**Get Started with ASP.NET Web API 2 (C#)**

HTTP is not just for serving up web pages. HTTP is also a powerful platform for building APIs that expose services and data. HTTP is simple, flexible, and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications.

ASP.NET Web API is a framework for building web APIs on top of the .NET Framework.

## **Adding a Model**

A model is an object that represents the data in your application. ASP.NET Web API can automatically serialize your model to JSON, XML, or some other format, and then write the serialized data into the body of the HTTP response message. As long as a client can read the serialization format, it can deserialize the object. Most clients can parse either XML or JSON. Moreover, the client can indicate which format it wants by setting the Accept header in the HTTP request message.

## **Adding a Controller**

In Web API, a controller is an object that handles HTTP requests. We'll add a controller that can return either a list of products or a single product specified by ID.

## **HttpResponseMessage**

If the action returns an [HttpResponseMessage](https://msdn.microsoft.com/library/system.net.http.httpresponsemessage.aspx), Web API converts the return value directly into an HTTP response message, using the properties of the **HttpResponseMessage** object to populate the response.

This option gives you a lot of control over the response message. For example, the following controller action sets the Cache-Control header.

C#Copy

public class ValuesController : ApiController

{

public HttpResponseMessage Get()

{

HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.OK, "value");

response.Content = new StringContent("hello", Encoding.Unicode);

response.Headers.CacheControl = new CacheControlHeaderValue()

{

MaxAge = TimeSpan.FromMinutes(20)

};

return response;

}

}

If you pass a domain model to the **CreateResponse** method, Web API uses a [media formatter](https://docs.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/media-formatters) to write the serialized model into the response body.

C#Copy

public HttpResponseMessage Get()

{

// Get a list of products from a database.

IEnumerable<Product> products = GetProductsFromDB();

// Write the list to the response body.

HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.OK, products);

return response;

}

The **IHttpActionResult** interface was introduced in Web API 2. Essentially, it defines an **HttpResponseMessage** factory. Here are some advantages of using the **IHttpActionResult** interface:

* Simplifies [unit testing](https://docs.microsoft.com/en-us/aspnet/web-api/overview/testing-and-debugging/unit-testing-controllers-in-web-api) your controllers.
* Moves common logic for creating HTTP responses into separate classes.
* Makes the intent of the controller action clearer, by hiding the low-level details of constructing the response.

**IHttpActionResult** contains a single method, **ExecuteAsync**, which asynchronously creates an **HttpResponseMessage** instance.

C#Copy

public interface IHttpActionResult

{

Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken);

}

If a controller action returns an **IHttpActionResult**, Web API calls the **ExecuteAsync** method to create an **HttpResponseMessage**. Then it converts the **HttpResponseMessage** into an HTTP response message.

Here is a simple implementaton of **IHttpActionResult** that creates a plain text response:

C#Copy

public class TextResult : IHttpActionResult

{

string \_value;

HttpRequestMessage \_request;

public TextResult(string value, HttpRequestMessage request)

{

\_value = value;

\_request = request;

}

public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)

{

var response = new HttpResponseMessage()

{

Content = new StringContent(\_value),

RequestMessage = \_request

};

return Task.FromResult(response);

}

}

## **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:

C#Copy

routes.MapHttpRoute(

name: "API Default",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

## **Routing Variations**

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

### HTTP verbs

Instead of using the naming convention for HTTP verbs, you can explicitly specify the HTTP verb for an action by decorating the action method with one of the following attributes:

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

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

C#Copy

public class ProductsController : ApiController

{

[HttpGet]

public Product FindProduct(id) {}

}

To allow multiple HTTP verbs for an action, or to allow HTTP verbs other than GET, PUT, POST, DELETE, HEAD, OPTIONS, and PATCH, use the [AcceptVerbs] attribute, which takes a list of HTTP verbs.

C#Copy

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 verb to select the action. However, you can also create a route where the action name is included in the URI:

C#Copy

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 verbs. For example, suppose your controller has the following method:

C#Copy

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:

C#Copy

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.

C#Copy

// Not an action method.

[NonAction]

public string GetPrivateData() { ... }

var jsonFormatter = config.Formatters.OfType<JsonMediaTypeFormatter>().FirstOrDefault();

jsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();

Circular reference cannot be solved by serialization.