EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

.NET Programming

Chapter 4: Authentication and authorization

TABLE CONTENT

- Authentication
- Authorization

AUTHENTICATION

AUTHENTICATION

Securing Web Applications with User Identity Verification

Definition: Authentication is the process of identifying and verifying a user's identity before granting access to an application.

Importance:

- Ensures that only authorized users can access sensitive resources.
- Forms the first layer of defense in securing web apps.
- Prevents unauthorized access and attacks like impersonation or account takeover.

AUTHENTICATION

What is Authentication?

Key Points:

- Authentication confirms the user's identity through credentials (e.g., username/password, tokens, etc.).
- It is typically the first step in the security process, followed by authorization.
- If authentication is successful, a user is granted access to the application based on their identity.

Example: User logs into an app with a username and password.

Methods of Authentication:

- Local login (username/password)
- Token-based authentication (JWT, OAuth)
- Social logins (e.g., Google, Facebook)

AUTHENTICATION

Authentication vs Authorization

Authentication (Who you are):

- Verifies the identity of a user.
- Example: Logging in with your username and password.

Authorization (What you can do):

- Determines what actions a user is allowed to perform after authentication.
- Example: Access to a dashboard for admin users only.

Key Differences:

- Authentication = Identifying **who** the user is.
- Authorization = Deciding **what** the user can do once authenticated.

AUTHENTICATION

Why Authentication Matters

Importance of Verifying User Identity:

- Protects against unauthorized access and security breaches.
- Ensures that sensitive information (e.g., user data, financial info) is accessed only by legitimate users.

Real-World Example:

- Without proper authentication, attackers could impersonate users and gain control over accounts.
- Common attacks prevented by authentication: brute force, phishing, and credential stuffing.

Key Benefits:

- Enhances user trust.
- Complies with security standards and regulations (e.g., GDPR, HIPAA).

AUTHENTICATION

Overview of Authentication in ASP.NET Core

Authentication Service:

- ASP.NET Core uses the `IAuthenticationService` interface to manage authentication tasks.

Authentication Middleware:

- The authentication middleware is responsible for validating user credentials and generating an identity (`ClaimsPrincipal`).

Authentication Schemes:

- Different schemes (e.g., JWT, cookies) are used to handle different authentication methods.
- Example: `AddJwtBearer` for token-based authentication, `AddCookie` for cookie-based authentication.

Integration: Authentication is added in `Program.cs` using `AddAuthentication()` and `UseAuthentication()`.

Authentication Workflow in ASP.NET Core

ASP.NET Core Authentication Pipeline

Request-Response Pipeline Overview:

- In ASP.NET Core, requests are processed through a pipeline of middleware components.
- The authentication middleware intercepts incoming requests to verify user identity.
- If the request contains valid credentials, the middleware generates a ClaimsPrincipal representing the user's identity.

Key Components in the Pipeline:

- **Request:** Incoming HTTP request from the user.
- **Middleware:** Components (including authentication) that process the request.
- **Response:** Generated after the request is handled, potentially containing user-specific data.

Order of Middleware:

- Middleware is processed in the order it's added, so authentication must be set up before authorization or routing.

Authentication Workflow in ASP.NET Core

IAuthenticationService in ASP.NET Core

Role of IAuthenticationService:

- The IAuthenticationService interface is the core service responsible for handling authentication operations.
- It abstracts authentication logic, making it flexible to integrate various authentication mechanisms.

Key Responsibilities:

- Authenticate users based on the provided credentials.
- Challenge unauthenticated users who attempt to access protected resources.
- Handle multiple authentication schemes (e.g., cookies, JWT, OAuth).

Methods:

- AuthenticateAsync: Verifies credentials and generates the ClaimsPrincipal.
- ChallengeAsync: Initiates the authentication challenge if credentials are missing or invalid.

Authentication Workflow in ASP.NET Core

Authentication Middleware in ASP.NET Core

What is Middleware?

- Middleware is software that handles requests and responses in the ASP.NET Core pipeline.

Authentication Middleware:

- Intercepts incoming requests to check for authentication tokens (e.g., JWT, cookies).
- Adds the authenticated user information (ClaimsPrincipal) to the HttpContext if the request is authenticated.
- Middleware must be added in the correct order: first authentication, then authorization.

Adding Middleware in Program.cs:

- UseAuthentication() is called to register the authentication middleware.

Why Order Matters:

- Authentication middleware must run before any middleware that depends on authenticated users (like authorization).

Authentication Workflow in ASP.NET Core

Authentication Handlers and Schemes

Authentication Schemes:

- Schemes are named configurations for handling authentication (e.g., "Cookies", "Bearer").
- They define how an app should authenticate requests, using specific authentication methods.

Authentication Handlers:

- Each scheme has an associated handler that implements the authentication logic.
- Examples of handlers:
  - **CookieAuthenticationHandler:** Handles cookie-based authentication.
  - **JwtBearerHandler:** Handles JWT-based authentication.

Custom Schemes and Handlers:

- Developers can create custom authentication handlers for unique scenarios.

Authentication Workflow in ASP.NET Core

Scheme Selection Logic

- **How ASP.NET Core Selects Schemes:**
  - The default authentication scheme is specified when configuring authentication.
  - The AddAuthentication() method sets a default scheme (e.g., "Bearer" for JWT, "Cookies" for cookie-based).
- **Multiple Schemes:**
  - Apps can use multiple schemes for different parts of the application.
  - Scheme selection can be enforced by attributes or policies in controllers (e.g., [Authorize(AuthenticationSchemes = "Bearer")]).
- **Common Scenarios:**
  - Single authentication scheme (e.g., only cookies).
  - Multiple schemes for different API endpoints (e.g., JWT for APIs, cookies for web UI).
- **Fallback Logic:**
  - If no scheme is specified for a request, the default scheme is used.

Authentication Options

Built-in Authentication Methods in ASP.NET Core

Overview of Built-in Options:

- ASP.NET Core offers various authentication methods that can be easily configured to secure applications.

Key Authentication Methods:

Cookies:

- Commonly used for web applications to store user identity in a cookie.
- Ideal for scenarios where users interact directly with the UI.

JWT (JSON Web Tokens):

- Used primarily for APIs.
- Tokens are stateless and contain claims, allowing verification without storing session data.

OAuth 2.0:

- Framework for third-party authorization, enabling users to authenticate using their existing accounts from other providers.
- Often used in combination with OpenID Connect for identity verification.

Authentication Options

Cookie Authentication

What is Cookie Authentication?

- A method that uses HTTP cookies to maintain user sessions after authentication.
- Stores user identity and session state in a cookie on the client-side.

How it Works:

- After a successful login, the server generates a cookie containing a unique identifier for the session.
- This cookie is sent to the client and stored in the browser.
- For subsequent requests, the cookie is included in the HTTP headers, allowing the server to identify the user.

Key Configuration:

- Configured using AddCookie() in Program.cs.
- Can set options such as expiration time, sliding expiration, and cookie policies.

Authentication Options

Cookie Authentication

```
// Add services to the container.
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/Account/Login";
        options.LogoutPath = "/Account/Logout";
        options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
    });

app.UseAuthentication();
```

Authentication Options

JWT Bearer Authentication

What is JWT?

- JSON Web Tokens (JWT) are a compact, URL-safe means of representing claims to be transferred between two parties.

How JWT Works:

- The server generates a JWT after successful user authentication.
- The token contains encoded user information and is signed to prevent tampering.
- The client stores the JWT (usually in local storage) and includes it in the Authorization header for API requests.

Validation:

- On receiving a request, the server validates the JWT signature and extracts the claims.
- If valid, the user is granted access; otherwise, the request is rejected.

Configuration:

- Implemented using AddJwtBearer() in Program.cs.

Authentication Options

JWT Bearer Authentication

```
// Add services to the container.
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(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"]))
        };
    });
```

* Remember to install the relative package in NuGet (If any)

Authentication Options

OAuth 2.0 and OpenID Connect

Overview of OAuth 2.0:

- A protocol for authorization that allows third-party applications to obtain limited access to user accounts.
- Enables users to authenticate via external providers (e.g., Google, Facebook) without sharing credentials.

OpenID Connect:

- Built on top of OAuth 2.0, it adds identity verification to the authorization process.
- Provides user identity information via ID tokens.

How it Works:

- The user is redirected to the external provider for authentication.
- Upon successful authentication, the provider returns an access token (and possibly an ID token) to the application.

Implementation:

- Configured using `AddAuthentication()` with specific methods for each provider.

Authentication Options

OAuth 2.0 and OpenID Connect

```
// Add services to the container.
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = CookieAuthenticationDefaults.AuthenticationScheme;
})
.AddCookie().AddGoogle(options =>
{
    options.ClientId = builder.Configuration["Google:ClientId"];
    options.ClientSecret = builder.Configuration["Google:ClientSecret"];
});
```

Authentication Options

Other Providers (Google, Facebook, Microsoft)

Using Social Media Logins:

- ASP.NET Core supports integration with various social media and third-party identity providers.
- Popular providers include:
  - **Google:** Users can log in using their Google account.
  - **Facebook:** Allows authentication through Facebook credentials.
  - **Microsoft:** Supports logging in with Microsoft accounts (e.g., Outlook, Azure).

Benefits:

- Reduces the friction of user registration and login.
- Leverages the security features of established providers.
- Enhances user experience by providing single sign-on (SSO).

Implementation:

- Set up via AddAuthentication() with specific methods for each provider.

Authentication Options

Other Providers (Google, Facebook, Microsoft)

```
// Add services to the container.
builder.Services.AddAuthentication()
    .AddFacebook(options =>
    {
        options.AppId = builder.Configuration["Facebook:AppId"];
        options.AppSecret = builder.Configuration["Facebook:AppSecret"];
    })
    .AddGoogle(options =>
    {
        options.ClientId = builder.Configuration["Google:ClientId"];
        options.ClientSecret = builder.Configuration["Google:ClientSecret"];
    });
```

Configuring Authentication in ASP.NET Core

Setting up Authentication in Program.cs

Introduction:

- In ASP.NET Core, authentication is configured in the Program.cs file.
- This includes registering authentication services and adding middleware to the request pipeline.

Key Steps:

- Add authentication services to the DI container using builder.Services.
- Configure middleware with app.UseAuthentication() and app.UseAuthorization() in the request pipeline.

```
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

// Adding authentication services
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();

var app = builder.Build();

// Adding authentication and authorization middleware
app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
app.Run();
```

Configuring Authentication in ASP.NET Core

Adding Schemes in Program.cs

Multiple Schemes in ASP.NET Core:

- ASP.NET Core supports multiple authentication schemes, such as Cookies and JWT Bearer, within the same application.
- Schemes allow the app to handle different types of authentication in various parts of the app.

Key Points:

- **Cookies:** Typically used for web applications with a UI.
- **JWT Bearer:** Common for securing APIs.

```
// Adding multiple authentication schemes
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddCookie()
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "your-issuer",
        ValidAudience = "your-audience",
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("your-secret-key"))
    };
});
```

Configuring Authentication in ASP.NET Core

Configuring Cookie Authentication

Cookie Authentication Overview:

- Cookie authentication is often used for traditional web applications.
- Stores user session data in cookies.

Configuration Details:

- Configure login and logout paths, cookie expiration, and security options.

```
// Configuring cookie authentication
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/Account/Login";
        options.LogoutPath = "/Account/Logout";
        options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
        options.SlidingExpiration = true;
        options.Cookie.HttpOnly = true; // Security setting to prevent XSS attacks
    });
```

Configuring Authentication in ASP.NET Core

Configuring JWT Authentication

JWT Authentication Overview:

- JWT (JSON Web Token) is commonly used for securing APIs.
- Tokens are issued by the server and passed between client and server in HTTP headers.

Key Configuration Settings:

- Issuer, audience, and signing key validation.
- Token expiration management.

```
// Configuring JWT authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(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"]))
        };
    });
```

Configuring Authentication in ASP.NET Core

Authentication Options: Summary

Recap of Key Authentication Schemes:

Cookies: Best for web applications with user sessions.

JWT: Ideal for APIs requiring stateless authentication.

Multiple Schemes: Flexibility to support both UI and API authentication.

Configuration Summary:

- Key differences in configuration between Cookies and JWT Bearer.
- Best practices for token and cookie security.

Key Takeaways:

- Understand the use case for each authentication scheme.
- Configure middleware correctly for seamless integration into the request pipeline.

Working with Authentication in MVC

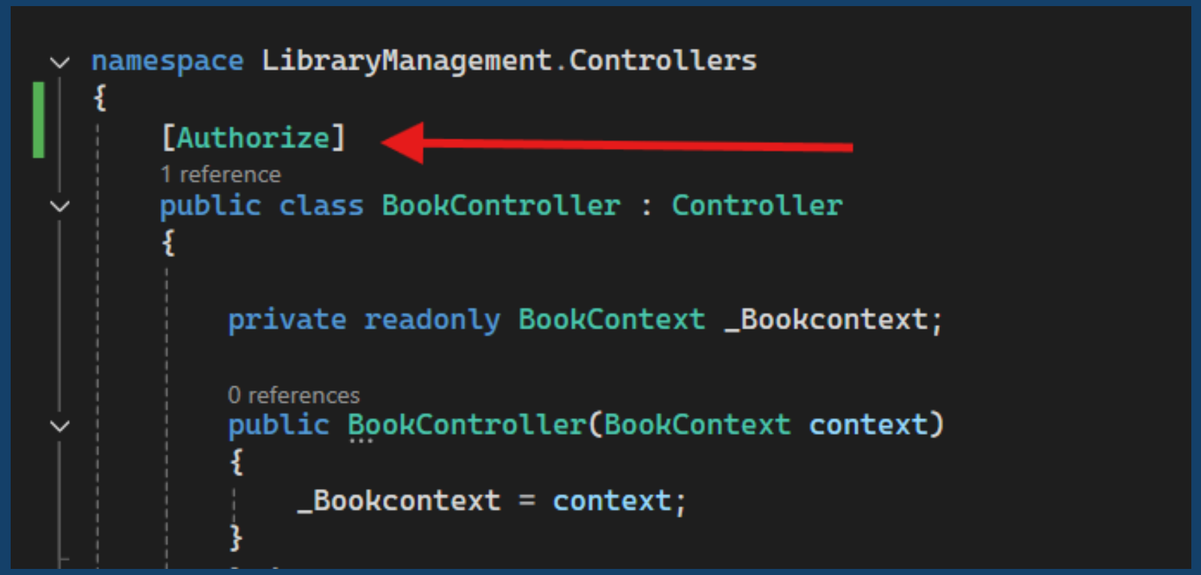
Authentication in ASP.NET Core MVC

Introduction:

- Authentication in MVC applications is essential for ensuring that only authenticated users can access specific parts of the application.
- ASP.NET Core MVC provides built-in support for authentication, leveraging middleware and attributes like [Authorize] to secure actions or controllers.

Key Points:

- MVC apps typically use cookie-based authentication.
- ASP.NET Core Identity or custom authentication schemes can be integrated.



Working with Authentication in MVC

Login and Logout Actions in MVC

Login in MVC:

- The login action authenticates a user and issues an authentication cookie.
- Typically, login views handle the username/password input and form submission.

Logout in MVC:

- The logout action removes the authentication cookie, effectively logging the user out.

E: > Desktop > CSE443 > C# code.cs

```
// Login action
[HttpPost]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await _signInManager
            .PasswordSignInAsync(
            model.Username,
            model.Password,
            model.RememberMe,
            lockoutOnFailure: false);
        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Home");
        }
    }
    return View(model);
}

// Logout action
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}
```

Working with Authentication in MVC

Handling Unauthorized Requests

Overview:

- When unauthenticated users try to access a secure page, they should be redirected to the login page.
- MVC applications can handle unauthorized requests by customizing the LoginPath in authentication options.

Handling Unauthorized Access:

- You can configure the middleware to redirect unauthorized users or display an error page.

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/Account/Login";  // Redirect here when unauthorized
    });
```

Working with Authentication in MVC

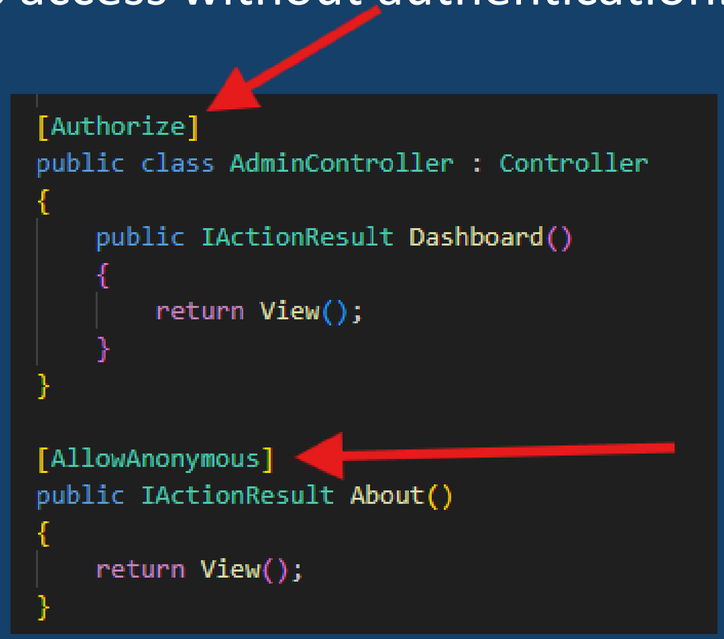
Securing Specific Controllers with Authentication

[Authorize] Attribute:

- ASP.NET Core MVC uses the [Authorize] attribute to protect specific controllers or actions.
- Only authenticated users can access these protected resources.

Key Options:

- **[Authorize]**: Requires any authenticated user.
- **[AllowAnonymous]**: Allows access without authentication.



```
[Authorize]
public class AdminController : Controller
{
    public IActionResult Dashboard()
    {
        return View();
    }
}

[AllowAnonymous]
public IActionResult About()
{
    return View();
}
```

Working with Authentication in MVC

Best Practices for MVC Authentication

Security Best Practices:

- **Use HTTPS** to secure user credentials and session data.
- **Strong Password Policies**: Enforce strong passwords for authentication.
- **Session Timeout Management**: Ensure sessions expire after inactivity.
- **Prevent Cross-Site Request Forgery (CSRF)**: Use anti-forgery tokens in forms.
- **Limit Cookie Scope**: Make authentication cookies HttpOnly and SameSite.

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.HttpOnly = true;  // Prevent XSS attacks
        options.Cookie.SameSite = SameSiteMode.Strict;  // Prevent CSRF
    });
```

Authentication in Web API

Authentication in ASP.NET Core Web API

Introduction:

- While MVC apps often use cookie-based authentication, Web APIs commonly rely on token-based authentication like JWT.
- Web APIs are stateless, so tokens must be passed with every request (usually in the Authorization header).

Key Differences:

- Web APIs typically do not have sessions or cookies for authentication.
- Authentication tokens (e.g., JWT) are used instead to authenticate each request.

Authentication in Web API

Using JWT Authentication in Web APIs

Introduction:

- JWT (JSON Web Tokens) is widely used in APIs because it’s stateless and easy to implement.
- The server issues a token after successful login, and clients must include it in the Authorization header for each request.

How JWT Authentication Works:

- The server verifies the token’s signature and extracts claims to identify the user.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "yourIssuer",
            ValidAudience = "yourAudience",
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("yourSecretKey"))
        };
    });
```

Authentication in Web API

Securing API Endpoints with [Authorize]

Using [Authorize] in Web API:

- Just like MVC, API endpoints can be secured using the [Authorize] attribute.
- Only authenticated users with a valid token can access these endpoints.

Custom Authorization Policies:

- You can also define custom policies based on claims or roles.

```
[Authorize]
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet]
    public IActionResult GetProducts()
    {
        return Ok(GetAllProducts());
    }
}
```

Authentication in Web API

API Authentication Error Handling

Handling Authentication Errors:

- When an API request contains an invalid or missing JWT token, the system should return an appropriate HTTP status code (401 Unauthorized or 403 Forbidden).

Common Scenarios:

- **401 Unauthorized:** The user is not authenticated (e.g., missing or invalid token).
- **403 Forbidden:** The user is authenticated but lacks permission to access the resource.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.Events = new JwtBearerEvents
        {
            OnAuthenticationFailed = context =>
            {
                context.Response.StatusCode = StatusCodes.Status401Unauthorized;
                return Task.CompletedTask;
            },
            OnChallenge = context =>
            {
                context.Response.StatusCode = StatusCodes.Status403Forbidden;
                return Task.CompletedTask;
            }
        };
    });
```

Authentication in Web API

Example: Securing a Web API with JWT

Complete Code Setup for JWT Authentication:Configure JWT in Program.cs:

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "yourIssuer",
            ValidAudience = "yourAudience",
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("yourSecretKey"
        };
    });
```

Authentication in Web API

API Authentication Error Handling

Issue JWT Token on Successful Login:

```
private string GenerateJwtToken(ApplicationUser user)
{
    var claims = new[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, user.UserName),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("yourSecretKey"));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
        issuer: "yourIssuer",
        audience: "yourAudience",
        claims: claims,
        expires: DateTime.Now.AddMinutes(30),
        signingCredentials: creds);

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

Authentication in Web API

API Authentication Error Handling

Secure API Endpoints with [Authorize]:

```
[Authorize]
[HttpGet("GetUserProfile")]
public IActionResult GetUserProfile()
{
    return Ok(new { Profile = "User Profile Data" });
}
```

Advanced Authentication Features

Advanced Authentication Scenarios

Multi-Factor Authentication (MFA):

- MFA adds an extra layer of security by requiring multiple forms of verification (e.g., password + SMS code).

Why MFA Matters:

- Helps prevent unauthorized access in case of compromised credentials.
- Can be combined with password-based or token-based authentication.

Popular MFA Methods:

- SMS, Email codes, Authenticator apps (Google Authenticator, Microsoft Authenticator).

Advanced Authentication Features

Token Refresh Mechanisms

Issue with Token Expiry:

- Short-lived tokens improve security but require mechanisms to refresh without frequent re-logins.

Refresh Tokens:

- Separate token issued along with the JWT to obtain a new access token without requiring re-authentication.
- Should be stored securely on the client side (e.g., in HttpOnly cookies).

```
[HttpPost("refresh-token")]
public IActionResult RefreshToken([FromBody] RefreshRequest request)
{
    // Validate refresh token and issue a new JWT token
    var newJwtToken = GenerateJwtToken(user);
    return Ok(new { Token = newJwtToken });
}
```

Advanced Authentication Features

Secure Token Storage

Importance of Secure Token Storage: Tokens, especially JWTs, must be stored securely to prevent theft and misuse (e.g., CSRF, XSS attacks).

Best Practices:

Client-Side Storage: Use HttpOnly and Secure cookies to store tokens instead of local storage or session storage.

Server-Side Validation: Always validate token expiration, issuer, audience, and signature.

```
var cookieOptions = new CookieOptions
{
    HttpOnly = true,
    Secure = true,
    SameSite = SameSiteMode.Strict
};
Response.Cookies.Append("jwt", token, cookieOptions);
```

Advanced Authentication Features

Working with External Providers (Google, Facebook)

Introduction to OAuth Providers:

- Many web applications allow users to authenticate via external providers like Google, Facebook, and Microsoft using OAuth 2.0.

Advantages of External Providers:

- Reduces the need to manage passwords.
- Provides a streamlined login experience for users.

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = GoogleDefaults.AuthenticationScheme;
})
.AddGoogle(options =>
{
    options.ClientId = "yourGoogleClientId";
    options.ClientSecret = "yourGoogleClientSecret";
});
```

Advanced Authentication Features

Authentication Across Multiple Tenants

What is Multi-Tenant Authentication?

- Multi-tenant applications serve multiple customers (tenants) with isolated authentication and authorization mechanisms.
- Each tenant may have separate identity providers or user stores.

Approaches to Multi-Tenant Authentication:

Database-per-tenant: Each tenant has a separate database for user credentials.

Tenant identification via domain/subdomain: Each tenant can have their own login page.

```
app.Use(async (context, next) =>
{
    var tenantId = context.Request.Headers["X-Tenant-ID"];
    // Load tenant-specific authentication options
    await next.Invoke();
});
```

Custom Authentication Handlers

Creating Custom Authentication Handlers

Why Create a Custom Handler?

- Default handlers (JWT, Cookie, OAuth, etc.) may not fit all use cases.
- Custom handlers allow you to implement unique authentication mechanisms (e.g., custom token formats, header-based authentication).

When to Use Custom Handlers?

- For specialized security requirements.
- When integrating with proprietary or legacy authentication systems.

Custom Authentication Handlers

Code Walkthrough: Custom Handler Implementation

Creating a Custom Authentication Handler:

- Inherit from AuthenticationHandler<TOptions> and implement the core logic for authentication.

Key Methods in Custom Handlers:

- HandleAuthenticateAsync: Main method to handle the authentication logic.
- HandleChallengeAsync and HandleForbiddenAsync: Handle cases for unauthenticated and unauthorized users.

```
public class CustomHeaderAuthenticationHandler : AuthenticationHandler<AuthenticationSchemeOptions>
{
    public CustomHeaderAuthenticationHandler(IOptionsMonitor<AuthenticationSchemeOptions> options,
        ILoggerFactory logger, UrlEncoder encoder, ISystemClock clock)
        : base(options, logger, encoder, clock) { }

    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        var token = Request.Headers["X-Custom-Token"];
        if (string.IsNullOrEmpty(token))
        {
            return Task.FromResult(AuthenticateResult.Fail("Token not provided"));
        }

        // Validate token logic here (e.g., check against database)
        var claims = new[] { new Claim(ClaimTypes.Name, "CustomUser") };
        var identity = new ClaimsIdentity(claims, Scheme.Name);
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, Scheme.Name);

        return Task.FromResult(AuthenticateResult.Success(ticket));
    }
}
```

Custom Authentication Handlers

Integrating Custom Handlers into Middleware

Adding the Custom Handler in Program.cs:

- The custom handler is registered similarly to built-in handlers, specifying its scheme and options.

Middleware Pipeline:

- The custom authentication handler is integrated into the middleware pipeline and processes each incoming request.
- Make sure to call UseAuthentication() before UseAuthorization().

```
builder.Services.AddAuthentication("CustomScheme")
    .AddScheme<AuthenticationSchemeOptions, CustomHeaderAuthenticationHandler>("CustomScheme", null);
```

Custom Authentication Handlers

Handling Authentication Failures in Custom Handlers

Common Failure Scenarios:

- Missing or invalid token/header.
- Expired tokens or session invalidation.
- Unrecognized authentication format.

Best Practices:

- Return meaningful error messages (e.g., 401 for unauthenticated, 403 for unauthorized).
- Log all failure events for security audits.

```
protected override Task<AuthenticateResult> HandleAuthenticateAsync()
{
    var token = Request.Headers["X-Custom-Token"];
    if (string.IsNullOrEmpty(token))
    {
        Logger.LogWarning("No token provided in request.");
        return Task.FromResult(AuthenticateResult.Fail("Token not provided"));
    }
    // Additional failure handling logic...
}
```

Custom Authentication Handlers

Final Review of Authentication in ASP.NET Core

Recap of Key Concepts:

- Authentication fundamentals and built-in options (Cookies, JWT, OAuth).
- Custom handlers for advanced scenarios.
- Securing MVC and Web API applications with authentication.

Best Practices:

- Use proper scheme selection and configuration.
- Secure token storage and refresh mechanisms.
- Ensure comprehensive logging and failure handling.

AUTHORIZATION

Introduction to Authorization

Authorization in ASP.NET Core

Definition:

- Authorization is the process of determining what a user can do after their identity has been authenticated.

Purpose:

- Controls access to resources and actions within an application based on user roles, claims, or policies.

Introduction to Authorization

What is Authorization?

Overview:

- Authorization restricts user access to specific resources, actions, or data based on their identity and permissions.

How it Works:

- Typically utilizes roles, claims, or policies defined within the application to enforce access control.

Example:

- An admin user might have access to all areas of an application, while a regular user might have restricted access.

Introduction to Authorization

Authentication vs. Authorization Recap

Definitions Recap:

- **Authentication:** Identifying who the user is.
- **Authorization:** Determining what an authenticated user is allowed to do.

Key Differences:

- Authentication is the first step; authorization follows and is contingent on successful authentication.

Introduction to Authorization

Authorization in MVC and Web API

Key Use Cases:

- **MVC Applications:**
  - Use of [Authorize] attribute to protect controllers and actions from unauthorized access.
  - Role and policy-based authorization for fine-grained control.
- **Web API Applications:**
  - Securing API endpoints with authorization filters to prevent unauthorized access.
  - Use of JWT and claims for securing Web API routes.

Introduction to Authorization

Overview of ASP.NET Core Authorization Options

Types of Authorization:

Role-based Authorization: Control access based on user roles.

Claim-based Authorization: Fine-grained access control based on user claims.

Policy-based Authorization: Custom authorization requirements can be defined and checked at runtime.

Code Example (ASP.NET Core 8.0): Registering Authorization Policies:

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));
    options.AddPolicy("ManagerAccess", policy =>
        policy.RequireClaim("Permission", "Manage"));
});
```

Role-Based Authorization

Introduction to Role-Based Authorization

Definition:

- Role-based authorization determines access rights based on user roles assigned within the application.

How It Works:

- Users are assigned one or more roles, and permissions are granted based on these roles.
- Common roles might include Admin, User, Manager, etc.

Role-Based Authorization

Configuring Roles in ASP.NET Core Identity

Setting Up Roles:

- Use RoleManager to create and manage roles in ASP.NET Core Identity.

Assigning Roles to Users:

- Users can be assigned roles using the UserManager.

```
public async Task SeedRoles(IServiceProvider serviceProvider)
{
    var roleManager = serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();
    string[] roleNames = { "Admin", "User", "Manager" };

    foreach (var roleName in roleNames)
    {
        var roleExist = await roleManager.RoleExistsAsync(roleName);
        if (!roleExist)
        {
            await roleManager.CreateAsync(new IdentityRole(roleName));
        }
    }
}
```

Role-Based Authorization

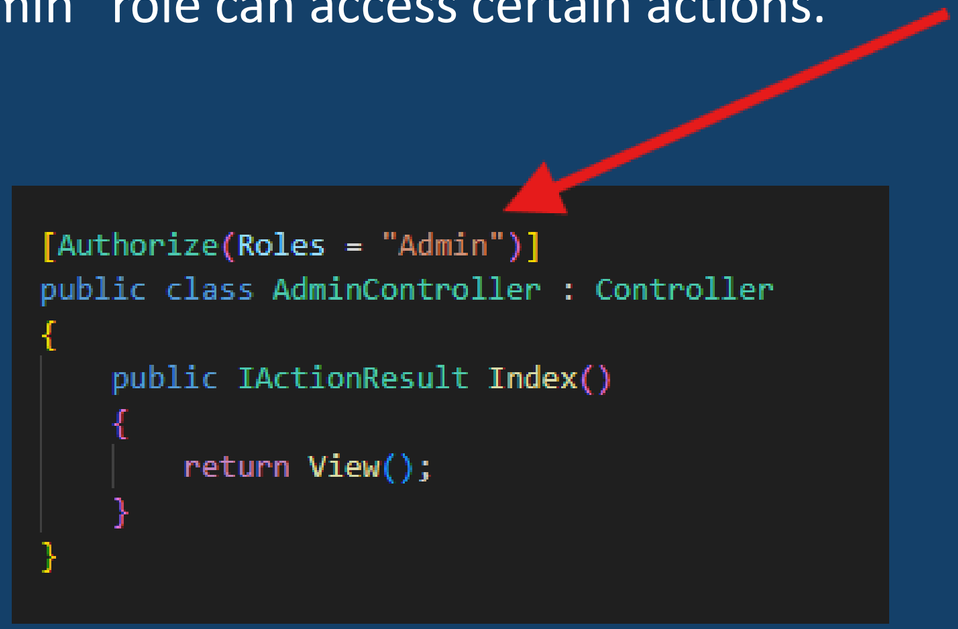
Applying [Authorize] by Role in MVC

Using the [Authorize] Attribute:

- Restrict access to controller actions based on roles.

Example:

- Only users in the "Admin" role can access certain actions.



```
[Authorize(Roles = "Admin")]
public class AdminController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Role-Based Authorization

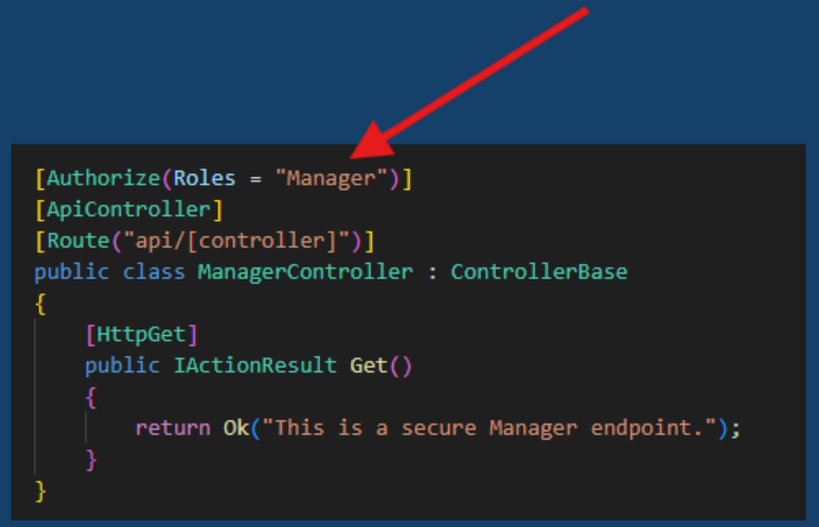
Role-Based Authorization in Web API

Securing API Endpoints:

- Apply role-based restrictions on Web API routes.

Example:

- Only users with the "Manager" role can access specific API endpoints.



```
[Authorize(Roles = "Manager")]
[ApiController]
[Route("api/[controller]")]
public class ManagerController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("This is a secure Manager endpoint.");
    }
}
```

Role-Based Authorization

Example: Role-Based Authorization in MVC

Complete MVC Example:

- Setting up roles and using the [Authorize] attribute.

```
public class HomeController : Controller
{
    // Action accessible to all authenticated users
    [Authorize]
    public IActionResult Index()
    {
        return View();
    }

    // Action restricted to Admin role
    [Authorize(Roles = "Admin")]
    public IActionResult AdminDashboard()
    {
        return View();
    }
}
```

Policy-Based Authorization

What is Policy-Based Authorization?

Definition:

- Policy-based authorization allows defining complex authorization rules based on multiple requirements (claims, roles, or custom logic).

How It Works:

- A policy consists of one or more requirements that the user must satisfy to gain access.

Use Case:

- Ideal for more granular and flexible authorization than role-based systems.

Policy-Based Authorization

Defining Policies in Program.cs

Registering Custom Policies:

- Policies are registered in Program.cs during service configuration.

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));
    options.AddPolicy("Over18Only", policy => policy.RequireClaim("Age", "18"));
});
```

Explanation:

- The AdminOnly policy ensures that only users with the "Admin" role can access certain resources.
- The Over18Only policy checks for the "Age" claim and restricts access to users over 18.

Policy-Based Authorization

Applying Policies in MVC and API

Using [Authorize] with Policies:

- Policies can be enforced by using the [Authorize] attribute in MVC or API controllers.

```
[Authorize(Policy = "AdminOnly")]
public class AdminController : Controller
{
    public IActionResult Dashboard()
    {
        return View();
    }
}

[Authorize(Policy = "Over18Only")]
[ApiController]
[Route("api/[controller]")]
public class AgeRestrictedController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("This endpoint is restricted to users over 18.");
    }
}
```

Policy-Based Authorization

Creating a Custom Policy Requirement

Custom Requirements:

- Custom logic can be defined in policy requirements to meet specific authorization needs.

Example:

- Custom policy to allow access only if a user has been a member for over a year.

Policy-Based Authorization

Creating a Custom Policy Requirement

Custom Requirements:

- Custom logic can be defined in policy requirements to meet specific authorization needs.

Example:

- Custom policy to allow access only if a user has been a member for over a year.

Policy-Based Authorization

Creating a Custom Policy Requirement

```
public class MinimumMembershipRequirement : IAuthorizationRequirement
{
    public int Months { get; }
    public MinimumMembershipRequirement(int months)
    {
        Months = months;
    }
}

public class MinimumMembershipHandler : AuthorizationHandler<MinimumMembershipRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, MinimumMembershipRequirement requirement)
    {
        var userJoinDate = DateTime.Parse(context.User.FindFirst(c => c.Type == "JoinDate").Value);
        if (userJoinDate.AddMonths(requirement.Months) <= DateTime.Now)
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}
```

This custom requirement ensures that users can only access resources if their membership duration is longer than the defined period (e.g., 12 months).

Policy-Based Authorization

Example: Policy-Based Authorization in Web API

Complete Example of Policy-Based Authorization:

- Combining multiple requirements into a policy.

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("MinimumMembershipPolicy", policy =>
        policy.Requirements.Add(new MinimumMembershipRequirement(12)));
});

[Authorize(Policy = "MinimumMembershipPolicy")]
[ApiController]
[Route("api/[controller]")]
public class MembersController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("You have been a member for over a year!");
    }
}
```

This example shows how to enforce the custom MinimumMembershipPolicy in a Web API controller.

Claim-Based Authorization

Introduction to Claim-Based Authorization

What Are Claims?

- Claims are key-value pairs representing information about the user (e.g., user role, age, email, or any other relevant detail).

How It Works:

- Claims are issued by an Identity provider and attached to the user's identity.
- Authorization decisions can be made based on the claims a user has.

Use Cases:

- Ideal for scenarios where access control is based on detailed user attributes (e.g., location, age).

Claim-Based Authorization

Configuring Claims in Identity

Claims in ASP.NET Core Identity: Claims can be added to users when they are created or modified.

```
var user = await _userManager.FindByEmailAsync("user@example.com");
var claim = new Claim("Department", "HR");
await _userManager.AddClaimAsync(user, claim);
```

Explanation: This example shows how to add a claim (e.g., "Department") to a user in an ASP.NET Core Identity system.

Claim-Based Authorization

Using [Authorize] with Claims

Claim-Based Authorization in MVC:

- The [Authorize] attribute can be used to restrict access based on claims.

```
[Authorize(Policy = "HRDepartmentOnly")]
public IActionResult HRDashboard()
{
    return View();
}
```

Policy Configuration in Program.cs:

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("HRDepartmentOnly",
    policy => policy.RequireClaim("Department", "HR"));
});
```

In this example, only users with the "HR" department claim can access the HRDashboard action.

Claim-Based Authorization

Example: Claim-Based Authorization in Web API

Claim-Based Security in Web API: Claims can be used to secure Web API endpoints in the same way as MVC controllers.

```
[Authorize(Policy = "AgeOver21")]
[ApiController]
[Route("api/[controller]")]
public class RestrictedContentController : ControllerBase
{
    [HttpGet]
    public IActionResult GetContent()
    {
        return Ok("This content is restricted to users over the age of 21.");
    }
}

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AgeOver21", policy => policy.RequireClaim("Age", "21"));
});
```

This example shows how to restrict access to a Web API endpoint based on the user's age claim.

Claim-Based Authorization

Best Practices for Claim-Based Authorization

Key Guidelines:

Minimize Claims Usage:

- Only use claims that are necessary for your authorization logic to avoid bloating user identities.

Secure Claims Sources:

- Ensure that claims are issued by trusted identity providers and are validated correctly.

Claim Storage:

- Use a secure and scalable mechanism to store and retrieve user claims, such as identity providers or claims databases.

Keep Claims Up-to-Date:

- Regularly update claims to reflect changes in user status or roles.

Authorization Policies in Depth

Advanced Policy-Based Authorization

Combining Multiple Policies:

- In ASP.NET Core, you can enforce multiple authorization policies on a single action or controller.
- Useful for scenarios where access control requires fulfilling multiple conditions.

Benefits of Combining Policies:

- Provides granular control over user access.
- Simplifies complex access logic by dividing conditions into smaller, reusable policies.

Authorization Policies in Depth

Applying Multiple Policies to Controllers

Applying Multiple Policies:

- Use multiple [Authorize] attributes or combine policies in one attribute.

```
[Authorize(Policy = "Over18")]
[Authorize(Policy = "HasEmployeeBadge")]
public IActionResult AccessEmployeeOnlySection()
    {
        return View();
    }
```

Explanation: In this example, the user must meet both the "Over18" and "HasEmployeeBadge" policies to access the action.

Authorization Policies in Depth

Custom Authorization Handlers

What is a Custom Authorization Handler?

- A custom handler allows you to write more complex logic for policy-based authorization.
- **Example Scenario:**
- You need to authorize based on a combination of user role, claim, and a database query.

```
public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   MinimumAgeRequirement requirement)
    {
        var birthDateClaim = context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth);
        if (birthDateClaim == null)
        {
            return Task.CompletedTask;
        }

        var birthDate = Convert.ToDateTime(birthDateClaim.Value);
        int calculatedAge = DateTime.Today.Year - birthDate.Year;
        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; }

    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
}
```

Explanation: This custom handler checks if the user meets a minimum age requirement based on their birth date claim.

Authorization Policies in Depth

Managing Complex Policies in MVC

Handling Complex Business Rules:

- Policies can be used to implement intricate business rules such as:
  - Hierarchical role structures.
  - Location-based access restrictions.
  - Time-based access (e.g., only allow access during working hours).

Tips:

- Break down complex rules into smaller policies and combine them.
- Keep business logic separate from authorization by using custom authorization handlers.

Example:

- Implement a policy that grants access only to managers during office hours, requiring both a role check and time check.

Authorization Policies in Depth

Example: Multi-Policy Authorization in Web API

Combining Policies in Web API:

- You can enforce multiple policies on API endpoints to control access more precisely.

```
[Authorize(Policy = "Over18")]
[Authorize(Policy = "HasValidSubscription")]
[HttpGet("api/securedata")]
public IActionResult GetSecureData()
{
    return Ok("This is secure data for users over 18 with a valid subscription.");
}

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy => policy.RequireClaim(ClaimTypes.DateOfBirth, CalculateAgeOver18));
    options.AddPolicy("HasValidSubscription", policy => policy.RequireClaim("SubscriptionStatus", "Active"));
});
```

The API endpoint /api/securedata is restricted to users who are over 18 and have an active subscription.

Securing Resources with Authorization

Introduction to Resource Authorization

What is Resource Authorization?

- Resource authorization focuses on controlling access to specific resources, such as files, data records, or API endpoints, based on user roles, claims, or policies.
- Unlike general authentication, this ensures users can only access resources they are permitted to interact with.

Importance:

- Essential for building secure systems where user access must be tightly controlled.

Securing Resources with Authorization

Role-Based Access to Resources

What is Role-Based Access?

- In role-based access, specific resources (e.g., files, data) are secured based on user roles.
- Example: Only users with the "Admin" role can delete records.

```
[Authorize(Roles = "Admin")]
public IActionResult DeleteRecord(int id)
{
    // Code to delete record
    return Ok();
}
```

Explanation:

- In this example, only users with the "Admin" role can access the DeleteRecord action.

Securing Resources with Authorization

Database Access Authorization

Controlling Database Operations:

- Restrict access to specific database records or operations based on the user’s roles or policies.
- Example: A user can only update or view their own records but cannot access others' data.

```
public IActionResult EditRecord(int id)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var record = _dbContext.Records.FirstOrDefault(r => r.Id == id && r.UserId == userId);

    if (record == null)
    {
        return Unauthorized();
    }

    // Code to edit the record
    return View(record);
}
```

Explanation: This example restricts users to only modify their own records in the database.

Securing Resources with Authorization

Securing Routes Dynamically: URL-Based Authorization

- Routes and URLs can be secured dynamically based on user identity, claims, or roles.
- Example: Restrict certain URLs or actions for users who do not meet certain conditions.

```
[Authorize(Policy = "AdminOnly")]
[Route("admin/{*url}")]
public IActionResult AdminAccess()
{
    return View();
}

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy =>
        policy.RequireRole("Admin"));
});
```

Explanation:

- In this example, all routes under /admin are secured, allowing only users with the "Admin" role.

Securing Resources with Authorization

API Resource Authorization

Fine-Grained Authorization for APIs:

- Implementing authorization at a more granular level, securing specific API resources or HTTP verbs (GET, POST, DELETE) based on user identity or policies.
- Example: Only users with specific claims can access sensitive API operations.

```
[Authorize(Policy = "CanEditData")]
[HttpPost("api/data/edit")]
public IActionResult EditData([FromBody] DataModel model)
{
    // Code to edit data
    return Ok();
}

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

Explanation:

- This example restricts access to the POST /api/data/edit API route to users who have the "EditData" claim.

Handling Authorization Failures

Dealing with Authorization Failures

What Happens When a User is Unauthorized:

- When a user tries to access a resource they are not authorized for, the system will block access.
- Unauthorized users will either be redirected to a login page (if not authenticated) or shown an error message (if authenticated but not authorized).

Common Responses:

- HTTP 403 Forbidden: When the user is authenticated but lacks permission.
- HTTP 401 Unauthorized: When the user is not authenticated.

Handling Authorization Failures

Customizing Access Denied Responses

Returning Custom Error Messages:

- ASP.NET Core allows customizing the response when users are denied access.
- You can show a friendly error page or return a custom JSON response in APIs.

```
options.AccessDeniedPath = "/Account/AccessDenied";
```

Redirect unauthorized users to a custom AccessDenied page.

Handling Authorization Failures

Forbid vs. Challenge

Difference Between Forbid and Challenge:

- **Forbid:** The user is authenticated but does not have permission to access the resource.
  - Returns **HTTP 403 Forbidden**.
- **Challenge:** The user is not authenticated and is prompted to log in.
  - Returns **HTTP 401 Unauthorized**.

```
if (User.Identity.IsAuthenticated)
{
    return Forbid();
}
else
{
    return Challenge();
}
```

Handling Authorization Failures

Handling Authorization in MVC

Graceful Handling of Authorization Failures in Views:

- Use custom error pages to inform users when they are denied access.
- Redirect users to relevant pages based on their roles or authorization status.

```
[Authorize(Roles = "Admin")]
public IActionResult AdminPage()
{
    if (!User.IsInRole("Admin"))
    {
        return RedirectToAction("AccessDenied", "Account");
    }

    return View();
}
```

Explanation: This redirects users to an "Access Denied" page when they try to access an unauthorized action.

Handling Authorization Failures

Handling Authorization in Web API

Returning HTTP 403 in APIs: In Web APIs, handle authorization failures by returning a **403 Forbidden** response or custom error messages in JSON format.

```
[Authorize(Policy = "AdminOnly")]
[HttpPost("api/secure-data")]
public IActionResult SecureEndpoint()
{
    if (!User.IsInRole("Admin"))
    {
        return Forbid();
    }

    return Ok("This is secured data");
}
```

Explanation: This example denies access with a **403 Forbidden** when users lack the required role.

Final Recap

Comparison of Authentication and Authorization

Authentication:

- Focuses on verifying *who* the user is.
- Involves methods like **passwords**, **tokens**, and **multi-factor authentication**.
- Common in login systems, where a user proves their identity (e.g., via email and password or external login).

Authorization:

- Focuses on determining *what* the user is allowed to do after they are authenticated.
- Role-based, claim-based, or policy-based restrictions control user access to resources.
- Example: Even after logging in, a user may only have access to certain pages or actions.

Interaction Between Authentication and Authorization:

- Authentication always comes first, followed by Authorization.
- Authorization only applies to users that have already been authenticated.

Final Recap

Best Practices for Securing ASP.NET Core Applications

For Authentication:

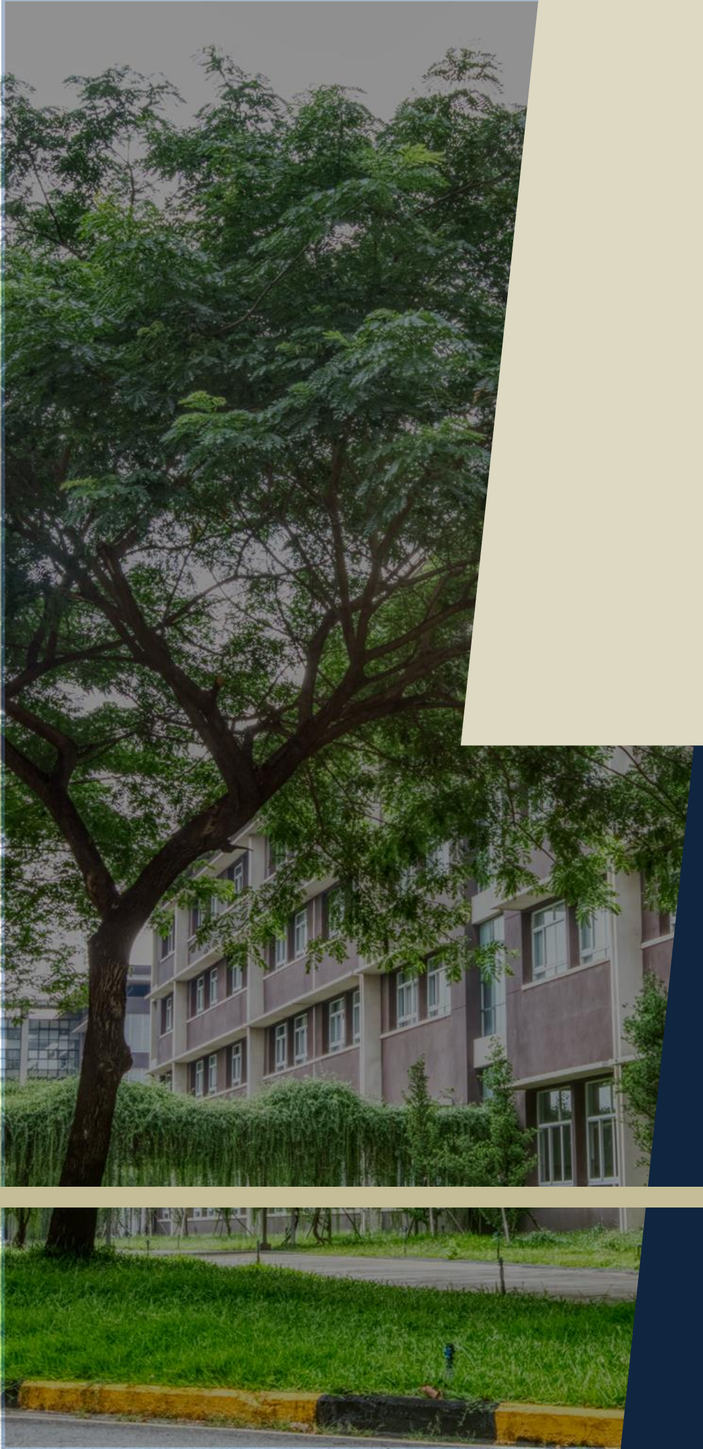
- Always use **HTTPS** to protect credentials and sensitive data in transit.
- Use **secure storage** for tokens and user credentials.
- Implement **multi-factor authentication (MFA)** where applicable.
- Keep external login providers (Google, Facebook, etc.) up to date with best practices.

For Authorization:

- Apply the **principle of least privilege**: only grant the permissions needed for each role.
- Use **role-based**, **claim-based**, or **policy-based authorization** appropriately for different scenarios.
- Keep sensitive routes, actions, and API endpoints secured with strict access controls.

General Security Tips:

- Regularly update the application to the latest version of **ASP.NET Core** to get security patches.
- Implement **logging and monitoring** to detect unauthorized access attempts or breaches.
- Perform regular **security audits** and **code reviews**.



EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

Thank You