Asp Dot Net Core

1. Content [Negotiation](#_How_Content_Negotiation)
2. [CROSS Origin](#_CROSS_Origin)
3. [REST Api - Idempotent](#_REST_Api_-)
4. [Middleware](#_Middleware)
5. Introduction
6. Run, Use and Map Methods
7. Add Custom Middleware
8. [HttpClient retry mechanism with .NET Core](#_HttpClient_retry_mechanism)

# How Content Negotiation Works

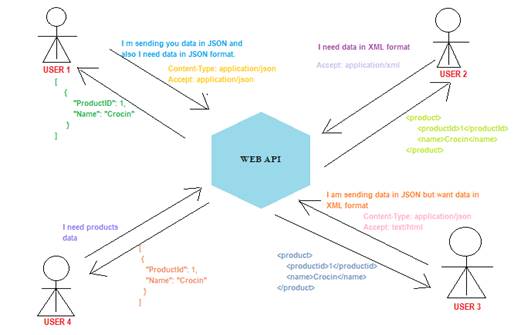
([https://www.dotnettricks.com/learn/webapi/content-negotiation-in-asp-net-web-ap](https://www.dotnettricks.com/learn/webapi/content-negotiation-in-asp-net-web-api)i)

There are two main headers which hold the responsibility of content negotiation

         Content-Type

         Accept

Let’s try to understand them. When a requester send a service request, the CONTENT-TYPE tells responder the format he will receive data whereas the ACCEPT header tells in which format the requester requires the data.



## Default Content Negotiator

In the above pictorial view, there are few points which should be noted down,

         User 2 didn’t mention Content-Type but then received the response in desired format. I.e. XML.

         **Whereas User 4 didn’t mention both Content-Type as well as Accept header. But then received the response. I.e. in JSON format. In short, JSON format is the default content negotiator in web api 2**.

         Also, User 3 requires data in text/html format but receives data in XML format. In short, text/html Accept header sends response in XML format by default.

# CROSS Origin

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the same-origin policy, and prevents a malicious site from reading sensitive data from another site. However, sometimes you might want to let other sites call your web API.

Cross Origin Resource Sharing (CORS) is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others.

What is "same origin"?

Two URLs have the same origin if they have identical schemes, hosts, and ports. (RFC 6454)

**These two URLs have the same origin:**

http://example.com/foo.html

http://example.com/bar.html

**These URLs have different origins than the previous two:**

http://example.net - Different domain

http://example.com:9000/foo.html - Different port

https://example.com/foo.html - Different scheme/Protocol

http://www.example.com/foo.html - Different sub domain

# REST API - Idempotent

In the context of REST APIs, when making multiple identical requests has the same effect as making a single request – then that REST API is called idempotent.

When you design REST APIs, you must realize that API consumers can make mistakes. Users can write client code in such a way that there can be duplicate requests coming to the API.

These duplicate requests may be unintentional as well as intentional some time (e.g. due to timeout or network issues). You have to design fault-tolerant APIs in such a way that duplicate requests do not leave the system unstable.

***"An idempotent HTTP method is an HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same. Idempotence essentially means that the result of a successfully performed request is independent of the number of times it is executed. For example, in arithmetic, adding zero to a number is an idempotent operation."***

## Idempotence with HTTP Methods:

**HTTP POST**

Generally – not necessarily – POST APIs are used to create a new resource on server. So when you invoke the same POST request N times, you will have N new resources on the server. So, POST is not idempotent.

**HTTP GET, HEAD, OPTIONS and TRACE**

GET, HEAD, OPTIONS and TRACE methods NEVER change the resource state on server. They are purely for retrieving the resource representation or meta data at that point of time. So invoking multiple requests will not have any write operation on server, so GET, HEAD, OPTIONS and TRACE are idempotent.

**HTTP PUT**

Generally – not necessarily – PUT APIs are used to update the resource state. If you invoke a PUT API N times, the very first request will update the resource; then rest N-1 requests will just overwrite the same resource state again and again – effectively not changing anything. Hence, PUT is idempotent.

# Middleware

<https://livebook.manning.com/book/asp-net-core-in-action/chapter-3/1>

<https://www.c-sharpcorner.com/article/overview-of-middleware-in-asp-net-core/>

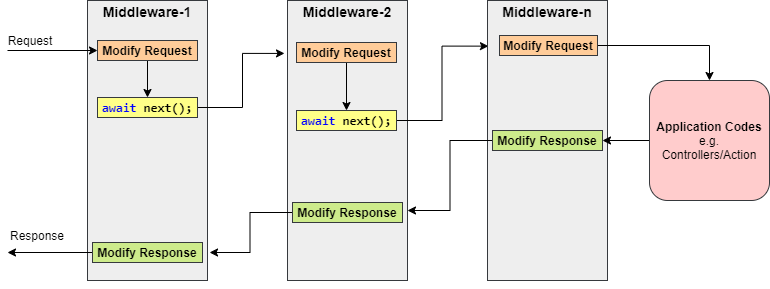
<https://jakeydocs.readthedocs.io/en/latest/migration/http-modules.html#loading-middleware-options-using-the-options-pattern>

Middleware is a piece of code in an application pipeline used to handle requests and responses.

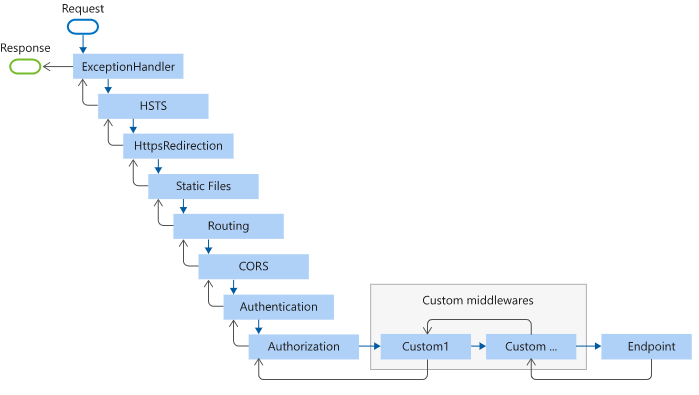
For example, we may have a middleware component to authenticate a user, another piece of middleware to handle errors, and another middleware to serve static files such as JavaScript files, CSS files, images, etc.

Middleware can be built-in as part of the .NET Core framework, added via NuGet packages, or can be custom middleware. These middleware components are configured as part of the application startup class in the configure method. Configure methods set up a request processing pipeline for an ASP.NET Core application. It consists of a sequence of request delegates called one after the other.

The following figure illustrates how a request process through middleware components.



## Middleware Ordering

The following diagram shows the complete request processing pipeline for ASP.NET Core MVC and Razor Pages apps. You can see how, in a typical app, existing middlewares are ordered and where custom middlewares are added. You have full control over how to reorder existing middlewares or inject new custom middlewares as necessary for your scenarios.

## Run, Use and Map Methods

### app.Run ()

This middleware component may expose Run [Middleware] methods that are executed at the end of the pipeline. Generally, this acts as a terminal middleware and is added at the end of the request pipeline, as it cannot call the next middleware.

### app.Use ()

This is used to configure multiple middleware. Unlike app.Run (), we can include the next parameter into it, which calls the next request delegate in the pipeline. We can also short-circuit (terminate) the pipeline by not calling the next parameter.

### app. Map ()

These extensions are used as a convention for branching the pipeline. The map branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

app.Map("/m1", HandleMapOne);

app.Map("/m2", appMap => {

appMap.Run(async context =>

{

await context.Response.WriteAsync("Hello from 2nd app.Map()");

});

});

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello from app.Run()");

});

}

private static void HandleMapOne(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Hello from 1st app.Map()");

});

}

## Create Custom Middleware

Middleware is generally encapsulated in a class and exposed with an extension method. The custom middleware can be built with a class with InvokeAsync () method and RequestDelegate type parameter in the constructor. RequestDelegate type is required in order to execute the next middleware in a sequence.

Let’s consider an example where we need to create custom middleware to log a request URL in a web application.

**Middleware class:**

using Microsoft.AspNetCore.Http;

using System.Globalization;

using System.Threading.Tasks;

namespace Culture

{

public class RequestCultureMiddleware

{

private readonly RequestDelegate \_next;

public RequestCultureMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

var cultureQuery = context.Request.Query["culture"];

if (!string.IsNullOrWhiteSpace(cultureQuery))

{

var culture = new CultureInfo(cultureQuery);

CultureInfo.CurrentCulture = culture;

CultureInfo.CurrentUICulture = culture;

}

// Call the next delegate/middleware in the pipeline

await \_next(context);

}

}

}

**Middleware Extension class:**

using Microsoft.AspNetCore.Builder;

namespace Culture

{

public static class RequestCultureMiddlewareExtensions

{

public static IApplicationBuilder UseRequestCulture(

this IApplicationBuilder builder)

{

return builder.UseMiddleware<RequestCultureMiddleware>();

}

}

}

**Use Middleware:**

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.UseRequestCulture();

app.Run(async (context) =>

{

await context.Response.WriteAsync(

$"Hello {CultureInfo.CurrentCulture.DisplayName}");

});

}

}

# HttpClient retry mechanism with .NET Core

## [HttpClient retry mechanism with .NET Core, Polly and IHttpClientFactory](https://briancaos.wordpress.com/2020/03/25/httpclient-retry-mechanism-with-net-core-polly-and-ihttpclientfactory/)

Polly is a .NET library that provides resilience and transient-fault handling capabilities. You can implement those capabilities by applying Polly policies such as Retry, Circuit Breaker, Bulkhead Isolation, Timeout, and Fallback. Polly targets .NET Framework 4.x and .NET Standard 1.0, 1.1, and 2.0 (which supports .NET Core and later).

**NuGet Packages:**

* Polly
* Microsoft.Extensions.Http.Polly

**Configure a client with Polly's Retry policy, in Startup**

//ConfigureServices() - Startup.cs

services.AddHttpClient<IBasketService, BasketService>()

.SetHandlerLifetime(TimeSpan.FromMinutes(5)) //Set lifetime to five minutes

.AddPolicyHandler(GetRetryPolicy());

The **AddPolicyHandler()** method is what adds policies to the HttpClient objects you'll use. In this case, it's adding a Polly's policy for Http Retries with exponential backoff.

To have a more modular approach, the Http Retry Policy can be defined in a separate method within the Startup.cs file, as shown in the following code:

static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()

{

return HttpPolicyExtensions

.HandleTransientHttpError()

.OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)

.WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,

retryAttempt)));

}

With Polly, you can define a Retry policy with the number of retries, the exponential backoff configuration, and the actions to take when there's an HTTP exception, such as logging the error. In this case, the policy is configured to try six times with an exponential retry, starting at two seconds.