ASP.Net Web API

ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

HTTP is not just for serving up web pages. It 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.

When we talk about the Database as a resource we usually talk in terms of CRUD operations. i.e. Create, Retrieve, Update and Delete. Now the philosophy of REST is that for a remote resource all these operations should be possible and they should be possible using simple HTTP protocols.

Now the basic CRUD operations are mapped to the HTTP protocols in the following manner:

* **GET**: This maps to the R(Retrieve) part of the CRUD operation. This will be used to retrieve the required data (representation of data) from the remote resource.
* **PUT**: This maps to the U(Update) part of the CRUD operation. This protocol will update the current representation of the data on the remote server.
* **POST**: This maps to the C(Create) part of the CRUD operation. This will create a new entry for the current data that is being sent to the server.
* **DELETE**: This maps to the D(Delete) part of the CRUD operation. This will delete the specified data from the remote server.

|  |  |
| --- | --- |
| **ASP.NET MVC** | **ASP.NET WebAPI** |
| used to create web applications that returns both view and data | used to create HTTP services, It returns only data. |
| return data in json format using jsonResult | return data in JSON, XML |
| Requests are mapped to actions name. | Requests are mapped to the actions based on HTTP verbs |

|  |
| --- |
|  |
| Web API REST | WCF REST |
| HTTP only. Suitable for access from various browsers, mobile devices etc enabling wide reach. | Support multiple transport protocols (HTTP, TCP, UDP, and custom transports) and allows switching between them. |
| Enables building Web APIs that support wide variety of media types including XML, JSON etc. | Support multiple encodings (Text, MTOM, and Binary) of the same message type and allows switching between them. |
| Uses basic protocol and formats such as HTTP, WebSockets, SSL, Equerry, JSON, and XML. There is no support for higher level protocols. | Supports building services with WS-\* standards like Reliable Messaging, Transactions, Message Security. |
| HTTP is request/response, for additional need SignalR and WebSockets integration. | Supports Request-Reply, One Way, and Duplex message exchange patterns |
| Ships with .NET framework but is open-source | Ships with the .NET framework. |

## **Creating a Resource**

public HttpResponseMessage PostProduct(Product item)

{

item = repository.Add(item);

var response = Request.CreateResponse<Product>(HttpStatusCode.Created, item);

string uri = Url.Link("DefaultApi", new { id = item.Id });

response.Headers.Location = new Uri(uri);

return response;

}

Notice that the method return type is now **HttpResponseMessage**. By returning an **HttpResponseMessage** instead of a Product, we can control the details of the HTTP response message, including the status code and the Location header.+

The **CreateResponse** method creates an **HttpResponseMessage** and automatically writes a serialized representation of the Product object into the body fo the response message.

**Updating a Resource**

public void PutProduct(int id, Product product)

{

product.Id = id;

if (!repository.Update(product))

{

throw new HttpResponseException(HttpStatusCode.NotFound);

}

}

The method name starts with "Put...", so Web API matches it to PUT requests. The method takes two parameters, the product ID and the updated product. The *id* parameter is taken from the URI path, and the *product* parameter is deserialized from the request body. By default, the ASP.NET Web API framework takes simple parameter types from the route and complex types from the request body.

## **Deleting a Resource**

To delete a resourse, define a "Delete..." method.

Copy

C#

public void DeleteProduct(int id)

{

Product item = repository.Get(id);

if (item == null)

{

throw new HttpResponseException(HttpStatusCode.NotFound);

}

repository.Remove(id);

}

If a DELETE request succeeds, it can return status 200 (OK) with an entity-body that describes the status; status 202 (Accepted) if the deletion is still pending; or status 204 (No Content) with no entity body. In this case, the DeleteProduct method has a void return type, so ASP.NET Web API automatically translates this into status code 204 (No Content).

Model Validation in ASP.Net Web API

[System.ComponentModel.DataAnnotations](https://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx)

namespace MyApi.Models

{

public class Product

{

public int Id { get; set; }

[Required]

public string Name { get; set; }

public decimal Price { get; set; }

[Range(0, 999)]

public double Weight { get; set; }

}

}

public class ProductsController : ApiController

{

public HttpResponseMessage Post(Product product)

{

if (ModelState.IsValid)

{

// Do something with the product (not shown).

return new HttpResponseMessage(HttpStatusCode.OK);

}

else

{

return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);

}

}

}

## **Handling Validation Errors**

Web API does not automatically return an error to the client when validation fails. It is up to the controller action to check the model state and respond appropriately.

public class ValidateModelAttribute : ActionFilterAttribute

{

public override void OnActionExecuting(HttpActionContext actionContext)

{

if (actionContext.ModelState.IsValid == false)

{

actionContext.Response = actionContext.Request.CreateErrorResponse(

HttpStatusCode.BadRequest, actionContext.ModelState);

}

}

}

To apply this filter to all Web API controllers, add an instance of the filter to the **HttpConfiguration.Filters** collection during configuration:

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

config.Filters.Add(new ValidateModelAttribute());

// ...

}

}

Another option is to set the filter as an attribute on individual controllers or controller actions:

public class ProductsController : ApiController

{

[ValidateModel]

public HttpResponseMessage Post(Product product)

{

// ...

}

}

Routing in Web API

routes.MapHttpRoute(

name: "API Default",

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

defaults: new { id = RouteParameter.Optional }

);

This route is defined in the WebApiConfig.cs file,

We create HttpClient instance as a proxy to access Web API methods.

### **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.

public class ProductsController : ApiController

{

[HttpGet]

public Product FindProduct(id) {}

}

### **Routing by Action Name**

With the default routing template, Web API uses the HTTP method to select the action.

C#

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.

# Preventing Cross-Site Request Forgery (CSRF) Attacks in ASP.NET Web API

Client Side

<script>

@functions{

public string TokenHeaderValue()

{

string cookieToken, formToken;

AntiForgery.GetTokens(null, out cookieToken, out formToken);

return cookieToken + ":" + formToken;

}

}

$.ajax("api/values", {

type: "post",

contentType: "application/json",

data: { }, // JSON data goes here

dataType: "json",

headers: {

'RequestVerificationToken': '@TokenHeaderValue()'

}

});

</script>

Server Side Token Validation

void ValidateRequestHeader(HttpRequestMessage request)

{

string cookieToken = "";

string formToken = "";

IEnumerable<string> tokenHeaders;

if (request.Headers.TryGetValues("RequestVerificationToken", out tokenHeaders))

{

string[] tokens = tokenHeaders.First().Split(':');

if (tokens.Length == 2)

{

cookieToken = tokens[0].Trim();

formToken = tokens[1].Trim();

}

}

AntiForgery.Validate(cookieToken, formToken);

}

Help Page in ASP.Net Web API

To create the help pages, expand the **Areas** node in Solution Explorer (shown in Figure 10) and then open the **HelpPageConfig.cs** file

config.SetDocumentationProvider(new XmlDocumentationProvider(HttpContext.Current.Server.MapPath("~/App\_Data/XmlDocument.xml")));

Next, enable XML documentation by selecting the **XML** **documentation** **file** option in the **Build** section of the project’s properties, as shown in Figure 11.

Documentation will build from the comments above the action method in the ApiController.

# HTTP Message Handlers

A message handler is a class that receives an HTTP request and returns an HTTP response. Message handlers derive from the abstract **HttpMessageHandler** class.

## **Server-Side Message Handlers**

On the server side, the Web API pipeline uses some built-in message handlers:

* **HttpServer** gets the request from the host.
* **HttpRoutingDispatcher** dispatches the request based on the route.
* **HttpControllerDispatcher** sends the request to a Web API controller.

## **Custom Message Handlers**

To write a custom message handler, derive from **System.Net.Http.DelegatingHandler** and override the **SendAsync** method.

Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request, CancellationToken cancellationToken);

## **Adding a Handler to the Pipeline**

To add a message handler on the server side, add the handler to the **HttpConfiguration.MessageHandlers** collection.

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

config.MessageHandlers.Add(new MessageHandler1());

config.MessageHandlers.Add(new MessageHandler2());

// Other code not shown...

}

}

Message handlers are called in the same order that they appear in **MessageHandlers** collection. Because they are nested, the response message travels in the other direction. That is, the last handler is the first to get the response message.

Content Negotiation in ASP.Net Web API

 content negotiation as "the process of selecting the best representation for a given response when there are multiple representations available." The primary mechanism for content negotiation in HTTP are these request headers:

* **Accept:** Which media types are acceptable for the response, such as "application/json," "application/xml," or a custom media type such as "application/vnd.example+xml"
* **Accept-Charset:** Which character sets are acceptable, such as UTF-8 or ISO 8859-1.
* **Accept-Encoding:** Which content encodings are acceptable, such as gzip.
* **Accept-Language:** The preferred natural language, such as "en-us".

## **Serialization**

If a Web API controller returns a resource as CLR type, the pipeline serializes the return value and writes it into the HTTP response body.

public Product GetProduct(int id)

{

var item = \_products.FirstOrDefault(p => p.ID == id);

if (item == null)

{

throw new HttpResponseException(HttpStatusCode.NotFound);

}

return item;

}

Request

GET http://localhost.:21069/api/products/1 HTTP/1.1

Host: localhost.:21069

Accept: application/json, text/javascript, \*/\*; q=0.01

Response

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

Content-Length: 57

Connection: Close

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}

Action Results in ASP.Net Web API

A Web API controller action can return any of the following:

1. void
2. **HttpResponseMessage**
3. **IHttpActionResult**
4. Some other type

## **void**

If the return type is void, Web API simply returns an empty HTTP response with status code 204 (No Content).

Example controller

public class ValuesController : ApiController

{

public void Post()

{

}

}

HTTP response

HTTP/1.1 204 No Content

Server: Microsoft-IIS/8.0

Date: Mon, 27 Jan 2014 02:13:26 GMT

## **HttpResponseMessage**

If the action returns an [HttpResponseMessage](https://msdn.microsoft.com/en-us/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.

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;

}

}

Response:

HTTP/1.1 200 OK

Cache-Control: max-age=1200

Content-Length: 10

Content-Type: text/plain; charset=utf-16

Server: Microsoft-IIS/8.0

Date: Mon, 27 Jan 2014 08:53:35 GMT

hello

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.

Web API uses the Accept header in the request to choose the formatter. For more information, see [Content Negotiation](https://docs.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/content-negotiation).

## **IHttpActionResult**

The **IHttpActionResult** interface was introducted in Web API 2. Essentially, it defines an **HttpResponseMessage** factory. Here are some advantages of using the **IHttpActionResult** interface:1

* 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.

## **Other Return Types**

For all other return types, Web API uses a [media formatter](https://docs.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/media-formatters) to serialize the return value. Web API writes the serialized value into the response body. The response status code is 200 (OK).

public class ProductsController : ApiController

{

public IEnumerable<Product> Get()

{

return GetAllProductsFromDB();

}

}

A disadvantage of this approach is that you cannot directly return an error code, such as 404. However, you can throw an **HttpResponseException** for error codes.

Media Formatter in ASP.Net Web API

A media type, also called a MIME type, identifies the format of a piece of data. In HTTP, media types describe the format of the message body. A media type consists of two strings, a type and a subtype. For example:+

* text/html
* image/png
* application/json

For example, if an HTTP response contains a PNG image, the response might have the following headers.

HTTP/1.1 200 OK

Content-Length: 95267

Content-Type: image/png

To create a media formatter, derive from one of these classes:+

* [MediaTypeFormatter](https://msdn.microsoft.com/en-us/library/system.net.http.formatting.mediatypeformatter.aspx). This class uses asynchronous read and write methods.
* [BufferedMediaTypeFormatter](https://msdn.microsoft.com/en-us/library/system.net.http.formatting.bufferedmediatypeformatter.aspx). This class derives from **MediaTypeFormatter** but uses sychronous read/write methods.

# Exception Handling in ASP.NET Web API

## **HttpResponseException**

The **HttpResponseException** type is a special case. This exception returns any HTTP status code that you specify in the exception constructor

public Product GetProduct(int id)

{

Product item = repository.Get(id);

if (item == null)

{

var resp = new HttpResponseMessage(HttpStatusCode.NotFound)

{

Content = new StringContent(string.Format("No product with ID = {0}", id)),

ReasonPhrase = "Product ID Not Found"

}

throw new HttpResponseException(resp);

}

return item;

}

## **Exception Filters**

An exception filter is executed when a controller method throws any unhandled exception that is not an **HttpResponseException** exception. The **HttpResponseException** type is a special case, because it is designed specifically for returning an HTTP response.

Exception filters implement the **System.Web.Http.Filters.IExceptionFilter** interface.

namespace ProductStore.Filters

{

using System;

using System.Net;

using System.Net.Http;

using System.Web.Http.Filters;

public class NotImplExceptionFilterAttribute : ExceptionFilterAttribute

{

public override void OnException(HttpActionExecutedContext context)

{

if (context.Exception is NotImplementedException)

{

context.Response = new HttpResponseMessage(HttpStatusCode.NotImplemented);

}

}

}

}

## **Registering Exception Filters**

There are several ways to register a Web API exception filter:

* By action
* By controller
* Globally

public class ProductsController : ApiController

{

[NotImplExceptionFilter]

public Contact GetContact(int id)

{

throw new NotImplementedException("This method is not implemented");

}

}

[NotImplExceptionFilter]

public class ProductsController : ApiController

{

// ...

}

GlobalConfiguration.Configuration.Filters.Add(

new ProductStore.NotImplExceptionFilterAttribute());

## **HttpError**

The **HttpError** object provides a consistent way to return error information in the response body.

public HttpResponseMessage GetProduct(int id)

{

Product item = repository.Get(id);

if (item == null)

{

var message = string.Format("Product with id = {0} not found", id);

return Request.CreateErrorResponse(HttpStatusCode.NotFound, message);

}

else

{

return Request.CreateResponse(HttpStatusCode.OK, item);

}

}

**CreateErrorResponse** is an extension method defined in the **System.Net.Http.HttpRequestMessageExtensions** class. Internally, **CreateErrorResponse** creates an **HttpError** instance and then creates an **HttpResponseMessage** that contains the **HttpError**.

HTTP/1.1 404 Not Found

Content-Type: application/json; charset=utf-8

Date: Thu, 09 Aug 2012 23:27:18 GMT

Content-Length: 51

{

"Message": "Product with id = 12 not found"

}

## **Authentication**

When the host authenticates the user, it creates a principal, which is an [IPrincipal](https://msdn.microsoft.com/en-us/library/System.Security.Principal.IPrincipal.aspx) object that represents the security context under which code is running. The host attaches the principal to the current thread by setting **Thread.CurrentPrincipal**. The principal contains an associated **Identity** object that contains information about the user. If the user is authenticated, the **Identity.IsAuthenticated** property returns **true**.

### **HTTP Message Handlers for Authentication**

Instead of using the host for authentication, you can put authentication logic into an [HTTP message handler](https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/http-message-handlers). In that case, the message handler examines the HTTP request and sets the principal.

When should you use message handlers for authentication? Here are some tradeoffs:

* An HTTP module sees all requests that go through the ASP.NET pipeline. A message handler only sees requests that are routed to Web API.
* You can set per-route message handlers, which lets you apply an authentication scheme to a specific route.
* HTTP modules are specific to IIS. Message handlers are host-agnostic, so they can be used with both web-hosting and self-hosting.
* HTTP modules participate in IIS logging, auditing, and so on.
* HTTP modules run earlier in the pipeline. If you handle authentication in a message handler, the principal does not get set until the handler runs. Moreover, the principal reverts back to the previous principal when the response leaves the message handler.

### **Setting the Principal**

If your application performs any custom authentication logic, you must set the principal on two places:

* **Thread.CurrentPrincipal**. This property is the standard way to set the thread's principal in .NET.
* **HttpContext.Current.User**. This property is specific to ASP.NET.

private void SetPrincipal(IPrincipal principal)

{

Thread.CurrentPrincipal = principal;

if (HttpContext.Current != null)

{

HttpContext.Current.User = principal;

}

}

**Authorization**Authorization happens later in the pipeline, closer to the controller. That lets you make more granular choices when you grant access to resources.

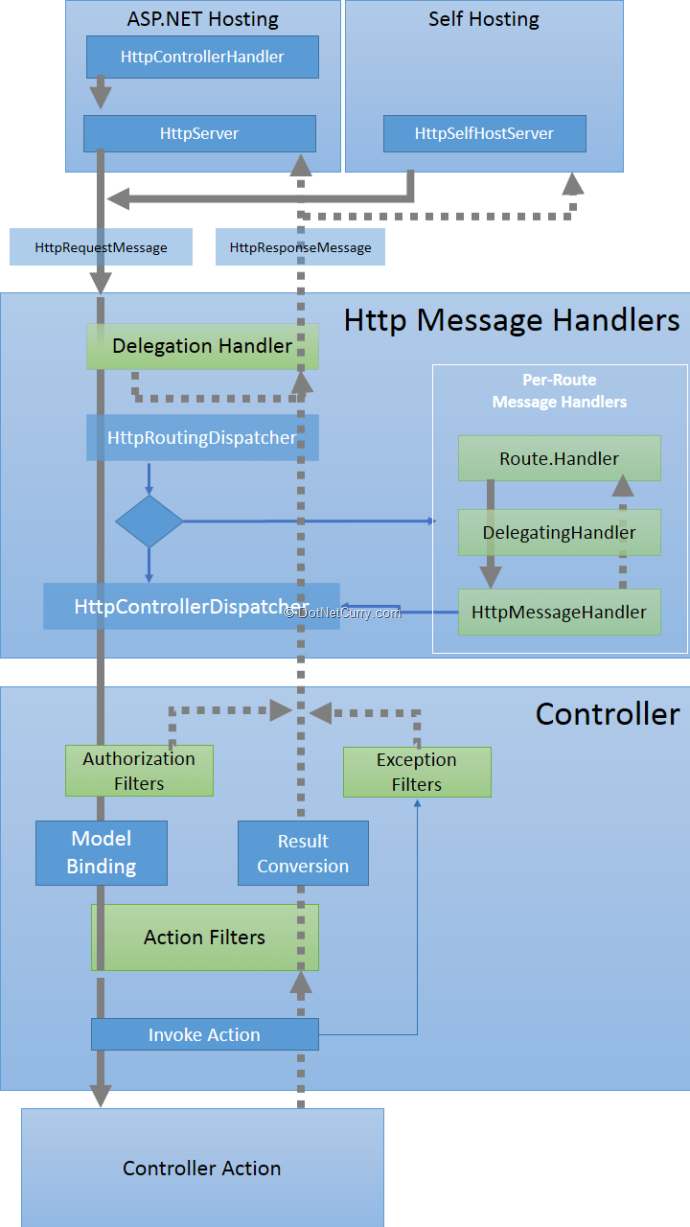
* Authorization filters run before the controller action. If the request is not authorized, the filter returns an error response, and the action is not invoked.
* Within a controller action, you can get the current principal from the **ApiController.User** property. For example, you might filter a list of resources based on the user name, returning only those resources that belong to that user.

### **Using the [Authorize] Attribute**

### **Custom Authorization Filters**

To write a custom authorization filter, derive from one of these types:

* **AuthorizeAttribute**. Extend this class to perform authorization logic based on the current user and the user's roles.
* **AuthorizationFilterAttribute**. Extend this class to perform synchronous authorization logic that is not necessarily based on the current user or role.
* **IAuthorizationFilter**. Implement this interface to perform asynchronous authorization logic; for example, if your authorization logic makes asynchronous I/O or network calls. (If your authorization logic is CPU-bound, it is simpler to derive from **AuthorizationFilterAttribute**, because then you don't need to write an asynchronous method.)

****