools` to the `<ItemGroup>` element of the `.csproj` file and run `dotnet restore`.

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.Extensions.Caching.SqlConfig.Tools" Version="1.0.0-msbuild3-
final" />
</ItemGroup>
```

Test SqlConfig.Tools by running the following command

```
C:\DistCacheSample\src\DistCacheSample>dotnet sql-cache create --help
```

sql-cache tool will display usage, options and command help, now you can create tables into sql server, running "sql-cache create" command :

```
C:\DistCacheSample\src\DistCacheSample>dotnet sql-cache create "Data Source=(localdb)\v11.0;Initial
Catalog=DistCache;Integrated Security=True;" dbo TestCache
info: Microsoft.Extensions.Caching.SqlConfig.Tools.Program[0]
      Table and index were created successfully.
```

The created table has the following schema:

Column Name	Data Type	Allow Nulls
Id	nvarchar(900)	<input type="checkbox"/>
Value	varbinary(MAX)	<input type="checkbox"/>
ExpiresAtTime	datetimeoffset(7)	<input type="checkbox"/>
SlidingExpirationInSecon...	bigint	<input checked="" type="checkbox"/>
AbsoluteExpiration	datetimeoffset(7)	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

Like all cache implementations, your app should get and set cache values using an instance of

`IDistributedCache`, not a `SqlServerCache`. The sample implements `SqlServerCache` in the `Production` environment (so it is configured in `ConfigureProductionServices`).

```
/// Use SQL Server Cache in Production
/// </summary>
/// <param name="services"></param>
public void ConfigureProductionServices(IServiceCollection services)
{
    services.AddDistributedSqlServerCache(options =>
    {
        options.ConnectionString = @"Data Source=(localdb)\v11.0;Initial Catalog=DistCache;Integrated
Security=True;";
        options.SchemaName = "dbo";
        options.TableName = "TestCache";
    });
}
```

NOTE

The `ConnectionString` (and optionally, `SchemaName` and `TableName`) should typically be stored outside of source control (such as UserSecrets), as they may contain credentials.

Recommendations

When deciding which implementation of `IDistributedCache` is right for your app, choose between Redis and SQL Server based on your existing infrastructure and environment, your performance requirements, and your team's experience. If your team is more comfortable working with Redis, it's an excellent choice. If your team prefers SQL Server, you can be confident in that implementation as well. Note that A traditional caching solution stores data in-memory which allows for fast retrieval of data. You should store commonly used data in a cache and store the entire data in a backend persistent store such as SQL Server or Azure Storage. Redis Cache is a caching solution which gives you high throughput and low latency as compared to SQL Cache.

Additional resources

- [Redis Cache on Azure](#)
- [SQL Database on Azure](#)
- [In-memory caching](#)
- [Detect changes with change tokens](#)
- [Response caching](#)
- [Response Caching Middleware](#)
- [Cache Tag Helper](#)
- [Distributed Cache Tag Helper](#)

Response caching in ASP.NET Core

11/29/2017 • 7 min to read • [Edit Online](#)

By [John Luo](#), [Rick Anderson](#), [Steve Smith](#), and [Luke Latham](#)

[View or download sample code](#) ([how to download](#))

Response caching reduces the number of requests a client or proxy makes to a web server. Response caching also reduces the amount of work the web server performs to generate a response. Response caching is controlled by headers that specify how you want client, proxy, and middleware to cache responses.

The web server can cache responses when you add [Response Caching Middleware](#).

HTTP-based response caching

The [HTTP 1.1 Caching specification](#) describes how Internet caches should behave. The primary HTTP header used for caching is [Cache-Control](#), which is used to specify cache *directives*. The directives control caching behavior as requests make their way from clients to servers and as responses make their way from servers back to clients. Requests and responses move through proxy servers, and proxy servers must also conform to the HTTP 1.1 Caching specification.

Common `Cache-Control` directives are shown in the following table.

DIRECTIVE	ACTION
public	A cache may store the response.
private	The response must not be stored by a shared cache. A private cache may store and reuse the response.
max-age	The client won't accept a response whose age is greater than the specified number of seconds. Examples: <code>max-age=60</code> (60 seconds), <code>max-age=2592000</code> (1 month)
no-cache	On requests: A cache must not use a stored response to satisfy the request. Note: The origin server re-generates the response for the client, and the middleware updates the stored response in its cache. On responses: The response must not be used for a subsequent request without validation on the origin server.
no-store	On requests: A cache must not store the request. On responses: A cache must not store any part of the response.

Other cache headers that play a role in caching are shown in the following table.

HEADER	FUNCTION
--------	----------

HEADER	FUNCTION
Age	An estimate of the amount of time in seconds since the response was generated or successfully validated at the origin server.
Expires	The date/time after which the response is considered stale.
Pragma	Exists for backwards compatibility with HTTP/1.0 caches for setting <code>no-cache</code> behavior. If the <code>Cache-Control</code> header is present, the <code>Pragma</code> header is ignored.
Vary	Specifies that a cached response must not be sent unless all of the <code>Vary</code> header fields match in both the cached response's original request and the new request.

HTTP-based caching respects request Cache-Control directives

The [HTTP 1.1 Caching specification for the Cache-Control header](#) requires a cache to honor a valid `Cache-Control` header sent by the client. A client can make requests with a `no-cache` header value and force the server to generate a new response for every request.

Always honoring client `Cache-Control` request headers makes sense if you consider the goal of HTTP caching. Under the official specification, caching is meant to reduce the latency and network overhead of satisfying requests across a network of clients, proxies, and servers. It isn't necessarily a way to control the load on an origin server.

There's no current developer control over this caching behavior when using the [Response Caching Middleware](#) because the middleware adheres to the official caching specification. [Future enhancements to the middleware](#) will permit configuring the middleware to ignore a request's `Cache-Control` header when deciding to serve a cached response. This will offer you an opportunity to better control the load on your server when you use the middleware.

Other caching technology in ASP.NET Core

In-memory caching

In-memory caching uses server memory to store cached data. This type of caching is suitable for a single server or multiple servers using *sticky sessions*. Sticky sessions means that the requests from a client are always routed to the same server for processing.

For more information, see [Introduction to in-memory caching in ASP.NET Core](#).

Distributed Cache

Use a distributed cache to store data in memory when the app is hosted in a cloud or server farm. The cache is shared across the servers that process requests. A client can submit a request that is handled by any server in the group and cached data for the client is available. ASP.NET Core offers SQL Server and Redis distributed caches.

For more information, see [Working with a distributed cache](#).

Cache Tag Helper

You can cache the content from an MVC view or Razor Page with the Cache Tag Helper. The Cache Tag Helper uses in-memory caching to store data.

For more information, see [Cache Tag Helper in ASP.NET Core MVC](#).

Distributed Cache Tag Helper

You can cache the content from an MVC view or Razor Page in distributed cloud or web farm scenarios with the Distributed Cache Tag Helper. The Distributed Cache Tag Helper uses SQL Server or Redis to store data.

For more information, see [Distributed Cache Tag Helper](#).

ResponseCache attribute

The `ResponseCacheAttribute` specifies the parameters necessary for setting appropriate headers in response caching. See [ResponseCacheAttribute](#) for a description of the parameters.

WARNING

Disable caching for content that contains information for authenticated clients. Caching should only be enabled for content that doesn't change based on a user's identity or whether a user is logged in.

`VaryByQueryKeys string[]` (requires ASP.NET Core 1.1 and higher): When set, the Response Caching Middleware varies the stored response by the values of the given list of query keys. The Response Caching Middleware must be enabled to set the `VaryByQueryKeys` property; otherwise, a runtime exception is thrown. There's no corresponding HTTP header for the `VaryByQueryKeys` property. This property is an HTTP feature handled by the Response Caching Middleware. For the middleware to serve a cached response, the query string and query string value must match a previous request. For example, consider the sequence of requests and results shown in the following table.

REQUEST	RESULT
<code>http://example.com?key1=value1</code>	Returned from server
<code>http://example.com?key1=value1</code>	Returned from middleware
<code>http://example.com?key1=value2</code>	Returned from server

The first request is returned by the server and cached in middleware. The second request is returned by middleware because the query string matches the previous request. The third request is not in the middleware cache because the query string value doesn't match a previous request.

The `ResponseCacheAttribute` is used to configure and create (via `IFilterFactory`) a `ResponseCacheFilter`. The `ResponseCacheFilter` performs the work of updating the appropriate HTTP headers and features of the response. The filter:

- Removes any existing headers for `Vary`, `Cache-Control`, and `Pragma`.
- Writes out the appropriate headers based on the properties set in the `ResponseCacheAttribute`.
- Updates the response caching HTTP feature if `VaryByQueryKeys` is set.

Vary

This header is only written when the `VaryByHeader` property is set. It's set to the `Vary` property's value. The following sample uses the `VaryByHeader` property:

```
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public IActionResult About2()
{
```

You can view the response headers with your browser's network tools. The following image shows the Edge F12

output on the **Network** tab when the `About2` action method is refreshed:

Headers	Body	Parameters	Cookies	Timings
Request URL: <code>http://localhost/Home/About2</code>				
Request Method: <code>GET</code>				
Status Code: <code>200 / OK</code>				
⌵ Request Headers				
Accept: <code>text/html, application/xhtml+xml, image/jxr, */*</code>				
Accept-Encoding: <code>gzip, deflate</code>				
Accept-Language: <code>en-US, en; q=0.8, zh-Hans-CN; q=0.5, zh-Hans; q=0.3</code>				
Connection: <code>Keep-Alive</code>				
Host: <code>localhost</code>				
User-Agent: <code>Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36...</code>				
⌵ Response Headers				
Cache-Control: <code>public, max-age=30</code>				
Content-Type: <code>text/html; charset=utf-8</code>				
Date: <code>Fri, 18 Nov 2016 21:45:14 GMT</code>				
Server: <code>Kestrel</code>				
Transfer-Encoding: <code>chunked</code>				
Vary: <code>User-Agent</code>				

NoStore and Location.None

`NoStore` overrides most of the other properties. When this property is set to `true`, the `Cache-Control` header is set to `no-store`. If `Location` is set to `None`:

- `Cache-Control` is set to `no-store, no-cache`.
- `Pragma` is set to `no-cache`.

If `NoStore` is `false` and `Location` is `None`, `Cache-Control` and `Pragma` are set to `no-cache`.

You typically set `NoStore` to `true` on error pages. For example:

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View();
}
```

This results in the following headers:

```
Cache-Control: no-store, no-cache
Pragma: no-cache
```

Location and Duration

To enable caching, `Duration` must be set to a positive value and `Location` must be either `Any` (the default) or `Client`. In this case, the `Cache-Control` header is set to the location value followed by the `max-age` of the response.

NOTE

`Location`'s options of `Any` and `Client` translate into `Cache-Control` header values of `public` and `private`, respectively. As noted previously, setting `Location` to `None` sets both `Cache-Control` and `Pragma` headers to `no-cache`.

Below is an example showing the headers produced by setting `Duration` and leaving the default `Location` value:

```
[ResponseCache(Duration = 60)]
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    return View();
}
```

This produces the following header:

```
Cache-Control: public,max-age=60
```

Cache profiles

Instead of duplicating `ResponseCache` settings on many controller action attributes, cache profiles can be configured as options when setting up MVC in the `ConfigureServices` method in `Startup`. Values found in a referenced cache profile are used as the defaults by the `ResponseCache` attribute and are overridden by any properties specified on the attribute.

Setting up a cache profile:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Default",
            new CacheProfile()
            {
                Duration = 60
            });
        options.CacheProfiles.Add("Never",
            new CacheProfile()
            {
                Location = ResponseCacheLocation.None,
                NoStore = true
            });
    });
}
```

Referencing a cache profile:

```
[ResponseCache(Duration = 30)]
public class HomeController : Controller
{
    [ResponseCache(CacheProfileName = "Default")]
    public IActionResult Index()
    {
        return View();
    }
}
```

The `ResponseCache` attribute can be applied both to actions (methods) and controllers (classes). Method-level attributes override the settings specified in class-level attributes.

In the above example, a class-level attribute specifies a duration of 30 seconds, while a method-level attribute references a cache profile with a duration set to 60 seconds.

The resulting header:

```
Cache-Control: public,max-age=60
```

Additional resources

- [Caching in HTTP from the specification](#)
- [Cache-Control](#)
- [In-memory caching](#)
- [Working with a distributed cache](#)
- [Detect changes with change tokens](#)
- [Response Caching Middleware](#)
- [Cache Tag Helper](#)
- [Distributed Cache Tag Helper](#)

Response Caching Middleware in ASP.NET Core

11/29/2017 • 6 min to read • [Edit Online](#)

By [Luke Latham](#) and [John Luo](#)

[View or download sample code \(how to download\)](#)

This document provides details on how to configure the Response Caching Middleware in ASP.NET Core apps. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the `ResponseCache` attribute, see [Response Caching](#).

Package

To include the middleware in a project, add a reference to the `Microsoft.AspNetCore.ResponseCaching` package or use the `Microsoft.AspNetCore.All` package.

Configuration

In `ConfigureServices`, add the middleware to the service collection.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddResponseCaching();
    })
    .Configure(app =>
    {
        app.UseResponseCaching();

        app.Run(async (context) =>
        {
            context.Response.GetTypedHeaders().CacheControl = new CacheControlHeaderValue()
            {
                Public = true,
                MaxAge = TimeSpan.FromSeconds(10)
            };
            context.Response.Headers[HeaderNames.Vary] = new string[] { "Accept-Encoding" };

            await context.Response.WriteAsync($"Hello World! {DateTime.UtcNow}");
        });
    })
    .Build();
```

Configure the app to use the middleware with the `UseResponseCaching` extension method, which adds the middleware to the request processing pipeline. The sample app adds a `Cache-Control` header to the response that caches cacheable responses for up to 10 seconds. The sample sends a `Vary` header to configure the middleware to serve a cached response only if the `Accept-Encoding` header of subsequent requests matches that of the original request.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

WebHost.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddResponseCaching();
    })
    .Configure(app =>
    {
        app.UseResponseCaching();

        app.Run(async (context) =>
        {
            context.Response.GetTypedHeaders().CacheControl = new CacheControlHeaderValue()
            {
                Public = true,
                MaxAge = TimeSpan.FromSeconds(10)
            };
            context.Response.Headers[HeaderNames.Vary] = new string[] { "Accept-Encoding" };

            await context.Response.WriteAsync($"Hello World! {DateTime.UtcNow}");
        });
    })
    .Build();

```

The Response Caching Middleware only caches 200 (OK) server responses. Any other responses, including [error pages](#), are ignored by the middleware.

WARNING

Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See [Conditions for caching](#) for details on how the middleware determines if a response is cacheable.

Options

The middleware offers three options for controlling response caching.

OPTION	DEFAULT VALUE
UseCaseSensitivePaths	Determines if responses are cached on case-sensitive paths. The default value is <code>false</code> .
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is <code>64 * 1024 * 1024</code> (64 MB).
SizeLimit	The size limit for the response cache middleware in bytes. The default value is <code>100 * 1024 * 1024</code> (100 MB).

The following example configures the middleware to cache responses smaller than or equal to 1,024 bytes using case-sensitive paths, storing the responses to `/page1` and `/Page1` separately.

```

services.AddResponseCaching(options =>
{
    options.UseCaseSensitivePaths = true;
    options.MaximumBodySize = 1024;
});

```

VaryByQueryKeys

When using MVC, the `ResponseCache` attribute specifies the parameters necessary for setting appropriate headers for response caching. The only parameter of the `ResponseCache` attribute that strictly requires the middleware is `VaryByQueryKeys`, which doesn't correspond to an actual HTTP header. For more information, see [ResponseCache Attribute](#).

When not using MVC, you can vary response caching with the `VaryByQueryKeys` feature. Use the `ResponseCachingFeature` directly from the `IFeatureCollection` of the `HttpContext`:

```
var responseCachingFeature = context.HttpContext.Features.Get<IResponseCachingFeature>();
if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

HTTP headers used by Response Caching Middleware

Response caching by the middleware is configured via HTTP headers. The relevant headers are listed below with notes on how they affect caching.

HEADER	DETAILS
Authorization	The response isn't cached if the header exists.
Cache-Control	<p>The middleware only considers caching responses marked with the <code>public</code> cache directive. You can control caching with the following parameters:</p> <ul style="list-style-type: none">• <code>max-age</code>• <code>max-stale</code>[†]• <code>min-fresh</code>• <code>must-revalidate</code>• <code>no-cache</code>• <code>no-store</code>• <code>only-if-cached</code>• <code>private</code>• <code>public</code>• <code>s-maxage</code>• <code>proxy-revalidate</code>[‡] <p>[†]If no limit is specified to <code>max-stale</code>, the middleware takes no action.</p> <p>[‡]<code>proxy-revalidate</code> has the same effect as <code>must-revalidate</code>.</p> <p>For more information, see RFC 7231: Request Cache-Control Directives.</p>
Pragma	A <code>Pragma: no-cache</code> header in the request produces the same effect as <code>Cache-Control: no-cache</code> . This header is overridden by the relevant directives in the <code>Cache-Control</code> header, if present. Considered for backward compatibility with HTTP/1.0.
Set-Cookie	The response isn't cached if the header exists.

HEADER	DETAILS
Vary	The <code>Vary</code> header is used to vary the cached response by another header. For example, you can cache responses by encoding by including the <code>Vary: Accept-Encoding</code> header, which caches responses for requests with headers <code>Accept-Encoding: gzip</code> and <code>Accept-Encoding: text/plain</code> separately. A response with a header value of <code>*</code> is never stored.
Expires	A response deemed stale by this header isn't stored or retrieved unless overridden by other <code>Cache-Control</code> headers.
If-None-Match	The full response is served from cache if the value isn't <code>*</code> and the <code>ETag</code> of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.
If-Modified-Since	If the <code>If-None-Match</code> header isn't present, a full response is served from cache if the cached response date is newer than the value provided. Otherwise, a 304 (Not Modified) response is served.
Date	When serving from cache, the <code>Date</code> header is set by the middleware if it wasn't provided on the original response.
Content-Length	When serving from cache, the <code>Content-Length</code> header is set by the middleware if it wasn't provided on the original response.
Age	The <code>Age</code> header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

Caching respects request Cache-Control directives

The middleware respects the rules of the [HTTP 1.1 Caching specification](#). The rules require a cache to honor a valid `Cache-Control` header sent by the client. Under the specification, a client can make requests with a `no-cache` header value and force a server to generate a new response for every request. Currently, there's no developer control over this caching behavior when using the middleware because the middleware adheres to the official caching specification.

[Future enhancements to the middleware](#) will permit configuring the middleware for caching scenarios where the request `Cache-Control` header should be ignored when deciding to serve a cached response. If you seek more control over caching behavior, explore other caching features of ASP.NET Core. See the following topics:

- [In-memory caching](#)
- [Working with a distributed cache](#)
- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Distributed Cache Tag Helper](#)

Troubleshooting

If caching behavior isn't as you expect, confirm that responses are cacheable and capable of being served from

the cache by examining the request's incoming headers and the response's outgoing headers. Enabling [logging](#) can help when debugging. The middleware logs caching behavior and when a response is retrieved from cache.

When testing and troubleshooting caching behavior, a browser may set request headers that affect caching in undesirable ways. For example, a browser may set the `Cache-Control` header to `no-cache` when you refresh the page. The following tools can explicitly set request headers, and are preferred for testing caching:

- [Fiddler](#)
- [Firebug](#)
- [Postman](#)

Conditions for caching

- The request must result in a 200 (OK) response from the server.
- The request method must be GET or HEAD.
- Terminal middleware, such as Static File Middleware, must not process the response prior to the Response Caching Middleware.
- The `Authorization` header must not be present.
- `Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
- The `Pragma: no-cache` header/value must not be present if the `Cache-Control` header isn't present, as the `Cache-Control` header overrides the `Pragma` header when present.
- The `Set-Cookie` header must not be present.
- `Vary` header parameters must be valid and not equal to `*`.
- The `Content-Length` header value (if set) must match the size of the response body.
- The `IHttpSendFileFeature` isn't used.
- The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
- Response buffering is successful, and the size of the response is smaller than the configured or default `SizeLimit`.
- The response must be cacheable according to the [RFC 7234](#) specifications. For example, the `no-store` directive must not exist in request or response header fields. See *Section 3: Storing Responses in Caches* of [RFC 7234](#) for details.

NOTE

The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks sets the `Cache-Control` and `Pragma` headers to `no-cache` so that responses aren't cached.

Additional resources

- [Application Startup](#)
- [Middleware](#)
- [In-memory caching](#)
- [Working with a distributed cache](#)
- [Detect changes with change tokens](#)
- [Response caching](#)
- [Cache Tag Helper](#)
- [Distributed Cache Tag Helper](#)

Response Compression Middleware for ASP.NET Core

1/10/2018 • 9 min to read • [Edit Online](#)

By [Luke Latham](#)

[View or download sample code \(how to download\)](#)

Network bandwidth is a limited resource. Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses.

When to use Response Compression Middleware

Use server-based response compression technologies in IIS, Apache, or Nginx. The performance of the middleware probably won't match that of the server modules. [HTTP.sys server](#) and [Kestrel](#) don't currently offer built-in compression support.

Use Response Compression Middleware when you're:

- Unable to use the following server-based compression technologies:
 - [IIS Dynamic Compression module](#)
 - [Apache mod_deflate module](#)
 - [NGINX Compression and Decompression](#)
- Hosting directly on:
 - [HTTP.sys server](#) (formerly called [WebListener](#))
 - [Kestrel](#)

Response compression

Usually, any response not natively compressed can benefit from response compression. Responses not natively compressed typically include: CSS, JavaScript, HTML, XML, and JSON. You shouldn't compress natively compressed assets, such as PNG files. If you attempt to further compress a natively compressed response, any small additional reduction in size and transmission time will likely be overshadowed by the time it took to process the compression. Don't compress files smaller than about 150-1000 bytes (depending on the file's content and the efficiency of compression). The overhead of compressing small files may produce a compressed file larger than the uncompressed file.

When a client can process compressed content, the client must inform the server of its capabilities by sending the `Accept-Encoding` header with the request. When a server sends compressed content, it must include information in the `Content-Encoding` header on how the compressed response is encoded. Content encoding designations supported by the middleware are shown in the following table.

<code>ACCEPT-ENCODING</code> HEADER VALUES	MIDDLEWARE SUPPORTED	DESCRIPTION
<code>br</code>	No	Brotli Compressed Data Format
<code>compress</code>	No	UNIX "compress" data format

ACCEPT-ENCODING HEADER VALUES	MIDDLEWARE SUPPORTED	DESCRIPTION
deflate	No	"deflate" compressed data inside the "zlib" data format
exi	No	W3C Efficient XML Interchange
gzip	Yes (default)	gzip file format
identity	Yes	"No encoding" identifier: The response must not be encoded.
pack200-gzip	No	Network Transfer Format for Java Archives
*	Yes	Any available content encoding not explicitly requested

For more information, see the [IANA Official Content Coding List](#).

The middleware allows you to add additional compression providers for custom `Accept-Encoding` header values. For more information, see [Custom Providers](#) below.

The middleware is capable of reacting to quality value (qvalue, `q`) weighting when sent by the client to prioritize compression schemes. For more information, see [RFC 7231: Accept-Encoding](#).

Compression algorithms are subject to a tradeoff between compression speed and the effectiveness of the compression. *Effectiveness* in this context refers to the size of the output after compression. The smallest size is achieved by the most *optimal* compression.

The headers involved in requesting, sending, caching, and receiving compressed content are described in the table below.

HEADER	ROLE
Accept-Encoding	Sent from the client to the server to indicate the content encoding schemes acceptable to the client.
Content-Encoding	Sent from the server to the client to indicate the encoding of the content in the payload.
Content-Length	When compression occurs, the <code>Content-Length</code> header is removed, since the body content changes when the response is compressed.
Content-MD5	When compression occurs, the <code>Content-MD5</code> header is removed, since the body content has changed and the hash is no longer valid.
Content-Type	Specifies the MIME type of the content. Every response should specify its <code>Content-Type</code> . The middleware checks this value to determine if the response should be compressed. The middleware specifies a set of default MIME types that it can encode, but you can replace or add MIME types.

HEADER	ROLE
Vary	When sent by the server with a value of <code>Accept-Encoding</code> to clients and proxies, the <code>Vary</code> header indicates to the client or proxy that it should cache (vary) responses based on the value of the <code>Accept-Encoding</code> header of the request. The result of returning content with the <code>Vary: Accept-Encoding</code> header is that both compressed and uncompressed responses are cached separately.

You can explore the features of the Response Compression Middleware with the [sample app](#). The sample illustrates:

- The compression of app responses using gzip and custom compression providers.
- How to add a MIME type to the default list of MIME types for compression.

Package

To include the middleware in your project, add a reference to the `Microsoft.AspNetCore.ResponseCompression` package or use the `Microsoft.AspNetCore.All` package. This feature is available for apps that target ASP.NET Core 1.1 or later.

Configuration

The following code shows how to enable the Response Compression Middleware with the with the default gzip compression and for default MIME types.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

WebHost.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddResponseCompression();
    })
    .Configure(app =>
    {
        app.UseResponseCompression();

        app.Run(async context =>
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync(LoremIpsum.Text);
        });
    })
    .Build();

```

NOTE

Use a tool like [Fiddler](#), [Firebug](#), or [Postman](#) to set the `Accept-Encoding` request header and study the response headers, size, and body.

Submit a request to the sample app without the `Accept-Encoding` header and observe that the response is uncompressed. The `Content-Encoding` and `Vary` headers aren't present on the response.

Headers	TextView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML
---------	----------	----------	---------	------	---------	-----	------	-----

Request Headers
GET / HTTP/1.1

Client
User-Agent: Fiddler

Transport
Host: localhost:5000

Get SyntaxView	Transformer	Headers	TextView	ImageView	HexView	WebView	Auth	Caching	Cookies	Raw
----------------	-------------	---------	----------	-----------	---------	---------	------	---------	---------	-----

```

HTTP/1.1 200 OK
Date: Wed, 11 Jan 2017 18:30:11 GMT
Content-Type: text/plain
Server: Kestrel
Content-Length: 2032
  
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed non risus. Suspendisse lectus tortor, digna elementum ultrices diam. Maecenas ligula massa, varius a, semper congue, euismod non, mi. Proin porttitor fermentum diam nisi sit amet erat. Duis semper. Duis arcu massa, scelerisque vitae, consequat in, preti pharetra tempor. Cras vestibulum bibendum augue. Praesent egestas leo in pede. Praesent blandit odio eu ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aliquam nibh. Mauris ac ma non diam sodales hendrerit. Ut velit mauris, egestas sed, gravida nec, ornare ut, mi. Aenean ut orci vel non tempus aliquam, nunc turpis ullamcorper nibh, in tempus sapien eros vitae ligula. Pellentesque rhonc diam. Integer quis metus vitae elit lobortis egestas. Lorem ipsum dolor sit amet, consectetur adipiscing sapien. Integer tortor tellus, aliquam faucibus, convallis id, congue eu, quam. Mauris ullamcorper felis purus iaculis lectus, et tristique ligula justo vitae magna. Aliquam convallis sollicitudin purus. Praes nisi, ac euismod nibh nisi eu lectus. Fusce vulputate sem at sapien. Vivamus leo. Aliquam euismod libero suscipit nulla in justo. Suspendisse cursus rutrum augue. Nulla tincidunt tincidunt mi. Curabitur iaculi ultricies lacus lorem varius purus. Curabitur eu amet.

Submit a request to the sample app with the `Accept-Encoding: gzip` header and observe that the response is compressed. The `Content-Encoding` and `Vary` headers are present on the response.

Headers	TextView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML
---------	----------	----------	---------	------	---------	-----	------	-----

Request Headers
GET / HTTP/1.1

Client
`Accept-Encoding: gzip`
User-Agent: Fiddler

Transport
Host: localhost:5000

Response body is encoded. Click to decode.

Get SyntaxView	Transformer	Headers	TextView	ImageView	HexView	WebView	Auth	Caching	Cookies	Raw
----------------	-------------	---------	----------	-----------	---------	---------	------	---------	---------	-----

```

HTTP/1.1 200 OK
Date: Wed, 11 Jan 2017 18:24:08 GMT
Content-Type: text/plain
Server: Kestrel
Transfer-Encoding: chunked
Content-Encoding: gzip
Vary: Accept-Encoding
  
```

a
 00000000
 384
 00U.9
 00}000d02000000^00000G]K*000rUW0Y0n0G0 000000
 0-000
 XH00V000Z.\ycI\@0500Z00
 00\$000000"00000000 X0QY00m0m\0000vNL0Bk80D00Q0200je000V0000R000000
 0 0;0000*0u_0a0000000:0000{0
 [0<0000*%_0J00J0\$0~0r00u80F000a00F]0sXPY0(PG000/0h0000N00(Sgy7000i20n00500I0000|000*0#0z0~0009`00tC00
 0000; iGP*[000w[, [000000a0#00I0*fd0cj>~000 000Vn@0_0VX000
 00i+f000^00*M3n0
 00000x10i00a.00&F0 00>YG00300Pdy0[000&(0' 000vg0R0毛Y @000z000k0000*0lyI0000~t0yE0X00+v00I00V0i00S>00b0
 0000d0'np0q0vt0(010Nz/0000Y0000#
 004w00S0000G;00=0500800000, 0&0000e
 0Yrv?00yY0G0x00000_0F0j00000ET0r 00Y.00V0?^00sx01/\u60&000X7ST0000d00;00000000f0ut00[000s0m0eL^0000+0R0+qu0
 0000K0000i000q0rM0Zux}0000}Q0<L0^<0t0}0h0:7vG?0Q_of000u0~S0id0Q000z1hno 00
 0

Providers

GzipCompressionProvider

Use the `GzipCompressionProvider` to compress responses with gzip. This is the default compression provider if none are specified. You can set the compression level with the `GzipCompressionProviderOptions`.

The gzip compression provider defaults to the fastest compression level (`CompressionLevel.Fastest`), which might not produce the most efficient compression. If the most efficient compression is desired, you can configure the middleware for optimal compression.

COMPRESSION LEVEL	DESCRIPTION
<code>CompressionLevel.Fastest</code>	Compression should complete as quickly as possible, even if the resulting output is not optimally compressed.
<code>CompressionLevel.NoCompression</code>	No compression should be performed.
<code>CompressionLevel.Optimal</code>	Responses should be optimally compressed, even if the compression takes more time to complete.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});

services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});
```

MIME types

The middleware specifies a default set of MIME types for compression:

- `text/plain`
- `text/css`
- `application/javascript`
- `text/html`
- `application/xml`
- `text/xml`
- `application/json`
- `text/json`

You can replace or append MIME types with the Response Compression Middleware options. Note that wildcard MIME types, such as `text/*` aren't supported. The sample app adds a MIME type for `image/svg+xml` and compresses and serves the ASP.NET Core banner image (*banner.svg*).

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});

services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});

```

Custom providers

You can create custom compression implementations with `ICompressionProvider`. The `EncodingName` represents the content encoding that this `ICompressionProvider` produces. The middleware uses this information to choose the provider based on the list specified in the `Accept-Encoding` header of the request.

Using the sample app, the client submits a request with the `Accept-Encoding: mycustomcompression` header. The middleware uses the custom compression implementation and returns the response with a `Content-Encoding: mycustomcompression` header. The client must be able to decompress the custom encoding in order for a custom compression implementation to work.

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});

services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});

```

```

public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}

```

Submit a request to the sample app with the `Accept-Encoding: mycustomcompression` header and observe the response headers. The `Vary` and `Content-Encoding` headers are present on the response. The response body (not shown) isn't compressed by the sample. There isn't a compression implementation in the `CustomCompressionProvider` class of the sample. However, the sample shows where you would implement such a compression algorithm.

Request Headers
GET / HTTP/1.1

Client
Accept-Encoding: mycustomcompression
User-Agent: Fiddler

Transport
Host: localhost:5000

Get SyntaxView | Transformer | Headers

Response Headers
HTTP/1.1 200 OK

Cache
Date: Thu, 19 Jan 2017 22:06:50 GMT
Vary: Accept-Encoding

Entity
Content-Encoding: mycustomcompression
Content-Type: text/plain

Miscellaneous
Server: Kestrel

Transport
Transfer-Encoding: chunked

Compression with secure protocol

Compressed responses over secure connections can be controlled with the `EnableForHttps` option, which is disabled by default. Using compression with dynamically generated pages can lead to security problems such as the [CRIME](#) and [BREACH](#) attacks.

Adding the Vary header

When compressing responses based on the `Accept-Encoding` header, there are potentially multiple compressed versions of the response and an uncompressed version. In order to instruct client and proxy caches that multiple versions exist and should be stored, the `Vary` header is added with an `Accept-Encoding` value. In ASP.NET Core 1.x, adding the `Vary` header to the response is accomplished manually. In ASP.NET Core 2.x, the middleware adds the `Vary` header automatically when the response is compressed.

ASP.NET Core 1.x only

```
// ONLY REQUIRED FOR ASP.NET CORE 1.x APPS
private void ManageVaryHeader(HttpContext context)
{
    // If the Accept-Encoding header is present, add the Vary header
    var accept = context.Request.Headers[HeaderNames.AcceptEncoding];
    if (!StringValues.IsNullOrEmpty(accept))
    {
        context.Response.Headers.Append(HeaderNames.Vary, HeaderNames.AcceptEncoding);
    }
}
```

Middleware issue when behind an Nginx reverse proxy

When a request is proxied by Nginx, the `Accept-Encoding` header is removed. This prevents the middleware from compressing the response. For more information, see [NGINX: Compression and Decompression](#). This issue is tracked by [Figure out pass-through compression for nginx \(BasicMiddleware #123\)](#).

Working with IIS dynamic compression

If you have an active IIS Dynamic Compression Module configured at the server level that you would like to disable for an app, you can do so with an addition to your `web.config` file. For more information, see [Disabling IIS modules](#).

Troubleshooting

Use a tool like [Fiddler](#), [Firebug](#), or [Postman](#), which allow you to set the `Accept-Encoding` request header and study the response headers, size, and body. The Response Compression Middleware compresses responses that meet the following conditions:

- The `Accept-Encoding` header is present with a value of `gzip`, `*`, or custom encoding that matches a custom compression provider that you've established. The value must not be `identity` or have a quality value (qvalue, `q`) setting of 0 (zero).
- The MIME type (`Content-Type`) must be set and must match a MIME type configured on the `ResponseCompressionOptions`.
- The request must not include the `Content-Range` header.
- The request must use insecure protocol (`http`), unless secure protocol (`https`) is configured in the Response Compression Middleware options. *Note the danger [described above](#) when enabling secure content compression.*

Additional resources

- [Application Startup](#)
- [Middleware](#)
- [Mozilla Developer Network: Accept-Encoding](#)
- [RFC 7231 Section 3.1.2.1: Content Codings](#)
- [RFC 7230 Section 4.2.3: Gzip Coding](#)
- [GZIP file format specification version 4.3](#)

Migration to ASP.NET Core, including ASP.NET 4.x, ASP.NET Core 2

1/10/2018 • 1 min to read • [Edit Online](#)

ASP.NET to ASP.NET Core 1.x

- [Migrate from ASP.NET MVC to ASP.NET Core MVC](#)
- [Migrate configuration](#)
- [Migrate authentication and Identity](#)
- [Migrate from ASP.NET Web API](#)
- [Migrate HTTP modules to middleware](#)

ASP.NET to ASP.NET Core 2.0

- [ASP.NET to ASP.NET Core 2.0](#)

ASP.NET Core 1.x to 2.0

- [Migrate from ASP.NET Core 1.x to 2.0](#)
- [Migrate authentication and Identity](#)

Migrating From ASP.NET MVC to ASP.NET Core MVC

10/13/2017 • 7 min to read • [Edit Online](#)

By [Rick Anderson](#), [Daniel Roth](#), [Steve Smith](#), and [Scott Addie](#)

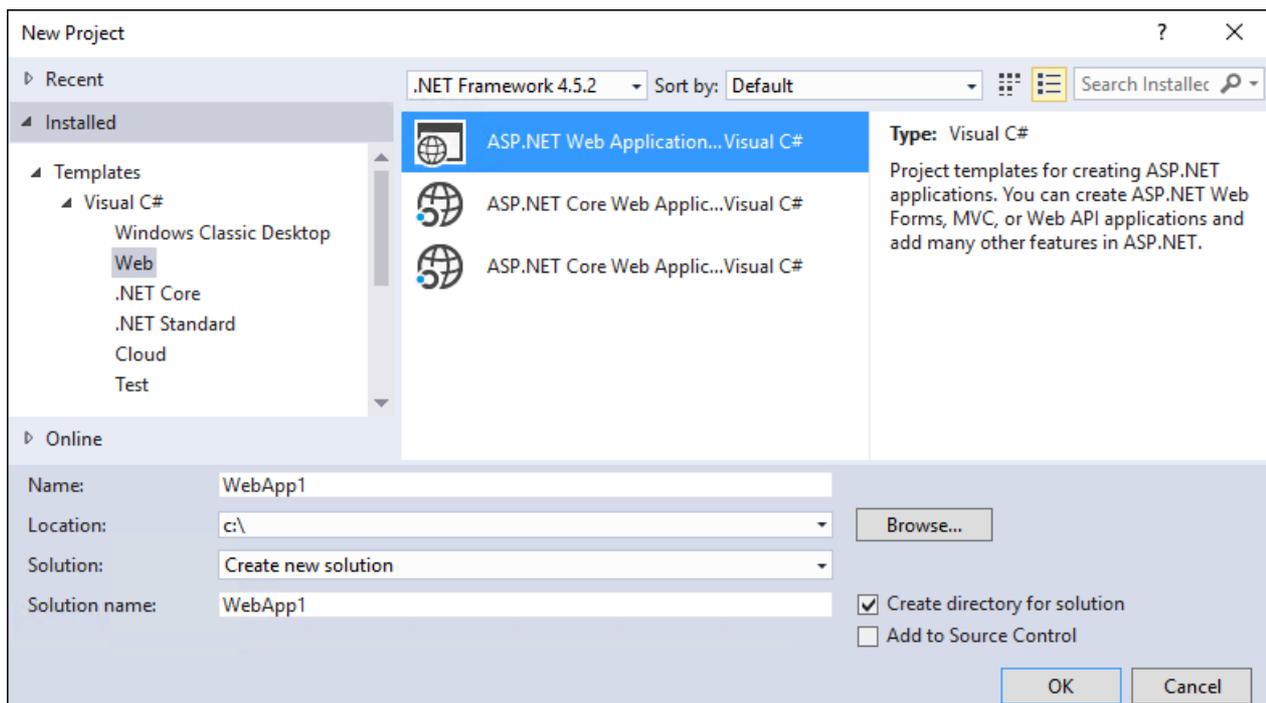
This article shows how to get started migrating an ASP.NET MVC project to [ASP.NET Core MVC](#). In the process, it highlights many of the things that have changed from ASP.NET MVC. Migrating from ASP.NET MVC is a multiple step process and this article covers the initial setup, basic controllers and views, static content, and client-side dependencies. Additional articles cover migrating configuration and identity code found in many ASP.NET MVC projects.

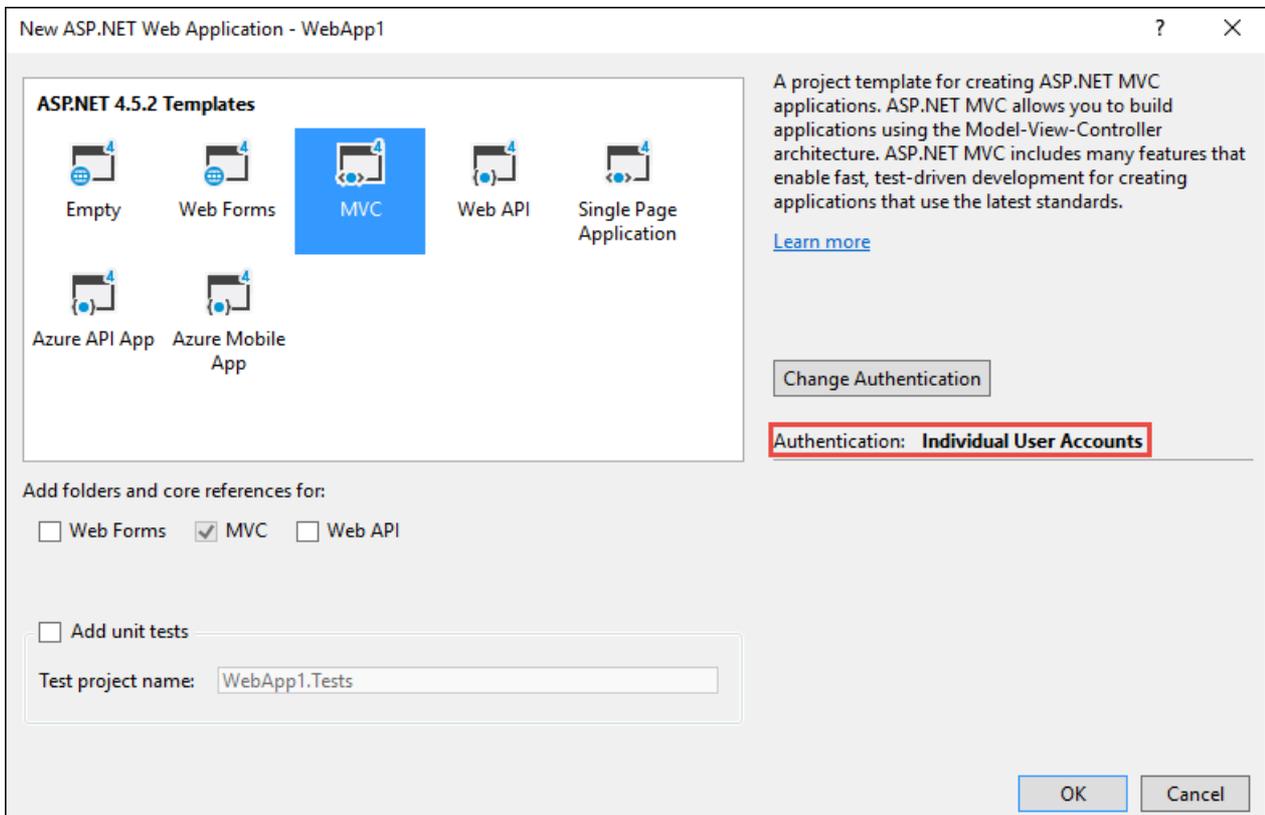
NOTE

The version numbers in the samples might not be current. You may need to update your projects accordingly.

Create the starter ASP.NET MVC project

To demonstrate the upgrade, we'll start by creating a ASP.NET MVC app. Create it with the name *WebApp1* so the namespace will match the ASP.NET Core project we create in the next step.

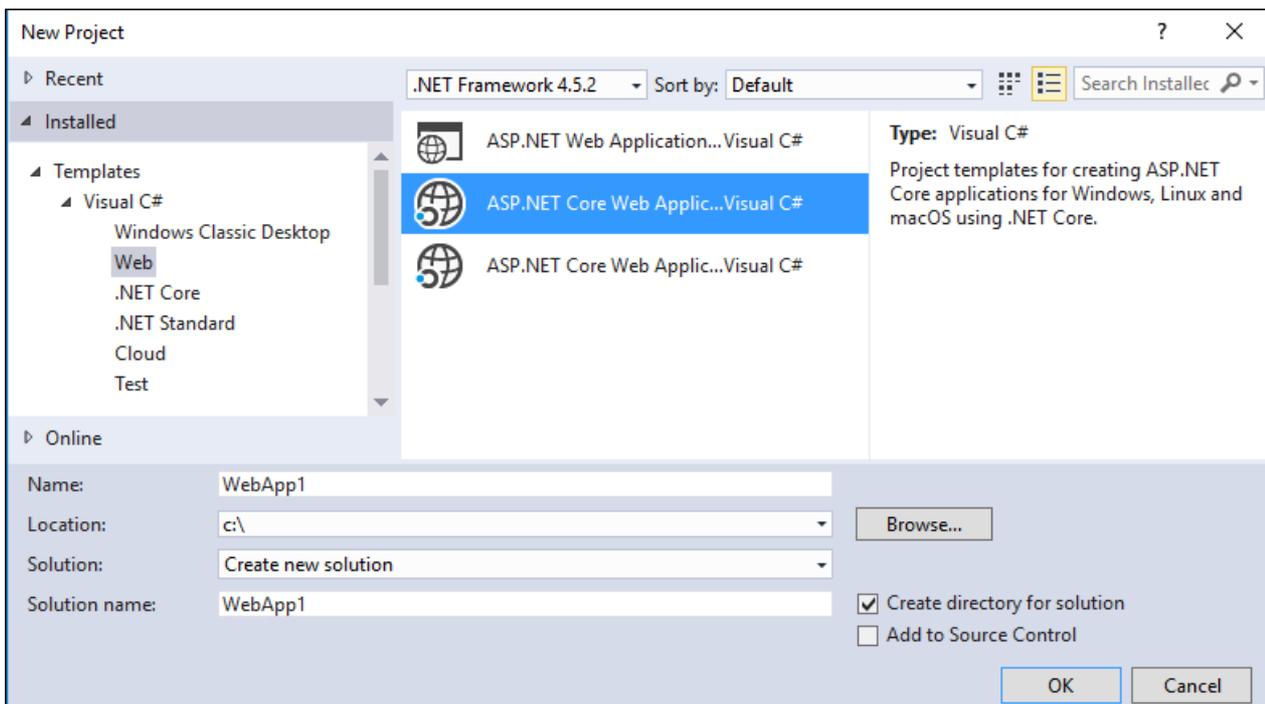


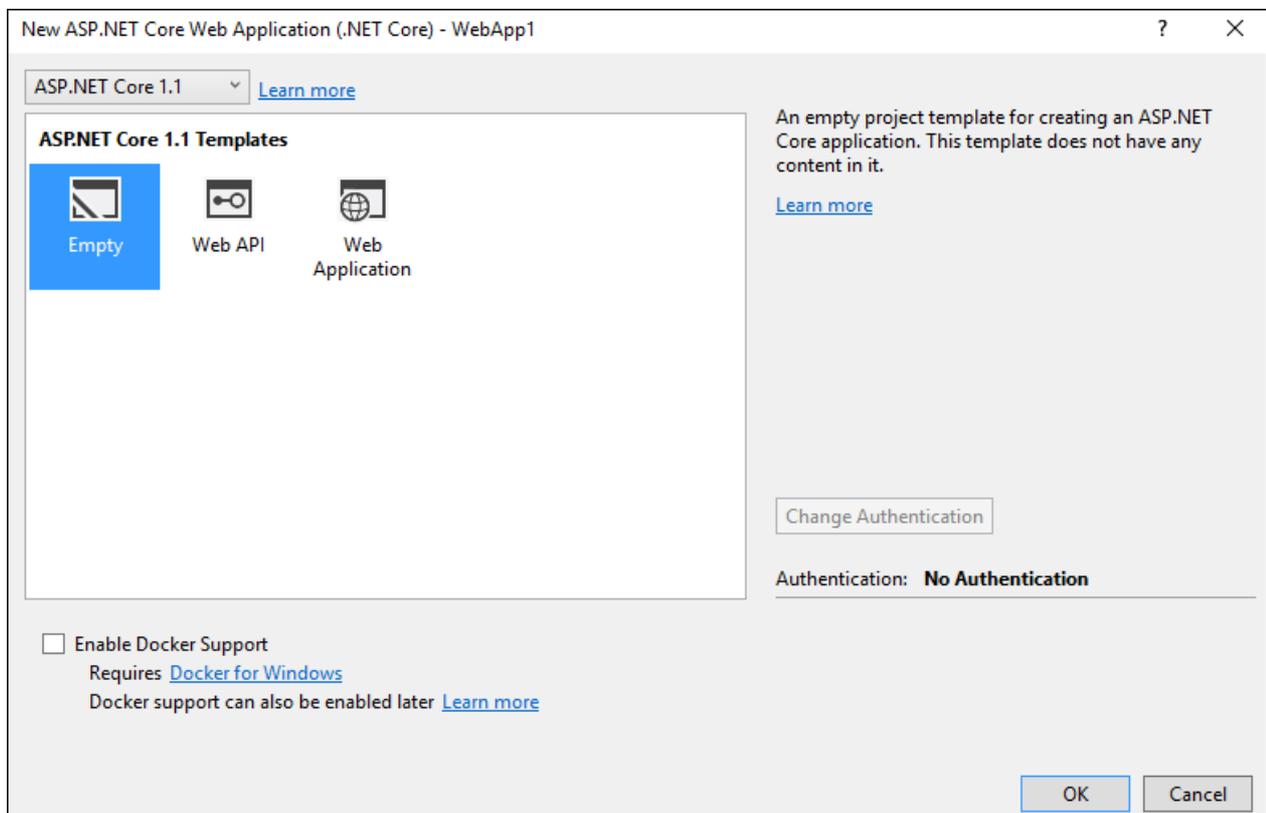


Optional: Change the name of the Solution from *WebApp1* to *Mvc5*. Visual Studio will display the new solution name (*Mvc5*), which will make it easier to tell this project from the next project.

Create the ASP.NET Core project

Create a new *empty* ASP.NET Core web app with the same name as the previous project (*WebApp1*) so the namespaces in the two projects match. Having the same namespace makes it easier to copy code between the two projects. You'll have to create this project in a different directory than the previous project to use the same name.





- *Optional:* Create a new ASP.NET Core app using the *Web Application* project template. Name the project *WebApp1*, and select an authentication option of **Individual User Accounts**. Rename this app to *FullAspNetCore*. Creating this project will save you time in the conversion. You can look at the template-generated code to see the end result or to copy code to the conversion project. It's also helpful when you get stuck on a conversion step to compare with the template-generated project.

Configure the site to use MVC

- Install the `Microsoft.AspNetCore.Mvc` and `Microsoft.AspNetCore.StaticFiles` NuGet packages.

`Microsoft.AspNetCore.Mvc` is the ASP.NET Core MVC framework. `Microsoft.AspNetCore.StaticFiles` is the static file handler. The ASP.NET runtime is modular, and you must explicitly opt in to serve static files (see [Working with Static Files](#)).

- Open the `.csproj` file (right-click the project in **Solution Explorer** and select **Edit WebApp1.csproj**) and add a `PrepareForPublish` target:

```
<Target Name="PrepublishScript" BeforeTargets="PrepareForPublish">
  <Exec Command="bower install" />
</Target>
```

The `PrepareForPublish` target is needed for acquiring client-side libraries via Bower. We'll talk about that later.

- Open the `Startup.cs` file and change the code to match the following:

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace WebApp1
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit
        https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
        loggerFactory)
        {
            loggerFactory.AddConsole();

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseStaticFiles();

            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}

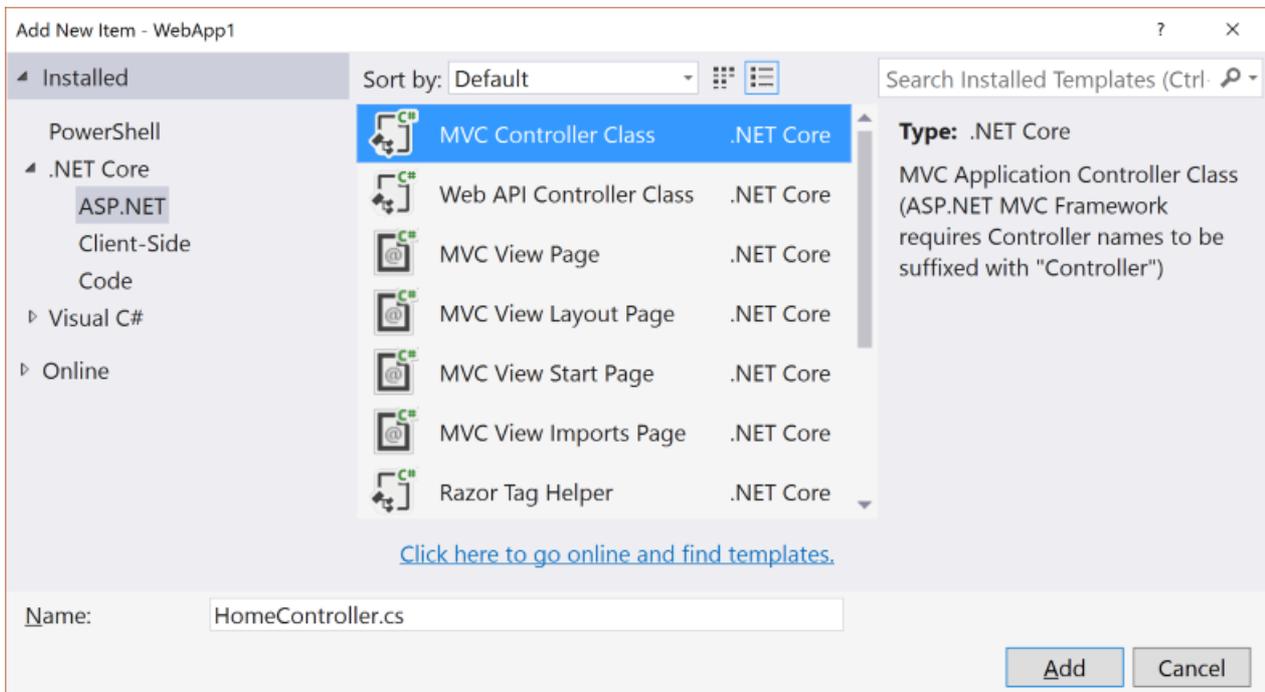
```

The `UseStaticFiles` extension method adds the static file handler. As mentioned previously, the ASP.NET runtime is modular, and you must explicitly opt in to serve static files. The `UseMvc` extension method adds routing. For more information, see [Application Startup](#) and [Routing](#).

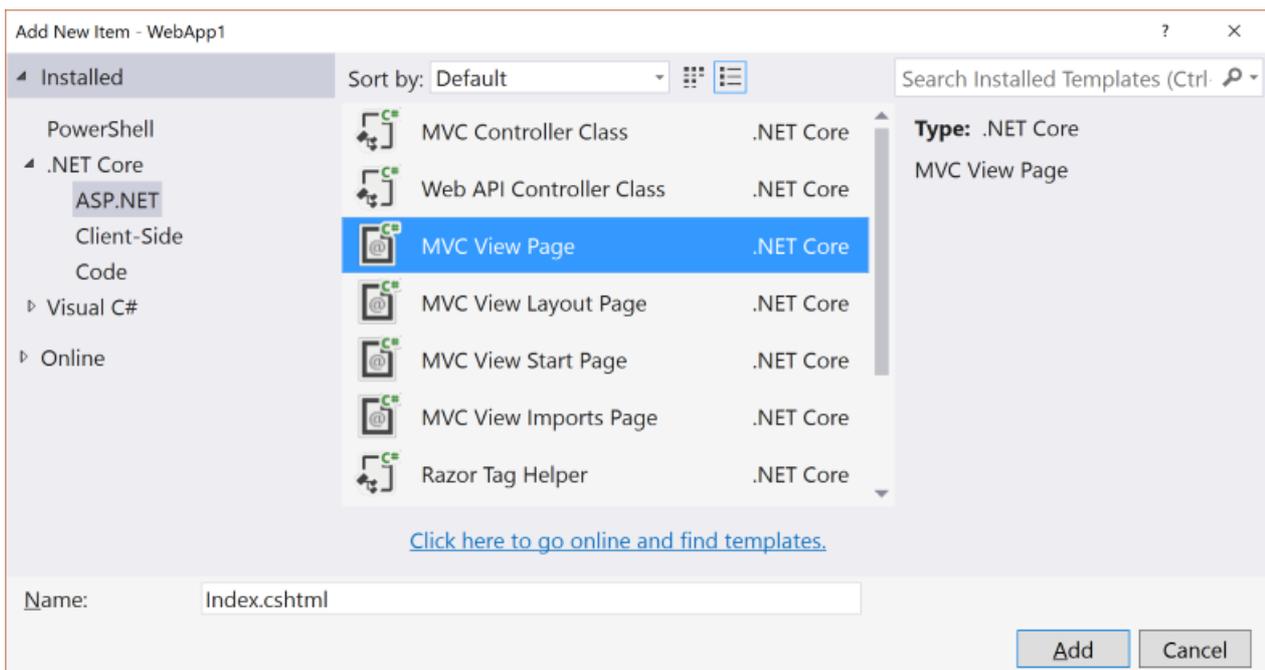
Add a controller and view

In this section, you'll add a minimal controller and view to serve as placeholders for the ASP.NET MVC controller and views you'll migrate in the next section.

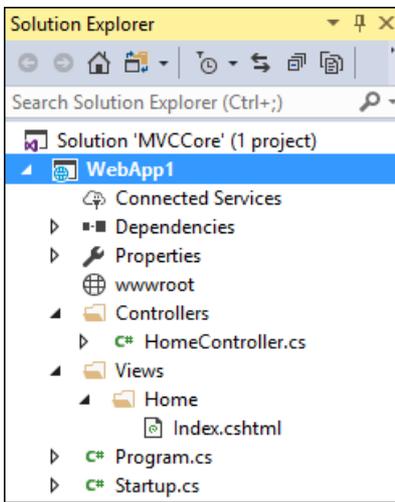
- Add a *Controllers* folder.
- Add an **MVC controller class** with the name *HomeController.cs* to the *Controllers* folder.



- Add a *Views* folder.
- Add a *Views/Home* folder.
- Add an *Index.cshtml* MVC view page to the *Views/Home* folder.



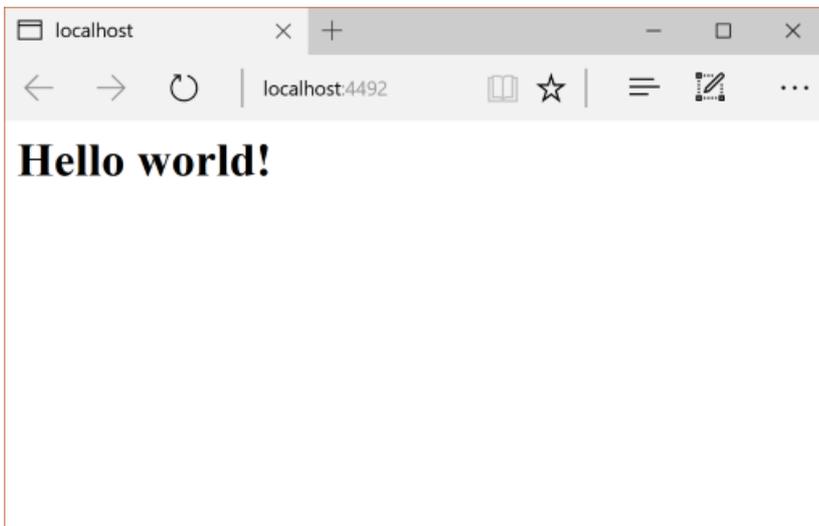
The project structure is shown below:



Replace the contents of the *Views/Home/Index.cshtml* file with the following:

```
<h1>Hello world!</h1>
```

Run the app.



See [Controllers](#) and [Views](#) for more information.

Now that we have a minimal working ASP.NET Core project, we can start migrating functionality from the ASP.NET MVC project. We will need to move the following:

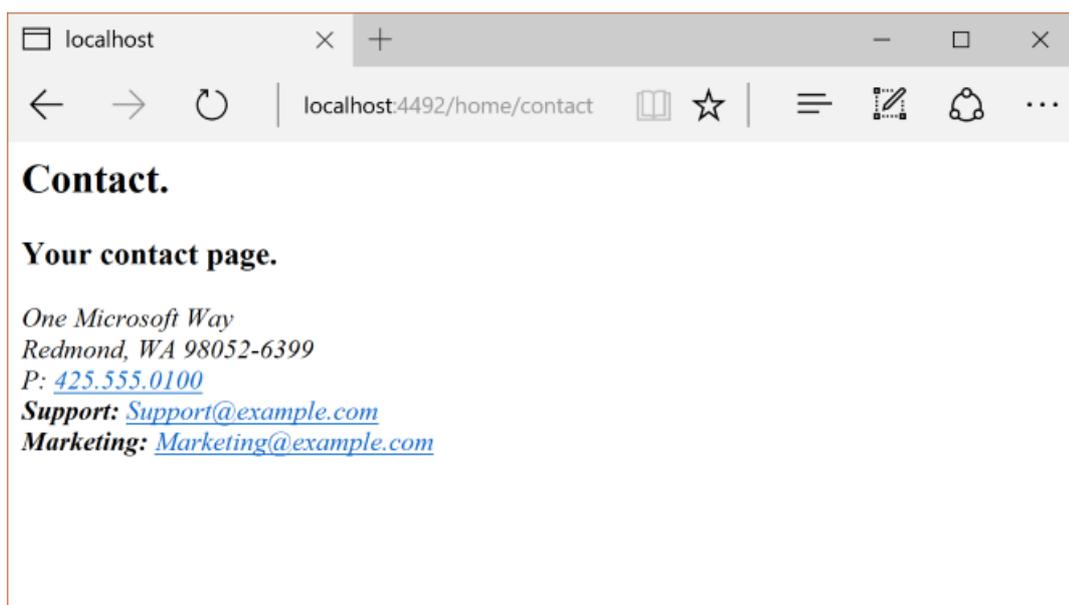
- client-side content (CSS, fonts, and scripts)
- controllers
- views
- models
- bundling
- filters
- Log in/out, identity (This will be done in the next tutorial.)

Controllers and views

- Copy each of the methods from the ASP.NET MVC `HomeController` to the new `HomeController`. Note that in

ASP.NET MVC, the built-in template's controller action method return type is [ActionResult](#); in ASP.NET Core MVC, the action methods return `IActionResult` instead. `ActionResult` implements `IActionResult`, so there is no need to change the return type of your action methods.

- Copy the *About.cshtml*, *Contact.cshtml*, and *Index.cshtml* Razor view files from the ASP.NET MVC project to the ASP.NET Core project.
- Run the ASP.NET Core app and test each method. We haven't migrated the layout file or styles yet, so the rendered views will only contain the content in the view files. You won't have the layout file generated links for the `About` and `Contact` views, so you'll have to invoke them from the browser (replace **4492** with the port number used in your project).
 - `http://localhost:4492/home/about`
 - `http://localhost:4492/home/contact`



Note the lack of styling and menu items. We'll fix that in the next section.

Static content

In previous versions of ASP.NET MVC, static content was hosted from the root of the web project and was intermixed with server-side files. In ASP.NET Core, static content is hosted in the *wwwroot* folder. You'll want to copy the static content from your old ASP.NET MVC app to the *wwwroot* folder in your ASP.NET Core project. In this sample conversion:

- Copy the *favicon.ico* file from the old MVC project to the *wwwroot* folder in the ASP.NET Core project.

The old ASP.NET MVC project uses [Bootstrap](#) for its styling and stores the Bootstrap files in the *Content* and *Scripts* folders. The template, which generated the old ASP.NET MVC project, references Bootstrap in the layout file (*Views/Shared/_Layout.cshtml*). You could copy the *bootstrap.js* and *bootstrap.css* files from the ASP.NET MVC project to the *wwwroot* folder in the new project, but that approach doesn't use the improved mechanism for managing client-side dependencies in ASP.NET Core.

In the new project, we'll add support for Bootstrap (and other client-side libraries) using [Bower](#):

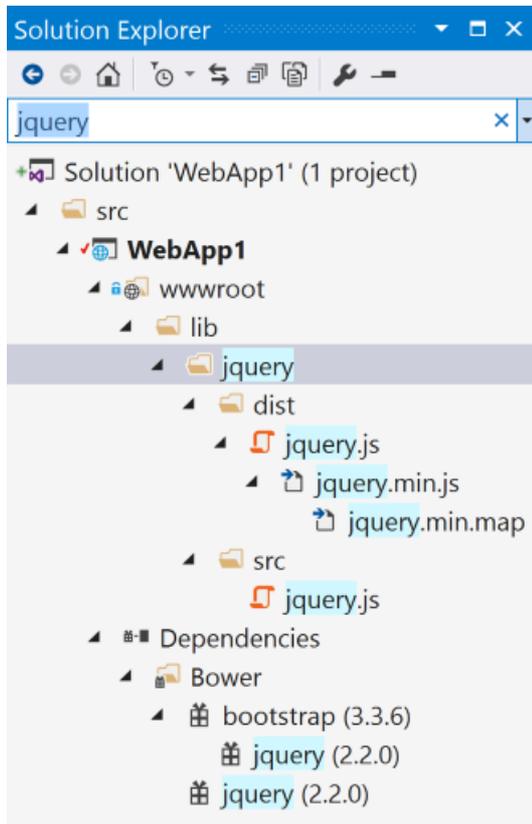
- Add a [Bower](#) configuration file named *bower.json* to the project root (Right-click on the project, and then **Add > New Item > Bower Configuration File**). Add [Bootstrap](#) and [jQuery](#) to the file (see the highlighted lines below).

```

{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "2.2.0"
  }
}

```

Upon saving the file, Bower will automatically download the dependencies to the `wwwroot/lib` folder. You can use the **Search Solution Explorer** box to find the path of the assets:



See [Manage Client-Side Packages with Bower](#) for more information.

Migrate the layout file

- Copy the `_ViewStart.cshtml` file from the old ASP.NET MVC project's `Views` folder into the ASP.NET Core project's `Views` folder. The `_ViewStart.cshtml` file has not changed in ASP.NET Core MVC.
- Create a `Views/Shared` folder.
- *Optional:* Copy `_ViewImports.cshtml` from the `FullAspNetCore` MVC project's `Views` folder into the ASP.NET Core project's `Views` folder. Remove any namespace declaration in the `_ViewImports.cshtml` file. The `_ViewImports.cshtml` file provides namespaces for all the view files and brings in [Tag Helpers](#). Tag Helpers are used in the new layout file. The `_ViewImports.cshtml` file is new for ASP.NET Core.
- Copy the `_Layout.cshtml` file from the old ASP.NET MVC project's `Views/Shared` folder into the ASP.NET Core project's `Views/Shared` folder.

Open `_Layout.cshtml` file and make the following changes (the completed code is shown below):

- Replace `@Styles.Render("~/Content/css")` with a `<link>` element to load `bootstrap.css` (see below).
- Remove `@Scripts.Render("~/bundles/modernizr")`.

- Comment out the `@Html.Partial("_LoginPartial")` line (surround the line with `@*...*@`). We'll return to it in a future tutorial.
- Replace `@Scripts.Render("~/bundles/jquery")` with a `<script>` element (see below).
- Replace `@Scripts.Render("~/bundles/bootstrap")` with a `<script>` element (see below)..

The replacement CSS link:

```
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
```

The replacement script tags:

```
<script src="~/lib/jquery/dist/jquery.js"></script>  
<script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
```

The updated `_Layout.cshtml` file is shown below:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - My ASP.NET Application</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />

</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class =
"navbar-brand" })
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li>@Html.ActionLink("Home", "Index", "Home")</li>
          <li>@Html.ActionLink("About", "About", "Home")</li>
          <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
        </ul>
        @@Html.Partial("_LoginPartial")*
      </div>
    </div>
  </div>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
    </footer>
  </div>

  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  @RenderSection("scripts", required: false)
</body>
</html>

```

View the site in the browser. It should now load correctly, with the expected styles in place.

- *Optional:* You might want to try using the new layout file. For this project you can copy the layout file from the *FullAspNetCore* project. The new layout file uses [Tag Helpers](#) and has other improvements.

Configure Bundling & Minification

For information about how to configure bundling and minification, see [Bundling and Minification](#).

Solving HTTP 500 errors

There are many problems that can cause a HTTP 500 error message that contain no information on the source of the problem. For example, if the *Views/_ViewImports.cshtml* file contains a namespace that doesn't exist in your project, you'll get a HTTP 500 error. To get a detailed error message, add the following code:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

See **Using the Developer Exception Page** in [Error Handling](#) for more information.

Additional Resources

- [Client-Side Development](#)
- [Tag Helpers](#)

Migrating Configuration

11/29/2017 • 2 min to read • [Edit Online](#)

By [Steve Smith](#) and [Scott Addie](#)

In the previous article, we began [migrating an ASP.NET MVC project to ASP.NET Core MVC](#). In this article, we migrate configuration.

[View or download sample code](#) ([how to download](#))

Setup Configuration

ASP.NET Core no longer uses the *Global.asax* and *web.config* files that previous versions of ASP.NET utilized. In earlier versions of ASP.NET, application startup logic was placed in an `Application_StartUp` method within *Global.asax*. Later, in ASP.NET MVC, a *Startup.cs* file was included in the root of the project; and, it was called when the application started. ASP.NET Core has adopted this approach completely by placing all startup logic in the *Startup.cs* file.

The *web.config* file has also been replaced in ASP.NET Core. Configuration itself can now be configured, as part of the application startup procedure described in *Startup.cs*. Configuration can still utilize XML files, but typically ASP.NET Core projects will place configuration values in a JSON-formatted file, such as *appsettings.json*. ASP.NET Core's configuration system can also easily access environment variables, which can provide a more secure and robust location for environment-specific values. This is especially true for secrets like connection strings and API keys that should not be checked into source control. See [Configuration](#) to learn more about configuration in ASP.NET Core.

For this article, we are starting with the partially-migrated ASP.NET Core project from [the previous article](#). To setup configuration, add the following constructor and property to the *Startup.cs* file located in the root of the project:

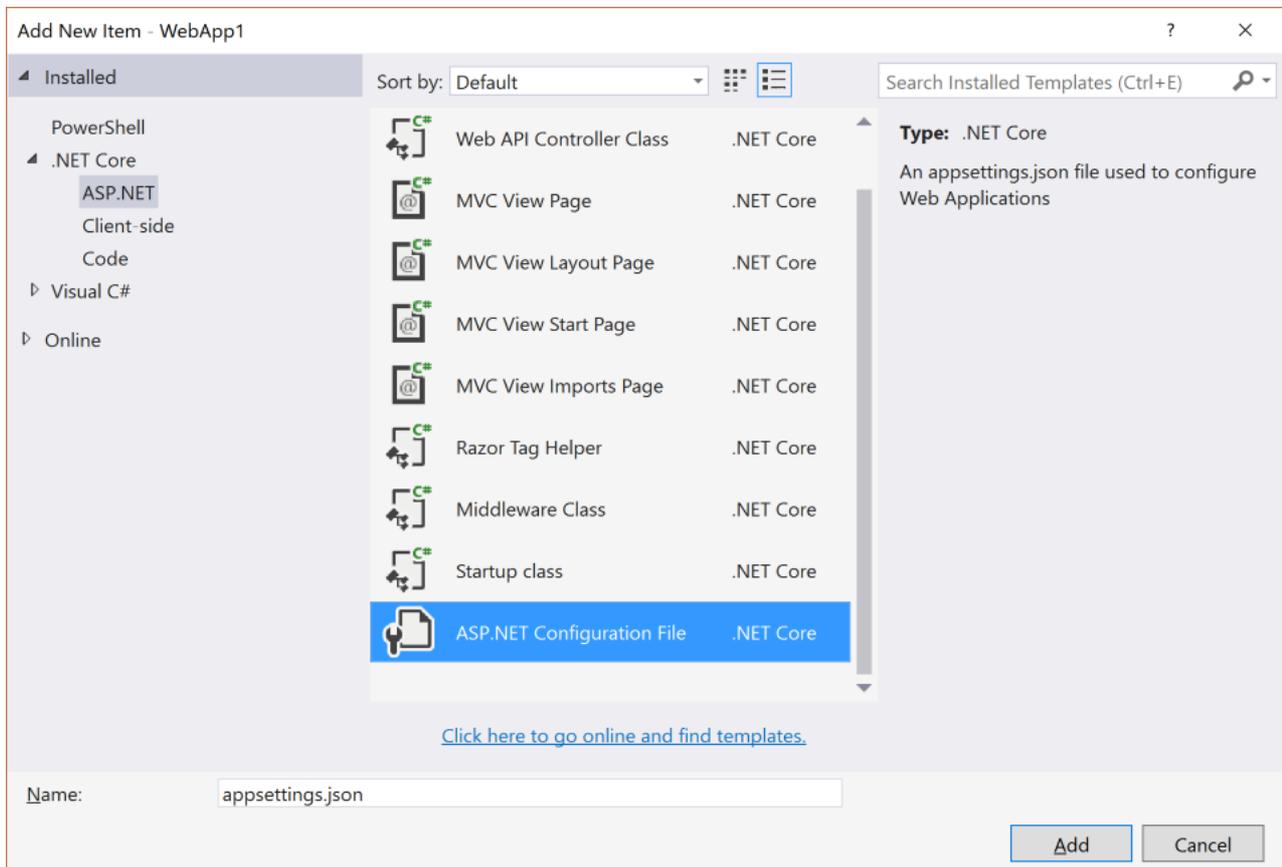
```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; }
```

Note that at this point, the *Startup.cs* file will not compile, as we still need to add the following `using` statement:

```
using Microsoft.Extensions.Configuration;
```

Add an *appsettings.json* file to the root of the project using the appropriate item template:



Migrate Configuration Settings from web.config

Our ASP.NET MVC project included the required database connection string in *web.config*, in the `<connectionStrings>` element. In our ASP.NET Core project, we are going to store this information in the *appsettings.json* file. Open *appsettings.json*, and note that it already includes the following:

```
{
  "Data": {
    "DefaultConnection": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trusted_Connection=True;"
    }
  }
}
```

In the highlighted line depicted above, change the name of the database from **_CHANGE_ME** to the name of your database.

Summary

ASP.NET Core places all startup logic for the application in a single file, in which the necessary services and dependencies can be defined and configured. It replaces the *web.config* file with a flexible configuration feature that can leverage a variety of file formats, such as JSON, as well as environment variables.

Migrating Authentication and Identity

10/13/2017 • 2 min to read • [Edit Online](#)

By [Steve Smith](#)

In the previous article we [migrated configuration from an ASP.NET MVC project to ASP.NET Core MVC](#). In this article, we migrate the registration, login, and user management features.

Configure Identity and Membership

In ASP.NET MVC, authentication and identity features are configured using ASP.NET Identity in `Startup.Auth.cs` and `IdentityConfig.cs`, located in the `App_Start` folder. In ASP.NET Core MVC, these features are configured in `Startup.cs`.

Install the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` and `Microsoft.AspNetCore.Authentication.Cookies` NuGet packages.

Then, open `Startup.cs` and update the `ConfigureServices()` method to use Entity Framework and Identity services:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
    services.AddEntityFramework(Configuration)
        .AddSqlServer()
        .AddDbContext<ApplicationDbContext>();

    // Add Identity services to the services container.
    services.AddIdentity<ApplicationUser, IdentityRole>(Configuration)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddMvc();
}
```

At this point, there are two types referenced in the above code that we haven't yet migrated from the ASP.NET MVC project: `ApplicationDbContext` and `ApplicationUser`. Create a new `Models` folder in the ASP.NET Core project, and add two classes to it corresponding to these types. You will find the ASP.NET MVC versions of these classes in `/Models/IdentityModels.cs`, but we will use one file per class in the migrated project since that's more clear.

`ApplicationUser.cs`:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace NewMvc6Project.Models
{
    public class ApplicationUser : IdentityUser
    {
    }
}
```

`ApplicationDbContext.cs`:

```

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.Data.Entity;

namespace NewMvc6Project.Models
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext()
        {
            Database.EnsureCreated();
        }

        protected override void OnConfiguring(DbContextOptionsBuilder options)
        {
            options.UseSqlServer();
        }
    }
}

```

The ASP.NET Core MVC Starter Web project doesn't include much customization of users, or the `ApplicationDbContext`. When migrating a real application, you will also need to migrate all of the custom properties and methods of your application's user and `DbContext` classes, as well as any other Model classes your application utilizes (for example, if your `DbContext` has a `DbSet`, you will of course need to migrate the `Album` class).

With these files in place, the `Startup.cs` file can be made to compile by updating its using statements:

```

using Microsoft.Framework.ConfigurationModel;
using Microsoft.AspNetCore.Hosting;
using NewMvc6Project.Models;
using Microsoft.AspNetCore.Identity;

```

Our application is now ready to support authentication and identity services - it just needs to have these features exposed to users.

Migrate Registration and Login Logic

With identity services configured for the application and data access configured using Entity Framework and SQL Server, we are now ready to add support for registration and login to the application. Recall that [earlier in the migration process](#) we commented out a reference to `_LoginPartial` in `_Layout.cshtml`. Now it's time to return to that code, uncomment it, and add in the necessary controllers and views to support login functionality.

Update `_Layout.cshtml`; uncomment the `@Html.Partial` line:

```

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
@*@Html.Partial("_LoginPartial")*@
</div>
</div>

```

Now, add a new MVC View Page called `_LoginPartial` to the `Views/Shared` folder:

Update `_LoginPartial.cshtml` with the following code (replace all of its contents):

```
@inject SignInManager<User> SignInManager
@Inject UserManager<User> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Account" asp-action="LogOff" method="post" id="logoutForm"
class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="" asp-controller="Manage" asp-action="Index" title="Manage">Hello
@UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-link">Log off</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="" asp-controller="Account" asp-action="Register">Register</a></li>
        <li><a asp-area="" asp-controller="Account" asp-action="Login">Log in</a></li>
    </ul>
}
```

At this point, you should be able to refresh the site in your browser.

Summary

ASP.NET Core introduces changes to the ASP.NET Identity features. In this article, you have seen how to migrate the authentication and user management features of an ASP.NET Identity to ASP.NET Core.

Migrating from ASP.NET Web API

10/2/2017 • 6 min to read • [Edit Online](#)

By [Steve Smith](#) and [Scott Addie](#)

Web APIs are HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Core MVC includes support for building Web APIs providing a single, consistent way of building web applications. In this article, we demonstrate the steps required to migrate a Web API implementation from ASP.NET Web API to ASP.NET Core MVC.

[View or download sample code](#) ([how to download](#))

Review ASP.NET Web API Project

This article uses the sample project, *ProductsApp*, created in the article [Getting Started with ASP.NET Web API](#) as its starting point. In that project, a simple ASP.NET Web API project is configured as follows.

In *Global.asax.cs*, a call is made to `WebApiConfig.Register`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Routing;

namespace ProductsApp
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

`WebApiConfig` is defined in *App_Start*, and has just one static `Register` method:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace ProductsApp
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

This class configures [attribute routing](#), although it's not actually being used in the project. It also configures the routing table which is used by ASP.NET Web API. In this case, ASP.NET Web API will expect URLs to match the format `/api/{controller}/{id}`, with `{id}` being optional.

The *ProductsApp* project includes just one simple controller, which inherits from `ApiController` and exposes two methods:

```

using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}

```

Finally, the model, *Product*, used by the *ProductsApp*, is a simple class:

```

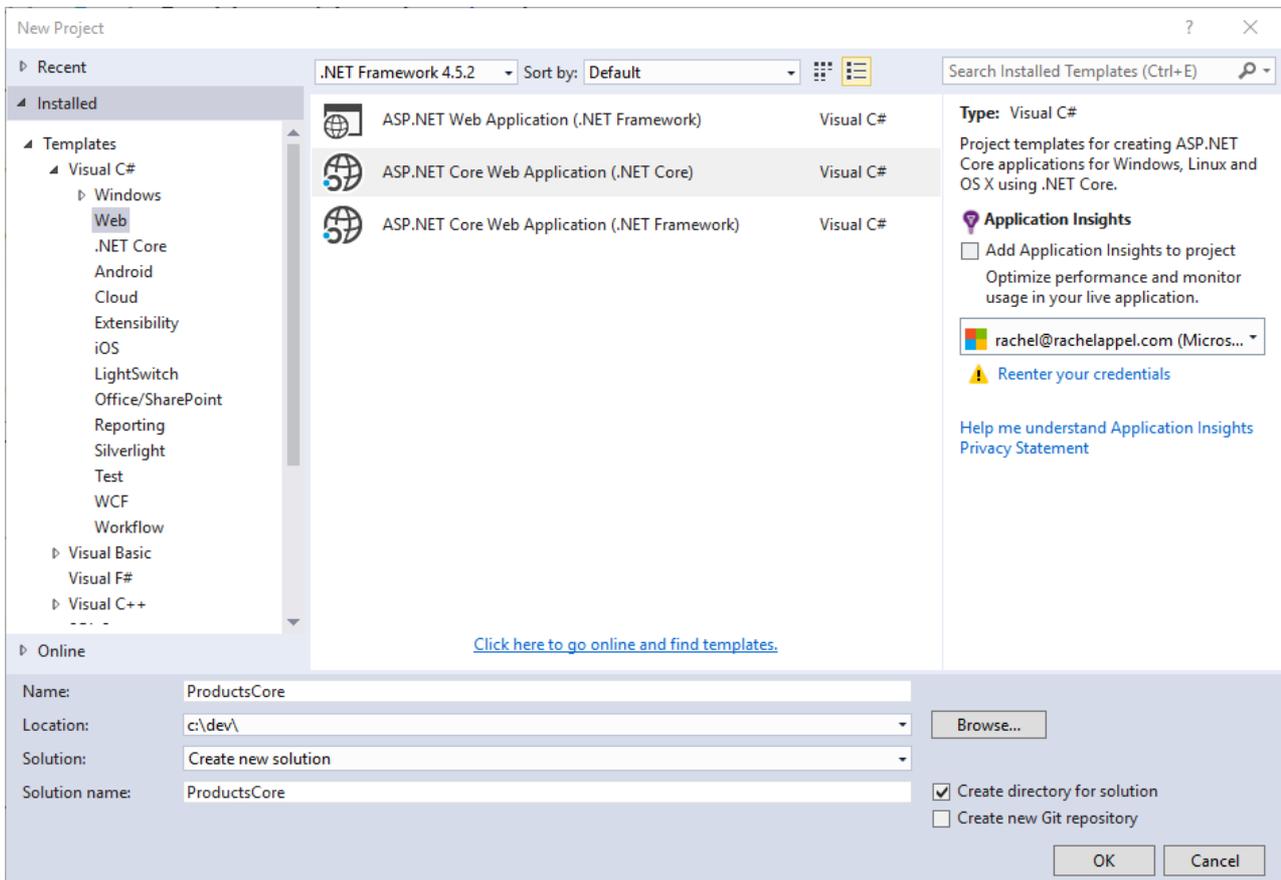
namespace ProductsApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}

```

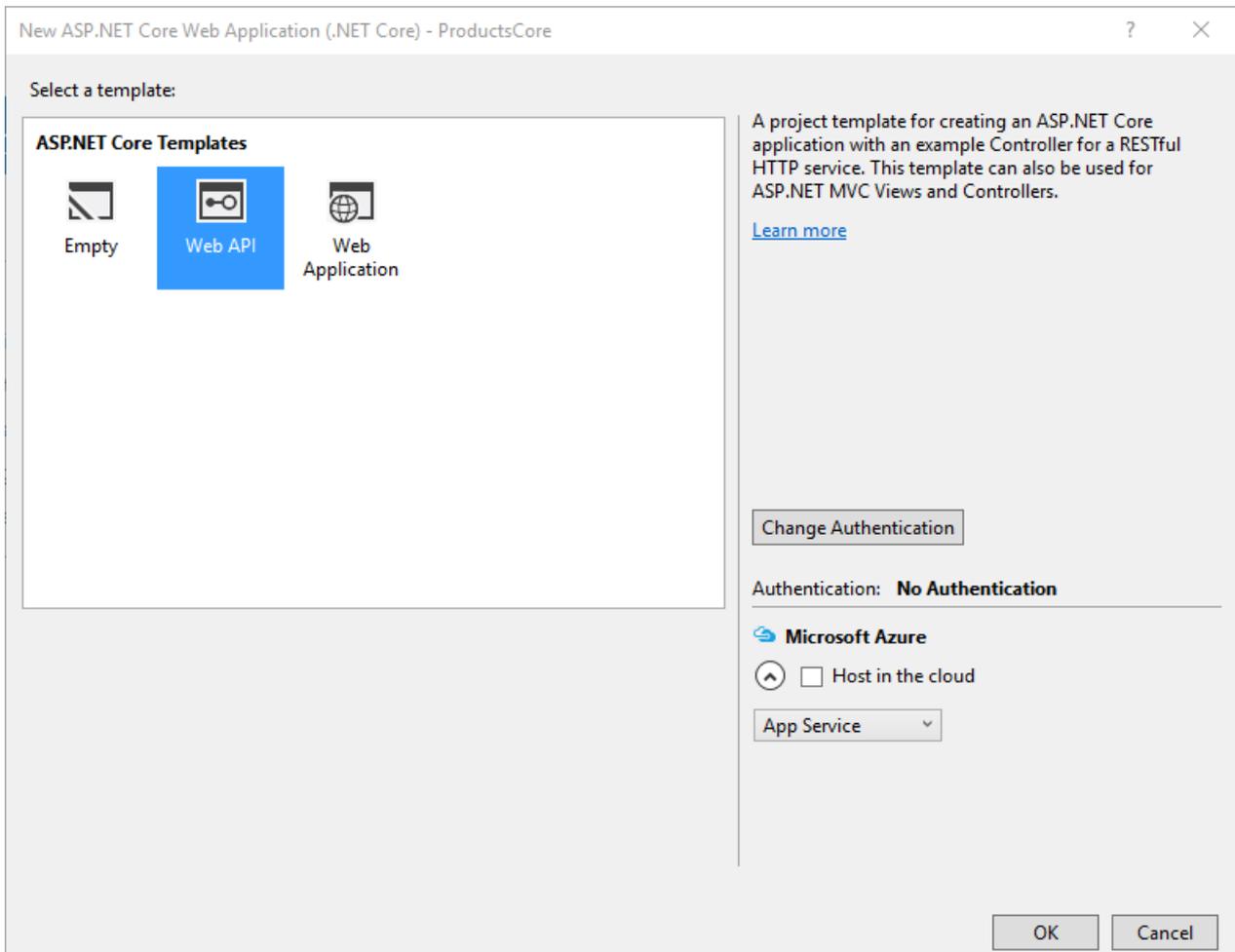
Now that we have a simple project from which to start, we can demonstrate how to migrate this Web API project to ASP.NET Core MVC.

Create the Destination Project

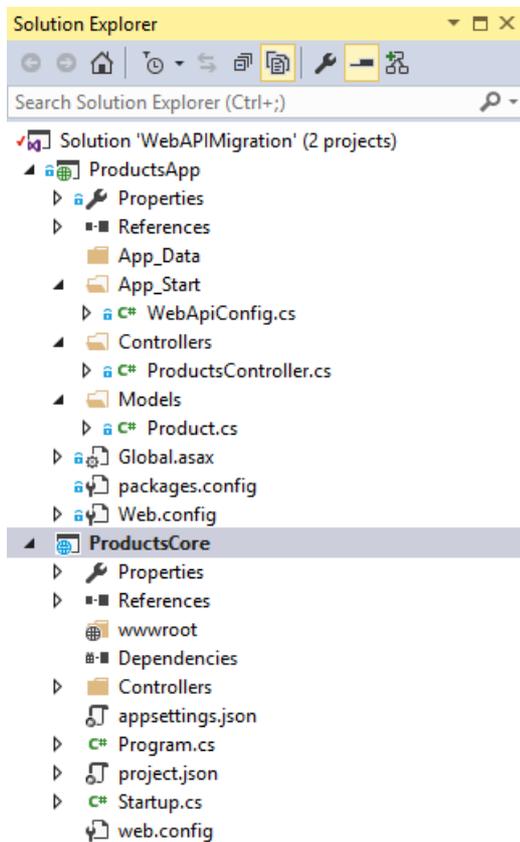
Using Visual Studio, create a new, empty solution, and name it *WebAPIMigration*. Add the existing *ProductsApp* project to it, then, add a new ASP.NET Core Web Application Project to the solution. Name the new project *ProductsCore*.



Next, choose the Web API project template. We will migrate the *ProductsApp* contents to this new project.



Delete the `Project_Readme.html` file from the new project. Your solution should now look like this:



Migrate Configuration

ASP.NET Core no longer uses *Global.asax*, *web.config*, or *App_Start* folders. Instead, all startup tasks are done in *Startup.cs* in the root of the project (see [Application Startup](#)). In ASP.NET Core MVC, attribute-based routing is now included by default when `UseMvc()` is called; and, this is the recommended approach for configuring Web API routes (and is how the Web API starter project handles routing).

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace ProductsCore
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
                .AddEnvironmentVariables();
            Configuration = builder.Build();
        }

        public IConfigurationRoot Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            // Add framework services.
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole(Configuration.GetSection("Logging"));
            loggerFactory.AddDebug();

            app.UseMvc();
        }
    }
}

```

Assuming you want to use attribute routing in your project going forward, no additional configuration is needed. Simply apply the attributes as needed to your controllers and actions, as is done in the sample `ValuesController` class that is included in the Web API starter project:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace ProductsCore.Controllers
{
    [Route("api/[controller]")]
    public class ValuesController : Controller
    {
        // GET api/values
        [HttpGet]
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        [HttpGet("{id}")]
        public string Get(int id)
        {
            return "value";
        }

        // POST api/values
        [HttpPost]
        public void Post([FromBody]string value)
        {
        }

        // PUT api/values/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
}

```

Note the presence of `[controller]` on line 8. Attribute-based routing now supports certain tokens, such as `[controller]` and `[action]`. These tokens are replaced at runtime with the name of the controller or action, respectively, to which the attribute has been applied. This serves to reduce the number of magic strings in the project, and it ensures the routes will be kept synchronized with their corresponding controllers and actions when automatic rename refactorings are applied.

To migrate the Products API controller, we must first copy `ProductsController` to the new project. Then simply include the route attribute on the controller:

```
[Route("api/[controller]")]
```

You also need to add the `[HttpGet]` attribute to the two methods, since they both should be called via HTTP Get. Include the expectation of an "id" parameter in the attribute for `GetProduct()`:

```
// /api/products
[HttpGet]
...

// /api/products/1
[HttpGet("{id}")]
```

At this point, routing is configured correctly; however, we can't yet test it. Additional changes must be made before *ProductsController* will compile.

Migrate Models and Controllers

The last step in the migration process for this simple Web API project is to copy over the Controllers and any Models they use. In this case, simply copy *Controllers/ProductsController.cs* from the original project to the new one. Then, copy the entire Models folder from the original project to the new one. Adjust the namespaces to match the new project name (*ProductsCore*). At this point, you can build the application, and you will find a number of compilation errors. These should generally fall into the following categories:

- *ApiController* does not exist
- *System.Web.Http* namespace does not exist
- *IHttpActionResult* does not exist

Fortunately, these are all very easy to correct:

- Change *ApiController* to *Controller* (you may need to add *using Microsoft.AspNetCore.Mvc*)
- Delete any using statement referring to *System.Web.Http*
- Change any method returning *IHttpActionResult* to return a *IActionResult*

Once these changes have been made and unused using statements removed, the migrated *ProductsController* class looks like this:

```

using Microsoft.AspNetCore.Mvc;
using ProductsCore.Models;
using System.Collections.Generic;
using System.Linq;

namespace ProductsCore.Controllers
{
    [Route("api/[controller]")]
    public class ProductsController : Controller
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        // /api/products
        [HttpGet]
        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        // /api/products/1
        [HttpGet("{id}")]
        public IActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}

```

You should now be able to run the migrated project and browse to */api/products*; and, you should see the full list of 3 products. Browse to */api/products/1* and you should see the first product.

Summary

Migrating a simple ASP.NET Web API project to ASP.NET Core MVC is fairly straightforward, thanks to the built-in support for Web APIs in ASP.NET Core MVC. The main pieces every ASP.NET Web API project will need to migrate are routes, controllers, and models, along with updates to the types used by controllers and actions.

Migrating HTTP handlers and modules to ASP.NET Core middleware

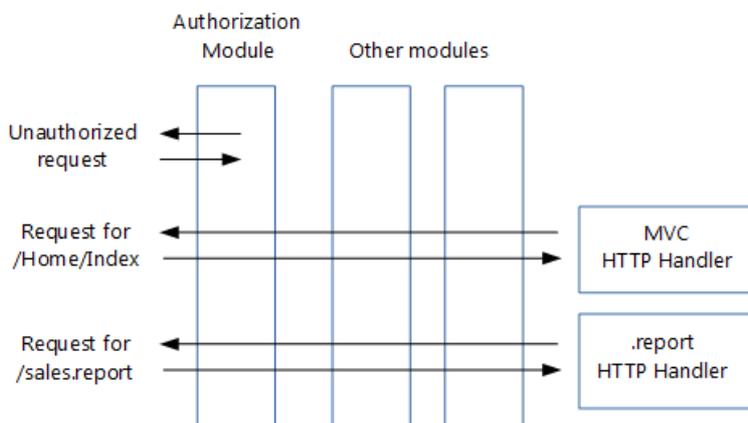
11/29/2017 • 15 min to read • [Edit Online](#)

By [Matt Perdeck](#)

This article shows how to migrate existing ASP.NET [HTTP modules and handlers](#) from `system.webserver` to ASP.NET Core [middleware](#).

Modules and handlers revisited

Before proceeding to ASP.NET Core middleware, let's first recap how HTTP modules and handlers work:



Handlers are:

- Classes that implement [IHttpHandler](#)
- Used to handle requests with a given file name or extension, such as `.report`
- [Configured](#) in `Web.config`

Modules are:

- Classes that implement [IHttpModule](#)
- Invoked for every request
- Able to short-circuit (stop further processing of a request)
- Able to add to the HTTP response, or create their own
- [Configured](#) in `Web.config`

The order in which modules process incoming requests is determined by:

1. The [application life cycle](#), which is a series of events fired by ASP.NET: [BeginRequest](#), [AuthenticateRequest](#), etc. Each module can create a handler for one or more events.
2. For the same event, the order in which they are configured in `Web.config`.

In addition to modules, you can add handlers for the life cycle events to your `Global.asax.cs` file. These handlers run after the handlers in the configured modules.

From handlers and modules to middleware

Middleware are simpler than HTTP modules and handlers:

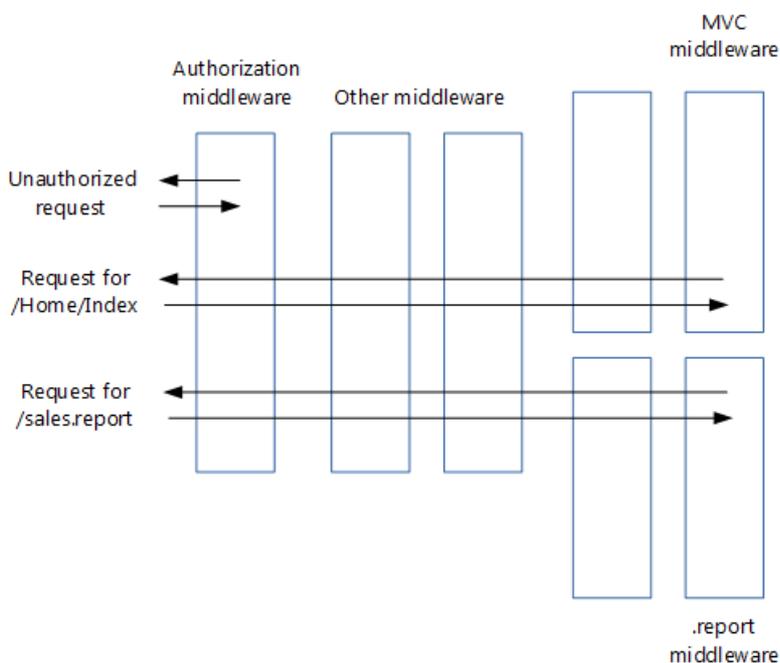
- Modules, handlers, *Global.asax.cs*, *Web.config* (except for IIS configuration) and the application life cycle are gone
- The roles of both modules and handlers have been taken over by middleware
- Middleware are configured using code rather than in *Web.config*
- [Pipeline branching](#) lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.

Middleware are very similar to modules:

- Invoked in principle for every request
- Able to short-circuit a request, by [not passing the request to the next middleware](#)
- Able to create their own HTTP response

Middleware and modules are processed in a different order:

- Order of middleware is based on the order in which they are inserted into the request pipeline, while order of modules is mainly based on [application life cycle](#) events
- Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses
- See [Creating a middleware pipeline with IApplicationBuilder](#)



Note how in the image above, the authentication middleware short-circuited the request.

Migrating module code to middleware

An existing HTTP module will look similar to this:

```

// ASP.NET 4 module

using System;
using System.Web;

namespace MyApp.Modules
{
    public class MyModule : IHttpModule
    {
        public void Dispose()
        {
        }

        public void Init(HttpApplication application)
        {
            application.BeginRequest += (new EventHandler(this.Application_BeginRequest));
            application.EndRequest += (new EventHandler(this.Application_EndRequest));
        }

        private void Application_BeginRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the beginning of request processing.
        }

        private void Application_EndRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the end of request processing.
        }
    }
}

```

As shown in the [Middleware](#) page, an ASP.NET Core middleware is a class that exposes an `Invoke` method taking an `HttpContext` and returning a `Task`. Your new middleware will look like this:

```

// ASP.NET Core middleware

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyMiddleware
    {
        private readonly RequestDelegate _next;

        public MyMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            // Do something with context near the beginning of request processing.

            await _next.Invoke(context);

            // Clean up.
        }
    }

    public static class MyMiddlewareExtensions
    {
        public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyMiddleware>();
        }
    }
}

```

The above middleware template was taken from the section on [writing middleware](#).

The `MyMiddlewareExtensions` helper class makes it easier to configure your middleware in your `Startup` class. The `UseMyMiddleware` method adds your middleware class to the request pipeline. Services required by the middleware get injected in the middleware's constructor.

Your module might terminate a request, for example if the user is not authorized:

```

// ASP.NET 4 module that may terminate the request

private void Application_BeginRequest(Object source, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;

    // Do something with context near the beginning of request processing.

    if (TerminateRequest())
    {
        context.Response.End();
        return;
    }
}

```

A middleware handles this by not calling `Invoke` on the next middleware in the pipeline. Keep in mind that this does not fully terminate the request, because previous middlewares will still be invoked when the response makes its way back through the pipeline.

```
// ASP.NET Core middleware that may terminate the request

public async Task Invoke(HttpContext context)
{
    // Do something with context near the beginning of request processing.

    if (!TerminateRequest())
        await _next.Invoke(context);

    // Clean up.
}
```

When you migrate your module's functionality to your new middleware, you may find that your code doesn't compile because the `HttpContext` class has significantly changed in ASP.NET Core. [Later on](#), you'll see how to migrate to the new ASP.NET Core `HttpContext`.

Migrating module insertion into the request pipeline

HTTP modules are typically added to the request pipeline using *Web.config*:

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <modules>
      <add name="MyModule" type="MyApp.Modules.MyModule"/>
    </modules>
  </system.webServer>
</configuration>
```

Convert this by [adding your new middleware](#) to the request pipeline in your `Startup` class:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}

```

The exact spot in the pipeline where you insert your new middleware depends on the event that it handled as a module (`BeginRequest`, `EndRequest`, etc.) and its order in your list of modules in *Web.config*.

As previously stated, there is no application life cycle in ASP.NET Core and the order in which responses are processed by middleware differs from the order used by modules. This could make your ordering decision more challenging.

If ordering becomes a problem, you could split your module into multiple middleware components that can be ordered independently.

Migrating handler code to middleware

An HTTP handler looks something like this:

```
// ASP.NET 4 handler

using System.Web;

namespace MyApp.HttpHandlers
{
    public class MyHandler : IHttpHandler
    {
        public bool IsReusable { get { return true; } }

        public void ProcessRequest(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            context.Response.Output.Write(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.QueryString["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }
}
```

In your ASP.NET Core project, you would translate this to a middleware similar to this:

```

// ASP.NET Core middleware migrated from a handler

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyHandlerMiddleware
    {
        // Must have constructor with this signature, otherwise exception at run time
        public MyHandlerMiddleware(RequestDelegate next)
        {
            // This is an HTTP Handler, so no need to store next
        }

        public async Task Invoke(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            await context.Response.WriteAsync(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.Query["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }

    public static class MyHandlerExtensions
    {
        public static IApplicationBuilder UseMyHandler(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyHandlerMiddleware>();
        }
    }
}

```

This middleware is very similar to the middleware corresponding to modules. The only real difference is that here there is no call to `_next.Invoke(context)`. That makes sense, because the handler is at the end of the request pipeline, so there will be no next middleware to invoke.

Migrating handler insertion into the request pipeline

Configuring an HTTP handler is done in *Web.config* and looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandler" verb="*" path="*.report" type="MyApp.HttpHandlers.MyHandler"
resourceType="Unspecified" preCondition="integratedMode"/>
    </handlers>
  </system.webServer>
</configuration>
```

You could convert this by adding your new handler middleware to the request pipeline in your `Startup` class, similar to middleware converted from modules. The problem with that approach is that it would send all requests to your new handler middleware. However, you only want requests with a given extension to reach your middleware. That would give you the same functionality you had with your HTTP handler.

One solution is to branch the pipeline for requests with a given extension, using the `MapWhen` extension method. You do this in the same `Configure` method where you add the other middleware:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}

```

`MapWhen` takes these parameters:

1. A lambda that takes the `HttpContext` and returns `true` if the request should go down the branch. This means you can branch requests not just based on their extension, but also on request headers, query string parameters, etc.
2. A lambda that takes an `IApplicationBuilder` and adds all the middleware for the branch. This means you can add additional middleware to the branch in front of your handler middleware.

Middleware added to the pipeline before the branch will be invoked on all requests; the branch will have no impact on them.

Loading middleware options using the options pattern

Some modules and handlers have configuration options that are stored in *Web.config*. However, in ASP.NET Core a new configuration model is used in place of *Web.config*.

The new [configuration system](#) gives you these options to solve this:

- Directly inject the options into the middleware, as shown in the [next section](#).
- Use the [options pattern](#):

1. Create a class to hold your middleware options, for example:

```
public class MyMiddlewareOptions
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}
```

2. Store the option values

The configuration system allows you to store option values anywhere you want. However, most sites use *appsettings.json*, so we'll take that approach:

```
{
  "MyMiddlewareOptionsSection": {
    "Param1": "Param1Value",
    "Param2": "Param2Value"
  }
}
```

MyMiddlewareOptionsSection here is a section name. It doesn't have to be the same as the name of your options class.

3. Associate the option values with the options class

The options pattern uses ASP.NET Core's dependency injection framework to associate the options type (such as `MyMiddlewareOptions`) with a `MyMiddlewareOptions` object that has the actual options.

Update your `Startup` class:

a. If you're using *appsettings.json*, add it to the configuration builder in the `Startup` constructor:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

b. Configure the options service:

```

public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}

```

c. Associate your options with your options class:

```

public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}

```

4. Inject the options into your middleware constructor. This is similar to injecting options into a controller.

```

public class MyMiddlewareWithParams
{
    private readonly RequestDelegate _next;
    private readonly MyMiddlewareOptions _myMiddlewareOptions;

    public MyMiddlewareWithParams(RequestDelegate next,
        IOptions<MyMiddlewareOptions> optionsAccessor)
    {
        _next = next;
        _myMiddlewareOptions = optionsAccessor.Value;
    }

    public async Task Invoke(HttpContext context)
    {
        // Do something with context near the beginning of request processing
        // using configuration in _myMiddlewareOptions

        await _next.Invoke(context);

        // Do something with context near the end of request processing
        // using configuration in _myMiddlewareOptions
    }
}

```

The [UseMiddleware](#) extension method that adds your middleware to the `IApplicationBuilder` takes care of dependency injection.

This is not limited to `IOptions` objects. Any other object that your middleware requires can be injected this way.

Loading middleware options through direct injection

The options pattern has the advantage that it creates loose coupling between options values and their consumers. Once you've associated an options class with the actual options values, any other class can get access to the options through the dependency injection framework. There is no need to pass around options values.

This breaks down though if you want to use the same middleware twice, with different options. For example an authorization middleware used in different branches allowing different roles. You can't associate two different options objects with the one options class.

The solution is to get the options objects with the actual options values in your `Startup` class and pass those directly to each instance of your middleware.

1. Add a second key to *appsettings.json*

To add a second set of options to the *appsettings.json* file, use a new key to uniquely identify it:

```
{
  "MyMiddlewareOptionsSection2": {
    "Param1": "Param1Value2",
    "Param2": "Param2Value2"
  },
  "MyMiddlewareOptionsSection": {
    "Param1": "Param1Value",
    "Param2": "Param2Value"
  }
}
```

2. Retrieve options values and pass them to middleware. The `Use...` extension method (which adds your middleware to the pipeline) is a logical place to pass in the option values:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}

```

3. Enable middleware to take an options parameter. Provide an overload of the `Use...` extension method (that takes the options parameter and passes it to `UseMiddleware`). When `UseMiddleware` is called with parameters, it passes the parameters to your middleware constructor when it instantiates the middleware object.

```

public static class MyMiddlewareWithParamsExtensions
{
    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>();
    }

    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder, MyMiddlewareOptions myMiddlewareOptions)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>(
            new OptionsWrapper<MyMiddlewareOptions>(myMiddlewareOptions));
    }
}

```

Note how this wraps the options object in an `OptionsWrapper` object. This implements `IOptions`, as expected by the middleware constructor.

Migrating to the new HttpContext

You saw earlier that the `Invoke` method in your middleware takes a parameter of type `HttpContext`:

```
public async Task Invoke(HttpContext context)
```

`HttpContext` has significantly changed in ASP.NET Core. This section shows how to translate the most commonly used properties of `System.Web.HttpContext` to the new `Microsoft.AspNetCore.Http.HttpContext`.

HttpContext

HttpContext.Items translates to:

```
IDictionary<object, object> items = httpContext.Items;
```

Unique request ID (no System.Web.HttpContext counterpart)

Gives you a unique id for each request. Very useful to include in your logs.

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

HttpContext.Request.HttpMethod translates to:

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString translates to:

```
IQueryCollection queryParameters = httpContext.Request.Query;

// If no query parameter "key" used, values will have 0 items
// If single value used for a key (...?key=v1), values will have 1 item ("v1")
// If key has multiple values (...?key=v1&key=v2), values will have 2 items ("v1" and "v2")
IList<string> values = queryParameters["key"];

// If no query parameter "key" used, value will be ""
// If single value used for a key (...?key=v1), value will be "v1"
// If key has multiple values (...?key=v1&key=v2), value will be "v1,v2"
string value = queryParameters["key"].ToString();
```

HttpContext.Request.Url and **HttpContext.Request.RawUrl** translate to:

```
// using Microsoft.AspNetCore.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();
```

HttpContext.Request.IsSecureConnection translates to:

```
var isSecureConnection = httpContext.Request.IsHttps;
```

HttpContext.Request.UserHostAddress translates to:

```
var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();
```

HttpContext.Request.Cookies translates to:

```
IRequestCookieCollection cookies = httpContext.Request.Cookies;
string unknownCookieValue = cookies["unknownCookie"]; // will be null (no exception)
string knownCookieValue = cookies["cookie1name"]; // will be actual value
```

HttpContext.Request.RequestContext.RouteData translates to:

```
var routeValue = httpContext.GetRouteValue("key");
```

HttpContext.Request.Headers translates to:

```
// using Microsoft.AspNetCore.Http.Headers;
// using Microsoft.Net.Http.Headers;

IHeaderDictionary headersDictionary = httpContext.Request.Headers;

// GetTypedHeaders extension method provides strongly typed access to many headers
var requestHeaders = httpContext.Request.GetTypedHeaders();
CacheControlHeaderValue cacheControlHeaderValue = requestHeaders.CacheControl;

// For unknown header, unknownheaderValues has zero items and unknownheaderValue is ""
IList<string> unknownheaderValues = headersDictionary["unknownheader"];
string unknownheaderValue = headersDictionary["unknownheader"].ToString();

// For known header, knownheaderValues has 1 item and knownheaderValue is the value
IList<string> knownheaderValues = headersDictionary[HeaderNames.AcceptLanguage];
string knownheaderValue = headersDictionary[HeaderNames.AcceptLanguage].ToString();
```

HttpContext.Request.UserAgent translates to:

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer translates to:

```
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType translates to:

```
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType; // ex. application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type; // ex. application
string contentSubType = mediaHeaderValue?.SubType; // ex. x-www-form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form translates to:

```
if (HttpContext.Request.HasFormContentType)
{
    IFormCollection form;

    form = HttpContext.Request.Form; // sync
    // Or
    form = await HttpContext.Request.ReadFormAsync(); // async

    string firstName = form["firstname"];
    string lastName = form["lastname"];
}
```

WARNING

Read form values only if the content sub type is *x-www-form-urlencoded* or *form-data*.

HttpContext.Request.InputStream translates to:

```
string inputBody;
using (var reader = new System.IO.StreamReader(
    HttpContext.Request.Body, System.Text.Encoding.UTF8))
{
    inputBody = reader.ReadToEnd();
}
```

WARNING

Use this code only in a handler type middleware, at the end of a pipeline.

You can read the raw body as shown above only once per request. Middleware trying to read the body after the first read will read an empty body.

This does not apply to reading a form as shown earlier, because that is done from a buffer.

HttpContext.Response

HttpContext.Response.Status and **HttpContext.Response.StatusDescription** translate to:

```
// using Microsoft.AspNetCore.Http;
httpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding and **HttpContext.Response.ContentType** translate to:

```
// using Microsoft.Net.Http.Headers;
var mediaType = new MediaTypeHeaderValue("application/json");
mediaType.Encoding = System.Text.Encoding.UTF8;
httpContext.Response.ContentType = mediaType.ToString();
```

HttpContext.Response.ContentType on its own also translates to:

```
httpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output translates to:

```
string responseContent = GetResponseContent();
await httpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

Serving up a file is discussed [here](#).

HttpContext.Response.Headers

Sending response headers is complicated by the fact that if you set them after anything has been written to the response body, they will not be sent.

The solution is to set a callback method that will be called right before writing to the response starts. This is best done at the start of the `Invoke` method in your middleware. It is this callback method that sets your response headers.

The following code sets a callback method called `SetHeaders`:

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
}
```

The `SetHeaders` callback method would look like this:

```

// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

private Task SetHeaders(object context)
{
    var httpContext = (HttpContext)context;

    // Set header with single value
    httpContext.Response.Headers["ResponseHeaderName"] = "headerValue";

    // Set header with multiple values
    string[] responseHeaderValues = new string[] { "headerValue1", "headerValue1" };
    httpContext.Response.Headers["ResponseHeaderName"] = responseHeaderValues;

    // Translating ASP.NET 4's HttpContext.Response.RedirectLocation
    httpContext.Response.Headers[HeaderNames.Location] = "http://www.example.com";
    // Or
    httpContext.Response.Redirect("http://www.example.com");

    // GetTypedHeaders extension method provides strongly typed access to many headers
    var responseHeaders = httpContext.Response.GetTypedHeaders();

    // Translating ASP.NET 4's HttpContext.Response.CacheControl
    responseHeaders.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = new System.TimeSpan(365, 0, 0, 0)
        // Many more properties available
    };

    // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}

```

HttpContext.Response.Cookies

Cookies travel to the browser in a *Set-Cookie* response header. As a result, sending cookies requires the same callback as used for sending response headers:

```

public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetCookies, state: httpContext);
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
}

```

The `SetCookies` callback method would look like the following:

```

private Task SetCookies(object context)
{
    var httpContext = (HttpContext)context;

    IResponseCookies responseCookies = httpContext.Response.Cookies;

    responseCookies.Append("cookie1name", "cookie1value");
    responseCookies.Append("cookie2name", "cookie2value",
        new CookieOptions { Expires = System.DateTime.Now.AddDays(5), HttpOnly = true });

    // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}

```

Additional Resources

- [HTTP Handlers and HTTP Modules Overview](#)
- [Configuration](#)
- [Application Startup](#)
- [Middleware](#)

Migrating from ASP.NET to ASP.NET Core 2.0

11/29/2017 • 7 min to read • [Edit Online](#)

By [Isaac Levin](#)

This article serves as a reference guide for migrating ASP.NET applications to ASP.NET Core 2.0.

Prerequisites

- [.NET Core 2.0.0 SDK](#) or later.

Target Frameworks

ASP.NET Core 2.0 projects offer developers the flexibility of targeting .NET Core, .NET Framework, or both. See [Choosing between .NET Core and .NET Framework for server apps](#) to determine which target framework is most appropriate.

When targeting .NET Framework, projects need to reference individual NuGet packages.

Targeting .NET Core allows you to eliminate numerous explicit package references, thanks to the ASP.NET Core 2.0 [metapackage](#). Install the `Microsoft.AspNetCore.All` metapackage in your project:

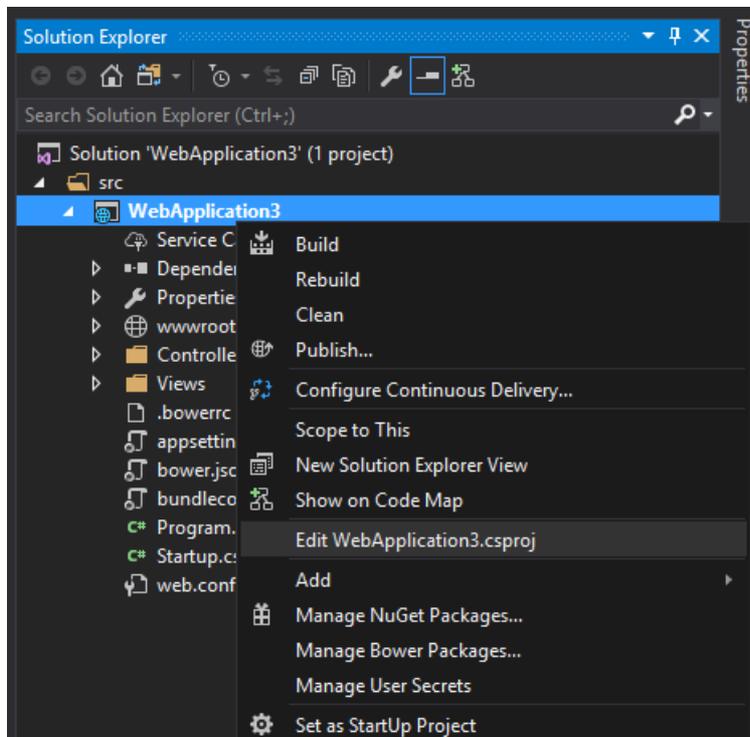
```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
</ItemGroup>
```

When the metapackage is used, no packages referenced in the metapackage are deployed with the app. The .NET Core Runtime Store includes these assets, and they are precompiled to improve performance. See [Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.x](#) for more detail.

Project structure differences

The `.csproj` file format has been simplified in ASP.NET Core. Some notable changes include:

- Explicit inclusion of files is not necessary for them to be considered part of the project. This reduces the risk of XML merge conflicts when working on large teams.
- There are no GUID-based references to other projects, which improves file readability.
- The file can be edited without unloading it in Visual Studio:



Global.asax file replacement

ASP.NET Core introduced a new mechanism for bootstrapping an app. The entry point for ASP.NET applications is the *Global.asax* file. Tasks such as route configuration and filter and area registrations are handled in the *Global.asax* file.

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

This approach couples the application and the server to which it's deployed in a way that interferes with the implementation. In an effort to decouple, [OWIN](#) was introduced to provide a cleaner way to use multiple frameworks together. OWIN provides a pipeline to add only the modules needed. The hosting environment takes a [Startup](#) function to configure services and the app's request pipeline. `Startup` registers a set of middleware with the application. For each request, the application calls each of the the middleware components with the head pointer of a linked list to an existing set of handlers. Each middleware component can add one or more handlers to the request handling pipeline. This is accomplished by returning a reference to the handler that is the new head of the list. Each handler is responsible for remembering and invoking the next handler in the list. With ASP.NET Core, the entry point to an application is `Startup`, and you no longer have a dependency on *Global.asax*. When using OWIN with .NET Framework, use something like the following as a pipeline:

```

using Owin;
using System.Web.Http;

namespace WebApi
{
    // Note: By default all requests go through this OWIN pipeline. Alternatively you can turn this off by
    // adding an appSetting owin:AutomaticAppStartup with value "false".
    // With this turned off you can still have OWIN apps listening on specific routes by adding routes in
    // global.asax file using MapOwinPath or MapOwinRoute extensions on RouteTable.Routes
    public class Startup
    {
        // Invoked once at startup to configure your application.
        public void Configuration(IAppBuilder builder)
        {
            HttpConfiguration config = new HttpConfiguration();
            config.Routes.MapHttpRoute("Default", "{controller}/{customerID}", new { controller = "Customer",
customerID = RouteParameter.Optional });

            config.Formatters.XmlFormatter.UseXmlSerializer = true;
            config.Formatters.Remove(config.Formatters.JsonFormatter);
            // config.Formatters.JsonFormatter.UseDataContractJsonSerializer = true;

            builder.UseWebApi(config);
        }
    }
}

```

This configures your default routes, and defaults to XmlSerialization over Json. Add other Middleware to this pipeline as needed (loading services, configuration settings, static files, etc.).

ASP.NET Core uses a similar approach, but doesn't rely on OWIN to handle the entry. Instead, that is done through the *Program.cs* `Main` method (similar to console applications) and `Startup` is loaded through there.

```

using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace WebApplication2
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

`Startup` must include a `Configure` method. In `Configure`, add the necessary middleware to the pipeline. In the following example (from the default web site template), several extension methods are used to configure the pipeline with support for:

- [BrowserLink](#)
- Error pages
- Static files
- ASP.NET Core MVC
- Identity

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}

```

The host and application have been decoupled, which provides the flexibility of moving to a different platform in the future.

Note: For a more in-depth reference to ASP.NET Core Startup and Middleware, see [Startup in ASP.NET Core](#)

Storing Configurations

ASP.NET supports storing settings. These settings are used, for example, to support the environment to which the applications were deployed. A common practice was to store all custom key-value pairs in the `<appSettings>` section of the *Web.config* file:

```

<appSettings>
  <add key="UserName" value="User" />
  <add key="Password" value="Password" />
</appSettings>

```

Applications read these settings using the `ConfigurationManager.AppSettings` collection in the `System.Configuration` namespace:

```

string userName = System.Web.Configuration.ConfigurationManager.AppSettings["UserName"];
string password = System.Web.Configuration.ConfigurationManager.AppSettings["Password"];

```

ASP.NET Core can store configuration data for the application in any file and load them as part of middleware bootstrapping. The default file used in the project templates is *appsettings.json*:

```

{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  // Here is where you can supply custom configuration settings, Since it is is JSON, everything is
  // represented as key: value pairs
  // Name of section is your choice
  "AppConfiguration": {
    "UserName": "UserName",
    "Password": "Password"
  }
}

```

Loading this file into an instance of `IConfiguration` inside your application is done in *Startup.cs*:

```

public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }

```

The app reads from `Configuration` to get the settings:

```

string userName = Configuration.GetSection("AppConfiguration")["UserName"];
string password = Configuration.GetSection("AppConfiguration")["Password"];

```

There are extensions to this approach to make the process more robust, such as using [Dependency Injection](#) (DI) to load a service with these values. The DI approach provides a strongly-typed set of configuration objects.

```

// Assume AppConfiguration is a class representing a strongly-typed version of AppConfiguration section
services.Configure<AppConfiguration>(Configuration.GetSection("AppConfiguration"));

```

Note: For a more in-depth reference to ASP.NET Core configuration, see [Configuration in ASP.NET Core](#).

Native Dependency Injection

An important goal when building large, scalable applications is the loose coupling of components and services. [Dependency Injection](#) is a popular technique for achieving this, and it is a native component of ASP.NET Core.

In ASP.NET applications, developers rely on a third-party library to implement Dependency Injection. One such library is [Unity](#), provided by Microsoft Patterns & Practices.

An example of setting up Dependency Injection with Unity is implementing `IDependencyResolver` that wraps a `UnityContainer` :

```

using Microsoft.Practices.Unity;
using System;
using System.Collections.Generic;
using System.Web.Http.Dependencies;

public class UnityResolver : IDependencyResolver
{
    protected IUnityContainer container;

    public UnityResolver(IUnityContainer container)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }
        this.container = container;
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return container.Resolve(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return container.ResolveAll(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return new List<object>();
        }
    }

    public IDependencyScope BeginScope()
    {
        var child = container.CreateChildContainer();
        return new UnityResolver(child);
    }

    public void Dispose()
    {
        Dispose(true);
    }

    protected virtual void Dispose(bool disposing)
    {
        container.Dispose();
    }
}

```

Create an instance of your `UnityContainer`, register your service, and set the dependency resolver of `HttpConfiguration` to the new instance of `UnityResolver` for your container:

```
public static void Register(HttpConfiguration config)
{
    var container = new UnityContainer();
    container.RegisterType<IProductRepository, ProductRepository>(new HierarchicalLifetimeManager());
    config.DependencyResolver = new UnityResolver(container);

    // Other Web API configuration not shown.
}
```

Inject `IProductRepository` where needed:

```
public class ProductsController : ApiController
{
    private IProductRepository _repository;

    public ProductsController(IProductRepository repository)
    {
        _repository = repository;
    }

    // Other controller methods not shown.
}
```

Because Dependency Injection is part of ASP.NET Core, you can add your service in the `ConfigureServices` method of *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IProductRepository, ProductRepository>();
}
```

The repository can be injected anywhere, as was true with Unity.

Note: For an in-depth reference to dependency injection in ASP.NET Core, see [Dependency Injection in ASP.NET Core](#)

Serving Static Files

An important part of web development is the ability to serve static, client-side assets. The most common examples of static files are HTML, CSS, Javascript, and images. These files need to be saved in the published location of the app (or CDN) and referenced so they can be loaded by a request. This process has changed in ASP.NET Core.

In ASP.NET, static files are stored in various directories and referenced in the views.

In ASP.NET Core, static files are stored in the "web root" (`<content root>/wwwroot`), unless configured otherwise.

The files are loaded into the request pipeline by invoking the `UseStaticFiles` extension method from

`Startup.Configure` :

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

Note: If targeting .NET Framework, install the NuGet package `Microsoft.AspNetCore.StaticFiles`.

For example, an image asset in the `wwwroot/images` folder is accessible to the browser at a location such as

```
http://<app>/images/<imageFileName>.
```

Note: For a more in-depth reference to serving static files in ASP.NET Core, see [Introduction to working with static files in ASP.NET Core](#).

Additional Resources

- [Porting Libraries to .NET Core](#)

Migrating from ASP.NET Core 1.x to ASP.NET Core 2.0

11/29/2017 • 7 min to read • [Edit Online](#)

By [Scott Addie](#)

In this article, we'll walk you through updating an existing ASP.NET Core 1.x project to ASP.NET Core 2.0. Migrating your application to ASP.NET Core 2.0 enables you to take advantage of [many new features and performance improvements](#).

Existing ASP.NET Core 1.x applications are based off of version-specific project templates. As the ASP.NET Core framework evolves, so do the project templates and the starter code contained within them. In addition to updating the ASP.NET Core framework, you need to update the code for your application.

Prerequisites

Please see [Getting Started with ASP.NET Core](#).

Update Target Framework Moniker (TFM)

Projects targeting .NET Core should use the TFM of a version greater than or equal to .NET Core 2.0. Search for the `<TargetFramework>` node in the `.csproj` file, and replace its inner text with `netcoreapp2.0`:

```
<TargetFramework>netcoreapp2.0</TargetFramework>
```

Projects targeting .NET Framework should use the TFM of a version greater than or equal to .NET Framework 4.6.1. Search for the `<TargetFramework>` node in the `.csproj` file, and replace its inner text with `net461`:

```
<TargetFramework>net461</TargetFramework>
```

NOTE

.NET Core 2.0 offers a much larger surface area than .NET Core 1.x. If you're targeting .NET Framework solely because of missing APIs in .NET Core 1.x, targeting .NET Core 2.0 is likely to work.

Update .NET Core SDK version in global.json

If your solution relies upon a [global.json](#) file to target a specific .NET Core SDK version, update its `version` property to use the 2.0 version installed on your machine:

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

Update package references

The `.csproj` file in a 1.x project lists each NuGet package used by the project.

In an ASP.NET Core 2.0 project targeting .NET Core 2.0, a single [metapackage](#) reference in the `.csproj` file replaces the collection of packages:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
</ItemGroup>
```

All the features of ASP.NET Core 2.0 and Entity Framework Core 2.0 are included in the metapackage.

ASP.NET Core 2.0 projects targeting .NET Framework should continue to reference individual NuGet packages. Update the `Version` attribute of each `<PackageReference />` node to 2.0.0.

For example, here's the list of `<PackageReference />` nodes used in a typical ASP.NET Core 2.0 project targeting .NET Framework:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.Cookies" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0"
PrivateAssets="All" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.0" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0"
PrivateAssets="All" />
</ItemGroup>
```

Update .NET Core CLI tools

In the `.csproj` file, update the `Version` attribute of each `<DotNetCliToolReference />` node to 2.0.0.

For example, here's the list of CLI tools used in a typical ASP.NET Core 2.0 project targeting .NET Core 2.0:

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

Rename Package Target Fallback property

The `.csproj` file of a 1.x project used a `PackageTargetFallback` node and variable:

```
<PackageTargetFallback>$(PackageTargetFallback);portable-net45+win8+wp8+wpa81;</PackageTargetFallback>
```

Rename both the node and variable to `AssetTargetFallback`:

```
<AssetTargetFallback>$(AssetTargetFallback);portable-net45+win8+wp8+wpa81;</AssetTargetFallback>
```

Update Main method in Program.cs

In 1.x projects, the `Main` method of `Program.cs` looked like this:

```
using System.IO;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore1App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseApplicationInsights()
                .Build();

            host.Run();
        }
    }
}
```

In 2.0 projects, the `Main` method of `Program.cs` has been simplified:

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The adoption of this new 2.0 pattern is highly recommended and is required for product features like [Entity Framework \(EF\) Core Migrations](#) to work. For example, running `Update-Database` from the Package Manager Console window or `dotnet ef database update` from the command line (on projects converted to ASP.NET Core 2.0) generates the following error:

```
Unable to create an object of type '<Context>'. Add an implementation of 'IDesignTimeDbContextFactory<Context>' to the project, or see https://go.microsoft.com/fwlink/?linkid=851728 for additional patterns supported at design time.
```

Add configuration providers

In 1.x projects, adding configuration providers to an app was accomplished via the `Startup` constructor. The steps involved creating an instance of `ConfigurationBuilder`, loading applicable providers (environment variables, app

settings, etc.), and initializing a member of `IConfigurationRoot`.

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}

public IConfiguration Configuration { get; }
```

The preceding example loads the `Configuration` member with configuration settings from `appsettings.json` as well as any `appsettings.<EnvironmentName>.json` file matching the `IHostingEnvironment.EnvironmentName` property. The location of these files is at the same path as `Startup.cs`.

In 2.0 projects, the boilerplate configuration code inherent to 1.x projects runs behind-the-scenes. For example, environment variables and app settings are loaded at startup. The equivalent `Startup.cs` code is reduced to `IConfiguration` initialization with the injected instance:

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

To remove the default providers added by `WebHostBuilder.CreateDefaultBuilder`, invoke the `Clear` method on the `IConfigurationBuilder.Sources` property inside of `ConfigureAppConfiguration`. To add providers back, utilize the `ConfigureAppConfiguration` method in `Program.cs`:

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureAppConfiguration((hostContext, config) =>
        {
            // delete all default configuration providers
            config.Sources.Clear();
            config.AddJsonFile("myconfig.json", optional: true);
        })
        .Build();
```

The configuration used by the `CreateDefaultBuilder` method in the preceding code snippet can be seen [here](#).

For more information, see [Configuration in ASP.NET Core](#).

Move database initialization code

In 1.x projects using EF Core 1.x, a command such as `dotnet ef migrations add` does the following:

1. Instantiates a `Startup` instance
2. Invokes the `ConfigureServices` method to register all services with dependency injection (including `DbContext` types)
3. Performs its requisite tasks

In 2.0 projects using EF Core 2.0, `Program.BuildWebHost` is invoked to obtain the application services. Unlike 1.x, this has the additional side effect of invoking `Startup.Configure`. If your 1.x app invoked database initialization code in its `Configure` method, unexpected problems can occur. For example, if the database doesn't yet exist, the seeding code runs before the EF Core Migrations command execution. This problem causes a `dotnet ef migrations list` command to fail if the database doesn't yet exist.

Consider the following 1.x seed initialization code in the `Configure` method of `Startup.cs`:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

SeedData.Initialize(app.ApplicationServices);
```

In 2.0 projects, move the `SeedData.Initialize` call to the `Main` method of `Program.cs`:

```
var host = BuildWebHost(args);

using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    try
    {
        // Requires using RazorPagesMovie.Models;
        SeedData.Initialize(services);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}

host.Run();
```

As of 2.0, it's bad practice to do anything in `BuildWebHost` except build and configure the web host. Anything that is about running the application should be handled outside of `BuildWebHost` — typically in the `Main` method of `Program.cs`.

Review your Razor View Compilation setting

Faster application startup time and smaller published bundles are of utmost importance to you. For these reasons, [Razor view compilation](#) is enabled by default in ASP.NET Core 2.0.

Setting the `MvcRazorCompileOnPublish` property to true is no longer required. Unless you're disabling view

compilation, the property may be removed from the `.csproj` file.

When targeting .NET Framework, you still need to explicitly reference the [Microsoft.AspNetCore.Mvc.Razor.ViewCompilation](#) NuGet package in your `.csproj` file:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0" PrivateAssets="All" />
```

Rely on Application Insights "Light-Up" features

Effortless setup of application performance instrumentation is important. You can now rely on the new [Application Insights](#) "light-up" features available in the Visual Studio 2017 tooling.

ASP.NET Core 1.1 projects created in Visual Studio 2017 added Application Insights by default. If you're not using the Application Insights SDK directly, outside of `Program.cs` and `Startup.cs`, follow these steps:

1. If targeting .NET Core, remove the following `<PackageReference />` node from the `.csproj` file:

```
<PackageReference Include="Microsoft.ApplicationInsights.AspNetCore" Version="2.0.0" />
```

2. If targeting .NET Core, remove the `UseApplicationInsights` extension method invocation from `Program.cs`:

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    host.Run();
}
```

3. Remove the Application Insights client-side API call from `_Layout.cshtml`. It comprises the following two lines of code:

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
@Html.Raw(JavaScriptSnippet.FullScript)
```

If you are using the Application Insights SDK directly, continue to do so. The 2.0 [metapackage](#) includes the latest version of Application Insights, so a package downgrade error appears if you're referencing an older version.

Adopt Authentication / Identity Improvements

ASP.NET Core 2.0 has a new authentication model and a number of significant changes to ASP.NET Core Identity. If you created your project with Individual User Accounts enabled, or if you have manually added authentication or Identity, see [Migrating Authentication and Identity to ASP.NET Core 2.0](#).

Additional Resources

- [Breaking Changes in ASP.NET Core 2.0](#)

Migrating Authentication and Identity to ASP.NET Core 2.0

10/28/2017 • 8 min to read • [Edit Online](#)

By [Scott Addie](#) and [Hao Kung](#)

ASP.NET Core 2.0 has a new model for authentication and [Identity](#) which simplifies configuration by using services. ASP.NET Core 1.x applications that use authentication or Identity can be updated to use the new model as outlined below.

Authentication Middleware and Services

In 1.x projects, authentication is configured via middleware. A middleware method is invoked for each authentication scheme you want to support.

The following 1.x example configures Facebook authentication with Identity in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseIdentity();
    app.UseFacebookAuthentication(new FacebookOptions {
        AppId = Configuration["auth:facebook:appid"],
        AppSecret = Configuration["auth:facebook:appsecret"]
    });
}
```

In 2.0 projects, authentication is configured via services. Each authentication scheme is registered in the `ConfigureServices` method of *Startup.cs*. The `UseIdentity` method is replaced with `UseAuthentication`.

The following 2.0 example configures Facebook authentication with Identity in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    // If you want to tweak Identity cookies, they're no longer part of IdentityOptions.
    services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
    services.AddAuthentication()
        .AddFacebook(options =>
        {
            options.AppId = Configuration["auth:facebook:appid"];
            options.AppSecret = Configuration["auth:facebook:appsecret"];
        });
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory) {
    app.UseAuthentication();
}
```

The `UseAuthentication` method adds a single authentication middleware component which is responsible for automatic authentication and the handling of remote authentication requests. It replaces all of the individual middleware components with a single, common middleware component.

Below are 2.0 migration instructions for each major authentication scheme.

Cookie-based Authentication

Select one of the two options below, and make the necessary changes in *Startup.cs*:

1. Use cookies with Identity

- Replace `UseIdentity` with `UseAuthentication` in the `Configure` method:

```
app.UseAuthentication();
```

- Invoke the `AddIdentity` method in the `ConfigureServices` method to add the cookie authentication services.
- Optionally, invoke the `ConfigureApplicationCookie` or `ConfigureExternalCookie` method in the `ConfigureServices` method to tweak the Identity cookie settings.

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
```

2. Use cookies without Identity

- Replace the `UseCookieAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddAuthentication` and `AddCookie` methods in the `ConfigureServices` method:

```
// If you don't want the cookie to be automatically authenticated and assigned to
HttpContext.User,
// remove the CookieAuthenticationDefaults.AuthenticationScheme parameter passed to
AddAuthentication.
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/Account/LogIn";
        options.LogoutPath = "/Account/LogOff";
    });
```

JWT Bearer Authentication

Make the following changes in *Startup.cs*:

- Replace the `UseJwtBearerAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddJwtBearer` method in the `ConfigureServices` method:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.Audience = "http://localhost:5001/";
        options.Authority = "http://localhost:5000/";
    });
```

This code snippet doesn't use Identity, so the default scheme should be set by passing

`JwtBearerDefaults.AuthenticationScheme` to the `AddAuthentication` method.

OpenID Connect (OIDC) Authentication

Make the following changes in *Startup.cs*:

- Replace the `UseOpenIdConnectAuthentication` method call in the `Configure` method with `UseAuthentication` :

```
app.UseAuthentication();
```

- Invoke the `AddOpenIdConnect` method in the `ConfigureServices` method:

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.Authority = Configuration["auth:oidc:authority"];
    options.ClientId = Configuration["auth:oidc:clientid"];
});
```

Facebook Authentication

Make the following changes in *Startup.cs*:

- Replace the `UseFacebookAuthentication` method call in the `Configure` method with `UseAuthentication` :

```
app.UseAuthentication();
```

- Invoke the `AddFacebook` method in the `ConfigureServices` method:

```
services.AddAuthentication()
    .AddFacebook(options =>
    {
        options.AppId = Configuration["auth:facebook:appid"];
        options.AppSecret = Configuration["auth:facebook:appsecret"];
    });
```

Google Authentication

Make the following changes in *Startup.cs*:

- Replace the `UseGoogleAuthentication` method call in the `Configure` method with `UseAuthentication` :

```
app.UseAuthentication();
```

- Invoke the `AddGoogle` method in the `ConfigureServices` method:

```
services.AddAuthentication()
    .AddGoogle(options =>
    {
        options.ClientId = Configuration["auth:google:clientid"];
        options.ClientSecret = Configuration["auth:google:clientsecret"];
    });
```

Microsoft Account Authentication

Make the following changes in *Startup.cs*:

- Replace the `UseMicrosoftAccountAuthentication` method call in the `Configure` method with `UseAuthentication` :

```
app.UseAuthentication();
```

- Invoke the `AddMicrosoftAccount` method in the `ConfigureServices` method:

```
services.AddAuthentication()
    .AddMicrosoftAccount(options =>
    {
        options.ClientId = Configuration["auth:microsoft:clientid"];
        options.ClientSecret = Configuration["auth:microsoft:clientsecret"];
    });
```

Twitter Authentication

Make the following changes in *Startup.cs*:

- Replace the `UseTwitterAuthentication` method call in the `Configure` method with `UseAuthentication` :

```
app.UseAuthentication();
```

- Invoke the `AddTwitter` method in the `ConfigureServices` method:

```
services.AddAuthentication()
    .AddTwitter(options =>
    {
        options.ConsumerKey = Configuration["auth:twitter:consumerkey"];
        options.ConsumerSecret = Configuration["auth:twitter:consumersecret"];
    });
```

Setting Default Authentication Schemes

In 1.x, the `AutomaticAuthenticate` and `AutomaticChallenge` properties of the `AuthenticationOptions` base class were intended to be set on a single authentication scheme. There was no good way to enforce this.

In 2.0, these two properties have been removed as properties on the individual `AuthenticationOptions` instance. They can be configured in the `AddAuthentication` method call within the `ConfigureServices` method of *Startup.cs*:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme);
```

In the preceding code snippet, the default scheme is set to `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookies").

Alternatively, use an overloaded version of the `AddAuthentication` method to set more than one property. In the

following overloaded method example, the default scheme is set to

`CookieAuthenticationDefaults.AuthenticationScheme`. The authentication scheme may alternatively be specified within your individual `[Authorize]` attributes or authorization policies.

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
});
```

Define a default scheme in 2.0 if one of the following conditions is true:

- You want the user to be automatically signed in
- You use the `[Authorize]` attribute or authorization policies without specifying schemes

An exception to this rule is the `AddIdentity` method. This method adds cookies for you and sets the default authenticate and challenge schemes to the application cookie `IdentityConstants.ApplicationScheme`. Additionally, it sets the default sign-in scheme to the external cookie `IdentityConstants.ExternalScheme`.

Use HttpContext Authentication Extensions

The `IAuthenticationManager` interface is the main entry point into the 1.x authentication system. It has been replaced with a new set of `HttpContext` extension methods in the `Microsoft.AspNetCore.Authentication` namespace.

For example, 1.x projects reference an `Authentication` property:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

In 2.0 projects, import the `Microsoft.AspNetCore.Authentication` namespace, and delete the `Authentication` property references:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Windows Authentication (HTTP.sys / IISIntegration)

There are two variations of Windows authentication:

1. The host only allows authenticated users
2. The host allows both anonymous and authenticated users

The first variation described above is unaffected by the 2.0 changes.

The second variation described above is affected by the 2.0 changes. As an example, you may be allowing anonymous users into your application at the IIS or [HTTP.sys](#) layer but authorizing users at the Controller level. In this scenario, set the default scheme to `IISDefaults.AuthenticationScheme` in the `ConfigureServices` method of `Startup.cs`:

```
services.AddAuthentication(IISDefaults.AuthenticationScheme);
```

Failure to set the default scheme accordingly prevents the authorize request to challenge from working.

IdentityCookieOptions Instances

A side effect of the 2.0 changes is the switch to using named options instead of cookie options instances. The ability to customize the Identity cookie scheme names is removed.

For example, 1.x projects use [constructor injection](#) to pass an `IdentityCookieOptions` parameter into `AccountController.cs`. The external cookie authentication scheme is accessed from the provided instance:

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IOptions<IdentityCookieOptions> identityCookieOptions,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _externalCookieScheme = identityCookieOptions.Value.ExternalCookieAuthenticationScheme;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

The aforementioned constructor injection becomes unnecessary in 2.0 projects, and the `_externalCookieScheme` field can be deleted:

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

The `IdentityConstants.ExternalScheme` constant can be used directly:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Add IdentityUser POCO Navigation Properties

The Entity Framework (EF) Core navigation properties of the base `IdentityUser` POCO (Plain Old CLR Object) have been removed. If your 1.x project used these properties, manually add them back to the 2.0 project:

```

/// <summary>
/// Navigation property for the roles this user belongs to.
/// </summary>
public virtual ICollection<IdentityUserRole<int>> Roles { get; } = new List<IdentityUserRole<int>>();

/// <summary>
/// Navigation property for the claims this user possesses.
/// </summary>
public virtual ICollection<IdentityUserClaim<int>> Claims { get; } = new List<IdentityUserClaim<int>>();

/// <summary>
/// Navigation property for this users login accounts.
/// </summary>
public virtual ICollection<IdentityUserLogin<int>> Logins { get; } = new List<IdentityUserLogin<int>>();

```

To prevent duplicate foreign keys when running EF Core Migrations, add the following to your `IdentityDbContext` class' `OnModelCreating` method (after the `base.OnModelCreating()` call):

```

protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
    // Customize the ASP.NET Identity model and override the defaults if needed.
    // For example, you can rename the ASP.NET Identity table names and more.
    // Add your customizations after calling base.OnModelCreating(builder);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Claims)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .onDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Logins)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .onDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Roles)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .onDelete(DeleteBehavior.Cascade);
}

```

Replace GetExternalAuthenticationSchemes

The synchronous method `GetExternalAuthenticationSchemes` was removed in favor of an asynchronous version. 1.x projects have the following code in `ManageController.cs`:

```

var otherLogins = _signInManager.GetExternalAuthenticationSchemes().Where(auth => userLogins.All(u1 =>
auth.AuthenticationScheme != u1.LoginProvider)).ToList();

```

This method appears in `Login.cshtml` too:

```

var loginProviders = SignInManager.GetExternalAuthenticationSchemes().ToList();
<div>
  <p>
    @foreach (var provider in loginProviders)
    {
      <button type="submit" class="btn btn-default" name="provider"
value="@provider.AuthenticationScheme" title="Log in using your @provider.DisplayName
account">@provider.AuthenticationScheme</button>
    }
  </p>
</div>
</form>
}

```

In 2.0 projects, use the `GetExternalAuthenticationSchemesAsync` method:

```

var schemes = await _signInManager.GetExternalAuthenticationSchemesAsync();
var otherLogins = schemes.Where(auth => userLogins.All(ul => auth.Name != ul.LoginProvider)).ToList();

```

In `Login.cshtml`, the `AuthenticationScheme` property accessed in the `foreach` loop changes to `Name` :

```

var loginProviders = (await SignInManager.GetExternalAuthenticationSchemesAsync()).ToList();
<div>
  <p>
    @foreach (var provider in loginProviders)
    {
      <button type="submit" class="btn btn-default" name="provider" value="@provider.Name"
title="Log in using your @provider.DisplayName account">@provider.DisplayName</button>
    }
  </p>
</div>
</form>
}

```

ManageLoginsViewModel Property Change

A `ManageLoginsViewModel` object is used in the `ManageLogins` action of `ManageController.cs`. In 1.x projects, the object's `OtherLogins` property return type is `IList<AuthenticationDescription>`. This return type requires an import of `Microsoft.AspNetCore.Http.Authentication` :

```

using System.Collections.Generic;
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore1App.Models.ManageViewModels
{
  public class ManageLoginsViewModel
  {
    public IList<UserLoginInfo> CurrentLogins { get; set; }

    public IList<AuthenticationDescription> OtherLogins { get; set; }
  }
}

```

In 2.0 projects, the return type changes to `IList<AuthenticationScheme>`. This new return type requires replacing the `Microsoft.AspNetCore.Http.Authentication` import with a `Microsoft.AspNetCore.Authentication` import.

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore2App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationScheme> OtherLogins { get; set; }
    }
}
```

Additional Resources

For additional details and discussion, see the [Discussion for Auth 2.0](#) issue on GitHub.

What's new in ASP.NET Core 2.0

1/10/2018 • 5 min to read • [Edit Online](#)

This article highlights the most significant changes in ASP.NET Core 2.0, with links to relevant documentation.

Razor Pages

Razor Pages is a new feature of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

For more information, see the introduction and tutorial:

- [Introduction to Razor Pages](#)
- [Getting started with Razor Pages](#)

ASP.NET Core metapackage

A new ASP.NET Core metapackage includes all of the packages made and supported by the ASP.NET Core and Entity Framework Core teams, along with their internal and 3rd-party dependencies. You no longer need to choose individual ASP.NET Core features by package. All features are included in the [Microsoft.AspNetCore.All](#) package. The default templates use this package.

For more information, see [Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0](#).

Runtime Store

Applications that use the `Microsoft.AspNetCore.All` metapackage automatically take advantage of the new .NET Core Runtime Store. The Store contains all the runtime assets needed to run ASP.NET Core 2.0 applications. When you use the `Microsoft.AspNetCore.All` metapackage, no assets from the referenced ASP.NET Core NuGet packages are deployed with the application because they already reside on the target system. The assets in the Runtime Store are also precompiled to improve application startup time.

For more information, see [Runtime store](#)

.NET Standard 2.0

The ASP.NET Core 2.0 packages target .NET Standard 2.0. The packages can be referenced by other .NET Standard 2.0 libraries, and they can run on .NET Standard 2.0-compliant implementations of .NET, including .NET Core 2.0 and .NET Framework 4.6.1.

The `Microsoft.AspNetCore.All` metapackage targets .NET Core 2.0 only, because it is intended to be used with the .NET Core 2.0 Runtime Store.

Configuration update

An `IConfiguration` instance is added to the services container by default in ASP.NET Core 2.0. `IConfiguration` in the services container makes it easier for applications to retrieve configuration values from the container.

For information about the status of planned documentation, see the [GitHub issue](#).

Logging update

In ASP.NET Core 2.0, logging is incorporated into the dependency injection (DI) system by default. You add providers and configure filtering in the *Program.cs* file instead of in the *Startup.cs* file. And the default `ILoggerFactory` supports filtering in a way that lets you use one flexible approach for both cross-provider filtering and specific-provider filtering.

For more information, see [Introduction to Logging](#).

Authentication update

A new authentication model makes it easier to configure authentication for an application using DI.

New templates are available for configuring authentication for web apps and web APIs using [Azure AD B2C](#).

For information about the status of planned documentation, see the [GitHub issue](#).

Identity update

We've made it easier to build secure web APIs using Identity in ASP.NET Core 2.0. You can acquire access tokens for accessing your web APIs using the [Microsoft Authentication Library \(MSAL\)](#).

For more information on authentication changes in 2.0, see the following resources:

- [Account confirmation and password recovery in ASP.NET Core](#)
- [Enabling QR Code generation for authenticator apps in ASP.NET Core](#)
- [Migrating Authentication and Identity to ASP.NET Core 2.0](#)

SPA templates

Single Page Application (SPA) project templates for Angular, Aurelia, Knockout.js, React.js, and React.js with Redux are available. The Angular template has been updated to Angular 4. The Angular and React templates are available by default; for information about how to get the other templates, see [Creating a new SPA project](#). For information about how to build a SPA in ASP.NET Core, see [Using JavaScriptServices for Creating Single Page Applications](#).

Kestrel improvements

The Kestrel web server has new features that make it more suitable as an Internet-facing server. We've added a number of server constraint configuration options in the `KestrelServerOptions` class's new `Limits` property. You can now add limits for the following:

- Maximum client connections
- Maximum request body size
- Minimum request body data rate

For more information, see [Kestrel web server implementation in ASP.NET Core](#).

WebListener renamed to HTTP.sys

The packages `Microsoft.AspNetCore.Server.WebListener` and `Microsoft.Net.Http.Server` have been merged into a new package `Microsoft.AspNetCore.Server.HttpSys`. The namespaces have been updated to match.

For more information, see [HTTP.sys web server implementation in ASP.NET Core](#).

Enhanced HTTP header support

When using MVC to transmit a `FileStreamResult` or a `FileContentResult`, you now have the option to set an `ETag` or a `LastModified` date on the content you transmit. You can set these values on the returned content with code

similar to the following:

```
var data = Encoding.UTF8.GetBytes("This is a sample text from a binary array");
var entityTag = new EntityTagHeaderValue("\"MyCalculatedEtagValue\"");
return File(data, "text/plain", "downloadName.txt", lastModified: DateTime.UtcNow.AddSeconds(-5), entityTag:
entityTag);
```

The file returned to your visitors will be decorated with the appropriate HTTP headers for the `Etag` and `LastModified` values.

If an application visitor requests content with a Range Request header, ASP.NET will recognize that and handle that header. If the requested content can be partially delivered, ASP.NET will appropriately skip and return just the requested set of bytes. You do not need to write any special handlers into your methods to adapt or handle this feature; it is automatically handled for you.

Hosting startup and Application Insights

Hosting environments can now inject extra package dependencies and execute code during application startup, without the application needing to explicitly take a dependency or call any methods. This feature can be used to enable certain environments to "light-up" features unique to that environment without the application needing to know ahead of time.

In ASP.NET Core 2.0, this feature is used to automatically enable Application Insights diagnostics when debugging in Visual Studio and (after opting in) when running in Azure App Services. As a result, the project templates no longer add Application Insights packages and code by default.

For information about the status of planned documentation, see the [GitHub issue](#).

Automatic use of anti-forgery tokens

ASP.NET Core has always helped HTML-encode your content by default, but with the new version we're taking an extra step to help prevent cross-site request forgery (XSRF) attacks. ASP.NET Core will now emit anti-forgery tokens by default and validate them on form POST actions and pages without extra configuration.

For more information, see [Preventing Cross-Site Request Forgery \(XSRF/CSRF\) Attacks in ASP.NET Core](#).

Automatic precompilation

Razor view pre-compilation is enabled during publish by default, reducing the publish output size and application startup time.

Razor support for C# 7.1

The Razor view engine has been updated to work with the new Roslyn compiler. That includes support for C# 7.1 features like Default Expressions, Inferred Tuple Names, and Pattern-Matching with Generics. To use C# 7.1 in your project, add the following property in your project file and then reload the solution:

```
<LangVersion>latest</LangVersion>
```

For information about the status of C# 7.1 features, see [the Roslyn GitHub repository](#).

Other documentation updates for 2.0

- [Visual Studio publish profiles for ASP.NET Core app deployment](#)

- [Key Management](#)
- [Configuring Facebook authentication](#)
- [Configuring Twitter authentication](#)
- [Configuring Google authentication](#)
- [Configuring Microsoft Account authentication](#)

Migration guidance

For guidance on how to migrate ASP.NET Core 1.x applications to ASP.NET Core 2.0, see the following resources:

- [Migrating from ASP.NET Core 1.x to ASP.NET Core 2.0](#)
- [Migrating Authentication and Identity to ASP.NET Core 2.0](#)

Additional Information

For the complete list of changes, see the [ASP.NET Core 2.0 Release Notes](#).

If you'd like to connect with the ASP.NET Core development team's progress and plans, tune in to the weekly [ASP.NET Community Standup](#).

What's new in ASP.NET Core 1.1

11/7/2017 • 1 min to read • [Edit Online](#)

ASP.NET Core 1.1 includes the following new features:

- [URL Rewriting Middleware](#)
- [Response Caching Middleware](#)
- [View Components as Tag Helpers](#)
- [Middleware as MVC filters](#)
- [Cookie-based TempData provider](#)
- [Azure App Service logging provider](#)
- [Azure Key Vault configuration provider](#)
- [Azure and Redis Storage Data Protection Key Repositories](#)
- [WebListener Server for Windows](#)
- [WebSockets support](#)

Choosing between versions 1.0 and 1.1 of ASP.NET Core

ASP.NET Core 1.1 has more features than 1.0. In general, we recommend you use the latest version.

Additional Information

- [ASP.NET Core 1.1.0 Release Notes](#)
- If you'd like to connect with the ASP.NET Core development team's progress and plans, tune in to the weekly [ASP.NET Community Standup](#).