Attribute Routing in ASP.NET Web API 2

Routing is how Web API matches a URI to an action. Web API 2 supports a new type of routing, called *attribute routing*. As the name implies, attribute routing uses attributes to define routes. Attribute routing gives you more control over the URIs in your web API. For example, you can easily create URIs that describe hierarchies of resources.

The earlier style of routing, called convention-based routing, is still fully supported. In fact, you can combine both techniques in the same project.

This topic shows how to enable attribute routing and describes the various options for attribute routing. For an end-to-end tutorial that uses attribute routing, see Create a REST API with Attribute Routing in Web API 2.

Prerequisites

Visual Studio 2017 Community, Professional, or Enterprise edition

Alternatively, use NuGet Package Manager to install the necessary packages. From the **Tools** menu in Visual Studio, select **NuGet Package Manager**, then select **Package Manager Console**. Enter the following command in the Package Manager Console window:

```
Install-Package Microsoft.AspNet.WebApi.WebHost
```

Why Attribute Routing?

The first release of Web API used *convention-based* routing. In that type of routing, you define one or more route templates, which are basically parameterized strings. When the framework receives a request, it matches the URI against the route template. For more information about convention-based routing, see Routing in ASP.NET Web API.

One advantage of convention-based routing is that templates are defined in a single place, and the routing rules are applied consistently across all controllers. Unfortunately, convention-based routing makes it hard to support certain URI patterns that are common in RESTful APIs. For example, resources often contain child resources: Customers have orders, movies have actors, books have authors, and so forth. It's natural to create URIs that reflect these relations:

```
/customers/1/orders
```

This type of URI is difficult to create using convention-based routing. Although it can be done, the results don't scale well if you have many controllers or resource types.

With attribute routing, it's trivial to define a route for this URI. You simply add an attribute to the controller action:

C#Copy

```
[Route("customers/{customerId}/orders")]

public IEnumerable<Order> GetOrdersByCustomer(int customerId) { ... }
```

Here are some other patterns that attribute routing makes easy.

API versioning

In this example, "/api/v1/products" would be routed to a different controller than "/api/v2/products".

/api/v1/products /api/v2/products

Overloaded URI segments

In this example, "1" is an order number, but "pending" maps to a collection.

/orders/1 /orders/pending

Multiple parameter types

In this example, "1" is an order number, but "2013/06/16" specifies a date.

/orders/1 /orders/2013/06/16

Enabling Attribute Routing

To enable attribute routing, call **MapHttpAttributeRoutes** during configuration. This extension method is defined in the **System.Web.Http.HttpConfigurationExtensions** class.

C#Copy

```
using System.Web.Http;


namespace WebApplication
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API routes
            config.MapHttpAttributeRoutes();


            // Other Web API configuration not shown.
        }
    }
}
```

Attribute routing can be combined with convention-based routing. To define convention-based routes, call the **MapHttpRoute** method.

C#Copy

```
public static class WebApiConfig
```

```
{
   public static void Register(HttpConfiguration config)
   {
      // Attribute routing.
      config.MapHttpAttributeRoutes();

      // Convention-based routing.
      config.Routes.MapHttpRoute(
         name: "DefaultApi",
         routeTemplate: "api/{controller}/{id}",
         defaults: new { id = RouteParameter.Optional }
      );
   }
}
```

For more information about configuring Web API, see [Configuring ASP.NET Web API 2]().

Note: Migrating From Web API 1

Prior to Web API 2, the Web API project templates generated code like this:

C#Copy

```
protected void Application_Start()
{
   // WARNING - Not compatible with attribute routing.
   WebApiConfig.Register(GlobalConfiguration.Configuration);
}
```

If attribute routing is enabled, this code will throw an exception. If you upgrade an existing Web API project to use attribute routing, make sure to update this configuration code to the following:

C#Copy

```
protected void Application_Start()
{
   // Pass a delegate to the Configure method.
   GlobalConfiguration.Configure(WebApiConfig.Register);
}
```

Note

For more information, see **Configuring Web API with ASP.NET Hosting**.

Adding Route Attributes

Here is an example of a route defined using an attribute:

C#Copy

```
public class OrdersController : ApiController
{
    [Route("customers/{customerId}/orders")]
    [HttpGet]
    public IEnumerable<Order> FindOrdersByCustomer(int customerId) { ... }
}
```

The string "customers/{customerId}/orders" is the URI template for the route. Web API tries to match the request URI to the template. In this example, "customers" and "orders" are literal segments, and "{customerId}" is a variable parameter. The following URIs would match this template:

- http://localhost/customers/1/orders
- http://localhost/customers/bob/orders
- http://localhost/customers/1234-5678/orders

You can restrict the matching by using constraints, described later in this topic.

Notice that the "{customerId}" parameter in the route template matches the name of the *customerId* parameter in the method. When Web API invokes the controller action, it tries to bind the route parameters. For example, if the URI is http://example.com/customers/1/orders, Web API tries to bind the value "1" to the *customerId* parameter in the action.

A URI template can have several parameters:

C#Copy

```
[Route("customers/{customerId}/orders/{orderId}")]
public Order GetOrderByCustomer(int customerId, int orderId) { ... }
```

Any controller methods that do not have a route attribute use convention-based routing. That way, you can combine both types of routing in the same project.

HTTP Methods

Web API also selects actions based on the HTTP method of the request (GET, POST, etc). By default, Web API looks for a case-insensitive match with the start of the controller method name. For example, a controller method named PutCustomers matches an HTTP PUT request.

You can override this convention by decorating the method with any of the following attributes:

- **[HttpDelete]**

- **[HttpGet]**
- **[HttpHead]**
- **[HttpOptions]**
- **[HttpPatch]**
- **[HttpPost]**
- **[HttpPut]**

In the following example, Web API maps the CreateBook method to HTTP POST requests.

C#Copy

```
[Route("api/books")]

[HttpPost]

public HttpResponseMessage CreateBook(Book book) { ... }
```

For all other HTTP methods, including non-standard methods, use the **AcceptVerbs** attribute, which takes a list of HTTP methods.

C#Copy

```
// WebDAV method

[Route("api/books")]

[AcceptVerbs("MKCOL")]

public void MakeCollection() { }
```

Route Prefixes

Often, the routes in a controller all start with the same prefix. For example:

C#Copy

```
public class BooksController : ApiController

{

   [Route("api/books")]

   public IEnumerable<Book> GetBooks() { ... }


   [Route("api/books/{id:int}")]

   public Book GetBook(int id) { ... }


   [Route("api/books")]

   [HttpPost]
```

```
    public HttpResponseMessage CreateBook(Book book) { ... }
}
```

You can set a common prefix for an entire controller by using the **[RoutePrefix]** attribute:

C#Copy

```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // GET api/books
    [Route("")]
    public IEnumerable<Book> Get() { ... }

    // GET api/books/5
    [Route("{id:int}")]
    public Book Get(int id) { ... }

    // POST api/books
    [Route("")]
    public HttpResponseMessage Post(Book book) { ... }
}
```

Use a tilde (~) on the method attribute to override the route prefix:

C#Copy

```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // GET /api/authors/1/books
    [Route("~/api/authors/{authorId:int}/books")]
    public IEnumerable<Book> GetByAuthor(int authorId) { ... }

    // ...
}
```

The route prefix can include parameters:

C#Copy

```
[RoutePrefix("customers/{customerId}")]

public class OrdersController : ApiController

{

   // GET customers/1/orders

   [Route("orders")]

   public IEnumerable<Order> Get(int customerId) { ... }

}
```

Route Constraints

Route constraints let you restrict how the parameters in the route template are matched. The general syntax is "{parameter:constraint}". For example:

C#Copy

```
[Route("users/{id:int}")]

public User GetUserById(int id) { ... }


[Route("users/{name}")]

public User GetUserByName(string name) { ... }
```

Here, the first route will only be selected if the "id" segment of the URI is an integer. Otherwise, the second route will be chosen.

The following table lists the constraints that are supported.

Constraint	Description	Example
alpha	Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z)	{x:alpha}
bool	Matches a Boolean value.	{x:bool}
datetime	Matches a **DateTime** value.	{x:datetime}
decimal	Matches a decimal value.	{x:decimal}
double	Matches a 64-bit floating-point value.	{x:double}
float	Matches a 32-bit floating-point value.	{x:float}
guid	Matches a GUID value.	{x:guid}
int	Matches a 32-bit integer value.	{x:int}
length	Matches a string with the specified length or within a specified range of lengths.	{x:length(6)} {x:length(1,20

Constraint	**Description**	**Example**
long	Matches a 64-bit integer value.	{x:long}
max	Matches an integer with a maximum value.	{x:max(10)}
maxlength	Matches a string with a maximum length.	{x:maxlength(10)}
min	Matches an integer with a minimum value.	{x:min(10)}
minlength	Matches a string with a minimum length.	{x:minlength(10)}
range	Matches an integer within a range of values.	{x:range(10,50)}
regex	Matches a regular expression.	{x:regex(^\d{3}-\d{3}-\d{4}$)}

Notice that some of the constraints, such as "min", take arguments in parentheses. You can apply multiple constraints to a parameter, separated by a colon.

C#Copy

```
[Route("users/{id:int:min(1)}")]

public User GetUserById(int id) { ... }
```

Custom Route Constraints

You can create custom route constraints by implementing the **IHttpRouteConstraint** interface. For example, the following constraint restricts a parameter to a non-zero integer value.

C#Copy

```
public class NonZeroConstraint : IHttpRouteConstraint

{

  public bool Match(HttpRequestMessage request, IHttpRoute route, string parameterName,

    IDictionary<string, object> values, HttpRouteDirection routeDirection)

  {

    object value;

    if (values.TryGetValue(parameterName, out value) && value != null)

    {

      long longValue;

      if (value is long)

      {

        longValue = (long)value;

        return longValue != 0;
```

```
            }


            string valueString = Convert.ToString(value, CultureInfo.InvariantCulture);
            if (Int64.TryParse(valueString, NumberStyles.Integer,
                CultureInfo.InvariantCulture, out longValue))
            {
                return longValue != 0;
            }
        }
        return false;
    }
}
```

The following code shows how to register the constraint:

C#Copy

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var constraintResolver = new DefaultInlineConstraintResolver();
        constraintResolver.ConstraintMap.Add("nonzero", typeof(NonZeroConstraint));


        config.MapHttpAttributeRoutes(constraintResolver);
    }
}
```

Now you can apply the constraint in your routes:

C#Copy

```
[Route("{id:nonzero}")]
public HttpResponseMessage GetNonZero(int id) { ... }
```

You can also replace the entire **DefaultInlineConstraintResolver** class by implementing the **IInlineConstraintResolver** interface. Doing so will replace all of the built-in constraints, unless your implementation of **IInlineConstraintResolver** specifically adds them.

Optional URI Parameters and Default Values

You can make a URI parameter optional by adding a question mark to the route parameter. If a route parameter is optional, you must define a default value for the method parameter.

C#Copy

```
public class BooksController : ApiController
{
    [Route("api/books/locale/{lcid:int?}")]
    public IEnumerable<Book> GetBooksByLocale(int lcid = 1033) { ... }
}
```

In this example, /api/books/locale/1033 and /api/books/locale return the same resource.

Alternatively, you can specify a default value inside the route template, as follows:

C#Copy

```
public class BooksController : ApiController
{
    [Route("api/books/locale/{lcid:int=1033}")]
    public IEnumerable<Book> GetBooksByLocale(int lcid) { ... }
}
```

This is almost the same as the previous example, but there is a slight difference of behavior when the default value is applied.

- In the first example ("{lcid:int?}"), the default value of 1033 is assigned directly to the method parameter, so the parameter will have this exact value.
- In the second example ("{lcid:int=1033}"), the default value of "1033" goes through the model-binding process. The default model-binder will convert "1033" to the numeric value 1033. However, you could plug in a custom model binder, which might do something different.

(In most cases, unless you have custom model binders in your pipeline, the two forms will be equivalent.)

Route Names

In Web API, every route has a name. Route names are useful for generating links, so that you can include a link in an HTTP response.

To specify the route name, set the **Name** property on the attribute. The following example shows how to set the route name, and also how to use the route name when generating a link.

C#Copy

```
public class BooksController : ApiController
{
    [Route("api/books/{id}", Name="GetBookById")]
    public BookDto GetBook(int id)
    {
        // Implementation not shown...
    }

    [Route("api/books")]
    public HttpResponseMessage Post(Book book)
    {
        // Validate and add book to database (not shown)

        var response = Request.CreateResponse(HttpStatusCode.Created);

        // Generate a link to the new book and set the Location header in the response.
        string uri = Url.Link("GetBookById", new { id = book.BookId });
        response.Headers.Location = new Uri(uri);
        return response;
    }
}
```

Route Order

When the framework tries to match a URI with a route, it evaluates the routes in a particular order. To specify the order, set the **Order** property on the route attribute. Lower values are evaluated first. The default order value is zero.

Here is how the total ordering is determined:

1. Compare the **Order** property of the route attribute.
2. Look at each URI segment in the route template. For each segment, order as follows:

a. Literal segments.

b. Route parameters with constraints.

c. Route parameters without constraints.

d. Wildcard parameter segments with constraints.

e. Wildcard parameter segments without constraints.

3. In the case of a tie, routes are ordered by a case-insensitive ordinal string comparison (OrdinalIgnoreCase) of the route template.

Here is an example. Suppose you define the following controller:

C#Copy

```
[RoutePrefix("orders")]
public class OrdersController : ApiController
{
   [Route("{id:int}")] // constrained parameter
   public HttpResponseMessage Get(int id) { ... }

   [Route("details")]  // literal
   public HttpResponseMessage GetDetails() { ... }

   [Route("pending", RouteOrder = 1)]
   public HttpResponseMessage GetPending() { ... }

   [Route("{customerName}")]  // unconstrained parameter
   public HttpResponseMessage GetByCustomer(string customerName) { ... }

   [Route("{*date:datetime}")]  // wildcard
   public HttpResponseMessage Get(DateTime date) { ... }
}
```

These routes are ordered as follows.

1. orders/details
2. orders/{id}
3. orders/{customerName}
4. orders/{*date}
5. orders/pending

Notice that "details" is a literal segment and appears before "{id}", but "pending" appears last because the **Order** property is 1. (This example assumes there are no customers named "details" or "pending". In general, try to avoid ambiguous routes. In this example, a better route template for GetByCustomer is "customers/{customerName}")