# Understanding Routing Precedence in ASP.NET MVC and Web API

Rion Williams

5 Jul 2016 CPOL

Rate this: ★★★★★ 4.86 (4 votes)

Understanding Routing Precedence in ASP.NET MVC and Web API

Routing can be a very tricky issue within ASP.NET MVC and Web API applications. This can be especially true if you have a variety of different routes with varying parameters defined in such a way that a single request could satisfy multiple routes.

This blog post will cover some of the precedence rules that are applied to routing in ASP.NET MVC and Web API and how to leverage these to ensure that the route you want to use gets used.

## What are Routes? How Do They Work?

A Route in ASP.NET simply consists of a pattern and this pattern is going to be mapped to a handler. The handlers themselves can be a variety of different mechanisms (e.g. a physical file, a Web Forms page, or an MVC Controller class). In simplest terms, **Routes define how requests are handled within your application**.

Routes will consist of the following three components:

1. **A name** to identify the Route itself.

2. **A pattern to match URLs** to match a request with its appropriate handler.
3. **A handler** to tell requests that match the pattern where to go.

You can see an example of a route declaration in ASP.NET MVC below, which contains all three of these criteria:

Hide Copy Code

```
routes.MapRoute(
        "Default",                                          // Name
        "{controller}/{action}/{id}",                       // Pattern
        new { controller = "Home", action = "Index", id = "" } // Handler
);
```

Routes can also be defined by using the `[Route]` attribute and decorating a particular Controller Action with it:

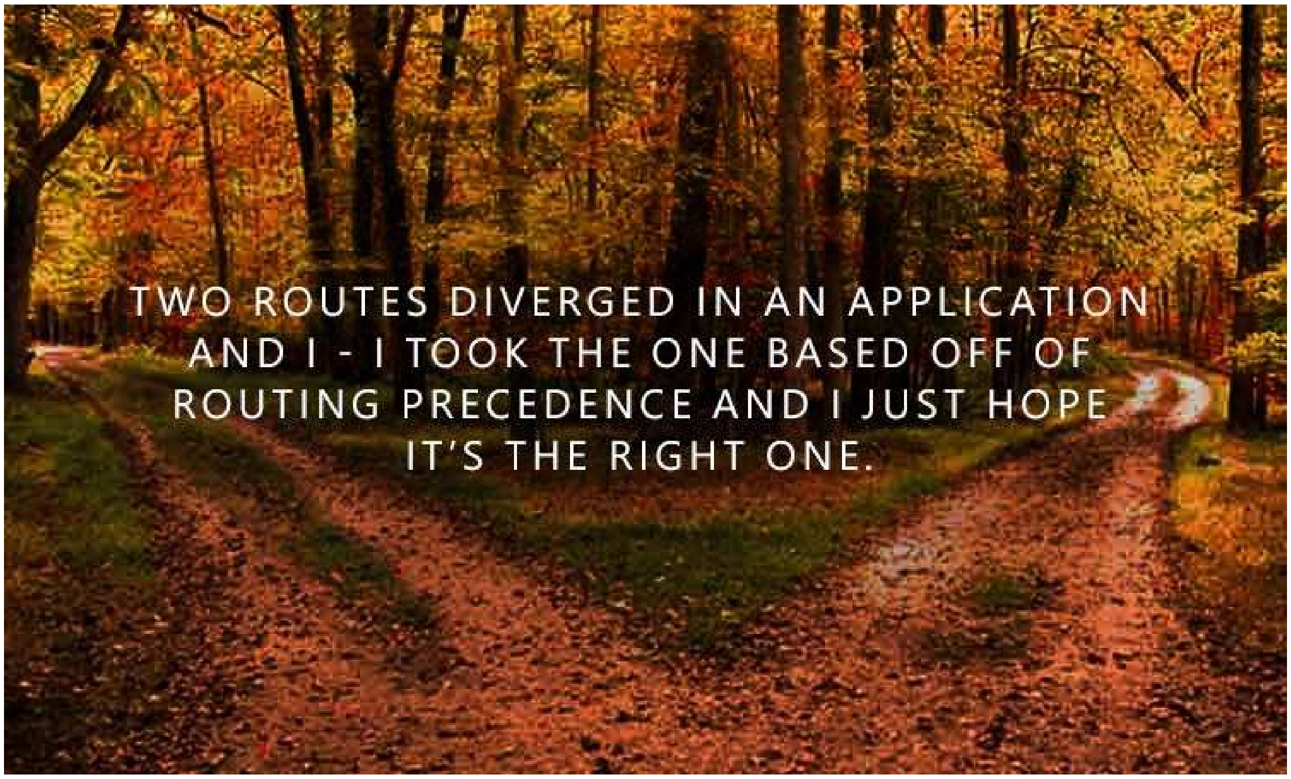
Hide Copy Code

```
[Route("widgets/{brand}", Order = 1)]
public IEnumerable<Widget> GetWidgetsByBrand(string brand) { ... }
```

Since the focus of this post isn't really about routing itself and more of routing precedence, I won't go into any more detail about declaring them.

ASP.NET will store all of the defined routes within a `Routes` Table and when a request comes, it will look through these routes to determine the one best suited to serve for the request. To know how this works, we need to know how routes are prioritized and that's why this blog post exists at all.

**tl;dr; Routes match requests using a pattern and tell them where to go.**


TWO ROUTES DIVERGED IN AN APPLICATION AND I - I TOOK THE ONE BASED OFF OF ROUTING PRECEDENCE AND I JUST HOPE IT'S THE RIGHT ONE.

## Routing Precedence

Routing can be a blessing and a curse within an application. It can provide you with the freedom to define all sorts of very creative and easily understandable approaches to accessing data, but when overused, things can get hairy. **The primary reason that routing can make you want to pull your hair out is that a single request can match multiple routes.**

Routing order can be broken down into the following steps:

1. Check the `Order` property (if available).
2. Order all routes without an explicit `Order` attribute as follows:

    1. Literal segments
    2. Route parameters with constraints
    3. Route parameters without constraints
    4. Wildcard parameter segments with constraints
    5. Wildcard parameter segments without constraints

3. As a tie-breaker, order the routes via a case-insensitive string comparison.

Confused yet? That's okay.

Let's talk about defining Route Order and the different types of routes for a moment to clear things up.

## Route Order

You can use the `[Route]` attribute to explicitly set when a particular route is going to be checked against via the `Order` property. **By default, all defined routes have an Order value of 0 and routes are processed from lowest to highest.** This is the primary reason that it is so important for establishing route precedence, as it's a single attribute that will govern the order that routes are processed.

Let's look at the following three routes:

Hide Copy Code

```
[Route("Alpha", Order = 1)]
public void Fizz() { ... }
[Route("Beta")]
public void Buzz() { ... }
[Route("Gamma")]
public void FizzBuzz() { ... }
```

Since the `Alpha` route has an `Order` property of `1`, it will always be evaluated last out of these three routes. The only scenarios where that would not be true would be if another route had a higher `Order` value or an equal one (and it received priority via the other criteria).

## Literal Segments

A Literal Segment route can be thought of as a "hard-coded" route. It contains no types of parameters and no constraints. A few examples of these types of routes might look like:

Hide Copy Code

```
/widgets/broken
/employees
```

## Route Parameters

`Route` parameters are going to have a bit more information that literals, but not by much. The only major difference is that they will have a "placeholder" that can be used to define parameters (similar to the String.Format() method).

Hide Copy Code

```
/widgets/{widgetId}
/reports/{year}/{month}
```

## Route Parameters with Constraints

`Route` parameters can also be constrained to a specific type. This is important as they will be evaluated prior to unconstrained ones:

Hide Copy Code

```
/widgets/{widgetId:int}
/reports/{year:int}/{month:int}
```

## Wildcard Parameters

Wildcard parameters are the last type of parameters to review and you generally won't see these as much as the aforementioned types. These function as "catch-all" parameters and may contain more than a single segment.

Consider the following route :

Hide Copy Code

```
/query/widgets/{*queryvalues}
```

Notice the `{*queryvalues}` segment, the leading asterisk indicates that this is a wildcard parameter and thus it can accept all sorts of additional segments in it:

Hide Copy Code

```
/query/widgets/broken
/query/widgets/byyear/2015
```

## Wildcard Parameters with Constraints

Wildcard parameters can also feature type constraints just like normal route parameters as well if you are expecting a certain type:

Hide Copy Code

```
/query/widgets/{*byyear:int}
```

## Applying Precedence in Action

Now to break this down, let's define several actions that meet the criteria at least one of each of these routes and we will look at each phase as we order the routes:

Hide Copy Code

```
[RoutePrefix("widgets")]
public class WidgetsController : ApiController
{
    [Route("{widgetId:int}")] // Constrained Route Parameter
    public HttpResponseMessage Get(int widgetId) { ... }

    [Route("new")] // Literal Segment
    public HttpResponseMessage GetNew() { ... }

    [Route("{*features})] // Unconstrained Wildcard Parameter
    public HttpResponseMessage GetByFeatures(string features) { ... }

    [Route("broken", RouteOrder = 1)] // Literal Segment (with Route Order)
    public HttpResponseMessage GetBroken() { ... }

    [Route("{brand}")]  // Unconstrained Route Parameter
    public HttpResponseMessage GetByBrand(string brand) { ... }

    [Route("{*date:datetime}")]  // Constrained Wildcard Parameter
    public HttpResponseMessage GetByManufacturedDate(DateTime date) { ... }
}
```

So with all of these routes defined, let's output all of them as they appear in the Controller and update them as we apply each rule:

Hide Copy Code

```
Get(int widgetId) { ... }            // widgets/{widgetId:int}
GetNew()                             // widgets/new
GetByFeatures(string features)       // widgets/{*features}
GetBroken()                          // widgets/broken
GetByBrand(string brand)             // widgets/{brand}
GetByManufacturedDate(DateTime date) // widgets/{*date:datetime}
```

Firstly, **check the routes for the `Order` attribute**. Now since the `GetBroken()` action has an `Order` of `1`, we know that this will be the last route to be evaluated (since the default is `0` and routes are processed in ascending order):

Hide Copy Code

```
Get(int widgetId) { ... }            // widgets/{widgetId:int}
GetNew()                             // widgets/new
GetByFeatures(string features)       // widgets/{*features}
GetByBrand(string brand)             // widgets/{brand}
GetByManufacturedDate(DateTime date) // widgets/{*date:datetime}
GetBroken()                          // widgets/broken
```

Next up, we will **check for any literal routes** and ensure that they will be the first to be processed. In this case, the `GetNew()` action meets that requirement, so move it to the beginning:

Hide Copy Code

```
GetNew()                             // widgets/new
Get(int widgetId) { ... }            // widgets/{widgetId:int}
GetByFeatures(string features)       // widgets/{*features}
GetByBrand(string brand)             // widgets/{brand}
GetByManufacturedDate(DateTime date) // widgets/{*date:datetime}
GetBroken()                          // widgets/broken
```

After literals, **the next routes to be processed are constrained route parameters**. The `Get(int)` action meets this requirement as it accepts an integer for its `widgetId` parameter, so it will fall next in line:

Hide Copy Code

```
GetNew()                             // widgets/new
Get(int widgetId) { ... }            // widgets/{widgetId:int}
GetByFeatures(string features)       // widgets/{*features}
GetByBrand(string brand)             // widgets/{brand}
GetByManufacturedDate(DateTime date) // widgets/{*date:datetime}
GetBroken()                          // widgets/broken
```

Next, **unconstrained route parameters are processed**. The `GetByBrand(string)` action meets this requirement as it contains a parameter, but it doesn't specify a type for it:

Hide Copy Code

```
GetNew()                             // widgets/new
Get(int widgetId) { ... }            // widgets/{widgetId:int}
GetByBrand(string brand)             // widgets/{brand}
GetByFeatures(string features)       // widgets/{*features}
GetByManufacturedDate(DateTime date) // widgets/{*date:datetime}
GetBroken()                          // widgets/broken
```

After all of the literals and route parameters have been routed, **the next to be evaluated are the wildcard parameters**. More specifically, the constrained wildcard parameters, which the `GetbyManufacturedDate(DateTime)` route matches:

Hide Copy Code

```
GetNew()                            // widgets/new
Get(int widgetId) { ... }           // widgets/{widgetId:int}
GetByBrand(string brand)            // widgets/{brand}
GetByManufacturedDate(DateTime date) // widgets/{*date:datetime}
GetByFeatures(string features)      // widgets/{*features}
GetBroken()                         // widgets/broken
```

And finally, **the last route type to evaluate is unconstrained wildcard parameters (or "catch-all" routes)**. The `GetByFeatures(string)` route fits under that category so it will be placed prior to the `GetBroken()` route, which had a higher `Order` property:

Hide Copy Code

```
GetNew()                            // widgets/new
Get(int widgetId) { ... }           // widgets/{widgetId:int}
GetByBrand(string brand)            // widgets/{brand}
GetByManufacturedDate(DateTime date) // widgets/{*date:datetime}
GetByFeatures(string features)      // widgets/{*features}
GetBroken()                         // widgets/broken
```

And that's it as we don't have any ties to break, but **if a tie between two routes did exist, then a case-insensitive comparison of the route template names would resolve it**.

Any requests that come through will be evaluated in that order (i.e. the first pattern to match against the URL will be action that handles the request).

## Your Mileage May Vary

One important thing to point out in a post like this is that generally you shouldn't have to deal with routing precedence too often. If you are responsible for building and designing your API, then try to keep your API as simple as possible.

Don't try to create this sprawling, verbose API of endless routes unless you absolutely have to. Complicated APIs can lead to complicated problems and in most cases if your API is too difficult to interact with, others won't likely want to use it (especially if you are building a public-facing API).

## License

## Share

## About the Author

**Rion Williams**

Software Developer (Senior)
United States

An experienced Software Developer and Graphic Designer with an extensive knowledge of object-oriented programming, software architecture, design methodologies and database design principles. Specializing in Microsoft Technologies and focused on leveraging a strong technical background and a creative skill-set to create meaningful and successful applications.

Well versed in all aspects o...

**show more**

## Comments and Discussions

**You must Sign In to use this message board.**

Search Comments

-- There are no messages in this forum --

Permalink
Advertise
Privacy
Cookies
Terms of Use

Layout: fixed | fluid