Routing in ASP.NET Core

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 app and configured when the app starts. 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 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 find the URL corresponding to a given route handler based on route handler information.

Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful. The default conventional route `{controller=Home}/{action=Index}/{id?}`:

- Supports a basic and descriptive routing scheme:
- Is a good starting point for web apps intended for use by browsers.

It's common to add additional terse routes to high traffic areas of the app in specialized situations (for example, blog, ecommerce) using attribute routing or dedicated conventional routes.

Web APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs. This means that many operations (for example, GET, POST) on the same logical resource will use the same URL. Attribute routing provides a level of control that is needed to carefully design an API's URL-space.

MVC's URL generation support allows the app to be developed without hard coding the URLs to link the app together. This allows for starting with a basic routing configuration, and modifying the routes after the app's shape is determined.

Important

This document covers low-level ASP.NET Core routing. For information on ASP.NET Core MVC routing, see **Routing to controller actions in ASP.NET Core**.

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 Core MVC adds routing to the middleware pipeline as part of its configuration. To learn about using routing as a standalone component, see Use Routing Middleware section.

URL matching

URL matching is the process by which routing dispatches an incoming request to a *handler*. This process is 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 app.

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 is 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 set after a route matches.

A match during `RouteAsync` also sets the properties of the `RouteContext.RouteData` to appropriate values based on the request processing performed thus far. If a route matches a request, the `RouteContext.RouteData` contains 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 app.

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 that the app 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 `RouteData.DataTokens` can be of any type, in contrast to `RouteData.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 of one another. 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 it starts with user or framework code calling into the GetVirtualPath method of the route collection. Each *route* then has 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 whether it's 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 obtain services or additional data associated with the current context.

Tip

Think of **VirtualPathContext.Values** as being a set of overrides for the **VirtualPathContext.AmbientValues**. URL generation attempts 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`. `VirtualPathData` contains the `VirtualPath` for the output URL and 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. If you want to render the generated URL in HTML, prepend the base path of the app.

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 match against the URL path when RouteAsync is called. `Route` uses the same route template to generate a URL when `GetVirtualPath` is called.

Most apps create routes by calling MapRoute or one of the similar extension methods defined on IRouteBuilder. All of these methods create an instance of Route and add it to the route collection.

`MapRoute` doesn't take a route handler parameter. `MapRoute` only adds routes that are handled by the DefaultHandler. Since the default handler is an `IRouter`, it may decide not to handle the request. For example, ASP.NET Core 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 in ASP.NET Core.

The following code example is an example of a `MapRoute` call used by a typical ASP.NET Core MVC route definition:
C#Copy
```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

This template matches a URL path like `/Products/Details/17` and extracts 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're defined by enclosing the parameter name in braces `{ ... }`.

The preceding template could also match the URL path `/` and would produce the values `{ controller = Home, action = Index }`. This occurs 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 don't 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*:
C#Copy
```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id:int}");
```

This template matches 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:
C#Copy
```
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

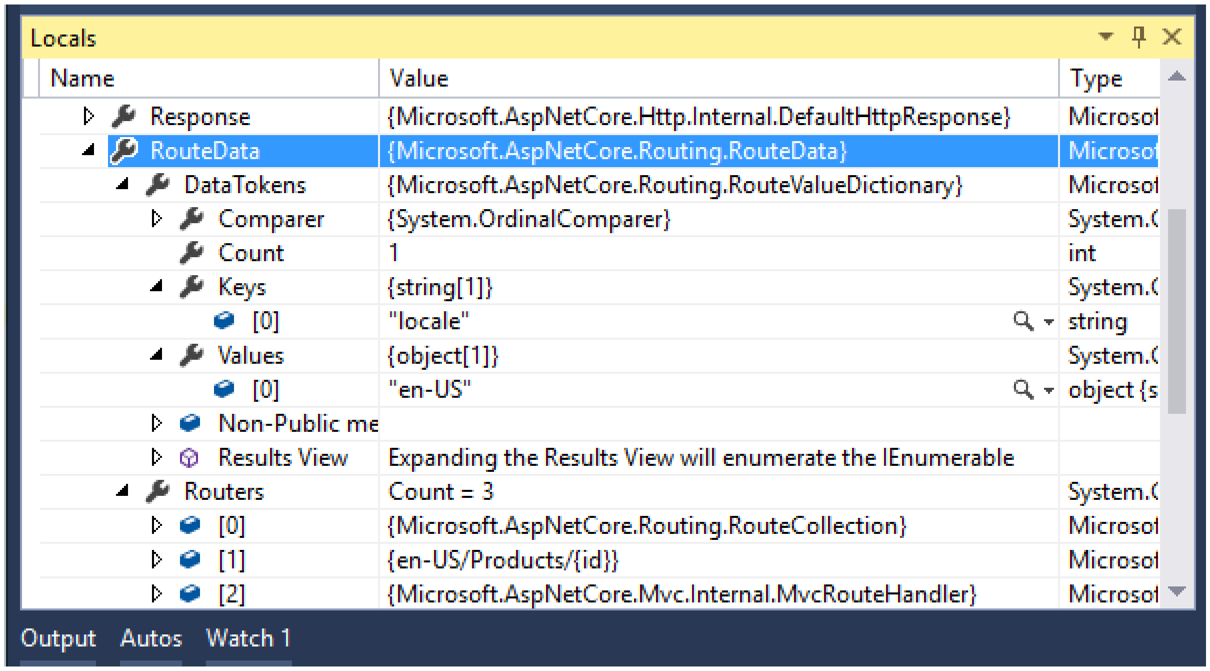
The inline syntax for defining constraints and defaults can be convenient for simple routes. However, there are features, such as data tokens, that aren't supported by inline syntax.

The following example demonstrates a few more scenarios:
C#Copy
```
routes.MapRoute(
    name: "blog",
    template: "Blog/{*article}",
    defaults: new { controller = "Blog", action = "ReadArticle" });
```

This template matches a URL path like `/Blog/All-About-Routing/Introduction` and extracts 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:
C#Copy
```
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 matches a URL path like `/en-US/Products/5` and extracts the values `{ controller = Products, action = Details, id = 5 }` and the data tokens `{ locale = en-US }`.



```
Locals
Name                      Value                                                        Type
▷ Response                {Microsoft.AspNetCore.Http.Internal.DefaultHttpResponse}    Microsof
◢ RouteData               {Microsoft.AspNetCore.Routing.RouteData}                     Microsof
  ◢ DataTokens            {Microsoft.AspNetCore.Routing.RouteValueDictionary}          Microsof
    ▷ Comparer            {System.OrdinalComparer}                                     System.(
      Count               1                                                            int
    ◢ Keys                {string[1]}                                                  System.(
        [0]               "locale"                                                     string
    ◢ Values              {object[1]}                                                  System.(
        [0]               "en-US"                                                      object {s
    ▷ Non-Public me
    ▷ Results View        Expanding the Results View will enumerate the IEnumerable
  ◢ Routers               Count = 3                                                    System.(
    ▷ [0]                 {Microsoft.AspNetCore.Routing.RouteCollection}               Microsof
    ▷ [1]                 {en-US/Products/{id}}                                        Microsof
    ▷ [2]                 {Microsoft.AspNetCore.Mvc.Internal.MvcRouteHandler}          Microsof
Output   Autos   Watch 1
```

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 Core MVC style route:

C#Copy

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

With the route values `{ controller = Products, action = List }`, this route generates 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 generates the URL `/`. The route values that were provided match the default values so that the segments corresponding to those values can be safely omitted. Both URLs generated 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 Core MVC should use **UrlHelper** to generate URLs instead of calling into routing directly.

For more information about URL generation, see url-generation-reference.

Use Routing Middleware

Reference the Microsoft.AspNetCore.Routing in the app's project file.

Add routing to the service container in `Startup.ConfigureServices`:

C#Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRouting();
}
```

Routes must be configured in the `Startup.Configure` method. The sample app uses these APIs:

- `RouteBuilder`
- `Build`
- `MapGet` – Matches only HTTP GET requests.
- `UseRouter`

C#Copy

```
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");
    // 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 following table 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're configuring a single route, call UseRouter passing in an `IRouter` instance. You won't need to use 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` is used as the *route handler* when the route matches. Other methods in this family allow configuring a middleware pipeline for use as the route handler. If the *Map* method doesn't accept a handler, such as `MapRoute`, then it uses 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* that are 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}` isn't a valid route, since there's 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 URLs 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, consider the template `files/{filename}.{ext?}`. When both `filename` and `ext` exist, both values are populated. If only `filename` exists in the URL, the route matches because the trailing period `.` is optional. The following URLs match this route:

- `/files/myFile.txt`
- `/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}` matches any URI that starts with `/blog` and had any value following it (which is 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 equals sign (`=`). For example, `{controller=Home}` defines `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 question mark (`?`) to the end of the parameter name, as in `id?`. The difference between optional values and default route parameters is that a route parameter with a default value always produces a value; an optional parameter has a value only when a value is provided by the request URL.

Route parameters may 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, they're enclosed in parentheses `( )` after the constraint name. Multiple inline constraints can be specified by appending another colon (`:`) and constraint name. The constraint name and arguments are 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 a `minlength` constraint with the argument `10`. For more information on route constraints and a listing of the constraints provided by the framework, see the Route constraint reference section.

The following table demonstrates some route templates and their behavior.

Route Template	Example Matching URL	Notes
hello	/hello	Only matches the single path `/hello`
{Page=Home}	/	Matches and sets `Page` to `Home`
{Page=Home}	/Contact	Matches and sets `Page` to `Contact`
{controller}/{action}/{id?}	/Products/List	Maps to `Products` controller and `List` action
{controller}/{action}/{id?}	/Products/Details/123	Maps to `Products` controller and `Details` action. `id` set to 123
{controller=Home}/{action=Index}/{id?}	/	Maps to `Home` controller and `Index` method; `id` 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.

Reserved routing names

The following keywords are reserved names and can't be used as route names or parameters:

- `action`
- `area`
- `controller`
- `handler`
- `page`

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 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 results in a *404 - Not Found* response instead of a *400 - Bad Request* with an appropriate error message. Route constraints are 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
`int`	`{id:int}`	`123456789`, `-123456789`	Matches any int
`bool`	`{active:bool}`	`true`, `FALSE`	Matches `true` o insensitive)

constraint	Example	Example Matches	Notes
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Matches a valid `DateTime`v invariant culture warning)
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Matches a valid `decimal`va invariant culture warning)
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Matches a valid (in the invariant warning)
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Matches a valid (in the invariant warning)
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	Matches a valid
`long`	`{ticks:long}`	`123456789`, `-123456789`	Matches a valid
`minlength(value)`	`{username:minlength(4)}`	`Rick`	String must be a characters
`maxlength(value)`	`{filename:maxlength(8)}`	`Richard`	String must be n 8 characters
`length(length)`	`{filename:length(12)}`	`somefile.txt`	String must be e characters long
`length(min,max)`	`{filename:length(8,16)}`	`somefile.txt`	String must be a no more than 16 long

constraint	Example	Example Matches	Notes
`min(value)`	`{age:min(18)}`	`19`	Integer value mu 18
`max(value)`	`{age:max(120)}`	`91`	Integer value mu more than 120
`range(min,max)`	`{age:range(18,120)}`	`91`	Integer value mu 18 but no more t
`alpha`	`{name:alpha}`	`Rick`	String must cons more alphabetica (`a`-`z`, case-insens
`regex(expression)`	`{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}`	`123-45-6789`	String must matc expression (see t defining a regula
`required`	`{name:required}`	`Rick`	Used to enforce parameter value during URL gen

Multiple, colon-delimited constraints can be applied to a single parameter. For example, the following constraint restricts a parameter to an integer value of 1 or greater:

C#Copy

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { }
```

Warning

Route constraints that verify the URL and are converted to a CLR type (such as `int` or `DateTime`) always use the invariant culture. These constraints assume that the URL is non-localizable. The framework-provided route constraints don't modify the values stored in route values. All route values parsed from the URL are stored as strings. For example, the `float` constraint attempts 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 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. To use the regular expression `^\d{3}-\d{2}-\d{4}$` in Routing, the expression must 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 must be escaped by doubling them to escape the Routing parameter delimiter characters. The table below shows a regular expression and the escaped version.

Expression	Escaped
`^\d{3}-\d{2}-\d{4}$`	`^\\d{{3}}-\\d{{2}}-\\d{{4}}$`
`^[a-z]{2}$`	`^[[a-z]]{{2}}$`

Regular expressions used in routing 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 match any substring within the string, which is often undesirable. The table below shows some examples and explains why they match or fail to match.

Expression	String	Match	Comment
`[a-z]{2}`	hello	Yes	Substring matches
`[a-z]{2}`	123abc456	Yes	Substring matches
`[a-z]{2}`	mz	Yes	Matches expression
`[a-z]{2}`	MZ	Yes	Not case sensitive
`^[a-z]{2}$`	hello	No	See `^` and `$` above
`^[a-z]{2}$`	123abc456	No	See `^` and `$` above

For more information on regular expression syntax, see .NET Framework Regular Expressions.

To constrain a parameter to a known set of possible values, use a regular expression. For example, `{action:regex(^(list|get|create)$)}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string `^(list|get|create)$` is 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 following example shows how to generate a link to a route given a dictionary of route values and a RouteCollection.

C#Copy

```csharp
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 preceding sample is `/package/create/123`. The dictionary supplies the `operation` and `id` route values of the "Track Package Route" template, `package/{operation}/{id}`. For details, see the sample code in the Use Routing Middlewaresection or the sample app.

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. In an ASP.NET Core 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` is 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"	`/Home/About`
controller="Home"	controller="Order",action="About"	`/Order/About`
controller="Home",color="Red"	action="About"	`/Home/About`
controller="Home"	action="About",color="Red"	`/Home/About?color=Red`

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:
C#Copy
```
routes.MapRoute("blog_route", "blog/{*slug}",
    defaults: new { controller = "Blog", action = "ReadPost" });
```

Link generation only generates a link for this route when the matching values for controller and action are provided.