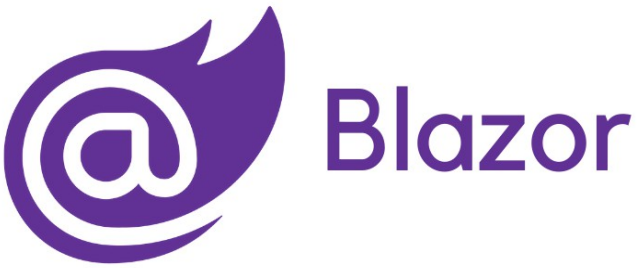

ASP520

Module 4

# Custom Authentication & Authorization

# 1 Authentication State Provider

## 1.1 Roles & Claims

First implement a model class for the information you need to identity the user and to be able to determine what the user is allowed or not allowed to do which we called as authorization. This information should be available once the user goes through a login process. For authorization, you can use the concept of roles and/or policies. A role is a simple indentifier while a policy is a set of rules/checks performed on information that is attached to the user which we call as claims. A role is basically a claim by itself. It is up to you decide how many roles each user can have. In the following class, a user is expected to have a single role.

Session model for single role

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

The following code would be needed to convert the above model into **ClaimsIdentity** which represent a collection of claims that can be used to identify and authorize the user.

Converting above session to ClaimsIdentity

```
List<Claim> claims = new List<Claim>();
claims.Add(new Claim(ClaimTypes.Name, session.UserId));
if (!string.IsNullOrEmpty(session.RoleId))
    claims.Add(new Claim(ClaimTypes.Role, session.RoleId));
var claimsIdentity = new ClaimsIdentity(claims);
```

The following shows an example model where a user is allowed to have multiple roles. We can use an array or collection to contain the roles. In a database usually you need two tables; a **users** table to store the user details and a **roles** table to store the roles for each user. You would need to retrieve data from both tables to provide the model shown below when the user login.

Session model for multiple roles

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

## Converting multiple roles to claims

```
foreach (string roleId in session.RoleIds)
    claims.Add(new Claim(ClaimTypes.Role, roleId));
```

You can also use an **enum** to represent a set of permissions where each permission is represented by one bit. Default enum is 32-bits but you can even have 64-bit enums so technically you can have up to 64 individual permissions per enum. For example, we can implement read and write permissions for **Event** table.

## A bit flag enum

```
[Flags]
public enum EventPermission {
    None = 0,                   //  0b00000000000000000000000000000000 (bitmask)
    Read = 0x01,                //  0b00000000000000000000000000000001
    Write = 0x02,               //  0b00000000000000000000000000000001
    All = Read | Write          //  0b00000000000000000000000000000011
}
```

## Extend from long or unsigned long to get 64-bit enum

```
public enum EventPermission : ulong {
    :
}
```

## You can use binary literals in C#7 (.NET 5 is C# 9)

```
[Flags]
public enum EventPermission {
    None    = 0,
    Read    = 0b1,
    Write   = 0b11
    All     = Read | Write
}
```

## Session model for permission-based roles

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

## Converting permissions into claims

```
if ((session.EventPermissions & EventPermission.Read) == EventPermission.Read)
    claims.Add(new Claim(ClaimType.Role, "event_reader"));
if (session.EventPermissions & EventPermission.Write) == EventPermission.Write)
    claims.Add(new Claim(ClaimType.Role, "event_writer"));
```

In the example above, any user with **event_reader** role can then be authorized to retrieve and view events but you need **event_writer** role to add, update and remove events.

As our application only needs to differentiate between an admin account and a basic user account, we can just use a simple single fixed admin role.

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

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

Converting a single fixed role to claim

```
if (session.IsAdmin) claims.Add(new Claim(ClaimTypes.Role, "admin"));
```

.NET security works with **ClaimsPrincipal** which contains a **ClaimsIdentity** and help provide role checking. A **ClaimsIdentity** contains a collection of claims and the type of authentication used. To make it easier to convert the model to a **ClaimsPrincipal**, we will add the following method and use our custom authentication type.

Add method to help convert model to ClaimsPrincipal

```
public const string AuthenticationType = "NewriseAuthentication";

public ClaimsPrincipal GetPrincipal() {
    var claims = new List<Claim>();
    claims.Add(new Claim(ClaimTypes.Name, UserId));
    if (IsAdmin) claims.Add(new Claim(ClaimTypes.Role, "admin"));
    var identity = new ClaimsIdentity(claims, AuthenticationType);
    return new ClaimsPrincipal(identity);
}
```

## 1.2 AuthenticationStateProvider

Blazor applications use an **AuthenticationStateProvider** to fetch the authentication state. We can implement custom authentication and authorization by simply creating our own authentication state provider.

We used **UserSession** to generate principal from data we accessed from a database during a login process. To remember that the user has already logged-in, we need to store this data and remove it when they logout. To remember only for current session we can use session storage and to remember across sessions, use local storage. Since this information is stored on the web browser we should encrypt it if we do not want users to be able to see what is in **UserSession**. Blazor provides encrypted browser storage with the **ProtectedSessionStorage** and **ProtectedLocalStorage** services. If you are using local storage, your **UserSession** should contain an expiry date as it is not safe to keep it forever. Once it is expired, it should no longer be used. The user is forced to login again. For our application we will use session storage which is more secure. When users close the web browser, they are automatically logged-out. We get access to the service using constructor injection.

Adding ProtectedSessionStorage service: Program.cs

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

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 protected session storage: ServerAuthenticationStateProvider.cs

```
private ProtectedSessionStorage _storage;

public ServerAuthenticationStateProvider(ProtectedSessionStorage storage) {
    _storage = storage;
}
```

We will first declare a default **ClaimsPrincipal** to create an unauthenticated state for when a user has not logged-in or has logged-out. We then add methods to return and update the authentication state.

Default authentication state for anonymous user

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

Method to return authentication state

```
public override async Task<AuthenticationState> GetAuthenticationStateAsync() {
    if (_state != null) return _state;
    try {
        var result = await _storage.GetAsync<UserSession>("UserSession");
        if (result.Success) return _state = new AuthenticationState(
            result.Value.GetPrincipal());
        return _anonymous; } catch {
        return _anonymous;
    }
}
```

Updating authentication state

```
public async Task UpdateAuthenticationStateAsync(UserSession userSession) {
    if (userSession == null) {
        await _storage.DeleteAsync("UserSession");
        NotifyAuthenticationStateChanged(Task.FromResult(_anonymous));
    }
    else {
        await _storage.SetAsync("UserSession", userSession);
        _state = new AuthenticationState(userSession.GetPrincipal());
        NotifyAuthenticationStateChanged(Task.FromResult(_state));
    }
}
```

We now need to register our custom authentication state provider that will override the original authentication state provider. Call **AddAuthentication** method to enable authentication services. Also call **AddCascadingAuthenticationState** to allow the authentication state to be passed down the entire application so all components have access to it.

Register authentication state provider

```
services.AddScoped<AuthenticationStateProvider, ServerAuthenticationStateProvider>();
services.AddAuthentication(UserSession.AuthenticationType);
services.AddCascadingAuthenticationState();
```

Enable authentication and authorization middleware by calling **UseAuthentication** and **UseAuthorization** methods. Always ensure authorization is enabled only after route mapping. Add all the following namespaces to access authorization services and components in the application.

Enable authentication and authorization middleware

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
app.UseAuthentication();
app.UseAuthorization();
```

Import authorization services and components namespaces

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

## 1.3 Authorization

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: Routes.razor

```
<Router AppAssembly="typeof(App).Assembly" Context="routeData">
    <Found>
        <AuthorizeRouteView RouteData="routeData" DefaultLayout="typeof(DefaultLayout)">
            <NotAuthorized>
                <MudAlert Severity="Severity.Error">
                    Access denied. Sorry, you are not authorized to access this page.
                </MudAlert>
            </NotAuthorized>
        </AuthorizeRouteView>
    </Found>
</Router>
```

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. For example, we can display the user name in the **AppBar** if user has been authenticated. You can provide different content for **NotAuthorized**, **Authorized** and **Authorizing**.

Using AuthorizeView component

```
<AuthorizeView>
    <MudText>@context.User.Identity.Name</MudText>
</AuthorizeView>
```

Provide alternative content

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

Allow access to only admin role: Pages/Event/Create.razor

```
@page "/events/create"
@attribute [Authorize(Roles = "admin")]
```

Rather than just restricting access to an 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 button only to admin role

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

# 2 Implementing Authentication

## 2.1 Users & Roles Management

Depending how we handle rows, we many need to add one or more extra columns to the **Participant** table or even an entire new **Roles** table. For our application, a user is either a normal user or an administrator so an extra **IsAdmin** column is enough.

Add admin role support: Models\Participant.cs

```
public class Participant {
                :
    public bool IsAdmin { get; set; }
                :
```

Update database

```
Add-Migration AddAdminRole
Update-Database
```

You now have to provide services to manage users and roles. If you support multiple roles, you may need to be able to add and remove roles from a **Roles** table. You then have to be able to add a user to a role or remove a user from a role. We do not need this because we just need to support a single admin role.

For users, you need to provide ability to hashed passwords, add users, remove users, authenticate to sign-in the user and ability to sign-out the user. We will implement a single **ParticipantDataService** class to access and manage the **Participants** table and also provide sign-in and sign-out functionality. This service require access to the database as well as the **AuthenticationStateProvider**. We want to use our custom one but note it was registered as a replacement for the original one so we will access it using the original class but cast to **ServerAuthenticationStateProvider**.

Data service class for Participants: Services\ParticipantDataService

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

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. A hashing algorithm determines the size of the hash and Base64 requires 4 characters to encode 3 bytes so you can calculate the size of the hash.

Method to hash a password

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

public static 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);
}
```

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",
            PasswordHash = 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.PasswordHash == hashedPassword);
        if (user == null) throw new Exception("Invalid user Id or password.");
        var userSession = new UserSession { UserId = user.Id, IsAdmin = user.IsAdmin };
        await _authenticationStateProvider.UpdateAuthenticationStateAsync(userSession);
    }
}
```

Sign-out method

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

Note that you need to register the service using **AddScoped** since each user has to be assigned a separate authentication state provider because the authentication state is kept per object.

Register service: Program.cs

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

## 2.2 Custom Login Page

Create a **User** folder under **Pages** put in user-related component pages and then add a new component 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** in development environmentto make sure the administrator account is created.

Custom Login page: Pages/User/Login.razor

```
@page "/login"
@inject IWebHostEnvironment Environment
@inject ParticipantDataService ParticipantDataService
@inject NavigationManager NavigationManager

@code {
    LoginInfo Model { get; set; } = new();
    string Error { get; set; } = string.Empty;
    bool LoggingIn { get; set; } = false;

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

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

The Login page UI

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

Form input controls inside MudItem

```
<MudCard Class="pa-8 ma-2">
    <MudCardHeader>
        <MudText Typo=Typo.h5>LOGIN</MudText>
    </MudCardHeader>
    <MudCardContent>
        <MudTextField T="string" Label="USER ID/EMAIL"
            InputType="InputType.Email"
            @bind-Value=Model.UserID /><br />
        <MudStack Row="true">
            <MudTextField T="string" Label="PASSWORD"
                InputType="InputType.Password"
                @bind-Value=Model.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>
```

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.

## 2.3 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> GetParticipantAsync(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> GetCurrentParticipantAsync() {
    var state = await _authenticationStateProvider.GetAuthenticationStateAsync();
    if (state != null) return await GetParticipantAsync(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 "/profile"
@attribute [Authorize]
@inject ParticipantDataService ParticipantDataService
@inject NavigationManager NavigationManager

@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.GetCurrentParticipantAsync();
    }

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

You can now update user name in the layout component to link 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.

Update link to profile page

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

## 2.4 User Registration

The following is the asynchronous method to add a new **Participant**. We will 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 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. 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.

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

```
public async Task AddParticipantAsync(NewParticipant 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.PasswordHash = HashPassword(participant.Password);
        dc.Participants.Add(participant);
        dc.SaveChanges();
    }
}
```

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 "/register"
@inject ParticipantDataService ParticipantDataService
@inject NavigationManager NavigationManager
```

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

```
NewParticipant 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);
            NavigationManager.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);
    NavigationManager.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 />
```

## 2.5 Event Participation

We want to add more features to the **EventDetails** component. We want a user to be able to enter and remove their participation in an event. We also need to check if the user is already participating in an event. You can go about this in two ways; locate the event and access the participants for that event or locate the participant instead and access the events for that participant. In Entity Framework use **Include** method when you wish to include related entities to a particular entity you are retrieving. The following shows how to retrieve an **Event** together with related **Participants**. We can then check for a particular participant of the event.

Check an event for a particular participant: Services\EventDataService.cs

```
public async Task<bool> HasParticipantAsync(string eventId, string participantId) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var eventItem = await dc.Events.Include(e => e.Participants).SingleOrDefaultAsync(
            e => e.Id == eventId);
        if (eventItem != null) return eventItem.Participants.SingleOrDefault(
            p => p.Id == participantId) != null;
        return false;
    }
}
```

Adding and removing a participant from an event can be quite an involved task. This is where creating stored procedures may be a better option instead of coding in C#. A database transaction is required since we need to ensure that no one else can update or access **AllocatedSeats** column in **Events** table until the end of the transaction by using the **RepeatableRead** isolation level.

Adding participant to event

```
public async Task AddParticipantAsync(string eventId, string participantId) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var transaction = await dc.Database.BeginTransactionAsync(
            IsolationLevel.RepeatableRead);
        try {
            var eventItem = await dc.Events.Include(e => e.Participants)
                .SingleOrDefaultAsync(e => e.Id == eventId);
            if (eventItem == null) throw new Exception($"Event '{eventId}' does not exist.");
            if (eventItem.Participants.SingleOrDefault(p => p.Id == participantId) != null)
                throw new Exception($"User '{participantId}' is already a participant of event
'{eventId}.");

            if (eventItem.RemainingSeats == 0)
                throw new Exception($"Event '{eventId}' is full.");
            var participant = await dc.Participants.FindAsync(participantId);
            if (participant == null) throw new Exception(
                $"Participant '{participantId}' does not exist.");
            eventItem.Participants.Add(participant);
            eventItem.AllocatedSeats++;
            await dc.SaveChangesAsync();
            await transaction.CommitAsync();
        }
        catch (Exception) { await transaction.RollbackAsync(); throw; }
    }
}
```

## Removing participant from event

```
public async Task RemoveParticipantAsync(string eventId, string participantId) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var transaction = await dc.Database.BeginTransactionAsync(
            IsolationLevel.RepeatableRead);
        try {
            var eventItem = await dc.Events.Include(e => e.Participants)
                .SingleOrDefaultAsync(e => e.Id == eventId);
            if (eventItem == null) throw new Exception($"Event '{eventId}' does not exist.");
            var participant = eventItem.Participants.SingleOrDefault(
                p => p.Id == participantId);
            if (participant == null) throw new Exception(
                $"User '{participantId}' is not a participant of event '{eventId}.");
            eventItem.Participants.Remove(participant);
            eventItem.AllocatedSeats--;
            await dc.SaveChangesAsync();
            await transaction.CommitAsync();
        }
        catch (Exception) {
            await transaction.RollbackAsync();
            throw;
        }
    }
}
```

In **EventDetails** we need to access the **ClaimsPrincipal** from authentication state as we need to get the user Id to map the current user to a participant. From this we will find out if the user is participating in the current event and set **IsParticipating** state to reflect this. We will cache the principal in **User** property even though we only need the user Id.

## Getting user information and event participation

```
@using System.Security.Claims;
@inject EventDataService EventDataService
@inject AuthenticationStateProvider AuthenticationStateProvider
@inject ISnackbar Snackbar
            :
@code {
            :
    ClaimsPrincipal User { get; set; }
    bool IsParticipating { get; set; }
    [Parameter] public Action<Event> EventUpdated { get; set; }

    protected override async Task OnParametersSetAsync() {
        User = (await AuthenticationStateProvider.GetAuthenticationStateAsync()).User;
        if (User.Identity.IsAuthenticated) {
            IsParticipating = await EventDataService.HasParticipantAsync(
                Event.Id, User.Identity.Name);
        }
    }
}
```

We will now add two methods; **ParticipateEventAsync** and **LeaveEventAsync** that will add and remove the current user as the participant in the current event based on the event Id retrieve from **Event** object and user Id retrieved from **User** object.

## Method to add user as participant

```
async Task ParticipateEventAsync() {
    try {
        IsProcessing = true;
        await EventDataService.AddParticipantAsync(Event.Id, User.Identity.Name);
        Snackbar.Add($"Participation in event '{Event.Id}' confirmed.", Severity.Success);
        IsParticipating = true; EventUpdated?.Invoke(Event);
    }
    catch (Exception ex) { Snackbar.Add(ex.Message, Severity.Error); }
    finally { MudDialog.Close(); }
}
```

## Method to remove user as participant

```
async Task LeaveEventAsync() {
    try {
        IsProcessing = true;
        await EventDataService.RemoveParticipantAsync(Event.Id, User.Identity.Name);
        Snackbar.Add($"Your '{Event.Id}' event participation is cancelled.", Severity.Success);
        IsParticipating = false; EventUpdated?.Invoke(Event);
    }
    catch (Exception ex) { Snackbar.Add(ex.Message, Severity.Error); }
    finally { MudDialog.Close(); }
}
```

We will now add in the additional buttons that will be available as long as the user is authenticated. Which button appears depends on the **IsParticipating** state. We will also add a tag to highlight event participation with an extra **MudChip**.

## Add event participation buttons

```
@if (IsProcessing) {
    <MudProgressCircular Class="mx-auto" Color=@Color.Primary Indeterminate=@true /> }
else {
    <AuthorizeView>
        @if (IsParticipating) {
            <MudButton Color=@Color.Primary
               Variant=@Variant.Filled OnClick=@LeaveEventAsync>
                LEAVE EVENT
            </MudButton>
        }
        else if (Event.RemainingSeats > 0) {
            <MudButton Color=@Color.Primary
               Variant=@Variant.Filled OnClick=@ParticipateEventAsync>
                PARTICIPATE EVENT
            </MudButton>
        }
    </AuthorizeView>
            :
```

## Add extra tags for participation info

```
<MudStack Row=@true>
    <MudSpacer />
    @if (Event.Online) { <MudChip Color=@Color.Info>ONLINE</MudChip>   }
    @if (Event.RemainingSeats == 0) { <MudChip Color=@Color.Info>FULL</MudChip> }
    @if (IsParticipating) { <MudChip Color=@Color.Secondary>PARTICIPATING</MudChip> }
</MudStack>
```

Currently we close the dialog everytime we complete the operation so we do not need to reset or refresh the UI state. If we choose to maintain the dialog we would need to make sure the UI state is refreshed property. In **EventDetails** dialog, we will need to make sure **IsParticipating** is updated, **IsProcessing** is reset and we need to fetch a new updated **Event**. Add the method to retrieve a specific **Event**.

Method to retrieve a specific event: Services\EventDataService.cs

```
public async Task<Event> GetEventAsync(string id) {
    using (var dc = _dcFactory.CreateDbContext()) {
        return await dc.Events.FindAsync(id);
    }
}
```

Update the finalization in the following two methods to fetch the updated **Event** and reset **IsProcessing** state. The UI will detect the changes and refresh automatically. If for some reason, some UI components shows prior state, call the **StateHasChanged** method to force a UI refresh.

Refresh dialog content

```
async Task ParticipateEventAsync() {
                :
    finally {
        Event = await EventDataService.GetEventAsync(Event.Id);
        IsProcessing = false;
    }
}

async Task LeaveEventAsync() {
                :
    finally {
        Event = await EventDataService.GetEventAsync(Event.Id);
        IsProcessing = false;
    }
}
```

Return back to **Events** page to set the extra **EventUpdated** parameter for the dialog and since both parameters run the same code, rename the **OnEventDeletedAsync** method to **OnEventUpdatedAsync** instead so we can use one delegate.

Rename event callback method: Pages\Events.razor

```
async void OnEventUpdatedAsync(EventModel item) {
    Items = await EventDataService.GetEventsAsync();
    StateHasChanged();
}
```

Using same delegate for multiple parameters

```
void ShowDetails(EventModel item) {
    var parameters = new DialogParameters();
    parameters.Add("Event", item);
    var action = new Action<EventModel>(OnEventUpdatedAsync);
    parameters.Add("EventDeleted", action);
    parameters.Add("EventUpdated", action);
    DialogService.Show<EventDetails>(null, parameters);
}
```