CHAPTER 16

Advanced Routing Features

In the previous chapter, I showed you how to use the routing system to handle incoming URLs, but this is only part of the story. You also need to be able use your URL schema to generate *outgoing URLs* you can embed in your views, so that users can click links and submit forms back to your application in a way that will target the correct controller and action. In this chapter, I will show you different techniques for generating outgoing URLs, show you how to customize the routing system by replacing the standard MVC routing implementation classes and use the MVC Framework *areas* feature, which allows you to break a large and complex MVC application into manageable chucks. I finish this chapter with some best-practice advice about URL schemas in MVC Framework applications. Table 16-1 provides the summary for this chapter.

Table 16-1. *Chapter Summary*

Problem	Solution	Listing
Generate an `a` element with an outgoing URL	Use the `Html.ActionLink` helper method	1–5, 9
Provide values for segment variables	Pass an anonymous object to the `ActionLink` helper whose properties correspond to the segment variable names.	6, 7
Define attributes for the `a` element	Pass an anonymous object to the `ActionLink` helper whose properties correspond to the attribute names	8
Generate an outgoing URL without the `a` element	Use the `Url.Action` helper method	10–13
Generate a URL from a specific route	Specify the route name when calling the helper	14, 15
Create a custom URL matching and generation policy	Derive from the `RouteBase` class	16–21
Create a custom mapping between URLs and action methods	Implement the `IRouteHandler` interface	22, 23
Break an application into sections	Create areas or apply the `RouteArea` attribute	24–27, 30
Resolve controller name ambiguity in areas	Give priority to a controller namespace	28, 29
Prevent IIS and ASP.NET processing requests for static files before they are passed to the routing system	Use the `RouteExistingFiles` property	31–33
Prevent the routing system from processing a request	Use the `IgnoreRoute` method	34

Preparing the Example Project

I am going to continue to use the UrlsAndRoutes project from the previous chapter, but I need to make a couple of changes. First, I deleted the AdditionalControllers folder and HomeController.cs file that it contains. To perform the deletion, right-click on the AdditionalControllers folder and select Delete from the pop-up menu.

Simplifying the Routes

I need to simplify the routes in the application. Edit your App_Start/RouteConfig.cs file so that it matches the content shown in Listing 16-1.

Listing 16-1. Simplifying the Example Routes in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapMvcAttributeRoutes();

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

Adding the Optimization Package

Later in the chapter I describe the *areas* feature, which requires a new package be installed into the project. Enter the following command into the NuGet console:

```
Install-Package Microsoft.AspNet.Web.Optimization -version 1.1.0
    -projectname UrlsAndRoutes
```

This package contains functionality for optimizing the JavaScript and CSS files in the project, which I describe in Chapter 26. I won't be using this feature directly in this chapter, but the areas feature has a dependency on it.

Updating the Unit Test Project

I need to make two changes to the unit test project. The first is to delete the TestIncomingRoutes method, which I won't be using since this chapter is about generating outgoing routes. To avoid failed tests, simply remove the method from the RouteTests.cs file.

The second change is to add a reference to the System.Web.Mvc namespace, which I do by installing the MVC NuGet package into the unit test project. Enter the following command into the NuGet console:

```
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -projectname UrlsAndRoutes.Tests
```

I need to add the MVC 5 package so that I can use some helper methods to generate outgoing URLs. I didn't need to do this in the last chapter because the support for dealing with incoming URLs is in the System.Web and System.Web.Routing namespaces.

Generating Outgoing URLs in Views

In almost every MVC Framework application, you will want to allow the user to navigate from one view to another, which will usually rely on including a link in the first view that targets the action method that generates the second view.

It is tempting to just add a static a element whose href attribute targets the action method, like this:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

With the standard routing configuration, the HTML element creates a link that will target the CustomVariable action method on the Home controller. Manually defined URLs like this one are quick and simple to create. They are also extremely dangerous and you will break all of the URLs you have hard-coded when you change the URL schema for your application. You then must trawl through all of the views in your application and update all of the references to your controllers and action methods, a process that is tedious, error-prone, and difficult to test. A better alternative is to use the routing system to generate outgoing URLs, which ensures that the URLs scheme is used to produce the URLs dynamically and in a way that is guaranteed to reflect the URL schema of the application.

Using the Routing System to Generate an Outgoing URL

The simplest way to generate an outgoing URL in a view is to call the Html.ActionLink helper method within as illustrated by Listing 16-2, which shows an addition I made to the /Views/Shared/ActionName.cshtml view.

Listing 16-2. Using the Html.ActionLink Helper Method in 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>
```

```
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariable")
    </div>
</body>
</html>
```

The parameters to the ActionLink method are the text for the link and the name of the action method that the link should target. You can see the result of this addition by starting the app and allowing the browser to navigate to the root URL, as illustrated by Figure 16-1.

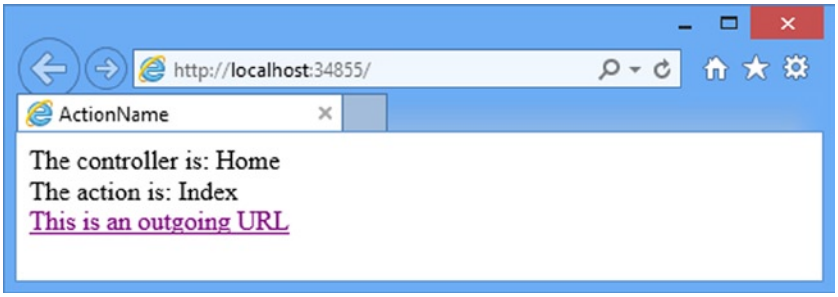

Figure 16-1. *Adding an outgoing URL to a view*

The HTML that the ActionLink method generates is based on the current routing configuration. For example, using the schema defined in Listing 16-1 (and assuming that the view is being rendered by a request to the Home controller) generates this HTML:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

Now, this may seem like a long path to recreate the manually defined URL I showed you earlier, but the benefit of this approach is that it automatically responds to changes in the routing configuration. As a demonstration, I have changed the route defined and added a new route to the RouteConfig.cs file, as shown in Listing 16-3.

Listing 16-3. Adding a Route to the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapMvcAttributeRoutes();

    routes.MapRoute("NewRoute", "App/Do{action}",
        new { controller = "Home" });

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

The new route changes the URL schema for requests that target the `Home` controller. If you start the app, you will see that this change is reflected in the HTML that is generated by the `ActionLink` HTML helper method, as follows:

```
<a href="/App/DoCustomVariable">This is an outgoing URL</a>
```

You can see how generating links in this way addresses the issue of maintenance. I am able to change the routing schema and have the outgoing links in the views reflect the change automatically. And, of course an outgoing URL becomes a regular request when you click on the link, and so the routing system is used again to target the action method correctly, as shown in Figure 16-2.

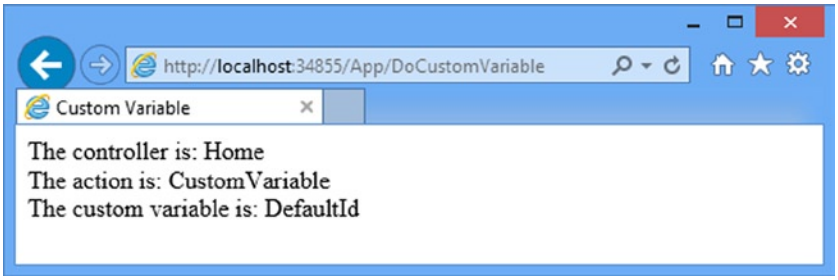

Figure 16-2. *The effect of clicking on a link is to make an outgoing URL into an incoming request*

UNDERSTANDING OUTBOUND URL ROUTE MATCHING

You have seen how changing the routes that define your URL schema changes the way that outgoing URLs are generated. Applications will usually define several routes, and it is important to understand just how routes are selected for URL generation. The routing system processes the routes in the order that they were added to the `RouteCollection` object passed to the `RegisterRoutes` method. Each route is inspected to see if it is a match, which requires three conditions to be met:

- A value must be available for every segment variable defined in the URL pattern. To find values for each segment variable, the routing system looks first at the values you have provided (using the properties of an anonymous type), then the variable values for the current request, and finally at the default values defined in the route. (I return to the second source of these values later in this chapter.)
- None of the values provided for the segment variables may disagree with the default-only variables defined in the route. These are variables for which default values have been provided, but which do not occur in the URL pattern. For example, in this route definition, `myVar` is a default-only variable:

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

For this route to be a match, I must take care to not supply a value for `myVar` or to make sure that the value I do supply matches the default value.

- The values for all of the segment variables must satisfy the route constraints. See the "Constraining Routes" section in the previous chapter for examples of different kinds of constraints.

To be clear: the routing system doesn't try to find the route that provides the *best* matching route. It finds only the *first* match, at which point it uses the route to generate the URL; any subsequent routes are ignored. For this reason, you should define your most specific routes first. It is important to test your outbound URL generation. If you try to generate a URL for which no matching route can be found, you will create a link that contains an empty `href` attribute, like this:

```
<a href="">About this application</a>
```

The link will render in the view properly, but won't function as intended when the user clicks it. If you are generating just the URL (which I show you how to do later in the chapter), then the result will be `null`, which renders as the empty string in views. You can exert some control over route matching by using named routes. See the "Generating a URL from a Specific Route" section later in this chapter for details.

The first `Route` object meeting these criteria will produce a non-`null` URL, and that will terminate the URL-generating process. The chosen parameter values will be substituted for each segment parameter, with any trailing sequence of default values omitted. If you have supplied explicit parameters that do not correspond to segment parameters or default parameters, then the method will append them as a set of query string name/value pairs.

Targeting Other Controllers

The default version of the `ActionLink` method assumes that you want to target an action method in the same controller that has caused the view to be rendered. To create an outgoing URL that targets a *different* controller, you can use an overload that allows you to specify the controller name, as illustrated by Listing 16-4.

Listing 16-4. Targeting a Different Controller in 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>
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariable")
    </div>
    <div>
        @Html.ActionLink("This targets another controller", "Index", "Admin")
    </div>
</body>
</html>
```

■ **Caution** The routing system has no more knowledge of the application when generating outgoing URLs than when processing incoming requests. This means that the values you supply for action methods and controllers are not validated, and you must take care not to specify nonexistent targets.

When you render the view, you will see the following HTML generated:

```
<a href="/Admin">This targets another controller</a>
```

My call for a URL that targets the `Index` action method on the `Admin` controller has been expressed as `/Admin` by the `ActionLink` method. The routing system is pretty clever and it knows that the route defined in the application will use the `Index` action method by default, allowing it to omit unneeded segments.

The routing system includes routes that have been defined using the `Route` attribute when determining how to target a given action method. In Listing 16-5, you can see how I have changed the controller name in the `ActionLink` call so that it targets the Index action in the `Customer` controller.

Listing 16-5. Targeting an Action Decorated with the Route Attribute in 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>
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariable")
    </div>
    <div>
        @Html.ActionLink("This targets another controller", "Index", "Customer")
    </div>
</body>
</html>
```

The link that is generated is as follows:

```
<a href="/Test">This targets another controller</a>
```

This corresponds to the Route attribute I applied to the Index action method in the Customer controller in Chapter 15:

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

Passing Extra Values

You can pass values for segment variables using an anonymous type, with properties representing the segments. Listing 16-6 provides an example, which I added to the ActionName.cshtml view file.

Listing 16-6. Supplying Values for Segment Variables in 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>
    <div>
        @Html.ActionLink("This is an outgoing URL",
            "CustomVariable", new { id = "Hello" })
    </div>
</body>
</html>
```

I have supplied a value for a segment variable called id. If the application uses the route shown in Listing 16-3, then the following HTML will be rendered in the view:

```
<a href="/App/DoCustomVariable?id=Hello">This is an outgoing URL</a>
```

Notice that the value I supplied has been added as part of the query string to fit into the URL pattern described by the route. This is because there is no segment variable that corresponds to id in that route. In Listing 16-7, I have edited the routes in the RouteConfig.cs file to leave only a route that *does* have an id segment.

Listing 16-7. Editing the Routes in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

   routes.MapMvcAttributeRoutes();

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

Start the application again and you will see that the call to the `ActionLink` helper method in the `ActionName.cshtml` view produces the following HTML element:

```
<a href="/Home/CustomVariable/Hello">This is an outgoing URL</a>
```

This time, the value I assigned to the `id` property is included as a URL segment, in keeping with the active route in the application configuration.

UNDERSTANDING SEGMENT VARIABLE REUSE

When I described the way that routes are matched for outbound URLs, I explained that when trying to find values for each of the segment variables in a route's URL pattern, the routing system will look at the values from the current request. This is a behavior that confuses many programmers and can lead to a lengthy debugging session.

Imagine the application has a single route, as follows:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{color}/{page}");
...
```

Now imagine that a user is currently at the URL `/Catalog/List/Purple/123`, and I render a link as follows:

```
...
@Html.ActionLink("Click me", "List", "Catalog", new {page=789}, null)
...
```

You might expect that the routing system would be unable to match the route, because I have not supplied a value for the `color` segment variable, and there is no default value defined. You would, however, be wrong. The routing system *will* match against the route I defined. It will generate the following HTML:

```
<a href="/Catalog/List/Purple/789">Click me</a>
```

The routing system is keen to make a match against a route, to the extent that it will reuse segment variable values from the incoming URL. In this case, I end up with the value `Purple` for the color variable, because of the URL from which my imaginary user started.

This is *not* a behavior of last resort. The routing system will apply this technique as part of its regular assessment of routes, even if there is a subsequent route that would match without requiring values from the current request to be reused. The routing system will reuse values only for segment variables that occur earlier in the URL pattern rather than any parameters that are supplied to the `Html.ActionLink` method. Suppose I tried to create a link like this:

```
...
@Html.ActionLink("Click me", "List", "Catalog", new {color="Aqua"}, null)
...
```

I have supplied a value for `color`, but not for `page`. But `color` appears before `page` in the URL pattern, and so the routing system *won't* reuse the values from the incoming URL, and the route will not match.

The best way to deal with this behavior is to prevent it from happening. I strongly recommend that you do not rely on this behavior, and that you supply values for all of the segment variables in a URL pattern. Relying on this behavior will not only make your code harder to read, but you end up making assumptions about the order in which your users make requests, which is something that will ultimately bite you as your application enters maintenance.

Specifying HTML Attributes

I have focused on the URL that the `ActionLink` helper method generates, but remember that the method generates a complete HTML anchor (`a`) element. I can set attributes for this element by providing an anonymous type whose properties correspond to the attributes I require. Listing 16-8 shows how I modified the `ActionName.cshtml` view to set an `id` attribute and assign a class to the HTML `a` element.

Listing 16-8. Generating an Anchor Element with Attributes in 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>
    <div>
        @Html.ActionLink("This is an outgoing URL",
            "Index", "Home", null, new {
                id = "myAnchorID",
```

```
                @class = "myCSSClass"
            })
    </div>
</body>
</html>
```

I created a new anonymous type that has `id` and `class` properties, and passed it as a parameter to the `ActionLink` method. I passed `null` for the additional segment variable values, indicating that I do not have any values to supply.

■ **Tip** Notice that I prepended the `class` property with a `@` character. This is a C# language feature that lets reserved keywords be used as the names for class members. This is the technique that I used to assign elements to Bootstrap classes in the SportsStore application in Part 1 of this book.

When this call to `ActionLink` is rendered, I get the following HTML:

```
<a class="myCSSClass" href="/" id="myAnchorID">This is an outgoing URL</a>
```

Generating Fully Qualified URLs in Links

All of the links that I have generated so far have contained relative URLs, but I can also use the `ActionLink` helper method to generate fully qualified URLs, as shown in Listing 16-9.

Listing 16-9. Generating a Fully Qualified URL in the ActionNane.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>
    <div>
        @Html.ActionLink("This is an outgoing URL", "Index", "Home",
            "https", "myserver.mydomain.com", " myFragmentName",
            new { id = "MyId" },
            new { id = "myAnchorID", @class = "myCSSClass" })
    </div>
</body>
</html>
```

This is the ActionLink overload with the most parameters, and accepts values for the protocol (https, in the listing), the name of the target server (myserver.mydomain.com), and the URL fragment (myFragmentName), as well as all of the other options you saw previously. When rendered in a view, the ActionLink helper generates the following HTML:

```
<a class="myCSSClass"
     href="https://myserver.mydomain.com/Home/Index/MyId#myFragmentName"
     id="myAnchorID">This is an outgoing URL</a>
```

I recommend using relative URLs wherever possible. Fully qualified URLs create dependencies on the way that your application infrastructure is presented to your users. I have seen large applications that relied on absolute URLs broken by uncoordinated changes to the network infrastructure or domain name policy, which are often outside the control of the programmers.

Generating URLs (and Not Links)

The Html.ActionLink helper method generates complete HTML <a> elements, which is usually what is required when creating views, but there will be times when you want just a URL, without the surrounding HTML. In these circumstances, the Url.Action method can be used to generate just the URL and not the surrounding HTML. Listing 16-10 shows the changes I made to the ActionName.cshtml file to create a URL with the Url.Action helper.

Listing 16-10. Generating a URL Without the Surrounding HTML in 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>
    <div>
        This is a URL:
        @Url.Action("Index", "Home", new { id = "MyId" })
    </div>
</body>
</html>
```

The Url.Action method works in the same way as the Html.ActionLink method, except that it generates only the URL. The overloaded versions of the method and the parameters they accept are the same for both methods, and you can do all of the things with Url.Action that I demonstrated with Html.ActionLink in the previous sections. You can see how the URL in Listing 16-10 is rendered in the view in Figure 16-3.

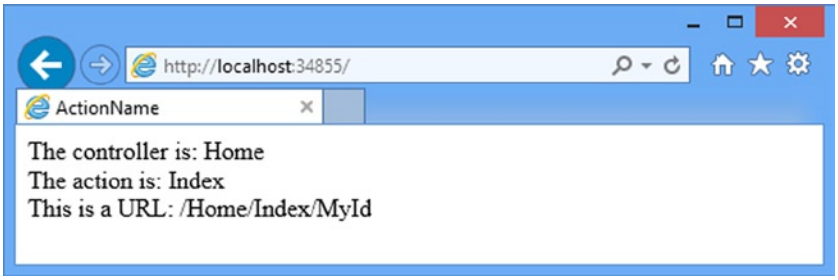

Figure 16-3. *Rendering a URL (as opposed to a link) in a view*

Generating Outgoing URLs in Action Methods

Mostly, you will need to generate outgoing URLs in views, but there are times when you may need to do something similar inside an action method. This can be achieved using the same helper method used in the view, as illustrated by Listing 16-11, which shows a new action method I added to the `Home` controller.

Listing 16-11. Generating an Outgoing URL 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 = "DefaultId") {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id;
            return View();
        }

        public ViewResult MyActionMethod() {
            string myActionUrl = Url.Action("Index", new { id = "MyID" });
            string myRouteUrl = Url.RouteUrl(new { controller = "Home",
                action = "Index" });
            //... do something with URLs...
            return View();
        }
    }
}
```

For the routing in the example app, the `myActionUrl` variable would be set to `/Home/Index/MyID` and the `myRouteUrl` variable would be set to `/`, which is consistent with the results that calling these helpers in a view would produce.

A more common requirement is to redirect the client browser to another URL, which can be achieved by returning the result of calling the RedirectToAction method, as shown in Listing 16-12.

Listing 16-12. Redirecting to Another Action in the HomeController.cs File

```
...
public RedirectToRouteResult MyActionMethod() {
    return RedirectToAction("Index");
}
...
```

The result of the RedirectToAction method is a RedirectToRouteResult, which instructs the MVC Framework to issue a redirect instruction to a URL that will invoke the specified action. There are the usual overloaded versions of the RedirectToAction method that specify the controller and values for the segment variables in the generated URL.

If you want to send a redirect using a URL generated from just object properties, you can use the RedirectToRoute method, as shown in Listing 16-13. This method also returns a RedirectToRouteResult object and has exactly the same effect as calling the RedirectToAction method.

Listing 16-13. Redirecting to a URL in the HomeController.cs File

```
...
public RedirectToRouteResult MyActionMethod() {
    return RedirectToRoute(new { controller = "Home", action = "Index", id = "MyID" });
}
...
```

Generating a URL from a Specific Route

In the previous examples, I left the routing system to select the route which will be used to generate a URL or a link. In this section, I will show you how to control this process and select specific routes. In Listing 16-14, I have changed the routing information in the RouteConfig.cs file to better demonstrate this feature.

Listing 16-14. Changing the Routing Configuration in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapMvcAttributeRoutes();
    routes.MapRoute("MyRoute", "{controller}/{action}");
    routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" });
}
...
```

There are two routes in this configuration and I have specified names for both of these routes: MyRoute and MyOtherRoute. There are two reasons for naming your routes:

- As a reminder of the purpose of the route
- So that you can select a specific route to be used to generate an outgoing URL

I have arranged the routes so that the least specific appears first in the list. This means that if I were to generate a link using the ActionLink method like this:

```
...
@Html.ActionLink("Click me", "Index", "Customer")
...
```

The outgoing link would always be generated using MyRoute, as follows:

```
<a href="/Customer/Index">Click me</a>
```

You can override the default route matching behavior by using the Html.RouteLink method, which lets you specify which route you want to use, as follows:

```
...
@Html.RouteLink("Click me", "MyOtherRoute","Index", "Customer")
...
```

The result is that the link generated by the helper looks like this:

```
<a Length="8" href="/App/Index?Length=5">Click me</a>
```

In this case, the controller I specified, Customer, is overridden by the route and the link targets the Home controller instead.

You can also give names to the routes you define with the Route attribute. In Listing 16-15, you can see how I have given a name to such a route in the Customer controller.

Listing 16-15. Naming a Route in the CustomerController.cs File

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

The addition in this example sets a value for the Name property that I described in Chapter 15. In this example, I assigned the name AddRoute to the route that the attribute creates, which allows me to generate routes by name.

THE CASE AGAINST NAMED ROUTES

The problem with relying on route names to generate outgoing URLs is that doing so breaks through the separation of concerns that is so central to the MVC design pattern. When generating a link or a URL in a view or action method, I want to focus on the action and controller that the user will be directed to, not the format of the URL that will be used. By bringing knowledge of the different routes into the views or controllers, I am creating dependencies that I would prefer to avoid. I tend to avoid naming my routes (by specifying null for the route name parameter) and prefer to use code comments to remind myself of what each route is intended to do.

Customizing the Routing System

You have seen how flexible and configurable the routing system is, but if it does not meet your requirements, you can customize the behavior. In this section, I will show you the two ways to do this.

Creating a Custom RouteBase Implementation

If you do not like the way that standard `Route` objects match URLs, or want to implement something unusual, you can derive an alternative class from `RouteBase`. This gives you control over how URLs are matched, how parameters are extracted, and how outgoing URLs are generated. To derive a class from `RouteBase`, you need to implement two methods:

- `GetRouteData(HttpContextBase httpContext)`: This is the mechanism by which *inbound URL matching* works. The framework calls this method on each `RouteTable.Routes` entry in turn, until one of them returns a non-`null` value.
- `GetVirtualPath(RequestContext requestContext, RouteValueDictionary values)`: This is the mechanism by which *outbound URL generation* works. The framework calls this method on each `RouteTable.Routes` entry in turn, until one of them returns a non-`null` value.

To demonstrate this kind of customization, I am going to create a `RouteBase` class that will handle legacy URL requests. Imagine that I have migrated an existing application to the MVC Framework, but some users have bookmarked the pre-MVC URLs or hard-coded them into scripts. I still want to support those old URLs. I could handle this using the regular routing system, but this problem provides a nice example for this section.

To begin, I need to create a controller that will receive the legacy requests. I have called the controller `LegacyController`, and its contents are shown in Listing 16-16.

Listing 16-16. The Contents of the LegacyController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {

    public class LegacyController : Controller {

        public ActionResult GetLegacyURL(string legacyURL) {
            return View((object)legacyURL);
        }
    }
}
```

In this simple controller, the `GetLegacyURL` action method takes the parameter and passes it as a view model to the view. If I were implementing this controller in a real project, I would use this method to retrieve the files that were requested. But as it is, I am simply going to display the URL in a view.

■ **Tip** Notice that I cast the parameter to the `View` method in Listing 16-16 to `object`. One of the overloaded versions of the `View` method takes a `string` specifying the name of the view to render and, without the cast, this would be the overload that the C# compiler thinks I want. To avoid this, I cast to `object` so that I unambiguously call the overload that passes a view model and uses the default view. I could also have solved this by using the overload that takes both the view name and the view model, but I prefer not to make explicit associations between action methods and views if I can help it.

Create a view called GetLegacyURL.cshtml within the Views/Legacy folder and set the content of the new file to match Listing 16-17.

Listing 16-17. The Contents of the GetLegacyURL.cshtml File

```
@model string

@{
    ViewBag.Title = "GetLegacyURL";
    Layout = null;
}

<h2>GetLegacyURL</h2>

The URL requested was: @Model
```

I want to demonstrate the custom route behavior, so I am not going to spend any time creating complicated actions and views and I just display the model value. I have now reached the point where I can create a custom derivation of the RouteBase class.

Routing Incoming URLs

I added a class file called LegacyRoute.cs to the Infrastructure folder (which is where I like to put support classes that do not really belong anywhere else). The contents of this file are shown in Listing 16-18.

Listing 16-18. The Contents of the LegacyRoute.cs File

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

namespace UrlsAndRoutes.Infrastructure {

    public class LegacyRoute : RouteBase {
        private string[] urls;

        public LegacyRoute(params string[] targetUrls) {
            urls = targetUrls;
        }

        public override RouteData GetRouteData(HttpContextBase httpContext) {
            RouteData result = null;

            string requestedURL =
                httpContext.Request.AppRelativeCurrentExecutionFilePath;
            if (urls.Contains(requestedURL, StringComparer.OrdinalIgnoreCase)) {
                result = new RouteData(this, new MvcRouteHandler());
                result.Values.Add("controller", "Legacy");
```

```
                result.Values.Add("action", "GetLegacyURL");
                result.Values.Add("legacyURL", requestedURL);
            }
            return result;
        }

        public override VirtualPathData GetVirtualPath(RequestContext requestContext,
            RouteValueDictionary values) {

            return null;
        }
    }
}
```

The constructor of this class takes a string array that represents the individual URLs that this routing class will support. I will specify these when I register the route later. Of note in this class is the `GetRouteData` method, which is what the routing system calls to see if the `LegacyRoute` class can match an incoming URL.

If the class cannot match the request, then I just return `null`, and the routing system will move on to the next route in the list and repeat the process. If the class *can* match the request, I return an instance of the `RouteData` class containing the values for the `controller` and `action` variables, and anything else I want to pass along to the action method.

When I create the `RouteData` object, I need to pass in the handler that I want to deal with the values that generated. I use the standard `MvcRouteHandler` class, which is what assigns meaning to the `controller` and `action` values:

```
...
result = new RouteData(this, new MvcRouteHandler());
...
```

For the vast majority of MVC applications, this is the class that you will require, as it connects the routing system to the controller/action model of an MVC application. But you can implement a replacement for `MvcRouteHandler`, as I will demonstrate in the Creating a Custom Route Handler section later in the chapter.

In this custom `RouteBase` implementation, I am willing to match any request for the URLs that were passed to the constructor. When I get such a URL, I add hard-coded values for the controller and action method to the `RouteValues` object. I also pass along the requested URL as the `legacyURL` property. Notice that the name of this property matches the name of the parameter of the action method in the `Legacy` controller, ensuring that the value I generate will be passed to the action method via the parameter.

The last step is to register a new route that uses the custom `RouteBase` class. You can see how to do this in Listing 16-19, which shows the addition to the `RouteConfig.cs` file.

Listing 16-19. Registering the Custom RouteBase Implementation in the RouteConfig.cs File

```
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.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));

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

    }
}
```

I create a new instance of the LegacyRoute class and pass in the URLs I want it to route. I then add the object to the RouteCollection using the Add method. Now when I start the application and request one of the legacy URLs I defined, the request is routed by the LegacyRoute class and directed toward the Legacy controller, as shown in Figure 16-4.

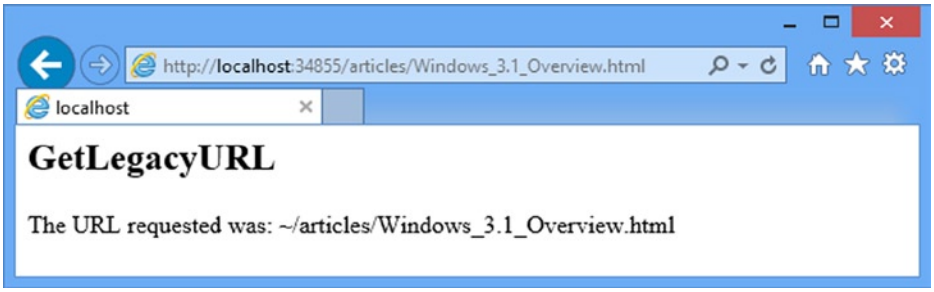

Figure 16-4. *Routing requests using a custom RouteBase implementation*

Generating Outgoing URLs

To support outgoing URL generation, I need to implement the GetVirtualPath method in the LegacyRoute class. If the class is unable to generate a specific URL, I let the routing system know by returning null. Otherwise, I return an instance of the VirtualPathData class. Listing 16-20 shows the implementation of this method.

Listing 16-20. Implementing the GetVirtualPath Method in the LegacyRoute.cs File

```
...
public override VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values) {

    VirtualPathData result = null;

    if (values.ContainsKey("legacyURL") &&
        urls.Contains((string)values["legacyURL"], StringComparer.OrdinalIgnoreCase)) {
        result = new VirtualPathData(this,
            new UrlHelper(requestContext)
                .Content((string)values["legacyURL"]).Substring(1));
    }
    return result;
}
...
```

I have been passing segment variables and other details around in earlier chapters using anonymous types. But behind the scenes, the routing system has been converting these into RouteValueDictionary objects so they can be processed by RouteBase implementations. Listing 16-21 shows an addition to the ActionName.cshtml view file that generates an outgoing URL using the custom routing class.

Listing 16-21. Generating an Outgoing URL via a Custom Route in 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>
    <div>
        This is a URL:
        @Html.ActionLink("Click me", "GetLegacyURL",
            new { legacyURL = "~/articles/Windows_3.1_Overview.html" })
    </div>
</body>
</html>
```

When this view is rendered the ActionLink helper generates the following HTML if you request a URL such as /Home/Index, just as you would expect:

```
<a href="/articles/Windows_3.1_Overview.html">Click me</a>
```

The anonymous type created with the legacyURL property is converted into a RouteValueDictionary class that contains a key of the same name. In this example, I decide I am able to deal with a request for an outbound URL if there is a key named legacyURL and if its value is one of the URLs passed to the constructor. I could be more specific and check for controller and action values, but for a simple example, this is sufficient.

If I get a match, I create a new instance of VirtualPathData, passing in a reference to the current object and the outbound URL. I have used the Content method of the UrlHelper class to convert the application-relative URL to one that can be passed to browsers. The routing system prepends an additional / to the URL, so I must take care to remove the leading character from the generated URL.

Creating a Custom Route Handler

I have relied on the MvcRouteHandler in my routes because it connects the routing system to the MVC Framework, the focus of this book. Even so, the routing system lets me define my own route handler by implementing the IRouteHandler interface. Listing 16-22 shows the content of the CustomRouteHandler.cs class file that I added to the Infrastructure folder in the example project.

Listing 16-22. Implementing the IRouteHandler Interface in the CustomRouteHandler.cs File

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

namespace UrlsAndRoutes.Infrastructure {

    public class CustomRouteHandler : IRouteHandler {
        public IHttpHandler GetHttpHandler(RequestContext requestContext) {
            return new CustomHttpHandler();
        }
    }

    public class CustomHttpHandler : IHttpHandler {
        public bool IsReusable {
            get { return false; }
        }

        public void ProcessRequest(HttpContext context) {
            context.Response.Write("Hello");
        }
    }
}
```

The purpose of the `IRouteHandler` interface is to provide a means to generate implementations of the `IHttpHandler` interface, which is responsible for processing requests. In the MVC implementation of these interfaces, controllers are found, action methods are invoked, views are rendered, and the results are written to the response. My implementation is a little simpler: it just writes the word `Hello` to the client (not an HTML document containing that word, but just the text).

■ **Note** The IHttpHandler interface is defined by the ASP.NET platform and is part of the standard request handling system, which I describe in my *Pro ASP.NET MVC 5 Platform* book, published by Apress in 2014. You don't need to understand the way that ASP.NET handles requests to write an MVC Framework application, but there are facilities to customize and extend the process that can be useful for advanced or complex applications.

I register the custom route handler in the `RouteConfig.cs` file when I define a route, as shown in Listing 16-23.

Listing 16-23. Using a Custom Routing Handler in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapMvcAttributeRoutes();

    routes.Add(new Route("SayHello", new CustomRouteHandler()));

    routes.Add(new LegacyRoute(
            "~/articles/Windows_3.1_Overview.html",
            "~/old/.NET_1.0_Class_Library"));
```

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

When I request the URL /SayHello, the custom route handler is used to process the request. Figure 16-5 shows the result.

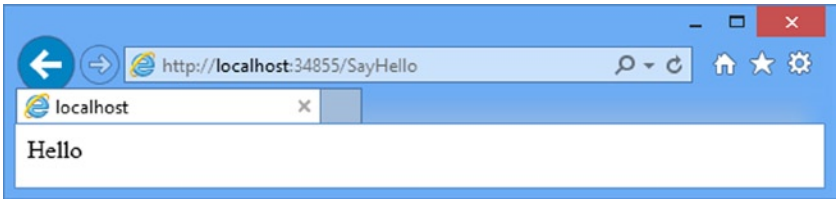

Figure 16-5. *Using a custom request handler*

Implementing custom route handling means taking on responsibility for functions that are usually handled for you, such as controller and action resolution. But it does give you incredible freedom: you can co-opt some parts of the MVC Framework and ignore others, or even implement an entirely new architectural pattern.

Working with Areas

The MVC Framework supports organizing a Web application into *areas*, where each area represents a functional segment of the application, such as administration, billing, customer support, and so on. This is useful in a large project, where having a single set of folders for all of the controllers, views, and models, and it can become difficult to manage.

Each MVC area has its own folder structure, allowing you to keep everything separate. This makes it more obvious which project elements relate to each functional area of the application, helping multiple developers to work on the project without colliding with one another. Areas are supported largely through the routing system, which is why I have chosen to cover this feature alongside URLs and routes. In this section, I will show you how to set up and use areas in your MVC projects.

Creating an Area

To add an area to the example MVC application, right-click the UrlsAndRoutes project item in the Solution Explorer window and select Add ➤ Area from the pop-up menu. Visual Studio will prompt you for the name of the area, as shown in Figure 16-6. In this case, I have specified an area called Admin. This is a pretty common area to create, because many Web applications need to separate the customer-facing and administration functions. Click the Add button to create the area.

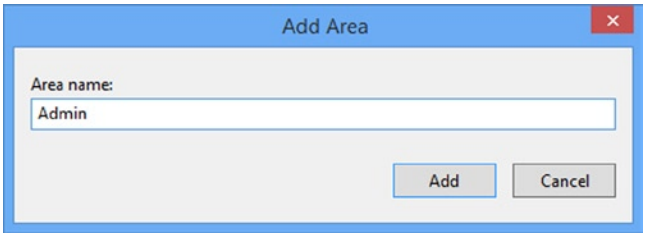

Figure 16-6. *Adding an area to an MVC application*

After you click Add, Visual Studio will add an `Areas` folder to the project. This contains a folder called `Admin`, which represents the area that I just created. If I were to create additional areas, other folders would be created here.

Inside the `Areas/Admin` folder, you will see a mini-MVC project. There are folders called `Controllers`, `Models`, and `Views`. The first two are empty, but the `Views` folder contains a `Shared` folder (and a `web.config` file that configures the view engine, but I am not interested in the view engine until Chapter 20).

The `Areas` folder also contains a file called `AdminAreaRegistration.cs`, which defines the `AdminAreaRegistration` class, as shown in Listing 16-24.

Listing 16-24. The Contents of the AdminAreaRegistration.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Areas.Admin {
    public class AdminAreaRegistration : AreaRegistration {

        public override string AreaName {
            get {
                return "Admin";
            }
        }

        public override void RegisterArea(AreaRegistrationContext context) {
            context.MapRoute(
                "Admin_default",
                "Admin/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

The interesting part of this class is the `RegisterArea` method. As you can see from the listing, this method registers a route with the URL pattern `Admin/{controller}/{action}/{id}`. I can define additional routes in this method, which will be unique to this area.

■ **Caution** If you assign names to your routes, you must ensure that they are unique across the entire application and not just the area for which they are intended.

I do not need to take any action to make sure that this registration method is called. Visual Studio added a statement to the Global.asax file that takes care of setting up areas when it created the project, which you can see in Listing 16-25.

Listing 16-25. Area Registration Setup in the Global.asax 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 call to the static AreaRegistration.RegisterAllAreas method causes the MVC Framework to go through all of the classes in the application, find those that are derived from the AreaRegistration class and call the RegisterArea method on each of them.

■ **Caution** Do not change the order of the statements related to routing in the Application_Start method. If you call RegisterRoutes before AreaRegistration.RegisterAllAreas is called, then your routes will be defined before the area routes. Given that routes are evaluated in order, this will mean that requests for area controllers are likely to be matched against the wrong routes.

The AreaRegistrationContext class that is passed to each area's RegisterArea method exposes a set of MapRoute methods that the area can use to register routes in the same way as your main application does in the RegisterRoutes method of Global.asax.

■ **Note** The MapRoute methods in the AreaRegistrationContext class automatically limit the routes you register to the namespace that contains the controllers for the area. This means that when you create a controller in an area, you must leave it in its default namespace; otherwise, the routing system will not be able to find it.

Populating an Area

You can create controllers, views, and models in an area just as you have seen in previous examples. To create a controller, right-click the Controllers folder within the Admin area and select Add ➤ Controller from the pop-up menu. Select MVC 5 Controller – Empty from the list of options, click the Add button, set the controller name and click the Add button to create the new controller class.

To demonstrate the way that areas isolate sections of the application, I added a Home controller to the Admin area. You can see the contents of the Areas/Admin/Controllers/HomeController.cs file in Listing 16-26.

Listing 16-26. The Contents of the Areas/Admin/Controllers/HomeController.cs File

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

namespace UrlsAndRoutes.Areas.Admin.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            return View();
        }
    }
}
```

I am not going to change the controller code. Just rendering the default view associated with the Index action method will be enough for this sample. Create the Areas/Admin/Views/Home folder, right-click it in the Solution Explorer and select Add ➤ MVC 5 View Page (Razor) from the menu. Set the name to Index.cshtml, click OK to create the file and edit the content to match those shown in Listing 16-27.

Listing 16-27. The Contents of the Areas/Admin/Views/Home/Index.cshtml File

```
@{
    ViewBag.Title = "Index";
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        <h2>Admin Area Index</h2>
    </div>
</body>
</html>
```

The point of all of this is to show that working inside an area is just the same as working in the main part of an MVC project. If you start the application and navigate to /Admin/Home/Index, you will see the view that was added to the Admin area, as shown in Figure 16-7.

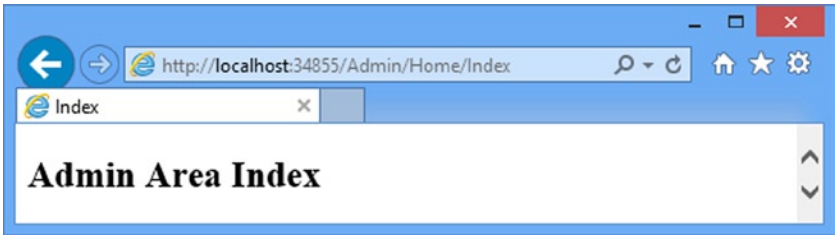

Figure 16-7. *Rendering an area view*

Resolving the Ambiguous Controller Issue

Okay, so I lied slightly: areas are not quite as self-contained as they might be. If you navigate to the `/Home/Index` URL, you will see the error shown in Figure 16-8.

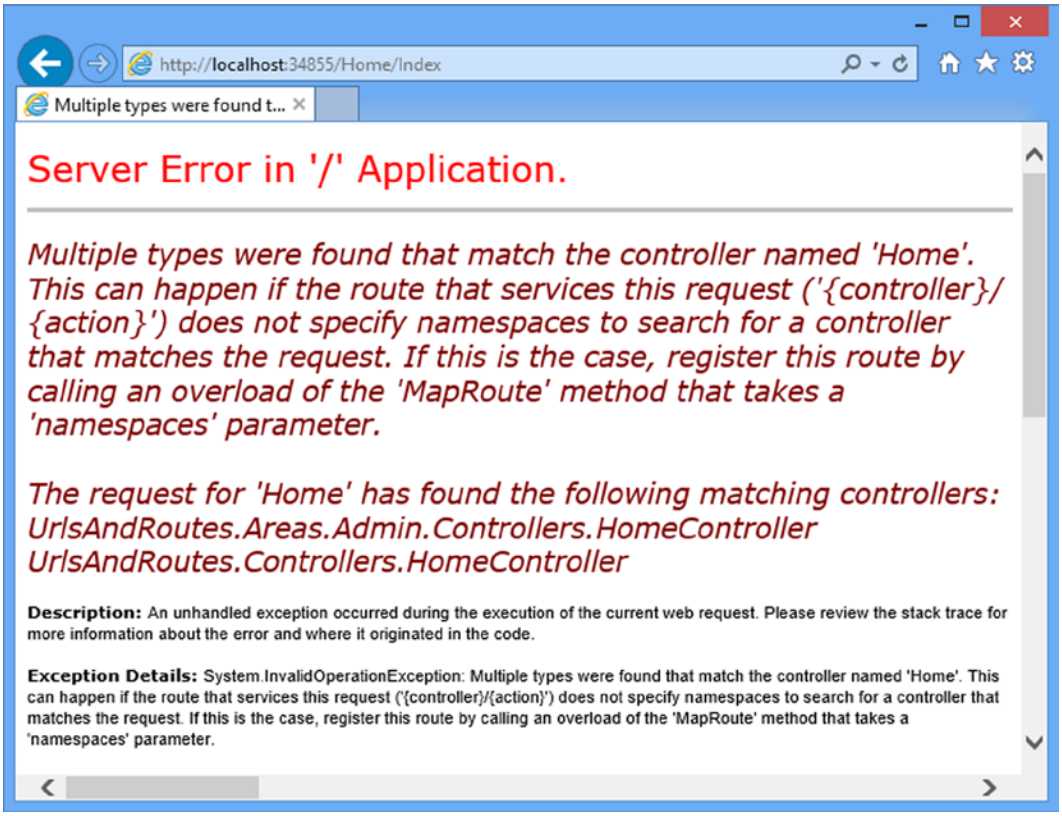

Figure 16-8. *The ambiguous controller error*

When an area is registered, any routes that it defines are limited to the namespace associated with the area. This is how I was able to request `/Admin/Home/Index` and get the `HomeController` class in the `UrlsAndRuutes.Areas.Admin.Controllers` namespace.

However, routes defined in the `RegisterRoutes` method of `RouteConfig.cs` are not similarly restricted. In Listing 16-28, as a reminder, I have listed the current routing configuration of the example app.

Listing 16-28. The Routing Configuration for the Example MVC App in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapMvcAttributeRoutes();

    routes.Add(new Route("SayHello", new CustomRouteHandler()));

    routes.Add(new LegacyRoute(
        "~/articles/Windows_3.1_Overview.html",
        "~/old/.NET_1.0_Class_Library"));

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

The route named `MyRoute` translates the incoming URL from the browser to the `Index` action on the `Home` controller. At that point, I get an error, because there are no namespace restrictions in place for this route and the MVC Framework can see two `HomeController` classes. To resolve this, I need to prioritize the main controller namespace in all of the routes which might lead to a conflict, as shown in Listing 16-29.

Listing 16-29. Resolving the Area Namespace Conflict in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapMvcAttributeRoutes();

    routes.Add(new Route("SayHello", new CustomRouteHandler()));

    routes.Add(new LegacyRoute(
            "~/articles/Windows_3.1_Overview.html",
            "~/old/.NET_1.0_Class_Library"));

    routes.MapRoute("MyRoute", "{controller}/{action}", null,
        new[] {"UrlsAndRoutes.Controllers"});
    routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" },
        new[] { "UrlsAndRoutes.Controllers" });
}
...
```

This change ensures that the controllers in the main project are given priority in resolving requests. Of course, if you want to give preference to the controllers in an area, you can do that instead.

Creating Areas with Attributes

You can also create areas by applying the RouteArea attribute to controller classes. You can see how I have assigned the action methods in the Customer controller to a new area called Services in Listing 16-30.

Listing 16-30. Creating an Area Using an Attribute in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    [RouteArea("Services")]
    [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}", Name="AddRoute")]
        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");
        }
    }
}
```

The RouteArea attribute moves all of the routes defined by the Route attribute into the specified area. The effect of this attribute combined with the RoutePrefix attribute means that to reach the Create action method, for example, I have to create a URL like this:

```
http://localhost:34855/Services/Users/Add/Adam/100
```

The RouteArea attribute doesn't affect routes that are defined by the Route attribute but that start with ~/. This means, for example, that I continue to reach the Index action method with this URL:

```
http://localhost:34855/Test/
```

The RouteArea has no effect on action methods to which the Route attribute has not been defined, which means that the routing for the List action method is determined by the RouteConfig.cs file and not attribute-based routing.

Generating Links to Actions in Areas

You do not need to take any special steps to create links that refer to actions in the same MVC area that the current request relates to. The MVC Framework detects that a request relates to a particular area and ensures that outbound URL generation will find a match only among routes defined for that area. For example, adding a call to the Html.ActionLink helper from a view in the Admin area like this:

```
...
@Html.ActionLink("Click me", "About")
...
```

will generate the following HTML:

```
<a href="/Admin/Home/About">Click me</a>
```

To create a link to an action in a *different* area, or no area at all, you must create a variable called area and use it to specify the name of the area you want, like this:

```
...
@Html.ActionLink("Click me to go to another area", "Index", new { area = "Support" })
...
```

It is for this reason that area is reserved from use as a segment variable name. The HTML generated by this call is as follows (assuming that you created an area called Support that has a suitable route defined):

```
<a href="/Support/Home">Click me to go to another area</a>
```

If you want to link to an action on one of the top-level controllers (a controller in the /Controllers folder), then you should specify the area as an empty string, like this:

```
...
@Html.ActionLink("Click me to go to another area", "Index", new { area = "" })
...
```

Routing Requests for Disk Files

Not all of the requests for an MVC application are for controllers and actions. Most applications need a way to serve content such as images, static HTML files, JavaScript libraries, and so on. As a demonstration, I created a Content folder and added a file called StaticContent.html to it using the HTML Page item template. Listing 16-31 shows the contents of the HTML file.

Listing 16-31. The Contents of the StaticContent.html File

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>Static HTML Content</title></head>
    <body>
        This is the static html file (~/Content/StaticContent.html)
    </body>
</html>
```

The routing system provides integrated support for serving such content. If you start the application and request the URL `/Content/StaticContent.html`, you will see the contents of this simple HTML file displayed in the browser, as shown in Figure 16-9.

Figure 16-9. *Requesting the static content file*

By default, the routing system checks to see if a URL matches a disk file *before* evaluating the application's routes, which is why I didn't have to add a route to get the result shown in Figure 16-9.

If there is a match between the requested URL and a disk on the file, then the disk file is served and the routes defined by the application are never used. This behavior can be reversed so that the routes are evaluated before disk files are checked by setting the `RouteExistingFiles` property of the `RouteCollection` to `true`, as shown in Listing 16-32.

Listing 16-32. Enabling Route Evaluation Before File-Checking in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.RouteExistingFiles = true;

            routes.MapMvcAttributeRoutes();

            routes.Add(new Route("SayHello", new CustomRouteHandler()));

            routes.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));
```

```
            routes.MapRoute("MyRoute", "{controller}/{action}", null,
                new[] { "UrlsAndRoutes.Controllers" });
            routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

The convention is to place this statement close to the top of the `RegisterRoutes` method, although it will take effect even if you set it after you have defined your routes.

Configuring the Application Server

Visual Studio uses IIS Express as the application server for MVC application projects. Not only do I have to set the `RouteExistingFiles` property to `true` in the `RegisterRoutes` method, I also have to tell IIS Express not to intercept requests for disk files before they are passed to the MVC routing system.

First of all, start IIS Express. The easiest way to do this is to start the MVC application from Visual Studio, which will cause the IIS Express icon to appear on the task bar. Right-click on the icon and select `Show All Applications` from the pop-up menu. Click on `UrlsAndRoutes` in the `Site Name` column to display the IIS configuration information, as shown in Figure 16-10.

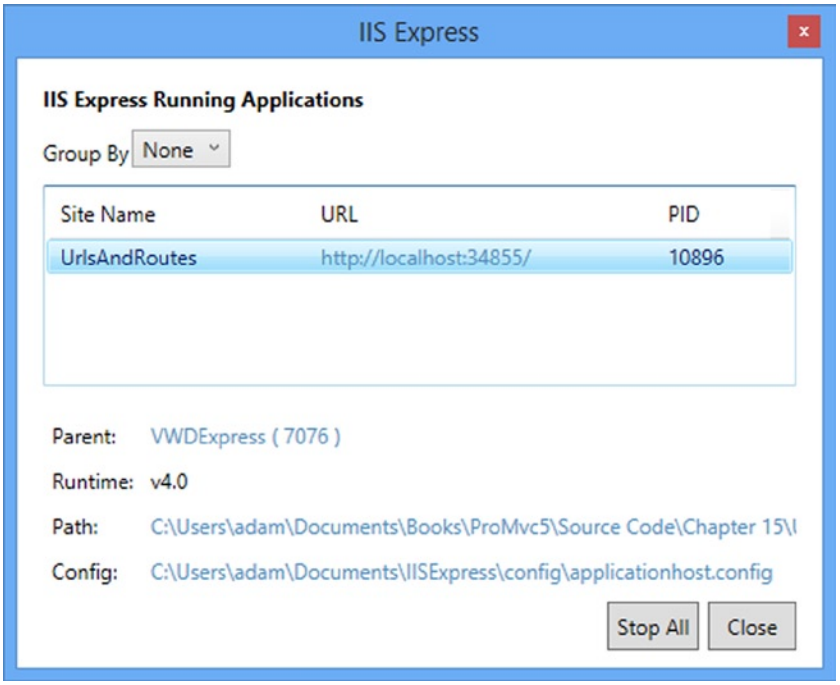

Figure 16-10. *The IIS Express configuration information*

Click on the Config link at the bottom of the window to open the IIS Express configuration file in Visual Studio. Now type Control+F and search for UrlRoutingModule-4.0. There will be an entry found in the modules section of the configuration file and you will need to set the preCondition attribute to the empty string, like this:

```
...
<add name="UrlRoutingModule-4.0" type="System.Web.Routing.UrlRoutingModule"
    preCondition="" />
...
```

Now restart the application in Visual Studio to let the modified settings take effect and navigate to the /Content/StaticContent.html URL. Rather than see the contents of the file, you will see the error message shown in Figure 16-11. This error occurs because the request for the HTML file has been passed to the MVC routing system but the route that matches the URL directs the request to the Content controller, which does not exist.

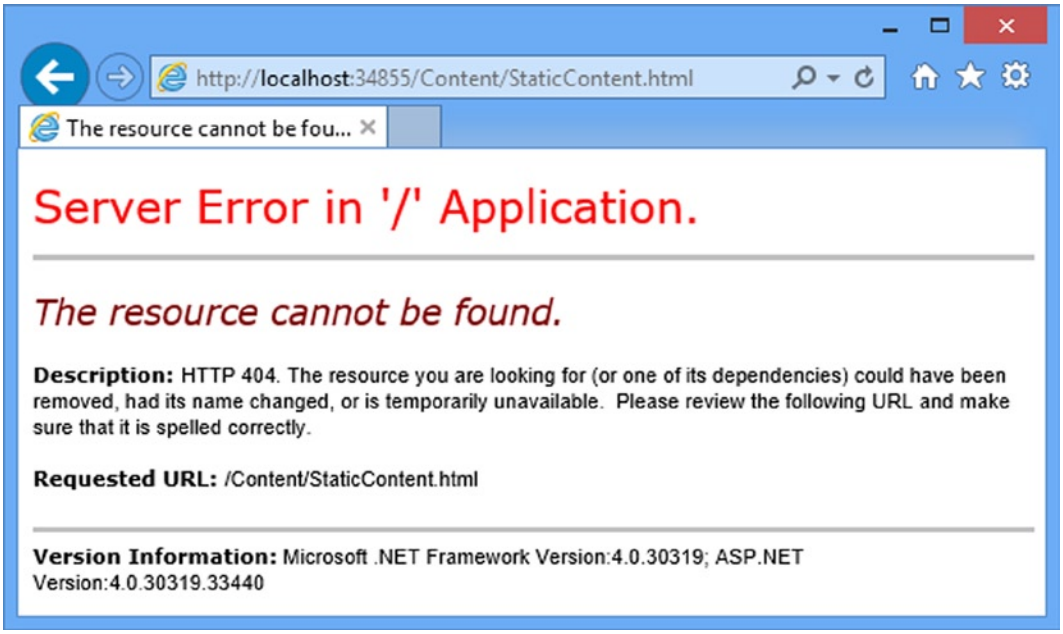

Figure 16-11. *Requesting a static content URL which is handled by the routing system*

Defining Routes for Disk Files

Once the property has been set to true, I can define routes that match URLs that correspond to disk files, such as the one shown in Listing 16-33.

Listing 16-33. A Route Whose URL Pattern Corresponds to a Disk File in the RouteConfig.cs File

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

```
namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.RouteExistingFiles = true;

            routes.MapMvcAttributeRoutes();

            routes.MapRoute("DiskFile", "Content/StaticContent.html",
                new {
                    controller = "Customer",
                    action = "List",
                });

            routes.Add(new Route("SayHello", new CustomRouteHandler()));

            routes.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));

            routes.MapRoute("MyRoute", "{controller}/{action}", null,
                new[] { "UrlsAndRoutes.Controllers" });
            routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

This route maps requests for the URL `Content/StaticContent.html` to the `List` action of the `Customer` controller. You can see the URL mapping at work in Figure 16-12, which I created by starting the app and navigating to the `/Content/StaticContent.html` URL again.

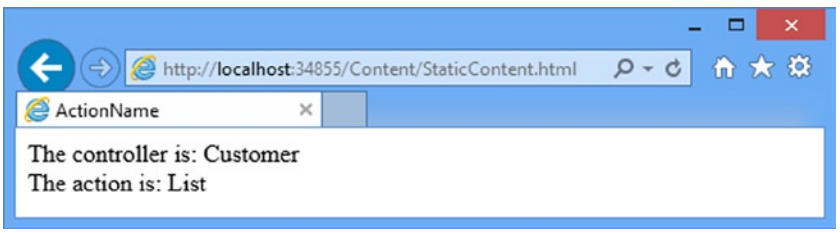

Figure 16-12. *Intercepting a request for a disk file using a route*

■ **Tip** Your browser may cache the old response, especially if you are using the browser link feature I described in Chapter 14. If this happens, reload the web page and you should see the response shown in the figure.

Routing requests intended for disk files requires careful thought, not least because URL patterns will match these kinds of URL as eagerly as any other. For example, as you saw in the previous section, a request for `/Content/StaticContent.html` will be matched by a URL pattern such as `{controller}/{action}`. Unless you are careful, you can end up with some exceptionally strange results and reduced performance. So, enabling this option is a last resort.

Bypassing the Routing System

Setting the RouteExistingFiles property, which I demonstrated in the previous section, makes the routing system more inclusive. Requests that would normally bypass the routing system are now evaluated against the routes the application defines.

The counterpart to this feature is the ability to make the routing system *less* inclusive and prevent URLs from being evaluated against routes. This is done using the IgnoreRoute method of the RouteCollection class, as shown in Listing 16-34.

Listing 16-34. Using the IgnoreRoute Method in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes) {

            routes.RouteExistingFiles = true;

            routes.MapMvcAttributeRoutes();

            routes.IgnoreRoute("Content/{filename}.html");

            routes.Add(new Route("SayHello", new CustomRouteHandler()));

            routes.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));

            routes.MapRoute("MyRoute", "{controller}/{action}", null,
                new[] { "UrlsAndRoutes.Controllers" });
            routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

You can use segment variables like {filename} to match a range of URLs. In this case, the URL pattern will match any two-segment URL where the first segment is Content and the second content has the .html extension.

The IgnoreRoute method creates an entry in the RouteCollection where the route handler is an instance of the StopRoutingHandler class, rather than MvcRouteHandler. The routing system is hard-coded to recognize this handler. If the URL pattern passed to the IgnoreRoute method matches, then no subsequent routes will be evaluated, just as when a regular route is matched. It follows, therefore, that the location of the call to the IgnoreRoute method is significant. If you start the app and navigate to the /Content/StaticContent.html URL again, you will see the contents of the HTML file because the StopRoutingHandler object is processed before any other route which might have matched the URL.

URL Schema Best Practices

After all of this, you may be left wondering where to start in designing your own URL schema. You could just accept the default schema that Visual Studio generates for you, but there are benefits in giving your schema some thought. In recent years, the design of an application's URLs have been taken increasingly seriously and a few important design principles have emerged. If you follow these design patterns, you will improve the usability, compatibility, and search-engine rankings of your applications.

Make Your URLs Clean and Human-Friendly

Users notice the URLs in your applications. If you do not agree, then just think back to the last time you tried to send someone an Amazon URL. Here is the URL for an earlier edition of this book:

```
http://www.amazon.com/Pro-ASP-NET-MVC-Professional-
Apress/dp/1430242361/ref=la_B001IUOSNK_1_5?ie=UTF8&qid=1349978167&sr=1-5
```

It is bad enough sending someone such a URL by e-mail, but try reading this over the phone. When I needed to do this recently, I ended up quoting the ISBN number and asking the caller to look it up for himself. It would be nice if I could access the book with a URL like this:

```
http://www.amazon.com/books/pro-aspnet-mvc5-framework
```

That is the kind of URL that I *could* read out over the phone and that doesn't look like I dropped something on the keyboard while composing an e-mail message.

■ **Note** To be clear, I have only the highest respect for Amazon, who sells more of my books than everyone else combined. I know for a fact that each and every member of the Amazon team is a strikingly intelligent and beautiful person. Not one of them would be so petty as to stop selling my books over something so minor as criticism of their URL format. I love Amazon. I adore Amazon. I just wish they would fix their URLs.

Here are some simple guidelines to make friendly URLs:

- Design URLs to describe their content, not the implementation details of your application. Use `/Articles/AnnualReport` rather than `/Website_v2/CachedContentServer/FromCache/AnnualReport`.
- Prefer content titles over ID numbers. Use `/Articles/AnnualReport` rather than `/Articles/2392`. If you must use an ID number (to distinguish items with identical titles, or to avoid the extra database query needed to find an item by its title), then use both (`/Articles/2392/AnnualReport`). It takes longer to type, but it makes more sense to a human and improves search-engine rankings. Your application can just ignore the title and display the item matching that ID.
- Do *not* use file name extensions for HTML pages (for example, `.aspx` or `.mvc`), but *do* use them for specialized file types (such as `.jpg`, `.pdf`, and `.zip`). Web browsers do not care about file name extensions if you set the MIME type appropriately, but humans still expect PDF files to end with `.pdf`.

- Create a sense of hierarchy (for example, `/Products/Menswear/Shirts/Red`), so your visitor can guess the parent category's URL.
- Be case-insensitive. (Someone might want to type in the URL from a printed page.) The ASP.NET routing system is case-insensitive by default.
- Avoid symbols, codes, and character sequences. If you want a word separator, use a dash (as in `/my-great-article`). Underscores are unfriendly, and URL-encoded spaces are bizarre (`/my+great+article`) or disgusting (`/my%20great%20article`).
- Do not change URLs. Broken links equal lost business. When you do change URLs, continue to support the old URL schema for as long as possible via permanent (301) redirections.
- Be consistent. Adopt one URL format across your entire application.

URLs should be short, easy to type, hackable (human-editable), and persistent, and they should visualize site structure. Jakob Nielsen, usability guru, expands on this topic at http://www.useit.com/alertbox/990321.html. Tim Berners-Lee, inventor of the Web, offers similar advice. (See http://www.w3.org/Provider/Style/URI.)

GET and POST: Pick the Right One

The rule of thumb is that `GET` requests should be used for all read-only information retrieval, while `POST` requests should be used for any operation that changes the application state. In standards-compliance terms, `GET` requests are for *safe* interactions (having no side effects besides information retrieval), and `POST` requests are for *unsafe* interactions (making a decision or changing something). These conventions are set by the World Wide Web Consortium (W3C), at http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

`GET` requests are *addressable*: all the information is contained in the URL, so it's possible to bookmark and link to these addresses.

Do not use `GET` requests for operations that change state. Many Web developers learned this the hard way in 2005, when Google Web Accelerator was released to the public. This application pre-fetched all the content linked from each page, which is legal within the HTTP because `GET` requests should be safe. Unfortunately, many Web developers had ignored the HTTP conventions and placed simple links to "delete item" or "add to shopping cart" in their applications. Chaos ensued.

One company believed their content management system was the target of repeated hostile attacks, because all their content kept getting deleted. They later discovered that a search-engine crawler had hit upon the URL of an administrative page and was crawling all the delete links. Authentication might protect you from this, but it wouldn't protect you from Web accelerators.

Summary

In this chapter, I have shown you the advanced features of the MVC Framework routing system, showing you how to generate outgoing links and URLs, and how to customize the routing system. Along the way, I introduced the concept of areas and set out my views on how to create a useful and meaningful URL schema. In the next chapter, I turn to controllers and actions, which are the heart of the MVC Framework. I explain how these work in detail and show you how to use them to get the best results in your application.