FREE eBook

LEARNING asp.net-web-api

Free unaffiliated eBook created from
Stack Overflow contributors.

#asp.net-web-api

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: asp-net-web-api

It is an unofficial and free asp.net-web-api ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official asp.net-web-api.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with asp.net-web-api

Remarks

This section provides an overview of what asp.net-web-api is, and why a developer might want to use it.

It should also mention any large subjects within asp.net-web-api, and link out to the related topics. Since the Documentation for asp.net-web-api is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Detailed instructions on getting asp.net-web-api set up or installed.

What and Why ASP.NET Web API ?

What? : A fully supported and extensible framework for building HTTP based endpoints. In the world of HTML5, mobile devices, and modern development techniques HTTP have become the default option for building rich, scalable services. The ASP.NET Web API provides an easy to use set of default options but also provides a deep extensibility infrastructure to meet the demands of any scenario using HTTP.

Why? :

- An HTML5 application that needs a services layer.
- A mobile application that needs a services layer.
- A client-server desktop application that needs a services layer.

To add Web API to an existing MVC application.

Use Nuget to find the Web Api Package.

You can do that either by using the Manage Nuget Packages and searching for the Web Api package or use Nuget Package Manager and type

```
PM> Install-Package Microsoft.AspNet.WebApi
```

Add WebApiConfig.cs to the App_Start/ folder The config file should contain this.

```
using System.Web.Http;
```

```
namespace WebApplication1
{
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(config =>
        {
            config.MapHttpAttributeRoutes();

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

Source : Configuring ASP.NET Web API

Add `GlobalConfiguration.Configure(WebApiConfig.Register);` in Application_Start of the Global.asax file.

Read Getting started with asp.net-web-api online: https://riptutorial.com/asp-net-web-api/topic/1058/getting-started-with-asp-net-web-api

Chapter 2: ASP.NET Web API Content Negotiation

Examples

ASP.NET Web API Content Negotiation Basic Information

Content Negotiation can be defined as the process of selecting best representation for a given resource. So Content negotiation means the client and server can negotiate between them so that client can get data according to their required format.

There are three points on which internet depends,

- The Resource
- A Pointer to resource(URL)
- Representation of resource

Third point is more important than other two, because everything is works on the basis of how we can see the resource. We can represent a resource in two formats.

1. XML(Extensible Markup Language) Format
2. JSON(JavaScript Object Notation) Format

One of the standards of the RESTful service is that, the client should have the ability to decide in which format they want the response either in JSON or XML. A request that is sent to the server includes an Accept header. Using the Accept header the client can specify the format for the response.

For example,

`Accept: application/xml` returns result in XML format

`Accept: application/json` returns result in JSON format

Depending on the Accept header value in the request, the server sends the response. This is called Content Negotiation.

What happens behind the scene when we request data in specific format?

The ASP.NET Web API controller generates the data that we want to send to the client and hands the data to the Web API pipeline which then look for Accept header of the client. Then, choose a appropriate formatter to format the data.

As ASP.NET Web API is greatly extensible, we can also specify multiple values for accept header in the request header.

```
Accept: application/xml,application/json
```

In the above case, server choose the first formatter to format the data of response.

We can also specify quality factor in the accept header. In this case, server choose a format which have higher quality factor.

```
Accept: application/json;q=0.8,application/xml;q=0.5
```

If we don't specify any Accept header, then by default server choose JSON formatter.

When the response is being sent to the client in the requested format, notice that the `Content-Type` header of the response is set to the appropriate value. For example, if the client has requested `application/xml`, the server send the data in XML format and also sets the `Content-Type=application/xml`.

The formatters are used by the server for both request and response messages. When the client sends a request to the server, we set the Content-Type header to the appropriate value to let the server know the format of the data that we are sending.

For example, if the client is sending JSON data, the Content-Type header is set to `application/json`. The server knows it is dealing with JSON data, so it uses JSON formatter to convert JSON data to .NET Type. Similarly when a response is being sent from the server to the client, depending on the Accept header value, the appropriate formatter is used to convert .NET type to JSON, XML etc.

Example of different types of response format:

application/json:

```
{
  "Email": "sample string 1",
  "HasRegistered": true,
  "LoginProvider": "sample string 3"
}
```

application/xml:

```
<UserInfoViewModel xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/WebApiDemo.Models">
  <Email>sample string 1</Email>
  <HasRegistered>true</HasRegistered>
  <LoginProvider>sample string 3</LoginProvider>
</UserInfoViewModel>
```

Modern web based applications can provide data in various languages and formats. So, if we develop our API to cover global users across the world, then Content Negotiation is relevant.

Content Negotiation in Web API

Understanding the concept

To understand content negotiation in Web API, it is important to understand the term `Resource`.

On the web, any information that we can access can be referred as `HTTP resource`. There is a tremendous amount of material to view on the web which has different content type such as html documents, images, video, audio, executable files, spreadsheets etc. We can get any resource by making an http request to the resource uri. The http response for the request, returns the resource and also specifies the content type, which is `also known as media type`.

In order to access any resource, client can make http request by providing specific resource uri and the http verbs. However, in addition to this, client can also specify the accept-type which is the format of the content the user is looking for. The “accept-type” can be defined in the http request headers as the `“accept”` header.

The server then checks the “accept” header from the requests and returns the response in the specified format, if available. Please note that the server can only return the response in the requested representation **if it is available**. If the requested representation is not available then it returns the resource in default representation. That is the reason it is called content negotiation.

A practical example

As an example, assume that you are making a request to http://example.com/customer/1 to get the information of customer with the id 1. If you don’t specify the “accept” header in the request, the server will return the default representation of this resource.

Assume that the server can return the customer information in `json and xml both`. Now, it is on the client to specify the required format of the customer information in the “accept” header in the request. The value of the `“accept”` header can be `“application/json”` for json representation, or `“text/xml”` for xml representation. The server will then return the response as per the requested format.

If the requested format is “text/html” which is not supported by this host (as in this example), then it will *`simply return the resource in the default format`*. The http response contains a header `“content-type”` which tells the client about the format of the resource.

Please note that even in the case when the requested representation of the resource is not available, the default representation of the resource is still returned.

That is why it is referred as content negotiation.

> The client negotiates the representation of the response, however, if it is not available then gets the default one.

How to configure in Web API

In Web API, content negotiation can be configured in the `WebAPIConfig` class as

```
config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));
```

You can also override the default content negotiation in Web API by implementing IContentNegotiator interface and its Negotiate method, and then setup this in the Web API request pipe line, in the WebAPI.config file as below:

```
GlobalConfiguration.Configuration.Services.Replace(typeof(IContentNegotiator), new
CustomContentNegotiator());
```

Following is a sample implemantation of Negotiate method.

```
public class CustomContentNegotiator : DefaultContentNegotiator
    {
        public override ContentNegotiationResult Negotiate(Type type, HttpRequestMessage
request, IEnumerable<MediaTypeFormatter> formatters)
        {
            var result = new ContentNegotiationResult(new JsonMediaTypeFormatter(), new
MediaTypeHeaderValue("application/json"));
            return result;
        }
```

Read ASP.NET Web API Content Negotiation online: https://riptutorial.com/asp-net-web-api/topic/7654/asp-net-web-api-content-negotiation

Chapter 3: ASP.NET WEB API CORS Enabling

Examples

Enabling CORS for WebAPI 2

```
// Global.asax.cs calls this method at application start
public static void Register(HttpConfiguration config)
{
    // New code
    config.EnableCors();
}

//Enabling CORS for controller after the above registration
[EnableCors(origins: "http://example.com", headers: "*", methods: "*")]
public class TestController : ApiController
{
    // Controller methods not shown...
}
```

Enabling CORS globally for Web API 2

```
public static void Register(HttpConfiguration config)
{
    var corsAttr = new EnableCorsAttribute("http://example.com", "*", "*");
    config.EnableCors(corsAttr);
}
```

Enabling CORS in Asp.Net 5 for all domains and methods

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(o => o.AddPolicy("MyPolicy", builder =>
    {
        builder.AllowAnyOrigin()
               .AllowAnyMethod()
               .AllowAnyHeader();
    }));

    // ...
}

public void Configure(IApplicationBuilder app)
{
    app.UseCors("MyPolicy");

    // ...
}
```

Enabling CORS in Asp.Net 5 for specific domains and methods

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddCors();
    services.ConfigureCors(options =>
         options.AddPolicy("AllowSpecific", p => p.WithOrigins("http://localhost:1233")
                                                  .WithMethods("GET")
                                                  .WithHeaders("name")));
}
```

Configure CORS for WebAPI 2 with Windows Authentication

The following server-side configuration allows CORS request to work along with Windows Authentication (no anonymous must be enabled in IIS).

web.config - allow unauthenticated (anonymous) preflight requests (OPTIONS)

```
<system.web>
    <authentication mode="Windows" />
    <authorization>
        <allow verbs="OPTIONS" users="*"/>
        <deny users="?" />
    </authorization>
</system.web>
```

global.asax.cs - properly reply with headers that allow caller from another domain to receive data

```
protected void Application_AuthenticateRequest(object sender, EventArgs e)
{
    if (Context.Request.HttpMethod == "OPTIONS")
    {
        if (Context.Request.Headers["Origin"] != null)
            Context.Response.AddHeader("Access-Control-Allow-Origin",
Context.Request.Headers["Origin"]);

        Context.Response.AddHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With,
Content-Type, Accept, MaxDataServiceVersion");
        Context.Response.AddHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE,
OPTIONS");
        Context.Response.AddHeader("Access-Control-Allow-Credentials", "true");

        Response.End();
    }
}
```

CORS enabling

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // all requests are enabled in this example. SupportsCredentials must be here to allow
authenticated requests
        var corsAttr = new EnableCorsAttribute("*", "*", "*") { SupportsCredentials = true };
        config.EnableCors(corsAttr);
    }
```

```
}

protected void Application_Start()
{
    GlobalConfiguration.Configure(WebApiConfig.Register);
}
```

Properly send authenticated request from jQuery against Web API 2 endpoint

The following example shows how to properly construct both GET and POST requests against Web API 2 (CORS must be configured server-side, if sent from another domain):

```
<script type="text/javascript" src="https://code.jquery.com/jquery-3.1.1.js"></script>
CORS with Windows Authentication test
<script type="text/javascript">

    // GET
    $.ajax({
        url: "endpoint url here",
        type: "GET",
        dataType: "json",
            xhrFields: {
            withCredentials: true
        }
    })
    .done(function (data, extra) {
      alert("GET result" + JSON.stringify(data));
    })
    .fail(function(data, extra) {
    });

    //POST
    $.ajax({
        url: "url here",
        type: "POST",
        contentType: 'application/json; charset=utf-8',
        data: JSON.stringify({testProp: "test value"}),
        xhrFields: {
            withCredentials: true
        },
        success: function(data) {
            alert("POST success - " + JSON.stringify(data));
        }
    })
    .fail(function(data) {
        alert("Post error: " + JSON.stringify(data.data));
    });

</script>
```

Server-side code:

```
    [System.Web.Http.HttpGet]
    [System.Web.Http.Route("GetRequestUsername")]
    public HttpResponseMessage GetRequestUsername()
    {
        var ret = Request.CreateResponse(
```

```
            HttpStatusCode.OK,
            new { Username = SecurityService.GetUsername() });
        return ret;
    }

    [System.Web.Http.HttpPost]
    [System.Web.Http.Route("TestPost")]
    public HttpResponseMessage TestPost([FromBody] object jsonData)
    {
        var ret = Request.CreateResponse(
            HttpStatusCode.OK,
            new { Username = SecurityService.GetUsername() });
        return ret;
    }
```

Properly send authenticated request from AngularJS against Web API 2 endpoint

```
<script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.6.1/angular.js"></script>
CORS with Windows Authentication test (Angular)
<script type="text/javascript">

    var app = angular.module('myApp', []);
    app.controller('myCtrl', function($http) {

        $http(
            {
                method: 'GET',
                url: 'url here',
                withCredentials: true,
            }
        )
        .then(function(data) {
            alert("Get result = " + JSON.stringify(data.data));
        },
        function(data, extra) {
            alert("Get failed: " + JSON.stringify(data.data));
        });

        $http(
            {
                method: 'POST',
                url: "url here",
                withCredentials: true,
                data: { url: "some url", message: "some message", type: "some type"}
            }
        )
        .then(function(data) {
            alert("POST success - " + JSON.stringify(data.data));
        },
        function(data) {
            alert("POST failed: " + JSON.stringify(data.data));
        });
    });

</script>

<div ng-app="myApp" ng-controller="myCtrl">
```

```
</div>
```

Read ASP.NET WEB API CORS Enabling online: https://riptutorial.com/asp-net-web-api/topic/4185/asp-net-web-api-cors-enabling

Chapter 4: ASP.NET Web API MediaTypeFormatter

Examples

MediaTypeFormatter Basic Information

`MediaTypeFormatter` is an abstract class from which `JsonMediaTypeFormatter` and `XmlMediaTypeFormatter` classes inherit from. Here, `JsonMediaTypeFormatter` class handles JSON objects and `XmlMediaTypeFormatter` class handles XML objects.

Return only JSON irrespective of the Accept Header value:

To return only JSON objects in the response of the request weather Accept Header value of request if `application/json` or `application/xml` write the following line in the `Register` method of `WebApiConfig` class.

```
config.Formatters.Remove(config.Formatters.XmlFormatter);
```

Here, `config` is a object of `HttpConfiguration` class. This line of code completely removes `XmlFormatter` which forces ASP.NET Web API to always return JSON irrespective of the Accept header value in the client request. Use this technique when you want your service to support only JSON and not XML.

Return only XML irrespective of the Accept Header value:

To return only XML objects in the response of the request weather Accept Header value of request if `application/json` or `application/xml` write the following line in the `Register` method of `WebApiConfig` class.

```
config.Formatters.Remove(config.Formatters.JsonFormatter);
```

Here, `config` is a object of `HttpConfiguration` class as described above. This line of code completely removes `JsonFormatter` which forces ASP.NET Web API to always return XML irrespective of the Accept header value in the client request. Use this technique when you want your service to support only XML and not JSON.

Return JSON instead of XML:

1. When a request is issued from the browser, the web API service should return JSON instead of XML.
2. When a request is issued from a tool like fiddler the Accept header value should be respected. This means if the Accept header is set to application/xml the service should return XML and if it is set to application/json the service should return JSON.

Method 1:

Include the following line in `Register` method of `WebApiConfig` class.

```
config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
MediaTypeHeaderValue("text/html"));
```

This instruct ASP.NET Web API to use `JsonFormatter` when request is made for `text/html` which is the default for most browsers. The problem with this approach is that `Content-Type` header of the response is set to `text/html` which is misleading.

Method 2:

Use Custom formatters. Make a class which is derived from `JsonMediaTypeFormatter` class and implement `SetDefaultContentHeaders` method.

Here is the example of custom JSON formatter class which returns JSON format in response.

```
public class CustomJsonFormatter : JsonMediaTypeFormatter
{
    public CustomJsonFormatter()
    {
        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));
    }

    public override void SetDefaultContentHeaders(Type type, HttpContentHeaders headers,
MediaTypeHeaderValue mediaType)
    {
        base.SetDefaultContentHeaders(type, headers, mediaType);
        headers.ContentType = new MediaTypeHeaderValue("application/json");
    }
}
```

And this is the example of Custom Media type formatter which returns CSV format in response.

```
public class CSVMediaTypeFormatter : MediaTypeFormatter {

    public CSVMediaTypeFormatter()
    {
        SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));
    }

    public CSVMediaTypeFormatter(MediaTypeMapping mediaTypeMapping) : this()
    {
        MediaTypeMappings.Add(mediaTypeMapping);
    }

    public CSVMediaTypeFormatter(IEnumerable<MediaTypeMapping> mediaTypeMappings) : this()
    {
        foreach (var mediaTypeMapping in mediaTypeMappings)
        {
            MediaTypeMappings.Add(mediaTypeMapping);
        }
    }
}
```

After, implementing the custom formatter class register it in `Register` method of `WebApiConfig` class.

```
config.Formatters.Add(new CustomJsonFormatter());
```

Now, according to your formatter you will get response and `Content-Type` from the server.

Read ASP.NET Web API MediaTypeFormatter online: https://riptutorial.com/asp-net-web-api/topic/7673/asp-net-web-api-mediatypeformatter

Chapter 5: Attribute Routing in WebAPI

Introduction

As the name suggests, this uses attributes to route. This gives the user more control over the URI's in the WebAPI. For example, you can describe hierarchies of the resource. However, the earlier 'Conventional Routing' is fully supported. Users can have a mixture of both too.

Syntax

- [RoutePrefix("api/books")] - for controller class
- [Route("getById")] - for actions
- [Route("~/api/authors/{authorId:int}/books")] - for overriding route prefix

Parameters

Parameter Name	Details
RoutePrefix	attribute to the controller class. all common url prefixes in actions are clubbed here. takes string as input
Route	attribute to the controller actions. each action will have route assosciated with(not necessarily)
Route("~/api/")	this overrides the Route Prefix

Remarks

Currently, Attribute Routes doesn't have `Controller specific Message Handlers`. As there is no way to specify Which handler to execute for which route at the time of declaration. This is possible in `Conventional Routing`.

Examples

Basic Attribute Routing

Simply add an attribute to the controller action

```
[Route("product/{productId}/customer")]
public IQueryable<Product> GetProductsByCustomer(int productId)
{
    //action code goes here
}
```

this will be queried as `/product/1/customer` and `productId=1` will be sent to the controller action.

Make sure the one within '{ }' and the action parameter are same. `productId` in this case.

before using this, you have to specify that you are using Attribute Routing by:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();
    }
}
```

Route Prefix Attribute

In cases where you need a common portion of the route for all routes within a controller, `RoutePrefix` attribute is used.

In the below example, api/students part of the code is common and so we can define `RoutePrefix` and avoid using it repeatedly.

```
[RoutePrefix("api/students")]
public class StudentController : ApiController
{
[Route("")]
public IEnumerable<Student> Get()
{
    //action code goes here
}

[Route("{id:int}")]
public Student Get(int id)
{
    //action code goes here
}

[Route("")]
public HttpResponseMessage Post(Student student)
{
    //action code goes here
}

}
```

Read Attribute Routing in WebAPI online: https://riptutorial.com/asp-net-web-api/topic/9440/attribute-routing-in-webapi

Chapter 6: Caching

Remarks

Caching is the process of storing data somewhere for the future requests, in our case we can avoid the unwanted hit to database to get the data if we cache the data somewhere, this way we can make sure that the data is served in a faster manner.

Examples

System.Runtime.Caching (MemoryCache)

Import the namespace System.Runtime.Caching(Make sure that you have added System.Runtime.Caching DLL to your project reference).

Create an instance of MemoryCache class.

```
MemoryCache memCache = MemoryCache.Default;
```

Add values to MemoryCache

```
public IQueryable<tblTag> GettblTags()
        {
            var ca = db.tblTags;
            memCache.Add("tag", ca, DateTimeOffset.UtcNow.AddMinutes(5));
            return db.tblTags;
        }
```

Here “tag” is my key and “ca” is my values and DateTimeOffset.UtcNow.AddMinutes(5) is for setting the cache for five minutes from now.

Get values from MemoryCache

```
var res = memCache.Get("tag");
            if (res != null)
            {
                return res;
            }
            else {
                var ca = db.tblTags;
                memCache.Add("tag", ca, DateTimeOffset.UtcNow.AddMinutes(5));
                return db.tblTags;
            }
```

We will get the cache values in the variable res, remember this values will be there only for five minutes. You can always change that as per need. If the value is not null, we will just return it and do the manipulation and if it is null we will go ahead and fetch the data from database and add the value to cache.

Remove values from MemoryCache

```
            if (memCache.Contains("tag"))
            {
                memCache.Remove("tag");
            }
```

Read Caching online: https://riptutorial.com/asp-net-web-api/topic/7983/caching

Chapter 7: Configure a Web API application to respond with pretty/formatted JSON data by defualt

Examples

Default JSON formatting: Efficiency at the cost of readability

Lets say you have a simple ApiController like this:

```
    [HttpGet]
    [Route("test")]
    public dynamic Test()
    {
        dynamic obj = new ExpandoObject();
        obj.prop1 = "some string";
        obj.prop2 = 11;
        obj.prop3 = "another string";

        return obj;
    }
```

The resulting JSON representation of this object will look like this:

```
{"prop1":"some string","prop2":11,"prop3":"another string"}
```

This is probably fine for simple responses like this, but imagine if you have a large/complex object sent as the response:

```
"response": { "version": "0.1", "termsofService":
"http://www.wunderground.com/weather/api/d/terms.html", "features": { "history": 1 } },
"history": { "date": { "pretty": "July 16, 2016", "year": "2016", "mon": "07", "mday": "16",
"hour": "12", "min": "00", "tzname": "America/Indianapolis" }, "utcdate": { "pretty": "July
16, 2016", "year": "2016", "mon": "07", "mday": "16", "hour": "16", "min": "00", "tzname":
"UTC" }, "observations": [{ "date": { "pretty": "12:15 AM EDT on July 16, 2016", "year":
"2016", "mon": "07", "mday": "16", "hour": "00", "min": "15", "tzname": "America/Indianapolis"
}, "utcdate": { "pretty": "4:15 AM GMT on July 16, 2016", "year": "2016", "mon": "07", "mday":
"16", "hour": "04", "min": "15", "tzname": "UTC" }, "tempm": "18.2", "tempi": "64.8",
"dewptm": "16.4", "dewpti": "61.5", "hum": "89", "wspdm": "9.3", "wspdi": "5.8", "wgustm": "-
9999.0", "wgusti": "-9999.0", "wdird": "20", "wdire": "NNE", "vism": "16.1", "visi": "10.0",
"pressurem": "1018.2", "pressurei": "30.07", "windchillm": "-999", "windchilli": "-999",
"heatindexm": "-9999", "heatindexi": "-9999", "precipm": "-9999.00", "precipi": "-9999.00",
"conds": "Clear", "icon": "clear", "fog": "0", "rain": "0", "snow": "0", "hail": "0",
"thunder": "0", "tornado": "0", "metar": "METAR KTYQ 160415Z AUTO 02005KT 10SM CLR 18/16 A3007
RMK AO2 T01820164" }, { "date": { "pretty": "12:35 AM EDT on July 16, 2016", "year": "2016",
"mon": "07", "mday": "16", "hour": "00", "min": "35", "tzname": "America/Indianapolis" },
"utcdate": { "pretty": "4:35 AM GMT on July 16, 2016", "year": "2016", "mon": "07", "mday":
"16", "hour": "04", "min": "35", "tzname": "UTC" }, "tempm": "17.7", "tempi": "63.9",
"dewptm": "16.3", "dewpti": "61.3", "hum": "91", "wspdm": "7.4", "wspdi": "4.6", "wgustm": "-
9999.0", "wgusti": "-9999.0", "wdird": "10", "wdire": "North", "vism": "16.1", "visi": "10.0",
```

```
"pressurem": "1018.2", "pressurei": "30.07", "windchillm": "-999", "windchilli": "-999",
"heatindexm": "-9999", "heatindexi": "-9999", "precipm": "-9999.00", "precipi": "-9999.00",
"conds": "Clear", "icon": "clear", "fog": "0", "rain": "0", "snow": "0", "hail": "0",
"thunder": "0", "tornado": "0", "metar": "METAR KTYQ 160435Z AUTO 01004KT 10SM CLR 18/16 A3007
RMK AO2 T01770163" } } }
```

That isn't what you would consider highly readable data. This is easily solved by setting setting a single property on the default JsonFormatter in App_Start/ApiConfig.cs:

```
    // Either one of these will format your JSON in a readable format.  Setting both provides
no additional benefit.
    config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
MediaTypeHeaderValue("text/html"));
    // OR
    config.Formatters.JsonFormatter.Indent = true;
```

Read Configure a Web API application to respond with pretty/formatted JSON data by defualt online: https://riptutorial.com/asp-net-web-api/topic/6682/configure-a-web-api-application-to-respond-with-pretty-formatted-json-data-by-defualt

Chapter 8: Creating A Custom ActionFilterAttribute

Introduction

Action Filters Attributes are a part of the ASP .NET Framework that I find useful to help follow the DRY principle. You can replace several lines of common logic with one simple declarative tag. The framework provides several useful Action Filter Attributes by default, such as the Authorize and Handle Error Attributes. This guide is intended to show you how to create your own custom attribute.

Examples

EnsurePresenseOfAttribute

This is an example of an attribute that I created to validate that required parameters have been assigned in the request object receive in a POST route. I decided on this approach because the standard ModelState.IsValid approach was not valid. This is because the required attributes vary based on what action is being called.

// ATTRIBUTE ONLY VALID FOR METHODS [AttributeUsage(AttributeTargets.Method)] // INHERIT ActionFilterAttribute public class EnsurePresencesOfAttribute : ActionFilterAttribute { // ReSharper disable once InconsistentNaming public string required { get; set; }

```
// VALIDATE REQUIRED ATTRIBUTES
// FOR NON-ASYNC REQUESTS
public override void OnActionExecuting(HttpActionContext context)
{
    Dictionary<string, object> model = context.ActionArguments;
    var serialstring = JsonConvert.SerializeObject(model);
    foreach (var requirement in required.Split(','))
    {
        if (serialstring.Contains($"{requirement}\":null"))
        {
            ValueError(context, requirement);
            return;
        }
    }
    base.OnActionExecuting(context);
}

// VALIDATE THE REQUIRED ATTRIBUTES ARE PRESENT
// FOR ASYNC REQUESTS
public override Task OnActionExecutingAsync(HttpActionContext context, CancellationToken
token)
{
    Dictionary<string, object> model = context.ActionArguments;
    var serialstring = JsonConvert.SerializeObject(model);
    foreach (var requirement in required.Split(','))
```

```
    {
        if (serialstring.Contains($"{requirement}\":null"))
        {
            ValueError(context, requirement);
            return Task.FromResult(0);
        }
    }
    return base.OnActionExecutingAsync(context, token);
}

// LOG ERROR AND RETURN AND SET ERROR RESPONSE
private static void ValueError(HttpActionContext context, string requirement)
{
    var action = context.ActionDescriptor.ActionName;
    AppUtils.LogError($"{action} Failed : Missing Required Attribute {requirement}. ");
    using (var controller = new BaseApiController { Request = new HttpRequestMessage() })
    {
        controller.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey, new
HttpConfiguration());
        context.Response = controller.InvalidInputResponse();
    }
}
```

Controller Before EnsuresPresenseOf Attribute

```
        [HttpPost]
        [Route("api/Fitbit/Activity/Stats")]
        public async Task<HttpResponseMessage> ActivityStats(FitbitRequestDTO request)
        {
            if (string.IsNullOrEmpty(request.PatientId) ||
string.IsNullOrEmpty(request.DeviceId))
                return InvalidInputResponse();
            try
            {
                var tokenErrorResponse = await EnsureToken(request);
                if (tokenErrorResponse != null)
                    return tokenErrorResponse;
                var client = GetFitbitClient();
                var stats = await client.GetActivitiesStatsAsync();
                return OkResponse(stats);
            }
            catch (Exception e)
            {
                const string function = " ActivityStats ";
                AppUtils.LogException(function, e);
                return SystemErrorResponse(function, e);
            }
        }
```

Update Controller

```
        public async Task<HttpResponseMessage> ActivityStats(FitbitRequestDTO request)
        {
            try
            {
                var tokenErrorResponse = await EnsureToken(request);
                if (tokenErrorResponse != null)
```

```
                return tokenErrorResponse;
            var client = GetFitbitClient();
            var stats = await client.GetActivitiesStatsAsync();
            return OkResponse(stats);
        }
        catch (Exception e)
        {
            const string function = " ActivityStats ";
            AppUtils.LogException(function, e);
            return SystemErrorResponse(function, e);
        }
    }
```

Read Creating A Custom ActionFilterAttribute online: https://riptutorial.com/asp-net-web-api/topic/8989/creating-a-custom-actionfilterattribute

Chapter 9: OData with Asp.net Web API

Examples

Install the OData Packages

From the Tools menu, select NuGet Package Manager > Package Manager Console. In the Package Manager Console window, type:

```
Install-Package Microsoft.AspNet.Odata
```

This command installs the latest OData NuGet packages.

Enable Entity Framework

For this tutorial, we'll use Entity Framework (EF) Code First to create the back-end database.

> Web API OData does not require EF. Use any data-access layer that can translate database entities into models.

First, install the NuGet package for EF. From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the Package Manager Console window, type:

```
Install-Package EntityFramework
```

Open the Web.config file, and add the following section inside the **configuration** element, after the **configSections** element.

```
<configuration>
  <configSections>
    <!-- ... -->
  </configSections>

  <!-- Add this: -->
  <connectionStrings>
    <add name="ProductsContext" connectionString="Data Source=(localdb)\v11.0;
        Initial Catalog=ProductsContext; Integrated Security=True;
MultipleActiveResultSets=True;
        AttachDbFilename=|DataDirectory|ProductsContext.mdf"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
```

This setting adds a connection string for a LocalDB database. This database will be used when you run the app locally.

Next, add a class named **ProductsContext** to the Models folder:

```
using System.Data.Entity;
```

```
namespace ProductService.Models
{
    public class ProductsContext : DbContext
    {
        public ProductsContext()
                : base("name=ProductsContext")
        {
        }
        public DbSet<Product> Products { get; set; }
    }
}
```

In the constructor, **"name=ProductsContext"** gives the name of the connection string.

Configure the OData Endpoint

Open the file App_Start/WebApiConfig.cs. Add the following **using** statements:

```
using ProductService.Models;
using System.Web.OData.Builder;
using System.Web.OData.Extensions;
```

Then add the following code to the **Register** method:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // New code:
        ODataModelBuilder builder = new ODataConventionModelBuilder();
        builder.EntitySet<Product>("Products");
        config.MapODataServiceRoute(
            routeName: "ODataRoute",
            routePrefix: null,
            model: builder.GetEdmModel());
    }
}
```

This code does two things:

- Creates an Entity Data Model (EDM).
- Adds a route.

An EDM is an abstract model of the data. The EDM is used to create the service metadata document. The **ODataConventionModelBuilder** class creates an EDM by using default naming conventions. This approach requires the least code. If you want more control over the EDM, you can use the **ODataModelBuilder** class to create the EDM by adding properties, keys, and navigation properties explicitly.

A route tells Web API how to route HTTP requests to the endpoint. To create an OData v4 route, call the **MapODataServiceRoute** extension method.

If your application has multiple OData endpoints, create a separate route for each. Give each

route a unique route name and prefix.

Add the OData Controller

A controller is a class that handles HTTP requests. You create a separate controller for each entity set in your OData service. In this tutorial, you will create one controller, for the Product entity.

In Solution Explorer, right-click the Controllers folder and select **Add** > **Class**. Name the class ProductsController.

> The version of this tutorial for OData v3 uses the Add Controller scaffolding. Currently, there is no scaffolding for OData v4.

Replace the boilerplate code in ProductsController.cs with the following.

```
using ProductService.Models;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.OData;
namespace ProductService.Controllers
{
    public class ProductsController : ODataController
    {
        ProductsContext db = new ProductsContext();
        private bool ProductExists(int key)
        {
            return db.Products.Any(p => p.Id == key);
        }
        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

The controller uses the **ProductsContext** class to access the database using EF. Notice that the controller overrides the **Dispose** method to dispose of the **ProductsContext**.

This is the starting point for the controller. Next, we'll add methods for all of the CRUD operations.

Performing CRUD on the Entity Set

Querying the Entity Set

Add the following methods to **ProductsController**.

```
[EnableQuery]
```

```
public IQueryable<Product> Get()
{
    return db.Products;
}

[EnableQuery]
public SingleResult<Product> Get([FromODataUri] int key)
{
    IQueryable<Product> result = db.Products.Where(p => p.Id == key);
    return SingleResult.Create(result);
}
```

The parameterless version of the Get method returns the entire Products collection. The Get method with a key parameter looks up a product by its key (in this case, the Id property).

The **[EnableQuery]** attribute enables clients to modify the query, by using query options such as $filter, $sort, and $page. For more information, see Supporting OData Query Options.

Adding an Entity to the Entity Set

To enable clients to add a new product to the database, add the following method to **ProductsController**.

```
public async Task<IHttpActionResult> Post(Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    db.Products.Add(product);
    await db.SaveChangesAsync();
    return Created(product);
}
```

Updating an Entity

OData supports two different semantics for updating an entity, PATCH and PUT.

- PATCH performs a partial update. The client specifies just the properties to update.
- PUT replaces the entire entity.

The disadvantage of PUT is that the client must send values for all of the properties in the entity, including values that are not changing. The OData spec states that PATCH is preferred.

In any case, here is the code for both PATCH and PUT methods:

```
public async Task<IHttpActionResult> Patch([FromODataUri] int key, Delta<Product> product)
{
    if (!ModelState.IsValid)
    {
```

```
        return BadRequest(ModelState);
    }
    var entity = await db.Products.FindAsync(key);
    if (entity == null)
    {
        return NotFound();
    }
    product.Patch(entity);
    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
    return Updated(entity);
}

public async Task<IHttpActionResult> Put([FromODataUri] int key, Product update)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    if (key != update.Id)
    {
        return BadRequest();
    }
    db.Entry(update).State = EntityState.Modified;
    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
    return Updated(update);
}
```

In the case of PATCH, the controller uses the **Delta<T>** type to track the changes.

Deleting an Entity

To enable clients to delete a product from the database, add the following method to **ProductsController**.

```
public async Task<IHttpActionResult> Delete([FromODataUri] int key)
{
    var product = await db.Products.FindAsync(key);
    if (product == null)
    {
        return NotFound();
    }
    db.Products.Remove(product);
    await db.SaveChangesAsync();
    return StatusCode(HttpStatusCode.NoContent);
}
```

Read OData with Asp.net Web API online: https://riptutorial.com/asp-net-web-api/topic/6019/odata-with-asp-net-web-api

Chapter 10: Quick Start: Working with JSON

Remarks

Examples to get you up and running quickly (and correctly) with ASP.NET WebAPI

Examples

Return JSON from GET using attributes

1. Setup your formatter and routing in `Register` of (`App_Start/WebApiConfig`)

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        GlobalConfiguration.Configuration.Formatters.Clear();
        GlobalConfiguration.Configuration.Formatters.Add(new JsonMediaTypeFormatter());

        config.MapHttpAttributeRoutes();
    }
}
```

2. Create methods in an `ApiController`

```
public class HelloWorldController : ApiController
{
  [HttpGet]
  [Route("echo/{message}")]
  public IHttpActionResult Echo(string message) {
    return Ok(new{ hello: message });
  }

  [HttpGet]
  [Route("echo/{digits:int}")]
  public IHttpActionResult Echo(int digits) {
    return Ok(new{ hello: digits });

}
```

executing `GET /echo/foo`

```
{
  "hello": "foo"
}
```

executing `GET /echo/1241290805`

```
{
  "hello": 1241290805
}
```

as the routing framework takes the most specific conditions (data type) when choosing a method

Read Quick Start: Working with JSON online: https://riptutorial.com/asp-net-web-api/topic/3851/quick-start--working-with-json

Chapter 11: Web API Url Routing

Examples

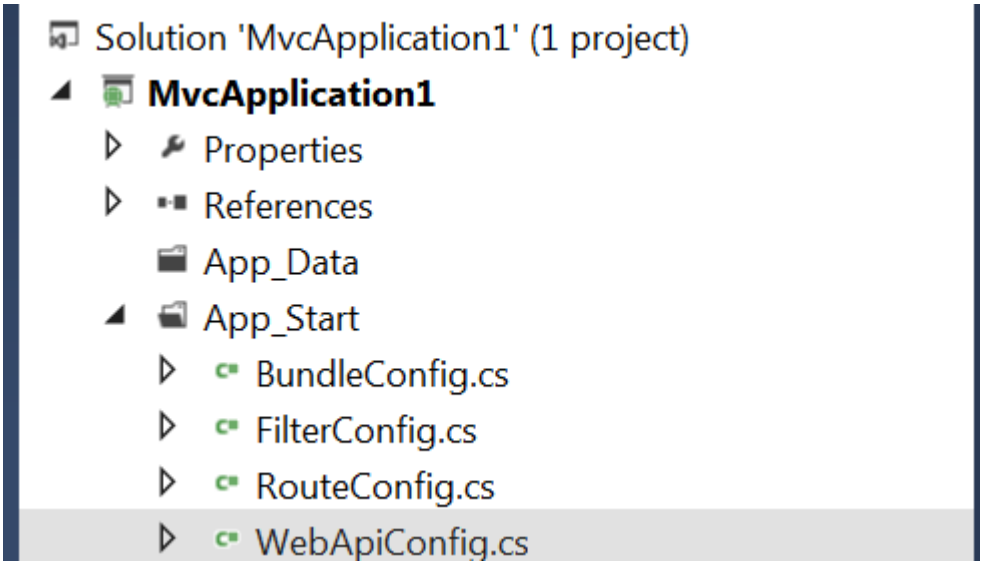
How Routing works in asp.net webapi

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:



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/values
- /api/values/1

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

- /values/1

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 "GetEmployee" or "GetAllEmployees". 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.

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 EmployeeGetEmployee method is mapped to GET requests:

```
 public class EmployeesController : ApiController
    {
        [HttpGet]
        public EmployeeGetEmployee(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 EmployeesController: ApiController
{
    [AcceptVerbs("GET", "HEAD")]
    public Employee GetEmployee (id) { }
}
```

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 EmployeesController: ApiController
{
    [HttpGet]
    public List<Employee> GetAllEmployees();
}
```

In this case, a GET request for “**api/Employees/GetAllEmployees**” would map to the GetAllEmployees method.

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

```
public class EmployeesController : ApiController
{
    [HttpGet]
    [ActionName("ShowAllEmployees")]
    public List<Employee> GetAll(int id);

    [HttpPost]
    [ActionName("ShowAllEmployees")]
    public void GetAll (int id);

}
```

Non-Actions

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

```
 [NonAction]
public string GetValues() { ... }
```

Verb based routing examples.

The same URI for different http methods acts differently. Below is a table depicting the same.

HTTP VERB	URL	DESCRIPTION
GET	/api/students	Returns all students
GET	/api/students/5	Returns details of Student Id =5
POST	/api/students	Add a new student
PUT	/api/students/5	Update student with Id=5
DELETE	/api/students/5	Delete student with Id = 5

Read Web API Url Routing online: https://riptutorial.com/asp-net-web-api/topic/2432/web-api-url-

routing

Credits

S. No	Chapters	Contributors
1	Getting started with asp.net-web-api	Arif, BehrouzMoslem, Community, The_Outsider
2	ASP.NET Web API Content Negotiation	Amit Shahani, Keyur Ramoliya
3	ASP.NET WEB API CORS Enabling	Alexei, Oluwafemi
4	ASP.NET Web API MediaTypeFormatter	Keyur Ramoliya
5	Attribute Routing in WebAPI	Ajay Aradhya, The_Outsider
6	Caching	Sibeesh Venu
7	Configure a Web API application to respond with pretty/formatted JSON data by defualt	Patrick
8	Creating A Custom ActionFilterAttribute	Antarr Byrd, Blanthor
9	OData with Asp.net Web API	Yushell
10	Quick Start: Working with JSON	Brett Veenstra
11	Web API Url Routing	ravindra, riteshmeher, The_Outsider