ASP.NetCore Identity

Dependencies:-

- Asp.NetCore
- EntityFrameWorkCore

ASP.NET Core Microsoft Identity is a robust and flexible membership system that enables you to add authentication and authorization features to your web applications.

It is an API that manages users, passwords, profile data, roles, tokens, email confirmation, external logins etc.

Packages Needed :

- EntityFramework Packages : Microsoft.EntityFramework.Core, Microsoft.EntityFramework.Core.Tools, Microsoft.EntityFramework.Core.SqlServer
- **Microsoft.AspNetCore.Identity** : acts as a membership system or platform for other packages that implement Identity , installed where DbContext class Exists
- **Microsoft.AspNetCore.Identity.EntityFrameworkCore** : as we will use EntityFrameworkCore to create and Store Identity tables to database, repo functionalities will be write using Linq , installed twice one with the above one and in the presentation layer or Asp.NetCore API/MVC project as it will be used by IOC container

IdentityUser<T>

Acts as a base class for ApplicationUser class that acts as model class to store user details.

You can add additional properties to the ApplicationUser class.

T represent the type of the Id will be created in most cases you will set it as string , Long, or Guid

basic common properties

- Id , UserName , PasswordHash , Email, PhoneNumber

IdentityRole<T>

Acts as a base class for ApplicationRole class that acts as model class to store role details. Eg: "admin"

You can add additional properties to the ApplicationRole class.

Built-in Properties

- Id, Name

Steps:

- **Create a Custom User Class (ApplicationUser)**
    - Inherit from IdentityUser<T> and specify the type of the primary key (e.g., Guid, int, string).
- public class ApplicationUser : IdentityUser<Guid>
- {
- // Add any additional properties here
- }
- **Create a Custom Role Class (ApplicationRole)**
    - Inherit from IdentityRole<T> and specify the same key type used in ApplicationUser.
- public class ApplicationRole : IdentityRole<Guid>
- {
- // Add any additional properties here
- }
- **Create a Custom DbContext Class** (ApplicationDbContext)
    - Inherit from IdentityDbContext<TUser, TRole, TKey> and use the custom classes and key type
    - IdentityDbContext itself inherit from DBContext
    - IdentityDbContext contains all DBSets required for creating identity tables .
- public class ApplicationDbContext : IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
- {

- //options parameter carry configuration for the DBContext class
- public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
- : base(options)
- {
- }
- // Define DbSet properties for other entities here
- }
-
- **Registering Identity Services in IOC Container**
    - in this step we register identity services in IOC container to be used by controllers or other services for applying authorization logic
- // use ApplicationUser as table for user details , use ApplicationRole for User
- //Role table
- builder.Services.AddIdentity<ApplicationUser, ApplicationRole>()
- .AddEntityFrameworkStores<ApplicationDbContext>();
    - AddIdentity<TUser, TRole>() sets up all the core Identity services and dependencies required for managing users, roles, and authentication logic.
    - .AddEntityFrameworkStores<ApplicationDbContext>() integrates Identity with Entity Framework Core using the specified DbContext to persist data in the database. it tells the Identity Services , UserStore, RoleStore that EntityFramework will be used for managing users , roles data , and the DbContext used is ApplicationDbContext
    - Services registered into the DI container include:
        - UserManager<ApplicationUser> – Manages user-related operations such as creation, deletion, password handling, and user validation.

        - RoleManager<ApplicationRole> – Handles role creation, deletion, and role assignment.
        - SignInManager<ApplicationUser> – Manages the sign-in workflow including password sign-in, two-factor authentication, and external logins.
        - IUserStore<ApplicationUser> – Provides the mechanism for storing and retrieving user information from the database.
        - IRoleStore<ApplicationRole> – Provides similar functionality for roles.
        - Token providers – Used for generating tokens for password reset, email confirmation, etc.
        - Validators and password hashing services – Used internally by UserManager to validate users and securely store passwords.
    - Core functionalities enabled by these services:
        - User registration and login
        - Secure password hashing and validation
        - User and role management (CRUD)
        - Claims and role-based authorization
        - Email confirmation and password reset via tokens
        - Sign-in and security checks for user authentication
    - if you want to use Different Custom UserStore , RolStore , or different ORM you need to register your own custom UserStore, RoleStore as the following
    - builder.Services.AddIdentity<ApplicationUser, ApplicationRole>()
    - //.AddEntityFrameworkStores<ApplicationDbContext>()
    - .AddUserStore<MyCustomUserStore>()
    - .AddRoleStore<MyCustomRoleStore>();
- **use UserManager service for managing users data**

- Provides business logic methods for managing users. It provides methods for creating, searching, updating and deleting users.
- Core Methods :
  - CreateAsync(user: ApplicationUser)
  - DeleteAsync()
  - UpdateAsync()
  - IsInRoleAsync()
  - FindByEmailAsync()
  - FindByIdAsync()
  - FindByNameAsync()

- **use SignInManager service for signing user**
  - Provides business logic methods for sign-in and sign-in functionality of the users
  - in MVC Project default authorization schema is cookie so when the user is signed in then server create Cookie contains user details and sent it to the browser, when sending request again to the browser cookies sent with the request for proving the the user is logged in , server validate on it
  - **Methods:**
    - SignInAsync(user: ApplicationUser) // used for signing user after creating it ,
    - PasswordSignInAsync(email : string, password: string, isPersistent : boolean, lockoutOnFailure) // used for sign in user using its credential (username and password)
      - ispersistent : if true then the created cookie will be persisted in the browser meaning that when user close the browser and open it again user will still logged in
      - lockoutOnFailure : if true then after three attempts for wrong signing then the login for that account will be locked out

        - when PsswordSignInAsync is executed it hash the passed password , it make database query through UserStore , fetch first record for users table that match email and password then return success other wise return failure, when success identity cookie will be created and send to the browser
    - SignOutAsync() // used for sign out the user, when user Signed out Identity Cookie will be deleted from the browser
        - SignInManager.SignOutAsync() is a wrapper that calls:
        - //This method comes from the AuthenticationService and is part of the ASP.NET Core authentication middleware.
        - await HttpContext.SignOutAsync(IdentityConstants.ApplicationScheme);
        - ASP.NET Core Sets a Set-Cookie Header with Expired Cookie This is how the server **notifies the browser** to delete the authentication cookie, The response will contain a header like:
        - Set-Cookie: .AspNetCore.Identity.Application=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/; httponly; samesite=lax
        - This instructs the browser to overwrite the existing .AspNetCore.Identity.Application cookie with an **empty** value and an **expired timestamp**, causing it to be removed from the browser.
    - IsSignedIn() // used for checking if the user is signed in or not
- **use Authentication Middleware for Cookie check and Authenticating the user**
    - app.useAuthentication() middleware used for authenticating the user , used after app.UseRouting() as microsoft recommendation middleware
    - in case of MVC it check on cookie sent with the request extract user details from it and validate if that user is exists or not , cookie contains Id of the user and user details mean that the user already logged in

    - server does not store identity cookie for the user as it the responsibility of the browser to send cookie with the request

- **Password Complexity Configuration (optional step) : you can configure password complexity using options parameter while registering IdentityServices as the following**
- services.AddIdentity<ApplicationUser, ApplicationRole>(options => {
- options.Password.RequiredLength = 6; //number of characters required in password
- options.Password.RequireNonAlphanumeric = true; //is non-alphanumeric characters (symbols)
- required in password
- options.Password.RequireUppercase = true; //is at least one upper case character required in password
- options.Password.RequireLowercase = true; //is at least one lower case character required in password
- options.Password.RequireDigit = true; //is at least one digit required in password
- options.Password.RequiredUniqueChars = 1; //number of distinct characters required in password
- })
- .AddEntityFrameworkStores<ApplicationDbContext>()
- **Add Authorization : validate if the particular user has access to specific action or not**
    - add authorization middleware to the request pipeline
    - app.UseAuthorization()
    - restrict access to all action methods , and inforce using authorization filter for accessing the action through registering the authorization to the ioc container
    - services.AddAuthorization(options =>
    - {

- var policy = new AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build();
- // fallback policy will be applied if no policy provided be
- // the resource using Authorize attribute above the action method
- options.FallbackPolicy = policy;
- });
- configure the application cookie , redirect the user to specific URL when user has no access to the requested action method or resource
- Services.ConfigureApplicationCookie(
- {
- options =>
- {
- options.LoginPath = "/path-of-login-page"
- }
- })
- if there is any action method need to be accessed without user to be logged in like login action method , register user action method ,… then add [AllowAnonymous] attribute to that action method
- Authentication Middleware check if the current working user has account on the system and logged in or not , check if user name and password is belong to active account and that user is logged in or not
- authorization Middleware evaluate access for the current working user on particular resource, is it has access to that resource or not

- **use RoleManager for adding Roles to Users (Role Based Authentication Feature)**
- User-role defines type of the user that has access to specific resources of the application. Examples: Administrator role, Customer role etc.

Core Methods in RoleManager<TRole>

1. CreateAsync(TRole role)

- **Purpose:** Creates a new role in the system. return IdentityResult

2. UpdateAsync(TRole role)

- **Purpose:** Updates the details of an existing role. Return IdentityResult

3. DeleteAsync(TRole role)

- **Purpose:** Deletes the specified role. Returns IdentityResult

4. FindByIdAsync(string roleId)

- **Purpose:** Retrieves a role using its ID. **Returns:** TRole (e.g., IdentityRole)

5. FindByNameAsync(string roleName)

- **Purpose:** Retrieves a role using its name. **Returns:** TRole

6. RoleExistsAsync(string roleName)

- **Purpose:** Checks whether a role with the given name exists., **Returns:** bool

7. GetRoleNameAsync(TRole role)

- **Purpose:** Gets the name of the given role., **Returns:** string

8. SetRoleNameAsync(TRole role, string name)

- **Purpose:** Sets a new name for the role., **Returns:** IdentityResult

9. GetRoleIdAsync(TRole role)

- **Purpose:** Gets the ID of the specified role., **Returns:** string

10. NormalizeKey(string key)

- **Purpose:** Returns the normalized version of a role name or ID (used internally for case-insensitive lookups)., **Returns:** string

11. GetClaimsAsync(TRole role)

- **Purpose:** Gets all claims associated with the specified role., **Returns:** IList<Claim>

---

12. AddClaimAsync(TRole role, Claim claim)

- **Purpose:** Adds a claim to the specified role.
- **Returns:** IdentityResult
- **Example:**
- csharp
- CopyEdit
- await _roleManager.AddClaimAsync(role, new Claim("Permission", "CanEdit"));
-

---

13. RemoveClaimAsync(TRole role, Claim claim)

- **Purpose:** Removes a claim from the specified role.
- **Returns:** IdentityResult
- **Example:**
- csharp
- CopyEdit
- await _roleManager.RemoveClaimAsync(role, new Claim("Permission", "CanEdit"));
-

---

14. Roles (Property)

- **Purpose:** Gets all roles as a queryable collection.
- **Returns:** IQueryable<TRole>
- **Example:**
- var allRoles = _roleManager.Roles.ToList();
-
- **UserManager.AddToRoleAsync(user: ApplicationRole, roleName : string) assign user to specific role**

- **use Authorize Filter for applying role based authentication**

Role-based authentication is a way to **restrict access to certain parts of an application** based on the **roles assigned to users** (e.g., Admin, Manager, Customer). Roles help define what actions users can perform.

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


// multipe roles


[Authorize(Roles = "Admin,Manager")]
public IActionResult Dashboard()
{
   return View();
}
```

ASP.NET Core Identity automatically sets the user's roles as claims when using the default cookie authentication, so [Authorize(Roles = "...")] just works.

Areas

Areas are a way to **organize large ASP.NET Core MVC applications** into smaller functional sections (or modules). Each Area has its **own controllers, views, and models**, allowing better separation of concerns and maintainability.

Common Use Cases:

- Admin vs. Public sections
- Modular application design (e.g., Shop, Blog, Dashboard)
- Multi-team development (each team works on a different area)

Folder Structure of Areas

/Areas

/Admin

/Controllers

DashboardController.cs

/Views

/Dashboard

Index.cshtml

/Customer

/Controllers

OrdersController.cs

/Views

/Orders

Index.cshtml

/Controllers

HomeController.cs

/Views

/Home

Index.cshtml

How to Configure and Use Areas

1-Create an Area Controller

csharp

CopyEdit

```
using Microsoft.AspNetCore.Mvc;


namespace YourApp.Areas.Admin.Controllers
{
   [Area("Admin")]
   public class DashboardController : Controller
   {
      public IActionResult Index()
      {
         return View();
      }
   }
}
```

- The [Area("Admin")] attribute is mandatory to indicate the controller belongs to the "Admin" area.

---

2. Add the View

Path: /Areas/Admin/Views/Dashboard/Index.cshtml

html

CopyEdit

```
@{
   ViewData["Title"] = "Admin Dashboard";
}
<h2>@ViewData["Title"]</h2>
```

Welcome to the Admin Dashboard.

---

3. Configure Routing in Program.cs

csharp

CopyEdit

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    // Route for Areas
    endpoints.MapControllerRoute(
        name: "areas",
        pattern: "{area:exists}/{controller=Dashboard}/{action=Index}/{id?}");

    // Default route
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

- {area:exists} ensures that this route is only matched when an area is present in the URL.

---

4. Access the Area Route

To access the Admin Dashboard:

pgsql

CopyEdit

<https://localhost:5001/Admin/Dashboard/Index>

Create Custom Authorization

1. Using RequireAssertion(): Simple Inline Logic

📌 **Definition:**

The RequireAssertion() method allows you to write **inline custom logic** when defining a policy — no need to create separate classes.

✅ **Example: Allow access if user is in Admin role or has a claim Department = IT**

```
builder.Services.AddAuthorization(options =>
{
   options.AddPolicy("AdminOrIT", policy =>
      policy.RequireAssertion(context =>
         context.User.IsInRole("Admin") ||
         context.User.HasClaim("Department", "IT")));
});
```

🔐 **Usage in Controller:**

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

🧠 **When to Use:**

- You need simple logic (roles, claims).
- No need to share logic across multiple policies.
- No dependency injection is required.

Behind-the-Scenes Flow

1. **User sends request** to /AdminDashboard.
2. ASP.NET Core **detects** the [Authorize(Policy = "AdminOnly")] attribute.
3. The framework looks up the policy named "AdminOnly" that was added to the authorization options.
4. That policy contains **one custom requirement** created via RequireAssertion().
5. The framework **internally wraps** your lambda expression into a class called AssertionRequirement, which implements IAuthorizationRequirement.
6. The lambda you provided is **compiled and executed at runtime** by the framework inside a handler called AssertionHandler.
7. The framework **passes the AuthorizationHandlerContext** (which contains user claims, roles, etc.) to your lambda.
8. The lambda returns a bool:
    - If true, it calls context.Succeed(requirement) behind the scenes.
    - If false, the requirement is not fulfilled.
9. If **all requirements in the policy are satisfied**, access is **granted**.

Otherwise, it's **denied (403 Forbidden)**.

2. Using IAuthorizationRequirement + AuthorizationHandler: Full Custom Handler

📌 **Definition:**

You create a **requirement class** (which represents the rule) and a **handler class** (which contains the logic). This is the **official, recommended** way for complex rules.

✅ **Step-by-Step Example: Only allow access if user is older than 18**

🧩 **Step 1: Create a Requirement**

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

    public MinimumAgeRequirement(int age) => MinimumAge = age;
}
```

🛠 Step 2: Create the Handler

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

        var dob = DateTime.Parse(dobClaim.Value);
        int age = DateTime.Today.Year - dob.Year;
        if (dob > DateTime.Today.AddYears(-age)) age--;

        if (age >= requirement.MinimumAge)
            context.Succeed(requirement);

        return Task.CompletedTask;
    }
```

}

⚙️ **Step 3: Register in Program.cs**

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Over18Only", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(18)));
});


builder.Services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
```

🔐 **Step 4: Use in Controller**

```
[Authorize(Policy = "Over18Only")] public IActionResult AdultSection() { return View(); }
```