CHAPTER 15

URL Routing

Early versions of ASP.NET assumed that there was a direct relationship between requested URLs and the files on the server hard disk. The job of the server was to receive the request from the browser and deliver the output from the corresponding file. This approach worked just fine for Web Forms, where each ASPX page is both a file and a self-contained response to a request.

It *doesn't* make sense for an MVC application, where requests are processed by action methods in controller classes and there is no one-to-one correlation to the files on the disk.

To handle MVC URLs, the ASP.NET platform uses the *routing system*, which has been overhauled for ASP.NET Core. In this chapter, I will show you how to use the routing system to create powerful and flexible URL handling for your projects. As you will see, the routing system lets you create any pattern of URLs you desire and express them in a clear and concise manner. The routing system has two functions.

- Examine an *incoming URL* and select the controller and action to handle the request.
- Generate *outgoing URLs*. These are the URLs that appear in the HTML rendered from views so that a specific action will be invoked when the user clicks the link (at which point it becomes an incoming URL again).

In this chapter, I will focus on defining routes and using them to process incoming URLs so that the user can reach the controllers and actions. There are two ways to create routes in an MVC application: *convention-based routing* and *attribute routing*. I explain both approaches in this chapter.

Then, in the next chapter, I will show you how to use those same routes to generate the outgoing URLs you will need to include in your views, as well as show you how to customize the routing system and use a related feature called *areas*. Table 15-1 puts routing into context.

Table 15-1. *Putting Routing in Context*

Question	Answer
What is it?	The routing system is responsible for processing incoming requests and selecting controllers and action methods to process them. The routing system is also used to generate routes in views, known as *outgoing URLs*.
Why is it useful?	The routing system allows requests to be handled flexibly without URLs being tied to the structure of classes in the Visual Studio project.
How is it used?	The mapping between URLs and the controllers and action methods is defined in the `Startup.cs` file or by applying the `Route` attribute to controllers.
Are there any pitfalls or limitations?	The routing configuration for a complex application can become hard to manage.
Are there any alternatives?	No. The routing system is an integral part of ASP.NET Core.

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_15

Table 15-2 summarizes the chapter.

Table 15-2. *Chapter Summary*

Problem	Solution	Listing
Map between URLs and action methods	Define a route	9
Allow URL segments to be omitted	Define default values for route segments	10–12
Match URL segments that don't have corresponding routing variables	Define static segments	13–16
Pass URL segments to action methods	Define custom segment variables	17–19
Allow URL segments for which there are no default values to be omitted	Define optional segments	20–21
Define routes that match any number of URL segments	Use a catchall segment	22–23
Restrict the URLs that a route can match	Apply route constraints	24–33
Define a route within a controller	Use attribute routing	34–38

Preparing the Example Project

For this chapter, I used the ASP.NET Core Web Application (.NET Core) template to create a new Empty project called UrlsAndRoutes. To add support for the MVC Framework, developer error pages, and static files, I add the statements shown in Listing 15-1 to the `Startup` class.

Listing 15-1. Configuring the Application in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc();
        }
    }
}
```

Creating the Model Class

All the effort in this chapter is about matching request URLs to actions. The only model class I need passes details about the controller and action method that has been selected to process a request. I created the Models folder and added a class file called Result.cs, which I used to define the class shown in Listing 15-2.

Listing 15-2. The Contents of the Result.cs File in the Models Folder

```
using System.Collections.Generic;

namespace UrlsAndRoutes.Models {

    public class Result {
        public string Controller { get; set; }

        public string Action { get; set; }

        public IDictionary<string, object> Data { get; }
            = new Dictionary<string, object>();
    }
}
```

The Controller and Action properties will be used to indicate how a request has been processed, and the Data dictionary will be used to store other details about the request produced by the routing system.

Creating the Example Controllers

I need some simple controllers to demonstrate how routing works. I created the Controllers folder and added a class file called HomeController.cs, the contents of which are shown in Listing 15-3.

Listing 15-3. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });
    }
}
```

The Index action method defined by the Home controller calls the View method to render a view called Result (which I define in the next section) and provides a Result object as the model object. The properties of the model object are set using the nameof function and will be used to indicate which controller and action method have been used to service a request.

I followed the same pattern by adding a CustomerController.cs file to the Controllers folder and using it to define the Customer controller shown in Listing 15-4.

Listing 15-4. The Contents of the CustomerController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });

        public ViewResult List() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(List)
            });
    }
}
```

The third and final controller is defined in a file called AdminController.cs, which I added to the Controllers folder, as shown in Listing 15-5. It follows the same pattern as the other controllers.

Listing 15-5. The Contents of the AdminController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class AdminController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(AdminController),
                Action = nameof(Index)
            });
    }
}
```

Creating the View

I specified the Result view in all the action methods defined in the previous section, which allows me to create one view that will be shared by all the controllers. I created the Views/Shared folder and added a new view called Result.cshtml to it, the contents of which are shown in Listing 15-6.

Listing 15-6. The Contents of the Result.cshtml File in the Views/Shared Folder

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Routing</title>
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
    <table class="table table-bordered table-striped table-sm">
        <tr><th>Controller:</th><td>@Model.Controller</td></tr>
        <tr><th>Action:</th><td>@Model.Action</td></tr>
        @foreach (string key in Model.Data.Keys) {
            <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
        }
    </table>
</body>
</html>
```

The view contains a table that displays the properties from the model object in a table that is styled using Bootstrap. To add Bootstrap to the project, I used the Bower Configuration File item template to create the `bower.json` file and added the Bootstrap package to the `dependencies` section, as shown in Listing 15-7.

Listing 15-7. Adding the Bootstrap Package in the bower.json File in the UrlsAndRoutes Folder

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

The final preparation is to create the `_ViewImports.cshtml` file in the `Views` folder, which sets up the built-in tag helpers for use in Razor views and imports the model namespace, as shown in Listing 15-8.

Listing 15-8. The Contents of the _ViewImports.cshtml File in the Views Folder

```
@using UrlsAndRoutes.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The configuration in the `Startup` class doesn't contain any instructions for how MVC should map HTTP requests to controllers and actions. When you start the application, any URL that you request will result in a `404 - Not Found` response, as shown in Figure 15-1.

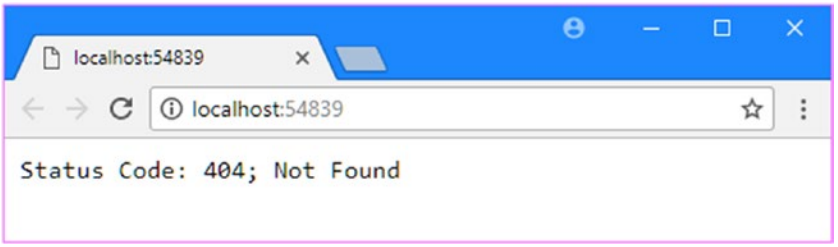

Figure 15-1. *Running the example application*

Introducing URL Patterns

The routing system works its magic using a set of *routes*. These routes collectively comprise the URL *schema* or *scheme* for an application, which is the set of URLs that your application will recognize and respond to.

I do not need to manually type out all of the individual URLs I am willing to support in my application. Instead, each route contains a *URL pattern*, which is compared to incoming URLs. If an incoming URL matches the pattern, then it is used by the routing system to process that URL. Here is a simple URL to get started with:

```
http://mysite.com/Admin/Index
```

URLs can be broken down into *segments*. These are the parts of the URL, excluding the hostname and query string, that are separated by the / character. In the example URL, there are two segments, as shown in Figure 15-2.

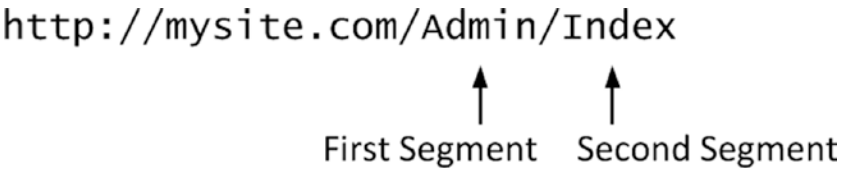

Figure 15-2. *The segments in an example URL*

The first segment contains the word `Admin`, and the second segment contains the word `Index`. To the human eye, it is obvious that the first segment relates to the controller and the second segment relates to the action. But, of course, I need to express this relationship using a URL pattern that can be understood by the routing system. Here is a URL pattern that matches the example URL:

```
{controller}/{action}
```

When processing an incoming HTTP request, the job of the routing system is to match the URL that has been requested to a pattern and extract values from the URL for the *segment variables* defined in the pattern.

The segment variables are expressed using braces (the { and } characters). The example pattern has two segment variables with the names `controller` and `action`, so the value of the `controller` segment variable will be `Admin`, and the value of the `action` segment variable will be `Index`.

An MVC application will usually have several routes, and the routing system will compare the incoming URL to the URL pattern of each route until it finds a match. By default, a pattern will match any URL that has the correct number of segments. For example, the pattern `{controller}/{action}` will match any URL that has two segments, as described in Table 15-3.

Table 15-3. *Matching URLs*

Request URL	Segment Variables
`http://mysite.com/Admin/Index`	`controller` = `Admin` `action` = `Index`
`http://mysite.com/Admin`	No match—too few segments
`http://mysite.com/Admin/Index/Soccer`	No match—too many segments

Table 15-3 highlights two key behaviors of URL patterns.

- URL patterns are *conservative* about the number of segments they match. They will match only URLs that have the same number of segments as the pattern. You can see this in the second and third examples in the table.
- URL patterns are *liberal* about the contents of segments they match. If a URL has the correct number of segments, the pattern will extract the value of each segment for a segment variable, whatever it might be.

These are the default behaviors, which are the keys to understanding how URL patterns function. I show you how to change the defaults later in this chapter.

Creating and Registering a Simple Route

Once you have a URL pattern in mind, you can use it to define a route. Routes are defined in the `Startup.cs` file and are passed as arguments to the `UseMvc` method that is used to set up MVC in the `Configure` method. Listing 15-9 shows a basic route that maps requests to the controllers in the example application.

Listing 15-9. Defining a Basic Route in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

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

```
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "default", template: "{controller}/{action}");
            });
        }
    }
}
```

Routes are created using a lambda expression passed as an argument to the `UseMvc` configuration method. The expression receives an object that implements the `IRouteBuilder` interface from the `Microsoft.AspNetCore.Routing` namespace, and routes are defined using the `MapRoute` extension method. To make routes easier to understand, the convention is to specify argument names when calling the `MapRoute` method, which is why I have explicitly named the `name` and `template` arguments in the listing. The `name` argument specified a name for a route, and the `template` argument is used to define the pattern.

■ **Tip** Naming your routes is optional, and there is a philosophical argument that doing so sacrifices some of the clean separation of concerns that otherwise comes from routing. I explain why this can be a problem in the "Generating a URL from a Specific Route" section in Chapter 16.

You can see the effect of the changes I made to the routing by starting the example application. There is no change when the application first starts—you will still see a 404 error—but if you navigate to a URL that matches the `{controller}/{action}` pattern, you will see a result like the one shown in Figure 15-3, which illustrates the effect of navigating to `/Admin/Index`.

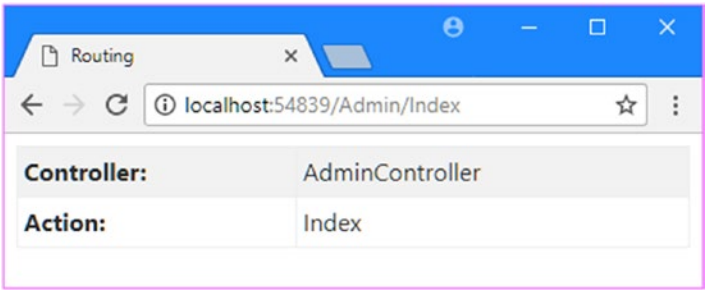

Figure 15-3. *Navigating using a simple route*

The reason that the root URL for the application doesn't work is that the route that I added to the `Startup.cs` file doesn't tell MVC how to select a controller class and action method when the requested URL has no segments. I'll fix this in the next section.

Defining Default Values

The example application returns a 404 message when the default URL is requested because it didn't match the pattern of the route defined in the Startup class. Since there are no segments in the default URL that can be matched to the controller and action variables defined by the routing pattern, the routing system doesn't make a match.

I explained earlier that URL patterns will match only URLs with the specified number of segments. One way to change this behavior is to use *default values*. A default value is applied when the URL doesn't contain a segment that can be matched by the routing pattern. Listing 15-10 defines a route that uses a default value.

Listing 15-10. Providing a Default Value in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller}/{action}",
                    defaults: new { action = "Index" });
            });
        }
    }
}
```

Default values are supplied as properties in an anonymous type, passed to the MapRoute method as the defaults argument. In the listing, I provided a default value of Index for the action variable.

This route will match all two-segment URLs, as it did previously. For example, if the URL http://mydomain.com/Home/Index is requested, the route will extract Home as the value for the controller and will extract Index as the value for the action.

But now that there is a default value for the action segment, the route will *also* match single-segment URLs. When processing a single-segment URL, the routing system will extract the controller value from the URL and use the default value for the action variable. In this way, the user can request /Home and MVC will invoke the Index action method on the Home controller, as shown in Figure 15-4.

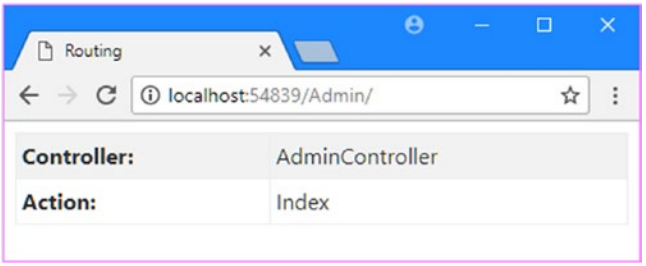

Figure 15-4. *Using a default action*

Defining Inline Default Values

Default values can also be expressed as part of the URL pattern, which is a more concise way to express routes, as shown in Listing 15-11. The inline syntax can be used only to provide defaults for variables that are part of the URL pattern, but, as you will learn, it is often useful to be able to provide defaults outside of that pattern. For this reason, it is useful to understand both ways of expressing defaults.

Listing 15-11. Defining Inline Default Values in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller}/{action=Index}");
            });
        }
    }
}
```

I can go further and match URLs that do not contain any segment variables at all, relying on just the default values to identify the action and controller. And as an example, Listing 15-12 shows how I have mapped the root URL for the application by providing default values for both segments.

Listing 15-12. Providing Default Values in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                template: "{controller=Home}/{action=Index}");
            });
        }
    }
}
```

By providing default values for both the `controller` and `action` variables, the route will match URLs that have zero, one, or two segments, as shown in Table 15-4.

Table 15-4. *Matching URLs*

Segments	Example	Maps To
0	**/**	`controller` = `Home` `action` = `Index`
1	**/Customer**	`controller` = `Customer` `action` = `Index`
2	**/Customer/List**	`controller` = `Customer` `action` = `List`
3	**/Customer/List/All**	No match—too many segments

The fewer segments received in the incoming URL, the more the route relies on the default values, up until the point where a URL with no segments is matched using only default values.

You can see the effect of the default values by starting the example app. When the browser requests the root URL for the application, the default values for the `controller` and `action` segment variables will be used, which will lead MVC to invoke the `Index` action method on the `Home` controller, as shown in Figure 15-5.

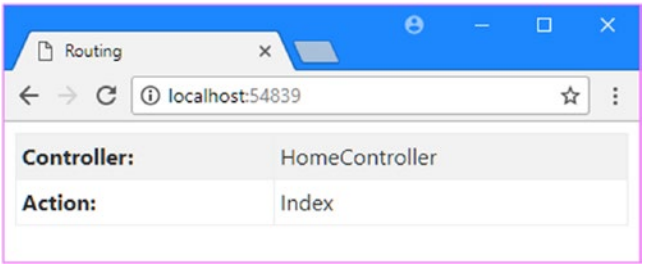

Figure 15-5. *Using default values to broaden the scope of a route*

Using Static URL Segments

Not all the segments in a URL pattern need to be variables. You can also create patterns that have *static segments*. Suppose that the application needs to match URLs that are prefixed with `Public`, like this:

```
http://mydomain.com/Public/Home/Index
```

This can be done by using a URL pattern like the one shown in Listing 15-13.

Listing 15-13. Using Static Segments in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
```

```
                    name: "default",
                    template: "{controller=Home}/{action=Index}");

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

This new pattern will match only URLs that contain three segments, the first of which *must* be `Public`. The other two segments can contain any value and will be used for the `controller` and `action` variables. If the last two segments are omitted, then the default values will be used.

You can also create URL patterns that have segments containing both static and variable elements, such as the one shown in Listing 15-14.

Listing 15-14. Mixing Segments in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute("", "X{controller}/{action}");

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

                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}/{action=Index}");

            });
        }
    }
}
```

The pattern in this route matches any two-segment URL where the first segment starts with the letter X. The value for `controller` is taken from the first segment, excluding the X. The `action` value is taken from the second segment. You can see the effect of this route if you start the application and navigate to `/XHome/Index`, the result of which is illustrated in Figure 15-6.

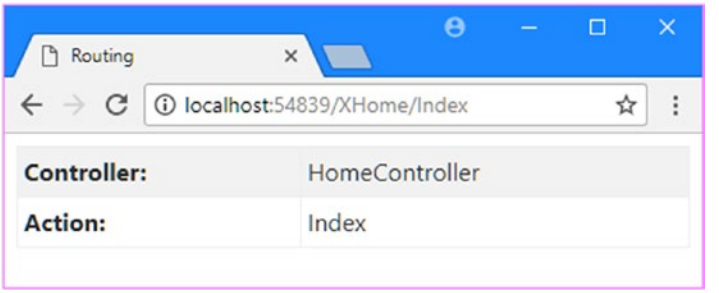

Figure 15-6. *Mixing static and variable elements in a single segment*

ROUTE ORDERING

In Listing 15-14, I defined a new route and placed it before all the others. I did this because routes are applied in the order in which they are defined. The `MapRoute` method adds a route to the end of the routing configuration, which means that routes are generally applied in the order in which they are defined. I say "generally" because there are methods that insert routes in specific locations. I tend not to use these methods because having routes applied in the order in which they are defined makes understanding the routing for an application simpler.

The routing system tries to match an incoming URL against the URL pattern of the route that was defined first and proceeds to the next route only if there is no match. The routes are tried in sequence until a match is found or the set of routes has been exhausted. As a consequence, the most specific routes must be defined first. The route I added in Listing 15-14 is more specific than the route that follows. Suppose that I reversed the order of the routes, like this:

```
...
routes.MapRoute("MyRoute", "{controller=Home}/{action=Index}");
routes.MapRoute("", "X{controller}/{action}");
...
```

Then the first route, which matches *any* URL with zero, one, or two segments, will always be the one that is used. The more specific route, which is now second in the list, will never be reached. The new route excludes the leading X of a URL, but this won't be done by the older route. Therefore, a URL such as this:

```
http://mydomain.com/XHome/Index
```

will be targeted to a controller called `XHome`, assuming that there is an `XHomeController` class in the application and it has an action method called `Index`.

Static URL segments and default values can be combined to create an alias for a specific URL. The URL schema that you use forms a contract with your users when you deploy your application, and if you subsequently refactor an application, you need to preserve the previous URL format so that any URL favorites, macros, or scripts the user has created continue to work.

Imagine that there used to be a controller called `Shop`, which has now been replaced by the `Home` controller. Listing 15-15 shows how I can create a route to preserve the old URL schema.

Listing 15-15. Segments and Default Values in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "ShopSchema",
                    template: "Shop/{action}",
                    defaults: new { controller = "Home" });

                routes.MapRoute("", "X{controller}/{action}");

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

                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}/{action=Index}");

            });
        }
    }
}
```

The route matches any two-segment URL where the first segment is `Shop`. The `action` value is taken from the second URL segment. The URL pattern doesn't contain a variable segment for `controller`, so the default value is used. The `defaults` argument provides the `controller` value because there is no segment to which the value can be applied to as part of the URL pattern.

The result is that a request for an action on the Shop controller is translated to a request for the Home controller. You can see the effect of this route by starting the app and navigating to the /Shop/Index URL. As Figure 15-7 shows, the new route causes MVC to target the Index action method in the Home controller.

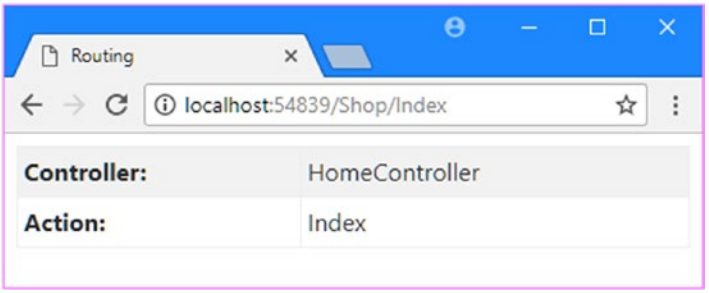

Figure 15-7. *Creating an alias to preserve URL schemas*

I can go one step further and create aliases for action methods that have been refactored away as well and are no longer present in the controller. To do this, I create a static URL and provide the controller and action values as defaults, as shown in Listing 15-16.

Listing 15-16. Aliasing a Controller and an Action in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

                routes.MapRoute(
                    name: "ShopSchema2",
                    template: "Shop/OldAction",
                    defaults: new { controller = "Home", action = "Index" });
```

```
                routes.MapRoute(
                    name: "ShopSchema",
                    template: "Shop/{action}",
                    defaults: new { controller = "Home" });

                routes.MapRoute("", "X{controller}/{action}");

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

                routes.MapRoute(name: "",
                    template: "Public/{controller=Home}/{action=Index}");

            });
        }
    }
}
```

Notice that the new route is defined first because it is more specific than the routes that follow. If a request for `Shop/OldAction` were processed by the next defined route, for example, I may get a different result from the one I want if there is a controller with an `OldAction` action method.

Defining Custom Segment Variables

The `controller` and `action` segment variables have special meaning in MVC applications and correspond to the controller and action method that will be used to service the request. These are only the built-in segment variables, and custom segment variables can also be defined, as shown in Listing 15-17. (I have removed the existing routes from the previous section so I can start over).

Listing 15-17. Defining Additional Variables in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
```

```
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller=Home}/{action=Index}/{id=DefaultId}");
            });
        }
    }
}
```

The URL pattern defines the standard `controller` and `action` variables, as well as a custom variable called `id`. This route will match any zero-to-three-segment URL. The contents of the third segment will be assigned to the `id` variable, and if there is no third segment, the default value will be used.

■ **Caution** Some names are reserved and not available for custom segment variable names. These are `controller`, `action`, `area`, and `page`. The meaning of the first two is obvious. I explain *areas* in the next chapter, and `page` is used by the Razor Pages feature.

The `Controller` class, which is the base for controllers, defines a `RouteData` property that returns a `Microsoft.AspNetCore.Routing.RouteData` object that provides details about the routing system and the way that the current request has been routed. Within a controller, I can access any of the segment variables in an action method by using the `RouteData.Values` property, which returns a dictionary containing the segment variables. To demonstrate, I have added an action method to the `Home` controller called `CustomVariable`, as shown in Listing 15-18.

Listing 15-18. Accessing a Segment Variable in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });

        public ViewResult CustomVariable() {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = RouteData.Values["id"];
            return View("Result", r);
        }
    }
}
```

This action method obtains the value of the custom `id` variable in the route URL pattern using the `RouteData.Values` property, which returns a dictionary of the variables produced by the routing system. The custom variable is added to the view model object and can be seen by running the application and requesting the following URL:

```
/Home/CustomVariable/Hello
```

The routing template matches the third segment in this URL as the value for the `id` variable, producing the results shown in Figure 15-8.

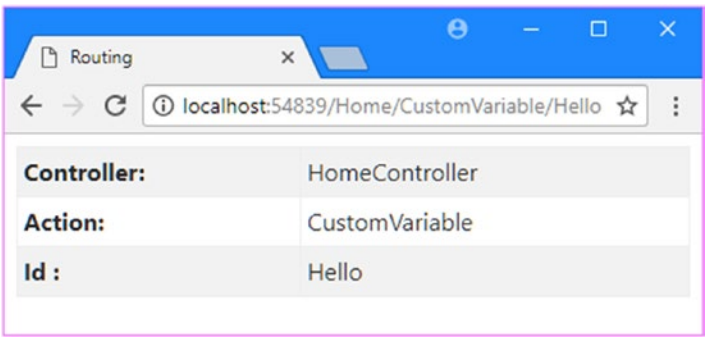

Figure 15-8. *Displaying the value of a custom segment variable*

The URL pattern in Listing 15-17 defines a default value for the `id` segment, which means that the route can also match URLs that have two segments. You can see the use of the default value by requesting this URL:

```
/Home/CustomVariable
```

The routing system uses the default value for the custom variable, as shown in Figure 15-9.

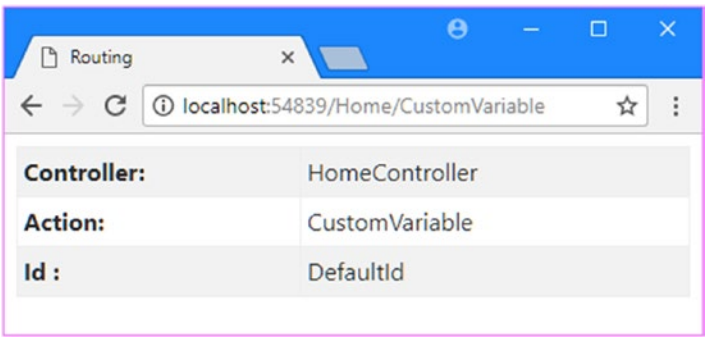

Figure 15-9. *The default value for a custom segment variable*

Using Custom Variables as Action Method Parameters

Using the `RouteData.Values` collection is only one way to access custom route variables, and the other way can be more elegant. If an action method defines parameters with names that match the URL pattern variables, MVC will automatically pass the values obtained from the URL as arguments to the action method.

The custom variable defined in the route in Listing 15-17 is called `id`. I can modify the `CustomVariable` action method in the `Home` controller so that it has a parameter of the same name, as shown in Listing 15-19.

Listing 15-19. Adding an Action Parameter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });

        public ViewResult CustomVariable(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = id;
            return View("Result", r);
        }
    }
}
```

When the routing system matches a URL against the route defined in Listing 15-17, the value of the third segment in the URL is assigned to the custom variable `id`. MVC compares the list of segment variables with the list of action method parameters and, if the names match, passes the values from the URL to the method.

The type of the `id` parameter is a `string`, but MVC will try to convert the URL value to whatever parameter type is used. If the action method declared the `id` parameter as an `int` or a `DateTime`, then it would receive the value from the URL converted to an instance of that type. This is an elegant and useful feature that removes the need for me to handle the conversion myself. You can see the effect of the action method parameter by starting the application and requesting `/Home/CustomVariable/Hello`, which produces the result shown in Figure 15-10. If you omit the third segment, then the action method will be provided with the default segment value, which is also shown in the figure.

■ **Note** MVC uses the *model binding* feature to convert the values contained in the URL to .NET types, and model binding can handle much more complex situations than shown in this example. I describe model binding in Chapter 26.

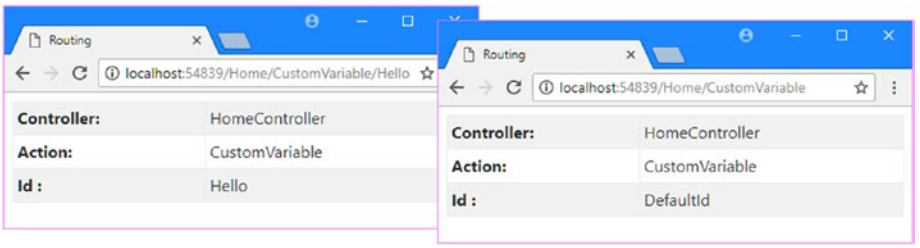

Figure 15-10. *Accessing segment variables using action method parameters*

Defining Optional URL Segments

An *optional* URL segment is one that the user does not need to specify and for which no default value is specified. An optional segment is denoted by a question mark (the ? character) after the segment name, as shown in Listing 15-20.

Listing 15-20. Specifying an Optional Segment in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

This route will match URLs whether or not the `id` segment has been supplied. Table 15-5 shows how this works for different URLs.

Table 15-5. *Matching URLs with an Optional Segment Variable*

Segments	Example URL	Maps To
0	**/**	controller = Home action = Index
1	**/Customer**	controller = Customer action = Index
2	**/Customer/List**	controller = Customer action = List
3	**/Customer/List/All**	controller = Customer action = List id = All
4	**/Customer/List/All/Delete**	No match—too many segments

As you can see from the table, the id variable is added to the set of variables only when there is a corresponding segment in the incoming URL. This feature is useful if you need to know whether the user supplied a value for a segment variable. When no value has been supplied for an optional segment variable, the value of the corresponding parameter will be null. I have updated the Home controller to respond when no value is provided for the id segment variable in Listing 15-21.

Listing 15-21. Checking for a Segment in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(HomeController),
                Action = nameof(Index)
            });

        public ViewResult CustomVariable(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(CustomVariable),
            };
            r.Data["Id"] = id ?? "<no value>";
            return View("Result", r);
        }
    }
}
```

Figure 15-11 shows the result of starting the application and navigating to the /Home/CustomVariable URL, which doesn't include a value for the id segment variable.

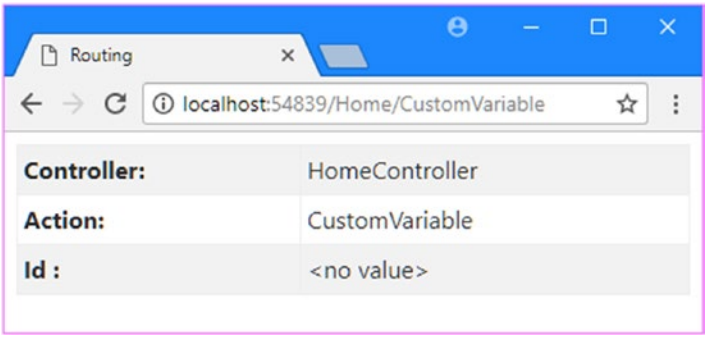

Figure 15-11. *Detecting when a URL doesn't contain a value for an optional segment variable*

UNDERSTANDING THE DEFAULT ROUTING CONFIGURATION

When you add MVC to the `Startup` class, you can do so using the `UseMvcWithDefaultRoute` method. This is just a convenience method for setting up the most common routing configuration and is equivalent to the following code:

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

This default configuration matches URLs that target controller classes and action method by name, with an optional `id` segment. If the `controller` or `action` segments are missing, then default values are used to target the `Home` controller and the `Index` action method, respectively.

Defining Variable-Length Routes

Another way of changing the default conservatism of URL patterns is to accept a variable number of URL segments. This allows you to route URLs of arbitrary lengths in a single route. You define support for variable segments by designating one of the segment variables as a *catchall*, done by prefixing it with an asterisk (the * character), as shown in Listing 15-22.

Listing 15-22. Designating a Catchall Variable in the Startup.cs File in the UrlsAndRoutes Folder

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

namespace UrlsAndRoutes {

    public class Startup {
```

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

        public void Configure(IApplicationBuilder app) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=Index}/{id?}/{*catchall}");
            });
        }
    }
}
```

I have extended the route from the previous example to add a catchall segment variable, which I imaginatively called `catchall`. This route will now match *any* URL, irrespective of the number of segments it contains or the value of any of those segments. The first three segments are used to set values for the `controller`, `action`, and `id` variables, respectively. If the URL contains additional segments, they are all assigned to the `catchall` variable, as shown in Table 15-6.

Table 15-6. *Matching URLs with a Catchall Segment Variable*

Segments	Example URL	Maps To
0	**/**	`controller` = `Home` `action` = `Index`
1	**/Customer**	`controller` = `Customer` `action` = `Index`
2	**/Customer/List**	`controller` = `Customer` `action` = `List`
3	**/Customer/List/All**	`controller` = `Customer` `action` = `List` `id` = `All`
4	**/Customer/List/All/Delete**	`controller` = `Customer` `action` = `List` `id` = `All` `catchall` = `Delete`
5	**/Customer/List/All/Delete/Perm**	`controller` = `Customer` `action` = `List` `id` = `All` `catchall` = `Delete/Perm`

In Listing 15-23, I have updated the `Customer` controller so that the `List` action passes the value of the `catchall` variable to the view via the model object.

Listing 15-23. Updating an Action in the CustomerController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });
```

```
        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["Id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catchall"];
            return View("Result", r);
        }
    }
}
```

To test the catchall segment, run the application and request the following URL:

```
/Customer/List/Hello/1/2/3
```

There is no upper limit to the number of segments that the URL pattern in this route will match. Figure 15-12 shows the effect of the catchall segment. Notice that the segments captured by the catchall are presented in the form *segment/segment/segment* and that I am responsible for processing the string to break out the individual segments.

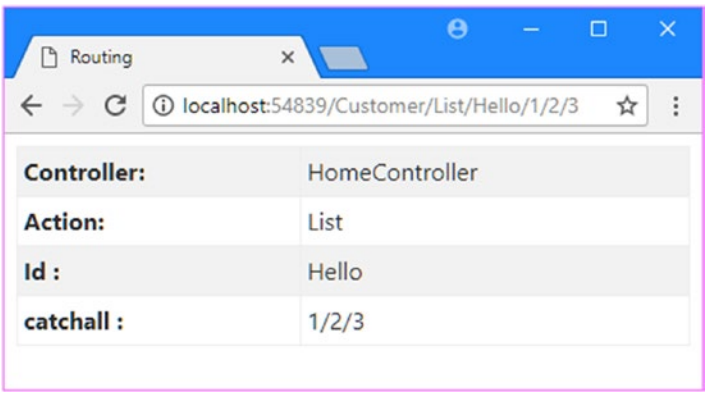

Figure 15-12. *Using a catchall segment*

Constraining Routes

At the start of the chapter, I described how URL patterns are conservative when they match the number of segments in the URL and liberal when they match the content of segments. The previous few sections have explained different techniques for controlling the degree of conservatism: making a route match more or fewer segments using default values, optional variables, and so on.

It is now time to look at how to control the liberalism in matching the *content* of URL segments, namely, how to restrict the set of URLs that a route will match against. Listing 15-24 demonstrates the use of a simple constraint that limits the URLs that a route will match.

Listing 15-24. Constraining a Route in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
```

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

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=Index}/{id:int?}");
            });
        }
    }
}
```

Constraints are separated from the segment variable name with a colon (the `:` character). The constraint in the listing is `int`, and it has been applied to the `id` segment. This is an example of an *inline constraint*, which is defined as part of the URL pattern applied to a single segment:

```
...
template: "{controller}/{action}/{id:int?}",
...
```

The `int` constraint only allows the URL pattern to match segments whose value can be parsed to an `integer` value. The `id` segment is optional, so the route will match segments that omit the `id` segment, but if the segment is present, then it must be an integer value, as summarized in Table 15-7.

Table 15-7. *Matching URLs with a Constraint*

Example URL	Maps To

/	`controller` = `Home`
	`action` = `Index`
	`id = null`

/Home/CustomVariable/Hello	No match—`id` segment cannot be parsed to an `int` value.

/Home/CustomVariable/1	`controller` = `Home`
	`action` = `CustomVariable`
	`id = 1`

/Home/CustomVariable/1/2	No match—too many segments

Constraints can also be specified outside of the URL pattern, using the `constraints` argument to the `MapRoute` method when defining a route. This technique is useful if you prefer to keep the URL pattern separate from its constraints or if you prefer to follow the routing style used by earlier versions of MVC, which did not support inline constraints. Listing 15-25 shows the same `integer` constraint on the `id` segment variable, expressed using a separate constraint. When using this format, the default values are also expressed externally.

Listing 15-25. Expressing a Constraint in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller}/{action}/{id?}",
                defaults: new { controller = "Home", action = "Index" },
                constraints: new { id = new IntRouteConstraint() });
            });
        }
    }
}
```

The `constraints` argument to the `MapRoute` method is defined using an anonymous type whose property names correspond to the segment variable being constrained. The `Microsoft.AspNetCore.Routing.Constraints` namespace contains a set of classes that can be used to define individual constraints. In Listing 15-25, the `constraints` argument is configured to use an `IntRouteConstraint` object for the `id` segment, creating the same effect as the inline constraint shown in Listing 15-24.

Table 15-8 describes the complete set of constraint classes in the `Microsoft.AspNetCore.Routing.Constraints` namespace and their inline equivalents for the constraints that can be applied to single segments in the URL pattern, some of which I describe in the sections that follow.

■ **Tip** You can restrict access to action methods to requests made with specific HTTP verbs, such as `GET` or `POST`, using a set of attributes provided by MVC, such as the `HttpGet` and `HttpPost` attributes. See Chapter 7 for details of using these attributes to handle forms in controllers, and see Chapter 20 for a full list of the attributes available.

Table 15-8. *Segment-Level Route Constraints*

Inline Constraint	Description	Class Name
`alpha`	Matches alphabet characters, irrespective of case (A–Z, a–z)	`AlphaRouteConstraint()`
`bool`	Matches a value that can be parsed into a `bool`	`BoolRouteConstraint()`
`datetime`	Matches a value that can be parsed into a `DateTime`	`DateTimeRouteConstraint()`
`decimal`	Matches a value that can be parsed into a `decimal`	`DecimalRouteConstraint()`
`double`	Matches a value that can be parsed into a `double`	`DoubleRouteConstraint()`
`float`	Matches a value that can be parsed into a `float`	`FloatRouteConstraint()`
`guid`	Matches a value to a globally unique identifier	`GuidRouteConstraint()`
`int`	Matches a value that can be parsed into an `int`	`IntRouteConstraint()`
`length(len)` `length(min, max)`	Matches a value with the specified number of characters or that is between `min` and `max` characters in length (inclusive)	`LengthRouteConstraint(len)` `LengthRouteConstraint (min, max)`
`long`	Matches a value that can be parsed into a `long`	`LongRouteConstraint()`
`maxlength(len)`	Matches a string with no more than `len` characters	`MaxLengthRouteConstraint(len)`
`max(val)`	Matches an `int` value if the value is less than `val`	`MaxRouteConstraint(val)`
`minlength(len)`	Matches a string with at least `len` characters	`MinLengthRouteConstraint(len)`
`min(val)`	Matches an `int` value if the value is more than `val`	`MinRouteConstraint(val)`
`range(min, max)`	Matches an `int` value if the value is between `min` and `max` (inclusive)	`RangeRouteConstraint (min, max)`
`regex(expr)`	Matches a regular expression	`RegexRouteConstraint(expr)`

Constraining a Route Using a Regular Expression

The constraint that offers the most flexibility is `regex`, which matches a segment using a regular expression. In Listing 15-26, I have constrained the controller segment to limit the range of URLs that it will match.

Listing 15-26. Using a Regular Expression in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller:regex(^H.*)=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

The constraint I used restricts the route so that it will only match URLs where the `controller` segment starts with the letter `H`.

■ **Note** Default values are applied before constraints are checked. So, for example, if I request the URL `/`, the default value for `controller`, which is `Home`, is applied. The constraints are then checked, and since the `controller` value begins with `H`, the default URL will match the route.

Regular expressions can constrain a route so that only specific values for a URL segment will cause a match. This is done using the bar (`|`) character, as shown in Listing 15-27. (I split the URL pattern into two so that it will fit onto the page, which you won't need to worry about in a real project).

Listing 15-27. Constraining a Route in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                  template: "{controller:regex(^H.*)=Home}/"
                    + "{action:regex(^Index$|^About$)=Index}/{id?}");
            });
        }
    }
}
```

This constraint will allow the route to match only URLs where the value of the action segment is `Index` or `About`. Constraints are applied together, so the restrictions imposed on the value of the `action` variable are combined with those imposed on the `controller` variable. This means that the route in Listing 15-27 will match URLs only when the `controller` variable begins with the letter `H` and the `action` variable is `Index` or `About`.

Using Type and Value Constraints

Most of the constraints are used to restrict routes so they only match URLs with segments that can be converted to specified types or have a specific format. The `int` constraint I used at the start of this section is a good example: it will match routes only when the value of the constrained segment can be parsed to a .NET `int` value. Listing 15-28 demonstrates the use of the `range` constraint, which restricts a route so that it matches URLs only when a segment value can be converted to an `int` and falls between specified values.

Listing 15-28. Constraining Based on Type and Value in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
```

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

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=Index}/{id:range(10,20)?}");
            });
        }
    }
}
```

The constraint in this example has been applied to the optional `id` segment. The constraint will be ignored if the request URL doesn't have at least three segments. If the `id` segment is present, the route will match the URL only if the segment value can be converted to an `int` and the value is between 10 and 20. The range constraint is inclusive, meaning that values of 10 and 20 are considered to be within the range.

Combining Constraints

If you need to apply multiple constraints to a single segment, then you chain them together so that each constraint is separated by a colon, as shown in Listing 15-29.

Listing 15-29. Combining Inline Constraints in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
    public class Startup {

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

```
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller=Home}/{action=Index}"
                        + "/{id:alpha:minlength(6)?}");
            });
        }
    }
}
```

In this listing, I have applied both the `alpha` and `minlength` constraints to the `id` segment. The question mark that denotes an optional segment is applied after all of the constraints. The effect of combining these constraints is that the route will match URLs only where the `id` segment is omitted (because it is optional) or when it is present and contains at least six alphabet characters.

If you are not using inline constraints, then you must use the `Microsoft.AspNetCore.Routing.CompositeRouteConstraint` class, which allows multiple constraints to be associated with a single property in an anonymously typed object. Listing 15-30 shows the combination of constraints that I used in Listing 15-29.

Listing 15-30. Combining Separate Constraints in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller}/{action}/{id?}",
                    defaults: new { controller = "Home", action = "Index" },
                    constraints: new {
                        id = new CompositeRouteConstraint(
                            new IRouteConstraint[] {
```

```
                        new AlphaRouteConstraint(),
                        new MinLengthRouteConstraint(6)
                    })
                });
            });
        }
    }
}
```

The constructor for the `CompositeRouteConstraint` class accepts an enumeration of objects that implement the `IRouteConstraint` objects, which is the interface that defines route constraints. The routing system will allow the route to match a URL only if all the constraints are satisfied.

Defining a Custom Constraint

If the standard constraints are not sufficient for your needs, you can define your own custom constraints by implementing the `IRouteConstraint` interface, which is defined in the `Microsoft.AspNetCore.Routing` namespace. To demonstrate this feature, I added an `Infrastructure` folder to the example project and created a new class file called `WeekDayConstraint.cs`, the contents of which are shown in Listing 15-31.

Listing 15-31. The Contents of the WeekDayConstraint.cs File in the Infrastructure Folder

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

namespace UrlsAndRoutes.Infrastructure {
    public class WeekDayConstraint : IRouteConstraint {
        private static string[] Days = new[] { "mon", "tue", "wed", "thu",
                                               "fri", "sat", "sun" };

        public bool Match(HttpContext httpContext, IRouter route,
            string routeKey, RouteValueDictionary values,
            RouteDirection routeDirection) {

            return Days.Contains(values[routeKey]?.ToString().ToLowerInvariant());
        }
    }
}
```

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 access to the request from the client, the route, the name of the segment that is being constrained, the segment variables that have been extracted from the URL, and whether the request is to check for an incoming or outgoing URL (I explain outgoing URLs in Chapter 16).

In the example, I use the `routeKey` parameter to get the value of the segment variable to which the constraint has been applied from the `values` parameter, convert it to a lowercase string, and see whether it matches one of the days of the week that are defined in the static `Days` field. Listing 15-32 applies the new constraint to the example route using the separate technique.

Listing 15-32. Applying a Custom Constraint in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                    template: "{controller}/{action}/{id?}",
                    defaults: new { controller = "Home", action = "Index" },
                    constraints: new { id = new WeekDayConstraint() });
            });
        }
    }
}
```

This route will match a URL only if the `id` segment is absent (such as `/Customer/List`) or if it matches one of the days of the week defined in the constraint class (such as `/Customer/List/Fri`).

Defining an Inline Custom Constraint

Setting up a custom constraint so that it can be used inline requires an additional configuration step, as shown in Listing 15-33.

Listing 15-33. Using a Custom Constraint Inline in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(name: "MyRoute",
                template: "{controller=Home}/{action=Index}/{id:weekday?}");
            });
        }
    }
}
```

In the `ConfigureService` method I configure the `RouteOptions` object, which controls some of the behaviors of the routing system. The `ConstraintMap` property returns the dictionary that is used to translate the names of inline constraints to the `IRouteConstraint` implementation classes that provide the constraint logic. I add a new mapping to the dictionary so that I can refer to the `WeekDayConstraint` class inline as `weekday`, like this:

```
...
template: "{controller=Home}/{action=Index}/{id:weekday?}",
...
```

The effect of the constraint is the same, but setting up the mapping allows custom classes to be used inline.

Using Attribute Routing

All the examples so far in this chapter have been defined using a technique known as *convention-based routing*. MVC also supports for a technique known as *attribute routing*, in which routes are defined by C# attributes that are applied directly to the controller classes. In the sections that follow, I show you how to create and configure routes using attributes, which can be mixed freely with the convention-based routes shown in earlier examples.

Preparing for Attribute Routing

Attribute routing is enabled when you call the `UseMvc` method in the `Startup.cs` file. MVC examines the controller classes in the application, finds any that have routing attributes, and creates routes for them.

For this section of the chapter, I have returned the example application to the default routing configuration described in the "Understanding the Default Routing Configuration" sidebar, as shown in Listing 15-34.

Listing 15-34. Using the Default Routing Configuration in the Startup.cs File in the UrlsAndRoutes Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(options =>
                options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

The default route will match URLs using the following pattern:

```
{controller}/{action}/{id?}
```

Applying Attribute Routing

The `Route` attribute is used to specify routes for individual controllers and actions. In Listing 15-35, I have applied the `Route` attribute to the `CustomerController` class.

Listing 15-35. Applying the Route Attribute in the CustomerController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("myroute")]
        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catchall"];
            return View("Result", r);
        }
    }
}
```

The `Route` attribute works by defining a route to the action method or controller it is applied to. In the listing, I applied the attribute to the `Index` action method and specified `myroute` as the route that should be used. The effect is to change the set of routes that are used to reach the action methods defined by the `Customer` controller, as described in Table 15-9.

Table 15-9. *The Routes for the Customer Controller*

Route	Description
/Customer/List	This URL targets the `List` action method, relying on the default route in the `Startup.cs` file.
/myroute	This URL targets the `Index` action method.

There are two important points to note. The first is that when you use the `Route` attribute, the value you provide to configure the attribute is used to define a complete route so that `myroute` becomes the complete URL to reach the `Index` action method. The second point to note is that using the `Route` attribute prevents the default routing configuration from being used so that the `Index` action method can no longer be reached by using the `/Customer/Index` URL.

Changing the Name of an Action Method

Defining a unique route for a single action method isn't useful in most applications, but the `Route` attribute can also be used more flexibly. In Listing 15-36, I have used the special `[controller]` token in the route to refer to the controller and set up the base section of the route.

■ **Tip** You can also change the name of an action using the `ActionName` attribute, which I describe in Chapter 31.

Listing 15-36. Renaming an Action in the CustomerController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("[controller]/MyAction")]
        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catchall"];
            return View("Result", r);
        }
    }
}
```

Using the `[controller]` token in the argument for the `Route` attribute is rather like using a `nameof` expression and allows for the route to the controller to be specified without hard-coding the class name. Table 15-10 describes the effect of the attribute in Listing 15-36.

Table 15-10. *The Routes for the Customer Controller*

Route	Description
`/Customer/List`	This URL targets the `List` action method.
`/Customer/MyAction`	This URL targets the `Index` action method.

Creating a More Complex Route

The `Route` attribute can also be applied to the controller class, allowing for the structure of the route to be defined, as shown in Listing 15-37.

Listing 15-37. Applying the Route Attribute in the CustomerController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    [Route("app/[controller]/actions/[action]/{id?}")]
    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catchall"];
            return View("Result", r);
        }
    }
}
```

This route defines mixes static segments and variable segments and uses the `[controller]` and `[action]` tokens to refer to the names of the controller class and the action methods. Table 15-11 shows the effect of the route.

Table 15-11. *The Routes for the Customer Controller*

Route	Description
`app/customer/actions/index`	This URL targets the `Index` action method.
`app/customer/actions/index/myid`	This URL targets the `Index` action method with the optional `id` segment set to `myid`.
`app/customer/actions/list`	This URL targets the `List` action method.
`app/customer/actions/list/myid`	This URL targets the `List` action method with the optional `id` segment set to `myid`.

Applying Route Constraints

Routes defined using attributes can be constrained just like those defined in the `Startup.cs` file, using the same inline technique used for convention-based routes. In Listing 15-38, I have applied the custom constraint created earlier in the chapter to the optional `id` segment defined with the `Route` attribute.

Listing 15-38. Constraining a Route in the CustomerController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

    [Route("app/[controller]/actions/[action]/{id:weekday?}")]
    public class CustomerController : Controller {

        public ViewResult Index() => View("Result",
            new Result {
                Controller = nameof(CustomerController),
                Action = nameof(Index)
            });

        public ViewResult List(string id) {
            Result r = new Result {
                Controller = nameof(HomeController),
                Action = nameof(List),
            };
            r.Data["id"] = id ?? "<no value>";
            r.Data["catchall"] = RouteData.Values["catchall"];
            return View("Result", r);
        }
    }
}
```

You can use all the constraints described in Table 15-8 or, as shown in the listing, use custom constraints that have been registered with the `RouteOptions` service. Multiple constraints can be applied by chaining them together and separating them with colons.

Summary

In this chapter, I took an in-depth look at the routing system. You have seen how routes are defined by convention or with attributes. You have seen how incoming URLs are matched and handled and how to customize routes by changing the way that they match URL segments and by using default values and optional segments. I also showed you how to constrain routes to narrow the range of requests that they will match, both using built-in constraints and using custom constraint classes.

In the next chapter, I show you how to generate outgoing URLs from routes in your views and how to use the areas feature, which relies on the routing system and which can be used to manage large and complex MVC applications.