Middlewares

1. What is Middleware?

Definition:

Middleware is software that sits between the incoming HTTP request and the outgoing HTTP response in an application. Each middleware in the pipeline processes requests **before** they reach the endpoint and can process responses **after** they leave the endpoint.

Think of middleware as a **chain of responsibility** where each link performs a specific task, such as:

- Logging
- Authentication
- Routing
- Error handling

Middleware Pipeline

The middleware pipeline is configured in `Program.cs` (or `Startup.cs` in older versions).

- Middleware is executed in the **order** it's added to the pipeline.
- Each middleware can:
  - **Pass the request to the next middleware**.
  - **Stop further processing** and return a response immediately.

Example pipeline:

```
var app = builder.Build();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

        endpoints.MapControllers();

});
```

Using Built-in Middleware

.NET Core provides several built-in middlewares to handle common tasks.

2.1 UseRouting

- **Purpose:** Matches the incoming request to an endpoint based on the route.
- **Required for:** Configuring endpoints for controllers or Razor Pages.

Example:

```
app.UseRouting(); // Enables routing
```

Without `UseRouting`, the application won't be able to map requests to specific controllers or endpoints.

2.2 UseAuthorization

- **Purpose:** Handles authorization for requests. It checks if the user is allowed to access a specific resource.
- **Works with:** `[Authorize]` attribute in controllers.

Example:

```
app.UseAuthorization(); // Enables authorization checks
```

Ensure `UseAuthorization` is added **after** `UseRouting`.

Complete Example with Built-in Middleware:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(); // Add controllers

builder.Services.AddAuthorization(); // Add authorization services

var app = builder.Build();

app.UseRouting(); // Adds routing middleware

app.UseAuthorization(); // Adds authorization middleware

app.UseEndpoints(endpoints =>

{

        endpoints.MapControllers(); // Maps controllers to endpoints

});

app.Run();
```

3. Creating Custom Middleware

Custom middleware allows you to add specific functionality to your application, such as logging or error handling.

3.1 Custom Logging Middleware

Step 1: Create the Middleware Class

```
public class LoggingMiddleware
{
    private readonly RequestDelegate _next;
    public LoggingMiddleware(RequestDelegate next)
    {       _next = next;       }
    public async Task Invoke(HttpContext context)
    {
    Console.WriteLine($"Incoming request: {context.Request.Method} {context.Request.Path}");
    await _next(context); // Pass to the next middleware
    Console.WriteLine($"Outgoing response: {context.Response.StatusCode}");
    }
}
```

Step 2: Register the Middleware You can register custom middleware in `Program.cs` using `app.UseMiddleware<T>()`.

```
var builder = WebApplication.CreateBuilder(args);


var app = builder.Build();


app.UseMiddleware<LoggingMiddleware>(); // Register custom logging middleware

app.UseRouting();

app.UseEndpoints(endpoints =>

{

        endpoints.MapControllers();

});


app.Run();
```

3.2 Custom Error Handling Middleware

Step 1: Create the Middleware Class

```
public class ErrorHandlerMiddleware
{       private readonly RequestDelegate _next;
        public ErrorHandlerMiddleware(RequestDelegate next)
        {       _next = next;    }
        public async Task Invoke(HttpContext context)
        {
        try
        {       await _next(context); // Pass to the next middleware  }
        catch (Exception ex)
        {       await HandleExceptionAsync(context, ex);        }
        }
        private static Task HandleExceptionAsync(HttpContext context, Exception
exception)

        {       context.Response.ContentType = "application/json";

        context.Response.StatusCode = StatusCodes.Status500InternalServerError;

        var result = Newtonsoft.Json.JsonConvert.SerializeObject(new

        {

                error = "An unexpected error occurred",

                details = exception.Message

        });

        return context.Response.WriteAsync(result);

        }

}
```

Step 2: Register the Middleware Add the error handling middleware **early in the pipeline** to catch exceptions.

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseMiddleware<ErrorHandlerMiddleware>(); // Register error handler

app.UseRouting();

app.UseEndpoints(endpoints =>

{

        endpoints.MapControllers();

});

app.Run();
```

4. Key Takeaways

1. **What is Middleware?**
    - Middleware is code that processes requests and responses in a pipeline.
    - It can modify requests, responses, or handle errors.
2. **Built-in Middleware:**
    - **UseRouting:** Matches requests to endpoints.
    - **UseAuthorization:** Checks if users have permission to access resources.
3. **Custom Middleware:**
    - Custom middleware lets you add application-specific logic, such as logging or error handling.

Final Example: Combining All Middleware

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddAuthorization();

var app = builder.Build();

app.UseMiddleware<LoggingMiddleware>(); // Custom logging middleware

app.UseMiddleware<ErrorHandlerMiddleware>(); // Custom error handling middleware

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{        endpoints.MapControllers();    });

app.Run();
```

This combines custom middleware for logging and error handling with built-in middleware for routing and authorization. You now have a robust middleware pipeline!