🧩 UNIT – 5: Middleware, Authentication & Authorization

🟢 1. Middleware

🔸 1.1 Introduction to Middleware

In **.NET Core** (now just called **.NET**), **middleware** is a very important part of the **HTTP request pipeline**.

Think of middleware as small components that handle requests and responses when you visit a website or API.

Each middleware:

- Runs when a request comes in.
- Can **process**, **change**, or **stop** a request.
- Can also **pass** the request to the next middleware in the chain.

So, middleware gives flexibility to handle how your application reacts to each HTTP request.

👉 **Middleware can:**

- Check or modify requests (like logging, authentication).
- Stop the request if needed (short-circuit).
- Do something before or after the next middleware runs.
- Modify the response before it is sent back.

🔹 Example of Middleware Pipeline

When a request comes to your website:

HTTP Request

↓

Middleware 1 (Logging)

↓

Middleware 2 (Authentication)

↓

Middleware 3 (Routing)

↓

Controller / Endpoint

Then on the way back:

Controller Response

↑

Middleware 3

↑

Middleware 2

↑

Middleware 1

↑

HTTP Response

Each middleware can do something before and after the next one runs.

🔸 1.2 Built-in Middlewares

ASP.NET Core already provides many **ready-made middlewares**.
You can use them easily without writing your own.

Some examples include:

- **UseRouting** → for routing requests to controllers.
- **UseAuthentication** → to check user identity.
- **UseAuthorization** → to check user permissions.
- **UseStaticFiles** → to serve static content like images or CSS.
- **UseExceptionHandler** → for handling errors globally.
- **UseEndpoints** → to map endpoints like controllers or pages.

You register these in your **Program.cs** file, and they will execute in the same order you add them.

🟠 1.3 Action Filters

Action filters are special attributes used in **controllers or actions** to run code *before or after* an action executes.
They help you handle **cross-cutting concerns** — things like logging, caching, error handling, or authorization that you need in many places.

Instead of writing the same code again and again, you just add a filter.

🔹 Types of Filters in ASP.NET Core

1. **Authorization filters** – Check permissions before anything else.
   Implements IAuthorizationFilter.

2. **Action filters** – Run before and after a controller action executes.
   Implements IActionFilter.

3. **Result filters** – Run before and after the result (like a view or JSON) is generated.
   Implements IResultFilter.

4. **Exception filters** – Catch and handle exceptions.
   Implements IExceptionFilter.

✅ **Order of execution:**
Authorization → Action → Result → Exception.

🔹 Base Class and Methods

To create a custom filter:

- Inherit from ActionFilterAttribute.
- Override these methods:

Method	Description
OnActionExecuting	Runs before the action method starts
OnActionExecuted	Runs after the action finishes
OnResultExecuting	Runs before the result is processed
OnResultExecuted	Runs after the result is processed

Example uses:

- Logging every action call
- Validating model data
- Measuring performance
- Handling errors

🔸 1.4 Middleware vs Action Filters

Feature	Middleware	Action Filter
Level	Works globally (for all requests)	Works on specific controllers or actions
Scope	Handles HTTP requests & responses	Handles controller/action logic
Location	Defined in Program.cs	Defined inside controllers
Order	Executed in registration order	Executed based on filter type

Feature	Middleware	Action Filter
Example	Logging, Routing, Auth	Validation, Authorization, Error Handling

Both are used to add reusable logic, but **middleware works globally**, while **filters work at controller level**.

🟢 1.5 Building Custom Middleware

You can create your own middleware in two ways:

1. By writing a **class**.
2. By using a **lambda expression** directly in Program.cs.

Key Concepts

- **Order matters** – Middleware runs in the same order it's added.
- **Terminal Middleware** – If one middleware ends the pipeline (using app.Run()), others after it will not execute.
- **Common Methods:**
  - Use() – Add middleware that can continue or stop the next.
  - Run() – Add terminal middleware (stops further execution).
  - Map() – Create a separate branch for a specific route (like /api).

Example Custom Middleware Class

```
public class MyCustomMiddleware
{
    private readonly RequestDelegate _next;
    public MyCustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        // Do something before
        await _next(context); // Call next middleware
        // Do something after
    }
}
```

You add it in Program.cs:

```
app.UseMiddleware<MyCustomMiddleware>();
```

Or by extension method:

```
app.UseMyCustomMiddleware();
```

Use(), Run(), Map() Comparison

Method	**Purpose**
Use()	Add middleware that can continue the chain
Run()	Last middleware – ends the pipeline
Map()	Creates a separate branch for specific route

Example:

```
app.Use(async (context, next) => { ... });
app.Run(async (context) => { ... });
app.Map("/admin", adminApp => { ... });
```

🧩 2. Authentication and Authorization

🔸 2.1 Authentication vs Authorization

Concept	**Authentication**	**Authorization**
Meaning	Verifies *who you are*	Checks *what you can do*
Example	Logging in with username & password	Allowing access to admin panel
Happens when	First	After authentication
Based on	Credentials (username/password, token)	Roles or permissions

Both are essential for security in ASP.NET Core APIs.

🔸 2.2 JWT (JSON Web Token) Authentication

JWT is a **popular, stateless method** for API authentication in .NET Core.

How JWT Works

1. **User Login**
   User sends username and password to server.
2. **Token Generation**
   If valid, server creates a JWT that includes:
   - **Header** (type, algorithm)
   - **Payload** (user data and roles)
   - **Signature** (to verify authenticity)
3. **Token Delivery**
   Server sends this token to the client.
4. **Subsequent Requests**
   Client sends the JWT with every request in the header like:
5. Authorization: Bearer <token>
6. **Authorization Check**
   Server verifies the token using the secret key:
   - If valid → access granted.
   - If invalid or expired → returns 401 or 403.

JWTs are **self-contained** and **don't need server memory** to track sessions — ideal for APIs.

🔸 2.3 Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) allows access based on **roles**, not individual users.

Instead of assigning permissions to each user, we assign roles (like *Admin*, *Editor*, *User*), and each role has its own permissions.

Key Concepts

Term	Meaning
User	A person using the app
Role	A collection of permissions (Admin, Editor, Guest)
Permission	Specific allowed action (Create, Edit, Delete)

Example:

- Admin → Can create, edit, delete
- Editor → Can edit only
- Guest → Can only read

In .NET Core

.NET Core uses **ASP.NET Core Identity** to manage:

- Users
- Roles
- Claims (user info like email, role, name)

How Authorization Works

After a user is authenticated:

- **Role-based Authorization:**
  Use [Authorize(Roles = "Admin")] to restrict actions.
- **Policy-based Authorization:**
  Create a policy in Program.cs and apply it:
- [Authorize(Policy = "CanEditPosts")]

Steps:

1. Define Policy
2. Apply it to controller/action

This makes code cleaner and easier to manage.

Flow of RBAC

1. User logs in → Authenticated.
2. Server assigns a JWT with user's roles.
3. When accessing resources → Authorization checks the user's roles/policies.
4. If role matches → Access granted.

🧩 3. Global Error Handling

🔸 3.1 Concept

Instead of writing try...catch in every controller, you can use **global error handling** to manage all exceptions in one place.

It improves:

- Code **cleanliness**
- **Security** (no sensitive info shown)
- **User experience** (friendly error page)

How It Works

ASP.NET Core’s middleware pipeline can catch exceptions globally.

You can add **UseExceptionHandler** middleware at the top of the pipeline to handle all unhandled exceptions.

Steps to Implement Global Error Handling

1. **UseExceptionHandler in Program.cs**
2. app.UseExceptionHandler("/Error");

→ Redirects all errors to /Error.

3. **Create Error Controller**
4. public IActionResult Error()
5. {
6. var feature = HttpContext.Features.Get<IExceptionHandlerFeature>();
7. // Log the error here
8. return View("Error");
9. }
10. **Handle Different Environments**
    - **Development:** Show detailed error page (DeveloperExceptionPage).
    - **Production:** Show friendly error page (no internal details).

Why Use Global Error Handling

- **Maintainability:** One place for all error logic.
- **Security:** No sensitive details shown.
- **User Experience:** Clean, friendly error messages.
- **DRY Principle:** Avoid repeating try-catch everywhere.

✅ Summary of Unit – 5

Topic	**Key Idea**
Middleware	Handles requests/responses globally in pipeline
Action Filters	Handle logic before/after controller actions
Custom Middleware	User-defined request handlers
Authentication	Verifies identity
Authorization	Checks access permissions
JWT	Stateless token-based authentication
RBAC	Role-based access control for permissions
Global Error Handling	Centralized way to catch and display errors