16

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

So far in this book we have stuck with the default routing configuration which comes with any new ASP.NET MVC project. We will cover the routing system in-depth and learn how to create custom routes for your 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, where we now see file extensions like .aspx and .php mapping to physical files in public URLs. 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 that 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. 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.

16.1 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 of "plants" in our URL 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 16.1.

Listing 16.1 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

16.1.1 What’s that curl command?

The curl command shown in listing 16.1 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 web developers 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 not 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.

16.1.2 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]](#footnote-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]](#footnote-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 16.1) and outbound routing (figure 16.2). 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.

I am a failure at footnotes. Author included footnotes in text. I moved them but cannot number them, nor can I get rid of that lone 3. Footnotes should be numbered 1 and 2 to match references in text.

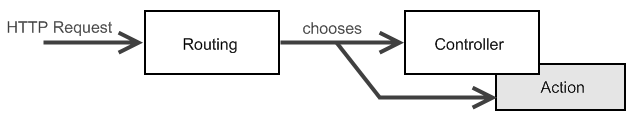


Figure 16.1 Inbound routing refers to taking an HTTP request (a URL) and mapping it to a controller and action.

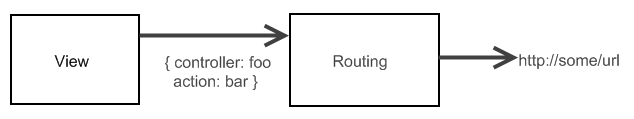


Figure 16.2 Outbound routing generates appropriate URLs from a given set of route data (usually controller and action).

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.

16.2 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 web site 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.

16.2.1 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 a non-ambiguous 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 the user expect? This is described as hacking the URL.

16.2.2 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 16.1.

Table 16.1 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?

16.2.3 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 16.3.3 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 about URL size. For URLs, size matters, and smaller is better.

16.2.4 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.

16.2.5 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.

16.2.6 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

This is 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.

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 sites relevancy for search terms.

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.

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:

Typesetter: the following should be a table within the sidebar…

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, as the complexity of SOAP often brought more problems than solutions.

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.

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

16.3 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 16.2). They are defined in Global.asax.cs. These routes cover an ignore route to take certain URLs out of the ASP.NET MVC pipeline and a generic dynamic route covering a standard /controller/action/id route.

Cueballs in code and text

Listing 16.2 Implementing 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);

}

}

In listing 16.2, the first operation is an IgnoreRoute (1). 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 (2), a URL pattern (3), default values (4), and constraints. The latter two are optional, but you will most likely use default values in your routes. The route in listing 16.2 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 3.16.

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 16.3 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.

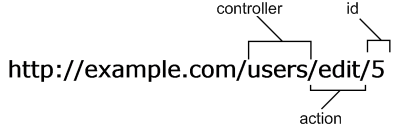


Figure 16.3 Decomposing a URL into route values using the default route of {controller}/{action}/{id}

Table 16.2 The route values are set to the values extracted from the URL

|  |  |
| --- | --- |
| Name | Value |
| Controller | "users" | |
| Action | "edit" | |
| Id | "5" | |

The route values, shown in table 16.2, 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 16.3 shows the action method that will be invoked as a result of the URL figure 16.3.

Listing 16.3 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 16.2, 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.

16.3.1 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 16.3.

Table 16.3 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 users 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?

16.3.2 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 16.4. Route 1 in our schema is handled by our route defaults, so we can leave that one exactly as is.

Table 16.4 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 16.4.

Listing 16.4 A static route

routes.MapRoute("privacy\_policy", "privacy", new {controller = "Help", action = "Privacy"});

The route in listing 16.4 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.

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.

16.3.3 Adding a custom dynamic route

Four dynamic routes are added in this section (shown in table 16.5); we’ll consider them two at a time.

Table 16.5 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 |
| 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 16.4 implements rules 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.

Cueball in code and text

Listing 16.4 Implementation of routes 3 and 4

routes.MapRoute("widgets", "{widgetCode}/{action}",

new {controller = "Catalog", action = "Show"},

new {widgetCode = @"WDG[0-9]{4}"}); #1

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 16.4 we are ensuring that the request will only match if the {widgetCode} parameter starts with WDG followed by exactly 4 digits (1). Listing 16.5 shows a controller that can handle a request that matches the route in listing 16.4.

Listing 16.5 The controller action handling the dynamic routes

public ActionResult Show(string widgetCode)

{

var widget = GetWidget(widgetCode); #A

if(widget == null)

{

Response.StatusCode = 404; #B

return View("404");

}

else

{

return View(widget); #C

}

}

#A Find widget by widget code

#B Return 404 if widget not found

#C Render view for widget

Listing 16.5 shows the action implementation in the controller for the route in listing 16.4. 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-up 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 16.6 is simply to stop unknown actions from being passed to the controller.

Listing 16.6 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. To handle this, we use catch-all routes.

16.3.4 Catch-all routes

The final route we’ll add to the sample application is a final 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 16.7, will catch anything, and as such should be the last route 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 16.7 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 16.8.

Listing 16.8 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 to be 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 16.7 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.

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.

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 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.

16.4 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.

16.5 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 16.9 demonstrates how you would test a route with NUnit and Rhino Mocks.

Listing 16.9 Testing routes can be a pain

using System.Web;

using System.Web.Routing;

using NUnit.Framework;

using NUnit.Framework.SyntaxHelpers;

using Rhino.Mocks;

namespace BadRoutingTestExample.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 testing. Those specific stubs on HttpContextBase and HttpRequestBase were not lucky guesses either. 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 16.10 is the same test, using MvcContrib:

Cueball in code and text

Listing 16.10 Cleaner route testing with MvcContrib’s TestHelper project

using System.Web.Routing;

using CodeCampServerRoutes.Controllers;

using MvcContrib.TestHelper;

using NUnit.Framework;

namespace BetterRouteTestExample.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 (1). 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://github.com/mvccontrib/MvcContrib/tree/master/src/MvcContrib.TestHelper/MvcContrib.TestHelper/>

Now it’s time to apply this to our Store Example routing rules and make sure that we have covered the desired cases. We do that in listing 16.11.

Listing 16.11 Testing Code Camp Server routes

using System.Web.Routing;

using StoreExample.Controllers;

using MvcContrib.TestHelper;

using NUnit.Framework;

namespace StoreExample.Tests

{

[TestFixture]

public class ComplexRouteTests

{

/\* Desired URL schema:

\* 1. example.com/ home page

\* 2. example.com/privacy static page w/ privacy policy

\* 3. example.com/widgets show a list of the widgets

\* 4. example.com/<widget code> Shows a product detail page

\* 5. example.com/<widget code>/buy Add to the shopping basket

\* 6. example.com/basket Shows the user's shopping basket

\* 7. example.com/checkout Starts the checkout process

\* 8. example.com/404 show a friendly 404 page

\*/

[TestFixtureSetUp]

public void FixtureSetup()

{

RouteTable.Routes.Clear();

MvcApplication.RegisterRoutes(RouteTable.Routes);

}

[Test]

public void root\_maps\_to\_home\_index()

{

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

}

[Test]

public void privacy\_should\_map\_to\_home\_privacy()

{

"~/privacy".ShouldMapTo<HomeController>(x => x.Privacy());

}

[Test]

public void widgets\_should\_map\_to\_catalog\_index()

{

"~/widgets".ShouldMapTo<CatalogController>(x => x.Index());

}

[Test]

public void widget\_code\_url()

{

"~/WDG-0002".ShouldMapTo<CatalogController>(x => x.Show("WDG-0002"));

}

[Test]

public void widget\_buy\_url()

{

"~/WDG-0002/buy".ShouldMapTo<CatalogController>(x => x.Buy("WDG-0002"));

}

[Test]

public void basket\_should\_map\_to\_catalog\_basket()

{

"~/basket".ShouldMapTo<CatalogController>(x => x.Basket());

}

[Test]

public void checkout\_should\_map\_to\_catalog\_checkout()

{

"~/checkout".ShouldMapTo<CatalogController>(x => x.CheckOut());

}

[Test]

public void \_404\_should\_map\_to\_error\_notfound()

{

"~/404".ShouldMapTo<ErrorController>(x => x.NotFound());

}

}

}

Note

We've separated each rule into a separate test. It might be tempting for all of these one-liners to just reside in a single test, but don't forget the value of understanding why a test is failing. If you make a mistake, only distinct tests will break, giving you much more information than a single broken test\_all\_routes() test.

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). Figure 16.4 shows the ReSharper test runner results, with all tests passing.

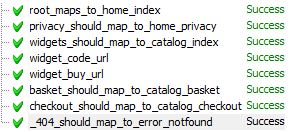


Figure 16.4 - Seeing the results of our route tests in the ReSharper Test Runner.

Armed with these tests, we are free to make some refactorings or clean up our route rules, confident that we aren't breaking existing URLs on our site. Could you imagine if all of a sudden, product links on Amazon.com were suddenly broken due to a typo in some route rule? Don't let that happen to you. It is much easier to write automated tests for your site than it is to do manual exploratory testing for each and every release.

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 a complete example 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!

16.6 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 web sites/applications and although you may not be in a position to be able 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 16.12 (and 16.13 for IIS 7) in to the assemblies and httpModules sections in your web.config.

Listing 16.12 Configuration for the UrlRoutingModule

<assemblies>

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

...

</assemblies>

...

<httpModules> #A

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

...

</httpModules>

...

#A For IIS6 or IIS7 classic mode

Listing 16.13 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 does not 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.

16.8 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 order in which routes are defined determines the order they are searched when a request is received and that you must give 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.

In the next chapter, we’ll see some advanced deployment concepts for your ASP.NET MVC applications.

1. URL Rewriting in ASP.NET - <http://msdn2.microsoft.com/en-us/library/ms972974.aspx> [↑](#footnote-ref-1)
2. Apache URL Rewriting module - <http://httpd.apache.org/docs/2.2/mod/mod_rewrite.html> [↑](#footnote-ref-2)