Blazor Cheat Sheet

A Quick Blazor Reference Guide Just For You. Thanks for the Support!!!

- The is accurate at the time of creation, information is subject to change

Created By
Karim Blanche
Just Blazor Programming

Check Out Just Blazor Programming On Youtube!!

Date Created
16AUG2022

What Is Blazor?

Blazor is a newer framework that allows you to build web apps with C# including the front-end. The initial release was in 2018, and Microsoft has been steadily increasing support and usage which includes adding it to .Net MAUI as an option its front-end for mobile development. Blazor is essentially an upgrade to an older product called Razor, and still uses Razor pages for its components. Blazor comes in two flavors (Hosting Models) Blazor Webassembly(WASM) and Blazor Server. Blazor Hybrid Apps (Like .NET MAUI) are technically another hosting model but are not covered here.

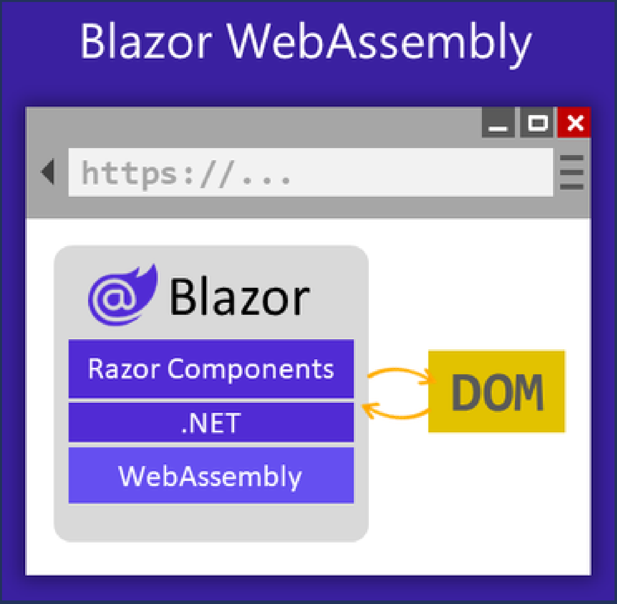
Supported Browsers[1]

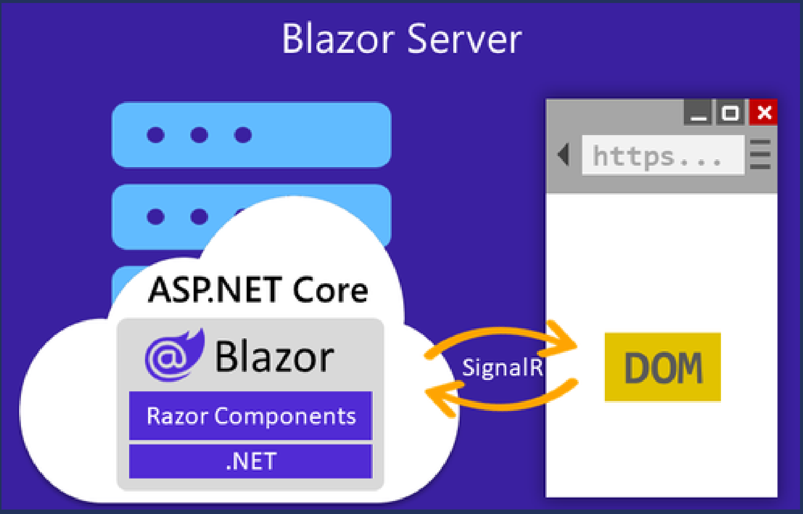
- Apple Safari, including iOS
- Google Chrome, Including Android
- Microsoft Edge
- Mozilla Firefox
- Microsoft Edge Legacy

Keep This In Mind

Blazor allows you to use C# in the front-end, but there might be times where you need Javascript either for a library you want to use, or for other uses. Some JS knowledge might be required and paying special attention to the JS Interop section will help you in these situations.

HOSTING MODELS AT A GLANCE[1]





Blazor Server vs Blazor WebAssembly

Blazor Server vs Blazor WebAssembly

The main differences between Blazor Server and Webassembly is essentially where the work of compiling .NET is happening. Blazor Server is similar to ASP.Net Razor in the sense that they both have compile code and access content via WebSockets protocol. Blazor Server handles this using a SignalR connection so all the works is still done in the server but it feels quick to load for the user. Blazor Server also uses Razor pages.

Blazor WebAssembly compiles the .NET code in the browser, but first it must be downloaded giving it a heavier initial payload. This will seem like its slow to start but after the load the experience is also rather quick and you don't have to have a server-side dependency to continue using the app. Its most similar to other JS frameworks in terms of how you would go about architecting an application.

Quick Comparison Chart

*Most Info From Microsoft Docs	Blazor Server	Blazor WebAssembly (WASM)
Initial Load Time	Blazor Server has a faster initial load time than Blazor WASM. The drawback of achieving this is you need a constant internet connection to continue usage.	Load time is the slowest with this hosting model, among other frameworks its also slower, however this is improving with updates. Once complete, it will feel as quick or quicker than Server. Works offline once loaded.
.NET Compatibility	Blazor Server Apps have complete .NET API Compatibility.	Blazor WASM apps have some limitations in compatibility. If theres a requirement where you need one of these limited .NET API's use Blazor Server
Server VS API Calls	Blazor Server is similar to ASP.NET Razor in the sense it has direct access to the server. It uses SignalR Connections. There's a certain limited number of connections and you need internet access to continue working	Blazor WASM works like any of the popular frameworks out there (React, Angular) in which you will be using API calls in order to interact with the backend over the network. You can also use gRPC-web or SignalR.
SignalR VS WebAssembly	Blazor Server works using a SignalR connection. Essentially it allows for server-side calls to push content to the client instantly. It allows you to run Blazor without needing the .NET to run in the browser. You will need a constant connection to ensure your app works.	WebAssembly allows .NET code to run on the browser. With this in mind your front-end portion of code will be compiled and downloaded to the browser. Any sensitive information or code should be delegated to the back-end. Enables C# the same development experience as other JS frameworks.

Need To Know Blazor Concepts

What Is In This Page

Here is a list of concepts that you should understand when it comes to working with Blazor. Referential information will be provided but each concept can be pretty in-depth and that will require more learning outside of the scope of this cheat sheet. Its a cheat sheet not a text book.

Data-Binding

Forms And Validation

Event Handling

JS Interop

Routing And Navigation

Components

Authorization

Life Cycle And Life Cycle Methods

Keep In Mind:

As Blazor is written in C# and uses .NET technology, to truly understand Blazor you need to understand C# and the .NET framework in general. Its easier if you go from a .NET related tech stack to Blazor because you already would have most of the knowledge needed and the rest is just Blazor conventions.

Data-Binding

- Data-Binding is how we couple the data (whether its input, data from the server, data from another source entirely, etc) to the visual element (in this case the DOM or Document Object Model).

- There are two forms of Data-Binding, One-way and Two-Way

- One-Way only allows you to either bind the data to the DOM or from the DOM to the variable. There is no exchange of data. Uni-Directional flow.

- Two-Way allows you to exchange data between the component and the DOM. This will allow you to both interact and see the changes in the data you are trying to do. Bi-directional flow.

- The three most important attributes to remember for Data-Binding (although there are more) are:
- @bind
- @bind-value
- @bind-value:event

@bind-value & @bind-value:event

@bind-value allows you to bind a variable and its value to an element

@bind-value:event binds a certain even that element triggers to a DOM event. **More Info In Event Handling**

```
<input @bind-value="InputValue" @bind-value:event="onchange" />
```

@bind

```
<input @bind="InputValue" />
```

This is the equivalent of :

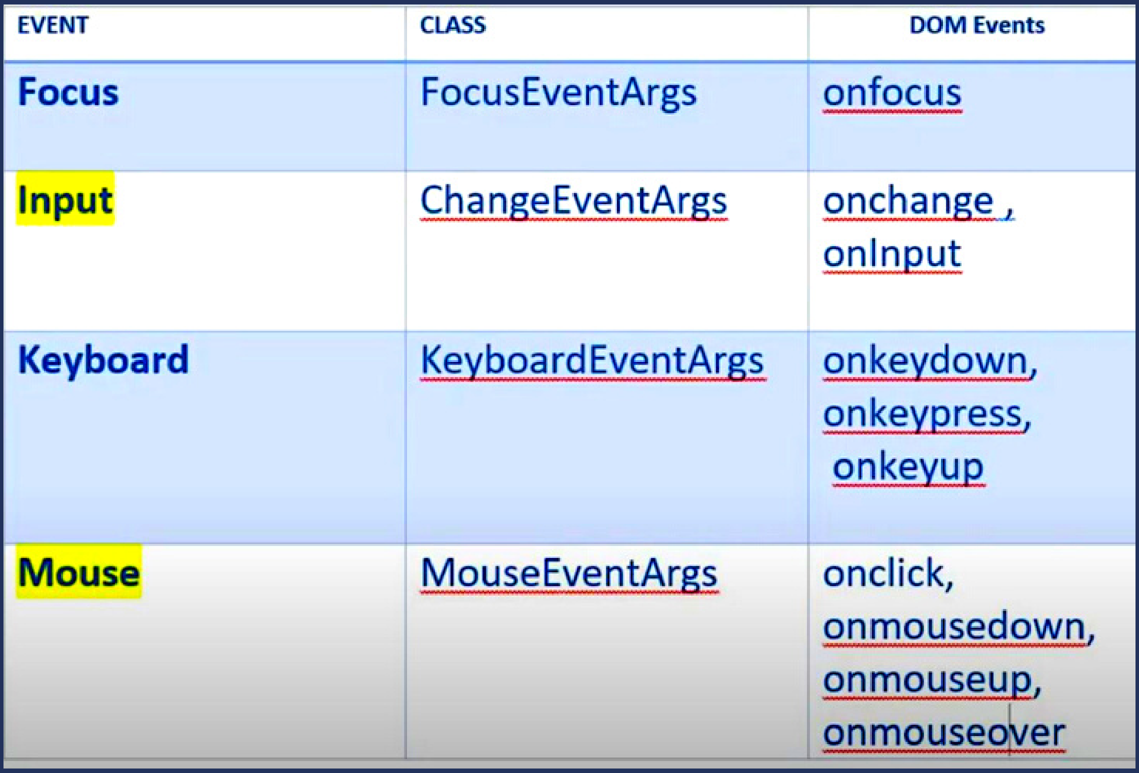
```
<input @bind-value="InputValue" @bind-value:event="onchange" />
```

Event Handling

- Event Handlers Allow UI To Track User Events And Add Responses To Them aka when you click, mouseover, change etc.

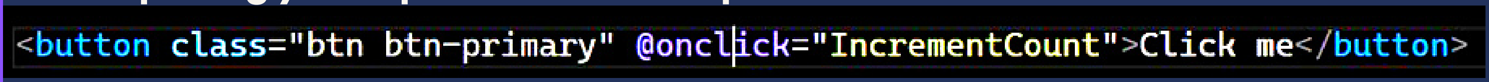
The Most Important Expression To Remember Is: @on[DOMEVENT] = "[DELEGATE]" this is how you invoke an event handler within the element.

A List Of Most Used Events & Associated Classes

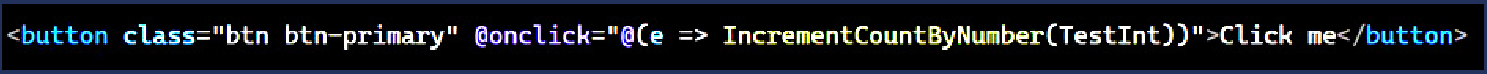


EVENT	CLASS	DOM Events
Focus	FocusEventArgs	onfocus
Input	ChangeEventArgs	onchange , onInput
Keyboard	KeyboardEventArgs	onkeydown, onkeypress, onkeyup
Mouse	MouseEventArgs	onclick, onmousedown, onmouseup, onmouseover

You can pass in functions into events, even those expecting you to pass in data via a parameter

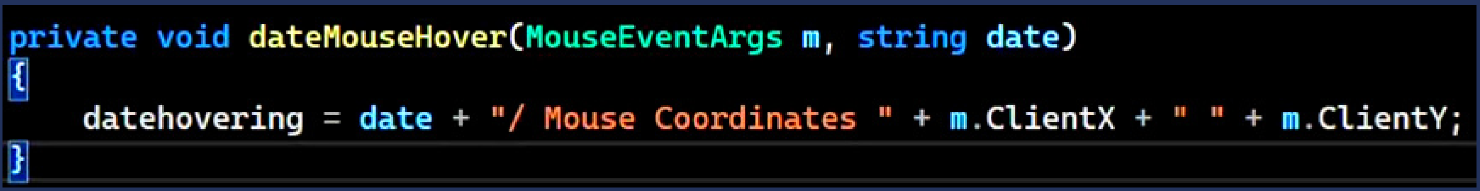


```
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```



```
<button class="btn btn-primary" @onclick="@(e => IncrementCountByNumber(TestInt))">Click me</button>
```

An Example passing in a class (MouseEventArgs), theres nothing you need to do except add it as a parameter in the function. You dont have to add it to the html element itself.

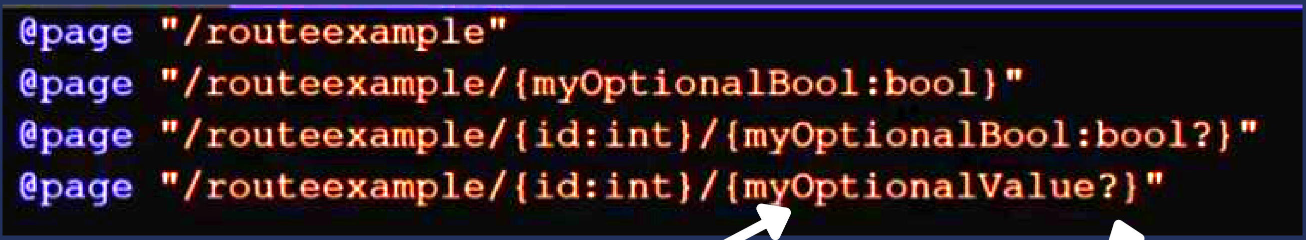


```
private void dateMouseHover(MouseEventArgs m, string date)
{
    datehovering = date + "/ Mouse Coordinates " + m.ClientX + " " + m.ClientY;
}
```

Routing And Navigation

- A page can have a page URI associated with them in order for you to be able to navigate to them. You can even pass parameters through these URIs.
- You may also navigate in code via the NavigationManager class

- Examples of URI's with and without parameters, you can have multiple URIs for the same page but those URIs must be unique

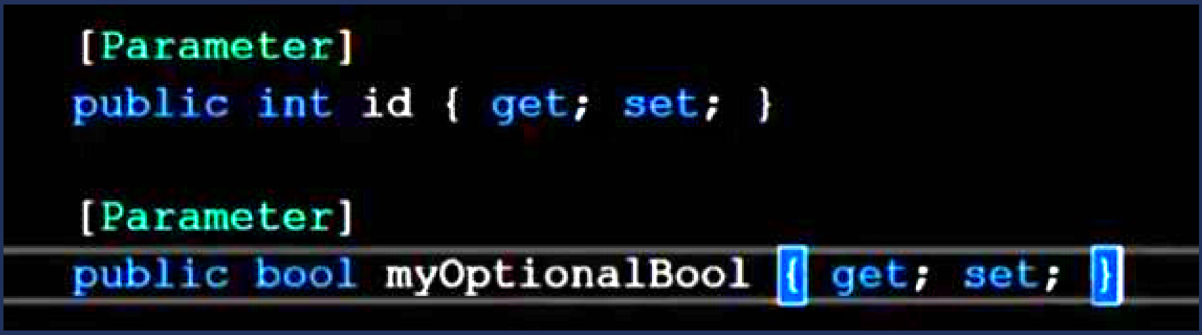


```
@page "/routeexample"
@page "/routeexample/{myOptionalBool:bool}"
@page "/routeexample/{id:int}/{myOptionalBool:bool?}"
@page "/routeexample/{id:int}/{myOptionalValue?}"
```

This Is How You Pass A String, you dont need to declare it.
Other value types must be declared

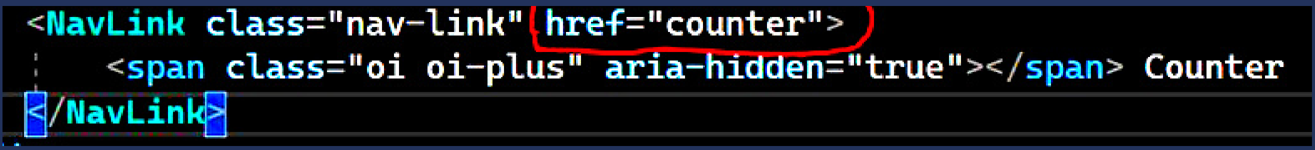
? means OPTIONAL

- In order to access the parameter's value being passed you must have a variable with their name and the data attribute [Parameter] associated.



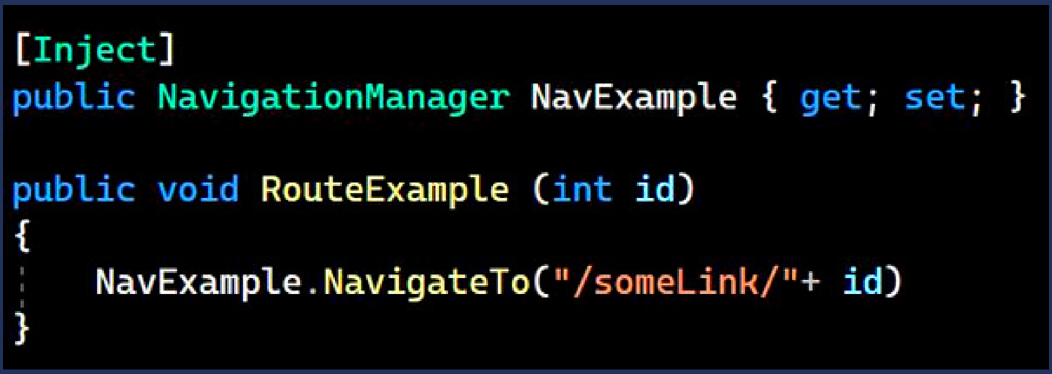
```
  [Parameter]
  public int id { get; set; }

  [Parameter]
  public bool myOptionalBool { get; set; }
```

- Two examples of navigation, this is via the NavLink component, you could also do this with other elements.



```
<NavLink class="nav-link" href="counter">
    <span class="oi oi-plus" aria-hidden="true"></span> Counter
</NavLink>
```

- This is how you can use the NavigationManager class in order to navigate in code



```
[Inject]
public NavigationManager NavExample { get; set; }

public void RouteExample (int id)
{
    NavExample.NavigateTo("/someLink/"+ id)
}
```

Authorization

- There is a difference between Authorization and Authentication. When it comes to Blazor, since we are dealing with the front-end what you care about here is the authorization step. Authentication is something that the back-end handles and there are many ways to implement this, however thats outside of the scope of this cheat sheet.

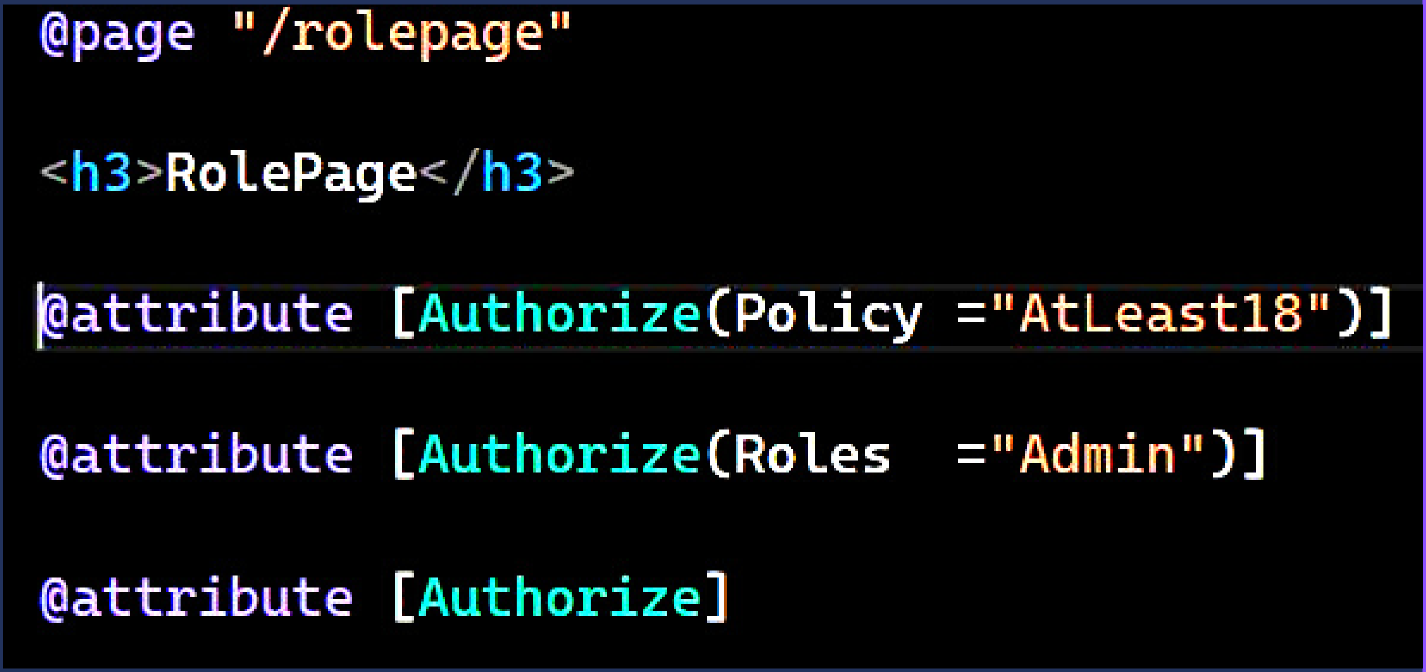
- Authentication: Process of proving that you are who you say you are[1]
- Authorization: The act of granting an authenticated party permission to do something. It specifies what data you're allowed to access and what you can do with that data[1]

Authorization can be enforced using:
- Attributes on the page
-AuthorizeView Component

You may also assign Roles and Policies in order to enforce Authorization for specific circumstances or needs.

These are added first to the Program.cs page through the Authorization Service. You have to add the policies and roles here first before implementing them in other places. You can also add logic to policies.

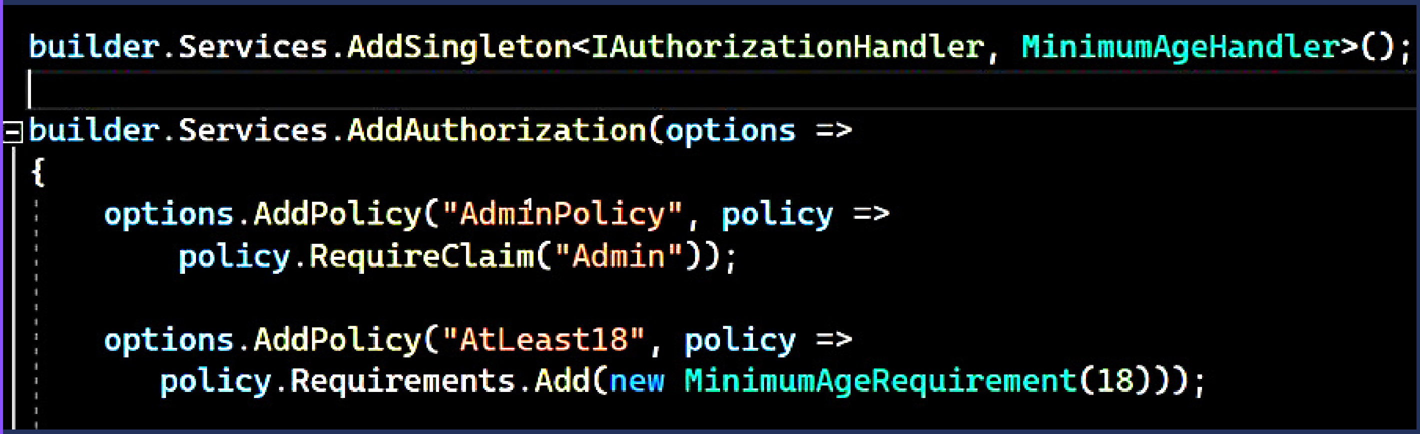
This is how to add authorization to a page with the @attribute key word. Note there are 3 different ways, one for Policy, Role, and unspecified authorization



```
@page "/rolepage"

<h3>RolePage</h3>

@attribute [Authorize(Policy ="AtLeast18")]

@attribute [Authorize(Roles  ="Admin")]

@attribute [Authorize]
```

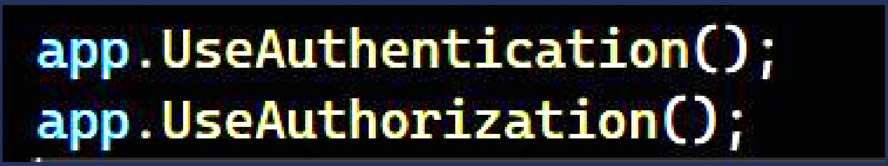
Authorization (cont.)

- Here is an example on adding the authorization functionality in Program.cs, you can add a policy this way, and there is an example of adding logic for one of the two policies. You must invoke that class within the singleton and the IAuthorizationHandler. Then add it via the AddAuthorization service.



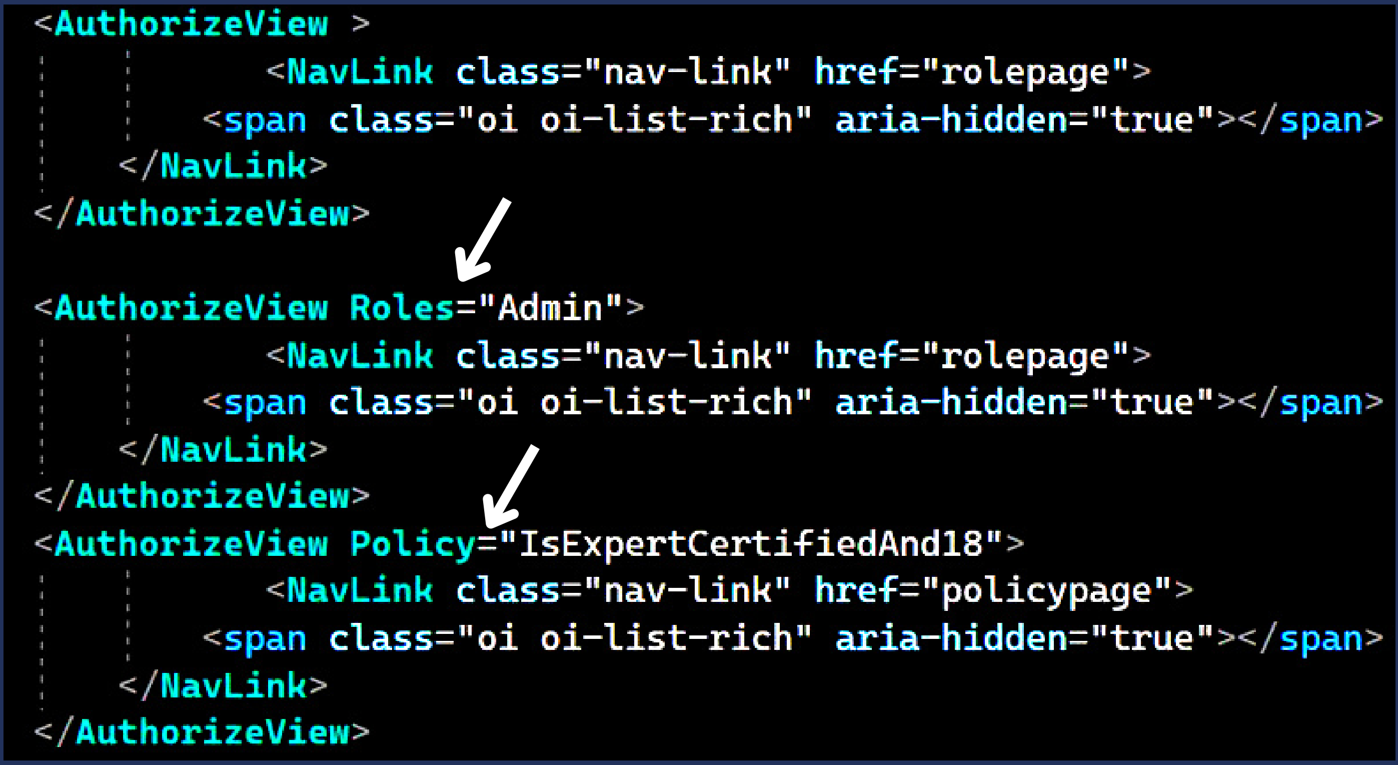
```
builder.Services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminPolicy", policy =>
        policy.RequireClaim("Admin"));

    options.AddPolicy("AtLeast18", policy =>
       policy.Requirements.Add(new MinimumAgeRequirement(18)));
```

You must add this below the AddAuthorization service.



```
app.UseAuthentication();
app.UseAuthorization();
```

Using the AuthorizeView component can allow you to enforce the authorization in your Razor pages. Example of Unspecified, Policy, and Role Authorization

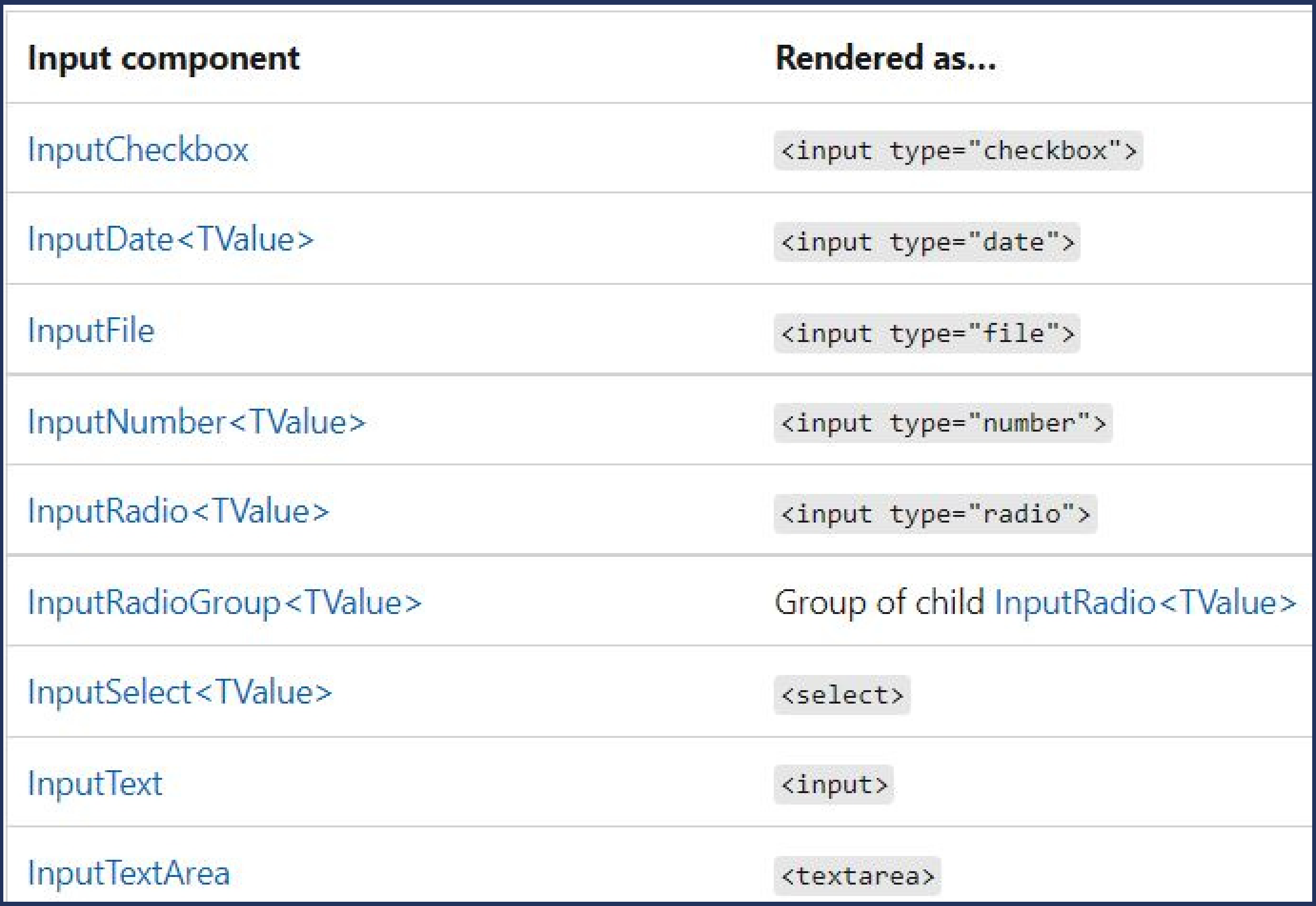


```
<AuthorizeView >
            <NavLink class="nav-link" href="rolepage">
        <span class="oi oi-list-rich" aria-hidden="true"></span>
    </NavLink>
</AuthorizeView>

<AuthorizeView Roles="Admin">
            <NavLink class="nav-link" href="rolepage">
        <span class="oi oi-list-rich" aria-hidden="true"></span>
    </NavLink>
</AuthorizeView>
<AuthorizeView Policy="IsExpertCertifiedAnd18">
            <NavLink class="nav-link" href="policypage">
        <span class="oi oi-list-rich" aria-hidden="true"></span>
    </NavLink>
</AuthorizeView>
```

Forms And Validation

- Blazor has the ability to create forms and validations out of the box. There is no need to download a separate package to achieve this.
- A Form must have an associated Model (a class to represent the properties you expect the form to have and pass data to).
- A Form has specific components that you can use in order to create and accept different data types or different functions. Like text, password, checkbox, etc.

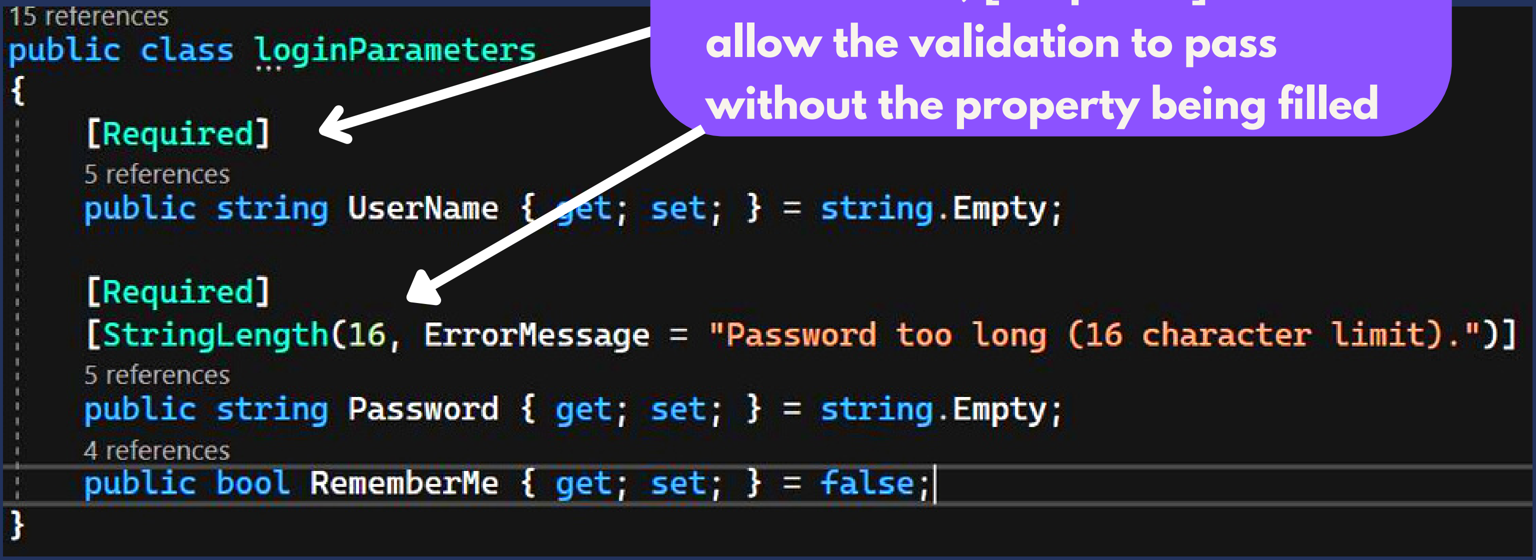
Table Of Input Components[1]



Input component	Rendered as…
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputFile	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
InputRadio<TValue>	`<input type="radio">`
InputRadioGroup<TValue>	Group of child InputRadio<TValue>
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

Forms And Validation (cont..)

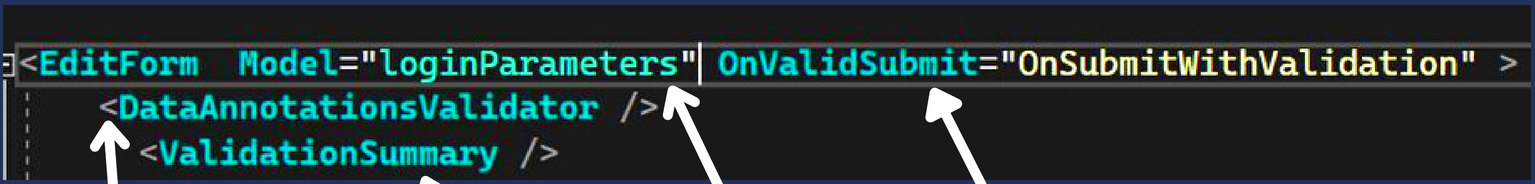
Example Of A Model



```
15 references
public class loginParameters
{
    [Required]
    5 references
    public string UserName { get; set; } = string.Empty;

    [Required]
    [StringLength(16, ErrorMessage = "Password too long (16 character limit).")]
    5 references
    public string Password { get; set; } = string.Empty;
    4 references
    public bool RememberMe { get; set; } = false;
}
```

Data Annotations To Add Logic To The Form, [Required] wont allow the validation to pass without the property being filled

Writing an EditForm With validation



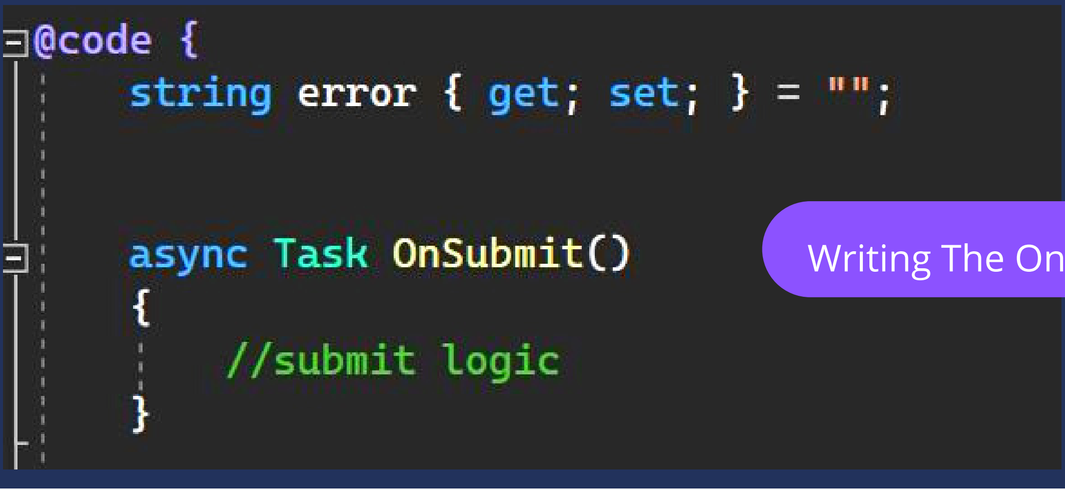
```
<EditForm  Model="loginParameters" OnValidSubmit="OnSubmitWithValidation" >
    <DataAnnotationsValidator />
      <ValidationSummary />
```

You can assign a function to trigger OnSubmit, OnValidSubmit, OnInvalidSubmit.

This is where your Model for the form goes that designated the properties expected.

DataAnnotations validations are now supported through this component

Shows the error messages you have assigned to the form if they triggered



```
@code {
    string error { get; set; } = "";


    async Task OnSubmit()
    {
        //submit logic
    }
```

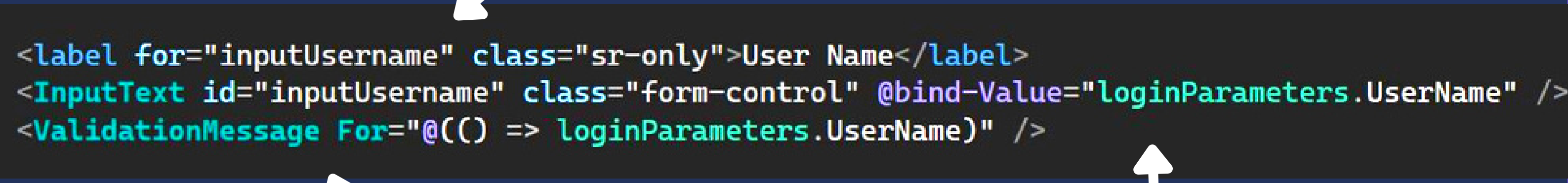
Writing The OnSubmit Function

Forms And Validation (cont..)

Writing an Input Component

This is a label associated to the inputText via the id associated.

Input Text



```
<label for="inputUsername" class="sr-only">User Name</label>
<InputText id="inputUsername" class="form-control" @bind-Value="loginParameters.UserName" />
<ValidationMessage For="@(() => loginParameters.UserName)" />
```

A Validation Message that has been associated with the component via its property

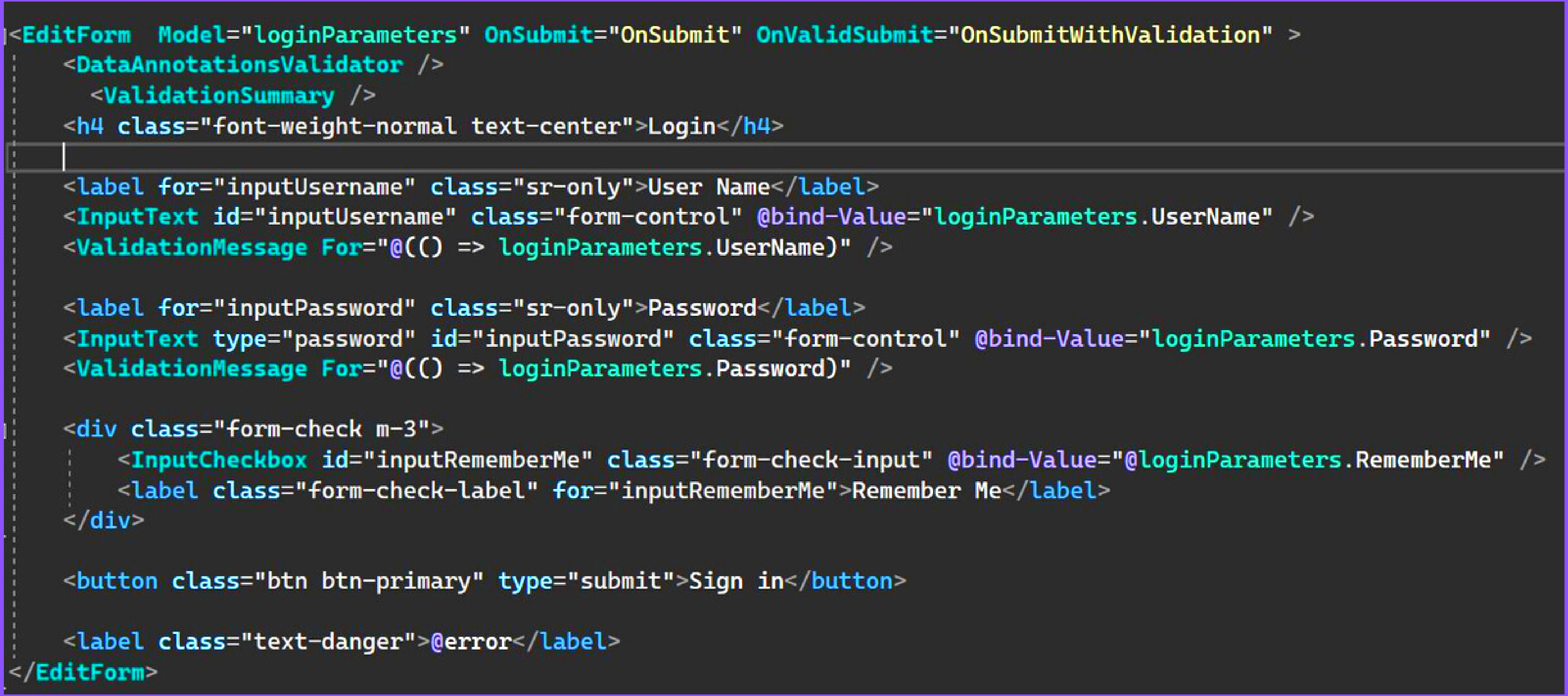
This is the Input Component Itselt. Here you will bind it to the property of the model you want and assign an id. Its best practice to have individual id's to allow for html associations.

Keep In Mind:

Every Input Component is a bit different and some have different data types that can be accepted. They all follow the same common conventions such as associating to a property in the model, and html attributes like id are present. InputCheckbox only accepts booleans while InputText is string. Refer to the chart to see their HTML counterparts

Forms And Validation (cont..)

A Complete Form Example

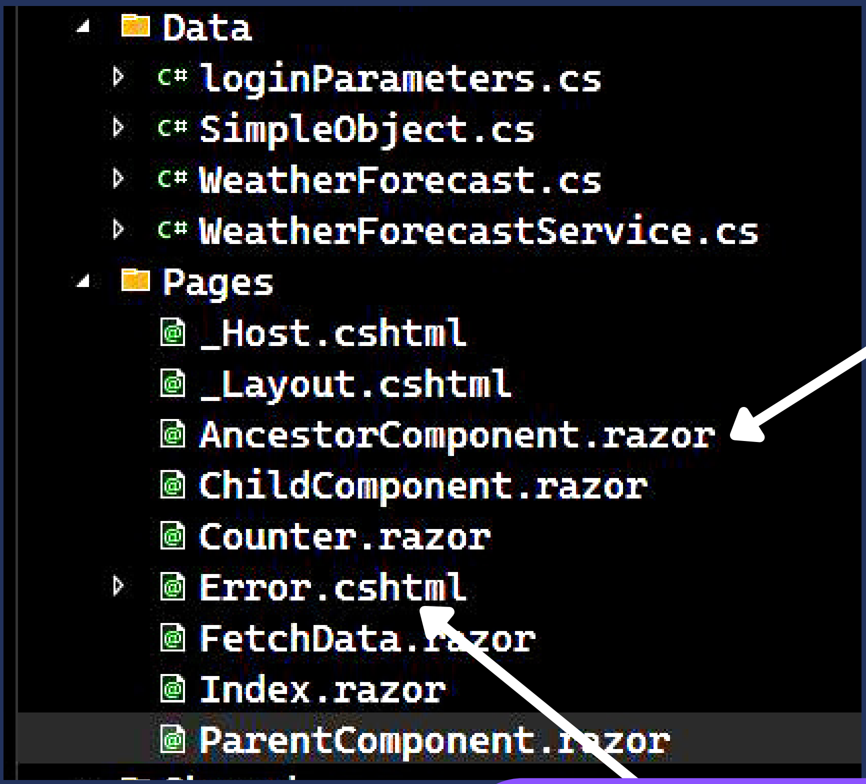


```
<EditForm  Model="loginParameters" OnSubmit="OnSubmit" OnValidSubmit="OnSubmitWithValidation" >
    <DataAnnotationsValidator />
      <ValidationSummary />
    <h4 class="font-weight-normal text-center">Login</h4>

    <label for="inputUsername" class="sr-only">User Name</label>
    <InputText id="inputUsername" class="form-control" @bind-Value="loginParameters.UserName" />
    <ValidationMessage For="@(() => loginParameters.UserName)" />

    <label for="inputPassword" class="sr-only">Password</label>
    <InputText type="password" id="inputPassword" class="form-control" @bind-Value="loginParameters.Password" />
    <ValidationMessage For="@(() => loginParameters.Password)" />

    <div class="form-check m-3">
        <InputCheckbox id="inputRememberMe" class="form-check-input" @bind-Value="@loginParameters.RememberMe" />
        <label class="form-check-label" for="inputRememberMe">Remember Me</label>
    </div>

    <button class="btn btn-primary" type="submit">Sign in</button>

    <label class="text-danger">@error</label>
</EditForm>
```

Components

- Components are a concept found in most modern-day web app frameworks. Blazor is no exception and this can be considered the backbone of working with Blazor
- A Component is basically HTML + Code (C# in Blazor). Blazor components are created as razor files.
- All Components have to start with a capital letter.
- You can treat components like html pages if you assign them a page URI to make them navigable in your solution.
- Components can pass in parameters and contain attributes as well, you may even nest components within other components. They are meant to be used modularly in your code.
- All components share a Component Lifecycle (See Lifecycle Page for more information) that determines the order of rendering objects and data.
- Blazor comes packed with all sorts of components ready to be used. You can also create them yourself in case that wasnt clear.
- All component logic must be within the @code block. This is where you can assign/create variables, objects, and create functions for this component

Solution Explorer



```
◢ Data
    ▷ C# loginParameters.cs
    ▷ C# SimpleObject.cs
    ▷ C# WeatherForecast.cs
    ▷ C# WeatherForecastService.cs
◢ Pages
       _Host.cshtml
       _Layout.cshtml
       AncestorComponent.razor
       ChildComponent.razor
       Counter.razor
    ▷  Error.cshtml
       FetchData.razor
       Index.razor
       ParentComponent.razor
```

All Components start with a capital letter and are .razor files. They are different from .cshtml because you cannot use any of the component conventions in those files.

.cshtml files are not components

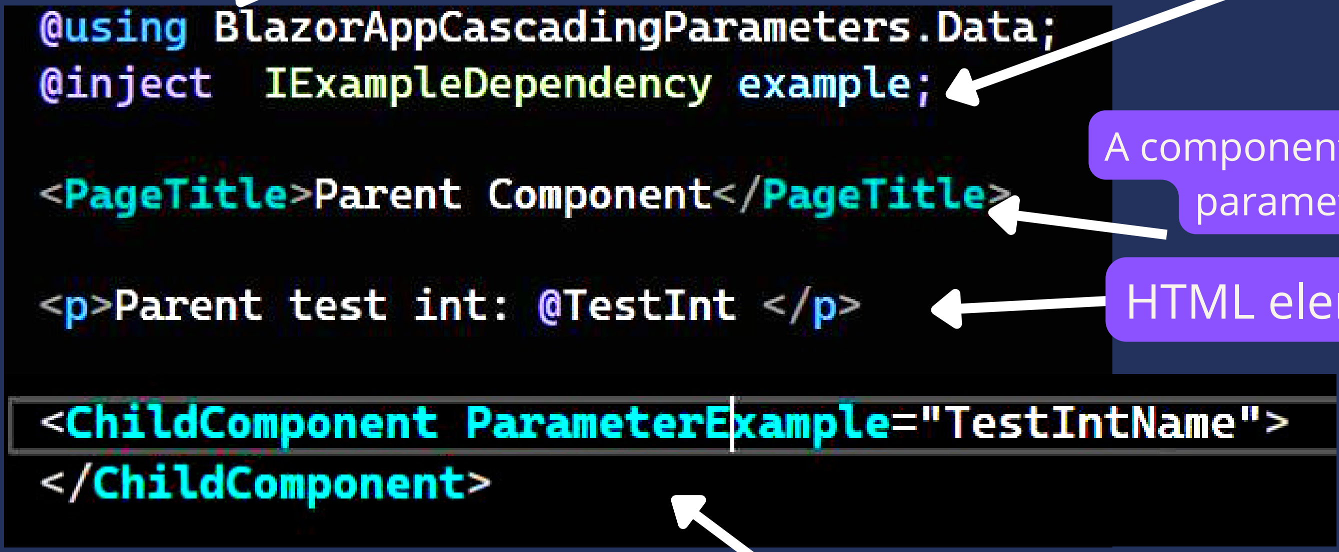
Components

@using can be used to access an external library or namespace, like in a .cs class

@inject Allows dependency injection of a service that has been registered in your Program.cs file

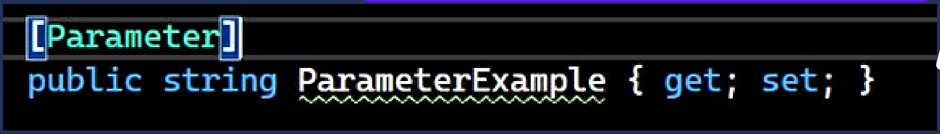
A component with no parameters

HTML element



```
@using BlazorAppCascadingParameters.Data;
@inject  IExampleDependency example;

<PageTitle>Parent Component</PageTitle>

<p>Parent test int: @TestInt </p>

<ChildComponent ParameterExample="TestIntName">
</ChildComponent>
```

A component with a parameter example, in the ChildCompoenent there is a defined variable called ParameterExample of type string with a [Parameter] attribute. This is how you assign parameters to a component. You can also use cascading paramaters as well



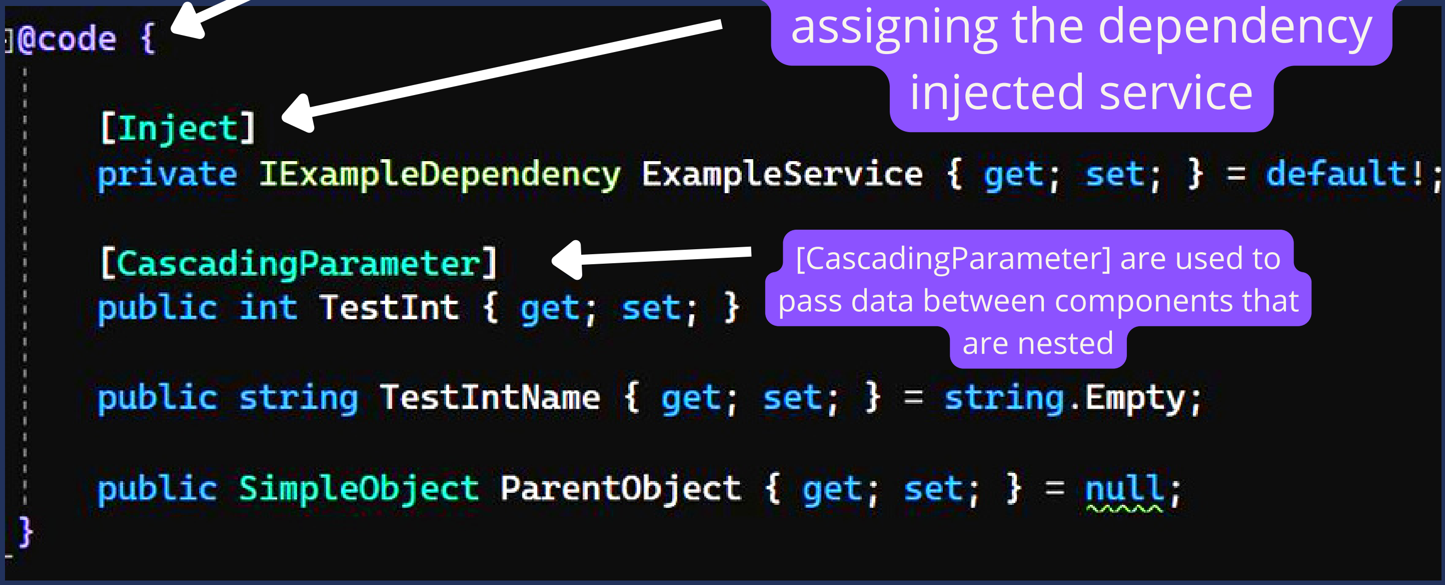
```
[Parameter]
public string ParameterExample { get; set; }
```

ChildComponent

@code block

[inject] is a data attribute for assigning the dependency injected service

[CascadingParameter] are used to pass data between components that are nested



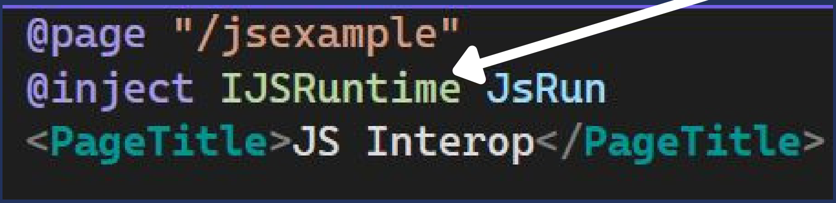
```
@code {

    [Inject]
    private IExampleDependency ExampleService { get; set; } = default!;

    [CascadingParameter]
    public int TestInt { get; set; }

    public string TestIntName { get; set; } = string.Empty;

    public SimpleObject ParentObject { get; set; } = null;
}
```

Javascript Interoperability (JS Interop)

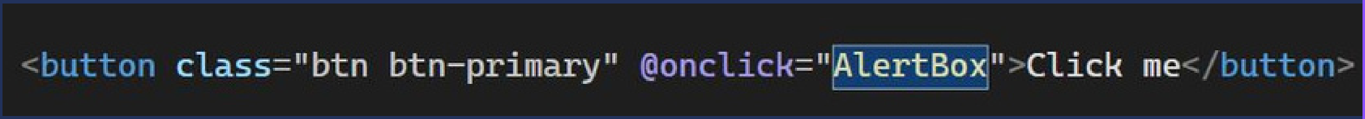
- JS Interop is how Blazor interacts with Javascript and how you will invoke JS functions.
- JS Interop calls are asynchronous by default. However for Blazor WASM you can create JS calls that are synchronous. You cannot do this in Blazor Server.
- The initial Blazor project comes with a javascript library already inside it, if you want to find it you can go to _Layout.cshtml or Index.cshml there should be a js library available. In the _Layout is also where you will any external dependencies like CDN's as well as javascript libraries you download or point to via CDN.
- When you use the InvokeAsync function from the JSRunTime interface, you need to make sure that the name of the function you want to invoke is the same as the function name in the js file. Also whatever parameters are expected have to be included as well

The javascript file that comes with Blazor is part of the nuget package so if you want to see the contents you need to look at the repos for blazor server and blazor WASM as they each have their own.

Inject the IJSRuntime service in order to get the ability to invoke js functions

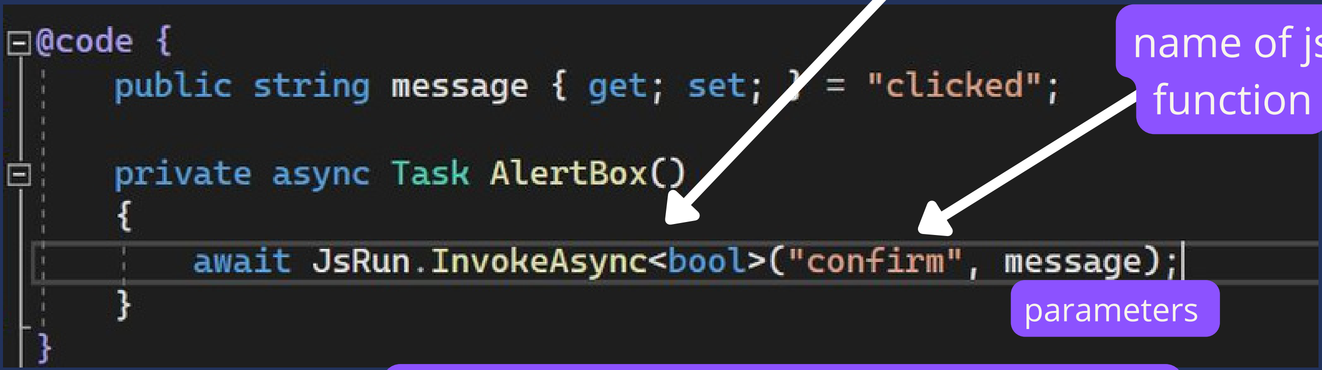


```
@page "/jsexample"
@inject IJSRuntime JsRun
<PageTitle>JS Interop</PageTitle>
```



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

by default these commands are asynchronous. InvokeAsync must be used and given an expected value, in this case bool



```
@code {
    public string message { get; set; } = "clicked";

    private async Task AlertBox()
    {
        await JsRun.InvokeAsync<bool>("confirm", message);
    }
}
```

name of js function

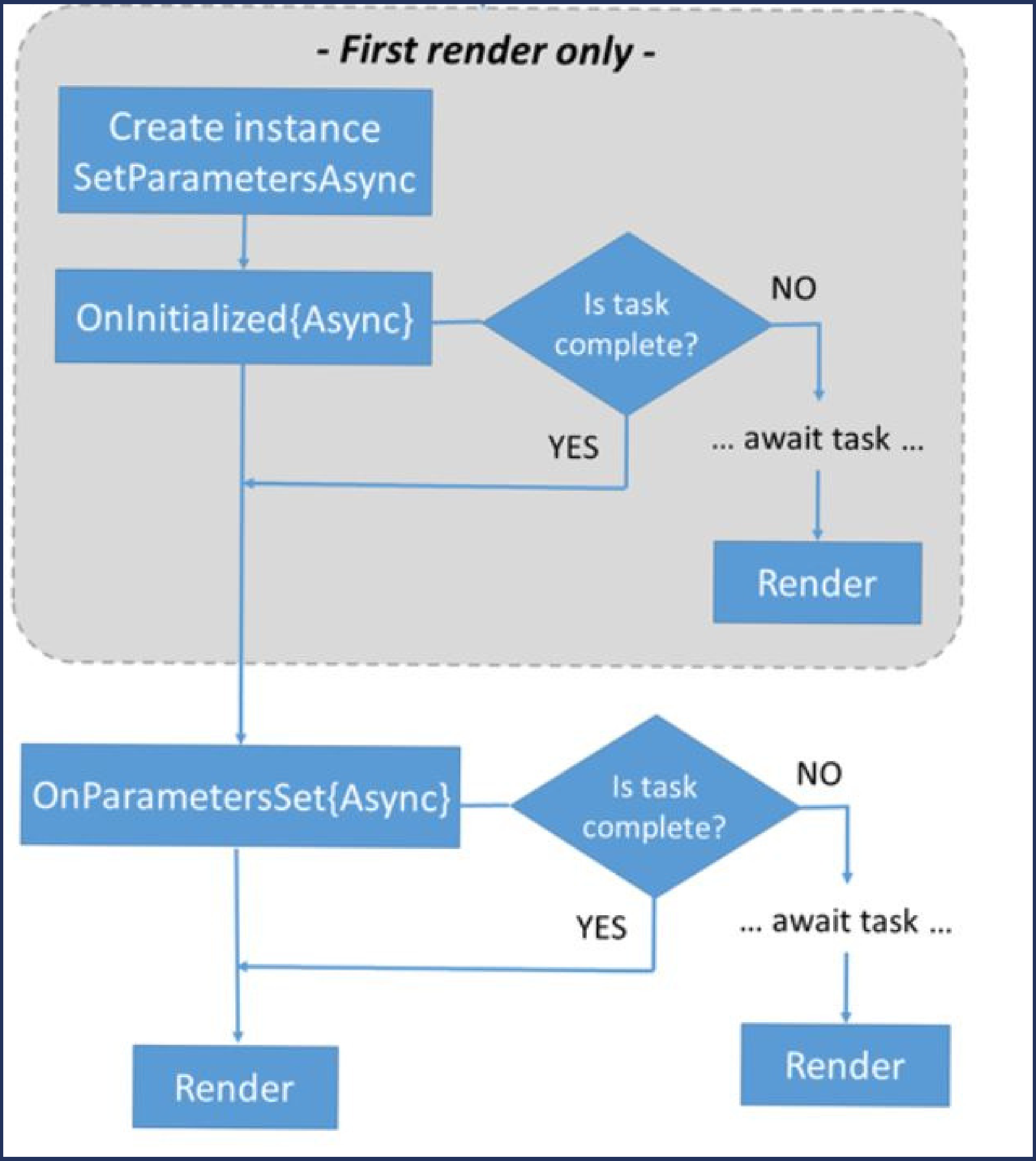
parameters

confirm is the name of a js function that triggers an Alert() that expects a string for the message

Life Cycle And Life Cycle Methods

- Every Component in Blazor has something call a Life Cycle. This determines what gets rendered when. You can even access Life Cycle methods in order to manipulate what happens during what step.
- Generally, you want to make sure you do your data gathering calls earlier in the lifecycle. And if you have any JS that manipulates the HTML in some way you must trigger those after you guarantee the HTML of the component has been rendered so you would have to trigger this later in the lifecycle.

Microsoft's Lifecycle Component Flow[1]



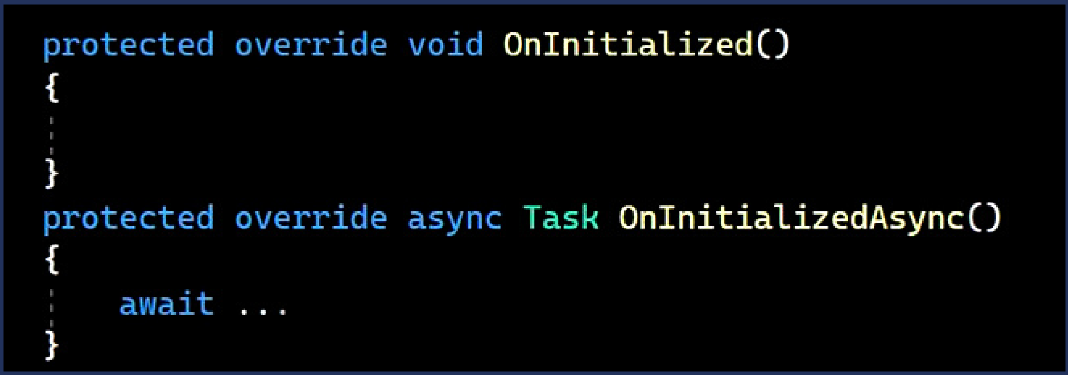
Life Cycle And Life Cycle Methods (Cont)

- Note This Order does not include the render tree.
- SetParameterAsync Will Trigger first whenever the parent renders. Then the rest of these methods are called and are more important in terms of how you can manipulate data and rendering for your component.
- There are Synchronous Calls and Asynchronous calls, the synchronous calls will be called first if you use both.

List Of LifeCycle Methods

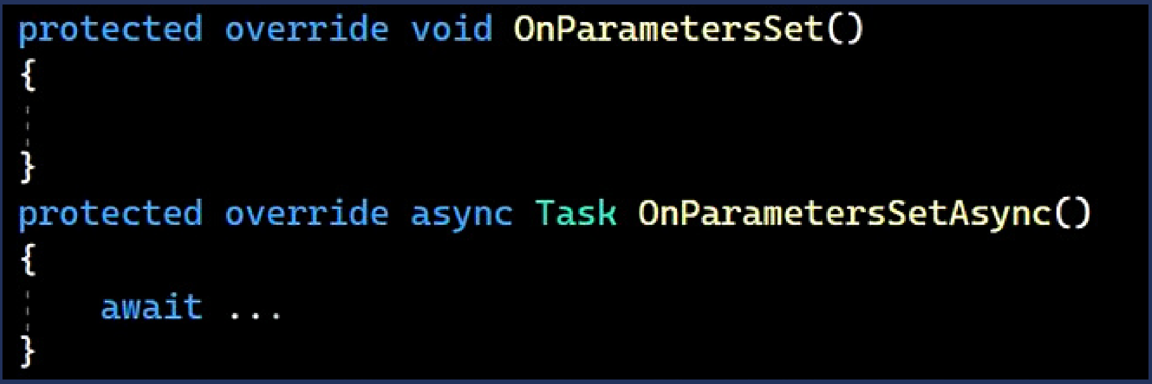
- OnInitialized()
- OninitializedAsync()
- OnParametersSet()
- OnParametersSetAsync()
- OnAfterRender()
- OnAfterRenderAsync()
- ShouldRender()
- StateHasChanged()

- OnInitialized will trigger once the component is loaded. Use this method in order to load data.



```
protected override void OnInitialized()
{

}
protected override async Task OnInitializedAsync()
{
    await ...
}
```

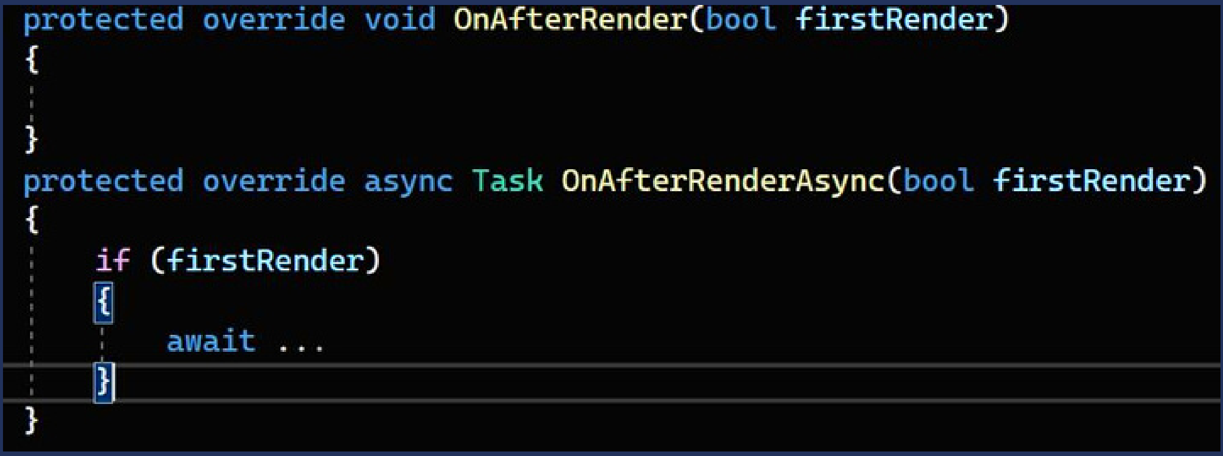
- Unlike OnInitialized, these do not wait for the component to completely render, instead they wait new parameters during initialization. They are then retriggered when new or updated parameters are received by a parent component



```
protected override void OnParametersSet()
{

}
protected override async Task OnParametersSetAsync()
{
    await ...
}
```

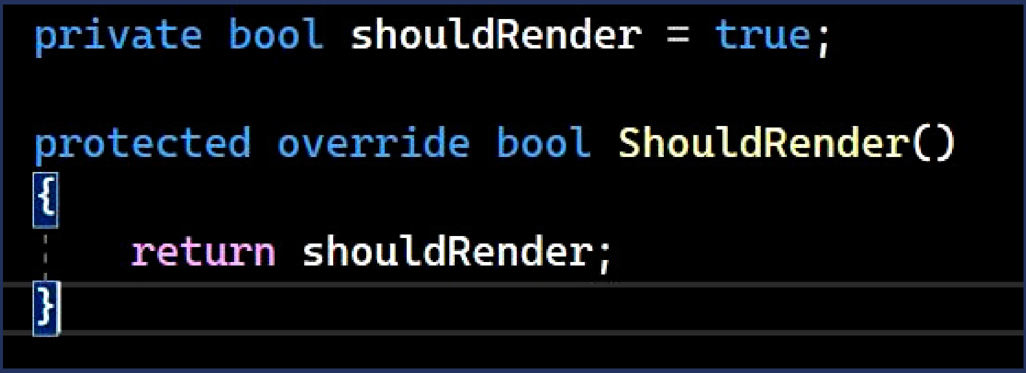
Life Cycle And Life Cycle Methods (Cont)

- Everytime the component re-renders and this finishes these will get triggered last. You should use this one in order to run your functions that require the component to be completely done rendering, aka using javascript libraries or functions that manipulate the DOM and need to make sure it exists.



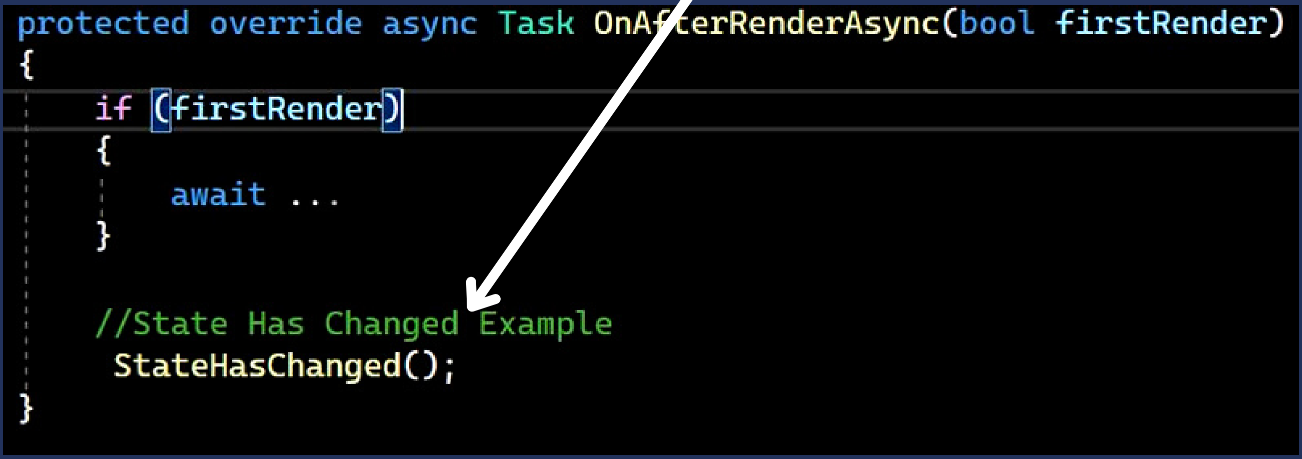
```
protected override void OnAfterRender(bool firstRender)
{

}
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        await ...
    }
}
```

- A way of optimizing Blazor rendering by manually forcing or denying the render of something like an identical output.



```
private bool shouldRender = true;

protected override bool ShouldRender()
{
    return shouldRender;
}
```

- A way of informing the component theres a change in state and that should be re-rendered. Useful when using something like Timers as whatever output is associated with them requires periodic re-rendering without user input.



```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        await ...
    }

    //State Has Changed Example
     StateHasChanged();
}
```

References

1) docs.microsoft.com

Useful And Informative Resources

0) Just Blazor Programming Youtube Channel (Yes Its My Channel)
https://www.youtube.com/channel/UC8EGEqngCWEfIvqNma5YsZQ

1) docs.microsoft.com

2) https://blazor-university.com/
(not my site but its very informative)

3) https://github.com/AdrienTorris/awesome-blazor
(A lot of blazor resources in one repo)