IMY320 Individual Assignment Written - WebAssembly

Name: Nicolaas Johan

Surname: Jansen van Rensburg

Student Number: 22590732

Repository Link: GitHub

Introduction

For students studying BIS Multimedia at the University of Pretoria, the 2nd-year first semester already covers a broad range of frontend and multimedia technologies: Vue, React, Angular, HTML, CSS, JavaScript, TypeScript, plus design patterns, search algorithms, and tools like Blender. These give strong grounding in how modern web apps are built, especially in JavaScript and framework-based approaches. However, despite JavaScript's ubiquity, many developers find it frustrating: its weak typing, odd scoping rules, asynchronous callback hell, inconsistent behavior across browsers, and the need to juggle many different libraries and toolchains can lead to steep learning curves and bugs.

How WebAssembly Works

WebAssembly code is compiled into a binary format (.wasm) that browsers can load and execute directly. The browser includes a WebAssembly runtime that validates and runs this binary inside a secure sandbox. For C#, the process differs from C/C++: Blazor compiles C# into .NET Intermediate Language (IL), which is then executed in the browser by a WebAssembly-based .NET runtime. This is why the framework file blazor.webassembly.js is required—it bootstraps the runtime, loads assemblies, and connects the compiled C# logic to the web page's DOM. In effect, WebAssembly lets non-JavaScript languages execute as first-class citizens inside the browser.

Benefits of WebAssembly

WebAssembly's primary benefit is performance. Since the .wasm binary format is compact and optimized, load times are often faster than equivalent JavaScript bundles, and execution is closer to native speed (DeClute, 2023). This makes it especially attractive for compute-intensive applications like 3D rendering, data visualisation, or multimedia processing—relevant areas for Multimedia students. Another advantage is language choice: developers are no longer confined to JavaScript. Strongly typed languages like C# or Rust can be used in browser-based applications, enabling clearer structure and reducing common bugs. Finally, WebAssembly ensures security by running in a sandbox that enforces browser safety policies (Fioretti, 2021).

Why C# and Blazor WebAssembly

Although C# is not part of the BIS Multimedia 2nd-year curriculum, it is my preferred programming language. Using Blazor WebAssembly allows me to apply that preference while still producing a modern, browser-based application. Unlike emscripten, which is suited to C/C++ workflows, Blazor compiles C# into .NET assemblies and executes them through a WebAssembly runtime in the browser. Blazor also provides a component-based model similar to React, with features such as routing, pages, shared layouts, and reusable components. This means I can take advantage of the convenience and structure of frameworks typically tied to JavaScript, while avoiding its well-known shortcomings.

Conclusion

After completing this assignment, I found myself preferring Razor (Blazor's component system) over traditional JavaScript frameworks. Razor's syntax, modular structure, and integration with routes and layouts make development more straightforward and enjoyable. For BIS Multimedia students, WebAssembly demonstrates that modern web development is not restricted to JavaScript: it is a bridge that allows familiar languages like C# to deliver high-performance, interactive applications directly in the browser.

Tutorial

Emscripten is a toolchain designed to compile C and C++ into WebAssembly. Since this project uses C#, emscripten is not applicable. Instead, Microsoft provides Blazor WebAssembly, which compiles C# to .NET Intermediate Language (IL), then executes it in the browser via a WebAssembly-based runtime. This allows C# developers to target the browser directly without rewriting their code in C or C++.

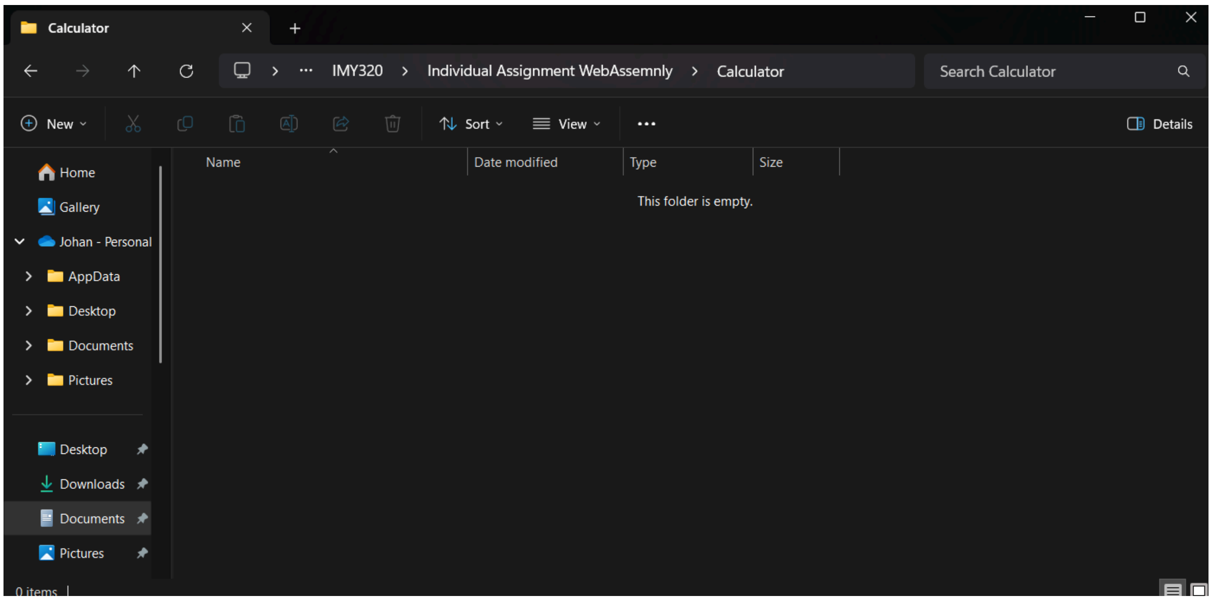
While some languages use emscripten for WebAssembly compilation, C# uses a different pipeline. When you run dotnet new blazorwasm, the build process compiles the C# code into .NET assemblies, which are then executed by the WebAssembly-based .NET runtime. This is why the _framework/blazor.webassembly.js file is included—it bootstraps the runtime and loads the compiled assemblies into the browser.

Setup

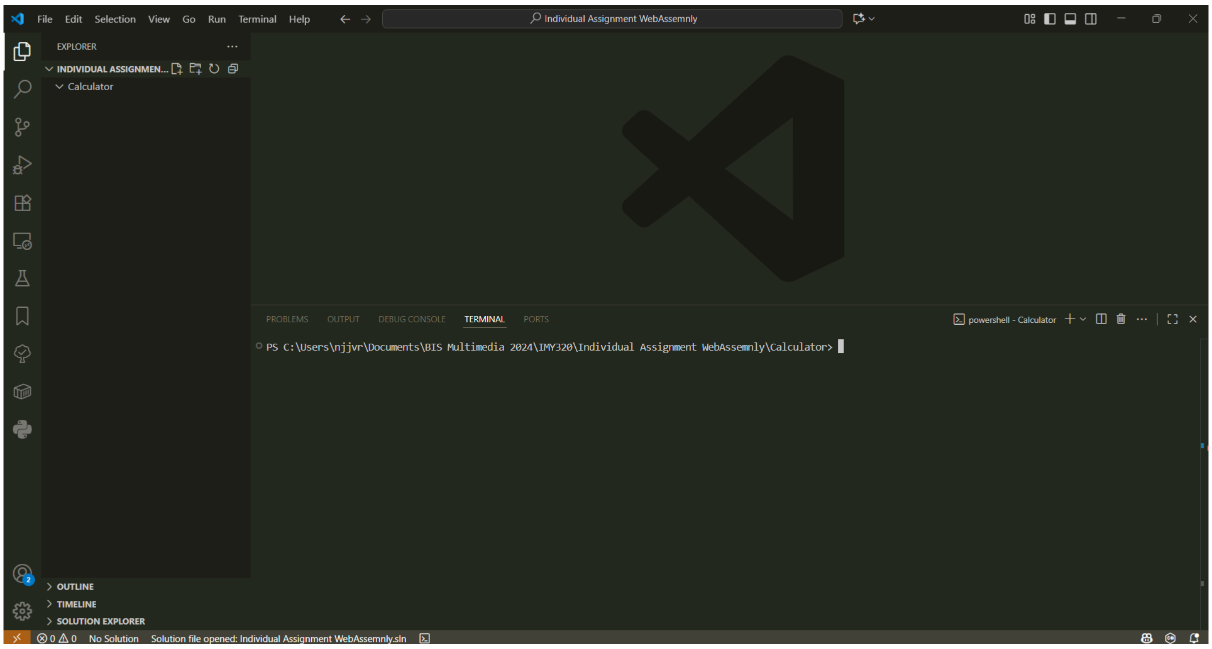
This tutorial requires you to have a basic understanding of HTML and C#. We are going to make a basic Calculator app.

Start

- Set up the project folder. Call the root folder "Calculator"



- Open the folder in a text editor of your choice. This tutorial will be using Visual Studio Code.
- Let’s do the initial setup. Open a PowerShell terminal in your folder. The project should look similar to this



```
PS C:\Users\njjvr\Documents\BIS Multimedia 2024\IMY320\Individual Assignment WebAssemnly\Calculator>
```

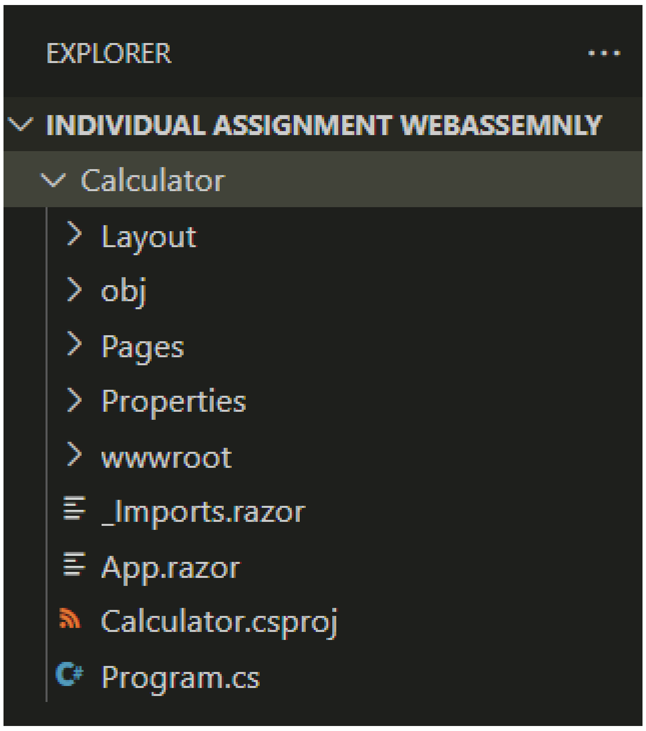
- Install .NET 8 SDK (or latest LTS)
    - Here is the download link
- To check if it is installed correctly, type the following in your terminal

```
dotnet --version
```

- If .NET was installed correctly, it should show you the version you installed.
- When .NET 8 SDK is properly installed, type the following in the terminal

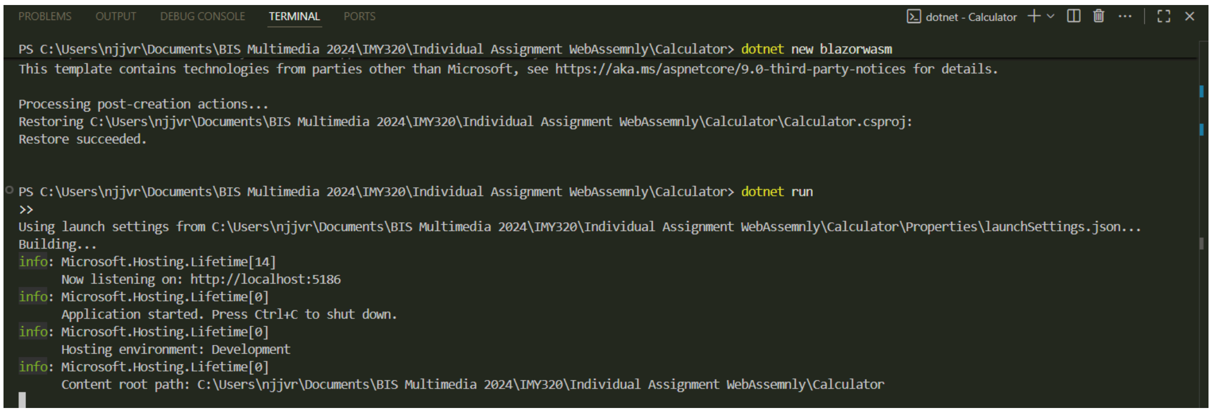
```
dotnet new blazorwasm
```

- Your project structure should look like this



- Run the following in the terminal

```
dotnet run
```

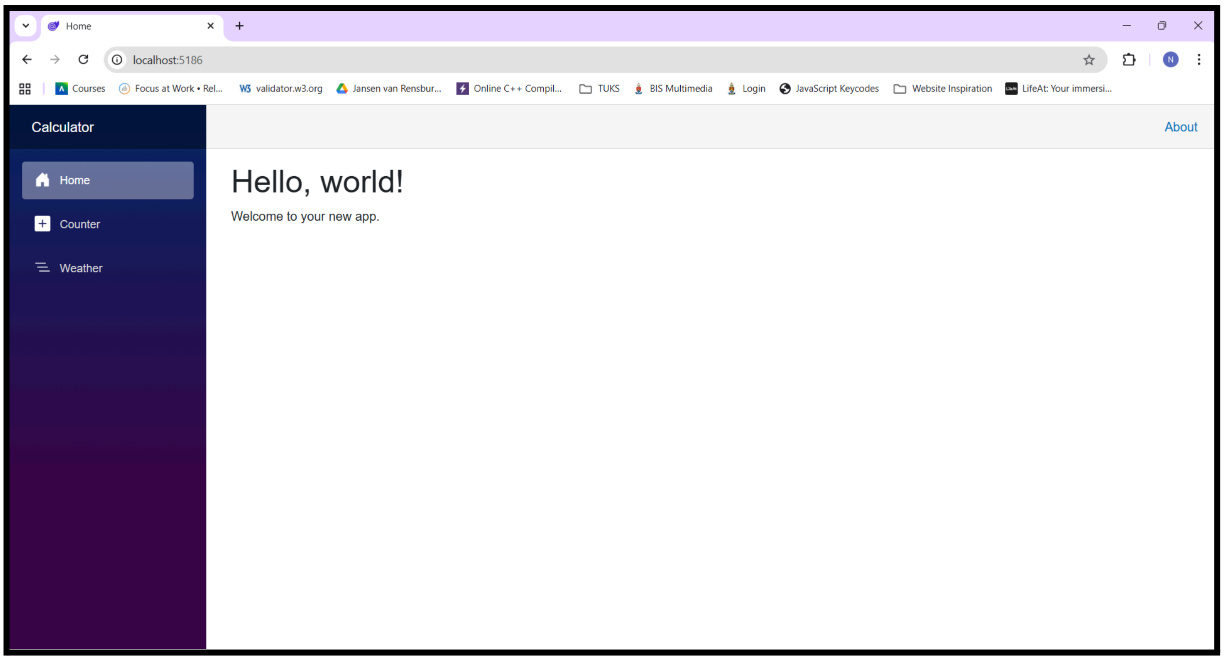


```
PS C:\Users\njjvr\Documents\BIS Multimedia 2024\IMY320\Individual Assignment WebAssemnly\Calculator> dotnet new blazorwasm
This template contains technologies from parties other than Microsoft, see https://aka.ms/aspnetcore/9.0-third-party-notices for details.

Processing post-creation actions...
Restoring C:\Users\njjvr\Documents\BIS Multimedia 2024\IMY320\Individual Assignment WebAssemnly\Calculator\Calculator.csproj:
Restore succeeded.


PS C:\Users\njjvr\Documents\BIS Multimedia 2024\IMY320\Individual Assignment WebAssemnly\Calculator> dotnet run
>>
Using launch settings from C:\Users\njjvr\Documents\BIS Multimedia 2024\IMY320\Individual Assignment WebAssemnly\Calculator\Properties\launchSettings.json...
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5186
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\njjvr\Documents\BIS Multimedia 2024\IMY320\Individual Assignment WebAssemnly\Calculator
```

- In the output of the line we just ran, you can find a port number. In the screenshot above, the following line tells us that the project runs on port 5186

```
Now listening on: http://localhost:5186
```

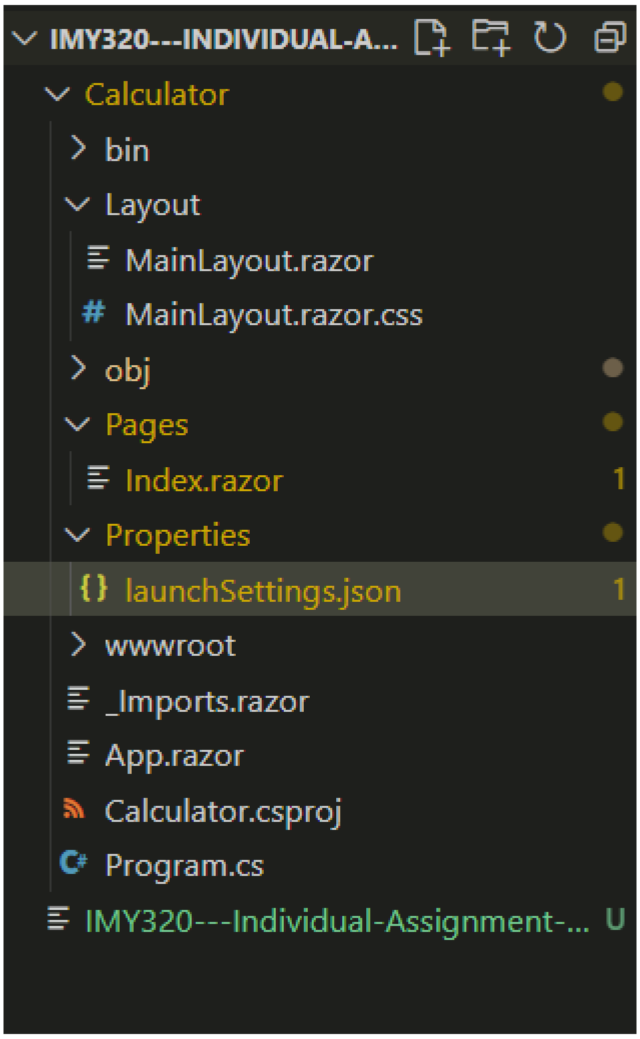
- The project should look like this



Hello, world!

Welcome to your new app.

- If you want to change the ports used for your project, locate the launchSettings.json file in your project and change the port number



- Let’s look at the details of the project before building our Calculator project.
- In the ./wwwroot folder, there is an index.html file. It should look like this

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Calculator</title>
    <base href="/" />
    <link rel="stylesheet" href="lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="css/app.css" />
    <link rel="icon" type="image/png" href="favicon.png" />
    <link href="Calculator.styles.css" rel="stylesheet" />
</head>

<body>
    <div id="app">
        <svg class="loading-progress">
            <circle r="40%" cx="50%" cy="50%" />
            <circle r="40%" cx="50%" cy="50%" />
        </svg>
        <div class="loading-progress-text"></div>
    </div>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="." class="reload">Reload</a>
        <span class="dismiss">✕</span>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

</html>
```

- There are two important lines here

```
<div id="app">
```

```
<script src="_framework/blazor.webassembly.js"></script>
```

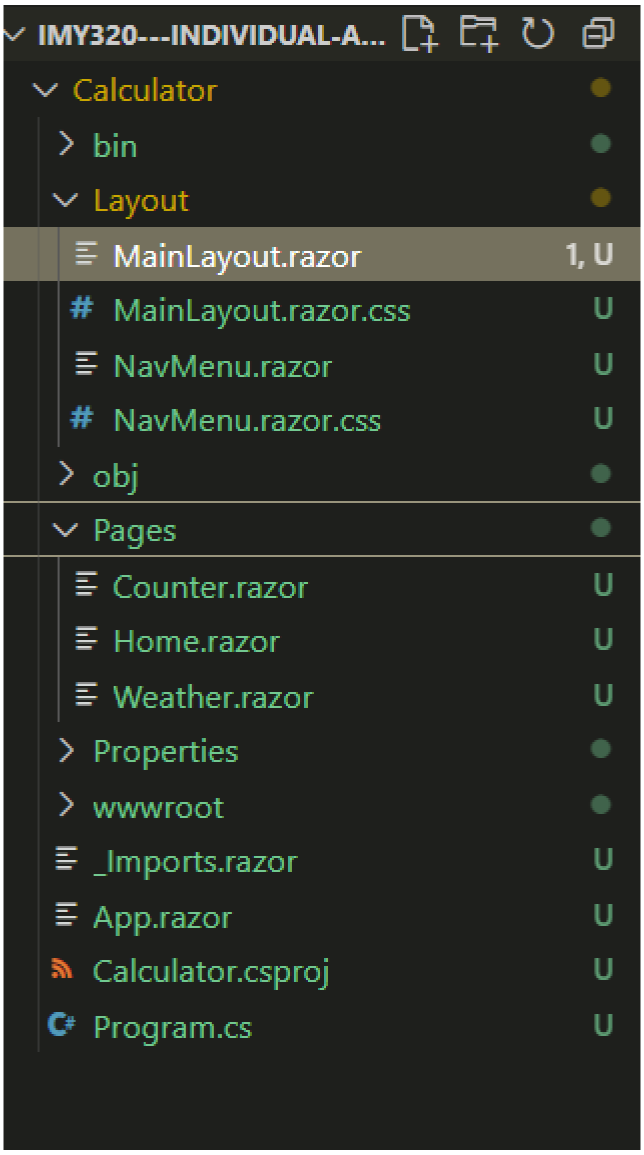
- This is the div that is the placeholder for the app’s UI

- The blazor.webassembly.js file is where the project bootstraps the WASM at runtime and loads the C# assemblies.
- Let's go into the App.razor to look at the start of our UI

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
    <NotFound>
        <PageTitle>Not found</PageTitle>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

- The App uses a Router to match the URL in the browser.
- If it finds a .razor file with a matching @page directive, it renders it.
- By default, all pages use the MainLayout layout.
    - This layout uses content that is auto-generated with the project, so we are going to make some changes:

- Let's recap the project structure



- We are going to delete the following files:
    - NavMenu.razor
    - NavMenu.razor.css
    - Counter.razor
    - Home.razor
    - Weather.razor
- Now that we have a blank project, let's start making changes.

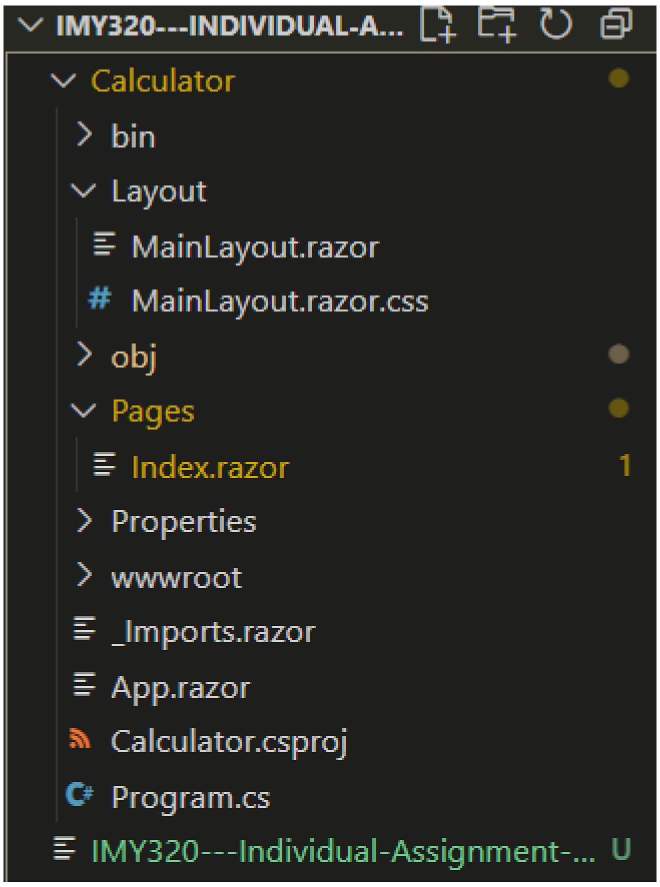
- Open MainLayout.razor

```
@inherits LayoutComponentBase

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <a href="https://learn.microsoft.com/aspnet/" target="_blank">About</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

- Since we deleted the NavMenu files, let's delete the sidebar div. Our final MainLayout should look something like this

```
@inherits LayoutComponentBase
<div class="page">
    <main>
        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

- The @Body value is the placeholder for the current page. By default, it looks for a .razor file with a root route (@page "/")
- Let's create our Calculator page now
    - In the projects folder, create a file Index.razor
    - The project folder should look similar to this


IMY320---INDIVIDUAL-A...

Calculator
- bin
- Layout
  - MainLayout.razor
  - MainLayout.razor.css
- obj
- Pages
  - Index.razor 1
- Properties
- wwwroot
- _Imports.razor
- App.razor
- Calculator.csproj
- Program.cs

IMY320---Individual-Assignment-... U

- Populate the Index.razor file with the following code

```
@page "/"
<h3>WebAssembly Calculator</h3>

<div class="calc-container">
    <input type="text" @bind="Input" class="calc-input" />
    <p>Result: @Result</p>
</div>

<div class="buttons">
    <button @onclick='() => Append("0")'>0</button>
    <button @onclick='() => Append("1")'>1</button>
    <button @onclick='() => Append("2")'>2</button>
    <button @onclick='() => Append("3")'>3</button>
    <button @onclick='() => Append("4")'>4</button>
    <button @onclick='() => Append("5")'>5</button>
    <button @onclick='() => Append("6")'>6</button>
    <button @onclick='() => Append("7")'>7</button>
    <button @onclick='() => Append("8")'>8</button>
    <button @onclick='() => Append("9")'>9</button>

    <button @onclick='() => Append("+")'>+</button>
    <button @onclick='() => Append("/")'>/</button>
    <button @onclick='() => Append("*")'>*</button>
    <button @onclick='() => Append("-")'>-</button>

    <button @onclick='() => Append(".")'>.</button>

    <button @onclick='Clear'>Clear</button>
    <button @onclick='Calculate'>=</button>
</div>


@code {
    private string Input = "";
    private string Result = "";

    void Append(string value)
    {
        Input += value;
    }

    void Clear()
    {
        Input = "";
        Result = "";
    }

    void Calculate()
    {
        try
        {
            var result = new System.Data.DataTable().Compute(Input, "");
            Result = result.ToString();
        }
        catch (Exception ex)
        {
            Result = "Error: " + ex.Message;
        }
    }
}
```

- Let’s go over the code

```
@page "/"
```

- This line tells us that Index.razor contains the content for the default route, or main page of our app.

```
<h3>WebAssembly Calculator</h3>

<div class="calc-container">
    <input type="text" @bind="Input" class="calc-input" />
    <p>Result: @Result</p>
</div>

<div class="buttons">
    <button @onclick='() => Append("0")'>0</button>
    <button @onclick='() => Append("1")'>1</button>
    <button @onclick='() => Append("2")'>2</button>
    <button @onclick='() => Append("3")'>3</button>
    <button @onclick='() => Append("4")'>4</button>
    <button @onclick='() => Append("5")'>5</button>
    <button @onclick='() => Append("6")'>6</button>
    <button @onclick='() => Append("7")'>7</button>
    <button @onclick='() => Append("8")'>8</button>
    <button @onclick='() => Append("9")'>9</button>

    <button @onclick='() => Append("+")'>+</button>
    <button @onclick='() => Append("/")'>/</button>
    <button @onclick='() => Append("*")'>*</button>
    <button @onclick='() => Append("-")'>-</button>

    <button @onclick='() => Append(".")'>.</button>

    <button @onclick='Clear'>Clear</button>
    <button @onclick='Calculate'>=</button>
</div>
```

- This section is the HTML of our Calculator page.
- The @ in the HTML code tells Blazor to switch from HTML to C#
    - @bind ties the value of the text input to the Input variable in the C# code section
    - @Result represents the value of the Result variable in the C# code section
    - @onclick is Blazor’s method of handling onclick events in the HTML
        - All the onclick events call an arrow function that calls the

Append function in the C# code section
  - The 'Clear' and 'Calculate' buttons call the Clear and Calculate functions in the C# code section.

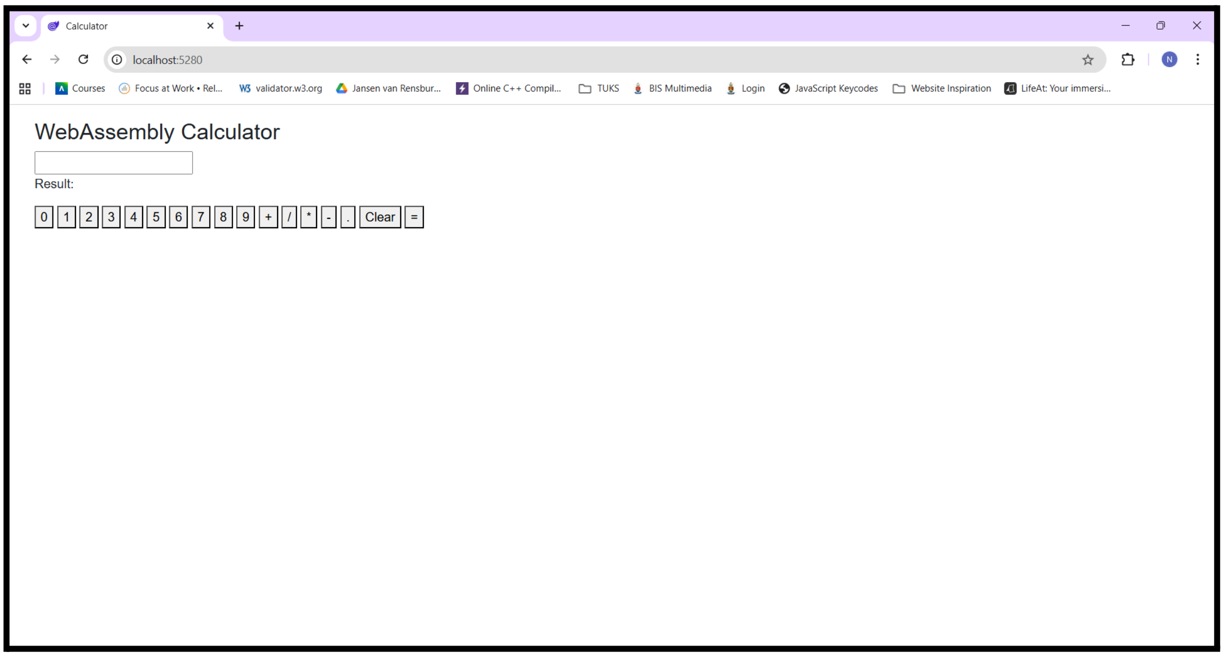
```
@code {
    private string Input = "";
    private string Result = "";

    void Append(string value)
    {
        Input += value;
    }

    void Clear()
    {
        Input = "";
        Result = "";
    }

    void Calculate()
    {
        try
        {
            var result = new System.Data.DataTable().Compute(Input, "");
            Result = result.ToString();
        }
        catch (Exception ex)
        {
            Result = "Error: " + ex.Message;
        }
    }
}
```

- This is the C# code section of the file
  - The @code is where the C# logic lives in this file
  - This section contains the string Input and Result variables, and the function for a basic Calculator app.

- Our Calculator app should now be complete! Let's go back to the terminal
- The project should still be running. Type Ctrl + C to stop the project
- Type the following to rebuild the project

```
dotnet run
```

- The project should look like this in the browser. The port number will depend on the port number shown in your terminal. You should be able to type a calculation into the input box, or use the buttons we created in the HTML to do calculations.



WebAssembly Calculator

Result:

- This concludes the Calculator tutorial using C# and WebAssembly

References

- Webassembly.org. (2025). *WebAssembly*. [online] Available at: https://webassembly.org/ [Accessed 21 Sep. 2025].
- Mozilla.org. (2025). *WebAssembly | MDN*. [online] Available at: https://developer.mozilla.org/en-US/docs/WebAssembly [Accessed 21 Sep. 2025].
- Fioretti, M. (2021) An Introduction to WebAssembly. The Linux Foundation. Available at: https://training.linuxfoundation.org/blog/an-introduction-to-webassembly/ (Accessed: 21 September 2025).
- DeClute, D. (2023). *Why WebAssembly? Top 11 Wasm benefits*. [online] TheServerSide.com. Available at: https://www.theserverside.com/tip/Why-WebAssembly-Top-Wasm-benefits [Accessed 21 Sep. 2025].