Routing and Action Selection in ASP.NET Web API

12/14/2018 • 9 minutes to read • +4

In this article

by Mike Wasson

This article describes how ASP.NET Web API routes an HTTP request to a particular action on a controller.

> **Note**
>
> For a high-level overview of routing, see **Routing in ASP.NET Web API**.

This article looks at the details of the routing process. If you create a Web API project and find that some requests don't get routed the way you expect, hopefully this article will help.

Routing has three main phases:

1. Matching the URI to a route template.
2. Selecting a controller.
3. Selecting an action.

You can replace some parts of the process with your own custom behaviors. In this article, I describe the default behavior. At the end, I note the places where you can customize the behavior.

Route Templates

A route template looks similar to a URI path, but it can have placeholder values, indicated with curly braces:

C# Copy

```
"api/{controller}/public/{category}/{id}"
```

When you create a route, you can provide default values for some or all of the placeholders:

C# Copy

```
defaults: new { category = "all" }
```

You can also provide constraints, which restrict how a URI segment can match a placeholder:

C# Copy

```
constraints: new { id = @"\d+" }   // Only matches if "id" is one or more
digits.
```

The framework tries to match the segments in the URI path to the template. Literals in the template must match exactly. A placeholder matches any value, unless you specify constraints. The framework does not match other parts of the URI, such as the host name or the query parameters. The framework selects the first route in the route table that matches the URI.

There are two special placeholders: "{controller}" and "{action}".

- "{controller}" provides the name of the controller.
- "{action}" provides the name of the action. In Web API, the usual convention is to omit "{action}".

Defaults

If you provide defaults, the route will match a URI that is missing those segments. For example:

C# Copy

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{category}",

    defaults: new { category = "all" }
```

```
    defaults: new { category = "all" }
);
```

The URIs `http://localhost/api/products/all` and `http://localhost/api/products` match the preceding route. In the latter URI, the missing `{category}` segment is assigned the default value `all`.

Route Dictionary

If the framework finds a match for a URI, it creates a dictionary that contains the value for each placeholder. The keys are the placeholder names, not including the curly braces. The values are taken from the URI path or from the defaults. The dictionary is stored in the **IHttpRouteData** object.

During this route-matching phase, the special "{controller}" and "{action}" placeholders are treated just like the other placeholders. They are simply stored in the dictionary with the other values.

A default can have the special value **RouteParameter.Optional**. If a placeholder gets assigned this value, the value is not added to the route dictionary. For example:

C# Copy

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

For the URI path "api/products", the route dictionary will contain:

- controller: "products"
- category: "all"

For "api/products/toys/123", however, the route dictionary will contain:

- controller: "products"
- category: "toys"
- id: "123"

The defaults can also include a value that does not appear anywhere in the route template. If the route matches, that value is stored in the dictionary. For example:

C# Copy

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

If the URI path is "api/root/8", the dictionary will contain two values:

- controller: "customers"
- id: "8"

Selecting a Controller

Controller selection is handled by the **IHttpControllerSelector.SelectController** method. This method takes an **HttpRequestMessage** instance and returns an **HttpControllerDescriptor**. The default implementation is provided by the **DefaultHttpControllerSelector** class. This class uses a straightforward algorithm:

1. Look in the route dictionary for the key "controller".
2. Take the value for this key and append the string "Controller" to get the controller type name.
3. Look for a Web API controller with this type name.

For example, if the route dictionary contains the key-value pair "controller" = "products", then the controller type is "ProductsController". If there is no matching type, or multiple matches, the framework returns an error to the client.

For step 3, **DefaultHttpControllerSelector** uses the **IHttpControllerTypeResolver** interface to get the list of Web API controller types. The default implementation of **IHttpControllerTypeResolver** returns all public classes that (a) implement **IHttpController**, (b) are not abstract, and (c) have a name that ends in "Controller".

Action Selection

After selecting the controller, the framework selects the action by calling the **IHttpActionSelector.SelectAction** method. This method takes an **HttpControllerContext** and returns an **HttpActionDescriptor**.

The default implementation is provided by the **ApiControllerActionSelector** class. To select an action, it looks at the following:

- The HTTP method of the request.
- The "{action}" placeholder in the route template, if present.
- The parameters of the actions on the controller.

Before looking at the selection algorithm, we need to understand some things about controller actions.

Which methods on the controller are considered "actions"? When selecting an action, the framework only looks at public instance methods on the controller. Also, it excludes "special name" methods (constructors, events, operator overloads, and so forth), and methods inherited from the **ApiController** class.

HTTP Methods. The framework only chooses actions that match the HTTP method of the request, determined as follows:

1. You can specify the HTTP method with an attribute: **AcceptVerbs**, **HttpDelete**, **HttpGet**, **HttpHead**, **HttpOptions**, **HttpPatch**, **HttpPost**, or **HttpPut**.
2. Otherwise, if the name of the controller method starts with "Get", "Post", "Put", "Delete", "Head", "Options", or "Patch", then by convention the action supports that HTTP method.
3. If none of the above, the method supports POST.

Parameter Bindings. A parameter binding is how Web API creates a value for a parameter. Here is the default rule for parameter binding:

- Simple types are taken from the URI.
- Complex types are taken from the request body.

Simple types include all of the .NET Framework primitive types, plus **DateTime**, **Decimal**, **Guid**, **String**, and **TimeSpan**. For each action, at most one parameter can read the request body.

> **Note**
>
> It is possible to override the default binding rules. See **WebAPI Parameter binding under the hood**.

With that background, here is the action selection algorithm.

1. Create a list of all actions on the controller that match the HTTP request method.
2. If the route dictionary has an "action" entry, remove actions whose name does not match this value.
3. Try to match action parameters to the URI, as follows:
   a. For each action, get a list of the parameters that are a simple type, where the binding gets the parameter from the URI. Exclude optional parameters.
   b. From this list, try to find a match for each parameter name, either in the route dictionary or in the URI query string. Matches are case insensitive and do not depend on the parameter order.
   c. Select an action where every parameter in the list has a match in the URI.
   d. If more that one action meets these criteria, pick the one with the most parameter matches.
4. Ignore actions with the **[NonAction]** attribute.

Step #3 is probably the most confusing. The basic idea is that a parameter can get its value either from the URI, from the request body, or from a custom binding. For parameters that come from the URI, we want to ensure that the URI actually contains a value for that parameter, either in the path (via the route dictionary) or in the query string.

For example, consider the following action:

C# Copy

```
public void Get(int id)
```

The *id* parameter binds to the URI. Therefore, this action can only match a URI that contains a value for "id", either in the route dictionary or in the query string.

Optional parameters are an exception, because they are optional. For an optional parameter, it's OK if the binding can't get the value from the URI.

Complex types are an exception for a different reason. A complex type can only bind to the URI through a custom binding. But in that case, the framework cannot know in advance whether the parameter would bind to a particular URI. To find out, it would need to invoke the binding. The goal of the selection algorithm is to select an action from the static description, before invoking any bindings. Therefore, complex types are excluded from the matching algorithm.

After the action is selected, all parameter bindings are invoked.

Summary:

- The action must match the HTTP method of the request.
- The action name must match the "action" entry in the route dictionary, if present.
- For every parameter of the action, if the parameter is taken from the URI, then the parameter name must be found either in the route dictionary or in the URI query string. (Optional parameters and parameters with complex types are excluded.)
- Try to match the most number of parameters. The best match might be a method with no parameters.

Extended Example

Routes:

C# Copy

```
routes.MapHttpRoute(
    name: "ApiRoot",
    routeTemplate: "api/root/{id}",
    defaults: new { controller = "products", id = RouteParameter.Optional }
);
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

Controller:

C# Copy

```
public class ProductsController : ApiController
{
    public IEnumerable<Product> GetAll() {}
    public Product GetById(int id, double version = 1.0) {}
    [HttpGet]
    public void FindProductsByName(string name) {}
    public void Post(Product value) {}
    public void Put(int id, Product value) {}
}
```

HTTP request:

Console Copy

```
GET http://localhost:34701/api/products/1?version=1.5&details=1
```

Route Matching

The URI matches the route named "DefaultApi". The route dictionary contains the following entries:

- controller: "products"
- id: "1"

The route dictionary does not contain the query string parameters, "version" and "details", but these will still be considered during action selection.

Controller Selection

From the "controller" entry in the route dictionary, the controller type is `ProductsController`.

Action Selection

The HTTP request is a GET request. The controller actions that support GET are `GetAll`, `GetById`, and `FindProductsByName`. The route dictionary does not contain an entry for "action", so we don't need to match the action name.

Next, we try to match parameter names for the actions, looking only at the GET actions.

Action	Parameters to Match
`GetAll`	none
`GetById`	"id"
`FindProductsByName`	"name"

Notice that the *version* parameter of `GetById` is not considered, because it is an optional parameter.

The `GetAll` method matches trivially. The `GetById` method also matches, because the route dictionary contains "id". The `FindProductsByName` method does not match.

The `GetById` method wins, because it matches one parameter, versus no parameters for `GetAll`. The method is invoked with the following parameter values:

- *id* = 1
- *version* = 1.5

Notice that even though *version* was not used in the selection algorithm, the value of the parameter comes from the URI query string.

Extension Points

Web API provides extension points for some parts of the routing process.

Interface	Description
IHttpControllerSelector	Selects the controller.
IHttpControllerTypeResolver	Gets the list of controller types. The **DefaultHttpControllerSelector** chooses the controller type from this list.
IAssembliesResolver	Gets the list of project assemblies. The **IHttpControllerTypeResolver** interface uses this list to find the controller types.
IHttpControllerActivator	Creates new controller instances.
IHttpActionSelector	Selects the action.
IHttpActionInvoker	Invokes the action.

To provide your own implementation for any of these interfaces, use the **Services** collection on the **HttpConfiguration** object:

C# Copy

```
var config = GlobalConfiguration.Configuration;
config.Services.Replace(typeof(IHttpControllerSelector), new
MyControllerSelector(config));
```

Is this page helpful?

Yes No