John Atten

ASP.NET

Customizing Routes in ASP.NET MVC

JOHN ATTEN AUGUST 21, 2013 6

Image by ~uminotorino Some rights reserved

Routing is one of the primary aspects of the MVC framework which makes MVC what it is. While that is possibly over-simplifying, the routing framework is where the MVC philosophy of “convention over configuration” is readily apparent.

Routing also represents one of the more potentially confounding aspects of MVC. Once we move beyond the basics, as our applications become more complex, it generally becomes necessary to customize our routes beyond the simple flexible default MVC route.

Route customization is a complex topic. In this article we will look at some of the basic ways we can modify the conventional MVC routing mechanism to suit the needs of an application which requires more flexibility.

***NOTE:** most of the examples in this post are somewhat contrived, and arbitrary. My goal is to demonstrate some basic route customization options without getting distracted by the business case-specifics under which they might be required. I recognize that in the real world, the examples may or may not represent reasonable cases for customized routes.*

In any sufficiently complex ASP.NET MVC project, it is likely you will need to add one or more custom routes which either supplement or replace the conventional MVC route. As we learned in Routing Basics in ASP.NET MVC, the default MVC route mapping is:

```
{controller}/{action}/{id}
```

This basic convention will handle a surprising number of potential routes. However, often there will be cases where more control is required, either to:

- Route incoming requests to the proper application logic in complex scenarios.
- Allow for well-structured URLs which reflect the structure of our site.
- Create URL structures that are easy to type, and “hackable” in the sense that a user, on examining the structure of a current URL, might reasonably make some navigation guesses based on that structure.

Route customization is achieved in a number of ways, usually by by combining modifications to the basic pattern of the route mapping, employing route ***default options*** to specify controllers and actions explicitly, and less frequently, using ***route constraints*** to limit the ways in which a route will “match” a particular URL segment or parameter.

URLs are Matched Against Routes in

Order – The First Match Wins

Incoming URLs are compared to route patters in the order the patters appear in the ***route dictionary*** (that is what we added the route maps to in our RouteConfig.cs file). The first route which successfully matches a controller, action, and action parameters to either the parameters in the URL or the defaults defined as part of the route map will call into the specified controller and action. This is important, and requires us to think our routes through carefully so that the wrong handler is not called inadvertently.

The basic process of route matching was covered in Routing Basics in ASP.NET MVC.

Modifying the URL Pattern

As we learned previously, in an MVC application, routes define patterns by which incoming URLs are matched to specific controllers, and actions (methods) on those controllers. Route mappings recognize URLs as a pattern of segments separated (or delimited) by a slash (/) character. Each segment may contain various combinations of ***literals*** (text values) and ***parameter placeholders***. Parameter placeholders are identified in a route definition by braces.

MVC recognizes the special parameter placeholders `{controller}` and `{action}` and uses these to locate the appropriate controller and method to call in response to an incoming URL. In addition to these two special placeholders, we can add just about anything to a route as a parameter placeholder to suit our purpose, so long as we adhere to good URL design principles, and of course, reasonably expect the parameter to be useful.

We are familiar with the default MVC pattern shown above. We can see that the route is composed of three segments, with no literals, and parameter placeholders for the special parameters `{controller}` and `{action}`, as well as an additional parameter called `{id}`.

When creating route URL patterns, we can combine literals with parameter placeholders in any number of ways, so long as parameter placeholders are always separated by either a delimiter, or at least one literal character. The following are examples of valid route patterns (whether they are sensible or not is another issue):

Examples of Valid Route Patterns in ASP.NET MVC:

Route Pattern	URL Example
`mysite/{username}/{action}`	~/mysite/jatten/login
`public/blog/{controller}-{action}/{postId}`	~/public/blog/posts-show/123
`{country}-`	~/us-

`{lang}/{controller}/{action}/{id}`	`en/products/show/123`
`products/buy/{productId}-{productName}`	`~/products/buy/2145-widgets`

The following pattern is not valid, because the {controller} parameter placeholder and the {action} parameter placeholder are not separated by either a slash (/) or another literal character. In this case, the MVC framework has no way to know where one parameter ends, and the next begins (assume the controller is named “People” and the action is “Show’):

Route Pattern	URL Example
`mysite/{controller}{action}/{id}`	`~/mysite/peopleshow/5`

Not all of the valid route examples above include either the `{controller}` parameter placeholder or the `{action}` parameter placeholder. The last example does not include either. Also, most include user-defined parameters, such as `{username}` or `{country}`. We’ll take a look at both in the next two sections.

Modifying Route Default Options

In conjunction with modifying the route URL pattern, we can also take advantage of the route defaults to create routes which are more specific, and in some cases which map only to a specific controller and/or action.

The standard MVC project file contains the route mapping configuration in a file named RouteConfig.cs:

The standard MVC RouteConfig.cs File:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollectio
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInf
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new {
                controller = "Home",
                action = "Index",
                id = UrlParameter.Optional }
        );
    }
}
```

The route above establishes a default value for both the controller and the action, in case one more both are not provided as part of an incoming URL.

We can take this a step further, and add a new, more specific route which, when it matches a specific URL pattern, calls into one specific controller. In the following, we have added a new route mapping in our `RouteConfig.cs` file:

Adding a More Restrictive Route:

```
public static void RegisterRoutes(RouteCollection ro
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}")
    // ALL THE PROPERTIES:
```

```
    // rentalProperties/
    routes.MapRoute(
        name: "Properties",
        url: "RentalProperties/{action}/{id}",
        defaults: new
        {
            controller = "RentalProperty",
            action = "All",
            id = UrlParameter.Optional
        }
    );
    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new {
            controller = "Home",
            action = "Index",
            id = UrlParameter.Optional }
    );
}
```

First off, take note that we have added the new, more specific route ***before the default in the file*** (which determines the order it is added to the route dictionary). This is because ***routes are evaluated for a match to an incoming URL in order***. The default MVC route a pretty general, and if route order is not carefully considered when adding additional routes to your project, the default route may inadvertently match a URL intended for a different handler.

Beyond this, what we see in the new route added above is that there is no `{controller}` parameter placeholder in the URL pattern of the “Properties” route. Since no controller can be passed to this route as a parameter, it will always map to the `RentalPropertiesController` , and to whichever Action is provided as a parameter.

Since there is a default action defined, if the incoming URL matches the route but does not contain an Action parameter, the default `All()` method will be called.

A Contrived Example

So far, in this rather contrived example, we haven't accomplished anything we couldn't have done using the default MVC route pattern. However, suppose in our application we wanted to define the following URL patterns and associated URL examples for accessing views in a property management application:

Desired URL patterns for Simple Property Management Application:

Behavior	URL Example
`Show all the rental properties`	`~/rentalproperties/`
`Show a specific rental property`	`~/rentalproperties/propertyname/`
`Show a specific unit at a property`	`~/rentalproperties/propertyname/units/unitNo`

We could define a `RentalPropertyController` class as follows:

Rental Property Controller Example:

```
public class RentalPropertiesController : Controller
{
    private RentalPropertyTestData _data = new Renta
    // List all the properties
    public ActionResult All()
    {
        var allRentalProperties = _data.RentalProper
        return View(allRentalProperties);
    }
    // get a specific property, display details and
    public ActionResult RentalProperty(string rental
    {
        var rentalProperty = _data.RentalProperties.
        return View(rentalProperty);
    }
    // get a specific unit at a specific property:
    public ActionResult Unit(string rentalPropertyNa
    {
        var unit = _data.Units.Find(u => u.RentalPro
        return View(unit);
    }
}
```

Given the above controller, and the desired URL patterns in the previous table, we can see that the default MVC route mapping would work for some, but not all of our URLs. We might be better served by adding the following controller-and-action-specific routes to our application, so that our `RouteConfig.cs` file looks like this:

Modified Route Config File for Property Management Example Application:

```
public static void RegisterRoutes(RouteCollection ro
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}")
    // RentalProperties/Boardwalk/Units/4A
    routes.MapRoute(
        name: "RentalPropertyUnit",
        url: "RentalProperties/{rentalPropertyName}/
        defaults: new
        {
            controller = "RentalProperties",
            action = "Unit",
        }
    );
    // RentalProperties/Boardwalk
    routes.MapRoute(
        name: "RentalProperty",
        url: "RentalProperties/{rentalPropertyName}"
        defaults: new
        {
            controller = "RentalProperties",
            action = "RentalProperty",
        }
    );
    // RentalProperties/
    routes.MapRoute(
        name: "RentalProperties",
        url: "RentalProperties",
        defaults: new
        {
            controller = "RentalProperties",
            action = "All",
            id = UrlParameter.Optional
        }
    );
    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new
        {
            controller = "Home",
            action = "Index",
            id = UrlParameter.Optional
        }
    );
}
```

As we see now, all of the newly added routes explicitly specify both the controller and action, and allow for our progressively enhanced URL structure which includes the property name as part of the URL.

Of course, in the real world, this URL scheme would likely not work the way we desire, because (depending upon how our database is going to be used) the names of rental properties may not be unique, even within a given management company. If the company operated rental properties in several states or even counties within a state, it is entirely possible that there would be more than one property named (for example) “Maple Glen.” It is a sad fact that despite all the wonderfully simple examples we can find while learning, ***the real world often fails to model in the way we want.***

Adding Route Constraints

Route constraints can be used to further restrict the URLs that can be considered a match for a specific route. While we have so far examined whether routes match based upon URL structure and parameter names, Route Constraints allow us to add some additional specificity.

The MVC framework recognizes two types of constraints, either a `string`, or a class which implements the interface `IRouteConstraint`.

Regular Expressions in Route

Constraints

When a string is provided as a constraint, the MVC framework interprets the string as a Regular Expression, which can be employed to limit the ways a URL might match a particular route. A common scenario in this regard would be to restrict the value of a route parameter to numeric values.

Example Route Mapping for Blog Application with no Constraint:

```
routes.MapRoute(
    name: "BlogPost",
    url: "blog/posts/{postId}",
    defaults: new
    {
        controller = "Posts",
        action = "GetPost",
    },
);
```

This route will call into the `PostsController` of a hypothetical blog application and retrieve a specific post, based on the unique postId. This route will properly match the following URL:

http://domain/blog/posts/123

Unfortunately, if will also match this:

http://domain/blog/posts/gimme

The basic Route Constraint allows us to test the value of the `{postId}` parameter to determine if it is a numeric value. We can re-write our route mapping thusly:

Example Route Mapping for Blog Application with Added Constraint:

```
routes.MapRoute(
    name: "BlogPost",
    url: "blog/posts/{postId}",
    defaults: new
    {
        controller = "Posts",
        action = "GetPost",
    },
    new {postId = @"\d+" }
);
```

The tiny Regular Expression `@"\d+` in the code above basically limits the matches for this route to URLs in which the postId parameter contains one or more digits. In other words, nothing but integers, please.

I'm not going to dig too deep into Regular Expressions here. Suffice it to say that Regular Expressions can be used to great effect in developing a complex routing scheme for your application.

A Note and More Resources for Regular Expressions

Regular Expressions are a powerful tool, and also a giant pain the the ass. As Jamie Zawinskie said, ***"Some people, when confronted with a problem, think 'I know,***

I'll use regular expressions.' Now they have two problems." However, Regular Expressions are sometimes the very best tool for the job – restricting route matches in an MVC application is one of those cases. I find three tools most helpful when writing more complex regular expressions. In order of importance:

- Google (doh!)
- Expresso Regular Expression Development Tool (older, but invaluable)
- The site Regular-Expressions.info (More than you ever wanted to know)

Custom Route Constraints using IRouteConstraint

The other option for using route constraints is to create a class which implements the IRouteConstraint interface.

We'll take a quick look at a custom route constraint which can be used to make sure a particular controller is excluded as a match for our default MVC route.

Note: The following code was adapted from a code project article by Vijaya Anand (Code Project member After2050), originally posted at his personal blog Proud Parrot.

In creating a custom constraint, we first create a new class which implements `IRouteConstraint`. `IRouteConstraint` defines a single method, `Match` for which we need to provide the implementation. In order for the constraint to work, we will also need to create a constructor with which we set the argument(s) required for the method:

The Custom Constraint Class ExcludeController:

```
public class ExcludeController : IRouteConstraint
{
    private readonly string _controller;
    public ExcludeController(string controller)
    {
        _controller = controller;
    }
    public bool Match(HttpContextBase httpContext,
        Route route, string parameterName,
        RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        // Does the _controller argument match the c
        // dictionary for the current request?
        return string.Equals(values["controller"].To
            _controller, StringComparison.OrdinalIgn
    }
}
```

Now, suppose we have a hypothetical controller `ConfigurationController` for which we have defined a special route which requires authentication. We don't want the MVC default route map to inadvertently allow access to the Configuration controller for unauthenticated users. We can make sure of this by modifying our default MVC route like so:

Adding a Custom Constraint to the Default MVC Route:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new
    {
        controller = "Home",
        action = "Index",
        id = UrlParameter.Optional
    },
    constraints: new { controller = new ExcludeContr
);
```

Now, if a URL which has not been matched by any previous route mapping is evaluated for a match with our default route, the MVC framework will call the Match method of our custom constraint `ExcludeController`. In the case above, if the URL contains the `{controller}` parameter with a value of “Configuration” the `Match` method will return a value of false, causing the route to reject the URL as a match.

Stephen Walther presents an excellent tutorial on Custom Route Constraints, and specifically, creating a authentication constraint at his blog in ASP.NET MVC Tip #30 – Create Custom Route Constraints.

8/23/2013 – UPDATE: *I forgot to mention that Attribute Routing is an alternative routing model which allows you to specify routes directly on the controller action handler for a specific URL pattern. You can use Attribute Routing now as a Nuget package, but it is actually going to be an integrated option shipped with the upcoming release of ASP.NET 5 and Web Api 2. There is some debate as to whether Attribute Routing is a good thing, as evidenced by K. Scott Allen’s recent post on OdeToCode. In this article I keep the focus on customization of the standard routing mechanism in*

ASP.NET.

Additional Resources

- Routing Basics in ASP.NET MVC
- Routing Basics in ASP.NET Web API
- Creating a Clean, Minimal-Footprint ASP.NET WebAPI Project with VS 2012 and ASP.NET MVC 4
- Building Out a Clean, REST-ful Web Api Service with a Minimal Web Api Project
- Deploying an Azure Website from Source Control
- Webmatrix 3: Integrated Git and Deployment to Azure

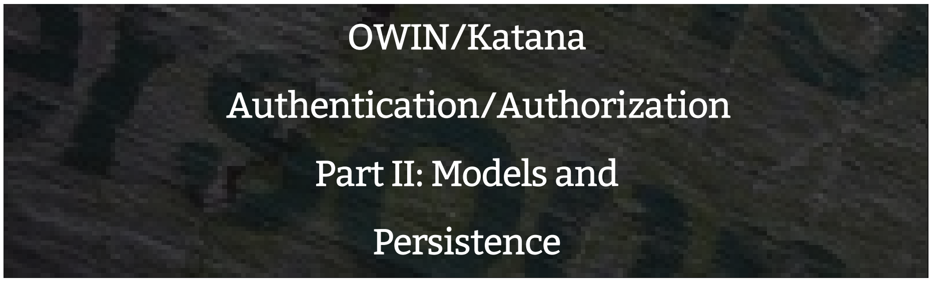
ASP.NET C# MVC

SHARE:

RELATED ARTICLES

ASP.NET

ASP.NET Web Api: Understanding



OWIN/Katana Authentication/Authorization Part II: Models and Persistence

CODEPROJECT

Java: Checked Exceptions, Revisited Part II

CODEPROJECT

Git: Combine and Organize Messy Commits Using Interactive Rebase

VIEW COMMENTS

PREVIOUS POST

C# – Wildcard Search Using LINQ

NEXT POST

Install Sublime Text 3 (beta) on Linux Mint or Ubuntu