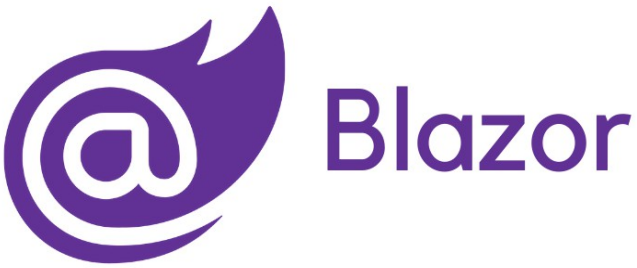

ASP520

Module 1

Building a Web Server Application with Blazor

Copyright ©
Symbolicon Systems
2008-2024

1 Blazor Server Application

1.1 Blazor Server Project

A Blazor Server web application runs completely on the web server and thus is more secure since no application logic code will be downloaded to the web browser, only the UI rendered on the web server will be sent to the web browser for display to the user. All user interactions and UI updates will be communicated between the web browser and the web server through SignalR, a real-time network communication system built on top of web sockets. Since all this is managed automatically by Blazor, developing a web server application is as simple as developing a desktop application using the full power of the C# programming language and feature complete .NET libraries including database access, data encryption, image processing, et cetera.

We will be using Visual Studio 2022 and .NET 8 for this course. To verify the version of the .NET SDKs that has been installed on your machine, enter the following console command using Windows Command Prompt. You can use Visual Studio 2022 Installer or visit dotnet.microsoft.com to install the latest stable version of .NET.

If a project is configured for HTTPS, your web server must have a certificate installed. A development certificate is provided by .NET to enable HTTPS for local development and testing. If it is not already installed, you will be prompted to install and trust it on your local IIS server on the first launch of the web application. You can also install the developer certificate from command prompt by running **dotnet** with **dev-certs https** command. It will inform you if the certificate is already present.

Checking .NET SDK versions & install certificate from console

```
C:\WINDOWS\system32> dotnet --list-sdks
C:\WINDOWS\system32> dotnet dev-certs https
```

Start up Visual Studio and create a new project by using the Blazor Web App template with the following details below.

Project information

```
Project name                    : Newrise
Project type                    : Blazor Web App
Location                        : <repository_folder>\src
Solution                        : Module1
Target Framework                : .NET 8.0
Authentication Type             : none
Interactive render mode         : Server
Interactive location            : Per page/component
Configure for HTTPS             : check
Do not use top-level statements : check
```

Double-click the **Newrise** project file in Solution Explorer to edit it. Set **Nullable** to **disable** as this setting is not conducive for a training environment. We do not wish to waste time changing or adding extra code to satisfy the compiler that we understand where and when object references can be or cannot be nullable.

Disable nullable warnings: Newrise.csproj

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>disable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

During development, you can use the .NET built-in web server Kestrel or external web server like IIS Express to test your web application. You can find their settings in the **launchSettings.json** file in Properties folder. Pay attention to the HTTP and HTTPS port numbers to be used by the web servers for your application. You can change the port numbers to make the application easier to access. I will be using **7000** for **https** and **5000** for **http** launch profiles for Kestrel. If you want to test with IIS Express use the **iisSettings** configuration section to configure the web server settings for the your application.

Launch settings: Properties\launchSettings.json

```
{
    "$schema": "http://json.schemastore.org/launchsettings.json",
    "iisSettings": {
        "windowsAuthentication": false,
        "anonymousAuthentication": true,
        "iisExpress": {
            "applicationUrl": "http://localhost:33183",
            "sslPort": 44374
        }
    },
    "profiles": {
        "http": {
            "commandName": "Project",
            "dotnetRunMessages": true,
            "launchBrowser": true,
            "applicationUrl": "http://localhost:5000",
            "environmentVariables": { "ASPNETCORE_ENVIRONMENT": "Development" }
        },
        "https": {
            "commandName": "Project",
            "dotnetRunMessages": true,
            "launchBrowser": true,
            "applicationUrl": "https://localhost:7000;http://localhost:5000",
            "environmentVariables": { "ASPNETCORE_ENVIRONMENT": "Development" }
        },
        "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": true,
            "environmentVariables": { "ASPNETCORE_ENVIRONMENT": "Development" }
        }
    }
}
```

1.2 Web Application Host

To run a .NET web application, you need to build a web application host. The following shows the code to build and run a very basic ASP.NET Core web application. Call the **CreateBuilder** method on **WebApplication** class to create a web application builder that comes with some built-in services and default settings. Additional services can be added at this point. These services can be used across the entire application. When all services has been added, call **Build** method to build the web application host then call the **Run** method on the host to run your application.

Building and running a .NET web application: Program.cs

```
public class Program {
    public static void Main(string[] args) {
        var builder = WebApplication.CreateBuilder(args);
        var app = builder.Build();
        app.Run();
    }
}
```

You need a web browser to test a web application. Visual Studio automatically detects what web browsers are installed on your local machine. You can then choose which of the browsers to use when you start the application from within Visual Studio. Always test your application against multiple web browsers to ensure consistent UI behavior across all browsers.

1.3 Middleware Components

If you execute your application at this moment, the web browser will show you a 404 page not found error. This is because no middleware components have been added to the application request pipeline. Web browsers communicate with web servers using a Request-Response model. For every HTTP request send by the browser, the server has to send back a HTTP response. The job of a middleware component is to process the request and return a response, if a component does not want to process a particular request, it can pass the request to the next component in the pipeline. The most basic middleware component provided by ASP.NET Core is the WelcomePage component. This should be the last component added as it will return a welcome page back as a response if no previous component in the pipeline has a response.

Using WelcomePage middleware component

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.UseWelcomePage();
app.Run();
```

Run the application again and you will now see a welcome page from ASP.NET Core. This is much better than getting a 404 error from your web browser and it proves that the your web application is running and accessible. A .NET web application is actually a console application so you can also check its console window to see if it's running or check for errors if it's not. Closing the console window will terminate the application if it's still running.

HTTPS should always be enabled for Internet-based applications. This will guarantee full privacy between the web client and the server. Web browsers will now warn users if they tried to access sites that do not support HTTPS. If your site do support HTTPS you can call **UseHsts** method to add a middleware to inform the client to only talk to the server through HTTPS. You should not enable this in development as browsers will cache the setting and you have to clear the cache if you need to revert back to HTTP during development for faster performance and easier debugging. Normally the value is cached for up to 30 days. Use properties on the application host **Environment** to check whether the application is running in development or production.

Middleware to inform client to communicate securely using HTTPS

```
var env = app.Environment;
if (env.IsDevelopment()) {
}
else {
    app.UseHsts();
}
```

Not every web browser supports HTTP Strict Transfer Security (HSTS), so you should also call **UseHttpsRedirection** to add a middleware to redirect the client to HTTPS if a HTTP request is received. The setting is not cached so safe to be used even within a development environment. Run the application and then try to use the HTTP URL and you will see that you will be redirected to HTTPS instead. You can still test HTTP using **http** launch profile instead but this would not work if HSTS is used. This is why it should not never be enabled in development environment.

Middleware to redirect HTTP to HTTPS

```
app.UseHttpsRedirection();
```

A web application does not completely run on the web server. Additional resource and code files have to be downloaded to the web browser for local access and execution. The **UseStaticFiles** method can add middleware to enable a client to download static files in a specific directory. This directory can be customized by passing in an optional **StaticFileOptions** instance that contains the directory name but the default directory will be **wwwroot**. Notice this directory may already be created for you if you selected any ASP.NET application project template. Add an **images** folder into the default web directory and then copy the following files that are provided by the instructor into that folder. Run the application and then try to use the URLs provided to access the files. You will see that you are not able to download those files by default.

Files copied into default web directory: \wwwroot\images

```
newrise-icon.png
newrise-logo.png
```

Accessing static files

```
https://localhost:7000/images/newrise-icon.png
https://localhost:7000/images/newrise-logo.png
```

Now call **UseStaticFiles** and run the web application again. You should now be able to access these files from the web content directory in your application. This directory should commonly be used to contain content files to be downloaded and used by your web browser like images, fonts, stylesheets and scripts. Do not use this folder for files that you do not want to be accessible directly by end users.

Middleware to enable static file access

```
app.UseStaticFiles();
```

Using the static files middleware is good enough to publish a static web site. However you would need to write code to implement a dynamic web site. Even though you can do this by creating custom middleware components, it would be too complicated and time-consuming to use this technique to implement a complete web site. This is why ASP.NET Core comes with web application frameworks to simplify the development of fully dynamic web sites. Currently there are 3 major web application frameworks for building a ASP.NET Core web application; MVC, Razor Pages and Blazor.

One major security problem with web applications is cross-site request forgery (CSRF) where users are fooled into using a fake web site that cross-posts requests to the real web site. This allow fake web sites to have full control over the real web site on behalf of the user to steal private data and perform transactions without the user knowing it. The following middleware does not allow cross-site posting. You can only post data to a site from a form on the same site. It generates an anti-forgery token in forms that will be checked when form data is posted to the server. This guarantees the the data is coming from the exact same form.

Middleware to stop CSRF attacks

```
app.UseAntiforgery();
```

1.4 Web Application Frameworks

Selecting the web application framework to use is determined by the services and the middleware components you add. The following shows the services and components for supporting MVC.

Services required for MVC

```
builder.Services.AddControllersWithViews();
```

Middleware components required for MVC

```
app.UseRouting();
app.UseEndpoints(endpoints => {
    endpoints.MapControllerRoute("default",
        "{controller=Home}/{action=Index}/{id?}");
});
```

The following shows required services and components for supporting Razor Pages. It is possible to use multiple frameworks by combining all the services and middleware components.

Services required for Razor pages

```
builder.Services.AddRazorPages();
```

Middleware components required for Razor pages

```
app.MapRazorPages();
```

The following shows the services and components required for Blazor server. Note the key term Razor Components represents Blazor. A Blazor application UI is constructed entirely from Razor components. Since this course is specifically targeted for Blazor, the following are the services and components that we will use.

Services required for Blazor server

```
builder.Services
    .AddRazorComponents()
    .AddInteractiveServerComponents();
```

Middleware components required for Blazor server

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

The application project we created has a default Razor component named **App.razor** to be used as the hosting page for the Blazor application. This is required in order to call **MapRazorComponents** method. It is considered the root component to load and start the Blazor server application. You can now remove the WelcomePage middleware as we no longer need it. Run the application and you should get a welcome message from the existing Razor components in the project.

1.5 Application Page

Blazor is considered a single-page web application. It only takes one request to start the application and the rest of the communication between the browser and server will go through SignalR. A Razor component is required to act as the application page to be send back to the browser on receiving a request. This should contain the HTML for a basic web page. The included script is required to setup SignalR communication between browser and server to update the page.

A basic HTML5 page: Components/App.razor

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <base href="/" />
    <title>Newrise</title>
</head>
<body>
    <h1>Welcome to Blazor</h1>
    <script src="_framework/blazor.web.js"></script>
</body>
</html>
```

1.6 Razor Components & Pages

The presentation of your application can be segregated across multiple components and each component can be used multiple times within other components. Location of the components doesn't matter. They can be created and used from another project. You can use the Razor Class Library project template to create libraries to specifically share Razor components across multiple applications. For the application project that we created, there is already a **Components** folder to put components though you can put your components anywhere. If there are too many components, you can always add additional folders and subfolders to organize them. Add the following components into the **Components** folder. Just add a H1 tag with the component name in each file.

Components to add

```
Home.razor
Events.razor
ContactUs.razor
```

You can easily embed one component into another component by using their names just as though they are HTML elements. The following shows the above components embedded into the **App** component. Run the application and see everything combined into a single page.

Embedding components into other components

```
<body>
    <h1>Welcome to Blazor</h1>
    <Home />
    <Events />
    <ContactUs />
    <script src="_framework/blazor.web.js"></script>
</body>
```

Sometimes you prefer each component to behave like a separate page. This is done in Blazor using a **Router** component to find a component based on it's route mappings and then use **RouteView** component to present the located component. You need to set the **AppAssembly** to which assembly to search for component pages. You can get the assembly by using **typeof** to get type information for a type in that assembly and then access it's **Assembly** property. This allows component pages to exist in another assembly. You also need to set the **Context** name on the **Router** to pass routing data to **RouteView** so it knows which component to present and what other parameters to pass to the component.

Implement component routing

```
<body>
    <Router AppAssembly="typeof(App).Assembly" Context="routeData">
        <Found>
            <RouteView RouteData="routeData" />
        </Found>
    </Router>
    <script src="_framework/blazor.web.js"></script>
</body>
```

You can now use the **page** directive to map your component to a route. You can have multiple route mappings per component. The **Router** locates your component based on the route. So any component can act as a page by using the **page** directive.

Route mappings for Home component

```
@page "/"
@page "/home"
<h1>Home</h1>
```

Route mapping for Events component

```
@page "/events"
<h1>Events</h1>
```

Route mapping for ContactUs component

```
@page "/contact"
<h1>Contact Us</h1>
```

Since the application base address is "/", the **Home** component becomes the default component to be presented by **RouteView** since it has a route mapping to "/". Test the following URLs to ensure the route mappings are correct.

Test route mappings

```
https://localhost:7000/home
https://localhost:7000/events
https://localhost:7000/contact
https://localhost:7000/
```

Since the **Router** can automatically locate components with **@page** route mappings, you can easily move them anywhere. Move all component pages to the **Pages** folder to organize them then rebuild and run the application again. The above URLs should still work.

1.7 Component Layout Page

Since you may have more than one component page, you may want multiple pages to share similar content like the same page header and page footer. This can be done by creating a special type of Razor component called as a layout component. If you have more than one layout, you can use a separate folder to organize them. For the project we created, there should already be **Layout** subfolder in **Components**. Add a Razor component named **DefaultLayout.razor** but to be a layout component you have to use the **inherits** directive to change the base class to **LayoutComponentBase**. The base class for normal components is **ComponentBase**. Then use the **Body** directive to specify the location of where the page content should be embedded in. Use **layout** directive to assign the same or different layout to each of the component pages.

Layout component: Components\Layout\DefaultLayout.razor

```
@inherits LayoutComponentBase
@Body
```

Add header and footer sections around page content

```
<header>
    <center>
        <img src="/images/newrise-logo.png" alt="Newrise Logo" />
    </center>
    <br />
</header>
@Body
<footer>
    <br />
    <center>
        <div>
            <a href="/home">Home</a> | 
            <a href="/events">Events</a> | 
            <a href="/contact">Contact Us</a>
        </div>
        <div><small>Copyright &copy Newrise 2024</small></div>
    </center>
</footer>
```

Assigning layout for Home page

```
@page "/"
@page "/home"
@layout Layout.DefaultLayout
<h1>Home</h1>
```

Assigning layout for Events page

```
@page "/events"
@layout Layout.DefaultLayout
<h1>Events</h1>
```

Assigning layout for ContactUs page

```
@page "/contact"
@layout Layout.DefaultLayout
<h1>Contact Us</h1>
```

Run the application and you should now see the page content embedded between the header and footer provided by the layout. You can now use the links in the footer to easily navigate between the pages. If all or most pages uses a single layout you can then set it as the default using **RouteView's DefaultLayout** property.

Setting default layout on RouteView

```
<RouteView RouteData="routeData" DefaultLayout="typeof(Layout.DefaultLayout)" />
```

1.8 Global Imports

You can create a special **_Imports** component to setup global namespace imports in Razor components so you do not have to use the full type names and you do not have to import namespaces per component. Add the **Layout** namespace as global import and removed it in the **RouteView**.

Global imports component: Components_Imports.razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using static Microsoft.AspNetCore.Components.Web.RenderMode
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using Newrise
@using Newrise.Components
@using Newrise.Components.Layout
```

Layout namespace reference/import no longer required: App.razor

```
<RouteView RouteData="routeData" DefaultLayout="typeof(DefaultLayout)" />
```

1.9 Blazor Components

Besides **Router** and **RouteView**, there are a number of other components provided with Blazor. **FocusOnNavigate** allows you to set focus to any element in a page after navigation by using CSS selectors. This is typically placed after **RouteView**.

FocusOnNavigate component

```
<RouteView RouteData="routeData" DefaultLayout="typeof(DefaultLayout)" />
<FocusOnNavigate RouteData="routeData" Selector="h1" />
```

To be able to change the HTML **head** section of a web page, add the **HeadOutlet** to the section as shown below. After that you can use **PageTitle** component to change the **title** element in any component.

HeadOutlet component

```
<head>
    <base href="/" />
    <meta charset="utf-8" />
    <title>Newrise</title>
    <HeadOutlet />
</head>
```

We will add a constant **AppName** field to **Program** class to contain the application's name so we can access it from multiple components.

Setting up the default application name

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

We will then modify **App** component to replace the static title with **AppName** field so that we do not need to hardcode the name across multiple components. If we need to change it, we do so in the **Program** class.

Remove static page title: App.razor

```
<head>
    <base href="/" />
    <meta charset="utf-8" />
    <HeadOutlet />
</head>
```

Remove the static title from the head section and then use **PageTitle** component so every page can have a different title but still use the same application name. You can easily embed code by using the Razor symbol (**@**) before the code. For more complex expressions, enclosed them within parentheses. Run the application and check page title after navigation between pages.

Changing page title in Home page

```
@page "/"
@page "/home"
<PageTitle>@(Program.AppName + "/Home")</PageTitle>
<h1>Home</h1>
```

Changing page title in Events page

```
@page "/events"
<PageTitle>@(Program.AppName + "/Events")</PageTitle>
<h1>Events</h1>
```

Changing page title in ContactUs page

```
@page "/contact"
<PageTitle>@(Program.AppName + "/Contact Us")</PageTitle>
<h1>Contact Us</h1>
```

You can replace normal HTML links with **NavLink** component which can detect if the link is already the active link by matching the current page route against the link to allow you to assign a separate CSS class for the active link by using the **ActiveClass** property.

Replacing HTML links with NavLink component: DefaultLayout.razor

```
<NavLink ActiveClass="active-link" href="/home">Home</NavLink> | 
<NavLink ActiveClass="active-link" href="/events">Events</NavLink> | 
<NavLink ActiveClass="active-link" href="/contact">Contact Us</NavLink>
```

When the project is created a stylesheet named **app.css** is already created containing some style classes used by Blazor. We will add the **active-link** style class we used for the above links. We will used the class to remove the underline and disable the link by using CSS **text-decoration** and **pointer-events** styles. We need to make sure that the stylesheet is also linked into the application in the HTML **head** section.

Style class to add: wwwroot/app.css

```
.active-link {
    text-decoration:none;
    pointer-events:none;
}
```

Link in the stylesheet: App.razor

```
<head>
    <base href="/" />
    <meta charset="utf-8" />
    <link rel="stylesheet" href="app.css" />
    <HeadOutlet />
</head>
```

Run the application and you should see that once you navigate to a specific page, it's matching link will look different and no longer interactable. Razor components usually have more features and simpler to use than using raw HTML elements. Generally the main activity in Blazor application development is creating and using components and the rest of time is implementing services.

2 MudBlazor UI Framework

2.1 UI Frameworks

Unless you are an expert at CSS, HTML and Javascript, you need a Web UI Framework to implement your front-end. Of course any UI framework built for Blazor applications is preferable where UI elements are Razor components. Following is a list of the free and open-source Blazor UI frameworks. For this training, 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 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 rel="stylesheet"
    href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" />
<link rel="stylesheet" href="_content/MudBlazor/MudBlazor.min.css" />
```

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 services = builder.Services;
services
    .AddRazorComponents()
    .AddInteractiveServerComponents();
services.AddMudServices();
```

You need to need to a **MudThemeProvider** to the page or its layout page to provide default or custom theme for MudBlazor components to use. To create custom themes, you can download and install **MudBlazor.ThemeManager** package. You can embed the theme manager as part of the application to perform live designing and editing of your application theme. You can now use MudBlazor components and services in your application.

Use default MudBlazor theme: Layout\DefaultLayout.razor

```
@inherits LayoutComponentBase
<MudThemeProvider />
    :
```

2.2 Page Layout

Use **MudLayout** component to define the layout of a page. In the layout you can use **AppBar** to create 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. Use **ClipMode** parameter to determine if the drawer overlays or always clipped within the bar. **MudMainContent** represent the main content area of a page. Lot of components have same parameters. 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.

A sample layout: Layout\DefaultLayout.razor

```
@inherits LayoutComponentBase
<MudThemeProvider />
<MudLayout>
    <MudAppBar Dense=true Elevation=2></MudAppBar>
    <MudDrawer Open=true ClipMode=DrawerClipMode.Always Elevation=1></MudDrawer>
    <MudMainContent>@Body</MudMainContent>
</MudLayout>
```

Some sample MudBlazor layouts

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

Following shows more MudBlazor components; **MudIconButton** to add icon buttons, **MudText** to display general text and **MudSpacer** to help space-out components. In our **AppBar** we will add two icon buttons, one to the left and the other on the right and the application name is displayed in the center.

Components in the application bar

```
<MudIconButton Icon=@Icons.Material.Filled.Menu Color=Color.Inherit />
<MudSpacer />
<MudText Typo=Typo.h5>@Program.AppName</MudText>
<MudSpacer />
<MudIconButton Icon=@Icons.Material.Filled.MoreVert Color=Color.Inherit />
```

There are many different ways to set component parameters. We can use the shortest method as long as Blazor can understand the values assigned to the parameters. For example, it may confuse between an enum member and a string.

Different ways to assigning values to parameters

```
Dense               existence of bool parameter defaults to true.
Dense=true          since parameter type is bool, true is bool value and not string.
Dense="true"        since parameter type is bool, true is bool value and not string.
Dense=@true         a simple programmatic value.
Dense=@(true)       a complex programmatic expression.
Dense="@true"       a wrapped simple programmatic value.
Dense="@(true)"     a wrapped complex programmatic expression.
```

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**. Use **MudImage** to display an image in the drawer, slightly scaled down and centered using **ObjectFit** and **ObjectPosition** parameters followed by a **MudNavMenu**. Relaunch the application and test out the links in the navigation menu.

Adding an image and navigation menu to drawer

```
<MudImage Src="/images/newrise-logo.png" Alt="Newrise Logo"
    ObjectFit=ObjectFit.ScaleDown ObjectPosition=ObjectPosition.Center />
<MudNavMenu>
    <MudNavLink Href="/" Match=@NavLinkMatch.All>Home</MudNavLink>
    <MudNavLink Href="/events">Events</MudNavLink>
    <MudNavLink Href="/contact">Contact Us</MudNavLink>
    <MudNavGroup Title="MudBlazor" Expanded=true>
        <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>
```

2.3 Rendermode & Interactivity

When you launch the application, notice that the **MudNavGroup** cannot be collapsed. This is because the rendermode for this component has not been set. Remember that when we created the project there is an Interactive location option initially set to Per page/component. We can then decide which components generates static content and which components are interactive. Components can also be interactive on the server or client. To set the default for the entire application, you need to set it for each Razor component you used in the application page. This requires moving the **Router** into a separate Razor component. Add a **Routes** component to contain the router.

Separate component for routing: Components\Routes.razor

```
<Router AppAssembly="typeof(App).Assembly" Context="routeData">
    <Found>
        <RouteView RouteData="routeData" DefaultLayout="typeof(DefaultLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

Update application page to use the above component and then use **@rendermode** on all components in the page. Since this is a Blazor server application we want the UI to run on the server so set rendermode to **InteractiveServer**. This is the same as if the Interactive location option was set to Global instead during project creation.

Setting rendermode to InteractiveServer: App.razor

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

Let us implement more interactivity. Add the following code block to the layout to add a **drawerOpened** field and a **ToggleDrawer** method that toggles the field between **false** and **true** each time it is called.

Add fields and methods to the layout component: DefaultLayout.razor

```
@code {
    bool drawerOpened = false;
    void ToggleDrawer() {
        drawerOpened = !drawerOpened;
    }
}
```

Assign the method to the **OnClick** event of the first **MudIconButton**. When the user clicks on the button, the code would be executed. Then assign **drawerOpened** field to the **Open** parameter of the **MudDrawer**. You can now control the opening and the closing of the drawer by clicking the icon button. Since the default value is **false** it is initially closed.

Attaching event handler to OnClick event

```
<MudIconButton
    Icon=@Icons.Material.Filled.Menu
    Color=Color.Inherit OnClick=ToggleDrawer />
```

Assigning field or property to component parameter

```
<MudDrawer Open=drawerOpened ClipMode=DrawerClipMode.Always Elevation=1>
```

This above assignment can be regarded as a one-way binding. The drawer can access the value of the field but cannot change it. We will now configure the drawer to open only temporarily using the **Variant** parameter. An overlay will cover the page content when the drawer is opened and should close together with the drawer. This can be set using the **OverlayAutoClose** parameter. Launch the application and you should see that the drawer will automatically close after a successful navigation to another page. However since it cannot change the **drawerOpened** field, the page still thinks that it is opened so you have to click on the icon button twice to re-open it. Use a two-way binding to attach the field to the drawer. This can be done using the **@bind** directive on the parameter assignment. This binding would allow the drawer to update the field or property when the drawer is closed.

Using two-way binding

```
<MudDrawer @bind-Open=drawerOpened ClipMode=DrawerClipMode.Always Elevation=1
    Variant=DrawerVariant.Temporary OverlayAutoClose>
```

2.4 Using Services

Add an **Event** sub-folder to **Pages** folder and then add a **Create** Razor component in that sub-folder with the following initial content. Then update the **Events** page to add a button to navigate to this new page.

Create event component page: Components\Pages\Event\Create.razor

```
@page "/events/create"
<MudText Typo=Typo.h5>New Event</MudText>
```

Button to navigate to above page: Events.razor

```
@page "/events"
<PageTitle>@(Program.AppName + "/Events")</PageTitle>
<MudText Typo=Typo.h5>Events</MudText>
<MudButton Variant=Variant.Filled Color=Color.Primary
    Href="/events/create">ADD EVENT</MudButton>
```

Sometimes you want or need to perform navigation from component code. To do this, you need to access to the **NavigationManager** service. This service is automatically registered by ASP.NET Core when you create the web application builder.

Standard services are automatically registered by the builder

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

You can use any registered service in a Razor component through property injection. Use the **inject** directive and identity the service that you want to use by class name or interface name used for registering the service with the application host builder and provide the name of the property to assign the service object to. This property is then added to your component and you can use it to access the service. You can use the same or different name for the property. The following code shows how to access the **NavigationManager** service and call **NavigateTo** method navigate to a different page.

Using NavigationManager service: Pages\Events.razor

```
@page "/events"
@inject NavigationManager Navigation
<PageTitle>@(Program.AppName + "/Events")</PageTitle>
<MudText Typo=Typo.h5>Events</MudText>
<MudButton Variant=Variant.Filled Color=Color.Primary
    OnClick=@(e=>Navigation.NavigateTo("/events/create")
    Href="/events/create">ADD EVENT</MudButton>
```

Here we have completed the basics of building a Blazor server application and using the MudBlazor UI framework to build the application user interface. We will look into more advanced features of Blazor and MudBlazor in the following modules.