[Url Routing in ASP.NET](http://blog.microsoft-j.net/2009/01/31/UrlRoutingInASPNET.aspx)

January 30, 2009 @ 06:16 PM

http://blog.microsoft-j.net/2009/01/31/UrlRoutingInASPNET.aspx

One of the new features in ASP.NET 3.5 SP1 is Url Routing … it’s the magic behind the pretty, clean url’s that you see with Dynamic Data and MVC. If you dig around a bit, almost all (if not all) of the material that’s out there focuses on using Routing with either MVC or Dynamic Data … I’ve found nada that actually talks about how it can be added to an existing ASP.NET WebForms application. In talks of .NET 3.5 SP1, Url Routing is even ignored some of the time … and if it’s not ignored, it’s barely mentioned in passing. And then there’s the documentation which is, IMHO, pretty lame and the “How To” articles on MSDN are only mildly better.

In spite of (or maybe because of) that, I found myself intrigued by the whole idea of Url Routing. Yes, I had seen it and tweaked it in MVC and Dynamic Data, but I knew that there had to be additional uses for it. So … I set about to build a demo for Houston Tech Fest that showed how to get started with Url Routing, adding it to an existing website that showed some data from Northwind. It’s not a pretty or even really functional app … that’s not the point … and has Url’s that are both plain vanilla and that require query strings. In addition, there was absolutely, positively no consistency between the pages or the query string parameters. I know *that* doesn’t happen in the real world! ;-)

There was one thing that I did do in the application that I don’t see done frequently; I added a static utility class that built and returned Url’s for the different pages in the app. Again, not something that I typically see, but it is definitely something that can make the maintenance of links and url’s easier and more elegant. Well, maybe not elegant but it sure beats “Find in project”. But then, url’s in the application never change, do they?

If you’re interested, you can find the PPT and the demos (as well as something resembling a script) on my [**SkyDrive**](http://blog.microsoft-j.net/ct.ashx?id=dc679fe3-5329-4857-bdb9-36e7b0b4fcf1&url=http%3a%2f%2fcid-7d9c5b1649900d36.skydrive.live.com%2fself.aspx%2fPublic%2fASPNET%2520Routing.zip). It’s the same place as the link that I posted the other day for my Houston TechFest presentations. However, I wanted to spend a little more time and explain what’s going on and how it all works.

**StaticRoutingHandler**

This is what does the work. All you have to do to create a routing handler is to implement the **IRouteHandler** interface. It’s pretty a simple interface – there’s only 1 method. This takes a url route and then transfers this to a static .aspx page that is configured for the route (1 route, 1 page). Since I needed to pass query strings to the pages, I also do some url rewriting in here. Technically, this is not necessary to pass information to the target page, but remember – I didn’t want to have to spend a lot of time making changes all over the app and changing the pages to no longer require query strings would be more work and get away from the point of the demo (i.e. making it easy). While you do create an instance of the page with Url Routing, the pages in this demo didn’t have any nice properties that encapsulated the query string parameters. No way to do that without url rewriting when the app is expecting to get query strings. It takes the data arguments specified in the route and turns them into query strings, using the name as the query string parameter. Here it is:

public IHttpHandler GetHttpHandler(RequestContext requestContext)

{

string finalPath = VirtualPath;

if (requestContext.RouteData.Values.Count > 0)

{

List<string> values = new List<string>();

//Add these to the virtual path as QS arguments

foreach (var item in requestContext.RouteData.Values)

{

values.Add(item.Key + "=" + item.Value);

}

finalPath += "?" + String.Join("&", values.ToArray());

}

//Rewrite the path to pass the query string values.

HttpContext.Current.RewritePath(finalPath);

var page = BuildManager.CreateInstanceFromVirtualPath(

VirtualPath, typeof(Page)) as IHttpHandler;

return page;

}

**Configuration**

This is where it could potentially get hairy. With MVC and Dynamic Data, it’s pretty easy to do it in the global.asax file since their paths and page names follow a clear and simple convention. Not so with the sample app. So each route/page combination needs to be registered separately because the page names have absolutely no consistency, not to mention the query string arguments. Ewwww … that’ll bloat your global.asax real quick. Since I didn’t like how that was going, I decided that I’d make it configuration-driven. This had the added benefit of allowing you to change the routes, arguments, etc. without redeploying code. I wrote a custom configuration section to handle this; this also makes the config read/write with the API which I thought might be a nice-to-have. So, the section looks like the following:

<StaticRoutes>

<Routes>

<add name="AXD" routeUrl="{resource}.axd/{\*pathInfo}"></add>

<add name="CategoryList" routeUrl ="CategoryList" virtualPath="~/CategoryList.aspx"/>

<add name="ProductList" routeUrl="Category/Products/{C}" virtualPath="~/ProductList.aspx">

<Restrictions>

<add name="C" value="\d"/>

</Restrictions>

</add>

<add name="ViewProduct" routeUrl="Product/{Id}" virtualPath="~/ViewProduct.aspx">

<Restrictions>

<add name="Id" value="\d"></add>

</Restrictions>

<Defaults>

<add name="Id" value="1"/>

</Defaults>

</add>

<add name="CustomerOrders" routeUrl="Customers/Orders/{Cu}" virtualPath="~/ListOrders.aspx">

</add>

<add name="CustomerList" routeUrl="Customers" virtualPath="~/Customers.aspx">

</add>

<add name="OrderDetails" routeUrl="Customers/Orders/{Id}" virtualPath="~/OrderDetails.aspx"/>

</Routes>

</StaticRoutes>

It’s got all of the information that we need to create our routes. Restrictions and Defaults are optional – not every route needs them. You’ll also notice that the “AXD” route doesn’t have any virtual path listed … when there is no virtualPath specified, the StaticRoutingHandler.Configure method will add a StopRoutingHandler rather than the StaticRoutingHandler. The StopRoutingHandler is the only handler (that I could find) that is in the API itself (MVC and Dynamic Data each have their own RoutingHandlers). It tells Routing to simply ignore the request and send it along it’s merry way as if there was no routing configured. The order of the routes in the config file does matter, but that has nothing to do with my code; when the ASP.NET routing framework looks for the handler that a particular url matches, it grabs the first that it finds on the list. So … that’s how you prioritize your routes. The query string parameters are surrounded with curly braces … so “{Cu}” in the CustomerOrders route above would get put in as a query string named “Cu” with the value that appears in that place in the url.

With the configuration, RegisterRoutes, rather than being a mess that looks more like my office desk than code, is clean, simple and sweet. We just need to call a static method on the StaticRoutingHandler class to read the configuration and add the routes.

public static void RegisterRoutes(RouteCollection routes)   
{   
    routes.Clear();   
  
    StaticRouting.StaticRoutingHandler.Configure(routes);   
  
}

The names (which are required) also allow us to build the Url using the Routing API, rather than having Url’s hardcoded in the application. It’s pretty simple and straightforward; below is one of the more “complex” examples as it builds the RouteValueDictionary with the value for the query string.

public static string GetProductsForCategory(int categoryId)

{

var values = new RouteValueDictionary();

values.Add("C", categoryId);

var path = RouteTable.Routes.GetVirtualPath(

null,

"ProductList",

values);

return path.VirtualPath;

}

I got a question when I did this presentation about wildcard parameters. I knew about them and how they worked, but somehow didn’t think to play with them in this sample app. First, you can do this (and I mentioned it in the presentation) by adding a wildcard parameter to the route. In our configuration, it would look like the following:

<add name="Wildcard" routeUrl ="Wildcard/{id}/{\* params}" virtualPath="~/Customers.aspx"></add>

It doesn’t have to have a set parameter (id in this case) in the routeUrl; I just put that there as a comparison. Everything else after that goes into a single item called (in this case) “params”. The “id” is exactly as we expect it to be. However, the wildcard doesn’t translate well into query string parameters. Yes, it is named “params”, but the value is *everything* in the path in the wildcard’s place, including the slashes in the path. So, with the url http://localhost/Wildcard/3/sasdf/asdf/as, the value of “params” is **sasdf/asdf/as**. Yes, the routing handler will very happily pass this along when it rewrites the url, but doesn’t really seem to make sense in this context. In this case, I’d say that you put each possible query string/route combination in as a different route to make sure that the page gets the parameters it expects the way that it expects them. I might, some time in the future, put some more thought into this and come up with something else, but for right now, I’m happy punting on it for now and just adding a route for each combination.