2. ASP.NET Core Request Processing Pipeline

❖ Request Pipeline Overview

The ASP.NET Core Request Processing Pipeline, often called the “Middleware Pipeline,” is a sequence of middleware components that handle an incoming HTTP request and generate an appropriate HTTP response in an ASP.NET Core Web application.

The Request Processing Pipeline plays a crucial role in processing requests and performing various tasks such as routing, authentication, authorization, caching, logging, and more. Each middleware component in the pipeline processes the request in a specific way and can modify the request or response as needed.

Every .NET core web app uses a Startup class to bootstrap the application. Startup class has two methods:

- ➢ **Configure Services**, to configure the dependencies
- ➢ **Configure**, to configure the request processing pipeline. This is the part which decides which middlewares would be invoked.

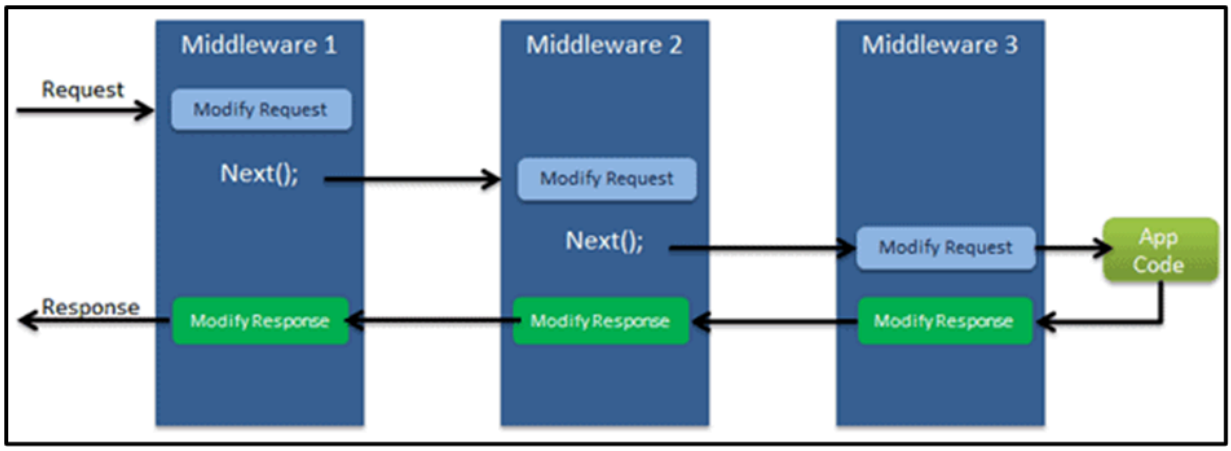
The middlewares are invoked in the order in which they are configured in the Configure method.

Below diagram shows overview of recommended order of middlewares for .NET Core web application. It includes some middlewares which we may not want to use with the .NET core web API app.

❖ Middleware

- ➢ ASP.NET Core introduced a new concept called Middleware.
- ➢ A middleware is nothing but a component (class) which is executed on every request in ASP.NET Core application.

- In the classic ASP.NET, HttpHandlers and HttpModules were part of the request pipeline.
- Middleware is similar to HttpHandlers and HttpModules where both need to be configured and executed in each request.
- there will be multiple middleware in ASP.NET Core web applications. It can be either framework provided middleware, added via NuGet or your own custom middleware.
- We can set the order of middleware execution in the request pipeline.
- Each middleware adds or modifies http request and optionally passes control to the next middleware component.
- But a middleware component can decide not to call the next piece of middleware in the pipeline. This is called **short-circuiting** or terminating the request pipeline.
- Short-circuiting is often desirable because it avoids unnecessary work.



Some common middleware components in the ASP.NET Core pipeline include:

- **Routing Middleware:** Handles URL routing and maps requests to appropriate controller actions or Razor Pages.
- **Authentication Middleware:** Handles user authentication and identity management.
- **Authorization Middleware:** Enforces access control rules and permissions.
- **Static Files Middleware:** Serves static files (e.g., CSS, JavaScript, images) directly from the file system.

- **CORS Middleware:** Enforces Cross-Origin Resource Sharing (CORS) policies.
- **Logging Middleware:** Logs information about the request and response.
- **Response Compression Middleware:** Compresses responses before sending them to the client.
- **Exception Handling Middleware:** Handles unhandled exceptions and provides error responses.

❖ Configure Middleware

➢ We can configure middleware in the Configure method of the Startup class using the IApplicationBuilder instance.

➢ The following example adds a single middleware using Run method which returns a string "Hello World!" on each request.

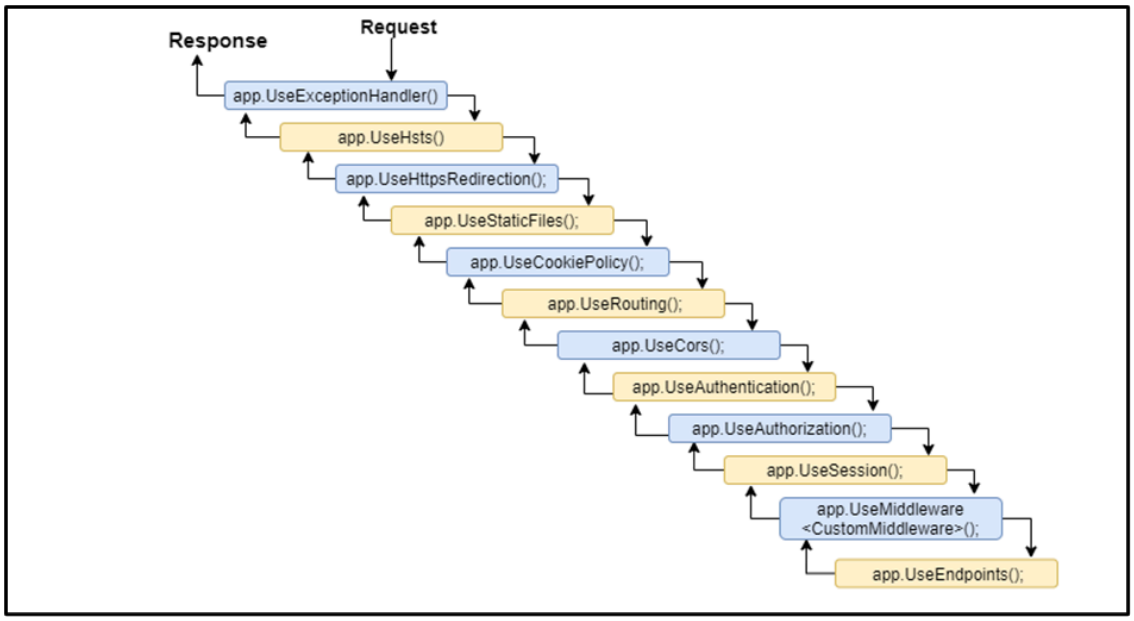
```
public class Startup
{
    public Startup()
    {
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
    {
        //configure middleware using IApplicationBuilder here..

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");

        });

        // other code removed for clarity..
    }
}
```

➢ In the above example, Run() is an extension method on an IApplicationBuilder instance which adds a terminal middleware to the application's request pipeline.

➢ The above configured middleware returns a response with a string "Hello World!" for each request

❖ Middleware Ordering

- ➢ Middleware components are executed in the order they are added to the pipeline and care should be taken to add the middleware in the right order otherwise the application may not function as expected.
- ➢ This ordering is critical for security, performance, and functionality.
- ➢ The first configured middleware has received the request, modified it (if required), and passes control to the next middleware.
- ➢ Similarly, the first middleware is executed at the last while processing a response if the echo comes back down the tube.
- ➢ That's why Exception-handling delegates need to be called early in the pipeline, so they can validate the result and displays a possible exception in a browser and client-friendly way.



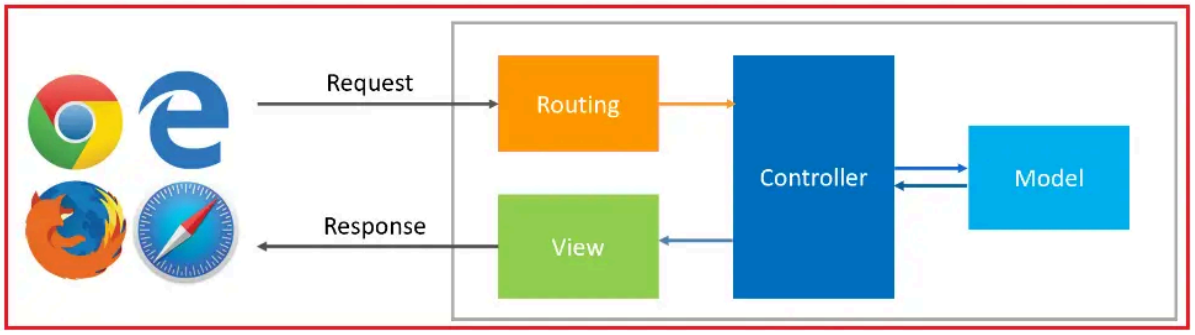
❖ Built-in Middleware Via NuGet

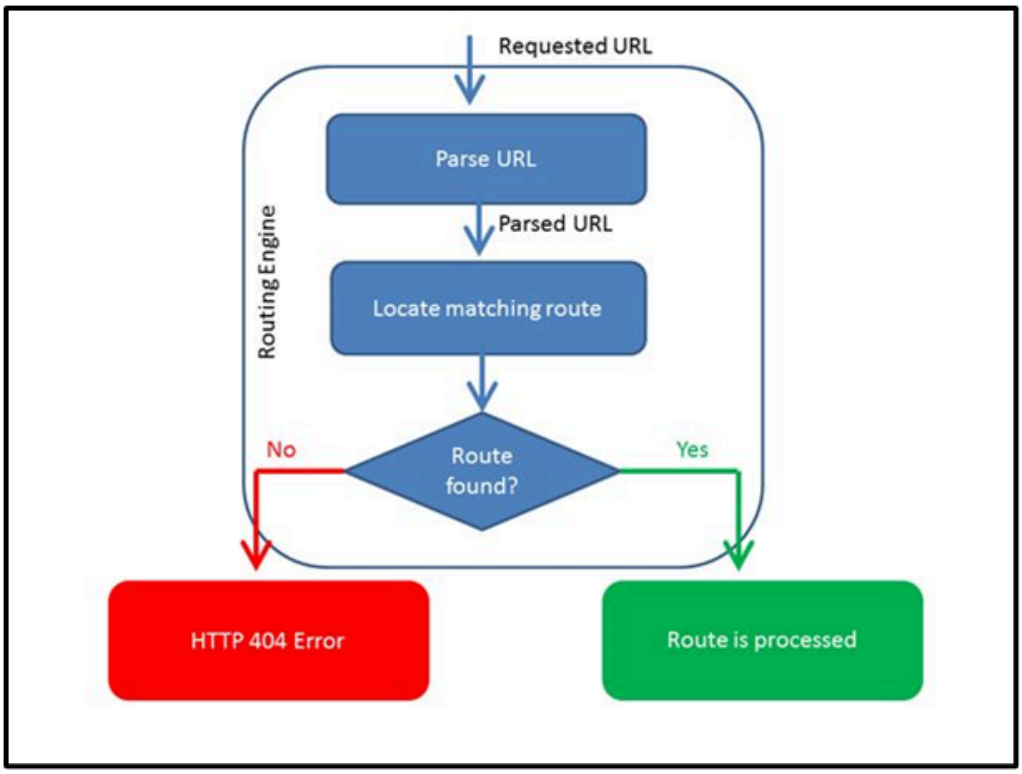
- ➢ ASP.NET Core is a modular framework. We can add server side features we need in our application by installing different plug-ins via NuGet. There are many middleware plug-ins available which can be used in our application.

Middleware	Description
Authentication	Adds authentication support.
CORS	Configures Cross-Origin Resource Sharing.
Routing	Adds routing capabilities for MVC or web form
Session	Adds support for user session.
Static Files	Adds support for serving static files and directory browsing.
Diagnostics	Adds support for reporting and handling exceptions and errors.

❖ Routing

- ➢ Routing is a pattern matching system that monitors the incoming request and figures out what to do with that request.
- ➢ Typically, it is a way to serve the user request.
- ➢ When a user requests URLs from the server then URLs are handled by the routing system.
- ➢ The Routing system tries to find out the matching route pattern of requested URL with already registered routes which are map to controller, actions, files, or other items.
- ➢ If there is a matching route entry, then it processes the request i.e. serve the resource, otherwise it returns a 404 error.





❖ Types of Routing

There are two main ways to define routes in ASP.NET Core:

- ➢ **1. Convention-based Routing**
- ➢ **2. Attribute Routing**

❖ Convention-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 Startup.cs file.

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

```
public class ProductController : Controller
{
    //Product
    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }
    //Product/Create
    0 references | 0 requests | 0 exceptions
    public IActionResult Create()
    {
        return View();
    }
    //Product/Edit/1
    0 references | 0 requests | 0 exceptions
    public IActionResult Edit(int id)
    {
        return View();
    }
}
```

- **Attribute 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

```
[Route("Home")]
0 references
public class HomeController : Controller
{
    [Route("")]       // "Home"
    [Route("Index")] // "Home/Index"
    [Route("/")]      //  ""
    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }
    [Route("About")] // "Home/About"
    0 references | 0 requests | 0 exceptions
    public IActionResult About()
    {
        return View();
    }
}
```

Difference Between Convention Based Routing and Attribute Routing

Point	Attribute Routing	Convention Based Routing
Meaning	Attribute routing, introduced in later versions of ASP.NET, allows for more granular control over the URLs by defining routes directly on actions and controllers using attributes. This approach offers greater flexibility in defining routes, including specifying complex route patterns and constraints directly on the action methods or controllers	Conventional routing uses a predefined pattern to map requests to controller actions. Conventional Routing involves defining routes in the Program.cs file. This approach relies on a predefined pattern to match the URL paths to the corresponding controller actions. It typically specifies placeholders for the controller name, action name, and optional parameters. This approach is useful for applications with a clear and consistent URL structure across the entire application.
Characteristics	Routes are defined using attributes on controllers and actions, allowing for detailed and customized route definitions. It offers greater flexibility in defining routes that do not follow a conventional pattern. Supports the definition of complex route templates, including using multiple parameters, constraints, and even dynamically generating URLs.	Routes are defined globally in a centralized location, typically in the Startup.cs file. It uses route templates that are associated with controllers and actions implicitly through naming conventions. Parameters can be specified within the route templates to capture values from the URL.
Advantages	Provides greater control and precision over routing, allowing for complex and custom route patterns that are not easily achieved with conventional routing. Facilitates the organization of routing	Simplifies route management by centralizing route definitions, making it easier to understand and manage the application's URL structure. Ideal for large applications with a hierarchical URL structure that

	logic by keeping it closer to the action methods it applies to, improving readability and maintainability.	follows a consistent pattern.
Disadvantages	This can lead to a scattered routing configuration if not managed carefully, as routes are defined across different controllers and actions. It requires more effort to understand the routing configuration as it is not centralized.	Less flexible in defining granular or action-specific routes directly on controllers or actions. This can lead to more complex route management in highly dynamic URL pattern applications.
When to Use	**Fine-grained URL Control:** When you need precise control over your application's URL patterns, attribute routing is the way to go. It's particularly useful for designing RESTful APIs where URLs represent resources and operations on those resources. **Complex Route Patterns:** Attribute routing shines when dealing with complex routes that conventional routing struggles with, such as routes that need to capture multiple parameters or have custom constraints. **Versioning APIs:** For applications that involve API versioning, attribute routing makes it easier to define different routes for different versions of the API within the same application. **Mixing Static and Dynamic Segments:** If you need to mix static URL segments with dynamic ones in a single route, attribute routing provides the flexibility to accomplish this easily.	**Large Applications with Standardized URL Patterns:** Conventional routing is ideal for large applications with many controllers and actions but standardized URL patterns. It reduces the need to decorate each action method with route attributes, keeping the code cleaner. **Centralized Route Configuration:** If you prefer to have a centralized location where routes are defined and managed, conventional routing provides this advantage. It allows for easier changes to route patterns without modifying individual controllers or actions. **Simple CRUD Operations:** For applications that follow a straightforward CRUD (Create, Read, Update, Delete) pattern, conventional routing can be more straightforward and less verbose than attribute routing.

❖ Filters

- ➢ 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.
- ➢ We can create custom filters as well.
- ➢ Filters help us to remove duplicate codes in our application.

❖ Advantages and Disadvantages of Filters in ASP.NET Core MVC

- ➢ Filters in ASP.NET Core MVC Application provide several advantages and disadvantages. Understanding the advantages and disadvantages of Filters is essential for making decisions about when and how to use them in your application. Here are the advantages and disadvantages of Filters in ASP.NET Core MVC:

➢ Advantages of Filters in ASP.NET Core MVC:

- **Modularity and Reusability:** Filters in ASP.NET Core allow us to encapsulate and separate concerns in our application, which makes our code more modular and easier to maintain. We can define filters for specific requirements of our application (e.g., Authentication, Authorization, Logging, Exception Handling, etc.) and apply them selectively to controllers or actions. This also promotes code reusability as we defined the filter once and applied it at multiple places.
- **Cross-Cutting Concerns:** Filters in ASP.NET Core provide a centralized and consistent way to handle cross-cutting concerns, such as Authentication, Authorization, Logging, Error Handling, Caching, etc, without duplicating code.
- **Customization and Extensibility:** ASP.NET Core MVC Framework provides many built-in filters that we can use directly in our controllers and action methods. It also allows us to create Custom filters to address specific requirements unique to your application, which are not served by the built-in filters.

- **Separation of Concerns:** Filters in ASP.NET Core help us to maintain the separation of concerns principle in our application architecture. By applying filters to specific actions or controllers, we can ensure that each application component focuses on its primary responsibility, making the codebase more maintainable and easier to understand.
- **Consistency and Standardization:** Filters in ASP.NET Core allow us to enforce consistent behavior and standards across our application. For example, we can use authorization filters to ensure that only authenticated users with specific roles can access certain actions. This consistency is especially important for security and compliance.
- **Performance Optimization:** Filters in ASP.NET Core Application can also be used for performance optimization tasks, such as caching frequently used data or results. Result filters, for instance, allow us to cache the output of an action and serve it directly from the cache for subsequent requests, reducing the load on our application and improving response times.
- **Security:** Filters in ASP.NET Core Application are essential for implementing security measures like authentication and authorization. Authorization filters can restrict access to specific actions or resources based on user roles, permissions, or policies, helping protect sensitive parts of our application.
- **Error Handling:** Exception Filters in the ASP.NET Core Application help us to handle unhandled exceptions, allowing us to log errors, provide user-friendly error messages, and maintain the stability of our application, providing different error pages based on different status codes, etc.
- **Logging and Monitoring:** Filters can also be used for logging important information about requests and responses, facilitating monitoring, debugging, and auditing of our application's behavior.

❖ Disadvantages of Filters in ASP.NET Core MVC:

- **Complexity:** As filters can be applied at different levels (Action, Controller, Global), and multiple filters can be combined, it can lead to increased complexity in understanding the overall behavior of a controller or action. Once you understand the concepts, then it will not be a problem at all. Yes, initially, you will find it a little difficult to understand.
- **Ordering:** Managing the execution order for multiple filters can be challenging, especially when filters with conflicting behaviors are applied to the same action.
- **Maintenance Overhead:** While filters promote code Reusability and Modularity, at the same time, having many filters can also make your application difficult to understand, maintain, and debug.

- **Performance Overhead:** Filters add a slight performance overhead to request processing by introducing additional method calls and processing steps. However, this overhead is usually negligible for most applications.
- **Limited Scope:** Filters operate within the ASP.NET Core MVC Framework's Request Processing Pipeline and may not be suitable for handling tasks that require broader control over the entire request processing, such as low-level routing or response compression. In that case, you might use Middleware Components.

❖ Filter Types

- ➢ Every filter type is executed at a different stage in the filter pipeline. Following are the filter types.
- ➢ 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.

Synchronous Filter Example

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

Asynchronous filter example

```
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
        }
    }
}
```

- **Authorization filters**
  - Are the first filters run in the filter pipeline.
  - Control access to action methods.
  - Have a before method, but no after method.
  - Custom authorization filters require a custom authorization framework.
  - Prefer configuring the authorization policies or writing a custom authorization policy over writing a custom filter.
  - The built-in authorization filter:
    - Calls the authorization system.
    - Does not authorize requests.
    - Do not throw exceptions within authorization filters:
    - The exception will not be handled.
    - Exception filters will not handle the exception.
- · **Resource filters**
  - Implement either the IResourceFilter or IAsyncResourceFilter interface.
  - Execution wraps most of the filter pipeline.
  - Only Authorization filters run before resource filters.
  - Resource filters are useful to short-circuit most of the pipeline. For example, a caching filter can avoid the rest of the pipeline on a cache hit.
- · **Action filters**
  - Implement either the IActionFilter or IAsyncActionFilter interface.
  - Their execution surrounds the execution of action methods.
  - The following code shows a sample action filter:

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

The ActionExecutingContext provides the following properties:

- Action Arguments - enables reading the inputs to an action method.
- Controller - enables manipulating the controller instance.
- Result - setting Result short-circuits execution of the action method and subsequent action filters.

Throwing an exception in an action method:

- Prevents running of subsequent filters.
- Unlike setting Result, is treated as a failure instead of a successful result.

The ActionExecutedContext provides Controller and Result plus the following properties:

- Cancelled - True if the action execution was short-circuited by another filter.
- Exception - Non-null if the action or a previously run action filter threw an exception. Setting this property to null:
  - Effectively handles the exception.
  - Result is executed as if it was returned from the action method.

The OnActionExecuting action filter can be used to:

- Validate model state.
- Return an error if the state is invalid.

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

The OnActionExecuted method runs after the action method:

- ➢ And can see and manipulate the results of the action through the Result property.
- ➢ Cancelled is set to true if the action execution was short-circuited by another filter.
- ➢ Exception is set to a non-null value if the action or a subsequent action filter threw an exception. Setting Exception to null:
  - Effectively handles an exception.
  - ActionExecutedContext.Result is executed as if it were returned normally from the action method.

❖ · **Exception filters**

- ➢ Implement IExceptionFilter or IAsyncExceptionFilter.
- ➢ Can be used to implement common error handling policies.

The following sample exception filter displays details about exceptions that occur when the app is in development:

```
public class SampleExceptionFilter : IExceptionFilter
{
    private readonly IHostEnvironment _hostEnvironment;

    public SampleExceptionFilter(IHostEnvironment hostEnvironment) =>
        _hostEnvironment = hostEnvironment;

    public void OnException(ExceptionContext context)
    {
        if (!_hostEnvironment.IsDevelopment())
        {
            // Don't display exception details unless running in Development.
            return;
        }

        context.Result = new ContentResult
        {
            Content = context.Exception.ToString()
        };
    }
}
```

- Exception filters:
  - Don't have before and after events.
  - Implement OnException or OnExceptionAsync.
  - Handle unhandled exceptions that occur in Razor Page or controller creation, model binding, action filters, or action methods.
  - Do not catch exceptions that occur in resource filters, result filters, or MVC result execution.

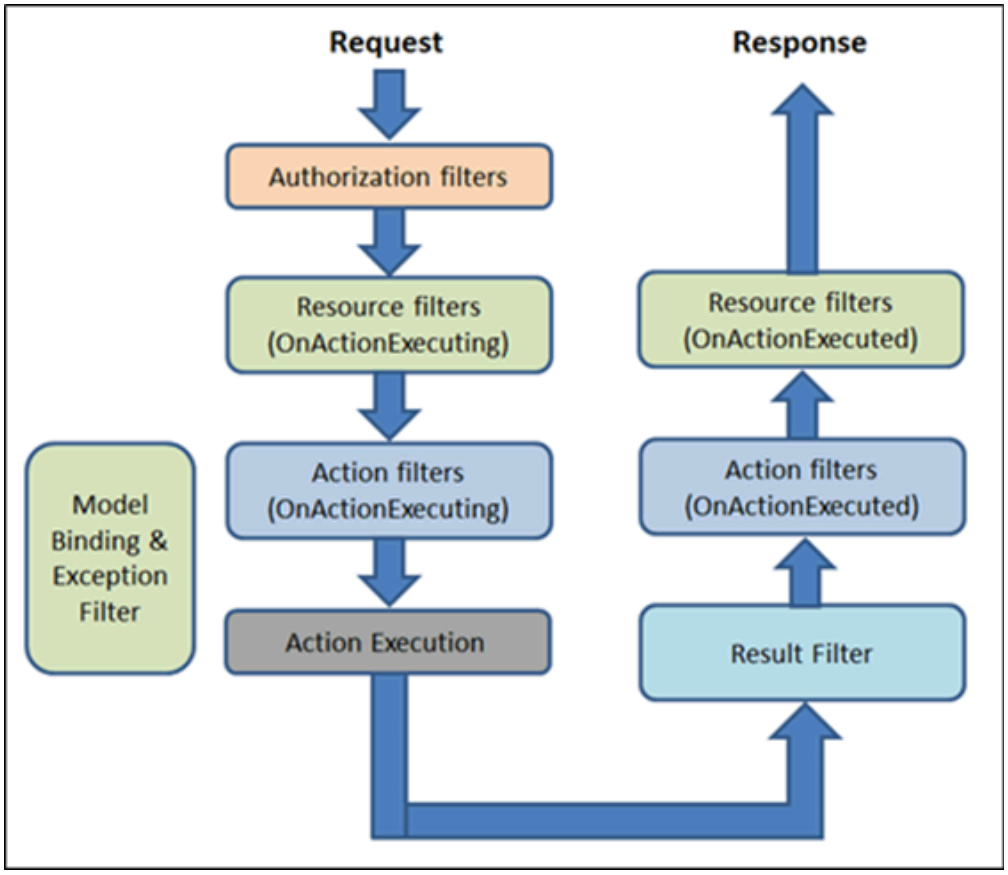
To handle an exception, set the ExceptionHandled property to true or assign the Result property. This stops propagation of the exception. An exception filter can't turn an exception into a "success". Only an action filter can do that.

- **Result filters**
  - Implement an interface:
    - IResultFilter or IAsyncResultFilter
    - IAlwaysRunResultFilter or IAsyncAlwaysRunResultFilter
  - Their execution surrounds the execution of action results.

```
public class SampleResultFilter : IResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        // Do something before the result executes.
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Do something after the result executes.
    }
}
```

The kind of result being executed depends on the action. An API method might perform some serialization as part of the execution of the result. Result filters are only executed when an action or action filter produces an action result. Result filters are not executed when:

- An authorization filter or resource filter short-circuits the pipeline.
- An exception filter handles an exception by producing an action result.



❖ Controller Initialization

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET Core MVC.

❖ What is a Controller?

- ➢ A controller is used to define and group a set of actions.
- ➢ An action (or action method) is a method on a controller which handles requests. Controllers logically group similar actions together.
- ➢ This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively.
- ➢ Requests are mapped to actions through routing.

❖ A controller is an instantiable class, usually public, in which at least one of the following conditions is true:

- ➢ The class name is suffixed with Controller.

- ➢ The class inherits from a class whose name is suffixed with Controller.
- ➢ The [Controller] attribute is applied to the class.
- ➢ A controller class must not have an associated [NonController] attribute.
- ➢ The controller is a UI-level abstraction. Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned.
- ➢ In well-factored apps, it doesn't directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.
- ➢ Every controller must be derived from this **abstract** Controller class. This base Controller class contains helper methods that can be used for various purposes.

❖ Controller Helper Methods

Controllers usually inherit from Controller, although this isn't required. Deriving from Controller provides access to three categories of helper methods:

❖ Methods resulting in an empty response body

- ➢ No Content-Type HTTP response header is included, since the response body lacks content to describe.
- ➢ There are two result types within this category: Redirect and HTTP Status Code.

❖ HTTP Status Code

- ➢ This type returns an HTTP status code.
- ➢ A couple of helper methods of this type are BadRequest, NotFound, and Ok.
- ➢ For example, return BadRequest(); produces a 400 status code when executed. When methods such as **BadRequest**, **NotFound** and 200 for **Ok** are overloaded, they no longer qualify as HTTP Status Code responders, since content negotiation is taking place.

➢ Redirect

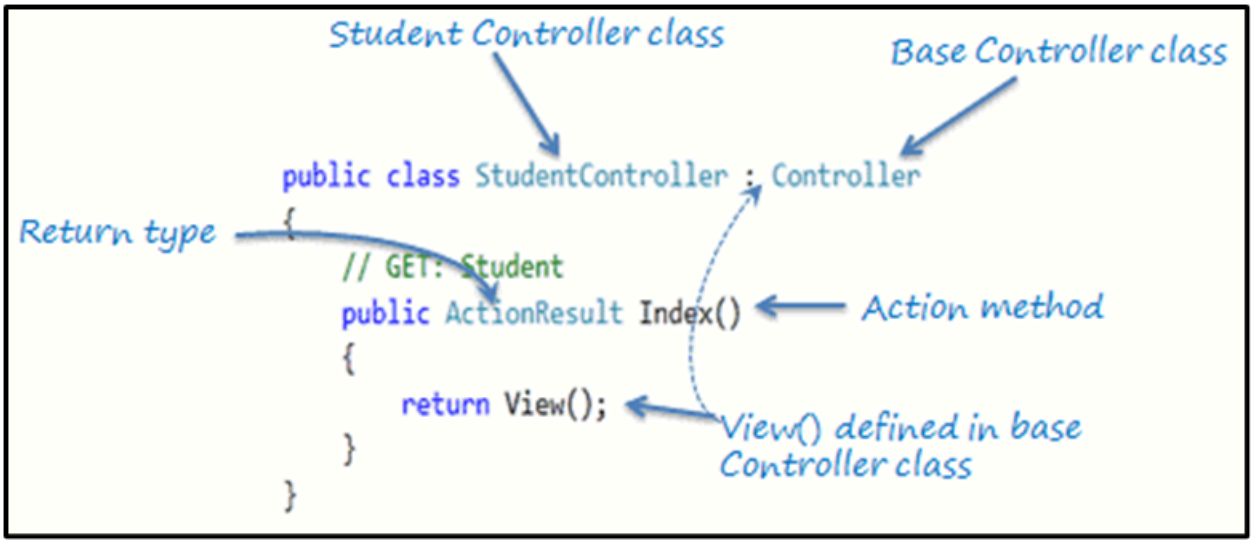
- ➢ This type returns a redirect to an action or destination (using Redirect, LocalRedirect, RedirectToAction, or RedirectToRoute).

- For example, return RedirectToAction("Complete", new {id = 123}); redirects to Complete, passing an anonymous object.
- The Redirect result type differs from the HTTP Status Code type primarily in the addition of a Location HTTP response header.

❖ Action method

All the public methods of the Controller class are called Action methods. They are like any other normal methods with the following restrictions:

- Action method must be public.
- Action method cannot be overloaded, a static method and private or protected.



❖ The Index() method is public, and it returns the ActionResult using the View() method. The View() method (use in MVC Model) is defined in the Controller base class, which returns the appropriate ActionResult.

- All the public methods in the Controller class are called Action methods.
- The Action method has the following restrictions.
  - Action method must be public. It cannot be private or protected.
  - Action method cannot be overloaded.
  - Action method cannot be a static method.
- ActionResult is a base class of all the result type which returns from the Action method.

- The base Controller class contains methods that returns appropriate result types e.g. View(), Content(), File(), JavaScript() etc.
- The Action method can include Nullable type parameters.