



.NET

.NET CORE

Documentation

[Abstract](#)

In this module I learn basic things about .NET Core

Dhruvil Dobariya
dhruvildobariya21@gmail.com

INDEX

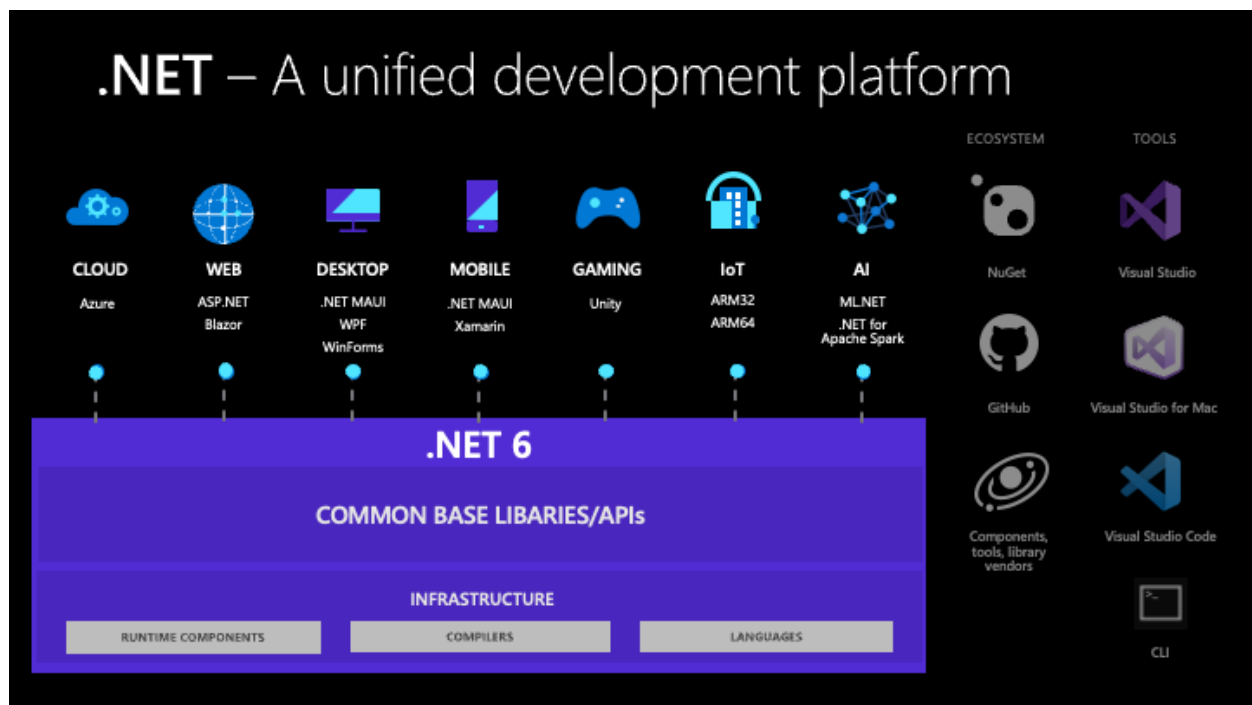
1	OVERVIEW OF .NET CORE	1
1.1	INTRODUCTION	1
1.2	WHY .NET CORE	2
1.3	CHARACTERISTICS	2
1.4	.NET CORE COMPOSITION.....	4
1.5	.NET CORE VERSIONS	4
2	OVERVIEW OF ASP.NET CORE	6
2.1	INTRODUCTION	6
2.2	WHY ASP.NET CORE	6
2.3	PROJECT STRUCTURE.....	7
3	CONTROLLER INITIALIZATION	12
3.1	INTRODUCTION	12
3.2	WHAT IS CONTROLLER	13
3.3	HOW TO CREATE CONTROLLER	14
4	ACTION METHOD.....	16
4.1	INTRODUCTION	16
4.2	ACTION RESULT	16
5	ROUTING	19
5.1	INTRODUCTION	19
6	MIDDLEWARE	23
6.1	INTRODUCTION	23
6.2	EXTENSION METHODS.....	23
6.3	BUILT IN MIDDLEWARE	25
6.4	CUSTOM MIDDLEWARE.....	26
7	FILTER.....	30
7.1	INTRODUCTION	30
7.2	SCOPE AND ORDER	33
7.3	CANCELLATION OR SHORT-CIRCUITING FILTERS	35
7.4	DEPENDENCY INJECTION IN FILTERS.....	35
8	DEPENDENCY INJECTION	38
8.1	INTRODUCTION	38
8.2	IOC CONTAINER	39
8.3	LIFETIME OF SERVICES	39
8.4	CONSTRUCTOR INJECTION	41
9	EXCEPTION HANDLING	43
9.1	INTRODUCTION	43

9.2	UseDeveloperExceptionPage	43
9.3	UseExceptionHandler	46
10	LOGGING API	48
10.1	INTRODUCTION	48
10.2	LOGGING API	48
10.3	LOGGING PROVIDERS	49

OVERVIEW OF .NET CORE

1.1 INTRODUCTION

- .NET Core is a new version of .Net framework.
- Which is a free, open-source, general-purpose development platform maintained by Microsoft.
- It is a cross-platform framework that runs on Windows, macOS, and Linux operating systems.
- .NET Core Framework can be used to build different types of applications such as mobile, desktop, web, cloud, IoT, machine learning, microservices, game, etc.
- .NET Core is written from scratch to make it modular, lightweight, fast, and cross-platform Framework.



- It includes the core features that are required to run a basic .NET Core app.
- Other features are provided as NuGet packages, which you can add it in your application as needed.
- In this way, the .NET Core application speed up the performance, reduce the memory footprint and becomes easy to maintain.

1.2 WHY .NET CORE

- There are some limitations with the .NET Framework.
- For example, it only runs on the Windows platform.
- Also, you need to use different .NET APIs for different Windows devices such as Windows Desktop, Windows Store, Windows Phone, and Web applications.
- In addition to this, the .NET Framework is a machine-wide framework.
- Any changes made to it affect all applications taking a dependency on it.
- Today, it's common to have an application that runs across devices.
- A backend on the web server, admin front-end on windows desktop, web, and mobile apps for consumers.
- So, there is a need for a single framework that works everywhere.
- So, considering this, Microsoft created .NET Core.
- The main objective of .NET Core is to make .NET Framework open-source, cross-platform compatible that can be used in a wide variety of verticals, from the data center to touch-based devices.

1.3 CHARACTERISTICS

Open-source Framework:

- .NET Core is an open-source framework maintained by Microsoft and available on GitHub under MIT and Apache 2 licenses.
- It is a .NET Foundation project.
- You can view, download, or contribute to the source code using the following GitHub repositories:
- Language compiler platform Roslyn: <https://github.com/dotnet/roslyn>
- .NET Core runtime: <https://github.com/dotnet/runtime>
- .NET Core SDK repository. <https://github.com/dotnet/sdk>
- ASP.NET Core repository. <https://github.com/dotnet/aspnetcore>

Cross-platform:

- .NET Core runs on Windows, macOS, and Linux operating systems.
- There are different runtime for each operating system that executes the code and generates the same output.

Consistent across Architectures:

- Execute the code with the same behavior in different instruction set architectures, including x64, x86, and ARM.

Wide-range of Applications:

- Various types of applications can be developed and run on .NET Core platform such as mobile, desktop, web, cloud, IoT, machine learning, microservices, game, etc.

Supports Multiple Languages:

- You can use C#, F#, and Visual Basic programming languages to develop .NET Core applications.
- You can use your favorite IDE, including Visual Studio 2019/2022, Visual Studio Code, Sublime Text, Vim, etc.

Modular Architecture:

- .NET Core supports modular architecture approach using NuGet packages.
- There are different NuGet packages for various features that can be added to the .NET Core project as needed.
- Even the .NET Core library is provided as a NuGet package.
- The NuGet package for the default .NET Core application model is Microsoft.NETCore.App.
- This way, it reduces the memory footprint, speeds up the performance, and easy to maintain.

CLI Tools:

- .NET Core includes CLI tools (Command-line interface) for development and continuous-integration.

Flexible Deployment:

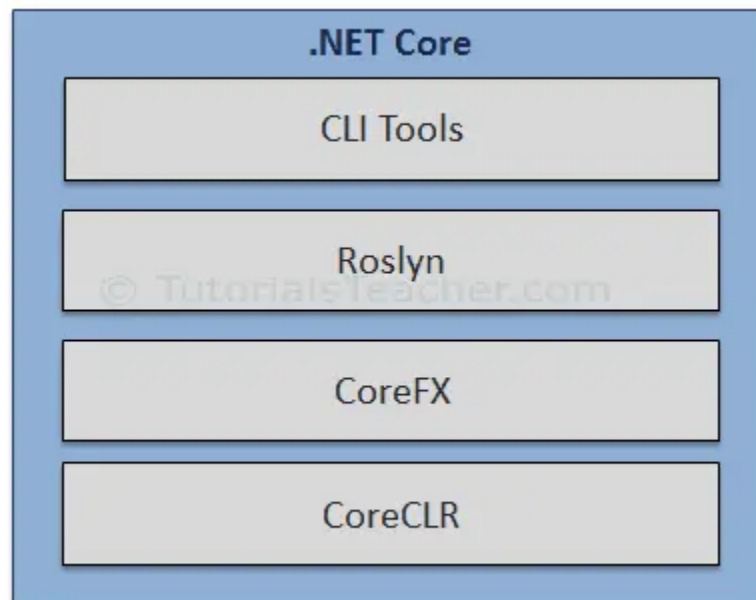
- .NET Core application can be deployed user-wide or system-wide or with Docker Containers.

Compatibility:

- Compatible with .NET Framework and Mono APIs by using .NET Standard specification.

ASP.NET Core

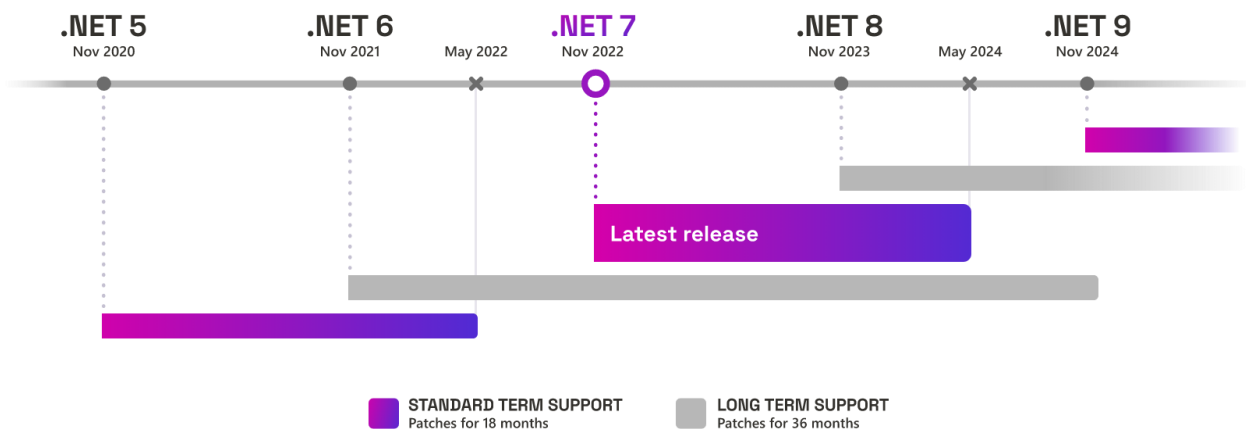
1.4 .NET CORE COMPOSITION



- CLI Tools: A set of tooling for development and deployment.
- Roslyn: Language compiler for C# and Visual Basic
- CoreFX: Set of framework libraries.
- CoreCLR: A JIT based CLR (Common Language Runtime).

1.5 .NET CORE VERSIONS

- It release in 27 June 2016.
- .Net core first version to .Net core 3.1 it know as .Net core after 4th version skip and release next version of .Net core which know as .Net 5.



ASP.NET Core

- After it continues it's sequence.
- .Net Core 3.1 is LTS and .Net 5 is Current term version, after every even version is LTS and odd version is Current term version.
- Every LTS version is supported by 3 year and current term is 18 months.
- Every year Microsoft release first preview version of next .Net core on 17 to 21 February.
- Every year Microsoft release 7 preview versions of next .Net core.
- Every year Microsoft release first release candidate version of next .Net core on 14 September.
- Every year Microsoft release 2 release candidate versions of next .Net core.
- Every year Microsoft release stable version new version on 8 November.

OVERVIEW OF ASP.NET CORE

2.1 INTRODUCTION

- ASP.NET Core is the new version of the ASP.NET web framework mainly targeted to run on .NET Core platform.
- ASP.NET Core is a free, open-source, and cross-platform framework for building cloud-based applications, such as web apps, IoT apps, and mobile backends.
- It is designed to run on the cloud as well as on-premises.
- Same as .NET Core, it was architected modular with minimum overhead, and then other more advanced features can be added as NuGet packages as per application requirement.
- This results in high performance, require less memory, less deployment size, and easy to maintain.

2.2 WHY ASP.NET CORE

Supports Multiple Platforms:

- ASP.NET Core applications can run on Windows, Linux, and Mac.
- So you don't need to build different apps for different platforms using different frameworks.

Fast:

- ASP.NET Core no longer depends on System.Web.dll for browser-server communication.
- ASP.NET Core allows us to include packages that we need for our application.
- This reduces the request pipeline and improves performance and scalability.

IoC Container:

- It includes the built-in IoC container for automatic dependency injection which makes it maintainable and testable.

Integration with Modern UI Frameworks:

- It allows you to use and manage modern UI frameworks such as Angular, ReactJS, Umber, Bootstrap, etc.

Hosting:

- ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache etc.
- It is not dependent only on IIS as a standard .NET Framework.

Code Sharing:

- It allows you to build a class library that can be used with other .NET frameworks such as .NET Framework 4.x or Mono.
- Thus a single code base can be shared across frameworks.

Side-by-Side App Versioning:

- ASP.NET Core runs on .NET Core, which supports the simultaneous running of multiple versions of applications.

Smaller Deployment Footprint:

- ASP.NET Core application runs on .NET Core, which is smaller than the full .NET Framework.
- So, the application which uses only a part of .NET CoreFX will have a smaller deployment size.
- This reduces the deployment footprint.

2.3 PROJECT STRUCTURE

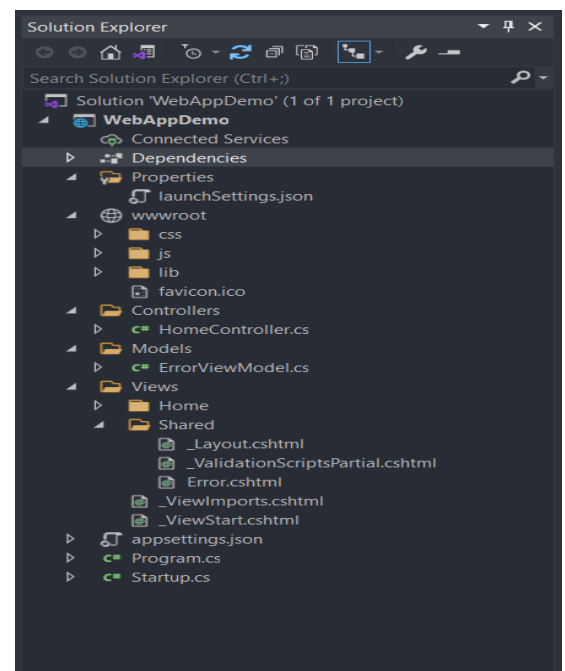
Properties:

- The Properties Folder in ASP.NET Core Web Application by default contains one JSON file called as launchSettings.json file as shown in the below image.

Models:

- The Models folder of an ASP.NET Core MVC application contains the class files which are used to store the domain data (you can also say business data) as well as business logic to manage the data.

View:



- The Views Folder of an ASP.NET Core MVC application contains all the “.cshtml” files of your application.
- In MVC, the .cshtml file is a file where we need to write the HTML code along with the C# code.
- The Views folder also includes separate folders for each and every controller for your application.
- For example: all the .cshtml files of the HomeController will be in the View => Home folder.
- We also have the Shared folder under the Views folder.
- The Shared Folder contains all the views which are needed to be shared by different controllers.
- For example: error files, layout files, etc.

Controllers:

- The ASP.NET Core Web API is a controller-based approach.
- All the controllers of your ASP.NET Core Web API Application should and must reside inside the Controllers folder.
- It contain business logic of Web API, It handles all requests and give corresponding response.
- This File inherited from **“Controller”** class or **“ControllerBase”** class.
- **“Controller”** class is derived from the **“ControllerBase”** class.
- **“Controller”** should use when we want to return View from **“ActionMethod”**, Because when we use **“ControllerBase”** class then we can't render View.

appsettings.json file:

- This is the same as web.config or app.config of our traditional .NET Application.
- The appsettings.json file is the application configuration file in ASP.NET Core Web Application used to store the configuration settings such as database connections strings, any application scope global variables, etc.

appsettings.Development.json:

- If you want to configure some settings based on the environments then you can do such settings in appsettings.{Environment}.json file.
- You can create n number of environments like development, staging, production, etc.
- If you set some settings in the appsettings.Development.json file, then such settings can only be used in the development environment, can not be used in other environments.

wwwroot:

- The “**wwwroot**” folder in the ASP.NET Core project is treated as a web root folder.
- Static files can be stored in any folder under the web root and accessed with a relative path to that root.
- **Libman.json**: This file contains the list of libraries for static file to download.
- Each library has a name, a version, a list of files to download, and the location where the file will be copied.

Program.cs:

- It has a public static void Main() method.
- The Main method is the entry point of our Application.
- Each ASP.NET Core Web API Application initially starts as a Console Application and the Main() method is the entry point to the application.
- So, when we execute the ASP.NET Core Web API application, it first looks for the Main() method and this is the method from where the execution starts for the application.
- The Main() method then configures ASP.NET Core and starts it. At this point, the application becomes an ASP.NET Core Web API application.

Startup.cs:

- The Startup class is like the Global.asax file of our traditional .NET application.
- As the name suggests, it is executed when the application starts.
- Startup class includes two public methods:
 - ConfigureServices
 - Configure

ConfigureServices:

- The ConfigureServices method of the Startup class is the place where we can register our dependent classes with the built-in IoC container.
- Once we register the dependent classes, they can be used anywhere within the application.
- The ConfigureServices method includes the IServiceCollection parameter to register services to the IoC container.

Configure:

ASP.NET Core

- The Configure method of the Startup class is the place where we configure the application request pipeline using the IApplicationBuilder instance that is provided by the built-in IoC container.
- ASP.NET Core introduced the middleware components to define a request pipeline, which will be executed on every request.
- If you look at the Configure method, it registered UseDeveloperExceptionPage, UseSwagger, UseSwaggerUI, UseHttpsRedirection, UseRouting, UseAuthorization, and UseEndpoints middleware components to the request processing pipeline.
- After coming .Net 6 “Startup.cs” file was removed and combine the functionality with the “Program.cs” file.
- So now after .Net 6 we have only “Program.cs”.


launchSettings.json:


- The launchsettings.json file contains some settings that are going to be used by .NET Core when we run the application either from Visual Studio or by using .NET Core CLI.


```

{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:63044",
      "sslPort": 44395
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "MyFirstWebAPIProject": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

 This URL will be used when we use IIS Express Profile to run our Application

 This setting is for IIS Express

 This setting is for Kestrel Web Server

ASP.NET Core

- The launchSettings.json file is only used within the local development machine.
- So, this file is not required when we publishing our ASP.NET Core Web API application into the production server.
- Now, open the launchSettings.json file, by default you will see the following settings.
- If you are running your application from the visual studio then IIS Express Profile will be used (for HTTP the port number will be 63044 and for HTTPS the port number will be 44395).
- if you are running your application using .NET Core CLI, then WebAPIDemo profile will be used which is nothing but using Kestrel Web Server and for HTTP protocol it uses the port number 5000 and for HTTPS protocol it uses the port number 5001.

CONTROLLER INITIALIZATION

3.1 INTRODUCTION

- In ASP.NET Core, controller initialization refers to the process of creating an instance of a controller class and preparing it for handling incoming HTTP requests.
- When a request is made to the server, ASP.NET Core follows a series of steps to initialize the controller before invoking the appropriate action method.
- Here's an overview of the controller initialization process in ASP.NET Core,

Routing:

- The request first goes through the routing middleware, which examines the URL and determines the appropriate controller and action method based on the defined routes in the application.

Controller Activation:

- Once the routing middleware identifies the controller, ASP.NET Core activates an instance of the controller class.
- By default, the framework uses the built-in dependency injection (DI) container to create the controller instance.
- The DI container resolves any dependencies required by the controller, injecting them through constructor parameters.

Action Method Selection:

- After the controller instance is created, ASP.NET Core selects the appropriate action method to handle the request.
- The action method is determined based on the HTTP verb (GET, POST, PUT, DELETE, etc.) and the name of the method.
- The framework also considers any route constraints or attribute-based routing when selecting the action method.

Model Binding:

- If the action method has parameters, ASP.NET Core performs model binding.
- Model binding maps the data from the request (query string parameters, form values, JSON payload, etc.) to the parameters of the action method.
- The framework uses model binders to perform this mapping automatically.

Action Method Execution:

- Once the model binding is complete, ASP.NET Core invokes the selected action method, passing in the bound parameters.
- The action method contains the logic to process the request and generate a response.

Result Execution:

- After the action method executes, it typically returns a result object, such as a view, JSON, or HTTP status code.
- ASP.NET Core handles the result and sends the appropriate response back to the client.
- It's important to note that ASP.NET Core supports extensibility points throughout the controller initialization process.
- We can customize the behavior by implementing filters, middleware, or overriding default conventions to suit your application's requirements.
- Overall, controller initialization in ASP.NET Core is a crucial step in handling HTTP requests, involving the creation of a controller instance, action method selection, model binding, execution, and result handling.

3.2 WHAT IS CONTROLLER

- In ASP.NET Core, a controller is a class that inherits from the Controller base class or ControllerBase for API controllers.
- It contains a set of action methods that are responsible for processing specific HTTP requests and generating appropriate responses.
- Each action method typically corresponds to a specific endpoint or route in the application.
- The controller receives the incoming HTTP request, determines the appropriate action method to execute based on the request's route and HTTP verb, and then executes the method.
- The action method performs the required business logic, such as retrieving data, updating the model, or interacting with other services, and generates a response that is sent back to the client.
- Controllers in ASP.NET Core are responsible for tasks such as,

Handling user input:

- Controllers receive user input from the client, such as form data, query parameters, or JSON payloads, and process that input as needed.

ASP.NET Core

Model binding:

- Controllers facilitate the process of binding incoming request data to action method parameters, allowing easy access to user input.

Coordinating actions:

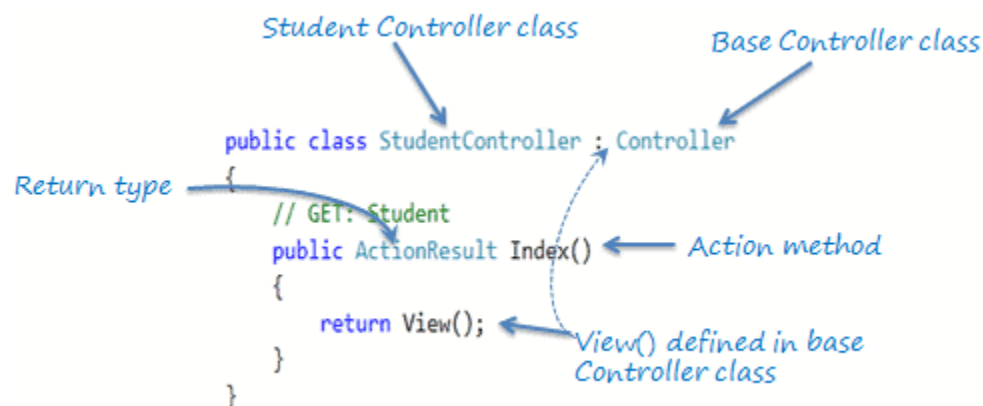
- Controllers coordinate the necessary actions to fulfill the user request, such as retrieving data from a database, performing calculations, or invoking other services.

Returning responses:

- Controllers generate and return responses to the client, which can include rendering views, returning JSON or XML data, or redirecting to other URLs.

3.3 HOW TO CREATE CONTROLLER

- A controller is a special class that can be created in web development.



- Here are some important things to know about controllers,

Naming Convention:

- A controller class typically has a name that ends with "Controller".
- This naming convention helps identify it as a controller.

Inheritance or Attribute:

- A controller class can inherit from another class with a "Controller" suffix or have the [Controller] attribute applied to it.
- This indicates that it is a controller.

Public and Instantiable:

- Controllers are usually declared as public classes that can be instantiated.
- This allows them to handle incoming requests.

UI-Level Responsibility:

- The main role of a controller is to handle user requests and ensure that the received data is valid.
- It decides which view or result should be returned based on the request.

Delegation of Responsibilities:

- In well-structured applications, controllers typically delegate data access and business logic tasks to separate services.
- The controller focuses on handling the request and coordinating with these services.

ACTION METHOD

4.1 INTRODUCTION

- It is used in controller classes to handle incoming HTTP requests and produce an HTTP response.
- It is responsible for executing the logic and returning the result to the client.
- The Action method is a public method defined within a controller class and is typically decorated with attributes to define the HTTP method it handles (such as [HttpGet], [HttpPost], etc.) and to specify route templates.
- Action method must be public.
- Action method cannot be overloaded, a static method and private or protected.

Syntax:

```
public class HomeController : Controller
{
    [HttpGet]
    public IActionResult Index()
    {
        // Perform some logic here
        // Return the appropriate IActionResult
        return View();
    }
}
```

- Action method return any thing like,
 - Void
 - Object – like int, List<Product>
 - ActionResult, or IActionResult

4.2 ACTION RESULT

- ActionResult and IActionResult are classes/interfaces that represent the result of an action method in a controller.
- They provide a way to encapsulate the result that will be sent back to the client in response to an HTTP request.
- The IActionResult interface contains a single method, ExecuteResultAsync, which is responsible for executing the result and generating the appropriate response.
- ActionResult is a base class that implements the IActionResult interface.

ASP.NET Core

- It provides a set of common functionality and properties that can be used by derived classes.
- It also provides a default implementation of the `ExecuteResultAsync` method.

Syntax:

```
public interface IActionResult
{
    Task ExecuteResultAsync(ActionContext context);
}
```

- Using `ActionResult` or `IActionResult` as the return type for an action method allows you to return different types of results based on the specific needs of your application.
- It provides flexibility in generating responses and enables you to handle various scenarios, such as rendering views, returning JSON data, redirecting to other actions or URLs, returning specific HTTP status codes, and more.
- It have different types of response which it can be return,

Return Type	Description
<code>PhysicalFileResult</code>	Represents an action result that returns a physical file from the server.
<code>ChallengeResult</code>	Represents an action result that initiates an authentication challenge.
<code>JsonResult</code>	Represents an action result that serializes an object to JSON format.
<code>StatusCodeResult</code>	Represents an action result that returns a specific HTTP status code.
<code>FileResult</code>	Represents an action result that returns a file to the client.
<code>ObjectResult</code>	Represents an action result that serializes an object to a specific format.
<code>RedirectResult</code>	Represents an action result that performs a redirection to a specified URL.
<code>ContentResult</code>	Represents an action result that returns a user-defined content.
<code>ViewResult</code>	Represents an action result that renders a specified view to the response stream.
<code>RedirectToActionResult</code>	Represents an action result that performs a redirection to a specified action and controller.
<code>ForbiddenResult</code>	Represents an action result that indicates an access denied response.
<code>VirtualFileResult</code>	Represents an action result that returns a virtual file from the server.
<code>OkResult</code>	Represents an action result that indicates a successful HTTP request with no content.
<code>CreatedResult</code>	Represents an action result that returns a HTTP 201 Created response.
<code>AcceptedResult</code>	Represents an action result that returns a HTTP 202 Accepted response.
<code>BadRequestResult</code>	Represents an action result that indicates a HTTP 400 Bad Request response.
<code>NoContentResult</code>	Represents an action result that indicates a HTTP 204 No Content response.
<code>NotFoundResult</code>	Represents an action result that indicates a HTTP 404 Not Found response.
<code>UnauthorizedResult</code>	Represents an action result that indicates a HTTP 401 Unauthorized response.

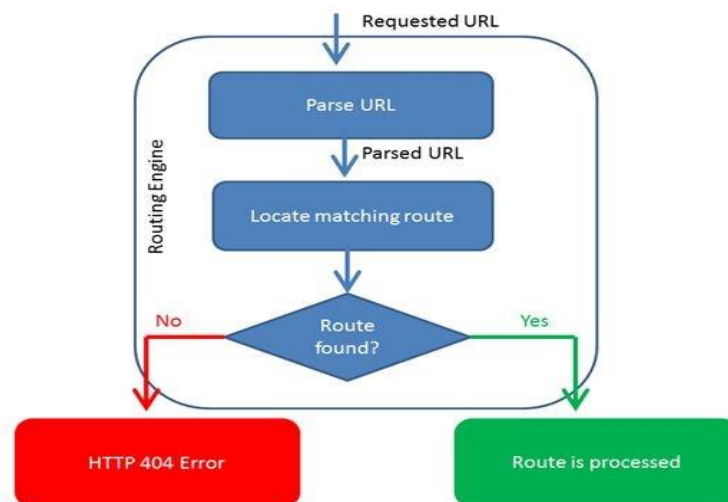
ASP.NET Core

RedirectToRouteResult	Represents an action result that performs a redirection to a specified route.
OkObjectResult	Represents an action result that returns a HTTP 200 OK response with an object.
ActionResult	Represents a generic base class for action results.
PartialViewResult	Represents an action result that renders a partial view to the response stream.

ROUTING

5.1 INTRODUCTION

- Routing is a system that matches incoming requests to specific actions or resources.
- When a user requests a URL, the routing system examines it and looks for a matching route.
- If a match is found, the system knows how to handle the request, such as serving a webpage or executing a controller action.
- If there's no matching route, it returns a 404 error.
- Essentially, routing helps determine what should be done with each user request.



- We have two types of routing,
 - Conventional based routing
 - Attribute routing

Conventional based routing:

- It creates routes based on a series of conventions which represent all the possible routes in your system.
- Convention-based are defined in the program.cs file.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{Id?}");
```

ASP.NET Core

Example:

```
using Microsoft.AspNetCore.Mvc;

namespace RoutingLearn.Controllers
{
    public class ProductController : Controller
    {
        // Product/Get
        [HttpGet]
        public IActionResult Get()
        {
            return View();
        }

        // Product/Get/1
        [HttpGet]
        public IActionResult GetById(int id)
        {
            return View();
        }

        // Product/Create
        [HttpPost]
        public IActionResult Create()
        {
            return View();
        }

        // Product/Update/1
        [HttpPut]
        public IActionResult Update(int id)
        {
            return View();
        }

        // Product/Delete/1
        [HttpDelete]
        public IActionResult Delete(int id)
        {
            return View();
        }
    }
}
```

ASP.NET Core

Attribute based routing:

- It creates routes based on attributes placed on controller or action level.
- Attribute routing provides us more control over the URLs generation patterns which helps us in SEO.

Example:

```
using Microsoft.AspNetCore.Mvc;

namespace RoutingLearn.Controllers
{
    // [Route("[controller]/[action]/{:id?}")]
    public class UserController : Controller
    {
        // User/Get
        // GetUsers
        [HttpGet]
        [Route("User/Get")]
        [Route("GetUsers")]
        public IActionResult Get()
        {
            return View();
        }

        // User/Get/1
        [HttpGet]
        [Route("User/GetById/{:id}")]
        public IActionResult GetById(int id)
        {
            return View();
        }

        // User/Create
        [HttpPost]
        [Route("User/Create")]
        public IActionResult Create()
        {
            return View();
        }

        // User/Update/1
        [HttpPut]
        [Route("User/Update/{:id}")]
    }
}
```


ASP.NET Core

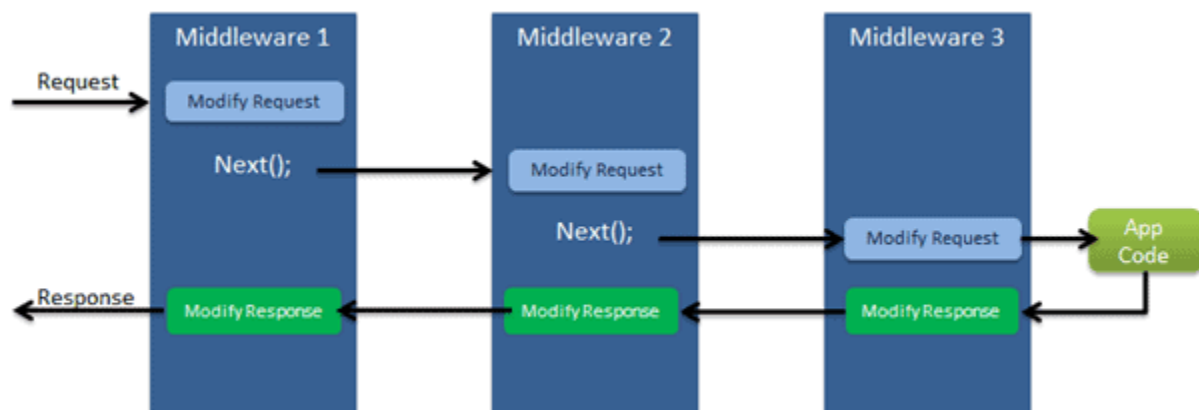
```
public IActionResult Update(int id)
{
    return View();
}

// User/Delete/1
[HttpDelete]
[Route("User/Delete/{:id}")]
public IActionResult Delete(int id)
{
    return View();
}
}
```

MIDDLEWARE

6.1 INTRODUCTION

- In ASP.NET Core, middleware refers to components that are placed in the request/response pipeline to process incoming requests and outgoing responses.
- Middleware components provide a way to handle cross-cutting concerns, such as authentication, logging, exception handling, and more.
- Middleware is executed in the order it is added to the pipeline.
- It allowing you to control the flow of the request processing.
- Each middleware component can choose to pass the request to the next component in the pipeline or short-circuit the pipeline and return a response immediately.
- The first middleware added will be the first one to handle the request and last handle response, and the last middleware added will be the last one to handle the request and first response.
- We can configure request-response pipeline in Program.cs file.



6.2 EXTENSION METHODS

- In ASP.NET Core, middleware components are added to the request/response pipeline using extension methods provided by the `IApplicationBuilder` interface.
- These extension methods make it easy to add and configure middleware components in a fluent and readable manner.
- We have some extension method we we can use to add middleware in request-response pipeline.

UseMiddleware<TMiddleware>():

- This method adds a middleware component of type `TMiddleware` to the pipeline.

Dhruvil A. Dobariya

ASP.NET Core

Example:

```
app.UseMiddleware<FirstMiddleware>();
```

UseMiddleware<TMiddleware>(params object[] args):

- This method adds a middleware component of type TMiddleware to the pipeline and passes additional parameters to its constructor.

Example:

```
app.UseMiddleware<FirstMiddleware>(arg1, arg2, ...);
```

Use(Func<HttpContext, Func<Task>, Task>):

- This method allows you to define inline middleware using a Func delegate.
- You can write custom middleware logic directly in this delegate.

Example:

```
app.Use(async (context, next) =>
{
    // Custom middleware logic

    await next();

    // Additional logic after the next middleware
});
```

UseWhen(Func<HttpContext, bool>, Action<IApplicationBuilder>):

- This method conditionally adds middleware based on a predicate.
- The specified middleware is only executed if the predicate evaluates to true.

```
app.UseWhen(context => context.Request.Path.StartsWithSegments("/admin"),
appBuilder =>
{
    appBuilder.UseMiddleware<AdminMiddleware>();
});
```

Map(string, Action<IApplicationBuilder>):

- This method maps a specific request path to a branch of the pipeline.

ASP.NET Core

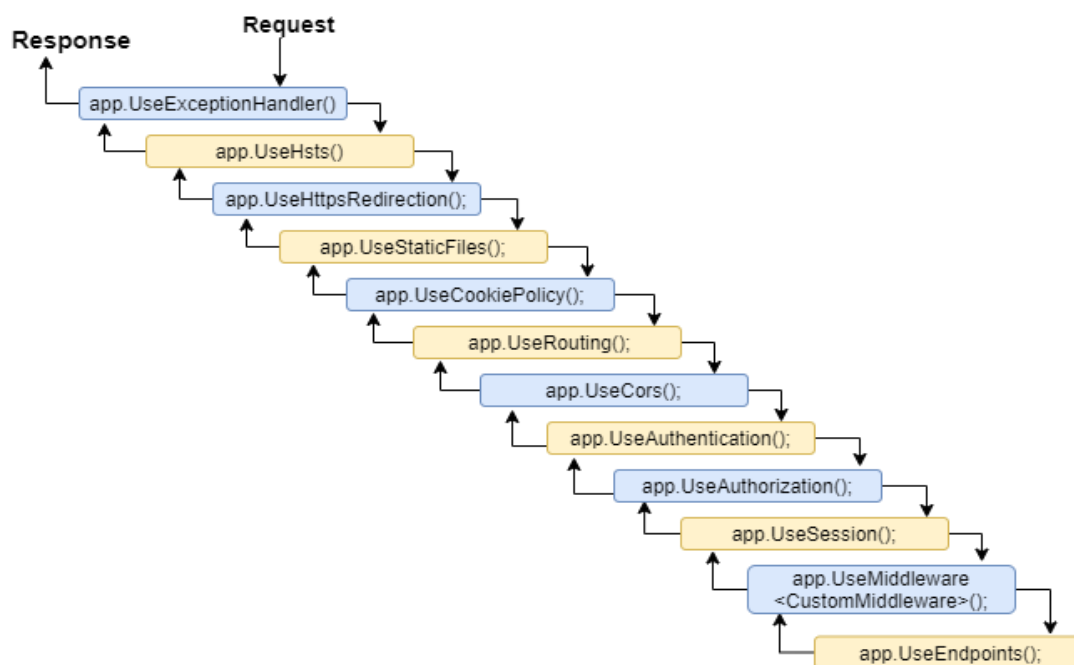
- It allows you to specify a different middleware pipeline for a particular URL or route.

Example:

```
app.Map("/api", apiApp =>
{
    // Middleware pipeline for "/api" requests
    apiApp.UseMiddleware<ApiMiddleware>();
});
```

6.3 BUILT IN MIDDLEWARE

- ASP.NET core provides built in middleware which helps use to run application in appropriate manner.
- All middleware add after we use app.Run() method for run whole application with request-response pipeline.



Middleware	Description
<code>app.UseExceptionHandler()</code>	Handles unhandled exceptions and generates an appropriate HTTP response with details of the exception.
<code>app.UseHsts()</code>	Adds the HTTP Strict Transport Security (HSTS) header to responses, instructing browsers to use HTTPS for future requests.
<code>app.UseHttpsRedirection()</code>	Redirects HTTP requests to HTTPS, based on the configuration.

<code>app.UseStaticFiles()</code>	Enables serving static files (HTML, CSS, JavaScript, images, etc.) from the specified folder or directory.
<code>app.UseCookiePolicy()</code>	Implements the cookie policy, including security and consent requirements for cookies.
<code>app.UseRouting()</code>	Sets up routing for incoming requests, allowing them to be dispatched to the appropriate endpoint.
<code>app.UseCors()</code>	Enables Cross-Origin Resource Sharing (CORS) and configures cross-origin policies for handling requests.
<code>app.UseAuthentication()</code>	Adds authentication middleware to the pipeline, allowing authentication of incoming requests.
<code>app.UseAuthorization()</code>	Adds authorization middleware to the pipeline, allowing authorization of incoming requests based on policies.
<code>app.UseSession()</code>	Enables the use of session state in the application, storing and retrieving session data for each user.
<code>app.UseMiddleware<TMiddleware>()</code>	Adds a custom middleware component of type <code>TMiddleware</code> to the pipeline.
<code>app.UseEndpoints()</code>	Configures endpoint routing for the application, mapping URLs to specific actions or handlers.
<code>app.UseEndpoints(configureEndpoints)</code>	An overload of <code>UseEndpoints()</code> that allows more advanced configuration of endpoints.

6.4 CUSTOM MIDDLEWARE

- Custom middleware in ASP.NET Core allows you to extend the request/response pipeline with your own logic.
- It provides flexibility to handle specific requirements or implement cross-cutting concerns unique to your application.
- Custom middleware is typically used to perform tasks such as logging, request/response transformation, caching, header manipulation, and more.
- There are two main types of custom middleware you can create in ASP.NET Core:
 - Inline Middleware
 - Class-based Middleware

6.4.1 INLINE MIDDLEWARE:

- Inline middleware is defined using a `Func<HttpContext, Func<Task>, Task>` delegate.
- This approach allows you to write middleware logic directly within the `Program.cs` file.

Example:

```
app.Use(async (context, next) =>
{
```

```
// Custom middleware logic

await next();

// Additional logic after the next middleware
});
```

6.4.2 CLASS-BASED MIDDLEWARE:

- Class-based middleware is implemented as a separate class.
- We can create class based middleware in two different ways,
 - Using Request Delegate
 - Using IMiddleware interface

Request Delegate:

- The custom middleware class should have a constructor that accepts a RequestDelegate parameter.
- It allowing the middleware to invoke the next delegate in the pipeline.
- This allows the request to continue processing through the subsequent middleware components.
- In InvokeAsync method, you can write custom logic that executes before and after invoking the next delegate.
- This gives you control over the request/response flow and allows you to perform tasks such as logging, request/response transformation, error handling, and more.

Example:

```
public class CustomMiddleware
{
    private readonly RequestDelegate _next;

    public CustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        // Custom middleware logic before invoking the next delegate

        await _next(context);
    }
}
```

ASP.NET Core

```

        // Custom middleware logic after invoking the next delegate
    }
}

```

IMiddleware:

- The IMiddleware interface provides a more structured approach to create middleware components.
- It combines the constructor injection and InvokeAsync method in a single interface.

Example:

```

public class CustomMiddleware : IMiddleware
{
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        // Custom middleware logic before the next middleware

        await next(context);

        // Custom middleware logic after the next middleware
    }
}

```

6.4.3 HOW TO ADD CUSTOM MIDDLEWARE IN REQUEST-RESPONSE PIPELINE:

- To add this custom middleware to the pipeline you can do following steps.
- First you need to register dependency in Program.cs file.
- After you need to use the UseMiddleware extension method in the Program.cs file.

Example:

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

// Register dependency
builder.Services.AddTransient<CustomMiddleware>();
// Or
builder.Services.AddScoped<CustomMiddleware>();
// Or
builder.Services.AddSingleton<CustomMiddleware>();

var app = builder.Build();

```

ASP.NET Core

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for
    // production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseMiddleware<CustomMiddleware>();

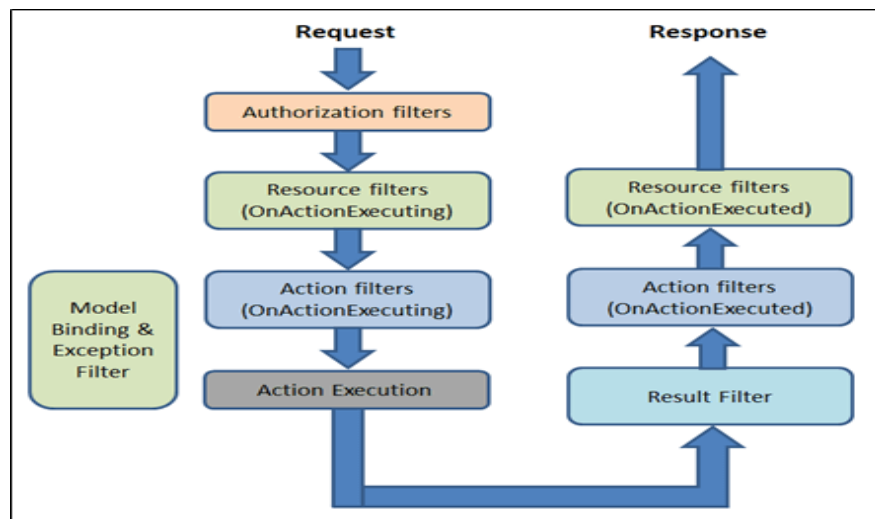
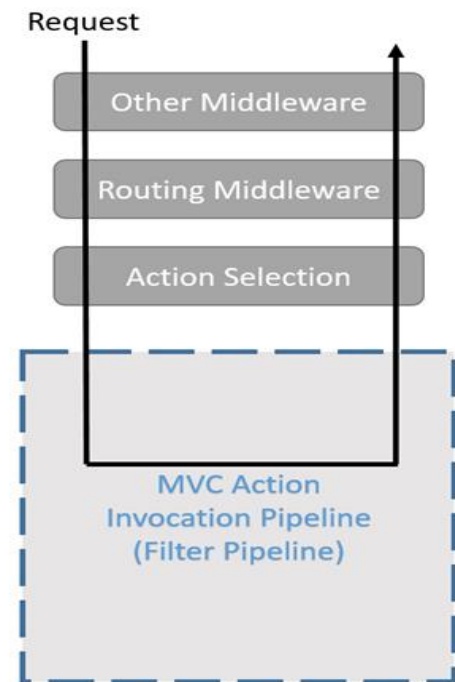
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```


FILTER

7.1 INTRODUCTION

- Filters allow us to run custom code before or after executing the action method.
- They provide ways to do common repetitive tasks on our action method.
- The filters are invoked on certain stages in the request processing pipeline.
- There are many built-in filters available with ASP.NET Core MVC, and we can create custom filters as well.
- Filters help us to remove duplicate codes in our application.
 - Authorization filter
 - Resource filter
 - Action filter
 - Exception filter
 - Result filter



Authorization filter:

- The Authorization filters are executed first.
- This filter helps us to determine whether the user is authorized for the current request or not.

- It can short-circuit a pipeline if a user is unauthorized for the current request.
- We can also create custom authorization filter.

Resource filters:

- 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.

Action filters:

- The Action filters run the code immediately before and after the 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.

Exception filters:

- The Exception filters are used to handle exception that occurred before anything written to the response body.

Result filters:

- The Result filters are used to run code after the execution of controller action results.
- They are executed only if the controller action method has been executed successfully.
- Filter supports two types of implementation.
 - Synchronous
 - Asynchronous

Synchronous:

- 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.

Example:

```
using Microsoft.AspNetCore.Mvc.Filters;  
namespace Filters
```

```
{  
    public class CustomActionFilter : IActionFilter  
    {  
        public void OnActionExecuting(ActionExecutingContext context)  
        {  
            //To do : before the action executes  
        }  
  
        public void OnActionExecuted(ActionExecutedContext context)  
        {  
            //To do : after the action executes  
        }  
    }  
}
```

Asynchronous:

- 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.

Example:

```
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Mvc.Filters;  
  
namespace Filters  
{  
    public class CustomAsyncActionFilter : IAsyncActionFilter  
    {  
        public async Task OnActionExecutionAsync(ActionExecutingContext  
context, ActionExecutionDelegate next)  
        {  
            //To do : before the action executes  
            await next();  
            //To do : after the action executes  
        }  
    }  
}
```

- 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.

Dhruvil A. Dobariya

ASP.NET Core

- 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.

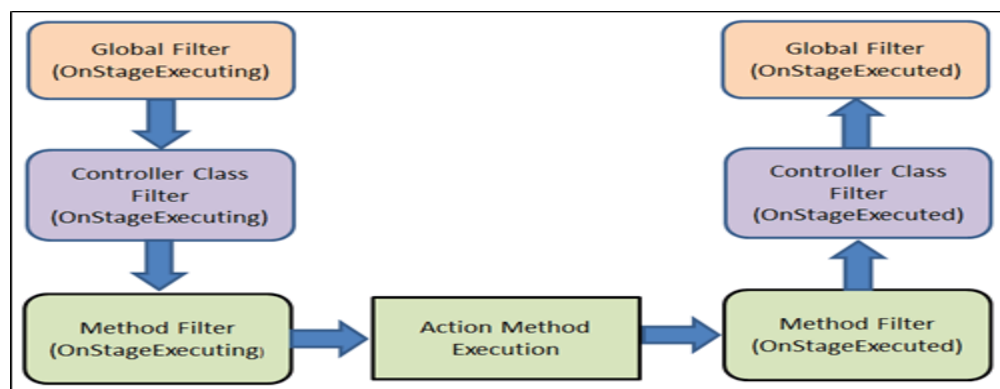
7.2 SCOPE AND ORDER

- A filter can be added to the pipeline at one of three scopes.
 - By action method
 - By controller class
 - Globally (which be applied to all the controller and actions).
- We need to register filters in to the MvcOption.Filters collection within ConfigureServices method.

Example:

```
// Add framework services.
builder.Services.AddMvc(options =>
{
    //an instant
    options.Filters.Add(new CustomActionFilter());
    //By the type
    options.Filters.Add(typeof(CustomActionFilter));
});
```

- When multiple filters are applied to the particular stage of the pipeline, scope of filter defines the default order of the filter execution.



- The global filter is applied first, then class level filter is applied and finally method level filter is applied.

Overriding the default order:

ASP.NET Core

- 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.

Example:

```
// ExampleFilter.cs

using System;
using Microsoft.AspNetCore.Mvc.Filters;
namespace Filters
{
    public class ExampleFilterAttribute : Attribute, IActionFilter,
    IOrderedFilter
    {
        public int Order { get; set; }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            //To do : before the action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            //To do : after the action executes
        }
    }
}
```

```
// HomeController.cs

using System;
using Microsoft.AspNetCore.Mvc;
using Filters;

namespace Filters.Controllers
{
    [ExampleFilter(Order = 1)]
    public class HomeController : Controller
```

ASP.NET Core

```
{
    public IActionResult Index()
    {
        return View();
    }
}
```

- When filters are run in pipeline, filters are sorted first by order and then scope.
- All built-in filters are implemented by `IOrderFilter` and set the default filter order to 0.

7.3 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.

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
namespace Filters
{
    public class Example1FilterAttribute : Attribute, IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            //To do : before the action executes
            context.Result = new ContentResult()
            {
                Content = "Short circuit filter"
            };
        }
        public void OnActionExecuted(ActionExecutedContext context)
        {
            //To do : after the action executes
        }
    }
}
```

7.4 DEPENDENCY INJECTION IN FILTERS

- As we learned, the filter can be added by the type or by the instance.
- If we added filter as an instance, this instance will be used for every request and if we add filter as a type, instance of the type will be created for each request.

ASP.NET Core

- Filter has constructor dependencies that will be provided by the DI.
- The filters that are implemented as attributes and added directly to the controller or action methods, cannot have constructor dependencies provided by the DI.
- In this case, contractor parameter must be supplied when they are applied.
- This is a limitation of attribute.
- There are many way to overcome this limitation.
- We can apply our filter to the controller class or action method using one of the following,
 - ServiceFilterAttribute
 - TypeFilterAttribute
 - IFilterFactory implemented on attribute

ServiceFilterAttribute:

- A ServiceFilter retrieves an instance of the filter from dependency injection (DI).
- We need to add this filter to the container in Program.cs and reference it in a ServiceFilter attribute in the controller class or action method.
- One of the dependencies we might require to get from the DI, is a logger. Within filter, we might need to log something happened.

Example:

```
// Filter
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;
namespace FiltersSample.Filters
{
    public class ExampleFilterWithDI : IActionFilter
    {
        private ILogger _logger;
        public ExampleFilterWithDI(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<ExampleFilterWithDI>();
        }
        public void OnActionExecuting(ActionExecutingContext context)
        {
            //To do : before the action executes
            _logger.LogInformation("OnActionExecuting");
        }
        public void OnActionExecuted(ActionExecutedContext context)
        {
            //To do : after the action executes
            _logger.LogInformation("OnActionExecuted");
        }
    }
}
```

```
}  
}  
}
```

```
// Register filter in Program.cs  
builder.Services.AddScoped<ExampleFilterWithDI>
```

```
// Use filter for Action method of Controller class  
[ServiceFilter(typeof(ExampleFilterWithDI))]  
public IActionResult Index()  
{  
    return View();  
}
```

TypeFilterAttribute:

- It is very similar to ServiceFilterAttribute and also implemented from IFilterFactory interface.
- Here, type is not resolved directly from the DI container but it instantiates the type using class "Microsoft.Extensions.DependencyInjection.ObjectFactory".
- Due to this difference, the types are referenced in TypeFilterAttribute need to be register first in Program.cs.
- The "TypeFilterAttribute" can be optionally accept constructor arguments for the type.

Example:

```
[TypeFilter(typeof(ExampleFilterAttribute), Arguments = new object[]  
{ "Argument if any" })]  
public IActionResult About()  
{  
    return View();  
}
```


DEPENDENCY INJECTION

8.1 INTRODUCTION

- Dependency injection (DI) is a design pattern used in software development to implement inversion of control (IoC) between components.
- This pattern promotes loose coupling and modular design by separating the creation and management of dependencies from the dependent components.

8.1.1 DI ROLES:

- In DI, there are typically three main roles involved,
 - Dependency
 - Dependent
 - Injector

Dependency:

- A dependency is an object or service that is required by a component to perform its functionality.
- Dependencies can be other classes, services, configurations, or any external resource needed by the component.

Dependent:

- A dependent is a component that relies on one or more dependencies to fulfill its responsibilities.
- It doesn't create or manage the dependencies itself but expects them to be provided from the outside.

Injector:

- The injector is responsible for creating and managing the dependencies and injecting them into the dependent components.
- It is responsible for satisfying the dependencies when creating instances of the dependent components.

8.1.2 ADVANTAGES:

Loose Coupling:

- Reduces coupling and allows for easy swapping or replacement of dependencies.

Modular development:

- Components can focus on specific functionality, leading to easier testing, reusability, and extensibility.

Testability:

- Easier unit testing with the ability to mock or substitute dependencies, enabling reliable and efficient testing.

Scalability and flexibility:

- Manages complex dependencies and accommodates changing requirements by adding or replacing dependencies without modifying dependent components.

8.2 IOC CONTAINER

- In ASP.NET Core, the built-in IoC (Inversion of Control) container is used to manage dependencies and provide instances of classes or services to other parts of your application.
- It helps to decouple components, improve testability, and promote modular development.

8.3 LIFETIME OF SERVICES

- Service lifetime determines how long an instance of a registered service should exist.
- It affects the behavior and performance of our application.
- We have three types of lifetime,
 - Transient
 - Scoped
 - Singleton

Transient:

- Services are created each time they are requested.
- It gets a new instance of the injected object, on each request of this object.
- Every time you inject this object is injected in the class, it will create a new instance.
- Suitable for lightweight and stateless services.

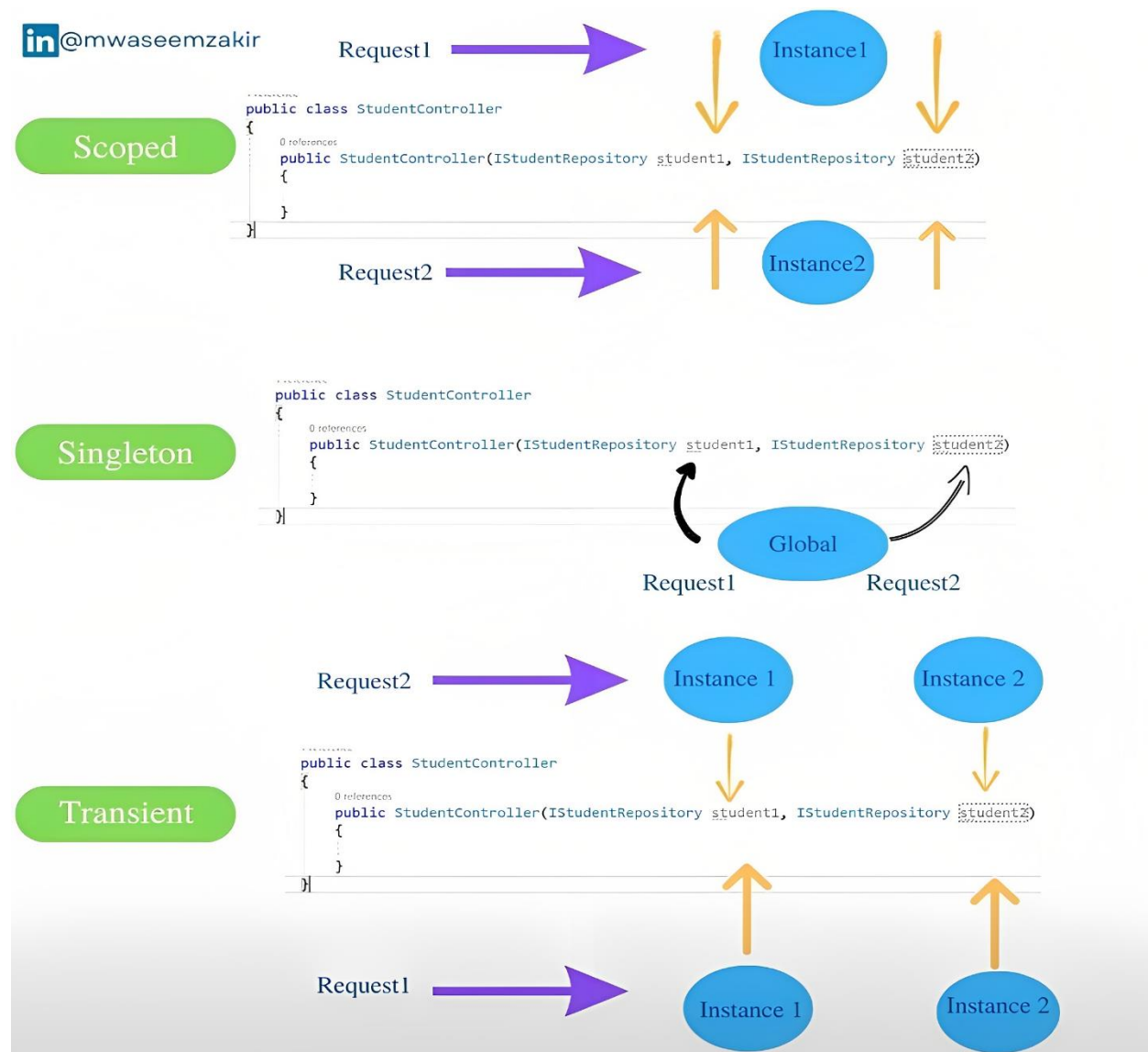
Scoped:

ASP.NET Core

- A single instance is created per request, and it's shared across components within the same HTTP request.
- Suitable for services that hold request-specific data.

Singleton:

- Services are created once for the lifetime of the application.
- A single instance is created for the entire application and shared across all requests.
- Suitable for stateless services or services that hold global state.



ASP.NET Core

8.3.1 EXTENSION METHODS:

- ASP.NET Core provides three different extension method for three different lifetime.
 - AddTransient
 - AddScoped
 - AddSingleton
- It's namespace is "Microsoft.Extensions.DependencyInjection".
- It help us to register dependency in Program.cs file.

Example:

```
// normal service
builder.Services.AddTransient<IMyService, MyService>();
builder.Services.AddScoped<IMyService, MyService>();
builder.Services.AddSingleton<IMyService, MyService>();

// generic service
builder.Services.AddTransient(typeof(IMyService<>), typeof(MyService<>));
builder.Services.AddScoped(typeof(IMyService<>), typeof(MyService<>));
builder.Services.AddSingleton(typeof(IMyService<>), typeof(MyService<>));
```

- We can have various way to register dependency for different usage.

Registration Method	Automatic Object Disposal	Multiple Implementations	Pass Arguments
services.AddSingleton<IMyDep, MyDep>();	Yes	Yes	No
services.AddSingleton<IMyDep>(sp => new MyDep());	Yes	Yes	Yes
services.AddSingleton<MyDep>();	Yes	No	No
services.AddSingleton<IMyDep>(new MyDep());	No	Yes	Yes
services.AddSingleton(new MyDep());	No	No	Yes

- AddSingleton is used as an example here, but the same patterns apply to other lifetime options like AddTransient and AddScoped.

8.4 CONSTRUCTOR INJECTION

- Once your services are registered, we can inject them into our classes or controllers using constructor injection.

ASP.NET Core

- The IoC container will automatically resolve and provide the required dependencies when creating an instance of a class.

```
public class MyController : Controller
{
    private readonly IMyService _myService;

    public MyController(IMyService myService)
    {
        _myService = myService;
    }
}
```

EXCEPTION HANDLING

9.1 INTRODUCTION

- We have two middleware for handling exception.
 - UseDeveloperExceptionPage
 - UseExceptionHandler

9.2 USEDEVELOPEREXCEPTIONPAGE

- This middleware is typically used during development to provide detailed error information for debugging purposes.
- When an exception occurs, it captures the exception details and generates an HTML error page with stack traces, exception messages, and other relevant information.
- This page is displayed in the browser for the developer to analyze and troubleshoot the issue.
- In this page we see five different component,
 - Stack,
 - Query
 - Cookies
 - Header
 - Routing

Stack:

- The Stack tab gives the information of stack trace which indicated where exactly the exception occurred, the file name, and the line number that causes the exception.

Query:

- The Query tab gives information about the query strings

Cookies:

- The Cookies tab displays the information about the cookies set by the request.

Header:

- The Header tab gives information about the headers which is sent by the client when makes the request.

Routing:

- The Route tab gives information about the Route Pattern and Route HTTP Verb type of the method, etc.

Example:

```
app.UseDeveloperExceptionPage();
```

ExceptionHandler Setup Gui | Home | Microsoft 365 | 4. Exception Handling.docx | DotNet Core Training.xlsx | Internal Server Error

localhost:44398/Home/Calculate?a=10&b=0

GitHub | Microsoft 365 | YouTube | .NET documentation | Bootstrap 5

An unhandled exception occurred while processing the request.

DivideByZeroException: Attempted to divide by zero.

ExceptionHandlerLearn.Controllers.HomeController.Calculate(int a, int b) in HomeController.cs, line 25

Stack Query Cookies Headers Routing

DivideByZeroException: Attempted to divide by zero.

ExceptionHandlerLearn.Controllers.HomeController.Calculate(int a, int b) in HomeController.cs

25. return Ok(a / b);

lambda_method18(Closure, object, object[])

Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor+SyncActionResultExecutor.Execute(ActionResultTypeMapper mapper, ObjectMethodExecutor executor, object controller, object[] arguments)

Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeActionMethodAsync()

Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(ref State next, ref Scope scope, ref object state, ref bool isCompleted)

Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeNextActionFilterAsync()

Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Rethrow(ActionExecutedContextSealed context)

Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(ref State next, ref Scope scope, ref object state, ref bool isCompleted)

Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync()

Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResourceFilter>g__Awaited|25_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope, object state, bool isCompleted)

Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResourceExecutedContextSealed context)

Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Next(ref State next, ref Scope scope, ref object state, ref bool isCompleted)

Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.InvokeFilterPipelineAsync()

Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Awaited|17_0(ResourceInvoker invoker, Task task, IDisposable scope)

ExceptionHandler Setup Gui | Home | Microsoft 365 | 4. Exception Handling.docx | DotNet Core Training.xlsx | Internal Server Error

localhost:44398/Home/Calculate?a=10&b=0

GitHub | Microsoft 365 | YouTube | .NET documentation | Bootstrap 5

An unhandled exception occurred while processing the request.

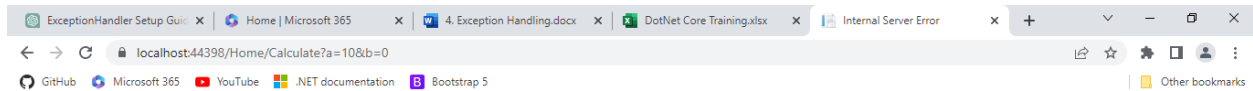
DivideByZeroException: Attempted to divide by zero.

ExceptionHandlerLearn.Controllers.HomeController.Calculate(int a, int b) in HomeController.cs, line 25

Stack **Query** Cookies Headers Routing

Variable	Value
a	10
b	0

ASP.NET Core



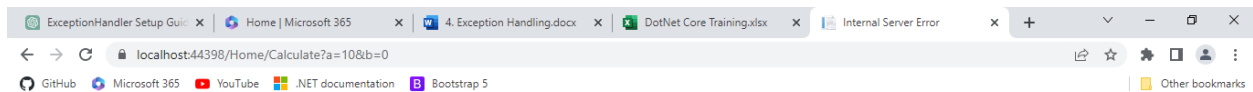
An unhandled exception occurred while processing the request.

DivideByZeroException: Attempted to divide by zero.

ExceptionHandlingLearn.Controllers.HomeController.Calculate(int a, int b) in HomeController.cs, line 25

Stack Query **Cookies** Headers Routing

No cookie data.



An unhandled exception occurred while processing the request.

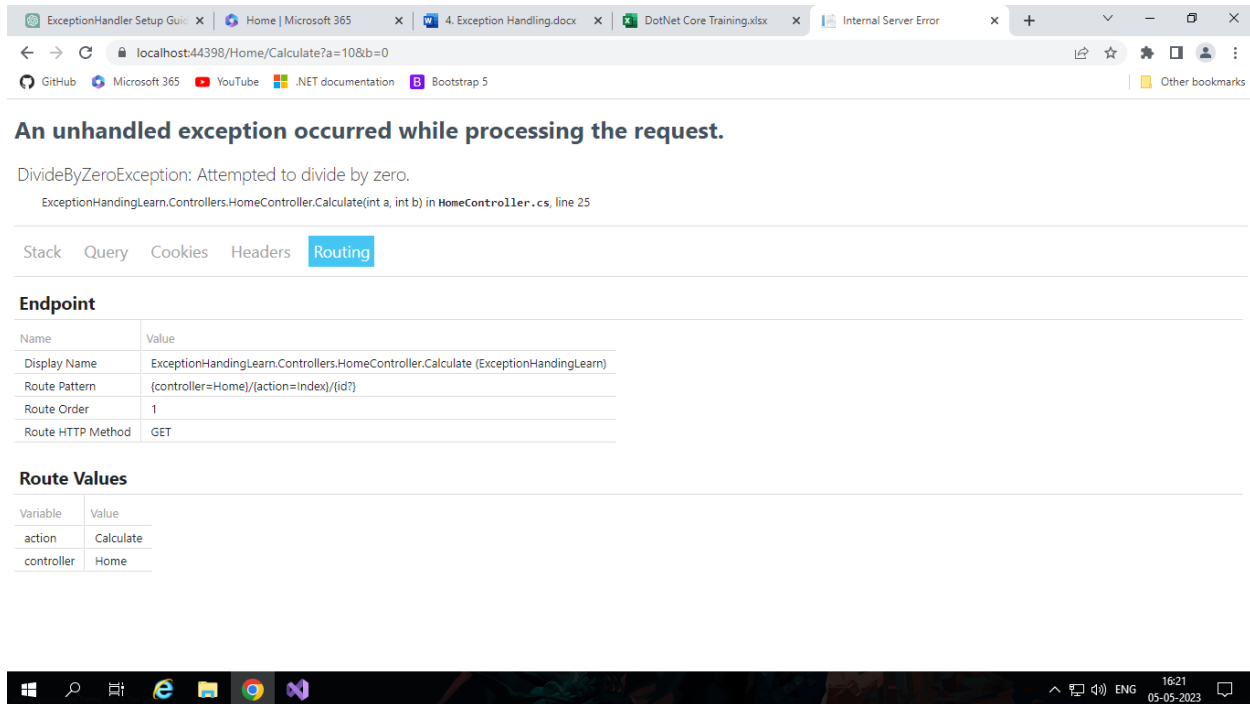
DivideByZeroException: Attempted to divide by zero.

ExceptionHandlingLearn.Controllers.HomeController.Calculate(int a, int b) in HomeController.cs, line 25

Stack Query Cookies **Headers** Routing

Variable	Value
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding	gzip, deflate, br
Accept-Language	en-US,en;q=0.9
Connection	close
Host	localhost:44398
sec-ch-ua	"Chromium";v="112", "Google Chrome";v="112", "Not-A-Brand";v="99"
sec-ch-ua-mobile	?0
sec-ch-ua-platform	"Windows"
sec-fetch-dest	document
sec-fetch-mode	navigate
sec-fetch-site	none
sec-fetch-user	?1
upgrade-insecure-requests	1
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.0.0 Safari/537.36





An unhandled exception occurred while processing the request.

DivideByZeroException: Attempted to divide by zero.

ExceptionHandlingLearn.Controllers.HomeController.Calculate(int a, int b) in HomeController.cs, line 25

Stack Query Cookies Headers **Routing**

Endpoint

Name	Value
Display Name	ExceptionHandlingLearn.Controllers.HomeController.Calculate (ExceptionHandlingLearn)
Route Pattern	(controller=Home)/(action=index)/{id?}
Route Order	1
Route HTTP Method	GET

Route Values

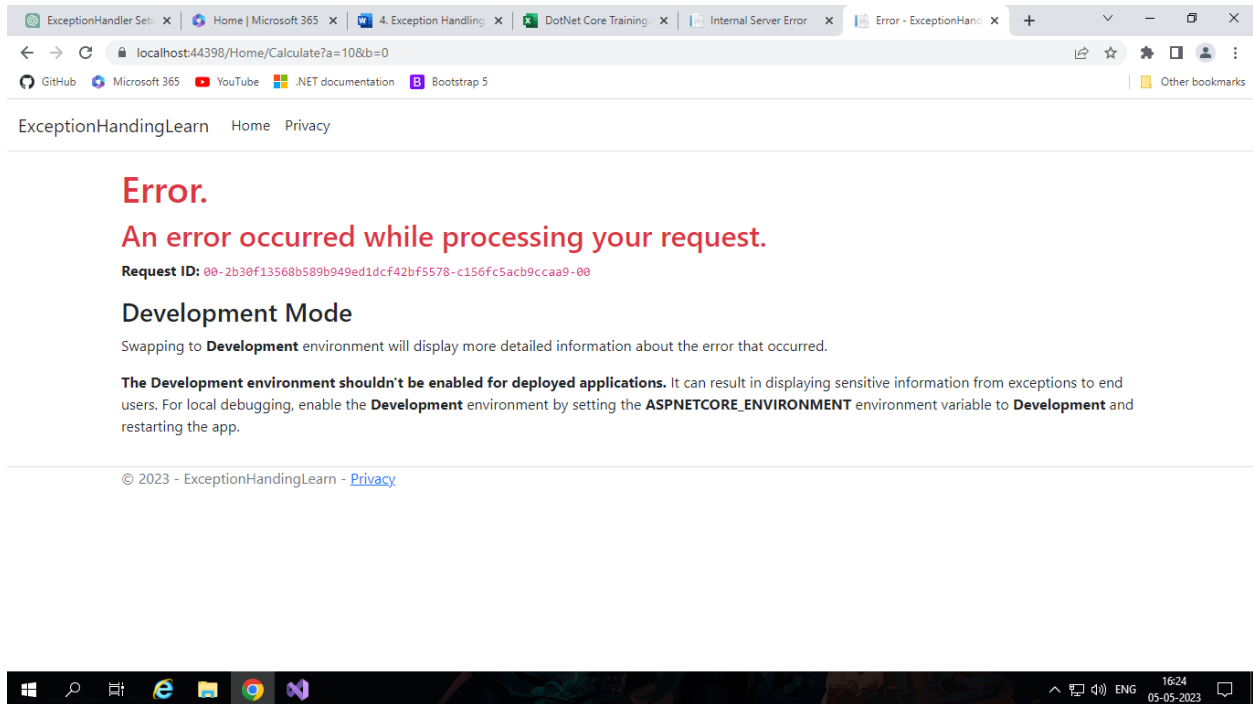
Variable	Value
action	Calculate
controller	Home

9.3 USEEXCEPTIONHANDLER

- This middleware is used in production environments to handle exceptions and display a custom error page or response.
- It allows you to specify a delegate or a custom error handling endpoint to handle exceptions and generate an appropriate response.
- This middleware is responsible for catching unhandled exceptions and executing the provided error handling logic.

Example:

```
app.UseExceptionHandler("/error");
// or
app.UseExceptionHandler(builder =>
{
    builder.Run(async context =>
    {
        // Custom error handling logic
        await context.Response.WriteAsync("An error occurred. Please try again later.");
    });
});
```



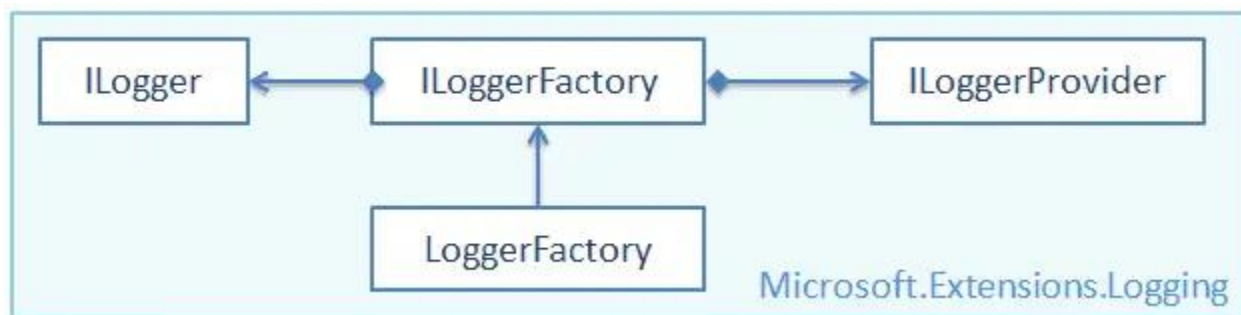
LOGGING API

10.1 INTRODUCTION

- ASP.NET Core provides a built-in logging API that allows us to easily log messages and exceptions in our applications.
- The logging API is designed to be extensible and flexible, with support for multiple logging providers and the ability to customize logging configuration.
- There are two important building blocks for implementing logging in a .NET Core based application.
 - Logging API
 - Logging Providers

10.2 LOGGING API

- Microsoft provides logging API as an extension in the wrapper Microsoft.Extensions.Logging which comes as a NuGet package.
- Microsoft.Extensions.Logging includes the necessary classes and interfaces for logging.
- The most important are the
 - ILogger,
 - ILoggerFactory
 - ILoggerProvider
 - LoggerFactory



10.2.1 ILOGGINGFACTORY:

- The ILoggerFactory is the factory interface for creating an appropriate ILogger type instance and also for adding the ILoggerProvider instance.
- The Logging API have the built-in LoggerFactory class that implements the ILoggerFactory interface.

ASP.NET Core

- We can use it to add an instance of type ILoggerProvider and to retrieve the ILogger instance for the specified category.

Syntax:

```
public interface ILoggerFactory : IDisposable
{
    ILogger CreateLogger(string categoryName);
    void AddProvider(ILoggerProvider provider);
}
```

10.2.2 ILoggerProvider:

- It is interface that is used for create our custom LoggerProvider by implementing interface.

Syntax:

```
public interface ILoggerProvider : IDisposable
{
    ILogger CreateLogger(string categoryName);
}
```

10.2.3 ILogger:

- This interface used to get different methods for log.

```
public interface ILogger
{
    void Log<TState>(LogLevel logLevel, EventId eventId, TState state,
        Exception exception, Func<TState, Exception, string> formatter);
    bool IsEnabled(LogLevel logLevel);
    IDisposable BeginScope<TState>(TState state);
}
```

10.3 LOGGING PROVIDERS

- We have some build in logging providers.

Logging Provider	Description
Console	Writes log messages to the console. Useful for development and debugging.
Debug	Writes log messages to the debug output window in Visual Studio. Useful for debugging and troubleshooting.
EventSource	Writes log messages to the Windows Event Log using the EventSource API. Useful for high-performance logging with low overhead.

Dhruvil A. Dobariya

EventLog	Writes log messages to the Windows Event Log using the <code>System.Diagnostics.EventLog</code> class. Useful for logging critical events that require monitoring by system administrators.
TraceSource	Writes log messages to the <code>System.Diagnostics.Trace</code> class. Useful for tracing and debugging complex application logic.
Azure App Service	Writes log messages to the Azure App Service logging system. Useful for cloud-based applications hosted on Azure.
Application Insights	Writes log messages to Microsoft Azure Application Insights, which is a cloud-based application performance monitoring service. Useful for monitoring and analyzing application performance in production environments.

- We also can use many third-party logging providers available for ASP.NET Core, such as Serilog, NLog, and Log4Net, among others.
- These providers offer additional features and customization options beyond what is provided by the built-in providers.
- We can also create your own custom logging providers by implementing the `ILoggerProvider` interface.
- We can provider with the logging system using the `AddProvider` method.
- This allows us to integrate with other logging systems or to implement custom logging behaviour.