24

Route Debugger

This chapter covers:

* Customizing and extending the routing system
* Inspecting route matches at runtime in the browser

In Chapter 16, you learned all about routing. You probably already understand that routing is a complex and important topic. What happens when routing doesn't behave the way you expect? In this chapter, we will extend the routing system to provide diagnostic information about which routes are being matched for a given web request.

24.1 Extending the Routing System

The UrlRoutingModule is an implementation of IHttpModule and represents the entry point into the ASP.NET MVC Framework. This module examines each request, builds up the RouteData for the request, finds an appropriate IRouteHandler for the given route matched, and finally redirects the request to the IRouteHandler's IHttpHandler. Make sense?

In any ASP.NET MVC application, the default route looks like the one in listing 24.1. The MapRoute method is actually a simplified way of specifying routes.

Listing 24.1 A simple way of specifying routes

routes.MapRoute("default", "{controller}/{action}/{id}",

new { Controller="home", Action="index",

id=UrlParameter.Optional});

Most of the applications you will work with will use this style of adding routes. There is also a more verbose method, which allows you to customize the classes that are used as part of the route. Listing 24.2 shows the same route, only without using the MapRoute helper method.

Listing 24.2 A more detailed way of specifying routes

routes.Add(new Route("{controller}/{action}/{id}",

new RouteValueDictionary(new {

Controller = "home", Action = "index",

id = UrlParameter.Optional }),

new MvcRouteHandler() #1

));

That third argument in listing (#1) is telling the framework which IRouteHandler to use for this route. We are using the built-in MvcRouteHandler that ships with the framework. By default this class is used when we call the MapRoute method. We can change this to be a custom route handler and take control in interesting ways. An IRouteHandler is responsible for creating an appropriate IHttpHandler to handle the request given the details of the request. This is a good place to change the way routing works, or perhaps to gain control extremely early in the request pipeline. The MvcRouteHandler simply constructs an MvcHandler to handle a request, passing it a RequestContext, which contains the RouteData and an HttpContextBase.

A quick example will help illustrate the need for a custom route handler. When starting to define your routes, you'll sometimes run across errors. Let's assume you have defined the route shown in listing 24.3.

Listing 24.3 Adding another route

routes.MapRoute("CategoryRoute", "{category}/{action}",

new { Controller = "Products", Action="index" });

Here we’ve added a new custom route at the top position that will accept URLs like /apparel/index, use the ProductController, and call the Index() action on it, passing in the category as a parameter to the action as shown in listing 24.4.

Listing 24.4 A controller action that handles the new route

public class ProductsController : Controller

{

public ActionResult Index(string category)

{

return View();

}

}

This is a good example of a custom route that makes your URLs more readable.

Now let's assume that we have another controller called Home. HomeController has an Index action to show the start page as shown in listing 24.5.

Listing 24.5 A controller action to respond to the default route

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

}

We'd like the URL for the action in listing 24.4 to look like /home/index. If we try this URL, we'll get a 404 error as shown in figure 24.1. Why?

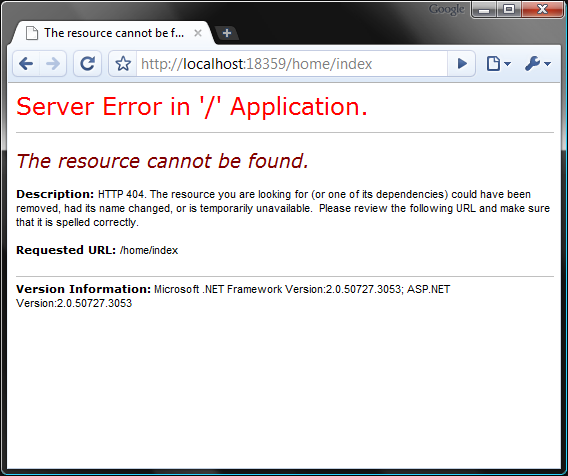


Figure 24.1 - This message doesn’t tell us much about what’s wrong. An action couldn’t be found on the controller, but which one?

The problem is not apparent from that error message. We certainly have a controller called HomeController, and it has an action method called Index. If you dig deep into the routes you can deduce that this URL was picked up by the first route, {category}/{action}, which was not what we intended. We should be able to quickly identify a routing mismatch, so that we can fix it speedily.

With many custom routes, it is easy for a URL to be caught by the wrong route. It would be nice if we had a diagnostics tool to display which routes are being matched (and used) for quickly catching these types of errors.

24.2 Inspecting Routes at Runtime

In order to allow us to see the route rules as they are matched at runtime, we can add a special query string parameter that we can tack on to the end of the URL. This will signify that instead of rendering the regular view, our custom Route Debugger should instead circumvent the request and provide a simple HTML view of the route information.

The current route information is stored in an object called RouteData, available to us in the IRouteHandler interface. The route handler is also first to get control of the request, so it is a great place to intercept and alter the behavior for any route as shown in listing 24.6.

Cueball in code and text

Listing 24.6 A custom route handler creates an associated IHttpHandler

public class CustomRouteHandler : IRouteHandler

{

public IHttpHandler GetHttpHandler(RequestContext requestContext)

{

if(HasQueryStringKey("routeInfo", #1

requestContext.HttpContext.Request))

{

OutputRouteDiagnostics(requestContext.RouteData,

requestContext.HttpContext);

}

var handler = new MvcHandler(requestContext);

return handler;

}

private bool HasQueryStringKey(string keyToTest,

HttpRequestBase request)

{

return Regex.IsMatch(request.Url.Query,

string.Format(@"^\?{0}$", keyToTest, RegexOptions.IgnoreCase));

}

}

#1 Check for query string parameter

A route handler’s normal responsibility is to construct and hand off the IHttpHandler that will handle this request. By default, this is MvcHandler. In our CustomRouteHandler we first check to see if the query string parameter is present (1) (we do this with a simple regular expression on the URL query section). The OutputRouteDiagnostics method is shown in listing 24.7.

Listing 24.7 Rendering route diagnostic information to the response stream

private void OutputRouteDiagnostics(RouteData routeData, HttpContextBase context)

{

var response = context.Response;

response.Write(

@"<style>body {font-family: Arial;}

table th {background-color: #359; color: #fff;}

</style>

<h1>Route Data:</h1>

<table border='1' cellspacing='0' cellpadding='3'>

<tr><th>Key</th><th>Value</th></tr>"); #A

foreach (var pair in routeData.Values)

{

response.Write(string.Format("<tr><td>{0}</td><td>{1}</td></tr>",

pair.Key, pair.Value));

}

response.Write(

@"</table>

<h1>Routes:</h1>

<table border='1' cellspacing='0' cellpadding='3'>

<tr><th></th><th>Route</th></tr>"); #B

bool foundRouteUsed = false;

foreach(Route r in RouteTable.Routes)

{

response.Write("<tr><td>");

bool matches = r.GetRouteData(context) != null;

string backgroundColor = matches ? "#bfb" : "#fbb"; #C

if(matches && !foundRouteUsed)

{

response.Write("&raquo;"); #D

foundRouteUsed = true;

}

response.Write(string.Format(

"</td><td style='font-family: Courier New;

background-color:{0}'>{1}</td></tr>",

backgroundColor, r.Url));

}

response.End();

}

#A Create an HTML table

#B Display the routes

#C Output green if matching, red otherwise

#D Display chevron (») next to route selected

This method outputs two tables, one for the current route data, and one for the routes in the system. Each route will return null for GetRouteData if the route doesn’t match the current request. The table is then colored to show which routes matched, and a little arrow indicates which route is in use for the current URL. The response is ended to prevent any further rendering.

To finalize this change, we have to alter the current routes to use our new handler as shown in listing 24.8.

Listing 24.8 Assigning routes to our custom route handler

private static RouteBase CreateRoute(string url, object defaults)

{

return new Route(url, new RouteValueDictionary(defaults),

new CustomRouteHandler());

}

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.Add(CreateRoute("{category}/{action}", new {

controller = "products",

action = "index"}));

routes.Add(CreateRoute("{controller}/{action}/{id}", new {

controller = "home",

action = "index",

id=UrlParameter.Optional}));

}

Here we are simply creating routes as we did before, only this time we are setting them up with our new CustomRouteHandler class. A helper method is used to avoid too much code duplication and to allow a similar experience to the MapRoute method we used previously.

The end result (shown in figure 24.2) is incredibly helpful. It shows us all of the routes that are defined, color coded by whether or not they match the current request. Let’s use the /home/index URL that resulted in a 404 in figure 24.1 but this time we’ll add ?routeInfo to the query string. We can see in the route data table that the value home was picked up as a conference key. The route table confirms that the conference key route was picked up first, since it matched.

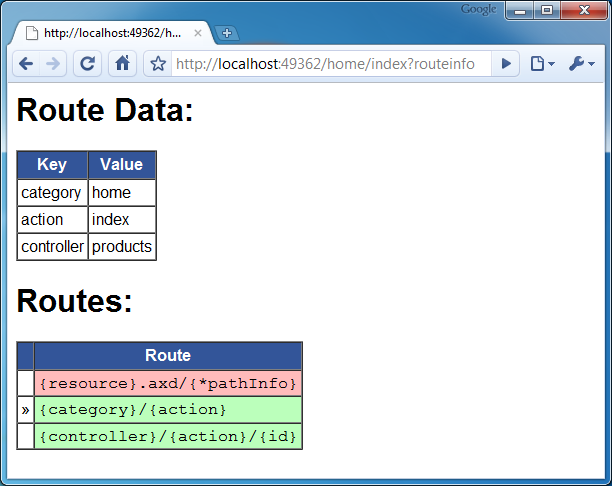


Figure 24.2 Appending the querystring parameter ?routeinfo to our URL gives us detailed information about the current request’s route. We can see now that the wrong route was chosen.

Now you can immediately tell that the current route used is not the one we intended. We can also tell whether or not other routes match this request by the color of the cell. If you’re reading the print version of this book this might not be apparent, but if you run the sample application you’ll see that rows 2 and 3 are green. We now quickly identify the issue as a routing problem and can fix it accordingly. In this case, if we add constraints to the first route such that {category} isn’t the same as one of our controllers, the problem is resolved.

Remember that order matters!

The first route matched is the one used.

Of course you wouldn’t want this information to be visible in a deployed application, so use it only to aid your development. You could also build a switch that changes the routes to the CustomRouteHandler if you’re in debug mode, which would be a more automated solution. Listing 24.9 shows a simple way of accomplishing this using preprocessor directives.

Listing 24.9 Switching the IRouteHandler implementation for debug mode

private static RouteBase CreateRoute(string url, object defaults)

{

IRouteHandler routeHandler = new MvcRouteHandler();

#if DEBUG

routeHandler = new CustomRouteHandler();

#endif

return new Route(url, new RouteValueDictionary(defaults), routeHandler);

}

In this example, we are simply modifying our helper method to change out the IRouteHandler implementation to the standard one if the code is built in Release mode.

NOTE

This example was inspired by Phil Haack’s route debugger that he posted on his blog when for an early preview of the ASP.NET MVC Framework. It is a great example of what you can do with the information provided to you by the routing system. You can see his original example of here:

<http://haacked.com/archive/2008/03/13/url-routing-debugger.aspx>

24.3 Summary

Routing is a complex topic, and a small mistake can mean that your entire site is inaccessible. Using this technique of extending via the IRouteHandler interface you learned how to customize the routing system and leverage it to create a nice route debugger. This tool is a great way to understand how your routes are being matched and also which one is being used for the current request. In the next chapter, you'll learn how to customize Visual Studio to take advantage of some advanced features of ASP.NET MVC.