INTRODUÇÃO AO DESENVOLVIMENTO WEB COM BLAZOR

INTRODUÇÃO

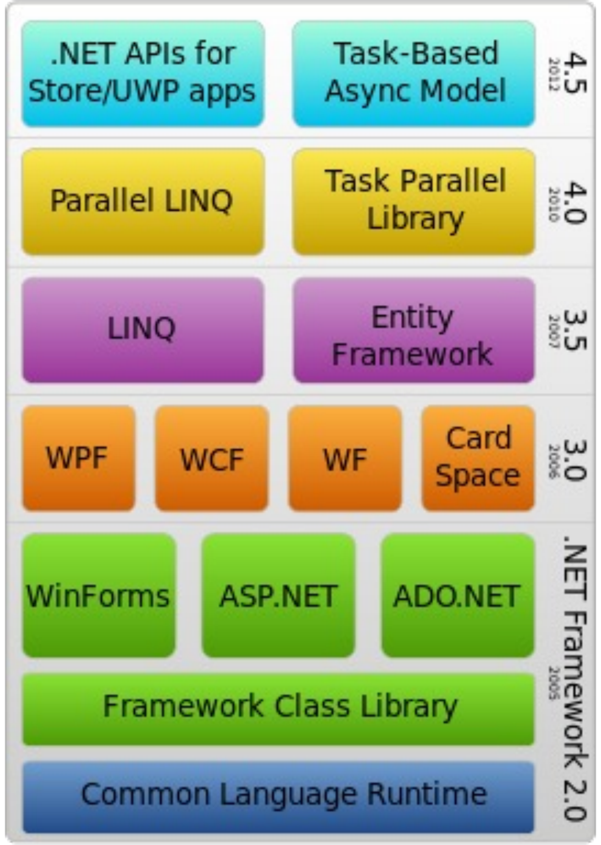
BRUNO OLIVEIRA – BRUNO.OLIVEIRA3@SAPO.PT

AGENDA

- Introdução à framework .NET;
- Introdução a Blazor
- Blazor Server VS WebAssembly
- Ferramentas/tecnologias *front-end*
- Caso de estudo

FRAMEWORK .NET

- .NET Framework é um ambiente de execução que fornece uma biblioteca de classes abrangente;
- Permite aos programadores desenvolver aplicações robustas com código confiável para todas as principais áreas de desenvolvimento;



FRAMEWORK .NET CORE

- A *framework* .NET Core é versão modular da .NET Framework;
- Uma das características principais desta biblioteca passa pela capacidade de instalar os componentes que são necessários para a aplicação;
- Permite que diferentes versões de uma aplicação possam coexistir na mesma máquina sem problemas de compatibilidade da *framework* .NET;
- A ASP.NET Core foi completamente reescrita a partir da *framework* ASP.NET de forma a ser *cross-platform*, *open source* e sem limitações de compatibilidade;

MICROSOFT WEB STACK

- Modelos de desenvolvimento:
  - Web Forms (2002): Permite a construção de websites dinâmicos utilizando uma interface *drag-and-drop* baseado num modelo orientado a eventos a eventos; Permite a criação de aplicações de forma rápida e simples;
  - ASP.NET MVC (2009): A framework MVC (MVC5) da Microsoft aplica o padrão MVC (Model-View-Controller) sobe o ASP.NET, permitindo o desenvolvimento de aplicações web que promovem a reutilização, organização e desempenho do código;
  - ASP.NET Core é diferente das versões anteriores.

BLAZOR

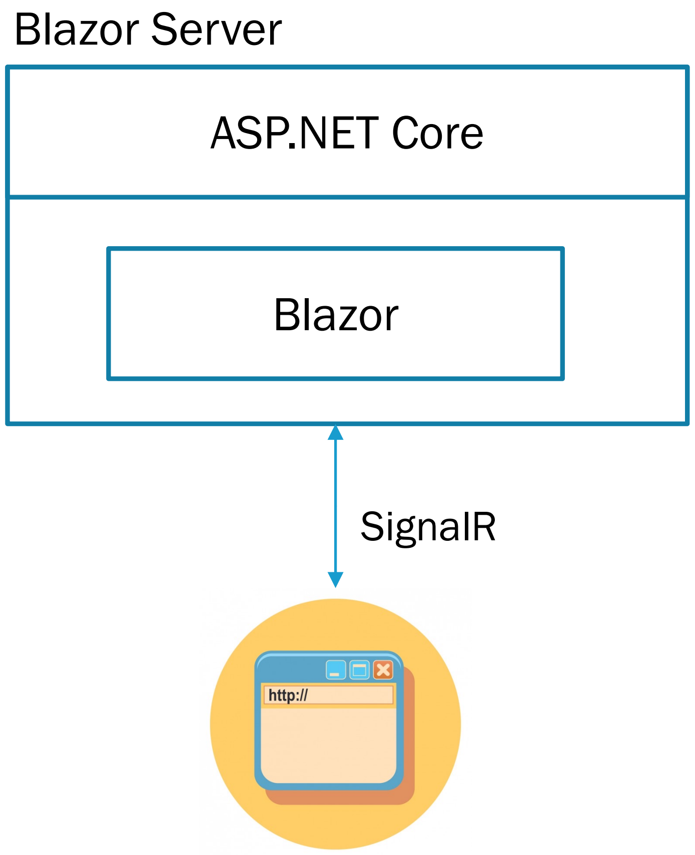
- **Blazor applications** are composed of components that are constructed using C#, HTML-based **Razor syntax**, and CSS.
- Blazor has two different **runtime modes**: **server-side Blazor** and **client-side Blazor**, also known as **Blazor WebAssembly**.
- Blazor brings a modern component-based version of **Razor**, making it possible to break your features down into **tiny components** which are small, focused, and therefore quicker to build.
- Once you have these components you can easily compose them together to **make a bigger feature** or an entire application.

References: https://www.telerik.com/blogs/difference-between-blazor-vs-razor

BLAZOR

- Both modes run in all modern web browsers, including web browsers on mobile phones
- Client-side Blazor is composed of the same code as server-side Blazor; however, it runs entirely in the web browser using a technology known as WebAssembly (https://webassembly.org/)
- The primary difference in Blazor applications that are created in server-side Blazor versus client-side Blazor is that the client-side Blazor applications need to make web calls to access server data, whereas the server-side Blazor applications can omit this step as all their code is executed on the server.

BLAZOR SERVER VS BLAZOR WEBASSEMBLY



Blazor Server

ASP.NET Core

Blazor

SignalR

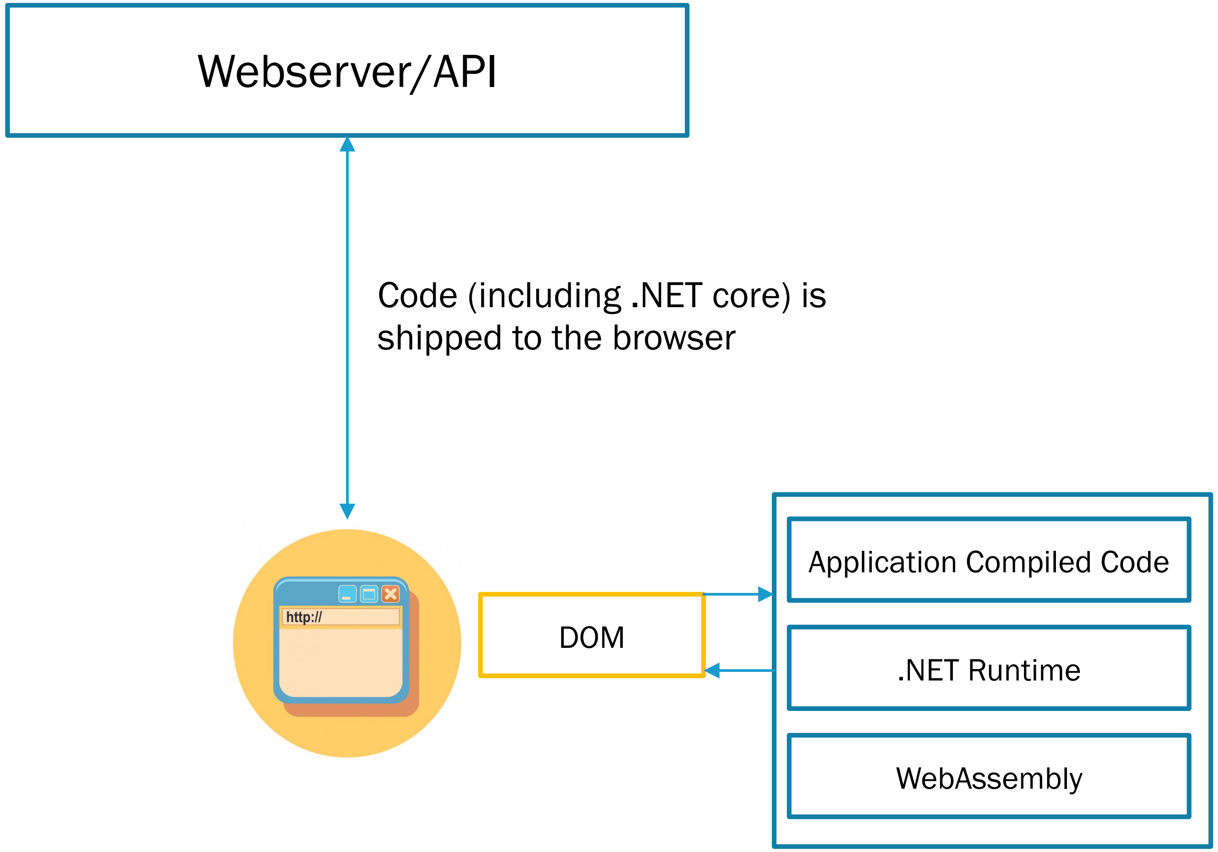
Application runs on the server

SinalR component enables the communication

Easier to develop since integrates all components for web app development

BLAZOR SERVER VS BLAZOR WEBASSEMBLY

Blazor WebAssembly


Webserver/API
Code (including .NET core) is shipped to the browser
http://
DOM
Application Compiled Code
.NET Runtime
WebAssembly

Webserver/API need to be developed separately (it is useful if you already have an existing API)

Runs on the client (displaying data)

Obviously, you cannot connect directly to the database

It is useful for front-end

Supports progressive web app (specially used if you want to ship web app to mobile phones)

BLAZOR - COMPONENTS AND ROUTING

- A component is a chunk of code **consisting of a user interface and** processing logic;
- Blazor features routing, where you can provide navigation to your controls using the **@page** directive followed by a unique route in quotes preceded by a slash;
- A Razor component is contained in a **.razor** file and can be **nested inside** of other components.

BLAZOR - COMPONENTS AND ROUTING - OVERVIEW

- For example, we can create a component named ComponentOne.razor using the following code.

```
<h4 style="background-color:goldenrod">
  This is ComponentOne
</h4>


@code {  }
```

We can alter ComponentExample.razor to contain ComponentOne.razor.

```
@page "/componentexample"
<h3>This is Component Example</h3>

<ComponentOne />

@code {
}
```

BLAZOR – PARAMETERS - OVERVIEW

- Razor components can pass values to other components using **parameters**.
- Component parameters are defined using the **[Parameter]** attribute, which must be declared as **public**.

```
<h4>Parameter Example Component</h4>


<h5 style="color:red">@Title</h5>
@code {
  [Parameter]
  public string Title { get; set; }
}
```

```
@page "/parameterexample"
<h4>Parameter Example</h4>
<ParameterExampleComponent
           Title="Passed from Parent" />
@code {
}
```

BLAZOR - DATA BINDING - OVERVIEW

- Simple, one-way binding in Blazor is achieved by declaring a parameter and referencing it using the @ symbol. An example of this is shown in the following code:

```
<b>BoundValue:</b> @BoundValue
@code {
  private string BoundValue { get; set; }

  protected override void OnInitialized() {
    BoundValue = "Initial Value";
  }
}
```

BLAZOR - DATA BINDING - OVERVIEW

- Two-way, dynamic data binding in Razor components is implemented using the @bind attribute.

```
<input @bind="BoundValue" @bind:event="oninput" />
<p>Display CurrentValue: @BoundValue</p>
@code {
  private string BoundValue { get; set; }
}
```

BLAZOR – EVENT - OVERVIEW

- Raising events in Razor components is straightforward. The following example demonstrates using the **@onclick** event handler to execute the method IncrementCount when the button is clicked

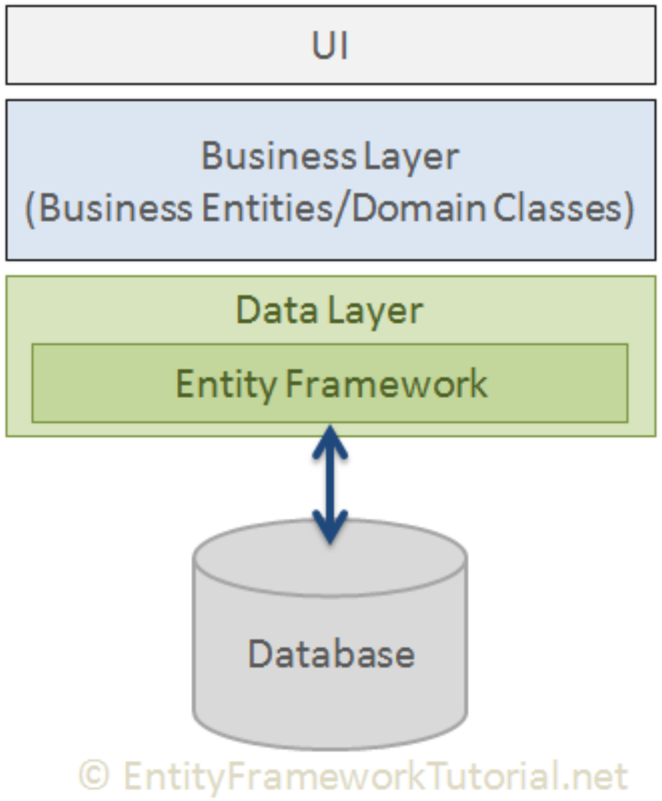
```
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount"> Click me </button>


@code {
  private int currentCount = 0;
  private void IncrementCount() {
    currentCount++;
  }
}
```

DATABASE MANAGEMENT

- Entity Framework: https://docs.microsoft.com/en-us/ef/
- Dapper: https://dapper-tutorial.net/dapper

ENTITY FRAMEWORK

- Entity Framework is an open-source ORM framework for .NET applications supported by Microsoft.
- It enables developers to work with data using objects of domain-specific classes without focusing on the underlying database tables and columns where this data is stored.
- With the Entity Framework, developers can work at a higher level of abstraction when they deal with data and can create and maintain data-oriented applications with less code compared with traditional applications.


UI
Business Layer
(Business Entities/Domain Classes)
Data Layer
Entity Framework
Database
© EntityFrameworkTutorial.net

https://www.entityframeworktutorial.net

http://www.entityframeworktutorial.net/what-is-entityframework.aspx

DAPPER

- Dapper is a simple object mapper for .NET and owns the title of **King of Micro ORM** in terms of speed and is virtually as fast as using a raw ADO.NET data reader.
- An ORM is an Object Relational Mapper, which is responsible for mapping between a database and a programming language.

https://dapper-tutorial.net/dapper

DEMO (STARTUP)

- Using CLI
  - **dotnet new blazorserver -o BlazorApp –no-https net7.0**
  - **cd BlazorApp**
  - **dotnet build**
  - **dotnet watch**
  - Wait for the command to display that it's listening on http://localhost:5000 and then, open a browser and navigate to that address.
  - https://github.com/brunobmo/Blazor_Course

KEY COMPONENTS

BLAZOR PROJECT

_HOST.CSHTML

- The root page of the app implemented as a Razor Page;
- When any page of the app is initially requested, this page is rendered and returned in the response.
- The Host page specifies where the root App component (App.razor) is rendered.

PROGRAM.CS

- The app's entry point that sets up the ASP.NET Core host and contains the app's startup logic, including service registrations and request processing pipeline configuration;
- Specifies the app's dependency injection (DI) services. Services are added by calling AddServerSideBlazor, and the WeatherForecastService is added to the service container for use by the example FetchData component.
- MapFallbackToPage("/_Host") is called to set up the root page of the app (Pages/_Host.cshtml) and enable navigation.

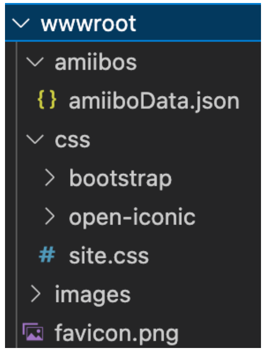
APP.RAZOR

- The root component of the app that sets up client-side routing using the Router component.
- The Router component intercepts browser navigation and renders the page that matches the requested address.

```
<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>
```

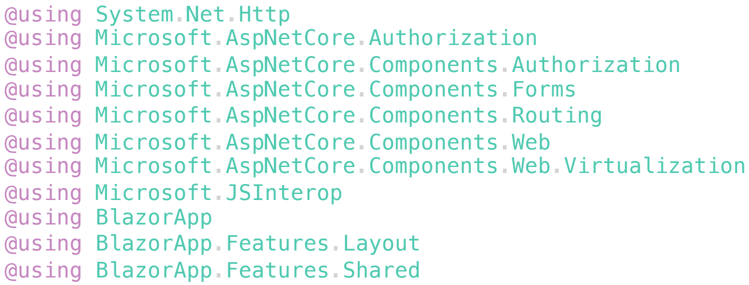
WWWROOT FOLDER AND _IMPORTS.RAZOR

- wwwroot: The Web Root folder for the app containing the app's public static assets.



```
wwwroot
  amiibos
    amiiboData.json
  css
    bootstrap
    open-iconic
    site.css
  images
  favicon.png
```

- _Imports.razor: Includes common Razor directives to include in the app's components (.razor), such as @using directives for namespaces.



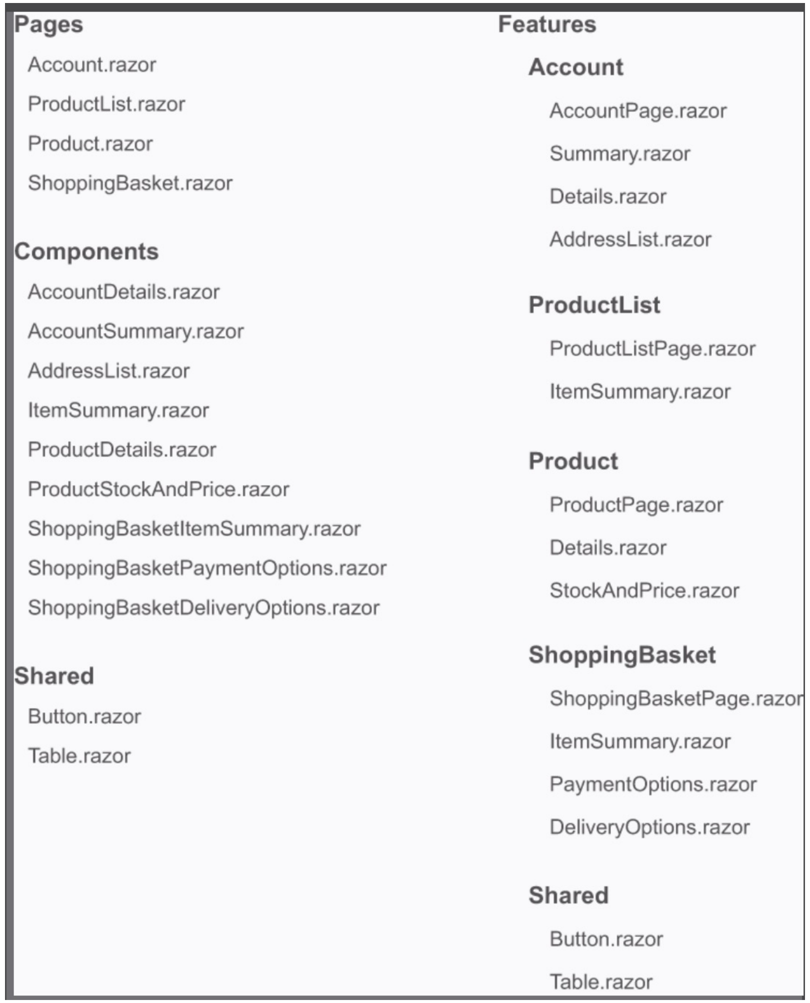
```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using BlazorApp
@using BlazorApp.Features.Layout
@using BlazorApp.Features.Shared
```

CASE STUDY

ORGANIZING FILES USING FEATURE FOLDERS

- By default, a Pages folder is used for routable components, and there's a Shared folder for anything that is used in multiple places;
- This kind of separation doesn't scale well and makes adding or changing functionality much more difficult, as files end up being spread out all over the place;
- When using feature folders, all the files relating to that feature are stored in the same place.
- This has two major benefits:
  - First, when you go to work on a particular feature, all the files you need are in the same place;
  - Second, it scales well. Every time you add a new feature to the app, you just add a new folder and everything goes in there. You can also arrange each feature with subfeatures if they contain a lot of files;

ORGANIZING FILES USING FEATURE FOLDERS

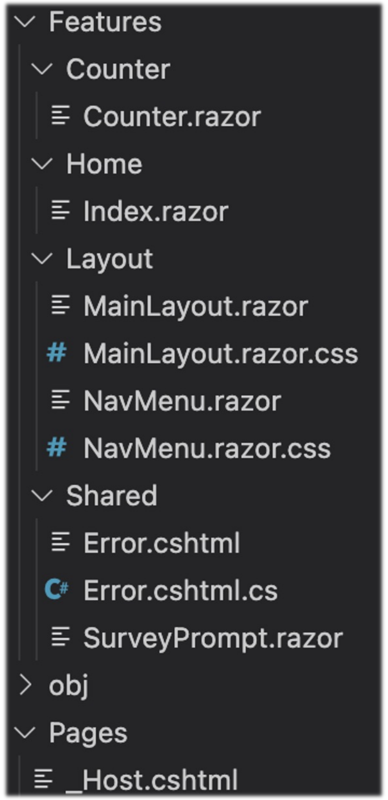


- **Pages**
  - Account.razor
  - ProductList.razor
  - Product.razor
  - ShoppingBasket.razor
- **Components**
  - AccountDetails.razor
  - AccountSummary.razor
  - AddressList.razor
  - ItemSummary.razor
  - ProductDetails.razor
  - ProductStockAndPrice.razor
  - ShoppingBasketItemSummary.razor
  - ShoppingBasketPaymentOptions.razor
  - ShoppingBasketDeliveryOptions.razor
- **Shared**
  - Button.razor
  - Table.razor

- **Features**
  - **Account**
    - AccountPage.razor
    - Summary.razor
    - Details.razor
    - AddressList.razor
  - **ProductList**
    - ProductListPage.razor
    - ItemSummary.razor
  - **Product**
    - ProductPage.razor
    - Details.razor
    - StockAndPrice.razor
  - **ShoppingBasket**
    - ShoppingBasketPage.razor
    - ItemSummary.razor
    - PaymentOptions.razor
    - DeliveryOptions.razor
  - **Shared**
    - Button.razor
    - Table.razor

ORGANIZING FILES USING FEATURE FOLDERS

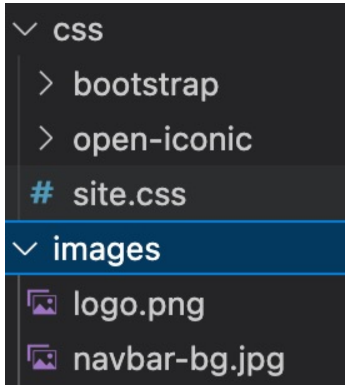
- Considering the template generated for Blazor Projects, we can arrange the files based on features;



```
Features
  Counter
    Counter.razor
  Home
    Index.razor
  Layout
    MainLayout.razor
    MainLayout.razor.css
    NavMenu.razor
    NavMenu.razor.css
  Shared
    Error.cshtml
    Error.cshtml.cs
    SurveyPrompt.razor
obj
Pages
  _Host.cshtml
```

STYLES CONFIGURATION

- We can add custom styles to the site.css file (inside wwwroot/css).
- By adding the styles here, they will affect the whole application.
- These styles will customize the look of some common elements, such as links and buttons, as well as the navbar.
- The images folder can also be created to store images we want to use on the website.



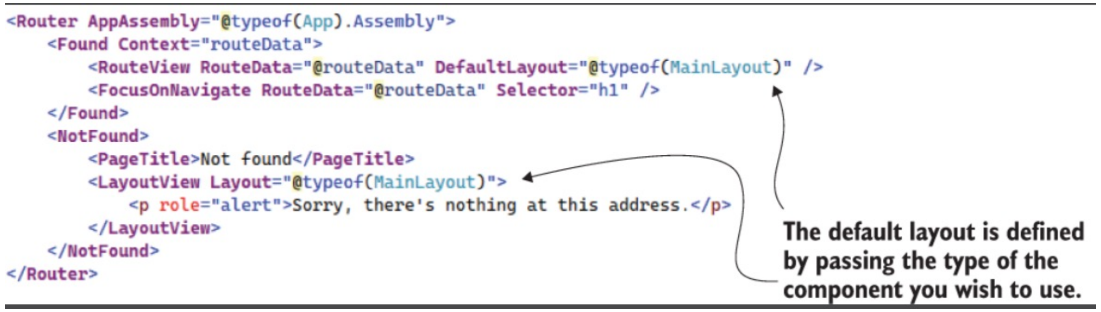
```
css
  bootstrap
  open-iconic
  site.css
images
  logo.png
  navbar-bg.jpg
```

LAYOUT

- Blazor borrows the concept of a layout from other parts of ASP.NET Core.
- Essentially it allows us to define common UIs, which is required by multiple pages.
- Things such as the header, footer, and navigation menu are all examples of things you might put in your layout.
- We also add a reference to a parameter called Body where we want page content to be rendered.
- This comes from a special base class that all layouts in Blazor must inherit from called LayoutComponentBase

LAYOUT

- We are not restricted to a single layout for your whole application; you can have multiple layouts for different parts of your app.
- So, if you wanted a particular layout for the public pages but a different one for the admin pages, you can do that.
- In Blazor, the default layout is defined within the Router component, which can be found in App.razor.

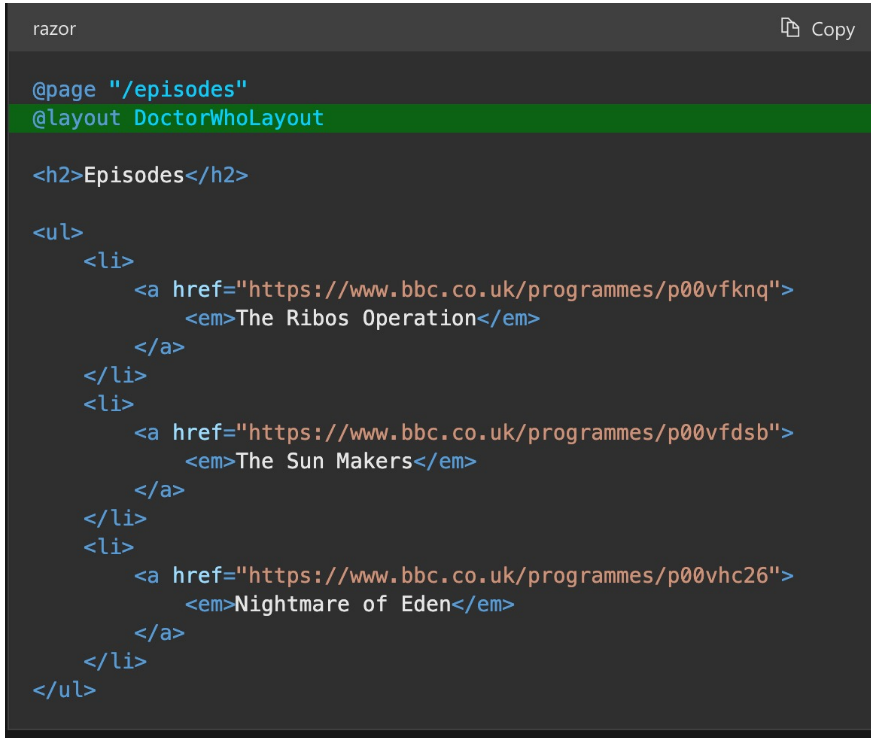


```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
    <NotFound>
        <PageTitle>Not found</PageTitle>
    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

The default layout is defined by passing the type of the component you wish to use.

LAYOUT

- If you want to use a different layout on certain pages, you can specify an alternative by applying the @layout directive.
- This goes at the top of the page, and you pass the name of the component you wish to use.
- For example, if we had an alternative layout called AdminLayout, our layout directive would look like this: @layout AdminLayout



```razor
@page "/episodes"
@layout DoctorWhoLayout

<h2>Episodes</h2>

<ul>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vfknq">
            <em>The Ribos Operation</em>
        </a>
    </li>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vfdsb">
            <em>The Sun Makers</em>
        </a>
    </li>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vhc26">
            <em>Nightmare of Eden</em>
        </a>
    </li>
</ul>
```

https://learn.microsoft.com/en-us/aspnet/core/blazor/components/layouts?view=aspnetcore-7.0

LAYOUT

- Let’s change the default template and add a Header.

```
@inherits LayoutComponentBase

<PageTitle>BlazorApp</PageTitle>

<div class="page">
   <main>
     <Header />

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

MainLayout.razor

```
<nav class="navbar mb-5 shadow">
    <a class="navbar-brand" href="/">
      <img src="/images/logo.png">
    </a>
</nav>
```

Header.razor

LAYOUT

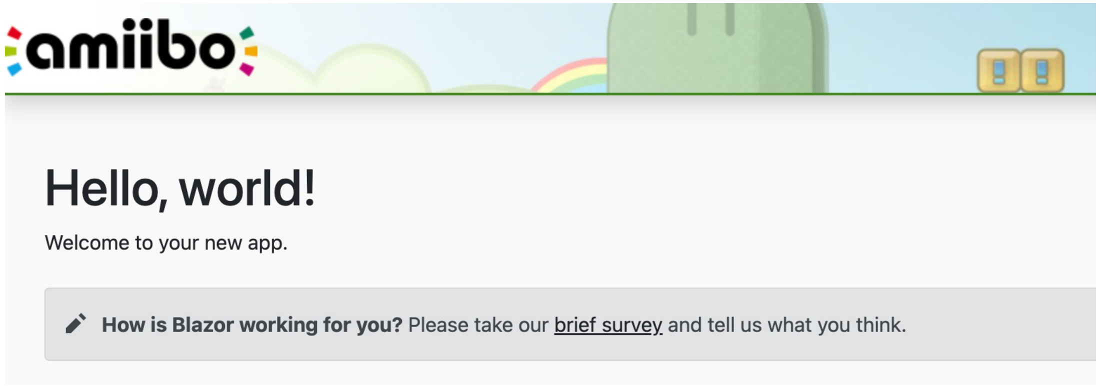
- Vamos agora criar/modificar as páginas do nosso site:
- Vamos observar o ficheiro Index.razor (features -> home)

```
@page "/"

<PageTitle>Index</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

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

When a route template contains only a forward slash (/), it tells the router that this is the root page of the application.

LAYOUT

- The result shows a new website presentation:



Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our brief survey and tell us what you think.

CONTENT LOGIC

- We need a way of representing data to show in the website.
- To do that, we will add a new class called Amiibo to the shared folder inside feature folder.
- It will contain the various data points for a Amiibo;

CONTENT LOGIC

```
namespace BlazorApp.Features.Home;

public class Amiibo
{
  public string AmiiboSeries { get; set; } = "";
  public string Character { get; set; } = "";
  public string GameSeries { get; set; } = "";
  public string Image { get; set; } = "";
  public string Name { get; set; } = "";
  public string Type { get; set; } = "";
}
```

CONTENT LOGIC

- Now that we have a definition for an Amiibo, we’ll define some test data to use.
- Currently, we don’t have a backend.
- To simulate the backend, we’ll define our test data in a JSON file.
- We can use the HttpClient to load the data from the JSON file in the same way we’d load data from the backend.

CONTENT LOGIC

- Put the file inside wwwroot (in this case in the wwwroot/amiibos/amiiboData.json)
- Example:

```json
[
  {
    "amiiboSeries": "Super Smash Bros.",
    "character": "Mario",
    "gameSeries": "Super Mario",
    "image": "https://raw.githubusercontent.com/N3evin/AmiiboAPI/master/images/icon_00000000-
00000002.png",
    "name": "Mario",
    "type": "Figure"
  },
  {
    "amiiboSeries": "Super Mario Bros.",
    "character": "Mario",
    "gameSeries": "Super Mario",
    "image": "https://raw.githubusercontent.com/N3evin/AmiiboAPI/master/images/icon_00000000-
00340102.png",
    "name": "Mario",
    "type": "Figure"
  }
]
```

CONTENT LOGIC

With our test data in place, we'll return to the HomePage component, where we need to load it.

We're going to load the data using the HttpClient, but to use it we need to get an instance of it using dependency injection.

Blazor makes this easy by providing an inject directive: @inject [TYPE] [NAME], where [Type] is the type of the object we want and [Name] is the name we'll use to work with that instance in our component.

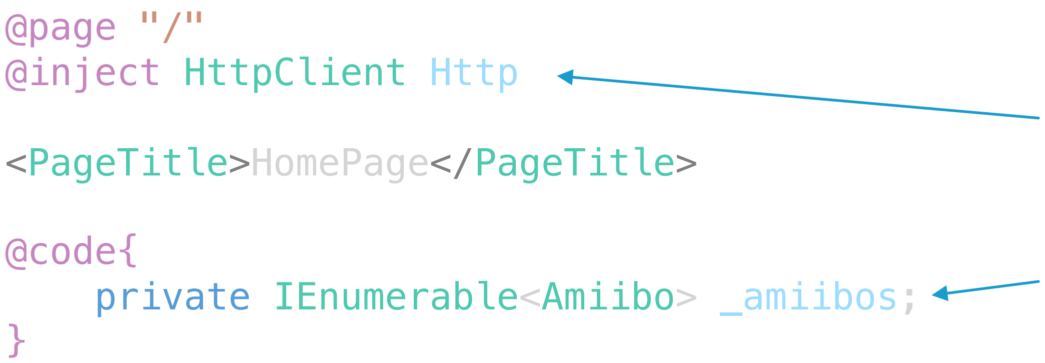
Under the page directive, add @inject HttpClient Http, which will give us an instance of the HttpClient to work with.

CONTENT LOGIC

Before we can use the HttpClient, we need somewhere to store the results returned by the call.

Our JSON test data is an array of trails, and as we're not going to modify what's returned, just listing it out, we can create a private field of type IEnumerable <Amiibo>.

This is done in the @code block of the component as shown in the following listing.



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

<PageTitle>HomePage</PageTitle>

@code{
    private IEnumerable<Amiibo> _amiibos;
}
```

The Inject directive is used to get instances of objects from the dependency injection container

The Private field holds trail data

CONTENT LOGIC

Now that we have somewhere to store our test data, we can make the call to retrieve it.

A great place to do this kind of thing is the OnInitialized life cycle method.

This method is provided by ComponentBase—which all Blazor components inherit from—and it's one of three primary life cycle methods.

The other two are OnParametersSet and OnAfterRender—they all have async versions as well.

CONTENT LOGIC

OnInitialized is run only once in the component’s lifetime, making it perfect for loading initial data like we need to.

To retrieve the data from the JSON file, we can make a GET request just like we would if we were reaching out to an API.

However, instead of passing the address of the API in the call, we pass the relative location of the JSON file.

As the file is in the wwwroot folder, it will be available as a static asset at run time, just like the CSS file. This means the path we need to pass in the GET request is ”amiibos/amiiboData.json”.

CONTENT LOGIC

Para aceder ao ficheiro através do browser:

http://localhost:5266/amiibos/amiiboData.json

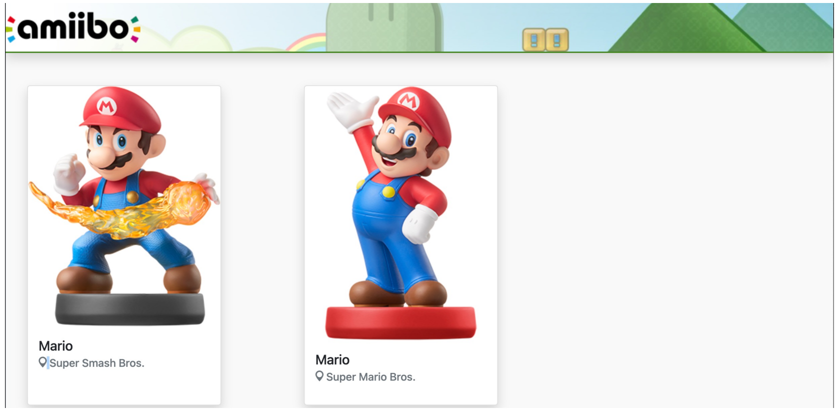
```
@code{
      private IEnumerable<Amiibo>? _amiibos;
      protected override async Task OnInitializedAsync()
      {
        try{
          _amiibos = await Http.GetFromJsonAsync
<IEnumerable<Amiibo>>("http://localhost:5266/amiibos/amiiboData.json");
        }
        catch (HttpRequestException ex) {
          Console.WriteLine($"There was a problem loading amiibos data: {ex.Message}");
        }
      }
}
```

Code block to process HomePage logic

CONTENT LOGIC

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

@if (_amiibos == null) {
       <p>Loading amiibos...</p>
}
else
{
       <div class="grid">
          @foreach (var amiibo in _amiibos) {
             <div class="card shadow" style="width: 18rem;">
                <img src="@amiibo.Image" class="card-img-top"
alt="@amiibo.Name">
                <div class="card-body">
                    <h5 class="card-title">@amiibo.Name</h5>
                    <h6 class="card-subtitle mb-3 text-muted">
                       <span class="oi oi-map-marker"></span>
                       @amiibo.AmiiboSeries
                    </h6>
                </div>
             </div>
          }
      </div>
}
```

Data presentation



Mario
Super Smash Bros.

Mario
Super Mario Bros.

CONTENT LOGIC

There’s a fair amount of code for creating the amiibo card;

While it’s all perfectly valid as is, wouldn’t it be nice to encapsulate it all in a component instead?

This would make the code in the HomePage component much easier to read.

Create a new component called AmiiboCard.razor in the Home feature folder. Then replace the boilerplate code with the markup for the card from the HomePage.

How do we get access to the current amiibo data?

The answer is parameters.

CONTENT LOGIC

We can pass data into components via parameters.

Think of these as the public API for a component, and they work one way, from parent to child.

We can define them in the code block by creating a public property and decorating it with the Parameter attribute.

We pass data into them from the parent using attributes on the component tag.

For our AmiiboCard component, we'll create a parameter that will allow us to pass in the current amiibo data from the parent.

CONTENT LOGIC

```
<div class="grid">
  @foreach (var amiibo in _amiibos)
  {
    <AmiiboCard amiibo="amiibo" />
  }
</div>
```

Index.razor

CONTENT LOGIC

```
<div class="card shadow" style="width: 18rem;">
    <img src="@amiibo.Image" class="card-img-top"
alt="@amiibo.Name">
    <div class="card-body">
       <h5 class="card-title">@amiibo.Name</h5>
       <h6 class="card-subtitle mb-3 text-muted">
          <span class="oi oi-map-marker"></span>
          @amiibo.AmiiboSeries
       </h6>
     </div>
</div>
@code {
  [Parameter, EditorRequired]
  public Amiibo amiibo { get; set; } = default!;
}
```

AmiiboCard.razor

In addition to using the Parameter attribute, we've also added another attribute called EditorRequired.

We can use it to indicate that a parameter is required.

If we try to use the AmiiboCard component now, without passing a parameter, we'll get a warning

COMPONENT LIFE CYCLE METHODS

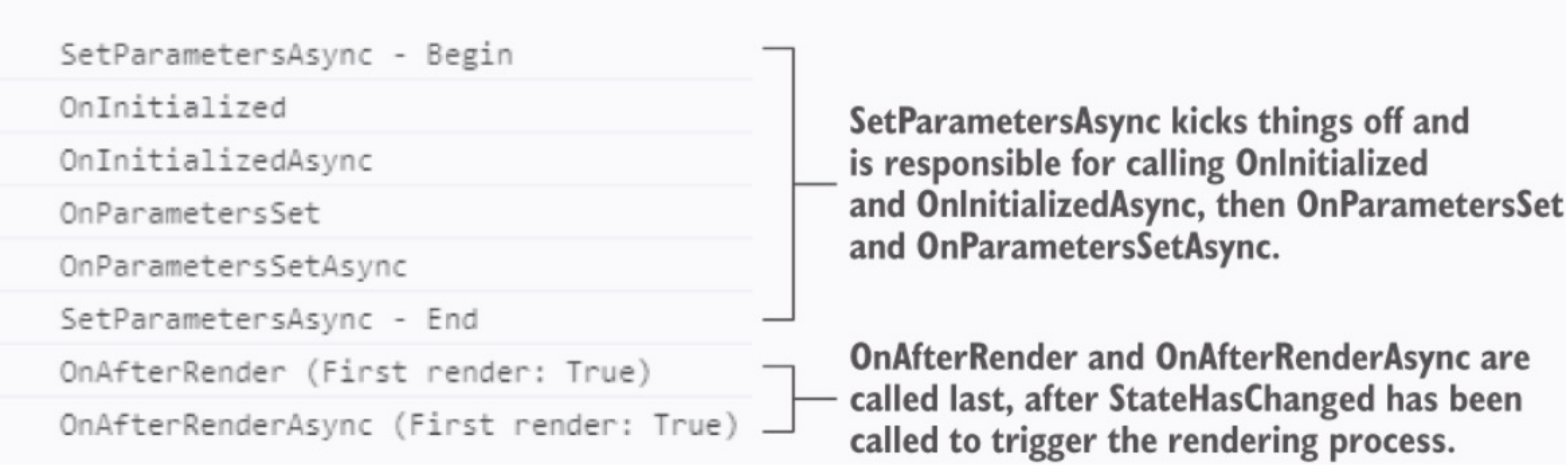
Components in Blazor have a life cycle: they're created, they exist for a period, and then they're destroyed;

Depending on what an application is doing, it may need to perform actions at certain points during this life cycle—for example, load initial data for the component to display when it is first created, or update the UI when a parameter has a certain value from the parent. Blazor supports this by giving us access to the component life cycle at specific points, which are:

- OnInitialized/OnInitializedAsync
- OnParametersSet/OnParametersSetAsync
- OnAfterRender/OnAfterRenderAsync

COMPONENT LIFE CYCLE METHODS

Each method has a synchronous and asynchronous version. The synchronous version is always called before the asynchronous version.



```
SetParametersAsync - Begin
OnInitialized
OnInitializedAsync
OnParametersSet
OnParametersSetAsync
SetParametersAsync - End
OnAfterRender (First render: True)
OnAfterRenderAsync (First render: True)
```

SetParametersAsync kicks things off and is responsible for calling OnInitialized and OnInitializedAsync, then OnParametersSet and OnParametersSetAsync.

OnAfterRender and OnAfterRenderAsync are called last, after StateHasChanged has been called to trigger the rendering process.

COMPONENT LIFE CYCLE METHODS

SetParametersAsync is not a life cycle method that is often used by developers. Commonly, it is just OnInitialized, OnParametersSet, and OnAfterRender.

During the first render, the component hasn't been initialized.

This means that On-Initialized and OnInitializedAsync will be called first—it is also the only time they will run.

- This pair of methods is the only one that runs once in a component's lifetime.
- You can think of these as constructors for your component.

COMPONENT LIFE CYCLE METHODS

Once the OnInitialized methods have run, OnParametersSet and On-ParametersSetAsync are called.

These methods allow developers to perform actions whenever a component's parameters change. In the case of a first render, the component's parameters have been set to their initial values

The final methods to run are OnAfterRender and OnAfterRenderAsync.

These methods both take a Boolean value indicating if this is the first time the component has been rendered.

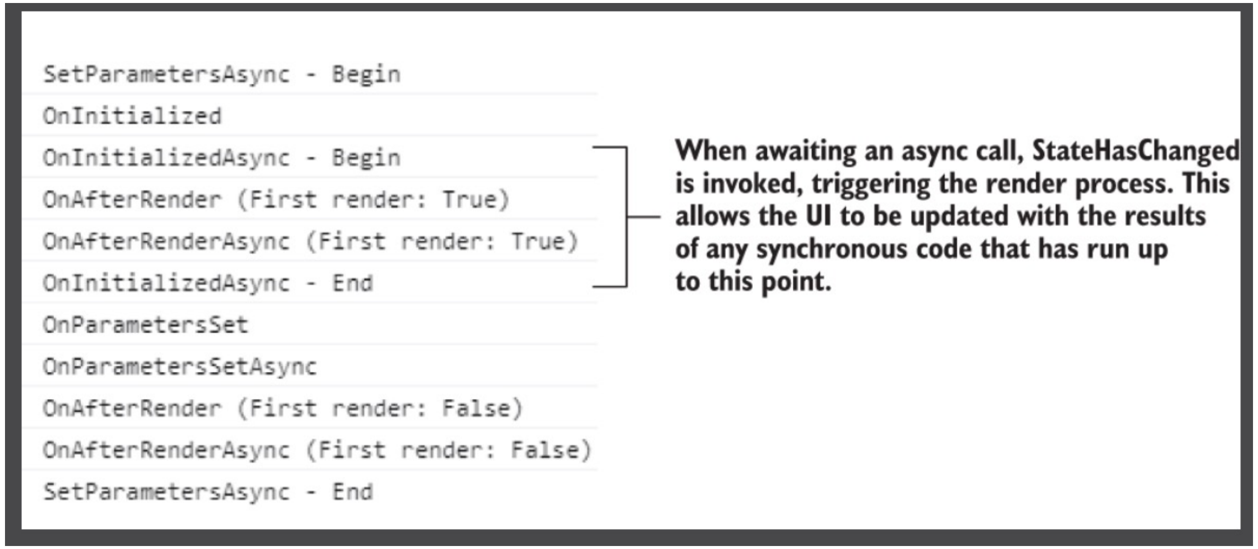
The primary use of the OnAfterRender methods is to perform JavaScript interop and other DOM-related operations, such as setting the focus on an element

THE LIFE CYCLE WITH ASYNC

One key point about the render we just covered is that it ran synchronously.

In the Lifecycle component, there are no awaited calls in any of the async life cycle methods, meaning each method ran in sequence.

However, when async calls are added, then things look a bit different.



```
SetParametersAsync - Begin
OnInitialized
OnInitializedAsync - Begin
OnAfterRender (First render: True)
OnAfterRenderAsync (First render: True)
OnInitializedAsync - End
OnParametersSet
OnParametersSetAsync
OnAfterRender (First render: False)
OnAfterRenderAsync (First render: False)
SetParametersAsync - End
```

When awaiting an async call, StateHasChanged is invoked, triggering the render process. This allows the UI to be updated with the results of any synchronous code that has run up to this point.

THE LIFE CYCLE WITH ASYNC

While Blazor was awaiting the async call, the component was rendered.

It was then rendered a second time after the OnParametersSet methods.

This is because Blazor checks to see if an awaitable task is returned from OnInitializedAsync.

If there is, it calls StateHasChanged to render the component with the results of any of the synchronous code that has been run so far, while awaiting the completion of the task.

This behavior is also true for async calls made in OnParametersSetAsync

THE LIFE CYCLE WITH ASYNC

When dealing with multiple asynchronous calls, rendering may not behave quite as you’d expect;

```
@foreach (var word in _greeting)
{
    <p>@word</p>
}
@code {
    List<string> _greeting = new List<string>();
    protected override async Task OnInitializedAsync()
    {
       _greeting.Add("Welcome");
       await Task.Delay(1000);
       _greeting.Add("to");
       await Task.Delay(1000);
_      greeting.Add("Blazor in Action");
    }
}
```

What happens is: the word Welcome is displayed, then after 2 seconds the words to Blazor in Action are added

THE LIFE CYCLE WITH ASYNC

The code up to the first awaited method is executed, and a call is made to StateHasChanged at this point to render the results of any synchronous code while awaiting that task.

This explains the render of the word Welcome but not why the word to isn't rendered after the first awaited call.

The reason for this is that Blazor doesn't understand our code.

There is no way for it to know that it should render after we add it to the greeting list.

Instead, the code continues to execute until the end of the method, and at this point, Blazor can perform a new render of the component.

THE LIFE CYCLE WITH ASYNC

If we want the UI to update after each word is added to the list, then we must manually call StateHasChanged to inform Blazor that the UI should be updated:

```
(...)
await Task.Delay(1000);
_greeting.Add("to");
StateHasChanged();
(...)
```

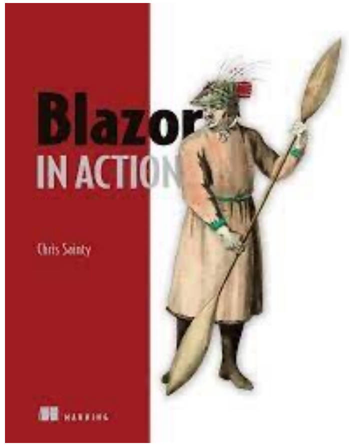
USEFUL RESOURCES

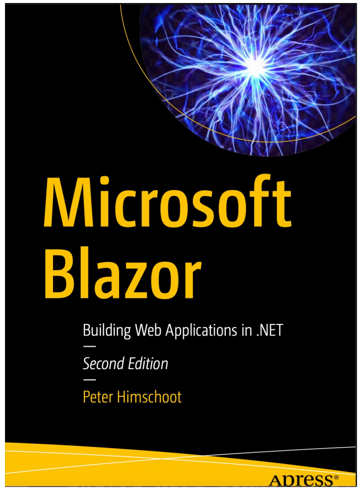
https://mudblazor.com

https://www.matblazor.com

https://blazor.radzen.com

RECURSOS





- https://dotnet.microsoft.com/learn/aspnet/blazor-tutorial/intro
- https://docs.microsoft.com/en-us/aspnet/core/tutorials/build-a-blazor-app?view=aspnetcore-5.0
- https://www.youtube.com/watch?v=8DNgdphLvag&t=2142s

INTRODUÇÃO AO DESENVOLVIMENTO WEB COM BLAZOR

INTRODUÇÃO

BRUNO OLIVEIRA – BRUNO.OLIVEIRA3@SAPO.PT