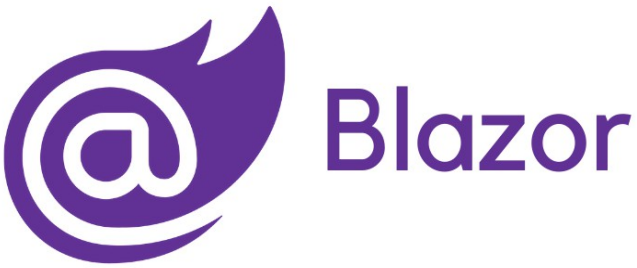



CSC530-A

Module 2

# Beginning a Blazor Server Application

# 1 Application Setup

## 1.1 Project Setup

Re-create the Blazor server application project using the following information. Enter the basic code in **Program** class to setup a basic Blazor server application host. Edit the **https** launch profile and change the port numbers to the ones below so URL links in the training notes can be copied without change.

### Project Information

```
Project Name     : Newrise
Project Template : Blazor Server App Empty
Location         : <repository>\src
Solution         : Module2
Options          : Configure for HTTPS
```

### Build and run a basic Blazor application host: Program.cs

```
public class Program {
    public static void Main(string[] args) {
        var builder = WebApplication.CreateBuilder(args);
        builder.Services.AddRazorComponents().AddInteractiveServerComponents();
        var app = builder.Build();
        if (!app.Environment.IsDevelopment()) {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAntiforgery();
        app.MapRazorComponents<App>().AddInteractiveServerRenderMode();
        app.Run();
    }
}
```

### HTTPS launch profile: Properties\launchSettings.json

```
"https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "applicationUrl": "https://localhost:7116;http://localhost:5174",
    "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
}
```

## 1.2 UI Framework

Unless you are an expert at CSS, HTML and Javascript, you need a Web UI Framework to implement your front-end. Of course, a UI framework that is specifically built for a web application framework will be preferable. Following is a list of the free and open-source UI frameworks that has support for Blazor. For this training course, we will be using MudBlazor, the most popular UI framework to date.

Free UI Frameworks for Blazor

```
Ant Design Blazor    (https://antblazor.com/en-US/)
Blazorise            (https://blazorise.com/)
BlazorStrap          (https://blazorstrap.io/)
Element Blazor       (https://element-blazor.github.io/)
MatBlazor            (https://www.matblazor.com/)
MudBlazor            (https://mudblazor.com/)
Radzen               (https://blazor.radzen.com/)
Skclusive Material   (https://skclusive.github.io/Skclusive.Blazor.Samples/Dashboard/)
```

Use the NuGet Package Manager to find and install **MudBlazor** for **Newrise** project. You can check the **Packages** folder under **Dependencies** of your project to verify that the UI framework has been installed. Open **_Imports.razor** and add **MudBlazor** namespace.

Import Mudblazor namespace: _Imports.razor

```
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using Newrise
@using MudBlazor
@using MudBlazor.Services
```

Add the following CSS stylesheets and Javascript library to your application host page for MudBlazor. You will also need to register services provided by MudBlazor by calling **AddMudServices** on the application host builder.

Adding component stylesheets: App.razor

```
<link href="css/site.css" rel="stylesheet" />
<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" />
```

Adding component script library

```
<script src="_framework/blazor.server.js"></script>
<script src="_content/MudBlazor/MudBlazor.min.js"></script>
```

Registering MudBlazor services: Program.cs

```
var builder = WebApplication.CreateBuilder(args);
        :
builder.Services.AddMudServices();
```

You can now use MudBlazor in Razor pages and components. We map **Index.razor** as the default component page of the application. Add a **Message** field or property to be displayed in **h1** heading and **UpdateMessage** method that will update the **Message**. Then add a **MudButton** to call the method from the **OnClick** event. Launch the application and test out the button. The button works but it doesn't seem to be styled properly. You need to need to a **MudThemeProvider** to the page or its layout page to provide a default or custom theme for MudBlazor components to use.

Testing a MudBlazor component: Pages\Index.razor

```
@page "/"
<h1>@Message</h1>
<MudButton Variant=@Variant.Filled Color=@Color.Primary
    OnClick=@UpdateMessage>Click Me</MudButton>
@code {
    string Message { get; set; } = string.Empty;
    void UpdateMessage() { Message = "Hello, MudBlazor!"; }
}
```

## 1.3 Page Layout

Use **MudLayout** component to define the layout of a page. To share layouts between pages, create a layout component page. A default is already provided with the project named **MainLayout.razor**. This default layout is assigned in **App.razor**. You can use a **@layout** directive to assign a different layout per page.

A sample layout: Layout\MainLayout.razor

```
@inherits LayoutComponentBase
<MudThemeProvider />
<MudLayout>
    <MudAppBar Dense=@true Elevation="0"></MudAppBar>
    <MudDrawer Open=@true Color=@Color.Primary></MudDrawer>
    <MudMainContent>@Body</MudMainContent>
    <MudAppBar Bottom=@true Dense=@true Elevation="0" Color=@Color.Secondary></MudAppBar>
</MudLayout>
```

Use **AppBar** to create application header or footer bars for putting logos, icons, links and menus. Use **MudDrawer** to create sidebars that can be fixed or slide-in from the left or right. The **MudMainContent** presented the main content area of the page. Lot of components have similar properties. Use **Elevation** to apply a shadow to separate the component from the background, to make it appear as it is floating on top of the background. Use **Dense** to reduce spacing around (*margin*) and within (*padding*) the component so it is more compact. Use **Color** property to determine the component's main color. Use the following link to see some sample layout templates that you can use for your pages.

Some sample MudBlazor layouts

```
https://www.mudblazor.com/getting-started/wireframes#main-layouts
```

Following shows more MudBlazor components; **MudIconButton** to add icon buttons; **MudSpacer** to help space-out components, **MudText** to display general text.

Our application layout: Layout\MainLayout.razor

```
@inherits LayoutComponentBase
<MudThemeProvider />
<MudLayout>
    <MudAppBar Elevation="1">
        <MudIconButton Icon=@Icons.Material.Filled.Menu
            Color=Color.Inherit OnClick=@ToggleSideBar />
        <MudSpacer />
        <MudText Typo=@Typo.h5>Newrise</MudText>
        <MudSpacer />
        <MudIconButton Icon=@Icons.Material.Filled.MoreVert
            Color=Color.Inherit Edge=@Edge.End />
    </MudAppBar>
    <MudDrawer @bind-Open=@sidebarOpened ClipMode=@DrawerClipMode.Always Elevation="2">
    </MudDrawer>
    <MudMainContent>@Body</MudMainContent>
</MudLayout>
```

To support an expandable/collapsible sidebar, declare a boolean field or property that is binded to **Open** property of a **MudDrawer** component. Add a method to toggle the boolean value and attach it to an icon **OnClick** event. Note that the **@bind** directive is required for two-way binding. This is important for binding to input components as the component needs to also update the value.

Implement expandable/collapsible sidebar

```
@code {
    bool sidebarOpened = true;
    void ToggleSideBar() { sidebarOpened = !sidebarOpened; }
}
```

Note that lambda expressions can be used to implement anonymous methods. These expressions can be directly assign to component events as well.

Using lambda expressions

```
<MudIconButton ... OnClick=@(e=>sidebarOpened = !sidebarOpened) />
```

## 1.4 Page Navigation

MudBlazor components have built-in navigation so there is no need to write any code to navigate between pages. Use the **Href** property on menu items and links to specify the navigation path to use when the component is selected. Add a new **Events** page also a **Contacts** page.

Add a new Events component page: Pages\Events.razor

```
@page "/events"
<MudText Typo=@Typo.h5>Events</MudText>
```

Add a new Contacts component page: Pages\Contacts.razor

```
@page "/contacts"
<MudText Typo=@Typo.h5>Contact Us</MudText>
```

Added navigation menu to drawer: Layout\MainLayout.razor

```
      :
<MudNavMenu>
    <MudNavLink Href="/" Match=@NavLinkMatch.All>Home</MudNavLink>
    <MudNavLink Href="/events">Events</MudNavLink>
    <MudNavLink Href="/contact">Contact Us</MudNavLink>
    <MudNavGroup Title="MudBlazor" Expanded=@false>
        <MudNavLink Target="_blank" Href="https://mudblazor.com/">Home</MudNavLink>
        <MudNavLink Target="_blank"
Href="https://mudblazor.com/features/icons#icons">Icons</MudNavLink>
        <MudNavLink Target="_blank"
Href="https://mudblazor.com/features/colors#theme-palette-colors">Colors</MudNavLink>
        <MudNavLink Target="_blank"
Href="https://mudblazor.com/features/breakpoints#breakpoints">Breakpoints</MudNavLink>
    </MudNavGroup>
</MudNavMenu>
    :
```

Use a **MudNavLink** to create a link to another component page or external site. You can use a **MudNavGroup** to group multiple links into a collapsible sub-menu. Links can be standalone or placed in a **MudNavMenu**. Add the above menu to the sidebar in the layout. Launch the application and test out the links.
To perform navigation in code, use **@inject** directive to use dependency injection to define a property where an **NavigationManager** service will be injected into. We use **Navigation** as the name of the property in the following example. Call a **NavigateTo** method to perform navigation.

Using NavigationManager service: Pages\Events.razor

```
@page "/events"
@inject NavigationManager Navigation

<MudText Typo=@Typo.h5>Events</MudText>
<MudButton OnClick=@(e=>Navigation.NavigateTo("/"))>Go Home</MudButton>
```

If you are not using a third-party UI framework, Blazor also comes with some built-in components. You can use a **NavLink** component to perform navigation instead. The path to navigate to is assigned to the **href** property.

Using Blazor NavLink component

```
<div><NavLink href="/">Home</NavLink></div>
```

# 2 File Access

## 2.1 Content Files

Not everything has to be stored in a database. A web application may need to display information that usually does not change very often. This information can be stored in an external data file so that it can still be changed easily without touching any pages and components and having to rebuild the application. If we do not wish this file to be directly accessible from the web browser, do not put it into web root folder. Create a **Content** folder instead and add the following JSON file into the folder.

JSON file containing office information: Content/offices.json

```
[
    {
        "Id"        : "Penang (mainland) office",
        "Address1"  : "484B, 2nd Floor, Jalan Permatang Rawa,",
        "Address2"  : "Bandar Perda, Bukit Mertajam,",
        "Address3"  : "14000, Penang, Malaysia.",
        "Email"     : "info@newrise.com.my",
        "Tel"       : "604-6042307",
        "Mobile"    : "016-4422957"
    },
    {
        "Id"        : "Penang (island) office",
        "Address1"  : "2-3-11, Bangunan Lip Sin,",
        "Address2"  : "Lebuh Pekaka 1, Sungai Dua,",
        "Address3"  : "11700 Gelugor, Penang, Malaysia.",
        "Email"     : "info@newrise.com.my",
        "Tel"       : "604-6042308",
        "Mobile"    : "012-4186092"
    }
]
```

We will now create a data model class to represent a single entity, which in this case is an office. Create a **Models** folder and add the following class to the folder. We can now implement a service class to deserialize **Office** objects from the file.

A data model class: Models\Office.cs

```
namespace Newrise.Models {
    public class Office {
        public string Id { get; set; } = string.Empty;
        public string Address1 { get; set; } = string.Empty;
        public string Address2 { get; set; } = string.Empty;
        public string Address3 { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public string Tel { get; set; } = string.Empty;
        public string Mobile { get; set; } = string.Empty;
    }
}
```

## 2.2 Content Provider

Add a folder named **Services** and create a **OfficeListProvider** class in it. To get the physical path to the application folder, you need access to **IWebHostEnvironment** which we can accomplish through constructor injection. Use **ContentRootPath** to get the application directory while **WebRootPath** will give you the **wwwroot** directory. We can then construct the physical location of the JSON file. Since it is a JSON file, we can use the **JsonSerializer** to deserialize it in the **Load** method, then add a **GetList** method to return the result back to the caller.

Office data provider service: Services\OfficeListProvider.cs

```
using Newrise.Models;
using System.Text.Json;

namespace Newrise.Services {
    public class OfficeListProvider {
        private readonly string _path;
        private List<Office> _offices;

        public OfficeListProvider(IWebHostEnvironment environment) {
            _path = Path.Combine(environment.ContentRootPath, @"Content\offices.json");
            Load();
        }
        public void Load() {
            using var stream = new FileStream(_path, FileMode.Open, FileAccess.Read);
            _offices = JsonSerializer.Deserialize<List<Office>>(stream);
        }
        public List<Office> GetList() {
            return _offices ??= new List<Office>();
        }
    }
}
```

We only need one instance of this object to load in the data. Once the data is loaded, it can be shared across multiple requests and users, so it makes sense to register the service as a singleton.

There are three methods that you can use to register services with the container. The method used will determine how many objects are created, when it is created and the expected activation lifetime of the object.

Service registration methods

```
Scoped      AddScoped<>()       Object is created for each request
Singleton   AddSingleton<>()    Only one object is created per application
Transient   AddTransient<>()    Object created per injection call
```

A scoped object is used to process a single request so the object is no longer active at after the response is send back to the client. This ensures that no two requests can use the same object regardless of whether the requests are from one user or different users. Transient guarantees each component and page will not share the same object even if they are processing the same request from the same user. Singleton means a single object shared across requests and users.

Registering the service: Program.cs

```
builder.Services.AddSingleton<OfficeListProvider>();
```

## 2.3 Content Presentation

To make it easy to access the model and service, import the component namespaces in the **_Imports** component. Finally, create a **Contacts** component page to display a list of offices. Use property injection to get a **OfficeListProvider** singleton. Call the **GetList** method to access and return the list of offices. Use **MudGrid** and **MudItem** to arrange content across columns and rows. We used **MudPaper** to create a border container for each office. Override **OnInitialized** or **OnInitializedAsync** method to retrieve the office list after the component is ready but before it is rendered.

Importing namespaces for models and services: _Imports.razor

```
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using Newrise
@using Newrise.Models;
@using Newrise.Services;
@using MudBlazor;
```

Contacts page: Pages\Contacts.razor

```
@page "/contacts"
@inject OfficeListProvider OfficeList

<MudGrid>
@foreach(var office in offices) {
    <MudItem>
        <MudPaper>
            <MudText Typo=@Typo.h5 Align=@Align.Center><b>@office.Id</b></MudText>
            <br />
            <MudText Typo=@Typo.overline>ADDRESS</MudText>
            <MudText Typo=@Typo.body1>@office.Address1</MudText>
            <MudText Typo=@Typo.body1>@office.Address2</MudText>
            <MudText Typo=@Typo.body1>@office.Address3</MudText>
            <br />
            <MudText Typo=@Typo.overline>CONTACT</MudText>
            <MudText Typo=@Typo.body1>
                <MudIcon Icon=@Icons.Material.Filled.Email />
                <span>@office.Email</span>
            </MudText>
            <MudText Typo=@Typo.body1>
                <MudIcon Icon=@Icons.Material.Filled.Phone />
                <span>@office.Tel</span>
            </MudText>
            <MudText Typo=@Typo.body1>
                <MudIcon Icon=@Icons.Material.Filled.Smartphone />
                <span>@office.Mobile</span>
            </MudText>
        </MudPaper>
    </MudItem>
}
</MudGrid>
```

Initializing fields before rendering

```
@code {
    List<Office> offices;

    protected override void OnInitialized() {
        offices = OfficeList?.GetList();
    }
}
```

There are a few issues with the current presentation. The office details are too close to the border and each office is too close to one another. A UI Framework provides a set of utility style classes for customization and adjustment. This can be assigned to any HTML element using the **class** property. For MudBlazor component, use the **Class** property instead. We will use 2 utility class groups; **p** for spacing within the border (padding) and **m** for spacing before the border (margin). Use **a** for all, **l** for left, **r** for right, **t** for top and **b** for bottom, followed by the size after the dash.

Setting padding and margin for MudPaper

```
<MudPaper Class="pa-8 ma-2">
```

Alternatively you can change the CSS style attributes of each element by using **Style** property on a MudBlazor component or style attribute on a HTML element. Following shows changing the alignment of the text to better align with the icon.

Solving misaligned icon and text

```
<MudText Typo=@Typo.body1>
    <MudIcon Icon=@Icons.Material.Filled.Email />
    <span style="vertical-align:top">@office.Email</span>
</MudText>
```

We may need to apply the same class and style setting to multiple elements, it would be better to declare fields for them and bind to the elements instead. It is also easier for you to change the settings later on.

Using fields to share settings between UI elements

```
@code {
    const string s1 = "vertical-align:top";
                :
```

Binding class and style to fields

```
<span style=@s1>@office.Email</span>
```

## 2.4 Responsive Content

A responsive web application can change and re-arrange content to fit the screen size of a device. This make an application work well across desktop, notebook, tablet and smartphone devices. The first thing that you need to do is to add a viewport meta tag to the application hosting page to fix the width of the page to the width of the device. A responsive UI framework will then adapt the page content to the device width.

Adding viewport meta tag: App.razor

```
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="~/" />
        :
```

If you wish to disable scaling, also set **maximum-scale** to **1.0**. Most UI frameworks uses a 12-column based responsive grid. A **MudGrid** can have up to 12 columns but then each column may be too small to show their content effectively on small devices. On a small device we may only want one **MudItem** to appear on a row but on large devices we may want up to 4. The following are breakpoints that are available in MudBlazor that you can use to determine the content width of each item for different devices.

MudBlazor Breakpoints

```
xs - Extra Small < 600px
sm - Small < 960px
md - Medium < 1280px
lg - Large < 1920px
xl - Extra Large < 2560x
xx - Extra Extra Large >= 2560px
```

Setting up the responsive size of a MudItem

```
<MudItem xs="12" md="6" xl="3">
            :
</MudItem>
```

Use breakpoint properties to set the number of logical columns that the component will take up. For XS device and above, the component will take up all 12 columns which means that only a single office will appear on one row. For MD device and above, two offices can appear on one row, and on XL devices and above, you can have up to four offices on one row.

## 2.5 Content Refresh

However if the file is changed, the user does not see the changes since the singleton will exist until the application server is restarted. There are a few ways to resolve this. One is use a memory cache to store and retrieve the office list. Another method is to setup a file watcher to watch the file for changes. Easiest solution is to use a scoped service but the file has to be reloaded for every request.

When using caching, an expiration period can be attached to the cache item so that it has to be reloaded after a certain amount of time has passed or has passed since last accessed. If for example, the expiration period is 1 hour, is it guaranteed that the data will be accurate after each hour or each hour since it was last accessed if you are using sliding expiration. Another method is to ask the file system to monitor the file and inform us if it has changed. You can do this by using the **FileSystemWatcher** class. Make changes to **OfficeListProvider** as shown  to use this class.

Add fields to store filename and watcher object: Services\OfficeListProvider.cs

```
public const string Filename = "offices.json";
private readonly FileSystemWatcher _watcher;
private readonly string _path;
private List<Office> _offices;
```

Setup watcher in constructor

```
public OfficeListProvider(IWebHostEnvironment environment) {
    string directory = Path.Combine(environment.ContentRootPath, "Content");
    _path = Path.Combine(directory, Filename);
    _watcher = new FileSystemWatcher(directory, Filename);
    _watcher.NotifyFilter = NotifyFilters.LastWrite;
    _watcher.EnableRaisingEvents = true;
    _watcher.Changed += OnListChanged;
    Load();
}
```

Create a **FileSystemWatcher** and pass in the directory and file filter to specify which directory and files to watch. Set **NotifyFilter** to apply additional filter on the changes to watch for. Ensure **EnableRaisingEvents** is set to **true** as we use event handler to process the change; **OnListChanged** event handler to the **Changed** event. There are also **Created**, **Deleted** and **Renamed** events. Note replacing an existing file does not trigger a Changed event, but a combination of Deleted and Created events.

Event hander to reload the office list

```
private void OnListChanged(object sender, FileSystemEventArgs e) {
    Load();
}
```

If your file filter covers more than a file, you can get the **Name** or the **FullPath** from the **FileSystemEventArgs** parameter to check which file the event is trigged for. To test this, build and reload the application and open the **Contacts** page, then open the **offices.json** file, edit some information, then close and save the file. If you switch to another page and switch back to the **Contacts** page, the changed content will appear.

## 2.6 Content Caching

Some content are not accessed frequently, so you do not want to load them nor keep them in memory for a long period of time. Call the **AddMemoryCache** method to add caching service to your application.

Adding caching service: Program.cs

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddMemoryCache();
```

We will now update our **OfficeListProvider** to use caching. Declare **IMemoryCache** field and use constructor dependency injection to obtain the cache service.

Declare a field for cache service: Services\OfficeListProvider.cs

```
public class OfficeListProvider {
    public const string Filename = "offices.json";
    private readonly FileSystemWatcher _watcher;
    private readonly string _path;
    private IMemoryCache _cache;
                :
}
```

Obtain cache service from constructor injection

```
public OfficeListProvider(IWebHostEnvironment environment, IMemoryCache cache) {
                :
    _cache = cache;
}
```

When the file is updated, we will manually remove any old office list from the cache. We will use the filename as the key for registering and retrieving the information from the cache.

Removing office list from the cache

```
private void OnListChanged(object sender, FileSystemEventArgs e) {_cache.Remove(Filename);}
```

When **GetList** is called, we will attempt to access the office list from the cache, if it is not available, then only we will load and store into the cache. So the office list will not be loaded unless this method is used. Subsequent calls will return the same office list without re-loading. Use **Get** and **Set** to retrieve or store objects into the cache.

Retrieving office list from the cache

```
public List<Office> GetList() {
    var offices = _cache.Get<List<Office>>(Filename);
    if (offices == null) {
        using var stream = new FileStream(_path, FileMode.Open, FileAccess.Read);
        offices = JsonSerializer.Deserialize<List<Office>>(stream);
        _cache.Set(Filename, offices); }
    return offices;
}
```

## 2.7 Cache Expiration

**IMemoryCache** service provides many auto-expiration options. When a cached item is expired, it will be automatically removed from the cache. You can expire based on a fixed time period from the moment you added the item to the cache. To set multiple expiration options including sliding expiration, use **MemoryCacheEntryOptions**.

Expire cached item in 15 minutes

```
_cache.Set(Filename, offices, TimeSpan.FromMinutes(15));
```

Expire cached item in 15 minutes from last access

```
_cache.Set(Filename, offices, new MemoryCacheEntryOptions {
    SlidingExpiration = TimeSpan.FromMinutes(15)});
```

**IMemoryCache** can also accept a change token. You can get this token when calling a **Watch** method on the environment **ContentRootFileProvider** property. This will allow you to watch for file changes without directly using **FileSystemWatcher**. Using FileSystemWatcher however gives you more options. You can use it along with other time expiration features of the cache.

Declaring a change token field

```
private IChangeToken _token;
```

Obtaining a change token from watching a file

```
_token = environment.ContentRootFileProvider.Watch(@"Content\offices.json");
```

Registering cache item with change token

```
_cache.Set(Filename, offices, _token);
```

## 2.8 Code Separation

At the moment, the Razor component page contains both presentation and logic code. If you wish to separate presentation and logic code, all you need to do is to create a class file following the full name of the Razor component. Mark the class as **partial** and you can then move the code from within the **@code** block into the class and then everything should work as before. You can move other directives to the class as well. Replace **@inject** directive with property injection.

Using partial class: Pages\Contact.razor.cs

```
public partial class Contact {
    const string s1 = "vertical-align:top";
    [Inject]OfficeListProvider OfficeList { get; set; }
    List<Office> offices;
    protected override void OnInitialized() { offices = OfficeList?.GetList(); }
}
```

Another choice is to define a base class to be inherited by the page using a **@inherits** directive. The class must extend from **ComponentBase** and properties and methods to be marked **protected** or **public**. This is useful to share the same members across multiple pages. You can also combine both, where shared members are in a separate inheritable class and specialized members in the page or partial page class.

Inheritable Razor component class

```
public class ContactBase : ComponentBase {
    public const string s1 = "vertical-align:top";
                :
```

Inheriting from component page

```
@inherits ContactBase
```

# 3 Database Access

## 3.1 Data Model

Using Entity Framework you can generate the data model from an existing database (database first) or generate a database from a manually designed data model (code first). For the project we will use a code first approach. First ensure that the following Entity Framework Core packages have been installed to the project.

Entity Framework Core packages

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Tools
```

We need to have a data model to represent the events organized by the company and users can sign up to participate in those events. First add an **EventType** enumeration to identify all the different types of events to be organized. Note we start from 1 so 0 can used for input validation as an indication that the user has not selected a value. Then add both an **Event** and the **Participant** classes as well.

Enumeration of event types: Models\EventType.cs

```
namespace Newrise.Models {
    public enum EventType {
        None = 0,
        Presentation,
        Training,
        Workshop,
        Forum
    }
}
```

Entity class for event: Models\Event.cs

```
namespace Newrise.Models {
    public class Event {
        public string Id { get; set; }
        public EventType Type { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime From { get; set; }
        public double Hours { get; set; }
        public DateTime To { get { return From.Add(TimeSpan.FromHours(Hours)); }}
        public int Seats { get; set; }
        public int AllocatedSeats { get; set; }
        public int RemainingSeats { get { return Seats - AllocatedSeats; }}
        public decimal Fee { get; set; }
        public bool Online { get; set; }
    }
}
```

Entity class for event participant: Models\Participant.cs

```
namespace Newrise.Models {
    public class Participant {
        public string Id { get; set; }
        public string Name { get; set; }
        public string Company { get; set; }
        public string Position { get; set; }
        public string Email { get; set; }
        public byte[] Photo { get; set; }
    }
}
```

In Entity Framework relationships can be formed between entities in a data model by using virtual properties. One-to-one, one-to-many or many-to-many relationships can be formed depending on whether type of property represents one entity or an entity collection. In our project one event can have multiple participants and one participant can participate in multiple events. Thus a many-to-many relationship will be required. Always use the **ICollection<T>** as property type but actual collection class can be **List<T>** or **HashSet<T>** if duplicates are not allowed. For example each participant cannot register more than once for each event. So every participant in one event will be unique and every event the participant is registered for is unique.

Each event can have multiple participants: Models\Event.cs

```
public class Event {
            :
    public virtual ICollection<Participant> Participants { get; set; } =
        new HashSet<Participant>();
}
```

Each participant can register for multiple events: Models\Participant.cs

```
public class Participant {
            :
    public virtual ICollection<Event> Events { get; set; } = new HashSet<Event>();
}
```

To create a new database or create tables in an existing database from the model you need to implement a **DbContext** class. Declare a **DbSet<T>** property for each entity type to generate a table in the database. Add a constructor to accept a configuration object and pass it to the base class. This object will provide configuration information such as the data provider to use and the connection string.

Database model class: Models\NewriseDbContext.cs

```
namespace Newrise.Models {
    public class NewriseDbContext : DbContext {
        public DbSet<Event> Events { get; set; }
        public DbSet<Participant> Participants { get; set; }
        public NewriseDbContext(DbContextOptions<NewriseDbContext> options) : base(options) {}
    }
}
```

## 3.2 Data Annotations

Our current data model does not provide enough details for the Entity Framework to create an optimal database. It will try to detect the primary key and the datatype for each column. However, all string columns will be nullable except for the primary key and the maximum length cannot be set as this information is not available from the data model. You can attach data annotations to the data model to provide additional information, not only for improving or customizing database generation but can also used for model validation and presentation.

Some data annotation attributes

```
Key                 Primary key - generation only
DataType            Customize the data type – generation only
Required            Cannot be null / blank – generation & validation
StringLength        String maximum and minimum length – generation & validation
EmailAddress        Matches standard email pattern – validation only
RegularExpression   Matches provided pattern – validation only
Range               Minimum to maximum value – validation only
```

The following are data annotation attributes that we will assign to **Event** members. If you want to use the data model as a view model for input and presentation purposes then **ErrorMessage** parameter is important for showing validation error messages.

Annotations for Event entity class: Models/Event.cs

```
[Key, Required(ErrorMessage = "{0} is required.")]
[StringLength(6, ErrorMessage ="{0} can only have {1} characters.")]
[RegularExpression(@"^[A-Z]{3}\d{3}$", ErrorMessage ="{0} is not correctly formatted.")]
public string Id { get; set; }

[Range(1, 4, ErrorMessage = "{0} is not valid.")]
public EventType Type { get; set; }

[Required(ErrorMessage = "{0} is required.")]
[StringLength(50, ErrorMessage = "{0} can only have {1} characters.")]
public string Title { get; set; }

[StringLength(1000, ErrorMessage = "{0} can only contain {1} characters.")]
public string Description { get; set; }

[Range(0.5, 8.0, ErrorMessage = "{0} must be from {1} to {2}.")]
public double Hours { get; set; }

[Range(1, 200, ErrorMessage = "{0} must be from {1} to {2}.")]
public int Seats { get; set; }

[Range(0, 5000, ErrorMessage = "{0} must be from {1} to {2}.")]
public decimal Fee { get; set; }
```

The following are the data annotation attributes added for the **Participant** members. You can also write code to customize database generation by overriding the inherited **OnModelCreating** method and use the builder to customize columns as well as add constraints and indexes. Use **HasAlternateKey** method to create a unique constraint on the **Email** column of **Participant** and customize precision of **Fee** column.

Annotations for Participant entity: Models\Participant.cs

```
[Key, Required(ErrorMessage = "{0} is required.")]
[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Id { get; set; }

[Required(ErrorMessage = "{0} is required.")]
[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Name { get; set; }

[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Company { get; set; }

[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Position { get; set; }

[Required(ErrorMessage = "{0} is required.")]
[StringLength(254, ErrorMessage = "{0} can only have {1} characters.")]
[EmailAddress(ErrorMessage = "{0} is not correctly formatted.")]
public string Email { get; set; }
```

Customize database generation: Models\NewriseDbContext.cs

```
protected override void OnModelCreating(ModelBuilder builder) {

    builder.Entity<Participant>().HasAlternateKey(p => p.Email);
    builder.Entity<Event>().Property("Fee").HasColumnType("decimal").HasPrecision(7, 2);
}
```

## 3.3 Database Migrations

It is time to apply our model to a new database or existing database. First construct the connection string to reference the database. Then call a **AddDbContextFactory** to register a factory to create **NewriseDbContext** instances. Setup the configuration options to use SQL Server and pass in the connection string to use.

Registering and configuring a DbContext factory: Program.cs

```
var connectionString =
    "Server=.;Database=NewriseDb;TrustServerCertificate=True;Trusted_Connection=True";
builder.Services.AddDbContextFactory<NewriseDbContext>(
    options => options.UseSqlServer(connectionString));
```

Open up the **Package Manager Console** and then type in the following command to create and name your initial database migration. This will be created in a **Migrations** folder. It generates a class so you can actually go through the generated code to see if it is correct. The **Up** method shows what operations will take place when migration is applied to the database and the **Down** method shows the operations to revert the migration. When you are ready to apply the migration to the database, you can use a **Update-Database** command. It will run the **Up** method in the migration class.

Add a database migration

```
Add-Migration InitialCreate
```

## Apply the latest migration

```
Update-Database
```

If you have not applied the migration, you can simply delete the migration class. Use **Remove-Migration** command to revert an already applied migration. This will then run the **Down** method in the migration class.

## Reverting an applied migration

```
Remove-Migration
```

If a migration is used to create a new database, you can just delete the migration and use the following command to delete the database. You would not need to revert any migration for a new data model and database.

## Deleting new database

```
Drop-Database
```

The connection string that we are using is only for local development and probably is not the same for testing or production environment. The location of the database will not be the same for development, testing and production. Secondly, you wouldn't be using a trusted connection for production since you wouldn't be creating a **Windows** account for every user in the world. Also connections cannot be recycled across users that are using their individual Windows account to connect to the database even when connection pooling is enabled. Data access performance is greatly reduced when new connections have to be opened and authenticated for every request.

The best method is to store the connection string externally in the **appsettings.json** configuration file. Note that each runtime environment has its own configuration file to be merged with the main configuration file. So for a development only connection strings, store them in **appsettings.Development.json** file instead. Add them to the **ConnectionStrings** section and give each one a key. It is common to use a database name as the connection string key.

## Development only configuration: appsettings.Development.json

```
{
   "ConnectionStrings": {
        "NewriseDb": "Server=.;Database=NewriseDb;TrustServerCertificate=True;Trusted_Connection=True"
    },
         :
}
```

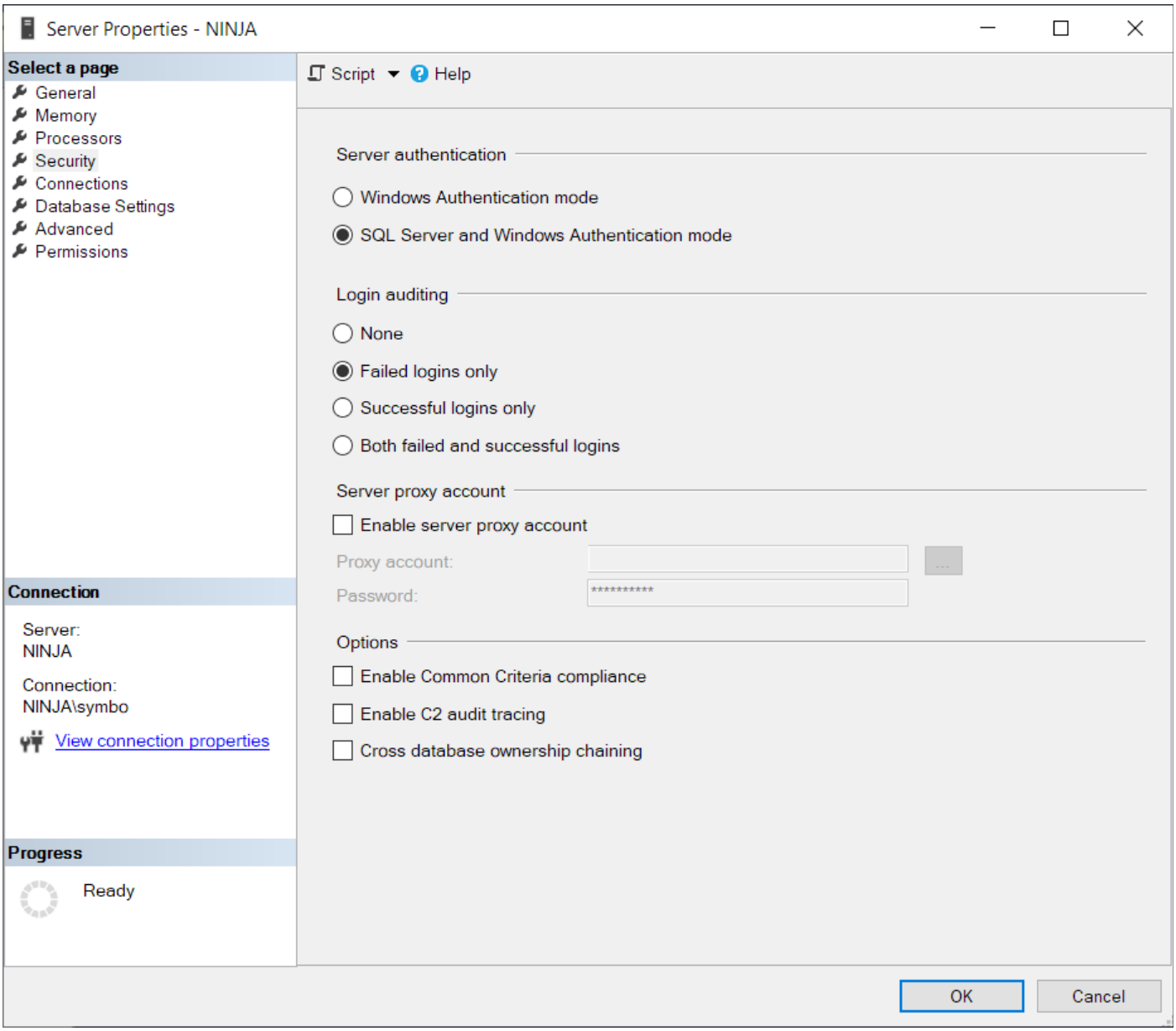
You can now access the connection string by calling a **GetConnectionString** method from **Configuration** property of the application host builder based on the what is the current environment.

## Retrieving connection string from configuration file: Program.cs

```
var connectionString = builder.Configuration.GetConnectionString("NewriseDb");
builder.Services.AddDbContextFactory<NewriseDbContext>(options =>
    options.UseSqlServer(connectionString));
```

Testing is not only done by the development team members but can also be company staff or external end users. It would be better for to use SQL Server authentication instead of Windows authentication to access the database in testing and production environments. First you need to make sure SQL Server authentication is enabled as it is disabled by default when you initially install the SQL Server database engine using default settings. You can do this in SQL Server Management Studio. Right-click on the server in **Object Explorer** and select **Properties** option. Switch to **Security** page to see the authentication options.

Retrieving connection string from configuration file: Program.cs




You can then go to the **Logins** under **Security** node in **Object Explorer** to create or manage existing SQL Server accounts. There is an administrative account named **sa** but it has been disabled. Enabled it and set the password if you want to use it instead of using your Windows account to manage SQL Server. However do not use it for your application. You should always create a separate SQL Server accounts for applications and set just enough permissions for the application to accomplish its data tasks.

For testing and production databases, add a new SQL Server login using the following information and map that login into the application database and assign it the security roles for that database. Assign the connection string to the main configuration file so it becomes the default for non-development environments. Use User Id and Password fields to specify the account name and password in the connection string.

New SQL Server account

```
Name            :   NewriseUser
Password        :   AF21631253214D62920AAD8C4F453BC0
User Mapping    :   NewriseDb
                    Database Roles:
                        db_datareader
                        db_datawriter
```

Setting the default connection string for non-development: appsettings.json

```
{
    "ConnectionStrings": {
        "NewriseDb" : "Server=NINJA;Database=NewriseDb;TrustServerCertificate=True;
User Id=NewriseUser;Password=AF21631253214D62920AAD8C4F453BC0"
    },
            :
```

## 3.4 Data Service

We want to isolate as much processing code as possible away from the UI layer. The UI layer should concentrate only on presentation and input. Input should include the validation of input data. There's also a need to be able to abstract code that has to run on a server in case the UI runs on the client rather than on the server. This would make it difficult to convert a Blazor Server project to a Blazor Webassembly project if the UI layer is directly accessing the database.

We will now implement a separate data service class to access the database through the entity framework model we have created. In the future we can easily abstract the functionality of this class by using an interface. The class will use constructor injection to get access to the **DbContext** factory for Newrise.

Access to DbContext factory: Services\EventDataService.cs

```
namespace Newrise.Services {
    public class EventDataService {
        readonly IDbContextFactory<NewriseDbContext> _dcFactory;
        public EventDataService(IDbContextFactory<NewriseDbContext> dcFactory) {
            _dcFactory = dcFactory;
        }
    }
}
```

When we need to access the database, we will use the factory to create an instance of **NewriseDbContext**. We need to make sure the instance is disposed to make sure that the database connection returns back to the connection pool after use. Objects that has a **Close** or **Dispose** method need these methods to be called after use so the resources used my the method are released back to the system immediately.

Even though the resources will definitely be released when the object is collected by the Garbage Collector, you have no idea when it will run. If the object is not collected, it will hold on to resources that cannot be recycled for another process or user. You must make sure the object is disposed regardless of success or failure. We can do this through a **try finally** block.

## Ensuring DbContext is disposed after use

```
public void AddEvent(Event item) {
    var dc = _dcFactory.CreateDbContext();
    try {
        dc.Events.Add(item);
        dc.SaveChanges();
    }
    finally {
        dc.Dispose();
    }
}
```

There is a simpler way of writing the above code, you can use a **using** statement to create an object and use that object within the **using** code block. When the block is exited, regardless of success or failure, the object will be disposed. The following code shows the correct way of creating and using a **DbContext** object to add an **Event** to the database. We can also implement an asynchronous version of the method.

## Using C# auto-dispose feature

```
public void AddEvent(Event item) {
    using (var dc = _dcFactory.CreateDbContext()) {
        dc.Events.Add(item);
        dc.SaveChanges();
    }
}
```

## Asynchronous version of the above method

```
public async Task AddEventAsync(Event item) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        await dc.Events.AddAsync(item);
        await dc.SaveChangesAsync();
    }
}
```

Following are the synchronous and asynchronous methods to remove an **Event** from the database. We should always try to avoid runtime errors whenever possible. There are many ways to ensure users are least likely to make mistakes in the UI layer.

## Removing an existing Event

```
public void RemoveEvent(string id) {
    using (var dc = _dcFactory.CreateDbContext()) {
        var item = dc.Events.Find(id);
        if (item != null) {
            dc.Events.Remove(item);
            dc.SaveChanges(); }
    }
}
```

## Asynchronous version

```
public async Task RemoveEventAsync(string id) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var item = await dc.Events.FindAsync(id);
        if (item != null) {
            dc.Events.Remove(item);
            await dc.SaveChangesAsync();
        }
    }
}
```

You cannot directly use **SaveChanges** on one **DbContext** to update a changed entity retrieved using another DbContext. There are two ways to resolve this; re-fetch the entity and transfer the properties from a previous entity or re-attach a previous entity to the current DbContext and set its **EntityState** and **SaveChanges** will then try to update the database according to that state.

## Refetch entity and update its properties

```
public void UpdateEvent(Event item) {
    using (var dc = _dcFactory.CreateDbContext()) {
        var source = dc.Events.Find(item.Id);
        if (source != null) {
            source.Type = item.Type;
            source.Title = item.Title;
            source.Description = item.Description;
            source.From = item.From;
            source.Hours = item.Hours;
            source.Seats = item.Seats;
            source.Fee = item.Fee;
            source.Online = item.Online;
            dc.SaveChanges();
        }
    }
}
```

## Reattach the previous entity and update its EntityState

```
public void UpdateEvent(Event item) {
    using (var dc = _dcFactory.CreateDbContext()) {
        var entry = dc.Events.Attach(item);
        entry.State = EntityState.Modified;
        dc.SaveChanges();
    }
}
```

## Asynchronous version of the above method

```
public async Task UpdateEventAsync(Event item) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var entry = dc.Events.Attach(item);
        entry.State = EntityState.Modified;
        await dc.SaveChangesAsync();
    }
}
```

### Returning all events as a List collection

```
public List<Event> GetEvents() {
    using (var dc = _dcFactory.CreateDbContext())
        return dc.Events.ToList();
}
```

### Asynchronous version

```
public async Task<List<Event>> GetEventsAsync() {
    using (var dc = await _dcFactory.CreateDbContextAsync())
        return await dc.Events.ToListAsync();
}
```

We will now register the data service class as a singleton. This object can be fetched by other services through constructor injection or by **Razor** components and pages through property injection.

### Registering our data service

```
builder.Services.AddSingleton<EventDataService>();
```

## 3.5 Forms & Validation

We will first setup the **Events** page and add a button to navigate to another page the user will use to add new events. We will update this page later to show the events as it is pointless to work on this page when there are no events. You will have to setup mock data if you want to work on this page first. We will add a folder named **events** and add a Razor component page named **Create.razor** in the folder.

### Adding a button to add new event: Pages\Events.razor

```
<MudGrid Class="ma-8">
    <MudItem xs="12">
        <MudText Typo=@Typo.h5>Events</MudText><br />
        <MudButton Variant="Variant.Filled" Color="Color.Primary"
            Href="/events/new">ADD EVENT</MudButton>
    </MudItem>
</MudGrid>
```

### Setup a new page to add event: Pages\events\Create.razor

```
@page "/events/new"
@inject EventDataService DataService

<EditForm Model="@Item" OnValidSubmit="@HandleSubmit">
    <DataAnnotationsValidator />
    <MudGrid>
        <MudItem xs="12">
            <MudCard Class="pa-8 ma-2">
                <MudCardContent></MudCardContent>
                <MudCardActions></MudCardActions>
            </MudCard>
        </MudItem>
    </MudGrid>
</EditForm>
```

Even though MudBlazor has a **MudForm** component, you need to use ASP.NET Core's built-in **EditForm** if you want to use data annotations to validate your model so you do not need to add specific validation components to your form. All you need to do is add a DataAnnotationsValidator to the form and it will generate the validation based on the model you assigned to the **Model** property of the form. Also assign a method to **OnValidSubmit** event to process the model that has passed validation. Assign a method to **OnInvalidSubmit** if you want to detect and process failed validations. In the form we will use a **MudCard** to contain the form contents in a **MudCardContent** section and action buttons in the **MudCardActions** section. To use the ASP.NET Core form and input components, you need to import the following namespace

Adding namespaces for ASP.NET Core form components: _Imports.Razor

```
@using Microsoft.AspNetCore.Components.Forms
                :
```

Initially we will add two **MudAlert** components to display success and error messages in the page after the page title. These components will not be added if the messages are empty. To submit the contents of the form, you need a **Submit** button which will trigger the form to validate the input and call one of the methods that are assigned to **OnValidSubmit** and **OnInvalidSubmit** depending on whether the form validation is successful or failed.

Initial contents of the card

```
<MudText Typo="Typo.h5">New Event</MudText><br />
@if (Success != string.Empty) {
    <MudAlert Severity="Severity.Success">@Success</MudAlert><br />
}
@if (Failure != string.Empty) {
    <MudAlert Severity="Severity.Error">@Failure</MudAlert><br />
}
```

Action buttons of the card

```
<MudButton Color="Color.Primary" Variant="Variant.Filled"
   ButtonType="ButtonType.Submit">SAVE</MudButton>
<MudButton Class="ml-2" Color="Color.Primary" Variant="Variant.Filled"
    Href="/events">VIEW EVENTS</MudButton>
```

Providing properties and methods to bind to

```
@code {
    string Success { get; set; } = string.Empty;
    string Failure { get; set; } = string.Empty;
    Event Item { get; set; } = new Event { From = DateTime.Now };

    void HandleSubmit(EditContext context) { DataService.AddEvent(Item); }
    void HandleSubmitAsync(EditContext context) { await DataService.AddEventAsync(Item); }
}
```

We have yet to add any input components for our **Event** model. You can use either ASP.NET Core's built-in input components or the ones provided by MudBlazor. We will be using MudBlazor input components so we don't have to use a Javascript UI library and CSS to create and style a responsive UI.

Normally each input component will be on a separate row, you can use **MudStack** to combine multiple input components into one row. However you should make sure that those components would fit well across all size breakpoints. Alternatively you can use **MudHidden** to create separate variation of the form inputs to cater for the different size breakpoints.

## Using MudStack to combine components into one row

```
<MudStack Row="true">
    <MudTextField Label="ID *" @ref="Id" @bind-Value="Item.Id" MaxLength="6"
        For="@(() => Item.Id)" AutoFocus="true" /><br />
    <MudSelect T="EventType" Label="EVENT TYPE *" AnchorOrigin="Origin.BottomCenter"
        @bind-Value="Item.Type" For="@(()=>Item.Type)">
        <MudSelectItem T="EventType" Value="EventType.Presentation" />
        <MudSelectItem T="EventType" Value="EventType.Training" />
        <MudSelectItem T="EventType" Value="EventType.Workshop" />
        <MudSelectItem T="EventType" Value="EventType.Forum" />
    </MudSelect>
    <MudCheckBox Class="pt-4" Label="Available Online" @bind-Checked="Item.Online" />
</MudStack><br />
```

## Using MudHidden to change presentation based on size breakpoints

```
<MudHidden Breakpoint="Breakpoint.SmAndDown">
    <MudStack Row="true">
        :
    </MudStack>
</MudHidden>
<MudHidden Breakpoint="Breakpoint.MdAndUp">
    <MudTextField Label="ID *" @ref="Id" @bind-Value="Item.Id" MaxLength="6"
        For="@(() => Item.Id)" AutoFocus="true" /><br />
    <MudSelect T="EventType" Label="EVENT TYPE *" AnchorOrigin="Origin.BottomCenter"
        :
    </MudSelect>
    <MudCheckBox Class="pt-4" Label="Available Online" @bind-Checked="Item.Online" />
</MudHidden>
```

We used three different input components to input **Id**, **Type** and **Online** properties of an **Event** object; **MudTextField**, **MudSelect** and **MudCheckBox**. The **For** property is to return the identity of the entity property that this input component is used for so that validation error messages can be attached to the correct component.

If you wish to access any input component from code, use **@ref** to bind it to a field or property. Note that the first **MudTextField** is assigned to an **Id** field or property so that we can change the focus to this component from code. Even though **AutoFocus** is enabled, it only works when the page is fetched for the first time and will not reset on every form submit.

## Declaring a field to bind to a component

```
@code {
    MudTextField<string> Id;
        :
}
```

## Input components for Title and Description

```
<MudTextField Label="TITLE *" @bind-Value="Item.Title" MaxLength="50"
    For="@(() => Item.Title)" /><br />
<MudTextField Label="DESCRIPTION" @bind-Value="Item.Description" MaxLength="1000"
    For="@(()=>Item.Description)" Lines="5" /><br />
```

We have a problem when using the MudBlazor components to input a date and time. There is no combined date and time picker component. The **MudDatePicker** will only a date to be selected and **MudTimePicker** will only work with **TimeSpan** and not a **DateTime** value and the types must be nullable. However our **Event** data model has a single non-nullable **DateTime** From property.

## Input components for From

```
<MudStack Row="true">
        <MudDatePicker Label="START DATE" @bind-Date="FromDate" />
        <MudTimePicker Label="START TIME" @bind-Time="FromTime" />
</MudStack><br />
```

One way to resolve this issue is to create wrapper properties that would satisfy all the requirements of the input component but the data is still retrieved and stored into the original data model property.

## Retrieve and store the date portion of From property

```
DateTime? FromDate {
    get { return Item.From;  }
    set {
        if (value != null) {
            Item.From = new DateTime(
                value.Value.Year,
                value.Value.Month,
                value.Value.Day,
                Item.From.Hour,
                Item.From.Minute,
                Item.From.Second);
        }
    }
}
```

## Retrieve and store the time portion of From property

```
TimeSpan? FromTime {
    get { return Item.From.TimeOfDay;  }
    set {
        if (value != null) {
            Item.From = new DateTime(
                Item.From.Year,
                Item.From.Month,
                Item.From.Day,
                value.Value.Hours,
                value.Value.Minutes,
                value.Value.Seconds);
        }
    }
}
```

Alternatively we can update our **Event** model class to include this two properties but we have to make sure Entity Framework does not generate a column for them as the data still comes from and stored into the existing **From** column. You can do this when you attached the **NotMapped** attribute.

## Adding properties but not columns: Models\Event.cs

```
[NotMapped]
public DateTime? FromDate {
    get { return From; }
    set {
        if (value != null) {
            From = new DateTime(
                value.Value.Year,
                value.Value.Month,
                value.Value.Day,
                From.Hour,
                From.Minute,
                From.Second
            );
        }
    }
}

[NotMapped]
public TimeSpan? FromTime {
    get { return From.TimeOfDay; }
    set {
        if (value != null) {
            From = new DateTime(
                From.Year,
                From.Month,
                From.Day,
                value.Value.Hours,
                value.Value.Minutes,
                value.Value.Seconds);
        }
    }
}
```

## Binding back to model properties

```
<MudStack Row="true">
        <MudDatePicker Label="START DATE" @bind-Date="Item.FromDate" />
        <MudTimePicker Label="START TIME" @bind-Time="Item.FromTime" />
</MudStack><br />
```

## Rest of the input components

```
<MudStack Row="true">
    <MudNumericField Label="HOURS" @bind-Value="Item.Hours"
        Min="0.5" Max="8.0" Step="1.0" For="@(()=>Item.Hours)" />
    <MudNumericField Label="SEATS" @bind-Value="Item.Seats"
        Min="1" Max="200" Step="10" For="@(()=>Item.Seats)" />
    <MudNumericField Label="FEE" @bind-Value="Item.Fee"
        Min="0" Max="5000" Step="100" For="@(()=>Item.Fee)" />
</MudStack><br />
```

Use **MudNumericField** to input or edit numerical type properties. You can use **Min** and **Max** to control the range of values so that you do not even need to use a **Range** validator. There are up and down buttons attached by default to adjust the value. You can set **HideSpinButtons** to **true** if you don't want the buttons to appear. Finally we finish up by implementing the method to be called when the model passes validation to update the database.

## Synchronous and asynchronous methods to update the database

```
void HandleSubmit(EditContext context) {
    try
    {
        Success = string.Empty;
        Failure = string.Empty;
        DataService.AddEvent(Item);
        Success = $"Event {Item.Id} added successfully";
        Item = new Event { From = DateTime.Now };
    }
    catch (Exception ex) {
        Failure = $"{ex.Message} Cannot add event.";
    }
    finally {
        StateHasChanged();
        Id.FocusAsync();
    }
}

async void HandleSubmitAsync(EditContext context) {
    try {
        Success = string.Empty;
        Failure = string.Empty;
        await DataService.AddEventAsync(Item);
        Success = $"Event {Item.Id} added successfully";
        Item = new Event { From = DateTime.Now };
    }
    catch (Exception ex) {
        Failure = $"{ex.Message} Cannot add event.";
    }
    finally {
        StateHasChanged();
        await Id.FocusAsync();
    }
}
```

We have exception handling in the above methods because input validation does not always guarantee that the database operation will be success. The operation may still fail certain table and column constraints. By default Entity Framework does not return the actual database error, it will return a general Entiry Framework error message and the actual error message can be retrieved from **InnerException** of the error. It will be pointless to do so unless the error is produced from a stored procedure, otherwise a constraint error is often too technical to be understood by a normal user. A common error that we will get from the above operation is a primary key violation error where you try to add more than one event with the same **Id**. What we can do is to check for a duplicate event Id before adding the event. In this way, we can create a better error message that the one from the database server or from Entity Framework.

Avoiding constraint errors by checking in code: Services\EventDataService.cs

```
public void AddEvent(Event item) {
    using (var dc = _dcFactory.CreateDbContext()) {
        if (dc.Events.Find(item.Id) != null)
            throw new Exception($"Event with ID '{item.Id}' already exists.");
        dc.Events.Add(item);
        dc.SaveChanges();
    }
}

public async Task AddEventAsync(Event item) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        if (await dc.Events.FindAsync(item.Id) != null)
            throw new Exception($"Event with ID '{item.Id}' already exists.");
        await dc.Events.AddAsync(item);
        await dc.SaveChangesAsync();
    }
}
```

## 3.6 Validation Summary

Currently validation errors is attached to the input components which is good enough. Optionally you can also add in a **ValidationSummary** component to render all of the error messages. The following demonstrates showing the validation summary inside a **MudAlert** component controlled by a **ValidationFailed** property. The property can be set when validation fails which we can detect through a **OnInvalidSubmit** method call. We will use an anonymous method to trigger the state change.

Displaying validation summary: Pages\events\Create.razor

```
@if (ValidationFailed) {
        <MudAlert Severity="Severity.Error"><ValidationSummary /></MudAlert>

}
```

Triggering the above section when validation fails

```
<EditForm Model="@Item"
    OnInvalidSubmit="()=>ValidationFailed=true"
    OnValidSubmit="HandleSubmitAsync">
```

ValidationSummary control property

```
bool ValidationFailed { get; set; }
```

Reset the property on every submit

```
void HandleSubmit(EditContext context) {
    try {
        ValidationFailed = false;
                :

async void HandleSubmitAsync(EditContext context) {
    try {
        ValidationFailed = false;
                :
```

We will now return back to the **Events** page to display the list of events added to the database. You can use **MudSimpleTable** or **MudTable**. Advantage of using the later is that you can assign items to be rendered and it will generate a row for each item using a **RowTemplate** without having to implement a **for-each** loop. It has support for paging, sorting, filtering, selection and editing features. It has many sections like **ToolBarContent**, **HeaderContent**, **FooterContent** and **PagerContent** as well.

Displaying a list of events: Pages\Events.razor

```
@page "/events"
@inject EventDataService DataService

<MudGrid>
    <MudItem Class="ma-8" xs="12">
        <MudText Typo="Typo.h5">List of Events</MudText><br />
        <MudTable Items="@Items" Bordered="true" Striped="true" RowsPerPage="10">
            <ToolBarContent>
                <MudButton Variant="Variant.Filled" Color="Color.Primary"
                           Href="/events/new">ADD EVENT</MudButton>
            </ToolBarContent>
            <HeaderContent>
                <MudTh>ID</MudTh>
                <MudTh>Type</MudTh>
                <MudTh>Title</MudTh>
                <MudTh>From</MudTh>
                <MudTh>Hours</MudTh>
                <MudTh>Seats</MudTh>
                <MudTh>Fee</MudTh>
            </HeaderContent>
            <RowTemplate>
                <MudTd DataLabel="ID">@context.Id</MudTd>
                <MudTd DataLabel="Type">@context.Type</MudTd>
                <MudTd DataLabel="Title">@context.Title</MudTd>
                <MudTd DataLabel="From">@context.From</MudTd>
                <MudTd DataLabel="Hours">@context.Hours</MudTd>
                <MudTd DataLabel="Seats">@context.Seats</MudTd>
                <MudTd DataLabel="Fee">@context.Fee</MudTd>
            </RowTemplate>
            <PagerContent>
                <MudTablePager />
            </PagerContent>
        </MudTable>
    </MudItem>
</MudGrid>
@code {
    List<Event> Items { get; set; }
    protected override async Task OnInitializedAsync() {
        Items = await DataService.GetEventsAsync();
    }
}
```

Set **RowsPerPage** and add a **MudTablePager** to **PagerContent** to allow the user to change page and page options. To support filtering, add a string field or property to store the search text and add an input text field for the user to enter it, then assign a method to **Filter** property of **MudTable** to perform the filtering based on the search text.

## Add a property to store search text

```
string SearchText { get; set; } = string.Empty;
```

## Add MudTextField to input the search text

```
<ToolBarContent>
    <MudButton Variant="Variant.Filled" Color="Color.Primary"
        Href="/events/new">ADD EVENT</MudButton>
    <MudSpacer />
    <MudTextField @bind-Value="SearchText" Placeholder="Search"
        Adornment="Adornment.Start" AdornmentIcon="@Icons.Material.Filled.Search"
        IconSize="Size.Medium"></MudTextField>
</ToolBarContent>
```

## Assign filter method to table

```
<MudTable ... Filter="item=>string.Concat(item.Id,item.Type,item.Title).Contains(SearchText)">
                :
</MudTable>
```

To support sorting, add a **MudTableSortLabel** to any header column and assign the method to return the sorting value to the **SortBy** property. The default sort order is ascending but you can change it with the **InitialDirection** property.

## Enable sorting on ID

```
<MudTh>
    <MudTableSortLabel
        SortBy="new Func<Event,object>(x => x.Id)">ID
    </MudTableSortLabel>
</MudTh>
```

## Enable sorting on Title

```
<MudTh>
    <MudTableSortLabel
        SortBy="new Func<Event,object>(x => x.Title)">Title
    </MudTableSortLabel>
</MudTh>
```

## Enable sorting on Fee

```
<MudTh>
    <MudTableSortLabel InitialDirection="SortDirection.Descending"
        SortBy="new Func<Event,object>(x => x.Fee)">Fee
    </MudTableSortLabel>
</MudTh>
```

We have not finished demonstrating all the features of **MudTable** yet. We will have more examples with this component to show more features in later part of the course. Basically in this training module, we have shown the general operations in building of a Blazor application where we implement models and services and build components to present those models and use the services to process user input and generate the results.