CHAPTER 2

Understanding URL Routing

IN THIS CHAPTER:

URL Routing is responsible for mapping a browser request to a particular controller and controller action. By taking advantage of URL Routing, you can control how the URLs that users enter into their browser address bars invoke the controller actions in your MVC applications.

In the first part of this chapter, you learn about the Default route that you get when you create a new ASP.NET MVC application. You learn about the standard parts of any route.

Next, you learn how to create custom routes. You learn how to extract custom parameters from a URL. You also learn how to create custom constraints that restrict the URLs that match a particular route

We also discuss how you can mix ASP.NET MVC pages with ASP.NET Web Forms pages. You learn how to incrementally migrate an existing ASP.NET Web Forms application to an ASP.NET MVC application.

You also learn how to configure URL Routing to work with different versions of Internet Information Services (IIS). In particular, you learn how to get URL Routing to work with IIS 6.0 and IIS 7.0 (in classic pipeline mode).

Finally, we discuss the very important topic of testing. You learn how to test custom routes, and custom route constraints, by building unit tests.

NOTE

URL Routing also can be used for generating URLs. We delay the exploration of this topic until Chapter 6, Using HTML Helpers.

Using the Default Route

You configure URL Routing in an application's Global.asax file. This makes sense because the Global.asax file contains event handlers for application lifecycle events such as the application Start and application End events. Because you want your routes to be enabled when an application first starts, routing is setup in the application Start event.

When you create a new ASP.NET MVC application, you get the Global.asax file in Listing 1.

Listing 1 – Global.asax.vb

```
Public Class GlobalApplication
    Inherits System.Web.HttpApplication

    Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

        ' MapRoute takes the following parameters, in order:
        ' (1) Route name
        ' (2) URL with parameters
        ' (3) Parameter defaults
        routes.MapRoute( _
            "Default", _
            "{controller}/{action}/{id}", _
            New With {.controller = "Home", .action = "Index", .id = ""} _
        )

    End Sub

    Sub Application_Start()
        RegisterRoutes(RouteTable.Routes)
    End Sub
End Class
```

The Global.asax file in Listing 2 includes two methods named Application_Start() and RegisterRoutes(). The Application_Start() method is called once, and only once, when an ASP.NET application first starts. In Listing 1, the Application_Start() method simply calls the RegisterRoutes() method.

NOTE

Why does the Global.asax include a separate method called RegisterRoutes()? Why isn't the code in the RegisterRoutes() method contained in the Application_Start() method? A separate method was created to improve testability. You can call the RegisterRoutes() method from your unit tests with a route table that is different than the default route table.

The RegisterRoutes() method is used to configure all of the routes in an application. The RegisterRoutes() method in Listing 1 configures the Default route with the following code:

```
routes.MapRoute( _
    "Default", _
    "{controller}/{action}/{id}", _
    New With {.controller = "Home", .action = "Index", .id = ""} _
)
```

You configure a new route by calling the MapRoute() method. This method accepts the following parameters:

Name – The name of the route

URL – The URL pattern for the route

Defaults – The default values of the route parameters

Constraints – A set of constraints that restrict the requests that match the route

The MapRoute() method has multiple overloads. You can call the MapRoute() method without supplying the Defaults or Constraints parameters.

The default route configured in the Global.asax file in Listing 1 is named, appropriate enough, *Default*. You aren't required to supply a route name. However, supplying a route name makes it easier to generate URLs from a route and unit test the route.

The URL parameter for the Default route matches URLs that satisfy the pattern {controller}/{action}/{id}. Therefore, the Default route matches URLs that look like this:

/Product/Insert/23

/Home/Index/1

/Do/Something/Useful

However, the Default route does not match a URL that looks like this:

/Product/Insert/Another/Item

The problem with this last URL is that it has too many segments. It has four different segments (four forward slashes) and the URL pattern {controller}/{action}/{id} only matches URLs that have three segments.

The URL pattern {controller}/{action}/{id} maps the first segment to a parameter named *controller*, the second segment to a parameter named *action*, and the final segment to a parameter named *id*.

The *controller* and *action* parameters are special. The ASP.NET MVC framework uses the controller parameter to determine which MVC controller to use to handle the request. The *action* parameter represents the action to call on the controller in response to the request.

If you create additional parameters – parameters that are not named *controller* or *action* – then the additional parameters are passed to an MVC controller action when the action is invoked. For example, the *id* parameter is passed as a parameter to a controller action.

Finally, the Default route includes a set of default values for the controller, action, and id parameters. By default, the controller parameter has the value Home, the action parameter has the value Index, and the id parameter has the value "" (empty string).

For example, imagine that you enter the following URL into the address bar of your browser:

http://www.MySite.com/Product

In that case, the controller, action, and id parameter would have the following values:

controller : Product

action: Index

id : ""

Now, imagine that you request the default page for a website:

http://www.MySite.com/

In that case, the controller, action, and id parameters would have the following values:

controller : Home

action: Index

id : ""

In this case, the ASP.NET MVC framework would invoke the Index() action method on the HomeController class.

NOTE

The code used to specify the defaults for a route might appear strange to you. This code is taking advantage of two new features of the Visual Basic .NET 9.0 and the C# 3.0 languages called anonymous types and property initializers.

Technically, the MapRoute() method expects an object for the Defaults parameter and you are taking advantage of the anonymous types feature to create a new object on the fly. If you prefer, you could rewrite the code so that you explicitly create a new HomeDefaults class this:

```
Public Class HomeDefaults

    Private _controller As String
    Private _action As String
```

```
    Public Property Controller() As String
        Get
            Return _controller
        End Get
        Set(ByVal value As String)
            _controller = value
        End Set
    End Property

    Public Property Action() As String
        Get
            Return _action
        End Get
        Set(ByVal value As String)
            _action = value
        End Set
    End Property

End Class
```

And then you can initialize and pass an instance of the HomeDefaults class to the MapRoute() method like this:

```
Dim homeDefaults As New HomeDefaults()
homeDefaults.Controller = "Home"
homeDefaults.Action = "Index"

routes.MapRoute( _
    "Home", _
    "{controller}/{action}", _
    homeDefaults _
)
```

But, this is a huge amount of work for something that can be done with much less fuss by taking advantage of an anonymous type.

Debugging Routes

In the next section, I show you how you can add custom routes to the Global.asax file. However, before we start creating custom routes, it is important to have some way of debugging our routes. Otherwise, things get confusing fast.

Included with the code that accompanies this book is a project named RouteDebugger. If you add a reference to the assembly generated by this project then you can debug the routes configured within any ASP.NET MVC application.

NOTE

The RouteDebugger project is included in the Code\VB\Common\RouteDebugger and the Code\CS\Common\RouteDebugger folder.

Here's how you add a reference to the RouteDebugger assembly. Select the menu option **Project**, **Add Reference** to open the Add Reference dialog box (see Figure 1). Select the Browse tab and browse to the assembly named RouteDebugger.dll located in the RouteDebugger\Bin\Debug folder. Click the OK button to add the assembly to your project.

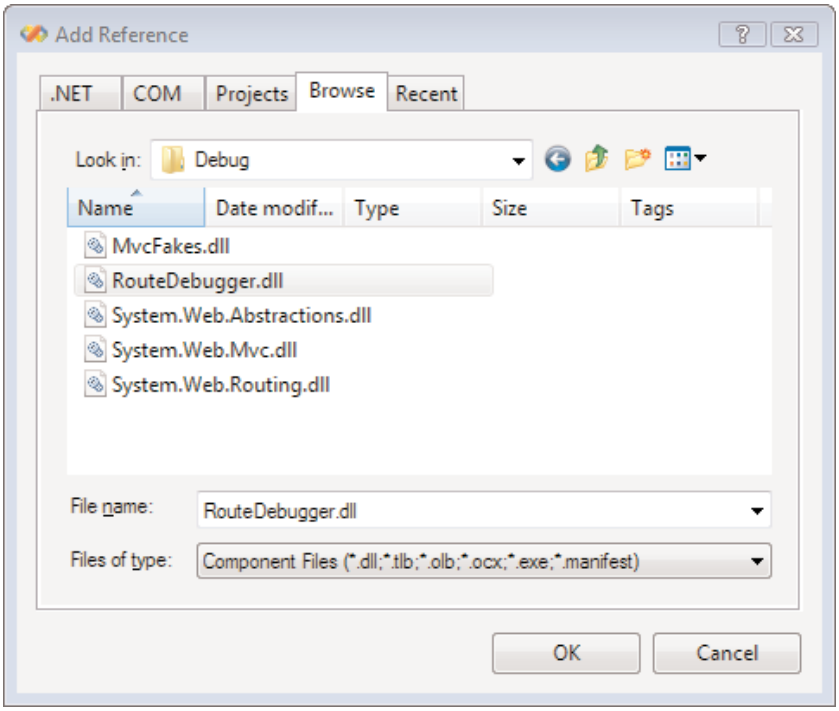

FIGURE 1 Using the Add Reference dialog box

After you add the RouteDebugger assembly to an ASP.NET MVC project, you can debug the routes in the project by entering the following URL into the address bar of your browser:

```
/RouteDebugger
```

Invoking the RouteDebugger displays the page in Figure 2. You can enter any relative URL into the form field and view the routes that match the URL. The URL should be an *application relative* URL and start with the tilde character ~.

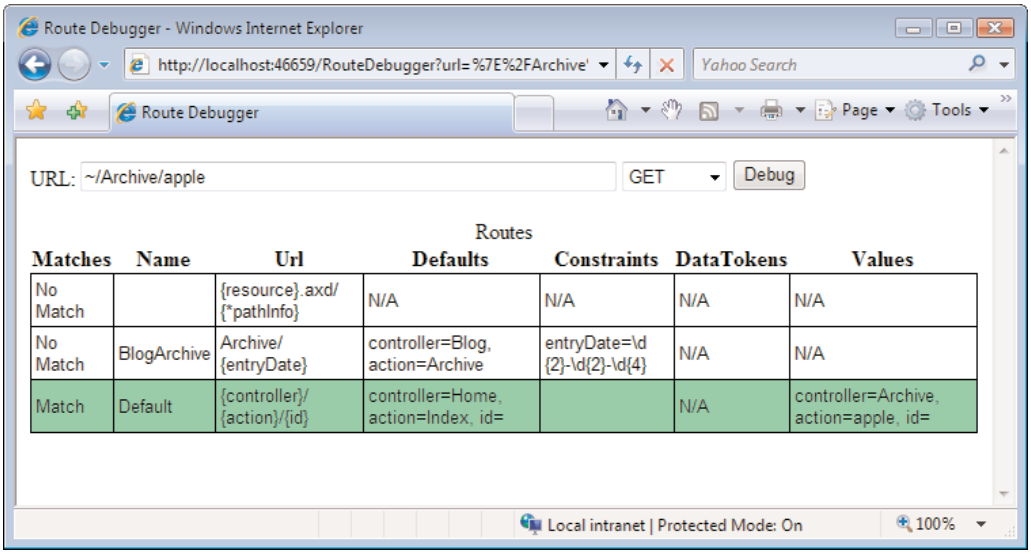

FIGURE 2 Using the Route Debugger

Whenever you enter a URL into the Route Debugger, the Route Debugger displays all of the routes from the application's route table. Each route that matches the URL is displayed with a green background. The first route that matches the URL is the route that would actually be invoked when the application is running.

WARNING

Be warned that you cannot invoke the RouteDebugger unless your application includes a route that maps to the RouteDebugger controller. If the RouteDebugger stops working then you can add the following route to the Global.asax file before any other route:

```
routes.MapRoute( _
    "RouteDebugger", _
    "RouteDebugger", _
    New With {.controller = "RouteDebugger", .action = "Index"} _
)
```

Creating Custom Routes

You can build an entire ASP.NET MVC application without creating a single custom route. However, there are situations in which it makes sense to create a new route. For example, imagine that you are creating a blog application and you want to route requests that look like this:

/Archive/12-25-2008

When someone requests this URL, you want to display blog entries for the date 12-25-2008.

The Default route defined in the Global.asax would extract the following parameters from this URL:

controller: Archive

action: 12-25-1966

This is wrong. You don't want to invoke a controller action named 12-25-1966. Instead, you want to pass this date to a controller action.

Listing 2 contains a custom route, named BlogArchive, which correctly handles requests for blog entries.

LISTING 2 Global.asax (with BlogArchive Route)

```
Public Class GlobalApplication
    Inherits System.Web.HttpApplication

    Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

        routes.MapRoute( _
            "BlogArchive", _
            "Archive/{entryDate}", _
            New With {.controller = "Blog", .action = "Archive"}
        )

        routes.MapRoute( _
            "Default", _
            "{controller}/{action}/{id}", _
            New With {.controller = "Home", .action = "Index", .id = ""} _
        )
    End Sub

    Sub Application_Start()
        RegisterRoutes(RouteTable.Routes)
    End Sub
End Class
```

The BlogArchive route matches any URL that satisfies the pattern /Archive/{entryDate}. The route invokes a controller named Blog and a controller action method named Archive(). The entryDate parameter is passed to the Archive() action method.

You can use the controller in Listing 3 with the BlogArchive route. This controller contains an Archive() action method that echoes back the value of the entryDate parameter.

LISTING 3 BlogController.vb

```
Imports System
Imports System.Web.Mvc

Public Class BlogController
        Inherits Controller

        Public Function Archive(ByVal entryDate As DateTime) As String
                        Return entryDate.ToString()
        End Function


End Class
```

The order that you add a custom route to the Global.asax file is important. The first route matched is used. For example, if you reverse the order of the BlogArchive and Default routes in Listing 2, then the Default route would always be executed instead of the BlogArchive route.

WARNING

The order of your routes in the Global.asax file matters.

Creating Route Constraints

When you create a custom route, you can include route constraints. A constraint restricts the requests that match a route. There are three basic types of constraints: regular expression constraints, the HttpMethod constraint, and custom constraints.

Using Regular Expression Constraints

You can use a regular expression constraint to prevent a request from matching a route unless a parameter extracted from the request matches a particular regular expression pattern. You can use regular expressions to match just about any string pattern including currency amounts, dates, times, and numeric formats.

For example, the BlogArchive custom route that we created in the previous section was created like this:

```
routes.MapRoute( _
    "BlogArchive", _
    "Archive/{entryDate}", _
    New With {.controller = "Blog", .action = "Archive"}
)
```

This custom route matches the following URLs:

/Archive/12-25-1966

/Archive/02-09-1978

Unfortunately, the route also matches these URLs:

/Archive/apple

/Archive/blah

There is nothing to prevent you from entering something that is not a date in the URL. If you request a URL like /Archive/apple, then you get the error page in Figure 3.

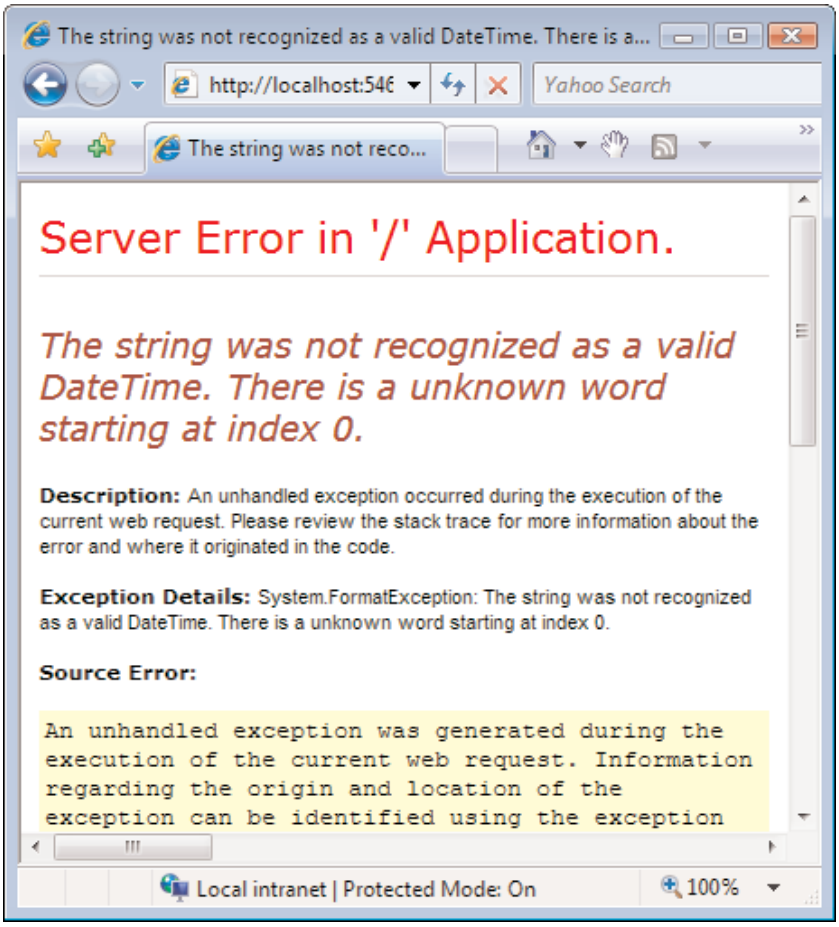

FIGURE 3 Entering a URL with an invalid date

We really need to prevent URLS that don't contain dates from matching our BlogArchive route. The easiest way to fix our route is to add a regular expression constraint. The following modified version of the BlogArchive route won't match URLs that don't contain dates in the format 01-01-0001:

```
routes.MapRoute( _
  "BlogArchive", _
  "Archive/{entryDate}", _
  New With {.controller = "Blog", .action = "Archive"}, _
  New With {.entryDate = "\d{2}-\d{2}-\d{4}"} _
)
```

The fourth parameter passed to the MapRoute() method represents the constraints. This constraint prevents a request from matching this route when the entryDate parameter does not match the regular expression \d{2}-\d{2}-d{4}. In other words, the entryDate must match the pattern of two decimals followed by a dash followed by two decimals followed by a dash followed by four decimals.

You can quickly test your new version of the BlogArchive route with the RouteDebugger. The page in Figure 4 shows the matched routes when an invalid date is entered. Notice that the BlogArchive route *is not* matched.

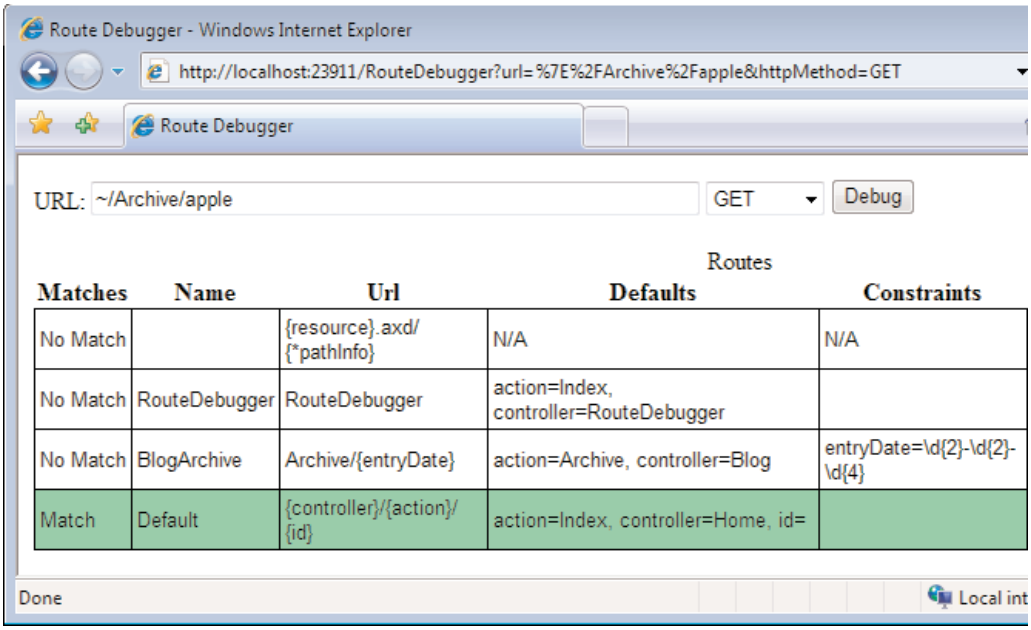

FIGURE 4 Using the RouteDebugger with the modified BlogArchive route

Using the HttpMethod Constraint

The URL Routing framework includes a special constraint named the HttpMethod constraint. You can use the HttpMethod constraint to match a route to a particular type of HTTP operation. For example, you might want to prevent a particular URL from being accessed when performing an HTTP GET but not when performing an HTTP POST.

For example, the following route, named ProductInsert, can be called only when performing an HTTP POST operation:

```
routes.MapRoute( _
    "ProductInsert", _
    "Product/Insert", _
     New With {.controller = "Product", .action = "Insert"}, _
     New With {.method = New HttpMethodConstraint("POST")} _
)
```

You can check whether the ProductInsert route really works by taking advantage of the RouteDebugger. The RouteDebugger enables you to pick an HTTP method that you want to simulate. The page in Figure 5 illustrates testing the ProductInsert route when performing an HTTP POST operation.

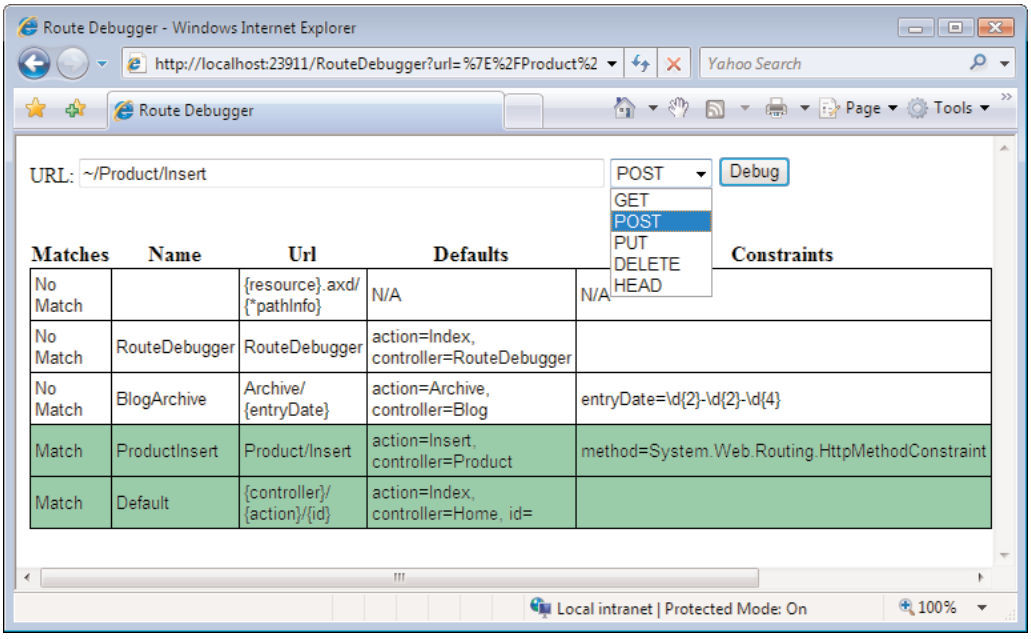

FIGURE 5 Matching the ProductInsert route when performing an HTTP POST

Creating an Authenticated Constraint

If you need to create a more complicated constraint — something that you cannot easily represent with a regular expression — then you can create a custom constraint. You create a custom constraint by creating a class that implements the IRouteConstraint interface. This interface is really easy to implement since it includes only one method: the Match() method.

For example, Listing 4 contains a new constraint named the AuthenticatedConstraint. The AuthenticatedConstraint prevents a request from matching a route when the request is not made by an authenticated user.

LISTING 4 Constraints\AuthenticatedConstraint.vb

```
Imports System.Web
Imports System.Web.Routing

Public Class AuthenticatedConstraint
    Implements IRouteConstraint

    Public Function Match _
    ( _
        ByVal httpContext As HttpContextBase, _
        ByVal route As Route, _
        ByVal parameterName As String, _
        ByVal values As RouteValueDictionary, _
        ByVal routeDirection As RouteDirection _
    ) As Boolean Implements IRouteConstraint.Match
        Return HttpContext.Request.IsAuthenticated
    End Function
End Class
```

In Listing 4, the Match() method simply returns the value of the HttpContext.Request.IsAuthenticated property to determine whether the current request is an authenticated request. If the Match() method returns the value False, then the request fails to match the constraint and the route is not matched.

After you create the AuthenticatedConstraint, you can use it with a route like this:

```
routes.MapRoute( _
    "Admin", _
    "Admin/{action}", _
    New With {.controller = "Admin"}, _
    New With {.Auth = New AuthenticatedConstraint()} _
)
```

It is important to understand that the AuthenticatedConstraint prevents only a particular route from matching a request. Another route, that does not include the

AuthenticatedConstraint, might match the very same request and invoke the very same controller action. In the next section, I show you how to create a constraint that prevents a route from ever invoking a particular controller.

Creating a NotEqual Constraint

If you want to create a route that will never match a particular controller action — or more generally, that will never match a particular route parameter value – then you can create a NotEqual constraint.

The code for the NotEqual constraint is contained in Listing 5.

LISTING 5 NotEqual.vb

```
Public Class NotEqual
    Implements IRouteConstraint

    Private _value As String

    Sub New(ByVal value As String)
        _value = value
    End Sub

    Public Function Match( _
        ByVal httpContext As HttpContextBase, _
        ByVal route As Route, _
        ByVal parameterName As String, _
        ByVal values As RouteValueDictionary, _
        ByVal routeDirection As RouteDirection _
    ) As Boolean Implements IRouteConstraint.Match
        Dim paramValue = values(parameterName).ToString()
        Return String.Compare(paramValue, _value, True) <> 0
    End Function

End Class
```

The NotEqual constraint performs a case-insensitive match of the value of a parameter against a field named _value. If there is a match, the constraint fails and the route is skipped.

After you create the NotEqual constraint, you can create a route that uses the constraint like this:

```
routes.MapRoute( _
    "DefaultNoAdmin", _
    "{controller}/{action}/{id}", _
    New With {.controller = "Home", .action = "Index", .id = ""}, _
```

```
        New With {.controller = New NotEqual("Admin")} _
    )
```

This route works just like the Default route except for the fact that it will never match when the controller parameter has the value Admin. You can test the NotEqual constraint with the RouteDebugger. In Figure 6, the URL /Admin/Delete matches the Default route, but it does not match the DefaultNoAdmin route.

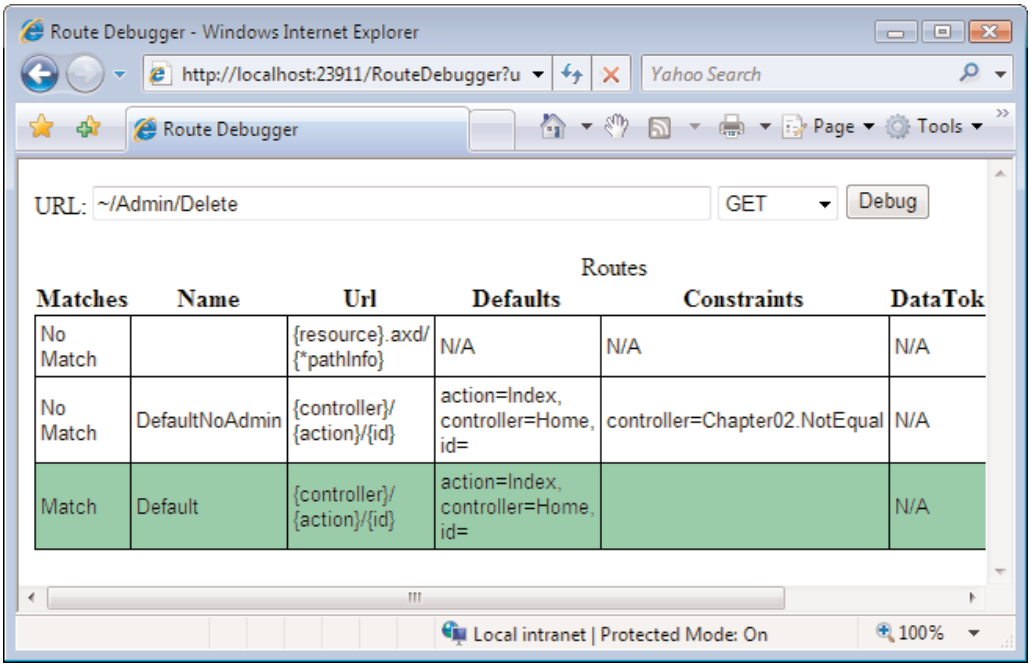

FIGURE 6 Using the NotEqual Constraint

Using Catch-All Routes

Normally, in order to match a route, a URL must contain a particular number of segments. For example, the URL /Product/Details matches the following route:

```
routes.MapRoute( _
    "Product1", _
    "Product/{action}", _
    New With {.controller = "Product"} _
)
```

However, this URL does not match the following route:

```
routes.MapRoute( _
    "Product2", _
```

```
        "Product/{action}/{id}", _
        New With {.controller = "Product"} _
    )
```

This route requires a URL to have 3 segments and the URL /Product/Details has only two segments (two forward slashes).

NOTE

The URL being requested is not required to have the same number of segments as a route's URL parameter. When a route parameter has a default value, a segment is optional. For example, the URL /Home/Index matches a route that has the URL pattern {controller}/{action}/{id} when the id parameter has a default value.

If you want to match a URL, regardless of the number of segments in the URL, then you need to create something called a catch-all parameter. Here's an example of a route that uses a catch-all parameter:

```
routes.MapRoute( _
    "SortRoute", _
    "Sort/{*values}", _
    New With {.controller = "Sort", .action = "Index"} _
)
```

Notice that the route parameter named values has a star in front of its name. The star marks the parameter as a catch-all parameter. This route matches any of the following URLs:

/Sort

/Sort/a/b/d/c

/Sort/Women/Fire/Dangerous/Things

All of the segments after the first segment are captured by the catch-all parameter.

NOTE

A catch-all parameter must appear as the very last parameter. Think of a catch-all parameter as a Visual Basic .NET parameter array.

The Sort controller in Listing 6 illustrates how you can retrieve the value of a catch-all parameter within a controller action.

LISTING 6 SortController.vb

```
Imports System
Imports System.Web.Mvc
```

```
Public Class SortController
    Inherits Controller

    Public Function Index(ByVal values As String) As String
        Dim brokenValues = values.Split("/"c)
        Array.Sort(brokenValues)
        Return String.Join(", ", brokenValues)
    End Function

End Class
```

Notice that the catch-all parameter is passed to the Index() action as a string (you *cannot* pass the value as an array). The Index() method simply sorts the values contained in the catch-all parameter and returns a string with the values in alphabetical order (see Figure 7).

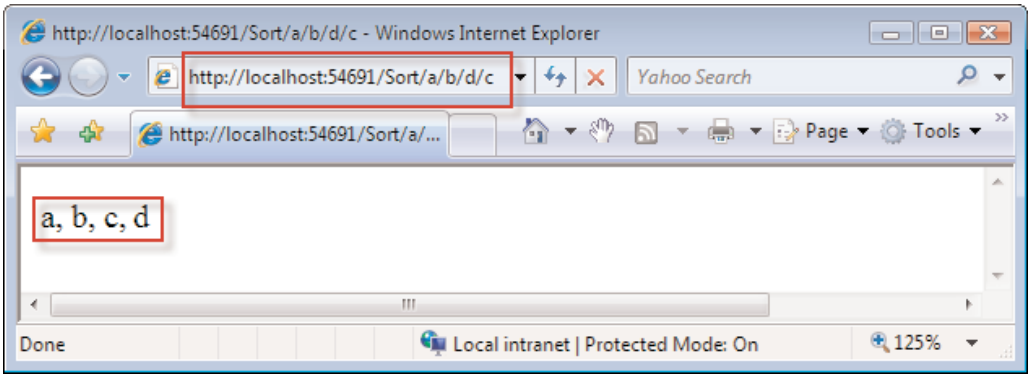

FIGURE 7 Using a catch-all parameter

Mixing MVC with Web Forms

You can gradually migrate an existing ASP.NET Web Forms application into an ASP.NET MVC application. ASP.NET Web Form pages and ASP.NET MVC views can be happily comingled in the same application.

In this section, we go through the steps required to convert an existing ASP.NET Web Forms application to an ASP.NET MVC application. You learn two methods of handling requests for Web Form pages from an ASP.NET MVC application.

Steps for Converting from Web Forms to MVC

The easiest way to convert a Web Forms application to an MVC application is to create a new ASP.NET MVC Application Project and copy and paste the files from the original

application to the new application. Follow these steps:

1. Open your original Web Forms Application
2. Select the menu option File, New Project to open the New Project dialog box (see Figure 8). Pick the ASP.NET MVC Web Application project template, give your new project a good name, select Add to Solution, and click the OK button.
3. Add all of the references from the original Web Forms application to the new ASP.NET MVC application.
4. Copy all of the files from your original Web Forms project to the new ASP.NET MVC Web Application project. *Don't overwrite the Web.config file in the new ASP.NET Web Application project.*

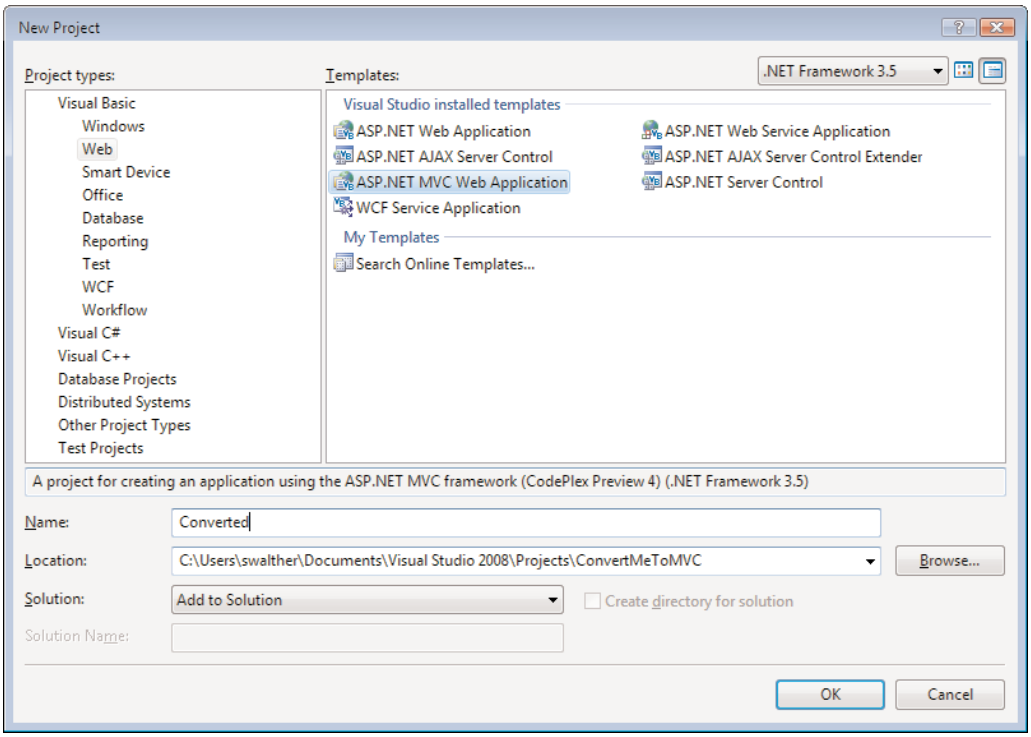

FIGURE 8 Add a new ASP.NET MVC Web Application Project

If your original Web Forms application was a website instead of a Web Application Project, then you need to do some additional work. You need to convert each .aspx file that you copied to the ASP.NET MVC Web Application. Right-click each .aspx file and select the menu option **Convert to Web Application**.

Converting an .aspx file adds an extra file to your project that ends with the extension .designer.vb or .designer.cs (depending on the language of your project). You can see these

new files by selecting the menu option **Project, Show All Files**.

NOTE

What's the difference between a website and a Web Application Project? You create websites by selecting the menu option **File, New Website** and you create Web Application Projects by selecting the menu option **File, New Project**)

In a Web Application Project, all of the files in the project are compiled into a single assembly in the /Bin folder. The files in a website, in contrast, are compiled dynamically.

Furthermore, a Web Application Project uses a project file to keep track of all of the files in the project. The project file is named either *project name*.vbproj or *project name*.csproj depending on the language.

WARNING

Do not add a folder named App_Code to a Web Application Project. Any classes in the App_Code file get compiled twice. Migrate all of the classes from the App_Code folder to a folder with a new name.

Routing Existing Files

After you copy a Web Form page into an ASP.NET MVC application, the Web Form Page should just work. You can request the page by entering the path to the page into your web browser just like you would in the case of a normal ASP.NET Web Forms application.

By default, the URL Routing module will respect requests for any file that actually exists on the hard drive. For example, if you make a request for a page named /Home/SomePage.aspx and the page SomePage.aspx actually exists in the Home folder, then this page is returned.

You can disable or enable this feature by modifying the value of the RouteExistingFiles property of the RouteCollection class. By default, the RouteExistingFiles property has the value False. In other words, URL Routing won't intercept request for existing files by default.

If you add the following line of code to the RegisterRoutes() method in the Global.asax file, then existing files in your ASP.NET MVC application are handled by the URL Routing module:

```
routes.RouteExistingFiles = True
```

This line of code prevents someone from requesting the /Home/SomePage.aspx page directly even when the page actually exists. Enabling the RouteExistingFiles property forces all requests to be processed by the URL Routing module.

Routing to Web Form Pages

Normally, the URL Routing module routes requests to ASP.NET MVC controllers. However, with a little work, you also can use the Routing Module to route requests to Web Form pages.

Before you can route browser requests to Web Form pages, you must create a new Route Handler. The class in Listing 7 implements a WebFormsRouteHandler.

LISTING 7 RouteHandlers\WebFormsRouteHandler.vb

```
Imports System.Security
Imports System.Web
Imports System.Web.Compilation
Imports System.Web.Routing
Imports System.Web.Security
Imports System.Web.UI

Public Class WebFormsRouteHandler
    Implements IRouteHandler

    Private _virtualPath As String

    Public Sub New(ByVal virtualPath As String)
        _virtualPath = virtualPath
    End Sub

    Public Function GetHttpHandler(ByVal requestContext As RequestContext) As
IHttpHandler Implements IRouteHandler.GetHttpHandler
        If (Not UrlAuthorizationModule.CheckUrlAccessForPrincipal(_virtualPath,
requestContext.HttpContext.User, requestContext.HttpContext.Request.HttpMethod))
Then
            Throw New SecurityException()
        End If

        Return TryCast(BuildManager.CreateInstanceFromVirtualPath(_virtualPath,
GetType(Page)), IHttpHandler)
    End Function

End Class
```

You implement the IRouteHandler interface by creating a class that includes a GetHttphandler() method. In Listing 7, this method instantiates a new Web Forms page given the path to the .aspx file.

Here's how you can use the WebFormsRouteHandler when creating a route:

```
routes.MapRoute( _
    "SomePage", _
    "Page/Some", _
    Nothing, _
    Nothing, _
    New WebFormsRouteHandler("~/SomePage.aspx") _
)
```

This route maps requests for Page/Some to the Web Forms page SomePage.aspx. In other words, if you enter the URL Page/Some into the address bar of your web browser, then SomePage.aspx is returned.

WARNING

You must add a reference to the RouteDebugger assembly in order to use the version of the MapRoute() method described in this section.

You can verify that the WebFormsRouteHandler is working correctly by taking advantage of the RouteDebugger. In Figure 10 illustrates that entering the URL Page/Some returns the SomePage.aspx Web Forms page.

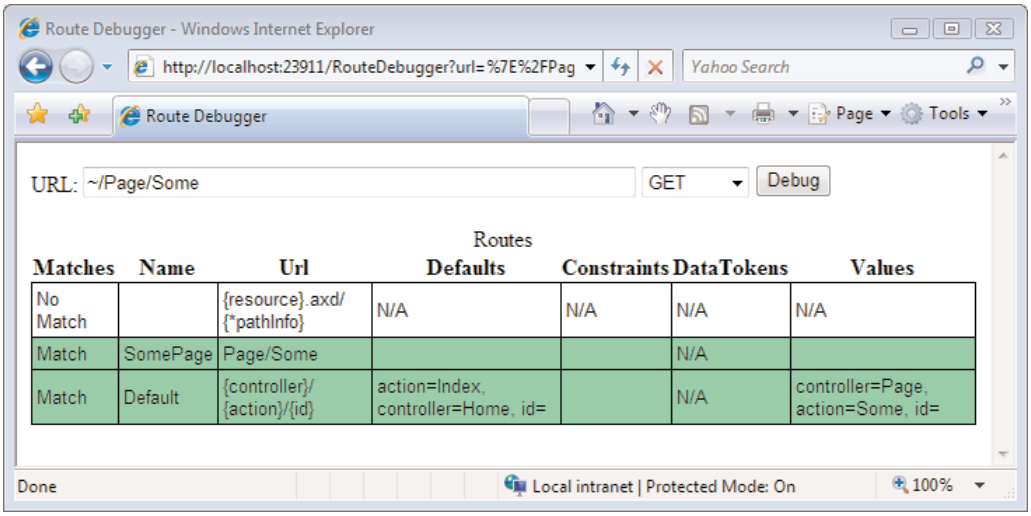

FIGURE 9 Routing to a Web Forms Page

NOTE

The WebFormsRouteHandler discussed in this section is based on Phil Haack's WebFormsRouteHandler. Phi Haack is a coworker of mine at Microsoft. To learn more, see:

http://haacked.com/archive/2008/03/11/using-routing-with-webforms.aspx

Using Routing with IIS

By default, when you run a project in Visual Studio 2008, Visual Studio launches a mini web server named the ASP.NET Development Server. Clearly, you don't need to do anything special to use URL Routing with the ASP.NET Development Web Server.

URL Routing also works just fine with Internet Information Services 7.0 (IIS 7.0). If IIS 7.0 is running in integrated pipeline mode, then you don't need to do anything special to get URL Routing to work.

NOTE

IIS 7.0 can run a Web application in two modes: integrated pipeline mode and classic pipeline mode. The pipeline mode is determined by the application pool associated with the Web application.

Here's how you can determine whether a particular application is running in integrated or classic mode. Within IIS 7.0, select an application and click the Basic Settings link. If the Application Pool setting has the value DefaultAppPool then the application is running in integrated pipeline mode. If, on the other hand, the Application Pool setting has the value Classic .NET App Pool, then your application is running in classic pipeline mode.

However, you will need to do some extra work to get URL Routing to work with IIS 6.0 (or IIS 7.0 when IIS 7.0 is running in classic pipeline mode). The problem, when using IIS 6.0, is that requests like /Home/Index never get passed to an ASP.NET application. These requests get handled by IIS itself before they ever get to the ASP.NET framework.

NOTE

IIS 6.0 is the version of IIS included with Microsoft Windows Server 2003.

There are two ways to get URL Routing to work with IIS 6.0. You can either configure a wildcard script map or you can change your routes so that they include a file extension.

If you have access to your IIS 6.0 web server, then you can enable a wildcard script map that maps all incoming requests to the ASP.NET framework. Follow these steps:

1. Right-click a website and select Properties
2. Select the **Home Directory** tab
3. Click the **Configuration** button
4. Select the **Mappings** tab
5. Click the **Insert** button (see Figure 10)
6. Paste the path to the aspnet_isapi.dll into the Executable field (you can copy this path from the script map for .aspx files)

7. Uncheck the checkbox labeled **Verify that file exists**
8. Click the **OK** button

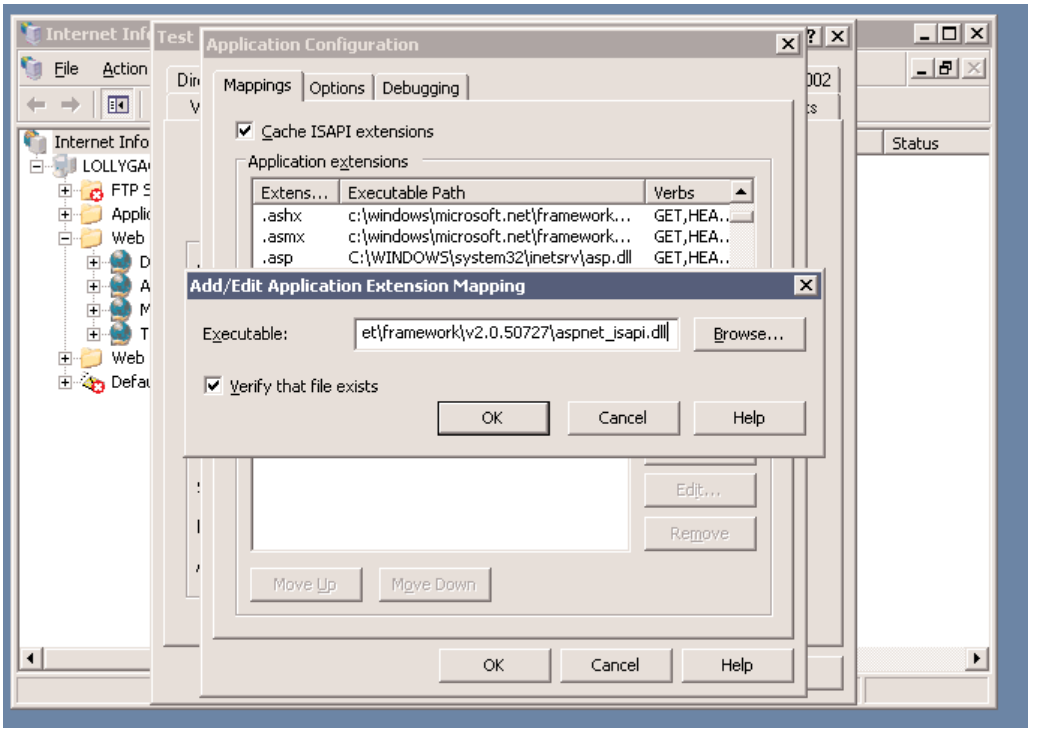

FIGURE 10 Adding a wildcard script map to IIS 6.0

You can follow a similar set of steps to enable a wildcard script map in IIS 7.0 when an application is configured to use classic pipeline mode:

1. Select your application in the Connections window
2. Make sure that the **Features** view is selected
3. Double-click the **Handler Mappings** button
4. Click the **Add Wildcard Script Map** link (see Figure 11)
5. Enter the path to the aspnet_isapi.dll file (You can copy this path from the PageHandlerFactory script map)
6. Enter the name MVC
7. Click the **OK** button

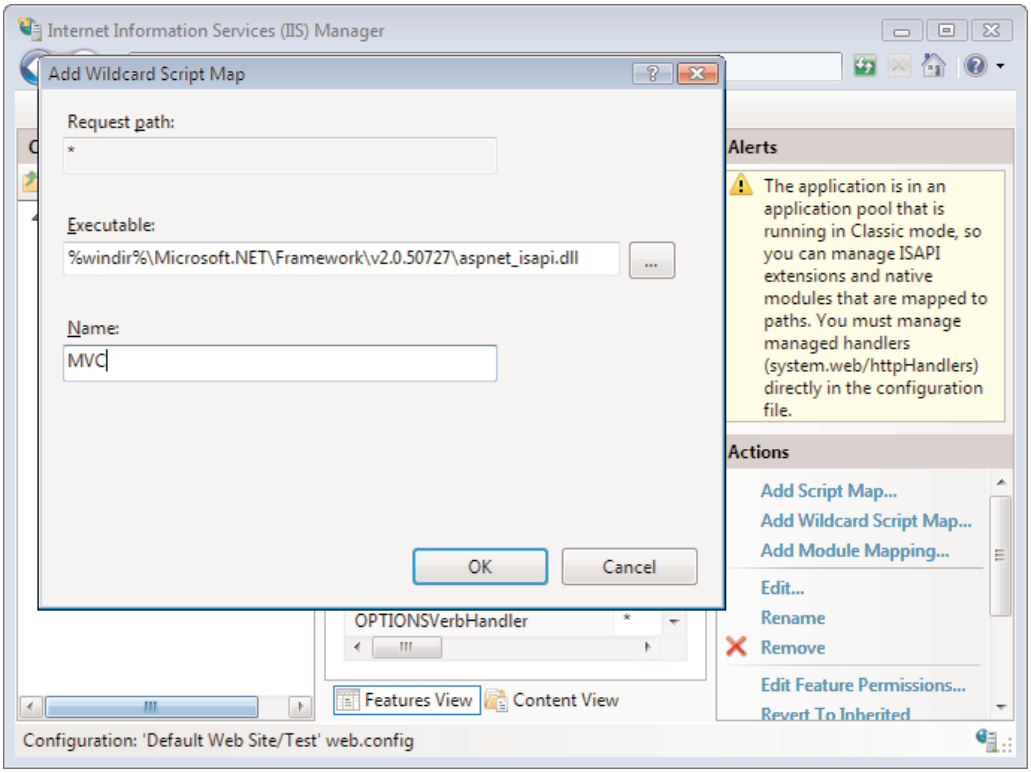

FIGURE 11 Adding a wildcard script map to IIS 7.0

However, there are situations in which you don't have access to the web server. For example, you might want to host an ASP.NET MVC application with an Internet Hosting Provider. What do you do when you don't have access to the web server?

One solution (a slightly clunky solution) is to change your routes so that they require users to enter the .aspx file extension. Requests for .aspx files get processed by the ASP.NET framework automatically. So, you don't need to perform any configuration on the web server.

Here's how you can modify the Default route so that it requires the .aspx extension:

```
routes.MapRoute( _
   "Default", _
   "{controller}.aspx/{action}/{id}", _
   New With {.controller = "Home", .action = "Index", .id = ""} _
)
```

Notice that the .aspx extension now appears in the URL pattern right after {controller}. This modified Default route works with the following URLs:

/Home.aspx

/Home.aspx/Index/23

/Product.aspx/Details

The requirement that you enter the .aspx extension is not optimal. However, this approach does work with all IIS web servers hosted at any Internet Service Provider.

NOTE

When you create a new ASP.NET MVC Web Application project, you get a Default.aspx page automatically. If you are using IIS 7.0 or you have enabled wildcard mappings then you should delete this file. If, on the other hand, you are using file extensions in your routes, then you need to keep this Default.aspx file or requests for ~/ won't work.

NOTE

Taking advantage of wildcard mapping does have performance implications. When you enable wildcard mapping, all requests – even requests for images and requests for classic ASP pages – get passed to the ASP.NET framework.

Testing Routes

Every feature of the ASP.NET MVC framework was designed to be highly testable and URL Routing is no exception. In this section, I describe how you can unit test both your routes and your route constraints.

Why would you want to build unit tests for your routes? If you build route unit tests then you can detect whether changes in your application break existing functionality automatically. For example, if your existing routes are covered by unit tests, then you know immediately whether introducing a new route prevents an existing route from ever being called.

Using the MvcFakes and RouteDebugger Assemblies

In order to unit test your custom routes, I recommend that you add references to two assemblies: the RouteDebugger and the MvcFakes assemblies.

If you want to test your routes by name, then you need to add a reference to the RouteDebugger assembly. The RouteDebugger assembly replaces the anonymous routes in your MVC application with named routes. That way, you can build tests that check whether a particular route is called by name.

I also recommend that you add a reference to the MvcFakes assembly. The MvcFakes assembly contains a set of fake objects that you can use in your unit tests. For example, MvcFakes includes a FakeHttpContext object. You can use the FakeHttpContext object to fake every aspect of a browser request.

Both of these assemblies are included with the code that accompanies this book in a folder named Common. You can add references to these assemblies to your test project by selecting the menu option **Project. Add Reference**, selecting the Browse tab, and browsing to the following two assemblies (see Figure 12):

\Common\RouteDebugger\Bin\Debug\RouteDebugger.dll

\Common\MvcFakes\Bin\Debug\MvcFakes.dll

To use the MvcFakes assembly, you'll also need to add a reference to the System.Web.dll assembly. Select the menu option **Project**, **Add Reference**, and select the .NET tab. Select the System.Web assembly and click the OK button.

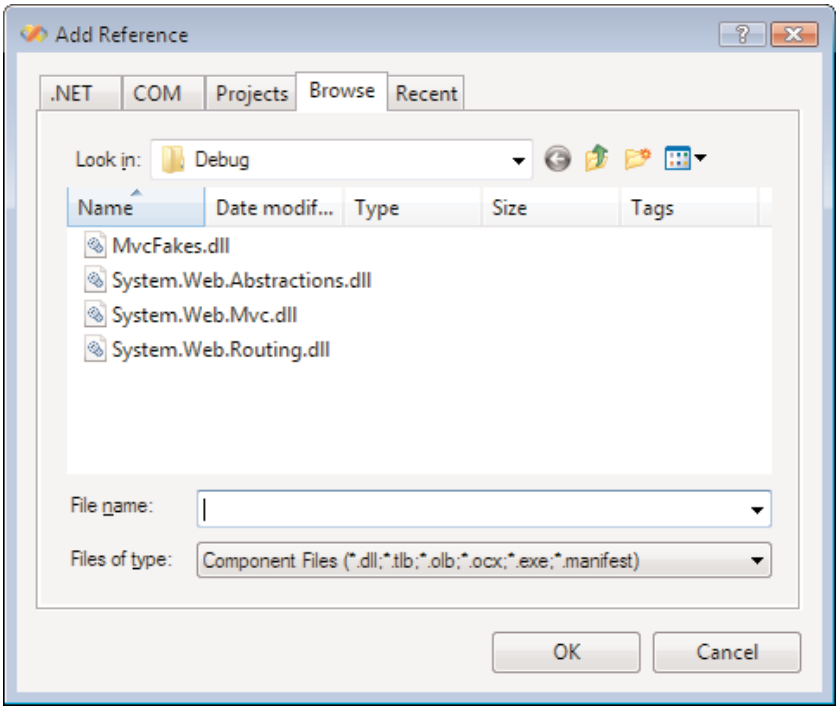

FIGURE 12 Adding a reference to an assembly

Testing If a URL Matches a Route

Let's start with a basic but very useful unit test. Let's create a unit test that verifies that a particular URL matches a particular route. The unit test is contained in Listing 8.

LISTING 8 Routes\RouteTest.vb

```
Imports System.Web.Routing
Imports Microsoft.VisualStudio.TestTools.UnitTesting
```

```
Imports MvcFakes
Imports RouteDebugger
Imports Chapter02

<TestClass()> _
Public Class RouteTest

    <TestMethod()> _
    Public Sub DefaultRouteMatchesHome()
        ' Arrange
        Dim routes = New RouteCollection()
        GlobalApplication.RegisterRoutes(routes)

        ' Act
        Dim context = New FakeHttpContext("~/Home")
        Dim routeData = routes.GetRouteData(context)

        ' Assert
        Dim matchedRoute = CType(routeData.Route, NamedRoute)
        Assert.AreEqual("Default", matchedRoute.Name)
    End Sub

End Class
```

You can add the unit test in Listing 8 to a Test project by selecting the menu option **Project, Add New Test** and selecting the Unit Test template (see Figure 13). Remember to add the assembly references discussed in the previous section or the unit test won't compile.

WARNING

Don't select the Unit Test Wizard. Also, don't select the tempting menu option **Project, Add New Unit Test**. Either option launches the Unit Test Wizard. The Unit Test Wizard creates a unit test that launches a web server (we don't want to do that). Instead, always pick the menu option Project, Add New Test and select the Unit Test template.

After you create the unit test in Listing 8, you can run it by entering the keyboard combination CTRL-R, A. Alternatively; you can click the **Run All Tests** in Solution button contained in the test toolbar (see Figure 14).

The test in Listing 8 verifies that the URL ~/Home matches the route named Default. The unit test consists of three parts.

The first part, the Arrange part, sets up the routes by creating a new route collection and passing the route collection to the RegisterRoutes() method exposed by the Global.asax file.

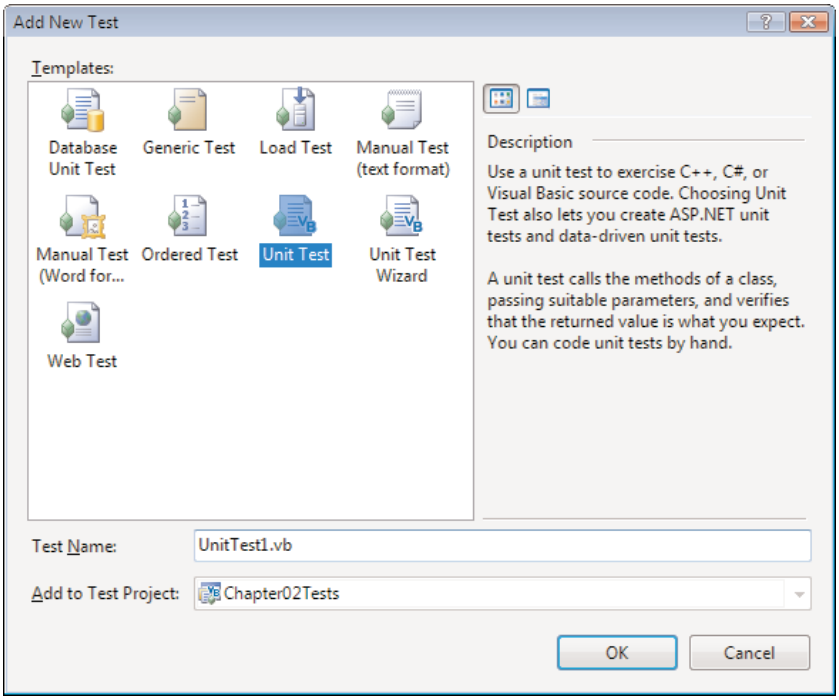

FIGURE 13 Adding a new unit test

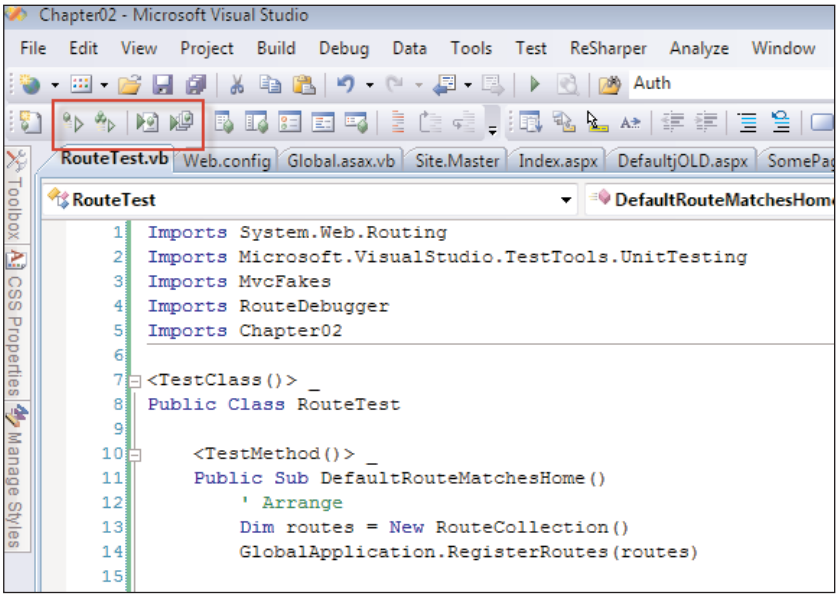

FIGURE 14 Run all tests in the solution

The second part, the Act part, sets up the fake HttpContext which represents the browser request for ~/Home. The FakeHttpContext object is part of the MvcFakes project. The FakeHttpContext object is passed to the GetRouteData() method. This method takes an HttpContext and returns a RouteData object that represents information about the route matched by the HttpContext.

Finally, the Assert part verifies that the RouteData represents a route named Default. At this point, the unit test either succeeds or fails. If it succeeds, then you get the test results in Figure 15.

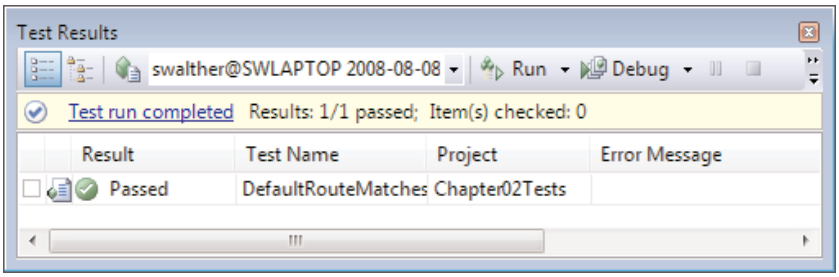

FIGURE 15 The test results

Testing Routes with Constraints

Let's try testing a slightly more complicated route. Earlier in this chapter, we discussed the HttpMethodConstraint which you can use to match a route only when the right HTTP method is used. For example, the following route should only match a browser request performed with an HTTP POST operation:

```
routes.MapRoute( _
    "ProductInsert", _
    "Product/Insert", _
    New With {.controller = "Product", .action = "Insert"}, _
    New With {.method = New HttpMethodConstraint("POST")} _
)
```

The HttpMethodConstraint restricts this route to match only POST requests. This is the intention, how do you test it? Easy, fake the HTTP operation with the HttpFakeContext object.

The unit test in Listing 9 contains two unit tests. The first test verifies that the ProductInsert route is matched when performing a POST operation. The second test verifies that the ProductInsert route is not matched when performing a GET operation.

LISTING 9 Routes\RouteTest.vb (with ProductInsert tests)

```
Imports System.Web.Routing
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports MvcFakes
```

```
Imports RouteDebugger
Imports Chapter02

<TestClass()> _
Public Class RouteTest

    <TestMethod()> _
    Public Sub ProductInsertMatchesPost()
        ' Arrange
        Dim routes = New RouteCollection()
        GlobalApplication.RegisterRoutes(routes)

        ' Act
        Dim context = New FakeHttpContext("~/Product/Insert", "POST")
        Dim routeData = routes.GetRouteData(context)

        ' Assert
        Dim matchedRoute = CType(routeData.Route, NamedRoute)
        Assert.AreEqual("ProductInsert", matchedRoute.Name)
    End Sub


    <TestMethod()> _
    Public Sub ProductInsertDoesNotMatchGet()
        ' Arrange
        Dim routes = New RouteCollection()
        GlobalApplication.RegisterRoutes(routes)

        ' Act
        Dim context = New FakeHttpContext("~/Product/Insert", "GET")
        Dim routeData = routes.GetRouteData(context)

        ' Assert
        If routeData IsNot Nothing Then
            Dim matchedRoute = CType(routeData.Route, NamedRoute)
            Assert.AreNotEqual("ProductInsert", matchedRoute.Name)
        End If
    End Sub

End Class
```

The second parameter passed to the constructor for the FakeHttpContext object determines the HTTP operation that the FakeHttpContext object represents.

Summary

In this chapter, you learned how to control how browser requests map to controllers and controller actions by taking advantage of URL Routing. We started by discussing the standard parts of a route.

You also learned how to debug the routes contained in the Global.asax file by taking advantage of the RouteDebugger assembly. We took advantage of the RouteDebugger when building our custom routes.

We also discussed how you can enable URL Routing in different version of Internet Information Services. You learned two methods of enabling URL Routing when using IIS 6.0.

Finally, you learned how to build unit tests for your custom routes. You learned how to take advantage of the FakeHttpContext object to fake different browser requests and test the routes that are matched.