ASP.NET Core Web Host

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. This topic covers the ASP.NET Core Web Host (IWebHostBuilder), which is useful for hosting web apps. For coverage of the .NET Generic Host (IHostBuilder), see .NET Generic Host.

Set up a host

Create a host using an instance of WebHostBuilder. Creating a host is typically performed in the app's entry point, the `Main` method. In the project templates, `Main` is located in *Program.cs*:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

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

`WebHostBuilder` requires a server that implements IServer. The built-in servers are Kestrel and HTTP.sys (prior to the release of ASP.NET Core 2.0, HTTP.sys was called WebListener). In this example, the UseKestrel extension method specifies the Kestrel server.

The *content root* determines where the host searches for content files, such as MVC view files. The default content root is obtained for `UseContentRoot` by Directory.GetCurrentDirectory. 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.

To use IIS as a reverse proxy, call UseIISIntegration as part of building the host. `UseIISIntegration` doesn't configure a *server*, like UseKestrel does. `UseIISIntegration` configures the base path and port the server listens on when using the ASP.NET Core Module to create a reverse proxy between Kestrel and IIS. To use IIS with ASP.NET Core, `UseKestrel` and `UseIISIntegration` must be specified. `UseIISIntegration` only activates when running behind IIS or IIS Express. For more information, see ASP.NET Core Module and ASP.NET Core Module configuration reference.

A minimal implementation that configures a host (and an ASP.NET Core app) includes specifying a server and configuration of the app's request pipeline:

C#
```
var host = new WebHostBuilder()
    .UseKestrel()
    .Configure(app =>
    {
        app.Run(context => context.Response.WriteAsync("Hello World!"));
    })
    .Build();

host.Run();
```

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 App 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_ENVIRONMENT`.
- Extensions such as UseContentRoot and UseConfiguration (see the Override configuration section).
- 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 Override configuration in the next section.

Application Key (Name)

The IHostingEnvironment.ApplicationName property is automatically set when UseStartup or Configure is called during host construction. The value is set to the name of the assembly containing the app's entry point. To set the value explicitly, use the WebHostDefaults.ApplicationKey:

Key: applicationName
Type: *string*
Default: The name of the assembly containing the app's entry point.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_APPLICATIONNAME`

C#

```
var host = new WebHostBuilder()
    .UseSetting("applicationName", "CustomApplicationName")
```

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.

C#
```
var host = new WebHostBuilder()
    .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.

C#
```
var host = new WebHostBuilder()
    .UseContentRoot("c:\\<content-root>")
```

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.

C#
```
var host = new WebHostBuilder()
    .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 Use multiple environments in ASP.NET Core.

C#
```
var host = new WebHostBuilder()
    .UseEnvironment(EnvironmentName.Development)
```

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.

C#
```
var host = new WebHostBuilder()
    .UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")
```

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.

C#
```
var host = new WebHostBuilder()
    .UseStartup("StartupAssemblyName")
```
C#
```
var host = new WebHostBuilder()
    .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`

C#
```
var host = new WebHostBuilder()
    .UseWebRoot("public")
```

Override configuration

Use Configuration to configure the web host. In the following example, host configuration is optionally specified in a *hostsettings.json* file. Any configuration loaded from the *hostsettings.json* file may be overridden by command-line arguments. The built configuration (in `config`) is used to configure the host with UseConfiguration. `IWebHostBuilder` configuration is added to the app's configuration, but the converse isn't true—`ConfigureAppConfiguration` doesn't affect the `IWebHostBuilder` configuration.

Overriding the configuration provided by `UseUrls` with *hostsettings.json* config first, command-line argument config second:

C#
```
public class Program
{
    public static void Main(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("hostsettings.json", optional: true)
            .AddCommandLine(args)
            .Build();

        var host = new WebHostBuilder()
            .UseUrls("http://*:5000")
            .UseConfiguration(config)
            .UseKestrel()
            .Configure(app =>
            {
                app.Run(context =>
                    context.Response.WriteAsync("Hello, World!"));
            })
            .Build();

        host.Run();
    }
}
```

hostsettings.json:

JSON
```
{
    urls: "http://*:5005"
}
```

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.

`UseConfiguration` only copies keys from the provided `IConfiguration` to the host builder configuration. Therefore, setting `reloadOnChange: true` for JSON, INI, and XML settings files has no effect.

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 *hostsettings.json* file, and the server listens on port 8080:

console
```
dotnet run --urls "http://*:8080"
```

Manage the host

Run

The `Run` method starts the web app and blocks the calling thread until the host is shut down:

C#
```
host.Run();
```

Start

Run the host in a non-blocking manner by calling its `Start` method:

C#
```
using (host)
{
    host.Start();
    Console.ReadLine();
}
```

If a list of URLs is passed to the `Start` method, it listens on the URLs specified:

C#
```
var urls = new List<string>()
{
    "http://*:5000",
    "http://localhost:5001"
};

var host = new WebHostBuilder()
    .UseKestrel()
    .UseStartup<Startup>()
    .Start(urls.ToArray());

using (host)
{
    Console.ReadLine();
}
```

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:

C#
```
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`:

C#
```
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. For more information, see Use multiple environments in ASP.NET Core.

The `IHostingEnvironment` service can also be injected directly into the `Configure` method for setting up the processing pipeline:

C#
```
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:

C#
```
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.

Cancellation Token	Triggered when…
ApplicationStarted	The host has fully started.
ApplicationStopped	The host is completing a graceful shutdown. All requests should be processed. Shutdown blocks until this event completes.
ApplicationStopping	The host is performing a graceful shutdown. Requests may still be processing. Shutdown blocks until this event completes.

C#

```
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
    }
}
```

StopApplication requests termination of the app. The following class uses `StopApplication` to gracefully shut down an app when the class's `Shutdown` method is called:

C#

```
public class MyClass
{
    private readonly IApplicationLifetime _appLifetime;

    public MyClass(IApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
```

```
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

Scope validation

When `ValidateScopes` is set to `true`, the default service provider performs checks to verify that:

- Scoped services aren't directly or indirectly resolved from the root service provider.
- Scoped services aren't directly or indirectly injected into singletons.

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app/server's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton because it's only disposed by the root container when app/server is shut down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

To always validate scopes, including in the Production environment, configure the ServiceProviderOptions with UseDefaultServiceProvider on the host builder:

C#
```
WebHost.CreateDefaultBuilder(args)
    .UseDefaultServiceProvider((context, options) => {
        options.ValidateScopes = true;
    })
```

Additional resources

- Host ASP.NET Core on Windows with IIS
- Host ASP.NET Core on Linux with Nginx
- Host ASP.NET Core on Linux with Apache
- Host ASP.NET Core in a Windows Service