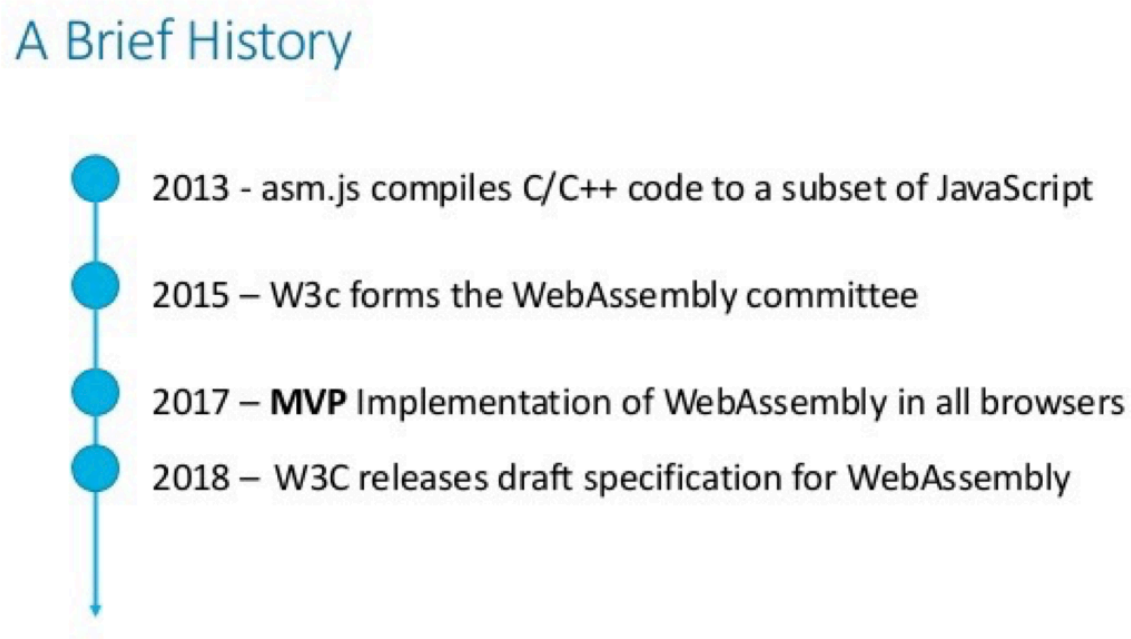
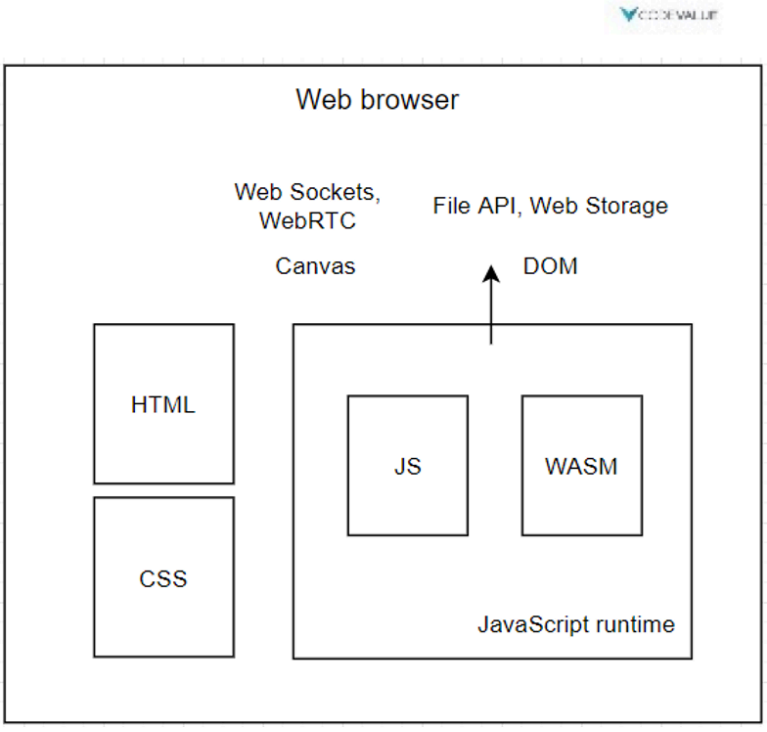
What is WebAssembly?

WebAssembly is a new type of code that can be run in modern web browsers and provides new features and major gains in performance. It is not primarily intended to be written by hand, rather it is designed to be an effective compilation target for source languages like C, C++, Rust, etc.

This has huge implications for the web platform — it provides a way to run code written in multiple languages on the web at near-native speed, with client apps running on the web that previously couldn't have done so.





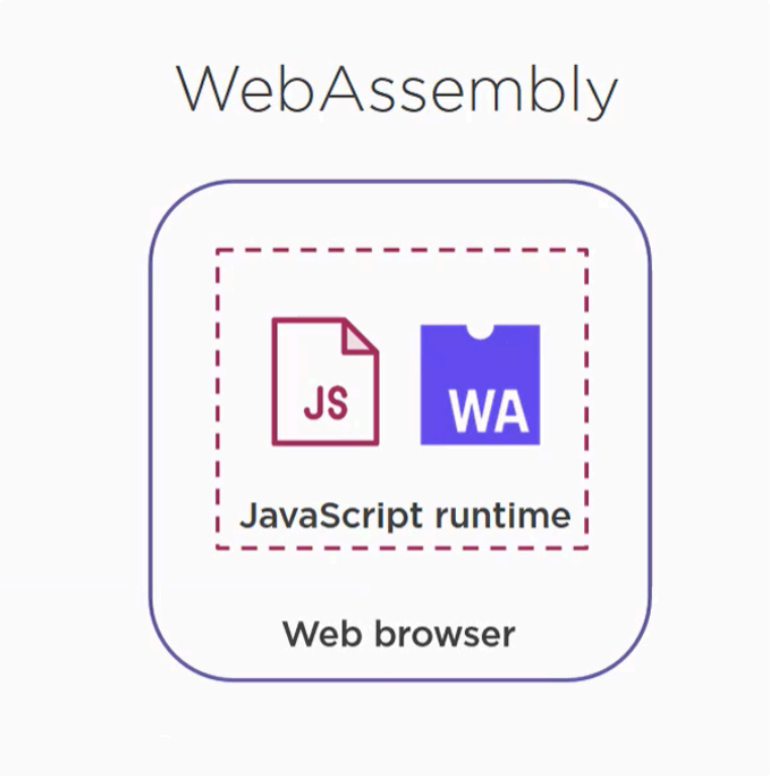
WASM

WebAssembly is a binary instruction format for a stack-based virtual machine.

WASM is designed as a portable target for compilation of high-level languages like C/C++/Rust enabling deployment on the web for client/server applications.

```
function ShowDate() {
   document.getElementById('demo')
           .innerHTML = Date();
}
```

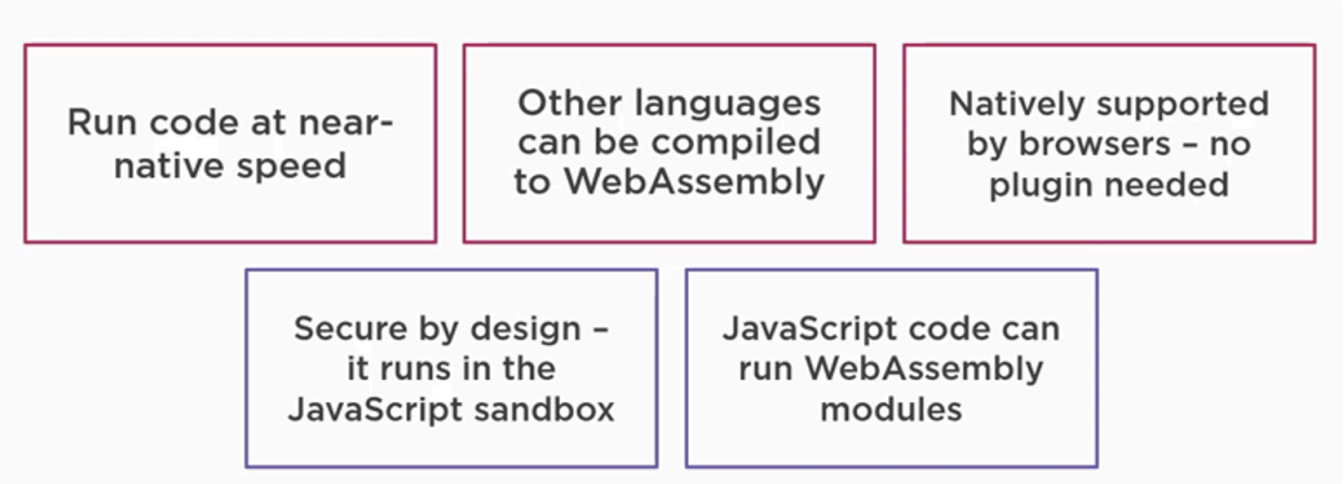
```
0x00000000 0061736D0100000001 .asm..........
0x00000010 7F0302010007070103 .........add....
0x00000020 010700200020016A0B ... . .j....name
0x00000030 010601000361646402 .....add.......l
0x00000040 68730103726873     hs..rhs
```



WebAssembly goals

WebAssembly is being created as an open standard inside the W3C WebAssembly Community Group with the following goals:

- Be fast, efficient, and portable — WebAssembly code can be executed at near-native speed across different platforms by taking advantage of common hardware capabilities.
- Be readable and debuggable — WebAssembly is a low-level assembly language, but it does have a human-readable text format (the specification for which is still being finalized) that allows code to be written, viewed, and debugged by hand.
- Keep secure — WebAssembly is specified to be run in a safe, sandboxed execution environment. Like other web code, it will enforce the browser's same-origin and permissions policies.
- Don't break the web — WebAssembly is designed so that it plays nicely with other web technologies and maintains backwards compatibility.

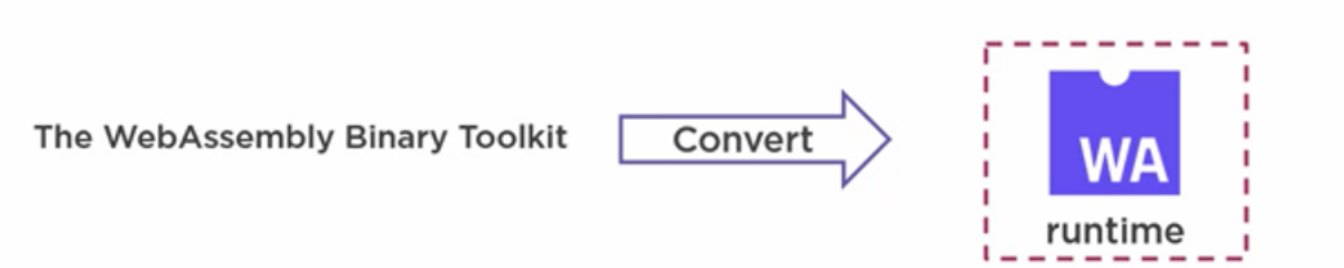


WebAssembly elements

WebAssembly Compatability with Web Platform. The web platform is thought of as having two parts:

- A virtual machine (VM) that runs the web applications code, e.g. the JS code that powers your apps.
- A set of Web APIs that a web application can call to regulate web browser/device functionality and make things happen (CSSOM, DOM, IndexedDB, WebGL, Web Audio API, etc.).

Example - writing WebAssembly yourself




```
(module
  (func $add (param $lhs i32)
  (param $rhs i32) (result i32)
    get_local $lhs
    get_local $rhs
    i32.add)
  (export "add" (func $add))
)
```

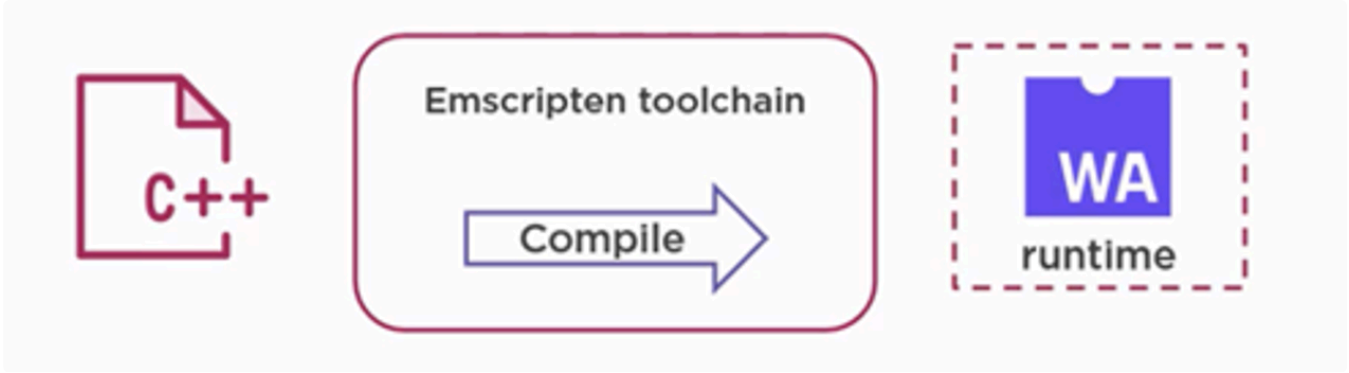
Main.wat

```
0x00000000 0061736D0100000001 .asm..........
0x00000010 7F0302010007070103 .........add....
0x00000020 010700200020016A0B ... . .j....name
0x00000030 010601000361646402 .....add.......l
0x00000040 68730103726873     hs..rhs
```

Main.wasm

https://webassembly-studio.kamenokosoft.com/

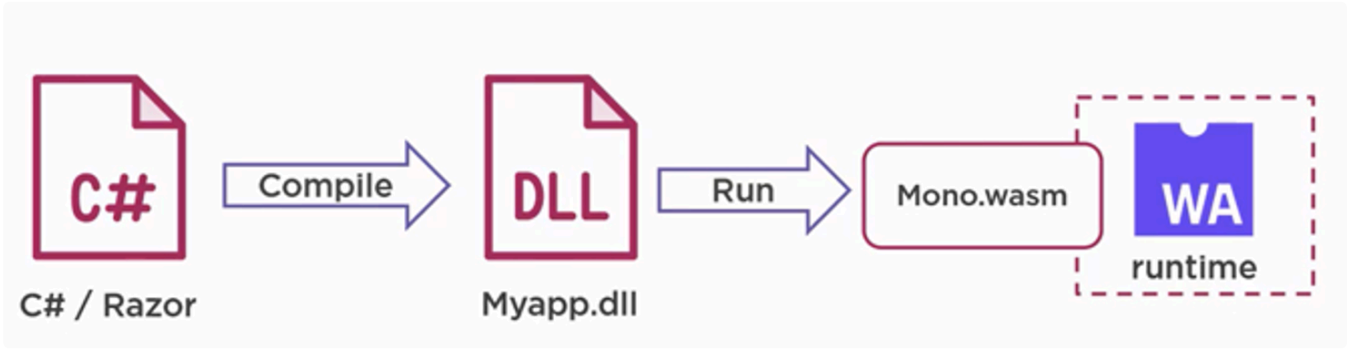
WebAssembly – C++



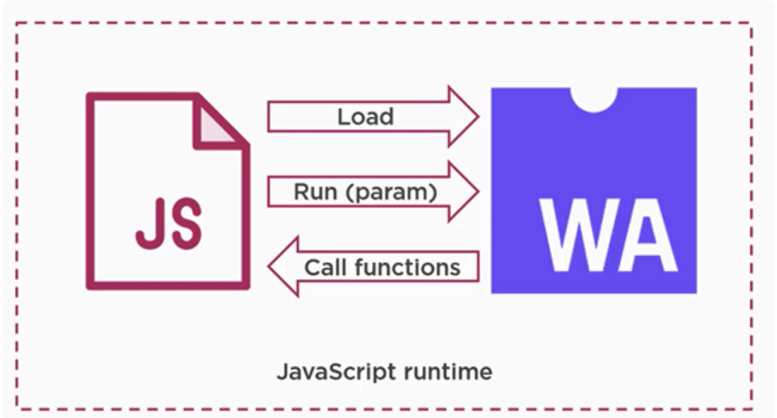

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!";
    return 0;
}
```

```
0x00000000 0061736D0100000001 .asm.....
0x00000010 7F0302010007070103 .....add....
0x00000020 010700200020016A0B ... . .j....name
0x00000030 010601000361646402 .....add.....l
0x00000040 68730103726873     hs..rhs
```

WebAssembly – CS



WebAssembly and JS



Blazor

Blazor is a web framework developed by Microsoft that allows developers to build interactive web UIs using C# and .NET instead of JavaScript. Blazor leverages the capabilities of .NET to create rich, modern web applications, providing a unified framework for server-side and client-side web development.

Key Features of Blazor:

1. **Single Language Full-Stack Development:** Use C# for both client and server-side code, reducing the need to switch between languages.
2. **Component-Based Architecture:** Blazor applications are composed of reusable web UI components implemented using C# and Razor syntax.

3. **WebAssembly (Wasm) Support:** Blazor WebAssembly runs .NET code directly in the browser using WebAssembly, enabling full client-side interactivity.
4. **Server-Side Blazor:** Provides an alternative hosting model where the client UI interactions are handled over a SignalR connection to a .NET server application.
5. **Integration with .NET Ecosystem:** Seamlessly use existing .NET libraries and tools.

Blazor components

Blazor apps are based on components. A component in Blazor is an element of UI, such as a page, dialog, or data entry form.

- Components are .NET C# classes built into .NET assemblies that:
- Define flexible UI rendering logic.
- Handle user events.
- Can be nested and reused.
- Can be shared and distributed as Razor class libraries or NuGet packages.

Blazor - hosting models

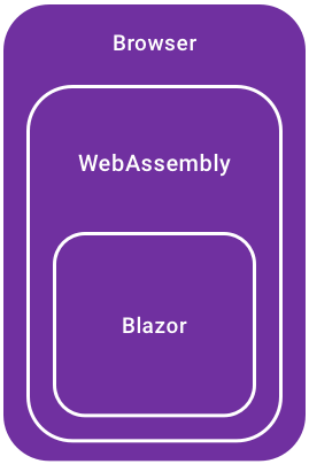
Blazor is an open source single-page web application development framework developed by Microsoft. Unlike other frameworks such as Angular, React and Vue which depend on JavaScript libraries, Blazor allows you to write and run C# code in web browsers via WebAssembly. In this blog, we will discuss about the different hosting models in Blazor.

Blazor currently has three hosting models:

- Blazor WebAssembly (client side).
- Blazor Server (server side).
- ASP.NET Core.

Blazor WebAssembly (Client Side)

According to Microsoft's official documentation , a client-side Blazor WebAssembly (Wasm) application runs in the browser. When a user opens a web page or web application, all the code related to the client-side logic will be downloaded. This means that all the dependencies will also be downloaded. So, the necessary execution time will be relative to run the application. Once we download everything, if we were to disconnect there would be no problem. Since the Blazor WebAssembly hosting model allows us to continue using the application in offline mode and we can synchronize the changes later.



Advantages

The advantages of the Blazor WebAssembly hosting model are as follows:

- WebAssembly allows you to use the client machine to execute the web application within the browser. Once you download the application, you can disconnect the server. The app will continue to work but will not communicate with the server to retrieve new data.
- By having the code run on the client side, we have fast load times since only the changes in the DOM (Document Object Model) are repainted. Therefore, this model considerably reduces the server load.
- This hosting model fully leverages the Customer resources and capabilities.

- You do not need to have an ASP.NET Core web server to host your application. There are serverless deployment scenarios, such as serving the application from a CDN.

Disadvantages

In spite of its several advantages, the Blazor Wasm hosting model does have some disadvantages:

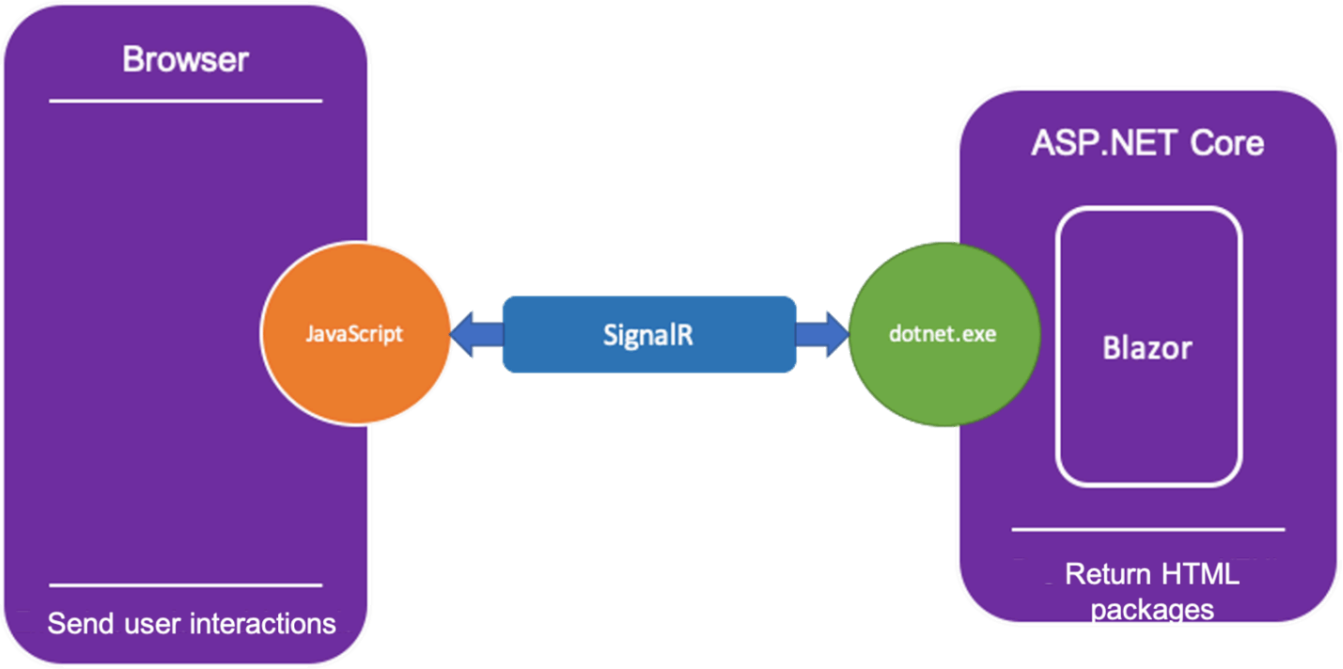
- The application is limited to the capabilities of the browser. This is because the Mono Framework interprets the .NET intermediate language, since the application runs entirely in the client's browser.

Note: Ahead-of-Time (AOT) compilation is planned for a future release.

- This model requires WebAssembly-compatible client hardware and software. In other words, Blazor Wasm only works on the latest browsers.
- The download size is much larger and the applications take longer time to load since the Wasm downloads all the required .NET DLL assemblies.
- Support for the .NET tools and runtime is less well developed. For example, there are limitations to the compatibility and debugging of .NET Standard.

Blazor Server (Server Side)

If we work with the server-side hosting model, the Blazor application will obviously run on the server and every change or event that happens on the client side will be sent to the server through SignalR communication. The server will then process the events or changes and update the client-side UI if necessary. This means that the UI rendering happens on the server side.



Advantages

- Blazor server-side applications load much faster because they pre-render the HTML content.
- You can take full advantage of the capabilities of the server.
- All that the client needs to use the application is a web browser since this model does not have restrictions on browser versions, meaning Blazor Server hosting model works with the oldest browsers.
- This model provides more security since it doesn't send the application code to the client.

Disadvantages

- This model requires an ASP.NET Core server.
- An active connection to the server is essential. The app cannot function without the Internet.
- Because it is constantly sending information both to and from the server, it has higher latency.

Which hosting model should I choose?

The answer is very simple: it depends on the application you are going to develop. One bit of advice I would give you is choose the Blazor server-side hosting model if your application is too complex and SEO is the most important thing. If your application is small and needs the ability to run offline, choose Blazor WebAssembly hosting model.

Blazor presents two clearly differentiated approaches to hosting models:

- Blazor Server: The DOM to be sent to the client from the server is built. It is the most traditional model, whose objective is to replace the .NET Web Forms model. Its main strength is the real-time interaction between the client and server through SignalR.
- Blazor WebAssembly: SPA model based on WebAssembly, i.e. the construction of the DOM is done on the client side. In turn, it allows operations on the server side and calling APIs to request data (with the intention of obtaining sensitive information that you don't want to calculate on the client). To understand this approach, you have to understand what WebAssembly is, which we learned about in the previous article : Blazor WebAssembly: An Overview.

@page directive

Routing tules helps us define which component will appear on the screen according to the URL in which the user is located. There routing rules are configured using the @page directive.

```
@page "/subjects1"

<h3>Subjects1</h3>
<p>This is some text</p>

@code {

}
```

App.razor

- App.razor defines our routing template.
- By default defines the view that will be displayed in case of a missing element.
- Otherwise displays the component based by the route.

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

App.razor- changing the 404 page

```
<Router AppAssembly="@typeof(Program).Assembly" PreferExactMatches="@true">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p style="color: red">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Route params

```
@page "/subjects/{SubjectId:int}/{Name}"

<h3>EditSubject</h3>
<p>
    The id is @SubjectId
</p>
<p>
    @Name
</p>

@code {
    [Parameter] public int SubjectId { get; set; }
    [Parameter] public string Name { get; set; }
}
```

Route constraints

Constraint	Example	Example Matches	Invariant culture matching
`bool`	`{active:bool}`	`true`, `FALSE`	No
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Yes
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Yes
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Yes
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Yes
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	No
`int`	`{id:int}`	`123456789`, `-123456789`	Yes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Yes

Catch-all route

```
@page "/catch-all/{*pageRoute}"

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

NavigationManager

Use NavigationManager to manage URIs and navigation in C# code. NavigationManager provides the event and methods shown in the following table.

Member	Description
Uri	Gets the current absolute URI.
BaseUri	Gets the base URI (with a trailing slash) that can be prepended to relative URI paths to produce an absolute URI. Typically, BaseUri corresponds to the `href` attribute on the document's `<base>` element in `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server).
NavigateTo	Navigates to the specified URI. If `forceLoad` is `true`: • Client-side routing is bypassed. • The browser is forced to load the new page from the server, whether or not the URI is normally handled by the client-side router.
LocationChanged	An event that fires when the navigation location has changed.
ToAbsoluteUri	Converts a relative URI into an absolute URI.
ToBaseRelativePath	Given a base URI (for example, a URI previously returned by BaseUri), converts an absolute URI into a URI relative to the base URI prefix.

```
@page "/subjects/{SubjectId:int}/{Name}"
@inject NavigationManager navigationManager

<h3>EditSubject</h3>
<button @onclick="OnSave">Save</button>
<p>
    @SubjectId
</p>
<p>
    @Name
</p>

@code {
    [Parameter] public int SubjectId { get; set; }
    [Parameter] public string Name { get; set; }

    private void OnSave()
    {
        Console.WriteLine("Save...");
        Console.WriteLine(navigationManager.Uri);
        navigationManager.NavigateTo("/subjects1");
    }
}
```

Working with Razor

Razor is a markup syntax for embedding server-based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a .cshtml file extension. Razor is also found in Razor components files (.razor).

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-5.0

```
@page "/razorsamples"

<h3>Razor samples 1</h3>
<p>@name</p>
<p>2+2=@(2 + 2)</p>

<button @onclick="WriteInLog">Do sth</button>
<button @onclick="@(()=>Console.WriteLine("Do sth 2"))">Do sth 2</button>

@code {
    string name = "Anne";

    void WriteInLog()
    {
        Console.WriteLine("Do sth...");
    }
}
```

Working with Razor - classes

Let's define a separate class.

```
namespace BlazorServerSideExample.Utilities
{
    public static class StringUtilities
    {
        public static string CustomToUpper(string value) => value.ToUpper();
    }
}
```

We can use _Imports.razor file to define the common imports available in all the components.

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using BlazorServerSideExample
@using BlazorServerSideExample.Shared
@using BlazorServerSideExample.Shared.Models
@using BlazorServerSideExample.Utilities
```

```
@page "/razorsamples"

<h3>Razor samples 1</h3>
<p>@StringHelpers.CustomToUpper(name)</p>
<p>2+2=@(2 + 2)</p>
```

Working with Razor – loops/conditionals

Lets create a Models folder with a student class.
Using MarkupString allows us to interpret corretly the HTML (carefull with injections).

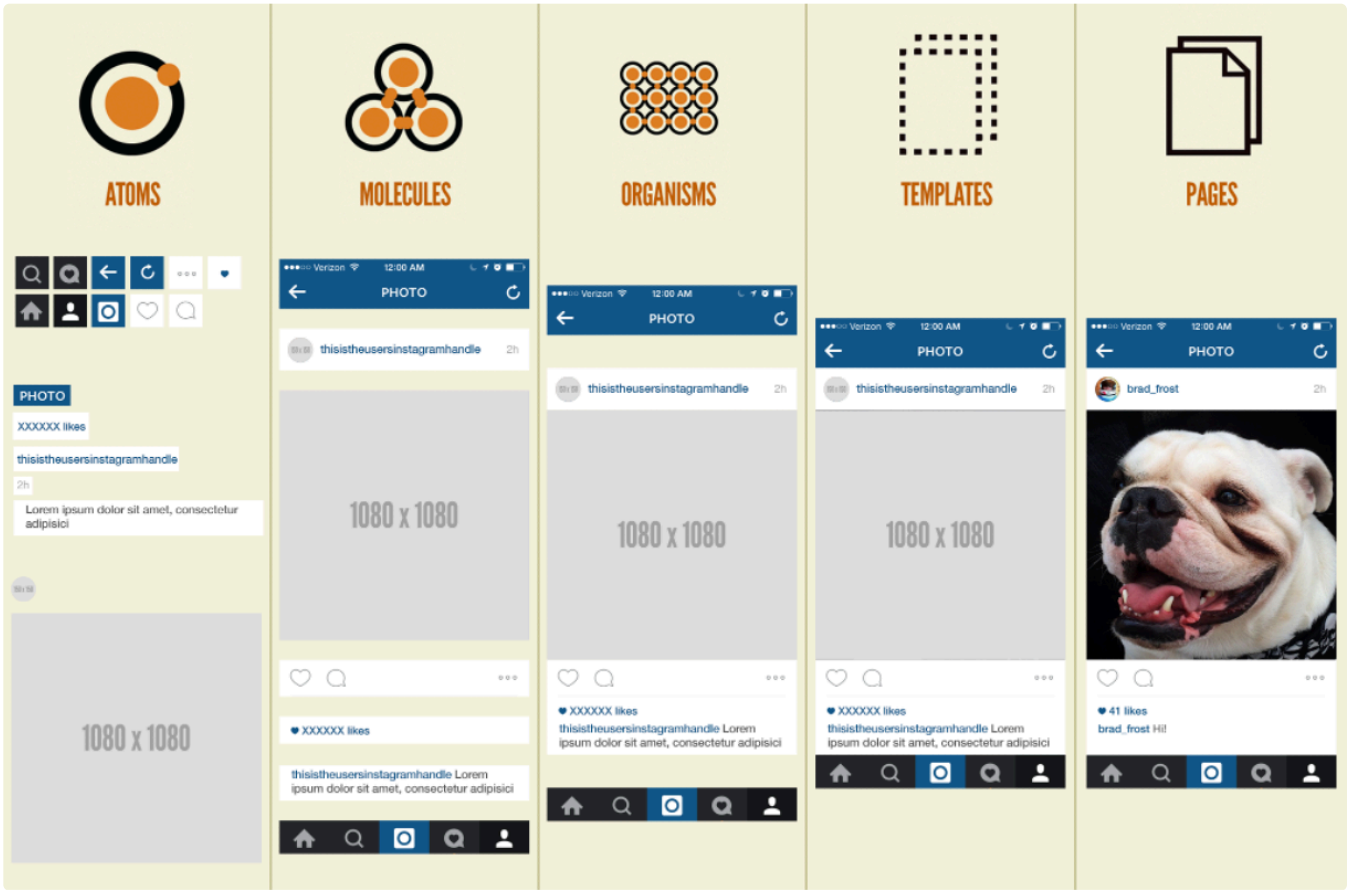
```razor
@page "/studentlist1"

<h3>Student list 1</h3>
@if (students == null)
{
    <p>There are no students</p>
}
else
{
    <ul>
        @foreach (var s in students)
        {
            <li>@(s.FirstName+" "+s.LastName)</li>
            <li>@((MarkupString)(s.FirstName+" "+s.LastName))</li>
        }
    </ul>
}

@code {
    List<Student> students;

    protected override async Task OnInitializedAsync()
    {
        await Task.Delay(4000);
        students = new List<Student>();
        students.Add(new Student
        {
            FirstName = "<b>John</b>",
            LastName = "Doe",
            Description = "Lorem ipsum..."
        });
        students.Add(new Student
        {
            FirstName = "Anne",
            LastName = "<b>Smith</b>",
            Description = "Lorem ipsum..."
        });
    }
}
```

Components

- A component is a reusable piece of user interface, which may contain logic.
- A component is a class.

- We can nest component to create complex interfaces.
- Similar idea to other frontend technologies like React/Angular/Vue.

Atomic design



First component - StudentsList

Let's extract the Students list to a separate razor component – Shared/StudentsList.razor

StudentsList.razor

```
<h3>Students list</h3>
@if (students == null)
{
    <p>There are no students</p>
}
else
{
    <table>
        @for (int i = 0; i < students.Count; i++)
        {
            <tr style="color: @(i%2==0 ? "red": "blue")">
                <td>@students[i].FirstName</td>
                <td>@students[i].LastName</td>
            </tr>
        }
    </table>
}
```

```
@code {
    List<Student> students;

    protected override async Task OnInitializedAsync()
    {
        await Task.Delay(4000);
        students = new List<Student>();
        students.Add(new Student
        {
            FirstName = "John",
            LastName = "Doe",
            Description = "Lorem ipsum..."
        });
        students.Add(new Student
        {
            FirstName = "Anne",
            LastName = "Smith",
            Description = "Lorem ipsum..."
        });
    }
}
```

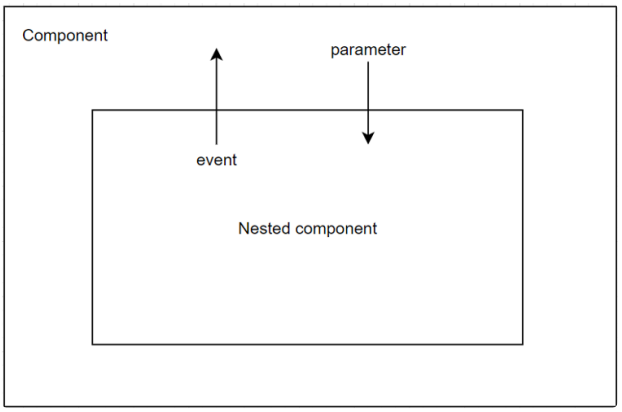
Using component

Then we can create a new page Students in which we can use our previously created StudentsList.razor component.

```
@page "/students"

<h3>Students</h3>
<StudentsList />

@code {

}
```

Parameters

- A component can receive parameters through its parameters.
- Parameters can be data, events, content.



Parameter - StudentList

Let's assume we can pass a list of students from parent component to child component.
To do that we have to add the [Parameter] property to the StudentList component.

```
        }
    </table>
}
```

```
@code {
    [Parameter] public List<Student> Students { get; set; }
```

Now we can pass a list of students from parent component.

```razor
@page "/students2"

<h3>Students2</h3>
<StudentList2 Students="students" />

@code {
    List<Student> students = new List<Student>();

    protected override void OnInitialized()
    {
        students.Add(new Student
        {
            FirstName = "John",
            LastName = "Doe",
            Description = "Lorem ipsum..."
        });
    }
}
```

Arbitrary parameters

Allows us to easily pass multiple parameters at once.

```razor
<h3>Dummy TextBox</h3>

<input type="text" @attributes="AdditionalParameters" />

@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public IDictionary<string, object> AdditionalParameters { get; set; }
}
```

```
@page "/students3"

<h3>Students3</h3>
<DummyTextBox placeholder="Some text..." disabled="true" />

@code {

}
```

Data binding

- Allows us to synchronise the control with the data.
- Razor components provide data binding features with the @bind Razor directive attribute with a field, property, or Razor expression value.

```
@page "/bindingexample"

<h3>BindingExample</h3>
<p>Current count: @currentCount</p>
<button @onclick="IncrementCount">Click me</button>
<input type="number" @bind="currentCount" />

@code {

    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }

}
```

Event callback

- Events allows us to pass data from child component to the parent component.
- Parent passed even handler to the child component. Then the child component calls the event handler and passes the data to the parent controller.

Event callback - child

```
<h3>SpecialButton</h3>

<button @onclick="@(()=>OnDeleteCallback.InvokeAsync("Dane"))">Usuń studenta</button>

@code {
    [Parameter]
    public EventCallback<string> OnDeleteCallback { get; set; }
}
```

Event callback - parent

```
@page "/eventcallback"

<h3>EventCallbackExample</h3>
<p>This is parent controller</p>
<SpecialButton OnDeleteCallback="OnDelete" />

@code {
    private void OnDelete(string data)
    {
        Console.WriteLine(data + " from parent");
    }
}
```

Render fragment

- RenderFragment is used to render components or content at run time in Blazor.
- The RenderFragment class allows you to create the required content or component in a dynamic manner at runtime. In the following code example, the content is created at runtime on OnInitialized.

```razor
<h3>Students list</h3>

@if (Students == null)
{
    @NullTemplate
}
else if (Students.Count == 0)
{
    @EmptyTemplate
}
else
{
    <table>
        @for (int i = 0; i < Students.Count; i++)
        {
            <StudentView Model="@Students[i]" Index="@i" />
            @if (displayButtons)
            {
                <button>Usuń</button>
            }
        }
    </table>
}

@code {
    bool displayButtons = false;

    [Parameter] public List<Student> Students { get; set; }

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

    [Parameter]
    public RenderFragment EmptyTemplate { get; set; }
}
```

```razor
@page "/students4"

<h3>Students4</h3>

<BlazorServerSideExample.Shared.StudentList4 Students="students">
    <NullTemplate>
        <span>This is null template</span>
    </NullTemplate>
    <EmptyTemplate>
        <span>This is empty template</span>
    </EmptyTemplate>
</BlazorServerSideExample.Shared.StudentList4>
<button @onclick="OnDelete">Usuń</button>

@code {
    List<Student> students = new List<Student>();

    protected override void OnInitialized()
    {
        students.Add(new Student
        {
            FirstName = "Anne",
            LastName = "Smith",
            Description = "Lorem ipsum..."
        });
    }
```

Lifecycle of a Component

- OnInitialized and OnInitializedAsync
- OnParameterSet and OnParameterSetAsync
- OnAfterRender and OnAfterRenderAsync
- ShouldRender

```razor
@page "/lifecycleexample"

<h3>LifecycleExample</h3>
<p>
    Lorem ipsum...
</p>

@code {
    protected override void OnInitialized()
    {
        Console.WriteLine("OnInitialized");
    }

    protected override void OnParametersSet()
    {
        Console.WriteLine("OnParametersSet");
    }

    protected override void OnAfterRender(bool firstRender)
    {
        Console.WriteLine("OnAfterRender");
    }

    protected override bool ShouldRender()
    {
        Console.WriteLine("ShouldRender");
        return true;
    }
}
```

Partial classes

```razor
namespace BlazorServerSideExample.Pages
{
    public partial class PartialClassesExample
    {
        public string Name { get; set; } = "Anna";
        public int Age { get; set; } = 22;
    }
}
```

```
@page "/PartialClassesExample"

<h3>PartialClassesExample</h3>
<p>
    @Name
</p>
<p>
    @Age
</p>
```