Developing Cross-Platform Web Apps With Blazor

Wael Kdouh - @waelkdouh

Senior Customer Engineer

v1.0

How to View This Presentation

- To switch to **Notes Page** view:
  - On the ribbon, click the **View** tab, and then click **Notes Page**
- To navigate through notes, use the Page Up and Page Down keys
  - Zoom in or zoom out, if required
- In the **Notes Page** view, you can:
  - Read any supporting text—now or after the delivery
  - Add notes to your copy of the presentation, if required
- Take the presentation files home with you

Module 3: Components and Layouts

Module Overview

Module 3: Components and Layouts

Section 1: Components

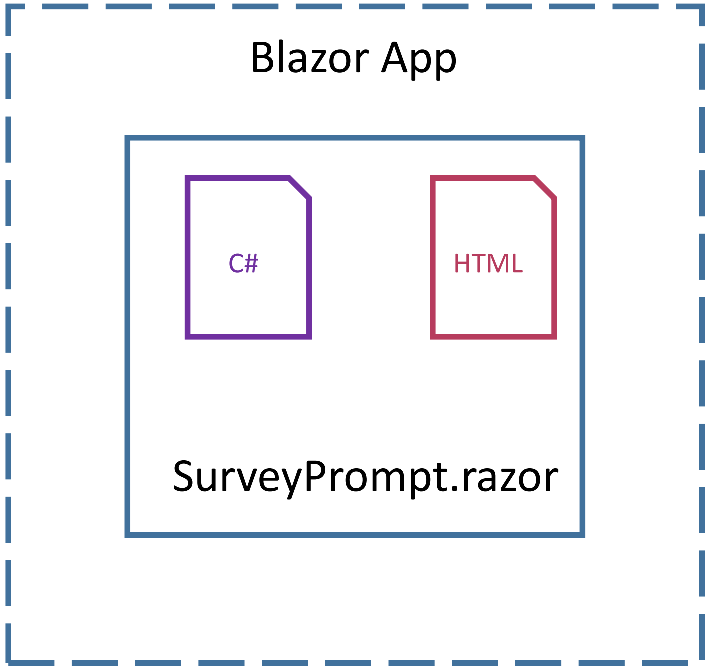
Lesson: Overview

Components

- Blazor apps are built using components
- A component is a self-contained chunk of user interface (UI), such as a page, dialog, or form
- A component includes HTML markup and the processing logic required to inject data or respond to UI events
- Components are flexible and lightweight. They can be nested, reused, and shared among projects

Implementing Components

- Components are implemented in Razor component files (.razor) using a combination of C# and HTML markup



SurveyPrompt.razor

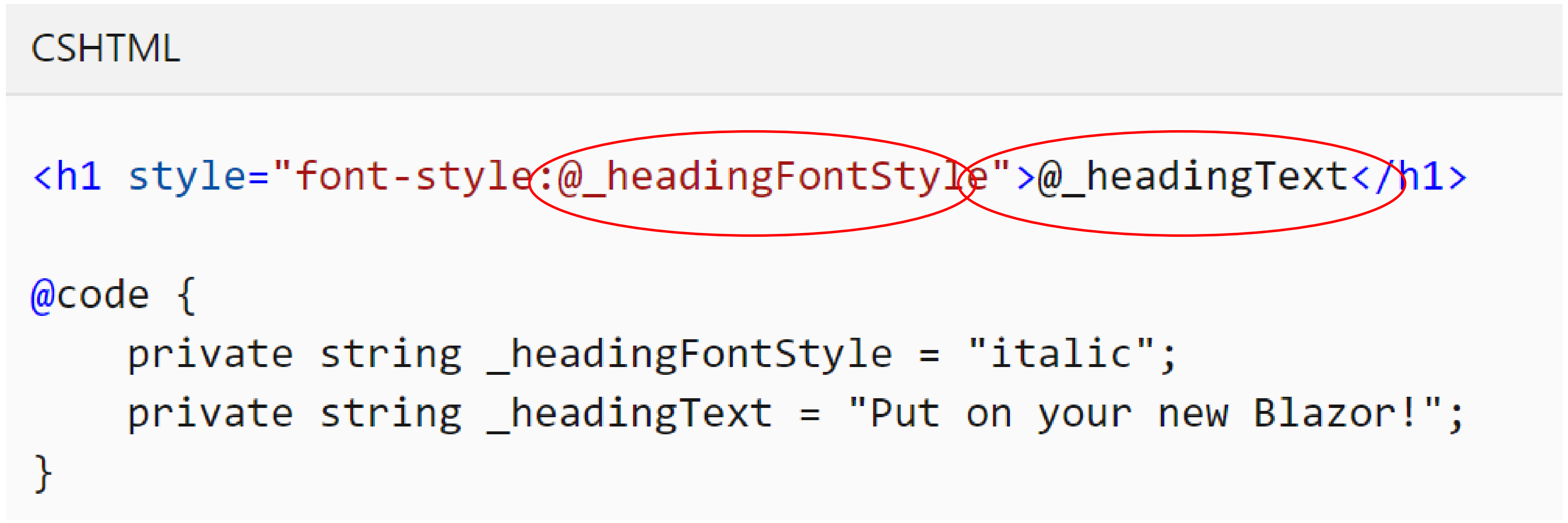
```
<div class="alert alert-secondary mt-4" role="alert">
    <span class="oi oi-pencil mr-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold"
           href="https://go.microsoft.com/fwlink/?linkid=2109206">
            brief survey
        </a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string Title { get; set; }
}
```

Implementing Components

- A component's name must start with an uppercase character. For example, MyCoolComponent.razor is valid, and myCoolComponent.razor is invalid
- The UI for a component is defined using HTML. Dynamic rendering logic (for example, loops, conditionals, expressions) is added using an embedded C# syntax called Razor
- When an app is compiled, the HTML markup and C# rendering logic are converted into a component class. The name of the generated class matches the name of the file

Implementing Components

- Members of the component class are defined in an @code block. In the @code block, component state (properties, fields) is specified with methods for event handling or for defining other component logic. More than one @code block is permissible
- Component members can be used as part of the component's rendering logic using C# expressions that start with @
- For example, a C# field is rendered by prefixing @ to the field name. The following example evaluates and renders:
  - _headingFontStyle to the CSS property value for font-style
  - _headingText to the content of the <h1> element



CSHTML

```cshtml
<h1 style="font-style:@_headingFontStyle">@_headingText</h1>

@code {
    private string _headingFontStyle = "italic";
    private string _headingText = "Put on your new Blazor!";
}
```

Using Components

- Components can include other components by declaring them using HTML element syntax

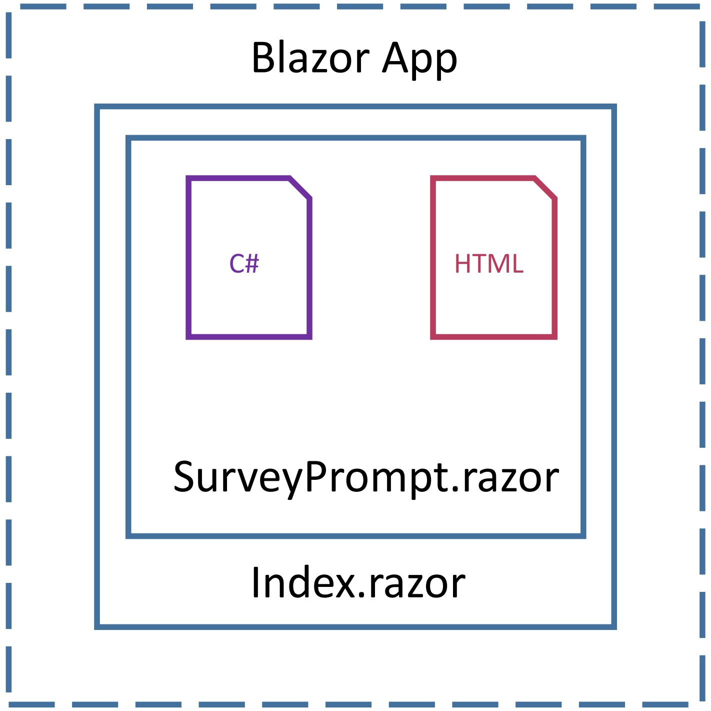
- The markup for using a component looks like an **HTML tag** where the **name of the tag is the component type**

Using Components

The following markup in Index.razor renders a SurveyPrompt instance

Index.razor

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```



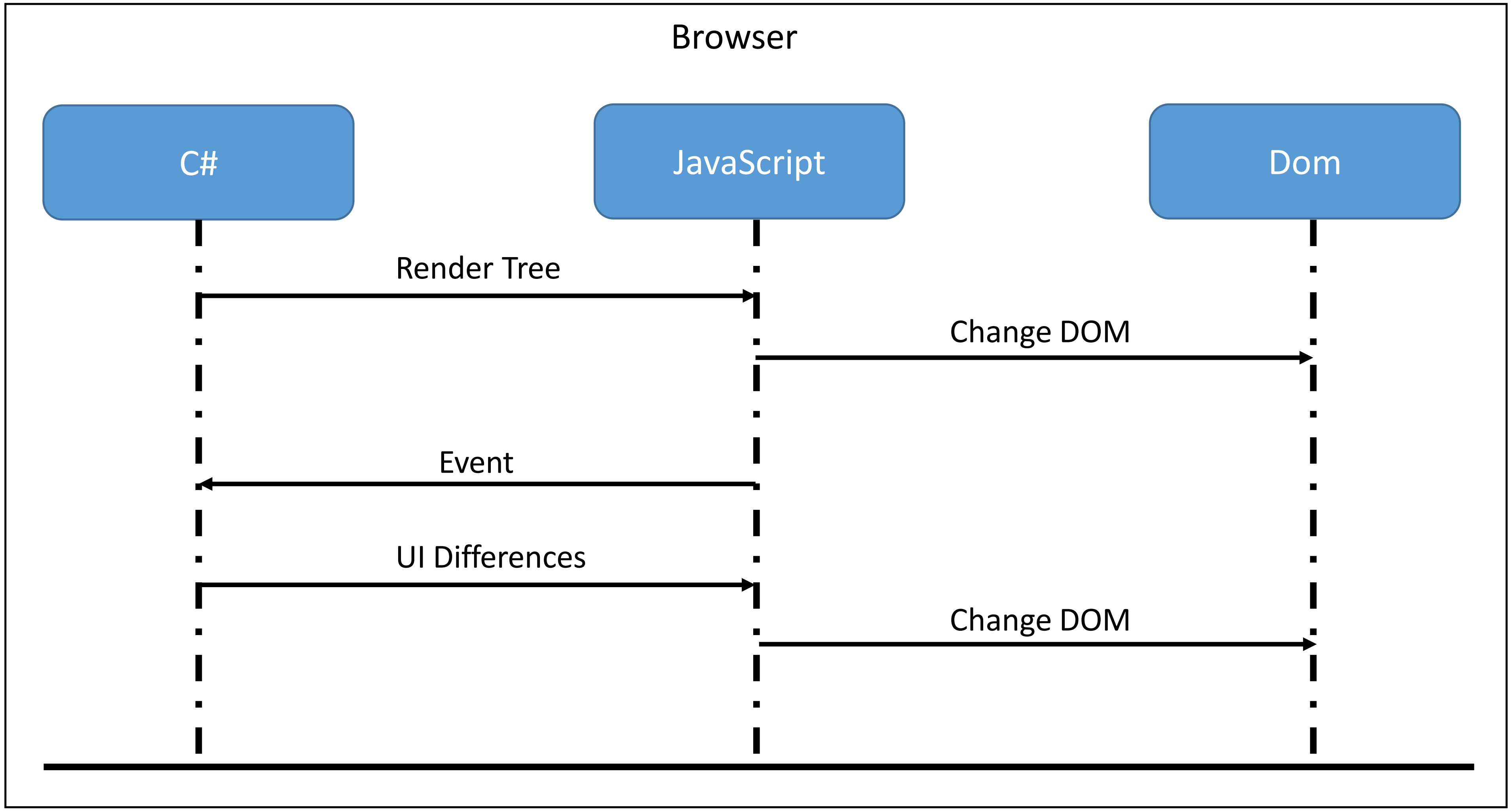
SurveyPrompt.razor

```
<div class="alert alert-secondary mt-4" role="alert">
    <span class="oi oi-pencil mr-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold"
           href="https://go.microsoft.com/fwlink/?linkid=2109206">
            brief survey
        </a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string Title { get; set; }
}
```

Rendering Components

- After the component is initially rendered, the component regenerates its render tree in response to events
- Blazor then compares the new render tree against the previous one and applies any modifications to the browser's Document Object Model (DOM)

Rendering Components



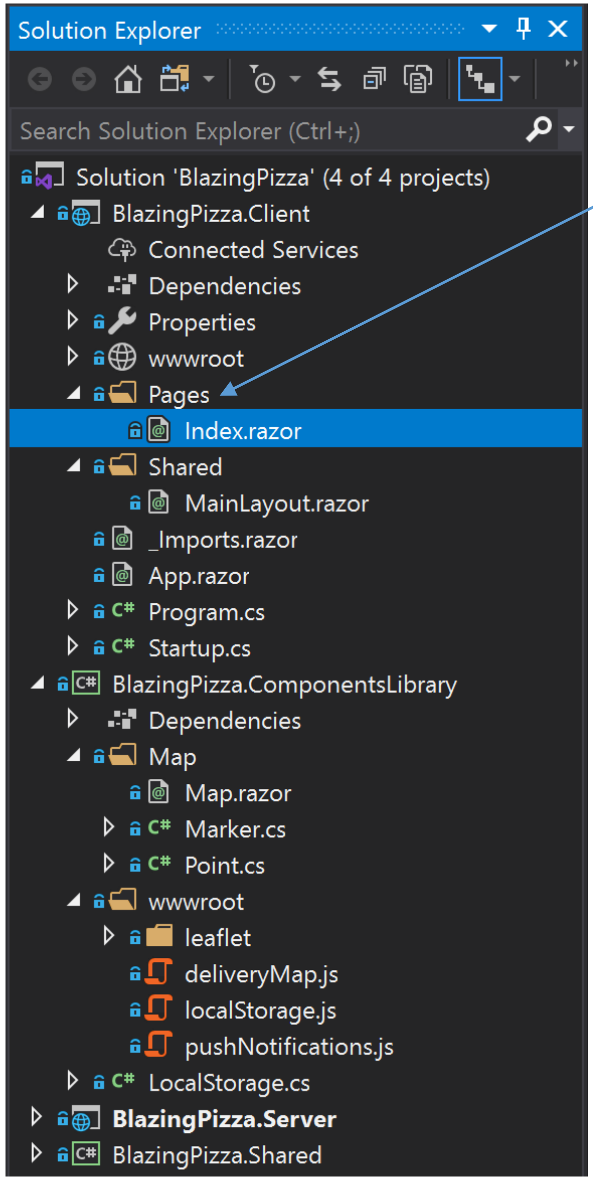
Organizing Components

- Components are ordinary C# classes and can be placed anywhere within a project
- Components that produce webpages usually reside in the Pages folder
- Non-page components are frequently placed in the Shared folder or a custom folder added to the project
- To use a custom folder, add the custom folder's namespace to either the parent component or to the app's _Imports.razor file
    - For example, the following namespace makes components in a Components folder available when the app's root namespace is WebApplication:

razor

```razor
@using WebApplication.Components
```

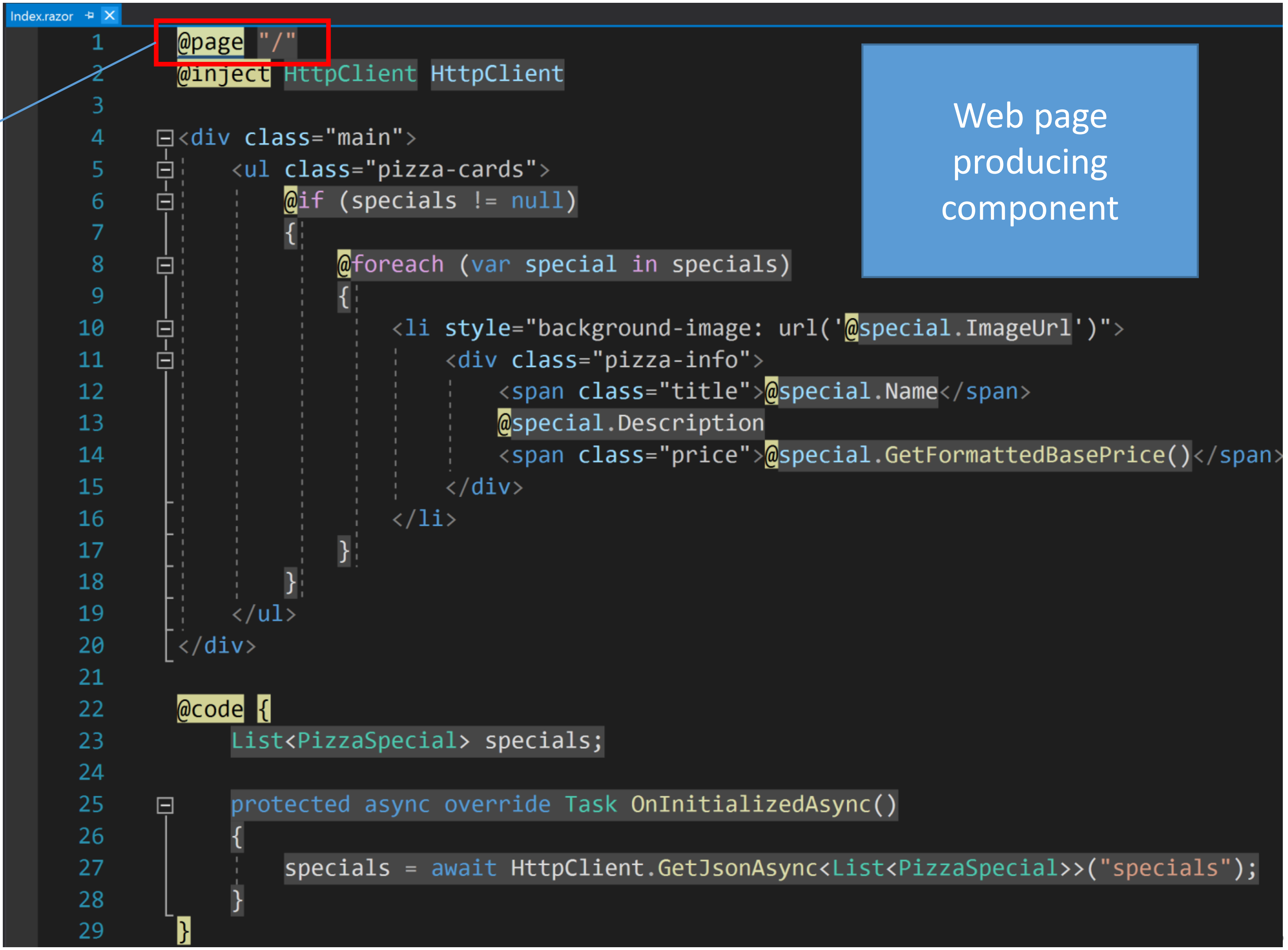
Organizing Components



Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'BlazingPizza' (4 of 4 projects)
  BlazingPizza.Client
      Connected Services
      Dependencies
      Properties
      wwwroot
      Pages
          Index.razor
      Shared
          MainLayout.razor
      _Imports.razor
      App.razor
      Program.cs
      Startup.cs
  BlazingPizza.ComponentsLibrary
      Dependencies
      Map
          Map.razor
          Marker.cs
          Point.cs
      wwwroot
          leaflet
          deliveryMap.js
          localStorage.js
          pushNotifications.js
      LocalStorage.cs
  BlazingPizza.Server
  BlazingPizza.Shared
```

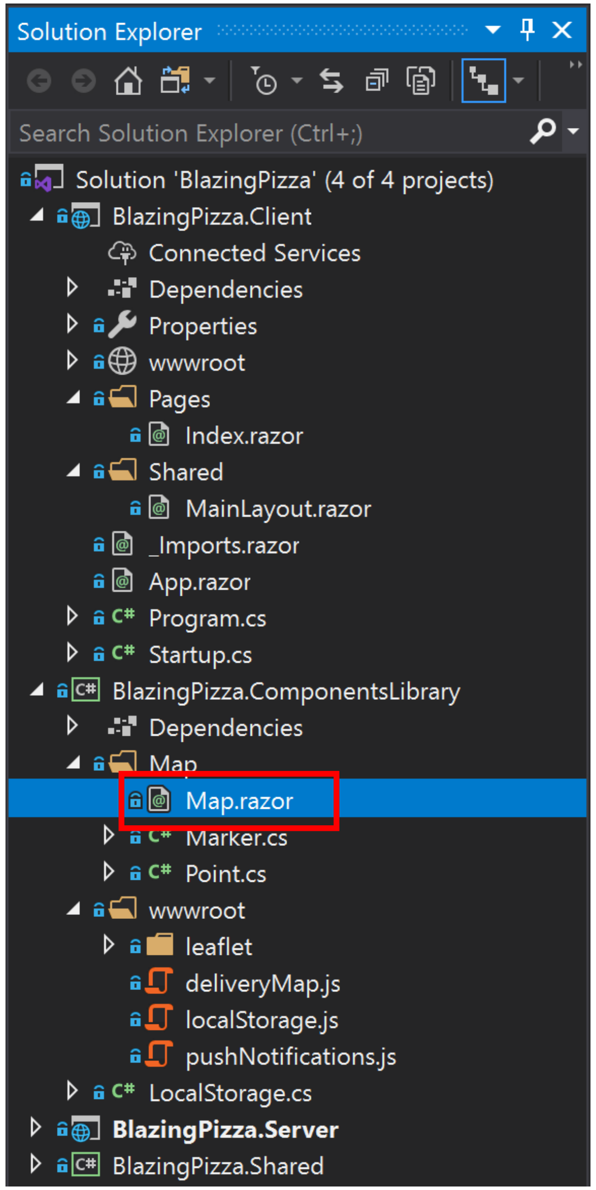


Index.razor

```
@page "/"
@inject HttpClient HttpClient

<div class="main">
    <ul class="pizza-cards">
        @if (specials != null)
        {
            @foreach (var special in specials)
            {
                <li style="background-image: url('@special.ImageUrl')">
                    <div class="pizza-info">
                        <span class="title">@special.Name</span>
                        @special.Description
                        <span class="price">@special.GetFormattedBasePrice()</span>
                    </div>
                </li>
            }
        }
    </ul>
</div>

@code {
    List<PizzaSpecial> specials;

    protected async override Task OnInitializedAsync()
    {
        specials = await HttpClient.GetJsonAsync<List<PizzaSpecial>>("specials");
    }
}
```

Web page producing component

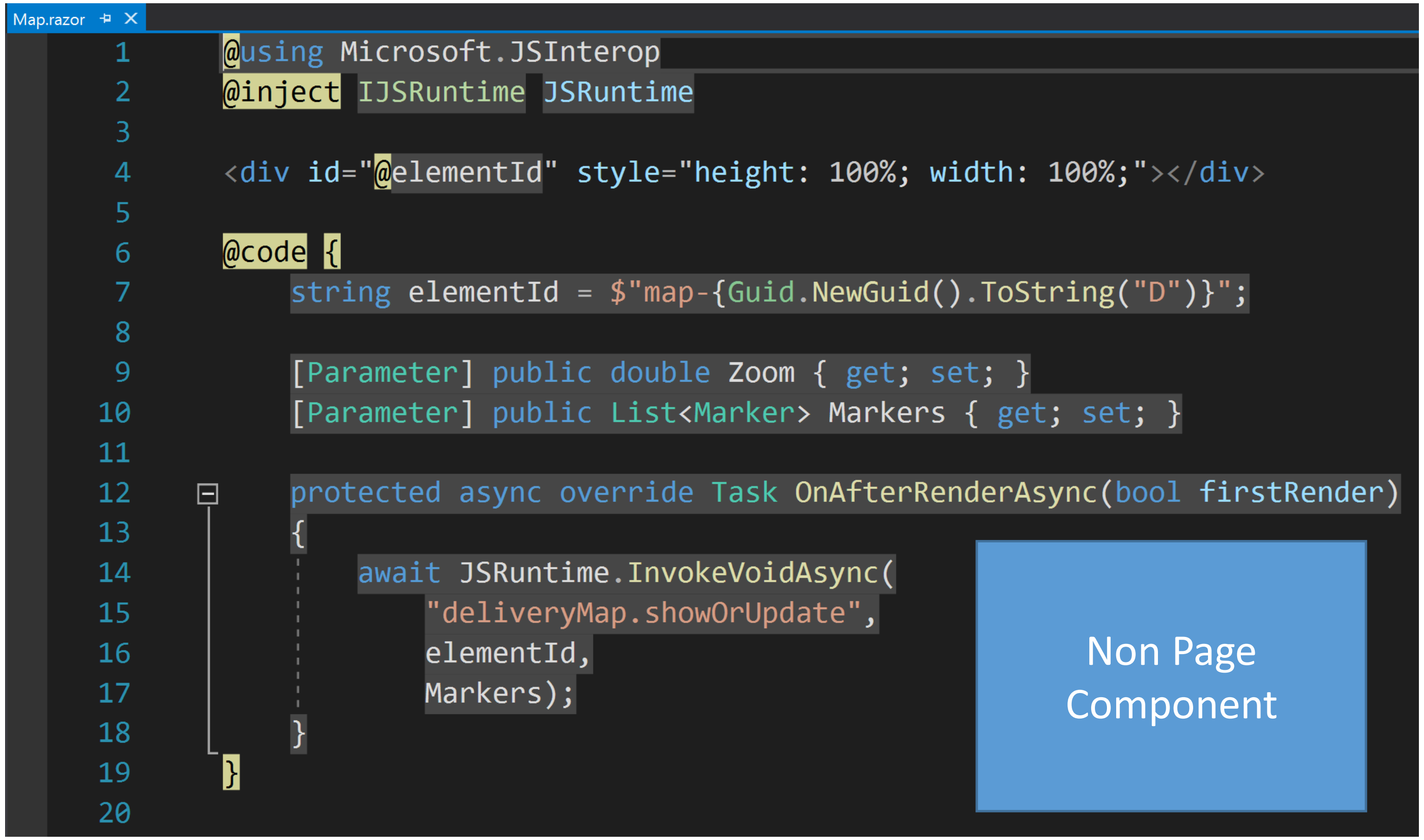
Organizing Components



Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'BlazingPizza' (4 of 4 projects)
  BlazingPizza.Client
      Connected Services
      Dependencies
      Properties
      wwwroot
      Pages
        Index.razor
      Shared
        MainLayout.razor
      _Imports.razor
      App.razor
      Program.cs
      Startup.cs
  BlazingPizza.ComponentsLibrary
      Dependencies
      Map
        Map.razor
        Marker.cs
        Point.cs
      wwwroot
        leaflet
        deliveryMap.js
        localStorage.js
        pushNotifications.js
      LocalStorage.cs
  BlazingPizza.Server
  BlazingPizza.Shared
```



Map.razor

```
@using Microsoft.JSInterop
@inject IJSRuntime JSRuntime

<div id="@elementId" style="height: 100%; width: 100%;"></div>

@code {
    string elementId = $"map-{Guid.NewGuid().ToString("D")}";

    [Parameter] public double Zoom { get; set; }
    [Parameter] public List<Marker> Markers { get; set; }

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        await JSRuntime.InvokeVoidAsync(
            "deliveryMap.showOrUpdate",
            elementId,
            Markers);
    }
}
```

Non Page Component

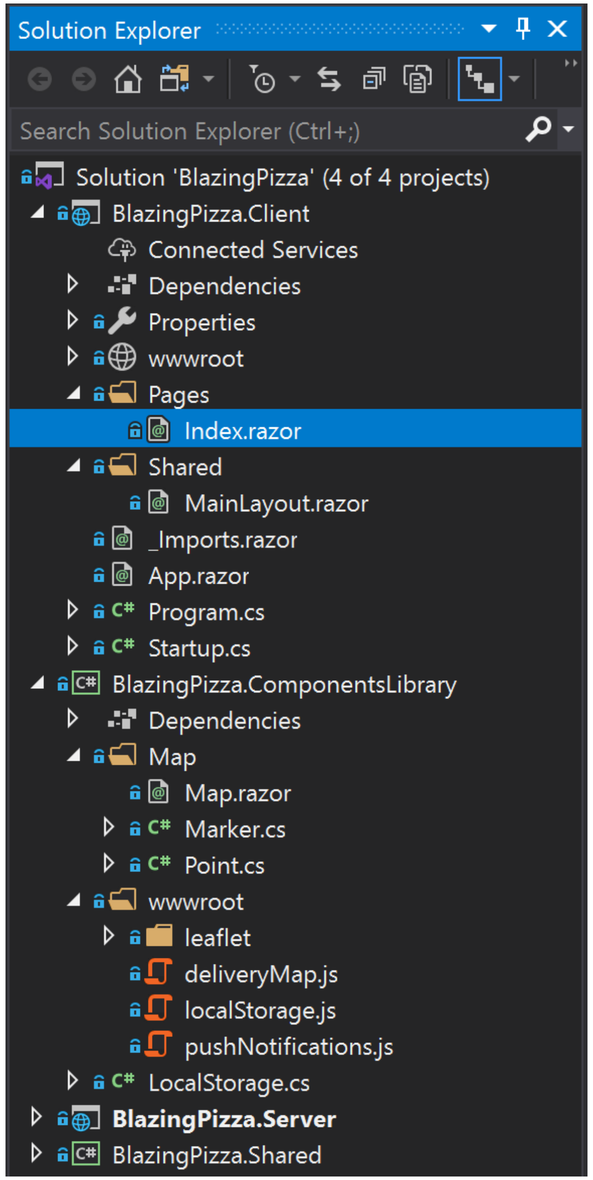
Demo: Web Page Vs Non Page Components

Module 3: Components and Layouts

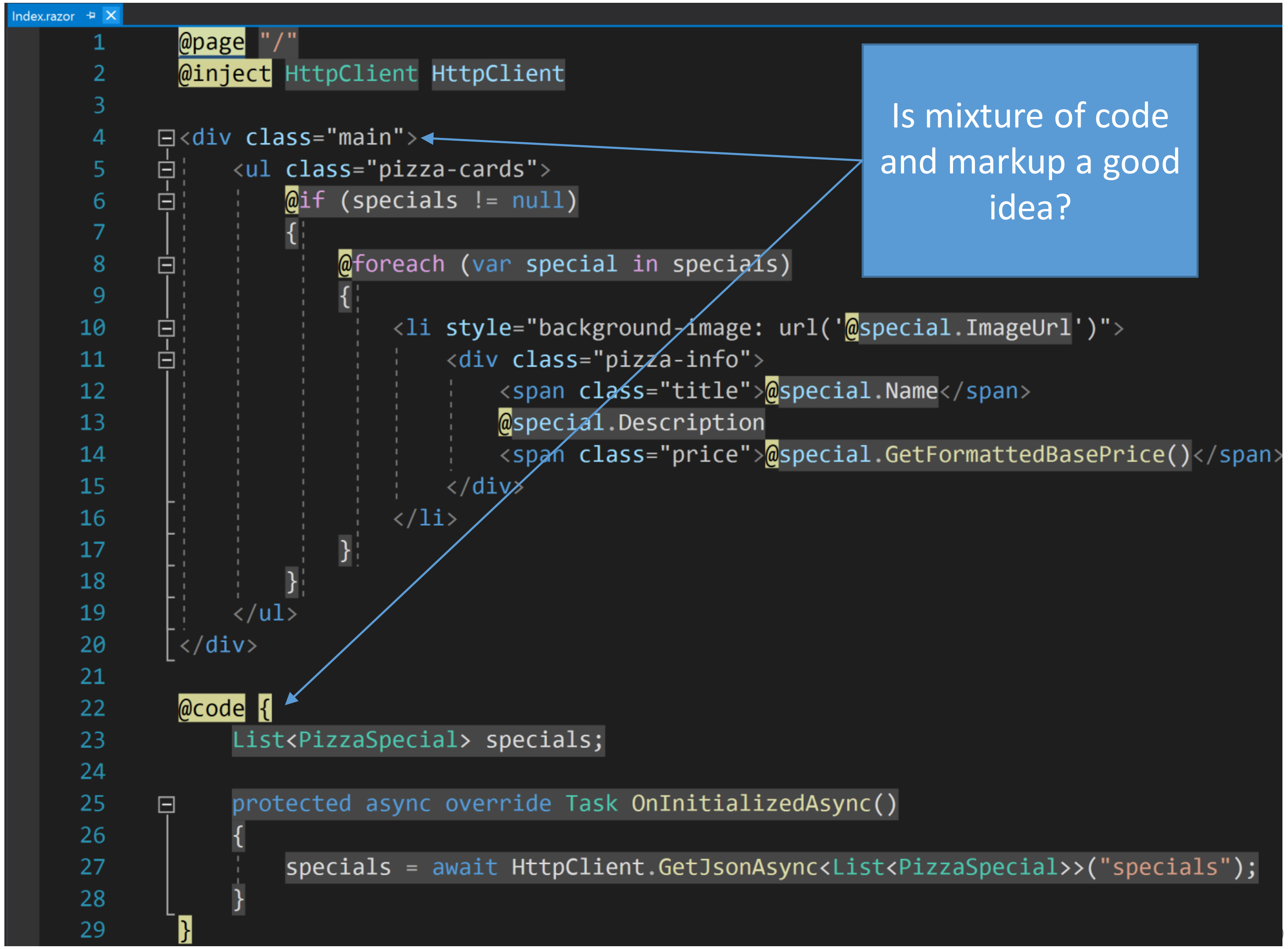
Section 1: Components

Lesson: **Component Code-Behind Approach**

Non Code-Behind Approach



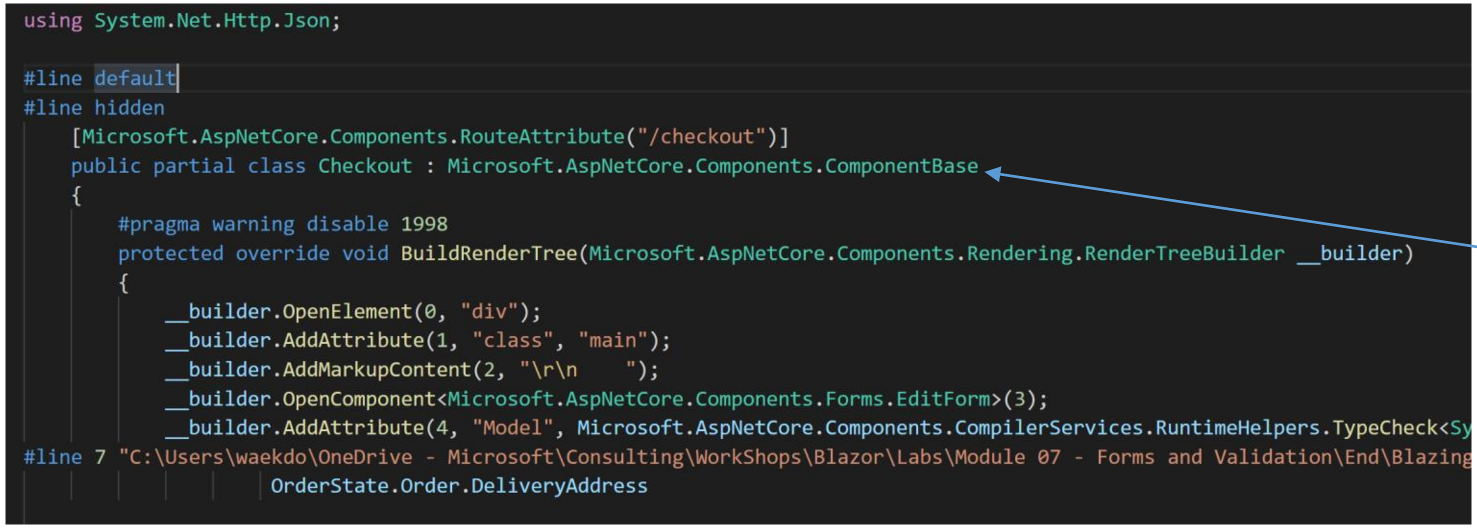
```
Solution 'BlazingPizza' (4 of 4 projects)
  BlazingPizza.Client
      Connected Services
      Dependencies
      Properties
      wwwroot
      Pages
          Index.razor
      Shared
          MainLayout.razor
      _Imports.razor
      App.razor
      Program.cs
      Startup.cs
  BlazingPizza.ComponentsLibrary
      Dependencies
      Map
          Map.razor
          Marker.cs
          Point.cs
      wwwroot
          leaflet
          deliveryMap.js
          localStorage.js
          pushNotifications.js
      LocalStorage.cs
  BlazingPizza.Server
  BlazingPizza.Shared
```



```
@page "/"
@inject HttpClient HttpClient

<div class="main">
    <ul class="pizza-cards">
        @if (specials != null)
        {
            @foreach (var special in specials)
            {
                <li style="background-image: url('@special.ImageUrl')">
                    <div class="pizza-info">
                        <span class="title">@special.Name</span>
                        @special.Description
                        <span class="price">@special.GetFormattedBasePrice()</span
                    </div>
                </li>
            }
        }
    </ul>
</div>

@code {
    List<PizzaSpecial> specials;

    protected async override Task OnInitializedAsync()
    {
        specials = await HttpClient.GetJsonAsync<List<PizzaSpecial>>("specials");
    }
}
```

Is mixture of code and markup a good idea?

Non Code-Behind Approach



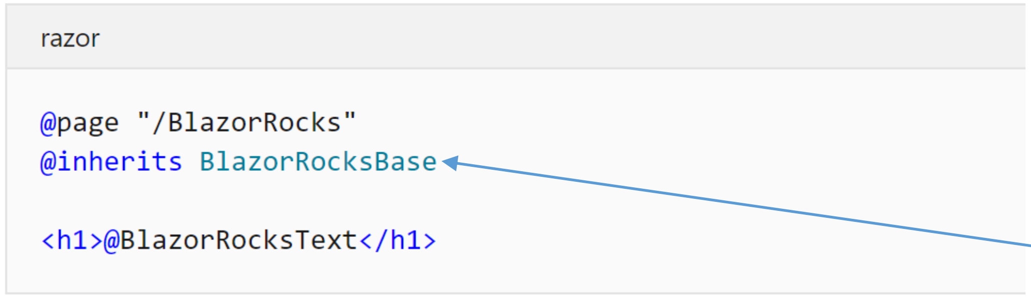
```
using System.Net.Http.Json;

#line default
#line hidden
    [Microsoft.AspNetCore.Components.RouteAttribute("/checkout")]
    public partial class Checkout : Microsoft.AspNetCore.Components.ComponentBase
    {
        #pragma warning disable 1998
        protected override void BuildRenderTree(Microsoft.AspNetCore.Components.Rendering.RenderTreeBuilder __builder)
        {
            __builder.OpenElement(0, "div");
            __builder.AddAttribute(1, "class", "main");
            __builder.AddMarkupContent(2, "\r\n    ");
            __builder.OpenComponent<Microsoft.AspNetCore.Components.Forms.EditForm>(3);
            __builder.AddAttribute(4, "Model", Microsoft.AspNetCore.Components.CompilerServices.RuntimeHelpers.TypeCheck<Sy
#line 7 "C:\Users\waekdo\OneDrive - Microsoft\Consulting\WorkShops\Blazor\Labs\Module 07 - Forms and Validation\End\Blazing
                        OrderState.Order.DeliveryAddress
```

Non Code-Behind approach generates a partial class for each component which derives from ComponentBase

Code-Behind Approach

- C# code is placed in a base class and then the @inherits directive can be used to specify a base class for a component

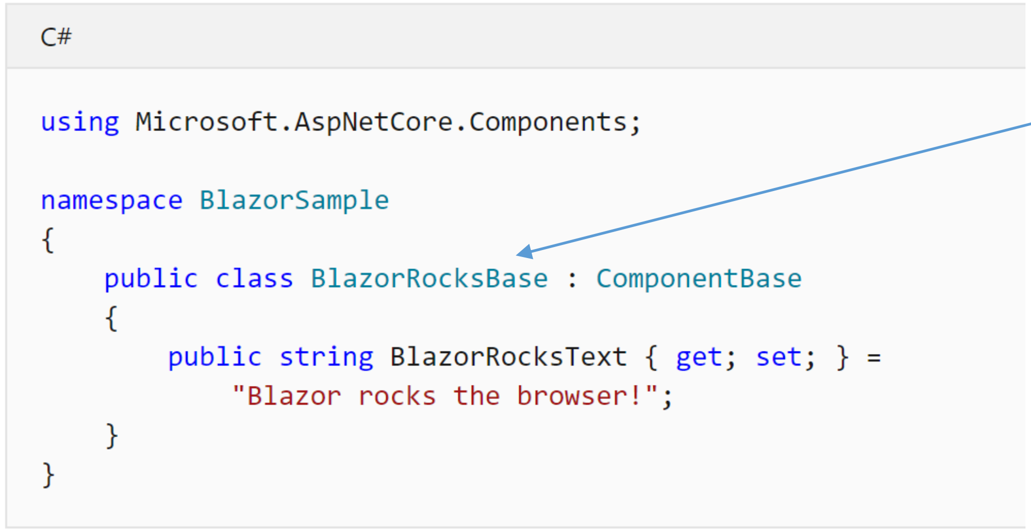
Pages/BlazorRocks.razor:



razor

```razor
@page "/BlazorRocks"
@inherits BlazorRocksBase

<h1>@BlazorRocksText</h1>
```

BlazorRocksBase.cs:



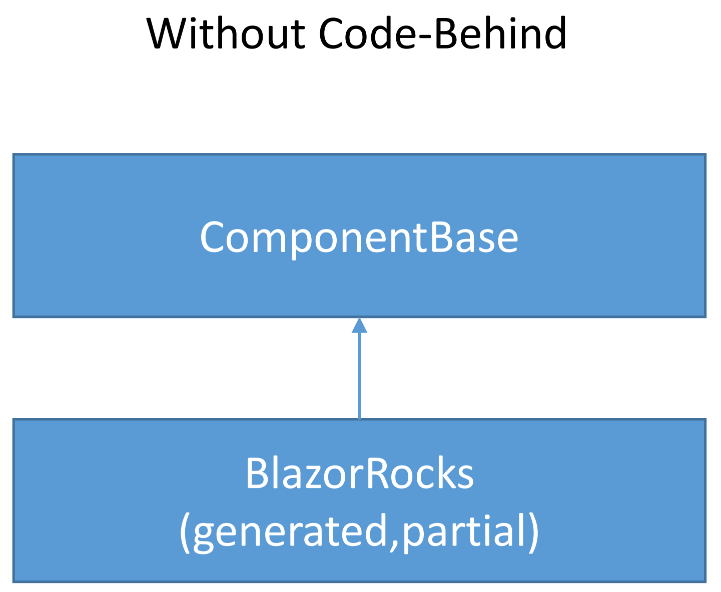
C#

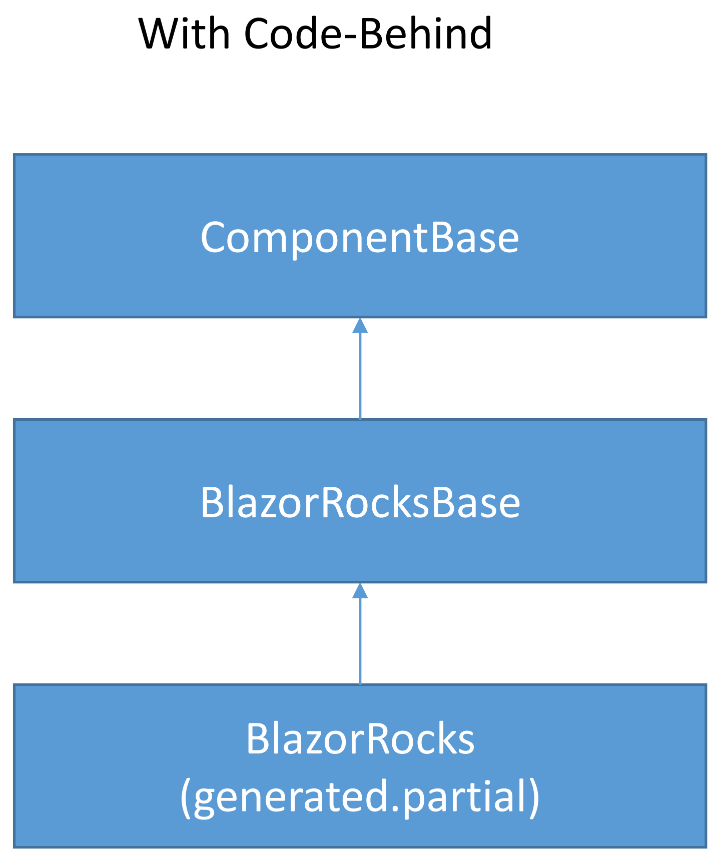
```csharp
using Microsoft.AspNetCore.Components;

namespace BlazorSample
{
    public class BlazorRocksBase : ComponentBase
    {
        public string BlazorRocksText { get; set; } =
            "Blazor rocks the browser!";
    }
}
```

Code-Behind classes have base as a suffix because they are the base class for the class generated from the razor file instead of ComponentBase

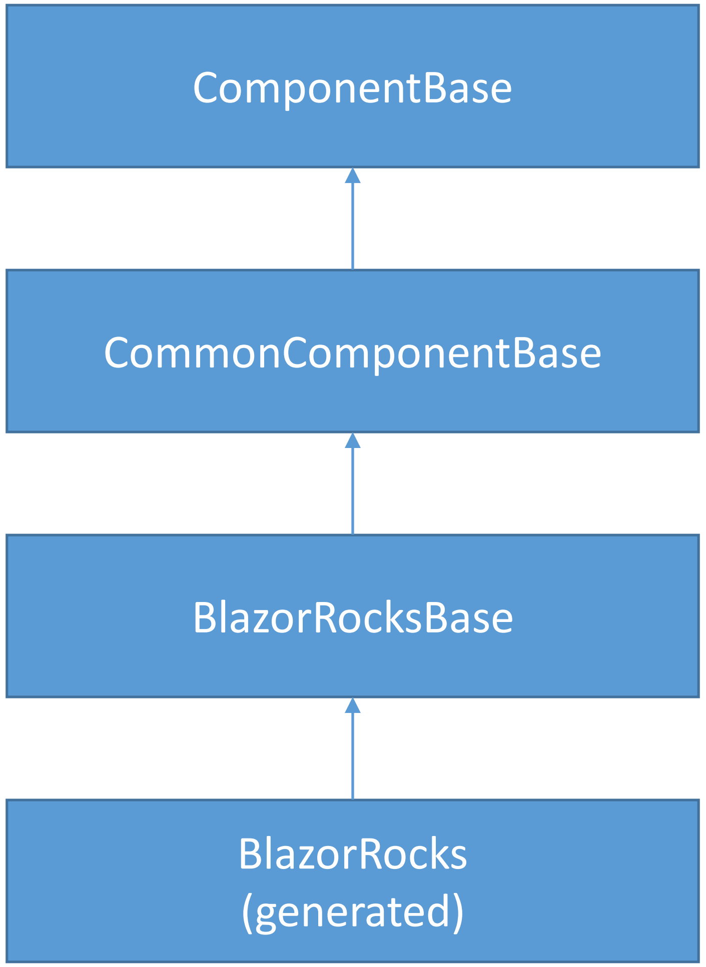
inside your razor page you inherit from the base class using @inherits directive

Partial Component Class Hierarchy With/Without Code-Behind





Advantage Of Using Code-Behind



Demo: Component Code-Behind Approach

Module 3: Components and Layouts

Section 1: Components

Lesson: **Component Parameters**

Component Parameters

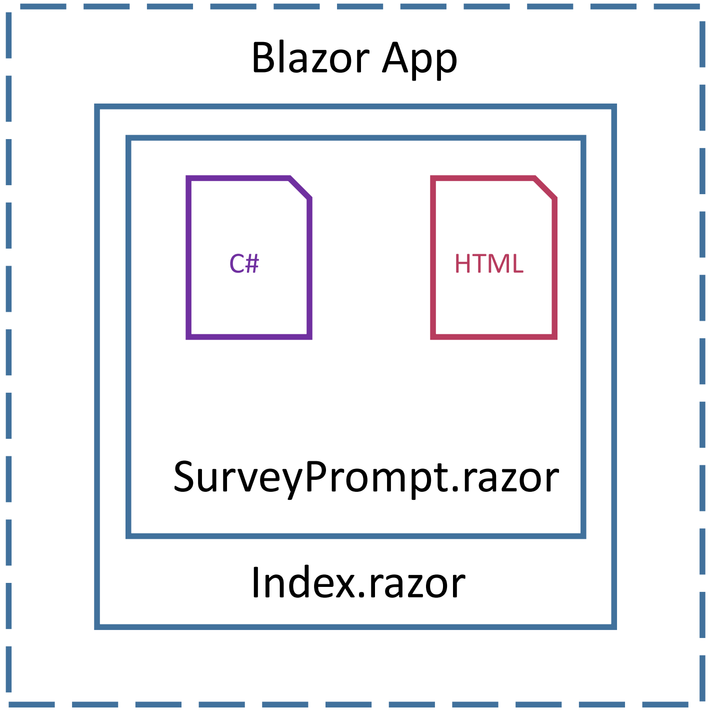
- Components can have parameters, which are defined using public properties on the component class with the [Parameter] attribute

- Use [Parameter] attributes to specify arguments for a component in markup

Component Parameters

Index.razor

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```



SurveyPrompt.razor

```
<div class="alert alert-secondary mt-4" role="alert">
    <span class="oi oi-pencil mr-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold"
           href="https://go.microsoft.com/fwlink/?linkid=2109206">
            brief survey
        </a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string Title { get; set; }
}
```

Module 3: Components and Layouts

Section 1: Components

Lesson: Templated Components

Templated Components

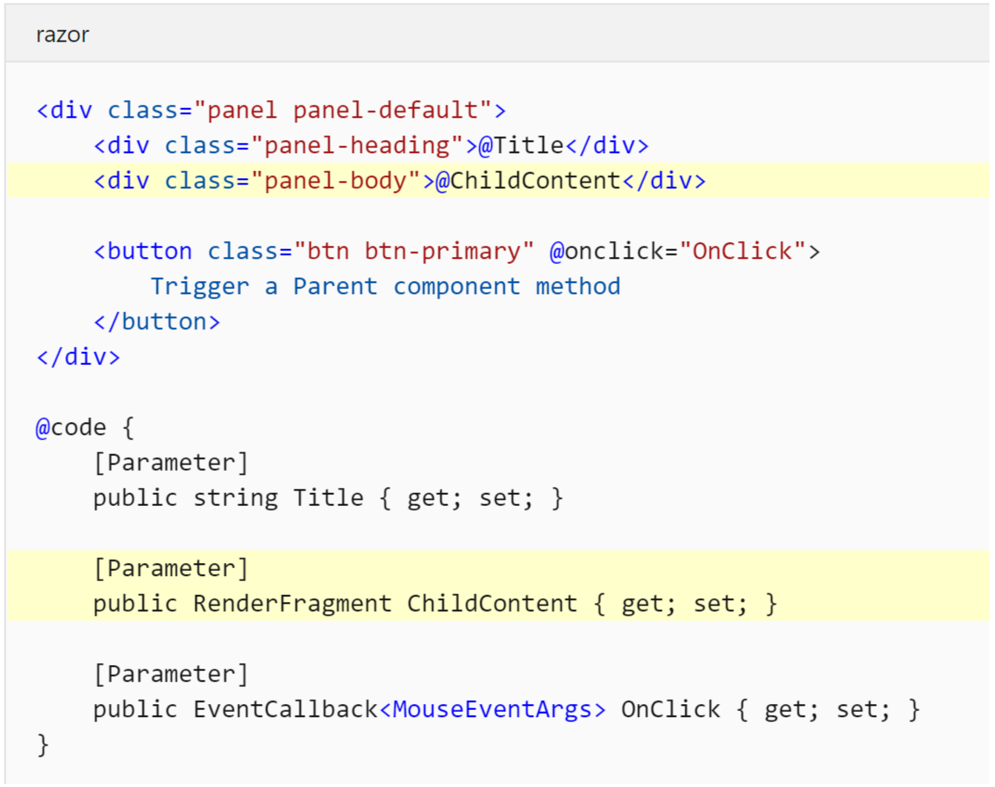
- Simple components can be configured through properties, but more complex components often need more rendering flexibility, and templates are a canonical way to achieve that

- Templated components expose customizable sections via parameters

- Parent components (consumers) then pass in their own templates for these sections that the child component (templated component) will use when rendering

Templated Components – Simple Use Case

Components/ChildComponent.razor:

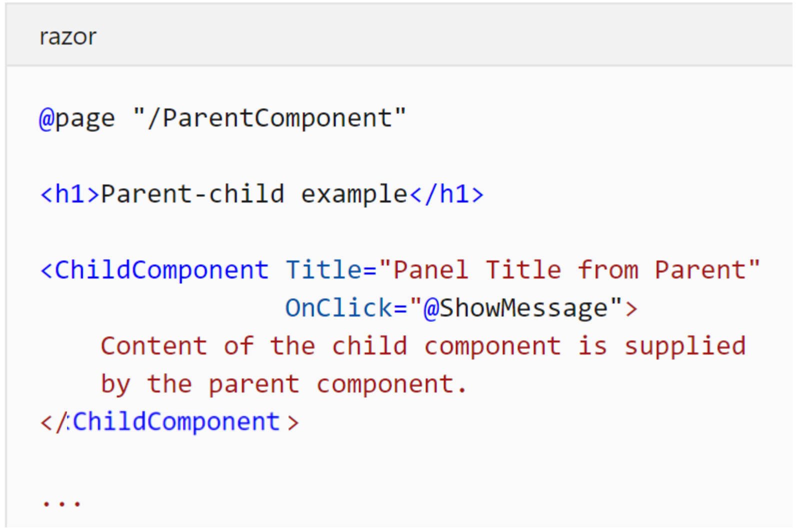


razor

```razor
<div class="panel panel-default">
    <div class="panel-heading">@Title</div>
    <div class="panel-body">@ChildContent</div>

    <button class="btn btn-primary" @onclick="OnClick">
        Trigger a Parent component method
    </button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClick { get; set; }
}
```

The property receiving the RenderFragment content must be named ChildContent by convention

Pages/ParentComponent.razor:



razor

```razor
@page "/ParentComponent"

<h1>Parent-child example</h1>

<ChildComponent Title="Panel Title from Parent"
                OnClick="@ShowMessage">
    Content of the child component is supplied
    by the parent component.
</ChildComponent>

...
```

The ParentComponent can provide content for rendering the ChildComponent by placing the content inside the <ChildComponent> tags

Templated Components – Advanced Use Case

Components/ChildComponent.razor:

```
@typeparam TItem

<div style="border: solid 4px #111;">
    <div class="table-responsive">
        <table class="table table-hover">
            <thead>
                <tr>
                    @if (HeaderTemplate != null)
                    {
                        @HeaderTemplate
                    }
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Items)
                {
                    <tr>
                        @RowTemplate(item)
                    </tr>
                }
            </tbody>
            <tfoot>
                <tr>
                    @if (FooterTemplate != null)
                    {
                        @FooterTemplate(Items)
                    }
                </tr>
            </tfoot>
        </table>
    </div>
</div>

@code {
    [Parameter]
    public RenderFragment HeaderTemplate { get; set; }

    [Parameter]
    public RenderFragment<TItem> RowTemplate { get; set; }

    [Parameter]
    public RenderFragment<IList<TItem>> FooterTemplate { get; set; }

    [Parameter]
    public IList<TItem> Items { get; set; }
}
```

Pages/ParentComponent.razor:

```
@page "/parent"
<h3>List of countries</h3>

<ChildComponent Items="@Countries" TItem="Country">
    <HeaderTemplate>
        <th>Name</th>
        <th>Capital</th>
    </HeaderTemplate>
    <RowTemplate>
        <td>@context.CountryName</td>
        <td>@context.Capital</td>
    </RowTemplate>
    <FooterTemplate>
        <td colspan="2">
            @context.Count countries found.
        </td>
    </FooterTemplate>
</ChildComponent>


@code {
    private List<Country> Countries = new List<Country>
    {
        new Country()
        {
            CountryName = "USA",
            Capital = "Washington, D.C."
        },

        new Country()
        {
            CountryName = "Lebanon",
            Capital = "Beirut"
        }
    };

}
```

Templated Components – Advanced Use Case

Components/ChildComponent.razor:

```
@typeparam TItem

<div style="border: solid 4px #111;">
    <div class="table-responsive">
        <table class="table table-hover">
            <thead>
                <tr>
                    @if (HeaderTemplate != null)
                    {
                        @HeaderTemplate
                    }
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Items)
                {
                    <tr>
                        @RowTemplate(item)
                    </tr>
                }
            </tbody>
            <tfoot>
                <tr>
                    @if (FooterTemplate != null)
                    {
                        @FooterTemplate(Items)
                    }
                </tr>
            </tfoot>
        </table>
    </div>
</div>

@code {
    [Parameter]
    public RenderFragment HeaderTemplate { get; set; }

    [Parameter]
    public RenderFragment<TItem> RowTemplate { get; set; }

    [Parameter]
    public RenderFragment<IList<TItem>> FooterTemplate { get; set; }

    [Parameter]
    public IList<TItem> Items { get; set; }
}
```

Use @typeparam directive to make the template-based components bound to a generic data type

The same grid component can realistically be used to present, search and page different collections of data

A Blazor template is an instance of the RenderFragment type

Put another way, it's a chunk of markup being rendered by the Razor view engine that can be treated like a plain instance of a .NET type

Templated Components – Advanced Use Case

Components/ChildComponent.razor:

```
@typeparam TItem

<div style="border: solid 4px #111;">
    <div class="table-responsive">
        <table class="table table-hover">
            <thead>
                <tr>
                    @if (HeaderTemplate != null)
                    {
                        @HeaderTemplate
                    }
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Items)
                {
                    <tr>
                        @RowTemplate(item)
                    </tr>
                }
            </tbody>
            <tfoot>
                <tr>
                    @if (FooterTemplate != null)
                    {
                        @FooterTemplate(Items)
                    }
                </tr>
            </tfoot>
        </table>
    </div>
</div>

@code {
    [Parameter]
    public RenderFragment HeaderTemplate { get; set; }

    [Parameter]
    public RenderFragment<TItem> RowTemplate { get; set; }

    [Parameter]
    public RenderFragment<IList<TItem>> FooterTemplate { get; set; }

    [Parameter]
    public IList<TItem> Items { get; set; }
}
```

A template can be made optional to implement in a client page, by wrapping its call with a plain check for existence before use

Templated Components – Advanced Use Case

TItem will match the name of the type parameter as declared through the @typeparam directive within the child component

Most templates are parameter-less, but you can make them generic, too

In this example the header template is parameter-less, but the row and the footer templates are generic

When rendering a parametric template, you use the “context” implicit name to reference the argument of the template

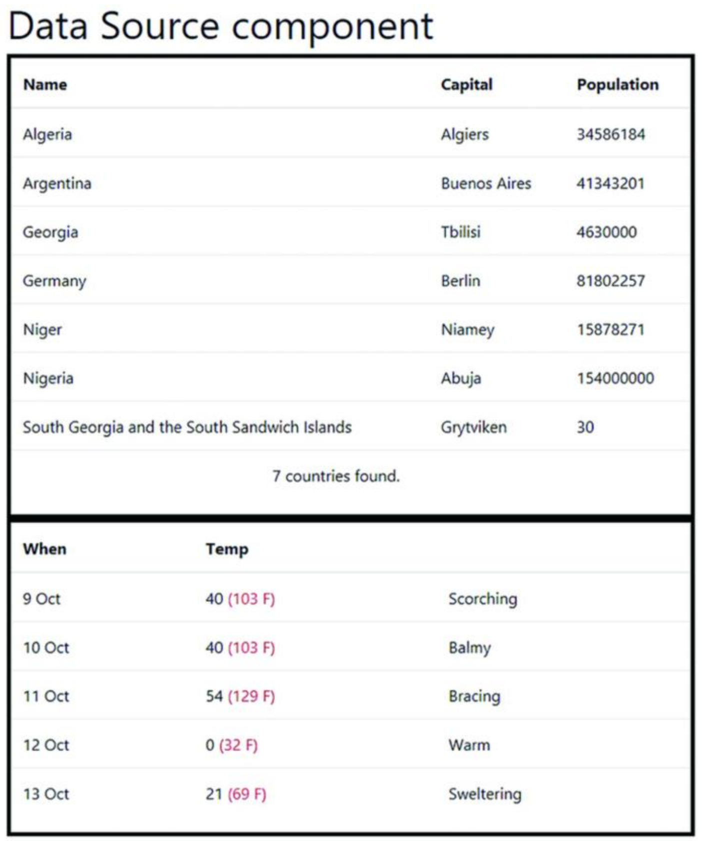
Pages/ParentComponent.razor:

```
@page "/parent"
<h3>List of countries</h3>

<ChildComponent Items="@Countries" TItem="Country">
    <HeaderTemplate>
        <th>Name</th>
        <th>Capital</th>
    </HeaderTemplate>
    <RowTemplate>
        <td>@context.CountryName</td>
        <td>@context.Capital</td>
    </RowTemplate>
    <FooterTemplate>
        <td colspan="2">
            @context.Count countries found.
        </td>
    </FooterTemplate>
</ChildComponent>


@code {
    private List<Country> Countries = new List<Country>
    {
        new Country()
        {
            CountryName = "USA",
            Capital = "Washington, D.C."
        },

        new Country()
        {
            CountryName = "Lebanon",
            Capital = "Beirut"
        }
    };

}
```

Templated Components – Advanced Use Case

As a result, the DataSource generic component can be used in the same Blazor view to populate data grids of different data types

```
<DataSource Items="@Countries" TItem="Country">
  ...
</DataSource>
<DataSource Items="@Forecasts" TItem="WeatherForecast">
  ...
</DataSource>
```



Data Source component

Name	Capital	Population
Algeria	Algiers	34586184
Argentina	Buenos Aires	41343201
Georgia	Tbilisi	4630000
Germany	Berlin	81802257
Niger	Niamey	15878271
Nigeria	Abuja	154000000
South Georgia and the South Sandwich Islands	Grytviken	30

7 countries found.

When	Temp	
9 Oct	40 (103 F)	Scorching
10 Oct	40 (103 F)	Balmy
11 Oct	54 (129 F)	Bracing
12 Oct	0 (32 F)	Warm
13 Oct	21 (69 F)	Sweltering

Demo: Templated Components

Module 3: Components and Layouts

Section 1: Components

Lesson: **Component Libraries**

Component Library

- Components can be shared in a Razor Class Library (RCL) across projects. A Razor components class library can be included from:
    - Another project in the solution
    - A NuGet package
    - A referenced .NET library

- Just as components are regular .NET types, components provided by an RCL are normal .NET assemblies

Demo: Component Library

Module 3: Components and Layouts

Section 1: Components

Lesson: **Component Lifecycle Hooks**

Component Lifecycle Hooks

- When you create a component in Blazor it should derive from ComponentBase. There are two reasons for this:
  - First, is that ComponentBase implements IComponent and Blazor uses this interface to locate components throughout your project as it doesn't rely on a folder convention
  - Second, is that **ComponentBase contains important lifecycle methods**

- Every component has a couple of methods you can override to capture the lifecycle of the component

- Putting code in the wrong lifecycle hook will likely break your component

SetParametersAsync

- After the component is created, this is the first method to be executed
- SetParametersAsync sets parameters supplied by the component's parent in the render tree:

```
public override async Task SetParametersAsync(ParameterView parameters)
{
    await ...

    await base.SetParametersAsync(parameters);
}
```

- ParameterView contains the entire set of parameter values each time SetParametersAsync is called
- Parameters are set when base.SetParametersAsync(parameters) is called

SetParametersAsync

- This is a good point at which to make asynchronous calls out to a server etc based on the state being passed into the component, before any of the component’s [Parameter] properties have been assigned any state

- It is also the correct place to assign default parameter values

OnInitialized and OnInitializedAsync

- Once the component has received its initial parameters from its parent in the render tree, the OnInitialized and OnInitializedAsync methods are called

- Both of these methods will only fire once in the component's lifecycle, as apposed the other lifecycle methods which will fire every time the component is re-rendered

- This is useful in the same way as SetParametersAsync, except it is possible to use the component's state

OnInitialized and OnInitializedAsync

- OnInitialized is called first, then OnInitializedAsync. Any asynchronous operations, which require the component to re-render once they complete, should be placed in the OnInitializedAsync method:

```
protected override void OnInitialized()
{
    ...
}
```

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

OnInitialized and OnInitializedAsync

- Note that **when a navigation is made to a new URL that resolves to the same type of component as the current page, the component will not be destroyed before navigation and the OnInitialized lifecycle methods will not be executed**. The navigation is simply seen as a change to the component's parameters

Previous URL	Current URL	OnInitialized Executed
/	/counter	Yes – Different Page
/counter	/counter/42	No – Same Page
/counter/42	counter/123	No – Same page

Demo: SetParametersAsync Vs OnInitializedAsync

OnParameterSet and OnParameterSetAsync

- The OnParametersSet and OnParametersSetAsync methods are called when a component is first initialized and each time new or updated parameters are received from the parent in the render tree

- By the time it fires new values would have been assigned to the component properties

- For example you can use it to set a private field to the value of which is dependent on the incoming properties

OnParameterSet and OnParameterSetAsync

- To perform an asynchronous operation, use OnParameterSetAsync and the await keyword on the operation:

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

- For a synchronous operation, use OnParameterSet:

```
protected override void OnParametersSet()
{
    ...
}
```

Demo: OnParameterSetAsync

OnAfterRender and OnAfterRenderAsync

- These two methods are executed every time Blazor has re-generated the component’s RenderTree

- This can be as a result of the component’s [Parameter] properties being changed in its parent’s HTML mark-up, as a consequence of the user interacting with the component (e.g. a mouse-click), or if the component executes its StateHasChanged method to invoke a re-render

- If you need to perform an action, such as attaching an event listener, which **requires the elements of the component to be rendered in the DOM** then these methods are where you can do it

- Another great use for these lifecycle methods is for JavaScript library initialization, **which requires DOM elements to be in place to work**

OnAfterRender and OnAfterRenderAsync

- These methods have a single parameter named firstRender. This parameter is true only the first time the method is called on the current component, from there onwards it will always be false
- In cases where additional component hook-ups are required (for example, via JavaScript) it is useful to know if this is the first render

Demo: OnAfterRender

StateHasChanged

- This method **doesn’t qualify as a lifecycle method**, **but it does trigger another method that is part of the lifecycle of a component**

- By default, Blazor will check if its state has changed after certain interactions (such as a button click). Sometimes Blazor cannot be aware of a change to state due to how it was triggered, for example when triggered by a Timer

- In these circumstances we are expected to call StateHasChanged, which will queue up a render request with Blazor for a re-render, and this will trigger OnAfterRender and OnAfterRenderAsync

Demo: StateHasChanged

Dispose

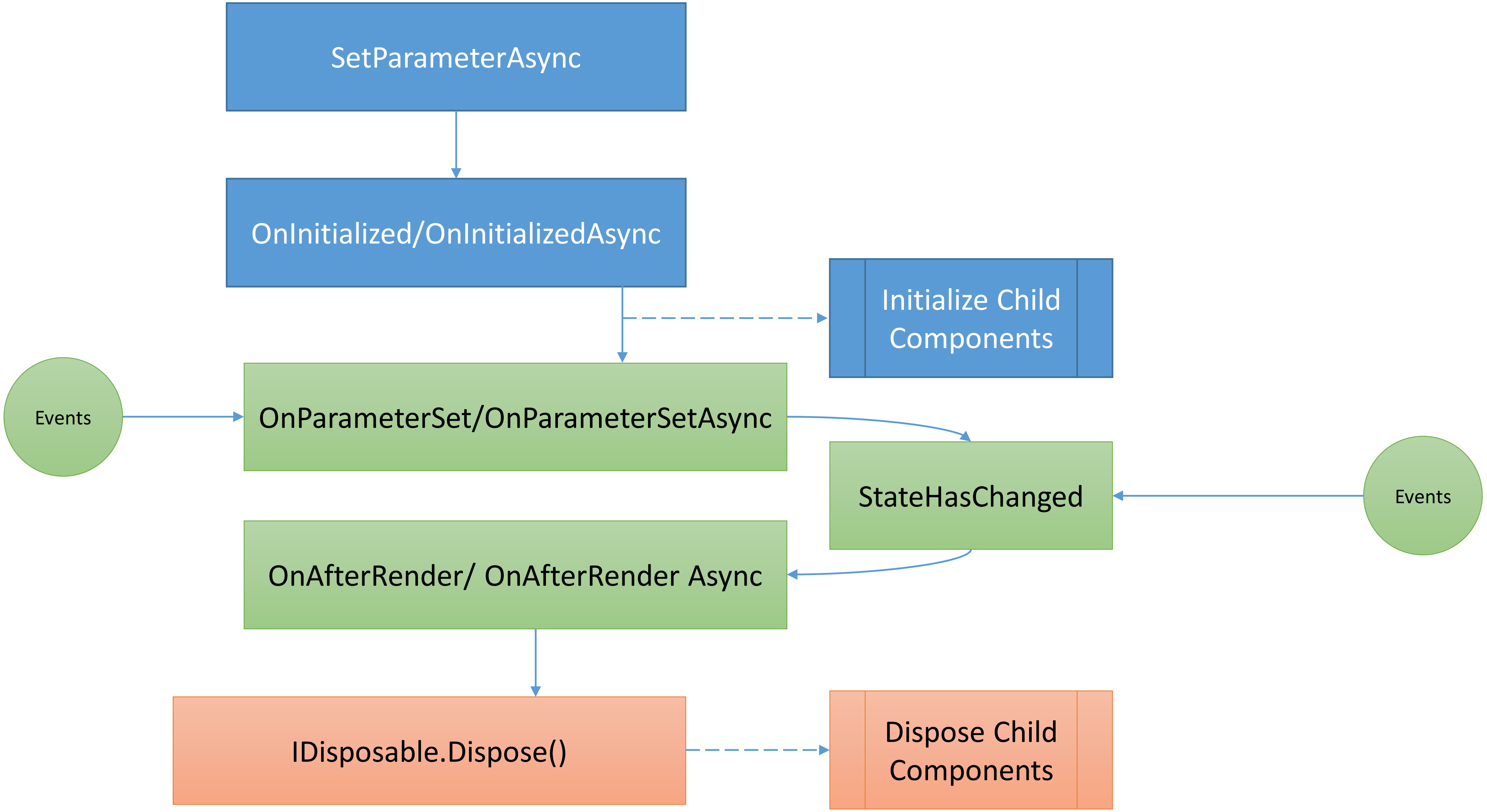
- Although this isn’t strictly one of the ComponentBase’s lifecycle methods, if a component implements IDisposable then Blazor will execute Dispose once the component is removed from its parent’s render tree

- To implement IDisposable add @implements IDisposable to the razor file

```
@implements IDisposable
<h1>This is MyComponent</h1>

@code {
  void IDisposable.Dispose()
  {
    // Code here
  }
}
```

Demo: StateHasChanged

Component Lifecycle Hooks – Complete Picture


SetParameterAsync
OnInitialized/OnInitializedAsync
Initialize Child Components
Events
OnParameterSet/OnParameterSetAsync
StateHasChanged
Events
OnAfterRender/ OnAfterRender Async
IDisposable.Dispose()
Dispose Child Components

Module 3: Components and Layouts

Section 1: Components

Lesson: Integrate components into Razor Pages and MVC apps

Integrate Components Into Razor Pages And MVC Apps

- Use components with existing Razor Pages and MVC apps
- There's no need to rewrite existing pages or views to use Razor components
- When the page or view is rendered, components are prerendered at the same time

Integrate Components Into Razor Pages And MVC Apps

- To render a component from a page or view, use the Component Tag Helper

CSHTML

```
<component type="typeof(Counter)" render-mode="ServerPrerendered"
    param-IncrementAmount="10" />
```

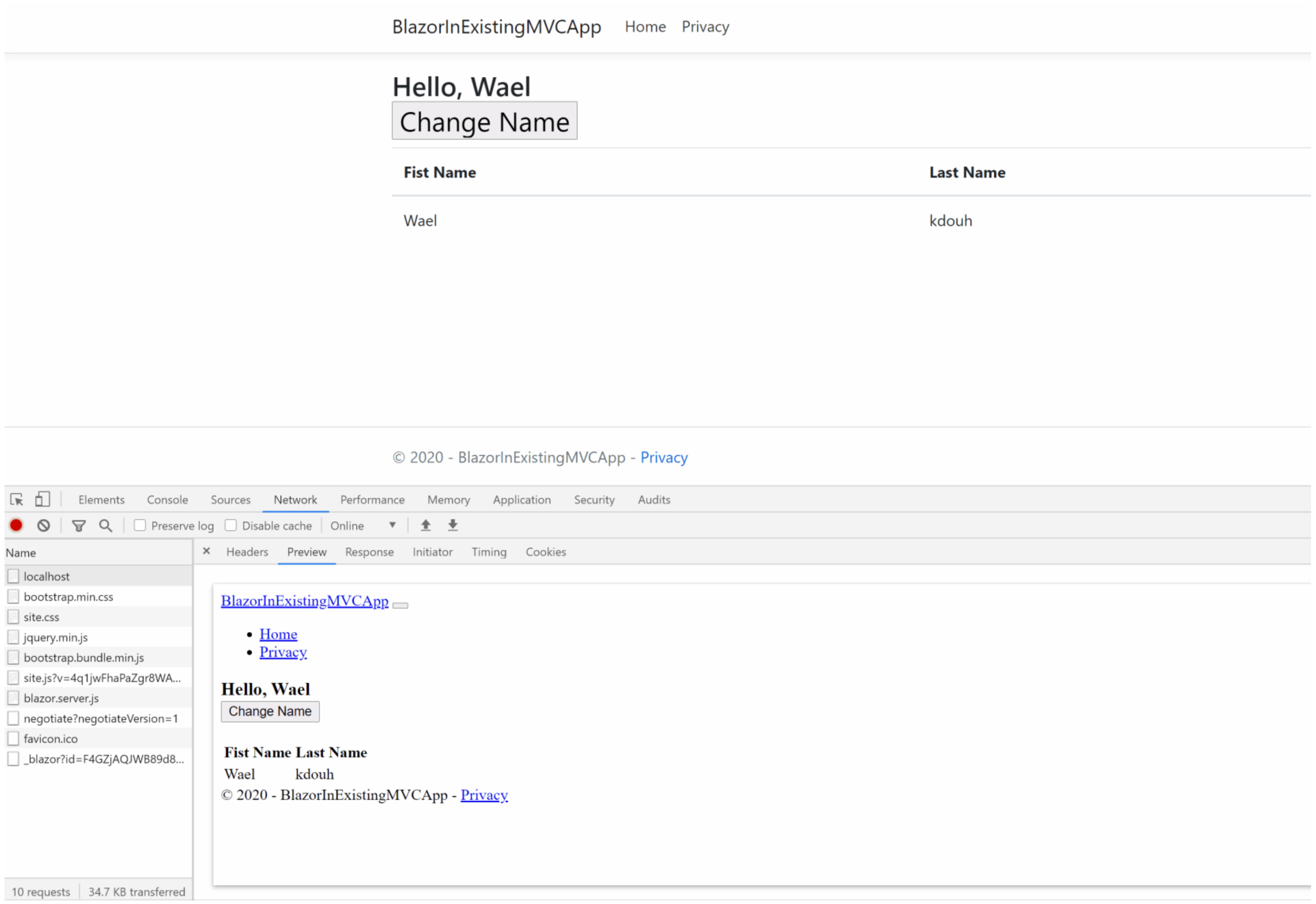
- Passing parameters (for example, IncrementAmount in the preceding example) is supported

Integrate Components Into Razor Pages And MVC Apps

- There are three RenderMode configurations that you can choose from. They basically differ in performance and how they work with Search Engines

RenderMode	Description
ServerPrerendered	Renders the component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app
Server	Renders a marker for a Blazor Server app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app
Static	Renders the component into static HTML

RenderMode - Static



BlazorInExistingMVCApp Home Privacy

Hello, Wael

Change Name

Fist Name	Last Name
Wael	kdouh

© 2020 - BlazorInExistingMVCApp - Privacy

```
localhost
bootstrap.min.css
site.css
jquery.min.js
bootstrap.bundle.min.js
site.js?v=4q1jwFhaPaZgr8WA...
blazor.server.js
negotiate?negotiateVersion=1
favicon.ico
_blazor?id=F4GZjAQJWB89d8...

10 requests | 34.7 KB transferred
```

BlazorInExistingMVCApp

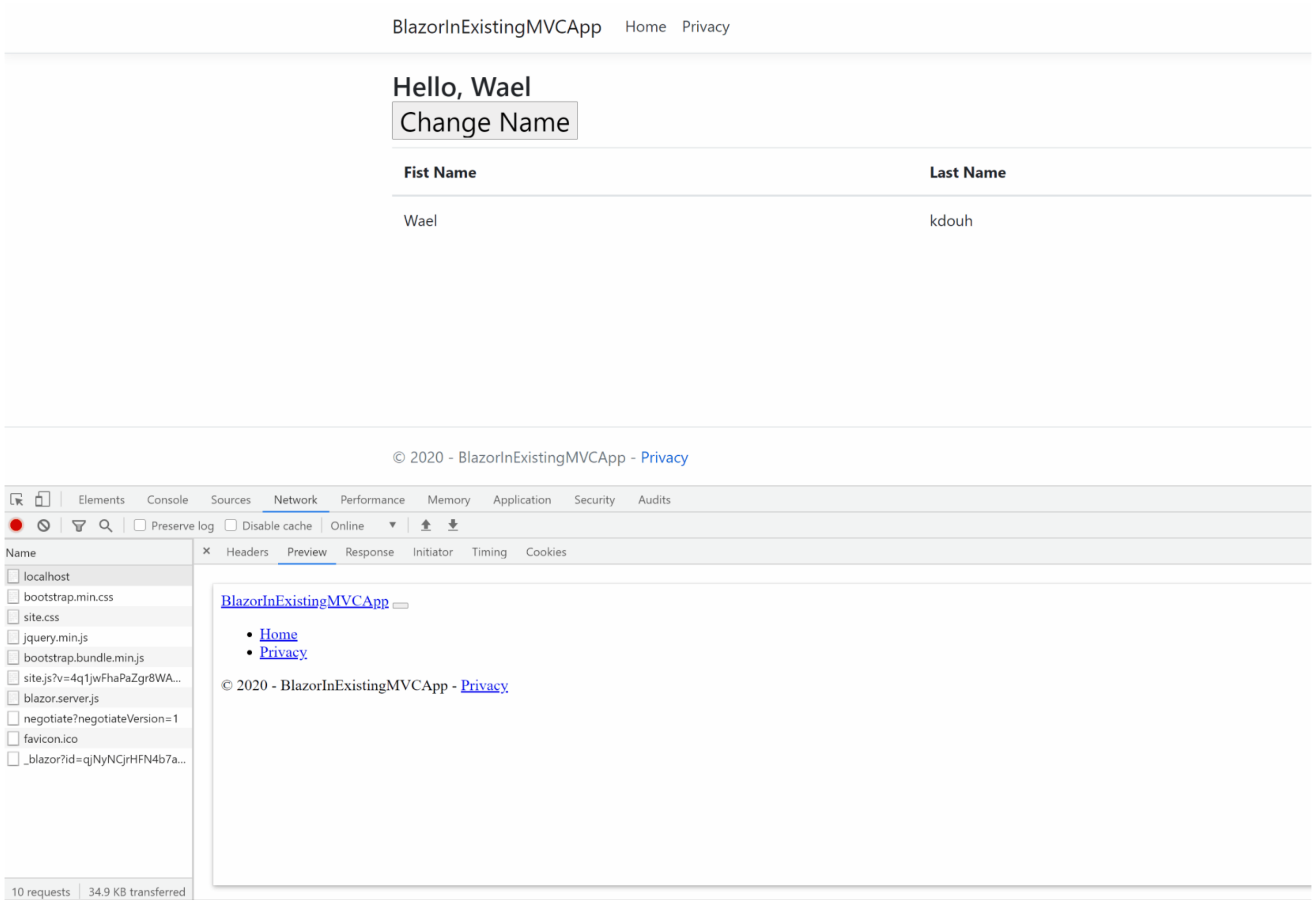
- Home
- Privacy

Hello, Wael

Change Name

Fist Name Last Name
Wael kdouh
© 2020 - BlazorInExistingMVCApp - Privacy

RenderMode - Server



BlazorInExistingMVCApp Home Privacy

Hello, Wael

Change Name

Fist Name	Last Name
Wael	kdouh

© 2020 - BlazorInExistingMVCApp - Privacy

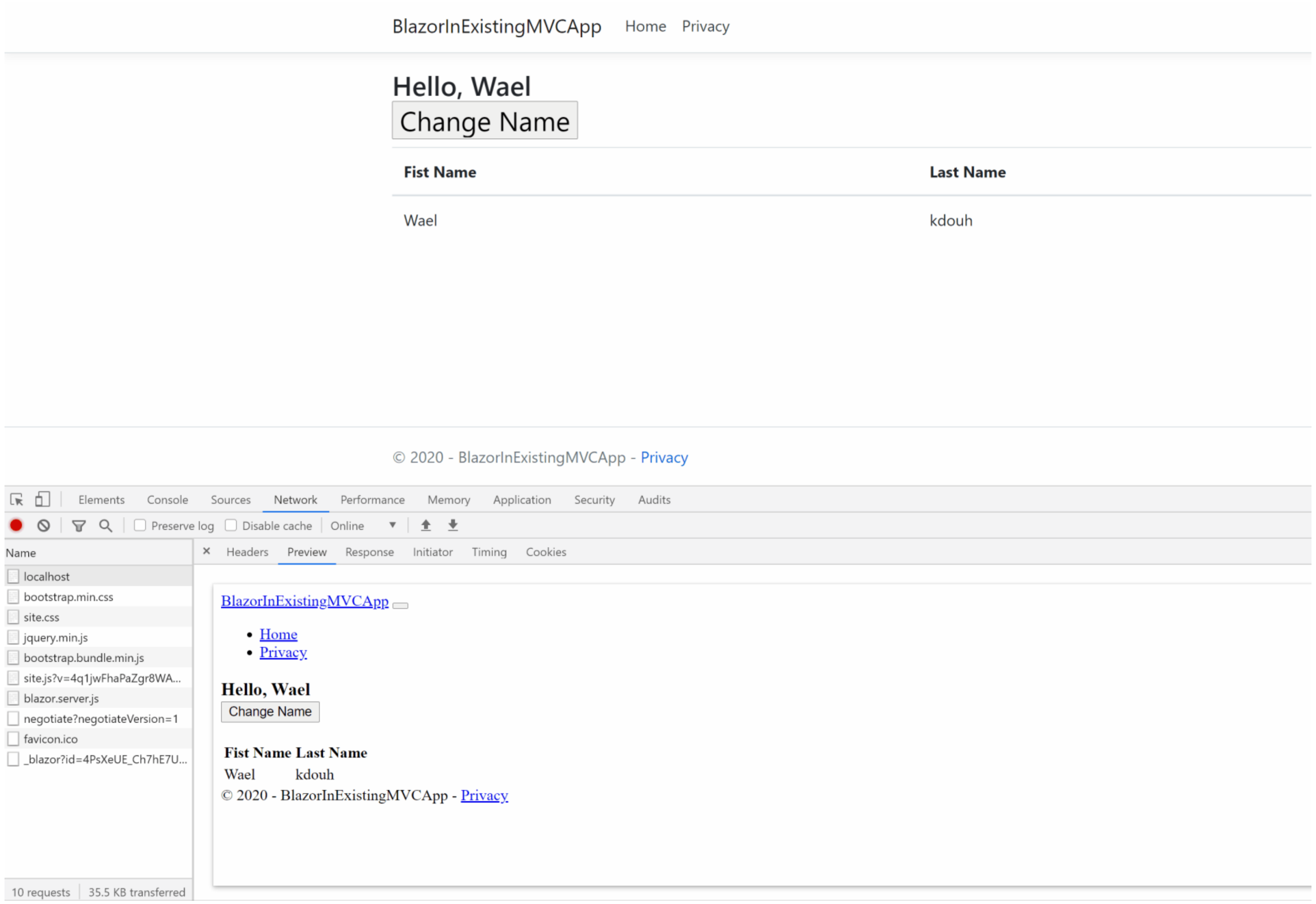
```
localhost
bootstrap.min.css
site.css
jquery.min.js
bootstrap.bundle.min.js
site.js?v=4q1jwFhaPaZgr8WA...
blazor.server.js
negotiate?negotiateVersion=1
favicon.ico
_blazor?id=qjNyNCjrHFN4b7a...

10 requests | 34.9 KB transferred
```

BlazorInExistingMVCApp

- Home
- Privacy

© 2020 - BlazorInExistingMVCApp - Privacy

RenderMode - ServerPrerendered



BlazorInExistingMVCApp Home Privacy

Hello, Wael

Change Name

Fist Name	Last Name
Wael	kdouh

© 2020 - BlazorInExistingMVCApp - Privacy

Name
- localhost
- bootstrap.min.css
- site.css
- jquery.min.js
- bootstrap.bundle.min.js
- site.js?v=4q1jwFhaPaZgr8WA...
- blazor.server.js
- negotiate?negotiateVersion=1
- favicon.ico
- _blazor?id=4PsXeUE_Ch7hE7U...

10 requests | 35.5 KB transferred

BlazorInExistingMVCApp

- Home
- Privacy

Hello, Wael

Change Name

Fist Name	Last Name
Wael	kdouh

© 2020 - BlazorInExistingMVCApp - Privacy

Integrate Components Into Razor Pages And MVC Apps

- While pages and views can use components, the converse isn't true. Components can't use view and page-specific scenarios, such as partial views and sections. To use logic from partial view in a component, factor out the partial view logic into a component

- Rendering server components from a static HTML page isn't supported

Demo: Adding Blazor Support to an Existing MVC Application

Module 3: Components and Layouts

Section 2: Layouts

Lesson: Overview

Layouts

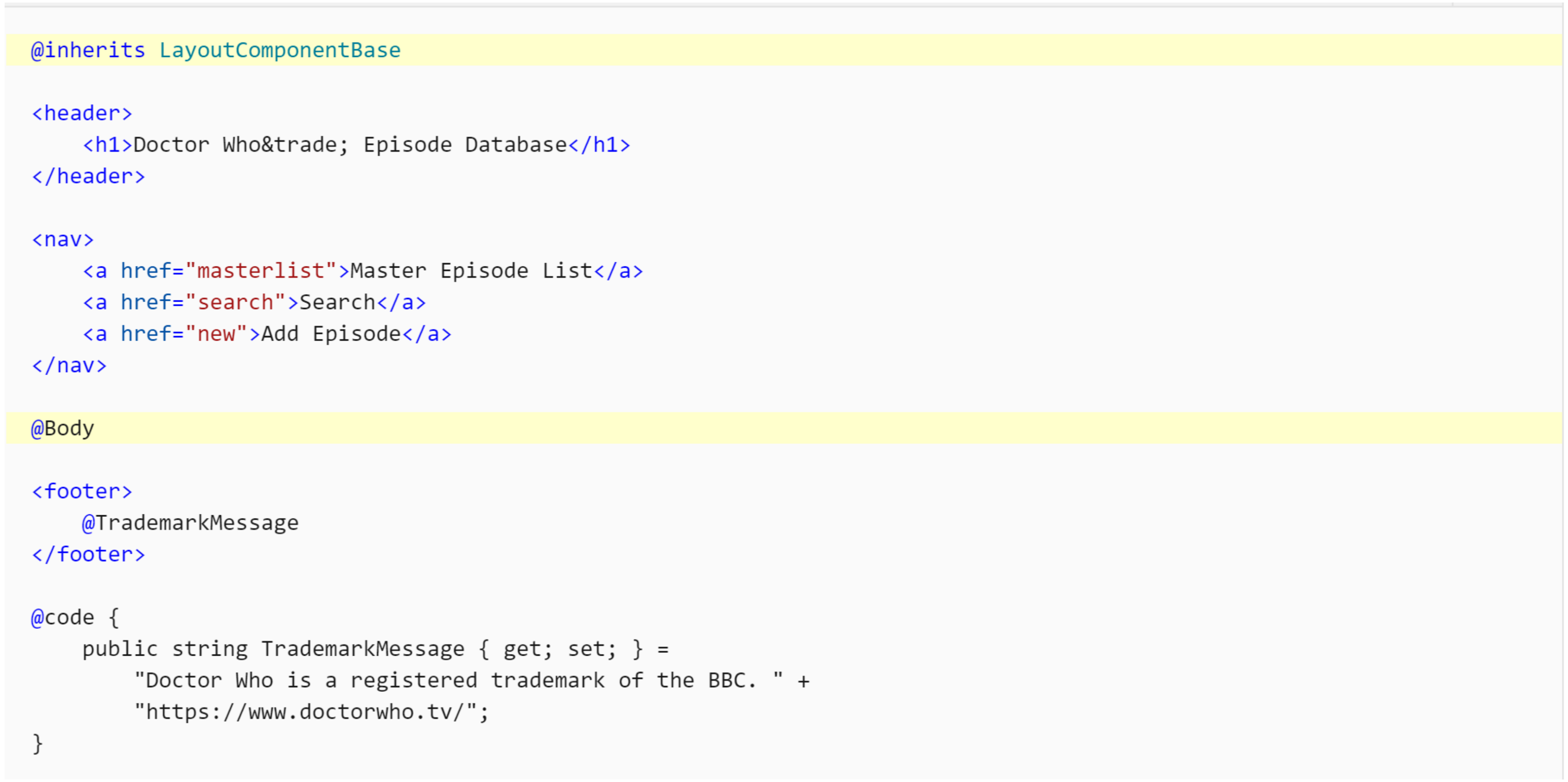
- A layout is just another component

- It is defined in a Razor template or in C# code and can use data binding, dependency injection, and other component scenarios

- To turn a component into a layout, the component:
  - Inherits from LayoutComponentBase, which defines a Body property for the rendered content inside the layout
  - Uses the Razor syntax @Body to specify the location in the layout markup where the content is rendered

Layouts

- The following code sample shows the Razor template of a layout component, MainLayout.razor
- The layout inherits LayoutComponentBase and sets the @Body between the navigation bar and the footer:



```
@inherits LayoutComponentBase

<header>
    <h1>Doctor Who&trade; Episode Database</h1>
</header>

<nav>
    <a href="masterlist">Master Episode List</a>
    <a href="search">Search</a>
    <a href="new">Add Episode</a>
</nav>

@Body

<footer>
    @TrademarkMessage
</footer>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/";
}
```

Default Layout

- Specify the default app layout in the Router component in the app's App.razor file. The following Router component, which is provided by the default Blazor templates, sets the default layout to the MainLayout component:

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

Default Layout

- To supply a default layout for NotFound content, specify a LayoutView for NotFound content:

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="typeof(MainLayout)">
            <h1>Page not found</h1>
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Specify A Layout In A Component

- Use the Razor directive @layout to apply a layout to a component. The compiler converts @layout into a LayoutAttribute, which is applied to the component class.

- The content of the following MasterList component is inserted into the MasterLayout at the position of @Body:

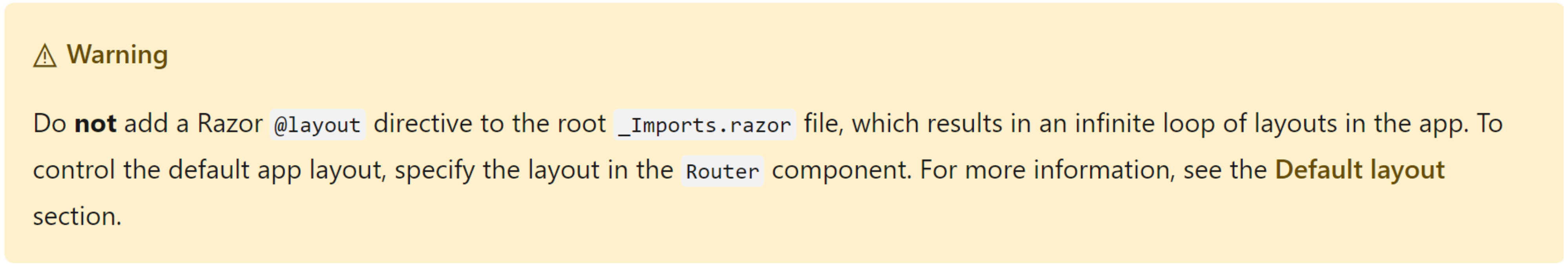
razor

```
@layout MasterLayout
@page "/masterlist"

<h1>Master Episode List</h1>
```

- Specifying the layout directly in a component overrides a default layout set in the router or an @layout directive imported from _Imports.razor.

Centralized Layout Selection

- Every folder of an app can optionally contain a template file named _Imports.razor
- The compiler includes the directives specified in the imports file in all of the Razor templates in the same folder and recursively in all of its subfolders
- An _Imports.razor file containing @layout MyCoolLayout ensures that all of the components in a folder use MyCoolLayout. There's no need to repeatedly add @layout MyCoolLayout to all of the .razor files within the folder and subfolders



⚠ **Warning**

Do **not** add a Razor `@layout` directive to the root `_Imports.razor` file, which results in an infinite loop of layouts in the app. To control the default app layout, specify the layout in the `Router` component. For more information, see the **Default layout** section.

Demo: Custom Layouts

Module Summary

- In this module, you learned about:
    - Building Components
    - Building Templated Components
    - Building Component Libraries
    - Understand the Different Lifecycle Events of a Component
    - Embed Blazor Components inside an Asp.Net Core MVC Application
    - Understand Layouts

Lab 3: Components and Layouts

References

- Microsoft Docs

- Blazor University

Microsoft