🔐 SECTION 8: AUTHENTICATION & AUTHORIZATION 🔐

1️⃣ What is Authentication?

🎯 Authentication is the process of verifying the identity of a user or system before granting access to an application.

In ASP.NET Core:

- Authentication is handled by **authentication middleware**
- Credentials such as JWT tokens, cookies, or certificates are validated
- On success, a **ClaimsPrincipal** is created
- This identity is attached to ***HttpContext.User***

ASP.NET Core supports multiple authentication schemes and allows applications to authenticate users using different mechanisms simultaneously.

🔁 Q1: Where does authentication happen in ASP.NET Core?

Answer: Authentication happens in the middleware pipeline, specifically when `UseAuthentication()` is executed. This middleware reads credentials from the request and builds the user identity.

🔁 Q2: What is ***ClaimsPrincipal***?

Answer: ***ClaimsPrincipal*** represents the authenticated user.

It contains one or more ***ClaimsIdentity*** objects, each holding a set of claims such as user ID, role, email, permissions, etc.

🔁 Q3: Can one application have multiple authentication schemes?

Answer: Yes. ASP.NET Core supports multiple authentication schemes, such as:

- JWT for APIs
- Cookies for MVC
- OAuth for external login

The appropriate scheme is selected using attributes or policies.

2️⃣ What is Authorization?

🎯 Authorization is the process of determining whether an authenticated user has permission to access a specific resource or action.

ASP.NET Core authorization:

- Occurs after authentication

- Evaluates user roles, claims, or policies
- Is enforced using:
    - `[Authorize]` attribute
    - Authorization middleware
    - Policy handlers

Authorization logic is declarative, centralized, and independent of business logic.

🔁 Q1: What happens when authorization fails?

Answer: If authorization fails:

- API returns 401 (unauthenticated) or 403 (forbidden)
- Controller action is not executed

🔁 Q2: Where does `[Authorize]` execute?

Answer: `[Authorize]` is implemented internally using the **authorization filter**, which works in coordination with the authorization middleware.

🔁 Q3: Can authorization run without authentication?

Answer: No. Authorization requires an authenticated identity. Without authentication, authorization automatically fails.

3️⃣ **Authentication vs Authorization.**

🎯 **Authentication:** Verifies identity --> "Who are you?" --> Creates ClaimsPrincipal --> Happens first

Authorization: Verifies permissions --> "What can you access?" --> Evaluates policies --> Happens after authentication

🔁 Q1: Which one is more critical?

Answer: Both are critical, but authorization errors are more dangerous because they may lead to privilege escalation or data exposure.

4️⃣ **Where do Authentication & Authorization run in the pipeline?**

🎯 They run as middleware components:

- `app.UseAuthentication();`
- `app.UseAuthorization();`

Correct order:

- Authentication → builds identity
- Authorization → checks access rules

🔁 Q1: What happens if order is reversed?

Answer: Authorization fails because no authenticated user exists in ***HttpContext.User***.

🔁 Q2: Is authentication executed for every request?

Answer: Yes, but only if the endpoint requires authentication or the middleware is configured globally.

5️⃣ What is Claims-based Authentication?

🧠 Instead of just username and role, we store facts about the user.

🎯 Claims-based authentication represents a user using a collection of claims, where each claim is a key-value pair describing identity or permissions.

Examples:

- UserId
- Email
- Role
- Permission
- Department

Claims allow fine-grained authorization and are widely used with JWT tokens.

🔁 Q1: Difference between **Role and Claim**?

Answer: Role is a coarse-grained claim and Claims allow fine-grained, flexible access control

🔁 Q2: Where are claims stored?

Answer: Claims are stored inside:

- *ClaimsPrincipal* in memory
- JWT payload during token transmission

6️⃣ Role-based Authorization

🎯 Role-based authorization restricts access using predefined roles.

Example:

```
[Authorize(Roles = "Admin")]
```

Pros:

- Simple
- Easy to implement

Cons:

- Not scalable
- Role explosion
- Difficult to represent complex business rules

🔁 Q1: Why role-based authorization fails at scale?

Answer: Because large systems require hundreds of roles, making maintenance and auditing extremely difficult.

🔁 Q2: Are roles implemented as claims?

Answer: Yes. Internally, roles are just claims with a specific claim type.

7️⃣ Policy-based Authorization (ENTERPRISE LEVEL)

🎯 Policy-based authorization defines rules (policies) instead of roles.

A policy may check:

- Claims
- Roles
- Custom logic

Example:

```
policy.RequireClaim("Permission", "EditOrder");
```

Policies are:

- Reusable
- Centralized
- Highly scalable
- Ideal for enterprise systems

🔁 Q1: Why policies are better than roles?

Answer: Policies allow business-oriented rules instead of static job titles.

🔁 Q2: Can a policy have multiple conditions?

Answer: Yes. Policies can combine multiple requirements.

8️⃣ Custom Authorization Handlers

🎯 Custom authorization handlers allow dynamic, context-aware authorization.

Used when:

- Ownership checks
- Resource-based access
- Data-dependent authorization

🔁 Q1: Difference between requirement and handler?

- Requirement defines what to check
- Handler defines how to check

🔁 Q2: How does handler access `HttpContext`?

Answer: Via `AuthorizationHandlerContext` or injected `IHttpContextAccessor`.

9️⃣ What is JWT Authentication?

🎯 JWT is a stateless authentication mechanism where identity and claims are stored in a signed token.

Structure:

- **Header**
- **Payload**
- **Signature**

ASP.NET Core validates:

- Signature
- Issuer
- Audience
- Expiry

🔁 Q1: Why JWT is stateless?

Answer: Server does not store session data; every request carries authentication info.

🔁 Q2: How refresh tokens work?

Answer: Short-lived access tokens + long-lived refresh tokens reduce risk if token is compromised.

1️⃣0️⃣ JWT vs Cookie Authentication

🎯 **JWT** : Stateless, Best for APIs, Stored client-side, CSRF resistant

Cookies : Stateful, Best for MVC, Stored server-side, CSRF vulnerable

1️⃣1️⃣ How do you secure JWT tokens?

🎯 Best practices:

- HTTPS only
- Short expiration
- Refresh tokens
- Strong signing keys
- Never store in localStorage

🔁 Q: Where should JWT be stored?

Answer: Prefer ***HttpOnly* secure cookies** or in-memory storage.

1️⃣2️⃣ Common Security Threats & Mitigation

Threat	Mitigation
CSRF	Anti-forgery / JWT
XSS	Encoding + CSP
SQL Injection	Parameterized queries
Broken Auth	Policies + validation

1️⃣3️⃣ Where authorization logic should NOT be written?

🎯 Never write authorization logic:

- Inside controllers using if
- Inside services using role checks

Use:

- Policies
- Handlers
- Attributes

🎯 **END** 🎯

🔐 HANDS-ON: AUTHENTICATION & AUTHORIZATION (JWT+POLICY) 🔐

🧱 STEP 1: Create ASP.NET Core Web API

```
dotnet new webapi -n SecureApi
cd SecureApi
```

👉 This creates:

- Program.cs
- Controllers
- appsettings.json

🧱 STEP 2: Add JWT Packages

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

📌 Interview point: ASP.NET Core uses middleware-based authentication, so JWT is added as middleware.

🧱 STEP 3: Configure JWT Settings (appsettings.json)

```
"JwtSettings": {
  "Issuer": "SecureApi",
  "Audience": "SecureApiUsers",
  "SecretKey": "THIS_IS_A_VERY_SECURE_KEY_12345",
  "ExpiryMinutes": 30
}
```

📌 Interview tip:

- Secret key must be long & random
- Stored securely in Key Vault / Environment variables in real systems

🧱 STEP 4: Create JWT Token Generator (Authentication Logic)

📁 Create folder: **Services**

```
public interface ITokenService
{
    string GenerateToken(string userId, string role);
}
```

Implementation:

```
public class TokenService : ITokenService
{
    private readonly IConfiguration _config;

    public TokenService(IConfiguration config)
    {
        _config = config;
    }

    public string GenerateToken(string userId, string role)
    {
        var claims = new[]
        {
            new Claim(ClaimTypes.NameIdentifier, userId),
            new Claim(ClaimTypes.Role, role),
            new Claim("Permission", "ReadData")
        };

        var key = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(_config["JwtSettings:SecretKey"])
        );

        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken(
            issuer: _config["JwtSettings:Issuer"],
            audience: _config["JwtSettings:Audience"],
            claims: claims,
            expires: DateTime.UtcNow.AddMinutes(
                int.Parse(_config["JwtSettings:ExpiryMinutes"])
            ),
            signingCredentials: creds
        );

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}
```

📌 Interview highlights:

- Claims-based identity
- Stateless authentication
- Token contains identity + permissions

🧱 STEP 5: Register Services in DI (Program.cs)

```
builder.Services.AddScoped<ITokenService, TokenService>();
```

📌 Interview follow-up:

- **Why scoped?** --> Because token service may depend on request-level services.

🧱 STEP 6: Configure Authentication Middleware (CRITICAL)

```
builder.Services.AddAuthentication("Bearer")
.AddJwtBearer("Bearer", options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,

        ValidIssuer = builder.Configuration["JwtSettings:Issuer"],
        ValidAudience = builder.Configuration["JwtSettings:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(builder.Configuration["JwtSettings:SecretKey"])
        )
    };
});
```

📌 Interview GOLD:

- This is where JWT is validated
- Token signature + expiry + issuer checked

🧱 STEP 7: Configure Authorization (Policy-Based)

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("CanReadData", policy =>
        policy.RequireClaim("Permission", "ReadData"));
});
```

📌 Interview line:

- Policy-based authorization scales better than role-based authorization.

🧱 STEP 8: Add Middleware in Correct Order

```
app.UseAuthentication();
app.UseAuthorization();
```

🚨 VERY IMPORTANT (INTERVIEW TRAP):

- ❌ Wrong order → authorization fails
- ✅ Authentication must come first

🧱 STEP 9: Create Auth Controller (Login Endpoint)

```
[ApiController]
[Route("api/auth")]
public class AuthController : ControllerBase
{
    private readonly ITokenService _tokenService;

    public AuthController(ITokenService tokenService)
    {
        _tokenService = tokenService;
    }

    [HttpPost("login")]
    public IActionResult Login()
    {
        // Simulating user validation
        var token = _tokenService.GenerateToken("123", "Admin");
        return Ok(new { access_token = token });
    }
}
```

📌 Interview clarification:

- Authentication logic usually validates user from DB
- Token issued only after successful validation

🧱 STEP 10: Secure an API Endpoint

```
[ApiController]
[Route("api/data")]
public class DataController : ControllerBase
{
    [HttpGet]
    [Authorize(Policy = "CanReadData")]
    public IActionResult GetData()
    {
        return Ok("Secure data accessed");
    }
}
```

📌 Interview insight:

- Authorization happens before controller executes
- If policy fails → 403 Forbidden

🧱 STEP 11: Test Flow (IMPORTANT FOR INTERVIEW)

1️⃣ Call `/api/auth/login` → Receive JWT token

2️⃣ Call `/api/data`

Add header:

```
Authorization: Bearer <JWT_TOKEN>
```

3️⃣ Result:

- Valid token → 200 OK
- Missing/invalid token → 401
- Missing permission → 403

🧠 INTERVIEW FLOW YOU CAN SAY

"Client authenticates via login endpoint, receives JWT.

Token is validated by authentication middleware, identity is built using claims, then authorization middleware enforces policies before controller execution."

🔥 This sentence alone impresses interviewers.

🧨 COMMON INTERVIEW FOLLOW-UPS (WITH ANSWERS)

❓ Where is JWT validated?
👉 In JwtBearer authentication middleware

❓ Why JWT is stateless?
👉 No server session; each request carries identity

❓ Why policy over role?
👉 Policies allow fine-grained permissions

❓ How to revoke JWT?
👉 Short expiry + refresh token + token blacklist (if needed)

🎯 **END** 🎯