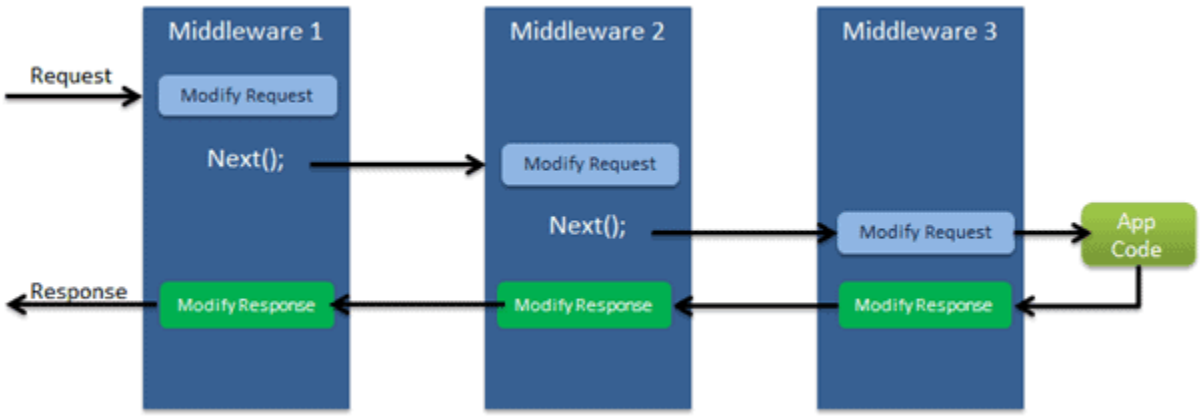
ASP.NET Core Request Processing Pipeline

1. Middleware

1.1 What is Middleware

- Middleware is a software component that sits between the web server and the application. It intercepts incoming requests and outgoing responses and allows you to modify them.
- Middleware can be used to perform a wide range of tasks such as authentication, logging, compression, and caching.
- Middleware works by using a pipeline model. When a request comes in, it is passed through a series of middleware components before it reaches the application.
- Each middleware component can modify the request before passing it on to the next component. Once the request has passed through all the middleware components, it reaches the application. Similarly, when a response is sent back to the client, it is passed through a series of middleware components before it is sent back to the client.



1.2 Types of Middleware

There are two types of middleware in .NET Core:

I. **Terminal middleware.**

The terminal middleware is responsible for sending the response back to the client. It is the final middleware component in the pipeline. Terminal middleware can be used to modify the outgoing response before it is sent back to the client.

- **Static files middleware:** This middleware is used to serve static files such as CSS, JavaScript, and images.

- **File server middleware:** This middleware is used to serve files from a specified directory.
- **MVC middleware:** This middleware is used to handle requests for MVC endpoints.

II. **Non-terminal middleware.**

Non-terminal middleware is any middleware component that is not the final component in the pipeline. Non-terminal middleware can be used to modify incoming requests and outgoing responses.

- **Authentication middleware:** This middleware is used to authenticate users.
- **Authorization middleware:** This middleware is used to authorize users to access certain resources.
- **Response compression middleware:** This middleware is used to compress the response before it is sent back to the client.
- **Request logging middleware:** This middleware is used to log requests.
- **Routing middleware:** This middleware is used to route requests to the appropriate endpoint.

1.3 Custom Middleware

- you can also create your own custom middleware components. Creating custom middleware allows you to add functionality to your application that is specific to your needs.
- To create custom middleware, you need to create a class that implements the IMiddleware interface. The IMiddleware interface has a single method called InvokeAsync that is called when the middleware component is invoked.

```
public class CustomHeaderMiddleware : IMiddleware
{
    public async Task InvokeAsync(HttpContext context, Func<Task> next)
    {
        context.Response.Headers.Add("X-Custom-Header", "Hello World");
        await next();
    }
}
```

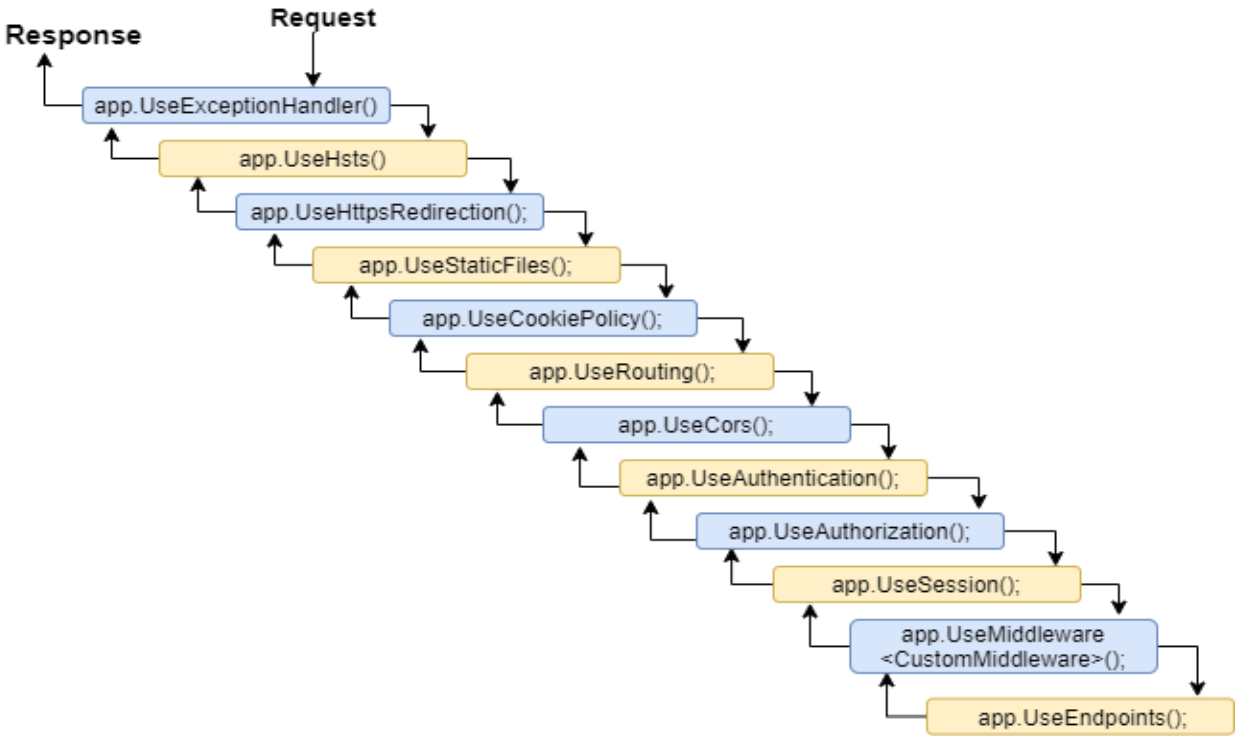
- The CustomHeaderMiddleware class implements the IMiddleware interface. The InvokeAsync method takes two parameters: the HttpContext and a Func<Task> delegate that represents the next middleware component in the pipeline.
- In this example, the middleware adds a custom header called “X-Custom-Header” to the outgoing response. It then calls the next middleware component in the pipeline by invoking the next delegate.
- To use the custom middleware component, you need to add it to the middleware pipeline in your application’s Startup class. You can do this by calling the UseMiddleware method on the IApplicationBuilder instance.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseMiddleware<CustomHeaderMiddleware>();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

- In this example, the UseMiddleware method is called with the CustomHeaderMiddleware class as the type parameter. This adds the custom middleware component to the pipeline before the routing middleware component.

1.4 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.



1.5 Understanding the Run, Use, and Map Method

1. **app.Run() Method**

   This middleware component may expose Run[Middleware] methods that are executed at the end of the pipeline. Generally, this acts as a terminal middleware and is added at the end of the request pipeline, as it cannot call the next middleware.

2. **app.Use() Method**

   This is used to configure multiple middleware, unlike app.Run(), We can include the next parameter into it, which calls the next request delegate in the pipeline. We can also short-circuit (terminate) the pipeline by not calling the next parameter.

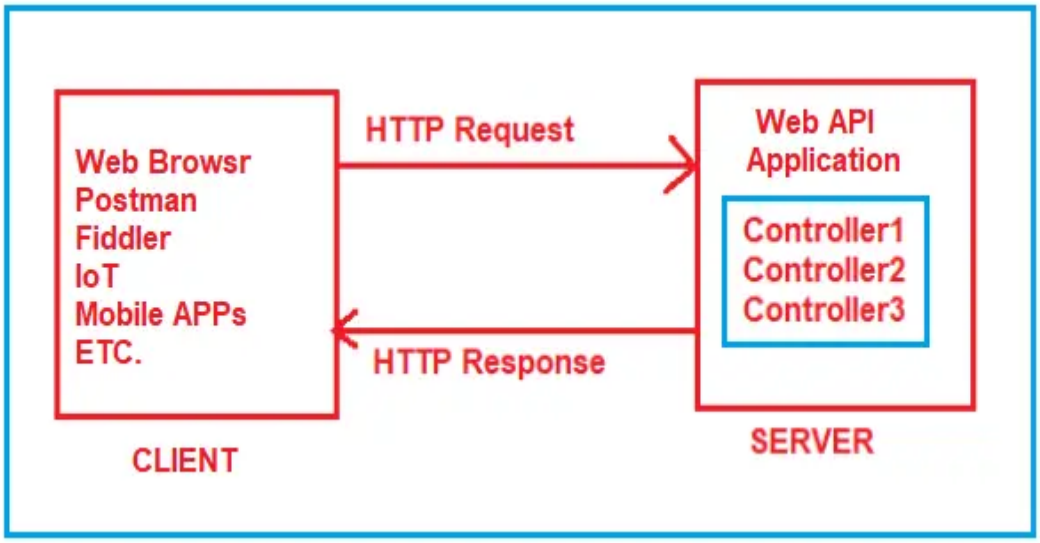
3. **app.Map() Method**

   These extensions are used as a convention for branching the pipeline. The map branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

2. Routing

2.1 What is routing

- Routing in is a fundamental concept that helps determine how incoming HTTP requests are matched to specific controller actions in your web application.



- Here, we need to understand how the application will come to know which request will be mapped to which controller action method. Basically, the mapping between the URL and resource (controller action method) is nothing but the concept of Routing.
- So, Routing in ASP.NET Core Web API is a concept that maps incoming HTTP requests to specific controllers and actions in your application. It determines how the Web API handles an HTTP request.

2.2 Configure Routing

- To Use Routing in your application you have to add service in your **ConfigureServices()** method.

```
services.AddControllers();
```

- To enable routing , you have to add given below middleware in your application's **Configure()** method.
  I. app.UseRouting();
  II. app.MapControllers();

I. **app.UseRouting():** This middleware enables routing capabilities in your ASP.NET Core application. It is responsible for matching incoming HTTP requests to routes that have been defined in your application. UseRouting should be added to the middleware pipeline before any middleware that requires knowledge of the requested endpoint, like authorization or endpoint-specific middleware.

II. **app.MapControllers():** This extension method is used to map attribute-routed controllers. It essentially tells the application to look for controllers in your project and creates routes for them based on the attributes you've defined (like [Route], [HttpGet], etc.). This is typically used when you have an API-centric application with controllers handling various HTTP requests.

2.3 Types of Routing

I. Convention-based Routing

Convention-based routing defines routes based on a set of conventions specified in the Program.cs file. Here, routes are defined globally for the application.

Advantages:

- Centralized Configuration: All routes are defined in one place, making it easier to see the big picture of the URL structure of your API.
- Consistency: By following conventions, the routes across different controllers and actions are consistent.

Disadvantages:

- Less Flexible: It is harder to define complex and custom routes that deviate from the established conventions.
- Refactoring Challenges: Changing controller or action names can break routes if not updated in the routing configuration.

Use Cases:

- It is best for applications with a straightforward and uniform URL structure.
- When you want a centralized place to manage routing.

II. Attribute Routing

Attribute routing uses attributes to define routes directly on controllers and actions. This approach provides more control by allowing custom and complex routes for each action.

Advantages:

- High Flexibility: Enables defining custom routes per action, allowing for complex URL structures.
- Self-documenting: Routes are defined where the action is defined, making it clear what URL maps to what action.
- Support for HTTP Verb Constraints: Easily specify HTTP methods (GET, POST, etc.) for actions.

Disadvantages:

- Scattered Configuration: Routes are spread across controllers and actions, making it harder to get an overview of the API's URL structure.
- Potential for Duplication: There is a possibility of duplicating route patterns, leading to conflicts or confusion.

Use Cases:

- Ideal for APIs with complex and non-uniform URL patterns.
- When you need fine-grained control over the routing of each action.
- Useful for versioning APIs by including the version in the route.

We can use both conventional and attribute-based Routing in ASP.NET Core Web API Applications. The action method that is decorated with a Route Attribute will use Attribute Routing, and the action method without the Route Attribute will use Conventional Routing.

3. Filters

3.1 What is Filter?

- Filters in ASP.NET Core allow code to run before or after specific stages in the request processing pipeline.
- Filters are used to inject extra logic at the different levels of MVC Framework request processing.
- Filters provide a way for cross-cutting concerns (logging, authorization, and caching).
- Filters allow you to modify the behavior of your MVC or Web API application without modifying the actual action method logic.

3.2 Configuration of Filter

1. Global Level (Applicable to all controllers and all action methods)
2. Controller Level (Applicable to all the action methods of the particular controller)
3. Action Level (Applicable to the specific action methods)

3.3 Types Of Filters

Filter Type	Interface	Description
Authentication	IAuthenticationFilter	These are Runs, before any other filters or the action method.
Authorization	IAuthorizationFilter	These Runs first, before any other filters or the action method.
Action	IActionFilter	These Runs before and after the action method.
Result	IResultFilter	Runs before and after the action result are executed.
Exception	IExceptionFilter	Runs only if another filter, the action method, or the action result throws an

1. **Authentication Filters**

   Authentication filter runs before any other filter or action method. Authentication confirms that you are a valid or invalid user. Action filters implement the IAuthenticationFilter interface.

```
public interface IAuthenticationFilter
{
    void OnAuthentication(AuthenticationContext filterContext);
    void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext);
}
```

2. **Authorization filters**

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

```
public interface IAuthorizationFilter
{
    void OnAuthorization(AuthorizationContext filterContext);
}
```

3. **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.

```
public interface IResourceFilter : IFilterMetadata
{
    void OnResourceExecuting(ResourceExecutingContext context);

    void OnResourceExecuted(ResourceExecutedContext context);
}
```

4. **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.

```
public interface IActionFilter
{
    void OnActionExecuted(ActionExecutedContext filterContext);
    void OnActionExecuting(ActionExecutingContext filterContext);
}
```

5. **Exception filters**

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

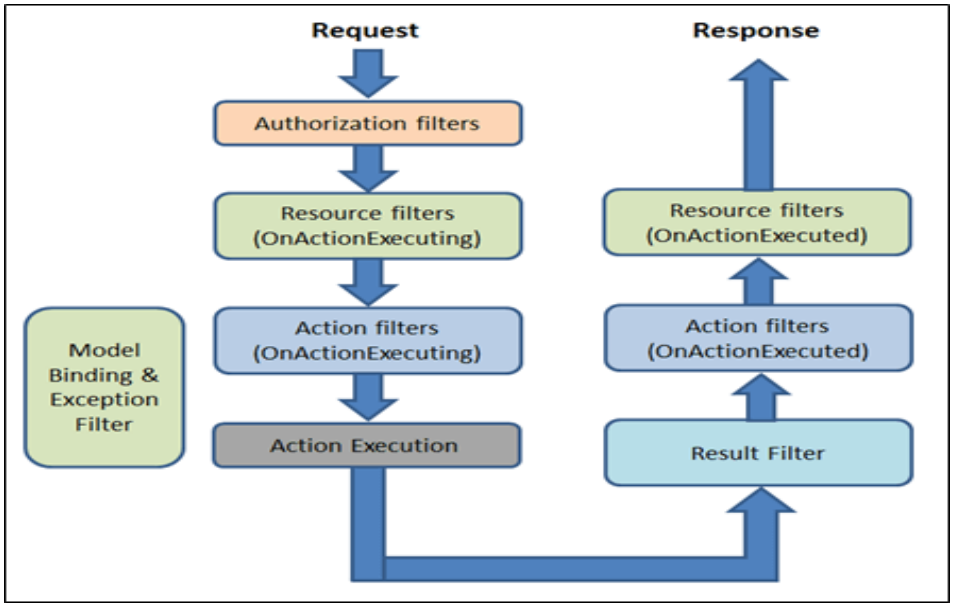
```
public interface IExceptionFilter
{
    void OnException(ExceptionContext filterContext);
}
```

6. **Result filters**

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

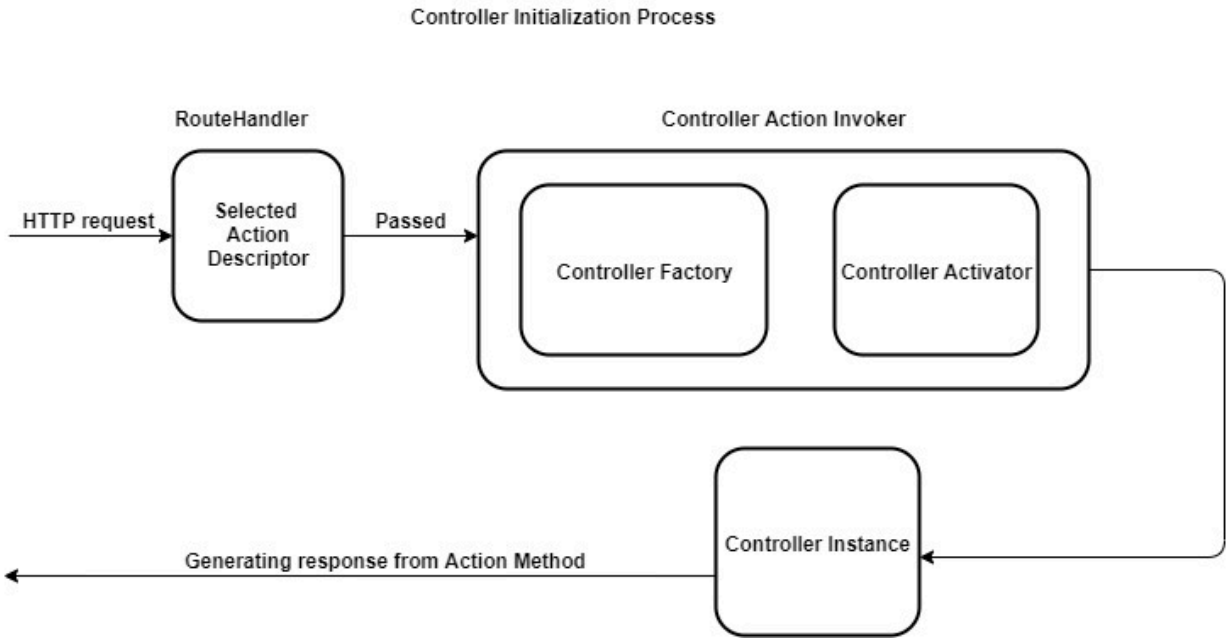
```
public interface IResultFilter
{
    void OnResultExecuted(ResultExecutedContext filterContext);
    void OnResultExecuting(ResultExecutingContext filterContext);
}
```

Order of Execution

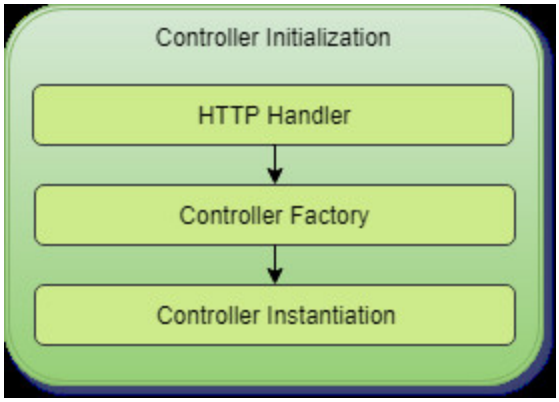


4. Controller Initialization

- Controllers are responsible for handling incoming requests which is done by mapping request to appropriate action method. The controller selects the appropriate action methods (to generate response) on the basis of route templates provided. A controller class inherits from controller base class. A controller class suffix class name with the Controller keyword.



- The MVC RouteHandler is responsible for selecting an action method candidate in the form of action descriptor.
- The RouteHandler then passes the action descriptor into a class called Controller action invoker.
- The class called Controller factory creates instance of a controller to be used by controller action method. The controller factory class depends on controller activator for controller instantiation.
- After action method is selected, instance of controller is created to handle request.
- Controller instance provides several features such as action methods, action filters and action result.
- The activator uses the controller type info property on action descriptor to instantiate the controller by name. once the controller is created, the rest of the action method execution pipeline can run.



- **HTTP Handler:**
  - An HTTP handler, often referred to simply as a handler, is a component responsible for processing incoming HTTP requests and generating corresponding responses.

- **Controller Factory:**
  - The controller factory is responsible for creating instances of controller classes to handle incoming requests.
  - The controller factory implements the IControllerFactory interface, which defines methods for creating and releasing controller instances
    - **i. CreateController**
      - This method is responsible for creating an instance of a controller class to handle an incoming request.
      - The 'CreateController' method is then invoked to create an instance of the appropriate controller class.
    - **ii. ReleaseController**
      - This method is responsible for releasing (or disposing of) the controller instance after the request has been handled.
      - Once the controller has finished processing the request and generating the response, the ReleaseController method is called to release any resources associated with the controller instance.
      - This may involve performing cleanup tasks, such as releasing database connections, closing file handles, or freeing up memory.

- The controller factory is responsible for determining which controller should handle a specific request based on the request's route and controller configuration.

- **Controller Instantiation:**
  - Controller instantiation refers to the process of creating an instance of a controller class to handle an incoming request.

5. Action Methods

5.1 What is Action Methods

- Action methods are responsible for handling HTTP requests and returning HTTP responses. They are the methods that perform specific actions in response to user requests.
- These methods are typically associated with URLs and are used to execute specific functionality based on the request type (e.g., GET, POST, PUT, DELETE).
- All the public methods of 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.

5.2 ActionResult

- MVC framework includes various Result classes, which can be returned from an action method.
- ActionResult is a base class of all the result type which returns from Action method.

Result Class	Description	Base Controller Method
ViewResult	Represents HTML and markup.	View()
EmptyResult	Represents No response.	
ContentResult	Represents string literal.	Content()
FileContentResult, FilePathResult, FileStreamResult	Represents the content of a file.	File()
JavaScriptResult	Represents a JavaScript script.	JavaScript()

JsonResult	Represents JSON that can be used in AJAX.	Json()
RedirectResult	Represents a redirection to a new URL.	Redirect()
RedirectToRouteResult	Represents redirection to another route.	RedirectToRoute()
PartialViewResult	Represents the partial view result.	PartialView()
HttpUnauthorizedResult	Represents HTTP 403 response.	