## Middleware

Middleware in ASP.NET Core is a component that sits in the request processing pipeline and can perform actions on requests and responses. Middleware can be used for logging, authentication, error handling, and more. Here's how you can create custom middleware in ASP.NET Core:

Register Middleware in Configure method Startup.

**Middleware Order:**

Middleware order is important. The order in which you add middleware determines the order in which they execute. Middlewares added later in the Configure method will execute after the ones added earlier.

## Routing

In ASP.NET Core, there are two primary types of routing: Conventional Routing and Attribute Routing. Each has its own use cases and benefits. Here’s a detailed explanation of both:

**Conventional Routing**

Conventional Routing is typically defined in the **Startup.cs** file within the **Configure** method. This type of routing uses URL patterns to match incoming requests to route handlers.

Conventional Routing is useful for applications where the URL patterns follow a predictable and standard structure.

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

});

**Attribute Routing**

Attribute Routing uses attributes to define routes directly on controller actions. This approach provides more flexibility and allows for more complex routing scenarios.

Attribute Routing is beneficial when you need fine-grained control over the URLs and is often used in RESTful APIs.

endpoints.MapControllers();

**Combining Conventional and Attribute Routing**

It's possible to use both Conventional and Attribute Routing in the same application. You might use Conventional Routing for general page navigation and Attribute Routing for specific controllers or actions that require customized routes.

**Route Constraints**

Route constraints can be applied to ensure that certain routes only match if specific conditions are met. Constraints can be applied in both Conventional and Attribute Routing.

[Route("{year:int}/{month:int}/{day:int}")]

[Route("{id:int:min(1)}")]

## Filters in ASP.NET Core

* Filters avoid duplicating code.
* E.g., an error handling exception filter => **can join together into one whole** error handling code.
* Filters runs within ASP.NET core **action invocation pipeline** (filter pipeline).
* Filter pipeline runs after the action to execute is selected.
* Other Middleware -> Routing Middleware -> Action Selection -> Filter pipeline.

### Filter Types

* Filter type run hote hai different stage mai in filter pipeline.

#### Authorization Filter

* Runs first in the filter pipeline.
* Role: Check krta hai ki yhe user authorized bhi hai ki nahi iss request ke liye.
* If not then yahi se pipeline ko short-circuit kardo.

#### Resource Fiter

* Authorization ke baad run hota hai.
* OnResourceExecuting baki bachi filer pipeline ke pehle execute hoga, and runs before model binding.
* OnResourceExecuted runs code after rest of pipeline completes.

#### Action Filter

* Yhe run hota hai just pehle and badme action method is called.
* Yhe action ko jo argument jane wale thai, unme change kr skta hai.
* Yhe action se jo result return hua, usko bhi change kr skta hai.
* **Are not supported in Razor Pages.**

#### Endpoint Filter

* Yhe bhi run hota hai immediately action method is called.
* Yhe bhi action ko jo argument jane wale thai, unme change kr skta hai.
* Yhe bhi action se jo result return hua, usko bhi change kr skta hai.
* **Yhe bhi, are not supported in Razor Pages.**
* Yhe action and **route handler-based endpoint** par invoke ho skta hai.

#### Exception Filter

* Yhe apply krta hai global policy on unhandled exception => jo occur hote hai response ke body likne se pehle.

#### Result Filter

* Yhe run hota hai immediately before and after the execution of **action results**.
* Tabhi run hoga jab action method successfully execute hoga.

### Implementation of filter

* A filter can be implemented as Synchronously and Asynchronously, interface definition ke help se.
* Tho, synchronous filters runs before and after their pipeline stage.
* E.g., OnActionExecuting call hota hai before execution of action method. And OnActionExecuted call hota hai after the action method returns.
* Asynchronous filter, On-Stage-ExecutionAsync method define krta hai.
* E.g., OnActionExecutionAsync.

### Multiple Filter stages

* Ek class mai aap **multiple filter stage** implement kar skte ho.
* i.e., ActionFilterAttribute class can implement:
  + Synchronously – IActionFilter and IResultFilter
  + Asynchronously – IAsyncActionFilter and IAsyncResultFilter
  + IOrderFilter.
* Sync ya Async dono mai ke hi implement krna.
* Actually, hota yhe hai ki runtime check krta hai ki yhe class ne konse filter ko implement kiya hai, if it is implementing async tho vho use call karta hai. If not then it calls the synchronous interface’s method(s).
* Agar jyada chatur ban rahe thai, aur dono sync and async se implement kara diya, tho sirf async ke method ko hi execute karayga.
* Agar aap ActionFilterAttribute abstract class jesa kuch use kr rhe ho tho then, only override either synch or async method for each filter type.
* Kai filter interfaces ke corresponding attributes hote hai, jo ki kiya ja ske hai as base class for custom implementations.
* E.g., IActionFilter -> ActionFilterAttribute -> ResponseFilterAttribute.

### Filter ka Scope and uska order of execution

* Koi filter ko ham pipeline mai 3 scopes pe daal skte hai.
  + Attribute ki trh **controller** pe.
  + Attribute ki trh **controller ke action** pe (Razor pages ke handler methods pe filter attribute nahi laga skte).
  + And globally, sare controllers, actions and razor pages par.

## Middleware VS Filters

In ASP.NET Core, middleware and filters are two mechanisms that help handle requests and responses. They serve different purposes and are used in different contexts. Here’s a detailed comparison:

### Middleware

**Purpose:**

Middleware components are used to handle requests and responses at a lower level. They are typically used to perform tasks such as request logging, authentication, error handling, etc.

**Execution Order:**

Middleware is executed in the order it is registered in the Startup.Configure method. Each middleware can choose to pass control to the next middleware in the pipeline or short-circuit the request.

**Scope:**

Middleware applies to the entire application and can handle requests for any endpoint.

**Customization:**

Middleware components are typically created as classes that implement a method called **Invoke** or **InvokeAsync**.

**Examples:**

* Authentication
* Authorization
* Request logging
* Response caching
* Error handling

**Usage:**

* Use middleware when you need to handle cross-cutting concerns that affect the entire application or a significant portion of it.

### Filters

**Purpose:**

* Filters are used to add logic before or after specific stages of the MVC request processing pipeline, such as before or after action execution, result execution, or exception handling.

**Execution Order:**

* Filters are executed based on their type and order within each stage of the request pipeline. There are several types of filters: authorization filters, resource filters, action filters, exception filters, and result filters.

**Scope:**

* Filters are scoped to controllers or actions, meaning they apply only to specific controllers or actions where they are applied.

**Customization:**

* Filters are typically created by implementing one of the filter interfaces (**IAuthorizationFilter**, **IActionFilter**, **IResultFilter**, **IExceptionFilter**, **IResourceFilter**) or by deriving from one of the base classes.

**Examples:**

* Authorization
* Action execution timing
* Result modification
* Exception handling specific to MVC

**Usage:**

* Use filters when you need to handle concerns specific to MVC actions or controllers, such as authorization, model validation, or logging specific to MVC actions.

### When to Use Middleware vs. Filters

* **Middleware**:
* When you need to handle requests or responses at a global level.
* For tasks that are not specific to MVC, such as logging, authentication, and error handling.
* When you need to interact with lower-level HTTP features.
* **Filters**:
* When you need to handle tasks specific to MVC actions or controllers.
* For tasks such as action authorization, input validation, and action-specific logging.
* When you need fine-grained control over the execution of actions and results within MVC.

## Controller Initialization

In ASP.NET Core, controllers are initialized by the framework when an HTTP request is made to an endpoint that is associated with a controller action. The initialization process involves several steps:

1. **Controller Discovery**: ASP.NET Core automatically discovers controllers in your application. Controllers are classes that derive from **ControllerBase** or **Controller** and are typically located in the **Controllers** folder or namespace.
2. **Controller Activation**: When a request is made, ASP.NET Core activates the controller by creating an instance of it. This is typically done using the built-in dependency injection (DI) container. Controllers should have their dependencies (services, etc.) injected through their constructors.
3. **Action Selection**: Once the controller instance is created, ASP.NET Core selects the appropriate action method to handle the request. The action method is selected based on the HTTP method of the request (**GET**, **POST**, etc.) and the route template defined for the action.
4. **Model Binding**: Before the action method is invoked, ASP.NET Core performs model binding. Model binding maps data from the request (query string, form fields, route data, etc.) to parameters of the action method. This allows you to access the request data in a strongly-typed manner.
5. **Action Execution**: Finally, the selected action method is executed to process the request. The action method typically returns an **IActionResult** or a specific derived type (e.g., **ViewResult** for rendering a view, **JsonResult** for returning JSON data). The result of the action method is used to generate the HTTP response.
6. **Response Generation**: Based on the result of the action method, ASP.NET Core generates an HTTP response, which is sent back to the client.

## Action Method

In ASP.NET Core, an action method is a method in a controller class that is responsible for handling incoming HTTP requests. Action methods are where you write the logic to process requests, interact with data, and generate responses. Here are some key points about action methods:

1. **Method Signature**: Action methods are public methods in a controller class. They typically return an **IActionResult** or a specific result type (e.g., **ViewResult**, **JsonResult**, **RedirectToActionResult**) that represents the response to the request.
2. **Route Mapping**: Action methods are associated with specific URLs using routing. You can use attribute routing or convention-based routing to map URLs to action methods.
3. **Request Handling**: Action methods can accept parameters to access data from the request, such as query string parameters, form data, route values, and request headers. ASP.NET Core uses model binding to map these values to method parameters.
4. **Response Generation**: Action methods generate responses by returning an **IActionResult** or a specific result type. The result can be a view, a JSON response, a file download, or a redirect to another URL, among others.
5. **Dependency Injection**: Action methods can use dependency injection to access services. Dependencies are typically injected into the controller's constructor, and the action method can use these dependencies to perform its logic.