Middleware

- Middleware is a part of the requesting pipeline of api.
- When api is called at that time it first goes to the middleware pipeline.
- If middleware processes the request after that it will go to the action method.
- Else middleware detects an error then it will redirect from there to the user and stop the further processing.
- Middleware is a class that is called before the real execution starts of the action method of the controller.
- Middleware can be created by two ways in C#.
- First is using the IMiddleware interface and second is the default class that has RequestDelegate as property in that.

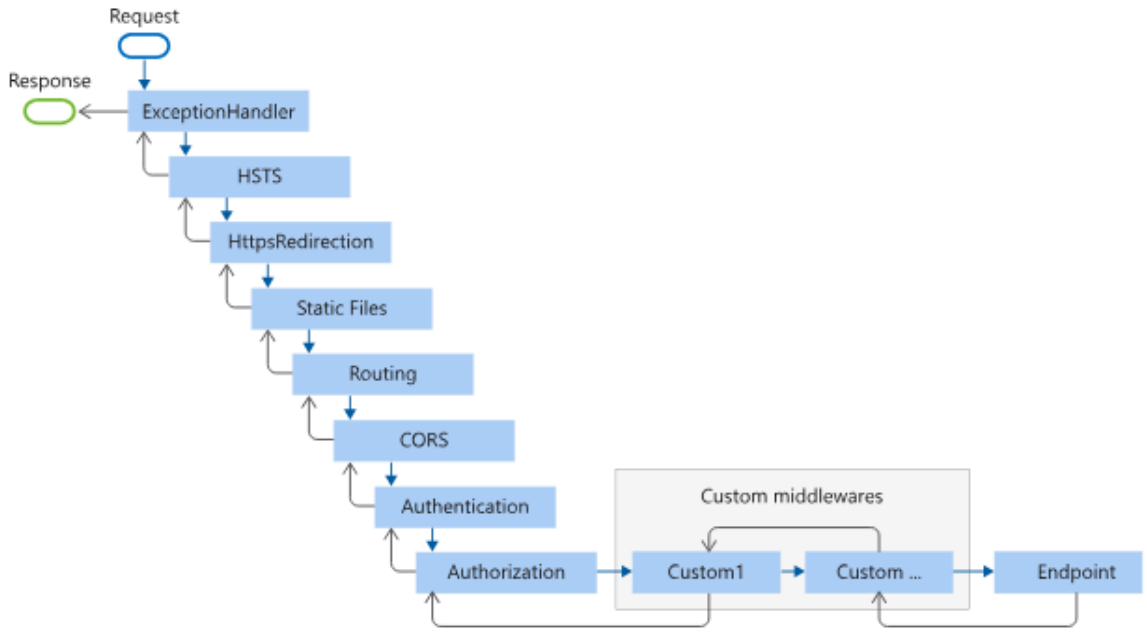
Example using IMiddleware

```
/// </summary>
2 references | DeepPatel25, 12 days ago | 1 author, 2 changes
public class CustomHeaderMiddleware : IMiddleware
{
    /// <summary>
    /// Adds a custom header to the HTTP response and performs a redirection.
    /// </summary>
    /// <param name="context">The HTTP context.</param>
    /// <param name="next">The delegate representing the next middleware in the pipeline.</param>
    0 references | DeepPatel25, 12 days ago | 1 author, 2 changes
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        // Add custom header to the response.
        context.Response.Headers.Append("Author", "Deep Patel");

        // Perform a redirection.
        context.Response.Redirect("https://www.google.com");

        // Call the next middleware in the pipeline.
        await next(context);
    }
```

Example without Using IMiddleware

```
namespace WebApiDemo.Middlewares
{
    2 references | Prince-Goswami, 10 days ago | 1 author, 1 change
    public class CustomMiddleware
    {
        private readonly RequestDelegate _next;

        0 references | Prince-Goswami, 10 days ago | 1 author, 1 change
        public CustomMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        0 references | Prince-Goswami, 10 days ago | 1 author, 1 change
        public async Task Invoke(HttpContext context)
        {
            // Apply custom logic before the request reaches the endpoint
            Console.WriteLine("Custom filter middleware: Before handling the request");

            // Call the next middleware in the pipeline
            await _next(context);

            // Apply custom logic after the response is generated
            Console.WriteLine("Custom filter middleware: After handling the response");
        }
    }

    // Extension method used to add the middleware to the HTTP request pipeline
    0 references | Prince-Goswami, 10 days ago | 1 author, 1 change
    public static class CustomFilterMiddlewareExtensions
    {
        1 reference | Prince-Goswami, 10 days ago | 1 author, 1 change
        public static IApplicationBuilder UseCustomMiddleware(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<CustomMiddleware>();
        }
    }
}
```

Middleware Order



Routing

- Routing is a way to specify which controller will be used for that specific request api.
- Routing is done by using Route Attribute on the above of any action method and controller
- Routing can be done in two ways.
- First is convention-based routing and second is attribute based routing.
- Convention based routing can be done by mapping app.MapControllerRoute

```
app.MapControllerRoute(
    name: "default",
    pattern: "api/{controller}/{action}/{id?}");
```

- Attribute based routing can be done by using Route attributes

```
/// </summary>
[ApiController]
[Route("api/[controller]")]

public class EmployeeController : ControllerBase
{
```

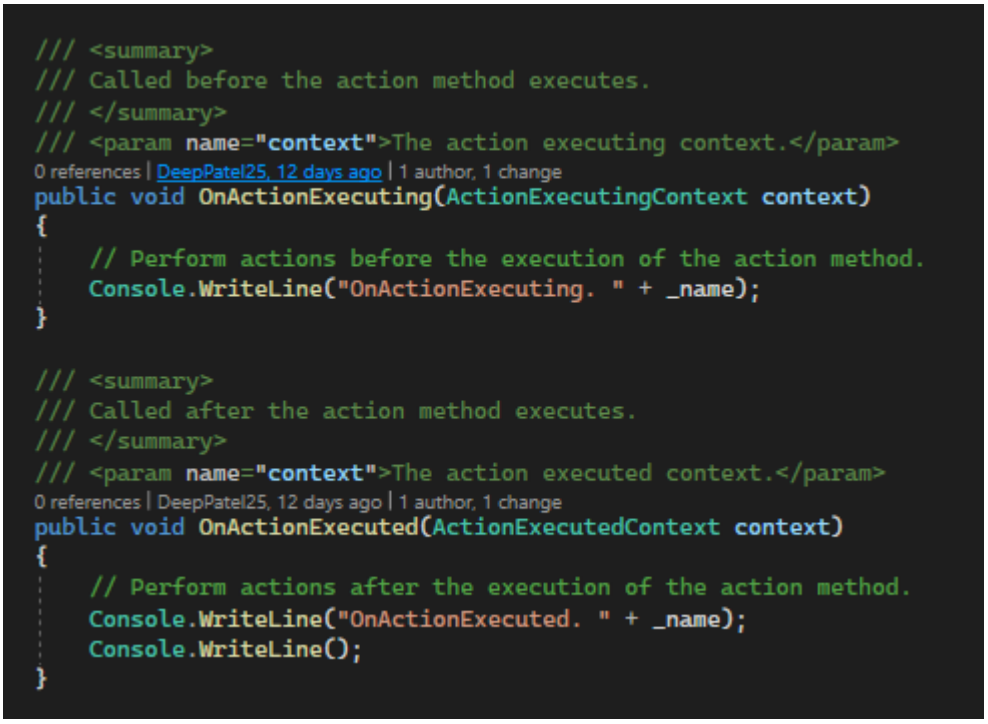
- And you can specify routes using HttpGet attribute template fields.

```
/// <returns>An action result containing
[HttpGet("GetAll")]
0 references | DeepPatel25, 12 days ago | 1 author, 1 change
public IActionResult Get()
{
    return Ok(_blEmployee.Get());
}
```

Filters

- Filters are classes that can be used to add specific details to response header, logging information, and etc.
- Filters have two types: Sync and Async.
- There are different filters for different types of tasks

1. **Action Filter**
    - Action filter has two methods one is OnActionExecuting and OnActionExecuted that are called before the action method is called and after it successfully finishes its execution.
    - Action filters default execution order is First Global level then controller and after that action methods action filter executes.
    - If you specify order value in those attributes then order can be changed accordingly to the Order Attribute which is part of IOrderedFilter.
    - IAsyncActionFilter has only one method which is OnActionExecutionAsync which has two arguments ActionExecutingContext, ActionExecutionDelegate.
    - ActionExecutingContext context contains the request and response information of the current request.
    - And ActionExecutionDelegate contains the reference of the next action filter or the action method.



```
/// <summary>
/// Called before the action method executes.
/// </summary>
/// <param name="context">The action executing context.</param>
0 references | DeepPatel25, 12 days ago | 1 author, 1 change
public void OnActionExecuting(ActionExecutingContext context)
{
    // Perform actions before the execution of the action method.
    Console.WriteLine("OnActionExecuting. " + _name);
}

/// <summary>
/// Called after the action method executes.
/// </summary>
/// <param name="context">The action executed context.</param>
0 references | DeepPatel25, 12 days ago | 1 author, 1 change
public void OnActionExecuted(ActionExecutedContext context)
{
    // Perform actions after the execution of the action method.
    Console.WriteLine("OnActionExecuted. " + _name);
    Console.WriteLine();
}
```

```
2 references | DeepPatel25, 12 days ago | 1 author, 1 change
public class MySampleAsyncActionFilterAttribute : Attribute, IAsyncActionFilter
{
    private readonly string _name;

    /// <summary>
    /// Initializes a new instance of the MySampleAsyncActionFilterAttribute class.
    /// </summary>
    /// <param name="name">The name associated with the filter.</param>
    1 reference | DeepPatel25, 12 days ago | 1 author, 1 change
    public MySampleAsyncActionFilterAttribute(string name)
    {
        _name = name;
    }

    /// <summary>
    /// Called asynchronously before the action method executes.
    /// </summary>
    /// <param name="context">The action executing context.</param>
    /// <param name="next">The delegate representing the next action filter or the action itself.</param>
    /// <returns>A task that represents the asynchronous on action execution operation.</returns>
    0 references | DeepPatel25, 12 days ago | 1 author, 1 change
    public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
    {
        // Perform actions before the execution of the action method.
        Console.WriteLine("Before Async Execution. " + _name);

        // Call the next action filter or the action method itself.
        await next();

        // Perform actions after the execution of the action method.
        Console.WriteLine("After Async Execution. " + _name);
    }
}
```

2. **Exception Filter**
    - It catches the exception which is thrown from the controllers and prints that where the developer wants to log it.
    - Its execution order is after the action method's executed method is completed.

```
public class LoggingExceptionFilter : IExceptionFilter
{
    private readonly IWebHostEnvironment _environment;

    /// <summary>
    /// Initializes a new instance of the LoggingExceptionFilter class.
    /// </summary>
    /// <param name="environment">The hosting environment.</param>

    public LoggingExceptionFilter(IWebHostEnvironment environment)
    {
        _environment = environment;
    }

    /// <summary>
    /// Handles the exception by logging it to a file.
    /// </summary>
    /// <param name="context">The exception context.</param>
    0 references | DeepPatel25, 7 days ago | 1 author, 2 changes
    public void OnException(ExceptionContext context)
    {
        Console.WriteLine("Exception occured.");

        // Log exception to file
        LogExceptionToFile(context.Exception);
    }
```

3. **IResourceFilter**
    - It has the same methods like IActionFilter but its executed after the authorization is completed then it will establish the connection to the database and other connectivities for the project and when the response is return back to the user at that time it executed OnResouceExecuted method to release the connections.

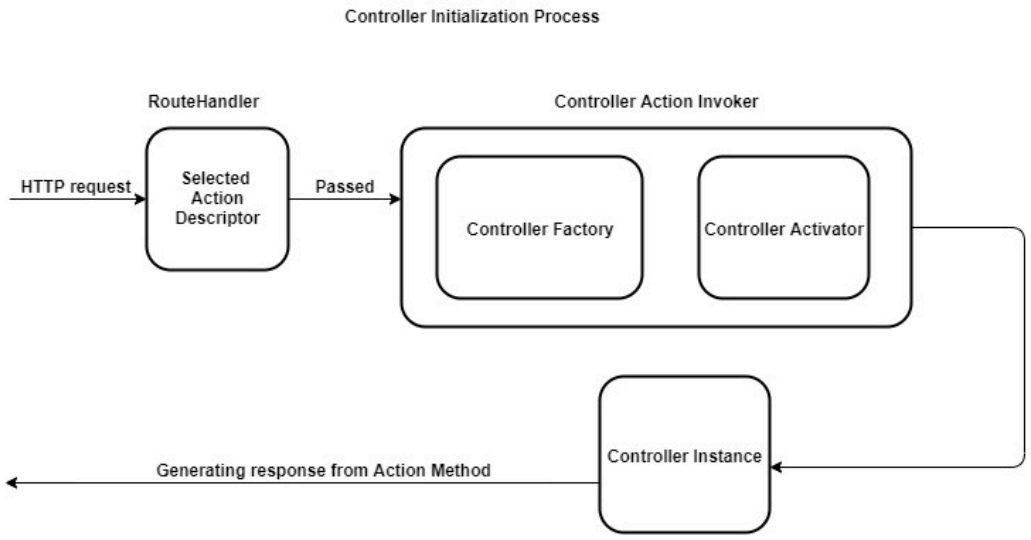
```
/// <summary>
/// For Establishing the database connectivity to the specified server.
/// </summary>
1 reference | DeepPatel25, 7 days ago | 1 author, 1 change
public class CustomResourceFilterAttribute : Attribute, IResourceFilter
{
    /// <summary>
    /// Called after the authentication process finished.
    /// </summary>
    /// <param name="context">The resource executing context.</param>
    0 references | DeepPatel25, 7 days ago | 1 author, 1 change
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        Console.WriteLine("Established database connection");
    }

    /// <summary>
    /// Called after the action filter and result is send back to the user for releasing the database connections.
    /// </summary>
    /// <param name="context">The resource executed context.</param>
    0 references | DeepPatel25, 7 days ago | 1 author, 1 change
    public void OnResourceExecuted(ResourceExecutedContext context)
    {
        Console.WriteLine("Released database connection");
    }
}
```

4. **IAuthorizationFilter**
    - It executes first and its priority is higher than all of the above filters.
    - It is used for authentication purposes.

```
public class BasicAuthenticationFilterAttribute : Attribute, IAuthorizationFilter
{

    public void OnAuthorization(AuthorizationFilterContext context)
    {
        bool allowAnnoymous = context.ActionDescriptor.EndpointMetadata
            .OfType<AllowAnonymousAttribute>().Any();

        if (allowAnnoymous)
        {
            return;
        }

        string authHeader = context.HttpContext.Request.Headers["Authorization"];

        if (authHeader != null && authHeader.StartsWith("Basic"))
        {
            // Extract credentials
            string encodedCredentials = authHeader.Substring("Basic ".Length).Trim();

            Tuple<string, string> usernameAndPassword =
                AuthenticationHelper.ExtractUserNameAndPassword(encodedCredentials);

            string username = usernameAndPassword.Item1;
            string password = usernameAndPassword.Item2;

            // Get the list of users.
            List<USR01> _lstUsers = BLUser.GetUsers();

            // Find user with matching credentials.
            USR01? objUser = _lstUsers.FirstOrDefault(u =>
                u.R01F02.Equals(username) && u.R01F03.Equals(password));
```

Controller Initialization

- The process of initialization and execution of controllers takes place. Controllers are responsible for handling incoming requests which is done by mapping requests to appropriate action methods. 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 an action descriptor. The RouteHandler then passes the action descriptor into a class called Controller action invoker. The class called Controller factory creates an instance of a controller to be used by the controller action method. The controller factory class depends on controller activator for controller instantiation.
- After the action method is selected, an instance of the controller is created to handle the request. Controller instances provide several features such as action methods, action filters and action results. The activator uses the controller type info property on the action descriptor to instantiate the controller by name. Once the controller is created, the rest of the action method execution pipeline can run.
- The controller factory is the component that is responsible for creating the controller instance. The controller factory implements an interface called IControllerFactory. This interface contains two methods which are called CreateController and ReleaseController.

Action Method

- Action method is a simple method that is called when the api request comes.
- An action method can be specified by using HttpGet, HttpPost and other attributes template fields.
- It has different types of return types but most of the time IActionResult and ActionResult is used.
- The Action Method binds the parameters of the request using FromQuery, FromBody, FromForm, and etc.

```
/// <summary>
/// Gets user information by ID.
/// </summary>
/// <param name="id">The ID of the user.</param>
/// <returns>An IActionResult containing user information.</returns>
[HttpGet("{id}")]
// [Authorize(Roles = "User")] // Authorize access to user role only.

public IActionResult Get(int id)
{
    return Ok(new { id });
}
```