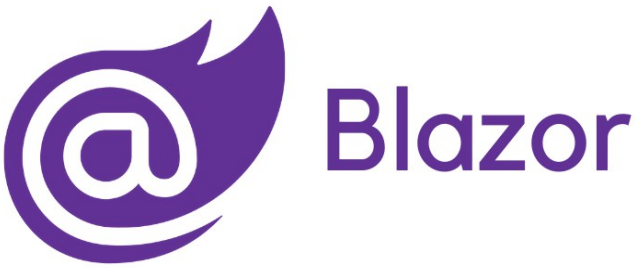

CSC530-A

Module 3

Implementing User Authentication & Authorization

1 Windows Authentication

1.1 Windows Authentication

Authentication is a process of being able to identify a user. For in-house applications, users can use their Windows account for authentication. The following package needs to be installed to support Windows authentication. Negotiate authentication scheme allow users to be authenticated through NTLM and Kerberos which are authentication protocols used in Windows and Windows Active Directory services.

Package to install

```
Microsoft.AspNetCore.Authentication.Negotiate
```

When building the application host, you can also build the authentication system used by the application. Call **AddAuthentication** to access the authentication builder and **AddNegotiate** to setup the Negotiate authentication service.

Setup the authentication service: Program.cs

```
builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme).AddNegotiate();
```

Note that your web browser will automatically login for you in an Intranet zone. You can disable this in the browser securty settings. To test the login process, change the **localhost** domain name to IP address **127.0.0.1** instead.

1.2 Authorization

Authentication is the means to identify the user but is not the means to control user's access to application resources which is called authorization. The following configures the authorization service to use the default authorization policy which requires users to be authenticated. As long as the user is authenticated, they are authorized access to the application.

Setting up the authorization policy

```
builder.Services.AddAuthorization(options => {
    options.FallbackPolicy = options.DefaultPolicy; });
```

ASP.NET comes with a set of authorization components that can be used in a Blazor application. You can import the following namespace to access those components in all your Razor components and pages.

Import authorization component namespace: _Imports.razor

```
@using Microsoft.AspNetCore.Components.Authorization
```

The authenticate state of the current user is not passed to all components in Blazor. Just like a theme provider can provide components with a standard theme to use, a authentication state provider can provide components with the authentication state. You can use **CascadingAuthenticationState** component to ensure components have access to the authentication state throughout the entire application.

Ensuring application has access to authentication state: App.razor

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(App).Assembly">
        :
    </Router>
</CascadingAuthenticationState>
```

To access authentication state, you can use a **AuthenticationStateProvider** service or through **HttpContext** which not only provide objects for **Request** and **Response** but also a **User** object to check authentication, retrieve user identity and to facilitate authorization through role checking. However **User** is only available to code running on the server that is activated as part of a HTTP request. Razor components only run on the server in a Blazor Server application.

You can replace **RouteView** with **AuthorizeRouteView** which is a combination of an **AuthorizeView** and **RouteView**. This stops the user from viewing a page activated by routing when the user is not authorized to access it. You can also provide alternate content to be shown.

Use AuthorizeRouteView to prohibit access to unauthorized pages

```
<AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
```

Provide alternate content when user is not authorized

```
<AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">
    <MudAlert Severity="Severity.Error">
        Access denied. Sorry, you are not authorized to access this page.
    </MudAlert>
</AuthorizeRouteView>
```

Even if user is allowed access to a page, you can use **AuthorizeView** to only provide content only to authorized users or provide alternative content.

Using AuthorizeView component

```
<MudAppBar Elevation="1">
            :
        <AuthorizeView><MudText>@context.User.Identity.Name</MudText></AuthorizeView>
        <MudIconButton Icon=@Icons.Material.Filled.MoreVert
            Color=Color.Inherit Edge=@Edge.End />
</MudAppBar>
```

Provide alternative content

```
<AuthorizeView>
    <Authorized><MudText>@context.User.Identity.Name</MudText></Authorized>
    <NotAuthorized><MudLink href="/login">Login</MudLink></NotAuthorized>
</AuthorizeView>
```

You can use the .NET **Authorize** attribute to control access to a page. You can import the following namespace to have direct access to this attribute in the page. Following shows

Import authorization namespace: _Imports.razor

```
@using Microsoft.AspNetCore.Authorization
```

Allow only authorized access to page

```
@page "/events"
@attribute [Authorize]
@inject EventDataService DataService
```

However this is pointless for Windows authentication as the default policy dictates the the user only needs to be authenticated. Since the user has to login to access the site they would automatically be guaranteed authorization for the entire application. So by default, you do not need to use **Authorize** attribute or **AuthorizeView** since users are already expected to be authorized due to the default authorization policy. You can create a custom authorization policy but this extra layer of complexity is not required as you can still customize the authorization through roles and claims. Window group accounts can be used as roles in authorization. Claims are additional information that is attached to each account, such as an email address or client certificate. Following shows that the page is only authorized for users that has a *WebAdmin* role. In terms of Windows authentication, their Windows account must be assigned to a *WebAdmin* local server Windows group. Use the Windows group *Everyone* for all users.

Allow access to only WebAdmin role: Pages/events/Create.razor

```
@page "/events/new"
@attribute [Authorize(Roles = "WebAdmin")]
```

Rather than restricting access to the entire page, you can also use **AuthorizeView** to hide content based on roles. Following shows how to hide the *Add Event* button in the **Events** page when the user does not have the specified role.

Show content only to WebAdmin role

```
<AuthorizeView Roles="WebAdmin">
    <MudButton Variant="Variant.Filled" Color="Color.Primary"
        Href="/events/new">ADD EVENT</MudButton>
</AuthorizeView>
```

In the case of Windows authentication, you should always use domain accounts and roles. The user can login into a domain server, which then allow access to all servers trusted under the domain, you can then use domain group accounts for the roles than only local groups on the local server you are accessing. Use the domain server name as part of the role name.

Using specific server or domain group

```
@attribute [Authorize(Roles = @"NewriseDomain\WebAdmin")]
```

Using authorization and roles, you can provide different navigation menus or links for the different group of users that are using the same application.

Customizing navigation menu based on roles: MainLayout.razor

```
<MudNavMenu>
    <MudNavLink Href="/" Match=@NavLinkMatch.All>Home</MudNavLink>
        <AuthorizeView Roles="WedAdmin">
                    :
        </AuthorizeView>
    <MudNavLink Href="/about">About</MudNavLink>
</MudNavMenu>
```

1.3 Authentication State Provider

For more complex authentication needs, you can access the authentication state using **AuthenticationStateProvider** service which is also used by **AuthorizeView**. Use **@inject** directive in a component or **Inject** attribute in a class for property injection to get access to the service. The authentication state will provide a **User** object that contains user identity, **IsInRole** method to check role membership and find methods to locate claims. It is the authenticator that provides the roles and claims for the user during the authentication process.

Service injection in component

```
@inject AuthenticationStateProvider AuthenticationStateProvider
```

Service injection in class

```
[Inject] public AuthenticationStateProvider AuthenticationStateProvider { get; set; }
```

Using AuthenticationStateProvider service

```
@page "/"
@inject AuthenticationStateProvider AuthenticationStateProvider

<MudText>@Message</MudText>

@code {
    string Message { get; set; } = string.Empty;

    protected override async Task OnInitializedAsync() {
        var state = await AuthenticationStateProvider.GetAuthenticationStateAsync();
        var isAuthenticated = state.User.Identity.IsAuthenticated;
        if (isAuthenticated) {
            var isAdmin = state.User.IsInRole("WebAdmin");
            var username = isAdmin ? "Administrator" : state.User.Identity.Name;
            Message = $"Welcome back {username}!";
        }   else Message = "Hello there!";
    }
}
```

Example of retrieving an email claim from user

```
var email = state.User.FindFirst(ClaimTypes.Email)?.Value;
```

2 ASP.NET Authentication

2.1 ASP.NET Identity

You can implement custom authentication but if you intend to authenticate against a database, ASP.NET already provides the facilities to use Entity Framework to create, manage users, roles and claims in a database and perform the authentication and to maintain the authentication state. This will reduce the amount of work that you need to implement authentication especially if you are building a new web application. First install the following package and then either create a new **DbContext** class or use an existing **DbContext** class to extend from **IdentityDbContext** class instead to inherit the **DbSet** properties for all the identity tables. If you did override **OnModelCreating** method, make sure to callback the base method that may contain code to configure the model for the identity tables.

Package to install

```
Microsoft.AspNetCore.Identity.EntityFrameworkCore
```

Ensure base context class is IdentityDbContext: Models\NewriseDbContext.cs

```
public class NewriseDbContext : IdentityDbContext {
                    :
        protected override void OnModelCreating(ModelBuilder builder) {
            base.OnModelCreating(builder);
            builder.Entity<Participant>().HasAlternateKey(p => p.Email);
            builder.Entity<Event>().Property("Fee").
                HasColumnType("decimal").HasPrecision(7, 2);
        }
                    :
```

We will need to add the migration for creating the identity tables in the database and then update the database using the migration class. You can check the migration class to determine what are the identity tables and what they contain.

Add migration and update the database

```
Add-Migration InitialIdentity
Update-Database
```

If you need to apply migrations quite often as you might be building the database as you work on your application, add the following package below to the project and call **AddDatabaseDeveloperPageExceptionFilter** on the application host builder which captures database-related issues that can be resolved through migrations and call the **UseMigrationsEndPoint** on the application host during development so it can detect if migrations are not applied.

Package to install

```
Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

Provide better diagnostics of database-related issues

```
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
```

Enable auto-detection of un-applied migrations

```
if (app.Environment.IsDevelopment()) {
    app.UseMigrationsEndPoint();
}
```

Note that if a migration has already been applied, you need to rollback the changes to the database by using **Update-Database** with a prior migration first.

Rollback and remove migration

```
Update-Database InitialCreate
Remove-Migration
```

You can customize the identity tables by creating your own user class extended from **IdentityUser** and role class extended from **IdentityRole** class. Then use the generic version of **IdentityDbContext** to utilize your custom classes.

Custom identity classes

```
public class MyUser : IdentityUser { ... }
public class MyRole : IdentityRole { ... }
public class MyDbContext : IdentityDbContext<MyUser, MyRole, string> { ... }
```

Install the following package to get the required services and components to integrate ASP.NET Identity into your application. Update your application host builder to add in the identity service and specify the storage used for the identity tables. Remember to replace **IdentityUser** if you are using a custom user class. Call **UseAuthorization** to enable authorization middleware.

Package to install

```
Microsoft.AspNetCore.Identity.UI
```

Add identity service and register DbContext

```
builder.Services.AddDefaultIdentity<IdentityUser>(options => {
        options.SignIn.RequireConfirmedAccount = true;
        options.User.RequireUniqueEmail = true;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = true;
        options.Password.RequireNonAlphanumeric = true;
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
}).AddEntityFrameworkStores<NewriseDbContext>();
```

Enable authorization middleware

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

2.2 Default Identity UI

We will first create a **LoginDisplay** razor component that display different content based on the authentication state of the user. Create a folder named **Shared** to store components that we can use from any page. Create the following component where it will display a link that shows the user account name that will redirect to a logout page if the user is authorized. If the user is not authorized, it will display two links; one to registration page and another to a login page.

Creating a login state component: Shared\LoginDisplay.razor

```
<AuthorizeView>
    <Authorized>
        <MudLink Color=@Color.Inherit Href="Identity/Account/Logout">
            @context.User.Identity?.Name</MudLink>
    </Authorized>
    <NotAuthorized>
        <MudLink Color=@Color.Inherit Href="Identity/Account/Register">Register</MudLink>
        <MudText> | </MudText>
        <MudLink Color=@Color.Inherit Href="Identity/Account/Login">Login</MudLink>
    </NotAuthorized>
</AuthorizeView>
```

Import namespace for shared components: _Import.razor

```
@using Newrise.Shared
```

Using the component in the layout: Shared\MainLayout.razor

```
<MudAppBar Elevation="1">
            :
    <MudText Typo=@Typo.h5>Newrise</MudText>
    <MudSpacer />
    <LoginDisplay />
    <MudIconButton Icon=@Icons.Material.Filled.MoreVert Color=Color.Inherit Edge=@Edge.End />
</MudAppBar>
```

Notice that you will get a runtime error when you click on any of the links. Notice that the error tells you that a particular **_LoginPartial** view is missing and the locations it is expected to be in. If you wish to use the default Identity UI you need to create this view in one of the locations shown. Create the directories and the file as shown below and then paste the default content as provided.

Expected location and name of the partial view

```
Areas/Identity/Account/Shared/_LoginPartial.cshtml
```

The following view generates a basic navigation bar and then uses a **SignInManager** service provided by Identity UI to verify if the user is already signed-in. If the user is signed-in, it will display the user account name and a form to submit to logout since it requires a POST request to do so since links produces a GET request. If user has not signed-in, a register form or a login form will be displayed depending on the original route using to access the page.

Content of the partial view: _LoginPartial.cshtml

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<ul class="navbar-nav">
    @if (SignInManager.IsSignedIn(User)) {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity"
                asp-page="/Account/Manage/Index" title="Manage">
                Hello @User.Identity?.Name!</a>
        </li>
        <li class="nav-item">
            <form class="form-inline" asp-area="Identity"
                asp-page="/Account/Logout" asp-route-returnUrl="/" method="post">
                <button type="submit" class="nav-link btn btn-link text-dark">Logout</button>
            </form>
        </li>
    }
    else {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity"
                asp-page="/Account/Register">Register</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity"
                asp-page="/Account/Login">Login</a>
        </li>
    }
</ul>
```

Update home page to show authentication status: Pages\Index.razor

```
@page "/"
<AuthorizeView>
    <Authorized><MudText>Welcome back!</MudText></Authorized>
    <NotAuthorized>
        <MudAlert Severity="Severity.Warning">
            You are not logged-in. Please log-in or register a new account.
        </MudAlert>
    </NotAuthorized>
    <Authorizing>
        <MudAlert Severity="Severity.Info">
            We're currently authenticating your account. Please wait...
        </MudAlert>
    </Authorizing>
</AuthorizeView>
```

You should now be able to register, login and logout using the default UI provided by ASP.NET Identity. Since we requested that the account requires confirmation before use but we did not provide any email server information, it will provide a confirmation page to confirm the account. Eventually we will configure and customize the UI but it is easier to use what is provided by default so we can get the authentication and the authorization up as soon as possible.

2.3 Sign-In Manager

The **AddDefaultIdentity** does not support **Roles** authorization. You will have access to **SignInManager** service to sign in and out and a **UserManager** service to manage users. It also provides default UI for registration, login, account confirmation, logout and so on. You can use **AddIdentity** instead but this would not provide a default UI but you can call **AddDefaultUI**. Alternatively, you can still use **AddDefaultIdentity** but call **AddRoles** additionally to provide support for roles including a **RoleManager** to manage roles. Note to replace **IdentityUser** and **IdentityRole** if you are using custom identity classes.

Using AddIdentity to support roles and optionally use DefaultUI

```
builder.Services.AddIdentity<IdentityUser, IdentityRole>(options => { ... })
    .AddDefaultUI()
    .AddEntityFrameworkStores<NewriseDbContext>();
```

Adding roles support to AddDefaultIdentity

```
builder.Services.AddDefaultIdentity<IdentityUser>(options => {...})
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<NewriseDbContext>();
```

If you want to implement your own UI, you would need to know how to make use of the **SignInManager**, **UserManager** and **RoleManager** services. You can get access to these services through constructor or property injection. Following is a service class that makes use of them to perform identity-related operations.

Using identity services: Services\UserAccountService.cs

```
public class UserAccountService {
    private SignInManager<IdentityUser> _signInManager;
    private UserManager<IdentityUser> _userManager;
    private RoleManager<IdentityRole> _roleManager;
    public UserAccountService(SignInManager<IdentityUser> signInManager,
        UserManager<IdentityUser> userManager, RoleManager<IdentityRole> roleManager) {
            _signInManager = signInManager;
            _userManager = userManager;
            _roleManager = roleManager;
    }
}
```

For signing in and out, use **PasswordSignInAsync** to sign in using a username and a password. If successful, it will create an authentication cookie and sent back to the web browser to maintain the authentication state. To create a persistent cookie, pass in **true** for the third parameter.

Using SignInManager to sign-in the user

```
public async Task LoginAsync(string userName, string password, bool rememberMe) {
    var result = await _signInManager.PasswordSignInAsync(
        userName, password, rememberMe, false);  if (result.Succeeded) return;
    if (result.IsLockedOut) throw new Exception("This account has been locked.");
    if (result.IsNotAllowed) throw new Exception("This account is not allowed for login");
    throw new Exception("Invalid user name or password.");
}
```

You can call **ConfigureApplicationCookie** to customize the authentication cookie. Use **Cookie.Name** to assign a custom name to the cookie. You may have multiple applications on your web server that you wish to share the same cookie. To support sub-domains, you should set the **Cookie.Domain** to a non-specific domain, example if you set the domain to *.newrise.com.my*, the cookie can be sent to multiple domains like *www.newrise.com.my* or *training.newrise.com.my* or *admin.newrise.com.my*. Use the following link to learn more about sharing cookies.

Learn more about sharing cookies

```
https://learn.microsoft.com/en-us/aspnet/core/security/cookie-sharing?view=aspnetcore-7.0
```

Configuring the authentication cookie: Program.cs

```
builder.Services.ConfigureApplicationCookie(configure => {
    configure.Cookie.Name = "newrise_app_cookie";
    configure.ExpireTimeSpan = TimeSpan.FromDays(7);
    configure.SlidingExpiration = true;
    configure.Cookie.HttpOnly = true;
});
```

In the above code, we change the name of the cookie, set an expiration of 7 days so the user can access the site for a week without having to login again. This expiration is sliding so the user gets another week if they accessed the site again anytime within that period. Setting **HttpOnly** ensures that client-scripts cannot access the cookie, it is only retrieved and stored through network communication, a feature to improve the security.

The following shows how to logout the user with the **SignInManager**. If the cookie is a session cookie, not a persistent cookie, closing the browser window will imply logout but to remove a persistent cookie, you must specifically sign-out using the method as shown below which expires the cookie immediately.

Removing the authentication cookie: Services\UserAccountService.cs

```
public async Task LogoutAsync() {
    await _signInManager.SignOutAsync();
}
```

2.4 UserManager & RoleManager

You can use **CreateAsync**, **UpdateAsync** and **DeleteAsync** methods available from **RoleManager** to manage roles. You do not have to update the identity tables directly. Following shows how to add a new role by using the **RoleManager**.

Adding a new identity role

```
public async Task AddRoleAsync(string roleId, string roleName = null) {
    if (string.IsNullOrEmpty(roleName)) roleName = roleId;
    var role = new IdentityRole { Id = "admin", Name = roleName };
    var result = await _roleManager.CreateAsync(role);
    if (!result.Succeeded) throw new Exception("Role may already exist.");
}
```

UserManager provides **CreateAsync**, **UpdateAsync** and **DeleteAsync** methods to manage users and also methods to assign and remove roles from users. The following code shows how to add a new user and optionally add the user to a specific role.

Add a new identity user

```
public async Task AddUserAsync(string userName, string email,
    string password, string roleName = null) {
    var isAdmin = roleName == "admin";
    if (roleName != null) {
        var role = await _roleManager.FindByIdAsync(roleName);
        if (role == null) throw new Exception("Invalid role name.");
    }
    if (string.IsNullOrEmpty(email)) email = userName;
    var user = new IdentityUser {
        Id = Guid.NewGuid().ToString(),
        UserName = userName, Email = email,
        EmailConfirmed = isAdmin,
        LockoutEnabled = !isAdmin
    };
    var result = await _userManager.CreateAsync(user, password);
    if (!result.Succeeded) throw new Exception("User may already exist.");
    if (roleName != null) await _userManager.AddToRoleAsync(user, roleName);
}
```

You would probably want to ensure some initial roles and administrative accounts are already created. We will add methods to check whether a particular role and user has already been created, if not we will create the role and user. Following is a **HasRole** and **HasUser** method to check for existing role and user, and an **Initialize** method that will add an *admin* role and an *Administrator* account.

Methods to setup initial roles and users

```
public async Task<bool> HasRoleAsync(string roleId) {
    var result = await _roleManager.FindByIdAsync(roleId);
    return result != null;
}

public async Task<bool> HasUserAsync(string userName) {
    var result = await _userManager.FindByNameAsync(userName);
    return result != null;
}

public async Task InitializeAsync() {
    if (!await HasRoleAsync("admin")) await AddRoleAsync("admin");
    if (!await HasUserAsync("admin@newrise.com.my")) await AddUserAsync(
        "admin@newrise.com.my", null, "P@ssw0rd", "admin");
}
```

We can register our user account service with the application host builder and use it to create our own custom register, login, logout pages as well as management pages for roles and users.

Register our user account service: Program.cs

```
builder.Services.AddScoped<UserAccountService>();
```

We will add code to our **Index.razor** home page to ensure that the initial admin role and admin user account will be created by calling the **InitializeAsync** method on our user account service. Once the page appears, you should now be able to login using the admin account.

Setup initial admin role and user account: Pages\Index.razor

```
@page "/"
@inject UserAccountService UserAccountService
                    :
@code {
    protected override async Task OnInitializedAsync() {
        try { await UserAccountService.InitializeAsync(); }
        catch { }
    }
}
```

Once the initial role and account has been created, it is not necessary to keep running the above code. Even though it would not cause any problems but you do not want to waste time to check the identity tables everytime the home page is accessed. You can remove or comment out the code.

2.5 AuthorizeAttribute & AuthorizeView

We can now test authorization with ASP.NET Identity. Use **AuthorizeView** in **Events** page to make sure that the add event button is only visible for users in admin role by assigning the **admin** role ID to the **Roles** property. The property can be assigned one or more roles separated by commas.

Authorize Add Event button for admin: Pages\Events.razor

```
<AuthorizeView Roles="admin">
    <MudButton Variant="Variant.Filled" Color="Color.Primary"
        Href="/events/new">ADD EVENT</MudButton>
</AuthorizeView>
```

Hiding or disabling a link button does not prohibit the user from accessing the page. We will need to use the .NET **Authorize** attribute to disable access to the entire page as well.

Authorize Create event page for admin: Pages\events\Create.razor

```
@page "/events/new"
@attribute [Authorize(Roles = "admin")]
@inject EventDataService DataService
```

A problem is with Identity in a Blazor application is login must be a Razor page, not a Razor component since it requires **HttpContext** to send an authentication cookie to the web browser to maintain authentication state throughout the session but Razor components are activated through SignalR, not with HTTP requests. Only the host page is a Razor page and it is fetched only once. This means that you cannot a Blazor UI Framework to construct the login page. You can still use style classes from the UI Framework to create a login page that is consistent with the rest of the application.

3 Custom Authentication

3.1 Authentication State Provider

In Blazor applications, we always use an **AuthenticationStateProvider** to fetch the authentication state. **AuthorizeAttribute** and **AuthorizeView** use it for authorizaton as well. Thus we can implement custom authentication and authorization by simply creating our own authentication state provider. Even though ASP.NET Identity provide a total solution for authentication with individual user accounts stored in a database, you would have to build your entire application around it. If instead of building a new application, you are migrating an existing application to Blazor, you might find it more difficult to replace your current application security with Identity then simply integrate it to Blazor. Identity may also be overly complicated for what your application requires and also the login and logout process has to be implementing by using Razor views or pages since it depends on **HTTPContext** and cookies while Razor components in a Blazor server application are activated through SignalR, not by using HTTP requests so no HTTPContext will be available to code executing in Razor components. Changes to UI is sent back through a SignalR connection, not a HTTP response.

We will now implement our custom **AuthenticationStateProvider**. Create the class as shown below in the **Services** directory. You need to extend your custom class from **AuthenticationStateProvider** and then override a **GetAuthenticationStateAsync** method.

Custom authentication state provider:Services\CustomAuthenticationStateProvider.cs

```
using Microsoft.AspNetCore.Components.Authorization;

namespace Newrise.Services {
    public class CustomAuthenticationStateProvider : AuthenticationStateProvider {
        public override Task<AuthenticationState> GetAuthenticationStateAsync() {
        }
    }
}
```

We will first declare a default **ClaimsPrincipal** to create an unauthenticated state for when a user has not logged-in or has logged-out.

Default authentication state for anonymous user

```
private readonly ClaimsPrincipal _anonymous = new ClaimsPrincipal(new ClaimsIdentity());
```

The default **AuthenticationStateProvider** obtains and maintain authentication state from the **HttpContext.User** property and make it available to all Razor components. The state is stored as a HTTP cookie but cookies require HTTP response to send back to the web client and HTTP request to send them back to the server. Alternatively we can use session storage so we do not need to use cookies.

We will declare a field to assign a **ProtectedSessionStorage** service to be received through the constructor. This service has be added to the application host builder so it can be injected into our authentication state provider.

Get access to ProtectedSessionStorage service

```
private readonly ProtectedSessionStorage _sessionStorage;

public CustomAuthenticationStateProvider(ProtectedSessionStorage sessionStorage) {
    _sessionStorage = sessionStorage;
}
```

Adding ProtectedSessionStorage service: Program.cs

```
builder.Services.AddScoped<ProtectedSessionStorage>();
```

You can now implement a model class for what you want to store into session storage which in this case is at least a User ID and role information. It is up to you to decide how to represent role information depending on whether the user can have a single or multiple roles or a specific role.

Session model for single role

```
namespace Newrise.Models {
    public class UserSession {
        public string UserId { get; set; }
        public string RoleId { get; set; }
    }
}
```

Session model for multiple roles

```
namespace Newrise.Models {
    public class UserSession {
        public string UserId { get; set; }
        public string[] RoleIds { get; set; }
    }
}
```

Session model for a specific role: Models/UserSession.cs

```
namespace Newrise.Models {
    public class UserSession {
        public string UserId { get; set; }
        public bool IsAdmin { get; set; }
    }
}
```

We will now implement the **GetAuthenticationStateAsync** method. We will initially try to fetch **UserSession** object from session storage. Since we are using protected session storage, the object is expected to be encrypted. If the object does not exist or if a runtime error occurs if the object cannot be decrypted due to tampering then we will return an authentication state for the anonymous principal. A principal contains identity and role. A **ClaimsPrincipal** can be constructed from a **ClaimsIdentity**. The identity must have a authentication type or it will be considered as unauthenticated.

Returning the authentication state

```
public override async Task<AuthenticationState> GetAuthenticationStateAsync() {
    try {
        var result = await _sessionStorage.GetAsync<UserSession>("UserSession");
        var userSession = result.Success ? result.Value : null;
        if (userSession == null) return await Task.FromResult(
            new AuthenticationState(_anonymous));
        List<Claim> claims = new List<Claim>();
        claims.Add(new Claim(ClaimTypes.Name, userSession.UserId));
        if (userSession.IsAdmin) claims.Add(new Claim(ClaimTypes.Role, "admin"));
        var principal = new ClaimsPrincipal(new ClaimsIdentity(claims, "CustomAuth"));
        return await Task.FromResult(new AuthenticationState(principal));
    }
    catch {
        return await Task.FromResult(new AuthenticationState(_anonymous));
    }
}
```

We will need to update the authentication state when the user logs in or when they log out. Following method will store or remove user session. UI may need to refresh upon authentication state change, so call the **NotifyAuthenticationStateChanged** to ensure that authorization knows the state has changed.

Updating authentication state

```
public async Task UpdateAuthenticationState(UserSession userSession) {
    if (userSession != null) await _sessionStorage.SetAsync("UserSession", userSession);
    else await _sessionStorage.DeleteAsync("UserSession");
    NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());
}
```

We can now add our **CustomAuthenticationStateProvider** to replace the default **AuthenticationStateProvider**. It will be fetched and used by **AuthorizeAttribute** and **AuthorizeView** instead. Check that **UseAuthentication** and **UseAuthorization** methods are called on the application host.

Add our custom authentication state provider

```
builder.Services.AddScoped<AuthenticationStateProvider,
    CustomAuthenticationStateProvider>();
```

Ensure authentication and authorization is enabled

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

Since we won't be using ASP.NET Identity, you can remove **AddDefaultIdentity** or **AddIdentity** as well as **ConfigureApplicationCookie**. You can also remove identity tables from the database by reverting back to **InitialCreate** migration. Remove also **UserAccountService** as it uses ASP.NET Identity objects.

Removing identity tables

```
Update-Database InitialCreate
Remove-Migration
```

3.2 Users & Roles

We will be using **Participant** as our users table. However the entity model class does not store a password or role information. We will first add a **Password** property and an **IsAdmin** property.

Add password and admin role support: Models\Participant.cs

```
public class Participant {
                    :
        [Required(ErrorMessage = "{0} is required.")]
        [StringLength(256, ErrorMessage = "{0} can only have {1} characters.")]
        public string Password { get; set; }
        public bool IsAdmin { get; set; }
                    :
}
```

Add migration and update database

```
Add-Migration AddPassword
Update-Database
```

3.3 Sign-In Manager

We will implement a single **ParticipantDataService** class to not only to access and manage the **Participants** table but also provide sign-in and sign-out functionality. Add the following class into **Services** folder and also register the service with access to the **DbContextFactory** and also **AuthenticationStateProvider**.

Data service class for Participants: Services\ParticipantDataService

```
public class ParticipantDataService {
    readonly IDbContextFactory<NewriseDbContext> _dcFactory;
    readonly CustomAuthenticationStateProvider _authenticationStateProvider;

    public ParticipantDataService(
        IDbContextFactory<NewriseDbContext> dcFactory,
        AuthenticationStateProvider  authenticationStateProvider) {
            _dcFactory = dcFactory;
            _authenticationStateProvider =
                (CustomAuthenticationStateProvider)
                authenticationStateProvider;
        }
}
```

Method to hash a password

```
const string PASSWORD_SALT = "$_{0}@newrise.921";

public string HashPassword(string password) {
    var ha = SHA256.Create();
    password = string.Format(PASSWORD_SALT, password);
    var hashedPassword = ha.ComputeHash(Encoding.UTF8.GetBytes(password));
    return Convert.ToBase64String(hashedPassword);
}
```

Passwords should never be stored in clear text. You can use any hashing algorithm to hash the password including MD5, SHA1 or SHA256. To ensure a stronger and more unique password, you can combine the original password with an additional password salt. Since hashing generate binary data but we intend to store into a text column you can convert the hashed password into a Base64-encoded string.

We will add a method to create an initial administrative account if it does not already exist. The account will be stored into the **Participants** table and **IsAdmin** set to true to indicate *admin* role.

Method to setup an initial admin account

```
public async Task InitializeAsync() {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        if (await dc.Participants.FindAsync("admin") == null) {
            var user = new Participant {
            Id = "admin",
            Name = "Administrator",
            Email = "symbolicon@live.com",
            Company = "Newrise Learning",
            Position = "Web Administrator",
            Password = HashPassword("P@ssw0rd"),
            IsAdmin = true
        };
        await dc.Participants.AddAsync(user);
        await dc.SaveChangesAsync();
    }
}
```

We will also provide methods to sign-in and sign-out. All that is required is to lookup the user with the correct id and password then then generate **UserSession** to update the **AuthenticationStateProvider**.

Sign-in method

```
public async Task SignInAsync(string userId, string password) {
    var hashedPassword = HashPassword(password);
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var user = dc.Participants.FirstOrDefault(p => (
            p.Id == userId || p.Email == userId) && p.Password == hashedPassword);
        if (user == null) throw new Exception("Invalid user Id or password.");
        var userSession = new UserSession { UserId = user.Id, IsAdmin = user.IsAdmin };
        await _authenticationStateProvider.UpdateAuthenticationState(userSession);
    }
}
```

Sign-out method

```
public async Task SignOutAsync() {
    await _authenticationStateProvider.UpdateAuthenticationState(null);
}
```

Register service: Program.cs

```
builder.Services.AddScoped<ParticipantDataService>();
```

3.4 Custom Login Page

Create a folder under **Pages** named **user** to put in user-related component pages and then add a new Razor component page named **Login.razor**. Configure the route and get access to the **ParticipantDataService** and **NavigationManager**. In the main UI there will be two sections; an alert while we are logging in and the form the user uses to login. The code section contains all the necessary properties to bind to facilitate the login operation. **LoginAsync** method will use the data service to sign-in. If successful it will use **NavigationManager** to redirect back to the home page. If not the error is displayed in the UI and user can attempt to login again. We also call **InitializeAsync** method to make sure the administrator account is created.

Custom Login page: Pages/user/Login.razor

```
@page "/user/login"
@inject ParticipantDataService ParticipantDataService
@inject NavigationManager Navigation

@if (LoggingIn) {
    <MudAlert Severity="Severity.Info">
        Attempting to login. Please wait...
    </MudAlert>
}
else {
    <MudForm>
        <MudGrid>
            <MudItem xs="12">
            </MudItem>
        </MudGrid>
    </MudForm>
}

@code {
    string UserId { get; set; } = string.Empty;
    string Password { get; set; } = string.Empty;
    string Error { get; set; } = string.Empty;
    bool LoggingIn { get; set; } = false;

    async Task LoginAsync() {
        try {
            LoggingIn = true;
            Error = string.Empty;
            await ParticipantDataService.SignInAsync(UserId, Password);
            Navigation.NavigateTo("/");
        }
        catch (Exception ex) {
            Error = $"Login failed. {ex.Message}";
        }
        finally {
            LoggingIn = false;
        }
    }

    protected override async Task OnInitializedAsync() {
        await ParticipantDataService.InitializeAsync();
    }
}
```

Form input controls inside MudItem:

```
<MudCard Class="pa-8 ma-2">
    <MudCardHeader>
        <MudText Typo="Typo.h5">LOGIN</MudText>
    </MudCardHeader>
    <MudCardContent>
        <MudTextField T="string" Label="EMAIL"
            InputType="InputType.Email"
            @bind-Value="UserName" /><br />
        <MudStack Row="true">
            <MudTextField T="string" Label="PASSWORD"
                InputType="InputType.Password"
                @bind-Value="Password" />
        </MudStack><br />
        @if (Error != string.Empty) {
            <MudAlert Severity="Severity.Error">@Error</MudAlert>
        }
    </MudCardContent>
    <MudCardActions>
         <MudButton Variant="Variant.Filled"
            Color="Color.Primary" Class="ml-auto"
            OnClick="LoginAsync">LOGIN</MudButton>
    </MudCardActions>
</MudCard>
```

Update the Login link to point to the above page: Shared\LoginDisplay.razor

```
<MudLink Color=@Color.Inherit Href="/user/login">Login</MudLink>
```

You should now be able to login using the administrator account and authorization will work as before. To logout, we will create a user profile page from where the user can perform a logout operation.

3.5 User Profile

First we need to update our data service to add a method to retrieve a **Participant**. It will match against the participant's **Id** or **Email**. Null will be return if the participant is not found. Then add another method to return the current participant.

Retrieving a specific participant

```
public async Task<Participant> GetParticipant(string id) {
    using (var dc = await _dcFactory.CreateDbContextAsync())
    return dc.Participants.FirstOrDefault(p => p.Id == id || p.Email == id);
}
```

Retrieving the current participant

```
public async Task<Participant> GetCurrentParticipant() {
    var state = await _authenticationStateProvider.GetAuthenticationStateAsync();
    if (state != null) return await GetParticipant(state.User.Identity.Name);
    return null;
}
```

The profile page is only accessible after the user is authenticated. It will use the data service to fetch the current participant. Since this operation is asynchronous we need to make sure that the participant details is only rendered when the item is fetched. A logout button is available to sign-out the user and navigate back to home page.

User profile page: Pages/user/Profile.razor

```
@page "/user/profile"
@attribute [Authorize]
@inject ParticipantDataService ParticipantDataService
@inject NavigationManager Navigation

@if (Item != null)
{
    <MudCard>
        <MudCardHeader>
            <CardHeaderAvatar>
                <MudAvatar Color="Color.Secondary">@Item.Id.Substring(0,1)</MudAvatar>
            </CardHeaderAvatar>
            <CardHeaderContent>
                <MudText Typo="Typo.body1">@Item.Id</MudText>
                <MudText Typo="Typo.body2">@Item.Name</MudText>
            </CardHeaderContent>
        </MudCardHeader>
        <MudCardContent>
            <MudTextField Label="COMPANY" ReadOnly=true Value=@Item.Company />
            <MudTextField Label="POSITION" ReadOnly=true Value=@Item.Position />
            <MudTextField Label="EMAIL" ReadOnly=true Value=@Item.Email />
        </MudCardContent>
        <MudCardActions>
            <MudButton Color=@Color.Primary OnClick=@LogoutAsync>LOGOUT</MudButton>
        </MudCardActions>
    </MudCard>
}

@code {
    Participant Item { get; set; }

    protected override async Task OnInitializedAsync() {
        Item = await ParticipantDataService.GetCurrentParticipant();
    }

    async void LogoutAsync() {
        await ParticipantDataService.SignOutAsync();
        Navigation.NavigateTo("/");
    }
}
```

You can now update the link in **LoginDisplay** component to point to the profile page instead. The only thing we do not have yet is the ability for the user to register for a new account. Once you implemented this feature, you can remove ASP.NET Identity totally from your application.

Update link to profile page: Shared\LoginDisplay.razor

```
<MudLink Color=@Color.Inherit Href="/user/profile">@context.User.Identity?.Name</MudLink>
```

3.6 Exercise

1. The **ParticipantDataService** class does not currently have a method to add a new participant. Create a method to add a new participant using the following signature. You should check to see if the **Id** and **Email** already exist before adding and throw an exception.

   Method signature

   ```
   public async Task AddParticipantAsync(Participant item) {
               :
   }
   ```

2. Create a **Register.razor** component page to allow the user to register as a new participant. Use **EditForm** with **Participant** as the data model and enable data annotation validation. Update the link in **LoginDisplay.razor** so you can view the component page as you are building it. You can use **events/Create.razor** and **user/Login.razor** as guides to create an input form.

3. Remove ASP.NET Identity completely from the application. Delete **Areas** folder and uninstall ASP.NET Identity related packages. Check the application to make sure it functions as normal.

4 Registering New Account

4.1 Participant Data Service

The following is the asynchronous method to add a new **Participant** into *Participants* table in the database.

Method to add a new participant: Services\ParticipantDataService.cs

```
public async Task AddParticipantAsync(Participant participant) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        if (await dc.Participants.FirstOrDefaultAsync(
            p => p.Id == participant.Id) != null)
            throw new Exception("User ID already taken. Use another ID.");
        if (await dc.Participants.FirstOrDefaultAsync(
            p => p.Email == participant.Email) != null)
            throw new Exception("Email already registered. Use another email.");
        participant.Password = HashPassword(participant.Password);
        dc.Participants.Add(participant);
        dc.SaveChanges();
    }
}
```

As usual we begin with using the **DbContextFactory** to instantiate a **DbContext** for our Newrise application to get access to the database. We will use a **using** block for the **DbContext** to make sure it is automatically disposed when the block is exited, no matter if the operation is successful or an exception has occurred.

Ensuring DbContext will be disposed

```
using (var dc = await _dcFactory.CreateDbContextAsync()) {
                :
}
```

We need to check whether a **Participant** with the same **Id** or same **Email** is already added to the *Participants* table before adding the new participant. If this is not done, the operation will still fail but the user will only see a primary key violation error and will not understand what went wrong. By checking it ourselves we can produce better error messages.

Checking for duplicate Participant Id

```
if (await dc.Participants.FirstOrDefaultAsync(p => p.Id == participant.Id) != null)
    throw new Exception("User ID already taken. Use another ID.");
```

Checking for duplicate Participant Email

```
if (await dc.Participants.FirstOrDefaultAsync(p => p.Email == participant.Email) != null)
    throw new Exception("Email already registered. Use another email.");
```

For security we only store hashed passwords in a database. Even if someone manages to see the hashed passwords, they do not know the real passwords and would not be able to login and impersonate the users.

Hash the password

```
participant.Password = HashPassword(participant.Password);
```

Finally we will use the standard **Entity Framework** way of adding a entity to a table and updating the database. Since the input has been validated and we ensure that no primary key or unique columns are violated, this should work if there are no network or server issues.

Adding participant and update database

```
dc.Participants.Add(participant);
dc.SaveChanges();
```

4.2 Register Page

We will now implement the UI for registering a new account. Add a Razor component named **Register.razor** to the **Pages\user** directory with the following basic content to set the route and get access to our data service and navigation manager.

Register page: Pages\user\Register.razor

```
@page "/user/register"
@inject ParticipantDataService ParticipantDataService
@inject NavigationManager Navigation
```

We can now update **LoginDisplay.razor** to change the **Register** link that previously navigates to ASP.NET Identity UI to our custom registration page. This would allow us to easily access the page as we build it.

Update link to register page: Shared\LoginDisplay.razor

```
<AuthorizeView>
        :
    <NotAuthorized>
        <MudLink Color=@Color.Inherit Href="/user/register">Register</MudLink>
                :
    </NotAuthorized>
</AuthorizeView>
```

First determine the different states of the page. When the page changes state, it may show different content. The initial state of the page will show a input form. However, we would not want the user to see or continue to use the form when the form is being submitted for processing. Since we are performing asynchronous operations, you do not want the user to change the data in the form or click on the submit button again which a submit is already in progress. If there are many exclusive states, you can use an **enum** to define the states so you can use one field or property to maintain a page state. If the page can be in only two states or states are non-exclusive, you can use a **bool** field or property for each state.

In our page, we will declare a **IsSubmitting** state property. When this state is **true**, the UI should just have an alert telling the user to wait for the submission to complete and the user can only see and use the registration form when the state is **false**.

Declaring a IsSubmitting state property

```
@code {
    bool IsSubmitting { get; set; }
}
```

Control form visibility and access with IsSubmitting

```
if (IsSubmitting) {
    <MudAlert Severity="Severity.Info">
        Attempting to register account. Please wait...
    </MudAlert>
}
else {
    <EditForm Model=@Item OnValidSubmit=@HandleSubmitAsync>
    <DataAnnotationsValidator />
        <MudCard Class="pa-8 ma-2">
            :
        </MudCard>
    </EditForm>
}
```

You can then declare additional properties for input and output purposes or to control components in the form. We will first declare an **Item** property, a **Participant** model object to store the main input data of the form. It is common to force a user to enter a password twice to ensure the password is correct. Since **Participant** can store only one password, we need another field or property for the second password. So declare a **string** property named **ConfirmPassword**. Data operations may succeed or fail so we may need to display success or error messages in the UI. Since we will navigate away from this page if the operation is successful, we will only show error messages so we will add a **Error** string property to contain the error message which will be then displayed somwhere in the UI.

Properties to add for page

```
Participant Item { get; set; } = new();
string ConfirmPassword { get; set; }
string Error { get; set; } = string.Empty;
```

Displaying error message

```
<MudCard Class="pa-8 ma-2">
    <MudCardContent>
        <MudText Typo="Typo.h5">Register Account</MudText><br />
        @if (Error != string.Empty) {
            <MudAlert Severity="Severity.Error">@Error</MudAlert><br /> }
    <MudCardContent>
</MudCard>
```

Sometimes we need direct access or to communicate with a component in the page. You can declare a field or property and then bind the component to it using the **@ref** directive. We will declare an **IdField** property to map to a input field so we can focus on the field from code.

We can then add buttons the user can use to validate and submit the input form or to perform other operations like canceling the form by navigating to another page or to change page state.

Adding buttons to perform operations or navigation

```
<MudCard>
        :
    <MudCardActions>
        <MudButton Color="Color.Primary" Variant="Variant.Filled"
            ButtonType="ButtonType.Submit">SAVE</MudButton>
        <MudButton Class="ml-2" Color="Color.Secondary" Variant="Variant.Filled"
            Href="/">CANCEL</MudButton>
    </MudCardActions>
</MudCard>
```

The following is the method to be called from **EditForm** when the form is submitted and validated. Since the method is asynchronous, note that the user can still access the page while the operation is being performed.

Method to process the form input

```
async Task HandleSubmitAsync(EditContext context) {
    try {
        IsSubmitting = true;
        Error = string.Empty;
        if (Item.Password != ConfirmPassword)
            throw new Exception("Passwords do not match.");
            await ParticipantDataService.AddParticipantAsync(Item);
            Navigation.NavigateTo("/user/login");
        }
        catch (Exception ex) {
            Error = $"{ex.Message} Registration failed.";
        }
        finally {
            IsSubmitting = false;
            await IdField.FocusAsync();
        }
}
```

We initially set **IsSubmitting** to **false** to switch the UI to display an alert to inform the user to wait for the processing to complete. The form will no longer be visible to the user. Then we set **Error** property to erase any previous error message. If we need to perform additional checking and validation with code, we can do so. Here we make sure both of the passwords entered into **Item.Password** and **ConfirmPassword** are the same. We will then pass the input data to our data service for processing. If the operation is successful we will automatically navigate to the login page.

Main operation to perform

```
IsSubmitting = true;
Error = string.Empty;
if (Item.Password != ConfirmPassword)
    throw new Exception("Passwords do not match.");
    await ParticipantDataService.AddParticipantAsync(Item);
    Navigation.NavigateTo("/user/login");
```

We expect a runtime error to occur if the operation fails so we have a **catch** section to fetch the exception to construct an error message. The error message will appear in the UI once the form re-appears.

Construct error message from exception

```
Error = $"{ex.Message} Registration failed.";
```

Note the form is not visible until we set **IsSubmitting** back to **false** so you must not access any components in the form until you do so. The **finally** section will switch the page state back to the form and we can then move the focus back to the component attached to **IdField** property. Since the component is part of the form, do not access the component until the form is visible again.

Switch page state to input and set input focus

```
IsSubmitting = false;
await IdField.FocusAsync();
```

We can now finish up the form by adding input controls and binding them to model or page properties. We will use **@ref** to bind the first input field to **IdField** property so we can move the input focus to this component from code. Even though we assigned the **AutoFocus** property, this works only when the page is initialized and not after the form is submitted. Remember to use **For** property to show validation errors when you are using **DataAnnotationsValidator**.

Input field for Participant Id

```
<MudTextField Label="USER ID *" @ref=IdField @bind-Value=Item.Id
    For=@(()=>Item.Id) MaxLength="40" AutoFocus />
```

Input fields for Participant Name and Email

```
<MudStack Row="true">
    <MudTextField Label="USER NAME *" @bind-Value=Item.Name
        For=@(()=>Item.Name) MaxLength="40" />
    <MudTextField Label="EMAIL *" @bind-Value=Item.Email
        For=@(()=>Item.Email) MaxLength="40" />
</MudStack><br />
```

Input fields for optional Participant Company and Position

```
<MudStack Row="true">
    <MudTextField Label="COMPANY" @bind-Value=Item.Company
        For=@(()=>Item.Company) MaxLength="40" />
    <MudTextField Label="POSITION" @bind-Value=Item.Position
        For=@(()=>Item.Position) MaxLength="40" />
</MudStack><br />
```

Input fields for Participant password and confirmation password

```
<MudStack Row="true">
    <MudTextField Label="PASSWORD" @bind-Value=Item.Password
        InputType=@InputType.Password For=@(()=>Item.Password) MaxLength="8" />
    <MudTextField Label="CONFIRM PASSWORD" @bind-Value=ConfirmPassword
        InputType=@InputType.Password MaxLength="8" />
</MudStack><br />
```

The form is now complete and users can now register new accounts by clicking on the **Register** link from **LoginDisplay** component in the layout to access the page. After adding the account, the user should be able to login to get authenticated. Use profile page to check the account and optionally logout.