**Blazor Fundemelets**

**What is Blazor?**

Blazor apps are composed of reusable web UI components built using C#, HTML, and CSS. With Blazor, developers can build client and server code with C#. Using C# for all code simplifies sharing data between the front end and back end, code reuse to accelerate development, and maintenance.

**You can use Blazor to generate:**

* Server-side code that handles UI interactions over a WebSocket connection.
* A client-side web app that runs directly in the browser via WebAssembly.

**What is WebAssembly?**

WebAssembly (WASM) is an open binary standard. It defines a portable code format for programs designed to run in web browsers. WebAssembly is a textual assembly language with a compact binary format for fast downloads and near-native performance.

WebAssembly provides a compilation target for languages such as C, C++, and Rust. It's designed to run alongside JavaScript so that both work together.

**What is Blazor WebAssembly?**

It's a single-page app framework that uses the WebAssembly open standards without requiring plug-ins or code generation.

.NET code executed via WebAssembly in a browser runs in the browser's JavaScript sandbox. The code includes all the security and protection that the sandbox provides.

Blazor uses a .NET runtime compiled to a WebAssembly module that is downloaded with an app. The module can execute .NET Standard code included in a Blazor app.

A Blazor WebAssembly app is restricted to the capabilities of the browser that executes the app, but the app can access full browser functionality via JavaScript interop.

**What is Blazor Server?**

Blazor Server provides support for hosting Razor components on the server in an ASP.NET Core app. UI updates are handled over a SignalR connection.

The runtime stays on the server and handles:

* Executing the app's C# code.
* Sending UI events from the browser to the server.
* Applying UI updates to the rendered components that are sent back by the server.
* The connection Blazor Server uses to communicate with the browser is also used to handle JavaScript interop calls.
* **blazor server app project**
* **Program.cs is the entry point for the app that starts the server and where you configure the app services and middleware.**
* **App.razor is the root component for the app.**
* **The Pages directory contains some example web pages for the app.**
* **BlazorApp.csproj defines the app project and its dependencies and can be viewed by double-clicking the BlazorApp project node in the Solution Explorer.**
* **The launchSettings.json file inside the Properties directory defines different profile settings for the local development environment. A port number is automatically assigned at project creation and saved on this file.**

**basic**

* @page- define the api url of this page(Route) . write in top of the page- first line.

example: @page "/counter"

* PageTitle- the title of the page tab- like in react.
* @code {...}- what u write here is the c# code that involve in the ui page- variables and functions declarations can be involve in the ui(html) code by add @ first.

example:

@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {

private int currentCount = 0;

private void IncrementCount()

{

currentCount++;

}

}

**Add a component**

Each of the .razor files defines a UI component that can be reused.

you can add a component to page by adding a <ComponentName /> element

**Modify a component**

Component parameters are specified using attributes or child content, which allow you to set properties on the child component.

Add a public property with a [Parameter] attribute.

**example:**

@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {

private int currentCount = 0;

[Parameter]

public int IncrementAmount { get; set; } = 1;

private void IncrementCount()

{

currentCount += IncrementAmount;

}

}

@page "/"

<h1>Hello, world!</h1>

<Counter IncrementAmount="10" />

**Marks**

* **@page-** pages can have multiply url addresses that will lead to same page. blazor can treat this.
* passing parameter in the url address: /{parameterName:type}
* **OnParametersSet()**-get and put the parameter in the c# code will be in function of OnParametersSet().
* **@layout**- you can modify the page layout by adding @layout LayoutName

**example:**

@page "/counter"

@page "/counter/{startingValue:int}"

@page "/dosamestuff"

@layout MainLayout

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {

private int currentCount = 0;

[Parameter]

public int StartingValue { get; set; } = 0;

[Parameter]

public int IncrementAmount { get; set; } = 1;

protected override void OnParametersSet()

{

currentCount = StartingValue;

base.OnParametersSet();

}

private void IncrementCount()

{

currentCount += IncrementAmount;

}

}

**Explaining App.razor and MainLayout.razor and host.cshtml files**

* **App.razor**

<Router AppAssembly="@typeof(App).Assembly">

<Found Context="routeData">

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

<FocusOnNavigate RouteData="@routeData" Selector="h1" />

</Found>

<NotFound>

<PageTitle>Not found</PageTitle>

<LayoutView Layout="@typeof(MainLayout)">

<p role="alert">Sorry, there's nothing at this address.</p>

</LayoutView>

</NotFound>

</Router>

<Router> is a component that is responsible for mapping URLs to corresponding components/views in the application. The AppAssembly parameter specifies the assembly where the application's components are defined.

Inside the <Router> component, there are two child components: <Found> and <NotFound>. <Found> is used to specify what to render when the requested URL matches a registered route (@Page), while <NotFound> is used to specify what to render when the requested URL does not match any registered route.

Inside <Found>, there are two child components: <RouteView> and <FocusOnNavigate>. <RouteView> is a component that renders the component/view associated with the matched route. RouteData specifies the data associated with the matched route. DefaultLayout specifies the layout component to be used for the component/view being rendered.

<FocusOnNavigate> is used to set focus on a specified element on navigation to a new page. In this case, it sets focus on the first <h1> element on navigation.

Inside <NotFound>, there are two child components: <PageTitle> and <LayoutView>. <PageTitle> specifies the title of the page. <LayoutView> is used to specify the layout component to be used for the page.

We can modify the default layout view, message of not found page and many more in this file…

* **MainLayout.razor**

@inherits LayoutComponentBase

<PageTitle>BlazorApp</PageTitle>

<div class="page">

<div class="sidebar">

<NavMenu />

</div>

<main>

<div class="top-row px-4">

<a href="https://docs.microsoft.com/aspnet/" target="\_blank">About</a>

</div>

<article class="content px-4">

@Body

</article>

</main>

</div>

A layout component is a component that defines the overall structure of a page in a Blazor application. It typically contains a header, a footer, and a content section.

@inherits LayoutComponentBase specifies that the component inherits from LayoutComponentBase, which is a base class that provides functionality for layout components.

<PageTitle> is a component that sets the title of the page.

<NavMenu> is a component of the menu- default component- file NavMenu.razor

The rest of the code defines the structure of the layout component. It contains a sidebar, a main section, and an article section that contains the page content. The @Body directive is used to render the content of the page.

We can modify the layout as we want…

* **\_Host.cshtml**

The host.cshtml file is the main HTML file that is used to host a Blazor application on the server. It contains the basic structure of the HTML document, links to CSS files, and JavaScript files required to run the application.

Here is a breakdown of the different parts of the file:

* @page "/": This directive sets the default route for the application to the root URL ("/").
* @using Microsoft.AspNetCore.Components.Web: This directive imports the namespace that is required for Blazor components.
* @namespace BlazorApp.Pages: This directive sets the namespace for the Razor components in the Pages folder of the Blazor application.
* @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers: This directive adds the Microsoft.AspNetCore.Mvc.TagHelpers tag helpers that are used to generate HTML elements for Blazor components.
* The HTML document structure contains a head and a body section.
* <base href="~/" />: This tag specifies the base URL for the application, which is the root directory of the server.
* The next few lines of code add links to CSS files required for the application to run.
* <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />: This tag specifies the HeadOutlet component that will be used to render the head section of the HTML document. The render-mode attribute specifies that the component will be pre-rendered on the server.
* <component type="typeof(App)" render-mode="ServerPrerendered" />: This tag specifies the App component that will be used as the root component of the Blazor application. The render-mode attribute specifies that the component will be pre-rendered on the server.
* The next few lines of code add an error UI that is displayed if an error occurs in the application. The UI contains a reload button and a dismiss button.
* <script src="\_framework/blazor.server.js"></script>: This tag specifies the JavaScript file that is required to run the Blazor application on the server.

**What is Razor?**

Markup syntax that uses HTML and C# for writing UI components of Blazor web apps.

Razor is based on ASP.NET and designed for creating web apps.

**What are Razor components?**

A Razor file defines components that make up a portion of the app UI. Components in Blazor are analogous to user controls in ASP.NET Web Forms.

At compile time, each Razor component is built into a .NET class. The class includes common UI elements like state, rendering logic, lifecycle methods, and event handlers.

**Data binding and events**

You can add C# code in separate .cs files or inline in your Razor components.

**C# code-behind in separate files**

You can add C# files directly to your app project, as with other .NET projects. Commonly called code-behind, this technique uses separate code files to store app logic. Separate files are a great strategy when your business logic is complex, long, or has multiple classes.

For simple logic, you don't always need to create new .cs files.

**C# inline in components**

It's a common practice to mix HTML and C# in a single Razor component file. To add code into a Razor file, you'll use directives.

**What are Razor directives?**

Razor directives are component markup used to add C# inline with HTML. With directives, developers can define single statements, methods, or larger code blocks.

**Code directives**

You can use @() to add a C# statement inline with HTML. If you require more code, use the @code directive to add multiple statements, enclosed by parentheses.

You can also add an @functions section to the template for methods and properties. They're added to the top of the generated class, where the document can reference them.

**Page directive**

The @Page directive is special markup that identifies a component as a page. You can use this directive to specify a route. The route maps to an attribute route that the Blazor engine recognizes to register and access the page.

**Razor data binding**

Within Razor components, you can data bind HTML elements to C# fields, properties, and Razor expression values. Data binding allows two-way synchronization between HTML and Microsoft .NET.

Data is pushed from HTML to .NET when a component is rendered. Components render themselves after event-handler code executes, which is why property updates are reflected in the UI immediately after an event handler is triggered.

You can use @bind markup to bind a C# variable to an HTML object. You'll define the C# variable by name as a string in the HTML.

**example:**

**separate cs file**

public class TodoItem

{

public string? Title { get; set; }

public bool IsDone { get; set; } = false;

}

**separate razor file**

@page "/todo"

<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>

<ul>

@foreach (var todo in todos)

{

<li>

<input type="checkbox" @bind="todo.IsDone" />

<input @bind="todo.Title" />

</li>

}

</ul>

<input placeholder="Something todo" @bind="newTodo" />

<button @onclick="AddTodo">Add todo</button>

@code {

private List<TodoItem> todos = new();

private string? newTodo;

private void AddTodo()

{

if (!string.IsNullOrWhiteSpace(newTodo))

{

todos.Add(new TodoItem { Title = newTodo });

newTodo = string.Empty;

}}}

**@bind options in Blazor:**

* @bind - binds the value of an input element to a property or field in the component
* @bind:event - specifies the event that triggers the two-way binding update (default is oninput)
* @bind:format - specifies a format string for formatting the value before it is displayed
* @bind:format-value - specifies a function that formats the value before it is displayed
* @bind:parse-value - specifies a function that parses the value before it is assigned to the property or field
* @bind:attributes - specifies additional attributes to add to the input element
* @bind:suppress-round-trip-formatting - suppresses the formatting of the value before it is sent to the server (useful for large values)

These options can be used with various input elements, such as <input>, <select>, and <textarea>, and can help customize the two-way binding behavior in Blazor.

**list of all the available event options for the @bind directive in Blazor:**

* oninput (default) - The event is triggered when the value of the input element changes.
* onchange - The event is triggered when the input element loses focus and its value has been changed since gaining focus.
* onblur - The event is triggered when the input element loses focus.
* onkeyup - The event is triggered when a key is released while the input element has focus.
* onkeydown - The event is triggered when a key is pressed while the input element has focus.
* onkeypress - The event is triggered when a key is pressed and released while the input element has focus.
* oninput:debounce(ms) - The event is triggered when the value of the input element changes, but only after a specified amount of time (in milliseconds) has passed without any other changes. This option can help reduce the number of server requests sent during user input.
* oninput:delay(ms) - The event is triggered when the value of the input element changes, but only after a specified amount of time (in milliseconds) has passed since the last change. This option can help reduce the number of server requests sent during user input.
* oninput:preventDefault - Prevents the default behavior of the input element on the oninput event. This can be useful for preventing unwanted updates during user input.

**Getting Data in blazor**

@page "/fetchdata"

@using BlazorApp.Data

@inject WeatherForecastService ForecastService

<PageTitle>Weather forecast</PageTitle>

@if (forecasts == null)

{

<p><em>Loading...</em></p>

}

else

{

<table class="table">

<thead>

<tr>

<th>Date</th>

<th>Temp. (C)</th>

<th>Temp. (F)</th>

<th>Summary</th>

</tr>

</thead>

<tbody>

@foreach (var forecast in forecasts)

{

<tr>

<td>@forecast.Date.ToShortDateString()</td>

<td>@forecast.TemperatureC</td>

<td>@forecast.TemperatureF</td>

<td>@forecast.Summary</td>

</tr>

}

</tbody>

</table>

}

@code {

private WeatherForecast[]? forecasts;

protected override async Task OnInitializedAsync()

{

forecasts = await ForecastService.GetForecastAsync(DateOnly.FromDateTime(DateTime.Now));

}

}

**in program.cs file.**

builder.Services.AddSingleton<WeatherForecastService>();

**in WeatherForecastService.cs file**

public class WeatherForecastService

{

private static readonly string[] Summaries = new[]

{

"Freezing", "Bracing", "Chilly", "Cool", "Mild","Warm", "Balmy", "Hot", "Sweltering", "Scorching"

};

public Task<WeatherForecast[]> GetForecastAsync(DateOnly startDate)

{

return Task.FromResult(Enumerable.Range(1, 5).Select(index => new WeatherForecast

{

Date = startDate.AddDays(index),

TemperatureC = Random.Shared.Next(-20, 55),

Summary =Summaries[Random.Shared.Next(Summaries.Length)]

}).ToArray());

}

}

This code is a Blazor component that demonstrates how to fetch data from a service and display it in a table. The component is defined in the file with the URL "/fetchdata".

The component uses the WeatherForecastService to get an array of WeatherForecast objects. The WeatherForecastService is defined in a separate file called WeatherForecastService.cs, and it generates random weather forecast data.

In the component, the @page directive sets the URL for the component. The **@using directive** imports the BlazorApp.Data namespace, and the **@inject directive** injects the WeatherForecastService into the component.

The @if statement displays a "Loading..." message if the forecasts array is null. Otherwise, a table is displayed with columns for the forecast data.

The @foreach statement iterates over each WeatherForecast object in the forecasts array and displays the values of its properties in the table rows.

The @code block contains the component's code. It initializes the forecasts array in the **OnInitializedAsync** method by calling the GetForecastAsync method of the WeatherForecastService. The method returns a Task<WeatherForecast[]>, which is awaited before assigning its result to the forecasts array.

In the program.cs file, the WeatherForecastService is added to the dependency injection container using the **AddSingleton** method. This makes it available for injection into other components and services in the application.

**More accurate- using Http Requests**

**adding controller in separate cs file.**

using Microsoft.AspNetCore.Mvc;

using BlazorApp.Data;

using Microsoft.AspNetCore.Http;

using System;

using System.Linq;

using System.Collections.Generic;

using System.Threading.Tasks;

using System.Runtime.CompilerServices;

namespace BlazorApp

{

[Route("api/[controller]")]

[ApiController]

public class ForecastController : ControllerBase

{

public async Task<ActionResult<IEnumerable<WeatherForecast>>> GetForecastAsync()

{

var svc = new WeatherForecastService();

return new OkObjectResult(await svc.GetForecastAsync(DateOnly.FromDateTime(DateTime.Now)));

}

}

}

**make changes in the razor file**

@page "/fetchdata"

@using BlazorApp.Data

@inject HttpClient Http

html code…

@code {

private IEnumerable<WeatherForecast> forecasts;

protected override async Task OnInitializedAsync()

{

forecasts = await Http.GetFromJsonAsync<IEnumerable<WeatherForecast>> ("http://localhost:5117/api/forecast");

}

}

**add service of HttpClient in the program file**

builder.Services.AddHttpClient();

**Summary**

1. **@using directive: The @using directive is a Razor syntax feature in ASP.NET Core that allows you to specify namespaces to be used within a Razor view or page. By using the @using directive, you can simplify your Razor code by avoiding the need to fully qualify the names of classes, interfaces, and other types that are defined in the specified namespace.**
2. **@inject directive: The @inject directive is another Razor syntax feature that allows you to declare a dependency injection service within a Razor view or page. When you use the @inject directive, you are able to access the services provided by the dependency injection container within your Razor code. This can be especially useful when you need to retrieve data or perform other operations that require services or dependencies.**
3. **OnInitializedAsync: OnInitializedAsync is a method that is defined in the Blazor component base class. It is called when a Blazor component is initialized, and can be used to perform any initialization logic that is required by the component. Because it is an asynchronous method, you can use it to perform long-running initialization tasks without blocking the UI thread.**
4. **AddSingleton method: AddSingleton is a method that is defined in the ASP.NET Core dependency injection container. It is used to register a singleton service with the container. A singleton service is a service that is instantiated once and then shared across all consumers that request it. This can be useful when you have a service that needs to maintain state or other information across multiple requests.**

**Blazor Events CallBack**

**Blazor allows you to add event handlers to components and trigger them in response to user interactions, such as button clicks or input changes. Blazor uses the C#** EventCallback **type to handle events and provide two-way data binding between components and their parent components or pages.**

**example of a button component that raises a custom event when clicked:**

**<button class="@CssClass" @onclick="OnClick">@ButtonText</button> @code { [Parameter] public string CssClass { get; set; } [Parameter] public string ButtonText { get; set; } [Parameter] public EventCallback OnClick { get; set; } private async Task OnClickHandler() { await OnClick.InvokeAsync(null); } }**

**In this example, we're defining a button component that takes in parameters for the button's CSS class, button text, and a callback function for the button click event. The** EventCallback **type allows us to raise custom events and pass data back to the parent component or page.**

**To use this component in a parent component or page, we can add an event handler:**

**<CustomButton CssClass="btn-primary" ButtonText="Click me" OnClick="@(() => HandleButtonClick())" /> @code { private async Task HandleButtonClick() { // Do something when the button is clicked } }**

**In this example, we're using the** CustomButton **component and passing in parameters for the button's CSS class and text, as well as a callback function for the** OnClick **event. The callback function is defined using a lambda expression that calls the** HandleButtonClick **method when the button is clicked.**

**Blazor also supports built-in event handlers for common UI interactions, such as button clicks, input changes, and form submissions. Here's an example of how to handle the** input **event in a text input component:**

**<input type="text" @bind-value="@TextInputValue" @bind-value:event="oninput" /> @code { private string TextInputValue { get; set; } }**

**In this example, we're using the** @bind-value **directive to two-way bind the value of the text input to the** TextInputValue **property in the component's C# code. We're also using the** @bind-value:event **directive to specify that the** oninput **event should trigger the binding. This means that every time the user types something in the text input, the** TextInputValue **property will be updated automatically.**

**Parameters and Sharing Data**

* **CascadingValue, CascadingParameter**

**In Blazor, the** CascadingValue **component is used to pass a value down the component tree to child components, regardless of their depth. This component creates a cascading value that can be consumed by child components, and when the value changes, it updates all the dependent components in the tree.**

**The** CascadingValue **component takes two parameters:** Value **and** ChildContent**. The** Value **parameter is the value to be cascaded down to the child components, and the** ChildContent **parameter is the content that will be rendered where the** CascadingValue **component is used.**

**Here's an example of how to use the** CascadingValue **component in Blazor:**

**<CascadingValue Value="5"> <MyChildComponent /> </CascadingValue>**

**In this example, we're passing the value** 5 **down to the** MyChildComponent **component using the** CascadingValue **component. The** MyChildComponent **component can access this value by using the** CascadingParameter **attribute:**

**<div> <h1>@Value</h1> </div>**

**@code {**

**[CascadingParameter(Name=”Value”)]**

**protected int Value { get; set; }**

**}**

**Managing User State**

Managing user state in Blazor involves keeping track of user-specific data, such as authentication status, session data, and user preferences, throughout the user's interaction with the application. There are several ways to manage user state.

**We can do it by injecting custom class state**

1. build the class that need to represent the data. cs file. - we need to give meaningful name and State in ending.
2. register the state in the program: builder.Services.AddScoped<NameOfStateClass>.
3. injecting to the components/pages we want to use.
4. use the injected directive parameter we declared.

public class UserState {

public bool IsAuthenticated { get; set; }

public string UserName { get; set; }

public string Email { get; set; }

}

builder.Services.AddScoped<UserState>();

@inject UserState userState

@if (userState.IsAuthenticated)

{ <h1>Welcome, @userState.UserName!</h1> <button @onclick="Logout">Logout</button> } else { <h1>Login</h1> <input type="text" @bind="userName" placeholder="Username" /><br /> <input type="password" @bind="password" placeholder="Password" /><br /> <button @onclick="Login">Login</button> } @code { private string userName = ""; private string password = ""; private void Login() { if (userName == "admin" && password == "password") { userState.IsAuthenticated = true; userState.UserName = userName; } else { // Show error message } } private void Logout() { userState.IsAuthenticated = false; userState.UserName = ""; } }

In this example, we're injecting the UserState class into our component using the @inject directive. We can then access the state properties of IsAuthenticated and UserName directly from the injected userState object. When the user logs in, we set the IsAuthenticated and UserName properties of the userState object, and when the user logs out, we reset those properties.

**another way- cockies:**

One common approach to managing user state in Blazor is to use a combination of server-side storage and cookies. The server-side storage can be used to store large amounts of user data, while cookies can be used to store smaller pieces of data that need to be accessed frequently.

@page "/"

@if (authenticated)

{

<h1>Welcome, @userName!</h1>

<button @onclick="Logout">Logout</button>

}

else

{

<h1>Login</h1>

<input type="text" @bind="userName" placeholder="Username" /><br />

<input type="password" @bind="password" placeholder="Password" /><br />

<button @onclick="Login">Login</button>

}

@code {

private bool authenticated = false;

private string userName = "";

private string password = "";

protected override void OnInitialized()

{

var cookie = GetCookie("auth");

if (cookie != null)

{

authenticated = true;

userName = cookie["userName"];

}

}

private void Login()

{

// Authenticate user

if (userName == "admin" && password == "password")

{

authenticated = true;

SetCookie("auth", new Dictionary<string, string> { { "userName", userName } }, TimeSpan.FromDays(1));

}

else

{

// Show error message

}

}

private void Logout()

{

authenticated = false;

RemoveCookie("auth");

}

private void SetCookie(string name, IDictionary<string, string> values, TimeSpan expires)

{

var options = new CookieOptions { Expires = DateTime.UtcNow.Add(expires) };

foreach (var kvp in values)

{

options.Append(key, value);

}

Response.Cookies.Append(name, options);

}

private IDictionary<string, string> GetCookie(string name)

{

Request.Cookies.TryGetValue(name, out string cookieValue);

if (cookieValue != null)

{

return JsonConvert.DeserializeObject<IDictionary<string, string>>(cookieValue);

}

return null;

}

private void RemoveCookie(string name)

{

Response.Cookies.Delete(name);

}

}

In this example, we're using cookies to store the user's authentication status and username. When the page is initialized, we check if there is an auth cookie and use its value to set the authenticated and userName fields. If the user logs in successfully, we set the authenticated field to true and create a new cookie with a one-day expiration. If the user logs out, we set the authenticated field to false and delete the auth cookie.

**Forms and Capturing User Data**

[**more about forms and input components**](https://learn.microsoft.com/en-us/aspnet/core/blazor/forms-and-input-components?WT.mc_id=academic-33200-cxa&view=aspnetcore-7.0)

**Blazor has a comprehensive system for data entry forms and validation. We'll now use this to apply the same DataAnnotations rules on the client that are already being enforced on the server.**

**The way Blazor's forms and validation system works is based around something called an EditContext. An EditContext tracks the state of an editing process, so it knows which fields have been modified, what data has been entered, and whether or not the fields are valid. Various built-in UI components hook into the EditContext both to read its state and to write to its state.**

**One of the most important built-in UI components for data entry is the EditForm. This renders as an HTML <form> tag, but also sets up an EditContext to track what's going on inside the form.**

**<div class="main">**

**<EditForm Model="OrderState.Order.DeliveryAddress" OnValidSubmit="PlaceOrder">**

**<div class="checkout-cols">**

**<div class="checkout-order-details">**

**<h4>Review order</h4>**

**<OrderReview Order="OrderState.Order" />**

**</div>**

**<div class="checkout-delivery-address">**

**<h4>Deliver to...</h4>**

**<AddressEditor Address="OrderState.Order.DeliveryAddress" />**

**</div>**

**</div>**

**<button type="submit" class="checkout-button btn btn-warning" disabled="@isSubmitting">**

**Place order**

**</button>**

**<DataAnnotationsValidator />**

**@\*<ValidationSummary />\*@**

**</EditForm>**

**</div>**

**AddressEditor.razor file:**

**<div class="form-field">**

**<label>Name:</label>**

**<div>**

**<InputText @bind-Value="Address.Name" />**

**<ValidationMessage For="@(() => Address.Name)" />**

**</div>**

**</div>**

**<div class="form-field">**

**<label>Line 1:</label>**

**<div>**

**<InputText @bind-Value="Address.Line1" />**

**<ValidationMessage For="@(() => Address.Line)" />**

**</div>**

**</div>**

**<div class="form-field">**

**<label>City:</label>**

**<div>**

**<InputText @bind-Value="Address.City" />**

**<ValidationMessage For="@(() => Address.City)" />**

**</div>**

**</div>**

**@code {**

**[Parameter] public Address Address { get; set; }**

**}**

**This code is an example of a Blazor component that displays a form for editing an address. The component uses the** @bind-Value **directive to two-way bind the values of the form fields to properties of the** Address **object that is passed in as a parameter.**

**The component contains several form fields for editing the name, address line, city of the address. Each form field is wrapped in a** div **element with a** form-field **class for styling purposes.**

**The** ValidationMessage **component is used to display validation error messages for each form field. The** For **attribute of the** ValidationMessage **component is used to specify which property of the** Address **object the validation message is for.**

**The** @code **block at the bottom of the component contains a** Parameter **attribute for the** Address **parameter, which allows the component to receive an** Address **object from its parent component or page.**

**The** EditForm **component is a built-in component in Blazor that provides support for model binding, validation, and submission of forms. It wraps around a form element and provides a way to submit the form data to a server-side method or to handle the form data in the component itself.**

**In this code, the** Model **attribute of the** EditForm **component is set to** OrderState.Order.DeliveryAddress**, which specifies the model that the form data will be bound to. The** OnValidSubmit **attribute is set to the name of a method (**PlaceOrder**) that will be called when the form is submitted and is valid.- when the user click on button which is type of sumbit**

**When the user submits the form, the** PlaceOrder **method will be called. The method can access the form data through the** Model **property of the** EditForm **component, which contains the bound data. The method can then process the form data and take any necessary actions, such as saving the data to a database or displaying a confirmation message to the user.**

**Validating Form Data with Blazor**

[**more about valiadating**](https://learn.microsoft.com/en-us/aspnet/core/blazor/forms-and-input-components?view=aspnetcore-5.0#basic-validation-1)

public class Address

{

public int Id { get; set; }

[Required, MaxLength(100)]

public string Name { get; set; }

[Required, MaxLength(100)]

public string Line1 { get; set; }

[Required(ErrorMessage = "Customize your message?"), MaxLength(50)]

public string City { get; set; }

}

**<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">**

**<DataAnnotationsValidator />**

**<div class="form-group">**

**<label for="name">Name</label>**

**<InputText id="name" class="form-control" @bind-Value="@model.Name" />**

**<ValidationMessage For="@(() => model.Name)" />**

**</div>**

**<div class="form-group">**

**<label for="email">Email</label>**

**<InputText id="email" class="form-control" @bind-Value="@model.Email" />**

**<ValidationMessage For="@(() => model.Email)" />**

**</div>**

**<button type="submit" class="btn btn-primary">Submit</button>**

**<ValidationSummary />**

**</EditForm>**

**In Blazor,** DataAnnotationsValidator **and** ValidationSummary **are components used to perform client-side validation on a form or input elements.**

DataAnnotationsValidator **is used to attach the validation rules specified in the** System.ComponentModel.DataAnnotations **namespace to a form or input element. This component listens to the changes made to the form and applies the validation rules accordingly. If the input values don't satisfy the validation rules, an error message is displayed next to the corresponding input element.**

ValidationSummary **is used to display a summary of all the validation errors in a form or input elements. It displays a list of error messages below the form, summarizing all the validation errors found in the input elements.**

* if you add to the require ErrorMessage then your costumize error message will be displayed and not the default.

**Blazing Pizza**

## **Session 1**

**The @inject directive essentially defines a new property on the component where the first token specifies the property type and the second token specifies the property name. The property is populated for you using dependency injection.**

**Override the OnInitializedAsync method in the @code block to retrieve the list. This method is part of the component lifecycle and is called when the component is initialized. Use the GetFromJsonAsync<T>() method to handle deserializing the response JSON:**

**Layouts in Blazor are also components. They inherit from LayoutComponentBase, which defines a Body property that can be used to specify where the body of the layout should be rendered.**

**To see how the layout is associated with your pages, look at the <Router> component in App.razor. Notice that the DefaultLayout parameter determines the layout used for any page that doesn't specify its own layout directly.**

**You can also override this DefaultLayout on a per-page basis. To do so, you can add a directive such as @layout SomeOtherLayout at the top of any .razor page component.**

**The NavLink component is provided by Blazor. Components can be used from components by specifying an element with the component's type name along with attributes for any component parameters.**

**The NavLink component is the same as an anchor tag, except that it adds an active class if the current URL matches the link address. NavLinkMatch.All means that the link should be active only when it matches the entire current URL (not just a prefix). We'll examine the NavLink component in more detail in a later session.**

## **Session 2- Event handling**

**When the user clicks a pizza special, a pizza customization dialog should pop up to allow the user to customize their pizza and add it to their order. To handle DOM UI events in a Blazor app, you specify which event you want to handle using the corresponding HTML attribute and then specify the C# delegate you want called. The delegate may optionally take an event specific argument, but it's not required.**

**The @ symbol is used in Razor files to indicate the start of C# code. Surround the C# code with parens if needed to clarify where the C# code begins and ends.**

**Since component is not a separate page, it does not need the @page directive.**

**Component parameters are defined by adding a writable property to the component decorated with the [Parameter] attribute.**

**Note: Component parameter values need to have a setter and be declared public because they get set by the framework.**

**two-way binding.**

**If you wanted to implement two-way binding manually, you could do so by combining value and @onchange.**

**<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" value="@Pizza.Size" @onchange="@((ChangeEventArgs e) => Pizza.Size = int.Parse((string) e.Value))" />**

**Easier way- In Blazor you can use the @bind directive attribute to specify a two-way binding with this same behavior. The equivalent markup using @bind looks like this:**

**<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" @bind="Pizza.Size" />**

**We'd prefer to see updates as the slider is moved. Data binding in Blazor allows for this by letting you specify which event triggers a change using the syntax @bind:<eventname>. So, to bind using the oninput event instead do this:**

**<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" @bind="Pizza.Size" @bind:event="oninput" />**

## **Component events**

**The Cancel and Order buttons don't do anything yet. We need some way to communicate to the Index component when the user adds the pizza to their order or cancels. We can do that by defining component events.**

**Component events are callback parameters that parent components can subscribe to.**

**Add two parameters to the ConfigurePizzaDialog component: OnCancel and OnConfirm. Both parameters should be of type EventCallback.**

**[Parameter] public EventCallback OnCancel { get; set; }**

**[Parameter] public EventCallback OnConfirm { get; set; }**

**Normally what happens when you trigger an event (like clicking the Cancel button) is that the component that defined the event handler delegate will rerender. You could define events using any delegate type like Action or Func<string, Task>. Sometimes you want to use an event handler delegate that doesn't belong to a component - if you used a normal delegate type to define the event then nothing will be rendered or updated.**

**EventCallback is a special type that is known to the compiler that resolves some of these issues. It tells the compiler to dispatch the event to the component that contains the event handler logic. EventCallback has a few more tricks up its sleeve, but for now just remember that using EventCallback makes your component smart about dispatching events to the right place.**

## **Session 3**

## **Adding a navigation link**

**Open Shared/MainLayout.razor. As an experiment, let's try adding a new link element *without* using a NavLink component. Add a plain HTML <a> tag pointing to myorders:**

**<div class="top-bar">**

**(leave existing content in place)**

**<a href="myorders" class="nav-tab">**

**<img src="img/bike.svg" />**

**<div>My Orders</div>**

**</a>**

**</div>**

**Notice how the URL we're linking to does *not* start with a /. If you linked to /myorders, it would appear to work the same, but if you ever wanted to deploy the app to a non-root URL the link would break. The <base href="/"> tag in index.html specifies the prefix for all non-slash-prefixed URLs in the app, regardless of which component renders them.**

**The only special thing a NavLink component does is toggle its own active CSS class depending on whether its href matches the current navigation state.**

**We can use the new PageTitle component to update the title**

**This works because inside the Program.cs file is an entry that adds a HeadOutlet component to the HTML presenting the BlazingPizza application. Blazor uses this HeadOutlet to write content into the header of the HTML page.**

**builder.RootComponents.Add<HeadOutlet>("head::after");**

### **What's a <text> element?**

**<text> is *not* an HTML element at all. Nor is it a component. Once the MyOrders component is compiled, the <text> tag won't exist in the result at all.**

**<text> is a special signal to the Razor compiler that you want to treat its contents as a markup string and *not* as C# source code. It's only used on rare occasions where the syntax would otherwise be ambiguous.**

**What's with href=""?**

**If <a href=""> (with an empty string for href) surprises you, remember that the browser will prefix the <base href="/"> value to all non-slash-prefixed URLs. So, an empty string is the correct way to link to the client app's root URL.**

**Why are we using OnParametersSetAsync?**

**Asynchronous work when applying parameters and property values must occur during the OnParametersSetAsync lifecycle event. We will be adding a parameter in a later session.**

**@page "/myorders/{orderId:int}"**

**This code illustrates how components can receive parameters from the router by declaring them as tokens in the @page directive. If you want to receive a string, the syntax is simply {parameterName}, which matches a [Parameter] name case-insensitively. If you want to receive a numeric value, the syntax is {parameterName:int}, as in the example above. The :int is an example of a *route constraint*. Other route constraints, such as bool, datetime and guid, are also supported.**

**how routing actually works:**

1. **When the app first starts up, code in Program.cs tells the framework to render App as the root component.**
2. **The App component (in App.razor) contains a <Router>. Router is a built-in component that interacts with the browser's client-side navigation APIs. It registers a navigation event handler that gets notification whenever the user clicks on a link.**
3. **Whenever the user clicks on a link, code in Router checks whether the destination URL is within the same SPA (i.e., whether it's under the <base href> value, and it matches some component's declared routes). If it's not, traditional full-page navigation occurs as usual. But if the URL is within the SPA, Router will handle it.**
4. **Router handles it by looking for a component with a compatible @page URL pattern. Each {parameter} token needs to have a value, and the value has to be compatible with any constraints such as :int.**
   * **If there is a matching component, that's what the Router will render. This is how all the pages in your application have been rendering all along.**
   * **If there's no matching component, the router tries a full-page load in case it matches something on the server.**
   * **If the server chooses to re-render the client-side Blazor app (which is also what happens if a visitor is initially arriving at this URL and the server thinks it may be a client-side route), then Blazor concludes that nothing matches on either server or client, so it displays whatever NotFound content is configured.**

**we'll poll the server every few seconds for updated data. This will make it possible to show the order status in (nearly) real-time, and later, to display the delivery driver's location on a moving map.**

**@using System.Threading**

**Unfortunately, Visual Studio does not yet add @using directives automatically in Razor files, so you do have to write them in yourself when needed.**

**@code {**

**[Parameter] public int OrderId { get; set; }**

**OrderWithStatus orderWithStatus;**

**bool invalidOrder;**

**CancellationTokenSource pollingCancellationToken;**

**protected override void OnParametersSet()**

**{**

**// If we were already polling for a different order, stop doing so**

**pollingCancellationToken?.Cancel();**

**// Start a new poll loop**

**PollForUpdates();**

**}**

**private async void PollForUpdates()**

**{**

**pollingCancellationToken = new CancellationTokenSource();**

**while (!pollingCancellationToken.IsCancellationRequested)**

**{**

**try**

**{**

**invalidOrder = false;**

**orderWithStatus = await HttpClient.GetFromJsonAsync<OrderWithStatus>($"orders/{OrderId}");**

**StateHasChanged();**

**if (orderWithStatus.IsDelivered)**

**{**

**pollingCancellationToken.Cancel();**

**}**

**else**

**{**

**await Task.Delay(4000);**

**}**

**}**

**catch (Exception ex)**

**{**

**invalidOrder = true;**

**pollingCancellationToken.Cancel();**

**Console.Error.WriteLine(ex);**

**StateHasChanged();**

**}**

**}**

**}**

**}**

* **This uses OnParametersSet instead of OnInitialized or OnInitializedAsync. OnParametersSet is another component lifecycle method, and it fires when the component is first instantiated *and* any time its parameters change value. If the user clicks a link directly from myorders/2 to myorders/3, the framework will retain the OrderDetails instance and simply update its OrderId parameter in place.**
  + **As it happens, we haven't provided any links from one "my orders" screen to another, so the scenario never occurs in this application, but it's still the right lifecycle method to use in case we change the navigation rules in the future.**
* **We're using an async void method to represent the polling. This method runs for arbitrarily long, even while other methods run. async void methods have no way to report exceptions upstream to callers , so it's important to use try/catch and do something meaningful with any exceptions that may occur.**
* **We're using CancellationTokenSource as a way of signalling when the polling should stop. Currently it stops if there's an exception, or once the order is delivered.**
* **We need to call StateHasChanged to tell Blazor that the component's data has (possibly) changed. The framework will then re-render the component. There's no way that the framework could know when to re-render your component otherwise, because it doesn't know about your polling logic.**

**Remember to Dispose!**

**If you deployed your app to production right now, bad things would happen. The OrderDetails logic starts a polling process, but doesn't end it. If the user navigated through hundreds of different orders (thereby creating hundreds of different OrderDetails instances), then there would be hundreds of polling processes left running concurrently, even though all except the last were pointless because no UI was displaying their results.**

**This is wasteful of client-side memory and CPU time, network bandwidth, and server resources.**

**To fix this, This is simply a matter of using the IDisposable interface.**

**In OrderDetails.razor, add the following directive at the top of the file, underneath the other directives:**

**@implements IDisposable**

**Now if you try to compile the application, the compiler will complain:**

**error CS0535: 'OrderDetails' does not implement interface member 'IDisposable.Dispose()'**

**Resolve this by adding the following method inside the @code block:**

**void IDisposable.Dispose()**

**{**

**pollingCancellationToken?.Cancel();**

**}**

**The framework calls Dispose automatically when any given component instance is torn down and removed from the UI.**

**Automatically navigating to order details**

**Right now, once users place an order, the Index component simply resets its state and their order appears to vanish without a trace. This is not very reassuring for users. We know the order is in the database, but users don't know that.**

**It would be nice if, once the order is placed, the app automatically navigated to the "order details" display for that order. This is quite easy to do.**

**Switch back to your Index component code. Add the following directive at the top:**

**@inject NavigationManager NavigationManager**

**The NavigationManager lets you interact with URIs and navigation state. It has methods to get the current URL, to navigate to a different one, and more.**

**To use this, update the PlaceOrder code so it calls NavigationManager.NavigateTo:**

**async Task PlaceOrder()**

**{**

**var response = await HttpClient.PostAsJsonAsync("orders", order);**

**var newOrderId = await response.Content.ReadFromJsonAsync<int>();**

**order = new Order();**

**NavigationManager.NavigateTo($"myorders/{newOrderId}");**

**}**

# **Session 4 -Refactor state management**

# **A problem**

# **You might have noticed this already, but our application has a bug! Since we're storing the list of pizzas in the current order on the Index component, the user's state can be lost if the user leaves the Index page. To see this in action, add a pizza to the current order (don't place the order yet) - then navigate to the MyOrders page and back to Index. When you get back, you'll notice the order is empty!**

# **A solution**

# **We're going to fix this bug by introducing something we've dubbed the *AppState pattern*. The *AppState pattern* adds an object to the DI container that you will use to coordinate state between related components. Because the *AppState* object is managed by the DI container, it can outlive the components and hold on to state even when the UI changes. Another benefit of the *AppState pattern* is that it leads to greater separation between presentation (components) and business logic.**

# **Create a new class called OrderState in the Client Project root directory - and register it as a scoped service in the DI container. In Blazor WebAssembly applications, services are registered in the Program class. Add the service just before the call to await builder.Build().RunAsync();.**

# **builder.Services.AddScoped<OrderState>();**

# **Note: the reason why we choose scoped over singleton is for symmetry with a server-side-components application. Singleton usually means *for all users*, where as scoped means *for the current unit-of-work*.**

# **Updating Index**

# **Now that this type is registered in DI, we can @inject it into the Index page.**

# **@inject OrderState OrderState**

# **Recall that @inject is a convenient shorthand to both retrieve something from DI by type, and define a property of that type.**

# **Now, let's add properties and methods to this class that will represent and manipulate the state of an Order and a Pizza.**

# **Move the configuringPizza, showingConfigureDialog and order fields to be properties on the OrderState class. Make them private set so they can only be manipulated via methods on OrderState.**

# **public class OrderState**

# **{**

# **public bool ShowingConfigureDialog { get; private set; }**

# **public Pizza ConfiguringPizza { get; private set; }**

# **public Order Order { get; private set; } = new Order();**

# **}**

# **Now let's move some of the methods from the Index to OrderState. We won't move PlaceOrder into OrderState because that triggers a navigation, so instead we'll just add a ResetOrder method.**

**public void ShowConfigurePizzaDialog(PizzaSpecial special)**

**{**

**ConfiguringPizza = new Pizza()**

**{**

**Special = special,**

**SpecialId = special.Id,**

**Size = Pizza.DefaultSize,**

**Toppings = new List<PizzaTopping>(),**

**};**

**ShowingConfigureDialog = true;**

**}**

**public void CancelConfigurePizzaDialog()**

**{**

**ConfiguringPizza = null;**

**ShowingConfigureDialog = false;**

**}**

**public void ConfirmConfigurePizzaDialog()**

**{**

**Order.Pizzas.Add(ConfiguringPizza);**

**ConfiguringPizza = null;**

**ShowingConfigureDialog = false;**

**}**

**public void ResetOrder()**

**{**

**Order = new Order();**

**}**

**public void RemoveConfiguredPizza(Pizza pizza)**

**{**

**Order.Pizzas.Remove(pizza);**

**}**

# **At this point it should be possible to get the Index component compiling again by updating references to refer to various bits attached to OrderState.**

# **Feel free to create convenience properties for things like OrderState.Order or OrderState.Order.Pizzas if it feels better to you that way.**

## **Exploring state changes**

## **This is a good opportunity to explore how state changes and rendering work in Blazor, and how EventCallback solves some common problems. The details of what is happening become more complicated now that OrderState is involved.**

## **EventCallback tells Blazor to dispatch the event notification (and rendering) to the component that defined the event handler. If the event handler is not defined by a component (OrderState) then it will substitute the component that *hooked up* the event handler (Index).**

## **Conclusion**

## **So let's sum up what the *AppState pattern* provides:**

* **Moves shared state outside of components into OrderState**
* **Components call methods to trigger a state change**
* **EventCallback takes care of dispatching change notifications**

**We've covered a lot of information as well about rendering and eventing:**

* **Components re-render when parameters change or they receive an event**
* **Dispatching of events depends on the event handler delegate target**
* **Use EventCallback to have the most flexible and friendly behavior for dispatching events**

**client-server (Blazor WebAssembly and WebAPI) using HttpClient**

* **Download Microsoft.AspNetCore.Blazor.HttpClient**

NuGet\Install-Package Microsoft.AspNetCore.Blazor.HttpClient -Version 3.2.0-preview3.20168.3

* **Add this in program.cs- Blazor WebAssembly**

builder.Services.AddSingleton<HttpClient>();

* **Inject this in the razor file- Blazor WebAssembly**

@inject HttpClient HttpClient;

* **Add the target class and async task function that use HttpClient.GetJsonAsync that fetch data from the api(When the WebApi project active) in the razor file- Blazor WebAssembly**

@code{

List<Student> Students = new List<Student>();

private async Task GetStudents()

{

Students = await HttpClient.GetJsonAsync<List<Student>>("https://localhost:7271/api/Students");

}

public class Student

{

public int StudentId { get; set; } = 0;

public string Name { get; set; } = "";

public string Roll { get; set; } = "";

}

}

* **In WebApi- server looks like:**

[Route("api/[controller]")]

[ApiController]

public class StudentsController : Controller

{

[HttpGet]

public IEnumerable<Student> Get()

{

return Students;

}

}

**Blazor Services, Dependency Injection & Page Parameters**

[blazor services-look in Services in client and program in client](https://github.com/patrickgod/BlazorBlog)

**upload files(.txt/.xml) from your PC and transfer it to the server through an API**

[Microsoft toturial](https://learn.microsoft.com/en-us/aspnet/core/blazor/file-uploads?view=aspnetcore-7.0&pivots=server)

**Blazor event handlers**

Blazor is a web framework for building interactive client-side web applications using C# and .NET. Blazor provides a number of event handlers that can be used to handle various events that occur during the lifecycle of a component. Here are some of the most commonly used event handlers in Blazor and their explanations:

1. OnInitializedAsync: This event is raised when the component is first initialized. It is typically used to initialize the component's state and perform any required setup.
2. OnParametersSetAsync: This event is raised when the component's parameters are updated. It is typically used to update the component's state based on the new parameter values.
3. OnAfterRenderAsync: This event is raised after the component has been rendered on the page. It is typically used to perform any required post-rendering cleanup or to interact with the rendered HTML.
4. OnAfterRenderAsync(bool): This event is similar to OnAfterRenderAsync, but it also includes a boolean parameter that indicates whether the component has been fully rendered or not.
5. OnInitialized: This event is raised synchronously when the component is first initialized. It is typically used to perform any synchronous initialization that is required.
6. OnParametersSet: This event is raised synchronously when the component's parameters are updated. It is typically used to update the component's state based on the new parameter values.
7. OnAfterRender: This event is raised synchronously after the component has been rendered on the page. It is typically used to perform any required post-rendering cleanup or to interact with the rendered HTML.
8. OnAfterRender(bool): This event is similar to OnAfterRender, but it also includes a boolean parameter that indicates whether the component has been fully rendered or not.

These are some of the most commonly used event handlers in Blazor. By handling these events, you can perform various tasks such as initializing component state, updating component state, and interacting with the rendered HTML.

**Blazor and EF and DbContext**

[in this module](https://learn.microsoft.com/en-us/training/modules/interact-with-data-blazor-web-apps/10-summary)

[also here](https://learn.microsoft.com/he-il/training/modules/use-pages-routing-layouts-control-blazor-navigation/3-exercise-change-navigation-blazor-using-page-directive)

**Routing Parameters**

Now suppose the user tried to specify two favorites by requesting the URI http://www.contoso.com/favoritepizza/margherita/hawaiian. The page displays the message "Your favorite pizza is: margherita" and ignores the subfolder hawaiian. You can change this behavior by using a *catch-all route parameter*, which captures paths across multiple URI folder boundaries (forward slashes). Prefix an asterisk (\*) to the route parameter name to make the route parameter catch-all:

@page "/FavoritePizza/{\*favorites}"

<h1>Choose a Pizza</h1>

<p>Your favorite pizzas are: @Favorites</p>

@code {

[Parameter]

public string Favorites { get; set; }

}

**Blazor Layouts**

[in this module](https://learn.microsoft.com/he-il/training/modules/use-pages-routing-layouts-control-blazor-navigation/6-build-reusable-component-using-layouts)

**Extras Tutorial for server side**

[Minimal Api](https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api?view=aspnetcore-6.0&tabs=visual-studio)

[Json Source Generator](https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/source-generation)