Asp Dot Net Core

1. [Why Asp.NetCore](#_Why_Asp.Net_Core)
2. Content [Negotiation](#_How_Content_Negotiation)
3. [CROSS Origin](#_CROSS_Origin)
4. [REST Api - Idempotent](#_REST_Api_-)
5. [Middleware](#_Middleware)
6. Introduction
7. Run, Use and Map Methods
8. Add Custom Middleware
9. [Filters](#_Filters)
10. What is XSRF or CSRF? How to prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core?
11. Autofac vs. Microsoft.Extensions.DependencyInjection (Default Di)
12. ASP.NET Web API: Media-Type Formatters
13. Asp.Net Core Http Request Pipeline

# Why Asp.Net Core (Features)

* Cross platform, provide ability to develop and run on Windows, Linux and MacOS.
* Open-source
* Unified Platform to develop Web UI and services.
* Built-in dependency injection.
* Ability to deploy on more than one server like IIS, Kestrel, Nginx, Docker, Apache etc
* Cloud enabled framework, provide support for environment based configuration systems.
* Lightweight, High performance and modern HTTP request pipelines.
* well suited architecture for testability
* Integration of many client-side frameworks like [Angular any version](https://www.qfles.com/interview-question/angular-interview-questions)
* Blazor allow you to use [C# code](https://www.qfles.com/interview-question/c-sharp-interview-questions) in browser with [JavaScript code](https://www.qfles.com/interview-question/java-script-interview-questions).

# How Content Negotiation Works

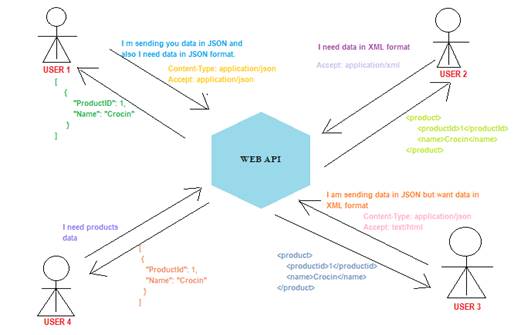
([https://www.dotnettricks.com/learn/webapi/content-negotiation-in-asp-net-web-ap](https://www.dotnettricks.com/learn/webapi/content-negotiation-in-asp-net-web-api)i)

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.

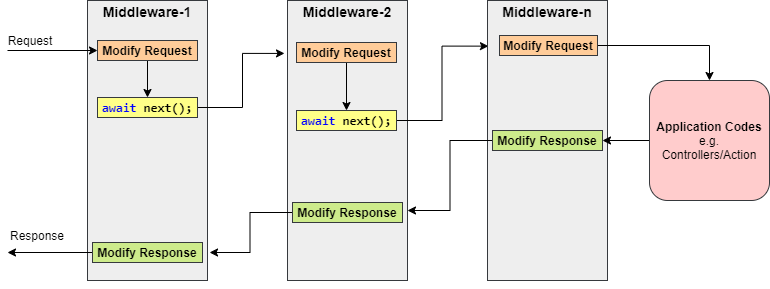
# Middleware

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

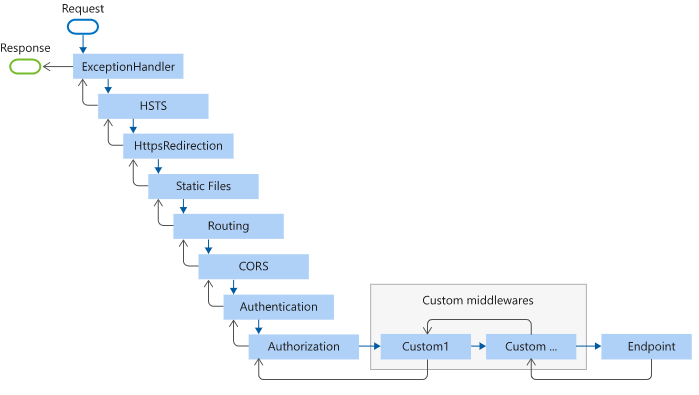
For example, we may have a middleware component to authenticate a user, another piece of middleware to handle errors, and another middleware to serve static files such as JavaScript files, CSS files, images, etc.

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()");

});

}

## Branching the HTTP pipeline in ASP.Net Core

The Map extension methods, i.e. Map and MapWhen, are used for branching the pipeline. While Map is used to branch based on a given request path, MapWhen is used to branch based on the result of a given predicate.

The following code snippet illustrates how the Map method can be used for branching the request pipeline.

**public** **class** **Startup**  
{  
    **private** **static** **void** **MapRequestA**(**IApplicationBuilder** app)  
    {  
        app.**Run**(async context =>  
        {  
            await context.**Response**.**WriteAsync**("This is MapRequestA");  
        });  
    }  
    **private** **static** **void** **MapRequestB**(**IApplicationBuilder** app)  
    {  
        app.**Run**(async context =>  
        {  
            await context.**Response**.**WriteAsync**("This is MapRequestB");  
        });  
    }  
    **private** **static** **void** **MapRequestC**(**IApplicationBuilder** app)  
    {  
        app.**Run**(async context =>  
        {  
            await context.**Response**.**WriteAsync**("This is MapRequestC");  
        });  
    }  
    **public** **void** **Configure**(**IApplicationBuilder** app)  
    {  
        app.**Map**("/mapRequestPathA", **MapRequestA**);  
        app.**Map**("/mapRequestPathB", **MapRequestB**);  
        app.**Map**("/mapRequestPathB", **MapRequestC**);  
        app.**Run**(async context =>  
        {  
            await context.**Response**.**WriteAsync**("Hello World!");  
        });  
    }  
   *//Other methods*  
}

## The UseWhen method in ASP.Net Core

The UseWhen method can be used to execute middleware conditionally. The following code snippet illustrates how the UseWhen method can be used to execute a middleware component if the request path starts with "/api".

app.**UseWhen**(context => context.**Request**.**Path**.**StartsWithSegments**("/api"), applicationBuilder =>  
{  
    applicationBuilder.**UseCustomMiddleware**();  
});

Note that unlike MapWhen, the UseWhen method continues to execute the later middleware irrespective of whether the UseWhen predicate was true or false. Let’s understand this with an example. Consider the following piece of code:

app.**UseMiddlewareA**();  
app.**UseWhen**(context => context.**Request**.**Path**.**StartsWithSegments**("/api"), applicationBuilder =>  
{  
    applicationBuilder.**UseMiddlewareB**();  
});  
app.**UseMiddlewareC**();

## 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}");

});

}

}

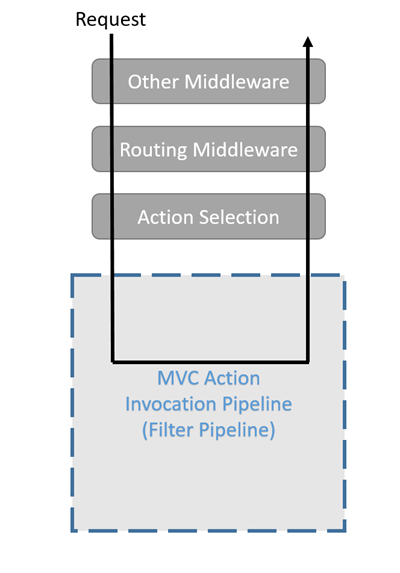
# Filters

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0>

Filters in ASP.NET Core allow code to be run before or after specific stages in the request processing pipeline.

**How Filters works:**

*Filters run within the ASP.NET Core action invocation pipeline, sometimes referred to as the filter pipeline. The filter pipeline runs after ASP.NET Core selects the action to execute.*

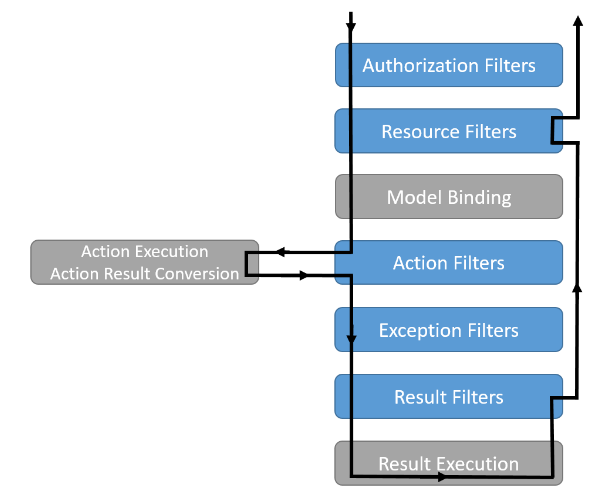


## Filter types

Each filter type is executed at a different stage in the filter pipeline:

* [Authorization filters](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0#authorization-filters) run first and are used to determine whether the user is authorized for the request. Authorization filters short-circuit the pipeline if the request is not authorized.
* [Resource filters](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0#resource-filters):
  + Run after authorization.
  + [OnResourceExecuting](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.filters.iresourcefilter.onresourceexecuting) runs code before the rest of the filter pipeline. For example, OnResourceExecuting runs code before model binding.
  + [OnResourceExecuted](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.filters.iresourcefilter.onresourceexecuted) runs code after the rest of the pipeline has completed.
* [Action filters](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0#action-filters):
  + Run code immediately before and after an action method is called.
  + Can change the arguments passed into an action.
  + Can change the result returned from the action.
  + Are **not** supported in Razor Pages.
* [Exception filters](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0#exception-filters) apply global policies to unhandled exceptions that occur before the response body has been written to.
* [Result filters](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0#result-filters) run code immediately before and after the execution of action results. They run only when the action method has executed successfully. They are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline.



## Action Filter Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions.

Synchronous filters run code before and after their pipeline stage. For example, [OnActionExecuting](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controller.onactionexecuting) is called before the action method is called. [OnActionExecuted](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controller.onactionexecuted) is called after the action method returns.

public class MySampleActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

// Do something before the action executes.

MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);

}

public void OnActionExecuted(ActionExecutedContext context)

{

// Do something after the action executes.

MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);

}

}

Asynchronous filters define an On-Stage-ExecutionAsync method. For example, [OnActionExecutionAsync](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controller.onactionexecutionasync):

public class SampleAsyncActionFilter : IAsyncActionFilter

{

public async Task OnActionExecutionAsync(

ActionExecutingContext context,

ActionExecutionDelegate next)

{

// Do something before the action executes.

// next() calls the action method.

var resultContext = await next();

// resultContext.Result is set.

// Do something after the action executes.

}

}

In the preceding code, the SampleAsyncActionFilter has an [ActionExecutionDelegate](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutiondelegate) (next) that executes the action method.

## Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*:

* Using an attribute on a controller action. Filter attributes cannot be applied to Razor Pages handler methods.
* Using an attribute on a controller or Razor Page.
* Globally for all controllers, actions, and Razor Pages as shown in the following code:

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews(options =>

{

options.Filters.Add(typeof(MySampleActionFilter));

});

}

# Autofac vs. Microsoft.Extensions.DependencyInjection

[.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection) gives you a built-in dependency injection framework. Though the default DI may offer enough functionality, there are a certain limitations like resolving a service with some associated Metadata, Named/Keyed services, Aggregate Services, Multi-tenant support, lazy instantiation, and much more. As the system grows you might need such features, and Autofac gives you all these features.

# ASP.NET Web API Media Type Formatter

As you have seen in the previous section that Web API handles JSON and XML formats based on Accept and Content-Type headers. But, how does it handle these different formats? The answer is: By using Media-Type formatters.

Media type formatters are classes responsible for serializing request/response data so that Web API can understand the request data format and send data in the format which client expects.

Web API includes following built-in media type formatters.

| Media Type Formatter Class | MIME Type | Description |
| --- | --- | --- |
| JsonMediaTypeFormatter | application/json, text/json | Handles JSON format |
| XmlMediaTypeFormatter | application/xml, text/json | Handles XML format |
| FormUrlEncodedMediaTypeFormatter | application/x-www-form-urlencoded | Handles HTML form URL-encoded data |
| JQueryMvcFormUrlEncodedFormatter | application/x-www-form-urlencoded | Handles model-bound HTML form URL-encoded data |

## Formatters In ASP.NET Core 2.0 Web API

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc(options =>

{

options.InputFormatters.Add(new XmlSerializerInputFormatter());

options.OutputFormatters.Add(new XmlSerializerOutputFormatter());

});

}

## How to create a custom formatter class

To create a formatter:

* Derive the class from the appropriate base class. The sample app derives from [TextOutputFormatter](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.textoutputformatter) and [TextInputFormatter](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.textinputformatter).
* Specify valid media types and encodings in the constructor.
* Override the [CanReadType](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.inputformatter.canreadtype) and [CanWriteType](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.outputformatter.canwritetype) methods.
* Override the [ReadRequestBodyAsync](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.inputformatter.readrequestbodyasync) and WriteResponseBodyAsync methods.

The following code shows the VcardOutputFormatter class from the [sample](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/web-api/advanced/custom-formatters/samples)