ASP.NET Core 3 - Blazor

Wael Kdouh - @waelkdouh

Senior Consultant

Module 3: Blazor

Module Overview

Module 3: Blazor

Section 1: WebAssembly

Lesson: WebAssembly Fundamentals

First a Disclaimer…

JavaScript isn’t going anywhere, but if you prefer .NET instead…

…let’s talk

What Is Webassembly?

WebAssembly is a new type of code that can be run in modern web browsers — it is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++ and Rust with a compilation target so that they can run on the web. It is also designed to run alongside JavaScript, allowing both to work together

"MDN"

Why WebAssembly?

- There is lots of code out there that is not javascript – C++, C#, Java

- Flash & Silverlight didn’t make the grade
- Filled a need, but required plugins

- What about ASM.js & transpilers?

- Emscripten can transpile C++ to javascript
- Is still JS (an optimized subset, but still)
- Some things are not easily expressed in JS

- Use cases – Games, VR/AR, Video/Image editing

Let’s Break that down…

low-level assembly-like language

compact binary format

near-native performance

Web Assembly Goals

- fast, efficient, portable
- readable, debuggable
- keep secure
- don’t break the web

Where Does It Run?

- Most modern browsers
  - https://caniuse.com/#search=webassembly

- Otherwise?
  - fallback to asm.js version

- Runs alongside JavaScript, does not replace it

Demo: Unity3D

Https://WWW.WEBASSEMBLYGAMES.COM/

Module 3: Blazor

Section 2: Blazor

Lesson: Introduction To Blazor

So… that's cool… Now what?

I'M A C# DEV, WHY DO I CARE?

As C# developers

- We love C#
- We love .NET
- We want the option to reuse the same code everywhere

What if (without plugins) we could

- Use C# on both client & server?
- Share DTOs with the client?
- Use the same toolchain on both?

Why .NET?

Stable, mature, productive

- .NET Standard
- MSBuild

Fast, scalable, reliable

- .NET Core for backend services

Modern languages

- Innovations in C# & F#
- Razor

First-rate dev tools

- Visual Studio
- Intellisense

Blazor History

- NDC Demo In 2017
- Lots Of Interest…
- Moved To ASP.Net’s Github As Official Experiment Jan 2018
- Server Side Blazor Components Shipped With .NET 3.0
- Client Side Blazor Shipping After .NET 3.0

Current Setup (October '19)

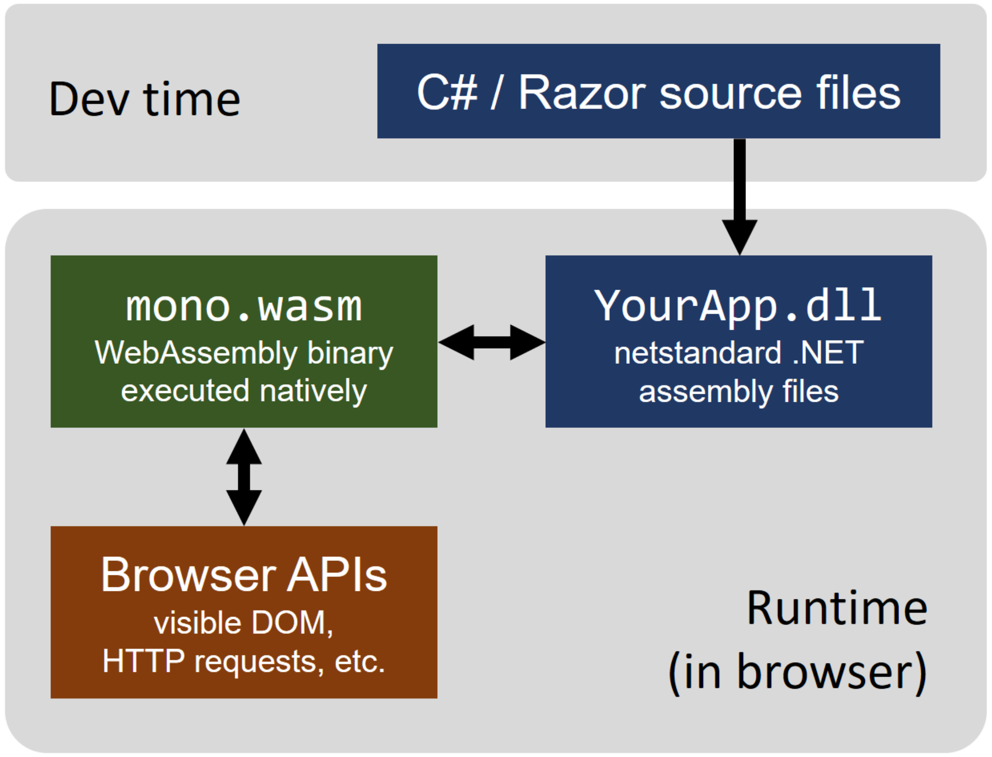
- .NET Core 3.0 SDK
- Visual Studio 2019 (16.3 or later)
- Latest Blazor Templates - simply install them from the command-line

What Is Blazor?

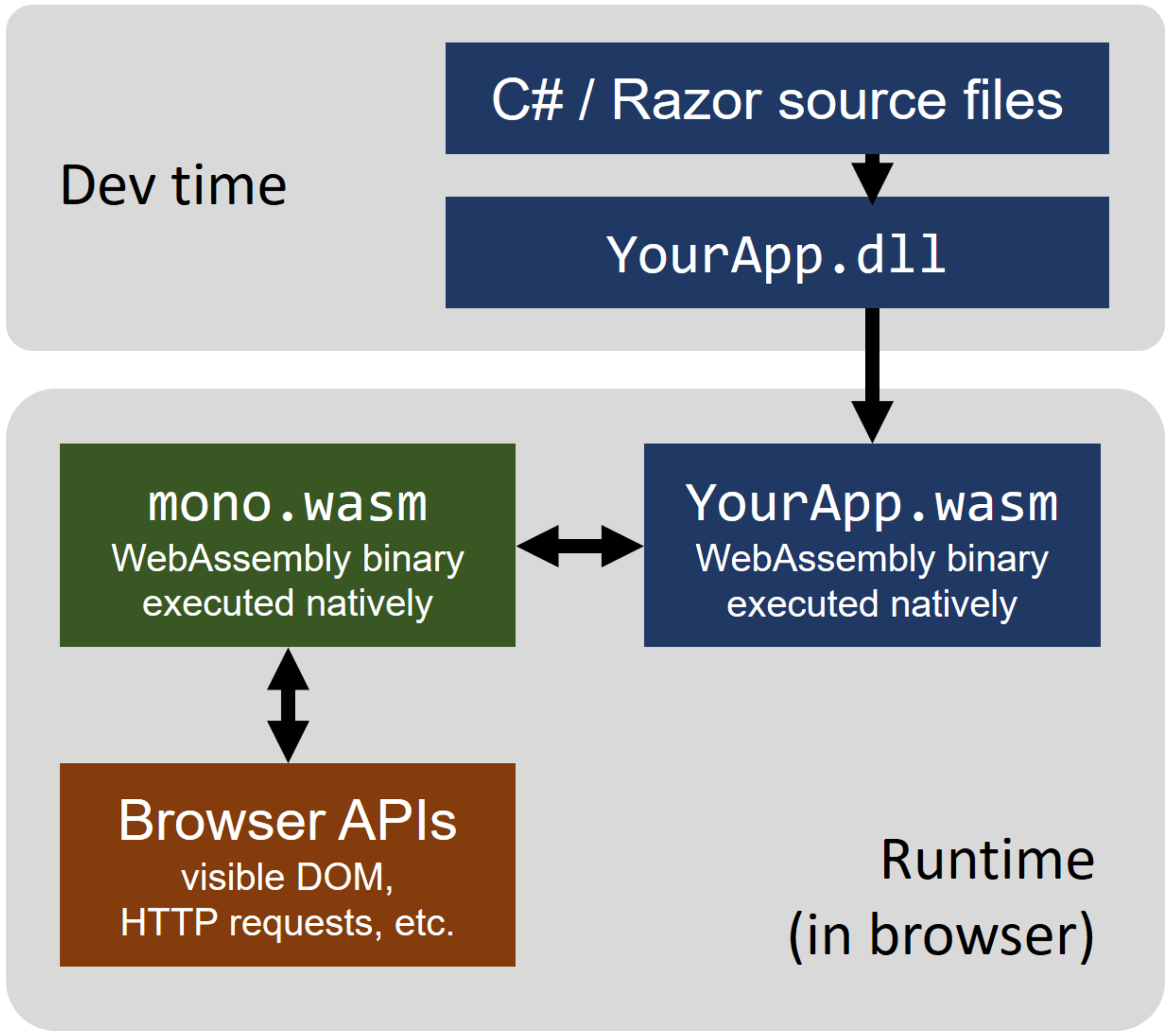
- Build client-side web UI with .NET instead of JavaScript
- Write reusable web UI components with C# and Razor
- Share .NET code with both the client and the server
- Call into JavaScript libraries & browser APIs as needed

What Is Going On Here?

- .NET runtime is compiled to web assembly
- Your app is compiled to a normal assembly
- Uses Mono runtime
  - Mono already had a project to compile to WASM
  - http://www.mono-project.com/news/2018/01/16/mono-static-webassembly-compilation/


Dev time
C# / Razor source files
mono.wasm
WebAssembly binary executed natively
YourApp.dll
netstandard .NET assembly files
Browser APIs
visible DOM, HTTP requests, etc.
Runtime (in browser)

Aot Compiling Coming Later


Dev time
C# / Razor source files
YourApp.dll
mono.wasm
WebAssembly binary executed natively
YourApp.wasm
WebAssembly binary executed natively
Browser APIs
visible DOM, HTTP requests, etc.
Runtime (in browser)

Is this Silverlight?

- No it is not
- No plug in
- All standard web technologies
- No XAML

It is just HTML/CSS & a VM

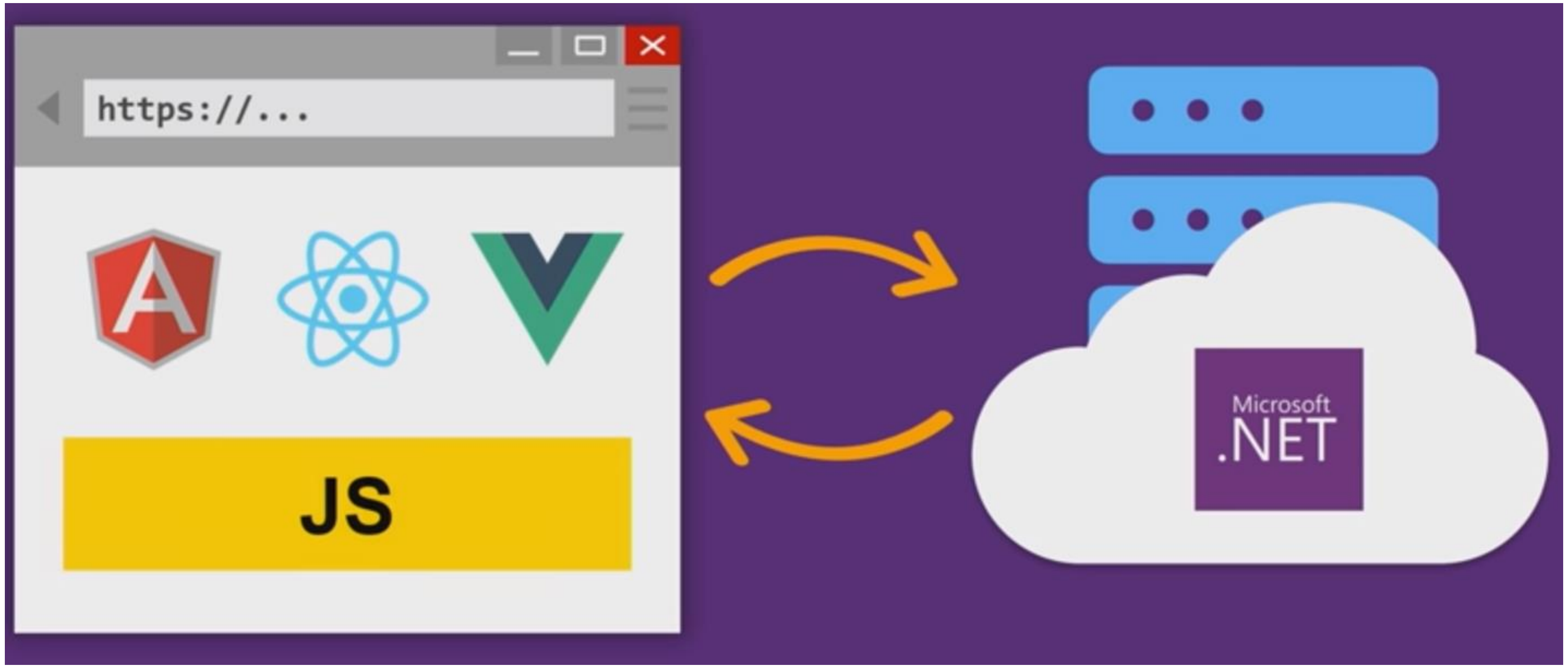
Demo: A Quick Hello World

Module 3: Blazor

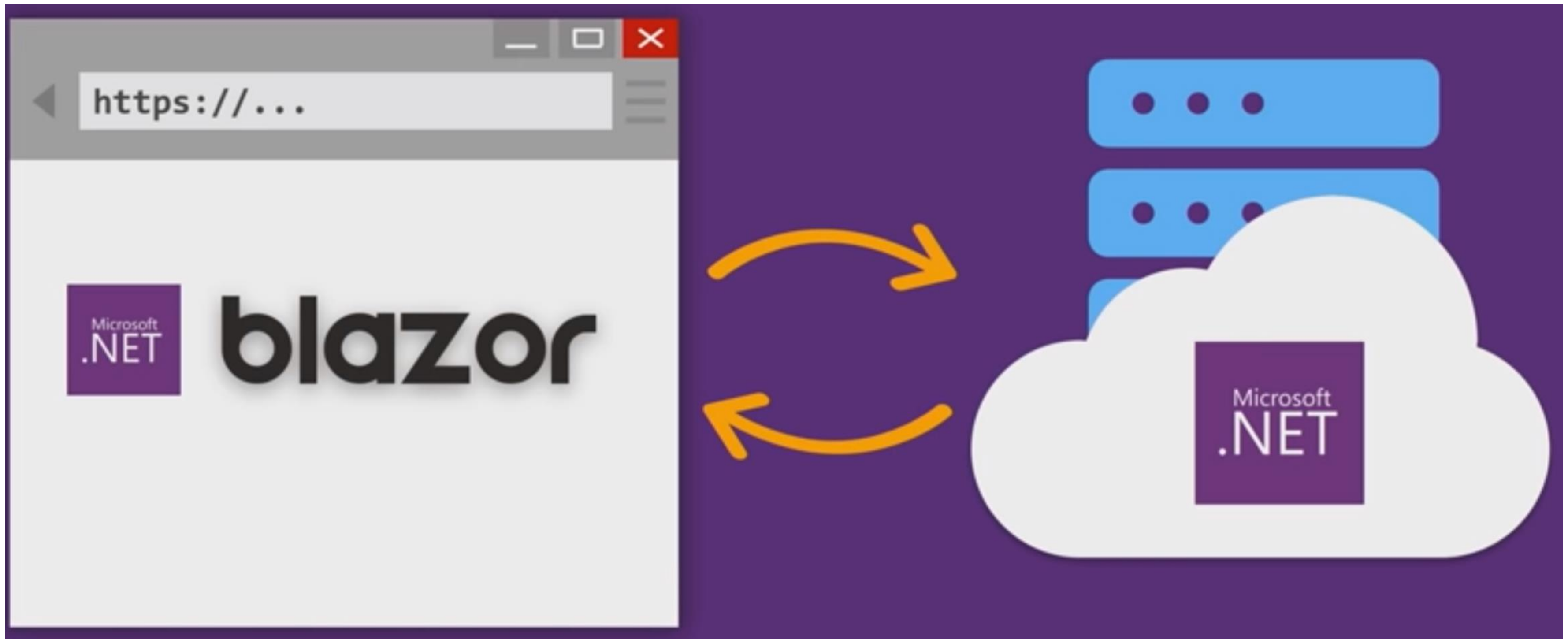
Section 2: Blazor

Lesson: Hosting Models

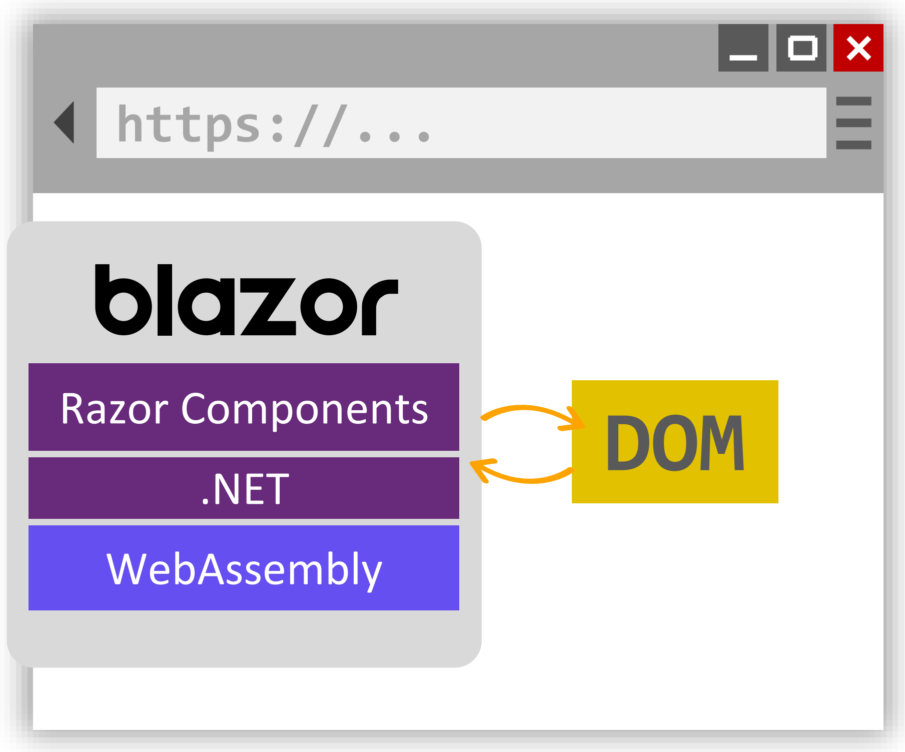
Client Side – In Preview As Of October 2019



Client Side – In Preview As Of October 2019



Client Side – In Preview As Of October 2019



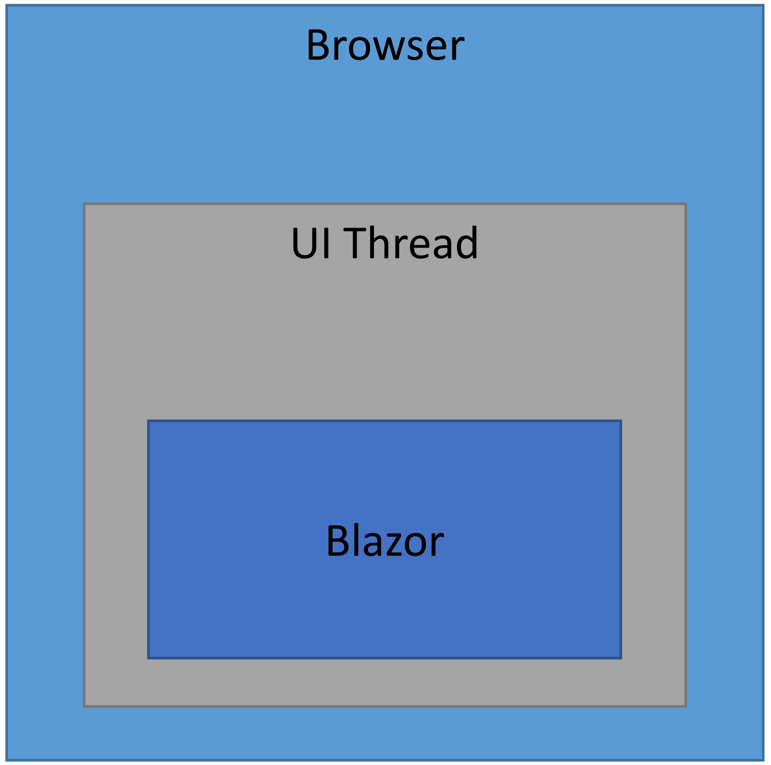
Browser provides the core WebAssembly runtime Support

On Top of that we built a .Net WebAssembly runtime

Razor Components run in the browser on top of the .Net runtime

Blazor receives UI events from the DOM and later sends back DOM diffs to be applied to the DOM

Client Side – In Preview As Of October 2019



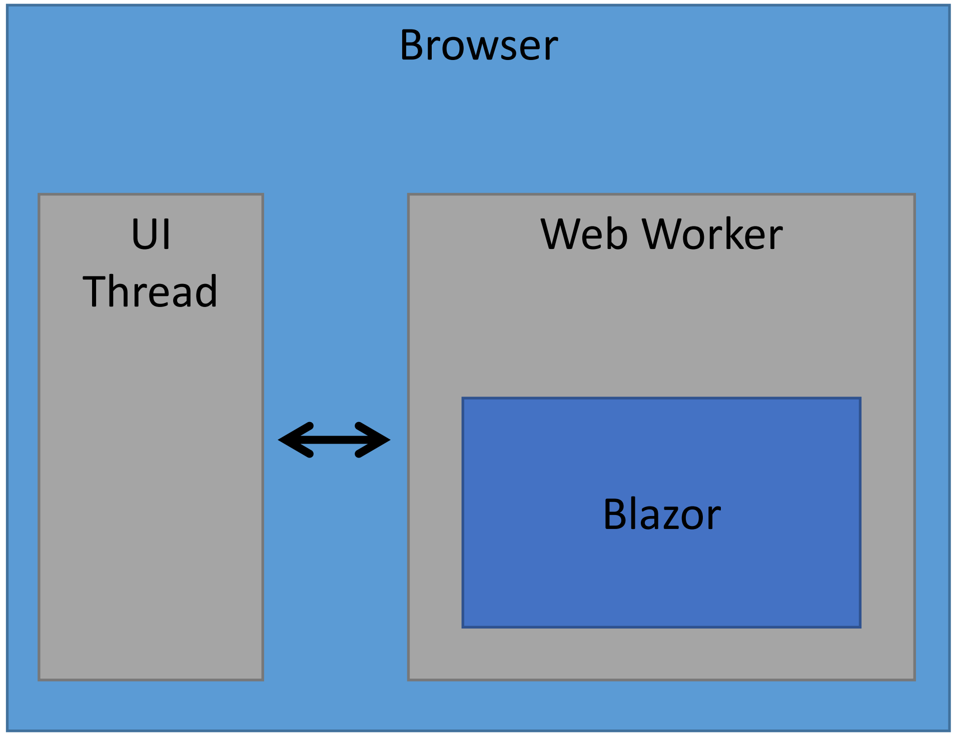
WebAssembly-based .NET runtime

Blazor WebAssembly

App is executed directly on the browser UI thread

UI updates and event handling occur within the same process

Client Side Blazor In A Web Worker



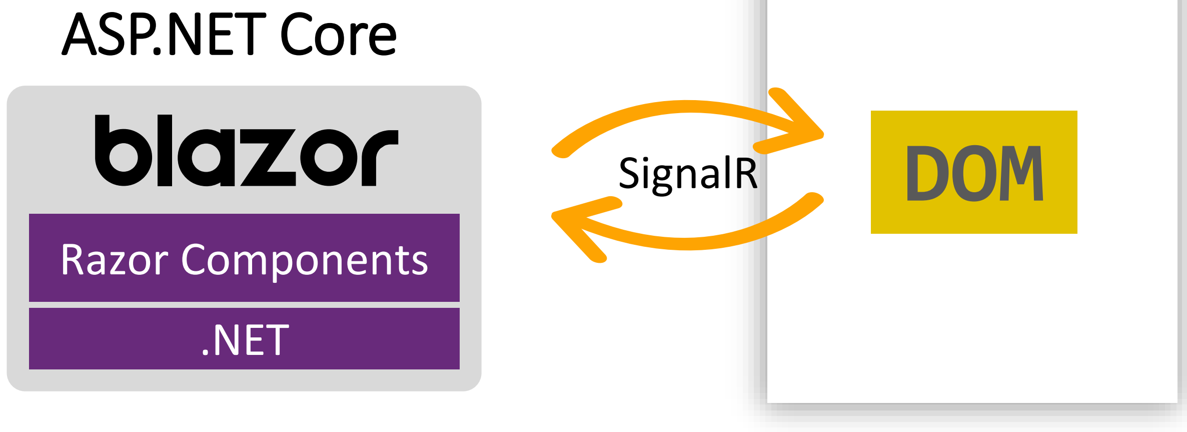
WebAssembly-based .NET runtime

Blazor WebAssembly

App is executed on a Web Worker

UI updates and event handling occur in different processes

Server-Side Blazor – Available With .Net Core 3



App is executed on the server from within an ASP.NET Core app

Browser Sets Up Realtime SignalR connection to handle and send all the UI events that the user creates

Blazor receives UI events from the DOM and later sends back DOM diffs using the SignalR connection to be applied to the Dom

Blazor On Client Or Server

Blazor WebAssembly

- Pro:
  - True SPA, full interactivity
  - Utilize client resources
  - Supports offline, static sites, PWA scenarios
- Con:
  - Larger download size
  - Requires WebAssembly
  - Limited Debugging Options
  - Still in preview

May 2020

Blazor Server

- Pro:
  - Smaller download size, faster load time
  - Running on fully featured .NET runtime
  - Code never leaves the server
  - Simplified architecture
- Con:
  - Latency
  - No offline support
  - Consumes more server resources

.NET Core 3.0

Demo: Hosting Models

Module 3: Blazor

Section 2: Blazor

Lesson: Basics

Blazor Features

- Components
- Layouts
- Routing
- Validation
- JavaScript Interop

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

Componens

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

Components

- 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

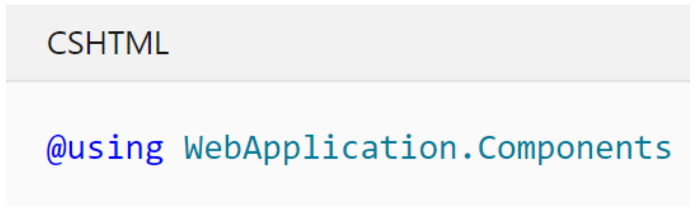
```cshtml
<h1 style="font-style:@_headingFontStyle">@_headingText</h1>

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

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)

Components

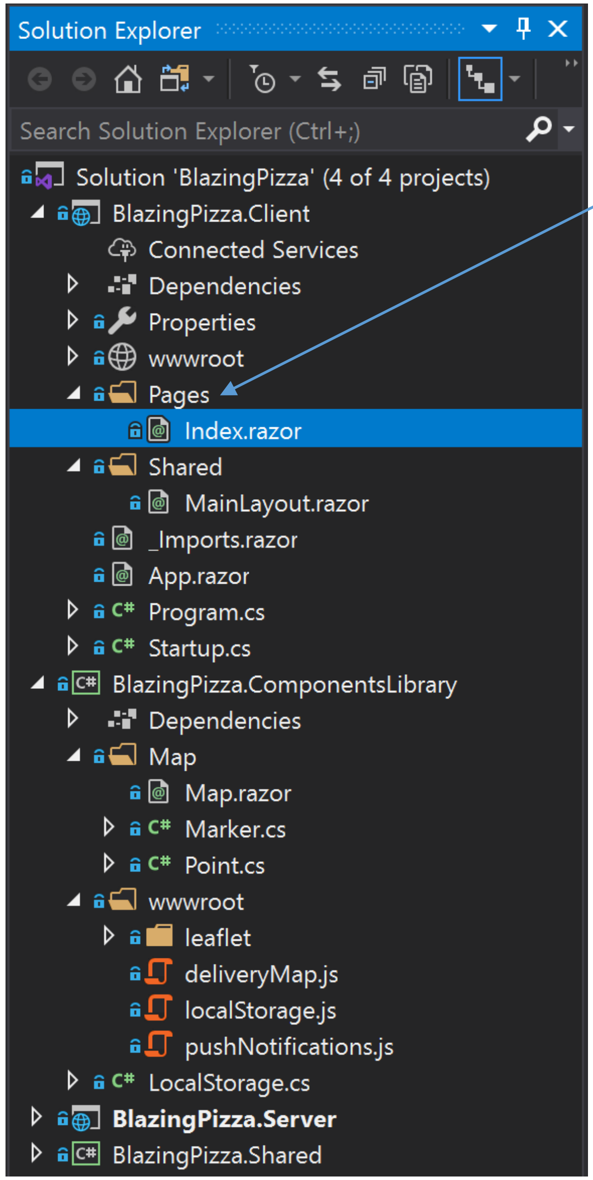
- Components are ordinary C# classes and can be placed anywhere within a project
- Components that produce webpages usually reside in the Pages folder
- Non-page components are frequently placed in the Shared folder or a custom folder added to the project
- To use a custom folder, add the custom folder's 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:



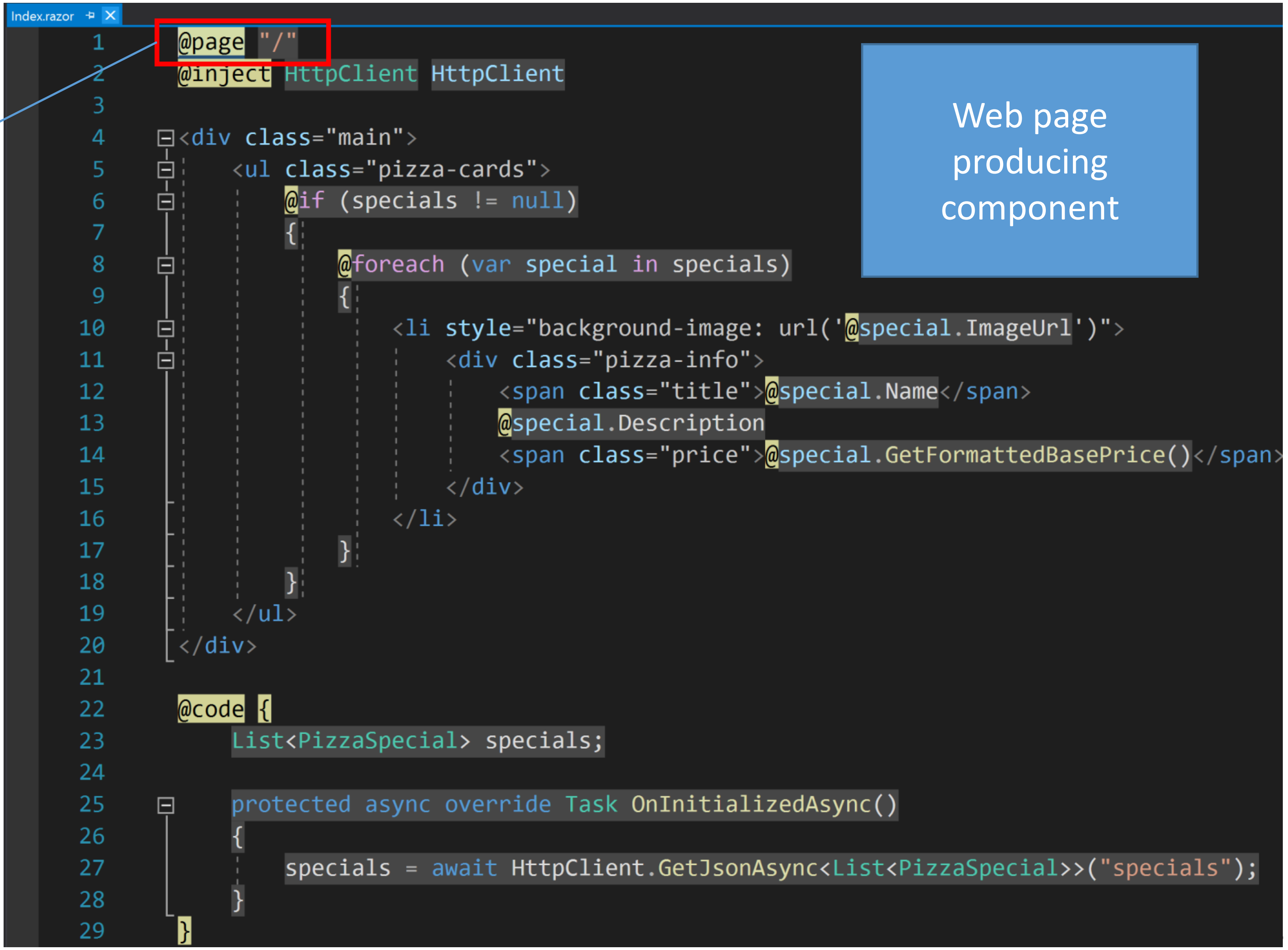
CSHTML

```
@using WebApplication.Components
```

Componens



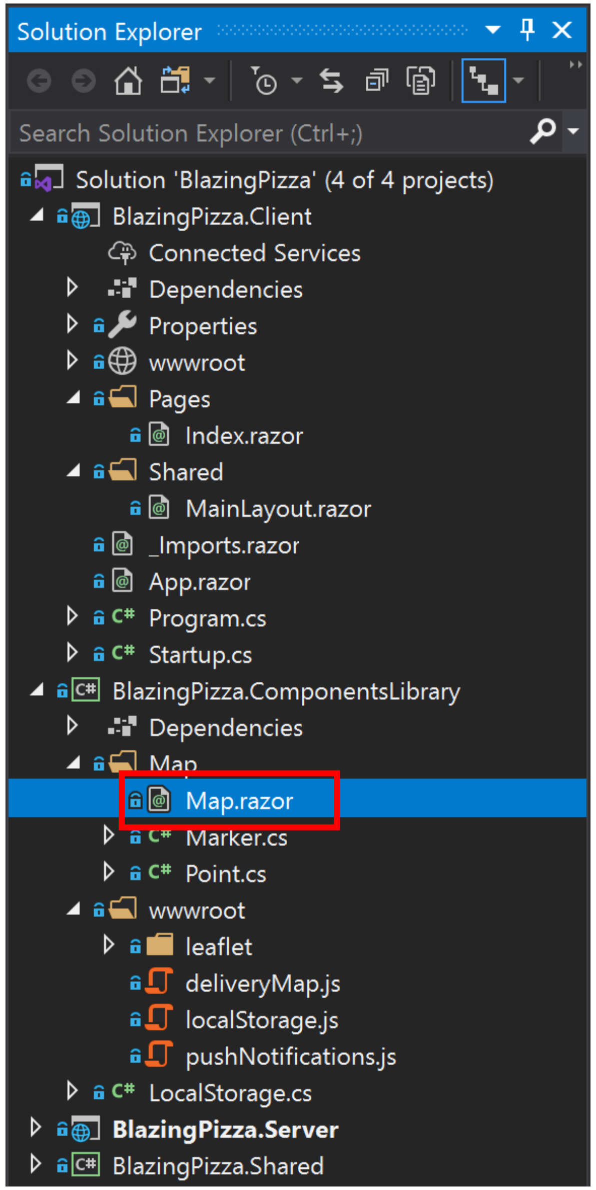
```
Solution 'BlazingPizza' (4 of 4 projects)
  BlazingPizza.Client
    Connected Services
    Dependencies
    Properties
    wwwroot
    Pages
      Index.razor
    Shared
      MainLayout.razor
    _Imports.razor
    App.razor
    Program.cs
    Startup.cs
  BlazingPizza.ComponentsLibrary
    Dependencies
    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");
    }
}
```

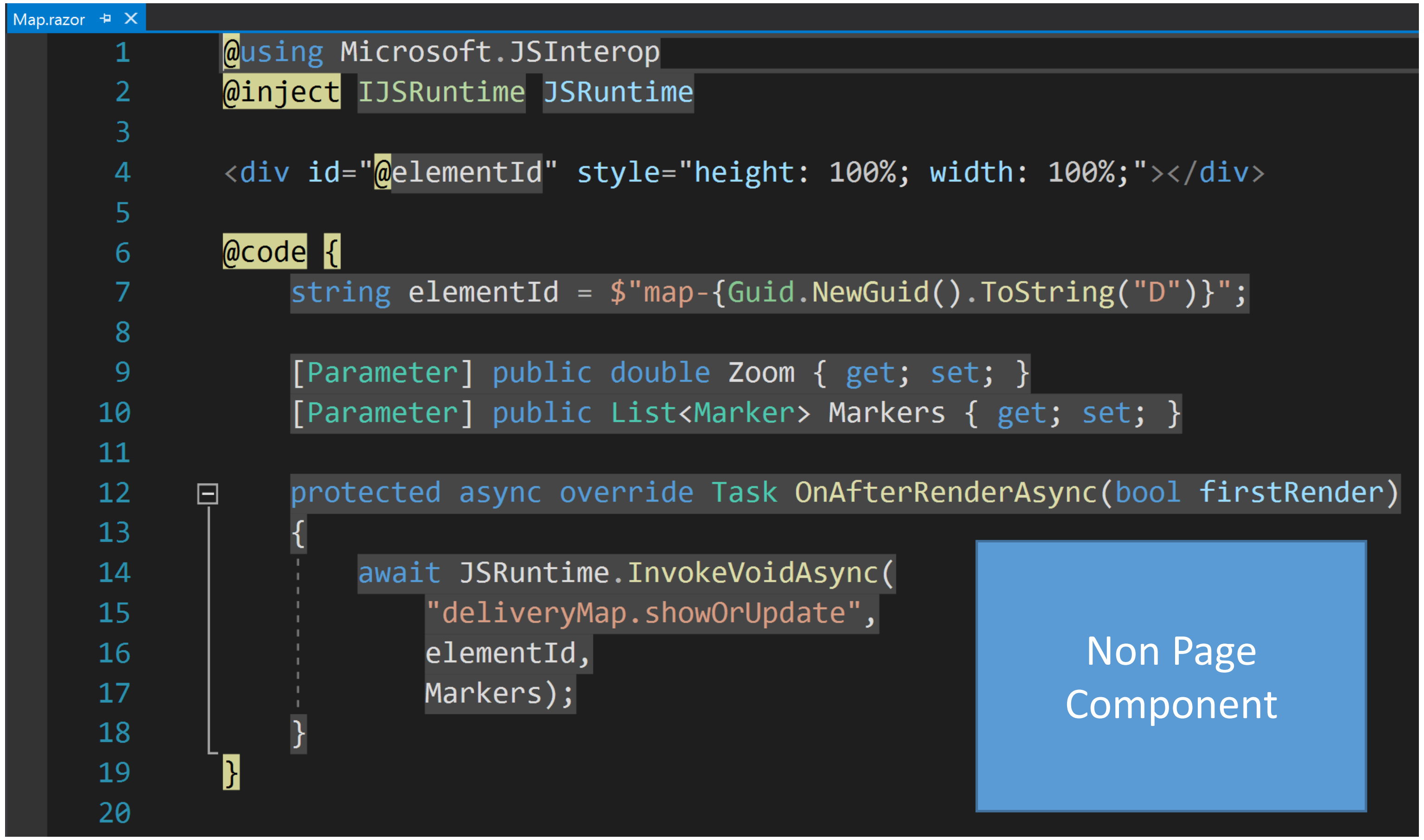
Web page producing component

Components


Solution Explorer

Search Solution Explorer (Ctrl+;)

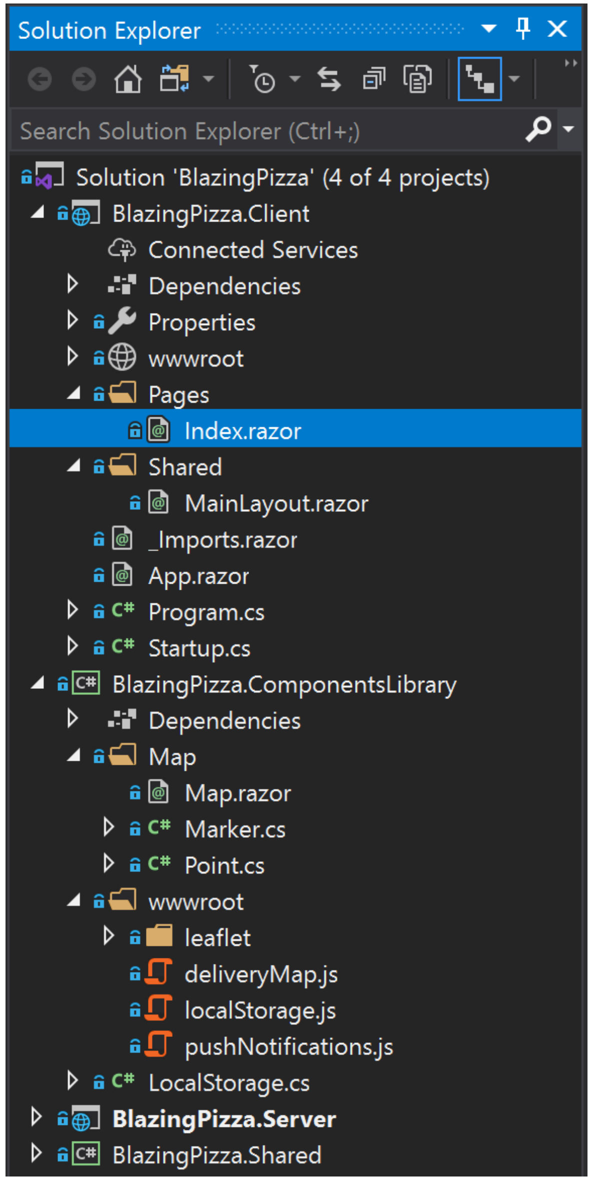
```
Solution 'BlazingPizza' (4 of 4 projects)
  BlazingPizza.Client
      Connected Services
      Dependencies
      Properties
      wwwroot
      Pages
          Index.razor
      Shared
          MainLayout.razor
      _Imports.razor
      App.razor
      Program.cs
      Startup.cs
  BlazingPizza.ComponentsLibrary
      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

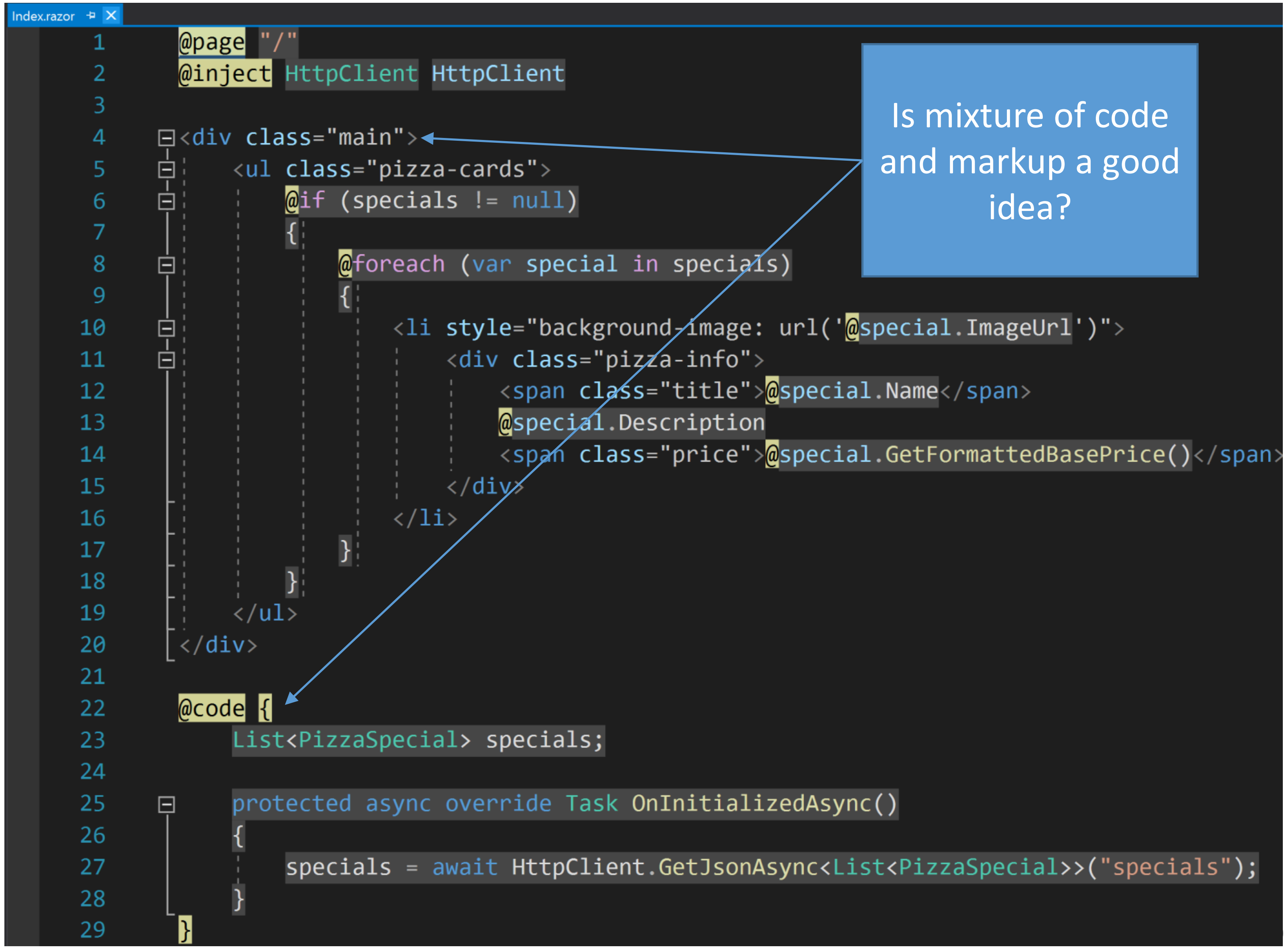
Components



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");
    }
}
```

Is mixture of code and markup a good idea?

Demo: Component Code-Behind Pattern

Components - Parameters

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

- Use attributes to specify arguments for a component in markup

Components - Parameters

Components/ChildComponent.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; }
}
```

Pages/ParentComponent.razor

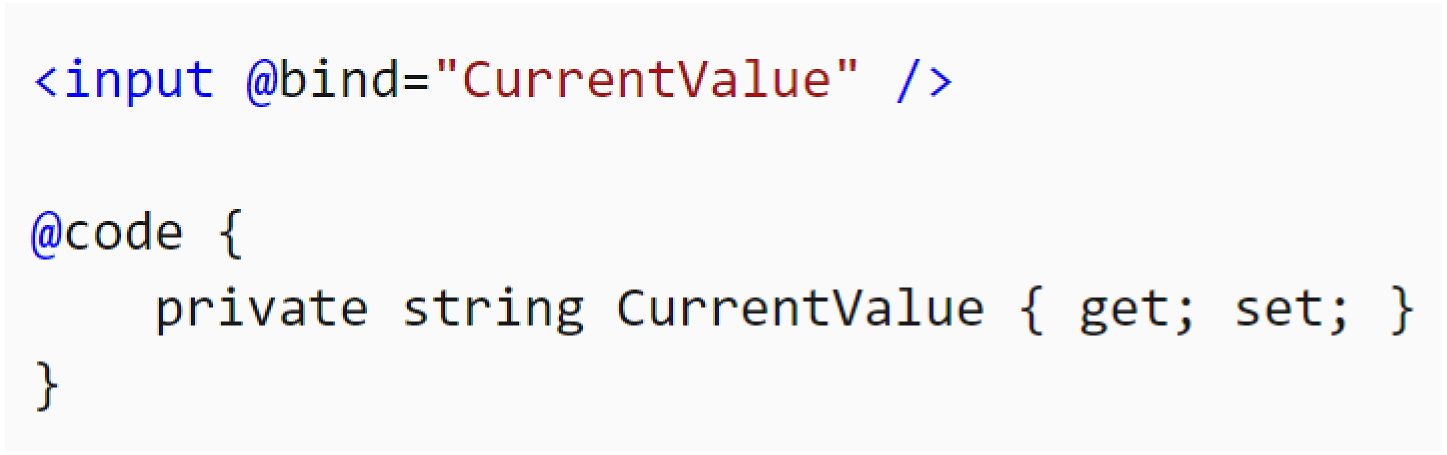
```
@page "/ParentComponent"

<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>

<p><b>@messageText</b></p>

@code {
    private string messageText;

    private void ShowMessage(MouseEventArgs e)
    {
        messageText = "Blaze a new trail with Blazor!";
    }
}
```

Demo: Parameters

Components - Data Binding

- Data binding to both components and DOM elements is accomplished with the @bind attribute

- The following example binds a CurrentValue property to the text box's value:



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

@code {
    private string CurrentValue { get; set; }
}
```

Demo: Data Binding

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>
```

Demo: Layouts

Routing

- Blazor Server is integrated into ASP.NET Core Endpoint Routing

- An ASP.NET Core app is configured to accept incoming connections for interactive components with MapBlazorHub in Startup.Configure:

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

Routing

- The most typical configuration is to route all requests to a Razor page, which acts as the host for the server-side part of the Blazor Server app

- By convention, the host page is usually named _Host.cshtml. The route specified in the host file is called a fallback route because it operates with a low priority in route matching

- The fallback route is considered when other routes don't match. This allows the app to use others controllers and pages without interfering with the Blazor Server app.

Route Templates

- The Router component enables routing to each component with a specified route

- The Router component appears in the App.razor file:

```
<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>
```

- When a .razor file with an @page directive is compiled, the generated class is provided a RouteAttribute specifying the route template

Route Templates

- At runtime, the RouteView component:
    - Receives the RouteData from the Router along with any desired parameters.
    - Renders the specified component with its layout (or an optional default layout) using the specified parameters.

- You can optionally specify a DefaultLayout parameter with a layout class to use for components that don't specify a layout

- The default Blazor templates specify the MainLayout component. MainLayout.razor is in the template project's Shared folder

Route Templates

- Multiple route templates can be applied to a component. The following component responds to requests for /BlazorRoute and /DifferentBlazorRoute:

```
@page "/BlazorRoute"
@page "/DifferentBlazorRoute"

<h1>Blazor routing</h1>
```

Demo: Routing

Validation

- Forms and validation are supported in Blazor using data annotations
- The following ExampleModel type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

Validation

- A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="@exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        Console.WriteLine("OnValidSubmit");
    }
}
```

Validation

- The form validates user input in the name field using the validation defined in the ExampleModel type. The model is created in the component's @code block and held in a private field (exampleModel). The field is assigned to the Model attribute of the <EditForm> element
- The DataAnnotationsValidator component attaches validation support using data annotations
- The ValidationSummary component summarizes validation messages
- HandleValidSubmit is triggered when the form successfully submits (passes validation)

Demo: Validation

JavaScript Interop

- A Blazor app can invoke JavaScript functions from .NET and .NET methods from JavaScript code

- There are times when .NET code is required to call a JavaScript function. For example, a JavaScript call can expose browser capabilities or functionality from a JavaScript library to the app

- To call into JavaScript from .NET, use the IJSRuntime abstraction. The InvokeAsync<T> method takes an identifier for the JavaScript function that you wish to invoke along with any number of JSON-serializable arguments. The function identifier is relative to the global scope (window). If you wish to call window.someScope.someFunction, the identifier is someScope.someFunction. There's no need to register the function before it's called. The return type T must also be JSON serializable

JavaScript Interop

- A Blazor app can invoke JavaScript functions from .NET and .NET methods from JavaScript code

- There are times when .NET code is required to call a JavaScript function. For example, a JavaScript call can expose browser capabilities or functionality from a JavaScript library to the app

- To call into JavaScript from .NET, use the IJSRuntime abstraction. The InvokeAsync<T> method takes an identifier for the JavaScript function that you wish to invoke along with any number of JSON-serializable arguments. The function identifier is relative to the global scope (window). If you wish to call window.someScope.someFunction, the identifier is someScope.someFunction. There's no need to register the function before it's called. The return type T must also be JSON serializable

Module Summary

- In this module, you learned about:
  - WebAssembly
  - Blazor Hosting Models
  - Blazor Basics

Lab: Blazor

Microsoft