Routing in ASP.NET Web API

By Mike Wasson | February 11, 2012

This article describes how ASP.NET Web API routes HTTP requests to controllers.

> If you are familiar with ASP.NET MVC, Web API routing is very similar to MVC routing. The main difference is that Web API uses the HTTP method, not the URI path, to select the action. You can also use MVC-style routing in Web API. This article does not assume any knowledge of ASP.NET MVC.

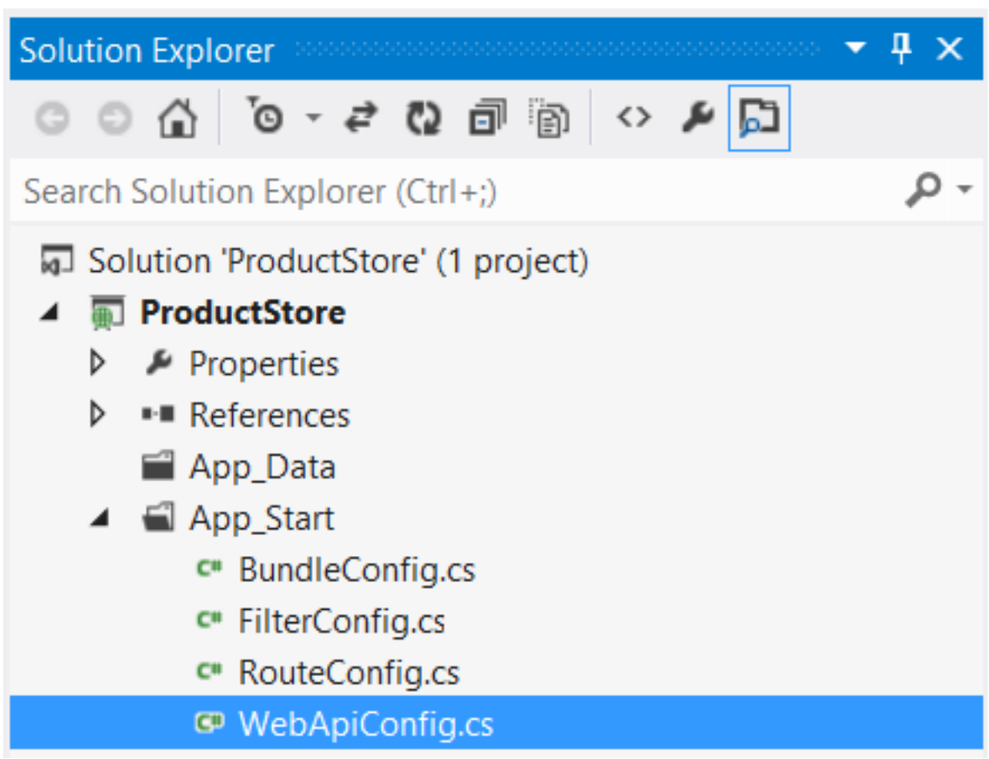
Routing Tables

In ASP.NET Web API, a *controller* is a class that handles HTTP requests. The public methods of the controller are called *action methods* or simply *actions*. When the Web API framework receives a request, it routes the request to an action.

To determine which action to invoke, the framework uses a *routing table*. The Visual Studio project template for Web API creates a default route:

```
routes.MapHttpRoute(
    name: "API Default",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

This route is defined in the WebApiConfig.cs file, which is placed in the App_Start directory:



For more information aboout the **WebApiConfig** class, see Configuring ASP.NET Web API .

If you self-host Web API, you must set the routing table directly on the **HttpSelfHostConfiguration** object. For more information, see Self-Host a Web API.

Each entry in the routing table contains a *route template*. The default route template for Web API is "api/{controller}/{id}". In this template, "api" is a literal path segment, and {controller} and {id} are placeholder variables.

When the Web API framework receives an HTTP request, it tries to match the URI against one of the route templates in the routing table. If no route matches, the client receives a 404 error. For example, the following URIs match the default route:

/api/contacts

/api/contacts/1

/api/products/gizmo1

However, the following URI does not match, because it lacks the "api" segment:

    /contacts/1

Note: The reason for using "api" in the route is to avoid collisions with ASP.NET MVC routing. That way, you can have "/contacts" go to an MVC controller, and "/api/contacts" go to a Web API controller. Of course, if you don't like this convention, you can change the default route table.

Once a matching route is found, Web API selects the controller and the action:

- To find the controller, Web API adds "Controller" to the value of the *{controller}* variable.
- To find the action, Web API looks at the HTTP method, and then looks for an action whose name begins with that HTTP method name. For example, with a GET request, Web API looks for an action that starts with "Get...", such as "GetContact" or "GetAllContacts". This convention applies only to GET, POST, PUT, and DELETE methods. You can enable other HTTP methods by using attributes on your controller. We'll see an example of that later.
- Other placeholder variables in the route template, such as *{id},* are mapped to action parameters.

Let's look at an example. Suppose that you define the following controller:

```
public class ProductsController : ApiController
{
    public void GetAllProducts() { }
    public IEnumerable<Product> GetProductById(int id) { }
    public HttpResponseMessage DeleteProduct(int id){ }
}
```

Here are some possible HTTP requests, along with the action that gets invoked for each:

HTTP Method	URI Path	Action	Parameter
GET	api/products	GetAllProducts	*(none)*
GET	api/products/4	GetProductById	4
DELETE	api/products/4	DeleteProduct	4
POST	api/products	*(no match)*	

Notice that the *{id}* segment of the URI, if present, is mapped to the *id* parameter of the action. In this example, the controller defines two GET methods, one with an *id* parameter and one with no parameters.

Also, note that the POST request will fail, because the controller does not define a "Post..." method.

Routing Variations

The previous section described the basic routing mechanism for ASP.NET Web API. This section describes some variations.

HTTP Methods

Instead of using the naming convention for HTTP methods, you can explicitly specify the HTTP method for an action by decorating the action method with the **HttpGet**, **HttpPut**, **HttpPost**, or **HttpDelete** attribute.

In the following example, the FindProduct method is mapped to GET requests:

```
public class ProductsController : ApiController
{
    [HttpGet]
    public Product FindProduct(id) {}
}
```

To allow multiple HTTP methods for an action, or to allow HTTP methods other than GET, PUT, POST, and DELETE, use the **AcceptVerbs** attribute, which takes a list of HTTP methods.

```
public class ProductsController : ApiController
{
    [AcceptVerbs("GET", "HEAD")]
    public Product FindProduct(id) { }

    // WebDAV method
    [AcceptVerbs("MKCOL")]
    public void MakeCollection() { }
}
```

Routing by Action Name

With the default routing template, Web API uses the HTTP method to select the action. However, you can also create a route where the action name is included in the URI:

```
routes.MapHttpRoute(
    name: "ActionApi",
    routeTemplate: "api/{controller}/{action}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

In this route template, the *{action}* parameter names the action method on the controller. With this style of routing, use attributes to specify the allowed HTTP methods. For example, suppose your controller has the following method:

```
public class ProductsController : ApiController
{
    [HttpGet]
    public string Details(int id);
}
```

In this case, a GET request for "api/products/details/1" would map to the Details method. This style of routing is similar to ASP.NET MVC, and may be appropriate for an RPC-style API.

You can override the action name by using the **ActionName** attribute. In the following example, there are two actions that map to "api/products/thumbnail/*id*. One supports GET and the other supports POST:

```
public class ProductsController : ApiController
{
    [HttpGet]
    [ActionName("Thumbnail")]
    public HttpResponseMessage GetThumbnailImage(int id);

    [HttpPost]
    [ActionName("Thumbnail")]
    public void AddThumbnailImage(int id);
}
```

Non-Actions

To prevent a method from getting invoked as an action, use the **NonAction** attribute. This signals to the framework that the method is not an action, even if it would otherwise match the routing rules.

```
// Not an action method.
[NonAction]
public string GetPrivateData() { ... }
```

Further Reading

This topic provided a high-level view of routing. For more detail, see Routing and Action Selection, which describes exactly how the framework matches a URI to a route, selects a controller, and then selects the action to invoke.

Author Information

Mike Wasson – Mike Wasson is a programmer-writer at Microsoft.