

Fundamentos do Blazor



[Antes de começar](#)

[André Baltieri](#)

[Onde aprender mais sobre Blazor](#)

[Capítulo 1 - Presente e Futuro](#)

[Por que ainda usamos o ASP.NET hoje?](#)

[Qual era o problema do ASP.NET?](#)

[SSR vs SPA](#)

[Server Side Rendering](#)

[Single Page Application](#)

[Apresentando o Blazor](#)

[Blazor Server](#)

[Blazor WebAssembly](#)

[Os problemas do Blazor](#)

[Como fica o ASP.NET?](#)

[Unificação dos projetos](#)

[Templates anteriores](#)

[Qual template escolher?](#)

Capítulo 2 - Visão Geral

Entendendo as segmentações do Blazor

Blazor SSR e Server

Blazor WASM

Blazor Auto

A mágica do RenderMode Auto

Capítulo 3 - Arquiteturas

Instalação do .NET

Criando um projeto com Blazor Server

wwwroot

Data

Pages

Shared

_Imports.razor

App.razor

Program.cs

Criando um projeto com Blazor Wasm

wwwroot

Components

Layout

Pages

_App.razor

_Imports.razor

App.razor

Program.cs

Criando um projeto com Blazor

Executando os demos

Capítulo 4 - Startup

Criando o projeto

Entendendo os pacotes

Startup.cs

App.razor

Head Outlet

Routes

blazor.web.js

Routes.razor

_Imports.razor

Adicionando suporte a Blazor no Razor Pages

Adicionando suporte a Razor Pages no Blazor

Blazor e Razor Pages juntos

Capítulo 5 - Páginas e Components

O que é uma página?

Estrutura de uma página

O que é um componente?

Estrutura de uma componente

O que é um Layout?

Estrutura de uma layout	
Interação entre elementos da página	
Binding e Two-way binding	
Separando o código da página	
Separando os estilos	
Entendendo o CSS dinâmico	
Parâmetros	
Capítulo 6 - Render Modes	
O que são Render Modes	
SSR	
Como é o processo de SEO	
Como usar SSR no Blazor	
SSR Streaming	
Como esta mágica acontece?	
Interactive Server	
Criando projetos no Visual Studio	
Interactive WebAssembly	
Inspecionando o projeto	
Executando o projeto	
Auto	
Inspecionando o projeto	
Executando o projeto	
Capítulo 7 - Rotas e Navegação	
Rotas	
Links	
NavLink	
Match	
ActiveClass	
NavigationManager	
FocusOnNavigate	
Rotas em diferentes projetos	
Parâmetros de Rotas	
Parâmetros opcionais	
Restrições de parâmetros	
Restrições e parâmetros opcionais	
Catch All	
Query Strings	
Capítulo 8 - Formulários	
Overview	
EditForm	
Validação do Formulário	
Context Binding	
Form Submit	
Componentes nativos disponíveis	
Conclusão	
Onde aprender mais sobre Blazor	

Antes de começar



André Baltieri

Me dedico ao desenvolvimento de software desde 2003, sendo minha maior especialidade o Desenvolvimento Web.

*Durante esta jornada pude trabalhar presencialmente aqui no Brasil e Estados Unidos, atender remotamente times da Índia, Inglaterra e Holanda, receber **11x Microsoft MVP** e realizar diversas consultorias em empresas e projetos de todos os tamanhos.*

[Instagram](#) | [LinkedIn](#) | [GitHub](#)

Onde aprender mais sobre Blazor

Temos uma carreira focada em [ASP.NET](#) e Blazor, com 4 cursos completos e mais de 12 horas de duração.

Carreira 03. ASP.NET/Blazor Developer

Aprenda criar APIs e aplicações web utilizando ASP.NET e Blazor



<https://balta.io/carreiras/aspnet-blazor-developer>



Recomendamos seguir todas as nossas carreiras na ordem, assim você aprenderá desde C# até Blazor, com o mesmo instrutor, mesma didática e muita mão na massa.

Carreiras

Comece sua carreira como desenvolvedor da forma correta!



<https://balta.io/carreiras>

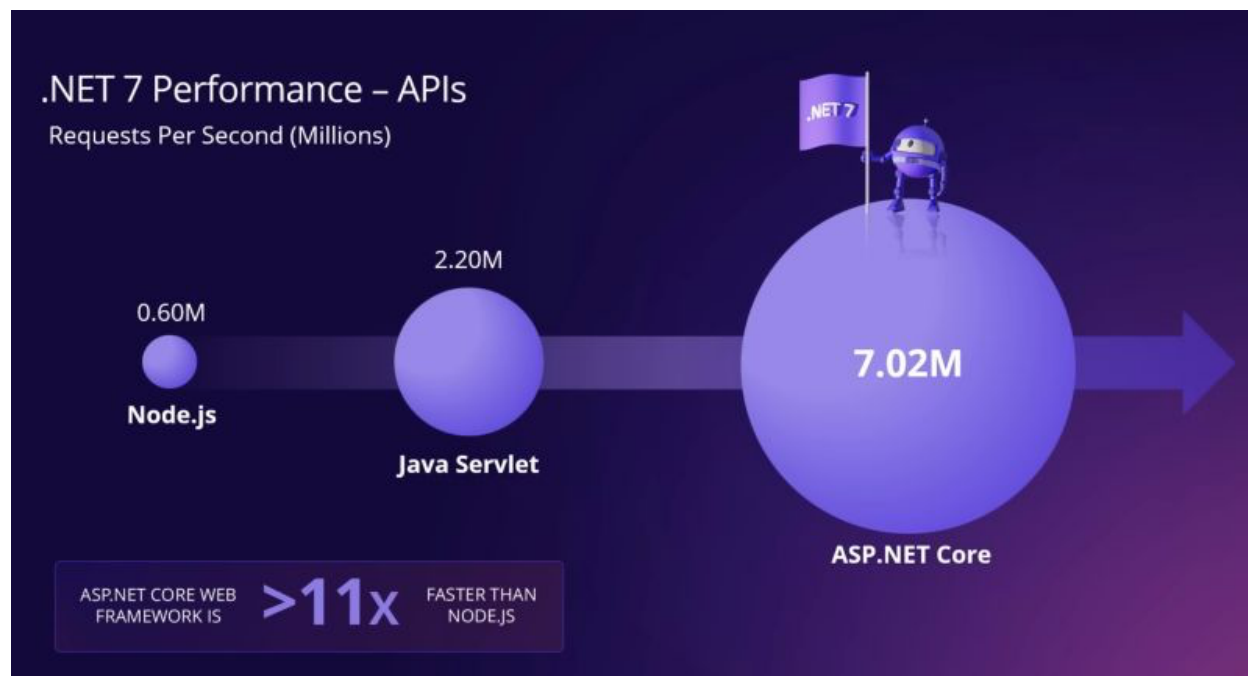
Capítulo 1 - Presente e Futuro

Por que ainda usamos o ASP.NET hoje?

Sabemos que C# é a linguagem de programação, .NET é o Framework (Conjunto de bibliotecas que torna o desenvolvimento mais rápido) e que o próprio .NET é escrito em C#.

O ASP.NET é o principal Framework para criação de Aplicações Web, APIs e WebSites da Microsoft, que faz uso do .NET como base (Ele é escrito em cima do .NET).

Assim como o C# e o .NET, o ASP.NET tem mais de 20 anos de existência e evoluiu muito nesse meio tempo. Atualmente ele é um dos Frameworks mais rápidos do mercado, ultrapassando com folga seus principais concorrentes.



Qual era o problema do ASP.NET?

Sendo sincero, não era bem um problema, era mais uma expansão que faltava. O ASP.NET sempre foi muito popular no Backend, para criação de APIs.

Porém, com a popularização de Frameworks SPA (Single Page Applications) o ASP.NET acabou sendo deixado de lado em diversos cenários.

Isto por que os navegadores executam apenas HTML, CSS e JS, tornando quase que obrigatório o uso do JavaScript.

O ASP.NET consegue servir muito bem a parte do SSR (Server Side Rendering) ou renderização do lado do servidor, mas ele para por aí.

Na verdade ele só faz isto! Toda e qualquer aplicação Web (Não API) que você criar com ASP.NET, vai ser renderizada do lado do servidor.

Isto significa que se você quiser atualizar um simples contador na tela por exemplo, precisa enviar uma requisição para o servidor, que vai gerar um HTML, CSS e JS e enviar de volta para tela.

Já nas aplicações SPA (Aplicações de página única), sua aplicação é renderizada uma vez e após isto, buscamos somente os dados no servidor (Requisição para uma API, normalmente JSON) e geramos o HTML necessário via

JavaScript.

SSR vs SPA

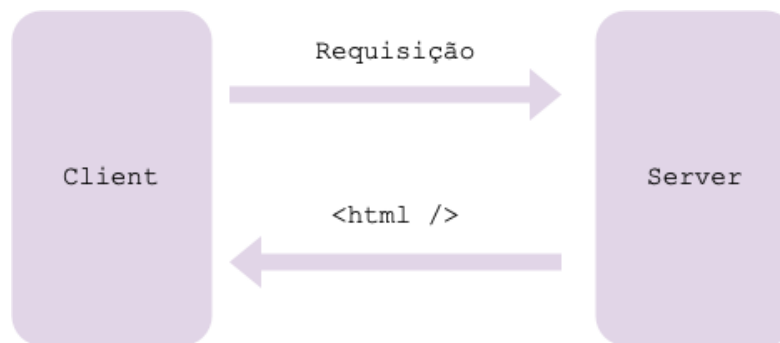
Tanto o SSR quanto o SPA desenvolvem um papel único no cenário Web, e ambos tem suas características.

Server Side Rendering

Do lado do SSR, podemos destacar diversos pontos fortes e fracos que já comentamos, mas em resumo eu elegeria dois principais:

- Carregamento mais rápido (Só o que precisa)
- Melhor otimização do SEO (Google e buscadores)

Abaixo temos uma ilustração de como funcionam as requisições e respostas em uma aplicação que se beneficia do SSR.



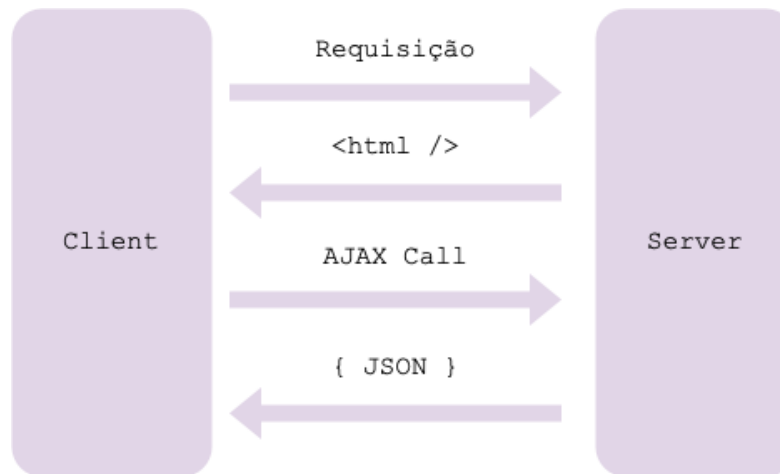
Podemos notar que é uma relação extremamente simples, que desfruta da base de toda aplicação Web, que é uma requisição e uma resposta.

Single Page Application

Do lado do SPA, também podemos destacar diversos pontos fortes e fracos que já comentamos, mas em resumo eu elegeria três principais:

- Carregamento inicial mais demorado
- Melhor interação com usuário
- Mais responsiva

Abaixo temos uma ilustração de como funcionam as requisições e respostas em uma aplicação que se beneficia do SPA.



Como podemos ver, neste modelo, nós temos o carregamento inicial similar ao SSR, porém, posteriormente fazemos apenas requisições assíncronas, recebendo um JSON como retorno e gerando o HTML necessário.

AJAX é a sigla para Asynchronous JavaScript and XML, é um termo que foi muito usado no passado, nos primórdios das requisições assíncronas. Hoje podemos chamar apenas de “requisição”, já que o XML nem é mais utilizado.

Apresentando o Blazor

Há alguns anos a Microsoft chegou com uma proposta legal e ousada, que inclusive eu comparei com Silverlight na época.

Embora simples, a proposta do Blazor era trazer suporte aos itens que o SPA tem de legais como responsividade, interação com usuário e também unir isto ao SSR com um carregamento mais rápido.

Mas estas de longe eram as propostas mais ousadas, a maior delas foi introduzir o C# no Frontend, não no lugar do JavaScript (Você ainda pode usar JS, inclusive interoperar com C#) mas sim como um complemento a ele.

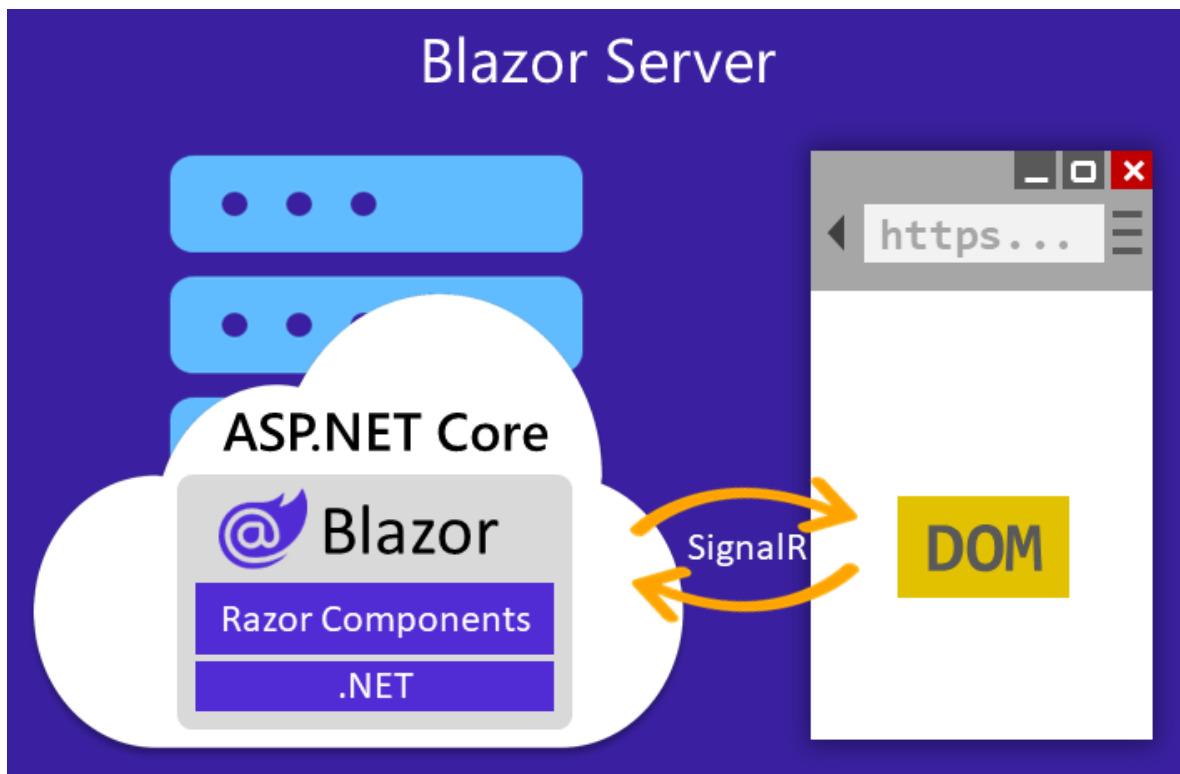
Então, ela fechou o ciclo, com ASP.NET no Backend e SSR e Blazor para SPA, dando de quebra a possibilidade de trabalhar com C# no Frontend.

O Blazor ainda foi segmentando em duas partes, uma chamada de Blazor Server e outra de Blazor Wasm, ambas com propósitos diferentes.

Blazor Server

A ideia principal do Blazor Server é rodar no servidor, mas diferente de como o ASP.NET faz com Razor Pages.

Enquanto o ASP.NET renderiza toda a página novamente, o Blazor foi mais esperto. Ele mantém uma conexão aberta e atualiza só o componente que está sendo manipulado.



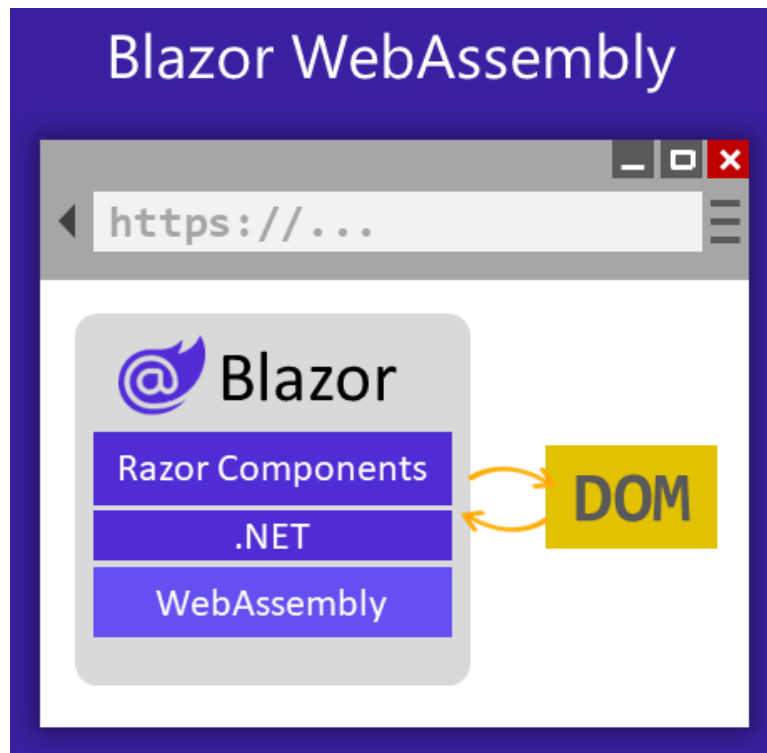
FONTE: Microsoft Learn

Isto significa que temos a aplicação servida inicialmente pelo servidor, de forma rápida e posteriormente, os componentes são atualizados através de um Socket (Conexão que fica aberta direta com servidor sendo muito mais rápida).

Blazor WebAssembly

Na outra mão temos o Blazor WebAssembly, ou Blazor WASM, que faz um trabalho bem diferente do seu irmão.

Web Assembly não é uma exclusividade da Microsoft, outras tecnologias como Rust se beneficiam dele, e ele basicamente permite a execução de códigos de máquina no navegador utilizando linguagens que não são necessariamente o JavaScript.



FONTE: Microsoft Learn

Isto não só implica em uma maior performance como em uma maior flexibilidade, dando a opção de reuso por exemplo, de regras de negócio escritas em C#.

Em resumo, o Blazor Wasm é literalmente o .NET rodando dentro do navegador, não há nada (Exceto os arquivos estáticos) sendo servidos no servidor.

Então após o carregamento inicial da aplicação, temos uma excelente performance e claro, o uso e abuso do nosso querido C#.

Os problemas do Blazor

Mas como nem tudo são flores, tanto o Blazor Server quanto o Blazor Wasm tem seus defeitos. Enquanto o Blazor Wasm tem uma ótima performance, sua fraqueza é o tamanho do seu App, que precisa do .NET para ser executado e consequentemente fica muito maior do que uma aplicação convencional.

No lado do Blazor Server, manter a conexão sempre aberta com o servidor pode ser custoso e renderizar novamente os componentes nem sempre é o jeito mais performático.

Além disso, perdemos a parte do SEO, já que só o “esqueleto” da página é renderizado inicialmente, o que deixa o Google triste.

Para finalizar, não conseguimos (Até o .NET 7) utilizar Blazor Server, WASM e SSR juntos, e esta é a grande novidade do .NET 8.

Como fica o ASP.NET?

Mas se no .NET 8 podemos utilizar SSR, WASM e recursos do Blazor Server, por que ainda precisamos do ASP.NET?

Nos últimos anos o ASP.NET sofreu diversas melhorias significativas, desde os Top-Level Statements (Não foi só para o ASP.NET) até a chegada dos Minimal APIs.

Sendo assim, eu encaro o ASP.NET com mais foco em APIs a partir do .NET 8, sendo mais específico em Minimal APIs.

Em caso de aplicações SSR (Websites) você ainda pode manter o Razor Pages, mas tem muita coisa que já pode ser feita com Blazor.

Para o MVC o cenário é um pouco menos favorável, visto que a arquitetura era considerada complexa para iniciar um projeto, e consome mais recursos que os Minimal APIs.

Então utilizar o MVC fica mais para projetos que realmente necessitem esta arquitetura. E antes de dizer que não tem como organizar bem as Minimal APIs, dá uma conferida neste repositório.

<https://github.com/balta-io/3001>

Unificação dos projetos

Indo além, no .NET 8 temos a unificação dos templates também, sendo que para criar uma nova aplicação Blazor com suporte a SSR, WASM e Blazor Server, basta usar o comando `dotnet new blazor`.



Este novo projeto combina recursos do ASP.NET, Blazor Server e Blazor Wasm, nos permitindo criar Razor Components, Razor Pages, Blazor Pages, Blazor Components e até utilizar o Blazor WebAssembly no projeto.

Templates anteriores

Ainda é plenamente possível utilizar templates anteriores como `blazorserver`, `blazorwasm`, `razorpages` e `mvc`. Nada foi removido, apenas adicionado.

Qual template escolher?

A opção mais flexível é o `dotnet new blazor` por possuir recursos tanto para SSR quanto SPA, dando ainda suporte ao WASM.

Porém é sempre legal fazer uma análise para entender o que seu projeto de fato precisa e adicionar somente o necessário.

De qualquer forma, não se preocupe, dá pra adicionar o Blazor a um projeto ASP.NET e adicionar ASP.NET a um projeto Blazor.

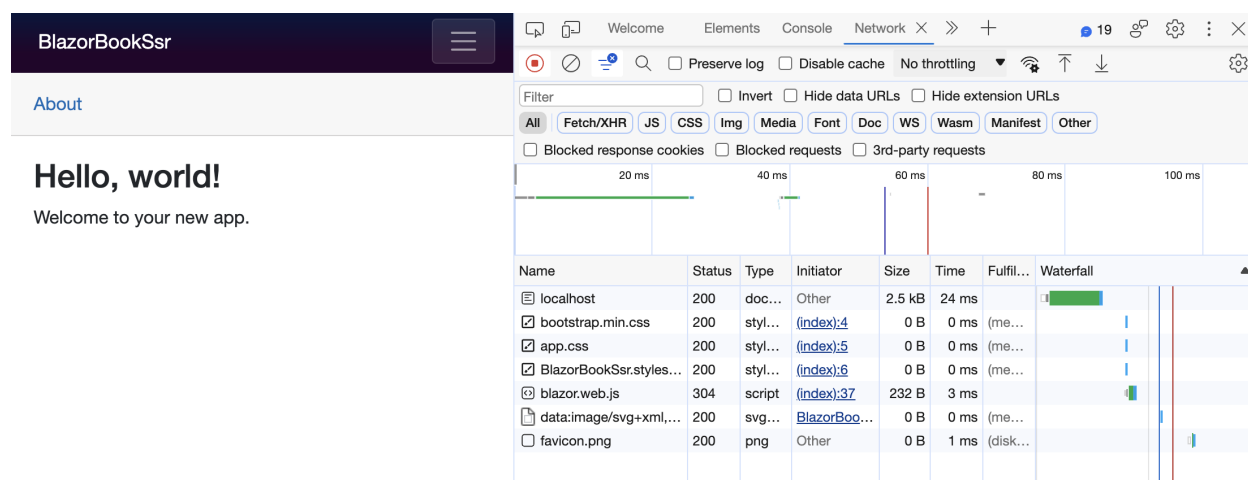
Capítulo 2 - Visão Geral

Entendendo as segmentações do Blazor

No capítulo anterior falamos sobre SPA, SSR, WASM e agora vamos entender com mais detalhes como o Blazor trata estes itens, vamos entender na prática a diferença entre uma aplicação WASM e um Server ou SSR.

Blazor SSR e Server

Quando servimos uma aplicação Blazor do lado do servidor nós temos um comportamento parecido (Explicado na parte de Streaming) que é a renderização da página por completo.



Este é um comportamento que acontece em basicamente todas as aplicações em um primeiro momento, e no Blazor SSR ou Server não é diferente.

Temos a requisição e carregamento da página, e como esperado, temos todo o conteúdo da página inserido no HTML, já que ele veio do servidor.

```
Line wrap ☐
1 <!DOCTYPE html>
2 <html lang="en"><head><meta charset="utf-8">
3   <meta name="viewport" content="width=device-width, initial-scale=1.0">
4   <base href="/">
5   <link rel="stylesheet" href="bootstrap/bootstrap.min.css">
6   <link rel="stylesheet" href="app.css">
7   <link rel="stylesheet" href="BlazorBookSsr.styles.css">
8   <link rel="icon" type="image/png" href="favicon.png">
9   <title>Home</title></head>
10
11 <body><div class="page" b-cfjni18mua><div class="sidebar" b-cfjni18mua><div class="top-row ps-3 navbar navbar-dark" b-m2vfk8trr><div cla
12
13 <input type="checkbox" title="Navigation menu" class="navbar-toggler" b-m2vfk8trr>
14
15 <div class="nav-scrollable" onclick="document.querySelector(&#x27;.navbar-toggler&#x27;).click()" b-m2vfk8trr><nav class="flex-column" b-
16 </a></div>
17
18   <div class="nav-item px-3" b-m2vfk8trr><a href="counter" class="nav-link"><span class="bi bi-plus-square-fill" aria-hidden="true"
19   </a></div>
20
21   <div class="nav-item px-3" b-m2vfk8trr><a href="weather" class="nav-link"><span class="bi bi-list-nested" aria-hidden="true" b-m
22   </a></div></nav></div></div>
23
24 <main b-cfjni18mua><div class="top-row px-4" b-cfjni18mua><a href="https://learn.microsoft.com/aspnet/core/" target="_blank" b-cfjni1
25
26   <article class="content px-4" b-cfjni18mua>
27
28   <h1>Hello, world!</h1>
29
30   Welcome to your new app.
31 </article></main></div>
32
33 <div id="blazor-error-ui" b-cfjni18mua>
34   An unhandled error has occurred.
35   <a href class="reload" b-cfjni18mua>Reload</a>
36   <a class="dismiss" b-cfjni18mua></a></div>
37
38 <script src="_framework/blazor.web.js"></script></body></html><!--Blazor-Server-Component-State:CfDj8GVYxZzGrB1Ohv4Q4oRSwYuFV7idQSG+il
```

O trecho que mais nos interessa (Além do cabeçalho) é o corpo da página, que no caso possui o seguinte HTML.

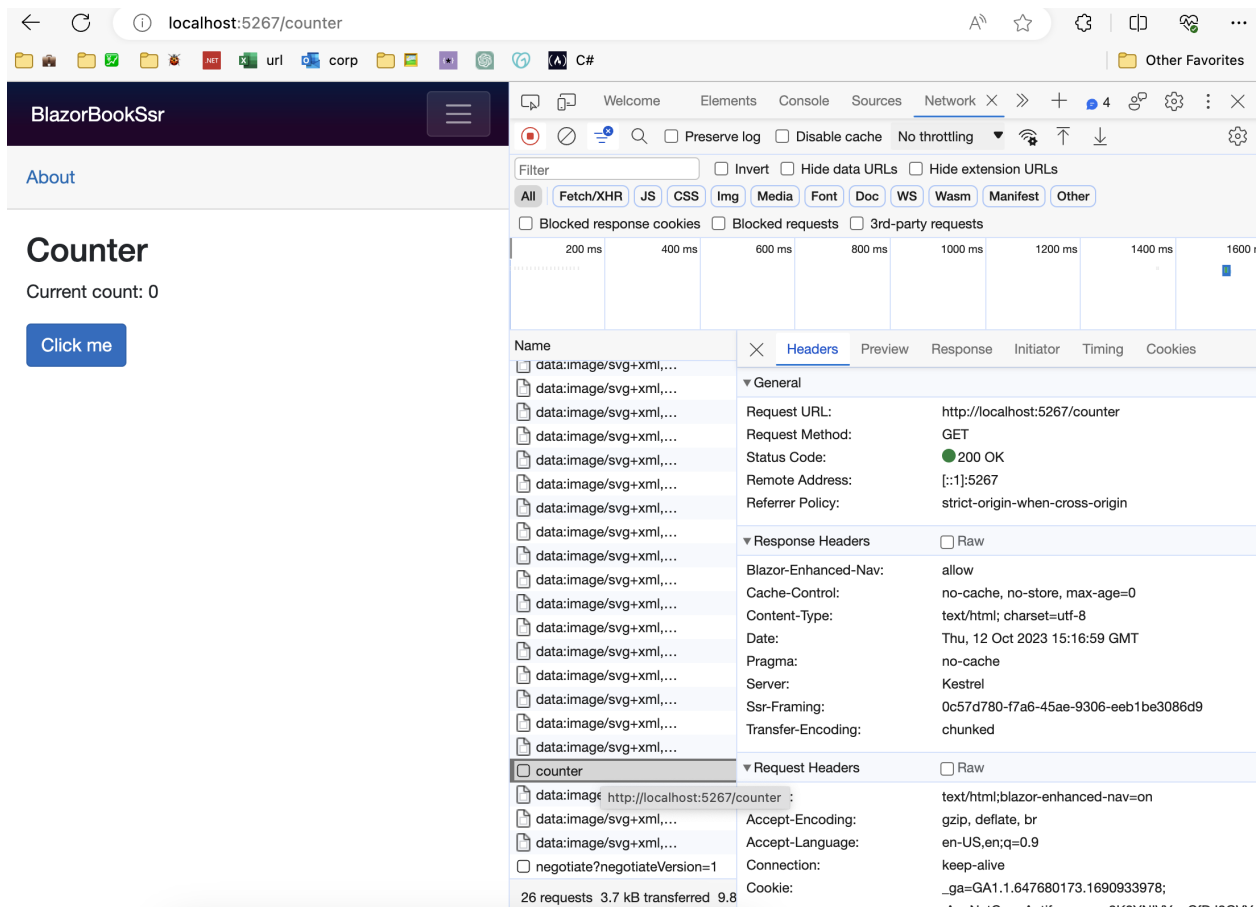
```
<h1>Hello, world!</h1>

Welcome to your new app.
</article></main></div>
```

Este conjunto (Cabeçalho e corpo) é que o facilita com que os mecanismos de busca (SEO - Search Engine Optimization) façam a indexação da nossa aplicação.

O problema nestes cenários é que para uma simples atualização, precisamos renderizar a página toda novamente (Calma que o Blazor não faz isto por padrão).

Vamos analisar o comportamento quando trocamos de página, por exemplo para o contador que vem no template padrão do Blazor.



Como podemos visualizar acima, somente um item chamado `counter` foi adicionado, o que significa que ele não baixou todo HTML e CSS novamente, apenas o “miolo” da página.

Este comportamento pode mudar de acordo com o modo de renderização (Render Mode) que escolhemos para página ou componente. Vamos tratar deles mais tarde.

De qualquer forma, surpreendentemente ainda assim temos um conteúdo do HTML atualizado, como mostrado na imagem abaixo.

```

1 <!DOCTYPE html>
2 <html lang="en"><head><meta charset="utf-8">
3   <meta name="viewport" content="width=device-width, initial-scale=1.0">
4   <base href="/">
5   <link rel="stylesheet" href="bootstrap/bootstrap.min.css">
6   <link rel="stylesheet" href="app.css">
7   <link rel="stylesheet" href="BlazorBookSsr.styles.css">
8   <link rel="icon" type="image/png" href="favicon.png">
9   <title>Counter</title></head>
10
11 <body><div class="page" b-cfjnl8mua><div class="sidebar" b-cfjnl8mua><div class="top-row ps-3 navbar navbar-dark" b-m2vfk8trr><div cl:
12
13 <input type="checkbox" title="Navigation menu" class="navbar-toggler" b-m2vfk8trr>
14
15 <div class="nav-scrollable" onclick="document.querySelector(&#x27;.navbar-toggler&#x27;).click()" b-m2vfk8trr><nav class="flex-column" }
16   </a></div>
17
18   <div class="nav-item px-3" b-m2vfk8trr><a href="counter" class="nav-link active" aria-current="page"><span class="bi bi-plus-sq
19     </a></div>
20
21   <div class="nav-item px-3" b-m2vfk8trr><a href="weather" class="nav-link"><span class="bi bi-list-nested" aria-hidden="true" b-
22     </a></div></nav></div></div>
23
24   <main b-cfjnl8mua><div class="top-row px-4" b-cfjnl8mua><a href="https://learn.microsoft.com/aspnet/core/" target="_blank" b-cfjni:
25     </a></div>
26   <article class="content px-4" b-cfjnl8mua><!--Blazor:{"type":"server","prerenderId":"854296b6197a4490b2661c463ae16e78","key":"7"
27
28 <h1>Counter</h1>
29
30 <p role="status">Current count: 0</p>
31
32 <button class="btn btn-primary">Click me</button><!--Blazor:{"prerenderId":"854296b6197a4490b2661c463ae16e78"}--></article></main></div>
33
34 <div id="blazor-error-ui" b-cfjnl8mua>
35   An unhandled error has occurred.
36   <a href class="reload" b-cfjnl8mua>Reload</a>
37   <a class="dismiss" b-cfjnl8mua></a></div>
38
39 <script src="framework/blazor.web.js"></script></body></html><!--Blazor-Server-Component-State:CfDJ8GVYxZZGrB1Ohv4Q4oRSwYsACUCr2jSp:

```

Novamente nosso foco é o cabeçalho e o corpo, e para nossa surpresa, o corpo da página no HTML também foi atualizado.

```

<h1>Counter</h1>

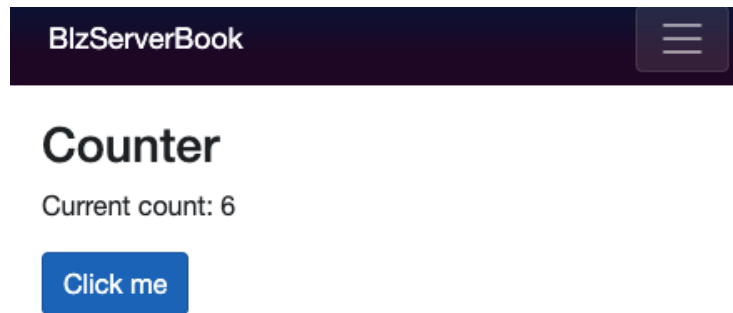
<p role="status">Current count: 0</p>

<button class="btn btn-primary">Click me</button><!--Blazor:{"prerenderId":"854296b6197a4490b2661c463ae16e78"}-

```

Isto se deve a um recurso do Blazor chamado Enhanced Navigation (Que também vamos explorar melhor adiante) e que nos permite transitar entre páginas de forma rápida mas com conteúdo do HTML sendo modificado.

Mas o que será que ocorre quando clicamos no botão “Click me” da página `Counter`, que tem que incrementar um número no texto acima dele?



Neste momento não vemos nenhum carregamento da página, nem um loading, nem nada, a página sequer pisca.

Isto ocorre por que uma vez a página carregada (No modelo Blazor Server), os componentes (Pedacos da página) são interativos.

Desta forma, quando clicamos no botão, ele simplesmente busca apenas os dados no servidor. Na verdade ele incrementa mais um no contador, e retorna o valor atualizado, mas tudo é feito do lado do servidor.

Este processo é feito por que as aplicações Blazor Server mantém uma conexão (Socket via SignalR) aberta com o servidor, para trafegar dados.

Então, se notarmos, embora a tela tenha se atualizado, o HTML por baixo dela não mudou.

```

1 <html lang= en >
2 <head>
3   <meta charset="utf-8" />
4   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
5   <base href="/" />
6   <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
7   <link href="css/site.css" rel="stylesheet" />
8   <link href="BlzServerBook.styles.css" rel="stylesheet" />
9   <link rel="icon" type="image/png" href="favicon.png"/>
10  <!--Blazor:{"sequence":0,"type":"server","prerenderId":"473d546430c049f
11 </head>
12 <body>
13   <!--Blazor:{"sequence":1,"type":"server","prerenderId":"896ed21729c044f
14
15   <div class="page" b-ikosa3nb0z><div class="sidebar" b-ikosa3nb0z><div clas
16     <button title="Navigation menu" class="navbar-toggler" b-vjmol2rww
17
18   <div class="collapse nav-scrollable" b-vjmol2rww><nav class="flex-column"
19     </a></div>
20     <div class="nav-item px-3" b-vjmol2rww><a href="counter" class="navi
21     </a></div>
22     <div class="nav-item px-3" b-vjmol2rww><a href="fetchdata" class="
23     </a></div></nav></div></div>
24
25     <main b-ikosa3nb0z><div class="top-row px-4" b-ikosa3nb0z><a href="http
26
27       <article class="content px-4" b-ikosa3nb0z>
28
29         <h1>Counter</h1>
30
31         <p role="status">Current count: 0</p>
32
33         <button class="btn btn-primary">Click me</button></article></main></div>
34       <!--Blazor:{"prerenderId":"896ed21729c044739ddfbefcddc7d428"}-->
35
36       <div id="blazor-error-ui">

```

Novamente focamos na sessão que nos interessa, e podemos notar que o `Current count:` ainda continua com o valor `0`.

```

<h1>Counter</h1>

<p role="status">Current count: 0</p>

<button class="btn btn-primary">Click me</button></article></main></div>
<!--Blazor:{"prerenderId":"896ed21729c044739ddfbefcddc7d428"}-->

```

Este é um comportamento normal em cenários SPA, onde o HTML é gerado dinamicamente, e todo conteúdo gerado desta forma não aparece no “Visualizar código fonte da página”, ou seja, afeta o SEO.

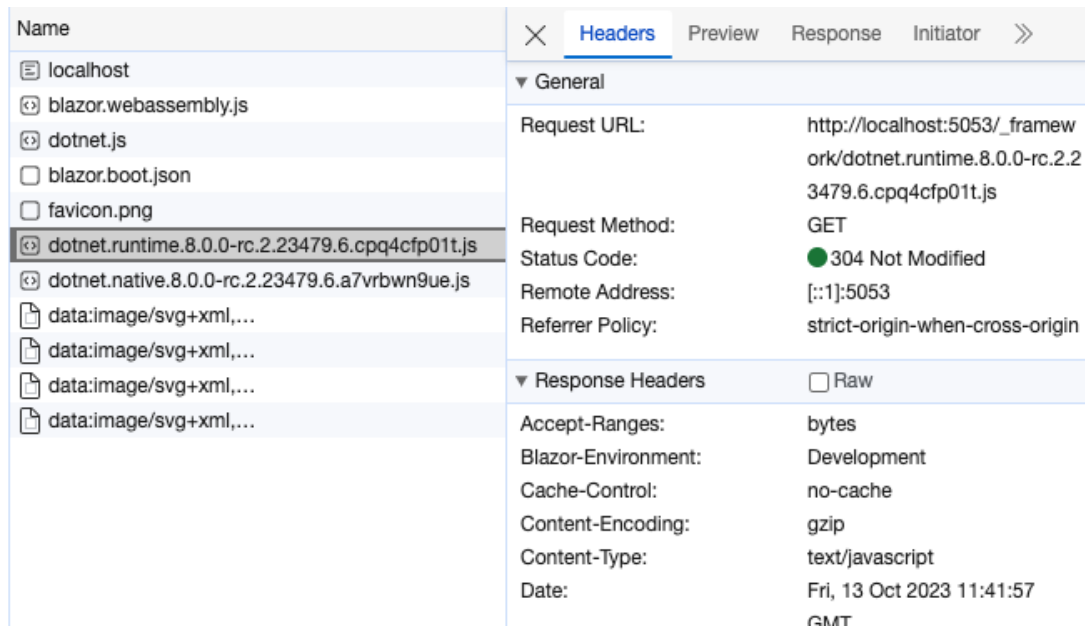
De qualquer forma, temos a opção de renderizar toda página novamente caso necessário, vai realmente do que você precisa naquele momento.

Blazor WASM

Na outra ponta temos o Blazor WASM ou Blazor WebAssembly, que como já comentamos é a possibilidade de rodar o .NET e C# direto no navegador.

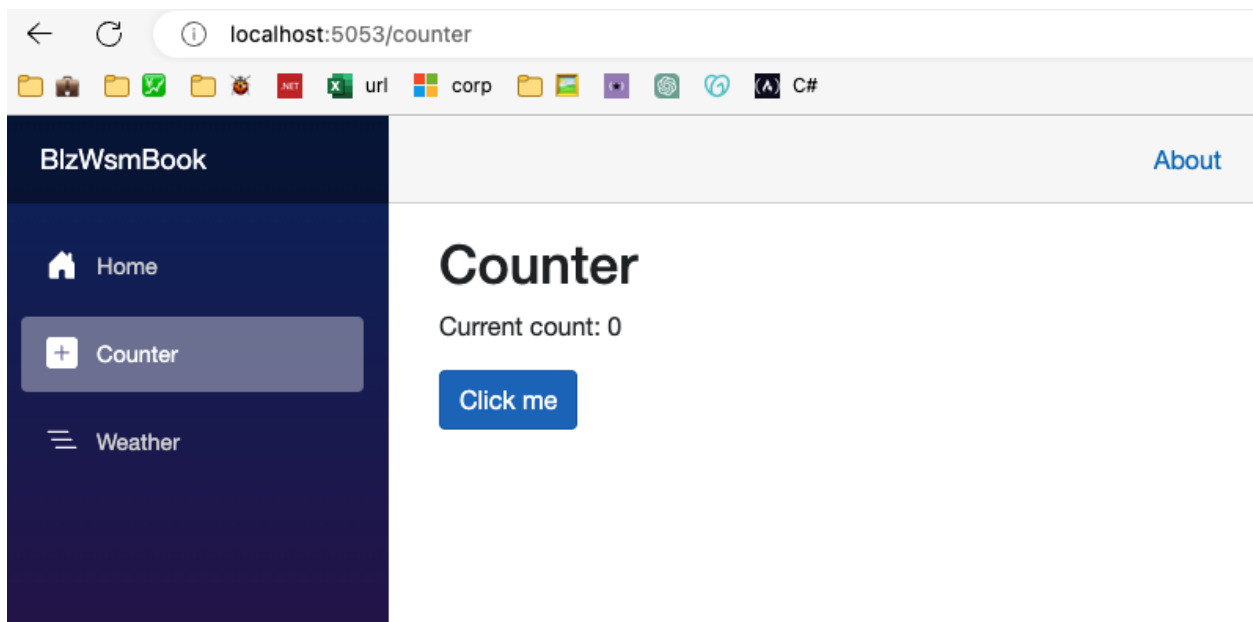
Não há como discutir a performance do WASM em relação ao Server, visto que todo conteúdo é processado localmente, na máquina do usuário.

O que precisamos discutir aqui são suas desvantagens em relação ao outro modelo, que começam com o tamanho do seu aplicativo.



Ao iniciar uma aplicação Blazor WASM, podemos notar a adição de um arquivo chamado `dotnet.runtime`, que conforme comentamos antes é o que faz o .NET rodar dentro do navegador.

Além disso, podemos notar que o conteúdo da página é igual ao que vimos anteriormente, no Blazor Server.



Porém, a maior diferença está por baixo dos panos, no HTML, onde temos o comportamento padrão de qualquer SPA.

Neste modelo, temos um HTML básico, feito para ser renderizado o mais rápido possível, e posteriormente vamos trocando o “miolo” dele sem atualizar a página.

```
line wrap
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="utf-8" />
6   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7   <title>BlzWsmBook</title>
8   <base href="/" />
9   <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
10  <link rel="stylesheet" href="css/app.css" />
11  <link rel="icon" type="image/png" href="favicon.png" />
12  <link href="BlzWsmBook.styles.css" rel="stylesheet" />
13 </head>
14
15 <body>
16   <div id="app">
17     <svg class="loading-progress">
18       <circle r="40%" cx="50%" cy="50%" />
19       <circle r="40%" cx="50%" cy="50%" />
20     </svg>
21     <div class="loading-progress-text"></div>
22   </div>
23
24   <div id="blazor-error-ui">
25     An unhandled error has occurred.
26     <a href="" class="reload">Reload</a>
27     <a class="dismiss"></a>
28   </div>
29   <script src="_framework/blazor.webassembly.js"></script>
30 </body>
31
32 </html>
33
```

Como podemos notar, o corpo da nossa página fica basicamente em branco, o que afeta o SEO, tendo como resultado apenas este HTML.

```
<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>
```

Não importa qual página acessarmos, nem quantos componentes renderizarmos na tela, no modelo WASM o HTML será sempre este.

Blazor Auto

Embora você possa escolher entre SSR e Server ou WASM, existe um novo modo no Blazor, chamado de automático.

Neste modo, o Blazor se encarrega de usar WASM ou Server para seus componentes ou páginas. Isto mesmo, ele pode ser definido por componente ou página utilizando o atributo `RenderMode`.

A mágica do RenderMode Auto

Quando definimos uma página ou componente com renderização automática, o Blazor sabe que componentes WebAssembly são mais rápidos por rodarem localmente.

Porém ele também sabe que o WASM demora mais para ser renderizado na primeira vez, devido a dependência do .NET.

Então ele faz o seguinte, renderiza este componente pela primeira vez no lado do servidor e deixa uma tarefa em segundo plano sendo executada.

Esta tarefa baixa o Runtime do .NET, que é a dependência mais pesada do Blazor WASM enquanto o usuário interage normalmente com o componente que foi servido.

A próxima vez que um componente ou página for renderizado, ele verifica se o Runtime já foi baixado, e então utiliza toda performance do WASM.

É uma saída muito inteligente, para manter performance e usabilidade nas aplicações de forma simples para nós (Só trocamos uma linha de código).

Capítulo 3 - Arquiteturas

Neste capítulo vamos entender quais as diferenças arquiteturais temos nos projetos Blazor e como podemos criar um novo projeto ou adicionar suporte ao Blazor em uma aplicação ASP.NET.

Instalação do .NET

Este livro se baseia na versão 8 do .NET, lançada em Novembro de 2023, e que pode ser baixada e instalada utilizando a URL abaixo:

Download .NET 8.0 (Linux, macOS, and Windows)

.NET 8.0 downloads for Linux, macOS, and Windows. .NET is a free, cross-platform, open-source developer platform for building many different types of applications.



<https://dotnet.microsoft.com/en-us/download/dotnet/8.0>



Versões anteriores e futuras podem apresentar diferenças, então recomendamos seguir sempre a mesma versão para um melhor entendimento.

Criando um projeto com Blazor Server

O primeiro projeto que vamos analisar é o Blazor Server, que conforme comentamos anteriormente trabalha apenas no lado do servidor.

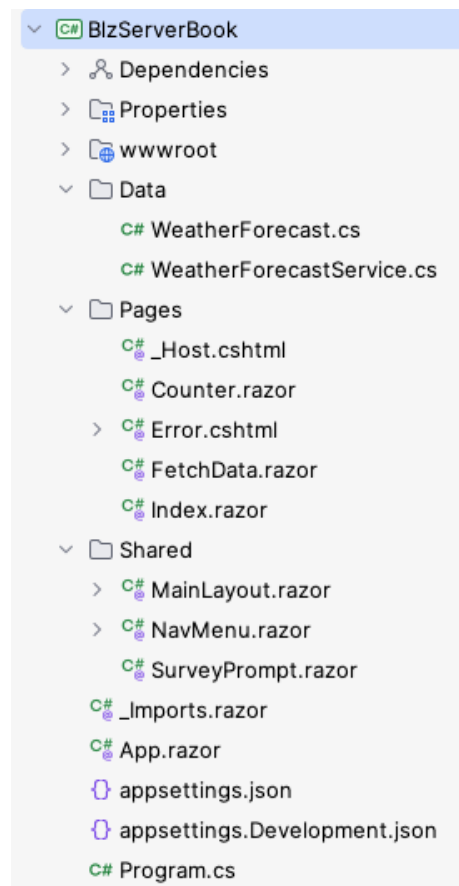
Então não se engane com a velocidade e usabilidade, todo seu código neste projeto está sendo executado no servidor.

Para isto, vamos executar o seguinte comando no terminal:

```
dotnet new blazorserver -o BlazorServerBookSample
```

O `-o` significa `output`, ou saída, e nada mais é do que o nome da nossa aplicação. O .NET vai criar uma pasta com este nome e colocar os arquivos lá dentro.

Como resultado, temos um projeto com a seguinte estrutura de pastas e arquivos.



wwwroot

Pasta utilizada para armazenar arquivos estáticos como JS, CSS, Imagens, entre outros.

Data

Pasta destinada ao acesso a dados ([Confira nossa carreira de acesso à dados](#))

Pages

Páginas da nossa aplicação

Shared

Componentes e Layouts ([Aprenda mais sobre Razor aqui](#))

_Imports.razor

Arquivo de importação de `namespaces`

App.razor

Componente inicial da nossa aplicação

Program.cs

Arquivo de execução do programa (Aplicação)

Criando um projeto com Blazor Wasm

Ao contrário do Blazor Server, o WASM é executado todo do lado do cliente (Navegador), então temos uma performance maior, mas não temos a segurança do servidor.

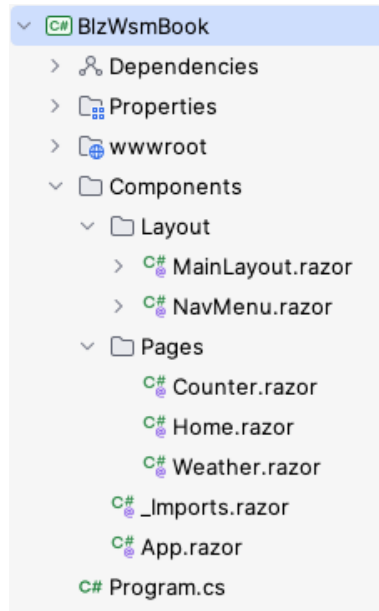
Isto significa que você não deve armazenar configurações sensíveis como **Connection Strings** neste projeto.

Em adicional, também devemos notar a necessidade de uma API para comunicação, visto que não teremos uma conexão direta com o banco no WASM.

Para criar um novo projeto Blazor WASM vamos executar o seguinte comando:

```
dotnet new blazorwasm -o BlazorWasmBookSample
```

Como podemos notar, a estrutura de uma aplicação WASM é diferente da Server, contendo menos pastas e arquivos e ainda sub-organizada em componentes.



wwwroot

Pasta utilizada para armazenar arquivos estáticos como JS, CSS, Imagens, entre outros.

Components

Componentes, Layouts e Páginas

Layout

Layouts

Pages

Páginas

_Imports.razor

_App.razor

_Program.cs

_Imports.razor

Arquivo de importação de `namespaces`

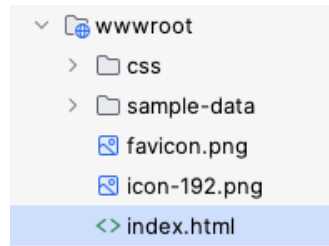
App.razor

Componente inicial da nossa aplicação

Program.cs

Arquivo de execução do programa (Aplicação)

Embora alguns arquivos e pastas sejam similares, temos algo bem diferente no WASM, que é o arquivo `index.html`. Este arquivo fica dentro da pasta `wwwroot`, que como vimos, é utilizada para servir arquivos estáticos.



O que acontece é que no Blazor Server as páginas são geradas no servidor, então, elas são dinâmicas, ou seja, a cada requisição geramos um HTML (Ou parte dele).

No Blazor WASM, o resultado final da nossa aplicação é apenas HTML, CSS e JS, ou seja, apenas arquivos estáticos.

Enquanto no Blazor Server você precisa de um servidor que suporte .NET para executar sua aplicação, no WASM isto não é necessário.

Você pode hospedar sua aplicação Blazor WASM em um servidor extremamente simples, como GitHub Pages por exemplo.

Criando um projeto com Blazor

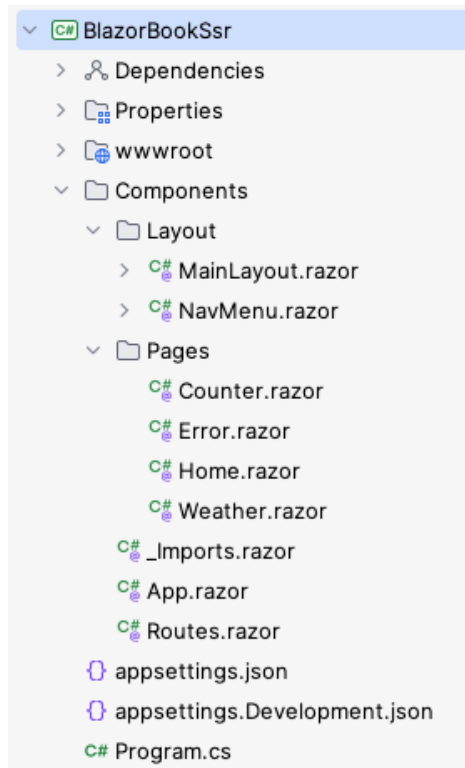
Parece confuso criar um projeto com Blazor, já que acabamos de criar dois (Server e WASM), mas o fato é que no .NET 8 temos uma opção a mais.

Lembra que comentamos sobre o modo “automático” do Blazor, pois é, isto mesmo que temos neste template, uma mescla entre o Server e WASM.

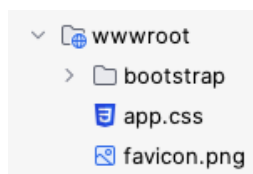
Dito isto, ao invés de especificarmos `blazorwasm` ou `blazorserver`, podemos usar apenas `blazor` e ter suporte a ambos.

```
dotnet new blazor -o BlazorBookSample
```

Como resultado deste comando, temos um projeto muito parecido com o gerado no Blazor WASM, mas com um detalhe importante.



Não temos o arquivo `index.html` na pasta **wwwroot** como anteriormente, já que para dar suporte ao Blazor Server, precisamos obrigatoriamente do .NET instalado no servidor.



Executando os demos

Uma das grandes vantagens de utilizar o .NET é que muitos dos seus conceitos, como injeção de dependência, são utilizados em todos os projetos.

Aliás, temos um [curso completo só sobre injeção de dependência](#) no balta.io, que vai te ajudar a entender conceitos como DI, DIP, IoC e Service Locator.

Além destas similaridades no código, os projetos escritos com .NET também são executados da mesma forma, bastando utilizar o comando `dotnet run` para isto.

```
cd NOME_DA_PASTA
dotnet run
```

Embora possamos executar a aplicação com o comando `dotnet run`, tem outro comando que nos ajuda muito, o `dotnet watch run`.


```
dotnet watch run
```

Quando adicionamos o `watch` ao comando, que significa observar em inglês, ele fica olhando nosso código e sempre que alteramos algo, ele reinicia a aplicação automaticamente.

Uma mão na roda quando estamos trabalhando com itens visuais, como é o caso do Blazor e precisamos checar a tela a todo momento.

Capítulo 4 - Startup

Ao decorrer deste livro vamos focar no Blazor (União de todos os Frameworks), já que ele tem tanto a parte WASM quanto a SSR.

Desta forma, vamos inspecionar o que faz o Blazor funcionar, inclusive em conjunto com o próprio ASP.NET de forma simples.

Criando o projeto

Nosso primeiro passo é criar um novo projeto com o Blazor, e como vimos anteriormente, faremos isto com o comando `dotnet new` seguido do nome do template, que no caso é `blazor`.

```
dotnet new blazor -o BlazorBook
```

O comando irá criar uma nova aplicação chamada **BlazorBook** com tudo o que precisamos para nosso projeto inicialmente.

Entendendo os pacotes

Investigando melhor o conteúdo gerado, podemos analisar o `BlazorBook.csproj`, arquivo principal da aplicação que define entre outros itens a versão do .NET e tipo de projeto.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Tanto o Blazor quanto o ASP.NET estão contidos no pacote **Microsoft.NET.Sdk.Web**, como mostrado na primeira linha do arquivo.

Em adicional, temos a atribuição do `TargetFramework` como `net8.0`, algo necessário para nosso projeto, visto que essa união do Blazor e ASP.NET só está presente a partir desta versão.

Razor e Blazor, qual a diferença?

Razor é o motor de renderização por trás do ASP.NET e Blazor, ele é o responsável por permitir a interpolação entre HTML e C#.

Razor Pages é a arquitetura utilizada para criar páginas Web com C# e HTML usando o ASP.NET.

Tanto o Razor Pages, MVC e Blazor, fazem uso do Razor para desenhar suas telas.

Startup.cs

Uma característica comum do .NET é compartilhar muita coisa entre os projetos, o que significa que sabendo como o Startup de uma aplicação funciona, você consegue evoluir em diversas frentes.

No caso, sempre teremos um builder (Construtor) e um app (Instância da aplicação), e o que varia é apenas como criamos esta instância.

No caso do Blazor, esta inicialização é exatamente a mesma do ASP.NET, dada pelo objeto `WebApplication`.

```
var builder = WebApplication.CreateBuilder(args);
```

Com o construtor instanciado, podemos adicionar os serviços que iremos utilizar, e aqui começa a magia da união do Blazor com o ASP.NET.

```
builder.Services
    .AddRazorComponents()
    .AddInteractiveServerComponents();
```

Como podemos notar, temos tanto a adição dos Razor Components (`AddRazorComponents`) quanto dos componentes interativos (`AddInteractiveServerComponents`).

Enquanto os Razor Components são componentes mais simples, sempre renderizados no formato SSR, os componentes interativos agem de forma diferente.

Embora eles sejam servidos via SSR assim como os Razor Components, eles tem a habilidade de serem interativos.

Isto significa que cada interação com o componente ou página não precisa renderizar todo HTML novamente, ele busca apenas os dados e atualiza só o que foi alterado no componente.

Por fim não podemos esquecer dos mapeamentos, eles serão responsáveis pelas rotas até as páginas e componentes que criarmos.

```
app.MapRazorComponents<App>( )
    .AddInteractiveServerRenderMode();
```

Existem mais coisas no arquivo `Startup.cs`, mas elas dizem respeito ao ASP.NET de forma geral e foram amplamente exploradas no CURSO GRATUITO DE RAZOR PAGES.

App.razor

Movendo adiante, temos o `App.razor`, arquivo que é a união do `App.razor` do **Blazor Server** com o `Index.html` do **Blazor WASM**.

Exceto a parte em que referenciamos o `BlazorBook.styles.css` que será explicado adiante aqui neste livro, nosso foco aqui fica em três partes principais, o `HeadOutlet`, `Routes` e `Script`.

```
<!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="bootstrap/bootstrap.min.css" />
  <link rel="stylesheet" href="app.css" />
  <link rel="stylesheet" href="BlazorBook.styles.css" />
  <link rel="icon" type="image/png" href="favicon.png" />
  <HeadOutlet />
</head>

<body>
  <Routes />
  <script src="_framework/blazor.web.js"></script>
</body>

</html>
```

Como podemos notar, este é um arquivo HTML, que não se distingue muito de qualquer outra página que usa a linguagem. O mais importante aqui é se atentar aos três itens abaixo.

Head Outlet

Sempre que trocamos de página, podemos alterar o corpo e cabeçalho da mesma, e isto é feito através do `Routes` e `HeadOutlet` respectivamente.

Em resumo, vamos encontrar itens especiais como o `PageTitle`, que são automaticamente alocados dentro do `HeadOutlet`.

```
<PageTitle>Home</PageTitle>
```

Além disso, podemos utilizar o `HeadContent` para colocar algum conteúdo dentro do espaço reservado pelo `HeadOutlet`.

```
<HeadContent>
  <meta name="META_NAME" content="META_VALUE">
</HeadContent>
```

Embora pareça uma funcionalidade simples, adicionar itens ao cabeçalho da página é algo extremamente importante para o SEO.

É importante lembrar que dependendo do modo de renderização da página (Render Mode) estas alterações podem não refletir diretamente no HTML, conforme mostramos no começo do livro, na sessão SPA.

Routes

A sessão `Routes` define literalmente qual página será renderizada aqui. O Blazor irá buscar em todos os arquivos `.razor` o atributo `@page` e montará uma tabela de rotas.

Talvez a maior diferença entre um componente e uma página seja apenas a decoração no começo do arquivo.

```
@page "/teste"
<h1>Eu sou uma página</h1>
```

Enquanto uma página sempre começará com o atributo `@page`, os componentes simplesmente ignoram esta linha.

```
<h1>Eu sou um componente</h1>
```

O que acontece é que sempre que digitamos no navegador uma URL como `https://localhost:1234/teste`, o Blazor buscará nos arquivos `.razor` que contém o atributo `@page` com o valor `"/teste"`.

Uma vez encontrado, ele irá renderizar a página substituindo o elemento `<Routes/>` pelo conteúdo da mesma.

blazor.web.js

Um ponto importante para notarmos no uso do Blazor é que muito do código executado (Byte Code) é feito por um arquivo JavaScript, visto que o navegador só interpreta HTML, CSS e JS.

Por isto há a necessidade da inclusão do `blazor.web.js` em nossa página principal, sempre que precisarmos executar algo no lado do cliente, assim como fazemos com qualquer outro JavaScript.

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

Outro fato interessante é que este arquivo não existe, ele é criado apenas quando executamos ou compilamos a aplicação.

Por padrão, o ASP.NET mantém arquivos estáticos na pasta `wwwroot`, e se notarmos, não há uma pasta `_framework` lá dentro, nem mesmo o arquivo `blazor.web.js`.

Este arquivo será gerado caso haja utilização de algum componente do Blazor WASM ou interoperabilidade com o JavaScript.

Routes.razor

O esquema de rotas no Blazor é relativamente simples e lembra muito o React (O que é bom ao meu ver).

Vamos explorar mais os esquemas de rotas adiante neste livro, mas no momento, precisamos saber onde tudo ocorre, que é no arquivo `Routes.razor`.

```
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData" DefaultLayout="@typeof(Layout.MainLayout)" />
    <FocusOnNavigate RouteData="@routeData" Selector="h1" />
  </Found>
</Router>
```

O primeiro elemento que encontramos é o `Router`, elemento principal, que tem como parâmetro o `AppAssembly`.

Ao utilizar o valor `@typeof(Program).Assembly` no `AppAssembly`, estamos dizendo ao Blazor para pesquisar arquivos `.razor` que contenham o atributo `@page` em todo o programa principal.

Dada uma rota, temos dois caminhos (Na verdade tem mais, mas inicialmente vamos focar no básico) que são **Found** e **NotFound**.

```
<Found Context="routeData">
  <RouteView RouteData="@routeData" DefaultLayout="@typeof(Layout.MainLayout)" />
  <FocusOnNavigate RouteData="@routeData" Selector="h1" />
</Found>
```

Sempre que uma rota é encontrada (**Found**), ela passa pelo elemento acima, que automaticamente injeta um objeto chamado `routeData` no contexto atual (**Context**).

Este objeto chamado `routeData` contém informações da rota, como parâmetros, segmentos, âncoras, dentre outros.

Dentro do elemento `Found` temos o `RouteView`, que é responsável por renderizar a página ou componente no elemento `Routes` que vimos anteriormente.

```
<RouteView RouteData="@routeData" DefaultLayout="@typeof(Layout.MainLayout)" />
```

Este elemento por sua vez, também repassa o objeto `routeData` adiante (Para seus filhos — páginas e componentes), além de ter um atributo adicional.

Assim como no **Razor Pages**, no Blazor podemos trabalhar com **Layouts**, e uma ótima opção é definir um layout padrão para as páginas.

Neste caso, utilizamos o atributo `DefaultLayout` para definir que o layout padrão de todas as páginas será o `MainLayout.razor`. Veremos mais sobre layouts adiante.

Por fim, podemos definir uma sessão para caso a rota não seja encontrada, utilizando o atributo `NotFound`.

Por padrão, o arquivo `Routes.razor` não traz esta implementação, mas podemos resolver isto de forma simples e fácil.

```
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
```

```

        <RouteView RouteData="@routeData" DefaultLayout="@typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
    <NotFound>
        <h1>Página não encontrada</h1>
    </NotFound>
</Router>

```

Desta forma, caso a página que buscamos não seja encontrada, o texto “*Página não encontrada*” será exibido na tela.

_Imports.razor

Assim como no Razor Pages, o arquivo `_Imports.razor` facilita o uso de *namespaces* que precisamos com frequência.

Então se estamos trabalhando com Entity Framework por exemplo e não queremos a todo momento ficar importando os namespaces em todas as páginas, podemos importar ele globalmente no `_Imports.razor`.

Adicionando suporte a Blazor no Razor Pages

Embora o Blazor dê suporte a SSR, como veremos a seguir neste livro, você pode adicionar suporte a componentes e páginas do Blazor em aplicações ASP.NET Razor Pages.

Dado que uma aplicação Razor Pages começa exatamente como uma aplicação Blazor, com o `WebApplication.CreateBuilder`.

```
var builder = WebApplication.CreateBuilder(args);
```

Desta forma, nada impede que haja a adição do suporte a Razor Pages no projeto, que pode ser feito pela adição do serviço abaixo.

```
builder.Services.AddRazorPages();
```

Também precisamos mapear do Razor Pages, que funcionam similar ao Blazor, com exceção do arquivo `Routes.razor` que vimos anteriormente.

```
app.MapRazorPages();
```

Vale lembrar que o esquema de pastas do Razor Pages é diferente do Blazor e precisa ser mantido. Pastas como `Pages` e `Shared`, obrigatórias para o funcionamento do Razor Pages, precisam ser criadas.

Adicionando suporte a Razor Pages no Blazor

Da mesma forma, podemos adicionar suporte ao Blazor em aplicações criadas com Razor Pages, basta adicionar os serviços.

```
builder.Services
    .AddRazorComponents()
    .AddInteractiveServerComponents();
```

E posteriormente configurar o esquema de rotas para dizer como encontrar páginas e componentes do Blazor.

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

Assim como no tópico anterior, precisamos adicionar a estrutura de pastas (Components, Pages Layout) necessária para que o Blazor possa ser executado.

Blazor e Razor Pages juntos

Com a chegada do SSR no Blazor a partir da versão 8 do .NET, o cenário fica mais favorável a criação de aplicações com o Framework.

Se temos a possibilidade de ter componentes interativos, páginas renderizadas no servidor e ainda suporte a Web Assembly, tudo em um único projeto, o uso do Razor Pages fica mais específico.

Claro que cada cenário é um caso a parte, mas agora o Blazor passa a ser uma primeira opção para Frontend com .NET, deixando Razor Pages para cenários que realmente tenham uma demanda completa de SSR.

Capítulo 5 - Páginas e Components

O coração do Blazor, assim como do Razor Pages, são páginas e componentes. Em ambos os casos o Razor é o motor de renderização utilizado.

Com ele, conseguimos interpolar C# e código HTML de forma simples. Podemos executar literalmente os dois juntos, algo como isto:

```
@ForEach(var item in items)
{
    <p>@item</p>
}
```

Caso queira aprender mais sobre Razor, temos um curso **GRATUITO** sobre ele no site.

Uma visão geral sobre o ASP.NET Razor Pages

Descubra o que é como funciona um dos Frameworks Web mais maduros e utilizados do mercado.



<https://balta.io/cursos/uma-visao-geral-do-aspnet-razor-pages>

#CURSO

**Uma visão geral do
ASP.NET Razor Pages**

GRATUITO E COM CERTIFICADO



O que é uma página?

As páginas, componentes e layouts no Blazor são bem parecidos, com exceção de alguns itens específicos de cada um.

No caso da página, podemos identificá-las pelo atributo `@page` já na primeira linha do arquivo, conforme podemos observar abaixo.

```
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.
```

O atributo `@page` sempre vem acompanhado de uma `string` que determina a rota até aquela página e que deve iniciar sempre com `" / "`.

Como esta string se tornará uma URL, ela não pode conter espaços ou caracteres especiais (Exceto os utilizados em parâmetros ou URLs).

```
@page "banana" // ❌ não começa com "/"
@page "/funções" // ⚠️ até funciona, mas evite ç, acentos e etc
@page "/minha agenda" // ❌ não deve conter espaços
@page "/produtos" // ✅ OK
@page "/categorias/docs" // ✅ OK
```

Estrutura de uma página

As páginas são segmentadas em três partes principais, o cabeçalho, que é onde importamos atributos, dependências e outras importações.

O corpo que é onde definimos o código HTML da página, que pode ser interpolado com C#, visto que estamos utilizando o Razor.

E o código em si que faz a interação com o HTML e pode conter chamada a funções, loops, condicionais e o que mais o C# nos proporcionar.

```
// Atributos, injeções de dependência, importações
@page "/"

// Corpo da página (HTML)
<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

// Código que fará interação com HTML da página
@code {
```



```
// Do some stuff  
}
```

O que é um componente?

Um componente nada mais é do que um pedaço menor de uma página, como se fosse um recorte dela.

Se precisamos repetir o mesmo elemento visual em diversas páginas por exemplo, podemos criar um componente e reusar ele.

A única diferença estrutural entre um componente e uma página é que os componentes não possuem o atributo `@page`, afinal, não temos uma rota até eles.

Podemos criar um arquivo na pasta `Components` e chamá-lo de `Greeting.razor`, com o seguinte código.

```
<h1>Hello, I`m a component</h1>
```

Note que não temos atributos e nem código, propositalmente, visto que este é nosso primeiro e mais simples componente.

Posteriormente, podemos utilizar este componente na página `Home.razor` por exemplo, apenas fazendo uma chamada a ele.

```
@page "/"  
  
<PageTitle>Home</PageTitle>  
  
<h1>Hello, world!</h1>  
  
Welcome to your new app.  
<Greeting></Greeting>
```

Estrutura de uma componente

A estrutura de um componente é ainda mais simples do que a apresentada nas páginas, visto que eles não tem um cabeçalho.

```
// Corpo do componente  
<h1>Hello, I`m a component</h1>  
  
// Código que fará interação com HTML  
@code {  
    // Do some stuff  
}
```

O que é um Layout?

Se um componente é um “pedaço da tela”, que pode compor uma página, que por sua vez também representa algo visual, os layouts são o que chamamos de Frames ou molduras.

Imagina que em sua aplicação você tem um menu superior e precisa utilizá-lo em todas as páginas. Por mais que os componentes sejam simples de usar, seria entediante repetir o processo em cada página da aplicação.

Os layouts nos ajudam exatamente nisso, e quando digo “os layouts” significa que podemos ter mais de um.

Vamos analisar o layout padrão, que já vem junto com a aplicação quando criamos um novo projeto com Blazor.

```
@inherits LayoutComponentBase

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

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

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

<div id="blazor-error-ui">
  An unhandled error has occurred.
  <a href="" class="reload">Reload</a>
  <a class="dismiss">X</a>
</div>
```

Estrutura de uma layout

A primeira mudança que encontramos e que define um layout é o uso do atributo `@inherits LayoutComponentBase`, que define esta página sendo um layout.

Depois disso, pouco importa o HTML contido na página, ele fica a seu critério desenhar a tela do jeito que mais te agrada.

Porém, para o Layout saber onde as páginas serão renderizadas, ele precisa de um elemento especial chamado `@Body`.

Durante a execução do código, o Razor pegará todo conteúdo da páginas (E consequentemente dos componentes dentro dela) e substituirá o `@Body` por eles.

Outra sessão importante nos layouts é a `div` com identificador `blazor-error-ui`. Se você já teve oportunidade de trabalhar com Blazor Web Assembly, já deve ter notado esse código lá.

```
<div id="blazor-error-ui">
  An unhandled error has occurred.
  <a href="" class="reload">Reload</a>
  <a class="dismiss">X</a>
</div>
```

Sempre que usamos componentes do Blazor WASM, seus erros são exibidos utilizando esta `div` acima, é muito importante manter ela.

Interação entre elementos da página

Criar componentes, páginas e layouts é algo relativamente simples, basta seguir as estruturas mencionadas anteriormente e pronto.

Indo mais além, exibir informações na tela, vindas de um banco de dados ou qualquer outro lugar, também não é um grande desafio.

A maior dificuldade está na interação com o usuário, em manter um estado, em esconder ou exibir elementos, em mudá-los.

Binding e Two-way binding

A forma mais simples que temos de fazer interações (Exibir no caso), é através de uma variável, como mostrado abaixo.

```
<p>Contador: @currentCount</p>

@code {
    private int currentCount = 0;
}
```

Neste caso, temos uma variável chamada `currentCount` que pode ser exibida em qualquer lugar da tela através do `@currentCount`.

Este processo é chamado de **Binding**, ou vínculo. Em resumo, se temos uma variável, propriedade, método, podemos interagir com ele.

Além disso, sempre que fizer uma alteração em uma variável, ela reflete nos elementos da tela, e sempre que alteramos um elemento na tela, ele também reflete na variável.

```
@rendermode InteractiveServer

<input type="text" @bind-value="_text"/>
<input type="text" @bind-value="_text"/>
<p>@_text</p>

@code {
    private string _text = "Olá mundo!";
}
```

Como resultado do código acima, temos o seguinte pedaço de tela (Página ou componente) que renderiza dois campos e um texto, vinculando os três.

Hello, world!

Welcome to your new app. Olá mundo! Olá mundo!
Olá mundo!

Sempre que alteramos o texto do campo, ao perder o foco, o outro campo e o texto abaixo deles também é alterado.

Hello, world!

Welcome to your new app. Hello world Hello world
Hello world

Essa magia é o resultado da junção do `@bind-value` com o **Render Mode**, sendo o primeiro para vincular um campo a uma variável ou propriedade e o segundo para dar interatividade a eles.

A partir da versão 8 do .NET, se você utilizar apenas o `@bind-value`, notará que nada acontece quando mudamos o texto no campo.

Isto ocorre por que mesmo os campos estando ligados (Binding), qualquer página ou componente é servido estático, ou seja, sem comportamento.

O que o trecho `@rendermode InteractiveServer` faz é literalmente dar vida a isto tudo, ele permite que a tela fique conectada ao servidor.

Desta forma, quando alteramos um campo, um evento é enviado para o servidor, que entende essa mensagem e manda uma atualização para tela.

A página ou componente por sua vez se atualiza, sem a necessidade de recarregar o aplicativo ou página, ele só altera o que realmente precisa.

Separando o código da página

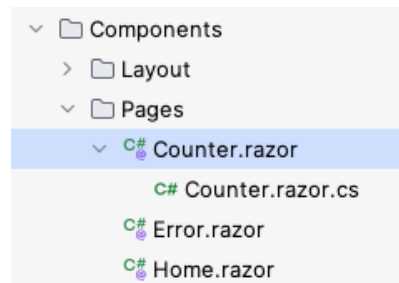
Como vimos nos tópicos anteriores, o código (Comportamento) das páginas e comportamentos fica no final do arquivo. Mas e se meu código crescer?

Realmente, em algumas situações fica complicado gerenciar todo código na página. Para ser sincero fica até confuso ter elementos visuais e código (Comportamento) juntos.

Neste caso, podemos criar um arquivo separado para o código, seguindo a convenção de manter o mesmo nome do arquivo, adicionando o sufixo `.cs`.

```
// Exemplo:  
-> Pages/Counter.razor  
-> Pages/Counter.razor.cs
```

Dependendo da IDE ou editor que você utiliza, ele já entenderá que se trata de um conjunto de itens e fará o agrupamento.



Agora temos dois arquivos, `Counter.razor` e `Counter.razor.cs`, sendo o primeiro para representação visual e o segundo a parte comportamental.

Tudo o que precisamos fazer é garantir que nossa classe herde de `ComponentBase` e pronto, já podemos mover o código do componente para cá.

```
using Microsoft.AspNetCore.Components;  
  
namespace BlazorBook.Components.Pages;  
  
public class CounterCode : ComponentBase  
{  
    protected int CurrentCount = 0;  
  
    protected void IncrementCount()  
    {  
        CurrentCount++;  
    }  
}
```

Quando optamos por utilizar essa abordagem, precisamos explicitamente utilizar `protected` ou `public` como modificar para as propriedades e métodos.

Exceto pela remoção do código da página, o resto do conteúdo continua o mesmo, com adição do `@inherits` `CounterCode`.

```
@page "/counter"  
@inherits CounterCode  
@rendermode InteractiveServer  
  
<PageTitle>Counter</PageTitle>  
  
<h1>Counter</h1>
```

```
<p role="status">Current count: @CurrentCount</p>

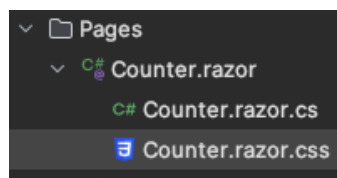
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

Esta herança é necessária para termos acesso ao código, que agora ficou em uma classe e arquivo separados.

Separando os estilos

Da mesma forma que fizemos anteriormente com o código, podemos aplicar estilos somente a componentes, páginas e layouts que desejarmos.

O processo é literalmente o mesmo, podemos criar um arquivo chamado `Counter.razor.css` e estilizar o que quisermos nele.

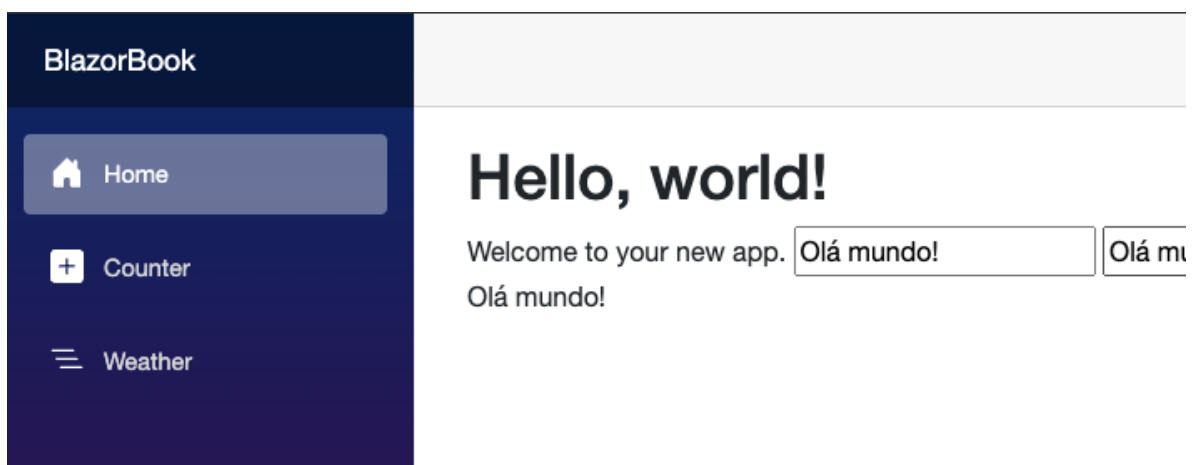


Quando a aplicação for executada, ela criará um CSS dinâmico, que só será aplicado neste componente.

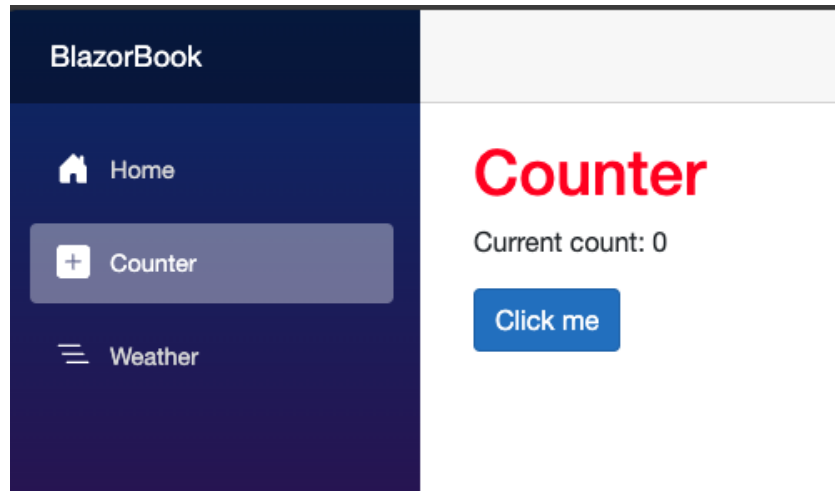
Para exemplificar, vamos popular o `Counter.razor.css` com o seguinte conteúdo, que vai deixar os títulos da página ou componente em vermelho.

```
h1 {
    color: #FF0000;
}
```

Ao executarmos a aplicação, podemos notar que o título **“Hello, world!”** na home se manteve na cor preta.



Quando movemos para página **Counter**, notamos que o título dela foi alterado para vermelho, conforme configuramos no `Counter.razor.css`.



Isto significa que o CSS que criamos só afetou a página Counter, deixando as outras páginas intactas.

Este é um recurso extremamente útil quando falamos em componentização. Podemos ter nossos estilos independentes com muito mais facilidade.

Entendendo o CSS dinâmico

Toda essa magia começa lá no arquivo `App.razor`, dentro do cabeçalho da página quando referenciamos os arquivos CSS.

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

Como comentado nos tópicos anteriores, existe um arquivo CSS que não existe na pasta `wwwroot`, que seria gerado junto com o build da aplicação.

```
<link rel="stylesheet" href="BlazorBook.styles.css" />
```

Este arquivo contém todos os estilos que ficam aninhados nos arquivos Razor (Os arquivos `.razor.css`)

Inspecionando a página Counter, podemos ver que um atributo chamado `b-f0mv87vb1r` foi adicionado ao cabeçalho.

```
<h1 b-f0mv87vb1r="">Counter</h1>
```

Este atributo, de nome aleatório é gerado pelo Razor justamente para aplicar um CSS específico para o componente ou página.

Desta forma, quando o arquivo `BlazorBook.styles.css` é gerado, o Razor agrupa os CSS aninhados utilizando este código.

```
/* _content/BlazorBook/Components/Pages/Counter.razor.rz.scoped.css */
h1[b-f0mv87vb1r] {
    color: #FF0000;
}
```

É isto que faz com que o CSS gerado dinamicamente pelo Razor afete somente um componente, Layout ou página.

Parâmetros

Existem várias formas de passar informações entre páginas e componentes, aqui vamos focar na parte de componentes, visto que adiante, na sessão de rotas, falaremos sobre parâmetros e páginas.

Um parâmetro nada mais é do que uma propriedade, decorada com o atributo `Parameter`, conforme mostrado no código abaixo.

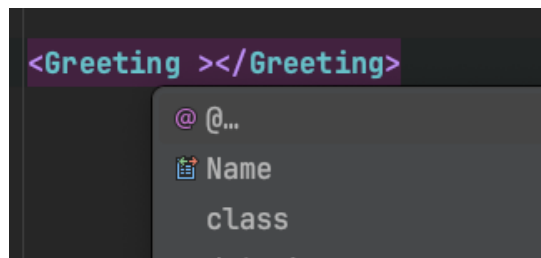
```
<h1>Hello, @Name</h1>

@code {

    [Parameter]
    public string Name { get; set; } = string.Empty;

}
```

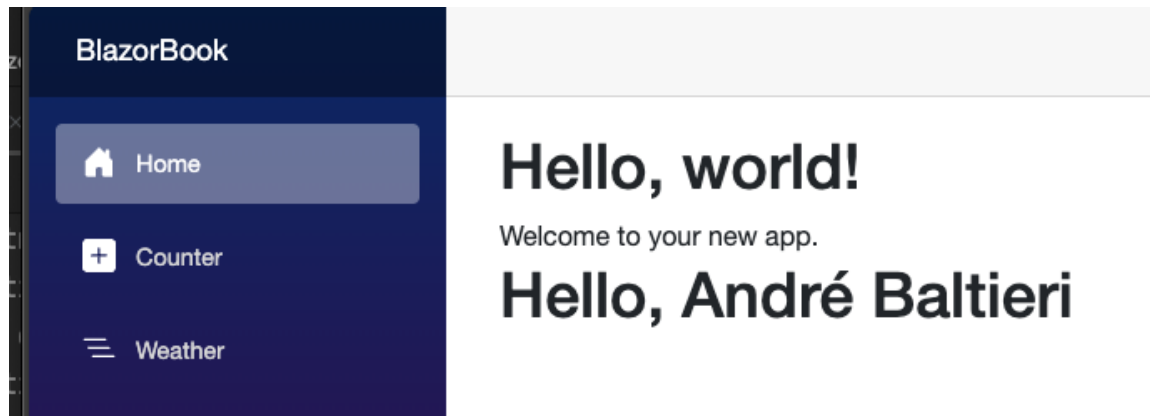
Não é possível utilizar o atributo `Parameter` em variáveis, somente em propriedades e com esta simples adição, sua IDE já deve identificar que o componente pode receber um parâmetro.



Desta forma, podemos renderizar nosso componente, passando alguma informação para ele, neste caso uma simples `string`.

```
<Greeting Name="André Baltieri"></Greeting>
```


Como resultado, temos o valor informado durante a importação do componente sendo exibido na tela como mostrado abaixo.



Você pode passar múltiplos parâmetros para um componente, ou mesmo passar um objeto complexo ao invés de um tipo primitivo.

Capítulo 6 - Render Modes

Se você me perguntasse sobre o Blazor há alguns anos atrás eu diria “Legal, mas ainda prefiro Angular”, simplesmente por que trocar um Framework consolidado como Angular por Blazor não me parecia uma boa ideia.

O tempo passou e o Blazor amadureceu, mas mesmo assim, minha recomendação era “Se você já tem código em C#, Blazor é uma ótima opção”.

Isto por que mesmo trabalhando com modelo SPA, o Blazor, assim como Angular, React e Vue, não tem um bom suporte a SSR, que sustenta o SEO.

No caso do balta por exemplo, o Blazor seria incrível na sessão de compras, para trazer uma maior interatividade para o usuário.

Avançando para o Player (Área que o aluno assiste os cursos no site) o Blazor seria devastador, mas mesmo assim, todo nosso site é feito em Razor Pages.

Esta decisão é por que nossa aplicação é relativamente pequena, não há motivos para quebrá-la em três só para ter o Blazor na jogada (Pagamento e Player).

Os benefícios não justificavam o esforço e custo de manter três aplicações, e isto era um dos motivos pelos quais não movemos para o Blazor.

Acredito que muitos outros negócios também se seguraram nestas argumentações até agora, já que o jogo mudou completamente.

O que são Render Modes

No Blazor 8 (.NET 8) foi incluso uma opção chamada Render Modes ou modo de renderização para componentes e páginas.

Esta opção permite definirmos o modelo de renderização de uma página ou componente (Na verdade por ser até para aplicação toda).

Particularmente, opto pela flexibilidade, então somente se fosse algo muito específico eu selecionaria apenas um Render Mode para aplicação toda.

Ao todo temos quatro modos de renderização, cada qual com suas particularidades, que afetam a forma como as páginas e componentes são desenhados.

- SSR e SSR Rendering
- Interactive Server
- Interactive WebAssembly
- Auto

SSR

Já comentamos sobre o SSR ou Server Side Rendering, mas vamos entender na prática o que ele é, e como ele funciona.

Como é o processo de SEO

SEO é a sigla para Search Engine Optimization e ele é um mundo a parte, existem livros só sobre o assunto, mas vamos resumir aqui.

Uma das validações que o Google faz para indexar nossas páginas e rankear elas em seu motor de busca é verificar o título de conteúdo da página.

Quando buscamos por “.NET” ele verifica as páginas que contém o título “.NET” e também analisa o corpo da página.

Esta última etapa é a mais complexa para nós, visto que se a página não possuir um corpo condizente com o título, você despenca no ranking.

Vale lembrar que no modelo WASM nós não temos a possibilidade de alterar o HTML gerado, já que ele é modificado no navegador.

Além disso, é importante frisar que os Render Modes são por componente (Ou página), então seja criativo, nem todo componente precisa ser SSR.

Como usar SSR no Blazor

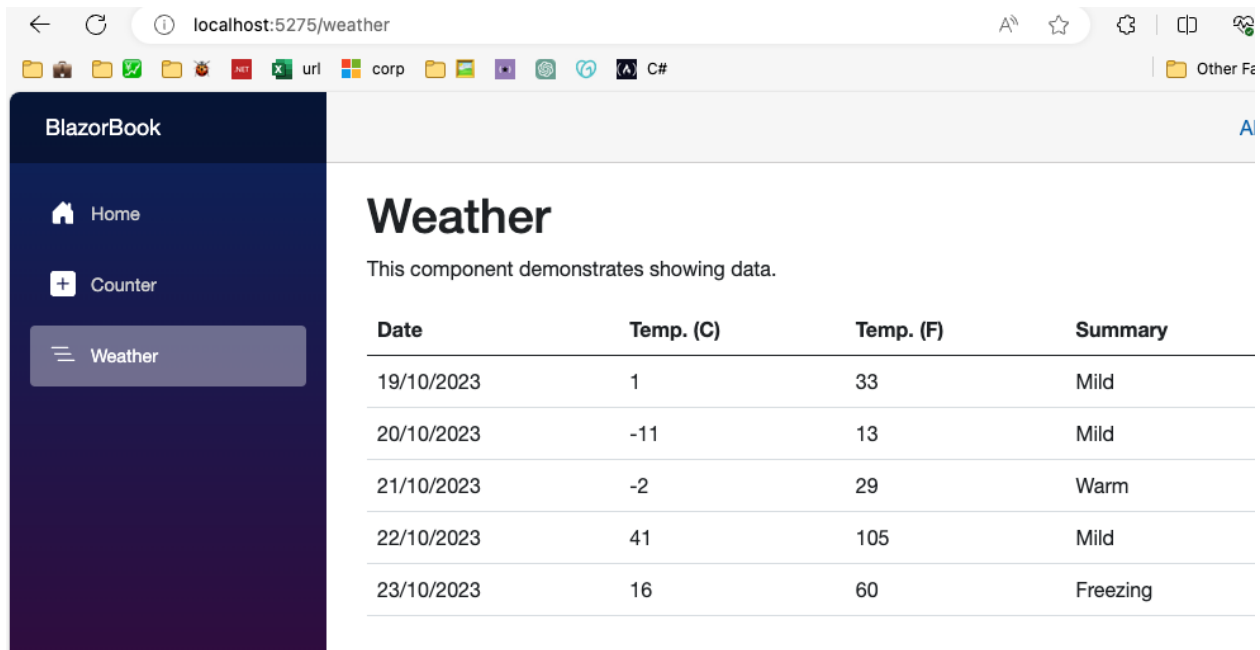
Lembra do tópico anterior onde trabalhamos no Counter e adicionamos um atributo nele chamado `@rendermode` `InteractiveServer` ?

Se sua página ou componente não declara de forma explícita que ele é interativo, o Blazor automaticamente renderizará ele usando SSR.

Se tomarmos como base por exemplo a página Weather, que carrega dados de um serviço, nós podemos comentar o atributo `StreamRendering` ou passar o valor `false` como parâmetro.

```
@* @attribute [StreamRendering(true)] *@  
  
ou  
  
@attribute [StreamRendering(false)]
```

Como resultado, teremos a página levando alguns segundos para carregar e depois exibindo os dados normalmente.



The screenshot shows a web browser window with the address bar displaying 'localhost:5275/weather'. The browser's taskbar at the bottom shows various application icons, including a file explorer, a terminal, and a C# application. The web application, titled 'BlazorBook', features a dark sidebar with navigation links for 'Home', 'Counter', and 'Weather'. The 'Weather' page is currently active, displaying a table of weather data. The table has four columns: 'Date', 'Temp. (C)', 'Temp. (F)', and 'Summary'. The data rows show weather information for the dates 19/10/2023, 20/10/2023, 21/10/2023, 22/10/2023, and 23/10/2023.

Date	Temp. (C)	Temp. (F)	Summary
19/10/2023	1	33	Mild
20/10/2023	-11	13	Mild
21/10/2023	-2	29	Warm
22/10/2023	41	105	Mild
23/10/2023	16	60	Freezing

Porém, ao inspecionar a página, podemos ver que os dados estão lá, ou seja, caso eles fossem relevantes para o SEO, estaríamos bem.

```
view-source:localhost:5275/weather

<main b-3p7n9vmdqf><div class="top-row px-4" b-3p7n9vmdqf><a href="https://learn.microsof
<article class="content px-4" b-3p7n9vmdqf>
<h1>Weather</h1>
<p>This component demonstrates showing data.</p><table class="table"><thead><tr><th>Date</th>
  <th>Temp. (C)</th>
  <th>Temp. (F)</th>
  <th>Summary</th></tr></thead>
  <tbody><tr><td>19/10/2023</td>
    <td>31</td>
    <td>87</td>
    <td>Bracing</td></tr><tr><td>20/10/2023</td>
    <td>-19</td>
    <td>-2</td>
    <td>Hot</td></tr><tr><td>21/10/2023</td>
    <td>-11</td>
    <td>13</td>
    <td>Hot</td></tr><tr><td>22/10/2023</td>
    <td>-5</td>
    <td>24</td>
    <td>Sweltering</td></tr><tr><td>23/10/2023</td>
    <td>15</td>
    <td>58</td>
    <td>Bracing</td></tr></tbody></table></article></main></div>
<div id="blazor-error-ui" b-3p7n9vmdqf>
  An unhandled error has occurred.
```

O que acontece neste cenário é que toda a página é gerada no servidor, ou seja, HTML, CSS e JS, e depois enviado para renderização.

É o mesmo processo que o ASP.NET Razor Pages faz (Talvez seja hora de migrar o balta para Blazor 😊).

SSR Streaming

O **Streaming** ou **Stream Rendering** executa um processo parecido, porém mais inteligente, trazendo a página em um primeiro momento e os dados depois.

Para utilizar o **StreamRendering** basta utilizar o atributo **StreamRendering(true)** conforme mostrado abaixo.

```
@attribute [StreamRendering(true)]
```

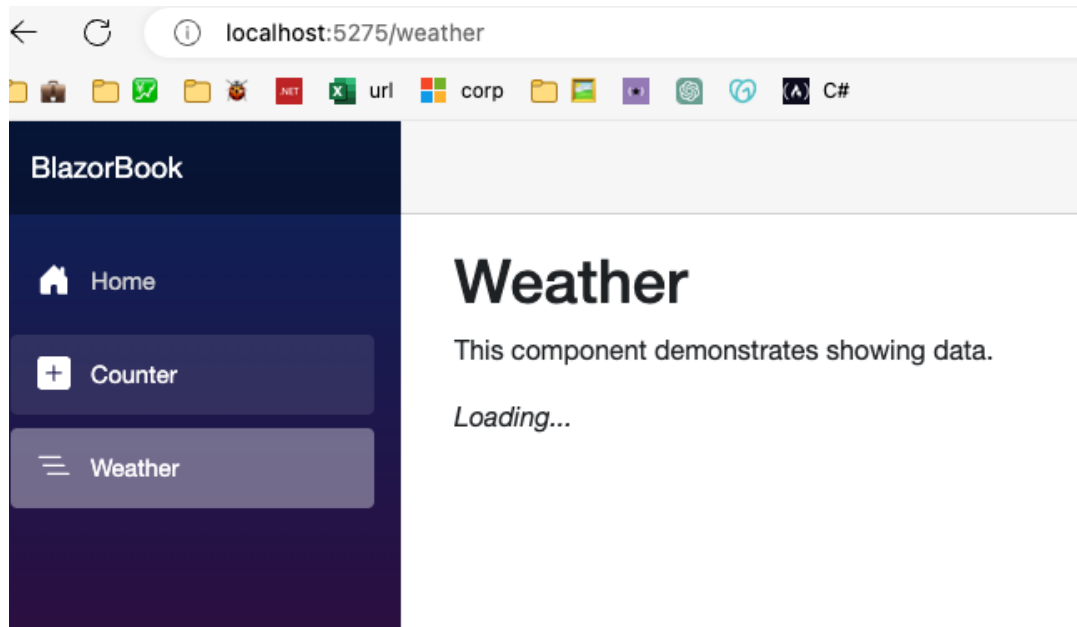
Para conseguir visualizar o efeito do StreamRendering, recomendo que altere o atraso na recuperação de dados.

```
// Linha 45
await Task.Delay(5000);
```

Esta página representa um uso extremamente comum em nosso dia-a-dia, onde precisamos ler dados do banco de dados ou de um serviço e exibir na tela.

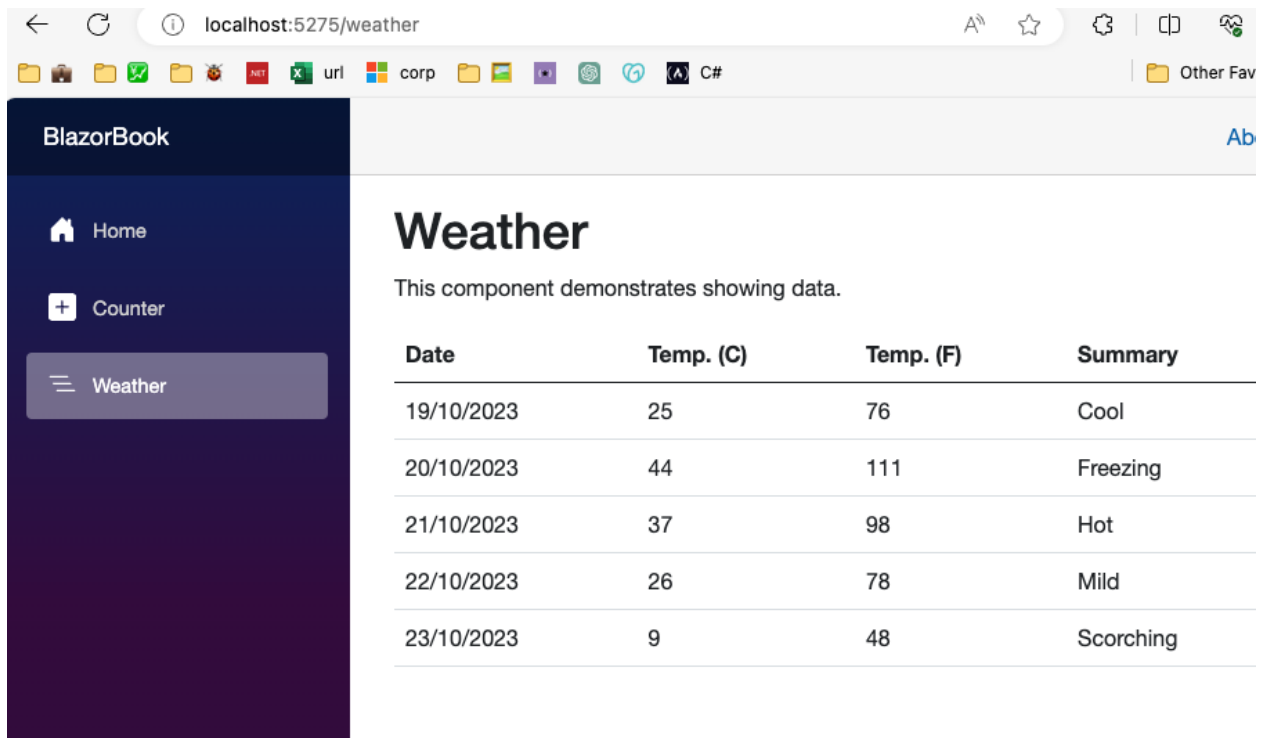
O que fizemos acima foi simular uma “demora” na recuperação destes dados, que é outro cenário comum.

Ao executar a aplicação, navegando para página de clima, temos o conteúdo carregado, mas os dados não, exibindo um “Loading” enquanto os mesmos são recuperados.

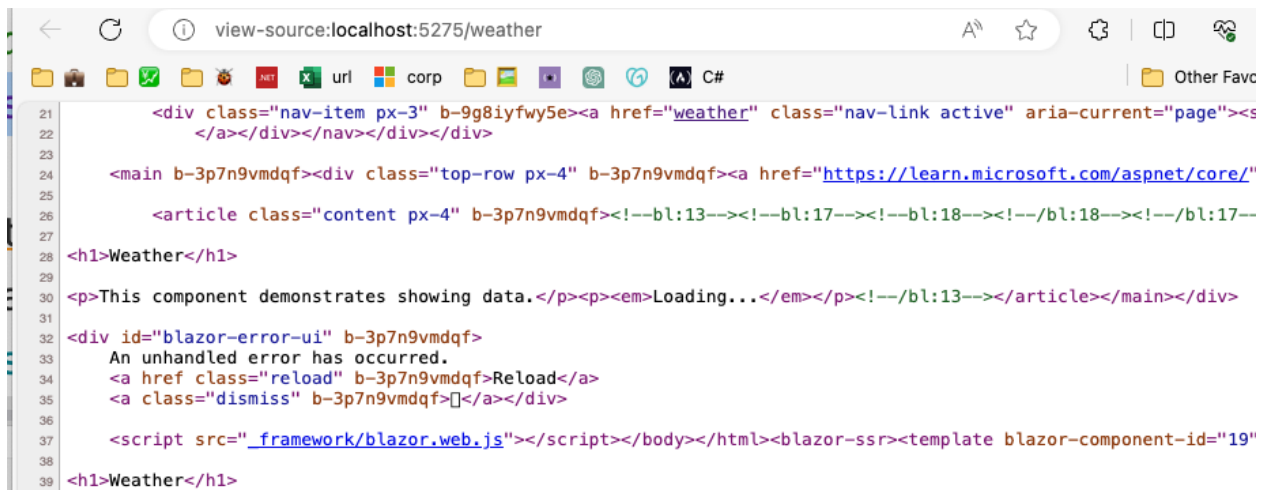


A principal diferença entre o `StreamRendering` e o SSR é que no exemplo anterior (SSR) a página não era carregada enquanto os dados não estivessem completos.

Neste caso, assim que os dados chegam do banco ou serviço, eles automaticamente vão para a tela, atualizando o componente ou página.



Porém, se você precisa destas informações para uma melhor indexação da sua página (SEO), é melhor olhar o HTML antes.



Como podemos ver, dado que o conteúdo foi injetado depois que a página foi carregada, o HTML não foi modificado.

Desta forma, o HTML sempre irá renderizar o conteúdo inicial que ele teve, quando foi servido direto do servidor via SSR.

```
<h1>Weather</h1>
```

```
<p>This component demonstrates showing data.</p><p><em>Loading...</em></p><!--/bl:13--></article></main></div>
```

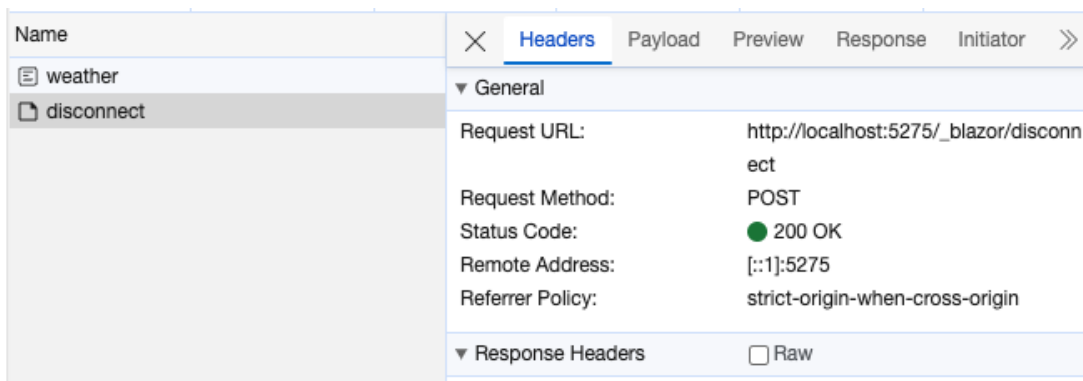
Como esta mágica acontece?

Sempre que o Blazor identifica um componente ou página com `StreamRendering` ele renderiza tudo o que pode, o que tem em mãos.

No caso da página `weather`, como ele não tem todas as informações, ele renderiza a página e fica aguardando pelos dados adicionais.

Assim que estes dados chegam, ele atualiza a tela e isto é feito por que a tela se mantém conectada ao servidor através de um `Socket`.

Ao inspecionarmos a execução da página, podemos ver que após navegar para `weather`, após o carregamento de todos os dados, temos um item `disconnect`.



Isto significa que mesmo após a página ter sido exibida na tela, ainda havia uma conexão ativa entre ela e o servidor, que foi fechada apenas momentos depois.

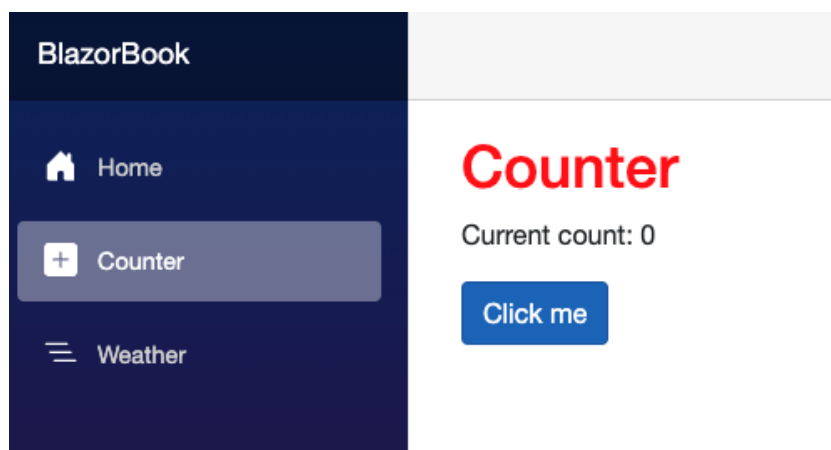
Interactive Server

Enquanto o SSR e Stream Rendering são excelentes para exibir informações, ainda temos a necessidade de interação.

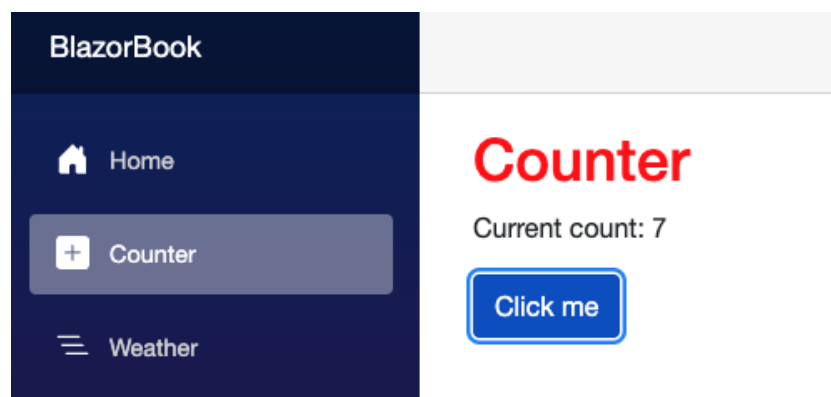
Sempre que precisamos dar vida aos componentes ou páginas, interagindo com botões, inputs, dentre outros itens, temos três opções, sendo a primeira o **Interactive Server**.

Neste formato, adicionamos o atributo `@rendermode InteractiveServer` ao componente ou página.

O comportamento é muito parecido com o do Stream Rendering, nosso componente irá renderizar mas ficará conectado ao servidor.



Desta forma, podemos processar nosso código no servidor, também utilizando Socket (SignalR) para manter a conexão aberta.



O componente responde rapidamente e se atualiza. Isto acontece quase que instantaneamente pois a conexão via Socket e a otimização na transferência de dados ajudam bastante.

Porém, assim como no Stream Rendering, como estamos modificando os elementos da tela após o HTML ser renderizado, o código da página não é alterado.

```
26 <article class="content px-4 b-3p7n9vmdqf">
27
28 <h1 b-f0mv87vb1r>Counter</h1>
29
30 <p role="status" b-f0mv87vb1r>Current count: 0</p>
31
32 <button class="btn btn-primary" b-f0mv87vb1r>Click me<
33
34 <div id="blazor-error-ui" b-3p7n9vmdqf>
35 An unhandled error has occurred
```

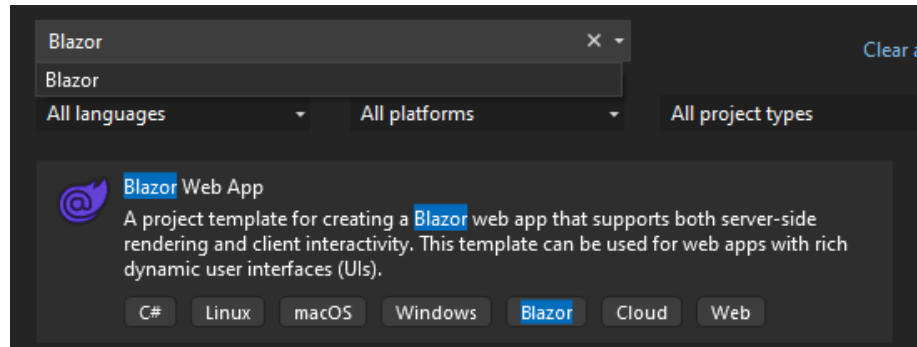
Criando projetos no Visual Studio

Durante a escrita deste eBook a versão mais recente do .NET 8 é a RC 2 (Release Candidate), que embora esteja muito próxima da final, ainda pode variar.

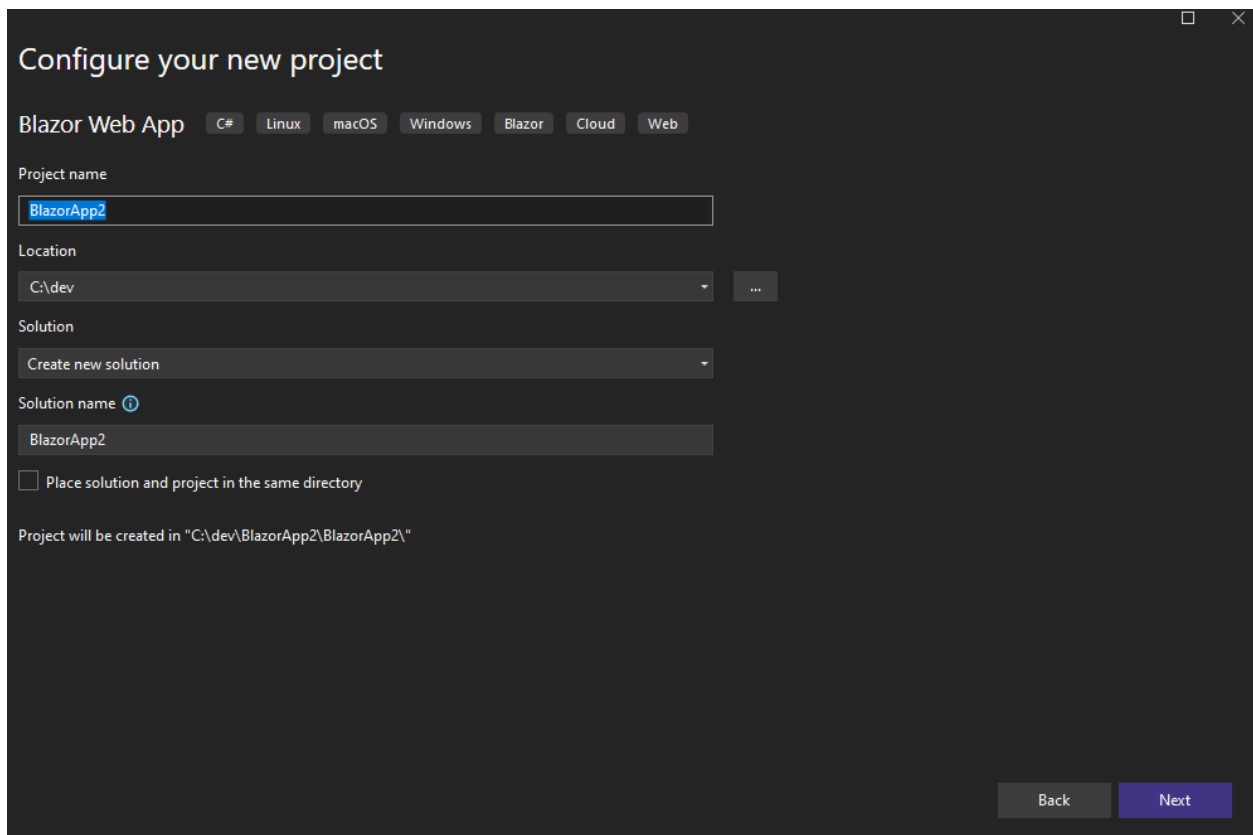
Desta forma, para entender os próximos modos de renderização, criaremos o projeto por dentro do Visual Studio Preview 17.0.4 (Ou superior).

Acredito que na versão final teremos comandos do `dotnet new` para criação destes templates, dispensando uso do Visual Studio neste passo.

Por enquanto, vamos selecionar a opção **File > New Project** e buscar pelo template **Blazor Web App**.

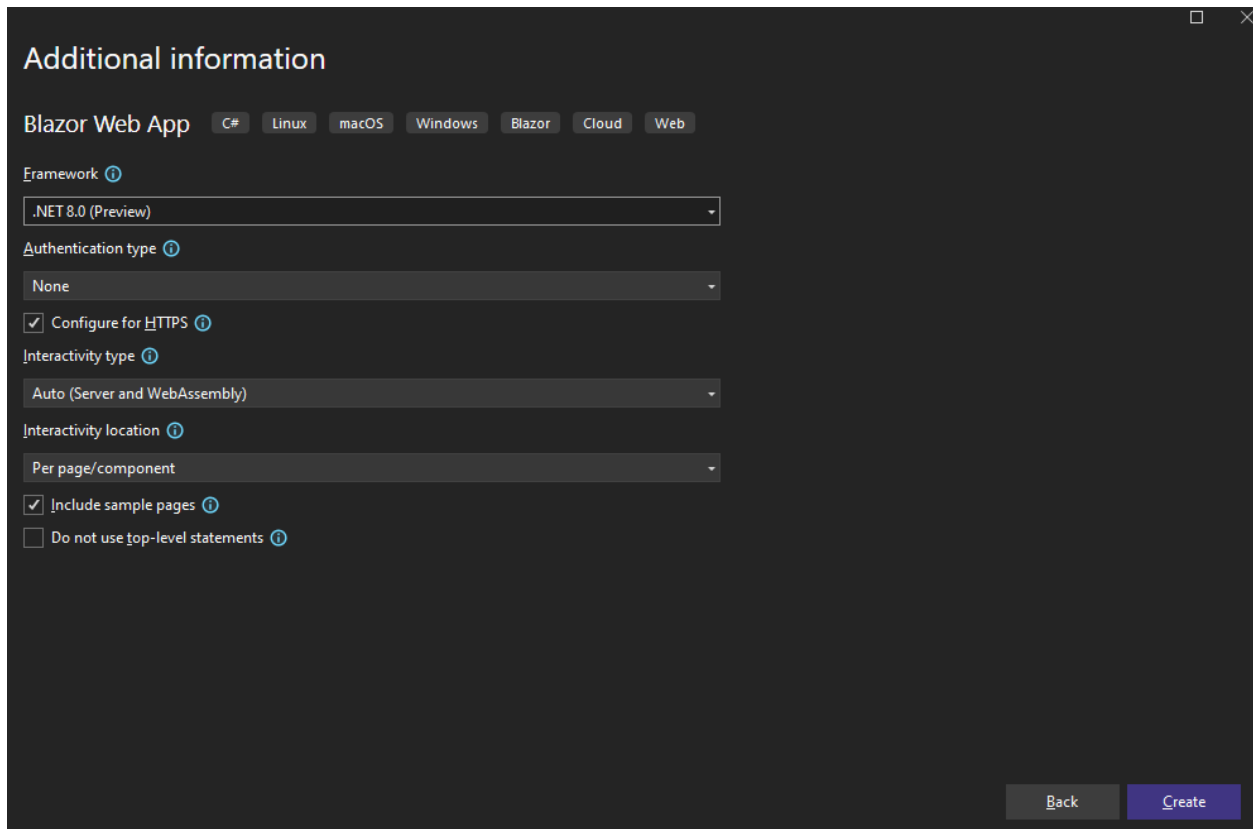


Após selecionar o template, vamos nomear o projeto e selecionar uma localização para o mesmo, fique à vontade nesta etapa.

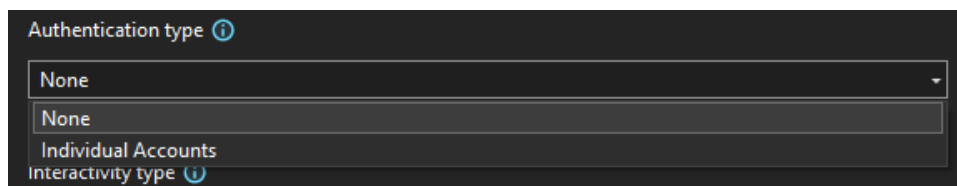


Movendo adiante, temos a tela que configura o tipo de interação que o projeto terá, que fica na opção **Interactivity Type**.

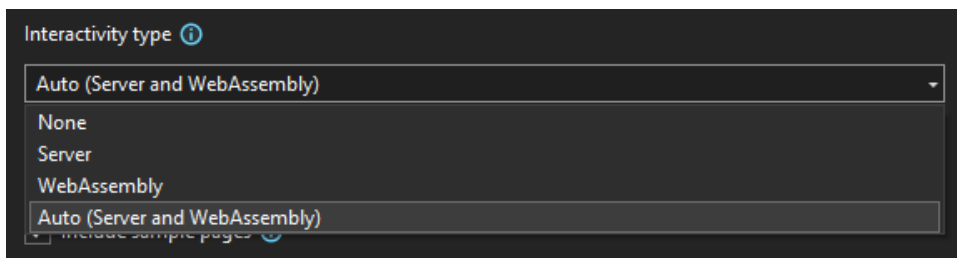
Além desta opção temos o Interactivity Location, que podemos definir como **Global** (Para todas as páginas e componentes) ou **Per Page/component** (Específico por página ou componente).



Neste momento vale ressaltar que o **Identity** também chegou ao Blazor, sendo possível configurar o tipo de autenticação como **None** ou **Individual Accounts**.



Por fim temos os tipos de interação que queremos, por enquanto vamos ignorar o **None** e **Server** que já vimos anteriormente e focar no **WebAssembly** e **Auto**.



Nos próximos tópicos tomaremos como base projetos criados com Interactivity Type Web Assembly e Auto respectivamente.

Interactive WebAssembly

Mas se podemos rodar o .NET direto no navegador utilizando WebAssembly (WASM), por que utilizar o modo anterior?

Sempre que rodamos nosso código no servidor, temos a segurança da execução em um ambiente externo.

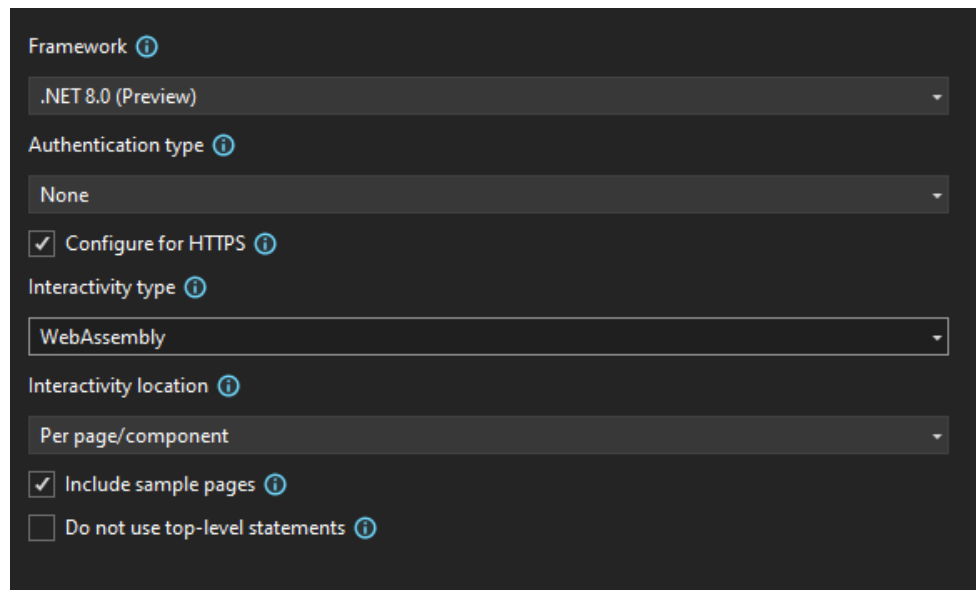
Por exemplo, não daria para deixar uma Connection String no WASM, visto que o código é executado no navegador, esta informação teria que ficar por lá.

Por outro lado, ficar conectado ao servidor (Modelo Interactive Server) demanda mais infraestrutura e pode ser mais lento.

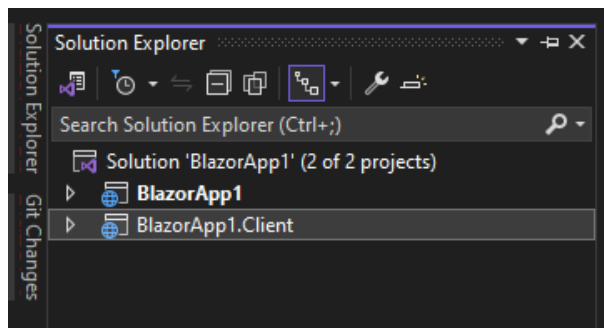
Além de não depender do servidor, o WASM ainda tem o poder de usar mais a capacidade da máquina do cliente (Que provavelmente é mais rápida que o servidor).

Inspeccionando o projeto

Para iniciar um projeto com **Interactivity Type** em **WebAssembly**, basta selecionar a opção conforme imagem abaixo.



O primeiro ponto que podemos notar é que alterando o **Interactivity Type** para **WebAssembly** temos agora dois projetos sendo criados, um para o **Server** e outro para o **Client**.



Começando nossa análise pelo projeto do servidor (`BlazorApp1`), podemos notar que no `Program.cs` temos a adição do `AddInteractiveWebAssemblyComponents` desta vez.

Este item só pode ser adicionado em projetos com *Interactivity Type* atribuídos como **WebAssembly** ou **Auto**.

Em adicional, como o WebAssembly roda apenas no lado do cliente (Navegador), precisamos de um projeto (`BlazorApp1.Client`) dedicado para ele.

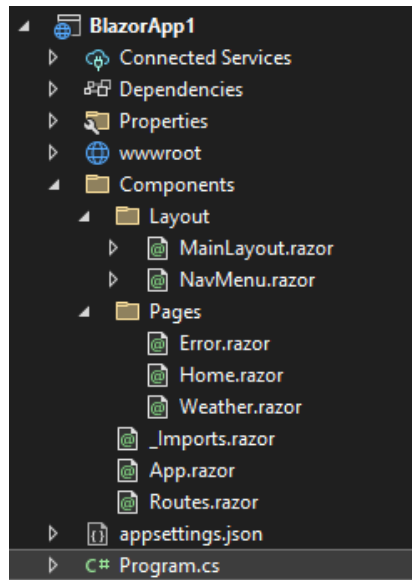
Continuando nossa análise, temos como resultado a adição do `AddInteractiveWebAssemblyComponents` e `AddInteractiveWebAssemblyRenderMode` no `Program.cs` .

```
// ...
builder
    .Services
    .AddRazorComponents()
    .AddInteractiveWebAssemblyComponents();

// ...
app.MapRazorComponents<App>( )
    .AddInteractiveWebAssemblyRenderMode( )
    .AddAdditionalAssemblies(typeof(Counter).Assembly);
```

Para finalizar, temos a adição do `AddAdditionalAssemblies(typeof(Counter).Assembly)` , que nos diz onde encontrar componentes que forem do tipo **WebAssembly**.

Podemos notar também, que dentro do projeto `BlazorApp1` , na pasta `Component/Pages` não existe mais o component `Counter.razor` .



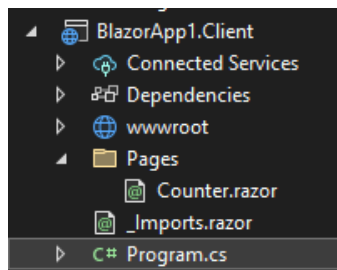
Esta mudança ocorre pois tratando-se de um projeto com suporte a **WebAssembly**, a fins de exemplo, o componente foi movido para o projeto `BlazorApp1.Client`.

Todos os componentes contidos neste projeto podem ser renderizados no formato **WebAssembly**, além da possibilidade de usar componentes no projeto **Server**.

Se analisarmos o `Program.cs` do projeto `BlazorApp1.Client`, podemos notar que esta é uma aplicação Web .NET extremamente simples.

```
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;  
  
var builder = WebAssemblyHostBuilder.CreateDefault(args);  
  
await builder.Build().RunAsync();
```

Por fim, expandindo a pasta `Pages`, podemos encontrar o componente `Counter.razor`, que nos exemplos anteriores estava junto aos outros componentes.



Fora esta mudança, o código do nosso componente se mantém o mesmo, com uma singela mudança no **Render Mode**.

```

@page "/counter"
@rendermode InteractiveWebAssembly

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;

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

```

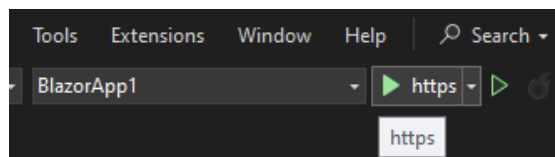
Desta vez, como podemos ver na segunda linha, utilizamos o atributo `InteractiveWebAssembly` ao invés do `InteractiveServer` como havíamos feito no passado.

Executando o projeto

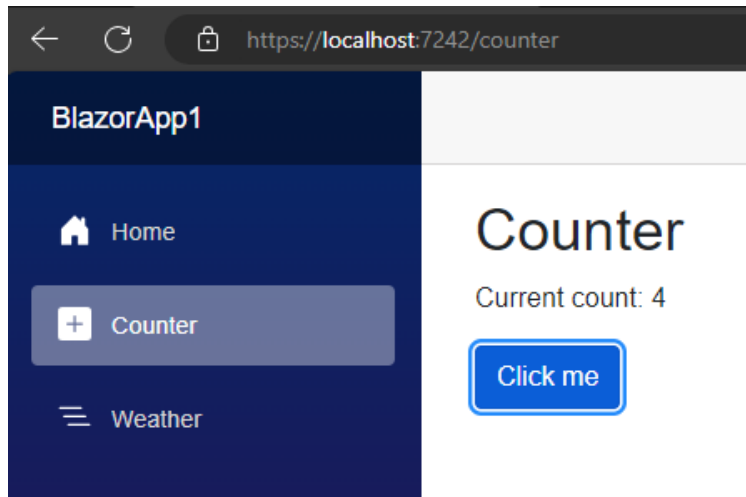
Algo que precisamos entender é que o projeto `BlazorApp1.Client` serve apenas como um local para armazenarmos nossas páginas e componentes que desejamos renderizar no formato **WebAssembly**.

Desta forma, vamos sempre executar o projeto `BlazorApp1`, que consequentemente carregará o projeto `BlazorApp1.Client` consigo.

O mesmo vale para a publicação do projeto, onde faremos sempre o deploy do `BlazorApp`, que é nossa aplicação e não do `BlazorApp1.Client`.



Como resultado da execução do projeto, temos o mesmo exemplo dos tópicos anteriores, porém, desta vez o componente `Counter.razor` está servido via **WebAssembly** (C# rodando no navegador).



Vale a pena ressaltar que a primeira vez que o componente é carregado, podemos experimentar uma leve lentidão, devido ao fato que o .NET estará sendo baixado no navegador, conforme explicamos anteriormente.

Auto

Finalizando nosso entendimento sobre **Render Modes**, temos o **Auto**, que é uma mescla entre o **Server** e o **WebAssembly**.

Neste modo, as páginas ou componentes são servidas primariamente no modelo **Server**, visto que este é o modo mais rápido para exibir um componente.

Dizemos isto pois no **WASM** precisamos baixar o .NET e isto pode levar algum tempo, dependendo da conexão do usuário com a internet.

Após exibido o componente na tela, o Blazor iniciará automaticamente o download do **.NET Runtime** em segundo plano.

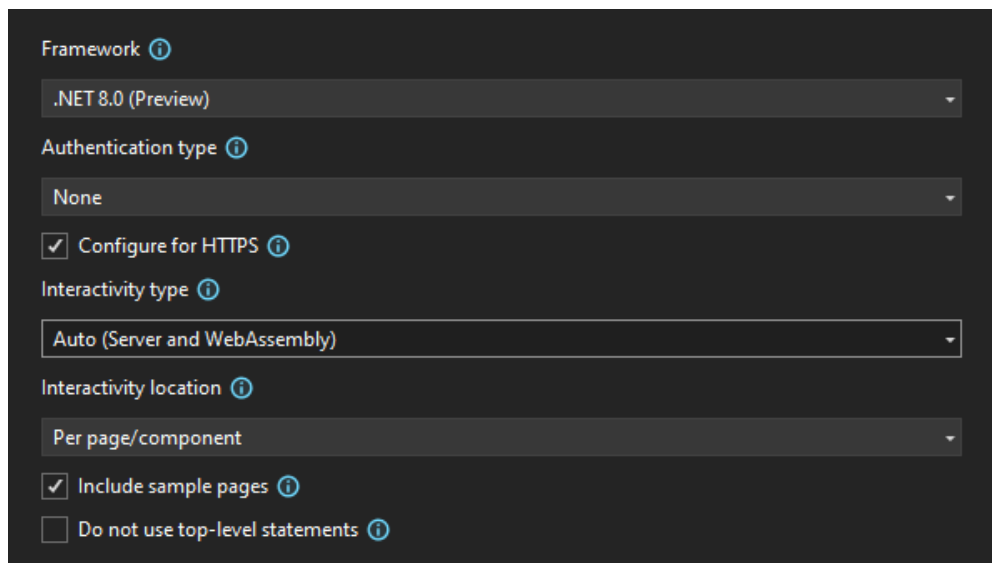
Nas próximas vezes que o usuário tentar renderizar este componente, o Blazor checará se o **.NET Runtime** já está presente na máquina.

Caso positivo, o componente então é renderizado utilizando **WebAssembly**, o que o torna mais performático após carregado.

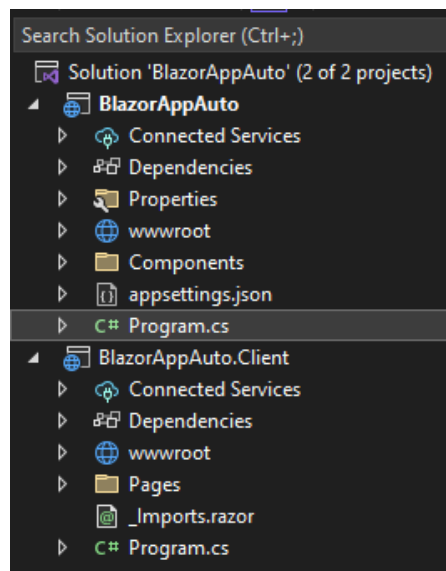
Em resumo, no modo **Auto** nós temos o melhor dos dois mundos, com a agilidade na renderização inicial do componente e a performance posterior do **WASM**.

Inspecionando o projeto

Para iniciar um projeto com o **Interactivity Type** em **Auto**, basta selecionar a opção conforme imagem abaixo.



Como você já deve imaginar, neste projeto temos a soma dos exemplos anteriores, suportando Server e WASM juntos.



Sendo assim, novamente precisamos de dois projetos, um para o **Server** e outro para o **WASM**, e seguimos os mesmos padrões do exemplo anterior.

Porém, desta vez podemos notar o uso tanto do `AddInteractiveServerComponents` quanto do `AddInteractiveWebAssemblyComponents`.

```
// ...  
  
builder  
    .Services  
    .AddRazorComponents()  
    .AddInteractiveServerComponents()
```



```

        .AddInteractiveWebAssemblyComponents();

// ...

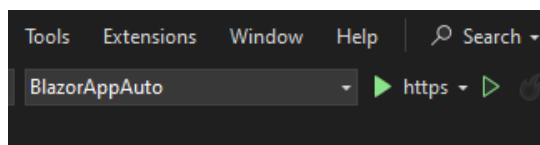
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof(Counter).Assembly);

```

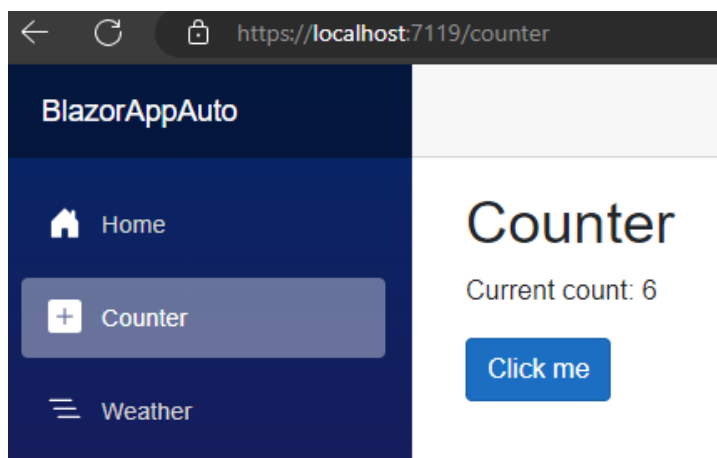
Da mesma forma, precisamos do `AddInteractiveServerRenderMode` e `AddInteractiveWebAssemblyRenderMode` para suportar os dois modos de renderização.

Executando o projeto

A execução deste projeto se dará pelo projeto `BlazorAppAuto` e não pelo projeto `BlazorAppAuto.Client`, igual vimos no exemplo anterior.



Como resultado, temos um comportamento exatamente idêntico ao anterior, com a renderização do componente **Counter** na tela.



Porém, existe um sutil detalhe aqui, sendo que na primeira execução deste componente, o mesmo foi servido no modo **Server**.

Enquanto o componente **Counter** era servido e o usuário navegava pelo aplicativo, o **.NET Runtime** foi baixado em segundo plano.

Agora, sempre que o usuário retornar ao componente **Counter**, o mesmo será renderizado no formato **WASM**, já que o **.NET Runtime** se encontra instalado no navegador.

Ao analisarmos o `Counter.razor` desta vez, temos o uso do `InteractiveAuto` ao invés do `InteractiveServer` ou `InteractiveWebAssembly`.

```

@page "/counter"
@rendermode InteractiveAuto

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;

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

```

Em resumo, temos basicamente o mesmo código para todos os modos de renderização, porém eles tem um comportamento bem diferente por baixo dos panos.

Capítulo 7 - Rotas e Navegação

O Blazor trabalha de forma extremamente simples com rotas e navegação, e se você já aprendeu um pouco sobre isto no **curso gratuito de Razor Pages**, estará mais do que em casa aqui.

Isto por que dado um atributo `@page` em um arquivo `.razor`, o mesmo se torna automaticamente uma rota.

Inclusive, precisamos nos policiar para não termos rotas iguais, desta forma, recomendo que estruture bem as pastas e organize bem seu projeto.

⚠ IMPORTANTE

Você não precisa seguir a estrutura de exemplo dos projetos Blazor, você pode criar as suas, inclusive recomendo que utilize a separação por funcionalidades ou áreas ao invés de ter uma única pasta para páginas e componentes.

Rotas

Indo direto ao ponto, sempre que definimos um atributo `@page` temos uma rota, que deve ser no formato `string` e começar com `"/"`.

```

@page "/counter"
// https://localhost:xxxx/counter

```

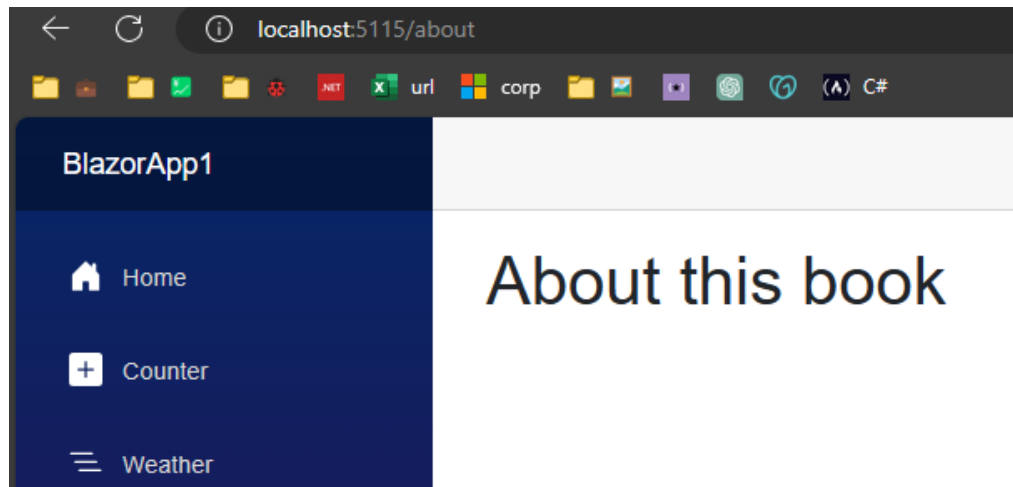
Desta forma, se criarmos um arquivo chamado `About.razor` na pasta `Components/Pages` com o seguinte conteúdo:

```
@page "/about"

<h1>About this book</h1>
```

Teremos como resultado o seguinte conteúdo sendo exibido, sempre que navegarmos para URL

<https://localhost:xxxx/about>.

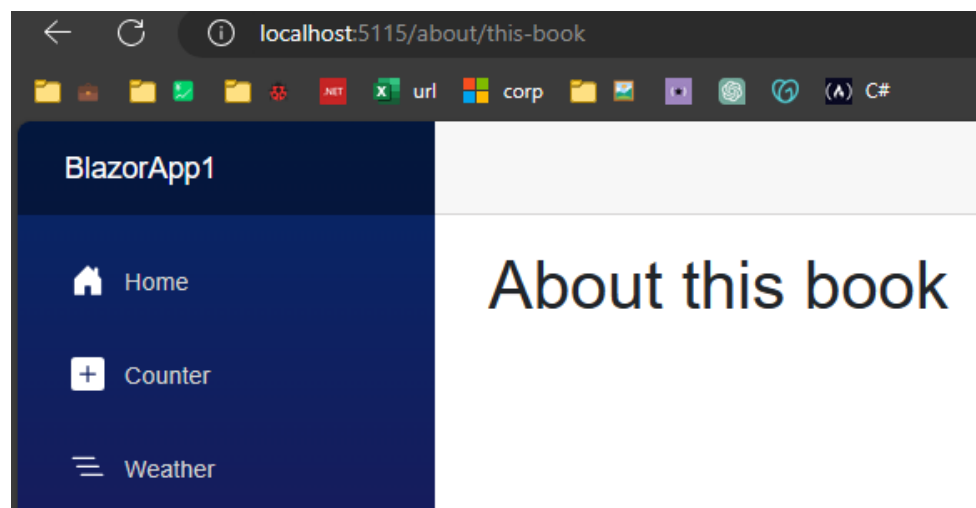


Da mesma forma, podemos alterar a URL para ter mais segmentos, e continuar com um resulta similar.

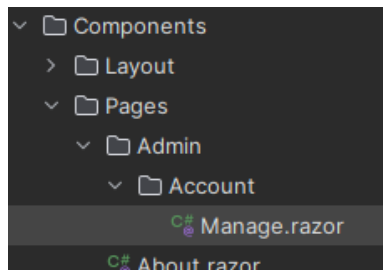
```
@page "/about/this-book"

<h1>About this book</h1>
```

Isto ocorre por que o uso de “/” e “-” na URL é plenamente permitido, então a renderização da página segue normalmente.



Por fim, pouco importa onde sua página está, se houver o atributo `@page` no começo do seu arquivo, ele será adicionado as rotas.

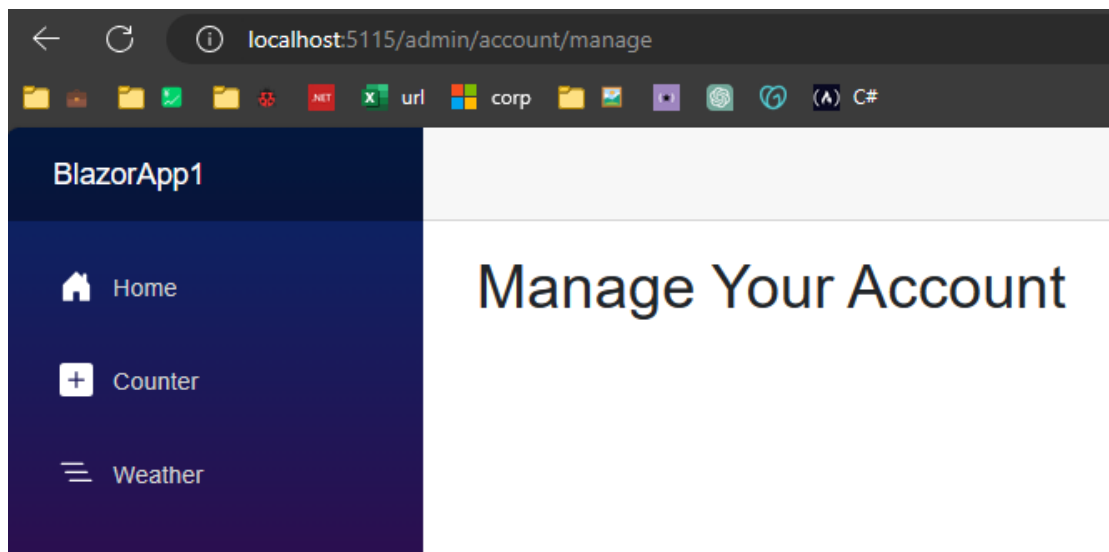


Desta forma, temos um arquivo `Manage.razor` em uma subpasta, dentro de outra subpasta, que contém o atributo `@page` como mostrado abaixo.

```
@page "/admin/account/manage"

<h1>Manage Your Account</h1>
```

E como resultado, temos a página sendo renderizada normalmente, visto que não faz diferença em qual pasta ela está alocada.



DICA

Você não precisa seguir a mesma convecção das pastas no nome da URL. No exemplo acima, embora o arquivo `Manage.razor` esteja na pasta `Admin/Account`, sua URL não necessariamente precisa seguir este padrão, embora seja recomendado.

Links

O Blazor possui duas formas de criar links para páginas, o tradicional `href` do HTML e o `NavLink`, componente exclusivo do Blazor.

Em ambos os casos devemos referenciar o caminho (URL) até a nossa página, que no exemplo é a `Counter.razor`.

```
<a class="nav-link" href="/counter">
    Counter (Href)
</a>
```

Da mesma forma, podemos utilizar o `NavLink`, que é o componente de navegação padrão do Blazor e nos fornece alguns recursos adicionais.

```
<NavLink class="nav-link" href="counter">
    Counter (NavLink)
</NavLink>
```

NavLink

A maior diferença é que o `NavLink` verifica a rota que estamos e tem a possibilidade de adicionar alguma classe do CSS ao elemento em questão.

No caso, o `NavLink` tem a possibilidade de alternar uma classe CSS do elemento utilizando os atributos `Match` e `ActiveClass`.

Caso o atributo `ActiveClass` não seja definido, o `NavLink` por padrão atribuirá uma classe chamada `active` do CSS ao elemento.

```
<NavLink class="nav-link" href="" Match="NavLinkMatch.All">
    Home
</NavLink>
```

Note que no `href` a "/" não está presente, isto por que estamos utilizando um template de rotas e ela se torna opcional.

Match

O atributor `Match` é um **enumerador** que possui dois valores, sendo eles `All` e `Prefix`, ambos utilizados para verificar a URL atual.

O `NavLinkMatch.All` define que a URL completa deve bater com o valor do `href` para que a classe `active` (Ou valor do `ActiveClass`) seja atribuído ao elemento.

Já o `NavLinkMatch.Prefix` define que a classe `active` (Ou valor do `ActiveClass`) deve ser aplicado quando o prefixo da URL bate com o valor `href` do elemento.

```
<li>
    <NavLink href="/pedidos" Match="NavLinkMatch.Prefix">
        Pedidos
```

```

        </NavLink>
    </li>
    <li>
        <NavLink href="/pedidos/detalhes" Match="NavLinkMatch.All">
            Detalhes do Pedido
        </NavLink>
    </li>
    <li>
        <NavLink href="/pedidos/editar" Match="NavLinkMatch.All">
            Editar Pedido
        </NavLink>
    </li>

```

No exemplo acima, quando o usuário navega para `/pedidos`, apenas o primeiro link fica ativo.

Quando o usuário navega para `/pedidos/detalhes`, o primeiro link permanece ativo, pois ainda corresponde ao prefixo da URL, e o segundo link também fica ativo porque corresponde a toda a URL.

Os valores dos parâmetros de rota são considerados para fins de correspondência de URLs, mas os valores e fragmentos da string de consulta são ignorados.

ActiveClass

O atributo `ActiveClass` permite adicionar uma classe (CSS) ao `NavLink`. Por padrão, uma classe chamada `active` é adicionada.

Para alterar esse valor padrão, basta informar o nome da classe (CSS) que deseja atribuir ao `NavLink` quando o `Match` for identificado.

```

<NavLink href="/pedidos" Match="NavLinkMatch.Prefix">
    Pedidos
</NavLink>

```

No exemplo acima, como nenhum valor foi atribuído ao `ActiveClass`, uma classe chamada `active` será atribuída assim que navegarmos para `/pedidos`.

```

<NavLink href="/pedidos" ActiveClass="link-ativo" Match="NavLinkMatch.Prefix">
    Pedidos
</NavLink>

```

Já neste outro exemplo, como um valor foi atribuído ao `ActiveClass`, assim que navegarmos para `/pedidos` uma classe chamada `link-ativo` será adicionada ao `NavLink`.

NavigationManager

Para realizar a navegação através de um código C#, podemos utilizar o `NavigationManager`, injetando ele na página.

```

@page "/"
@Inject NavigationManager navigationManager
<button @onclick="Navigate">Click</button>

```

```
@code{
    void Navigate()
    {
        navigationManager.NavigateTo("/other");
    }
}
```

Para realizar a navegação podemos utilizar algum dos métodos presentes no `NavigationManager`, como o `NavigateTo` e especificar uma URL.

FocusOnNavigate

Outro elemento que podemos encontrar é o `FocusOnNavigate`, presente no arquivo `Routes.razor` e que nos auxilia na seleção de elementos.

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
</Router>
```

No exemplo acima, definimos que sempre após navegar para uma rota (Desde que ela exista), iremos selecionar o primeiro `H1` (Headline) assim que a página for carregada.

Este comportamento nos ajuda na questão de usabilidade e acessibilidade, tornando mais fácil a navegação sem mouse ou simplesmente a rolagem da tela para uma sessão importante após a navegação.

Rotas em diferentes projetos

É possível ter componentes e páginas em projetos distintos, seja por motivo de reuso ou simplesmente uma organização do seu projeto.

Neste caso, podemos utilizar o `AdditionalAssemblies` para especificar projetos (Assemblies) adicionais ao atual.

```
<Router
    AppAssembly="@typeof(App).Assembly"
    AdditionalAssemblies="new[] { typeof(Component1).Assembly }">
    @* ... Router component elements ... *@
</Router>
```

Neste caso, o Blazor também buscará páginas em outros projetos especificados nesta propriedade.

Parâmetros de Rotas

Uma ação muito comum que temos é a passagem de parâmetros através das rotas durante a navegação.

No Blazor, assim como no Razor Pages, os parâmetros podem ser especificados através das chaves na URL.

```
@page "/route-parameter-1/{text}"
```

Posteriormente podemos atribuir (Binding) um parâmetro a uma propriedade simplesmente usando o atributo `Parameter` para decorá-la.

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

É importante frisar que o atributo `Parameter` só pode ser vinculado a propriedades. Variáveis não irão funcionar neste caso.

Em adicional, é importante manter o mesmo nome entre a propriedade e parâmetro para uma transição sem atritos. O exemplo final fica assim:

```
@page "/route-parameter-1/{text}"  
  
<h1>Blazor is @Text!</h1>  
  
@code {  
    [Parameter]  
    public string? Text { get; set; }  
}
```

Desta forma, ao chamarmos a URL `https://localhost:xxxx/route-parameter-1/awesome` teríamos o texto ***Blazor is awesome*** escrito na tela.

Em contrapartida, ao tentar acessar a mesma URL anterior sem informar um parâmetro (`https://localhost:xxxx/route-parameter-1`), teríamos um erro **404**, de página não encontrada.

Parâmetros opcionais

Como vimos no exemplo anterior, assim que declaramos um parâmetro em uma rota, ele se torna parte dela.

Desta forma, caso o parâmetro não seja informado, a rota se torna incompleta, sendo impossível acessá-la.

Assim como no C# temos os **Nullable Types**, aqui temos os parâmetros opcionais, onde podemos definir sua obrigatoriedade utilizando um `?`.

```
@page "/route-parameter-1/{text?}"
```

Adicionando o `?` ao fim do parâmetro, ele será declarado como `Nullable`, ou seja, se tornará opcional, tornando a URL `https://localhost:xxxx/route-parameter-1` válida.

Vale lembrar que a URL `https://localhost:xxxx/route-parameter-1/awesome` também continua válida, visto que ela informa um parâmetro.

Para receber este parâmetro, é importante declarar a propriedade como `Nullable` também, já que o valor pode vir nulo e causar um *NullReferenceException*.

```
...
@code {
    [Parameter]
    public string? Text { get; set; }
}
```

Por fim, podemos fazer uso do método `OnInitialized` que é executado sempre que o componente é renderizado na tela.

Neste método podemos aproveitar para inicializar nossa propriedade com algum valor caso nenhum parâmetro seja fornecido.

```
protected override void OnParametersSet()
{
    Text = Text ?? "awesome";
}
```

O código final do nosso componente fica conforme mostrado abaixo:

```
@page "/route-parameter-1/{text?}"

<h1>Blazor is @Text!</h1>

@code {
    [Parameter]
    public string? Text { get; set; }

    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

Restrições de parâmetros

Além de especificar um parâmetro, podemos limitar seu tipo e adicionar algumas restrições a ele, facilitando ainda mais a validação.

Com as restrições de rotas, além de informar o parâmetro, caso o mesmo contenha alguma restrição, também será retornar o erro 404.

Para adicionar uma restrição ao parâmetro, basta especificar um tipo (Veja tabela adiante) após o uso do separador

`:`.

```
@page "/user/{Id:int}"
```

Desta forma, se tentarmos abrir a URL `/user/test`, teríamos um erro **404**, pois embora fora informado um parâmetro para o `Id`, o mesmo não é do tipo **inteiro** (Informamos `test`, uma **string**).

Abaixo encontra-se uma tabela com todas as restrições disponíveis e alguns exemplos de uso das mesmas.

Restrição	Exemplo	Validação	Cultura invariante
<code>bool</code>	<code>{active:bool}</code>	<code>true</code> , <code>FALSE</code>	No
<code>datetime</code>	<code>{dob:datetime}</code>	<code>2016-12-31</code> , <code>2016-12-31 7:32pm</code>	Yes
<code>decimal</code>	<code>{price:decimal}</code>	<code>49.99</code> , <code>-1,000.01</code>	Yes
<code>double</code>	<code>{weight:double}</code>	<code>1.234</code> , <code>-1,001.01e8</code>	Yes
<code>float</code>	<code>{weight:float}</code>	<code>1.234</code> , <code>-1,001.01e8</code>	Yes
<code>guid</code>	<code>{id:guid}</code>	<code>CD2C1638-1638-72D5-1638-DEADBEEF1638</code> , <code>{CD2C1638-1638-72D5-1638-DEADBEEF1638}</code>	No
<code>int</code>	<code>{id:int}</code>	<code>123456789</code> , <code>-123456789</code>	Yes
<code>long</code>	<code>{ticks:long}</code>	<code>123456789</code> , <code>-123456789</code>	Yes

Restrições e parâmetros opcionais

É possível utilizar restrições de parâmetros e parâmetros opcionais juntos, só devemos nos atentar a ordem das sinalizações.

Neste caso, o sinalizador de parâmetro opcional (`?`) vem sempre no fim da rota, enquanto o sinalizador de restrição (`:`) permanece no mesmo local anterior.

```
@page "/user/{Id:int}/{Option:bool?}"
```

Catch All

Além dos parâmetros convencionais, existe o chamado **Wildcard** ou **Coringa**, que é representado por um asterisco (`*`).

Quando utilizamos o `*`, ignoramos todos os parâmetros e passamos a assumir apenas a URL atual.

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

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

Desta forma, se chamássemos a URL `/catch-all/this/is/a/test`, todos os segmentos depois de `catch-all` seriam ignorados e o resultado da propriedade `PageRoute` seria `this/is/a/test`.

Este é um cenário para situações onde você quer ter uma segmentação na URL por conta de SEO mas quer manter uma mesma rota para um segmento todo.

Query Strings

Você com certeza já acessou algum site que na URL havia um `?chave=valor`. Esta combinação é conhecida como **Query String**.

Desta forma, podemos passar parâmetros para nossas páginas que não afetam suas rotas, mas sim complementam a URL.

Por exemplo, podemos chamar a URL `https://balta.io/busca?termo=SQL` para informar um termo de busca via **Query String**.

```
// https://balta.io/busca/termo
//                               🙌 -> Parâmetro via URL

// https://balta.io/busca?termo=SQL
//                               🙌 -> Parâmetro via Query String
```

Diferente dos parâmetros de rota, quando utilizamos Query String, nossas rotas não precisam de nada especial.

```
@page "/busca"
```

Em adicional, precisamos do atributo `SupplyParameterFromQuery`, além do atributo `Parameter` que já vimos anteriormente.

```
@code {
    [Parameter]
    [SupplyParameterFromQuery]
    public string? Term { get; set; }
}
```

Desta forma, dizemos que estamos recebendo um parâmetro chamado `Term` e que este será recebido via **Query String**.

Agora ao chamar a URL `https://balta.io/busca?termo=SQL` teremos como resultado o texto `Termo: SQL` sendo exibido na tela.

O código completo desta página fica conforme exibido abaixo:

```
@page "/busca"

<h1>Exemplo de Busca</h1>

<p>Termo: @Term</p>

@code {
    [Parameter]
    [SupplyParameterFromQuery]
    public string? Term { get; set; }
}
```

Capítulo 8 - Formulários

Overview

Talvez uma das funcionalidades mais desejadas em qualquer Framework seja a capacidade de manipular formulários com facilidade.

Aplicar validações e submeter informações através destes componentes, é algo que realizamos com frequência, e o Blazor nos ajuda muito nesta parte.

EditForm

O Blazor possui diversos componentes para se trabalhar com formulários, desde simples Input Text até o próprio formulário em si.

Neste caso, o componente que dá vida aos formulários é o `EditForm`, que adiciona funcionalidades como **Binding** e **Validation** ao elemento `Form` do HTML.

Em suma, o `EditForm` é um componente relativamente simples, que precisa de apenas dois atributos para funcionar.

Enquanto no atributo `Model` ligamos um objeto (Que representa os dados) em nosso formulário, no `OnSubmit` definimos qual função será executada quando o formulário for submetido.

```
<EditForm Model="@Model" OnSubmit="@Submit">
    ...
</EditForm>
```

Nosso próximo passo é criar uma propriedade para o `Model`, que neste caso é definido por uma instância do `CreateProductViewModel` e o método `Submit`.

```
@code {
    public CreateProductViewModel Model { get; set; } = new();

    private void Submit()
    {
        // ...
    }
}
```

Por fim, podemos vincular propriedades do `CreateProductViewModel` em componentes dentro do formulário e utilizar um botão para enviar as informações.

```
@page "/products/new"

<EditForm Model="@Model" OnSubmit="@Submit">
    <InputText @bind-Value="Model.Title" />
    <button type="submit">Cadastrar</button>
</EditForm>

@code {
```

```

public CreateProductViewModel Model { get; set; } = new();

private void Submit()
{
    // ...
}
}

```

Por fim, vale notar o uso do `InputText`, um tipo específico do Blazor que permite fazer o vínculo entre uma propriedade do `CreateProductViewModel` e esta caixa de texto.

Neste caso, o vínculo é feito através do atributo `@bind-Value`, onde podemos informar o objeto e propriedade (`Model.Title`).

Validação do Formulário

A forma mais fácil que temos de validar formulários é fazendo o uso dos `DataAnnotations`, um recurso presente nativamente no .NET.

```

public class CreateProductViewModel
{
    [Required]
    [StringLength(36, ErrorMessage = "Título muito grande.")]
    public string Title { get; set; } = string.Empty;
}

```

Neste exemplo, utilizamos o `Required` para definir que o campo é obrigatório e o `StringLength` para definir seu tamanho.

Note que podemos utilizar o `ErrorMessage` para personalizar a mensagem de erro que será exibida durante a validação.

O próximo passo é utilizar o `DataAnnotationsValidator`, um componente que fará uso das validações que acabamos de criar.

```

<EditForm Model="@Model" OnSubmit="@Submit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText @bind-Value="Model.Title" />
    <button type="submit">Cadastrar</button>
</EditForm>

```

Agora, ao tentar submeter o formulário sem informar nenhum texto, receberemos a mensagem ***“The Title field is required.”***, visto que não informamos nenhum `ErrorMessage` no atributo `Required` desta propriedade.

- The Title field is required.

O mesmo acontece se tentarmos submeter um texto com mais de 36 caracteres, mas desta vez temos uma mensagem personalizada conforme informamos no `StringLength` do `CreateProductViewModel`.

- Título muito grande.

Estas mensagens serão inseridas no componente `ValidationSummary`, que como o nome diz é um resumo de todas as validações que deram errado.

Além disso, podemos utilizar o `ValidationMessage` para exibir as mensagens de erro específicas de uma propriedade, em um local específico.

```
<ValidationMessage For="@(() => Model.Title)"/>
```

Podemos utilizar o `ValidationMessage` em conjunto com o `ValidationSummary`, que terá um resultado como mostrado abaixo.

• Título muito grande. **Validation Summary**

Título muito grande. **ValidationMessage**

Salvar

Context Binding

Existe uma propriedade no `EditForm` chamada de `EditContext`, que além do vínculo com uma **Model**, nos permite ter recursos adicionais no formulário.

Até o momento, fizemos a submissão do formulário através de um botão (**Tipo Submit**) e as validações foram aplicadas automaticamente.

Mas e se precisarmos aplicar estas validações em outro momento? E se precisarmos verificar se o formulário foi modificado ou não?

O `EditContext` nos ajuda nisso, pois além de fornecer acesso ao modelo, ele nos trás métodos e propriedades adicionais.

```
private EditContext? editContext;
```

Desta forma, o primeiro item que precisamos é um objeto do tipo `EditContext`, que será inicializado com algum modelo.

```
@code {
    private EditContext? editContext;

    public CreateProductViewModel Model { get; set; } = new();

    protected override void OnInitialized()
    {
        editContext = new(Model);
    }
}
```

Agora, além do modelo e possibilidade de vínculo com suas propriedades, temos os métodos adicionais mencionados anteriormente.

Podemos então vincular nosso formulário com o `EditContext` ao invés de vincular com o modelo diretamente.

```
<EditForm EditContext="@editContext" ...>
    ...
</EditForm>
```

Agora sempre que precisarmos saber se o formulário foi modificado, podemos utilizar o `IsModified` e caso queira executar a validação, chamamos o `Validate`.

```
_context.IsModified();
_context.Validate();
```

Existem outros métodos e propriedades no `EditContext` que podem ser explorados com mais calma durante seus estudos.

Form Submit

Outro detalhe interessante do componente `EditForm` é a possibilidade de ter diferentes ações (Na verdade reações) ao enviá-lo.

Isto por que existem três eventos que podemos mapear em nossos formulários, sendo eles:

- `OnValidSubmit` que é executado somente quando o formulário é válido.
- `OnInvalidSubmit` que é executado somente quando o formulário é inválido.
- `OnSubmit` que é executado independente do estado do formulário.

Caso opte pela última opção, você pode verificar se o formulário está válido chamando o `EditContext.Validate`.

Componentes nativos disponíveis

Abaixo está a lista de componentes disponíveis para uso nos formulários com o Blazor.

Input component	Renderizado como...
<code>InputCheckbox</code>	<code><input type="checkbox"></code>
<code>InputDate<TValue></code>	<code><input type="date"></code>
<code>InputFile</code>	<code><input type="file"></code>
<code>InputNumber<TValue></code>	<code><input type="number"></code>
<code>InputRadio<TValue></code>	<code><input type="radio"></code>
<code>InputRadioGroup<TValue></code>	Grupo de <code>InputRadio<TValue></code>
<code>InputSelect<TValue></code>	<code><select></code>
<code>InputText</code>	<code><input></code>
<code>InputTextArea</code>	<code><textarea></code>

Conclusão

Espero que você tenha se divertido aprendendo Blazor tanto quanto eu me diverti escrevendo este eBook.

O Blazor tem algum tempo de mercado mas ainda é considerada uma tecnologia emergente e podemos nos beneficiar disso saindo na frente com este conhecimento.

Onde aprender mais sobre Blazor

Temos uma carreira focada em [ASP.NET](https://balta.io/carreiras/aspnet-blazor-developer) e Blazor, com 4 cursos completos e mais de 12 horas de duração.

Carreira 03. ASP.NET/Blazor Developer

Aprenda criar APIs e aplicações web utilizando ASP.NET e Blazor



<https://balta.io/carreiras/aspnet-blazor-developer>



Recomendamos seguir todas as nossas carreiras na ordem, assim você aprenderá desde C# até Blazor, com o mesmo instrutor, mesma didática e muita mão na massa.

Carreiras

Comece sua carreira como desenvolvedor da forma correta!

 <https://balta.io/carreiras>

Obrigado ❤️