CHAPTER 13

■ ■ ■

Using URL Routing

The URL routing feature is provided by a pair of middleware components that make it easier to generate responses by consolidating the processing and matching of request URLs. In this chapter, I explain how the ASP.NET Core platform supports URL routing, show its use, and explain why it can be preferable to creating your own custom middleware components. Table 13-1 puts URL routing in context.

■ **Note** This chapter focuses on URL routing for the ASP.NET Core platform. See Part 3 for details of how the higher-level parts of ASP.NET Core build on the features described in this chapter.

Table 13-1. *Putting URL Routing in Context*

Question	Answer
What is it?	URL routing consolidates the processing and matching of URLs, allowing components known as *endpoints* to generate responses.
Why is it useful?	URL routing obviates the need for each middleware component to process the URL to see whether the request will be handled or passed along the pipeline. The result is more efficient and easier to maintain.
How is it used?	The URL routing middleware components are added to the request pipeline and configured with a set of routes. Each route contains a URL path and a delegate that will generate a response when a request with the matching path is received.
Are there any pitfalls or limitations?	It can be difficult to define the set of routes matching all the URLs supported by a complex application.
Are there any alternatives?	URL routing is optional, and custom middleware components can be used instead.

Table 13-2 summarizes the chapter.

Table 13-2. *Chapter Summary*

Problem	Solution	Listing
Handling requests for a specific set of URLs	Define a route with a pattern that matches the required URLs	1–6
Extracting values from URLs	Use segment variables	7–10, 14
Generating URLs	Use the link generator to produce URLs from routes	11–13, 15
Matching URLs with different numbers of segments	Use optional segments or catchall segments in the URL routing pattern	16–18
Restricting matches	Use constraints in the URL routing pattern	19–21, 23–26
Matching requests that are not otherwise handled	Define fallback routes	22
Seeing which endpoint will handle a request	Use the routing context data	27

Preparing for This Chapter

In this chapter, I continue to use the `Platform` project from Chapter 12. To prepare for this chapter, add a file called `Population.cs` to the `Platform` folder with the code shown in Listing 13-1.

■ **Tip**   You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/apress/pro-asp.net-core-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 13-1. The Contents of the Population.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Population {
        private RequestDelegate next;

        public Population() { }

        public Population(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            string[] parts = context.Request.Path.ToString()
                .Split("/", StringSplitOptions.RemoveEmptyEntries);
            if (parts.Length == 2 && parts[0] == "population") {
                string city = parts[1];
                int? pop = null;
                switch (city.ToLower()) {
                    case "london":
                        pop = 8_136_000;
                        break;
                    case "paris":
                        pop = 2_141_000;
                        break;
                    case "monaco":
                        pop = 39_000;
                        break;
                }
                if (pop.HasValue) {
                    await context.Response
                        .WriteAsync($"City: {city}, Population: {pop}");
                    return;
                }
            }
            if (next != null) {
                await next(context);
            }
        }
    }
}
```

This middleware component responds to requests for /population/<city> where <city> is london, paris, or monaco. The middleware component splits up the URL path string, checks that it has the expected length, and uses a switch statement to determine it is a request for a URL that it can respond to. If the URL matches the pattern the middleware is looking for, then a response is generated; otherwise, the request is passed along the pipeline.

Add a class file named Capital.cs to the Platform folder with the code shown in Listing 13-2.

Listing 13-2. The Contents of the Capital.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Capital {
        private RequestDelegate next;

        public Capital() { }

        public Capital(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            string[] parts = context.Request.Path.ToString()
                .Split("/", StringSplitOptions.RemoveEmptyEntries);
            if (parts.Length == 2 && parts[0] == "capital") {
                string capital = null;
                string country = parts[1];
                switch (country.ToLower()) {
                    case "uk":
                        capital= "London";
                        break;
                    case "france":
                        capital = "Paris";
                        break;
                    case "monaco":
                        context.Response.Redirect($"/population/{country}");
                        return;
                }
                if (capital != null) {
                    await context.Response
                        .WriteAsync($"{capital} is the capital of {country}");
                    return;
                }
            }
            if (next != null) {
                await next(context);
            }
        }
    }
}
```

This middleware component is looking for requests for /capital/<country>, where <country> is uk, france, or monaco. The capital cities of the United Kingdom and France are displayed, but requests for Monaco, which is a city and a state, are redirected to /population/monaco.

Listing 13-3 replaces the middleware examples from the previous chapter and adds the new middleware components to the request pipeline.

Listing 13-3. Replacing the Contents of the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseMiddleware<Population>();
            app.UseMiddleware<Capital>();
            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the `Platform` project folder (which contains the `Platform.csproj` file), and running the command shown in Listing 13-4.
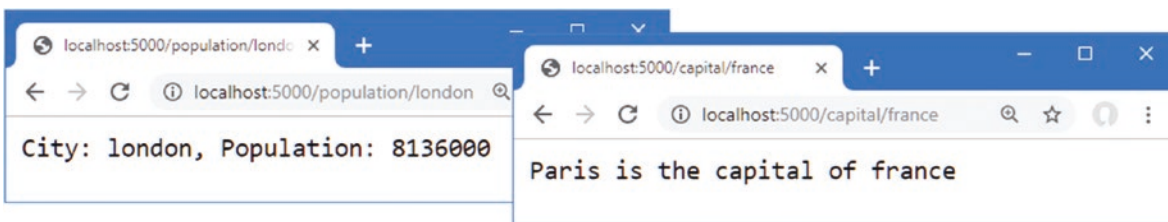
Listing 13-4. Starting the ASP.NET Core Runtime

```
dotnet run
```

Navigate to `http://localhost:5000/population/london`, and you will see the output on the left side of Figure 13-1. Navigate to `http://localhost:5000/capital/France` to see the output from the other middleware component, which is shown on the right side of Figure 13-1.

Figure 13-1. *Running the example application*

Understanding URL Routing

Each middleware component decides whether to act on a request as it passes along the pipeline. Some components are looking for a specific header or query string value, but most components—especially terminal and short-circuiting components—are trying to match URLs.

Each middleware component has to repeat the same set of steps as the request works its way along the pipeline. You can see this in the middleware defined in the prevision section, where both components go through the same process: split up the URL, check the number of parts, inspect the first part, and so on.

This approach is inefficient, it is difficult to maintain, and it breaks easily when changed. It is inefficient because the same set of operations is repeated to process the URL. It is difficult to maintain because the URL that each component is looking for is hidden in its code. It breaks easily because changes must be carefully worked through in multiple places. For example, the Capital component redirects requests to a URL whose path starts with /population, which is handled by the Population component. If the Population component is revised to support the /size URL instead, then this change must also be reflected in the Capital component. Real applications can support complex sets of URLs, and working changes fully through individual middleware components can be difficult.

URL routing solves these problems by introducing middleware that takes care of matching request URLs so that components, called *endpoints*, can focus on responses. The mapping between endpoints and the URLs they require is expressed in a *route*. The routing middleware processes the URL, inspects the set of routes and finds the endpoint to handle the request, a process known as *routing*.

Adding the Routing Middleware and Defining an Endpoint

The routing middleware is added using two separate methods: UseRouting and UseEndpoints. The UseRouting method adds the middleware responsible for processing requests to the pipeline. The UseEndpoints method is used to define the routes that match URLs to endpoints. URLs are matched using patterns that are compared to the path of request URLs, and each route creates a relationship between one URL pattern and one endpoint. Listing 13-5 shows the use of the routing middleware and contains a simple route.

> ■ **Tip** I explain why there are two methods for routing in the "Accessing the Endpoint in a Middleware Component" section.

Listing 13-5. Using the Routing Middleware in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseMiddleware<Population>();
            app.UseMiddleware<Capital>();

            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("routing", async context => {
                    await context.Response.WriteAsync("Request Was Routed");
                });
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```

There are no arguments to the UseRouting method. The UseEndpoints method receives a function that accepts an IEndpointRouteBuilder object and uses it to create routes using the extension methods described in Table 13-3.


■ **Tip** There are also extension methods that set up endpoints for other parts of ASP.NET Core, such as the MVC Framework, as explained in Part 3.

Table 13-3. *The IEndpointRouteBuilder Extension Methods*

Name	Description
MapGet(pattern, endpoint)	This method routes HTTP GET requests that match the URL pattern to the endpoint.
MapPost(pattern, endpoint)	This method routes HTTP POST requests that match to the URL pattern to the endpoint.
MapPut(pattern, endpoint)	This method routes HTTP PUT requests that match the URL pattern to the endpoint.
MapDelete(pattern, endpoint)	This method routes HTTP DELETE requests that match the URL pattern to the endpoint.
MapMethods(pattern, methods, endpoint)	This method routes requests made with one of the specified HTTP methods that match the URL pattern to the endpoint.
Map(pattern, endpoint)	This method routes all HTTP requests that match the URL pattern to the endpoint.

Endpoints are defined using RequestDelegate, which is the same delegate used by conventional middleware, so endpoints are asynchronous methods that receive an HttpContext object and use it to generate a response. This means that all the features described in earlier chapters for middleware components can also be used in endpoints.

Restart ASP.NET Core and navigate to http://localhost:5000/routing to test the new route. When matching a request, the routing middleware applies the route's URL pattern to the path section of the URL. The path is separated from the hostname by the / character, as shown in Figure 13-2.
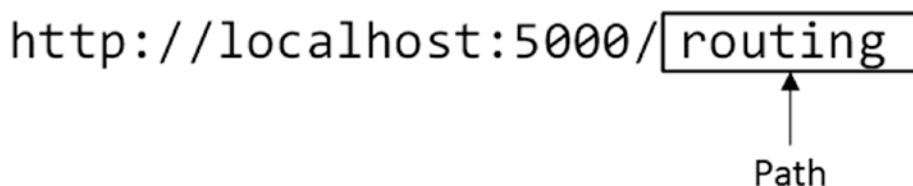

Figure 13-2. *The URL path*

The path in the URL matches the pattern specified in the route.

```
...
endpoints.MapGet("routing", async context => {
...
```

URL patterns are expressed without a leading / character, which isn't part of the URL path. When the request URL path matches the URL pattern, the request will be forwarded to the endpoint function, which generates the response shown in Figure 13-3.
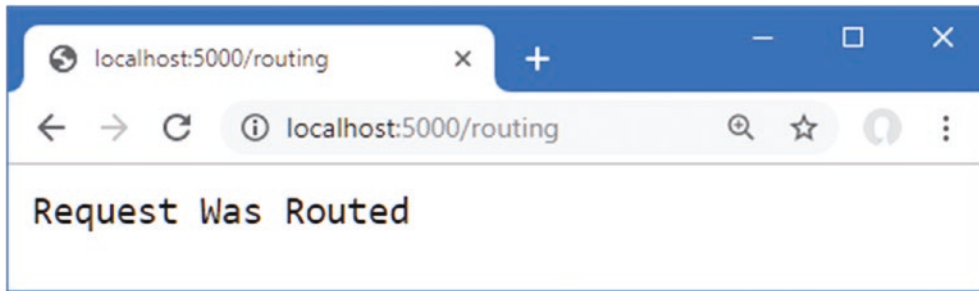
Figure 13-3. *Using an endpoint to generate a response*

The routing middleware short-circuits the pipeline when a route matches a URL so that the response is generated only by the route's endpoint. The request isn't forwarded to other endpoints or middleware components that appear later in the request pipeline.

If the request URL isn't matched by any route, then the routing middleware passes the request to the next middleware component in the request pipeline. To test this behavior, request the `http://localhost:5000/notrouted` URL, whose path doesn't match the pattern in the route defined in Listing 13-5.

The routing middleware can't match the URL path to a route and forwards the request, which reaches the terminal middleware, producing the response shown in Figure 13-4.
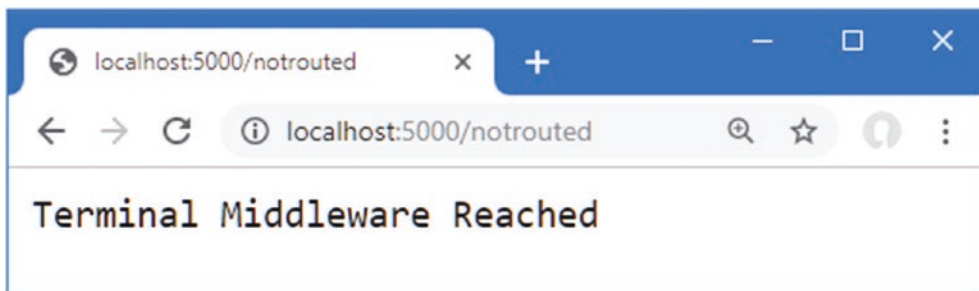
Figure 13-4. *Requesting a URL for which there is no matching route*

Endpoints generate responses in the same way as the middleware components demonstrated in earlier chapters: they receive an `HttpContext` object that provides access to the request and response through `HttpRequest` and `HttpResponse` objects. This means that any middleware component can also be used as an endpoint. Listing 13-6 adds a route that uses the `Capital` and `Population` middleware components as endpoints.

Listing 13-6. Using Middleware Components as Endpoints in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            // app.UseMiddleware<Population>();
            // app.UseMiddleware<Capital>();
```

```
        app.UseRouting();

        app.UseEndpoints(endpoints => {
            endpoints.MapGet("routing", async context => {
                await context.Response.WriteAsync("Request Was Routed");
            });
            endpoints.MapGet("capital/uk", new Capital().Invoke);
            endpoints.MapGet("population/paris", new Population().Invoke);
        });

        app.Use(async (context, next) => {
            await context.Response.WriteAsync("Terminal Middleware Reached");
        });
    }
  }
}
```

Using middleware components like this is awkward because I need to create new instances of the classes to select the Invoke method as the endpoint. The URL patterns used by the routes support only some of the URLs that the middleware components support, but it is useful to understand that endpoints rely on features that are familiar from earlier chapters. To test the new routes, restart ASP.NET Core and navigate to `http://localhost:5000/capital/uk` and `http://localhost:5000/population/paris`, which will produce the results shown in Figure 13-5.
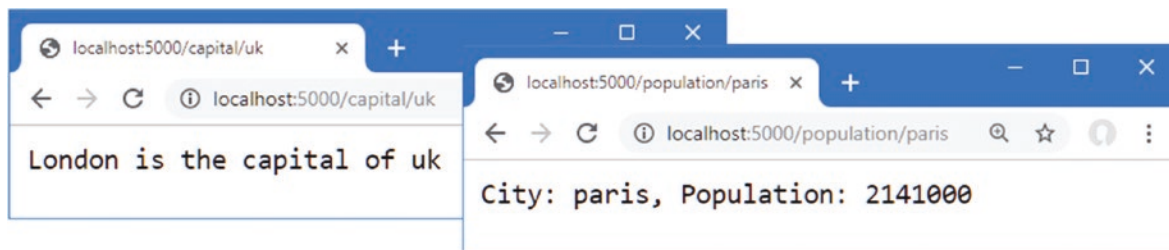

Figure 13-5. *Using middleware components as endpoints*

Understanding URL Patterns

Using middleware components as endpoints shows that the URL routing feature builds on the standard ASP.NET Core platform. Although the URLs that the application handles can be seen by examining the routes, not all of the URLs understood by the Capital and Population classes are routed, and there have been no efficiency gains since the URL is processed once by the routing middleware to select the route and again by the Capital or Population class in order to extract the data values they require.
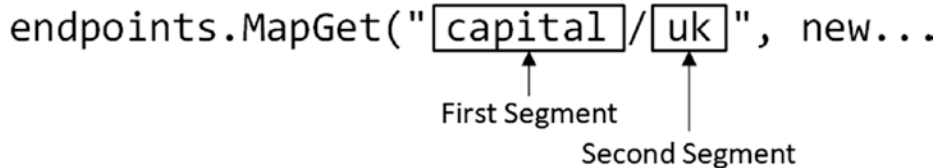
Making improvements requires understanding more about how URL patterns are used. When a request arrives, the routing middleware processes the URL to extract the segments from its path, which are the sections of the path separated by the / character, as shown in Figure 13-6.
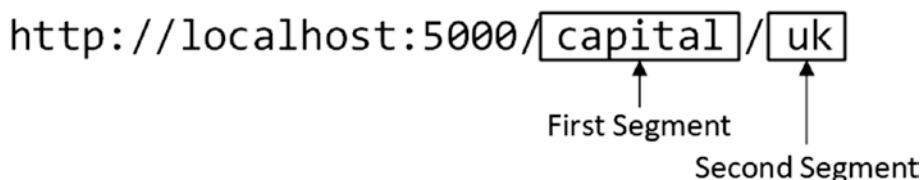
Figure 13-6. *The URL segments*

The routing middleware also extracts the segments from the URL routing pattern, as shown in Figure 13-7.

Figure 13-7. *The URL pattern segments*

To route a request, the segments from the URL pattern are compared to those from the request to see whether they match. The request is routed to the endpoint if its path contains the same number of segments and each segment has the same content as those in the URL pattern, as summarized in Table 13-4.

Table 13-4. *Matching URL Segments*

URL Path	Description
/capital	No match—too few segments
/capital/europe/uk	No match—too many segments
/name/uk	No match—first segment is not capital
/capital/uk	Matches

Using Segment Variables in URL Patterns

The URL pattern used in Listing 13-6 uses *literal segments*, also known as *static segments*, which match requests using fixed strings. The first segment in the pattern will match only those requests whose path has capital as the first segment, for example, and the second segment in the pattern will match only those requests whose second segment is uk. Put these together and you can see why the route matches only those requests whose path is /capital/uk.

Segment variables, also known as *route parameters*, expand the range of path segments that a pattern segment will match, allowing more flexible routing. Segment variables are given a name and are denoted by curly braces (the { and } characters), as shown in Listing 13-7.

Listing 13-7. Using Segment Variables in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("{first}/{second}/{third}", async context => {
                    await context.Response.WriteAsync("Request Was Routed\n");
```

```
            foreach (var kvp in context.Request.RouteValues) {
                await context.Response
                    .WriteAsync($"{kvp.Key}: {kvp.Value}\n");
            }
        });
        endpoints.MapGet("capital/uk", new Capital().Invoke);
        endpoints.MapGet("population/paris", new Population().Invoke);
    });

    app.Use(async (context, next) => {
        await context.Response.WriteAsync("Terminal Middleware Reached");
    });
        }
    }
}
```

The URL pattern {first}/{second}/{third} matches URLs whose path contains three segments, regardless of what those segments contain. When a segment variable is used, the routing middleware provides the endpoint with the contents of the URL path segment they have matched. This content is available through the HttpRequest.RouteValues property, which returns a RouteValuesDictionary object. Table 13-5 describes the most useful RouteValuesDictionary members.

■ **Tip** There are some reserved words that cannot be used as the names for segment variables: action, area, controller, handler, and page.

*Table 13-5.* *Useful RouteValuesDictionary Members*

Name	Description
[key]	The class defines an indexer that allows values to be retrieved by key.
Keys	This property returns the collection of segment variable names.
Values	This property returns the collection of segment variable values.
Count	This property returns the number of segment variables.
ContainsKey(key)	This method returns true if the route data contains a value for the specified key.

The RouteValuesDictionary class is enumerable, which means that it can be used in a foreach loop to generate a sequence of KeyValuePair<string, object> objects, each of which corresponds to the name of a segment variable and the corresponding value extracted from the request URL. The endpoint in Listing 13-7 enumerates the HttpRequest.RouteValues property to generate a response that lists the names and value of the segment variables matched by the URL pattern.

The names of the segment variables are first, second, and third, and you can see the values extracted from the URL by restarting ASP.NET Core and requesting any three-segment URL, such as http://localhost:5000/apples/oranges/cherries, which produces the response shown in Figure 13-8.
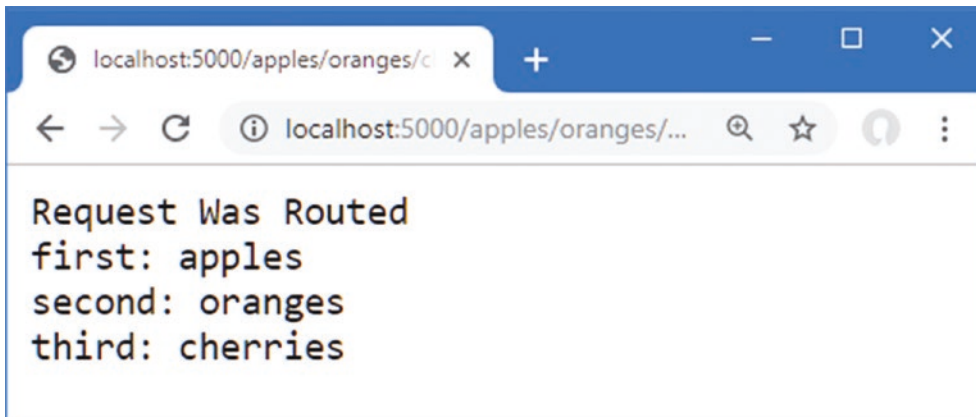
Figure 13-8. *Using segment variables*

UNDERSTANDING ROUTE SELECTION

When processing a request, the middleware finds all the routes that can match the request and gives each a score, and the route with the lowest score is selected to handle the route. The scoring process is complex, but the effect is that the most specific route receives the request. This means that literal segments are given preference over segment variables and that segment variables with constraints are given preference over those without (constraints are described in the "Constraining Segment Matching" section later in this chapter). The scoring system can produce surprising results, and you should check to make sure that the URLs supported by your application are matched by the routes you expect.

If two routes have the same score, meaning they are equally suited to routing the request, then an exception will be thrown, indicating an ambiguous routing selection. See the "Avoiding Ambiguous Route Exceptions" section later in the chapter for details of how to avoid ambiguous routes.

Refactoring Middleware into an Endpoint

Endpoints usually rely on the routing middleware to provide specific segment variables, rather than enumerating all the segment variables. By relying on the URL pattern to provide a specific value, I can refactor the Capital and Population classes to depend on the route data, as shown in Listing 13-8.

Listing 13-8. Depending on the Route Data in the Capital.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public static class Capital {

        public static async Task Endpoint(HttpContext context) {
            string capital = null;
            string country = context.Request.RouteValues["country"] as string;
            switch ((country ?? "").ToLower()) {
                case "uk":
                    capital = "London";
                    break;
                case "france":
                    capital = "Paris";
```

```
                break;
            case "monaco":
                context.Response.Redirect($"/population/{country}");
                return;
        }
        if (capital != null) {
            await context.Response
                .WriteAsync($"{capital} is the capital of {country}");
        } else {
            context.Response.StatusCode = StatusCodes.Status404NotFound;
        }
    }
}
}
```

Middleware components can be used as endpoints, but the opposite isn't true once there is a dependency on the data provided by the routing middleware. In Listing 13-8, I used the route data to get the value of a segment variable named country through the indexer defined by the RouteValuesDictionary class.

```
...
string country = context.Request.RouteValues["country"] as string;
...
```

The indexer returns an object value that is cast to a string using the as keyword. The listing removes the statements that pass the request along the pipeline, which the routing middleware handles on behalf of endpoints.

The use of the segment variable means that requests may be routed to the endpoint with values that are not supported, so I added a statement that returns a 404 status code for countries the endpoint doesn't understand.

I also removed the constructors and replaced the Invoke instance method with a static method named Endpoint, which better fits with the way that endpoints are used in routes. Listing 13-9 applies the same set of changes to the Population class, transforming it from a standard middleware component into an endpoint that depends on the routing middleware to process URLs.

Listing 13-9. Depending on Route Data in the Population.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Population {

        public static async Task Endpoint(HttpContext context) {
            string city = context.Request.RouteValues["city"] as string;
            int? pop = null;
            switch ((city ?? "").ToLower()) {
                case "london":
                    pop = 8_136_000;
                    break;
                case "paris":
                    pop = 2_141_000;
                    break;
                case "monaco":
                    pop = 39_000;
                    break;
            }
            if (pop.HasValue) {
                await context.Response
                    .WriteAsync($"City: {city}, Population: {pop}");
            } else {
```

```
                context.Response.StatusCode = StatusCodes.Status404NotFound;
            }
        }
    }
}
```

The change to static methods tidies up the use of the endpoints when defining routes, as shown in Listing 13-10.

Listing 13-10. Updating Routes in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first}/{second}/{third}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country}", Capital.Endpoint);
    endpoints.MapGet("population/{city}", Population.Endpoint);
});
...
```

The new routes match URLs whose path has two segments, the first of which is `capital` or `population`. The contents of the second segment are assigned to the segment variables named `country` and `city`, allowing the endpoints to support the full set of URLs that were handled at the start of the chapter, without the need to process the URL directly. To test the new routes, restart ASP. NET Core and request `http://localhost:5000/capital/uk` and `http://localhost:5000/population/london`, which will produce the responses shown in Figure 13-9.
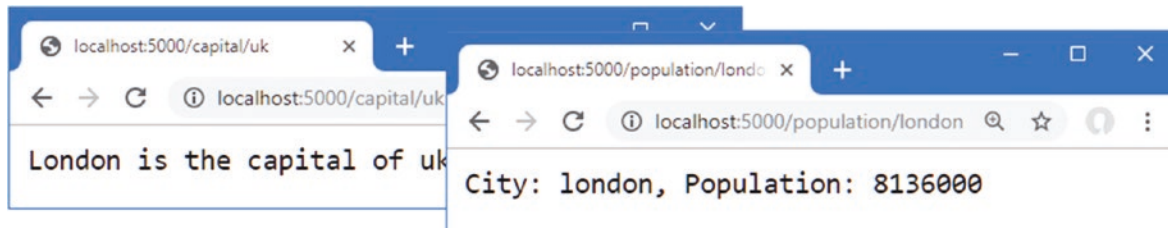
Figure 13-9. *Using segment variables in endpoints*

These changes address two of the problems I described at the start of the chapter. Efficiency has improved because the URL is processed only once by the routing middleware and not by multiple components. And it is easier to see the URLs that each endpoint supports because the URL patterns show how requests will be matched.

Generating URLs from Routes

The final problem was the difficulty in making changes. The `Capital` endpoint still has a hardwired dependency on the URL that the `Population` endpoint supports. To break this dependency, the routing system allows URLs to be generated by supplying data values for segment variables. The first step is to assign a name to the route that will be the target of the URL that is generated, as shown in Listing 13-11.

Listing 13-11. Naming a Route in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
```

```
namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("{first}/{second}/{third}", async context => {
                    await context.Response.WriteAsync("Request Was Routed\n");
                    foreach (var kvp in context.Request.RouteValues) {
                        await context.Response
                            .WriteAsync($"{kvp.Key}: {kvp.Value}\n");
                    }
                });
                endpoints.MapGet("capital/{country}", Capital.Endpoint);
                endpoints.MapGet("population/{city}", Population.Endpoint)
                    .WithMetadata(new RouteNameMetadata("population"));
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```

The `WithMetadata` method is used on the result from the `MapGet` method to assign metadata to the route. The only metadata required for generating URLs is a name, which is assigned by passing a new `RouteNameMetadata` object, whose constructor argument specifies the name that will be used to refer to the route. In Listing 13-11, I have named the route `population`.

■ **Tip**   Naming routes helps to avoid links being generated that target a route other than the one you expect, but they can be omitted, in which case the routing system will try to find the best matching route. You can see an example of this approach in Chapter 17.

In Listing 13-12, I have revised the `Capital` endpoint to remove the direct dependency on the `/population` URL and rely on the routing features to generate a URL.

Listing 13-12. Generating a URL in the Capital.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public static class Capital {
        public static async Task Endpoint(HttpContext context) {
            string capital = null;
            string country = context.Request.RouteValues["country"] as string;
            switch ((country ?? "").ToLower()) {
                case "uk":
```

```
                    capital= "London";
                    break;
                case "france":
                    capital = "Paris";
                    break;
                case "monaco":
                    LinkGenerator generator =
                        context.RequestServices.GetService<LinkGenerator>();
                    string url = generator.GetPathByRouteValues(context,
                        "population", new { city = country });
                    context.Response.Redirect(url);
                    return;
            }
            if (capital != null) {
                await context.Response
                    .WriteAsync($"{capital} is the capital of {country}");
            } else {
                context.Response.StatusCode = StatusCodes.Status404NotFound;
            }
        }
    }
}
```

URLs are generated using the LinkGenerator class. You can't just create a new LinkGenerator instance; one must be obtained using the dependency injection feature that is described in Chapter 14. For the purposes of this chapter, it is enough to know that this statement obtains the LinkGenerator object that the endpoint will use:

```
...
LinkGenerator generator = context.RequestServices.GetService<LinkGenerator>();
...
```

The LinkGenerator class provides the GetPathByRouteValues method, which is used to generate the URL that will be used in the redirection.

```
...
generator.GetPathByRouteValues(context, "population", new { city = country });
...
```

The arguments to the GetPathByRouteValues method are the endpoint's HttpContext object, the name of the route that will be used to generate the link, and an object that is used to provide values for the segment variables. The GetPathByRouteValues method returns a URL that will be routed to the Population endpoint, which can be confirmed by restarting ASP.NET Core and requesting the http://localhost:5000/capital/monaco URL. The request will be routed to the Capital endpoint, which will generate the URL and use it to redirect the browser, producing the result shown in Figure 13-10.


Figure 13-10. *Generating a URL*

The benefit of this approach is that the URL is generated from the URL pattern in the named route, which means a change in the URL pattern is reflected in the generated URLs, without the need to make changes to endpoints. To demonstrate, Listing 13-13 changes the URL pattern.

Listing 13-13. Changing a URL Pattern in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first}/{second}/{third}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country}", Capital.Endpoint);
    endpoints.MapGet("size/{city}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

The name assigned to the route is unchanged, which ensures that the same endpoint is targeted by the generated URL. To see the effect of the new pattern, restart ASP.NET Core and request the `http://localhost:5000/capital/monaco` URL again. The redirection is to a URL that is matched by the modified pattern, as shown in Figure 13-11. This feature addresses the final problem that I described at the start of the chapter, making it easy to change the URLs that an application supports.
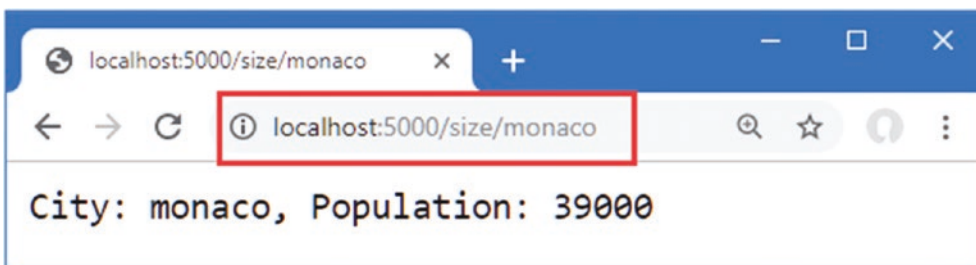
Figure 13-11. *Changing the URL pattern*

URL ROUTING AND AREAS

The URL routing system supports a feature called *areas*, which allows separate sections of the application to have their own controllers, views, and Razor Pages. I have not described the areas feature in this book because it is not widely used and, when it is used, it tends to cause more problems than it solves. If you want to break up an application, then I recommend creating separate projects.

Managing URL Matching

The previous section introduced the basic URL routing features, but most applications require more work to ensure that URLs are routed correctly, either to increase or to restrict the range of URLs that are matched by a route. In the sections that follow, I show you the different ways that URL patterns can be adjusted to fine-tune the matching process.

Matching Multiple Values from a Single URL Segment

Most segment variables correspond directly to a segment in the URL path, but the routing middleware is able to perform more complex matches, allowing a single segment to be matched to a variable while discarding unwanted characters. Listing 13-14 defines a route that matches only part of a URL segment to a variable.

294

Listing 13-14. Matching Part of a Segment in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("files/{filename}.{ext}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country}", Capital.Endpoint);
    endpoints.MapGet("size/{city}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

A URL pattern can contain as many segment variables as you need, as long as they are separated by a static string. The requirement for a static separator is so the routing middleware knows where the content for one variable ends and content for the next starts. The pattern in Listing 13-14 matches segment variables named `filename` and `ext`, which are separated by a period; this pattern is often used by process file names. To see how the pattern matches URLs, restart ASP.NET Core and request the `http://localhost:5000/files/myfile.txt` URL, which will produce the response shown in Figure 13-12.
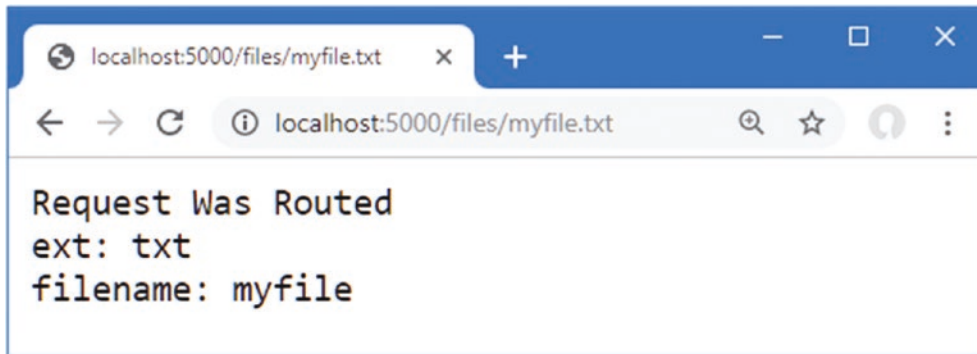
Figure 13-12. *Matching multiple values from a single path segment*

AVOIDING THE COMPLEX PATTERN MISMATCHING PITFALL

The order of the segment variables shown in Figure 13-12 shows that pattern segments that contain multiple variables are matched from right to left. This isn't important most of the time, because endpoints can't rely on a specific key order, but it does show that complex URL patterns are handled differently, which reflects the difficulty in matching them.

In fact, the matching process is so difficult that there can be unexpected matching failures. The specific failures change with each release of ASP.NET Core as the matching process is adjusted to address problems, but the adjustments often introduce new issues. At the time of writing, there is a problem with URL patterns where the content that should be matched by the first variable also appears as a literal string at the start of a segment. This is easier to understand with an example, as shown here:

```
...
endpoints.MapGet("example/red{color}", async context => {
...
```

This pattern has a segment that begins with the literal string `red`, followed by a segment variable named `color`. The routing middleware will correctly match the pattern against the URL path `example/redgreen`, and the value of the `color` route variable will be `green`. However, the URL path `example/redredgreen` won't match because the matching process confuses the position of the literal content with the first part of the content that should be assigned to the `color` variable. This problem may be fixed by the time you read this book, but there will be other issues with complex patterns. It is a good idea to keep URL patterns as simple as possible and make sure you get the matching results you expect.

Using Default Values for Segment Variables

Patterns can be defined with default values that are used when the URL doesn't contain a value for the corresponding segment, increasing the range of URLs that a route can match. Listing 13-15 shows the use of default values in a pattern.

Listing 13-15. Using Default Values in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("files/{filename}.{ext}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

Default values are defined using an equal sign and the value to be used. The default value in the listing uses the value `France` when there is no second segment in the URL path. The result is that the range of URLs that can be matched by the route increases, as described in Table 13-6.

Table 13-6. *Matching URLs*

URL Path	Description
/	No match—too few segments
/city	No match—first segment isn't `capital`
/capital	Matches, `country` variable is `France`
/capital/uk	Matches, `country` variable is `uk`
/capital/europe/italy	No match—too many segments

To test the default value, restart ASP.NET Core and navigate to `http://localhost:5000/capital`, which will produce the result shown in Figure 13-13.
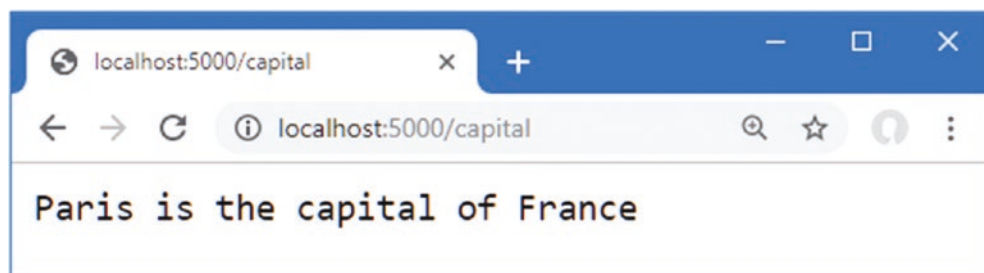
Figure 13-13. *Using a default value for a segment variable*

Using Optional Segments in a URL Pattern

Default values allow URLs to be matched with fewer segments, but the use of the default value isn't obvious to the endpoint. Some endpoints define their own responses to deal with URLs that omit segments, for which *optional segments* are used. To prepare, Listing 13-16 updates the `Population` endpoint so that it uses a default value when no `city` value is available in the routing data.

Listing 13-16. Using a Default Value in the Population.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Population {

        public static async Task Endpoint(HttpContext context) {
            string city = context.Request.RouteValues["city"] as string ?? "london";
            int? pop = null;
            switch (city.ToLower()) {
                case "london":
                    pop = 8_136_000;
                    break;
                case "paris":
                    pop = 2_141_000;
                    break;
                case "monaco":
                    pop = 39_000;
                    break;
            }
            if (pop.HasValue) {
                await context.Response
                    .WriteAsync($"City: {city}, Population: {pop}");
            } else {
                context.Response.StatusCode = StatusCodes.Status404NotFound;
            }
        }
    }
}
```

The change uses london as the default value because there is no city segment variable available. Listing 13-17 updates the route for the Population endpoint to make the second segment optional.

Listing 13-17. Using an Optional Segment in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("files/{filename}.{ext}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

Optional segments are denoted with a question mark (the ? character) after the variable name and allow the route to match URLs that don't have a corresponding path segment, as described in Table 13-7.

Table 13-7. *Matching URLs*

URL Path	Description
/	No match—too few segments.
/city	No match—first segment isn't `size`.
/size	Matches. No value for the `city` variable is provided to the endpoint.
/size/paris	Matches, `city` variable is `paris`.
/size/europe/italy	No match—too many segments.

To test the optional segment, restart ASP.NET Core and navigate to `http://localhost:5000/size`, which will produce the response shown in Figure 13-14.
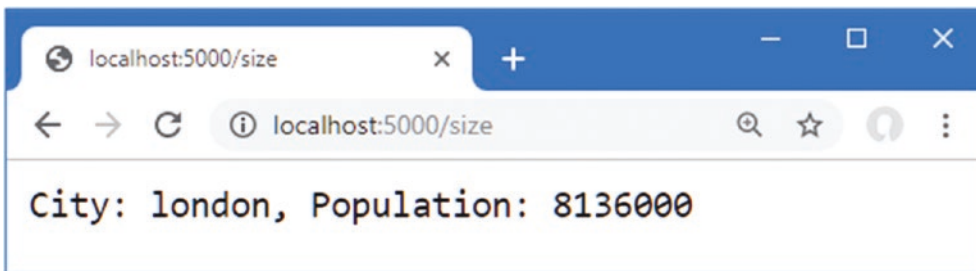
Figure 13-14. *Using an optional segment*

Using a catchall Segment Variable

Optional segments allow a pattern to match shorter URL paths. A *catchall* segment does the opposite and allows routes to match URLs that contain more segments than the pattern. A catchall segment is denoted with an asterisk before the variable name, as shown in Listing 13-18.

Listing 13-18. Using a Catchall Segment in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first}/{second}/{*catchall}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

The new pattern contains two-segment variables and a catchall, and the result is that the route will match any URL whose path contains two or more segments. There is no upper limit to the number of segments that the URL pattern in this route will match, and the contents of any additional segments are assigned to the segment variable named `catchall`. Restart ASP.NET Core and navigate to `http://localhost:5000/one/two/three/four`, which produces the response shown in Figure 13-15.

■ **Tip** Notice that the segments captured by the catchall are presented in the form *segment/segment/segment* and that the endpoint is responsible for processing the string to break out the individual segments.
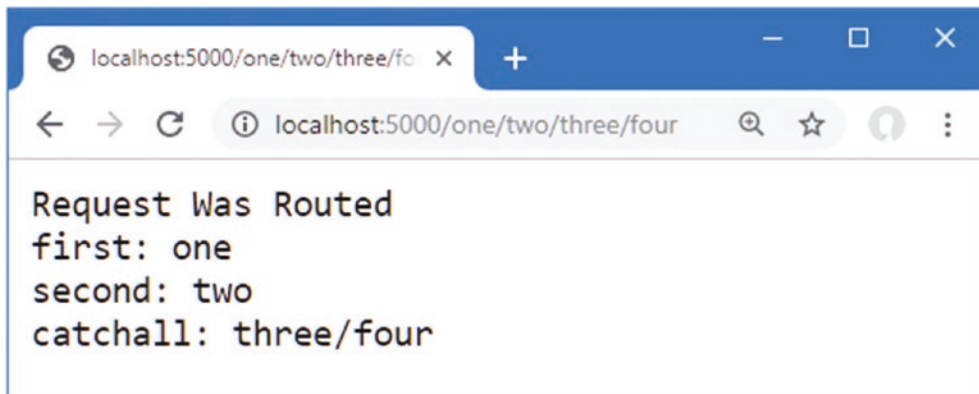
Figure 13-15. *Using a catchall segment variable*

Constraining Segment Matching

Default values, optional segments, and catchall segments all increase the range of URLs that a route will match. Constraints have the opposite effect and restrict matches. This can be useful if an endpoint can deal only with specific segment contents or if you want to differentiate matching closely related URLs for different endpoints. Constraints are applied by a colon (the : character) and a constraint type after a segment variable name, as shown in Listing 13-19.

Listing 13-19. Applying Constraints in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:int}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

This example constrains the first segment variable so it will match only the path segments that can be parsed to an `int` value, and it constrains the second segment so it will match only the path segments that can be parsed to a `bool`. Values that don't match the constraints won't be matched by the route. Table 13-8 describes the URL pattern constraints.


■ **Note**   Some of the constraints match types whose format can differ based on locale. The routing middleware doesn't handle localized formats and will match only those values that are expressed in the invariant culture format.

Table 13-8. *The URL Pattern Constraints*

Constraint	Description
alpha	This constraint matches the letters *a* to *z* (and is case-insensitive).
bool	This constraint matches true and false (and is case-insensitive).
datetime	This constraint matches DateTime values, expressed in the nonlocalized invariant culture format.
decimal	This constraint matches decimal values, formatted in the nonlocalized invariant culture.
double	This constraint matches double values, formatted in the nonlocalized invariant culture.
file	This constraint matches segments whose content represents a file name, in the form name.ext. The existence of the file is not validated.
float	This constraint matches float values, formatted in the nonlocalized invariant culture.
guid	This constraint matches GUID values.
int	This constraint matches int values.
length(len)	This constraint matches path segments that have the specified number of characters.
length(min, max)	This constraint matches path segments whose length falls between the lower and upper values specified.
long	This constraint matches long values.
max(val)	This constraint matches path segments that can be parsed to an int value that is less than or equal to the specified value.
maxlength(len)	This constraint matches path segments whose length is equal to or less than the specified value.
min(val)	This constraint matches path segments that can be parsed to an int value that is more than or equal to the specified value.
minlength(len)	This constraint matches path segments whose length is equal to or more than the specified value.
nonfile	This constraint matches segments that do not represent a file name, i.e., values that would not be matched by the file constraint.
range(min, max)	This constraint matches path segments that can be parsed to an int value that falls between the inclusive range specified.
regex(expression)	This constraint applies a regular expression to match path segments.

To test the constraints, restart ASP.NET Core and request `http://localhost:5000/100/true`, which is a URL whose path segments conform to the constraints in Listing 13-19 and that produces the result shown on the left side of Figure 13-16. Request `http://localhost:5000/apples/oranges`, which has the right number of segments but contains values that don't conform to the constraints. None of the routes matches the request, which is forwarded to the terminal middleware, as shown on the right of Figure 13-16.
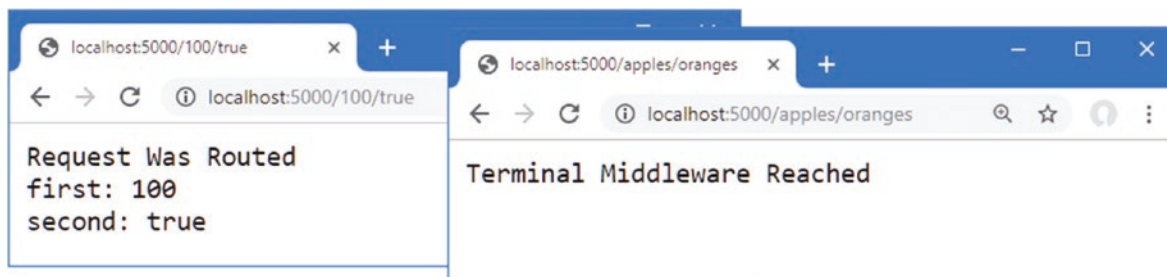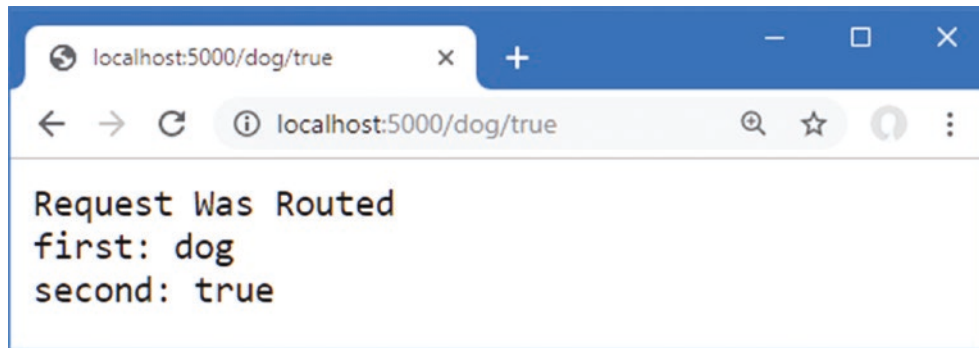

Figure 13-16. *Testing constraints*

Constraints can be combined to further restrict matching, as shown in Listing 13-20.

Listing 13-20. Combining URL Pattern Constraints in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:alpha:length(3)}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

The constraints are combined, and only path segments that can satisfy all the constraints will be matched. The combination in Listing 13-20 constrains the URL pattern so that the first segment will match only three alphabetic characters. To test the pattern, restart ASP.NET Core and request `http://localhost:5000/dog/true`, which will produce the output shown in Figure 13-17. Requesting the URL `http://localhost:5000/dogs/true` won't match the route because the first segment contains four characters.

Figure 13-17. Combining constraints

Constraining Matching to a Specific Set of Values

The `regex` constraint applies a regular expression, which provides the basis for one of the most commonly required restrictions: matching only a specific set of values. In Listing 13-21, I have applied the regex constraint to the routes for the `Capital` endpoint, so it will receive requests only for the values it is able to process.

Listing 13-21. Matching Specific Values in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:alpha:length(3)}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });

    endpoints.MapGet("capital/{country:regex(^uk|france|monaco$)}",
        Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

The route will match only those URLs with two segments. The first segment must be capital, and the second segment must be uk, france, or monaco. Regular expressions are case-insensitive, which you can confirm by restarting ASP.NET Core and requesting http://localhost:5000/capital/UK, which will produce the result shown in Figure 13-18.

■ **Tip** You may find that your browser requests /capital/uk, with a lowercase uk. If this happens, clear your browser history and try again.
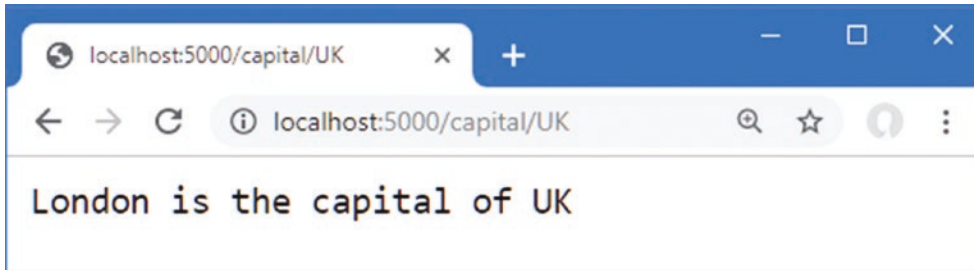
Figure 13-18. *Matching specific values with a regular expression*

Defining Fallback Routes

Fallback routes direct a request to an endpoint only when no other route matches a request. Fallback routes prevent requests from being passed further along the request pipeline by ensuring that the routing system will always generate a response, as shown in Listing 13-22.

Listing 13-22. Using a Fallback Route in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:alpha:length(3)}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });

    endpoints.MapGet("capital/{country:regex(^uk|france|monaco$)}",
        Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));

    endpoints.MapFallback(async context => {
        await context.Response.WriteAsync("Routed to fallback endpoint");
    });
});
...
```

The MapFallback method creates a route that will be used as a last resort and that will match any request. The methods for creating fallback routes are described in Table 13-9. (There are also methods for creating fallback routes that are specific to other parts of ASP.NET Core and that are described in Part 3.)

Table 13-9. *The Methods for Creating Fallback Routes*

Name	Description
MapFallback(endpoint)	This method creates a fallback that routes requests to an endpoint.
MapFallbackToFile(path)	This method creates a fallback that routes requests to a file.

With the addition of the route in Listing 13-22, the routing middleware with handle all requests, including those that match none of the regular routes. Restart ASP.NET Core and navigate to a URL that won't be matched by any of the routes, such as http://localhost:5000/notmatched, and you will see the response shown in Figure 13-19.
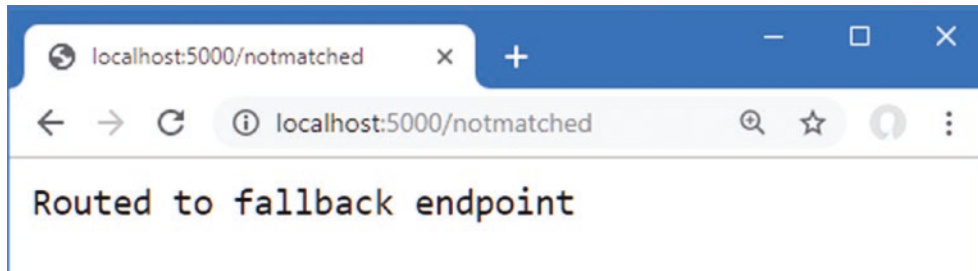
Figure 13-19. *Using a fallback route*

There is no magic to fallback routes. The URL pattern used by fallbacks is {path:nofile}, and they rely on the Order property to ensure that the route is used only if there are no other suitable routes, which is a feature described in the "Avoiding Ambiguous Route Exceptions" section.

Advanced Routing Features

The routing features described in the previous sections address the needs of most projects, especially since they are usually accessed through higher-level features such as the MVC Framework, described in Part 3. There are some advanced features for projects that have unusual routing requirements, which I describe in the following sections.

Creating Custom Constraints

If the constraints described in Table 13-8 are not sufficient, you can define your own custom constraints by implementing the IRouteConstraint interface. To create a custom constraint, add a file named CountryRouteConstraint.cs to the Platform folder and add the code shown in Listing 13-23.

Listing 13-23. The Contents of the CountryRouteConstraint.cs File in the Platform Folder

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;

namespace Platform {

    public class CountryRouteConstraint: IRouteConstraint {
        private static string[] countries = { "uk", "france", "monaco" };

        public bool Match(HttpContext httpContext, IRouter route, string routeKey,
                RouteValueDictionary values, RouteDirection routeDirection) {
            string segmentValue = values[routeKey] as string ?? "";
            return Array.IndexOf(countries, segmentValue.ToLower()) > -1;
        }
    }
}
```

303

The IRouteConstraint interface defines the Match method, which is called to allow a constraint to decide whether a request should be matched by the route. The parameters for the Match method provide the HttpContext object for the request, the route, the name of the segment, the segment variables extracted from the URL, and whether the request is to check for an incoming or outgoing URL. The Match method returns true if the constraint is satisfied by the request and false if it is not. The constraint in Listing 13-23 defines a set of countries that are compared to the value of the segment variable to which the constraint has been applied. The constraint is satisfied if the segment matches one of the countries. Custom constraints are set up using the options pattern, as shown in Listing 13-24. (The options pattern is described in Chapter 12.)

Listing 13-24. Using a Custom Constraint in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(opts => {
              opts.ConstraintMap.Add("countryName",
                  typeof(CountryRouteConstraint));
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("{first:alpha:length(3)}/{second:bool}",
                    async context => {
                        await context.Response.WriteAsync("Request Was Routed\n");
                        foreach (var kvp in context.Request.RouteValues) {
                            await context.Response
                                .WriteAsync($"{kvp.Key}: {kvp.Value}\n");
                        }
                    });

                endpoints.MapGet("capital/{country:countryName}", Capital.Endpoint);

                endpoints.MapGet("size/{city?}", Population.Endpoint)
                    .WithMetadata(new RouteNameMetadata("population"));

                endpoints.MapFallback(async context => {
                    await context.Response.WriteAsync("Routed to fallback endpoint");
                });
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```

The options pattern is applied to the RouteOptions class, which defines the ConstraintMap property. Each constraint is registered with a key that allows it to be applied in URL patterns. In Listing 13-24, the key for the CountryRouteConstraint class is countyName, which allows me to constraint a route like this:

```
...
endpoints.MapGet("capital/{country:countryName}", Capital.Endpoint);
...
```

Requests will be matched by this route only when the first segment of the URL is capital and the second segment is one of the countries defined in Listing 13-23.

Avoiding Ambiguous Route Exceptions

When trying to route a request, the routing middleware assigns each route a score. As explained earlier in the chapter, precedence is given to more specific routes, and route selection is usually a straightforward process that behaves predictably, albeit with the occasional surprise if you don't think through and test the full range of URLs the application will support.

If two routes have the same score, the routing system can't choose between them and throws an exception, indicating that the routes are ambiguous. In most cases, the best approach is to modify the ambiguous routes to increase specificity by introducing literal segments or a constraint. There are some situations where that won't be possible, and some extra work is required to get the routing system to work as intended. Listing 13-25 replaces the routes from the previous example with two new routes that are ambiguous, but only for some requests.

Listing 13-25. Defining Ambiguous Routes in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(opts => {
                opts.ConstraintMap.Add("countryName",
                    typeof(CountryRouteConstraint));
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.Map("{number:int}", async context => {
                    await context.Response.WriteAsync("Routed to the int endpoint");
                });
                endpoints.Map("{number:double}", async context => {
                    await context.Response
                        .WriteAsync("Routed to the double endpoint");
                });
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```

These routes are ambiguous only for some values. Only one route matches URLs where the first path segment can be parsed to a double, but both routes match for where the segment can be parsed as an int or a double. To see the issue, restart ASP.NET Core and request http://localhost:5000/23.5. The path segment 23.5 can be parsed to a double and produces the response shown on the left side of Figure 13-20. Request http://localhost:5000/23, and you will see the exception shown on the right of Figure 13-20. The segment 23 can be parsed as both an int and a double, which means that the routing system cannot identify a single route to handle the request.
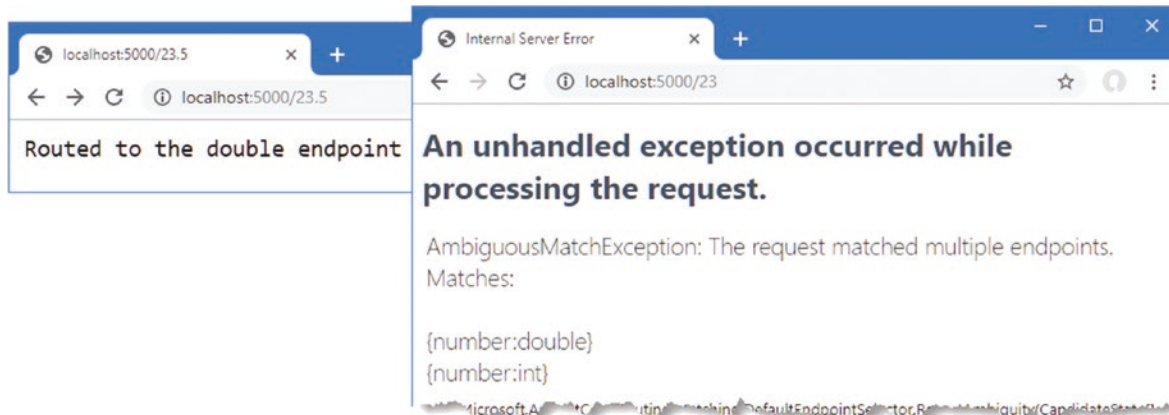
*Figure 13-20.* *An occasionally ambiguous routing configuration*

For these situations, preference can be given to a route by defining its order relative to other matching routes, as shown in Listing 13-26.

Listing 13-26. Breaking Route Ambiguity in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.Map("{number:int}", async context => {
        await context.Response.WriteAsync("Routed to the int endpoint");
    }).Add(b => ((RouteEndpointBuilder)b).Order = 1);

    endpoints.Map("{number:double}", async context => {
        await context.Response.WriteAsync("Routed to the double endpoint");
    }).Add(b => ((RouteEndpointBuilder)b).Order = 2);
});
...
```

The process is awkward and requires a call to the Add method, casting to a RouteEndpointBuilder and setting the value of the Order property. Precedence is given to the route with the lowest Order value, which means that the changes in Listing 13-26 tell the routing system to use the first route for URLs that both routes can handle. Restart ASP.NET Core and request the http://localhost:5000/23 URL again, and you will see that the first route handles the request, as shown in Figure 13-21.
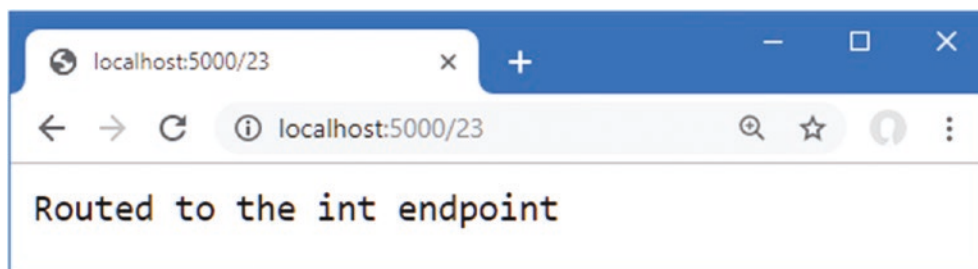
Figure 13-21. *Avoiding ambiguous routes*

Accessing the Endpoint in a Middleware Component

As earlier chapters demonstrated, not all middleware generates responses. Some components provide features used later in the request pipeline, such as the session middleware, or enhance the response in some way, such as status code middleware.

One limitation of the normal request pipeline is that a middleware component at the start of the pipeline can't tell which of the later components will generate a response. The routing middleware does something different. Although routes are registered in the UseEndpoints method, the selection of a route is done in the UseRouting method, and the endpoint is executed to generate a response in the UseEndpoints method. Any middleware component that is added to the request pipeline between the UseRouting method and the UseEndpoints method can see which endpoint has been selected before the response is generated and alter its behavior accordingly.

In Listing 13-27, I have added a middleware component that adds different messages to the response based on the route that has been selected to handle the request.

Listing 13-27. Adding a Middleware Component in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(opts => {
              opts.ConstraintMap.Add("countryName",
                  typeof(CountryRouteConstraint));
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.Use(async (context, next) => {
                Endpoint end = context.GetEndpoint();
                if (end != null) {
                    await context.Response
                        .WriteAsync($"{end.DisplayName} Selected \n");
                } else {
                    await context.Response.WriteAsync("No Endpoint Selected \n");
                }
                await next();
            });

            app.UseEndpoints(endpoints => {
                endpoints.Map("{number:int}", async context => {
                    await context.Response.WriteAsync("Routed to the int endpoint");
                })
                .WithDisplayName("Int Endpoint")
                .Add(b => ((RouteEndpointBuilder)b).Order = 1);

                endpoints.Map("{number:double}", async context => {
                    await context.Response
                        .WriteAsync("Routed to the double endpoint");
                })
                .WithDisplayName("Double Endpoint")
```

```
                .Add(b => ((RouteEndpointBuilder)b).Order = 2);
        });

        app.Use(async (context, next) => {
            await context.Response.WriteAsync("Terminal Middleware Reached");
        });
    }
  }
}
```

The GetEndpoint extension method on the HttpContext class returns the endpoint that has been selected to handle the request, described through an Endpoint object. The Endpoint class defines the properties described in Table 13-10.

■ **Caution** There is also a SetEndpoint method that allows the endpoint chosen by the routing middleware to be changed before the response is generated. This should be used with caution and only when there is a compelling need to interfere with the normal route selection process.


Table 13-10. *The Properties Defined by the Endpoint Class*

Name	Description
DisplayName	This property returns the display name associated with the endpoint, which can be set using the WithDisplayName method when creating a route.
Metadata	This property returns the collection of metadata associated with the endpoint.
RequestDelegate	This property returns the delegate that will be used to generate the response.

To make it easier to identify the endpoint that the routing middleware has selected, I used the WithDisplayName method to assign names to the routes in Listing 13-27. The new middleware component adds a message to the response reporting the endpoint that has been selected. Restart ASP.NET Core and request the http://localhost:5000/23 URL to see the output from the middleware that shows the endpoint has been selected between the two methods that add the routing middleware to the request pipeline, as shown in Figure 13-22.
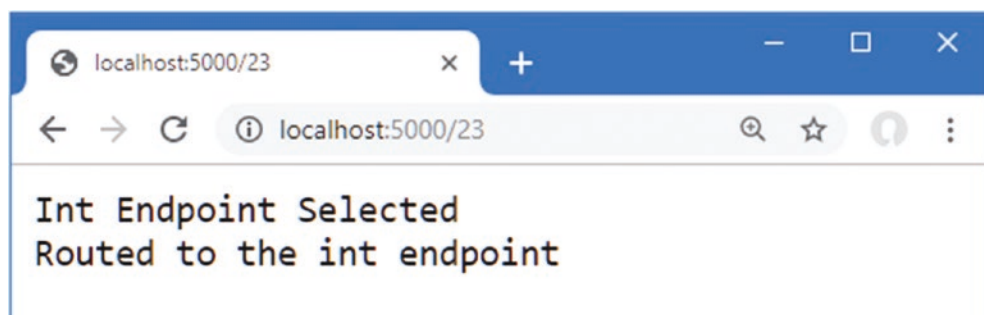
Figure 13-22. *Determining the endpoint*

Summary

In this chapter, I introduced the endpoint routing system and explained how it deals with some common problems arising in regular middleware. I showed you how to define routes, how to match and generate URLs, and how to use constraints to restrict the use of routes. I also showed you some of the advanced uses of the routing system, including custom constraints and avoiding route ambiguity. In the next chapter, I explain how ASP.NET Core services work.