COMP 2139

Authentication and Authorization – Part 1

Agenda

- Understanding Authentication and Authorization
- Introduction to ASP.NET Core Identity
- Implementing Authentication
- Implementing Authorization
- Customizing ASP.NET Core Identity
- Security Best Practices

Understanding Authentication and Authorization

Defining Authentication and Authorization

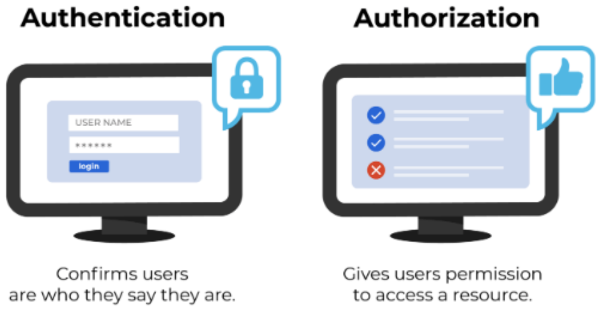
- **Authentication**: The process of verifying the identity of a user, device, or other entity in a computer system, often as a prerequisite to allowing access to resources in that system.
- **Authorization**: The process of determining whether a particular user, device, or entity has the right (**role**) to perform a specific action or access specific data within a system.

Key Points

Authentication answers the question, "**Who are you?**" while authorization answers, "**What are you allowed to do?**".

Differences Between Authentication and Authorization

- **Operational Phase:**
  - **Authentication** is the first step, ensuring that users are who they claim to be.
  - **Authorization** comes after, determining the permissions of the authenticated user.
- **Data Involved: Authentication** involves credentials like usernames and passwords, biometrics, or other verification codes. **Authorization** involves permissions and policies that control access to resources.

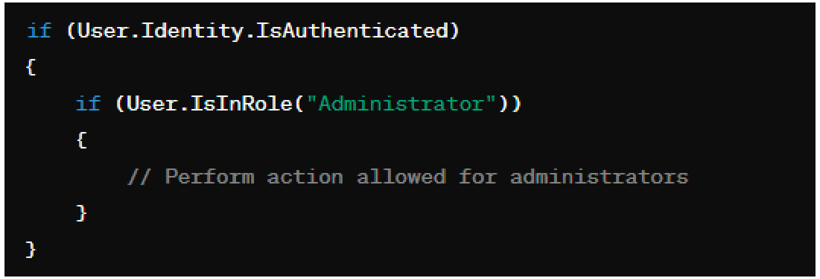

Authentication
USER NAME
******
login
Confirms users are who they say they are.
Authorization
Gives users permission to access a resource.

Role of Authentication and Authorization in Web Security

- **Security Foundation:** Authentication and authorization are foundational elements of web security, ensuring that sensitive information and critical functionalities are protected from unauthorized access.
- **Regulatory Compliance:** Many regulations require robust authentication and authorization mechanisms to protect user data and privacy.

Best Practices:

- Implement multi-factor authentication (**MFA**) to enhance security.
- Use **role-based access control** (RBAC) for fine-grained authorization.



```
if (User.Identity.IsAuthenticated)
{
    if (User.IsInRole("Administrator"))
    {
        // Perform action allowed for administrators
    }
}
```

Explanation: This code checks if a user is authenticated and then checks if the user belongs to the "**Administrator**" role to perform certain actions.

Introduction of ASP.NET Core Identity

Overview of ASP.NET Core Identity

- **Definition:** ASP.NET Core Identity is an extensible system that enables login functionality in ASP.NET Core applications. It supports user **authentication** and **authorization**.
- **Purpose:** Designed to integrate easily with ASP.NET Core applications, providing a robust framework for **managing users**, **passwords**, **profile data**, **roles**, **claims**, **tokens**, and more.

Key Points

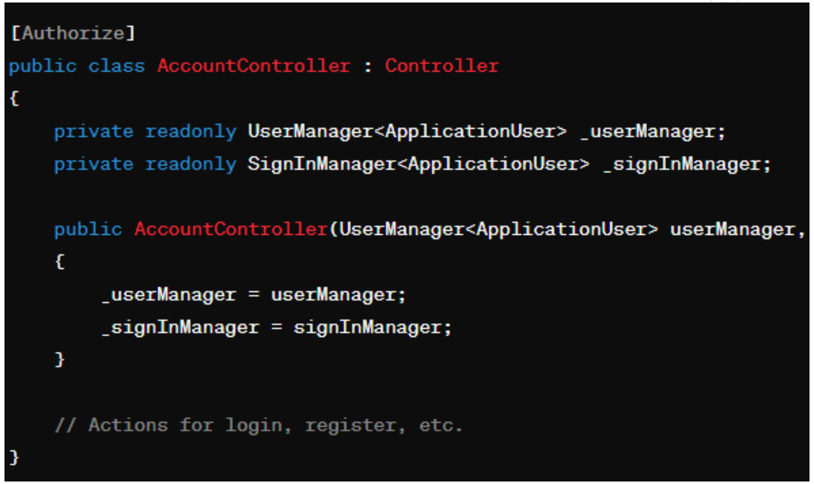
- Built-in support for storing user data in a database.
- Provides UI and APIs for common tasks such as **user registration**, **password recovery**, and **account management**.
- Documentation

Features and Capabilities of ASP.NET Core Identity

- **User Management:** Supports user creation, update, deletion, and querying.
- **Password Management:** Includes features like password hashing, password validation policies, and account lockout after multiple failed login attempts.
- **Security and Authentication:** Offers two-factor authentication (2FA), external login providers (Google, Facebook, etc.), and claims-based authentication.
- **Authorization:** Role-based and claims-based authorization for fine-grained access control.

How ASP.NET Core Identity Integrates with MVC

- **Seamless Integration:** ASP.NET Core Identity can be easily added to MVC applications through middleware. It works with the MVC pattern to secure web apps.
- **Use in Controllers and Views:** Utilize the **[Authorize]** attribute on controllers or actions to restrict access.
- Identity information is accessible via **UserManager** and **SignInManager** classes.



```
[Authorize]
public class AccountController : Controller
{
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly SignInManager<ApplicationUser> _signInManager;

    public AccountController(UserManager<ApplicationUser> userManager,
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }

    // Actions for login, register, etc.
}
```

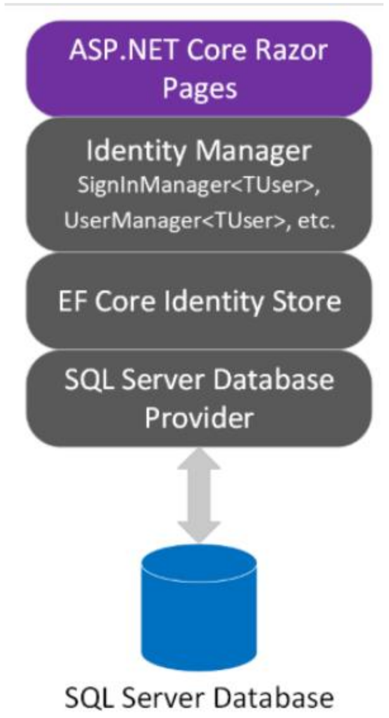
Explanation: This controller example demonstrates how ASP.NET Core Identity's **UserManager** and **SignInManager** are injected into a controller to manage user information and sign-in operations. The **[Authorize]** attribute restricts access to the controller to authenticated users.

ASP.NET Core Identity

- ASP.NET Core Identity (aka Identity) is Microsoft’s membership framework widely used by .NET developers for managing application users.
- Managing means everything that has to do with a **user accounts**, such as:
    - Creating an account
    - Login functionality (cookies, tokens, Multi-Factor Authentication, etc..)
    - Email confirmations
    - Resetting passwords
    - Using external login providers
    - Providing access to certain resources.
- **Identity, provides easy access to extremely useful helper and routine methods (centered around authentication and authorization) that would otherwise be cumbersome to implement independently.**
- Identity can be used with an ORM, EF Core with MS SQL Server being the default assumption.
- Identity works out-of-the-box (OOTB), that is without any customization.

ASP.NET Core Identity Architecture Basics

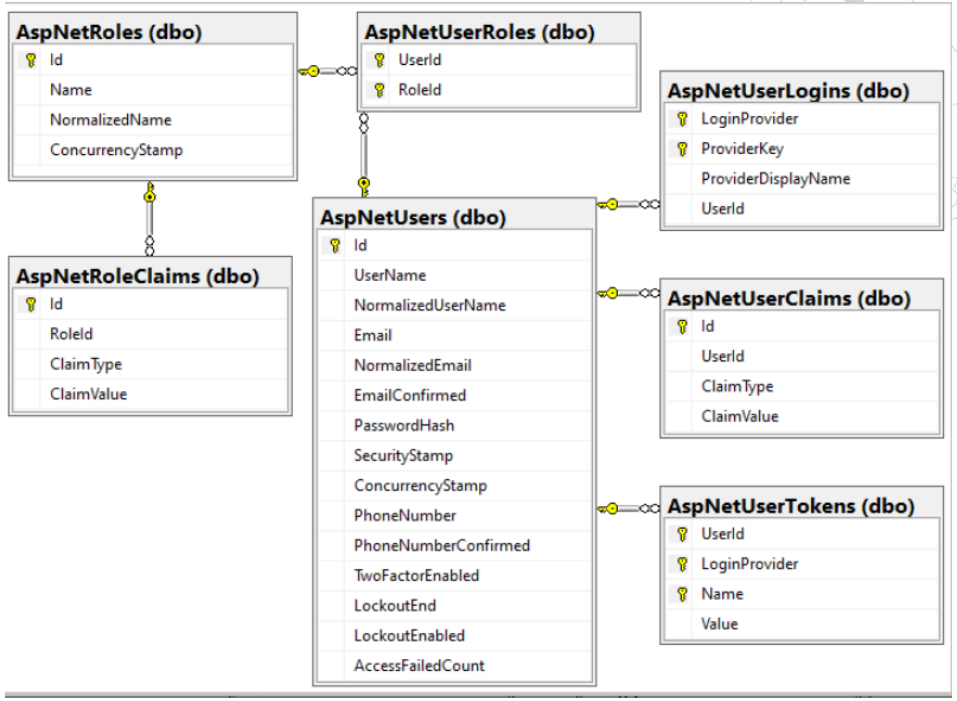
ASP.NET Core Identity



- **ASP.NET Core Razor Pages Layer:** Represent the UI to which Identity support will be added in this module
- **Identity Manager Layer:** Contains classes used from the **Microsoft.AspNetCore.Identity** namespace.
  - Example include:
    - **UserManager<IUser>**
    - **RoleManager<IUser>**
    - **SignInManager<IUser>**
- **EF Core Identity Store Layer:** contains classes from Microsoft.AspNetCore.Identity.EntityFrameworkCore namespace (ex IdentityUser etc..).
- **The Database Provider:** the database library - accepts SQL from EF Core and executes it

ASP.NET Core Identity Architecture

- After Applying the initial EF Core Identity migration, the supporting database tables, and their respective relationship created are as follows:



AspNetRoles (dbo)
- Id
- Name
- NormalizedName
- ConcurrencyStamp

AspNetUserRoles (dbo)
- UserId
- RoleId

AspNetUserLogins (dbo)
- LoginProvider
- ProviderKey
- ProviderDisplayName
- UserId

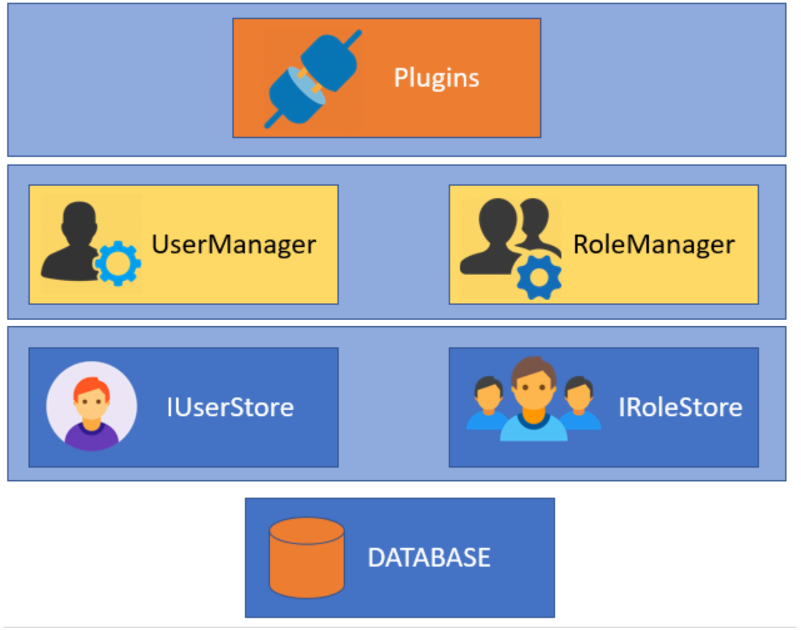
AspNetRoleClaims (dbo)
- Id
- RoleId
- ClaimType
- ClaimValue

AspNetUsers (dbo)
- Id
- UserName
- NormalizedUserName
- Email
- NormalizedEmail
- EmailConfirmed
- PasswordHash
- SecurityStamp
- ConcurrencyStamp
- PhoneNumber
- PhoneNumberConfirmed
- TwoFactorEnabled
- LockoutEnd
- LockoutEnabled
- AccessFailedCount

AspNetUserClaims (dbo)
- Id
- UserId
- ClaimType
- ClaimValue

AspNetUserTokens (dbo)
- UserId
- LoginProvider
- Name
- Value

ASP.NET Core Identity Architecture…



Extensions

Business Layer

Data Access Layer

Data

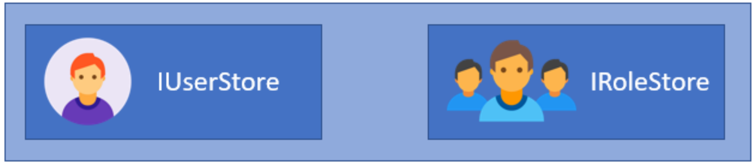
Data Access Layer: Identity Architecture Basics

<u>**Data Access Layer:**</u>

- These interfaces abstract the way the Identity membership schema is implemented in the database which means that this is <u>where you may write your own data access layer that saves and managing users on your own store and custom schema</u>.

- **IUserStore** is a required dependency for the next layer to work which means that an implementation must be provided for the library to work.

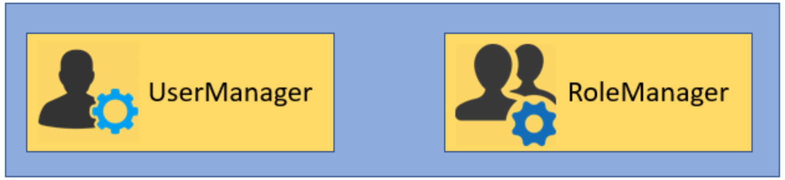
- Entity Framework provides an IUserStore implementation out-of-the-box which models a user as an **IdentityUser** in the database.



Business Layer: Identity Architecture Basics

Business Layer:

- This layer is utilized the **most**
- The business layer and is divided essentially into the **UserManager** and **RoleManager**.
- These managers hold all the business logic such as **validating user passwords** based on configuration or **checking that a user with the same username doesn't exist in the database during registration**.
- Under the hood managers **make calls to the data access layer**

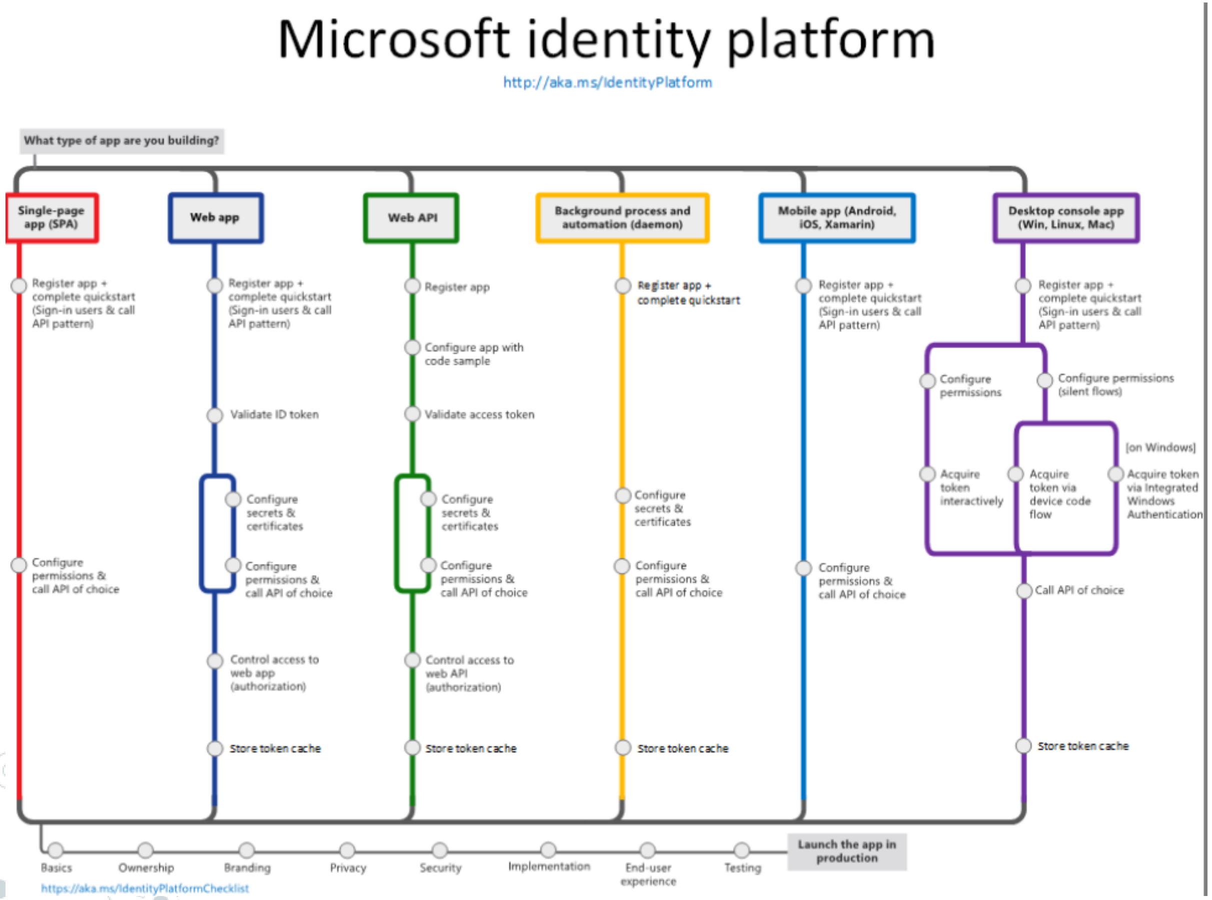


Plugins: Identity Architecture Basics

Plugins:

- The Plugins layer is a set of extensions.
- The most used plugin is the **SignInManager** which manages **sign-in operations for users**.
- These extensions sit on top of the managers by abstracting their logic or adding integration with other libraries.
- Signing a user in, using an external login is a simple example of these extensions.

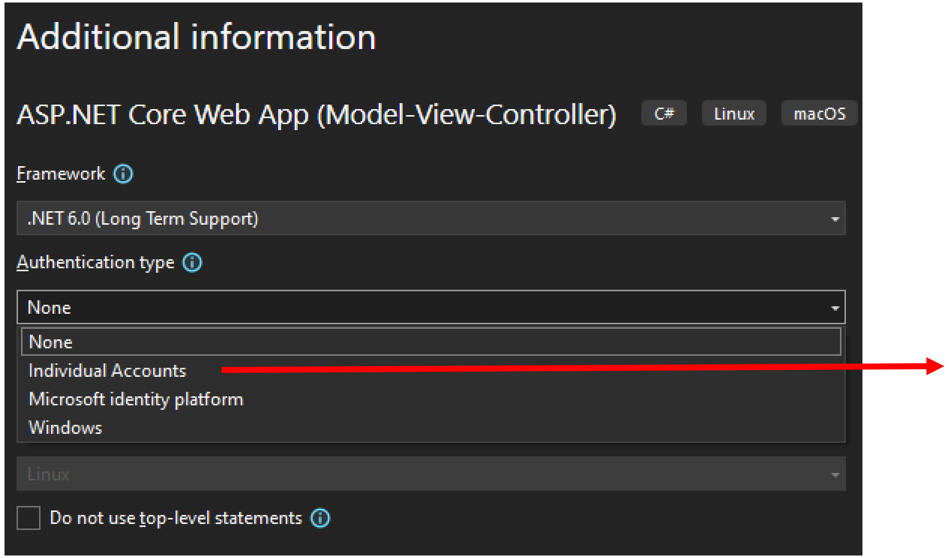
Common Authentication Scenarios


Microsoft identity platform
http://aka.ms/IdentityPlatform
What type of app are you building?
Single-page app (SPA)
Web app
Web API
Background process and automation (daemon)
Mobile app (Android, iOS, Xamarin)
Desktop console app (Win, Linux, Mac)
Register app + complete quickstart (Sign-in users & call API pattern)
Configure permissions & call API of choice
Register app + complete quickstart (Sign-in users & call API pattern)
Validate ID token
Configure secrets & certificates
Configure permissions & call API of choice
Control access to web app (authorization)
Store token cache
Register app
Configure app with code sample
Validate access token
Configure secrets & certificates
Configure permissions & call API of choice
Control access to web API (authorization)
Store token cache
Register app + complete quickstart
Configure secrets & certificates
Configure permissions & call API of choice
Store token cache
Register app + complete quickstart (Sign-in users & call API pattern)
Configure permissions & call API of choice
Register app + complete quickstart (Sign-in users & call API pattern)
Configure permissions
Configure permissions (silent flows)
[on Windows]
Acquire token interactively
Acquire token via device code flow
Acquire token via Integrated Windows Authentication
Call API of choice
Store token cache
Basics
Ownership
Branding
Privacy
Security
Implementation
End-user experience
Testing
Launch the app in production
https://aka.ms/IdentityPlatformChecklist

The most common app scenarios and their identity components

Creating an Identity Project

Authentication Types


Additional information

ASP.NET Core Web App (Model-View-Controller)

Framework

.NET 6.0 (Long Term Support)

Authentication type

None

Do not use top-level statements

“Individual Accounts” activates Microsoft Identity by default

Authentication Types…

Individual Accounts Authentication

- Is a traditional individual authentication platform.
- The application creates and manages users and ultimately allows those users to access and authenticate to a (one) specific application.
- Allows developers to code a **login page** that gets a username and password
- Encrypts the username and password entered by the user if the login page uses a secure connection
- Doesn’t rely on Windows user accounts

Authentication Types…

Microsoft Identity Platform

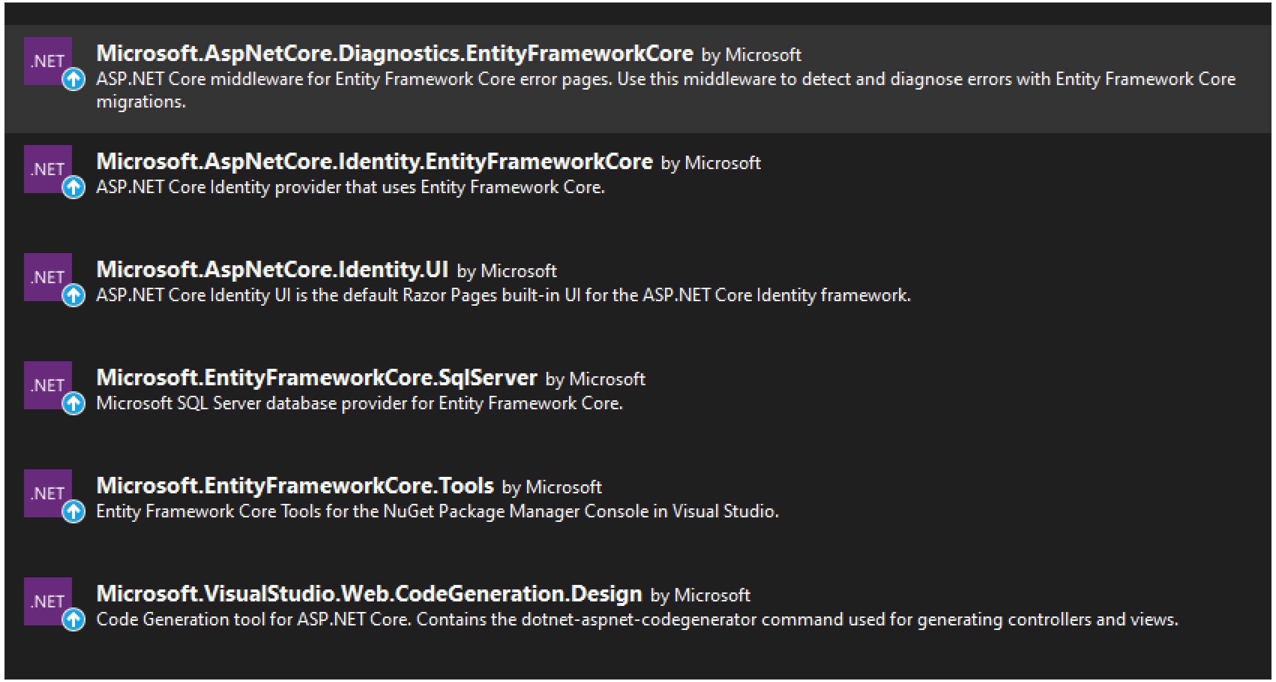
- Is a **centralized authentication** and authorization platform, independent of any one particular application.
- Typically, this is incorporated with the use of third party such as **Google**, **Facebook** and Microsoft using technologies like **OpenID** and **Oauth2**
- Allows users to use their existing logins and free developers from having to worry about the secure storage of user credentials
- Can issue identities or accept identities from other web applications and access user data on other servers.

Authentication Types…

Windows Authentication

- Causes the browser to display a login dialog when the user attempts to access a restricted page.
- Is supported by most browsers
- Is configured through IIS (Intranet Information Services) management console
- Uses Windows user accounts and directory rights to grant access to restricted resources
- Is (at most) only appropriate for internal intranet applications

Mandatory Packages



Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore by Microsoft
ASP.NET Core middleware for Entity Framework Core error pages. Use this middleware to detect and diagnose errors with Entity Framework Core migrations.

Microsoft.AspNetCore.Identity.EntityFrameworkCore by Microsoft
ASP.NET Core Identity provider that uses Entity Framework Core.

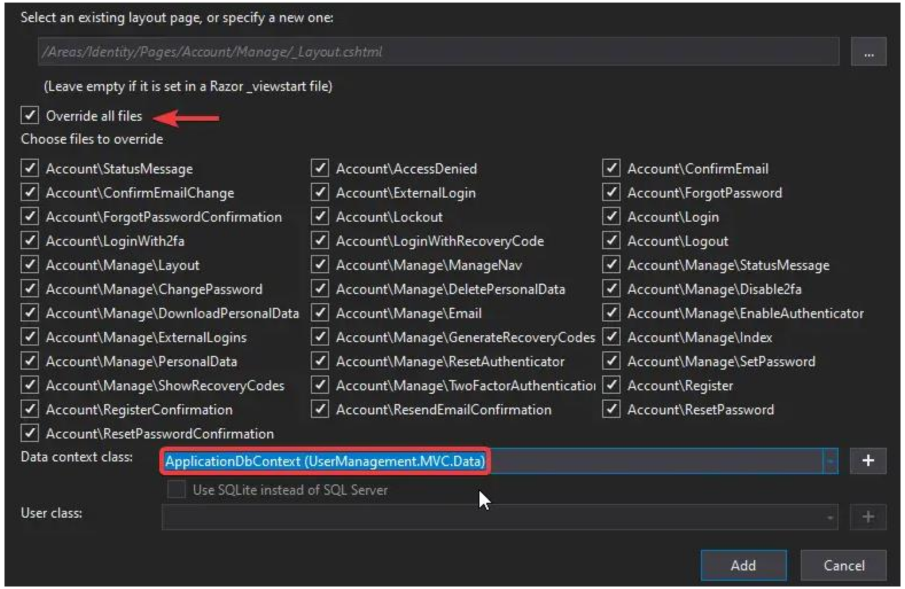
Microsoft.AspNetCore.Identity.UI by Microsoft
ASP.NET Core Identity UI is the default Razor Pages built-in UI for the ASP.NET Core Identity framework.

Microsoft.EntityFrameworkCore.SqlServer by Microsoft
Microsoft SQL Server database provider for Entity Framework Core.

Microsoft.EntityFrameworkCore.Tools by Microsoft
Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.

Microsoft.VisualStudio.Web.CodeGeneration.Design by Microsoft
Code Generation tool for ASP.NET Core. Contains the dotnet-aspnet-codegenerator command used for generating controllers and views.

Identity Pages



Select an existing layout page, or specify a new one:

/Areas/Identity/Pages/Account/Manage/_Layout.cshtml

(Leave empty if it is set in a Razor _viewstart file)

✓ Override all files

Choose files to override

✓ Account\StatusMessage	✓ Account\AccessDenied	✓ Account\ConfirmEmail
✓ Account\ConfirmEmailChange	✓ Account\ExternalLogin	✓ Account\ForgotPassword
✓ Account\ForgotPasswordConfirmation	✓ Account\Lockout	✓ Account\Login
✓ Account\LoginWith2fa	✓ Account\LoginWithRecoveryCode	✓ Account\Logout
✓ Account\Manage\Layout	✓ Account\Manage\ManageNav	✓ Account\Manage\StatusMessage
✓ Account\Manage\ChangePassword	✓ Account\Manage\DeletePersonalData	✓ Account\Manage\Disable2fa
✓ Account\Manage\DownloadPersonalData	✓ Account\Manage\Email	✓ Account\Manage\EnableAuthenticator
✓ Account\Manage\ExternalLogins	✓ Account\Manage\GenerateRecoveryCodes	✓ Account\Manage\Index
✓ Account\Manage\PersonalData	✓ Account\Manage\ResetAuthenticator	✓ Account\Manage\SetPassword
✓ Account\Manage\ShowRecoveryCodes	✓ Account\Manage\TwoFactorAuthenticatio	✓ Account\Register
✓ Account\RegisterConfirmation	✓ Account\ResendEmailConfirmation	✓ Account\ResetPassword
✓ Account\ResetPasswordConfirmation		

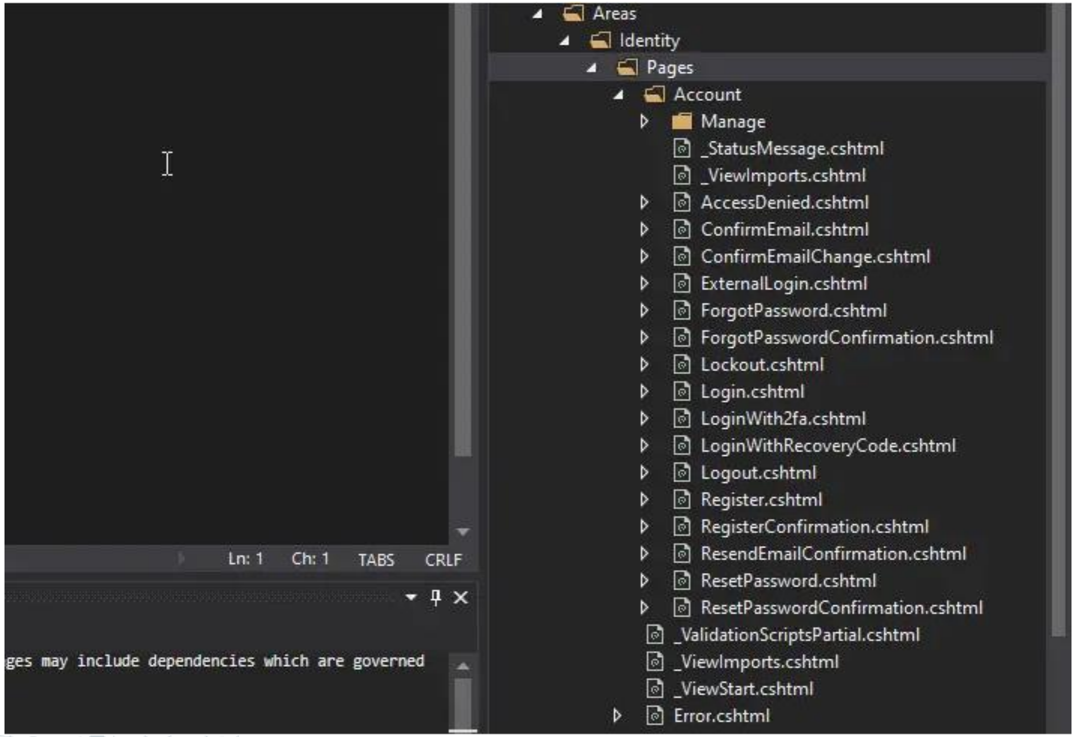
Data context class: ApplicationDbContext (UserManagement.MVC.Data)

Use SQLite instead of SQL Server

User class:

- In order to modify the existing Identity, Visual Studio/Rider allows you to generate the Identity Pages.
- You can achieve this by right-clicking the project and Adding a new Scaffolded Item (code generation).
- After that, you will be presented with a form containing over 50+ options to select from
- These are the Razor Page Versions (**not** Views only) of the Identity UI. You can choose the files you want to add to your project.

Identity Pages



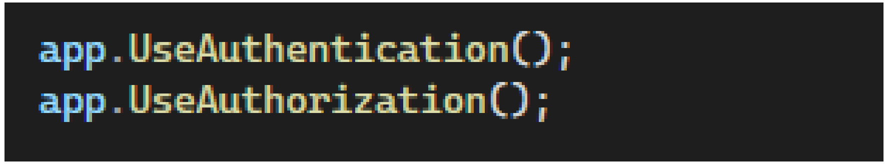
```
◢ Areas
  ◢ Identity
    ◢ Pages
      ◢ Account
        ▷ Manage
          _StatusMessage.cshtml
          _ViewImports.cshtml
        ▷ AccessDenied.cshtml
        ▷ ConfirmEmail.cshtml
        ▷ ConfirmEmailChange.cshtml
        ▷ ExternalLogin.cshtml
        ▷ ForgotPassword.cshtml
        ▷ ForgotPasswordConfirmation.cshtml
        ▷ Lockout.cshtml
        ▷ Login.cshtml
        ▷ LoginWith2fa.cshtml
        ▷ LoginWithRecoveryCode.cshtml
        ▷ Logout.cshtml
        ▷ Register.cshtml
        ▷ RegisterConfirmation.cshtml
        ▷ ResendEmailConfirmation.cshtml
        ▷ ResetPassword.cshtml
        ▷ ResetPasswordConfirmation.cshtml
        _ValidationScriptsPartial.cshtml
        _ViewImports.cshtml
        _ViewStart.cshtml
      ▷ Error.cshtml
```

```
ges may include dependencies which are governed
```

RazorPages

Configure ASP.Net Core with Authentication

Program.cs



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

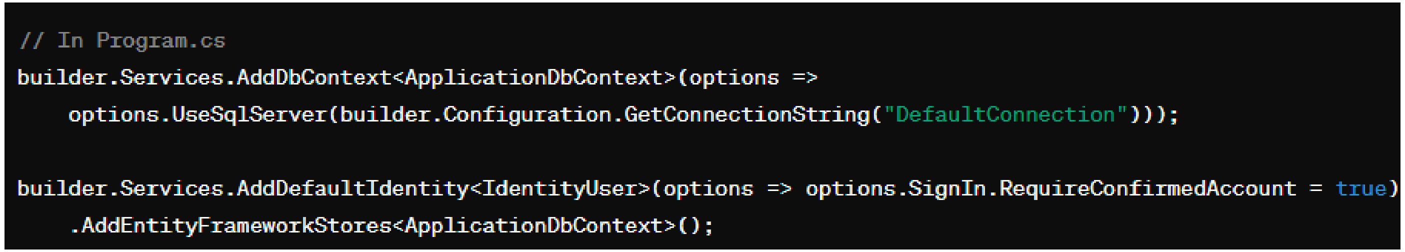
Implementing Authentication

Setting up ASP.NET Core Identity in an MVC Project

Introduction: ASP.NET Core Identity adds authentication and authorization functionality to ASP.NET Core applications.

Setup Steps:

1. **Install Identity**: Add the ASP.NET Core Identity package to your project.
2. **Configure Services**: Register Identity services in the **Program.cs**.
3. **Configure Identity Options**: Customize options such as password complexity and lockout settings.



```
// In Program.cs
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

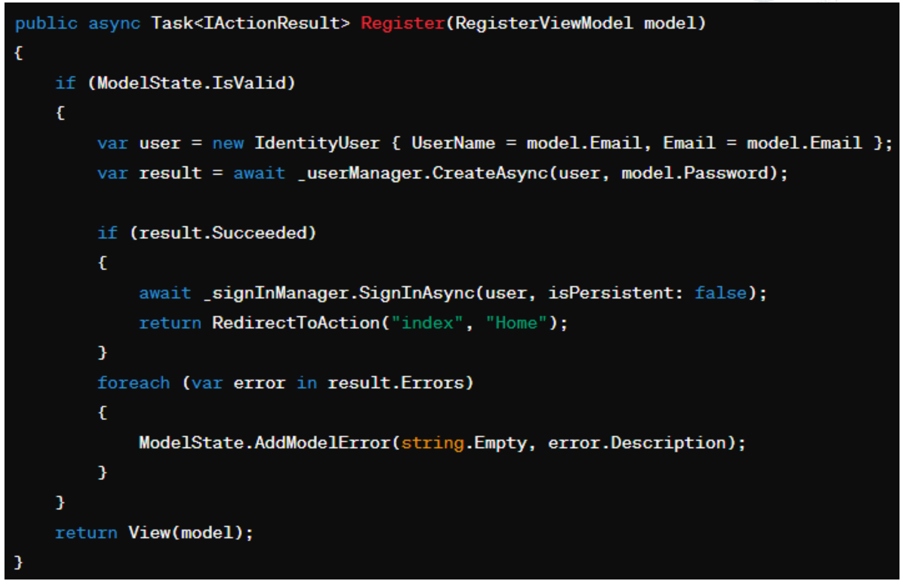
Explanation: This snippet shows the basic setup for ASP.NET Core Identity, configuring it with a database context and setting an option for sign-in.

Managing Users: Registration, Login, and Account Management

User Management: ASP.NET Core Identity simplifies user **registration**, **login**, and **account management** through built-in templates and scaffolding.

Implementing User Registration and Login:

- Utilize the **UserManager** and **SignInManager** for handling user and authentication operations.



```
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);

        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            return RedirectToAction("index", "Home");
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }
    return View(model);
}
```

Explanation: This example demonstrates handling a user registration request. It creates a user and signs them in upon successful registration.

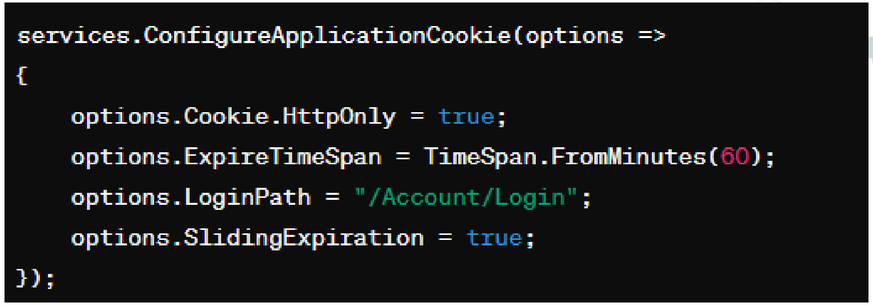
Understanding Cookies and Session Management

Cookies in Authentication: ASP.NET Core Identity uses cookies to maintain authentication state across HTTP requests.

Session Management: Configure session behavior such as timeout periods and persistence through Identity options.

Key Points:

- Authentication cookies are encrypted and secure.
- Session and authentication cookie settings can be customized for application needs.



```
services.ConfigureApplicationCookie(options =>
{
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Account/Login";
    options.SlidingExpiration = true;
});
```

Explanation: This configuration sets the properties of the authentication cookie, including making it **HTTP-only**, setting its **expiration time**, specifying the **login path**, and enabling **sliding expiration**.

Any questions?