

Blazor

Introduction

Blazor Server

Building an Application

Bruno Oliveira - bruno.oliveira3@sapo.pt

2023

Outline

- Introduction
- Blazor Component
- Hosting models
- Creating a Blazor application
- Key Components
- Organizing files using feature folders
- Defining Layout
- Routable Components

Introduction



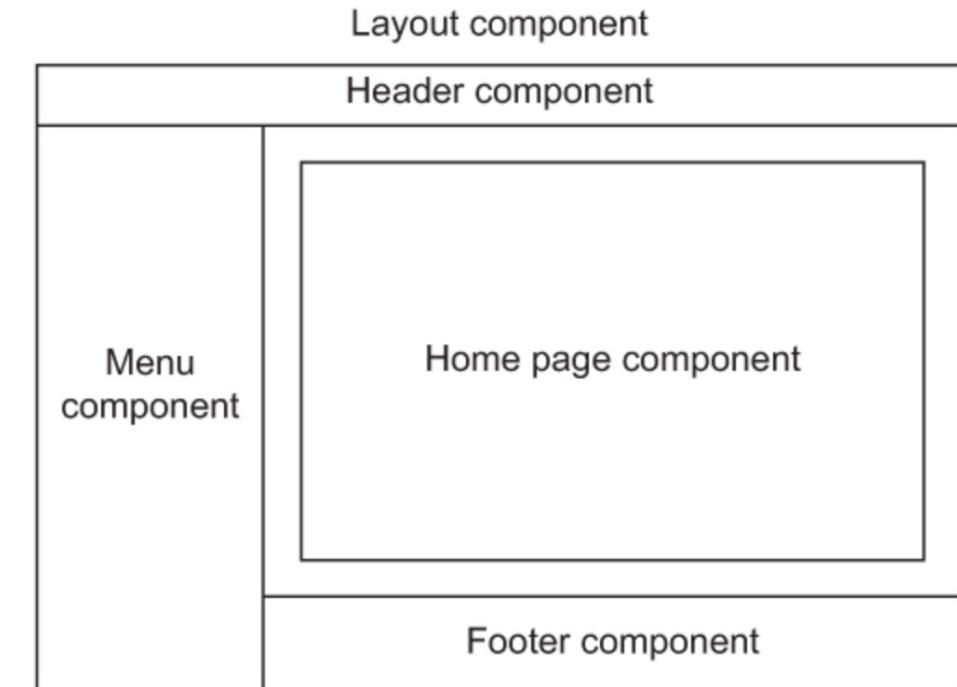
Introduction

Blazor, as with many modern [frontend frameworks](#), uses the concept of components to build the UI.

Everything is a component—[pages](#), [parts of a page](#), layouts.

Think of a component as a [building block](#).

Building blocks can be as big or as small as you decide; however, building an entire UI as a single component [isn't a good idea](#).



Anatomy of a Blazor component

This particular component is known as a [routable component](#), as it has a page directive declared at the top.

```
@page "/counter" ①

<h1>Counter</h1> ②

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0; ③

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Section 1—Used to define directives, add using statements, inject dependencies, or any other general configuration that applies to the whole component.

Section 2—Defines the [markup](#) of the component; this is written using the Razor language, a mix of C# and [HTML](#). Here we define the visual elements that make up the component.

Section 3—The [code](#) block. This is used to define the logic of the component. It is possible to write any valid C# code into this section. You can define fields, properties, or even [entire classes](#) if you wish.

Understanding hosting models

When first getting started with Blazor, you will immediately come across hosting models:

[Blazor WebAssembly](#)

[Blazor Server](#).

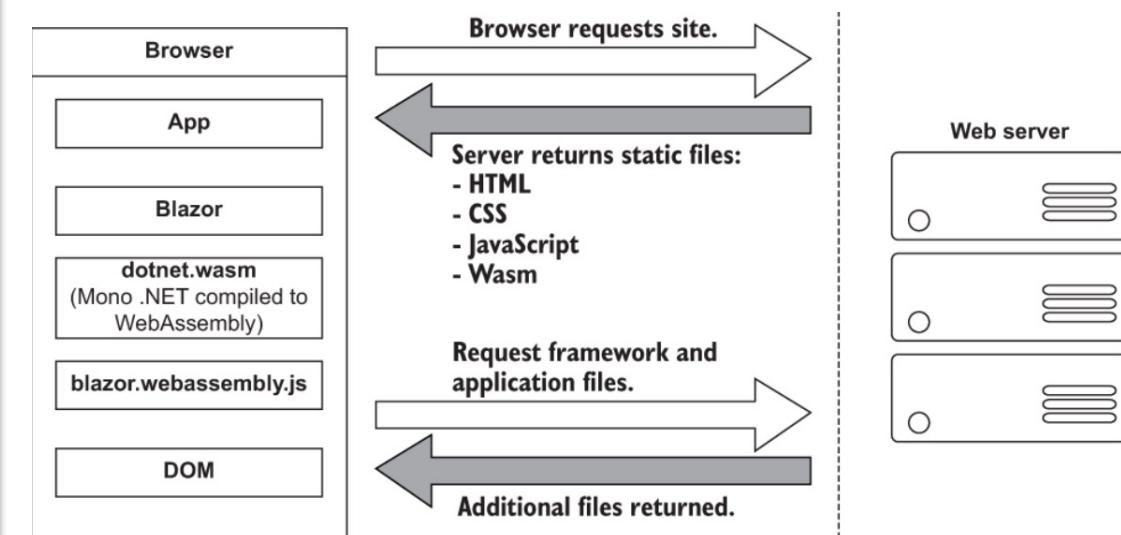
Regardless of which model you choose for your application, the [component model](#) is the same, meaning components are written the same way and can be interchanged between either hosting model

Blazor WebAssembly

Blazor WebAssembly allows your application to run entirely inside the client's browser

This part of the framework does three main things:

- Loads and initializes the Blazor application in the browser
- Provides direct DOM manipulation so Blazor can perform UI updates
- Provides APIs for JavaScript interop scenarios

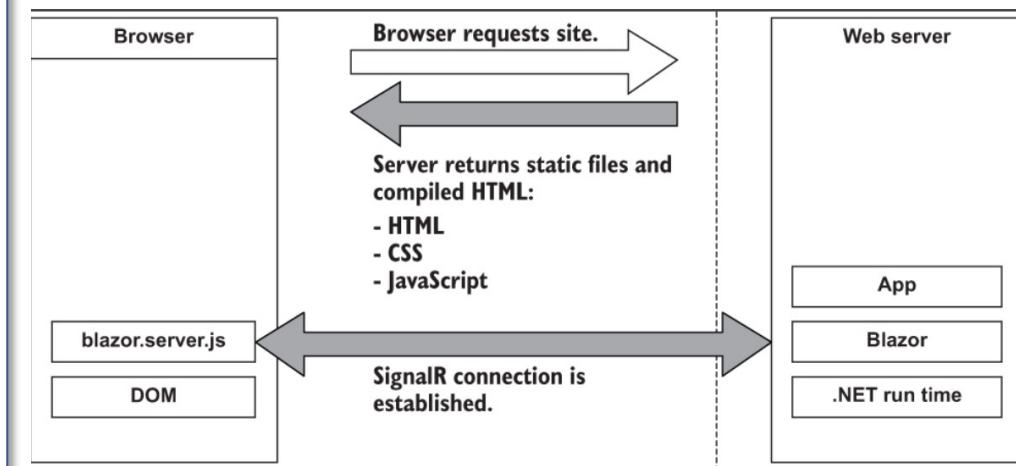


Blazor Server

Unlike Blazor WebAssembly, which behaves more like a [desktop application](#) with each user having their own instance, [Blazor Server](#) runs one instance of the app that [all users](#) connect to;

Once the [initial payload](#) is returned to the browser, the files are processed and the DOM is created—then a file called [blazor.server.js](#) is executed.

The job of this run time is to establish a [SignalR connection](#) back to the Blazor application running on the server.

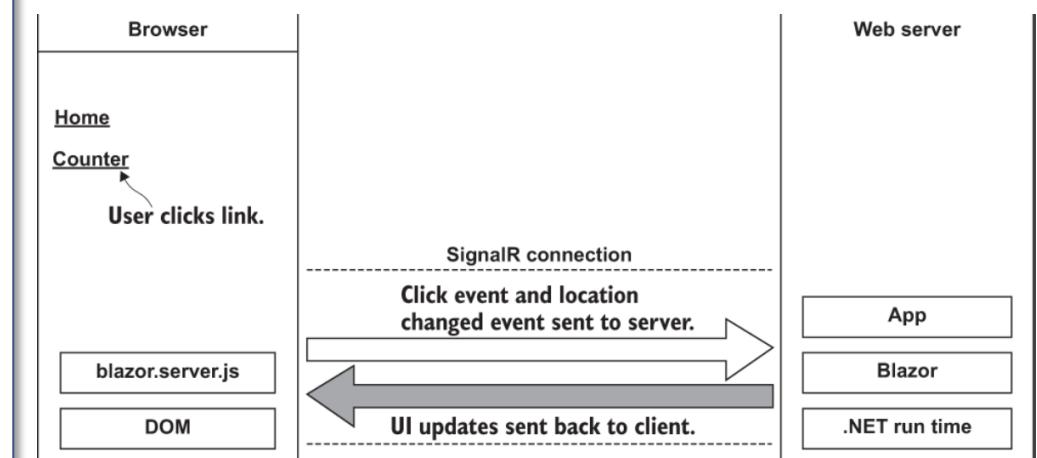


Blazor Server

The [user clicks](#) the link in the menu, and the [click event](#) is intercepted by Blazor's run time on the client.

Then processes the event to understand what happened: [mouse click](#) event and a navigation event, due to it being a clicked hyperlink.

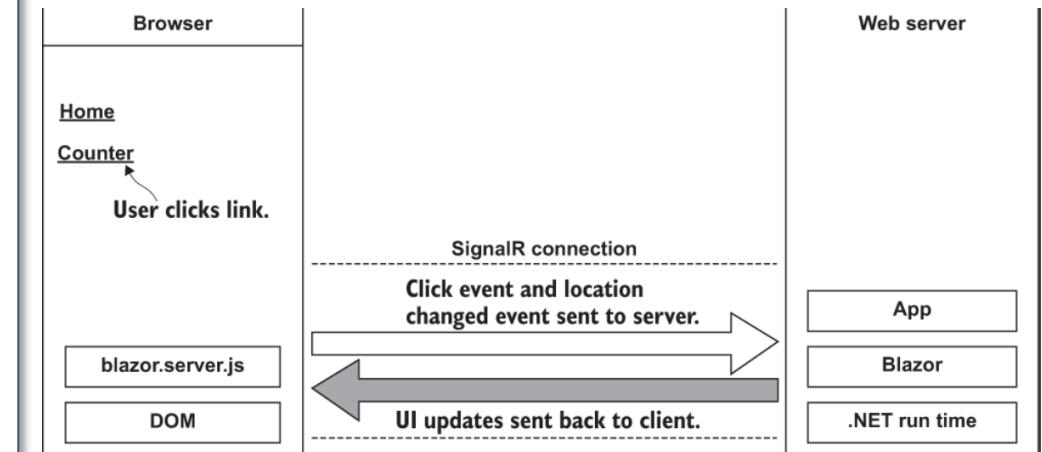
These [two events](#) are then bundled up and [sent back](#) to the server over the [SignalR](#) connection that was established when the application started.



Blazor Server

On the server, the message sent from the client is unpacked and processed.

The Blazor framework then calls any application code necessary.

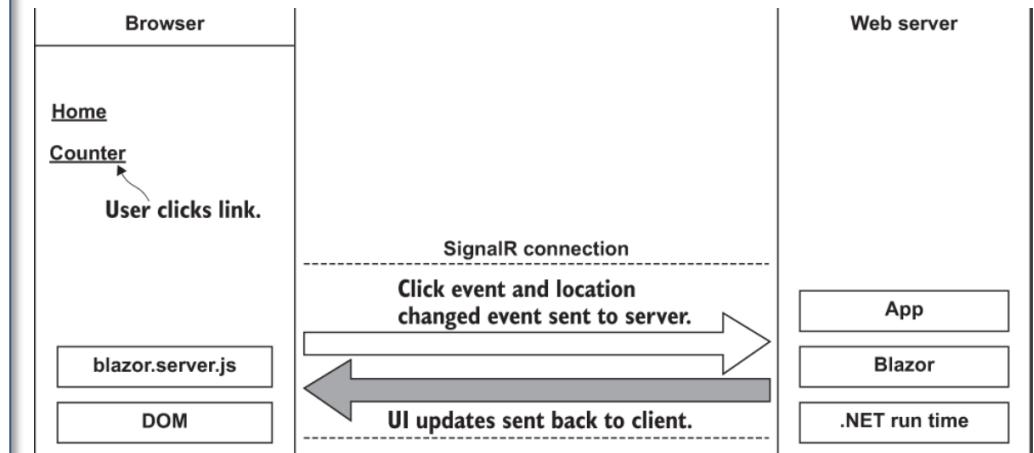


Blazor Server

Blazor will **not** send back an entirely new page to the client.

It will send back only the **minimum number of instructions** needed to update the current DOM to match the Counter page

Once back on the client, the changes are **unpacked** and the required changes are applied to the physical **DOM**.



Creating the Application

- In the terminal, run the following command to **create** your app:

```
dotnet new blazorserver -o app --no-https -f net7.0
```

- The **dotnet watch** command will **build** and **start** the app, and then **update** the app whenever you make code changes. You can stop the app at any time by selecting Ctrl+C.

```
cd app  
dotnet watch
```

Key components of a Blazor application

- `_Host.cshtml`
 - The root page of the app
 - 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
- `App.razor`
 - The **root component** of the app that sets up routing using the **Router component**. The Router component **intercepts** browser navigation and renders the page that matches the requested address.

Key components of a Blazor application

- **wwwroot folder and _Imports.razor**
 - All ASP.NET Core applications have a [wwwroot](#) folder, which is used to store [public static assets](#).
 - This is the place where you can put things such as images, CSS files, JavaScript files, or any other static files you need
 - Its job is to store [using statements](#). The benefit is that those using statements are made available to all the components in the file's directory and any subdirectories.

Organizing files using feature folders

- By default, the app structure used by the template divides files by [responsibility](#).
- There's a [Pages folder](#) for [routable components](#), and there's a [Shared folder](#) for anything that is used in [multiple places](#) or is a [global concern](#).
- 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](#).
- Instead, we're going to use a structure called [feature folders](#) to organize our application.

Organizing files using feature folders

- When using **feature folders**, all the files relating to that feature are stored in the **same place**.
- When you go to work on a particular feature, all the files you need are in the same place, making everything easier to understand and more **discoverable**.
- 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.

Pages	Features
Account.razor	Account
ProductList.razor	AccountPage.razor
Product.razor	Summary.razor
ShoppingBasket.razor	Details.razor
	AddressList.razor
	ProductList
AccountDetails.razor	ProductListPage.razor
AccountSummary.razor	ItemSummary.razor
AddressList.razor	
ItemSummary.razor	
ProductDetails.razor	Product
ProductStockAndPrice.razor	ProductPage.razor
ShoppingBasketItemSummary.razor	Details.razor
ShoppingBasketPaymentOptions.razor	StockAndPrice.razor
ShoppingBasketDeliveryOptions.razor	
	ShoppingBasket
Button.razor	ShoppingBasketPage.razor
Table.razor	ItemSummary.razor
	PaymentOptions.razor
	DeliveryOptions.razor
	Shared
Button.razor	
Table.razor	

Organizing files using feature folders

- You should put **any files** that relate to that feature in the folder—C# classes, TypeScript files, CSS files, anything at all.
- **Static assets** such as images are the only exception to this. These need to be placed in the wwwroot folder; otherwise they will not be available at run time.
- However, you can **mirror** your feature folder structure in the wwwroot folder if you wish.

Organizing files using feature folders

- The other useful little thing to do when using this organization system with Blazor is to append any routable component with the word [Page](#).
- When a feature has several other components in it, it's almost impossible to identify the routable component easily.

Organizing files using feature folders

Original structure

```
> bin  
> Data  
> obj  
> Pages  
> Properties  
> Shared  
  < wwwroot  
    > css  
    favicon.png  
    < Imports.razor  
    < app.csproj  
    < App.razor  
  {} appsettings.Development.json  
  {} appsettings.json  
  C# Program.cs
```

One folder for feature

Pages is preserved for holding _Host.cshtml and Error.cshtml

Remove shared folder and adjust references in _Imports.razor

Feature Folder

```
> bin  
> Data  
  < Features  
    < Counter  
      < Counter.razor  
    < FetchData  
    < Home  
      < Index.razor  
    < Layout  
  > obj  
> Pages  
> Properties  
  < wwwroot  
    < css  
      > bootstrap  
      > open-iconic  
      # site.css  
    favicon.png  
    < Imports.razor  
    < app.csproj  
    < App.razor  
  {} appsettings.Development.json  
  {} appsettings.json  
  C# Program.cs
```

Setting styling

- The Blazor templates ship with a CSS framework called [Bootstrap](#).
- You can place images in [wwwroot](#) in a folder called [images](#).
- You can add some custom styles to the [site.css](#) file.
- By adding the styles here, they will affect the [whole](#) application.
- These styles will customize the look of some common elements, and are placed in the [wwwroot > css folder](#).

Setting styling

- Open the `site.css` in the `wwwroot > css` folder. Then copy in the following code:

```
(...)
:root {
--brand: #448922;
--brand-hover: #5da030;
}

body {
background-color: #f9f9f9;
}

a {
color: var(--brand);
text-decoration: underline;
}
(...)
```

<https://gist.github.com/brunobmo/4760c85761f0fb91ffd5d1109bbb048e>

Setting styling

- You can make some adjustments to the `_Host.cshtml` page.
- First, we'll add a reference to Bootstrap Icons (<https://icons.getbootstrap.com/>).
- In order to use it, we'll add the following line to the head element of the index.html page in `wwwroot`:

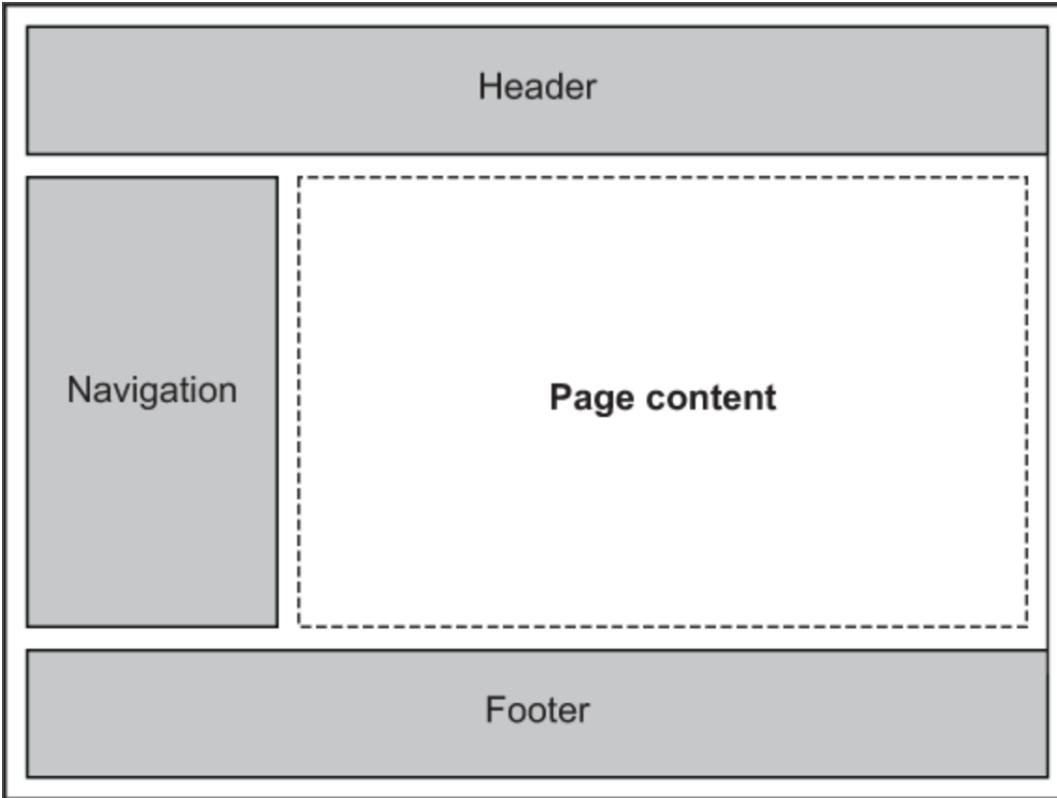
```
@page "/"
@using Microsoft.AspNetCore.Components.Web
@namespace app.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="app.styles.css" rel="stylesheet" />
    <link rel="icon" type="image/png" href="favicon.png"/>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.4.1/font/bootstrap-icons.css" rel="stylesheet"/>
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
</head>
```

Defining Layout

- The concept of a layout 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**.

Defining Layout



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>
```

Defining 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`.

Defining Layout

- We're going to update the **MainLayout** component.

```
@inherits LayoutComponentBase
```

```
<main class="container mt-5 mb-5">  
    @Body  
</main>
```



Defines the component as a layout component



Marks the location where page content is rendered in the layout

Defining Layout

- The Header will be defined in a [separate component](#): `Header.razor`

```
└─ Layout
    └─ Header.razor
    └─ MainLayout.razor
```

```
<nav class="navbar mb-5 shadow">
  <a class="navbar-brand" href="/">
    
  </a>
</nav>
```

<https://gist.github.com/brunobmo/d121e1d18fac84fcc1081cc527fb9876>

- We can now add that to the `MainLayout`:

```
@inherits LayoutComponentBase

<Header />

<main class="container mt-5 mb-5">
  @Body
</main>
```

<https://gist.github.com/brunobmo/a0b57ba04f60177b61be73fdcdf45f83>

Routable Components

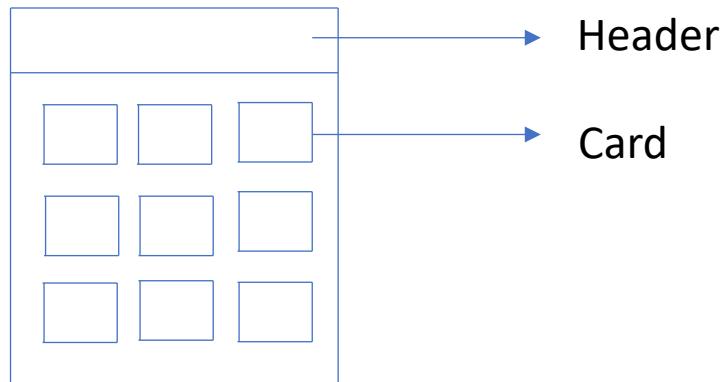
- To make a component **routable**, we need to use the `@page` directive and a `route template` that specifies the route it will be responsible for:

```
@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.

Creating a page

- Let's create the concept of "Card" to create a page with the following structure:



Creating a page

```
namespace app.Features.Home;

public class Card
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public string Description { get; set; } = "";
    public string Image { get; set; } = "";
}
```

<https://gist.github.com/brunobmo/cca963ea4e3f717a603fcdbc6e7997db>

Creating a page

- Let's create some sample data.
- We will do an http call to load data from the backend.

```
[  
 {  
   "id": 1,  
   "name": "Card 1",  
   "description": "Card 1 description",  
   "image": "(...)"  
 },  
 {  
   "id": 2,  
   "name": "Card 2",  
   "description": "Card 2 description",  
   "image": "(...)"  
 }]  
https://gist.github.com/brunobmo/8623e8b9744356b960fcc9f5f968f9ba
```

Creating a page

- 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;

Creating a page

- The `inject` directive:
 - The `inject` directive allows us to quickly and easily [inject instances of objects](#), registered with the service container in [Program.cs](#), into our components.
 - So, if we use the `Inject` directive for http client, it ends up compiled to this:

```
[Inject]  
public HttpClient Http { get; set; }
```

Creating a page

- Let's change the Index.Razor page (inside Home folder) to this:

```
@page "/"
@inject HttpClient Http
<PageTitle>Index</PageTitle>
<h1>Hello, world!</h1>
```

Welcome to your new app.

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

```
@code{
    private IEnumerable<Card>? _cards;
```

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

The Private field holds cards data

<https://gist.github.com/brunobmo/7143d6093dc6275055f750cc6de0ad10>

Creating a page

- For using the Http client you need to:

- Install the Nugget package:

- `dotnet add package System.Net.Http --version 4.3.4`

- Registration of the service in program.cs file:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;
using app.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();
builder.Services.AddHttpClient(); ( ... )
```

Creating a page

- 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](#).
- [OnInitialized](#) is run only once in the component's lifetime, making it perfect for [loading initial data](#);

Creating a page

- 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.
- As the file is in the **wwwroot** folder, it will be available as a **static asset** at run time, just like the CSS file.

Creating a page

- Let's change the Index.razor:

```
@inject HttpClient Http
@Inject NavigationManager NavigationManager
}

@code {
    private I Enumerable<Card>? _cards;
    protected override async Task OnInitializedAsync(){
        try{
            string domainName = NavigationManager.Uri;
            _cards = await Http.GetFromJsonAsync<I Enumerable<Card>>(domainName+"cards/cards.json");
        }
        catch (HttpRequestException ex){
            Console.WriteLine($"There was a problem loading cards data: {ex.Message}");
        }
    }
}
```

<https://gist.github.com/brunobmo/dc1a1e666ec0d72111feabd8aefd70c9>

Note: Inject is used to manage dependencies between objects, while Using is used to manage the lifecycle of IDisposable objects.

Creating a page

- For showing data, we can use a simple if statement in our markup to check the value of the `_cards` field.
- If it's `null`, then we can surmise that the data is still being loaded, excluding an error scenario, of course.
- If the value is `not null`, then we have some data, and we can go ahead and display it (see the following listing).

Creating a page

Index.razor

```
@page "/"
(....)
@if (_cards == null) {
    <p>Loading cards...</p>
}
else{
    <div class="grid">
        @foreach (var card in _cards) {
            <div class="card shadow" style="width: 18rem;">
                
                <div class="card-body">
                    <h5 class="card-title">@card.Name</h5>
                    <h6 class="card-subtitle mb-3 text-muted">
                        <span class="oi oi-map-marker"></span>
                        @card.Name
                    </h6>
                (...)
            </div>
        }
    </div>
}
(....)
```

<https://gist.github.com/brunobmo/cec57870da873ad48e289375722c5399>

Final Result



Source code available at: https://github.com/brunobmo/Blazor_Course

The screenshot shows a Blazor application interface. At the top center is a circular logo for "DATA CITY" with the tagline "WHERE DATA LIVES". Below it is a horizontal green line. The main content area contains two cards, each featuring a logo and some descriptive text.

Card 1
📍 Card 1
⌚ Card 1 description ⚡ Card 1 description

Card 2
📍 Card 2
⌚ Card 2 description ⚡ Card 2 description

The first card's logo consists of five stylized human figures in purple, blue, and orange. The second card's logo features two hands, one teal and one orange, forming a heart shape.

Resources

Component Libraries

- <https://mudblazor.com>
- <https://www.matblazor.com>
- <https://blazor.radzen.com>

Learning

- <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>
- Book: Blazor in Action

Blazor

Introduction

Blazor Server

Building an Application

Bruno Oliveira - bruno.oliveira3@sapo.pt

2023