CHAPTER 15

URL Routing

Before the introduction of the MVC Framework, 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 works 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*. 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 figure out for which controller and action the request is intended.
- 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 has become 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 your controllers and actions. There are two ways to create routes in an MVC Framework application: *convention-based routing* and *attribute routing*. You will be familiar with convention-based routing if you have used earlier versions of the MVC Framework, but attribute routing is new to MVC 5. 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 provides the summary for this chapter.

Table 15-1. *Chapter Summary*

Problem	Solution	Listing
Map between URLs and action methods	Define a route	1–8
Allow URL segments to be omitted	Define default values for segment variables	9, 10
Match URL segments that don't have corresponding routing variables	Use static segments	11–14
Pass URL segments to action methods	Define custom segment variables	15–18
Allow URL segments for which there is no default value to be omitted	Define optional segments	19–22
Define routes that match any number of URL segments	Use a catchall segment	23
Avoid controller name confusion	Specify priority namespaces in a route	24–27
Limit the URLs that a route can match	Apply a route constraint	28–34
Enable attribute routing	Call the `MapMvcAttributeRoutes` method	35
Define a route within a controller	Apply the `Route` attribute to the action methods	36, 37
Constrain an attribute route	Apply a constraint to the segment variable in the route pattern	38, 39
Define a common prefix for all of the attribute routes in a controller	Apply the `RoutePrefix` attribute to the controller class	40

Preparing the Example Project

To demonstrate the routing system, I need a project to which I can add routes. I created a new MVC application using the `Empty` template, and I called the project `UrlsAndRoutes`. I added a test project to the Visual Studio solution called `UrlsAndRoutes.Tests` by checking the `Add Unit Tests` option, as shown in Figure 15-1.

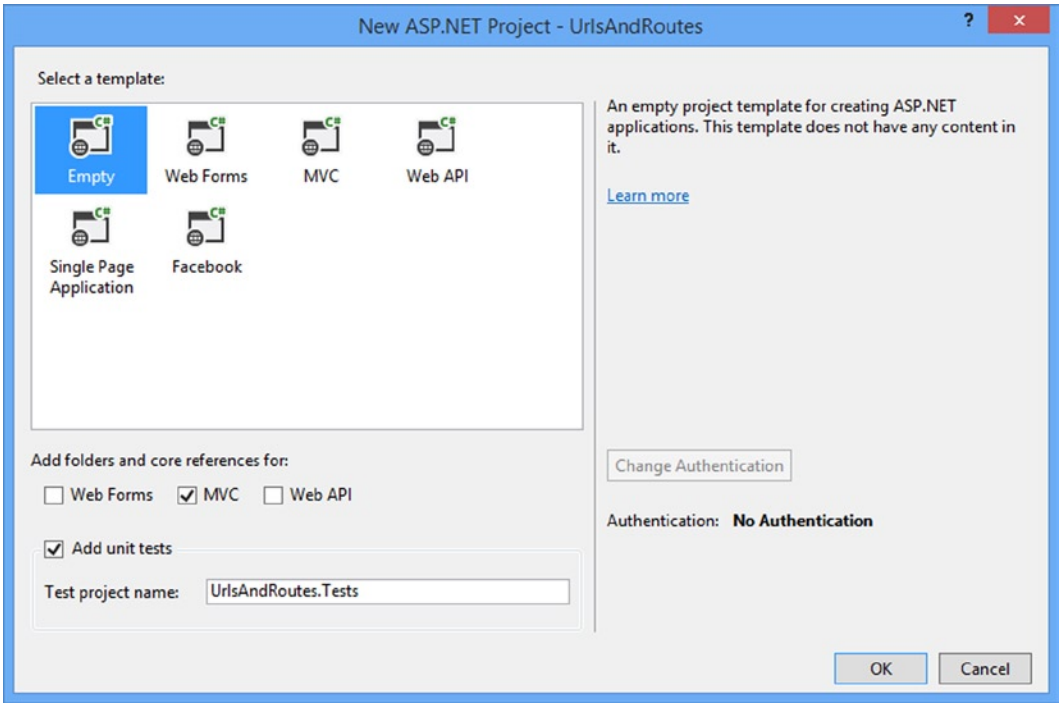

Figure 15-1. *Creating an Empty MVC application project with unit tests*

I showed you how to create the unit tests manually for the SportsStore chapter, but this produces the same result and handles the references between projects automatically. You will still need to add Moq, however, and so enter the following command in the NuGet console:

```
Install-Package Moq -version 4.1.1309.1617 -projectname UrlsAndRoutes.Tests
```

Creating the Example Controllers

To demonstrate the routing feature, I am going to add some simple controllers to the example application. I only care about the way in which URLs are interpreted in order to call action methods, so the view models I use are string values in the view bag which report the controller and action method name. First, create a Home controller and set its contents to match those in Listing 15-1.

Listing 15-1. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {
```

```
        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

Create a `Customer` controller and set its contents to match Listing 15-2.

Listing 15-2. The Contents of the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

Create an `Admin` controller and edit its contents to match the code shown in Listing 15-3.

Listing 15-3. The Contents of the AdminController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class AdminController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Admin";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

Creating the View

I specified the `ActionName` view in all of the action methods in these controllers, which allows me to define one view and use it throughout the example application. Create a folder called `Shared` in the `Views` folder and add a new view called `ActionName.cshtml` to it, setting the contents of the view to match Listing 15-4.

Listing 15-4. The Contents of the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
</body>
</html>
```

Setting the Start URL and Testing the Application

As I explained in Part 1 of this book, Visual Studio will try to figure out the URL you want the browser to request based on the file you are editing when you start the debugger. This is a good idea that quickly becomes annoying and is a feature that I always disable. Select `UrlsAndRoutes Properties` from the Visual Studio `Project` menu, switch to the `Web` tab and check the `Specific Page` option in the `Start Action` section. You don't have to provide a value—just checking the option is enough. If you start the example app, you will see the response shown in Figure 15-2.

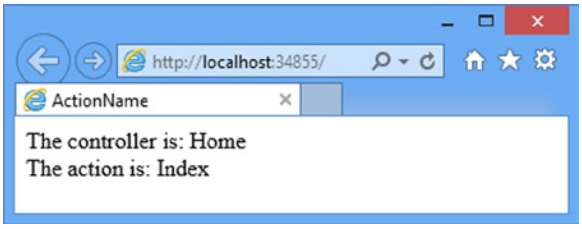

Figure 15-2. *Running the example app*

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 applications. Instead, each route contains a *URL pattern*, which is compared to incoming URLs. If a URL matches the pattern, then it is used by the routing system to process that URL. Let's start with a URL for the example application:

```
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 15-3.

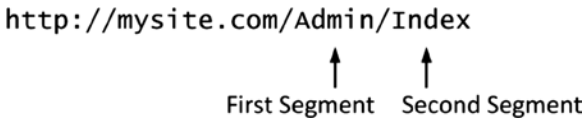

Figure 15-3. *The segments in an example URL*

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 in a way that the routing system can understand. Here is a URL pattern that does this:

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

When processing an incoming 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`, and so the value of the `controller` segment variable will be `Admin` and the value of the `action` segment variable will be `Index`.

I say match to *a* pattern, because 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.

■ **Note** The routing system does not have any special knowledge of controllers and actions. It just extracts values for the segment variables. It is later in the request handling process, when the request reaches the MVC Framework proper, that meaning is assigned to the `controller` and `action` variables. This is why the routing system can be used with Web Forms and the Web API. (I introduce the Web API in Chapter 27 and I describe the ASP.NET request handling process in detail in my *Pro ASP.NET MVC 5 Platform* book.)

By default, a URL 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 illustrated by Table 15-2.

Table 15-2. *Matching URLs*

Request URL	Segment Variables
http://mysite.com/**Admin/Index**	controller = Adminaction = Index
http://mysite.com/**Index/Admin**	controller = Indexaction = Admin
http://mysite.com/**Apples/Oranges**	controller = Applesaction = Oranges
http://mysite.com/**Admin**	No match—too few segments
http://mysite.com/**Admin/Index/Soccer**	No match—too many segments

Table 15-2 highlights two key behaviors of URL patterns:

- URL patterns are *conservative*, and will match only URLs that have the same number of segments as the pattern. You can see this in the fourth and fifth examples in the table.
- URL patterns are *liberal*. If a URL *does* have the correct number of segments, the pattern will extract the value for the 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.

As already mentioned, the routing system does not know anything about an MVC application, and so URL patterns will match even when there is no controller or action that corresponds to the values extracted from a URL. You can see this demonstrated in the second example in Table 15-2. I transposed the `Admin` and `Index` segments in the URL, and so the values extracted from the URL have also been transposed, even though there is no `Index` controller in the example project.

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 `RouteConfig.cs` file, which is in the `App_Start` project folder. You can see the initial content that Visual Studio defines for this file in Listing 15-5.

Listing 15-5. The Default Contents of the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional }
            );
        }
    }
}
```

The static `RegisterRoutes` method that is defined in the `RouteConfig.cs` file is called from the `Global.asax.cs` file, which sets up some of the core MVC features when the application is started. You can see the default contents of the `Global.asax.cs` file in Listing 15-6, and I have highlighted the call to the `RouteConfig.RegisterRoutes` method, which is made from the `Application_Start` method.

Listing 15-6. The Default Contents of the Global.asax.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The `Application_Start` method is called by the underlying ASP.NET platform when the MVC application is first started, which leads to the `RouteConfig.RegisterRoutes` method being called. The parameter to this method is the value of the static `RouteTable.Routes` property, which is an instance of the `RouteCollection` class, which I describe shortly.

■ **Tip** The other call made in the `Application_Start` method sets up a related feature called *areas*, which I describe in the next chapter.

Listing 15-7 shows how to create a route using the example URL pattern from the previous section in the `RegisterRoutes` method of the `RouteConfig.cs` file. (I have removed the other statements in the method so I can focus on the example.)

Listing 15-7. Registering a Route in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            Route myRoute = new Route("{controller}/{action}", new MvcRouteHandler());
            routes.Add("MyRoute", myRoute);
        }
    }
}
```

I created a new Route using a URL pattern as a constructor parameter, which I express as a string. I also pass an instance of MvcRouteHandler to the constructor. Different ASP.NET technologies provide different classes to tailor the routing behavior, and this is the class used for ASP.NET MVC applications. Once I have created the route, I add it to the RouteCollection object using the Add method, passing in the name I want the route to be known by and the route itself.

■ **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 am relaxed about naming, but I explain why this can be a problem in the "Generating a URL from a Specific Route" section in Chapter 16.

A more convenient way of registering routes is to use the MapRoute method defined by the RouteCollection class. Listing 15-8 shows how I can use this method to register a route, which has the same effect as the previous example, but has a cleaner syntax.

Listing 15-8. Registering a Route Using the MapRoute Method in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.MapRoute("MyRoute", "{controller}/{action}");
        }
    }
}
```

This approach is more compact, mainly because I do not need to create an instance of the MvcRouteHandler class (it is done for me, behind the scenes). The MapRoute method is solely for use with MVC applications. ASP.NET Web Forms applications can use the MapPageRoute method, also defined in the RouteCollection class.

Using the Simple Route

You can see the effect of the changes I made to the routing by starting the example application. You will see an error when the browser tries to navigate to the root URL for the application, but if you navigate to a route that matches the {controller}/{action} pattern, you will see a result like the one shown in Figure 15-4, which illustrates the effect of navigating to /Admin/Index.

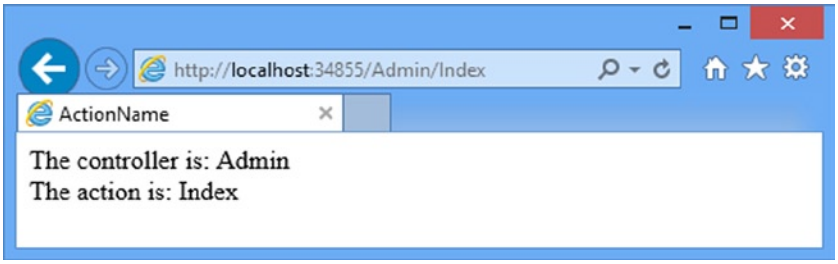

Figure 15-4. *Navigating using a simple route*

My simple route in Listing 15-8 does not tell the MVC Framework how to respond to requests for the root URL and only supports a single, specific, URL pattern. I have temporarily taken a step back from the functionality that Visual Studio adds to the `RouteConfig.cs` file when it creates the project, but I will show you how to build more complex patterns and routes throughout the rest of this chapter.

UNIT TEST: TESTING INCOMING URLS

I recommend that you unit test your routes to make sure they process incoming URLs as expected, even if you choose not to unit test the rest of your application. URL schemas can get pretty complex in large applications, and it is easy to create something that has unexpected results.

In previous chapters, I avoided creating common helper methods to be shared among tests in order to keep each unit test description self-contained. For this chapter, I am taking a different approach. Testing the routing schema for an application is most readily done when you can batch several tests in a single method, and this becomes much easier with some helper methods.

To test routes, I need to mock three classes from the MVC Framework: `HttpRequestBase`, `HttpContextBase`, and `HttpResponseBase`. (This last class is required for testing outgoing URLs, which I cover in the next chapter.) Together, these classes recreate enough of the MVC infrastructure to support the routing system. I added a new Unit Tests file called `RouteTests.cs` to the `UrlsAndRoutes.Tests` unit test project and my first addition is the helper method that creates the mock `HttpContextBase` objects, as follows:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System;
using System.Reflection;
using System.Web;
using System.Web.Routing;

namespace UrlsAndRoutes.Tests {
    [TestClass]
    public class RouteTests {

        private HttpContextBase CreateHttpContext(string targetUrl = null,
                                                  string httpMethod = "GET") {
            // create the mock request
            Mock<HttpRequestBase> mockRequest = new Mock<HttpRequestBase>();
```

```
            mockRequest.Setup(m => m.AppRelativeCurrentExecutionFilePath)
                .Returns(targetUrl);
            mockRequest.Setup(m => m.HttpMethod).Returns(httpMethod);

            // create the mock response
            Mock<HttpResponseBase> mockResponse = new Mock<HttpResponseBase>();
            mockResponse.Setup(m => m.ApplyAppPathModifier(
                It.IsAny<string>())).Returns<string>(s => s);

            // create the mock context, using the request and response
            Mock<HttpContextBase> mockContext = new Mock<HttpContextBase>();
            mockContext.Setup(m => m.Request).Returns(mockRequest.Object);
            mockContext.Setup(m => m.Response).Returns(mockResponse.Object);

            // return the mocked context
            return mockContext.Object;
        }
    }
}
```

The setup here is simpler than it looks. I expose the URL I want to test through the `AppRelativeCurrentExecutionFilePath` property of the `HttpRequestBase` class, and expose the `HttpRequestBase` through the `Request` property of the mock `HttpContextBase` class. My next helper method lets me test a route:

```
...
private void TestRouteMatch(string url, string controller, string action,
    object routeProperties = null, string httpMethod = "GET") {

    // Arrange
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    // Act - process the route
    RouteData result
        = routes.GetRouteData(CreateHttpContext(url, httpMethod));
    // Assert
    Assert.IsNotNull(result);
    Assert.IsTrue(TestIncomingRouteResult(result, controller,
        action, routeProperties));
}
...
```

The parameters of this method let me specify the URL to test, the expected values for the `controller` and `action` segment variables, and an `object` that contains the expected values for any additional variables I have defined. I will show you how to create such variables later in the chapter and in the next chapter. I also defined a parameter for the HTTP method, which I will explain in the "Constraining Routes" section.

The TestRouteMatch method relies on another method, TestIncomingRouteResult, to compare the result obtained from the routing system with the segment variable values I expect. This method uses .NET reflection so that I can use an anonymous type to express any additional segment variables. Do not worry if this method doesn't make sense, as this is just to make testing more convenient; it is not a requirement for understanding MVC. Here is the TestIncomingRouteResult method:

```
...
private bool TestIncomingRouteResult(RouteData routeResult,
    string controller, string action, object propertySet = null) {

    Func<object, object, bool> valCompare = (v1, v2) => {
        return StringComparer.InvariantCultureIgnoreCase
            .Compare(v1, v2) == 0;
    };

    bool result = valCompare(routeResult.Values["controller"], controller)
        && valCompare(routeResult.Values["action"], action);

    if (propertySet != null) {
        PropertyInfo[] propInfo = propertySet.GetType().GetProperties();
        foreach (PropertyInfo pi in propInfo) {
            if (!(routeResult.Values.ContainsKey(pi.Name)
                    && valCompare(routeResult.Values[pi.Name],
                    pi.GetValue(propertySet, null)))) {

                result = false;
                break;
            }
        }
    }
    return result;
}
...
```

I also need a method to check that a URL does not work. As you will see, this can be an important part of defining a URL schema.

```
...
private void TestRouteFail(string url) {
    // Arrange
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    // Act - process the route
    RouteData result = routes.GetRouteData(CreateHttpContext(url));
    // Assert
    Assert.IsTrue(result == null || result.Route == null);
}
...
```

TestRouteMatch and TestRouteFail contain calls to the Assert method, which throws an exception if the assertion fails. Because C# exceptions are propagated up the call stack, I can create simple test methods that test a set of URLs and get the test behavior I require. Here is a test method that tests the route I defined in Listing 15-8:

```
...
[TestMethod]
public void TestIncomingRoutes() {

    // check for the URL that is hoped for
    TestRouteMatch("~/Admin/Index", "Admin", "Index");
    // check that the values are being obtained from the segments
    TestRouteMatch("~/One/Two", "One", "Two");

    // ensure that too many or too few segments fails to match
    TestRouteFail("~/Admin/Index/Segment");
    TestRouteFail("~/Admin");
}
...
```

This test uses the TestRouteMatch method to check the URL I am expecting and also checks a URL in the same format to make sure that the controller and action values are being obtained properly using the URL segments. I use the TestRouteFail method to make sure that the application won't accept URLs that have a different number of segments. When testing, I must prefix the URL with the tilde (~) character, because this is how the ASP.NET Framework presents the URL to the routing system.

Notice that I didn't need to define the routes in the test methods. This is because I am loading them directly using the RegisterRoutes method in the RouteConfig class.

Defining Default Values

The reason that I got an error when I requested the default URL for the application is that it didn't match the route I had defined. The default URL is expressed as ~/ to the routing system and there are no segments in this string that can be matched to the controller and action variables defined by the simple route pattern.

I explained earlier that URL patterns are conservative, in that they will match only URLs with the specified number of segments. I also said that this was the default behavior and 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 to the value. Listing 15-9 provides an example of a route that contains a default value.

Listing 15-9. Providing a Default Value in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
```

```
namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { action = "Index" });
        }
    }
}
```

Default values are supplied as properties in an anonymous type. In Listing 15-9, 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 `Index` as the value for the `action`.

Now that I have provided a default value for the `action` segment, the route will *also* match single-segment URLs as well. When processing a single-segment URL, the routing system will extract the `controller` value from the sole URL segment, and use the default value for the `action` variable. In this way, I can request the URL `http://mydomain.com/Home` and invoke the `Index` action method on the `Home` controller.

I can go further and define 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 15-10 shows how I have mapped the root URL for the application by providing default values for both segments.

Listing 15-10. Providing Action and Controller Default Values in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

By providing default values for both the `controller` and `action` variables, I have created a route that will match URLs that have zero, one, or two segments, as shown in Table 15-3.

Table 15-3. *Matching URLs*

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

The fewer segments I receive in the incoming URL, the more I rely on the default values, up until the point I receive a URL with no segments and only default values are used. You can see the effect of the default values by starting the example app again. This time, 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 the MVC Framework to invoke the `Index` action method on the `Home` controller, as shown in Figure 15-5.

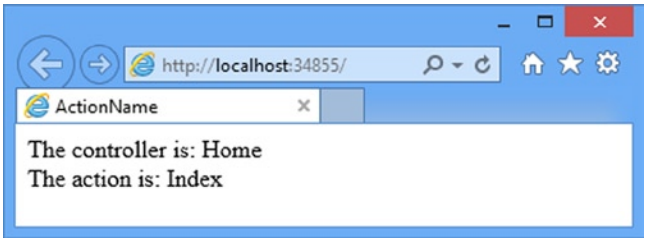

Figure 15-5. *Using default values to broaden the scope of a route*

UNIT TESTING: DEFAULT VALUES

I do not need to take any special actions to use the helper methods to test routes that define default values. Here are the revisions I made to the `TestIncomingRoutes` test method in the `RouteTests.cs` file for the route I defined in Listing 15-10:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteFail("~/Customer/List/All");
}
...
```

The only point of note is that I must specify the default URL as `~/`, as this is how ASP.NET presents the URL to the routing system. If I specify the empty string (`""`) that I used to define the route or `/`, the routing system will throw an exception, and the test will fail.

Using Static URL Segments

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

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

I can do so by using a pattern like the one shown in Listing 15-11.

Listing 15-11. A URL Pattern with Static Segments in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

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

            routes.MapRoute("", "Public/{controller}/{action}",
               new { 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.

I can also create URL patterns that have segments containing both static and variable elements, such as the one shown in Listing 15-12.

Listing 15-12. A URL Pattern with a Mixed Segment in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

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

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

            routes.MapRoute("", "Public/{controller}/{action}",
               new { 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 by Figure 15-6.

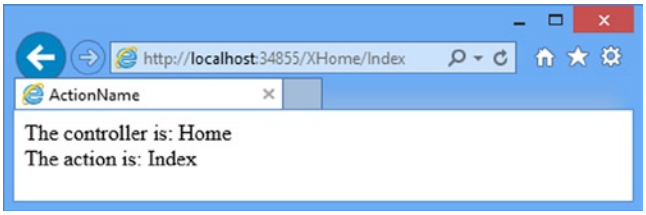

Figure 15-6. *Mixing static and variable elements in a single segment*

ROUTE ORDERING

In Listing 15-12, I defined a new route and placed it before all of the others in the `RegisterRoutes` method. I did this because routes are applied in the order in which they appear in the `RouteCollection` object. The `MapRoute` method adds a route to the end of the collection, 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 route 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. The result of this is that the most specific routes must be defined first. The route I added in Listing 15-12 is more specific than the route that follows. Suppose that I reversed the order of the routes, like this:

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

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

Then the first route, which matches *any* URL with zero, one, or two segments, will 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`, which does not exist, and so will lead to a `404–Not Found` error being sent to the user.

I can combine static URL segments and default values to create an alias for a specific URL. This can be useful if you have published your URL schema publicly and it has formed a contract with your user. If you refactor an application in this situation, 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 I used to have a controller called Shop, which has now been replaced by the Home controller. Listing 15-13 shows how I can create a route to preserve the old URL schema.

Listing 15-13. Mixing Static URL Segments and Default Values in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });

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

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

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

The route I added 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 I have supplied is used. This means 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 the MVC Framework to target the Index action method in the Home controller.

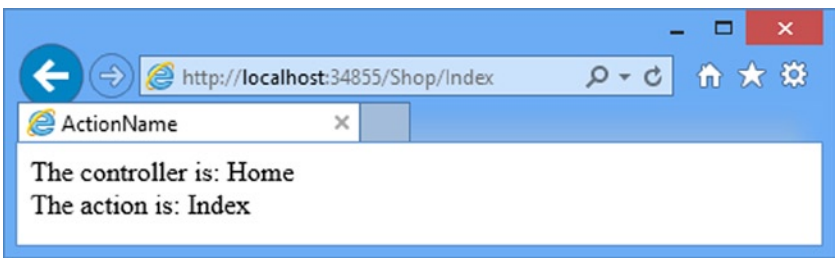

Figure 15-7. *Creating an alias to preserve URL schemas*

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 15-14.

Listing 15-14. Aliasing a Controller and an Action in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

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

            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });

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

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

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

Notice that, once again, I have placed the new route so that it is defined first. This is 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 would get a different result from the one I want. The request would be dealt with using a `404–Not Found` error, rather than being translated in order to preserve a contract with my clients.

UNIT TEST: TESTING STATIC SEGMENTS

Once again, I can use my helper methods to routes whose URL patterns contain static segments. Here is the addition I made to the `TestIncomingRoutes` unit test method to test the route added in Listing 15-14:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
```

```
        TestRouteFail("~/Customer/List/All");
        TestRouteMatch("~/Shop/Index", "Home", "Index");
    }
    ...
```

Defining Custom Segment Variables

The `controller` and `action` segment variables have special meaning to the MVC Framework and, obviously, they correspond to the controller and action method that will be used to service the request. But these are only the built-in segment variables. I can also define my own variables, as shown in Listing 15-15. (I have removed the existing routes from the previous section so I can start over.)

Listing 15-15. Defining Additional Variables in a URL Pattern in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index",
                    id = "DefaultId" });
        }
    }
}
```

The route's 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`, and `area`. The meaning of the first two is obvious, and I will explain *areas* in the next chapter.

I can access any of the segment variables in an action method by using the `RouteData.Values` property. To demonstrate this, I have added an action method to the `Home` controller called `CustomVariable`, as shown in Listing 15-16.

Listing 15-16. Accessing a Custom Segment Variable in an Action Method in the HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {
`
```

```
        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = RouteData.Values["id"];
            return View();
        }
    }
}
```

This method obtains the value of the custom variable in the route URL pattern and passes it to the view using the ViewBag. To create the view for the action method, create the Views/Home folder, right-click on it, select Add ➤ MVC 5 View Page (Razor) from the pop-up menu and set the name to CustomVariable.cshtml. Click the OK button to create the view and edit the contents to match Listing 15-17.

Listing 15-17. The Contents of the CustomVariable.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Custom Variable</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>The custom variable is: @ViewBag.CustomVariable</div>
</body>
</html>
```

To see the effect of the custom segment variable, start the application and navigate to the URL /Home/CustomVariable/Hello. The CustomVariable action method in the Home controller is called, and the value of the custom segment variable is retrieved from the ViewBag and passed to the view. You can see the results in Figure 15-8.

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

I have provided a default value for the `id` segment variable in the route, which means that you will see the results shown in Figure 15-9 if you navigate to `/Home/CustomVariable`.

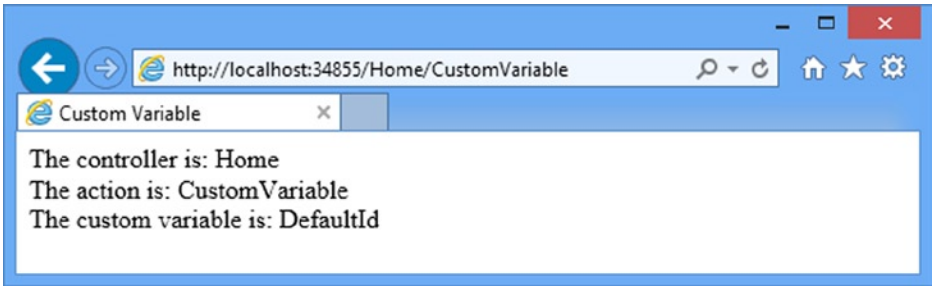

Figure 15-9. *The default value for a custom segment variable*

UNIT TEST: TESTING CUSTOM SEGMENT VARIABLES

I included support for testing custom segment variables in the test helper methods. The `TestRouteMatch` method has an optional parameter that accepts an anonymous type containing the names of the properties I want to test for and the values I expect. Here are the changes I made to the `TestIncomingRoutes` test method to test the route defined in Listing 15-15:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index", new { id = "DefaultId" });
    TestRouteMatch("~/Customer", "Customer", "index", new { id = "DefaultId" });
    TestRouteMatch("~/Customer/List", "Customer", "List",
        new { id = "DefaultId" });
    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteFail("~/Customer/List/All/Delete");
}
...
```

Using Custom Variables as Action Method Parameters

Using the `RouteData.Values` property is only one way to access custom route variables. The other way is much more elegant. If I define parameters to the action method with names that match the URL pattern variables, the MVC Framework will pass the values obtained from the URL as parameters to the action method. For example, the custom variable I defined in the route in Listing 15-15 is called `id`. I can modify the `CustomVariable` action method in the Home controller so that it has a matching parameter, as shown in Listing 15-18.

Listing 15-18. Adding an Action Method Parameter in the HomeController.cs File

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

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable(string id) {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id;
            return View();
        }
    }
}
```

When the routing system matches a URL against the route defined in Listing 15-18, the value of the third segment in the URL is assigned to the custom variable `id`. The MVC Framework 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.

I have defined the `id` parameter as a `string`, but the MVC Framework will try to convert the URL value to whatever parameter type I define. If I declared the `id` parameter as an `int` or a `DateTime`, then I would receive the value from the URL parsed to an instance of that type. This is an elegant and useful feature that removes the need for me to handle the conversion myself.

■ **Note** The MVC Framework uses the *model binding* feature to convert the values contained in the URL to .NET types and can handle much more complex situations than shown in this example. I cover model binding in Chapter 24.

Defining Optional URL Segments

An *optional* URL segment is one that the user does not need to specify, but for which no default value is specified. Listing 15-19 shows an example, and you can see that I specify that a segment variable is optional by setting the default value to `UrlParameter.Optional`.

Listing 15-19. Specifying an Optional URL Segment in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional });
        }
    }
}
```

This route will match URLs whether or not the `id` segment has been supplied. Table 15-4 shows how this works for different URLs.

Table 15-4. *Matching URLs with an Optional Segment Variable*

Segments	Example URL	Maps To
0	mydomain.com	controller = Homeaction = Index
1	mydomain.com/Customer	controller = Customeraction = Index
2	mydomain.com/Customer/List	controller = Customeraction = List
3	mydomain.com/Customer/List/All	controller = Customeraction = Listid = All
4	mydomain.com/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 controller to respond when no value is provided for the `id` segment variable in Listing 15-20.

Listing 15-20. Checking for an Optional Segment Variable in the HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
```

```
        public ActionResult CustomVariable(string id) {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id ?? "<no value>";
            return View();
        }

    }
}
```

You can see the result of starting the application and navigating to the `/Home/CustomVariable` controller URL (which doesn't define a value for the `id` segment variable) in Figure 15-10.

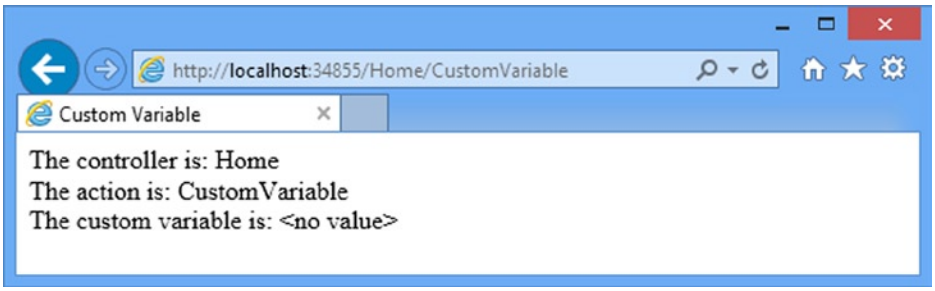

Figure 15-10. *Detecting when a URL doesn't contain a value for an optional segment variable*

Using Optional URL Segments to Enforce Separation of Concerns

Some developers who are focused on the separation of concerns in the MVC pattern do not like putting the default values for segment variables into the routes for an application. If this is an issue, you can use C# optional parameters along with an optional segment variable in the route to define the default values for action method parameters. As an example, Listing 15-21 shows the `CustomVariable` action method to define a default value for the `id` parameter that will be used if the URL doesn't contain a value.

Listing 15-21. Defining a Default Value for an Action Method Parameter in the HomeController.cs File

```
...
public ActionResult CustomVariable(string id = "DefaultId") {
    ViewBag.Controller = "Home";
    ViewBag.Action = "CustomVariable";
    ViewBag.CustomVariable = id;
    return View();
}
...
```

There will always be a value for the `id` parameter (either one from the URL or the default), so I am able to remove the code which deals with the `null` value. This action method combined with the route I defined in Listing 15-21 is the functional equivalent to the route shown in Listing 15-22:

Listing 15-22. An Equivalent Route

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

The difference is that the default value for the `id` segment variable is defined in the controller code and not in the routing definition.

UNIT TESTING: OPTIONAL URL SEGMENTS

The issue to be aware of when testing optional URL segments is that the segment variable will not be added to the `RouteData.Values` collection unless a value was found in the URL. This means that you should not include the variable in the anonymous type unless you are testing a URL that contains the optional segment. Here are the changes to the `TestIncomingRoutes` unit test method for the route defined in Listing 15-22.

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteFail("~/Customer/List/All/Delete");
}
...
```

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 (*), as shown in Listing 15-23.

Listing 15-23. Designating a Catchall Variable in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
```

```
                    id = UrlParameter.Optional });
        }
    }
}
```

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 15-5.

Table 15-5. *Matching URLs with a Catchall Segment Variable*

Segments	Example URL	Maps To
0	/	controller = Homeaction = Index
1	/Customer	controller = Customeraction = Index
2	/Customer/List	controller = Customeraction = List
3	/Customer/List/All	controller = Customeraction = Listid = All
4	/Customer/List/All/Delete	controller = Customeraction = Listid = Allcatchall = Delete
5	/Customer/List/All/Delete/Perm	controller = Customeraction = Listid = Allcatchall = Delete/Perm

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

UNIT TEST: TESTING CATCHALL SEGMENT VARIABLES

I can treat a catchall variable just like a custom variable. The only difference is that I must expect multiple segments to be concatenated in a single value, such as *segment/segment/segment*. Notice that I will not receive the leading or trailing / character. Here are the changes to the `TestIncomingRoutes` method that demonstrate testing for a catchall segment, using the route defined in Listing 15-23 and the URLs shown in Table 15-5:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteMatch("~/Customer/List/All/Delete", "Customer", "List",
        new { id = "All", catchall = "Delete" });
    TestRouteMatch("~/Customer/List/All/Delete/Perm", "Customer", "List",
        new { id = "All", catchall = "Delete/Perm" });
}
...
```

Prioritizing Controllers by Namespaces

When an incoming URL matches a route, the MVC Framework takes the value of the `controller` variable and looks for the appropriate name. For example, when the value of the `controller` variable is `Home`, then the MVC Framework looks for a controller called `HomeController`. This is an *unqualified* class name, which means that the MVC Framework doesn't know what to do if there are two or more classes called `HomeController` in different namespaces.

To demonstrate the problem, create a new folder in the root of the example project called `AdditionalControllers` and add a new `Home` controller, setting the contents to match those in Listing 15-24.

Listing 15-24. The Contents of the AdditionalControllers/HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.AdditionalControllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Additional Controllers - Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

When you start the app, you will see the error shown in Figure 15-11.

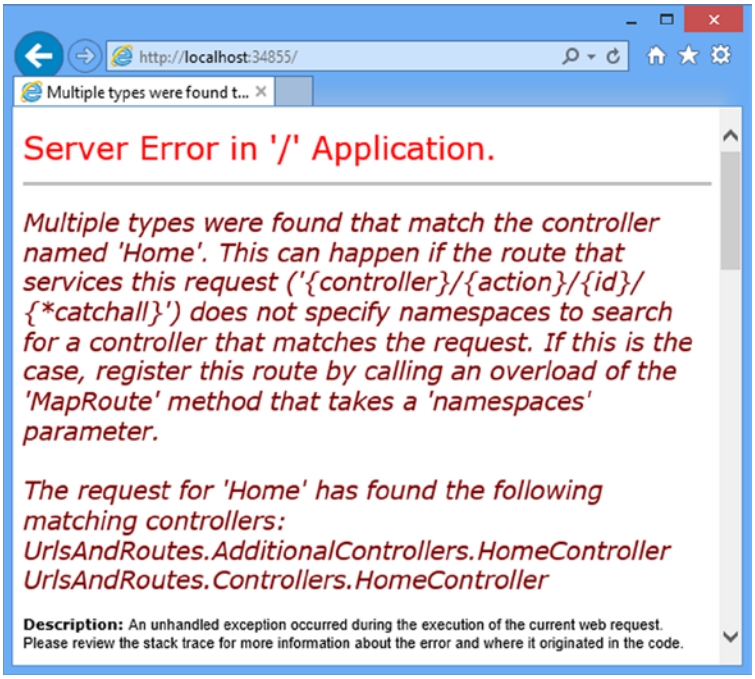

Figure 15-11. *The error displayed when there are two controllers with the same name*

The MVC Framework searched for a class called HomeController and found two: one in the original RoutesAndUrls.Controllers namespace and one in the new RoutesAndUrls.AdditionalControllers namespace. If you read the text of the error shown in Figure 15-11, you can see that the MVC Framework helpfully reports which classes it has found.

This problem arises more often than you might expect, especially if you are working on a large MVC project that uses libraries of controllers from other development teams or third-party suppliers. It is natural to name a controller relating to user accounts AccountController, for example, and it is only a matter of time before you encounter a naming clash.

To address this problem, I can tell the MVC Framework to give preference to certain namespaces when attempting to resolve the name of a controller class, as demonstrated in Listing 15-25.

Listing 15-25. Specifying Namespace Resolution Order in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional
                }, new[] { "URLsAndRoutes.AdditionalControllers" });
        }
    }
}
```

I express the namespaces as an array of strings and in the listing I have told the MVC Framework to look in the URLsAndRoutes.AdditionalControllers namespace before looking anywhere else.

If a suitable controller cannot be found in that namespace, then the MVC Framework will default to its regular behavior and look in all of the available namespaces. If you start the app once you have made this addition to the route, you will see the result shown in Figure 15-12, which shows that the request for the root URL, which is translated in to a request for the Index action method in the Home controller, has been sent to the controller in the AdditionalControllers namespace.

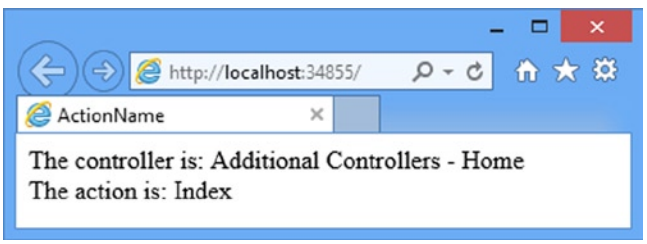

Figure 15-12. *Giving priority to controllers in a specified namespaces*

The namespaces added to a route are given equal priority. The MVC Framework does not check the first namespace before moving on to the second and so forth. For example, suppose that I added both of the project namespaces to the route, like this:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional },
    new[] { "URLsAndRoutes.AdditionalControllers", "UrlsAndRoutes.Controllers" });
...
```

I would see the same error as shown in Figure 15-11, because the MVC Framework is trying to resolve the controller class name in *all* of the namespaces added to the route. If I want to give preference to a single controller in one namespace, but have all other controllers resolved in another namespace, I need to create multiple routes, as shown in Listing 15-26.

Listing 15-26. Using Multiple Routes to Control Namespace Resolution in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("AddContollerRoute", "Home/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "URLsAndRoutes.AdditionalControllers" });

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

The first route applies when the user explicitly requests a URL whose first segment is `Home` and will target the `Home` controller in the `AdditionalControllers` folder. All other requests, including those where no first segment is specified, will be handled by controllers in the `Controllers` folder.

I can tell the MVC Framework to look *only* in the namespaces that I specify. If a matching controller cannot be found, then the framework will not search elsewhere. Listing 15-27 shows how this feature is used.

Listing 15-27. Disabling Fallback Namespaces in the RouteConfig.cs File

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

```
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            Route myRoute = routes.MapRoute("AddContollerRoute",
                    "Home/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "URLsAndRoutes.AdditionalControllers" });

            myRoute.DataTokens["UseNamespaceFallback"] = false;
        }
    }
}
```

The `MapRoute` method returns a `Route` object. I have been ignoring this in previous examples, because I didn't need to make any adjustments to the routes that were created. To disable searching for controllers in other namespaces, I take the `Route` object and set the `UseNamespaceFallback` key in the `DataTokens` collection property to `false`.

This setting will be passed along to the component responsible for finding controllers, which is known as the *controller factory* and which I discuss in detail in Chapter 19. The effect of this addition is that requests that cannot be satisfied by the `Home` controller in the `AdditionalControllers` folder will fail.

Constraining Routes

At the start of the chapter, I described how URL patterns are conservative in how they match segments and liberal in how 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: how to restrict the set of URLs that a route will match against. Once I have control over both of these aspects of the behavior of a route, I can create URL schemas that are expressed with laser-like precision.

Constraining a Route Using a Regular Expression

The first technique is constraining a route using regular expressions. Listing 15-28 contains an example.

Listing 15-28. Using a Regular Expression to Constrain a Route in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
```

```
namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
               new { controller = "Home", action = "Index", id = UrlParameter.Optional },
               new { controller = "^H.*" },
               new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

Constraints are defined by passing them as a parameter to the `MapRoute` method. Like default values, constraints are expressed as an anonymous type, where the properties of the type correspond to the names of the segment variables they constrain. In this example, I have used a constraint with a regular expression that matches URLs only where the value of the `controller` variable begins 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.

Constraining a Route to a Set of Specific Values

Regular expressions can constrain a route so that only specific values for a URL segment will cause a match. I do this using the bar (|) character, as shown in Listing 15-29.

Listing 15-29. Constraining a Route to a Specific Set of Segment Variable Values in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
               new { controller = "Home", action = "Index", id = UrlParameter.Optional },
               new { controller = "^H.*", action = "^Index$|^About$" },
               new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

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 15-29 will match URLs only when the `controller` variable begins with the letter `H` and the `action` variable is `Index` or `About`. So now you can see what I mean about creating precise routes.

Constraining a Route Using HTTP Methods

Routes can be constrained so that they match a URL only when it is requested using a specific HTTP method, as demonstrated in Listing 15-30.

Listing 15-30. Constraining a Route Based on an HTTP Method in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
               new { controller = "Home", action = "Index", id = UrlParameter.Optional },
               new {
                   controller = "^H.*", action = "Index|About",
                   httpMethod = new HttpMethodConstraint("GET")
               },
               new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

The format for specifying an HTTP method constraint is slightly odd. It does not matter what name is given to the property, as long as it is assigned to an instance of the `HttpMethodConstraint` class. In the listing, I called the constraint property `httpMethod` to help distinguish it from the value-based constraints I defined previously.

■ **Note** The ability to constrain routes by HTTP method is unrelated to the ability to restrict action methods using attributes such as `HttpGet` and `HttpPost`. The route constraints are processed much earlier in the request pipeline, and they determine the name of the controller and action required to process a request. The action method attributes are used to determine which specific action method will be used to service a request by the controller. I provide details of how to handle different kinds of HTTP methods (including the more unusual ones such as `PUT` and `DELETE`) in Chapter 16.

I pass the names of the HTTP methods I want to support as string parameters to the constructor of the `HttpMethodConstraint` class. In the listing, I limited the route to `GET` requests, but I could have easily added support for other methods, like this:

```
...
httpMethod = new HttpMethodConstraint("GET", "POST") },
...
```

UNIT TESTING: ROUTE CONSTRAINTS

When testing constrained routes, it is important to test for both the URLs that will match and the URLs you are trying to exclude, which you can do by using the helper methods introduced at the start of the chapter. Here are the changes to the `TestIncomingRoutes` test method that I used to test the route defined in Listing 15-30:

```
...
[TestMethod]
public void TestIncomingRoutes() {

    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Home", "Home", "Index");
    TestRouteMatch("~/Home/Index", "Home", "Index");

    TestRouteMatch("~/Home/About", "Home", "About");
    TestRouteMatch("~/Home/About/MyId", "Home", "About", new { id = "MyId" });
    TestRouteMatch("~/Home/About/MyId/More/Segments", "Home", "About",
        new {
            id = "MyId",
            catchall = "More/Segments"

});

    TestRouteFail("~/Home/OtherAction");
    TestRouteFail("~/Account/Index");
    TestRouteFail("~/Account/About");
}
...
```

Using Type and Value Constraints

The MVC Framework contains a number of built-in constraints that can be used to restrict the URLs that a route matches based on the type and value of segment variables. In Listing 15-31, you can see how I have applied one of these constraints to the routing configuration of the example application.

Listing 15-31. Using a Built-in Type/Value Constraint in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.Mvc.Routing.Constraints;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
               new { controller = "Home", action = "Index", id = UrlParameter.Optional },
               new {
                   controller = "^H.*", action = "Index|About",
                   httpMethod = new HttpMethodConstraint("GET"),
                   id = new RangeRouteConstraint(10, 20)
               },
               new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

In the constraint classes, which are in the `System.Web.Mvc.Routing.Constraints` namespace, check to see if segment variables are values for different C# types and can perform basic checks. In the listing, I have used the `RangeRouteConstraint` class, which checks that the value provided for a segment variable is a valid `int` value that falls between two bounds – , in this case `10` and `20`. Table 15-6 describes the complete set of constraint classes. Not all of them accept arguments and so I have shown the class names as they would be used to configure routes. Ignore the *Attribute Constraint* column for the moment. I'll refer back to it when I introduce the attribute routing feature later in this chapter.

Table 15-6. *The route constraint classes*

Name	Description	Attribute Constraint
`AlphaRouteConstraint()`	Matches alphabet characters, irrespective of case (A–Z, a–z)	`alpha`
`BoolRouteConstraint()`	Matches a value that can be parsed into a `bool`	`bool`
`DateTimeRouteConstraint()`	Matches a value that can be parsed into a `DateTime`	`datetime`
`DecimalRouteConstraint()`	Matches a value that can be parsed into a `decimal`	`decimal`
`DoubleRouteConstraint()`	Matches a value that can be parsed into a `double`	`double`
`FloatRouteConstraint()`	Matches a value that can be parsed into a `float`	`float`
`IntRouteConstraint()`	Matches a value that can be parsed into an `int`	`int`

(*continued*)

Table 15-6. (*continued*)

Name	Description	Attribute Constraint
`LengthRouteConstraint(len)` `LengthRouteConstraint(min, max)`	Matches a value with the specified number of characters or that is between `min` and `max` characters in length.	`length(len)` `length(min, max)`
`LongRouteConstraint()`	Matches a value that can be parsed into a `long`	`long`
`MaxRouteConstraint(val)`	Matches an `int` value if the value is less than `val`	`max(val)`
`MaxLengthRouteConstraint(len)`	Matches a string with no more than `len` characters	`maxlength(len)`
`MinRouteConstraint(val)`	Matches an `int` value if the value is more than `val`	`min(val)`
`MinLengthRouteConstraint(len)`	Matches a string with at least `len` characters	`minlength(len)`
`RangeRouteConstraint(min, max)`	Matches an `int` value if the value is between `min` and `max`	`range(min, max)`

You can combine different constraints for a single segment variable by using the `CompoundRouteConstraint` class, which accepts an array of constraints as its constructor argument. In Listing 15-32, you can see how I have used this feature to apply both the `AlphaRouteConstraint` and the `MinLengthRouteConstraint` to the `id` segment variable to ensure that the route will only match string values that contain solely alphabetic characters and have at least six characters.

Listing 15-32. Combining Route Constraints in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.Mvc.Routing.Constraints;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
               new { controller = "Home", action = "Index", id = UrlParameter.Optional },
               new {
                   controller = "^H.*", action = "Index|About",
                   httpMethod = new HttpMethodConstraint("GET"),
                   id = new CompoundRouteConstraint(new IRouteConstraint[] {
                        new AlphaRouteConstraint(),
                        new MinLengthRouteConstraint(6)
                    })
               },
               new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

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. To demonstrate this feature, I added an `Infrastructure` folder to the example project and created a new class file called `UserAgentConstraint.cs`, the contents of which are shown in Listing 15-33.

Listing 15-33. The Contents of the UserAgentConstraint.cs File

```
using System.Web;
using System.Web.Routing;

namespace UrlsAndRoutes.Infrastructure {

    public class UserAgentConstraint : IRouteConstraint {

        private string requiredUserAgent;

        public UserAgentConstraint(string agentParam) {
            requiredUserAgent = agentParam;
        }

        public bool Match(HttpContextBase httpContext, Route route, string parameterName,
                          RouteValueDictionary values, RouteDirection routeDirection) {

            return httpContext.Request.UserAgent != null &&
                httpContext.Request.UserAgent.Contains(requiredUserAgent);
        }
    }
}
```

The `IRouteConstraint` interface defines the `Match` method, which an implementation can use to indicate to the routing system if its constraint has been satisfied. The parameters for the `Match` method provide access to the request from the client, the route that is being evaluated, the parameter name of the constraint, the segment variables extracted from the URL, and details of whether the request is to check an incoming or outgoing URL. For the example, I check the value of the `UserAgent` property of the client request to see if it contains a value that was passed to the constructor. Listing 15-34 shows the custom constraint used in a route.

Listing 15-34. Applying a Custom Constraint in a Route in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
```

```
using System.Web.Mvc.Routing.Constraints;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("ChromeRoute", "{*catchall}",
                new { controller = "Home", action = "Index" },
                new { customConstraint = new UserAgentConstraint("Chrome") },
                new[] { "UrlsAndRoutes.AdditionalControllers" });

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
               new { controller = "Home", action = "Index", id = UrlParameter.Optional },
               new {
                   controller = "^H.*", action = "Index|About",
                   httpMethod = new HttpMethodConstraint("GET"),
                   id = new CompoundRouteConstraint(new IRouteConstraint[] {
                        new AlphaRouteConstraint(),
                        new MinLengthRouteConstraint(6)
                    })
               },
               new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

In the listing, I have constrained the first route so that it will match only requests made from browsers whose user-agent string contains `Chrome`. If the route matches, then the request will be sent to the `Index` action method in the `Home` controller defined in the `AdditionalControllers` folder, irrespective of the structure and content of the URL that has been requested. The URL pattern consists of just a catchall segment variable, which means that the values for the `controller` and `action` segment variables will always be taken from the defaults and not the URL itself.

The second route will match all other requests and target controllers in the `Controllers` folder, subject to the type and value constraints I applied in the previous section. The effect of these routes is that one kind of browser always ends up at the same place in the application. You can see this in Figure 15-13, which shows the effect of navigating to the app using Google Chrome.

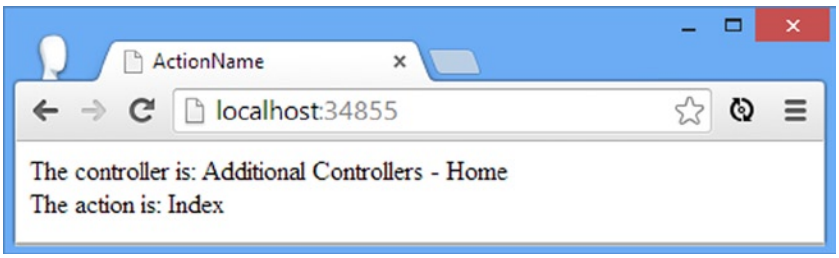

Figure 15-13. *Navigating to the app using the Google Chrome browser*

Figure 15-14 shows the result of navigating to the example application using Internet Explorer. (Notice that I have to add a third segment that contains six or more alpha characters to make the second route match the URL because of the constraints I applied in the last section.)

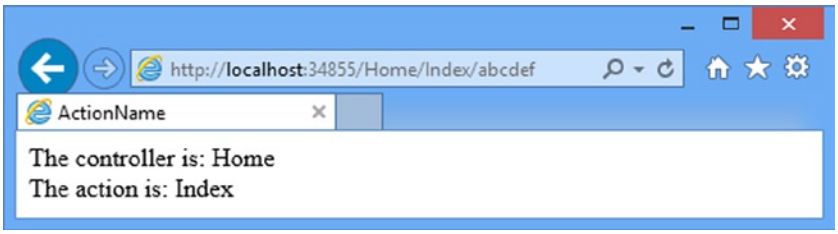

Figure 15-14. *Navigating to the app using Internet Explorer*

■ **Note** To be clear, because this is the kind of thing I get angry letters about, I am not suggesting that you restrict your application so that it supports only one kind of browser. I used user-agent strings solely to demonstrate custom route constraints and believe in equal opportunities for all browsers. I really hate Web sites that try to force their preference for browsers on users.

Using Attribute Routing

All of the examples so far in this chapter have been defined using a technique known as *convention-based routing*. MVC 5 adds support for a new 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'll show you how to create and configure routes using attributes, which can be mixed freely with the standard convention-based routes.

CONVENTION VERSUS ATTRIBUTE ROUTING

Attribute routing is one of the major additions to MVC 5, but I must admit that I am not a fan. One of the main goals of the MVC pattern is, as I described in Chapter 3, to separate out different parts of the application to make them easier to write, test and maintain. I prefer convention-based routing because the controllers have no knowledge or dependency on the routing configuration of the application. By contrast, I find attribute routing muddies the waters and blurs the lines between two important components of an application.

That said, attribute routing is supported as part of the MVC Framework as of MVC 5 and you should read the following sections and make up your own mind. The fact that I don't like a feature shouldn't deter you from giving it due consideration for use in your projects.

The good news is that both approaches to creating routes use the same underlying infrastructure and that means, as you'll see in the sections that follow, that you can use both approaches in a single project with no ill effects.

Enabling and Applying Attribute Routing

Attribute routing is disabled by default and is enabled by the `MapMvcAttributeRoutes` extension method, which is called on the `RouteCollection` object passed as the argument to the static `RegisterRoutes` method. I have added this method call to the `RouteConfig.cs` file in Listing 15-35, as well as simplifying the routes in the application so that I can focus on using attributes.

Listing 15-35. Enabling Attribute Routing in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapMvcAttributeRoutes();

            routes.MapRoute("Default", "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

Calling the `MapMvcAttributeRoutes` method causes the routing system to inspect the controller classes in the application and look for attributes that configure routes. The most important attribute is called `Route` and you can see how I have applied it to the `Customer` controller in Listing 15-36.

Listing 15-36. Applying the Route Attribute in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
```

```
        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

This is the basic use of the `Route` attribute, which is to define a static route for an action method. The `Route` defines two properties, as described in Table 15-7.

Table 15-7. *The Parameters Supported by the Route Attribute*

Name	Description
`Name`	Assigns a name to the route, used for generating outgoing URLs from a specific route
`Template`	Defines the pattern that will be used to match URLs that target the action method

If you supply just one value when applying the `Route` attribute—as I did in the listing—then the value is assumed to be the pattern that will be used to match routes. Patterns for the `Route` attribute follow the same structure as for convention-based routing, although there are some differences when it comes to constraining route matching (which I describe in the *Applying Route Constraints* section, later in the chapter). In this example, I used the `Route` attribute to specify that the `Index` action on the `Customer` controller can be accessed through the URL `/Test`. You can see the result in Figure 15-15. I show you how to use the `Name` property in Chapter 16.

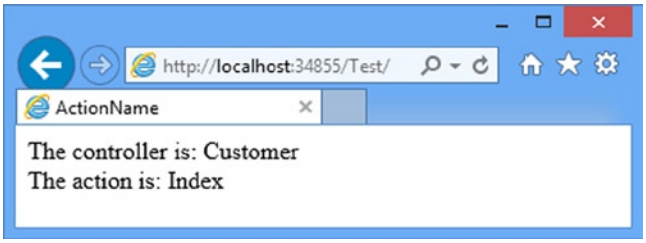

Figure 15-15. *The effect of applying the Route attribute to create a static route*

When an action method is decorated with the `Route` attribute, it can no longer be accessed through the convention-based routes defined in the `RouteConfig.cs` file. For my example, this means that the `Index` action of `Customer` controller can no longer be reached through the `/Customer/Index` URL.

■ **Caution** The `Route` attribute stops convention-based routes from targeting an action method even if attribute routing is disabled. Take care to call the `MapMvcAttributeRoutes` method in the `RouteConfig.cs` file or you will create unreachable action methods.

The `Route` attribute only affects the methods that it is applied to, which means that although the `Index` action method in the `Customer` controller is reachable via the `/Test` URL, the `List` action must still be targeted using the `/Customer/List` URL.

■ **Tip** You can apply the `Route` attribute to the same action method multiple times and each instance will create a new route.

Creating Routes with Segment Variables

The attribute routing feature supports all of the same features as convention-based routing, albeit expressed through attributes. This includes creating routes that contain segment variables and you can see an example of such a route in Listing 15-37.

Listing 15-37. Creating an Attribute Route with a Segment Variable in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        [Route("Users/Add/{user}/{id}")]
        public string Create(string user, int id) {
            return string.Format("User: {0}, ID: {1}", user, id);
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

I have added an action method called `Create` that takes `string` and `int` arguments. For simplicity, I return a `string` result from the method so that I don't have to create a view. The route I defined with the `Route` attribute mixes a static prefix (`Users/Add`) with `user` and `id` segment variables that correspond to the method arguments. The MVC Framework uses the model binding feature, which I describe in Chapter 25, to convert the segment variable values to the correct types in order to invoke the `Create` method. Figure 15-16 shows the effect of navigating to the URL `/Users/Add/Adam/100`.

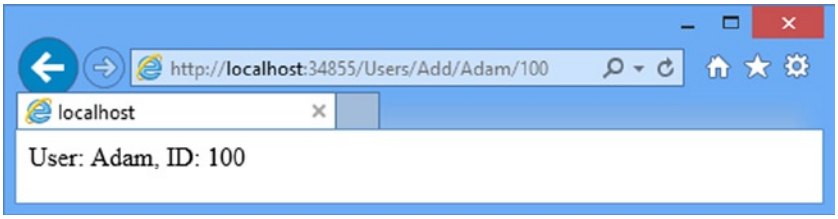

Figure 15-16. *Navigating to a URL with segment variables*

Notice that each instance of the `Route` attribute operates independently, and that means that I am able to create entirely different routes to target each of the action methods in the controller, as described in Table 15-8.

Table 15-8. *The Actions in the Customer Controller and the Routes that Target Them*

Action	URL
`Index`	`/Test`
`Create`	`/Users/Add/Adam/100` (or any values for the last two segments)
`List`	`/Customer/List` (through the route defined in the `RouteConfig.cs` file)

Applying Route Constraints

Routes defined using attributes can be constrained just like those defined in the `RouteConfig.cs` file, although the technique is more direct. To demonstrate how this works, I have added an additional action method to the `Customer` controller, as shown in Listing 15-38.

Listing 15-38. Adding an Action Method and Route to the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        [Route("Users/Add/{user}/{id:int}")]
        public string Create(string user, int id) {
            return string.Format("Create Method - User: {0}, ID: {1}", user, id);
        }
```

```
        [Route("Users/Add/{user}/{password}")]
        public string ChangePass(string user, string password) {
            return string.Format("ChangePass Method - User: {0}, Pass: {1}",
                user, password);
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

The new action method, called `ChangePass`, takes two string arguments. But I have used the `Route` attribute to associate the action with the same URL pattern as for the `Create` action method: a static prefix of `/Users/Add`, followed by two segment variables. To differentiate between the actions, I applied a constraint to the `Route` attribute for the `Create` method, as follows:

```
...
[Route("Users/Add/{user}/{id:int}")]
...
```

I followed the name of the segment variable—`id`—with a colon and then `int`. This tells the routing system that the `Create` action method should only be targeted by requests where the value provided for the `id` segment is a valid `int` value. The `int` constraint corresponds to the `IntRouteConstraint` constraint class and Table 15-6 includes the set of constraint names you can use to access the built-in type and value constraints.

You can see the effect of my constraints by starting the application and requesting the `/Users/Add/Adam/100` and `/Users/Add/Adam/Secret` URLs. The final segment of the first URL is a valid `int` and is directed to the `Create` method. The final segment of the second URL isn't an `int` value and so is directed to the `ChangePass` method, as shown in Figure 15-17.

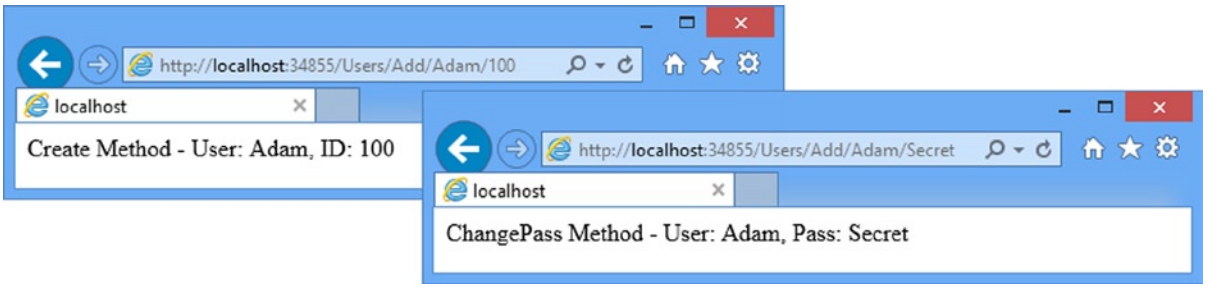

Figure 15-17. *The effect of applying a constraint through the Route attribute*

Combining Constraints

You can apply multiple constraints to a segment variable to further restrict the range of values that the route will match. In Listing 15-39, you can see how I have combined the `alpha` and `length` constraints on the route for the `ChangePass` method.

Listing 15-39. Applying Multiple Constraints to a Route in the CustomerController.cs File

```
...
[Route("Users/Add/{user}/{password:alpha:length(6)}")]
public string ChangePass(string user, string password) {
    return string.Format("ChangePass Method - User: {0}, Pass: {1}",
        user, password);
}
...
```

Multiple constraints are chained together using the same format as for a single constraint: a colon followed by the name of the constraint and, if required, a value in parentheses. The route created by the attribute in this example will only match alphabetic strings that have exactly six characters.

■ **Caution** Be careful when applying constraints. The routes defined by the `Route` attribute work in just the same way as those defined in the `RouteConfig.cs` file and a `404–Not Found` result will be sent to the browser for URLs that can't be matched to an action method. Always define a fallback route that will match irrespective of the values that the URL contains.

Using a Route Prefix

You can use the `RoutePrefix` attribute to define a common prefix that will be applied to all of the routes defined in a controller, which can be useful when you have multiple action methods that should be targeted using the same URL root. You can see how I have applied the `RoutePrefix` attribute to the `CustomerController` in Listing 15-40.

Listing 15-40. Setting a Common Route Prefix in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    [RoutePrefix("Users")]
    public class CustomerController : Controller {

        [Route("~/Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        [Route("Add/{user}/{id:int}")]
        public string Create(string user, int id) {
            return string.Format("Create Method - User: {0}, ID: {1}", user, id);
        }
```

```
        [Route("Add/{user}/{password}")]
        public string ChangePass(string user, string password) {
            return string.Format("ChangePass Method - User: {0}, Pass: {1}",
                user, password);
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

I used the `RoutePrefix` attribute to specify that the routes for the action method should be prefixed with `Users`. With the prefix defined, I am able to update the `Route` attribute for the `Create` and `ChangePass` action methods to remove the prefix. The MVC Framework will combine the prefix with the URL pattern automatically when the routes are created.

Notice that I have also changed the URL pattern for the `Route` attribute applied to the `Index` action method, as follows:

```
...
[Route("~/Test")]
...
```

Prefixing the URL with `~/` tells the MVC Framework that I don't want the `RoutePrefix` attribute applied to the `Index` action method, which means that it will still be accessible through the URL `/Test`.

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, 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, using both 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 MVC Framework *areas* feature, which relies on the routing system and which can be used to manage large and complex MVC Framework applications.