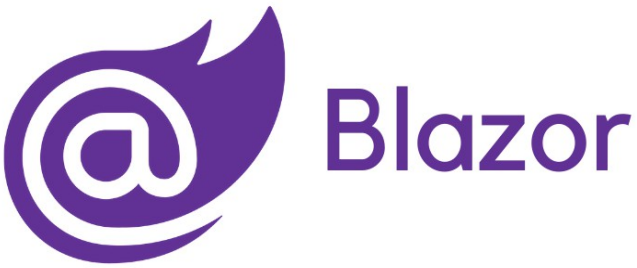

ASP520

Module 7

Implementing API Client & Blazor Web Assembly Application

1 API Client

1.1 Blazor WebAssembly

WebAssembly is a web browser technology that allows running programs on a web browser that is not written using Javascript. You can use any programming language as long as there is a compiler than can compile the program to webassembly code. Since .NET code can be transformed into webassembly code, you can now write web browser applications using C#.

Blazor is a web application framework to build a web application that runs on a web server called as a **Blazor Server** application or to run on a web browser that is called a **Blazor Webassembly** application. A Blazor application is constructed from a set of Razor components. A Razor component can be a standalone page or a component or sub-component of a page. Even though Razor component pages are commonly placed in a **Pages** folder, the actual location and name does not matter as routing is used to activate the correct page. Just like any ASP.NET web server application project you start by creating an application host to host your Blazor web assembly application but call WebAssemblyHostBuilder

Blazor web assembly startup code: Program.cs

```
public class Program {
    static async Task Main(string[] args) {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        await builder.Build().RunAsync();
    }
}
```

1.2 API Client

A Web API is to expose services that runs on the web server. To make it possible and also simple for a web browser application to use server-side services, you can create an API client to communicate with a Web API through **HTTP**. Register a **HTTPClient** service that you can use to send **HTTP** requests and process **HTTP** responses. Add a **Services** folder with two classes, **EventDataClient** and **ParticipantDataClient**. Use constructor injection is to get access to **HttpClient** service.

Register HTTPClient service: Program.cs

```
var services = builder.Services;
services.AddScoped(sp => new HttpClient {
    BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
});
```

Event API client: Services\EventDataClient.cs

```
public class EventDataClient {
    private const string _route = "api/events";
    private readonly HttpClient _client;
    private readonly JsonSerializerOptions _json_options;
    public EventDataClient(HttpClient client) {
        _client = client; _json_options = new JsonSerializerOptions {
            PropertyNameCaseInsensitive = true
        };
    }
}
```

Participant API client: Services\ParticipantDataClient.cs

```
public class ParticipantDataClient {
    private const string _route = "api/participants";
    private readonly HttpClient _client;
    private readonly JsonSerializerOptions _json_options;
    public ParticipantDataClient(HttpClient client) {
        _client = client; _json_options = new JsonSerializerOptions {
            PropertyNameCaseInsensitive = true
        };
    }
}
```

If you examine all the requests and responses with Swagger, notice that the names of the data model properties passed or return in JSON format does not have the same case as the original data model class. This is why a **JsonSerializerOptions** object is created in the above classes. We need this to inform the JSON deserializer to ignore the case or otherwise the JSON fields will not map correctly to data model properties and all the properties will be null or zero after deserialization.

Our next step is to implement methods in the above classes to construct and sent the HTTP requests required to trigger the action methods implemented in the Web APIs as shown below. We need to know the request method, the route, parameters and the request body in order to generate a correct request.

It would be easier to convert a Blazor Server to a Blazor Webassembly application if there are no code changes. One issue is that server components access data services directly while client components access data services through a Web API client. For a component to be able to use both, you could use an interface to register and inject a service. The interfaces can be created in **Newrise.Shared** so both server and client have access to them.

Event API client interface: Services\ParticipantDataClient.cs

```
public interface IEventDataService {
    Task<Event> GetEventAsync(string id);
    Task<Event> AddEventAsync(Event item);
    Task<Event> RemoveEventAsync(string id);
    Task<Event> UpdateEventAsync(Event item);
    Task<List<Event>> GetEventsAsync();
    Task<bool> HasParticipantAsync(string eventId, string participantId);
    Task<bool> AddParticipantAsync(string eventId, string participantId);
    Task<bool> RemoveParticipantAsync(string eventId, string participantId);
}
```

Participant API client interface: Services\ParticipantDataClient.cs

```
public interface IParticipantDataService {
    Task<UserSession> SignInAsync(string userId, string password);
    Task<Participant> AddParticipantAsync(NewParticipant participant);
    Task<bool> UpdatePhotoAsync(string id, byte[] photo);
    Task<Participant> GetParticipantAsync(string id);
    Task<List<Event>> GetParticipantEventsAsync(string id);
    Task<Participant> GetCurrentParticipantAsync();
    Task<List<Event>> GetCurrentParticipantEventsAsync();
    Task SignOutAsync();
}
```

Make Event client compatible to server

```
public class EventDataClient : IEventDataService {
            :
```

Make Participant client compatible to server

```
public class ParticipantDataClient : IParticipantDataService {
            :
```

For Blazor server, we would register **EventDataService** class but for Webassembly we register **EventDataClient**. As long as both classes implement the same interface, there is no code changes required for components using the service to switch to using the client.

Registering data services on server

```
services.AddScoped<IEventDataService, EventDataService>()>;
services.AddScoped<IParticipantDataService, ParticipantDataService()>;
```

Registering data services on client

```
services.AddScoped<IEventDataService, (EventDataClient()>;
services.AddScoped<IParticipantDataService, ParticipantDataClient>()>;
```

You can use the **HTTPClient** to send requests to the Web API. You can sent different type of requests including GET, PUT, POST and DELETE. When you receive a response, check the status code to make sure that the Web API has received and processed the request which will be 200 (OK). 404 means not found so your route may be incorrect. 401 means unauthorized. You can then obtain the result by reading and deserializing the response **Content**.

Method to retrieve a single event: EventDataClient.cs

```
public async Task<Event> GetEventAsync(string id) {
    var response = await _client.GetAsync($"{_route}/{id}");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<Event>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

To send an object, it has to be serialized to JSON and placed into the body section of a HTTP request. This is done using **PutAsJsonAsync** and **PostAsJsonAsync**. Even if some HTTP methods have a body section does not mean you need to sent something in the body, it can be an empty. Sometimes we just want to use that HTTP method to trigger a specific action method.

Method to add a new event

```
public async Task<Event> AddEventAsync(Event item) {
    var response = await _client.PutAsJsonAsync(_route, item);
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to add an event.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<Event>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to update an event

```
public async Task<Event> UpdateEventAsync(Event item) {
    var response = await _client.PostAsJsonAsync(_route, item);
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to add an event.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<Event>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
        return result.Value;
}
```

Method to remove an event

```
public async Task<Event> RemoveEventAsync(string id) {
    var response = await _client.DeleteAsync($"{_route}/{id}");
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to remove an event.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<Event>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to retrieve a list of event

```
public async Task<List<Event>> GetEventsAsync() {
    var response = await _client.GetAsync(_route);
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<List<Event>>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to check event participation

```
public async Task<bool> HasParticipantAync(string eventId, string participantId) {
    var response = await _client.GetAsync($"{_route}/{eventId}/participants/{participantId}");
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to check event participant.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<bool>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to add participant to an event

```
public async Task<bool> AddParticipantAsync(string eventId, string participantId) {
    var response = await _client.PutAsJsonAsync(
        $"{_route}/{eventId}/participants/{participantId}", string.Empty);
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to add event participant.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<bool>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to remove participant from an event

```
public async Task<bool> RemoveParticipantAsync(string eventId, string participantId) {
    var response = await _client.DeleteAsync(
        $"{_route}/{eventId}/participants/{participantId}");
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to remove event participant.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<bool>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to retrieve a single participant: ParticipantDataClient.cs

```
public async Task<List<Event>> GetParticipantEventsAsync(string id) {
    var response = await _client.GetAsync($"{_route}/{id}/events");
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to access participant events.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<List<Event>>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to add a participant

```
public async Task<Participant> AddParticipantAsync(NewParticipant item) {
    var response = await _client.PutAsJsonAsync(_route, item);
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<Participant>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to retrieve events for a participant

```
public async Task<Participant> GetParticipantAsync(string id) {
    var response = await _client.GetAsync($"{_route}/{id}");
    if (response.StatusCode == HttpStatusCode.Unauthorized)
        throw new Exception("You are not allowed to access the participant.");
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<Participant>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    return result.Value;
}
```

Method to upload photo for a participant

```
public async Task<bool> UpdatePhotoAsync(string id, byte[] photo) {
    var response = await _client.PostAsJsonAsync($"{_route}/{id}", photo);
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Network or server error ({response.StatusCode})");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<bool>>(content, _json_options);
        if (!result.Success) throw new Exception(result.Error);
        return result.Value;
}
```

Methods that require authorization will not work if the request does not have a JWT bearer token attached in the HTTP header. We still need to implement a login method to obtain the token and we also need to integrate client-side authorization to control which part of the application can or cannot access.

Participant methods that work with authentication state

```
Task<UserSession> SignInAsync(string userId, string password);
Task<Participant> GetCurrentParticipantAsync();
Task<List<Event>> GetCurrentParticipantEventsAsync();
Task SignOutAsync();
```

2 Client-Side Authorization

2.1 Authentication State Provider

AuthenticationStateProvider is a service used by Blazor authorization components to provide different content or content access depending on if the user is authorized or not. If you are not using Blazor, you just need the remember login user details and the JWT token. For our application, this would be the **UserSession** object. For Blazor, you need to install the following package in **Newrise.Client** to use authorization and authorization components for Blazor as well as accessing browser storage.

Packages to install on Client project

```
Microsoft.AspNetCore.Components.Authorization
Blazored.LocalStorage
Blazored.SessionStorage
```

Register browser storage services

```
services.AddBlazoredLocalStorage();
services.AddBlazoredSessionStorage();
```

Our authentication state provider: Services\CustomAuthenticationStateProvider.cs

```
public class ClientAuthenticationStateProvider : AuthenticationStateProvider {
    private readonly AuthenticationState _anonymous;
    private readonly ISessionStorageService _storage;
    private readonly HttpClient _client;
    private AuthenticationState _state;

    public ClientAuthenticationStateProvider(
        ISessionStorageService storage, HttpClient client) {
        var anonymous_user = new ClaimsPrincipal(new ClaimsIdentity());
        _anonymous = new AuthenticationState(anonymous_user);
        _storage = storage; _client = client;
    }

    public override Task<AuthenticationState> GetAuthenticationStateAsync() {
        throw new NotImplementedException();
    }

    public Task UpdateAuthenticationStateAsync(UserSession userSession) {
        throw new NotImplementedException();
    }
}
```

Register services for Blazor authorization

```
services.AddAuthorizationCore();
services.AddScoped<AuthenticationStateProvider, ClientAuthenticationStateProvider>();
services.AddCascadingAuthenticationState();
```

2.2 Authentication State

A successful login process will result in a **UserSession** object which we can generate a **ClaimsPrincipal**. This would be enough for us create an **AuthenticationState** for Blazor authorization. To remember the login we can store the **UserSession** into local storage or session storage. To access the Web API, we need to set **HTTPClient** to use bearer authentication with the JWT token.

Method to update authentication state

```
public async Task UpdateAuthenticationStateAsync(UserSession userSession) {
    if (userSession != null) {
        await _storage.SetItemAsync("UserSession", userSession);
        _state = new AuthenticationState(userSession.GetPrincipal());
        _client.DefaultRequestHeaders.Authorization =
            new AuthenticationHeaderValue("bearer", userSession.Token);
        NotifyAuthenticationStateChanged(Task.FromResult(_state));
    }
    else {
        await _storage.RemoveItemAsync("UserSession"); _state = null;
        _state = null; _client.DefaultRequestHeaders.Authorization = null;
        NotifyAuthenticationStateChanged(Task.FromResult(_anonymous));
    }
}
```

Method to return the current authentication state

```
public override async Task<AuthenticationState> GetAuthenticationStateAsync() {
    if (_state != null) return _state;
    try {
        var userSession = await _storage.GetItemAsync<UserSession>("UserSession");
        if (userSession != null) {
            _client.DefaultRequestHeaders.Authorization =
                new AuthenticationHeaderValue("bearer", userSession.Token);
            return _state = new AuthenticationState(userSession.GetPrincipal());
        }   return _anonymous; } catch { return _anonymous; }
}
```

2.3 Login & Logout

We will now inject **AuthenticationStateProvider** into **ParticipantDataClient** since this is where it will be used so we can complete the API client.

Accessing AuthenticationStateProvider service: ParticipantDataClient.cs

```
public class ParticipantDataClient : IParticipantDataService {
    private readonly ClientAuthenticationStateProvider _authenticationStateProvider;
                    :
    public ParticipantDataClient(
        AuthenticationStateProvider authenticationStateProvider, HttpClient client) {
            _authenticationStateProvider =
                (ClientAuthenticationStateProvider)authenticationStateProvider;
                    :
    }               :
}
```

Completing login method

```
public async Task<UserSession> SignInAsync(string userId, string password) {
    var body = new LoginInfo { UserID = userId, Password = password };
    var response = await _client.PostAsJsonAsync($"{_route}/login", body);
    if (response.StatusCode != HttpStatusCode.OK)
        throw new Exception($"Server or network error ({response.StatusCode}).");
    var content = await response.Content.ReadAsStringAsync();
    var result = JsonSerializer.Deserialize<ApiResult<UserSession>>(content, _json_options);
    if (!result.Success) throw new Exception(result.Error);
    await _authenticationStateProvider.UpdateAuthenticationStateAsync(result.Value);
    return result.Value;
}
```

Completing logout method

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

Certain operations depend on the current participant's ID. This can be accessed from authentication state. Following is the method to obtain the ID and the operations that uses it.

Accessing and using current participant ID

```
private async Task<string> GetCurrentParticipantId() {
    var state = await _authenticationStateProvider.GetAuthenticationStateAsync();
    return state.User.Identity.Name;
}

public async Task<Participant> GetCurrentParticipantAsync() {
    var id = await GetCurrentParticipantId();
    return await GetParticipantAsync(id);
}

public async Task<List<Event>> GetCurrentParticipantEventsAsync() {
    var id = await GetCurrentParticipantId();
    return await GetParticipantEventsAsync(id);
}
```

While you can bring almost all pages and components over from your Blazor server application to the **Newrise.Client**, the **ContactUs** page would require a API client for **Office** services which you can try to implement as an exercise. You can still bring the other pages and components first. Minimal changes are required unless your server-side services are very different from your client-side services.

3 Razor Components

3.1 Content Files

Content files remain on the server. So copy all files from your Blazor server **wwwroot** and **Content** folder to **Newrise** project. Since your UI runs on the client MudBlazor package should be installed to **Newrise.Client** project. However the hosting page is still on the server since it is used to download the Blazor webassembly to the client to run. So all content files for MudBlazor should be included from **Newrise** project.

Application host page: Newrise/Components/App.razor

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="/" />
    <link rel="stylesheet" href="app.css" />
    <link rel="stylesheet" href="Newrise.styles.css" />
    <link href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap"
rel="stylesheet" />
    <link href="_content/MudBlazor/MudBlazor.min.css" rel="stylesheet" />
    <HeadOutlet @rendermode="InteractiveWebAssembly" />
</head>
<body>
    <Routes @rendermode="InteractiveWebAssembly" />
    <script src="_framework/blazor.web.js"></script>
    <script src="_content/MudBlazor/MudBlazor.min.js"></script>
</body>
</html>
```

MudBlazor services should be enabled where you use them. In this case it should only be registered in **Newrise.Client** project. It is possible to create a hybrid application where some part of the UI runs on the server while the rest runs on the client then it makes sense to install and setup MudBlazor on both server and client projects. In this case we are moving our server application UI entirely to webassembly.

Adding MudBlazor services: Newrise.Client/Program.cs

```
services.AddMudServices();
```

Global import of MudBlazor namespaces: _Imports.razor

```
@using MudBlazor
@using MudBlazor.Services
```

3.2 Hybrid Application

When using the Blazor Web App project template for webassembly, it is always setup to support server components as well even though it is configured for webassembly so you can then easily switch to support both. This means that whatever services you need on the webassembly side is also needed on the server side even though they are not technically utilized as the webassembly would only used the client-side registered services. Thus the following services have to be registered for the application to start without errors.

Services to register on server

```
services.AddScoped<ProtectedSessionStorage>();
services.AddScoped<AuthenticationStateProvider, ServerAuthenticationStateProvider>();
services.AddCascadingAuthenticationState();
services.AddMudServices();
```

This is a hassle if you never intended to run both the Blazor server and webassembly application together. In that case, use the ASP.NET Web Core API project template for the server and Blazor Webassembly Standalone App project template for the client instead.

3.3 Shared Components

Move your layout component, **Routes** component and the entire **Shared** folder for all shared components over to root directory of **Newrise.Client**. Then it is just a matter of getting the namespaces imported. Since client namespaces may be different from server, don't copy **_Imports** component, just go through all the components, import the missing namespaces and move them over to the new **_Imports** component. The following namespaces should resolve any issue in **Routes** component.

Import authorization services and components

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

Import layout components

```
@using Newrise.Client.Layout
```

Many pages including the layout accesses an **AppName** string from **Program** class. Add it into the class and it should remove any issues left in **DefaultLayout** and other pages.

Adding the AppName constant

```
public class Program {
    public const string AppName = "Newrise";
        :
}
```

Many pages and component uses models and services. Thus the following namespace imports should help resolve them all. Pages and components using data services need to replace **EventDataService** to **IEventDataService** and **ParticipantDataService** to **IParticipantDataService**. This should resolve issues in **EventDetails** component. All other shared components have no issues. You can now import the namespace for the shared components as well.

Importing models and services

```
@using Newrise.Shared.Models
@using Newrise.Shared.Services
@using Newrise.Client.Services
```

Importing shared components

```
@using Newrise.Client.Shared
```

3.4 Component Pages

You can now move over the **Home**, **Events** page over. Change **EventDataService** to use the interface instead should solve all issues with **Events** page. Move **ContactUs** page over only if you have completed the API client to access the Office Web API. You can also move **Event\Create** page over and just change the data service injection to **IEventDataService**. Finally move the **Pages\User** folder. There are more changes required in these pages. For the **Login** page, remove **IHostEnvironment** injection and use **IParticipantDataService** instead. Then remove **OnInitializedAsync** since this is only for setting up the administrator account which should never be done at the client which is why we never expose the method through the Web API.

Changes to Login page: User\Login.razor

```
@page "/login"
@inject IParticipantDataService ParticipantDataService
@inject NavigationManager NavigationManager
                :
@code {
    //  protected override async Task OnInitializedAsync() {
    //      if (Environment.IsDevelopment())
    //          await ParticipantDataService.InitializeAsync();
    //  }
}
```

Changes to Register page: User\Register.razor

```
@page "/register"
@inject IParticipantDataService ParticipantDataService
@inject NavigationManager NavigationManager
                    :
@code {
    NewParticipant Item { get; set; } = new NewParticipant {
//      PasswordHash = ParticipantDataService.HashPassword("P@ssw0rd")
        PasswordHash = "OwWEA4SwLUjJnhy9GhtLE10qoss7aBHYivNUagcSOEk="
    };
            :
}
```

For **Register** page, fix the initial value of **PasswordHash** since we will only hash the password on the server and we do not want to expose our algorithm nor allow its use through the Web API for security. The hash doesn't even need to be a real hash, just make sure it's not empty to pass validation.

There are much more changes to **Profile** page. We add an **Events** property to allow events to be separately fetched thus we can refresh the events without refreshing the participant.

Changes to Profile page: User\Profile.razor

```
@page "/profile"
@attribute [Authorize]
@inject IEventDataService EventDataService
@inject IParticipantDataService ParticipantDataService
@inject NavigationManager NavigationManager
@inject ISnackbar Snackbar
@inject IDialogService DialogService
```

Events is separately fetched

```
@code {
    List<Event> Events { get; set;  }

    async void SaveImageAsync(byte[] image) {
        try
        {
            await ParticipantDataService.UpdatePhotoAsync(Item.Id, image);
            Events = await ParticipantDataService.GetParticipantEventsAsync(Item.Id);
            Snackbar.Add("Profile image updated.", Severity.Success); StateHasChanged();
        }   catch(Exception) { Snackbar.Add("Profile image updated.", Severity.Error); }
    }

    async void LeaveEventAsync(Event eventItem) {
        try {
            var result = await DialogService.ShowMessageBox(
                "Warning", $"Leave event '{eventItem.Id}'?");
            if (result != true) return;
            await EventDataService.RemoveParticipantAsync(eventItem.Id, Item.Id);
            Events = await ParticipantDataService.GetParticipantEventsAsync(Item.Id);
            StateHasChanged();
        }
        catch { Snackbar.Add($"Error leaving event '{eventItem.Id}'.", Severity.Error); }
    }

    protected override async Task OnInitializedAsync() {
        Item = await ParticipantDataService.GetCurrentParticipantAsync();
        Events = await ParticipantDataService.GetParticipantEventsAsync(Item.Id);
    }
}
```

Bind EventList to separate Events property

```
<EventList Items=Events OnLeaveEvent=LeaveEventAsync />
```

Components that uses **ProtectedSessionStorage** or **ProtectedLocalStorage** must be converted to use Blazored components instead as the former components will only work on server.

3.5 Blazor Server or WebAssembly

Even though we demonstrated building **Blazor WebAssembly** applications, you can also build **Blazor Server** applications that runs on the server. Implementation is the same except no JWT authentication or Web API is required. Blazor server application can access the services directly. The custom **AuthenticationStateProvider** together with login and logout process is all implemented on the server-side.

There are pros and cons with either Blazor application type. In WebAssembly the UI runs and updates completely on the client-side, thus you may find the UI much more responsive. However this means that the entire UI together with all the packages that it references have to be downloaded to the browser. Packages are cached so the next time you access the application, the loading time will be faster. Or you can implement a Blazor app that runs on the server while downloading to the client, and switches to client once download is complete which is now possible using Blazor Web App project template.

Blazor-Server

- Fast startup time
- Less responsive UI
- No Web API needs to be implemented
- No Web API client needs to be implemented
- Direct database access
- More secure

Blazor-Webassembly

- Slow startup time
- More responsive UI
- Requires implementation of Web API
- Requires implementation of Web API Client
- Data access through Web API
- Less secure

Both Server and Webassembly

- Fast startup time on server
- Webassembly downloaded in the background
- More responsive UI once Webassembly is downloaded
- Server-rendered components can run on server only
- Client-rendered components can run on client only
- Components that can run on both must be adaptable to use server/client services

4 Publishing

4.1 Folder

If you do not have the permission to directly publish your application to the staging or production web server, you can then publish into a local folder or network folder. You can zip up the folder and send it to a Web administrator who will then simply use FTP or web administration tools provided by the web host to upload the contents into the actual web application directory. You can then customize the publish settings that are applicable regardless of publishing target.

Configuration

If you want to deploy for testing, you probably want the **Debug** version, you will get more detailed information when errors occur. For staging and production, a **Release** version will be more performant and more secure.

Target Framework

If you are not allowed to deploy frameworks to the web server that you need to use whatever is provided by the web host. If you are using public web hosting you need to check their hosting plans to see supported frameworks and versions.

Deployment Mode

Framework-dependent will compile and deploy your project in accordance to the standard requirements of the framework. **Self-contained** will attempt to compile as much as possible into a single executable so there will be few dependencies to other files. The result still depends on the additional options below.

Target-Runtime

You can select the target runtime to use. By default you can use the portable runtime which can run on any CPU and OS, since it is not pre-compiled to native code. Using using a pre-compiled native code runtime should give you better overall performance from start to end, however you will be restricted on which CPU and OS you can deploy to.

File Publish options

Produce Single File option will attempt to compile your application into one single executable file but there will still be additional files that are external depending on the packages that you use. **ReadyToRun** will attempt to compile as much as possible into native code to give you better startup performance but code size may increase up to 3 times. You can select **Trim unused code** to remove code that you never use.

Choosing **Delete all existing files prior to publish** will ensure that extra files and folders that are not part of your project that you publish will be removed. Be careful that you may accidentally remove data files generated by the application. To be safe, the application should use a folder outside of the web application directory to store non-fixed data files.

Database

Basically this allows you to select a connection string to use since we usually do not connect to the same database for development/testing, staging and production.

Entity Framework migrations

If your development database server is not the same as staging or production it can automatically apply migration to synchronize the structure of the databases. You can select the connection string to use to apply migrations.

After publishing, you should see an executable file in the target folder. ASP.NET Core has a built-in web server named Kestrel. This is the web server that we usually use during development, even though you can switch to IIS Express. Thus the executable itself is self-hosting so when you launch it, you can actually just use a web browser to access the application.

4.2 IIS - Web Deploy

You can directly publish to Microsoft IIS web servers or any server that supports Web Deploy. However you must make sure to install on your web server to ensure that all the necessary IIS modules, frameworks and runtimes are installed to allow IIS to host ASP.NET Core web applications.

Link to ASP.NET Core IIS Hosting Bundle

```
https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/hosting-bundle?
view=aspnetcore-7.0
```

The following shows typical Web Deploy connection information to use when deploying to a local IIS server. Make sure to switch off HSTS and HTTPS redirect options if the IIS server is not configured to support HTTPS.

Web Deploy connection information

```
Server      : localhost
Site Name   : Default Web Site
User Name   : <admin_username>
Password    : <admin_password>
URL         : http://localhost
```

If you are using public web hosting, the Control Panel provided by the web host would provide you with the above information if they support Web Deploy. Following shows a sample of connection information from **Smarterasp.net** web host.

Web Deploy information from Smarterasp.net sites

```
Server      : https://win8020.site4now.net:8172/MsDeploy.axd?site=xnamp99-003-site1
Site Name   : xnamp99-003-site1
User Name   : xnamp99-003
Password    : <account_password>
URL         : http://xnamp99-003-site1.ftempurl.com/
```

Of course you would also need the connection string to your database. Depending on your hosting package, you can use the Control Panel to create one or more databases and you can assign a database name and password. The following is the connection information for sample database created at **Smarterasp.net**. You should be able to access the server from your local SQL Server Management Studio.

Database Connection information

```
Data-Source : SQL8005.site4now.net
Database    : db_a97a30_newrise
User Id     : db_a97a30_newrise_admin
Password    : <db_password> (e.g. nMhrq0HFNN)
```

If the database is empty, just add a new migration before publishing. Web Deploy can apply the migrations to the database, creating the same tables as in the development database. However, migration would only synchronize the structure not the data. You can use SQL Server Management Studio to create a script of the data in the tables in your development database and run the script on the new database.