Blazor

Blazing past the competition

Johan Litsfeldt

Contents

Chapter 1

Introduction

JavaScript is the duct tape of the Internet.
- Charlie Campbell

1.1 Pain points

Let us begin by having a look at the issues of modern web development today. Typically a JavaScript framework like Angular is needed for data-binding, routing etc.

For a back-end developer looking to transition to front-end development, the learning curve can be rather steep. Designing a stable architecture let alone figuring out where to begin can be the most daunting task often leading to a lot of trial and error before gaining some sort of expertise in the area. And by the time you have it all figured out, a new framework comes out and replaces it.

Current web development frameworks are very opinionated. Usually they give the impression that you have the freedom to create anything you want in any way you want to but in reality one typically has to follow strict paradigms. The introduction of hooks in the React framework is one example of how developers to some extent become restricted to guidelines set by the creators of the framework.

Another problem with modern web development is the dependencies on 3rd party software. For a typical React project you have to choose a toolchain including a packet manager (npm, Yarn etc.), a bundler (webpack, Parcel, etc.) and a transpiler such as Babel. Picking the right tools is a very difficult task in which you have to take a lot of information into consideration e.g. to what degree you want to allow yourself to depend on them and what the consequences might be.

Modern frameworks and third party software become obsolete very fast if not maintained. Web development has evolved at a very quick rate in the last decade and is still constantly changing. There is always some update to your npm packages and you never know how they will affect your software or if the tutorial you are following is up to date.

The JavaScript language itself has its issues being weakly dynamically typed, having some quirky behaviour and not being the most intuitive language. You typically need Typescript to make it bearable (i.e. another dependency) and by then it almost starts to look and behave like C#. As WebAssembly starts to become a viable option for running code in the web browser, we might start to see a decline of the importance of JavaScript.

Another problem is the way there is a gap between the frontend and the backend which needs to be bridged. We typically need to translate back-end C# classes/models to Typescript interfaces and do lots of serialization and deserialization for conversion between types and classes.

1.2 History

2002	.NET Framework 1.0
2006	Silverlight
	C# apps could run in the browser
2007	Android & iOS
	No extensions allowed in mobile browsers
2007	HTML 5
2010	Angular
	Viable option for apps in mobile browsers
2011	Silverlight 5
	Silverlight discontinued
2013	asm.js
	C/C++ code in the browser
2013	React
2016	.NET Core
2017	WebAssembly
	C# code in the browser
2017	Blazor announced (Steve Sandersson)
2019	Blazor Preview (Daniel Roth)
2019	.NET Core 3.0
2019	**Blazor Server**
2020	.NET Core 3.1
2020	**Blazor WebAssembly**
2021	.NET 6
2021	**Blazor LTS**

[1]Inspiration from William Liebenberg `https://www.ssw.com.au/people/william-liebenberg`

1.3 What is Blazor?

Blazor is a single page web app framework built on ASP.NET that runs in the browser via WebAssembly. The name Blazor is a combination of Browser and Razor.

Razor

Razor is a markup syntax for embedding server-based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Blazor uses Razor in component files with the .razor extension.

Blazor lets you build interactive web UIs with the use of C# instead of JavaScript. This includes typical concepts like reusable components, data-binding, routing and more. Components are implemented using the familiar cornerstones C#, HTML, and CSS. Both client and server code is written in C# allowing code and libraries to be shared seamlessly.

Blazor comes in two flavours. You can run your client-side C# code directly in the browser using WebAssembly much like the React, Angular or Vue frameworks. This is called Blazor WebAssembly (or Blazor WASM).

The other alternative is to run client-side code on the server which outputs HTML to the browser. Client UI events are communicated using SignalR technology and the changes are merged into the DOM dynamically. [1]

1.4 Why Blazor?

- **.NET framework in the browser:** All familiar tools of .NET Core can be used with no other dependencies. Code can also be shared between client and server enabling less duct taping.
- **No plugins required:** Blazor is using open web standards on both older and new browsers running HTML and CSS with no requirement for any external plugin. Blazor is NOT Silverslight.
- **.NET based SPA framework:** Like all modern web development frameworks Blazor operates dynamically on the DOM, requires no refreshing of the web browser and supports routing.
- **Razor templating engine:** Components are easily expressed using Razor markup, C# and HTML making switching from Razor pages to Blazor a pleasant experience.
- **Use existing .NET NuGet packages:** All your favorite NuGet packages can be reused to empower your Blazor applications.
- **No JavaScript necessary:** If required JavaScript is easily integrated into your Blazor apps using Interop calls but is not at all required for a fully functional app.
- **WebAssembly:** WebAssembly is designed to enable

near-native code execution speed in web browsers. Since WebAssembly is valuable other contexts than browsers it has a lot of potential for future web systems.

- **Built to be stable for years:** Blazor is free and open source with a strong community being backed by Microsoft. It is a modern, fun and exciting addition to the current flora of web development frameworks.

Chapter 2

Background

2.1 The .NET Framework

The .NET framework is a stable, free, cross-platform, open source developer platform for building applications for desktop, mobile, games, web and more. The framework is heavily tested, well supported and has been optimized for performance, reliability and security over its lifetime.

.NET includes a managed runtime, a standard set of libraries and support for multiple modern and innovative programming languages like C# and F# making programming versatile and fun for beginners as well as experts.

.NET 5 is the latest version of the framework and is the successor of .NET Core 3.x. It is not to be confused with

.NET Framework 4.x although it incorporates most of its features making .NET 5 the main implementation of .NET going forward.

Note

.NET Framework 4.x is still supported as technologies like Web Forms (replaceable by Blazor), Windows Communication Foundation (WCF) and Windows Workflow (WF) are still used.

ASP.NET Core 5 is a .NET web development framework offering standard tools and libraries for building web applications. Web applications can be built using ASP.NET Core using Razor pages, MVC or Blazor. It can also be used to build RESTful HTTP API endpoints, remote procedure call app (gRPC) or real time applications using SignalR (also used for Blazor Server apps).

The Visual Studio IDE provides a solid experience for development on Windows, Linux and macOS enabling the developer seamless integration between client and server.

- **Desktop**
  - WPF
  - Windows Forms
  - UWP
- **Web**
  - ASP.NET
  - **Blazor**
- **Cloud**
  - Azure
- **Mobile**
  - Xamarin
- **Gaming**
  - Unity
- **IoT**
  - ARM32
  - ARM64
- **AI**
  - ML.NET

- **.NET Standard**
  - **.NET 5**
  - **Infrastructure**
    - Runtime Components
    - Compilers
    - Languages

- **Visual Studio**
- **VS for Mac**
- **VS Code**
- **CLI**

Figure 2.1: The .NET platform.

2.2 WebAssembly

WebAssembly (WASM) is a portable binary-code language resembling the .NET IL code first announced in 2015 and being a standard web technology since 2017. It became a World Wide Web Consortium recommendation in December 2019 and is now supported by all modern web browsers.

Unlike the other languages running natively in the browser (HTML, CSS and JavaScript), WebAssembly runs at near native speed and requires no parsing or compilation steps before execution as WebAssembly is a binary format.

Both WebAssembly and JavaScript runs within a controlled environment called the JavaScript Runtime (V8 engine) providing a single thread for each tab or domain. This enables WebAssembly to call JavaScript functions and vice-versa efficiently.

In order to run .NET binaries code in the web browser, Blazor comes with a .NET Runtime compiled in WebAssembly called *dotnet.wasm*. This runtime is based on the mono runtime and enables IL code to be executed within a WebAssembly context.

When your browser loads a Blazor WebAssembly app it loads the script *dotnet.webassembly.js*. This script downloads the required system DLLs and boots the .NET runtime. Since a WebAssembly app cannot directly manipulate the DOM, this script also wires upp all DOM elements (buttons etc.) with the IL code.

Intermediate Language

During compilation, source code (e.g. C#) is converted into Intermediate Lanuage (IL) platform independent code. The IL code is then converted into machine code by the JIT (just-in-time) compiler.

2.3 SignalR

Blazor Server apps utilize the ASP.NET Core SignalR technology which is a real-time messaging framework. Each client using your Blazor Server app uses one or more SignalR connections.

Blazor uses an abstraction over the SignalR connections called a circuit. Circuits allow temporary network interruptions and attempts to reconnect should the connection be disconnected. The circuit will then create a new SignalR connection. [2]

Each browser screen (tab or iframe) connected to your app will use a SignalR connection (as well as a separate instance of component state). When closing a tab or navigating to an external URL, Blazor Server will try to perform a graceful termination. When a graceful termination occurs, the circuit and associated resources are immediately released. In the case of non-graceful disconnections (e.g. due to network interruption), Blazor Server will store disconnected circuits for a configurable interval to allow the client to reconnect without losing state.

A Blazor Server app will prerender in response to the first request from the client. This will in turn set up the UI state on the server and the client will attempt to create a SignalR connection to that server. The client must then reconnect to the same server.

An app which requires more than one backend server should implement sticky sessions for SignalR connections. You can do this via the Azure SignalR Service for Blazor Server apps or similar. This will allow for scaling up the Blazor Server app to large amounts of concurrent SignalR connections. [3]

2.4 The Virtual DOM

Any time an attribute of an HTML element is changed (e.g. width, height, padding, margin etc.) the browser must reflow the elements on the page before rendering them. This render tree is called the document-object-model (or DOM) and the act of updating it is usually very CPU intensive when performing lots of updates.

Client-side tools like React and Angular both implement something called the Virtual DOM used as an in-memory representation of the elements that will make up the HTML page. This data creates a tree of HTML elements as if they had been specified by an HTML mark-up page.

Both Blazor Server and Blazor WebAssembly uses a virtual DOM generated by the compiler. Each razor page contains a `BuildRenderTree` method accepting a `RenderTreeBuilder` object which contains code to build the virtual DOM. [4]

The virtual DOM offers two benefits:

1. The attribute values of virtual HTML elements can be updated many times in code during complex update processes without the browser having to re-render and reflow the view until the process is finished. This greatly improves the performance of apps.
2. Render trees can be created by comparing two trees and building a new tree that is the difference between the two. The diff tree offers the ability to represent changes to the view using the smallest number of changes possible to update the DOM. This is called the Incremental DOM technique and improves the UX (e.g. when changing the display).

For Blazor WebAssembly apps the virtual DOM is also used to optimize calls between JavaScript and WebAssembly minimizing the amount of data exchanged.

For Blazor Server the events and the diffs in the client DOM are transferred to the server holding the virtual DOM which is then updated and rendered appropriately. The Incremental DOM technique enables Blazor Server apps to send fewer bytes over the network making them more usable on slow networks.

Chapter 3

Blazor variants

Both Blazor WebAssembly and Blazor Server are based on ASP.NET Razor syntax which is a mix of HTML, C# and specific Blazor tags in order to create dynamic web pages. Both variants compiles to regular .NET Core and .NET Standard DLLs.

3.1 Choosing your flavour

Before creating your application you should decide which variant of Blazor to use. This section describes the main points to take into consideration.

3.1.1 Blazor WebAssembly

Pros

- **.NET:** .NET Code in the browser with no need for external plugins.
- **WebAssembly:** Near native speed thanks to WebAssembly performance and in-browser computation.
- **Shared code:** Code previously used on the server like validation can be shared via libraries and used in the client.
- **Reduced server load:** No server-side dependency means any server load is significantly reduced.
- **Scalability and offline support:** Scalable for serverless and offline scenarios. The whole page can be provided via Content Delivery Network (CDN).
- **PWA:** It can run as a Progressive Web App (PWA) meaning the client can choose to install the app onto their device and run it without network access.

Cons

- **Size:** Initial download size is around 700kb which is not good for users with limited connection, search engines and algorithms alike.
- **Browser compatability:** Blazor WebAssembly is not supported by Microsoft Internet Explorer. All

modern browsers support Blazor.

- **Limited browser sandbox:** Blazor runs in the secure but somewhat limited WebAssembly sandbox. WebAssembly is not aware of the DOM but re-renders pages whenever needed.
- **Tooling still needs to mature:** Debugging is mainly done through the debugging tab in the browser and bugs can be hard to find.
- **Secrets:** Code executed within the browser cannot embed secrets like connection strings

3.1.2 Blazor Server

Pros

- **Size:** No initial download size and the page load is lightweight.
- **Server-Side Rendering (SSR):** All elements are compiled on the server and served as HTML to the client.
- **Faster load time:** Heavy-lifting is done on the server making the client download significantly smaller.
- **Browser compatability:** Works on older browsers. Even those without WebAssembly support.
- **Tooling:** Debugging is done on the server side inside the familiar and robust Visual Studio IDE.

- **Secrets:** API/Server code is private allowing for secrets in the code.

Cons

- **Hosting:** You need an ASP.NET Core server and the app cannot be served from a CDN.
- **Latency:** Every user interaction requires a network hop (using SignalR).
- **Scalability:** States of connected clients must be managed, typically requiring about 85kb of process memory per connection.
- **No offline support:** If the connection is lost, the application stops working.

3.2 Setting up your environment

Start by downloading and installing the .NET SDK (Software Development Kit) from the official Microsoft website.

Also make sure to download and install Visual Studio or Visual Studio Code for your OS of choice (you can create Blazor apps from Visual Studio if you prefer it over the console). Verify that the installation is successful by outputting the .NET version in the console:

```
dotnet --version
```

3.3 Blazor WebAssembly

With Blazor WebAssembly, the browser itself hosts Blazor. When the app is opened in the browser, several files are downloaded:

The files *mono.js* and *mono.wasm* make up the WebAssembly version of the Mono compiler.

The script *blazor.webassembly.js* downloads the .NET runtime, the app with its dependencies and initializes the runtime. It uses the Mono runtime to execute .NET code (located in the *Blazorwasm.dll* file) directly in the browser.

The application itself is then mostly autonomous handling user actions events directly in the browser without any callbacks to a server.

3.3.1 Your first Blazor WebAssembly App

Open your console and navigate to a path of your choice. Then type:

```
dotnet new blazorwasm -o BlazorWasmApp
```

This will create a new Blazor web assembly app. The `-o` command tells dotnet to create a new folder called BlazorWasmApp in which it will place the code. Navigate to this folder using the command:

```
cd BlazorWasmApp
```

Running the app is now as simple as typing:

```
dotnet run
```

Open your web browser of choice and navigate to:

```
http://localhost:5000
```

(or whatever the console says) and voila! Your Blazor WebAssembly App is up and running. Exit by typing Ctrl+C or equivalent.

Tip

Type `dotnet watch run` in the console for automatic builds!

3.3.2 Project Structure

Program.cs: The entry point of the app. The `WebAssemblyHostBuilder` builds the hosting environment of the web application. This is the location where services are added to the app.

wwwroot folder: Contains the static resources of the app. The file index.html contains the basic HTML structure most notably the `<app>` element specifying the location of the app component and the *blazor.webassembly.js* script element which downloads DLLs and initializes the runtime of the app.

App.razor: The root component of the app used for setting up the routing using the `<Router>` component which in turn intercepts browser navigation and renders the page matching the requested address.

Shared folder: Contains UI components used by the app. *MainLayout.razor* is the layout component of the app which in turn sets up the side bar (`<NavMenu>`), the top bar and the main body (`@Body`) of the application where the pages are rendered. The NavLink component of *NavMenu.razor* is used for rendering navigation links to other components.

Pages folder: Contains the routable components and pages used for your Blazor app. The route for each page is specified using the `@page` directive at the top of component files.

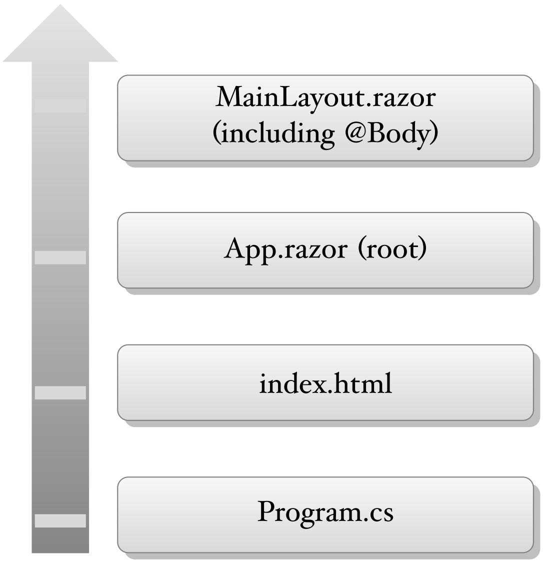

Figure 3.1: Blazor WebAssembly application flow.

3.4 Blazor Server

When running Blazor Server, an ASP.NET Core application hosts it on the server side. Most user events from the browser requires communication via SignalR which is a two-way connection. Only 165KB of data is transferred to the client making it lighter than its Blazor WebAssembly counterpart (2.2 MB) and similar in size to tools like React and Angular.

No actual C# runs on the client side and the browser only has to deal with JavaScript, CSS and HTML. The script *blazor.server.js* is served to the browser from an embedded resource in the ASP.NET Core shared framework. This file is used for intercepting user events as well as establishing the SignalR connection used for bi-directional communication with the server. The client side of the app is expected to persist and restore app state when needed.

SignalR messages can be tracked from the network tab of your browser.

3.4.1 Your first Blazor Server App

Open your terminal and navigate to a path of your choice. Then type:

```
dotnet new blazorserver -o BlazorServerApp
```

This will create a new Blazor server app. The `-o` command tells dotnet to create a new folder called BlazorServerApp in which it will place the code. Navigate to this folder using the command:

```
cd BlazorServerApp
```

Running the app is now as simple as typing:

```
dotnet run
```

Open your web browser of choice and navigate to:

```
http://localhost:5000
```

(or whatever the console says) and voila! Your Blazor Server App is up and running. Exit by typing Ctrl+C or equivalent.

> Tip
>
> Type `dotnet watch run` in the console for automatic builds!

3.4.2 Project Structure

Program.cs: The entry point of the app used for setting up the ASP.NET Core host.

Startup.cs: Contains the startup logic of the app defining two methods: `ConfigureServices` used for configuring dependency injection (DI) of the app. This is the location where services are added. The `Configure` method sets up request handling pipelines of the app (including the

`MapBlazorHub` endpoint used for establishing the SignalR connection with the browser).

Pages folder: Contains the routable components/pages used for your Blazor app along with the root Razor page *Pages/_Host.cshtml* and the Error page. The route for each page is specified using the `@page` directive at the top of component files.

App.razor: The root component of the app used for setting up the client-side routing using the `<Router>` component which in turn intercepts browser navigation and renders the page matching the requested address.

Shared folder: Contains UI components used by the app. *MainLayout.razor* is the layout component of the app which in turn sets up the side bar (`<NavMenu>`), the top bar and the main body (`@Body`) of the application where the pages are rendered. The NavLink component of *NavMenu.razor* is used for rendering navigation links to the other components.

_Imports.razor: Contains common directives to include in components e.g. layouts, components and namespaces.

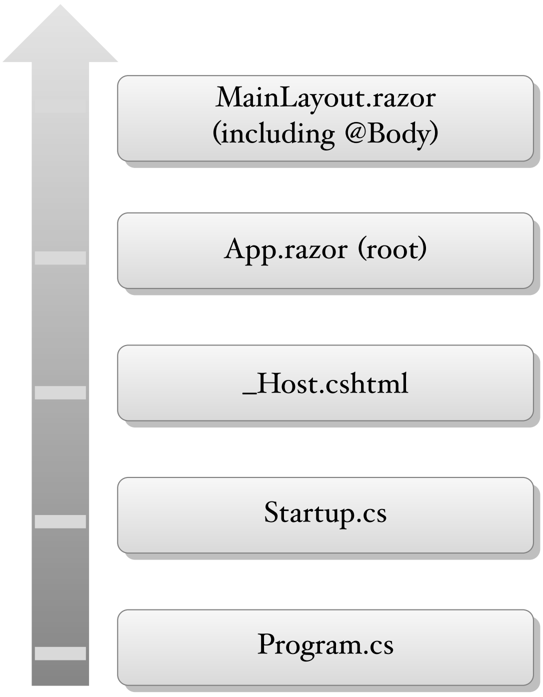

Figure 3.2: Blazor Server application flow.

Chapter 4

Blazor in-depth

4.1 Components

Blazor components are used as smaller units of reusable code for applications. A component consists of three parts:

- **Directives:** Directives are used for routing, importing libraries, injecting services etc. and are located at the top of components.
- **Markup & Razor:** The markup is located in the middle of the component and represents what is displayed to the user.
- **Code & Logic :** At the bottom the variables, methods and lifecycle events of components are handled.

The initial Blazor template comes with three pages (Index, FetchData and Counter). These are components with the `@page` directive defined at the top of the files. This means we can access them in the browser through routing (e.g. `localhost:5000/Counter`).

You can create a component which is not a page by omitting the `@page` directive. Both non-page components and page components can be added to any other component through HTML (e.g. `<Counter></Counter>`). A non-page component cannot be accessed through routing.

4.2 Code-behind

The default component architecture is to have all markup and logic within a single .razor file. This works for simple components but as complexity increases it is preferable to separate the parts of components into separate files. This is achieved using the code-behind approach:

Create a file *[componentName].razor.cs* where *[componentName]* is the component for which you want to use code-behind. The .razor.cs extension is recognized by Visual Studio and is properly nested within the file explorer.

Because the component already occupies the class name the new class must use the partial keyword e.g.:

```
public partial class Counter {
...
}
```

Most of the code from the `@code` block can be copied from the .razor file to the partial class with a few possible changes:

- Additional `using` statements may be required.
- Dependency Injection is written differently using the `[Inject]` attribute applied to properties directly instead of using the `@inject` directive.

4.3 Passing data between components

Imagine a shopping cart containing multiple component items. If you increase the quantity of one item we want the shopping cart to respond to this action and recalculate the total price.

It is often the case that components include sub-components that needs to communicate with each other. This is where data passing comes into the picture.

4.3.1 Parameters

By passing parameters to a component we can communicate from parent to child.

Change the private variable `currentCount` of *Counter.razor* to:

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

The initial value of the counter can now be set from the parent by adding the component with an initial value:

```
<Counter CurrentCount="4"></Counter>
<Counter CurrentCount="7"></Counter>
```

Create a new page *DataPassing.razor* and add the two lines above. The two counters on the page will now start out with displaying the values 4 and 7.

4.3.2 Routing parameters

Routing parameters are used to pass data to a page component. Let us begin by making the Counter header clickable.

Add the following code to the Counter component:

```
<h1>
  <a href="@($"/Counter/{CurrentCount}")">
    Counter
  </a>
</h1>
```

We will now pass the current count of the component into the routing parameter of the Counter page.

Tip

To navigate from one page to another using C# code we can inject a navigation manager `@inject NavigationManager Nav` and then call it from the @code block: `Nav.NavigateTo("Counter");`.

To allow the routing parameter into the component, we need to add it as follows:

```
@page "/counter"
@page "/counter/{CurrentCount:int}"
```

Note that we still allow the previous routing parameter without the current count (first directive).

Also note that if the routing parameter is present it will be cast to an integer. Since the current count is a property it will otherwise be set to the default value of 0 on page load.

Tip

Routing parameters can have different types e.g. `datetime`, `int`, `guid`, `long`, `bool`, `decimal`, `float` and `double`.

4.3.3 Cascading parameters

Say you have a larger tree of parent/child components and you want to pass data to the leaf component. This could be achieved through regular parameters but another way to do it is to use cascading parameters.

Create a new folder called Components and create a file *DisplayCounterComponent.razor*. Make sure to include the folder in *_Imports.razor*:

```
@using BlazorWasmApp.Components
```

Open the newly created *DisplayCounterComponent.razor* and add the following code:

```
<p style="color: @CounterColor">
  Current count: @CounterValue
</p>

@code {
    [CascadingParameter(Name = "CounterValue")]
    public int CounterValue { get; set; }

    [CascadingParameter(Name = "CounterColor")]
    public string CounterColor { get; set; }
}
```

This will display the `CounterValue` of a counter in the color `CounterColor`. Both values will be passed down from its parent components. Open *Counter.razor* and replace the previous counter HTML code to:

```
<CascadingValue Name="CounterValue"
  Value="@CurrentCount">
    <DisplayCounterComponent>
    </DisplayCounterComponent>
</CascadingValue>
```

Here we cascade the `CurrentCount` value as `CounterValue` for the DisplayCounterComponent component. As we still lack the `CounterColor` value, we need to add it in *DataPassing.razor*:

```
<CascadingValue Name="CounterColor"
  Value="@("red")" IsFixed="true">
    <Counter CurrentCount="4"></Counter>
</CascadingValue>
<CascadingValue Name="CounterColor"
  Value="@("blue")" IsFixed="true">
    <Counter CurrentCount="7"></Counter>
</CascadingValue>
```

The first counter will now have a red color and the second a blue. Note that the Counter component does not know anything about `CounterColor`. This is the power of cascading parameters.

Cascading parameters can have a performance hit on applications as it needs to keep track of more data. To improve upon performance, add `IsFixed="true"` wherever you know the value will not change during runtime.

Another thing to keep in mind is that cascading parameters can be hard to maintain if overused as they cannot be tracked in Visual Studio.

4.3.4 Event callbacks

To pass data from a child component to a parent we can use something called Event callbacks. Event callbacks are similar to delegates but which also re-renders the parent component after called.

Add the event callback to the Counter component as follows:

```
@code {
  ...
  [Parameter]
  public EventCallback<int>
    SetCounter { get; set; }

  private void IncrementCount()
  {
    ...
    SetCounter.InvokeAsync(CurrentCount);
  }
}
```

The `SetCounter` callback will be invoked when the value of the button component is incremented. Now, let's add functionality to *DataPassing.razor*:

```
<CascadingValue>
  <Counter ...
    SetCounter=”@((val) => counters[0] = val)”>
  </Counter>
</CascadingValue>
<CascadingValue>
  <Counter ...
    SetCounter=”@((val) => counters[1] = val)”>
  </Counter>
</CascadingValue>

<p>Sum: @(counters[0] + counters[1])</p>

@code {
  private int[] counters = new int[] { 4, 7 };
}
```

In the code block we add an array for the counter values held by the DataPassing component. We display the sum of the two counters in the paragraph.

The `SetCounter` parameter is set for the two Counter components and the value `val` is passed from child to parent. These values are then used to update the counters array and output the sum.

4.3.5 The ref attribute

Elements in the HTML can be referenced by code using the `ref` attribute. In *DataPassing.razor* add the following code:

```
<CascadingValue ...>
  <Counter @ref="counterComponent1" ...>
  </Counter>
</CascadingValue>
<CascadingValue ...>
  <Counter @ref="counterComponent2" ...>
  </Counter>
</CascadingValue>
...
<button
  class="btn btn-secondary"
  @onclick="DisableCounters">
    Disable counters
  </button>

@code {
  ...
  private Counter counterComponent1;
  private Counter counterComponent2;

  private void DisableCounters()
  {
    counterComponent1.DisableCounter();
    counterComponent2.DisableCounter();
  }
}
```

We add the `@ref` attribute to the HTML Counter elements which in turn links them to the `Counter` variables in the code block.

The ”Disable counters”-button calls the `DisableCounters()` method when clicked. This method in turn invokes the method `DisableCounter()` on the two `Counter` components.

Let us add the code to disable the ”Increment”-button in *Counter.razor*:

```
...
<button ... disabled=”@Disabled”>Click me</button>

@code {
  ...
  private bool Disabled { get; set; }
  ...
  public void DisableCounter()
  {
    Disabled = true;
  }
}
```

The ”Disable counters”-button on the DataPassing page will now trigger the `DisableCounter()` method in the Counter child components which disables their respective buttons.

4.4 Data binding

Data binding is used for communicating information between the user interface and the data layer. In a one-way data binding, data is only transferred from the data layer to the UI. In a two-way data binding, data is transferred in both ways (e.g. when the user edits an input field).

Let us begin by creating a new page called *DataBinding.razor* defining a `@code` block:

```
@code {
  class Person
  {
    public string Name { get; set; }
    public int Age { get; set; }
    public bool OpenToWork { get; set; }
    public List<string> Skills { get; set; }
    public int MainSkill { get; set; }
  }

  Person person = null;

  public string NewSkill;

  public void AddSkill()
  {
    person.Skills.Add(NewSkill);
    NewSkill = "";
  }
  protected override void OnInitialized()
  {
    base.OnInitialized();

    person = new Person()
    {
      Name = ”Johan”,
      Age = 31,
      OpenToWork = true,
      Skills = new List<string> {
        ”C#”,
        ”Java”,
        ”Python” },
      MainSkill = 1
    };
  }
}
```

We define a class `Person` with members `Name`, `Age`, a boolean `OpenToWork`, a list of skills and an integer representing the main skill of the person. The method `OnInitialized()` runs when the component is initialized and is in this example used for initializing the person object.

Note the `NewSkill` variable, it will be used for the two-way binding of the input field where new skills are added and in the `AddSkill` method.

4.4.1 One-way data binding

A simple way to bind the data of the `Person` class to the interface is to implement it as follows:

```
<h2>One-way</h2>
<p>
  <b>Person: </b> @person.Name
  @if (person.OpenToWork)
  {
    <span>- Open to work!</span>
  }
</p>
<p>
  <b>Age: </b> @person.Age
</p>
@if (person.Skills.Count > 0) {
  <b>Skills: </b>
  <ol>
    @for (var i = 0;
      i < person.Skills.Count;
      i++)
    {
        <li>
          @person.Skills[i]
          @if (i == person.MainSkill)
           {
             <span> - Main</span>
           }
        </li>
    }
    </ol>
}
```

The `@person` directive references the person variable. We can also reference its class variables using `@person.Name`, `@person.Age` and `@person.OpenToWork` as shown in the example above.

If- and for statements are handled by prepending the statements with `@` (i.e. `@if`, `@for` or `@foreach`). Note that it is possible to call methods directly on the objects
(e.g. `person.Skills.Count`).

4.4.2 Two-way data binding

Let us add some input elements to the interface for updating the data layer as follows:

```
<h2>Two-way</h2>
<p>
  <input type="number"
    @bind-value="person.Age"
    @bind-value:event="oninput" />

  <label>
    <input type="checkbox"
      @bind="person.OpenToWork" />
    Open for work
  </label>
</p>

<p>
    @for (var i = 0;
    i < person.Skills.Count;
    i++)
    {
      var id = i;
      <label>
        <input type=”radio”
          value=”@id”
          name=”mainskill”
          checked=
            ”@(person.MainSkill == id)”
          @onclick=
            ”@(() => person.MainSkill = id)”
        />
        @(id + 1)
      </label>
    }

  <select @bind=”person.MainSkill”>
    @for (var i = 0;
      i < person.Skills.Count;
      i++)
    {
      var id = i;
      <option value=”@id”>
```

```
        @person.Skills[id]
      </option>
    }
  </select>
</p>

<p>
  <label>
    <input type="text" @bind-value="@NewSkill" />
      <button @onclick="AddSkill">Add skill</button>
    </label>
    @NewSkill
</p>
```

The first and second input fields are used for manipulating the person's Age and OpenToWork-status. Note the usage of `@bind-value` along with `@bind-value:event` set to oninput. `@bind` is an override of `@bind-value` with the event set to `onchange`. Based on the event type the value will be updated whenever the input is edited or when the user is done editing (has clicked somewhere else).

In the second paragraph we create radio buttons for the skills of the person using a for-loop. Note that we need to store the index in a separate variable (`id`) if it is to be used in function calls (e.g. `@(() => person.MainSkill = id)`).

Below the radio buttons we have a drop-down variant of the radio-button solution for changing the main skill among the skills.

In the last paragraph we have an input field which binds to the variable `NewSkill`. When the ”Add skill”-button is clicked the `AddSkill` method is called which in turn uses the `NewSkill` value for adding a new skill to the person.

Note how the data layer is updated dynamically when age, status, main skill or skills are added in the user interface.

4.5 Templated Components

A templated component takes one or more UI templates as parameters which are then used as part of the templated component. This allows for higher-level components that are more reusable than regular components. In this chapter, we will work with the component *FetchData.razor* included in default Blazor projects.

4.5.1 Template parameters

Create a new component called *TableTemplate.razor* and add the following code:

```
@typeparam TItem

<table class=”table”>
  <thead>
    <tr>@TableHeader</tr>
  </thead>
  <tbody>
```

```
    @foreach (var item in Items)
      {
        <tr>@RowTemplate(item)</tr>
      }
  </tbody>
</table>

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

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

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

The `@typeparam` is used to specify the type parameters for the component.

Tip

If the type cannot be inferred automatically, we need to add the attribute `TItem="WeatherForecast"` to the TableTemplate component in *FetchData.razor*.

The component uses two render fragments representing segments of the UI to be rendered (`TableHeader` and

`RowTemplate`). These are used for the header and the rows of the component.

The table data to be displayed is stored in the list `Items`. Because `RowTemplate` and `Items` uses generics, we are not interested in what the items represent in this component.

Open *FetchData.razor* and change the previous HTML for the table to:

```
<TableTemplate Items="forecasts">
  <TableHeader>
    <th>Date</th>
    <th>Temp. (C)</th>
    <th>Temp. (F)</th>
    <th>Summary</th>
  </TableHeader>
  <RowTemplate>
    <td>@context.Date.ToShortDateString()</td>
    <td>@context.TemperatureC</td>
    <td>@context.TemperatureF</td>
    <td>@context.Summary</td>
  </RowTemplate>
</TableTemplate>
```

The forecasts are now passed through the `Items` attribute and the type of its elements is generally inferred. We now have the same output as before but with more reusable code.

4.5.2 Template context parameters

The `@context` parameter in the previous example (4.5.1) is implicit and represents the `TItem` type for the render fragments of the `TableTemplate` component.

By adding the `Context` attribute on child elements we can achieve better readability:

```
...
<RowTemplate Context=”forecast”>
  <td>@forecast.Date.ToShortDateString()</td>
...
```

Alternatively, we can add the `Context` attribute to the component and have it apply to all specified template parameters:

```
<TableTemplate Items=”forecasts” Context=”forecast”>
  ...
  <RowTemplate>
    <td>@forecast.Date.ToShortDateString()</td>
...
```

4.6 Lifecycle & rendering triggers

Lifecycle events are used to perform additional operations on components when initialized, rendered etc.

4.6.1 Component Lifecycle Events

If the app is rendering for its first time the instance of the component is created and then enters its lifecycle.

Setting parameters

The parent of the component in the render tree sets the parameters of a component using the `SetParametersAsync` method.

The `SetParametersAsync` method accepts a `ParameterView` object which in turn has corresponding values to the `[Parameter]` and `[CascadingParameter]` attribute properties explained in previous chapters (4.3.1, 4.3.3).

The `SetParametersAsync` method by default invokes `base.SetParametersAsync`. If removed there will be no requirement to assign the incoming parameters to properties of the class.

Component initialization

After parameters are set, the async method `OnInitialized` or `OnInitializedAsync` is performed. Use the async variant when the component should be refreshed upon performing asynchronous operations:

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

Tip

Blazor Server applications that pre-render their content will call `OnInitializedAsync` twice: Statically as part of the page and when the connection to the server is established.

After parameters are set

A call to async method `OnParametersSet` or `OnParametersSetAsync` is performed after parameters are set. If no `Task` is returned, the component is rendered immediately. Otherwise the `Task` is awaited followed by the component being rendered.

`OnParametersSet` is also called when the parent component is re-rendered and supplies primitive immutable types where at least one parameter has changed along with any complex-typed parameters.

After component is rendered

The methods `OnAfterRender` and `OnAfterRenderAsync` are called when the component has finished rendering. Since element component references are populated at this point it is a good place to make JS Interop calls (4.14) on rendered DOM elements.

> **Tip**
>
> The `firstRender` parameter of `OnAfterRender` and `OnAfterRenderAsync` is set to true if the component is rendered for its first time. Use it to your advantage.

Note also the `ShouldRender` method called each time the component is rendered except on initial rendering. If overridden, the developer can suppress rendering by returning the value `false`.

DOM event processing

Document Object Model (DOM) processing is performed whenever UI events are triggered (e.g. a button is clicked):

The event handler is run. If no Task is returned, the component is rendered immediately. Otherwise the Task is awaited followed by the component being rendered.

The Render lifecycle

The Render lifecycle is performed when component state is changed:

If the component hasn't rendered for the first time or the component member `ShouldRender` is evaluated as false no further operations are performed on the component. Otherwise the render tre difference (diff) is built followed by the component being rendered.

Finally the DOM is awaited to update followed by a call to async method `OnAfterRender`.

The `ShouldRender` variable can be set using code:

```
@code {
  protected override bool ShouldRender() => false;
  ...
}
```

State changes

Use the method `StateHasChanged()` to notify the app that its state has changed. This will cause the component to be re-rendered.

`StateHasChanged()` is called automatically for `EventCallback` methods.

4.7 Forms and Validations

Blazor handles forms and validation using data annotations. Create a new folder *Models* and add the file *Character.cs* with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace BlazorWasmApp.Models
{
  public class Character
  {
    [Required]
    [StringLength(16, ErrorMessage =
      ”Name too long (16 character limit).”)]
    public string Name { get; set; }

    public string Description { get; set; }

    [Required]
    public string Class { get; set; }

    [Range(1, 100, ErrorMessage = ”
      Age invalid (1-100).”)]
    public int Age { get; set; }

    [Required]
    [Range(typeof(bool), ”true”, ”true”,
      ErrorMessage =
        ”Character is not approved.”)]
    public bool IsApproved { get; set; }

    [Required]
    public DateTime CreatedDate { get; set; }
  }
}
```

Validation attributes allows for validation rules on model properties. Following are the built-in validation attributes:

- `[Compare]`: Two properties in the model match.
- `[EmailAddress]`: Has email format.
- `[Phone]`: Has telephone number format.
- `[Range]`: Falls within specified range.
- `[RegularExpression]`: Matches specified regular expression.
- `[Required]`: Is not null.
- `[StringLength]`: Does not exceed specified length.
- `[Url]`: Has URL format.
- `[Remote]`: Validates by calling an action method on server.

Make sure to add the namespace to *_Imports.razor*:

```
...
@using BlazorWasmApp.Models
```

Create a new page *FormsValidation.razor* and add the code:

```
@page ”/FormsValidation”

<h1>New Character Entry Form</h1>

<EditForm Model=”@character”
  OnValidSubmit=”@HandleValidSubmit”>
  <DataAnnotationsValidator />
  <ValidationSummary />

  <p>
    <label>
      Name:
      <InputText @bind-Value=”character.Name” />
    </label>
  </p>
  <p>
    <label>
      Description (optional):
        <InputTextArea @bind-Value=
          ”character.Description” />
    </label>
  </p>
  <p>
```

```
    <label>
      Class:
      <InputSelect @bind-Value=”character.Class”>
        <option value=””>
          Select class ...
        </option>
        <option value=”Warrior”>
         Warrior
        </option>
        <option value=”Thief”>
          Thief
        </option>
        <option value=”Magician”>
          Magician
        </option>
      </InputSelect>
    </label>
  </p>
  <p>
    <label>
      Age:
      <InputNumber
        @bind-Value=”character.Age” />
    </label>
  </p>
  <p>
    <label>
      Is approved:
      <InputCheckbox
```

```
        @bind-Value="character.IsApproved" />
    </label>
  </p>
  <p>
    <label>
      Creation date:
      <InputDate
        @bind-Value="character.CreatedDate" />
    </label>
  </p>

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

</EditForm>

@code {
  private Character character = new Character() {
    CreatedDate = DateTime.UtcNow
  };

  private void HandleValidSubmit()
  {
    ...
  }
}
```

The `EditForm` component is used to define a form which is then validated using the data annotations defined in *Character.cs*. Make sure to add the `DataAnnotationsValidator` component to support validation. The `ValidationSummary` component is used to display the validation messages.

Tip

The `EditForm` component creates an `EditContext` as a cascading value that tracks metadata about the edit process (e.g. which fields have been modified and the current validation messages).

`EditForm` also provides the useful events `OnValidSubmit` and `OnInvalidSubmit` the former used to trigger the `HandleValidSubmit` method when the form is successfully submitted and has passed validation.

The value properties of the input elements are bound to the model properties using `@bind-Value`. It also binds a change event delegate to the input component's `ValueChanged` property.

4.7.1 Built-in form components

The built-in form components support arbitrary attributes and provides default validation behavior when the field is changed. This includes updating the CSS classes as well as useful parsing logic (e.g. `InputNumber<TValue>` only treats numbers as valid).

Some of the built-in form components support display names using the `DisplayName` parameter. This allows its value to be used in validation errors.

Following are the built-in components available:

- `InputCheckbox`
- `InputDate<TValue>` (Supports `DisplayName`)
- `InputFile`
- `InputNumber<TValue>` (Supports `DisplayName`)
- `InputRadio`
- `InputRadioGroup`
- `InputSelect<TValue>` (Supports `DisplayName`)
- `InputText`
- `InputTextArea`

Tip

The `InputFile` component by default selects single files but can be extended to multiple files using the `multiple` attribute. When files are selected, the component fires an `OnChange` event and passes in `InputFileChangeEventArgs` which provides access to the selected file list including file details. Files are then read using `Streams`. For more information visit `https://docs.microsoft.com/en-us/aspnet/core/blazor/file-uploads?view=aspnetcore-5.0`

4.7.2 Fluent validation

Another way of doing validation is to use the *FluentValidation* package. Start by adding it to your project:

```
dotnet add package Blazored.FluentValidation
```

Make sure to add it to *Imports.razor*:

```
@using Blazored.FluentValidation
```

Open *Character.cs* and add the validator:

```
using FluentValidation;
...
public class CharacterValidator :
  AbstractValidator<Character>
  {
    public CharacterValidator()
    {
      RuleFor(x => x.Name).NotEmpty().Length(3, 16);
      RuleFor(x => x.Class).NotEmpty();
    }
  }
```

Open *FormsValidation.razor* and add the fluent validator:

```
...
@using Blazored.FluentValidation
...
<EditForm ...>
  <FluentValidationValidator />
...
```

4.8 Dependency Injection

DI (Dependency Injection) is a technique used for accessing services configured in a central location. DI is useful for decoupling components from concrete service classes and more generally for code to be swapped more easily.

Blazor applications support the use of built in- and custom services using DI. Following are default services added to Blazor applications:

- **HttpClient:** Provides methods for sending and receiving HTTP requests and responses from a resource. Only supplied by default in Blazor WebAssembly apps.

- **IJSRuntime:** Provides the JavaScript runtime from which JavaScript calls are dispatched.
- **NavigationManager:** Contains helpers for working with URIs and navigation state.

Service lifetimes

- **Scoped:** The scoped lifetime is a concept in Blazor Server apps where the service is to be used for the scope of a connection (user). The service itself works just like a Singleton service.
- **Singleton:** All components requiring a Singleton service receive an instance of the same service. DI creates a single instance of the service.
- **Transient:** The component receives a new instance of the service whenever the Transient service is obtained from the service container.

4.8.1 Blazor WebAssembly DI

Create a new folder *Services* and add the file *EmailService.cs*. Open the newly created file and set up the basic structure:

```
namespace BlazorWasmApp.Services
{
  public interface IEmailService
  {
    bool SendEmail(
      string sender,
      string address,
      string subject,
      string body);
  }

  public class EmailService : IEmailService
  {
    public bool SendEmail(
      string sender,
      string address,
      string subject,
      string body)
    {
      // insert implementation
      return false;
    }
  }
}
```

Also make sure to add the line `@using BlazorWasmApp.Services` to *_Imports.razor*.

We can now add the service to the `Main`-method of *Program.cs*:

```
public static async Task Main(string[] args)
{
  ...
  builder.Services.AddSingleton<EmailService>();

  await builder.Build().RunAsync();
}
```

To use the service in a page. Inject the service using `@inject EmailService Email` at the top of the file and call it using `Email.SendEmail(”...”, ”...”, ”...”, ”...”)`.

Note that services are injected after the component instance is created and before it is initialized. You may therefore want to call the service from the `OnInitialized` or `OnInitializedAsync` method:

```
@page ...
@inject EmailService Email
...

@code {
  ...
  protected override async Task OnInitializedAsync()
  {
    ...
    Email.SendEmail("", "", "", "");
  }
  ...
}
```

4.8.2 Blazor Server DI

Blazor Server dependecy injection works similar to Blazor WebAssembly with a small difference in how the service is added. Open *Startup.cs* and add the following code to the `ConfigureServices` method:

```
public void ConfigureServices(
  IServiceCollection services)
{
  ...
  services.AddSingleton<IEmailService,
    EmailService>();
  ...
}
```

The `IServiceCollection` is a list of service descriptor objects to which services are added. The services can then be used in components in the same way described for Blazor WebAssembly (4.8.1).

To inject a service into another service, the constructor is used e.g.:

```
public class EmailService : IEmailService
{
  public EmailService(HttpClient http)
  {
    ...
  }
}
```

4.9 Authentication & Authorization

Configuration and management of security in Blazor apps is supported by existing ASP.NET Core mechanisms.

> **Tip**
>
> Authentication confirms that users are who they say they are. Authorization gives users permission to access a resource (typically after authentication).

4.9.1 Blazor Server Auth

Authentication in SignalR based apps is handled when the connection from client to server is established and can be based on a cookie or some other token.

Create a new Blazor Server App project in Visual Studio and select Individual Authentication (in-app) to create a project including local user account store.

Open *appsettings.json* and locate the `DataSource` value of the `Connection String`. This is the location of the database. Also notice that the folder *Areas/Identity* has been created which contains pages for logging users in and out along with an identity provider. The folder *Data* contains database migrations and an *ApplicationDbContext* used for migrating the database with Entity Framework.

Tip

To manage Entity Framework migrations run `dotnet tool install --global dotnet-ef` and perform updates using `dotnet ef database update`.

Run the application and create two users. Name them **user@blazor.com** and **admin@blazor.com**. The *app.db* file has now been updated with the two users. Open *app.db* using any database browser and run the following SQL:

```
INSERT INTO AspNetRoles (Id, Name, NormalizedName)
VALUES (”User”, ”User”, ”User”),
  (”Admin”, ”Admin”, ”Admin”);

INSERT INTO AspNetUserRoles (UserId, RoleId)
SELECT Id, ”User” as RoleId
FROM AspNetUsers;

INSERT INTO AspNetUserRoles (UserId, RoleId)
SELECT Id, ”Admin” as RoleId
FROM AspNetUsers
WHERE UserName = ’admin@blazor.com’;
```

This adds the roles **User** and **Admin** to the database. Open *Startup.cs* add add the roles to the application:

```
services.AddDefaultIdentity<IdentityUser>(options =>
  options.SignIn.RequireConfirmedAccount = true)
```

```
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

In *Index.razor* add:

```
  <AuthorizeView>
    <Authorized>
        <p>Is authorized</p>
        <p>User: @User.IsInRole("User");</p>
        <p>Admin: @User.IsInRole("Admin");</p>
    </Authorized>
    <NotAuthorized>
        <p>Is not authorized</p>
    </NotAuthorized>
</AuthorizeView>
```

When the application is run the home page should now display whether the user is authenticated or not and if authenticated which roles it has.

The `AuthorizeView` can be passed a `Roles` parameter in order to only display the content for a user with the specific role. This can be useful for hiding certain menu items from users:

```
<AuthorizeView Roles="User">
  ...
  <NavLink class="nav-link" href="counter">
  ...
</AuthorizeView>
```

Since the user can still access a page without clicking the menu item (e.g. `localhost:5000/Counter`) the component itself can use authorization to disallow access. Add the following line to the top of *Counter.razor*:

```
@attribute [Authorize(Roles = ”User”)]
```

The component is now inaccessible for any user without the **User** role.

You can change the message displayed to unauthenticated users in *App.razor* as follows:

```
...
<AuthorizeRouteView RouteData=”@routeData”
  DefaultLayout=”@typeof(MainLayout)”>
  <NotAuthorized>
    Denied!
  </NotAuthorized>
</AuthorizeRouteView>
...
```

4.9.2 AuthenticationStateProvider service

Use `AuthenticationStateProvider` to get user identity and role (claims) for usage in code:

```
@using System.Security.Claims
@inject AuthenticationStateProvider Auth
...
@code {

  private ClaimsPrincipal User;
  ...

  private void IncrementCount() {
    ...
    if (User.Identity.IsAuthenticated
      && User.IsInRole(”Admin”)) {
      currentCount++;
    }
  }

  protected async override Task
  OnInitializedAsync() {
   var auth = await
     Auth.GetAuthenticationStateAsync();

   User = auth.User;
  }
}
```

The `AuthenticationStateProvider` is injected at the top of the page and is used for retrieving the authentication state in the `OnInitializedAsync()` method.

The `User` is retrieved as a `ClaimsPrincipal` holding a collection of `ClaimsIdentity` identities. The `ClaimsPrincipal` can be queried for authentication and authorization as seen in the code above.

Only authenticated users with the **Admin** role are now allowed to click the button.

4.9.3 Blazor WebAssemby Auth

Create a new Blazor WebAssembly App project in Visual Studio and select Individual Authentication (in-app) to create a project that includes a local user account store. Make sure to select the ”ASP.NET Core Hosted” checkbox.

Basic authentication works out of the box but a couple of additional steps are required for adding authorization to the app. Follow the steps for Blazor Server Authentication & Authorization (4.9.1) to set up the server side part of the WebAssembly application.

The server part uses Identity Server for serving roles as a JSON array in a single role claim. A single role is sent as a string value in the claim. To handle this we need to implement a custom user factory for unboxing this data on the client side. We also need to enable roles in the server part.

In the client part create the file *CustomUserFactory.cs* in the root folder:

```
using System.Linq;
using System.Security.Claims;
using System.Text.Json;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.
WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.
WebAssembly.Authentication.Internal;

public class CustomUserFactory
    : AccountClaimsPrincipalFactory<RemoteUserAccount>
{
  public CustomUserFactory
  (IAccessTokenProviderAccessor accessor)
    : base(accessor)
  {
  }

  public async override ValueTask<ClaimsPrincipal>
  CreateUserAsync(
    RemoteUserAccount account,
    RemoteAuthenticationUserOptions options). {

    var user = await
    base.CreateUserAsync(account, options);
```

```
if (user.Identity.IsAuthenticated)  {
  var identity = (ClaimsIdentity)user.Identity;
  var roleClaims =
  identity.FindAll(identity.RoleClaimType)
  .ToArray();

  if (roleClaims != null && roleClaims.Any())  {
    foreach (var existingClaim in roleClaims)  {
      identity.RemoveClaim(existingClaim);
    }

    var rolesElem =
    account.AdditionalProperties[
    identity.RoleClaimType];

    if (rolesElem is JsonElement roles). {
      if (roles.ValueKind ==
      JsonValueKind.Array)  {
        foreach (var role in
        roles.EnumerateArray())  {
          identity.AddClaim(
            new Claim(options.RoleClaim,
            role.GetString()));
        }
      }
      else  {
        identity.AddClaim(
          new Claim(options.RoleClaim,
          roles.GetString()));
          }
        }
      }
    }

    return user;
  }
}
```

Make sure to add the service to *Program.cs* in the client project:

```
builder.Services.AddApiAuthorization()
  .AddAccountClaimsPrincipalFactory
  <CustomUserFactory>();
```

In *Startup.cs* of the server project configure Identity Server to handle roles:

```
services.AddIdentityServer()
  .AddApiAuthorization
  <ApplicationUser, ApplicationDbContext>
  (options =>
  {
    options.IdentityResources["openid"]
    .UserClaims.Add("name");
    options.ApiResources.Single()
    .UserClaims.Add("name");
    options.IdentityResources["openid"]
    .UserClaims.Add("role");
    options.ApiResources.Single()
    .UserClaims.Add("role");
  });
```

Make sure to add the required libraries to *_Imports.razor* in the client project:

```
@using Microsoft.AspNetCore.Authorization
@using System.SecurityClaims
```

Authentication and authorization now works in the Blazor WebAssembly.

4.10 State management

The user state include the hierarchy of component instances including fields, properties and their most recent render output as well as data held in DI service instances and values set through JavaScript Interop calls. The user state for a Blazor WebAssembly app is held in the memory of the browser and is lost when the browser is closed or reloaded.

To preserve state across sessions the app must persist data to some other storage location than the browser memory. This is useful for multi-step web forms, shopping carts etc.

4.10.1 Server-side storage

Use independent server-side storage accessed via web API for permanent data persistance spanning multiple users and devices. Options for server-side storage include: Blob storage, key-value storage, relational database, table storage etc.

Additional measures of security are needed in Blazor WebAssembly apps as they run completely in the browser. Typically, the app authenticates via OAuth/OpenID Connect (OIDC) and then interacts with the server via web API calls. The server acts as the mediator between the Blazor WebAssembly app and the storage device or database.

4.10.2 URL

Use the URL for transferring data representing navigation state. This is useful for IDs of viewed entities, the current page number in a paged grid etc.

Tip

Contents of the address bar in the browser is retained when the user reloads the page.

4.10.3 Browser storage

A common location for storage in the browser is the `localStorage` and `sessionStorage`. These storages can be used in Blazor WebAssembly apps but only by writing custom code or by usage of third-party packages.

The `localStorage` is scoped to the window of a browser and is persisted when the user closes or reloads the browser.

Also note that the `localStorage` is shared across tabs and is persisted until explicitly cleared.

The `sessionStorage` is scoped to the tab of a browser and is persisted when a tab is reloaded. The `sessionStorage` state is lost when the tab or browser is closed. Also note that each tab has its own independent `sessionStorage` state.

Use `sessionStorage` to avoid the risk of bugs in state storage across tabs or tabs overwriting the state of other tabs. Use `localStorage` for persisting state across closing and opening the browser.

Warning

The data stored in `localStorage` and `sessionStorage` can be altered manually by the user. Do not store secret information in these locations.

4.10.4 State container service

Components can share access to data using a registered in-memory state container. Create a custom state container class with an assignable `Action` to notify components of state changes:

```
public class StateContainer
{
    public string Property { get; set; }
      = "Initial value";

    public event Action OnChange;

    public void SetProperty(string value)
    {
        Property = value;
        NotifyStateChanged();
    }

    private void NotifyStateChanged() =>
      OnChange?.Invoke();
}
```

Make sure to add the service to `Program.Main` (Blazor WebAssembly):

```
builder.Services.AddSingleton<StateContainer>();
```

Or `Startup.ConfigureServices` (Blazor Server):

```
services.AddSingleton<StateContainer>();
```

Use the `StateContainer` in your component:

```
@inject StateContainer StateContainer
@implements IDisposable
...
<button @onclick=”ChangePropertyValue”>
  Set Property
</button>
...
@code {
    protected override void OnInitialized()
    {
        StateContainer.OnChange +=
          StateHasChanged;
    }

    private void ChangePropertyValue()
    {
        StateContainer.SetProperty(
          $”New value set {DateTime.Now}”);
    }

    public void Dispose()
    {
        StateContainer.OnChange -=
          StateHasChanged;
    }
}
```

4.11 Component Virtualization

Blazor supports built-in virtualization which is a technique for limiting UI rendering to just the parts currently visible. This is useful when the app is rendering long lists of items where only a subset of the items are required to be visible.

This is best explained in the FetchData component of a new Blazor Server project. Open *WeatherForecastService.cs* and increase the number of forecasts to 500:

```
public class WeatherForecastService
{
  public Task<WeatherForecast[]>
  GetForecastAsync(DateTime startDate)
  {
    ...
    return Task.FromResult(
    Enumerable.Range(1, 500).Select(...
    ...
  }
  ...
```

In *FetchData.razor* switch out the for-loop to a `Virtualize` component:

```
<Virtualize
  Items="forecasts"
  Context="forecast">
  <tr>
    <td>@forecast.Date.ToShortDateString()</td>
    <td>@forecast.TemperatureC</td>
    <td>@forecast.TemperatureF</td>
    <td>@forecast.Summary</td>
  </tr>
</Virtualize>
```

The `Virtualize` component automatically calculates how many items to render based on the height of the container and the size of rendered items.

Note that even though the DOM only renders a set number of items, all the items are still loaded into memory. To prevent this, you can provide an items provider delegate method to the `ItemsProvider` parameter of the `Virtualize` component:

```
<Virtualize
  ItemsProvider="@LoadForecasts"
  Context="forecast">
  <ItemContent>
    <tr>
      ...
    </tr>
  </ItemContent>
  <Placeholder>
    <p>
      Loading...
    </p>
  </Placeholder>
</Virtualize>

@code {
  ...
  protected async ValueTask<ItemsProviderResult
  <WeatherForecast>> LoadForecasts
  (ItemsProviderRequest request)
    {
        return new ItemsProviderResult
        <WeatherForecast>(
        forecasts.Skip(request.StartIndex)
        .Take(request.Count),
        forecasts.Count());
    }
}
```

The items provider now receives an `ItemsProviderRequest` specifying the required number of items starting at `request.StartIndex` requiring `request.Count` items. This information can be used to lazily load data as needed.

Notice the use of `<Placeholder>` in the HTML. The content of the placeholder is displayed when data is loaded which improves the user experience.

Tip

The Virtualize component can be customized using parameters `ItemSize` to specify the item size in pixels (default 50px) or `OverscanCount` to determine how many additional items are rendered before and after the visible region (default 3).

4.12 CSS Isolation

CSS isolation is used to prevent dependencies on global styles and to avoid styling conflicts among components and libraries.

Add the file *FetchData.razor.css* to your pages folder with the following code:

```
h1 {
  color: green;
}
```

Blazor will now apply the CSS in the file exclusively to the FetchData component. If the app is run you will notice that only the FetchData page now shows the header in a green color.

When the application is built, Blazor rewrites CSS selectors to match markup rendered by the component. The CSS styles are then bundled and produced as the static asset `{PROJECT NAME}.styles.css`.

Use the `::deep` combinator in the CSS file to apply CSS rules to descendant components:

```
::deep h1 {
  color: green;
}
```

Also note that descendant components must be wrapped by an outer element:

```
<div>
  <h1>Parent>
  <Child />
</div>
```

4.13 Debugging

Blazor WebAssembly applications can be debugged in both the browser dev tools (Microsoft Edge/Google Chrome) and in the Visual Studio IDE. Blazor Server apps are mainly debugged in the IDE as the browser is used for rendering.

4.13.1 IDE debugging

For Blazor Server applications, IDE debugging works out of the box as the application is self-contained.

Open *Counter.razor* and add a breakpoint where the `CurrentCount` is incremented (`CurrentCount++;`). Whenever the button on the Counter page is clicked you can now inspect its value in the IDE. Use single stepping (F10) or resume code execution (F8).

To enable debugging for your Blazor WebAssembly app, open *launchSettings.json* and add the following line to each launch profile:

```
"inspectUri": "{wsProtocol}://{url.hostname}:
{url.port}/_framework/debug/ws-proxy?
browser={browserInspectUri}"
```

This tells the IDE that the app is a Blazor WebAssembly app and instructs the script debugging infrastructure to connect to the browser through Blazor's debugging proxy.

To start debugging, press F5 (or equivalent) in Visual Studio. Breakpoints can now be set on lines in the code.

Browser debugging

For Blazor WebAssembly apps you might want to use your browser for debugging.

Run a Debug build of your app in the Development environment and launch Google Chrome or Microsoft Edge. Navigate to the URL of the app (`localhost:5000` or similar).

In the browser, commence remote debugging by pressing Shift+Alt+d. If remote debugging is not enabled in your browser you will be shown an error page with instructions on how to enable it.

The Sources tab will now show a list of the .NET assemblies of your app within the file:// node. Any breakpoints set in component code (.razor) and C# code (.cs) are hit when the code executes. Use single stepping (F10) or resume code execution (F8).

For Blazor Server apps the Networks tab of the developer tools shows data received from the server like static resources, CSS styling and fonts. It also shows the WebSocket connection established to the server.

4.13.2 Logging

Logging to the browser is a great way of keeping track of user activity and for debugging related issues. This is easily achieved in Blazor.

Basic logging

Basic logging to the browser console from C# is as simple as typing:

```
Console.WriteLine(”Logging message”);
```

Ilogger

Another way of logging is to use Ilogger which comes out of the box. Open *Program.cs* and set the logging level:

```
builder.Services.AddLogging(
  builder => builder.SetMinimumLevel
  (LogLevel.Trace));
```

In your component, inject and use the Ilogger:

```
...
@inject Ilogger<Counter> Logger
...
@code{
   ...
   Logger.LogWarning(”Logging message”);
   ...
}
```

Serilog

Serilog has been around for a long time and is a good example of how existing NuGet packages can be reused in your Blazor applications. Since Serilog is based on sinks we can also use the logging data on the server which is highly useful.

Add the Serilog package by running:

```
dotnet add package Serilog
dotnet add package Serilog Sinks.BrowserConsole
```

Open *Program.cs* and add the logger:

```
using Serilog;
...
Log.Logger = new LoggerConfiguration()
  .MinimumLevel.Debug()
  .WriteTo.BrowserConsole()
  .CreateLogger();
```

Inject and use Serilog in your component:

```
@inject Serilog
...
@code{
   ...
   Logger.Information(”Information message”);
   Logger.Warning(”Warning message”);
   Logger.Error(”Error message”);
}
```

4.14 JavaScript Interop

For specific scenarios you might want to use snippets of JavaScript code for your Blazor apps (even though Blazor is built to be independent from it).

This is called JavaScript Interoperability or JavaScript Interop. Blazor handles calls to JavaScript from .NET as well as calls to .NET from JavaScript.

4.14.1 Calling JavaScript from .NET

Create a new page and name it *JsInterop.razor*. Add the following code:

```
@page ”/js”
@inject IJSRuntime JS

<h1>JS Interop</h1>

<input type=”number”
  @bind-value=”num1”
  @bind-value:event=”oninput” />

<input type=”number”
  @bind-value=”num2”
  @bind-value:event=”oninput” />

<button
  class=”btn btn-primary”
  @onclick=”Add”>
  Add
</button>
<button
  class=”btn btn-primary”
  @onclick=”Subtract”>
  Subtract
</button>

<p>Result: @result</p>

@code {
  private int num1;
  private int num2;
  private int result;

  private async Task Add()
  {
    result = await JS.InvokeAsync<int>
      (”math.add”, num1, num2);
  }

  private async Task Subtract()
  {
    result = await JS.InvokeAsync<int>
      (”math.subtract”, num1, num2);
  }
}
```

For this example, the JavaScript runtime is injected using the `@inject IJSRuntime` abstraction. Alternatively the runtime can be injected and used directly into a class:

```
public class JsInteropClasses
{
  private readonly IJSRuntime js;

  public JsInteropClasses(IJSRuntime js)
  {
    this.js = js;
  }
  ...
```

The `InvokeAsync` method accepts an identifier for the JavaScript function to be called along with any number of argument parameters. In this example, we use the values `num1` and `num2`. The function is expected to return an integer.

Tip

Return type `T` of `InvokeAsync<T>` must be JSON serializable.

Tip

JavaScript functions returning a Promise are called using `InvokeAsync` which in turn unwraps the Promise and returns the value awaited by the Promise.

To add the JavaScript functions we create a folder *www-root/js* in which we add a file called *math.js*. Add the following code to *math.js*:

```
window.math = {
  add: function (a, b) {
    return a + b;
  },
  subtract: function (a, b) {
    return a - b;
  }
};
```

The functions `add` and `subtract` are added to the global scope (`window`) which makes them accessible to from our page. Reference this script from *wwwroot/index.html* (WebAssembly) or *Pages/_Host.cshtml* (Server):

```
<!DOCTYPE html>
<html>
<head>
  ...
  <script src="js/math.js"></script>
</head>
...
```

Run your application and visit `localhost:5000/js`. The very basic calculator should work as intended.

4.14.2 Calling .NET from JavaScript

Blazor also handles .NET calls from JavaScript code. In this section we will add decimal to binary functionality in the component triggered from our previous JavaScript code (4.14.1).

Static .NET calls

Open *JsInterop.razor* and add the following code:

```
...
<button type="button" class="btn btn-primary"
  onclick="math.logBinaryAsync(@result)">
  Console.log @result to binary
</button>
...
@code {
  ...
  private int resultBin;
  ...
  [JSInvokable]
  public static Task<string>
  ReturnBinaryAsync(int dec)
  {
    return Task.FromResult(
    Convert.ToString(dec, 2));
  }
}
```

Here we add the method `ReturnBinaryAsync` which accepts a decimal value and converts it into a binary string.

When the new button is clicked it is supposed to trigger the JavaScript function `math.logBinaryAsync`. Let us add it in *wwwroot/js/math.js*:

```
window.math = {
  ...
  logBinaryAsync: function (dec) {
    DotNet.invokeMethodAsync('BlazorWasmApp',
    'ReturnBinaryAsync', dec)
      .then(data => {
        console.log(data);
      });
    }
};
```

The `logBinaryAsync` function invokes the static .NET method `ReturnBinaryAsync` passing the decimal value as a parameter and returning a promise. We then log the calculated binary string to the console.

Component instance .NET calls

Remove the "Log to binary"-button and rewrite the result paragraph in *JsInterop.razor*:

```
<p>Result: @result
    @if (resultBin != null)
    {
    <span>(bin @resultBin)</span>
    }
</p>
```

Also rewrite and add the following code in the `@code` block:

```
@code {
  private string resultBin;
  private static Action<string> action;
  ...
  protected override void OnInitialized()
  {
    action = UpdateResultBin;
  }

  private void UpdateResultBin(string bin)
  {
    resultBin = bin;
    StateHasChanged();
  }

  [JSInvokable]
  public static Task<string> ReturnBinaryAsync(int dec)
  {
    var bin = Convert.ToString(dec, 2);
    action.Invoke(bin);
    return Task.FromResult(bin);
  }
}
```

The `ReturnBinaryAsync` method will now wrap the call to the instance method `UpdateResultBin` as an invoked action. The `UpdateResultBin` method will then change the actual state of the component.

We can now remove the `console.log` call and clean upp *math.js* as follows:

```
window.math = {
  add: function (a, b) {
  let result = a + b;
  DotNet.invokeMethodAsync('BlazorWasmApp',
    'ReturnBinaryAsync', result)
    return result;
  },
  subtract: function (a, b) {
    let result = a - b;
    DotNet.invokeMethodAsync('BlazorWasmApp',
      'ReturnBinaryAsync', result);
        return result;
    }
};
```

Tip

Blazor Server apps with several concurrent users using the same component should use a helper class to invoke instance methods. For more information visit: `https://docs.microsoft.com/en-us/aspnet/core/blazor/call-dotnet-from-javascript`.

4.14.3 npm packages in Blazor

Create a folder *NpmJs* in the root of the project. Navigate to this folder in the command line and run:

```
npm init -y
```

This will create a *package.json* file in the *NpmJs* directory. Next up we need a bundler for the JavaScript files. Install webpack and the webpack CLI as developer dependencies by running:

```
npm install webpack webpack-cli --save-dev
```

Add a folder named *src* under the *NpmJs* directory and create a file *index.js* in it.

Modify *package.json* to use *index.js* as the source file and set the output directory to be placed in the *js* folder of the *wwwroot* directory:

```
...
"scripts": {
  "build": "webpack ./src/index.js --output-path
  ../wwwroot/js --output-filename index.bundle.js"
}
```

Modify the .csproj file to set up a pre-build step for installing and building the npm packages:

```
...
<Target Name="PreBuild"
BeforeTargets="PreBuildEvent">
  <Exec Command="npm install"
  WorkingDirectory="NpmJS" />
  <Exec Command="npm run build"
  WorkingDirectory="NpmJS" />
</Target>
```

Finally make sure to include the *index.bundle.js* file to your application in *wwwroot/index.html*:

```
<script src="js/index.bundle.js"></script>
```

The bundled JavaScript file will now be called *index.bundle.js*.

We can now install any npm packages in the *NpmJs* directory using:

```
npm install ...
```

Use *index.js* to initialize the JavaScript components or to set up functions for interacting with them as explained in the previous section (4.14.2).

Chapter 5

Blazor in the real world

5.1 In the Cloud

In this section we will discuss Blazor hosting in the cloud and more specifically using Microsoft Azure.

Azure is a cloud computing service for building, testing, deploying and managing applications and services through Microsoft managed data centers. Azure provides software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS).

5.1.1 Publishing to Azure

In the Azure Portal, create a resource group and give it a name, subscription and location. Optionally you can create a new server with a database and use its connection strings in your application.

From Visual Studio, right-click the project and select "Publish to Azure...". Select "App Service" and "Create New". Log in to your Azure account and fill out the fields for the app. Make sure to select your newly created resource group and note that the public URL will be:
`<App-Name>.azurewebsites.net.`

After the deployment is successful the website will be launched in the browser. Subsequent publishes will be a 1-click operation and not require filling out any information.

5.1.2 Azure AD B2C

From the Azure Portal, create a new resource "Azure Active Directory B2C" and create a tenant. Also make sure to link the tenant to an Azure subscription. Each tenant in Azure can access the same services independently as long as they are linked to a subscription.

Azure AD B2C allows for 3rd party service providers as the identity provider (e.g. Facebook, Google, Twitter etc.). Navigate to the "Identity providers" tab and select your connectors. Typically you need to fill in a client ID and a secret to configure them.

Navigate to ”User flows” to set up policies for sign up, sign out, password reset etc. Create a user flow named `signup_signin` and select the identity provider(s) you want to use along with the user attributes and token claims to be collected and returned. Do the same for a user flow named `password_reset`.

Go to ”App registrations” and register a new app. Type `https://localhost:5000/signin-oidc` or similar in the Redirect URI field. Note down the Application-ID.

Create a new Blazor Server application and select ”Individual User Accounts” and ”Connect to an existing user store in the cloud”. Fill out the form as follows:

- **Domain Name:** `<Tenant-Name>.onmicrosoft.com`
- **Application ID:** `<Application-ID>`
- **Sign-up or Sign-in Policy:** `signup_signin`
- **Reset Password Policy:** `password_reset`

Run the application and you should be re-directed to the Azure AD B2C UI for authentication without writing a single line of code!

[1]Inspiration from Carl Franklin `https://blazortrain.com/`

5.1.3 Serverless architectures

The standard web app architecture relies on a server for processing requests from the user. The issues with this approach is that the server needs to be up and running constantly and that more servers are needed when the app scales.

In serverless web app architectures the user requests data from a storage location which simply returns the files needed for interactive apps to run on the client side. Prior to WebAssembly this was only possible using JavaScript.

The problem with serverless web apps is that secrets cannot be stored in the client. This can be solved using Azure Functions which enables APIs for storing data (e.g. databases) without the need of an explicit server.

By using the serverless approach the need for a server to be up and running is eliminated along with the problem of scaling. The actual processing is instead distributed among the clients.

You can also put a Azure CDN in front of the storage account. This will help with caching and let you use a custom domain with a free certificate if you own the domain.

5.2 Mobile & Desktop

Blazor WebAssembly applications can run as Mobile or Desktop Applications either as a Progressive Web Application (PWA) or by using frameworks like Electron.

5.2.1 Progressive Web Applications

A PWA is typically a Single Page Application (SPA) that uses modern browser APIs and capabilities to behave like a desktop app.

As Blazor WebAssembly is a standards-based client-side web app platform it can use any browser API including PWA APIs required for capabilities like:

- Work offline and load instantly.
- Run in its own app window.
- Launch from operating system start menu, dock or home screen.
- Receive push notifications from a backend server even while the app is not being used.
- Automatically update in the background.

Apps with these capabilities are described "Progressive" because the user might first discover and use the app within their web browser and then install it in their OS and enable push notifications.

To create a Progressive Blazor Web Application, create it using Visual Studio or run the following command in the console:

```
dotnet new blazorwasm -o BlazorPWA --pwa
```

When the application is launched, users have the option to install the app into their OS start menu, dock or home screen.

Once installed, the app appears in its own window without an address bar. This window can be customized (title, color scheme, icon etc.) using the *manifest.json* file in the *wwwroot* directory.

Tip

Use the framework ElectronNET to build cross platform desktop apps for accessing the file system and more. [5]

5.3 Blazor Libraries

Blazor already has a large community of 3rd party libraries for components, theming, testing and more. Following is a compilation of the most prominent libraries:

- **Blazored:** Libraries and Components designed for the Blazor Framework. [6]
- **bUnit:** A testing library for Blazor Components. [7]
- **Radzen:** Desktop application for rapid web app development that hides the complexity of code behind its user-friendly interface. [8]
- **MatBlazor:** Material Design components for Blazor. [9]
- **Blazorize:** Blazorise is a component library built on top of Blazor with support for CSS frameworks like Bootstrap, Bulma, AntDesign and Material. [10]
- **Telerik:** Software tools for web, mobile, desktop application development, tools and subscription services for cross-platform application development. [11]
- **Syncfusion:** UI Component Suite for Building Powerful Web, Desktop, and Mobile Apps. [12]
- **DevExpress:** Blazor UI Component Library with over 30 native Blazor components (including a DataGrid, Pivot Grid, Scheduler, Chart, Data Editors, and Reporting). [13]

5.4 What's Next?

While WebAssembly was never intended to replace JavaScript it has paved the way forward for the mess web development has become today.

WebAssembly still lacks direct access to the DOM and other browser APIs but will surely mature as we head into the future. It is efficient, compact and modern making it the perfect candidate to replace JavaScript.

Try browsing the web today with JavaScript turned off and notice how JavaScript has essentially ”become” the browser. As developers we need to start thinking about how we want the web to operate in the future.

Microsoft is currently working on ways of compiling C# code directly to WebAssembly in the build phase something known as ahead of time compilation (AoT) which is a step in the right direction. This is scheduled for the release of .NET 6 in late 2021.

Blazor which now is part of the .NET ecosystem offers an intermediate step enabling C# to finally be used wholly throughout organizations leading to easier onboarding and training of employees.

With the latest addition to the C# language (9.0) Source Generators were added which can be used to inspect and produce additional files compiled together with the rest of your code during compilation phase.

In the future Blazor will potentially be able to render to other formats than HTML as the renderer is extensible and can be replaced. We could for example render to native controls. Today we can use ElectronNET to build Blazor applications for Windows, Mac or Linux.

It is clear that Microsoft since Typescript has hinted for a new and better way of doing web development and we are starting to see the fruits of their labour with Blazor being the shiny beacon of light in the forest of modern web development.

Bibliography

[1] Microsoft. *Blazor: Build client web apps with C#*
`https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor`

[2] Microsoft. *ASP.NET Core Blazor advanced scenarios*
`https://docs.microsoft.com/en-us/aspnet/core/blazor/advance`

[3] Microsoft. *Host and deploy Blazor Server*
`https://docs.microsoft.com/en-us/aspnet/core/blazor/host-an`

[4] Peter Morris. *Render trees*
`https://blazor-university.com/components/render-trees/`

[5] ElectronNET *Electron.NET*
`https://github.com/ElectronNET/Electron.NET`

[6] Chris Sainty. *Blazored*
`https://github.com/Blazored`

[7] Egil Hansen. *bUnit*
`https://github.com/egil/bUnit`

[8] Radzen Ltd. *radzen*
`https://www.radzen.com/`

[9] Vladimir Samoilenko. *MatBlazor*
`https://www.matblazor.com/`

[10] Mladen Macanovic. *Blazorise*
`https://blazorise.com/`

[11] Progress Software Corporation. *Telerik*
`https://www.telerik.com/`

[12] Syncfusion Inc. *Syncfusion*
`https://www.syncfusion.com/`

[13] Developer Express Inc. *DevExpress*
`https://www.devexpress.com/blazor/`

[14] SSW *SSW TV*
`https://tv.ssw.com/`

[15] Free Code Camp *freeCodeCamp*
`https://www.freecodecamp.org/`