**What is ASP.NET Core and what are it's benefits**  
ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications.

**Cross Platform :** ASP.NET 4.x applications run only on windows platform, where as ASP.NET Core applications can be developed and run across different platforms like Windows, macOS, or Linux. ASP.NET 4.x applications can be hosted only on IIS, where as ASP.NET Core applications can be hosted on IIS, Apache, Docker, or even self-host in your own process.

**Unified Programming Model for MVC and Web API :** With ASP.NET core, we use the same unified programming model to create MVC style web applications and ASP.NET Web API's. In both the cases, the Controller that we create inherits from the same Controller base class and returns IActionResult. As the name implies IActionResult is an interface and it has got several implementations. ViewResult and JsonResult are just 2 examples of the built-in result types that implement IActionResult interface. So, in the case of a Web API, the controller returns a JsonResult and in the case of an MVC style web application it returns a ViewResult.

**Dependency Injection :** Out of the box, ASP.NET Core has built-in support for dependency injection.

**Testability :** With built-in dependency injection and the unified programming model for creating Web Applications and Web API's, unit testing ASP.NET Core applications is easy.  
  
**Open-source and community-focused :** ASP.NET Core is fully open source and is being actively developed by the .NET team in collaboration with a vast community of open source developers. So, ASP.NET core is continually evolving as the vast community behind it is suggesting ways to improve it and help fix bugs and problems. This means we have a more secure and better quality software.  
  
**Modular HTTP Request Pipeline :** ASP.NET Core Provides Modularity with Middleware Components. In ASP.NET Core, we compose the request and response pipeline using the middleware components. It includes a rich set of built-in middleware components. We can also write our own custom middleware components.

**.Csproj File**

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp2.2</TargetFramework>

<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore.App" />

<PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />

</ItemGroup>

</Project>

**TargetFramework :** As the name implies this element is used to specify the target framework for your application.

To specify a target framework we use something called **Target Framework Moniker (TFM)**. As you can see in the above example, our application targets one frameowrk **netcoreapp2.2**. netcoreapp2.2 is the Moniker for .NET Core 2.2. When we created this application. For .Net 4.5 framewrok – net45

**AspNetCoreHostingModel :** This element specifies how the asp.net core application should be hosted. Should it be hosted **InProcess** or **OutOfProcess**.

The value of InProcess specifies that we want to use in-process hosting model i.e host our asp.net core app inside of the IIS worker process (**w3wp.exe**). The value of OutOfProcess specifies that we want to use  out-of-process hosting model i.e forward web requests to a back-end ASP.NET Core app running the **Kestrel** server.

**PackageReference :** As the name implies, this element is used to include a reference to all the NuGet packages that are installed for your application.

The significance of **Main() method in an ASP.NET Core application**

In an ASP.NET Core project we have a file with name **Program.cs**. In this file we have a **public static void Main()** method

public class Program

{

public static void Main(string[] args)

{

CreateWebHostBuilder(args).Build().Run();

}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>();

}

* An **asp.net core application initially starts as a console application** and the **Main()** method in **Program.cs** file is the entry point.   
    
  So, when the .net runtime executes our application **it looks for this Main() method** and this where the execution starts.  
    
  This **Main() method configures asp.net core** and starts it and at that point it becomes an asp.net core web application.  
    
  So, if you take a look at the **Main()** method, it calls CreateWebHostBuilder() method passing it the command line arguments.  
    
  As you can see, CreateWebHostBuilder() method returns an object that implements IWebHostBuilder.  
    
  On this object, **Build()** method is called which builds a web host that hosts our asp.net core web application.  
    
  On the web host **Run()** method is called, which runs the web application and it begins listening for incoming HTTP requests.  
    
  **CreateWebHostBuilder()** method calls CreateDefaultBuilder() static method of the WebHost class.  
  CreateDefaultBuilder() method creates a web host with pre-configured defaults.
* Setting up the web server
* Loading the host and application configuration from various configuration sources and  Configuring logging
* CreateDefaultBuilder() method **sets up a web host with certain defaults**.

**What is Kestrel**  
Kestrel is a cross-platform web server for ASP.NET Core. It is supported on all platforms and versions that .NET Core supports. It is included by default as internal server in ASP.NET Core. Kestrel can be used, by itself as an edge server i.e Internet-facing web server that can directly process the incoming HTTP requests from the client. In Kestrel, the process used to host the app is **dotnet.exe.**

**InProcess hosting in ASP.NET Core**  
To configure InProcess hosting, add <AspNetCoreHostingModel> element to the app's project file with a value of InProcess  
**<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>**

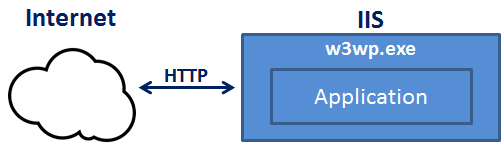
In case of InProcess hosting, CreateDefaultBuilder() method calls UseIIS() method and host the app inside of the IIS worker process (w3wp.exe or iisexpress.exe).

* From a **performance standpoint**, InProcess hosting delivers significantly higher request throughput than OutOfProcess hosting
* In the case of IIS, the process name that executes the app is **w3wp** and in the case of IIS Express it is **iisexpress**

IIS Express is a lightweight, self-contained version of IIS, optimized for application development. We do not use it for production. In production we use IIS.

With **InProcess hosting**, there is only **one web server** i.e the IIS that hosts the asp.net core application.

With **InProcess hosting**, the application is hosted in the IIS worker process (w3wp.exe or iisexpress.exe). With InProcess hosting, there is only one web server and that is the IIS server that hosts our application.



**Out of Process Hosting in ASP.NET Core**   
  
There are 2 ways to configure **Out of Process hosting**   
  
**Option 1 :** Add <AspNetCoreHostingModel> element to the app's project file with a value of OutOfProcess

<AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>

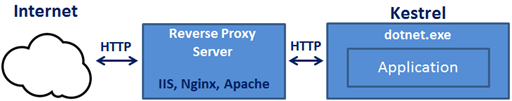
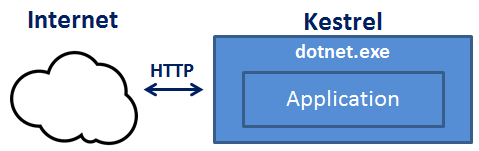
**Option 2 :** The default is OutOfProcess hosting. So if we remove the <AspNetCoreHostingModel> element from the project file, OutOfProcess hosting will be used by default.

**With out of process hosting**

* There are 2 web servers - An **internal web server** and an **external web server**.
* The internal web server is **Kestrel** and the external web server can be IIS, Nginx or Apache.
* So, **we do not have the performance penalty** of proxying requests between internal and external web server.

Depending on how you are running the asp.net core application, the external web server may or may not be used

**Kestrel can be used as the internet facing web server** that process the incoming HTTP requests directly. In this model we are not using an external web server. Only Kestrel is used and it is this server that faces the internet, to directly handle the incoming HTTP requests. When we run the asp.net core application using the .NET core CLI, Kestrel is the only web server that is used to handle and process the incoming HTTP request.   
  
  
  
**Kestrel can also be used in combination with a reverse proxy server**, such as IIS, Nginx, or Apache.



**If Kestrel can be used by itself as a web server, why do we need a reverse proxy server.**

With **Out of Process Hosting**, using a reverse proxy server is a good choice as it provides an additional layer of configuration and security. It might integrate better with the existing infrastructure. It can also be used for load balancing.    
  
So, with a reverse proxy server in place, it receives incoming HTTP requests from the network and forwards them to the Kestrel server for processing. Upon processing the request, the Kestrel server sends the response to the reverse proxy server which then ultimately sends the response to the requested client over the network.

**IIS Express** takes the incoming HTTP request and forwards it to Kestrel for processing. Kestrel process the request and sends the response to IIS Express. IIS Express, in turn sends that response to the browser.

This value along with the value of **AspNetCoreHostingModel** element in the project file, specifies the internal and external web server (reverse proxy server) to launch.

|  |  |  |  |
| --- | --- | --- | --- |
| **commandName** | **AspNetCoreHostingModel** | **Internal Web Server** | **External Web Server** |
| Project | Hosting Setting Ignored | Only one web server is used - Kestrel | |
| IISExpress | InProcess | Only one web server is used - IIS Express | |
| IISExpress | OutOfProcess | Kestrel | IIS Express |
| IIS | InProcess | Only one web server is used - IIS | |
| IIS | OutOfProcess | Kestrel | IIS |

**Startup class**

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{ }

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

}

}

**Startup class does the following 2 very important things**

* **ConfigureServices()** method configures services required by the application
* **Configure()** method sets up the application's request processing pipeline

**launchsettings.json file**

* You will find this file in the **"Properties"** folder in the project root folder.
* The settings in this file are used when we run this ASP.NET core project either from Visual Studio or by using .NET Core CLI.
* This file is **only used on local development machine**. We do not need it for publishing our asp.net core application.
* If there are certain settings that you want your asp.net core application to use when you publish and deploy your app, store them in **appsettings.json file**. We usually store our application configuration settings in this file.
* We can also have **environment specific appsettings.json files**. For example, appsettings.Staging.json for the staging environment. In ASP.NET Core, in addition to appsettings.json file, we also have other configuration sources like Environment variables, User Secrets, Command Line Arguments and even our own custom configuration source.

**Lanuch Profiles in ASP.NET Core**   
  
At the moment, the following are the settings in **launchSettings.json file**

{

"iisSettings": {

"windowsAuthentication": false,

"anonymousAuthentication": true,

"iisExpress": {

"applicationUrl": "http://localhost:48118",

"sslPort": 0

}

},

"profiles": {

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": true,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"EmployeeManagement": {

"commandName": "Project",

"launchBrowser": true,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

},

"applicationUrl": "http://localhost:5000"

}

}

}

Notice, we have 2 profiles - **IIS Express** and **EmployeeManagement**   
  
When we run the project from Visual Studio by pressing **CTRL + F5** or just **F5**, by default, the profile with "commandName": "IISExpress" is used. On the other hand, if we run the project using .NET Core CLI (dotnet run), the profile with the  "commandName": "Project" is used.

**Static Files**

* By default, an asp.net core application will not serve static files
* The default directory for static files is wwwroot and this directory must be in the root project folder

The middleware that we need is UseStaticFiles() middleware to access the static file.

By default, UseStaticFiles() middleware only serves the static files that are in wwwroot folder. We can also server static files outside of the wwwroot folder if you want to.

To be able to serve default page we have to plug in the UseDefaultFiles() middleware in our application's request processing pipeline.

The following are the default files which UseDefaultFiles middleware looks for

index.htm  
index.html  
default.htm  
default.html

// Add Default Files Middleware  
app.UseDefaultFiles();  
// Add Static Files Middleware  
app.UseStaticFiles();

**Please Note** : UseDefaultFiles must be called before UseStaticFiles to serve the default file. UseDefaultFiles is a URL rewriter that doesn't actually serve the file. It simply rewrites the URL to the default document which will then be served by the Static Files Middleware. The URL displayed in the address bar still reflects the root URL and not the rewritten URL.

If you want to use another document like foo.html for example as your default document, you can do so using the following code.

// Specify foo.html as the default document  
DefaultFilesOptions defaultFilesOptions = new DefaultFilesOptions();  
defaultFilesOptions.DefaultFileNames.Clear();  
defaultFilesOptions.DefaultFileNames.Add("foo.html");  
// Add Default Files Middleware  
app.UseDefaultFiles(defaultFilesOptions);  
// Add Static Files Middleware  
app.UseStaticFiles();

**UseFileServer Middleware**

**UseFileServer** combines the functionality of **UseStaticFiles**, **UseDefaultFiles** and **UseDirectoryBrowser** middleware. DirectoryBrowser middleware, enables directory browsing and allows users to see files within a specified directory. We could replace UseStaticFiles and UseDefaultFiles middlewares with UseFileServer Middleware.

/ Use UseFileServer instead of UseDefaultFiles & UseStaticFiles  
FileServerOptions fileServerOptions = new FileServerOptions();  
fileServerOptions.DefaultFilesOptions.DefaultFileNames.Clear();  
fileServerOptions.DefaultFilesOptions.DefaultFileNames.Add("foo.html");  
app.UseFileServer(fileServerOptions);

The important point to note here is the pattern that we use to add middleware to our application's request processing pipeline. In most cases we add middleware using the extension methods that start with the word USE. For example,

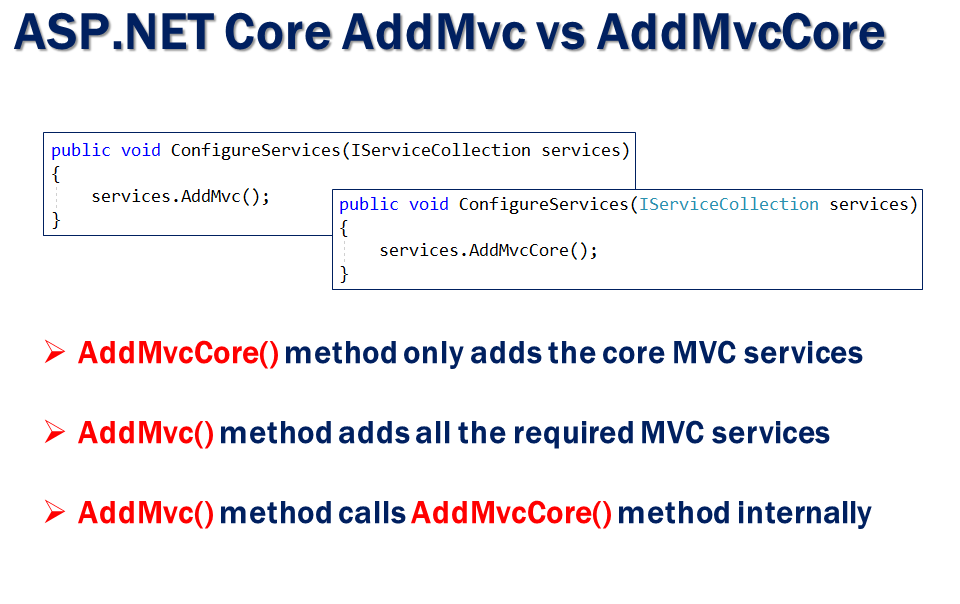
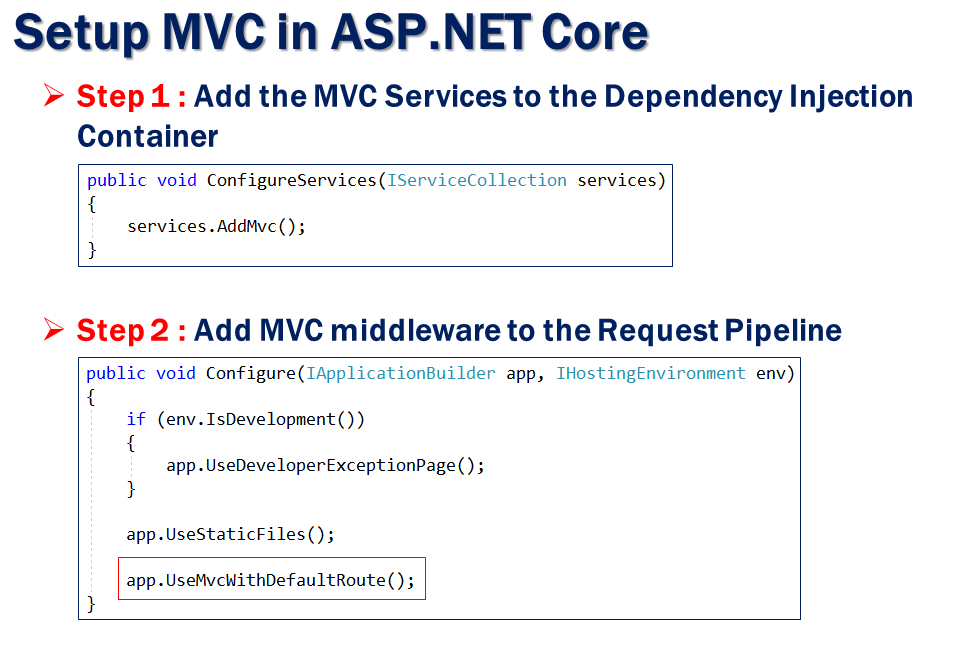
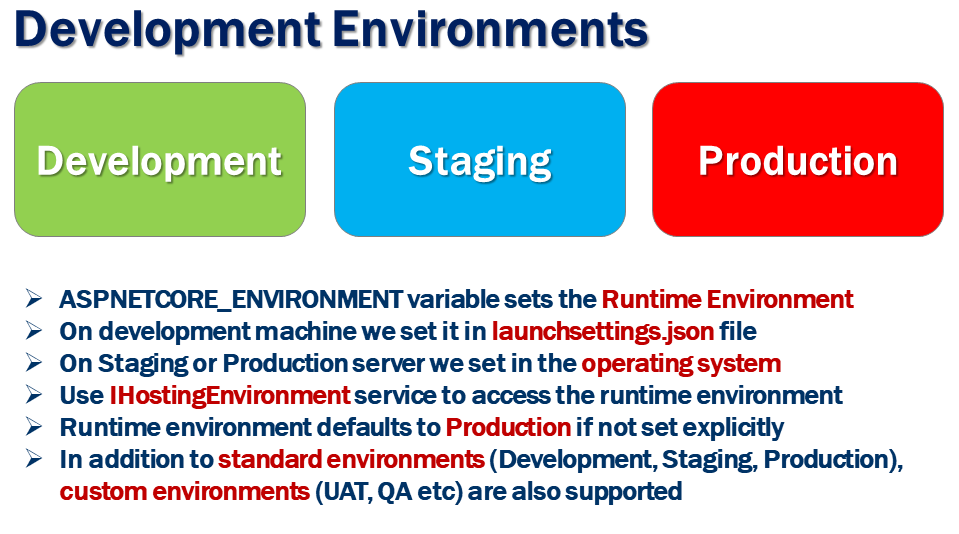
* UseDeveloperExceptionPage()
* UseDefaultFiles()
* UseStaticFiles()
* UseFileServer()

If you want to customize these middleware components, we use the respective OPTIONS object. For example, notice the respective OPTIONS objects we use.

|  |  |
| --- | --- |
| **Middleware** | **Options Object** |
| UseDeveloperExceptionPage | DeveloperExceptionPageOptions |
| UseDefaultFiles | DefaultFilesOptions |
| UseStaticFiles | StaticFileOptions |
| UseFileServer | FileServerOptions |

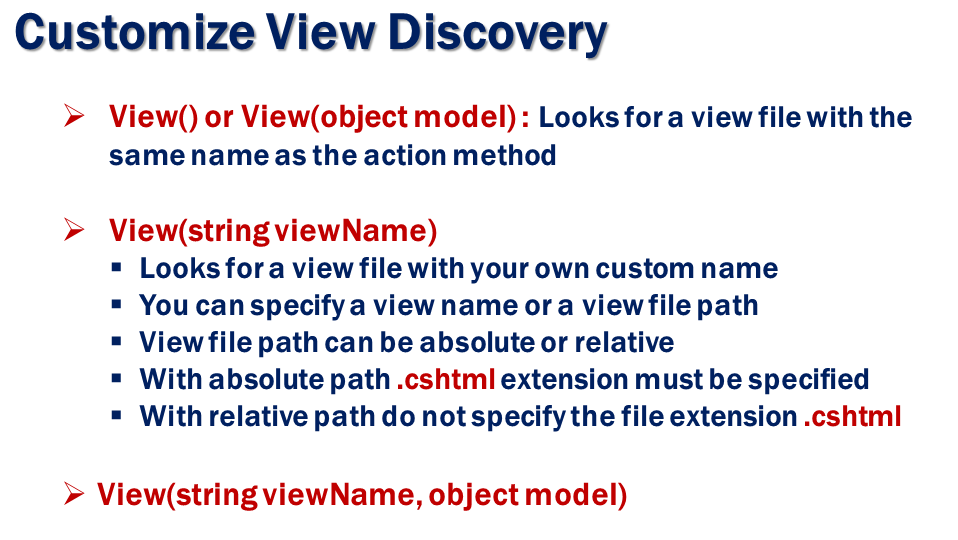
**UseDeveloperExceptionPage Middleware**

UseDeveloperExceptionPage Middleware must be plugged into the request processing pipeline as early as possible, so it can handle the exception and display the Developer Exception Page if the subsequent middleware components in the pipeline raises an exception.



**Registering Services with the ASP.NET Core Dependency Injection Container :**

**AddSingleton()** - As the name implies, AddSingleton() method creates a **Singleton service**. A Singleton service is created when it is first requested. This same instance is then used by all the subsequent requests. So in general, a Singleton service is created only one time per application and that single instance is used throughout the application life time.  
  
**AddTransient()** - This method creates a **Transient service**. A new instance of a Transient service is created each time it is requested.   
  
**AddScoped()** - This method creates a **Scoped service**. A new instance of a Scoped service is created once per request within the scope. For example, in a web application it creates 1 instance per each http request but uses the same instance in the other calls within that same web request.



**Filter In Dot net Core**

In dot core 5 type of filters are exist.

1] **Authorization Filter**

The Authorization filters are executed first. This filter helps us to determine whether the user is authorized for the current request. It can short-circuit a pipeline if a user is unauthorized for the current request. We can also create custom authorization filter.

2] **Resource Filter**

The Resource filters handle the request after authorization. It can run the code before and after the rest of the filter is executed. This executes before the model binding happens. It can be used to implement caching.

3] **Action Filter**

The Action filters run the code immediately controller action method is called. It can be used to perform any action before or after execution of the controller action method. We can also manipulate the arguments passed into an action.

4] **Exception Filter**

The Exception filters run the code immediately after the system exception. It can be used to handle exception.

5] **Result Filter**

The Result filters are used to run code before and after the execution of controller action results. They are executed only if the controller action method has been executed successfully.

**Scope of filter**

You can register the filter at global level, controller level and action level.

**To register the filter at global** you have to add the filter configuration section in startup.cs file.

Services.AddMvc(options=>

{

// Register the filter by creating the instance of filter

options.Filters.Add(**new FilterInstance**);

// Register the filter by typeof

options.Filters.Add( **typeof(FilterInstance)** );

});

**Register the filter at controller level**

[**FilterName**]

Public class LoginController : ControllerBase

{

// Some Action

}

**Register the filter at controller level**

Public class LoginController : ControllerBase

{

[**FilterName**]

Public string GetName(string id) { // Some logic}

}

**Order Of Filter Execution**

1] Authorization 2] Resource 3] Action 4] Exception 5] Result

**Overriding the default order Filter**

We can override the default sequence of filter execution by using implementing interface IOrderedFilter. This interface has property named "Order" that use to determine the order of execution. The filter with lower order value execute before the filter with higher order value. We can setup the order property using the constructor parameter.

1. **public** **class** ExampleFilterAttribute : Attribute, IActionFilter, IOrderedFilter
2. {
3. **public** **int** Order { **get**; **set**; }
5. **public** **void** OnActionExecuting(ActionExecutingContext context)
6. {
7. //To do : before the action executes
8. }
10. **public** **void** OnActionExecuted(ActionExecutedContext context)
11. {
12. //To do : after the action executes
13. }
14. }
15. **namespace** Filters.Controllers
16. {
17. [ExampleFilter(Order = 1)]
18. **public** **class** HomeController : Controller
19. {
20. **public** IActionResult Index()
21. {
22. **return** View();
23. }
24. }
25. }

**Cancellation or short circuiting Filters**

We can short circuit the filter pipeline at any point of time by setting the "Result" property of the "Context" parameter provided to the filter's methods.

1. **public** **class** Example1FilterAttribute : Attribute, IActionFilter
2. {
3. **public** **void** OnActionExecuting(ActionExecutingContext context)
4. {
5. //To do : before the action executes
6. context.Result = **new** ContentResult()
7. {
8. Content = "Short circuit filter"
9. };
10. }
11. **public** **void** OnActionExecuted(ActionExecutedContext context)
12. {
13. //To do : after the action executes
14. }
15. }

**Note**

Filter supports two types of implementation: **synchronous** and **asynchronous**; Both the implementations use different interface definitions.

The Synchronous filters run the code before and after their pipeline stage defines OnStageExecuting and OnStageExecuted. For example, ActionFilter. The OnActionExecuting method is called before the action method and OnActionExecuted method is called after the action method.

Asynchronous filters are defined with only single method: OnStageExecutionAsync, that takes a FilterTypeExecutingContext and FilterTypeExecutionDelegate as The FilterTypeExecutionDelegate execute the filter’s pipeline stage. For example, ActionFilter ActionExecutionDelegate calls the action method and we can write the code before and after we call action method.

We can implement interfaces for multiple filter types (stage) in single class. We can either implement synchronous or the async version of a filter interface, not both. The .net framework checks first for async filter interface, if it finds it, it called. If it is not found it calls the synchronous interface's method(s). If we implement both, synchronous interface is never called.

**Why We use ServiceFilter instead of attribute?**

The ServiceFilter attribute allows us to specify the type of our action filter and have it automatically resolved the class dependent object from DI

When your filter class constructor injection in that case ServiceFilter is used. Because Attributes don't support constructor injection via dependency injection.

Example :

public class LoggingActionFilter : IActionFilter

{

ILogger \_logger;

public LoggingActionFilter(ILoggerFactory loggerFactory)

{

\_logger = loggerFactory.CreateLogger<LoggingActionFilter>();

}

}

public class LoggingActionFilter1 : IActionFilter

{

}

**Exception Filter**

You can handled the exception in dot net core by using custom exception filter.

**Creating custom exception filter**

public class ExceptionFilterEample : IExceptionFilter

{

private readonly IHostingEnvironment \_hostingEnvironment;

private readonly IModelMetadataProvider \_modelMetadataProvider;

public ExceptionFilterEample(IHostingEnvironment hostingEnvironment, IModelMetadataProvider modelMetadataProvider)

{

\_hostingEnvironment = hostingEnvironment;

\_modelMetadataProvider = modelMetadataProvider;

}

public int Order { get; set; }

public void OnException(ExceptionContext context)

{

if (!\_hostingEnvironment.IsDevelopment())

{

// do nothing

return;

}

HttpStatusCode status = HttpStatusCode.InternalServerError;

var message = "Server error occurred.";

var exceptionType = context.Exception.GetType();

//You can enable logging error

context.ExceptionHandled = true;

HttpResponse response = context.HttpContext.Response;

response.StatusCode = (int)status;

response.ContentType = "application/json";

context.Result = new ObjectResult(new ApiResponse { Message = message, Data = null });

}

}

public class ExceptionFilterEampleWithAttribute : ExceptionFilterAttribute

{

private readonly IHostingEnvironment \_hostingEnvironment;

private readonly IModelMetadataProvider \_modelMetadataProvider;

public string ErrorActionName { get; set; }

public ExceptionFilterEampleWithAttribute(IHostingEnvironment hostingEnvironment, IModelMetadataProvider modelMetadataProvider)

{

\_hostingEnvironment = hostingEnvironment;

\_modelMetadataProvider = modelMetadataProvider;

}

public override void OnException(ExceptionContext context)

{

if (!\_hostingEnvironment.IsDevelopment())

{

// do nothing

return;

}

HttpStatusCode status = HttpStatusCode.InternalServerError;

var message = "Server error occurred.";

var exceptionType = context.Exception.GetType();

//You can enable logging error

context.ExceptionHandled = true;

HttpResponse response = context.HttpContext.Response;

response.StatusCode = (int)status;

response.ContentType = "application/json";

context.Result = new ObjectResult(new ApiResponse { Message = message, Data = null });

You can create custom exception filter by using IException interface Or ExceptionFilterAttribute.

You can create the handler and middleware for the same. See below example.

**Using ExcptionHandler**

public static class ExceptionMiddlewareExtensions

{

public static void ConfigureExceptionHandler(this IApplicationBuilder app) //, ILoggerManager logger

{

app.UseExceptionHandler(appError =>

{

appError.Run(async context =>

{

context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;

context.Response.ContentType = "application/json";

var contextFeature = context.Features.Get<IExceptionHandlerFeature>();

if (contextFeature != null)

{

//logger.LogError($"Something went wrong: {contextFeature.Error}");

await context.Response.WriteAsync(new ErrorDetails()

{

StatusCode = context.Response.StatusCode,

Message = "Internal Server Error.Exception of Exception filter handle by ExceptionHandle."

}.ToString());

}

});

});

}

}

Create the handler register the same in configure method of startup class in startup.cs file.

Ex.

public void Configure(IApplicationBuilder app, IHostingEnvironment env

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

//Handling the excepion using ExceptionHandler

app.ConfigureExceptionHandler();

app.UseMvc();

}

**Using Creating Custom Middleware**

public class ExceptionMiddleware

{

private readonly RequestDelegate \_next;

//private readonly ILoggerManager \_logger;

public ExceptionMiddleware(RequestDelegate next) //, ILoggerManager logger

{

//\_logger = logger;

\_next = next;

}

public async Task InvokeAsync(HttpContext httpContext)

{

try

{

await \_next(httpContext);

}

catch (Exception ex)

{

// \_logger.LogError($"Something went wrong: {ex}");

await HandleExceptionAsync(httpContext, ex);

}

}

private static Task HandleExceptionAsync(HttpContext context, Exception exception)

{

context.Response.ContentType = "application/json";

context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;

return context.Response.WriteAsync(new ErrorDetails()

{

StatusCode = context.Response.StatusCode,

Message = "Internal Server Error from the custom middleware."

}.ToString());

}

}

Create the handler register the same in configure method of startup class in startup.cs file.

Ex.

public void Configure(IApplicationBuilder app, IHostingEnvironment env

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

//Handling the excepion using Custom exception Middleware app.UseMiddleware<ExceptionMiddleware>();

app.UseMvc();

}

**Action Filter**

Action filters are executed after or before execution of a specific action or all action.

You can create the action filter by inheriting by IActionFilter interface or ActonFilterAttribute.

There are three type of action filter

1] Attribute Type

2] ServiceFilter Type

3]TypeFilter type

Attribute type filter can not instantiate the constructor of filter class.

[ActionFilterExample(Order =3)] // register at controller level

public class ValuesController : ControllerBase

{

[ActionFilterExample(Order =3)] // register at action level

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

return new string[] { "value1", "value2" };

}

}

**Register filter Globally**

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

config.filters.add(new actionfilterexample()) // Using Instance

config.filters.add(tyepof(actionfilterexample)) // Using Typeof

}

}

Service Type filter instantiate the constructor of filter class using dependency injection.

[ServiceFilter(typeof(ServiceTypeActionFilter),Order =2)] // register at controller level

public class ValuesController : ControllerBase

{

[ServiceFilter(typeof(ServiceTypeActionFilter),Order =2)] // register at action level

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

return new string[] { "value1", "value2" };

}

}

**Register filter Globally**

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

config.Filters.Add(typeof(ServiceTypeActionFilter),2)

services.AddScoped<ServiceTypeActionFilter>();

}

}

TypeFilter Type filter can instantiate the constructor of filter class using dependency injection

and passed the argument

register at controller level

[TypeFilter(typeof(ActionTypeFilter),Arguments = new object[] { 15, "Gaurav" })]

public class ValuesController : ControllerBase

{

register at action level

[TypeFilter(typeof(ActionTypeFilter),Arguments = new object[] { 15, "Gaurav" })]

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

return new string[] { "value1", "value2" };

}

}

**Result Filter**

[Result filters](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-2.2#result-filters) can run code immediately before and after the execution of individual action results. They run only when the action method has executed successfully. They are useful for logic that must surround view or formatter execution.

You can create the result filter by inheriting by IResultFilter interface or ResultFilterAttribute.

[ResultFilter(Order =3)] // register at controller level

public class ValuesController : ControllerBase

{

[ResultFilter (Order =3)] // register at action level

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

return new string[] { "value1", "value2" };

}

}

**Register filter Globally**

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

config.filters.add(new ResultFilter()) // Using Instance

config.filters.add(tyepof(ResultFilter)) // Using Typeof

}

}

**Resource Filter**

Resource filters are the first to handle a request after authorization. They can run code before the rest of the filter pipeline, and after the rest of the pipeline has completed.

Resource filter is useful to implement caching or otherwise short-circuit the filter pipeline for performance reasons. **It runs before model binding**, so it can influence model binding.

**Properties of Resource filters**

Implement either the IResourceFilter or IAsyncResourceFilter interface,

Their execution wraps most of the filter pipeline.Only Authorization filters run before Resource filters. Caching filter can avoid the rest of the pipeline if the response is in the cache i.e **ShortCircuiting.**

Can be used for,

**Prevents model binding from accessing the form data.**

**Useful for large file uploads and prevent a form from being read into memory.**

A common use of Resource Filters in,

**Logging**

**Caching**

**Modifying model binding**

**Creating the Resource Filter**

public class ResourceFilter : Attribute, IResourceFilter

{

public void OnResourceExecuting(

ResourceExecutingContext context)

{

context.Result = new ContentResult()

{

Content = "Resource unavailable - header should not be set"

};

}

public void OnResourceExecuted(ResourceExecutedContext context)

{

}}

[ResourceFilter (Order =3)] // register at controller level

public class ValuesController : ControllerBase

{

[ResourceFilter (Order =3)] // register at action level

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

return new string[] { "value1", "value2" };

}

}

**Register filter Globally**

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

config.filters.add(new ResourceFilter ()) // Using Instance

config.filters.add(tyepof(ResourceFilter)) // Using Typeof

}

}

**Custom Constraint in .CORE**

**Need to implement the** IRouteConstraint **interface and override the** Match **method of interface.**

The Match method has 5 parameters . Let's discuss each, in details.

1. *HttpContext* encapsulates all http specific information about an http request like request, response, sessions, and more. You can check the httpcontext as well if the request is authenticated or not, and take the appropriate action accordingly.
2. *IRouter* is the router which belongs to constraints.
3. *RouteKey* is the name of the parameter that is being checked. The same name is defined in route template.
4. *Routevalues* contain the parameters of the URL.
5. *RouteDirection* indicates whether ASP.NET routing is processing a URL from an HTTP request or generating a URL. This is an enum class that has two directions.
   1. IncomingRequest  
      A URL from a client is being processed.
   2. UrlGeneration  
      A URL is being created based on the route definition.
6. **public** **class** AlphaNumericConstraint : IRouteConstraint
7. {
8. **private** **static** **readonly** TimeSpan RegexMatchTimeout = TimeSpan.FromSeconds(10);
10. **public** **bool** Match(HttpContext httpContext,**string** routeKey,
11. RouteValueDictionary values,RouteDirection routeDirection)
12. {
13. **object** routeValue;
14. **if**(values.TryGetValue(routeKey, **out** routeValue))
15. {
16. var parameterValueString = Convert.ToString(routeValue, CultureInfo.InvariantCulture);
17. **Var result =** **new** Regex(@"^[a-zA-Z0-9]\*$",
18. RegexOptions.CultureInvariant  | RegexOptions.IgnoreCase,  RegexMatchTimeout).IsMatch(parameterValueString);
19. Return result
20. }
21. **return** **false**;    }
22. }

Add the constraint into ConstraintMap using route option in configure services method in the startup class.

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
3. // Add framework services.
4. services.AddMvc();
6. // add here your route constraint
7. services.Configure<RouteOptions>(routeOptions =>
8. {
9. routeOptions.ConstraintMap.Add("alphanumeric", **typeof**(AlphaNumericConstraint));
10. });
11. }

[HttpGet("{id:**alphanumeric**}")]

public void GetValue(string id)

{

var d = id;

}