With MvcContrib, NHibernate, and more

ASP.NET MVC IN ACTION

Jeffrey Palermo
Ben Scheirman
Jimmy Bogard

FOREWORD BY PHIL HAACK

SAMPLE CHAPTER

MANNING

ASP.NET MVC in Action

by Jeffrey Palermo
Ben Scheirman
and Jimmy Bogard

Chapter 5

brief contents

5 Routing

This chapter covers

- Routing as a solution to URL issues
- Designing a URL schema
- Using routing in ASP.NET MVC
- Route testing
- Using routing in Web Forms applications

Routing is all about the URL and how we use it as an external input to the applications that we build. The URL has led a short but troubled life and the HTTP URL is currently being tragically misused by current web technologies. As the web began to change from being a collection of hyperlinked static documents into dynamically created pages and applications, the URL has been kidnapped by web technologies and undergone terrible changes. The URL is in trouble and as the web becomes more dynamic we, as software developers, can rescue it to bring back the simple, logical, readable, and beautiful resource locator that it was meant to be.

Rescuing the URL means controlling those that control applications. Although routing is not core to all implementations of the MVC pattern, it is often implemented as a convenient way to add an extra level of separation between external

inputs and the controllers and actions which make up an application. The code required to implement routing using the ASP.NET MVC Framework is reasonably trivial but the thought behind designing a schema of URLs for an application can raise many issues. In this chapter, we'll go over the concept of routes and their relationship with MVC applications. We'll also briefly cover how they apply to Web Forms projects. We'll examine how to design a URL schema for an application, and then apply the concepts to create routes for Code Camp Server, our sample application. Because routes are the *front door* of your web application, we'll discover how to test routes to ensure they are working as intended. Now that you have an idea of how important routing is, we can start with the basics.

5.1 What are routes?

The history of the URL can be traced back to the very first web servers, where it was primarily used to point directly to documents in a folder structure. This URL would have been typical of an early URL and it's reasonably well structured and descriptive:

```
http://example.com/plants/roses.html
```

It seems to be pointing to information on roses and the domain also seems to have a logical hierarchy. But hold on, what is that .html extension on the end of the URL? This is where things started to go wrong for our friend the URL. Of course .html is a file extension because the web server is mapping the path in the URL directly to a folder of files on the disk of the web server. The category is being created by having a folder called *plants* containing all documents about plants.

The key thing here is that the file extension of .html is probably redundant in this context, as the content type is being specified by the `Content-Type` header returned as part of the HTTP response. An example HTTP header is shown in listing 5.1.

Listing 5.1 HTTP headers returned for a .html file

```
C:\> curl -I http://example.com/index.html

HTTP/1.1 200 OK
Date: Thu, 10 Jan 2008 09:03:29 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 15 Nov 2005 13:24:10 GMT
ETag: "280100-1b6-80bfd280"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

5.1.1 What's that curl command?

The `curl` command shown in listing 5.1 is a Unix command that allows you to issue an HTTP GET request for a URL and return the output. The `-I` switch tells it to display the HTTP response headers. This and other Unix commands are available on Windows via the Cygwin shell for Windows (http://cygwin.com).

The response returned contained a `Content-Type` header set to `text/html; charset=UTF8`, which specifies both a MIME type for the content and the character encoding. The file extension has no meaning in this situation.

> **File extensions are not all bad!**
>
> Reading this chapter so far, you might think that all file extensions are bad, but this is not the case. Knowing when information would be useful to the user is key to understanding when to use a file extension. Is it useful for the user to know that HTML has been generated from an .aspx source file? No, the MIME type is sufficient to influence how that content is displayed, so no extension should be shown. However, if a Word document is being served it would be good practice to include a .doc extension in addition to setting the correct MIME type, as that will be useful when the file is downloaded to the user's PC.

Mapping the path part of a URL directly to a disk folder is at the root of the problems that we face today. As dynamic web technologies have developed, .html files that contain information changed to be .aspx files containing source code. Suddenly the URL is not pointing to a document but to source code which fetches information from a database, and the filename must be generic as one source file can fetch any information it wants: what a mess!

Consider the following URL:

```
http://microsoft.com/downloads/details.aspx?FamilyID=9ae91ebe-3385-447c-8a30-
➥    081805b2f90b&displaylang=en
```

The file path is /download/details.aspx, which is a reasonable attempt to be descriptive with the source code name, but as it's a generic page which fetches the actual download details from a database, the file name can't possibly contain the important information that the URL should contain. Even worse, an unreadable GUID is used to identify the actual download and at this point the URL has lost all meaning.

This is a perfect opportunity to create a beautiful URL. Decouple the source code file name from the URL and it can become a resource locator again with the resource being a download package for Internet Explorer. The user never needs to know that this resource is served by a page called *details.aspx*. The result would look like this:

```
http://microsoft.com/downloads/windows-internet-explorer-7-for-windows-xp-sp2
```

This is clearly an improvement but we are making an assumption that the description of the item is unique. Ideally, in the design of an application, we could make some human-readable information like the title or description unique to support the URL schema. If this were not possible, we could implement another technique to end up with something like the following URL:

```
http://microsoft.com/downloads/windows-internet-explorer-7-for-windows-xp-
➥    sp2/1987429874
```

In this final example, both a description of the download and a unique identifier are used. When the application comes to process this URL, the description *can* be ignored and the download looked up on the unique identifier. You might want to enforce agreement between the two segments for search engine optimization. Having multiple URLs pointing to the same logical resource yields poor results for search engines. Let's see how we can apply these ideas to create better URLs.

5.1.2 Taking back control of the URL with routing

For years, the server platform has dictated portions of the URL, such as the *.aspx* at the end. This problem has been around since the beginning of the dynamic web and affects almost all current web technologies, so you should not be surprised that many solutions to the problem have been developed. Although ASP.NET *does* offer options for URL rewriting,[1] many ASP.NET developers ignore them. URL rewriting is discussed again in chapter 10.

Many web technologies such as PHP and Perl, hosted on the Apache web server, solve this problem by using mod_rewrite.[2] Python and Ruby developers have taken to the MVC frameworks and both Django and Rails have their own sophisticated routing mechanisms.

A routing system in any MVC framework manages the decoupling of the URL from the application logic. It must manage this in both directions so that it can

- Map URLs to a controller/action and any additional parameters
- Construct URLs which match the URL schema from a controller, action, and additional parameters

This is more commonly referred to as *inbound* routing (figure 5.1) and *outbound* routing (figure 5.2). Inbound routing describes the URL invocation of a controller action; outbound routing describes the framework generating URLs for links and other elements on your site.

When the routing system performs both of these tasks, the URL schema can be truly independent of the application logic. As long as it's never bypassed when constructing links in a view, the URL schema should be trivial to change independent of the application logic. Now let's take a look at how to build a meaningful URL schema for our application.

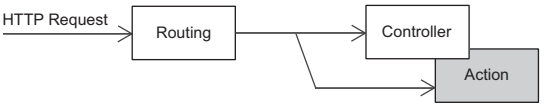

Figure 5.1 Inbound routing refers to taking an HTTP request (a URL) and mapping it to a controller and action.

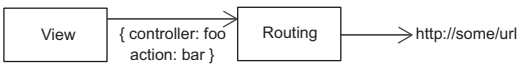

Figure 5.2 Outbound routing generates appropriate URLs from a given set of route data (usually controller and action).

[1] URL Rewriting in ASP.NET—http://msdn2.microsoft.com/en-us/library/ms972974.aspx
[2] Apache Module mod_rewrite—http://httpd.apache.org/docs/2.2/mod/mod_rewrite.html

5.2 Designing a URL schema

As a professional developer, you would not start coding a new project before mapping out what the application will do and how it will look. The same should apply for the URL schema of an application. Although it's hard to provide a definitive guide on designing URL schema (every website and application is different) we'll discuss general guidelines with an example or two thrown in along the way.

Here is a list of simple guidelines:

- Make simple, clean URLs.
- Make hackable URLs.
- Allow URL parameters to clash.
- Keep URLs short.
- Avoid exposing database IDs wherever possible.
- Consider adding unnecessary information.

These guidelines will not all apply to every application you create. You should run through a process similar to this before deciding on your final application URL schema.

5.2.1 Make simple, clean URLs

When designing a URL schema, the most important thing to remember is that you should step back from your application and consider it from the point of view of your end user. Ignore the technical architecture you will need to implement the URLs. Remember that by using routing, your URLs can be completely decoupled from your underlying implementation. The simpler and cleaner a permalink is, the more usable a site becomes.

> **Permalinks and deep linking**
>
> Over the past few years permalinks have gained popularity, and it's important to consider them when designing a URL schema. A permalink is simply an unchanging direct link to a resource within a website or application. For example, on a blog the URL to an individual post would usually be a permalink such as `http://example.com/blog/post-1/hello-world`.

Let's take the example of our events management sample application. In a Web Forms world we might have ended up with a URL something like this:

```
http://example.com/eventmanagement/events_by_month.aspx?year=2008&month=4
```

Using a routing system it's possible to create a cleaner URL like this:

```
http://example.com/events/2008/04
```

This gives us the advantage of having an unambiguous hierarchical format for the date in the URL, which raises an interesting point. What would happen if we omitted that `04` in the URL? What would you (as a user) expect? This is described as *hacking* the URL.

5.2.2 Make hackable URLs

When designing a URL schema, it's worth considering how a URL could be manipulated or "hacked" by the end user in order to change the data displayed. In the following example URL, it might reasonably be assumed that removing the parameter `04` from the URL might present all events occurring in 2008.

```
http://example.com/events/2008
```

By the same logic this could be expanded into the more comprehensive list of routes shown in table 5.1.

Table 5.1 Partial URL schema for the events management application

URL	Description
http://example.com/events	Displays all events
http://example.com/events/<year>	Displays all events in a specific year
http://example.com/events/<year>/<month>	Displays all events in a specific month
http://example.com/events/<year>/<month>/<date>	Displays all events on a specific single day

Being this flexible with your URL schema is great but it can lead to having an enormous number of potential URLs in your application. When you build your application views you should always give appropriate navigation; remember it may not be necessary to include a link to every possible URL combination on every page. It's all right for some things to be a happy surprise when a user tries to hack a URL and for it to work!

> **Slash or dash ?**
>
> It's a general convention that if a slash is used to separate parameters, the URL should be valid if parameters are omitted. If the URL `/events/2008/04/01/` is presented to users, they could reasonably assume that removing the last "day" parameter could increase the scope of the data shown by the URL. If this is not what is desired in your URL schema, consider using dashes instead of slashes as `/events/2008-04-01/` would not suggest the same hackability.

The ability to hack URLs gives power back to the users. With dates this is very easy to express, but what about linking to named resources?

5.2.3 Allow URL parameters to clash

Let's expand the routes and allow events to be listed by category. The most usable URL from the user's point of view would probably be something like this:

```
http://example.com/events/meeting
```

But now we have a problem! We already have a route that matches `/events/<something>` used to list the events on a particular year, month, or day and how are we now going to try to use `/events/<something>` to match a category as well? Our second route segment can now mean something entirely different; it *clashes* with the existing route. If the routing system is given this URL, should it treat that parameter as a category or a date? Luckily, the routing system in ASP.NET MVC allows us to apply conditions. The syntax for this can be seen in section 5.3.3 but for now it's sufficient to say that we can use regular expressions to make sure that routes only match certain patterns for a parameter. This means that we could have a single route that allows a request like `/events/2009-01-01` to be passed to an action that shows events by date and a request like `/events/asp-net-mvc-in-action` to be passed to an action that shows events by category. These URLs should "clash" with each other but they don't because we have made them distinct based on what characters will be contained in the URL.

This starts to restrict our model design, however. It will now be necessary to constrain event categories so that category names made entirely of numbers are not allowed. You'll have to decide if in your application this is a reasonable concession to make for such a clean URL schema.

The next principle we'll learn about is URL size. For URLs, size matters, and smaller is better.

5.2.4 Keep URLs short

Permalinks are passed around millions of times every day through email, instant messenger, micromessaging services such as SMS and Twitter, and even in conversation. Obviously for a URL to be spoken (and subsequently remembered!), it must be simple, short, and clean. Even when transmitting a permalink electronically this is important, as many URLs are broken due to line breaks in emails.

Short URLs are nice; however you shouldn't sacrifice readability for the sake of brevity. Remember that when a link to your application is shared, it's probably going to have only the limited context provided by whoever is sharing it. By having a clear, meaningful URL that is still succinct you can provide additional context that may be the difference between the link being ignored or clicked.

The next guideline is both the most useful in terms of maintaining clarity, and the most violated, thanks to the default routes in the ASP.NET MVC Framework.

5.2.5 Avoid exposing database IDs wherever possible

When designing the permalink to an individual event, the key requirement is that the URL should uniquely identify the event. We obviously already have a unique identifier for every object that comes out of a database in the form of a primary key. This is usually some sort of integer, autonumbered from 1, so it might seem obvious that the URL schema should include the database ID.

```
http://example.com/events/87
```

Unfortunately, the number `87` means nothing to anyone except the database administrator, and wherever possible you should avoid using database-generated IDs in URLs. This doesn't mean you cannot use integer values in a URL where relevant, but try to make them meaningful.

In the `Conference` model of Code Camp Server, there are two possible properties which are suitable for the permalink identifier that are not database generated: `Name` and `Key`. `Name` could be made to be unique without too much trouble but will probably include spaces, apostrophes, or other punctuation, so `Key` seems like a more logical choice as a short unique text string for an event.

```
http://example.com/events/houstonTechFest2008
```

Sometimes creating a meaningful identifier for a model adds benefits only for the URL and has no value apart from that. In cases like this, you should ask yourself if having a clean permalink is important enough to justify additional complexity not only on the technical implementation of the model, but also in the UI, as you will usually have to ask a user to supply a meaningful identifier for the resource.

This is a great technique, but what if you don't have a nice unique name for the resource? What if you need to allow duplicate names and the only unique identifier is the database ID? This next trick will show you how to utilize both a unique identifier *and* a textual description to create a URL that is both unique and readable.

5.2.6 Consider adding unnecessary information

If you must use a database ID in a URL, consider adding additional information which has no purpose other than to make the URL readable. Look at the URL for a specific session in our events application. The `Title` property is not necessarily going to be unique, and it's probably not practical to have people add a text identifier for a session. If we add the word *session* just for readability, the URL might look something like

```
http://example.com/houstonTechFest2008/session-87
```

This isn't good enough, though, as it gives no indication what the session is about; let's add another superfluous parameter to it. The addition has no purpose other than description. It will not be used at all while processing the controller action. The final URL could look like

```
http://example.com/houstonTechFest2008/session-87/an-introduction-to-mvc
```

Much more descriptive, and the `session-87` parameter is still there so we can look up the session by database ID. Of course we'd have to convert the session name to a more URL-friendly format, but this would be trivial.

The routing principles covered in this section will guide you through your choice of URLs in your application. Decide on a URL schema before going live on a site, as URLs are the entry point into your application. If you have links out there in the wild and you change your URLs, you risk breaking these links and losing referral traffic from other sites. You also lose any reputation for your URLs from the search engines.

Search engine optimization (SEO)

It's worth mentioning the value of a well-designed URL when it comes to optimizing your site for the search engines. It's widely accepted that placing relevant keywords in a URL has a direct effect on search engine ranking, so bear the following tips in mind when you are designing your URL schema.

1. Use descriptive, simple, commonly used words for your controllers and actions. Try to be as relevant as possible and use keywords which you would like to apply to the page you are creating.
2. Replace all spaces (which are encoded to an ugly *%20* in a URL) to dashes (-) when including text parameters in a route.
3. Strip out all nonessential punctuation and unnecessary text from string parameters.
4. Where possible, include additional, meaningful information in the URL. Additional information like titles and descriptions provide context and search terms to search engines that can improve the site's relevancy for search terms.

Now that you've learned what kind of routes you'll use, let's create some with ASP.NET MVC.

5.3 Implementing routes in ASP.NET MVC

When you first create a new ASP.NET MVC project, two default routes are created with the project template (shown in listing 5.2). They are defined in `Global.asax.cs`. These routes cover

- An ignore route to take certain URLs out of the ASP.NET MVC pipeline
- A generic dynamic route covering a standard /controller/action/id route

Listing 5.2 Default routes

```
public class MvcApplication : HttpApplication
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");     (1)

        routes.MapRoute(
            "Default",                                        (2)
            "{controller}/{action}/{id}",                    (3)

            new { controller = "Home", action = "Index", id = "" }     (4)
        );

    }

    protected void Application_Start()
    {
        RegisterRoutes(RouteTable.Routes);
    }
}
```

REST and RESTful architectures

A style of architecture called REST (or RESTful architecture) is a recent trend in web development. REST stands for *representational state transfer*. The name may not be very approachable, but the idea behind it absolutely is.

REST is based on the principle that every notable "thing" in an application should be an addressable *resource*. Resources can be accessed via a single, common URI, and a simple set of operations is available to those resources. This is where REST gets interesting. Using lesser-known HTTP verbs like PUT and DELETE in addition to the ubiquitous GET and POST, we can create an architecture where the URL points to the resource (the "thing" in question) and the HTTP verb can signify the method (what to do with the "thing"). For example, if we use the URI /speakers/5, with the verb GET, this would show the speaker (in HTML if it were viewed in a web browser). Other operations might be as shown in this chart:

URL	VERB	ACTION
/sessions	GET	List all sessions
/sessions	POST	Add a new session
/sessions/5	GET	Show session with id 5
/sessions/5	PUT	Update session with id 5
/sessions/5	DELETE	DELETE session with id 5
/sessions/5/comments	GET	List comments for session with id 5

REST isn't useful just as an architecture for rendering web pages. It's also a means of creating reusable services. These same URLs can provide data for an AJAX call or a completely separate application. In some ways, REST is a backlash against the more complicated SOAP-based web services.

If you are coming from Ruby on Rails and are smitten with its built-in REST support, you'll be disappointed to find that ASP.NET MVC has no built-in support for REST. However, due to the extensibility provided by the framework, it's not difficult to achieve a RESTful architecture. MvcContrib has an implementation called SimplyRestful that contains a usable REST implementation. Look it up if you are interested in REST.

In listing 5.2, the first operation is an `IgnoreRoute` ❶. We don't want Trace.axd, WebResource.axd, and other existing ASP.NET handlers routed through the MVC framework, so the route `{resource}.axd/{*pathInfo}` ensures any request coming in with an extension of .axd will not be served by ASP.NET MVC.

The second operation defines our first route. Routes are defined by calling `MapRoute` on a `RouteCollection`, which adds a `Route` object to the collection. So, what comprises a route? A route has a name ❷, a URL pattern ❸, default values ❹,

and constraints. The latter two are optional, but you will most likely use default values in your routes. The route in listing 5.2 is named *Default*, has a URL pattern of `{controller}/{action}/{id}`, and a default value dictionary that identifies the default controller and action. These default values are specified in an anonymous type, which is new in .NET 3.5.

If we pick apart this route, we can easily see its components: the first segment of the URL will be treated as the *controller*, the second segment as the *action*, and the third segment as the *id*. Notice how these values are surrounded in curly braces. When a URL comes in with the following format, what do you think the values will be for controller, action, and id?

http://example.com/users/edit/5

Figure 5.3 shows how the values are pulled out of the URL. Remember, this is only the default route template. You are free to change this for your own applications.

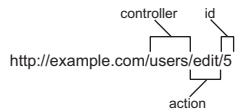

Figure 5.3 Decomposing a URL into route values using the default route of {controller}/{action}/{id}

Name	Value
`Controller`	`"users"`
`Action`	`"edit"`
`Id`	`"5"`

Table 5.2 The route values are set to the values extracted from the URL

The route values, shown in table 5.2, are all strings. The controller will be extracted out of this URL as *users*. The "`Controller`" part of the class name is implied by convention; thus the controller class created will be `UsersController`. As you can probably already tell, routes are not case sensitive. The action describes the name of the method to call on our controller. In ASP.NET MVC, an action is defined as a public method on a controller that returns an `ActionResult`. By convention the framework will attempt to find a method on the specified controller that matches the name supplied for action. If none is found it will also look for a method that has the `ActionNameAttribute` applied with the specified action. The remaining values defined in a route are pumped into the action method as parameters, or left in the `Request.Params` collection if no method parameters match.

Notice that the `id` is also a string; however if your action parameter is defined as an integer, a conversion will be done for you.

Listing 5.3 shows the action method that will be invoked as a result of the URL in figure 5.3.

Listing 5.3 An action method matching http://example.com/users/edit/5

```
public class UsersController : Controller
{
```

```
    public ActionResult Edit(int id)
    {
        return View();
    }
}
```

What happens if we omit the `id` or `action` from our URL? What will the URL http://example.com/users match? To understand this we have to look at the route *defaults*. In our basic route defined in listing 5.2, we can see that our defaults are defined as

```
new { controller = "Home", action = "Index", id = "" }
```

This allows the value of "Index" to be assumed when the value for action is omitted in a request that matches this route. You can assign a default value for any parameter in your route.

We can see that the default routes are designed to give a reasonable level of functionality for an average application but in almost any real world application you want to design and customize a new URL schema. In the next section we'll design a URL schema using custom static and dynamic routes.

5.3.1 URL schema for an online store

Now we are going to implement a route collection for a sample website. The site is a simple store stocking widgets for sale. Since the routes for Code Camp Server are a bit more complex, we'll first examine a slightly simpler case and continue our examples with Code Camp Server later in the chapter. Using the guidelines covered in this chapter we have designed a URL schema shown in table 5.3.

Table 5.3 The URL schema for sample widget store

	URL	Description
1	http://example.com/	Home page, redirects to the widget catalog list
2	http://example.com/privacy	Displays a static page containing site privacy policy
3	http://example.com/<widget code>	Shows a product detail page for the relevant <widget code>
4	http://example.com/<widget code>/buy	Adds the relevant widget to the shopping basket
5	http://example.com/basket	Shows the current user's shopping basket
6	http://example.com/checkout	Starts the checkout process for the current user

There is a new kind of URL in there that we have not yet discussed. The URL in route 4 is not designed to be seen by the user. It's linked via form posts. After the action has processed, it immediately redirects and the URL is never seen on the address bar. In cases like this it is still important for the URL to be consistent with the other routes defined in the application. How do we add a route?

5.3.2 Adding a custom static route

Finally it's time to start implementing the routes that we have designed. We'll tackle the static routes first as shown in table 5.4. Route 1 in our schema is handled by our route defaults, so we can leave that one exactly as is.

Table 5.4 Static routes

	URL	Description
1	http://example.com/	Home page, redirects to the widget catalog list
2	http://example.com/privacy	Static page containing site privacy policy

The first route that we'll implement is number 2 which is a purely static route linking http://example.com/privacy to the `privacy` action of the `Help` controller. Let's look at it in listing 5.4.

Listing 5.4 A static route

```
routes.MapRoute("privacy_policy", "privacy", new {controller = "Help", action
➥    = "Privacy"});
```

The route in listing 5.4 does nothing more than map a completely static URL to an action and controller. Effectively it maps http://example.com/privacy to the `Privacy` action of the `Help` controller.

NOTE *Route priorities* The order in which routes are added to the route table determines the order in which they will be searched when looking for a match. This means routes should be listed in source code from highest priority with the most specific conditions down to lowest priority or a catch-all route.

This is a common place for routing bugs to appear. Watch out for them!

Static routes are useful when there are a small number of URLs that deviate from the general rule. If a route contains information relevant to the data being displayed on the page, look at dynamic routes.

5.3.3 Adding a custom dynamic route

Four dynamic routes are added in this section (shown in table 5.5); we'll consider them two at a time.

Table 5.5 Dynamic routes

	URL	Description
1	http://example.com/<widget code>	Shows a product detail page for the relevant <widget code>
2	http://example.com/<widget code>/buy	Adds the relevant widget to the shopping basket

Table 5.5 Dynamic routes *(continued)*

	URL	Description
3	http://example.com/basket	Shows the current user's shopping basket
4	http://example.com/checkout	Starts the checkout process for the current user

Listing 5.5 implements routes 3 and 4. The route sits directly off the root of the domain, just as the privacy route did. It does not simply accept any and all values. Instead, it makes use of a route constraint. By convention, if we place a string value here it will be treated as a regular expression. We can create our own custom constraints by implementing `IRouteConstraint`, as we'll see later in this chapter. A request will only match a route if the URL pattern matches *and* all route constraints pass.

Listing 5.5 Implementation of routes 3 and 4

```
routes.MapRoute("widgets", "{widgetCode}/{action}",
                new {controller = "Catalog", action = "Show"},
                new {widgetCode = @"WDG[0-9]{4}"});          ❶
```

TIP If you are planning to host an ASP.NET MVC application on IIS6, mapping issues will cause the default routing rules not to work. For a quick fix, simply change the URLs used to have an extension such as `{controller}.mvc/{action}/{id}`. Chapter 10 presents more detail on this.

The `Constraints` parameter in `MapRoute` takes a dictionary in the form of an anonymous type which can contain a property for each named parameter in the route. In listing 5.5 we are ensuring that the request will only match if the `{widgetCode}` parameter starts with `WDG` followed by exactly 4 digits ❶. Listing 5.6 shows a controller that can handle a request that matches the route in listing 5.5.

Listing 5.6 The controller action handling the dynamic routes

```
public ActionResult Show(string widgetCode)
{
    var widget = GetWidget(widgetCode);

    if(widget == null)
    {
        Response.StatusCode = 404;
        return View("404");
    }
    else
    {
        return View(widget);
    }
}
```

Find widget by widget code

Return 404 if widget not found

Render view for widget

Listing 5.5 shows the action implementation in the controller for the route in listing 5.4. Although it's simplified from a real world application, it's straightforward until we get

to the case of the widget not being found. That's a problem. The widget does not exist and yet we have already assured the routing engine that we would take care of this request. As the widget is now being referred to by a direct resource locator, the HTTP specification says that if that resource does not exist, we should return *HTTP 404 not found*. Luckily, this is no problem and we can just change the status code in the `Response` and render the same 404 view that we have created for the catch-all route. (We'll cover catch-all routes later in this chapter.)

NOTE You may have noticed in the previous example that we appear to have directly manipulated the `HttpResponse`, but this is not the case. The `Controller` base class provides us with a shortcut property to an instance of `HttpResponseBase`. This instance acts as a façade to the actual `HttpResponse`, but allows you to easily use a mock if necessary to maintain testability. For an even cleaner testing experience, consider using a custom `ActionResult`.

TIP It's good practice to make constants for regular expressions used in routes as they are often used to create several routes.

Finally, we can add routes 5 and 6 from the schema. These routes are almost static routes but they have been implemented with a parameter and a route constraint to keep the total number of routes low. There are two main reasons for this. First, each request must scan the route table to do the matching, so performance can be a concern for large sets of routes. Second, the more routes you have, the higher the risk of route priority bugs appearing. Having few route rules is easier to maintain. The regular expression used for validation in listing 5.7 is simply to stop unknown actions from being passed to the controller.

Listing 5.7 Shopping basket and checkout rules

```
routes.MapRoute("catalog", "{action}",
                new{controller="Catalog"},
                new{action=@"basket|checkout"});
```

We've now added static and dynamic routes to serve up content for various URLs in our site. What happens if a request comes in and doesn't match any requests? In this event, an exception is thrown, which is hardly what you'd want in a real application. For exceptions, we can use catch-all routes.

5.3.4 Catch-all routes

The final route we'll add to the sample application is a catch-all route to match any URL not yet matched by another rule. The purpose of this route is to display our HTTP 404 error message. Global catch-all routes, like the one in listing 5.8, will catch anything, and as such should be the *last* routes defined.

NOTE The standard ASP.NET custom errors section is still useful. For example if a URL matches your standard `{controller}/{action}` route, but the controller doesn't exist, the framework will render the 404 page registered in that section. If a URL comes in and doesn't match any route, we'll get an exception stating, "The incoming request does not match any route." Catch-all routes can help give you even more control in these situations.

Listing 5.8 The catch-all route

```
routes.MapRoute("catch-all", "{*catchall}", new {controller = "Error",
    action = "NotFound"});
```

The value `"catchall"` gives a name to the information that the catch-all route picked up. You can retrieve this value by providing an action parameter with the same name.

The action code for the 404 error can be seen in listing 5.9.

Listing 5.9 The controller action for the HTTP 404 custom error

```
public class ErrorController : Controller
{
    public ActionResult Notfound()
    {
        Response.StatusCode = 404;
        return View("404");
    }
}
```

Catch-all routes can be used for other scenarios as well. If you wanted to match a certain string first, and then have everything else past the URL captured, you add the catch-all parameter to the end of the route definition. We saw this earlier: `routes.IgnoreRoute("{resource}.axd/{*pathInfo}")` will capture anything after the first segment. Another interesting use for a catch-all route is for dynamic hierarchies, such as product categories. When you reach the limits of the routing system, create a catch-all route and do it yourself.

The example in listing 5.8 is a true catch-all route and will literally match any URL that has not been caught by the higher priority rules. It's valid to have other catch-all parameters used in regular routes such as `/events/{*info}` which would catch every URL starting with /events/. Be cautious using these catch-all parameters as they will include *any* other text on the URL, including slashes and period characters. It's a good idea to use a regular expression parameter wherever possible so you remain in control of the data being passed into your controller action rather than just grabbing everything.

At this point, the default `{controller}/{action}/{id}` route can be removed as we have completely customized the routes to match our URL schema. You might choose to keep it around to serve as a default way to access your other controllers.

We have now customized the URL schema for our website. We have done this with complete control over our URLs, and without modifying where we keep our controllers and actions. This means that any ASP.NET MVC developer can come and look at

> **Internet Explorer's "friendly" HTTP error messages**
>
> If you are using Internet Explorer to develop and browse your application, be careful that you are not seeing Internet Explorer's "friendly" error messages when developing these custom 404 errors, as IE will replace your custom page with its own. To avoid this, go into Tools > Internet Options and untick "Show friendly HTTP error messages" under browsing options on the Advanced tab. Your custom 404 page should appear. Don't forget, though, that users of your application using Internet Explorer may not see your custom error pages.

our application and know exactly where everything is. This is a powerful concept. Next, we'll discover how to use the routing system from *within* our application.

5.4 Using the routing system to generate URLs

Nobody likes broken links. And since it's so easy to change the URL routes for your entire site, what happens if you directly use those URLs from within your application (for example, linking from one page to another)? If you changed one of your routes, these URLs could be broken. Of course the decision to change URLs does not come lightly; it's generally believed that you can harm your reputation in the eyes of major search engines if your site contains broken links. Assuming that you may have no choice but to change your routes, you'll need a better way to deal with URLs in your applications.

Instead, whenever we need a URL in our site, we'll ask the framework to give it to us, rather than hard-coding it. We'll need to specify a combination of controller, action, and parameters. The `ActionLink` method does the rest. It's a method on the `HtmlHelper` class included with the MVC framework which generates a full HTML `<a>` element with the correct URL inserted to match a route specified from the object parameters passed in.

```
<%= Html.ActionLink("WDG0001", "show", "catalog", new { widgetCode =
    "WDG0001" }, null) %>
```

This example generates a link to the `show` action on the `catalog` controller with an extra parameter specified for `widgetCode`. The output from this is shown next.

```
<a href="/WDG0001">WDG0001</a>
```

Similarly, if you use the `HtmlHelper` class' `BeginForm` method to build your form tags, it will generate your URL for you. As you saw in the last section, the controller and action may not be the only parameters that are involved in defining a route. Sometimes additional parameters are needed to match a route.

Occasionally it's useful to be able to pass parameters to an action that has not been specified as part of the route.

```
<%= Html.ActionLink("WDG0002 (French)", "show", "catalog",
    new { widgetCode = "WDG0002", language = "fr" }, null) %>
```

This example shows that passing additional parameters is as simple as adding extra members to the object passed to `ActionLink`. The link generated by this code is shown

next. If the parameter matches something in the route, it will become part of the URL. Otherwise it will be appended to the query string, as you can see in this example:

```
<a href="/WDG0002?language=fr">WDG0002 (French)</a>
```

When using `ActionLink`, your route will be determined for you, based on the first matching route defined in the route collection. Most often this will be sufficient, but if you want to request a specific route, you can use `RouteLink`. `RouteLink` accepts a parameter to identify the route requested, like this:

```
<%= Html.RouteLink("WDG003", "special-widget-route",
    new { widgetCode = "WDG003" }, null) %>
```

This will look for a route with the name *special-widget-route*. Most often you will not need to use this technique unless the URL generated by routing is not the desired one. Try to solve the issue by altering route ordering or with route constraints. Use `RouteLink` as a last resort.

Sometimes you need to obtain a URL, but not for the purposes of a link or form. This often happens when you are writing AJAX code, and the request URL needs to be set. The `UrlHelper` class can generate URLs directly, and in fact the `UrlHelper` is used by the `ActionLink` methods and others. Here is an example:

```
<%= Url.Action("show", "catalog",
    new { widgetCode="WDG0002", language="fr" }) %>
```

This will return the same URL as above, but without any surrounding tags.

5.5 Creating routes for Code Camp Server

Now that we are armed with the techniques for building routes for an application, let's apply this to Code Camp Server. Table 5.6 shows the desired URLs for the application. It's important to list all of the desired entry points to your website. We'll use this as a basis for route testing later.

Table 5.6 Desired URLs for Code Camp Server

Example	URL	Purpose
1	/	Redirect to current conference
2	/<conferenceKey>	See the details for the conference specified
3	/<conferenceKey>/edit	Edit the conference (admin only)
4	/<conferenceKey>/speakers	See the list of speakers for the conference
5	/<conferenceKey>/speakers/<id>/<personKey>	See the details of a speaker
6	/<conferenceKey>/sessions	See the list of sessions
7	/<conferenceKey>/sessions/new	Create a new session (admin only)

Table 5.6 Desired URLs for Code Camp Server *(continued)*

Example	URL	Purpose
8	/<conferenceKey>/sessions/<id>/<sessionKey>	See the details of a session
9	/<conferenceKey>/schedule	See the schedule of the conference
10	/<conferenceKey>/attendees	See who's coming
11	/<conferenceKey>/attendees/new	Create a new attendee
12	/<conferenceKey>/attendees/<id>/<personKey>	See the details of an attendee
13	/login	Log in to the site
14	/conference/list	List the conferences on the site (admin only)
15	/conference/new	Create a new conference (admin only)

This is a pretty exhaustive list of the URLs that we would like to have for our conference application. You can see that some of these do not follow the pattern that we are given by default (`{controller}/{action}/{id}`), so we have to customize.

Listing 5.10 defines the route that will handle requests at the root of our site. As you can see, we rely on the defaults to determine what controller and action to route to.

Listing 5.10 Default values help define which controller/action is invoked for site root

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute("root", "",
        new {controller = "Conference", action = "Current"});
}
```

In listing 5.11 we want to match a route where the first (and only) segment is the word *login*. Because this is a static route, we should define it before other dynamic routes. This way it is matched before a different, dynamic route matches it.

Listing 5.11 Defining a static route before other dynamic routes

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

    routes.MapRoute("root", "",
        new {controller = "Conference", action = "Current"});

    routes.MapRoute("login", "login",
        new { controller = "Account", action = "Login" });
}
```

The second and third URL in our list are intended for an action on the conference controller. We do not explicitly set the controller name in this URL, so we'll have to set a default value for it. Listing 5.12 shows this.

Listing 5.12 Routes for satisfying examples 3 and 4

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

    routes.MapRoute("root", "",
        new {controller = "Conference", action = "Current"});

    routes.MapRoute("login", "login",
        new { controller = "Account", action = "Login" });

    routes.MapRoute("conference", "{conferenceKey}/{action}",
        new {controller = "Conference", action = "Index"});
}
```

Looks as if this will work for examples 3 and 4, but it will incorrectly match the following routes. We'll treat rules 5 through 12 as static routes, which means they should be defined above the route we just added. Listing 5.13 shows these new routes.

Listing 5.13 Static routes for sessions, speakers, attendees, and more

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute("root", "",
                    new {controller = "Conference", action = "Current"});
    routes.MapRoute("login", "login",
                    new { controller = "Account", action = "Login" });
    routes.MapRoute("sessions", "{conferenceKey}/sessions/{action}",
                    new {controller = "Sessions", action = "index"});
    routes.MapRoute("attendees", "{conferenceKey}/attendees/{action}",
                    new { controller = "Attendees", action = "index" });
    routes.MapRoute("speakers", "{conferenceKey}/speakers/{action}",
                    new {controller = "Speakers", action = "Index"});

    routes.MapRoute("schedule", "{conferenceKey}/schedule/{action}",
                    new {controller = "Schedule", action = "Index"});
    routes.MapRoute("conference", "{conferenceKey}/{action}",
                    new {controller = "Conference", action = "Index"});
}
```

The new routes (bolded) define the general routes for sessions, attendees, speakers, and the schedule. These are similar enough that you might already be looking to combine them with a single, dynamic route. We'll visit that later in the chapter.

These routes also will not match the SEO-friendly routes we wanted, namely examples 5, 8, and 12. Listing 5.14 contains the definitions for these. Notice how the listing uses constraints on the `{id}` segment so that it will not incorrectly match as an action on another route. Because these routes are more specific, we'll need to add them before the more general (but very similar) routes.

Listing 5.14 Providing SEO-friendly routes that take extra information

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

```
    routes.MapRoute("root", "",
        new {controller = "Conference", action = "Current"});

    routes.MapRoute("login", "login",
        new { Controller = "Account", Action = "Login" });

    routes.MapRoute("single_session",
                   "{conferenceKey}/sessions/{id}/{sessionKey}",
                    new {controller = "Sessions", action = "show"},
                    new {id = @"\d+"});

    routes.MapRoute("sessions", "{conferenceKey}/sessions/{action}",
                    new {controller = "Sessions", action = "index"});

    routes.MapRoute("single_attendee",
                    "{conferenceKey}/attendees/{id}/{personKey}",
                    new { controller = "Attendees", action = "show" });

    routes.MapRoute("attendees", "{conferenceKey}/attendees/{action}",
                    new { controller = "Attendees", action = "index" });

    routes.MapRoute(null, "{conferenceKey}/speakers/{id}/{personKey}",
                    new {controller = "Speakers", action = "Show"},
                    new {id = @"\d+"});

    routes.MapRoute("speakers", "{conferenceKey}/speakers/{action}",
                    new {controller = "Speakers", action = "Index"});

    routes.MapRoute("schedule", "{conferenceKey}/schedule/{action}",
                    new {controller = "Schedule", action = "Index"});

    routes.MapRoute("conference", "{conferenceKey}/{action}",
                    new {controller = "Conference", action = "Index"});
}
```

The bolded routes now allow our SEO-friendly URLs for sessions, speakers, and attendees. This allows nice URLs such as http://example.com/sessions/129/introduction-to-asp-net-mvc. The constraint is a simple regular expression that matches one or more integers.

We have now addressed each URL in our table of desired URLs (table 5.6). Now it's time to write unit tests to ensure that these routes work as intended. This will also ensure that future routes will not break our URL structure.

5.6 Testing route behavior

When compared with the rest of the ASP.NET MVC Framework, testing routes is not easy or intuitive. Although ASP.NET MVC has advanced the functions interfaces and abstract base classes, many elements still must be mocked out before route testing is possible. Luckily, MvcContrib has a nice, fluent route testing API, which we can use to make testing these routes easier. But before we look at that, listing 5.15 demonstrates how you would test a route with NUnit and Rhino Mocks.

Listing 5.15 Testing routes can be a pain.

```
using System.Web;
using System.Web.Routing;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
```

```
using Rhino.Mocks;

namespace CodeCampServerRoutes.Tests
{
    [TestFixture]
    public class NaiveRouteTester
    {
        [Test]
        public void root_matches_conference_controller_and_current_action()
        {
            const string url = "~/";
            var request = MockRepository.GenerateStub<HttpRequestBase>();
            request.Stub(x => x.AppRelativeCurrentExecutionFilePath)
               .Return(url).Repeat.Any();
            request.Stub(x => x.PathInfo)
               .Return(string.Empty).Repeat.Any();

            var context = MockRepository.GenerateStub<HttpContextBase>();
            context.Stub(x => x.Request).Return(request).Repeat.Any();

            RouteTable.Routes.Clear();
            MvcApplication.RegisterRoutes(RouteTable.Routes);
            var routeData = RouteTable.Routes.GetRouteData(context);

            Assert.That(routeData.Values["controller"],
               Is.EqualTo("Conference"));
            Assert.That(routeData.Values["action"], Is.EqualTo("Current"));
        }
    }

}
```

If all of our route tests looked like that, nobody would even bother. Those specific stubs on `HttpContextBase` and `HttpRequestBase` were not lucky guesses. It took a peek inside of Reflector to find out exactly what to mock. This is not how a testable framework should behave! Luckily, we do not have to deal with this if we are smart. MvcContrib's fluent route testing API makes this a lot easier. Listing 5.16 is the same test, using MvcContrib:

Listing 5.16 Much better route testing with MvcContrib's TestHelper project

```
using System.Web.Routing;
using CodeCampServerRoutes.Controllers;
using MvcContrib.TestHelper;
using NUnit.Framework;

namespace CodeCampServerRoutes.Tests
{
    [TestFixture]
    public class FluentRouteTester
    {
        [Test]
        public void root_matches_conference_controller_and_current_action()
        {
            MvcApplication.RegisterRoutes(RouteTable.Routes);
            "~/".ShouldMapTo<ConferenceController>(x => x.Current());          (1)
```

```
        }
    }
}
```

This is all done with the magic and power of extension methods and lambda expressions.

You can't get away so easily! What kind of magic are you talking about?

Inside of MvcContrib there is an extension method on the `string` class that builds up a `RouteData` instance based on the parameters in the URL. The `RouteData` class has an extension method on it to assert that the route values match a controller and action ❶. You can see from the example that the controller comes from the generic type argument to the `ShouldMapTo<TController>()` method. The action is then specified with a lambda expression. The expression is parsed to pull out the method call (the action) and any arguments passed to it.

The arguments are matched with the route values. See the code for yourself here: http://code.google.com/p/mvccontrib/source/browse/trunk/src/MvcContrib.TestHelper/MvcContrib.TestHelper/Extensions/RouteTestingExtensions.cs.

Now it's time to apply this to our Code Camp Server routing rules and make sure that we have covered the desired cases. We do that in listing 5.17.

Listing 5.17 Testing Code Camp Server routes

```
using System.Web.Routing;
using CodeCampServerRoutes.Controllers;
using MvcContrib.TestHelper;
using NUnit.Framework;

namespace CodeCampServerRoutes.Tests
{
    [TestFixture]
    public class RouteTester
    {
        [SetUp]
        public void SetUp()
        {
            RouteTable.Routes.Clear();
            MvcApplication.RegisterRoutes(RouteTable.Routes);
        }

        [Test]
        public void incoming_routes()
        {
            "~/".ShouldMapTo<ConferenceController>(x => x.Current());

            "~/boiseCodeCamp".ShouldMapTo<ConferenceController>(
                x => x.Index("boiseCodeCamp"));

            "~/boiseCodeCamp/edit".ShouldMapTo<ConferenceController>(
                x => x.Edit("boiseCodeCamp"));

            "~/portlandTechFest/speakers".ShouldMapTo<SpeakersController>(
                x => x.Index("portlandTechFest"));

            "~/portlandTechFest/speakers/12/barney-rubble"
                .ShouldMapTo<SpeakersController>(
```

```
                x => x.Show("portlandTechFest", 12));

            "~/michigandayofdotnet/sessions"
                .ShouldMapTo<SessionsController>(
                x => x.Index("michigandayofdotnet"));

            "~/michigandayofdotnet/sessions/82/learning-nunit"
                .ShouldMapTo<SessionsController>(
                x => x.Show("michigandayofdotnet", 82));

            "~/houstonTechFest/schedule".ShouldMapTo<ScheduleController>(
                x => x.Index("houstonTechFest"));

            "~/austinCodeCamp/attendees".ShouldMapTo<AttendeesController>(
                x => x.Index("austinCodeCamp"));

            "~/austinCodeCamp/attendees/new"
                .ShouldMapTo<AttendeesController>(
                x => x.New("austinCodeCamp"));

            "~/austinCodeCamp/attendees/123/bob-johnson"
                .ShouldMapTo<AttendeesController>(
                x => x.Show("austinCodeCamp", 123));

            "~/login".ShouldMapTo<AccountController>(x => x.Login());

            "~/conference/new".ShouldMapTo<ConferenceController>(
                x => x.New());
        }
    }
}
```

NOTE You should probably separate out your tests into logical test methods. I have combined them for the sake of brevity in the example.

After running this example, we see that all of our routes are working properly (the output may look slightly different depending on your testing framework and runner):

```
------ Test started: Assembly: CodeCampServerRoutes.Tests.dll ------

1 passed, 0 failed, 0 skipped, took 2.03 seconds.
```

Remember earlier when we noticed an opportunity to make those other routes more dynamic? This would avoid duplication in the route rules and hopefully make them easier to understand. Armed with tests to back us up, we can now attempt to combine these routes.

We can probably match these all with a single route along the lines of `{conferenceKey}/{controller}/{action}`; however this route would prevent the preceding one (`{conferenceKey}/{action}`) from matching the routes on `ConferenceController`. It's clear that we'll have to leverage route constraints to get the URLs we want.

If we add the `{conferenceKey}/{action}` route first, then all we need to do is make it not match when the second segment—`{action}`—is equal to one of the other controller names. This can be accomplished fairly easily with regular expressions. Listing 5.18 shows the modified route.

Listing 5.18 Constraining the route so it will not be so greedy

```
routes.MapRoute("conference", "{conferenceKey}/{action}",
       new { controller = "Conference", action = "Index" },
       new { conferenceKey = "(?!conference|account).*",
             action = "(?!speakers|schedule|sessions|sttendees).*" });
```

Don't match conference or account

Don't match other controllers

This route will no longer match any route that has one of the values specified in the `action` constraint. This will allow us to match the request in a later route, which we have defined in listing 5.19.

Listing 5.19 Matching our other controllers with a single route

```
routes.MapRoute("other_controllers",
    "{conferenceKey}/{controller}/{action}",
        new {action="index"},
        new { conferenceKey = "(?!conference|account).*" });
```

This route must also contain the `conferenceKey` constraint, so that the final route (which is the default route provided in ASP.NET MVC applications) will match URLs that have *Conference* or *Account* in the first segment.

Now that we're done with the change, we'll run the tests again to make sure we didn't break anything.

```
------ Test started: Assembly: CodeCampServerRoutes.Tests.dll ------

1 passed, 0 failed, 0 skipped, took 1.97 seconds.
```

Excellent! Only by having unit tests can we be this confident this fast that we didn't break anything.

There is an important facet of route testing that we have paid little attention to so far: *outbound routing*. As defined earlier, outbound routing refers to the URLs that are generated by the Framework, given a set of route values. Look to projects like MvcContrib to eventually provide helpers for this type of route testing in the future. At the time of writing, no examples of outbound route testing were available.

Now that you've seen two complete examples of realistic routing schemas, you are prepared to start creating routes for your own applications. You have also seen some helpful unit testing extensions to make unit testing inbound routes *much* easier. We haven't yet mentioned that all of this routing goodness is available to Web Forms projects as well!

5.7 Using routing with existing ASP.NET projects

The URL problems discussed at the start of this chapter (URLs tied directly to files on disk, no ability to embed dynamic content in the URL itself, and so on) can affect all websites/applications and although you may not be in a position to adopt a full MVC pattern for an application, you should still care about your application's URL usability. `System.Web.Routing` is a separate assembly released as part of .NET 3.5 SP1, and as you might guess, it's available for use in Web Forms as well.

Luckily, by importing the `UrlRoutingModule` from the `System.Web.Routing` assembly, we can use the routing mechanism from the MVC framework in existing ASP.NET Web Forms applications. To get started, open an existing ASP.NET Web Forms project and add the lines from listing 5.20 (and 5.21 for IIS 7) in to the `assemblies` and `httpModules` sections in your web.config.

Listing 5.20 Configuration for the `UrlRoutingModule`

```
<assemblies>
   <add assembly="System.Web.Routing, Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=31BF3856AD364E35" />
   ...
</assemblies>

...
<httpModules>
   <add name="UrlRoutingModule" type="System.Web.Routing.UrlRoutingModule,
     System.Web.Routing,   Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=31BF3856AD364E35"/>
   ...
</httpModules>

...
```

For IIS6 or IIS7 classic mode

Listing 5.21 Configuration for IIS 7 integrated mode

```
<system.webServer>
  <handlers>
    <add name="UrlRoutingHandler" preCondition="integratedMode" verb="*"
        path="UrlRouting.axd"
        type="System.Web.HttpForbiddenHandler, System.Web, Version=2.0.0.0,
            Culture=neutral,
            PublicKeyToken=b03f5f7f11d50a3a" />
    ...
  </handlers>
  ...
  <modules>
    <remove name="UrlRoutingModule" />
    <add name="UrlRoutingModule" type="System.Web.Routing.UrlRoutingModule,
        System.Web.Routing,
        Version=3.5.0.0, Culture=neutral,
        PublicKeyToken=31BF3856AD364E35"/>
    ...
  </modules>
</system.webServer>
```

Next, we need to define a custom route handler that will—you guessed it—handle the route! You may have a custom route handler for each route, or you might choose to make it more dynamic. It's entirely up to you.

Defining the route is similar to methods we've seen earlier, except that there are no controllers and actions to specify. Instead you just specify a page. A sample route for Web Forms might look like this:

```
RouteTable.Routes.Add("ProductsRoute", new Route
    (
       "products/apparel",
       new CustomRouteHandler("~/Products/ProductsByCategory.aspx",
           "category=18")
    ));
```

The custom route handler simply needs to build the page. Here is a bare-bones handler that will work:

```
public class CustomRouteHandler : IRouteHandler
{
    public CustomRouteHandler(string virtualPath, string queryString)
    {
        this.VirtualPath = virtualPath;
        this.QueryString = queryString;
    }

    public string VirtualPath { get; private set; }
    public string QueryString { get; private set; }

    public IHttpHandler GetHttpHandler(RequestContext
          requestContext)
    {
        requestContext.HttpContext.RewritePath(
            String.Format("{0}?{1}", VirtualPath, QueryString));

        var page = BuildManager.CreateInstanceFromVirtualPath
             (VirtualPath, typeof(Page)) as IHttpHandler;
        return page;
    }
}
```

Now, requests for */products/apparel* will end up being served by the URL in the example.

NOTE When using `UrlRoutingModule` to add routing capabilities to your Web Forms application, you are essentially "directing traffic" around parts of the normal ASP.NET request processing pipeline. This means that the normal URL-based authorization features of ASP.NET will be circumvented, and even if users don't have access to a particular page, they can view it if the `CustomRouteHandler` does not implement authorization checking or the route is not listed in the authorization rules in the `web.config`. Although the complete implementation is outside the scope of this text, you can use the `UrlAuthorizationModule.CheckUrlAccessForPrincipal()` method to verify a user has access to a particular resource.

5.8 Summary

In this chapter we have seen how the routing module in the ASP.NET MVC Framework gives us virtually unlimited flexibility when designing routing schemas able to implement both static and dynamic routes. Best of all, the code needed to achieve this is relatively insignificant.

Designing a URL schema for an application is the most challenging thing we have covered in this chapter and there is never a definitive answer to what routes should be implemented. Although the code needed to generate routes and URLs from routes is simple, the process of designing that schema is not. Ultimately every application is different. Some will be perfectly happy with the default routes created by the project template, whereas others will have complex, custom route definition spanning multiple C# classes.

We saw that the order in which routes are defined determines the order they are searched when a request is received and that you must carefully consider the effects of adding new routes to the application. As more routes are defined, the risk of breaking existing URLs increases. Your insurance against this problem is route testing. Although route testing can be cumbersome, helpers like the fluent route testing API in MvcContrib can certainly help.

The most important thing to note from this chapter is that there should be no application written with the ASP.NET MVC Framework that is limited in its URL by the technical choices made by source code layout, and that can only be a good thing! Separation of the URL schema from the underlying code architecture gives ultimate flexibility and allows you to focus on what would make sense for the user on the URL rather than what the layout of your source code requires.

We'd like to offer a special note of thanks to Dave Verwer, who wrote the initial version of this chapter. In the next chapter, we'll see how to customize and extend the ASP.NET MVC Framework.

ASP.NET/WEB DEVELOPMENT

ASP.NET MVC IN ACTION

Jeffrey Palermo • Ben Scheirman • Jimmy Bogard
FOREWORD BY Phil Haack

Free eBook
SEE INSERT

ASP.NET MVC implements the Model-View-Controller pattern on the ASP.NET runtime. It works well with open source projects like NHibernate, Castle, StructureMap, AutoMapper, and MvcContrib.

ASP.NET MVC in Action is a guide to pragmatic MVC-based web development. After a thorough overview, it dives into issues of architecture and maintainability. The book assumes basic knowledge of ASP.NET (v. 3.5) and expands your expertise. Some of the topics covered:

- How to effectively perform unit and full-system tests.
- How to implement dependency injection using StructureMap or Windsor.
- How to work with the domain and presentation models.
- How to work with persistence layers like NHibernate.

The book's many examples are in C#.

Jeffrey Palermo is co-creator of MvcContrib. **Jimmy Bogard** and **Ben Scheirman** are consultants and .NET community leaders. All are Microsoft MVPs and members of ASPInsiders.

For online access to the authors and a free ebook for owners of this book, go to manning.com/ASP.NETMVCinAction

"Shows how to put all the features of ASP.NET MVC together to build a great application."
—From the Foreword by Phil Haack
Senior Program Manager
ASP.NET MVC Team, Microsoft

"This book put me in control of ASP.NET MVC."
—Mark Monster
Software Engineer, Rubicon

"Of all the offerings, this one got it right!"
—Andrew Siemer
Principal Architect, OTX Research

"Highly recommended for those switching from Web Forms to MVC."
—Frank Wang, Chief Software Architect, DigitalVelocity LLC

ISBN 13: 978-1-933988-62-7
ISBN 10: 1-933988-62-2
54499
9 781933 988627

MANNING $44.99 / Can $62.99 [INCLUDING eBOOK]