

Interagir com dados em aplicativo Web Blazor

Crie elementos de interface do usuário para um aplicativo Web usando componentes Blazor. Você também obterá dados e os exibirá para o usuário em páginas Web dinâmicas.

Objetivos de aprendizagem

Ao final deste módulo, você saberá como:

- Montar uma interface do usuário para um aplicativo Web criando componentes Blazor.
- Acessar dados para exibi-los em seu aplicativo Web.
- Compartilhar dados em seu aplicativo Web entre vários componentes Blazor.
- Associar um elemento HTML a uma variável em um componente Blazor.

Pré-requisitos

- Familiaridade com HTML e CSS para desenvolvimento na Web
- Capacidade de escrever código C# em um nível de iniciante
- Capacidade de criar um aplicativo Web Blazor

Este módulo faz parte destes roteiros de aprendizagem

- Criar aplicativos Web com o Blazor
 - Introdução 3 min
 - Criar uma interface do usuário com componentes Blazor 7 min
 - Exercício: criar uma interface do usuário com componentes Blazor 5 min
 - Acessar dados de um componente Blazor 7 min
 - Exercício: acessar dados de um componente Blazor 5 min
 - Compartilhar dados em aplicativos Blazor 7 min
 - Exercícios: compartilhar dados em aplicativos Blazor 6 min
 - Associar controles aos dados em aplicativos Blazor 7 min
 - Exercício: associar controles aos dados em aplicativos Blazor 5 min
 - Resumo 7 min

Introdução

O Blazor cria aplicativos Web interativos usando o código .NET, que permite o compartilhamento da lógica do aplicativo no lado do servidor e do cliente, sem a complexidade de gerenciar bibliotecas JavaScript do lado do cliente.

Suponha que você tenha sido contratado por uma empresa de entrega de pizzas para modernizar o site da empresa voltado para o cliente. Você recebeu modelos de páginas da Web dos designers gráficos e discutiu detalhadamente a funcionalidade do site com todos os stakeholders. Agora, você quer começar a criar o site com as páginas principais de pesquisa de pizza. Sua equipe trabalha com C# há muitos anos e tem menos experiência com JavaScript. Por isso, você quer escrever o máximo de código possível em .NET. Em módulos posteriores neste roteiro de aprendizagem, você criará as páginas de autenticação e de finalização de compra. Neste módulo, você aprenderá sobre os componentes Blazor, bem como usá-los para criar uma interface do usuário que exiba dados dinâmicos.

Objetivos de aprendizagem

Ao final deste módulo, você saberá como:

- Montar uma interface do usuário para um aplicativo Web criando componentes Blazor.
- Acessar dados para exibi-los em seu aplicativo Web.
- Compartilhar dados em seu aplicativo Web entre vários componentes Blazor.
- Associar um elemento HTML a uma variável em um componente Blazor.

Criar uma interface do usuário com componentes Blazor

Com os componentes Blazor, você pode definir páginas Web ou partes de HTML que incluem conteúdo dinâmico usando o código .NET. No Blazor, é possível formular conteúdo dinâmico usando C#, em vez de usar JavaScript.

Suponha que você esteja trabalhando para criar um novo site moderno para uma empresa de entrega de pizza. Você começa com uma página de boas-vindas, que se tornará a página de aterrissagem para a maioria dos usuários do site. Sua ideia é exibir ofertas especiais e pizzas populares nessa página.

Nesta unidade, você aprenderá a criar componentes no Blazor e escrever códigos que renderizam conteúdo dinâmico nesses componentes.

Entender os componentes Blazor

Blazor é uma estrutura que os desenvolvedores podem usar para criar uma interface do usuário sofisticada e interativa escrevendo código em C#. Com o Blazor, você pode usar a mesma linguagem em todo o seu código, tanto no lado do servidor quanto no lado do cliente, e renderizá-lo para exibição em navegadores diversos, incluindo navegadores em dispositivos móveis.

📌 Observação

Há dois modelos de hospedagem de código em aplicativos Blazor:

- **Blazor Server:** neste modelo, o aplicativo é executado no servidor Web dentro de um aplicativo ASP.NET Core. Atualizações de interface do usuário, eventos e chamadas JavaScript no lado do cliente são enviados por meio de uma conexão de **SignalR** entre o cliente e o servidor. Neste módulo, vamos discutir e codificar para esse modelo.
- **Blazor WebAssembly:** neste modelo, o aplicativo Blazor, suas dependências e o runtime do .NET são baixados e executados no navegador.

No Blazor, você compila a interface do usuário de partes autocontidas de código chamadas *componentes*. Cada componente pode conter uma combinação de código HTML e C#. Os componentes são escritos usando a *sintaxe Razor*, em que o código é marcado com a diretiva `@code`. Outras diretivas podem ser usadas para acessar variáveis, se vincular a valores e realizar outras tarefas de renderização. Quando o aplicativo é compilado, o HTML e o código são compilados em uma classe de componente. Os componentes são gravados como arquivos com uma extensão `.razor`.

📌 Observação

A sintaxe Razor é usada para inserir código .NET em páginas da Web. Você pode usá-la em aplicativos MVC ASP.NET, em que os arquivos têm uma extensão `.cshtml`. A sintaxe Razor é usada no Blazor para gravar componentes. Em vez disso, esses componentes têm a extensão `.razor` e não há nenhuma separação estrita entre controladores e exibições.

Veja um exemplo simples de um componente Blazor:

```
razor                                                                    Copiar

@page "/index"

<h1>Welcome to Blazing Pizza</h1>

<p>@welcomeMessage</p>

@code {
    private string welcomeMessage = "However you like your pizzas, we can deliver them blazing fast!";
}
```

Nesse exemplo, o código define o valor de uma variável de cadeia de caracteres chamada `welcomeMessage`. Essa variável é renderizada dentro de tags `<p>` no HTML. Examinaremos exemplos mais complexos mais adiante nesta unidade.

```
Bash                                                                    Copiar

dotnet new blazorserver -o BlazingPizzaSite -f net6.0
```

Os componentes padrão incluem a página inicial `Index.razor` e o componente de demonstração `Counter.razor`. Ambos os componentes são colocados na pasta `Páginas`. Você pode modificar essas exibições para atender às suas necessidades ou excluí-las e substituí-las por novos componentes.

Para adicionar um novo componente a um aplicativo Web existente, use este comando:

📌 Importante

O nome de um componente Blazor deve começar com um caractere maiúsculo.

Depois de criar o componente, você poderá abri-lo para edição no Visual Studio Code:

```
Bash Copiar
code Pages/PizzaBrowser
```

Escrever código em um componente Blazor

Ao compilar uma interface do usuário no Blazor, você misturará a marcação HTML estática e CSS com código C#, geralmente no mesmo arquivo. Para diferenciar esses tipos de código, use a sintaxe Razor. A sintaxe Razor inclui diretivas, prefixadas com o símbolo @, que delimitam o código C#, os parâmetros de roteamento, os dados vinculados, as classes importadas e outros recursos.

Vamos considerar esse exemplo de componente novamente:

```
razor Copiar
@page "/index"

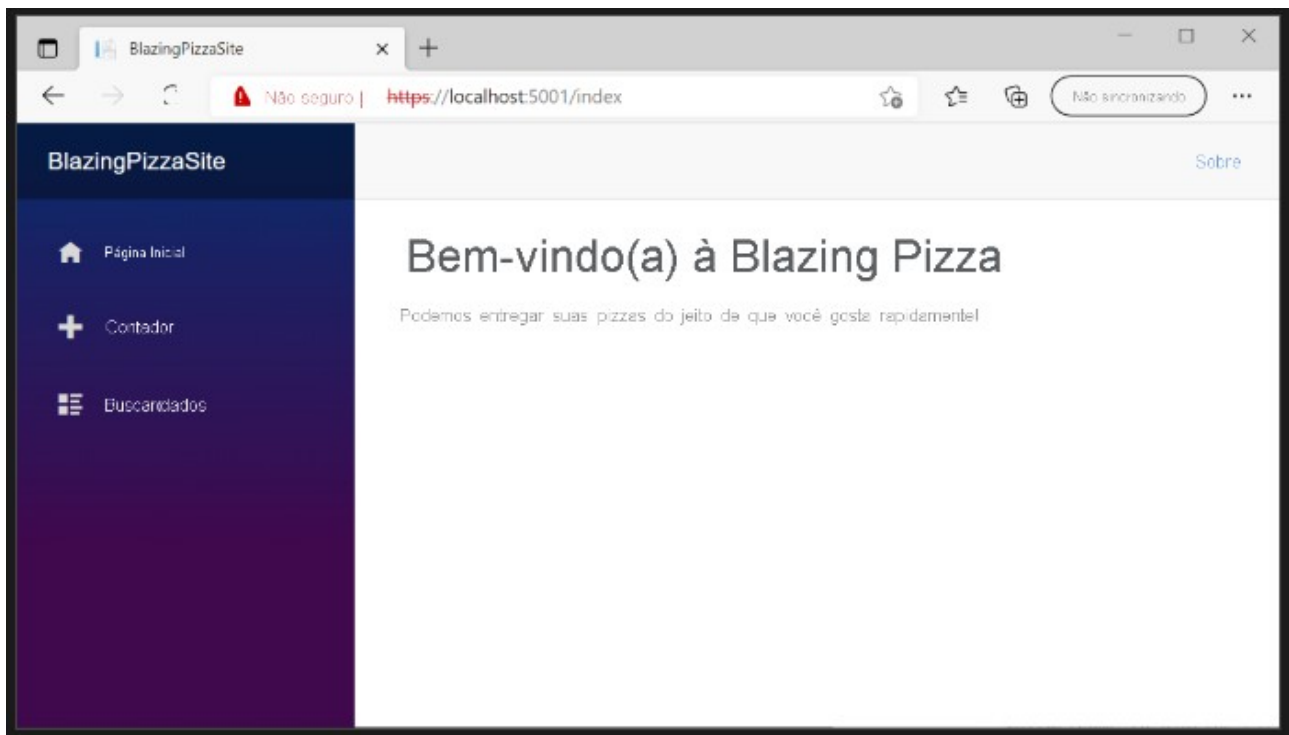
<h1>Welcome to Blazing Pizza</h1>

<p>@welcomeMessage</p>

@code {
    private string welcomeMessage = "However you like your pizzas, we can deliver them fast!";
}
```

Você pode reconhecer a marcação HTML com as tags <h1> e <p>. Essa marcação é a estrutura estática da página, na qual seu código inserirá o conteúdo dinâmico. A marcação Razor consiste em:

- A diretiva @page: esta diretiva fornece um modelo de rota para o Blazor. Em runtime, o Blazor localiza uma página a ser renderizada por meio da correspondência desse modelo à URL solicitada pelo usuário. Nesse caso, ele pode corresponder a uma URL no formato `http://yourdomain.com/index`.
- A diretiva @code: esta diretiva declara que o texto no bloco a seguir é um código C#. Você pode colocar quantos blocos de código precisar em um componente. Você pode definir membros de classe de componente nesses blocos de código e definir seus valores de cálculo, operações de pesquisa de dados ou outras fontes. Nesse caso, o código define um único membro de componente chamado `welcomeMessage` e define seu valor de cadeia de caracteres.
- Diretivas de acesso de membro: se você quiser incluir o valor de um membro em sua lógica de renderização, use o símbolo @ seguido por uma expressão C#, como o nome do membro. Nesse caso, a diretiva `@welcomeMessage` é usada para renderizar o valor do membro `welcomeMessage` nas tags <p>.



Exercício: criar uma interface do usuário com componentes Blazor

Neste exercício, você começará a criar um novo aplicativo Blazing Pizza para a empresa de entrega de pizza. A empresa forneceu o CSS, as imagens e o HTML atuais do site antigo para o trabalho.

Observação

🕒 Observação

Este módulo usa a CLI do .NET e o Visual Studio Code para o desenvolvimento local. Depois de concluir este módulo, você poderá aplicar os conceitos usando o Visual Studio para Windows ou o Visual Studio para Mac (macOS). Para o desenvolvimento contínuo, você pode usar o Visual Studio Code para Windows, Linux e macOS.

Se você ainda não criou um aplicativo Blazor, siga as instruções de instalação do Blazor para instalar a versão correta do .NET e verificar se o computador está configurado corretamente. Pare na etapa Criar seu aplicativo.

Criar um aplicativo Blazor

Este módulo usa o SDK do .NET 6.0. Verifique se tem o .NET 6.0 instalado, executando o seguinte comando em seu terminal preferido:

```
CLI do .NET Copiar

dotnet --list-sdks
```

Saída semelhante à seguinte exibida:

```
Console
3.1.100 [C:\program files\dotnet\sdk]
5.0.100 [C:\program files\dotnet\sdk]
6.0.100 [C:\program files\dotnet\sdk]
```

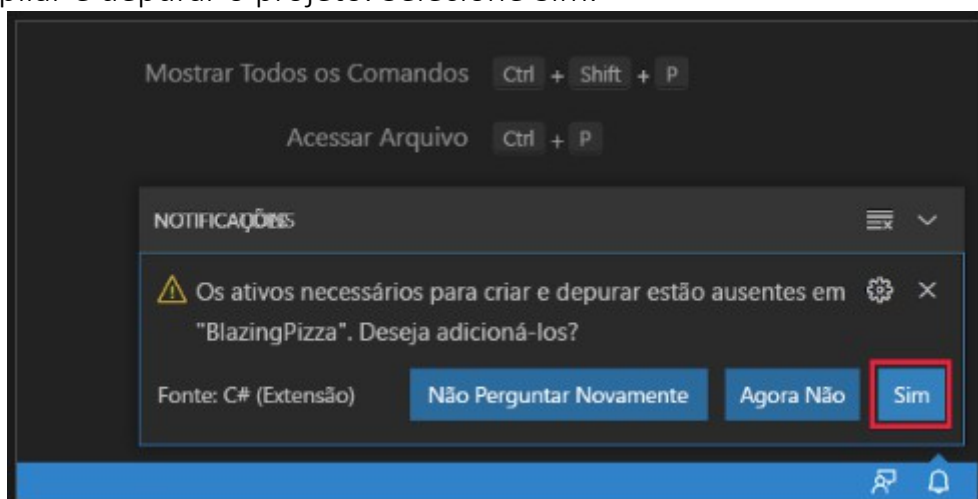
Verifique se uma versão que começa com 6 está listada. Se nenhum estiver listado ou o comando não for encontrado, instale o SDK do .NET 6.0 mais recente.

O .NET permite que você crie novos projetos com qualquer versão do Visual Studio ou comandos de terminal. Os exercícios a seguir mostram as etapas usando o terminal e o Visual Studio Code.

1. Abra o Visual Studio Code.
2. Abra o terminal integrado no Visual Studio Code selecionando Exibir. Em seguida, no menu principal, escolha Terminal.
3. No terminal, vá para onde você deseja criar o projeto.
4. Execute o comando de terminal do dotnet:

```
CLI do .NET
dotnet new blazorserver -o BlazingPizza --no-https true -f net6.0
```

1. Este comando cria um novo projeto de servidor Blazor em uma pasta chamada BlazingPizza. Ele também informa o projeto para desabilitar o HTTPS.
2. Selecione Arquivo>Abrir pasta.
3. Na caixa de diálogo Abrir, vá para a pasta BlazingPizza e escolha Selecionar Pasta.
4. O Visual Studio Code solicita a você a adição dos ativos necessários para compilar e depurar o projeto. Selecione Sim.



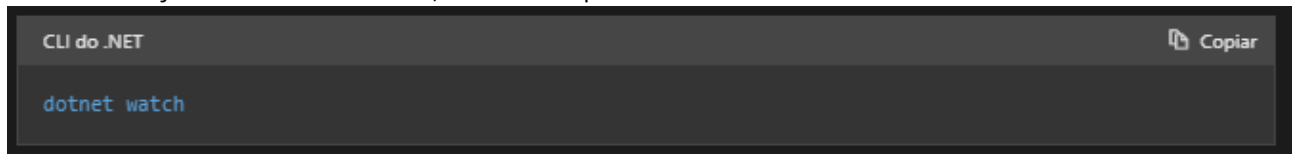
1. O Visual Studio Code adiciona launch.json e tasks.json na pasta .vscode do seu projeto.

Esses arquivos permitem que você execute e depure seu aplicativo Blazor com as ferramentas de depuração do Visual Studio Code.

Testar a configuração

Você pode optar por usar o terminal ou o Visual Studio Code para executar seu aplicativo.

1. Na janela do terminal, inicie o aplicativo Blazor com:



1. Este comando compila e inicia o aplicativo. O comando watch informa ao dotnet para inspecionar todos os seus arquivos de projeto. As alterações feitas nos arquivos de projetos disparam automaticamente uma recompilação e reiniciam seu aplicativo.

O navegador padrão de seus computadores deve abrir uma nova página em <http://localhost:5000>.

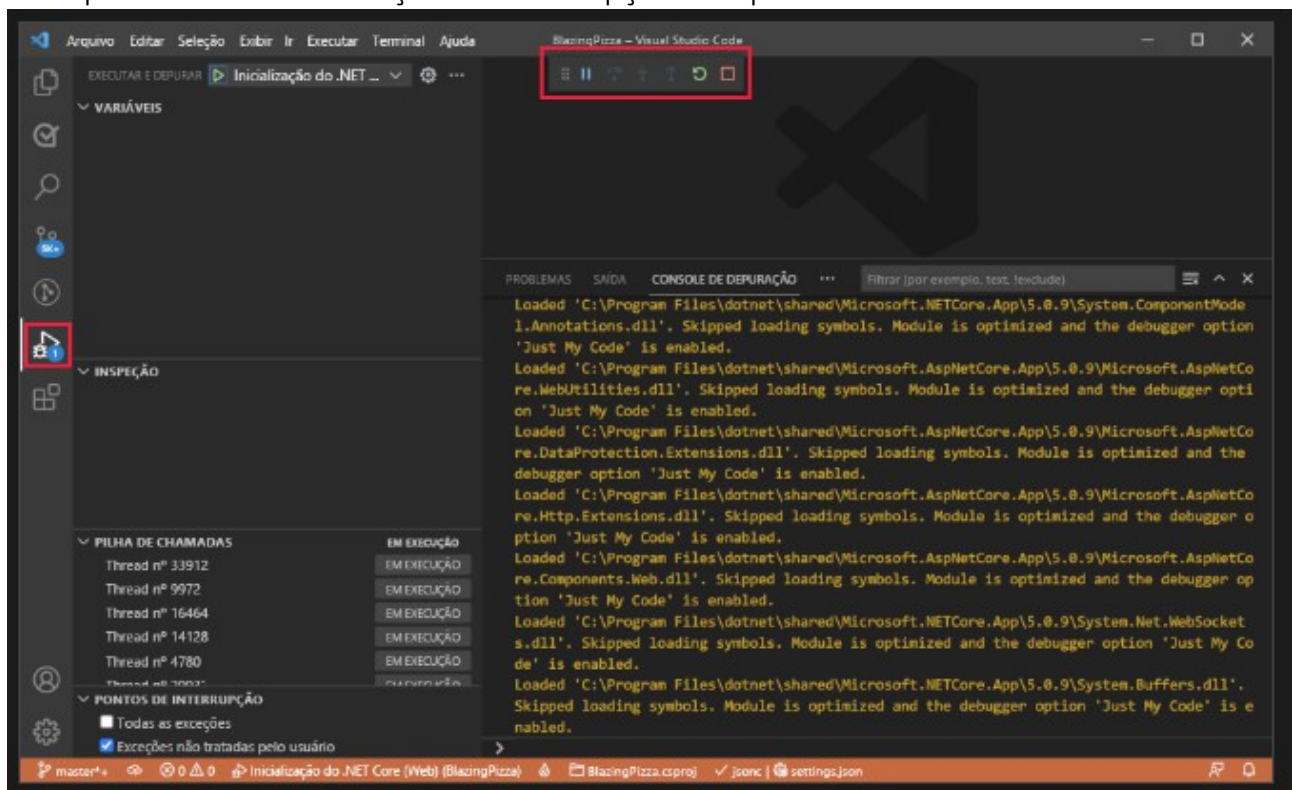
2. Para interromper o aplicativo, selecione Ctrl + C na janela do terminal.

Você também pode executar e depurar seu projeto com o Visual Studio Code.

1. No Visual Studio Code, selecione F5. Ou, então, no menu Executar, selecione Iniciar Depuração.

O aplicativo deve compilar e abrir uma nova página do navegador.

2. O Visual Studio Code também muda para a janela Executar e Depurar, que permite a reinicialização ou interrupção do aplicativo.

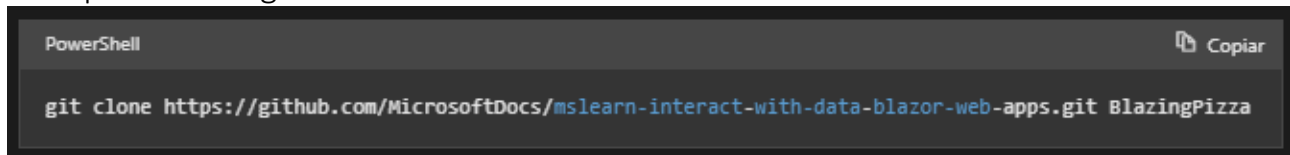


1. Selecione Shift + F5 para interromper o aplicativo.

Baixar os ativos de Blazing Pizza e os arquivos iniciais

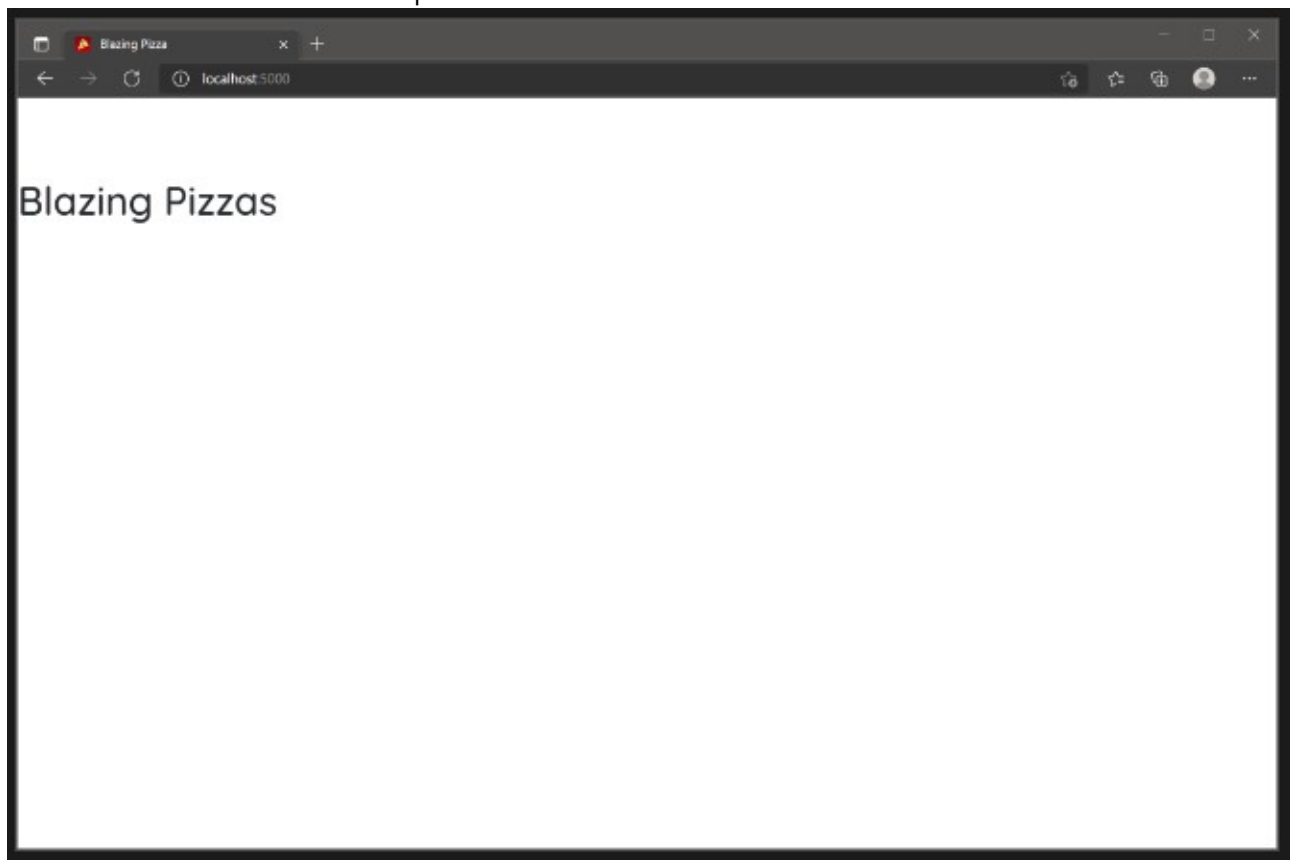
Agora, você clonará os arquivos de projeto do aplicativo Blazor existentes de suas equipes do repositório do GitHub.

- 1.Exclua sua pasta BlazingPizza usando o explorador de arquivos ou no Visual Studio Code.
- 2.No terminal, clone os arquivos de trabalho atuais em uma nova pasta BlazingPizza.

A screenshot of a PowerShell terminal window. The title bar says "PowerShell" and there is a "Copiar" button. The command entered is `git clone https://github.com/MicrosoftDocs/mslearn-interact-with-data-blazor-web-apps.git BlazingPizza`.

```
PowerShell Copiar
git clone https://github.com/MicrosoftDocs/mslearn-interact-with-data-blazor-web-apps.git BlazingPizza
```

Execute a versão atual do aplicativo. Selecione F5.



Faça algumas pizzas

O componente Pages/Index.razor permite que os clientes selecionem e configurem as pizzas que desejam solicitar. O componente responde à URL raiz do aplicativo.

A equipe também criou classes para representar os modelos no aplicativo. Examine o modelo PizzaSpecial atual.

- 1.No Visual Studio Code, no explorador de arquivos, expanda a pasta Modelo. Selecione PizzaSpecial.

C#

 Copiar

```
namespace BlazingPizza;

/// <summary>
/// Represents a pre-configured template for a pizza a user can order
/// </summary>
public class PizzaSpecial
{
    public int Id { get; set; }

    public string Name { get; set; }

    public decimal BasePrice { get; set; }

    public string Description { get; set; }

    public string ImageUrl { get; set; }

    public string GetFormattedBasePrice() => BasePrice.ToString("0.00");
}
```

1. Observe que um pedido de pizza tem Name, BasePrice, Description e ImageUrl.
2. No explorador de arquivos, expanda Páginas e selecione Index.razor.

razor

 Copiar

```
@page "/"

<h1>Blazing Pizzas</h1>
```

1. No momento, há apenas uma única tag H1 para o título. Você adicionará código para criar pizzas especiais.
2. Na tag <h1>, adicione este código C#:

razor

 Copiar

```
@code {
    List<PizzaSpecial> specials = new();

    protected override void OnInitialized()
    {
        specials.AddRange(new List<PizzaSpecial>
        {
            new PizzaSpecial { Name = "The Baconatorazor", BasePrice = 11.99M, Description = "It has EVE",
            new PizzaSpecial { Name = "Buffalo chicken", BasePrice = 12.75M, Description = "Spicy chicke",
            new PizzaSpecial { Name = "Veggie Delight", BasePrice = 11.5M, Description = "It's like sala",
            new PizzaSpecial { Name = "Margherita", BasePrice = 9.99M, Description = "Traditional Italia",
            new PizzaSpecial { Name = "Basic Cheese Pizza", BasePrice = 11.99M, Description = "It's chee",
            new PizzaSpecial { Name = "Classic pepperoni", BasePrice = 10.5M, Description = "It's the pi",
        });
    }
}
```

1. O bloco @code cria uma matriz para conter as pizzas especiais. Quando a página é inicializada, ela adiciona seis pizzas à matriz.

2.Selecione F5 ou Executar. Selecione Iniciar Depuração.

O aplicativo será compilado e executado e você verá que nada mudou. O código não está sendo usado por nada no HTML do lado do cliente. Vamos corrigir isso.

3.Selecione Shift + F5 ou selecione Parar Depuração.

4.Em Index.razor, substitua <h1>Blazing Pizzas</h1> por este código:

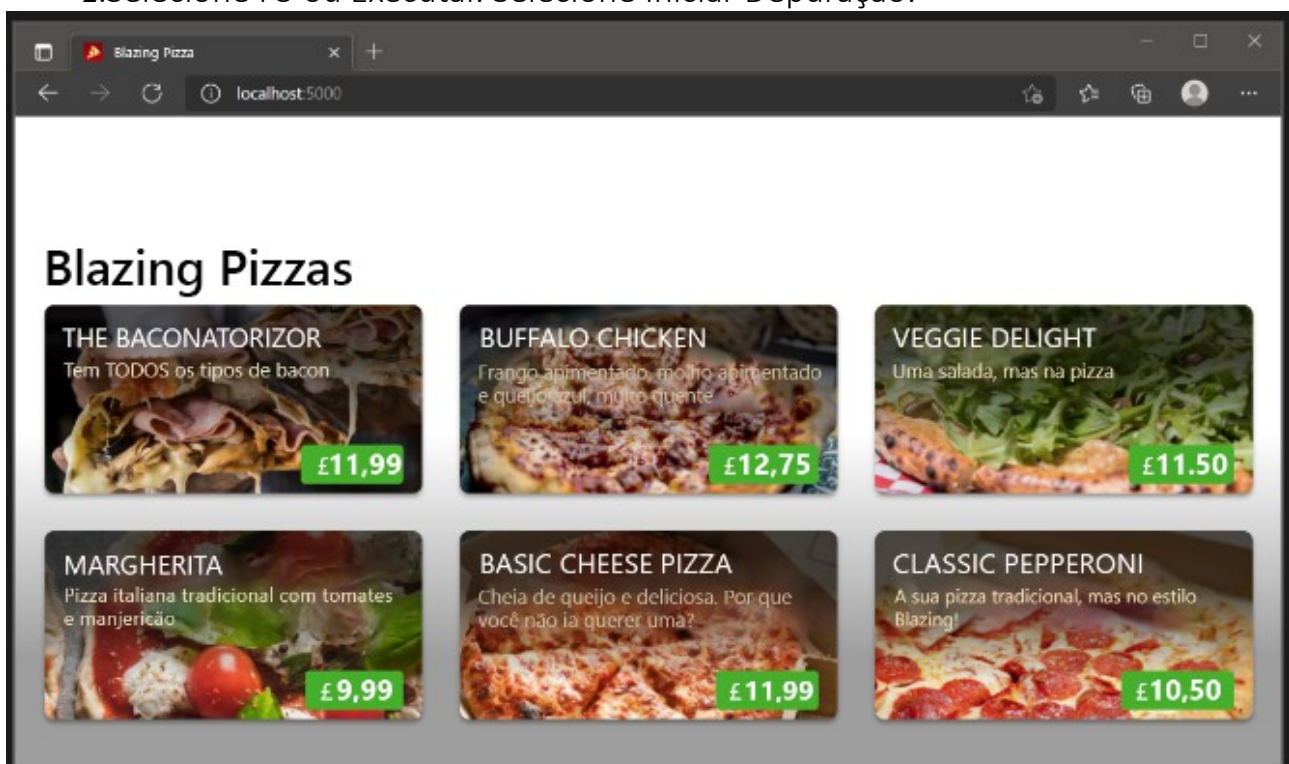
```
razor Copiar

<div class="main">
  <h1>Blazing Pizzas</h1>
  <ul class="pizza-cards">
    @if (specials != null)
    {
      @foreach (var special in specials)
      {
        <li style="background-image: url('@special.ImageUrl')">
          <div class="pizza-info">
            <span class="title">@special.Name</span>
            @special.Description
            <span class="price">@special.GetFormattedBasePrice()</span>
          </div>
        </li>
      }
    }
  </ul>
</div>
```

1.Esse código combina HTML sem formatação com diretivas de loop e de acesso de membro. O loop @foreach cria uma tag para cada pizza na matriz specials.

Dentro do loop, cada pizza especial exibe seu nome, descrição, preço e imagem com diretivas de membro.

2.Selecione F5 ou Executar. Selecione Iniciar Depuração.



Agora você tem um componente base de pizza para permitir que os clientes peçam uma pizza. Você melhorará esse componente nos exercícios a seguir.

Acessar dados de um componente Blazor

Os sites envolvidos precisam exibir o conteúdo dinâmico que pode ser alterado o tempo todo. A obtenção de dados de uma fonte dinâmica, como um banco de dados ou serviço Web, é uma técnica fundamental no desenvolvimento para a Web.

Suponha que você esteja trabalhando no site atualizado voltado para o cliente de uma empresa de entrega de pizza. Você tem uma variedade de páginas da Web criadas e projetadas como componentes Blazor. Agora, você quer preencher essas páginas com informações sobre pizzas, recheios e pedidos que serão obtidas de um banco de dados.

Nesta unidade, você aprenderá a acessar dados e renderizá-los na marcação HTML para exibição ao usuário.

Como criar um serviço de dados registrado

Se você quiser criar um site dinâmico que mostra informações variadas para os usuários, escreva o código para obter esses dados de algum lugar. Por exemplo, suponha que você tenha um banco de dados que armazena todas as pizzas vendidas por sua empresa. Como as pizzas estão sempre mudando, é uma má ideia codificá-las no HTML do site. Em vez disso, use o código C# e o Blazor para consultar o banco de dados e, em seguida, formatar os detalhes como HTML, para que o usuário possa escolher a preferida.

Em um aplicativo Blazor Server, é possível criar um serviço registrado para representar uma fonte de dados e obter dados dele.

🕒 Observação

As fontes de dados que você pode usar em um aplicativo Blazor incluem bancos de dados relacionais, bancos de dados NoSQL, serviços Web, vários serviços do Azure e muitos outros sistemas. É possível usar tecnologias .NET para consultar essas fontes, como Entity Framework, clientes HTTP e ODBC. Essas técnicas estão além do escopo deste módulo. Aqui, você aprenderá a formatar e a usar dados obtidos de uma dessas fontes e tecnologias.

A criação de um serviço registrado começa escrevendo uma classe que define suas propriedades. Veja um exemplo que você pode escrever para representar uma pizza:

```
C# Copiar

namespace BlazingPizza.Data;

public class Pizza
{
    public int PizzaId { get; set; }

    public string Name { get; set; }

    public string Description { get; set; }

    public decimal Price { get; set; }

    public bool Vegetarian { get; set; }

    public bool Vegan { get; set; }
}
```

A classe define as propriedades e os tipos de dados da pizza. Você deve verificar se essas propriedades correspondem ao esquema de pizza na fonte de dados. Faz sentido criar essa classe na pasta Dados do seu projeto e usar um namespace de membro chamado Dados. Se preferir, você pode escolher outras pastas e namespaces.

Em seguida, você definiria o serviço:

```
C# Copiar

namespace BlazingPizza.Data;

public class PizzaService
{
    public Task<Pizza[]> GetPizzasAsync()
    {
        // Call your data access technology here
    }
}
```

Observe que o serviço usa uma chamada assíncrona para acessar dados e retornar uma coleção de objetos Pizza. A fonte de dados pode ser remota do servidor em que o código Blazor está em execução. Nesse caso, use uma chamada assíncrona. Se a fonte de dados responder lentamente, outro código poderá continuar a ser executado enquanto você aguarda a resposta.

Você também deve registrar o serviço adicionando uma linha à seção Add Services to the container no arquivo Program.cs:

```
C# Copiar

...
// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
// Register the pizzas service
builder.Services.AddSingleton<PizzaService>();
...
```

Como usar um serviço para obter dados

Agora use o serviço que você definiu chamando-o em um componente Blazor e obtendo dados. Vamos supor que você tenha o seguinte código de componente e queira exibir pizzas nele:

```
razor Copiar

@page "/pizzas"

<h1>Choose your pizza</h1>

<p>We have all these delicious recipes:</p>
```

Como injetar o serviço

Antes de chamar o serviço do componente, use a injeção de dependência para adicionar o serviço. Injete o serviço adicionando o seguinte código após a diretiva @page:

```
razor Copiar

@using BlazingPizza.Data
@inject PizzaService PizzaSvc
```

Normalmente, o componente e o serviço estão em membros diferentes do namespace. Portanto, você deve incluir a diretiva @using. Essa diretiva funciona da mesma maneira que uma instrução using na parte superior de um arquivo de código C#. A diretiva @inject adiciona o serviço ao componente atual e inicia uma instância dele. Na diretiva, especifique o nome da classe de serviço. Siga pelo nome que você deseja usar para a instância do serviço neste componente.

Substituir o método OnInitializedAsync

Um bom lugar para chamar o serviço e obter dados é no método OnInitializedAsync. Esse evento dispara após a conclusão da inicialização do componente e após ele ter recebido parâmetros iniciais, mas antes de a página ser renderizada e exibida para o usuário. O evento é definido na classe base do componente Blazor. Você pode substituí-lo em um bloco de código como neste exemplo:

C#

Copiar

```
protected override async Task OnInitializedAsync()
{
    \\ Call the service here
}
```

Chamar o serviço para obter dados

Ao chamar o serviço, use a palavra-chave await, pois a chamada é assíncrona:

C#

Copiar

```
private Pizza[] todaysPizzas;

protected override async Task OnInitializedAsync()
{
    todaysPizzas = await PizzaSvc.GetPizzasAsync();
}
```

Como exibir dados para o usuário

Depois de obter alguns dados do serviço, convém exibi-los para o usuário. No exemplo de pizzas, esperamos que o serviço retorne uma lista de pizzas para escolha dos usuários. O Blazor inclui um conjunto avançado de diretivas que você pode usar para inserir esses dados na página visto pelo usuário.

Como verificar se há dados

Primeiro, determine o que a página exibe antes que as pizzas sejam carregadas. Podemos fazer isso verificando se a coleção todaysPizzas é null. Para executar o código de renderização condicional em um componente Blazor, use a diretiva @if:

razor

Copiar

```
@if (todaysPizzas == null)
{
    <p>We're finding out what pizzas are available today...</p>
}
else
{
    <!-- This markup will be rendered once the pizzas are loaded -->
}
```

A diretiva @if renderizará a marcação em seu primeiro bloco de código somente se a expressão C# retornar true. Você também poderá usar um bloco de código else if para executar outros testes e renderizar a marcação, se eles forem verdadeiros. Por fim, é possível especificar um bloco de código else para renderizar o código se nenhuma das condições anteriores tiver retornado true. Ao verificar se há null no bloco de código @if, você garantirá que o Blazor não tentará exibir detalhes da pizza antes que os dados sejam obtidos do serviço.

Observação

O Blazor também inclui a diretiva `@switch` para renderização de marcação com base em um teste que pode retornar vários valores. A diretiva `@switch` funciona de forma semelhante à instrução `switch` do C#.

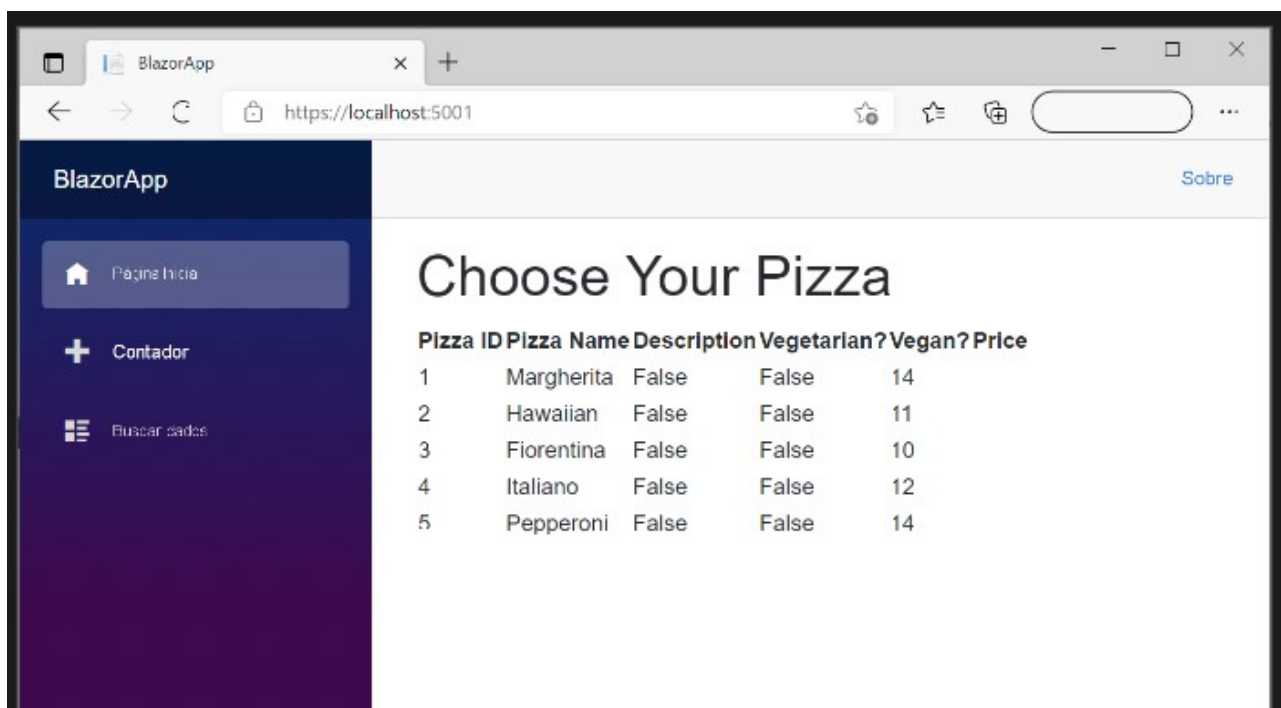
Como renderizar uma coleção de objetos

Se o Blazor executar a instrução else no código anterior, você saberá que algumas pizzas foram obtidas do serviço. A próxima tarefa é exibir essas pizzas para o usuário. Vamos examinar como exibir os dados em uma tabela HTML simples.

Não sabemos quantas pizzas estarão disponíveis quando codificarmos esta página. Podemos usar a diretiva `@foreach` para executar um loop em todos os objetos da coleção `todayPizzas` e renderizar uma linha para cada um:

```
razor Copiar

<table>
  <thead>
    <tr>
      <th>Pizza Name</th>
      <th>Description</th>
      <th>Vegetarian?</th>
      <th>Vegan?</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var pizza in todaysPizzas)
    {
      <tr>
        <td>@pizza.Name</td>
        <td>@pizza.Description</td>
        <td>@pizza.Vegetarian</td>
        <td>@pizza.Vegan</td>
        <td>@pizza.Price</td>
      </tr>
    }
  </tbody>
</table>
```



Verificar seu conhecimento

1.Qual manipulador de eventos Blazor é um bom local para buscar dados?

☐PageLoad

☐OnInitializedAsync

☐OnAfterRenderAsync

2.Qual diretiva do Blazor você deve usar para trabalhar com um serviço de acesso a dados em uma página do Blazor?

☐@inject

☐@page

☐@using

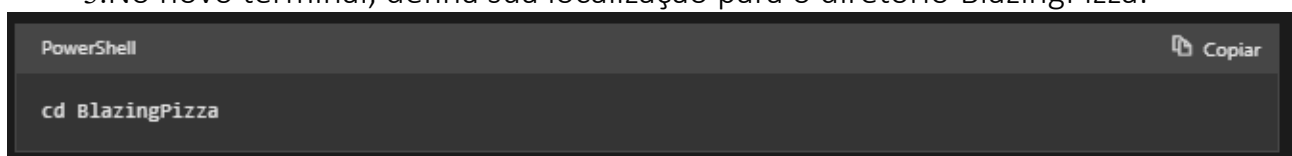
Exercício: acessar dados de um componente Blazor

As pizzas atuais codificadas no aplicativo precisam ser substituídas por um banco de dados. Você pode usar o Microsoft Entity Framework para adicionar conexões a fontes de dados. Neste aplicativo, usaremos um banco de dados SQLite para armazenar as pizzas.

Nesse exercício, você adicionará pacotes para dar suporte à funcionalidade do banco de dados, conectará nossas classes a um banco de dados de back-end e adicionará uma classe auxiliar para pré-carregar dados das pizzas da empresa.

Adicionar pacotes para dar suporte ao acesso ao banco de dados

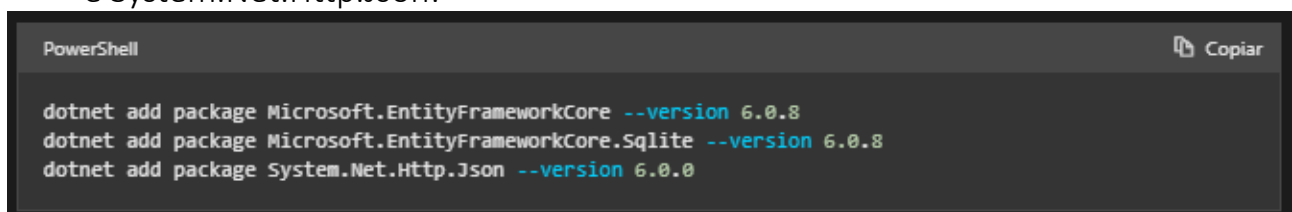
- 1.Interrompa o aplicativo se ele ainda estiver em execução.
- 2.No Visual Studio Code, selecione Terminal>Novo Terminal.
- 3.No novo terminal, defina sua localização para o diretório BlazingPizza.

A screenshot of a PowerShell terminal window. The title bar says "PowerShell" and there is a "Copiar" button. The command "cd BlazingPizza" has been entered and executed.

```
PowerShell
cd BlazingPizza
```

cd BlazingPizza

- 1.Execute estes comandos para adicionar os pacotes Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Sqlite e System.Net.Http.Json:

A screenshot of a PowerShell terminal window. The title bar says "PowerShell" and there is a "Copiar" button. Three commands are entered to add NuGet packages.

```
PowerShell
dotnet add package Microsoft.EntityFrameworkCore --version 6.0.8
dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version 6.0.8
dotnet add package System.Net.Http.Json --version 6.0.0
```

Estes comandos adicionam referências de pacote ao arquivo BlazingPizza.csproj:


```
XML Copiar

<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.8" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="6.0.8" />
  <PackageReference Include="System.Net.Http.Json" Version="6.0.0" />
</ItemGroup>
```

Adicionar um contexto de banco de dados

- 1.No Visual Studio Code, crie uma nova pasta em BlazingPizza. Nomeie-a Dados.
- 2.Crie um arquivo dentro da pasta de Dados. Nomeie-o como PizzaStoreContext.cs.
- 3.Insira este código para a classe:

```
C# Copiar

using Microsoft.EntityFrameworkCore;

namespace BlazingPizza.Data;

public class PizzaStoreContext : DbContext
{
    public PizzaStoreContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<PizzaSpecial> Specials { get; set; }
}
```

Essa classe cria um contexto de banco de dados que podemos usar para registrar um serviço de banco de dados. O contexto também nos permite ter um controlador que acessa o banco de dados.

- 1.Salve suas alterações.

Adicionar um controlador

- 1.Crie uma nova pasta BlazingPizza. Nomeie-a como Controladores.
- 2.Crie um novo arquivo na pasta Controladores. Nomeie-o como SpecialsController.cs.
- 3.Insira este código para a classe:

C#

 Copiar

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using BlazingPizza.Data;

namespace BlazingPizza.Controllers;

[Route("specials")]
[ApiController]
public class SpecialsController : Controller
{
    private readonly PizzaStoreContext _db;

    public SpecialsController(PizzaStoreContext db)
    {
        _db = db;
    }

    [HttpGet]
    public async Task<ActionResult<List<PizzaSpecial>>> GetSpecials()
    {
        return (await _db.Specials.ToListAsync()).OrderByDescending(s => s.BasePrice).ToList();
    }
}
```

Essa classe cria um controlador que permite consultar o banco de dados em busca de pizzas especiais e retorná-las como JSON na URL (<http://localhost:5000/specials>).

1. Salve suas alterações.

Carregar dados no banco de dados

O aplicativo verifica se há um banco de dados SQLite existente e cria um com algumas pizzas pré-preparadas.

1. Crie um novo arquivo no diretório de Dados. Nomeie-o como SeedData.cs.

2. Insira este código para a classe:

C#Copiar

1. namespace BlazingPizza.Data;

```
public static class SeedData
{
    public static void Initialize(PizzaStoreContext db)
    {
        var specials = new PizzaSpecial[]
        {
            new PizzaSpecial()
            {
                Name = "Basic Cheese Pizza",
```

```
        Description = "It's cheesy and delicious. Why wouldn't you want one?",
        BasePrice = 9.99m,
        ImageUrl = "img/pizzas/cheese.jpg",
    },
    new PizzaSpecial()
    {
        Id = 2,
        Name = "The Baconatorizer",
        Description = "It has EVERY kind of bacon",
        BasePrice = 11.99m,
        ImageUrl = "img/pizzas/bacon.jpg",
    },
    new PizzaSpecial()
    {
        Id = 3,
        Name = "Classic pepperoni",
        Description = "It's the pizza you grew up with, but Blazing hot!",
        BasePrice = 10.50m,
        ImageUrl = "img/pizzas/pepperoni.jpg",
    },
    new PizzaSpecial()
    {
        Id = 4,
        Name = "Buffalo chicken",
        Description = "Spicy chicken, hot sauce and bleu cheese, guaranteed to
warm you up",
        BasePrice = 12.75m,
        ImageUrl = "img/pizzas/meaty.jpg",
    },
    new PizzaSpecial()
    {
```

```

        Id = 5,
        Name = "Mushroom Lovers",
        Description = "It has mushrooms. Isn't that obvious?",
        BasePrice = 11.00m,
        ImageUrl = "img/pizzas/mushroom.jpg",
    },
    new PizzaSpecial()
    {
        Id = 7,
        Name = "Veggie Delight",
        Description = "It's like salad, but on a pizza",
        BasePrice = 11.50m,
        ImageUrl = "img/pizzas/salad.jpg",
    },
    new PizzaSpecial()
    {
        Id = 8,
        Name = "Margherita",
        Description = "Traditional Italian pizza with tomatoes and basil",
        BasePrice = 9.99m,
        ImageUrl = "img/pizzas/margherita.jpg",
    },
};

db.Specials.AddRange(specials);
db.SaveChanges();
}
}

```

A classe usa um contexto de banco de dados passado, cria alguns objetos `PizzaSpecial` em uma matriz e os salva.

2.No explorador de arquivos, selecione `Program.cs`.

3.Na parte superior, adicione uma referência a um novo `PizzaStoreContext`:

```
C# Copiar

using BlazingPizza.Data;
```

Essa instrução permite que o aplicativo use o novo serviço.

1. Insira este segmento acima do método `app.Run();`:

```
C# Copiar

...
// Initialize the database
var scopeFactory = app.Services.GetRequiredService<IServiceScopeFactory>();
using (var scope = scopeFactory.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<PizzaStoreContext>();
    if (db.Database.EnsureCreated())
    {
        SeedData.Initialize(db);
    }
}

app.Run();
```

Essa alteração cria um escopo do banco de dados com o `PizzaStoreContext`. Se não houver um banco de dados já criado, ele chamará a classe estática `SeedData` para criar um.

1. No momento, o aplicativo não funciona porque não inicializamos o `PizzaStoreContext`. Na seção superior `Add Services to the container` do arquivo `Program.cs`, adicione este código nos serviços atuais (as linhas que iniciam `builder.Services`):

```
C# Copiar

builder.Services.AddHttpClient();
builder.Services.AddSqlite<PizzaStoreContext>("Data Source=pizza.db");
```

Esse código registra dois serviços. A primeira instrução `AddHttpClient` permite que o aplicativo acesse comandos HTTP. O aplicativo usa um `HttpClient` para obter o JSON para pizzas especiais. A segunda instrução registra o novo `PizzaStoreContext` e fornece o nome do arquivo para o banco de dados SQLite.

Usar o banco de dados para exibir pizzas

Agora, podemos substituir a pizza codificada na página `index.razor`.

1. No explorador de arquivos, selecione `Index.razor`.
2. Substitua o método `OnInitialized()` existente por:

```
C# Copiar

protected override async Task OnInitializedAsync()
{
    specials = await HttpClient.GetFromJsonAsync<List<PizzaSpecial>>(NavigationManager.BaseUri + "special
}

```

Há alguns erros que você precisa corrigir. Adicione essas instruções `@inject` na diretiva `@page`:

Observação

Esse código substituiu `OnInitialized()` por `OnInitializedAsync()`. Agora, as especiais retornarão como JSON do aplicativo de forma assíncrona.

Salve todas as alterações e selecione F5 ou Executar. Selecione Iniciar Depuração.

```
razor Copiar

@Inject HttpClient HttpClient
@Inject NavigationManager NavigationManager

```

Ocorre um erro de runtime ao executar o aplicativo. O `JsonReader` gerou uma exceção.

1. Lembre-se de que o aplicativo deve criar o JSON em (<http://localhost:5000/specials>). Vá para essa URL.

O aplicativo não sabe como rotear essa solicitação. Você aprenderá sobre roteamento no módulo sobre roteamento do Blazor. Vamos corrigir o erro.

2. Selecione Shift + F5 ou selecione Parar Depuração.

3. No explorador de arquivos, selecione `Program.cs`.

4. No meio do arquivo, após as linhas iniciadas `app.`, adicione este ponto de extremidade:

```
C# Copiar

app.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");

```

Agora, o código deve ser:

```
C# Copiar

...
app.MapRazorPages();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");
app.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
...

```

Selecione F5 ou Executar. Selecione Iniciar Depuração.

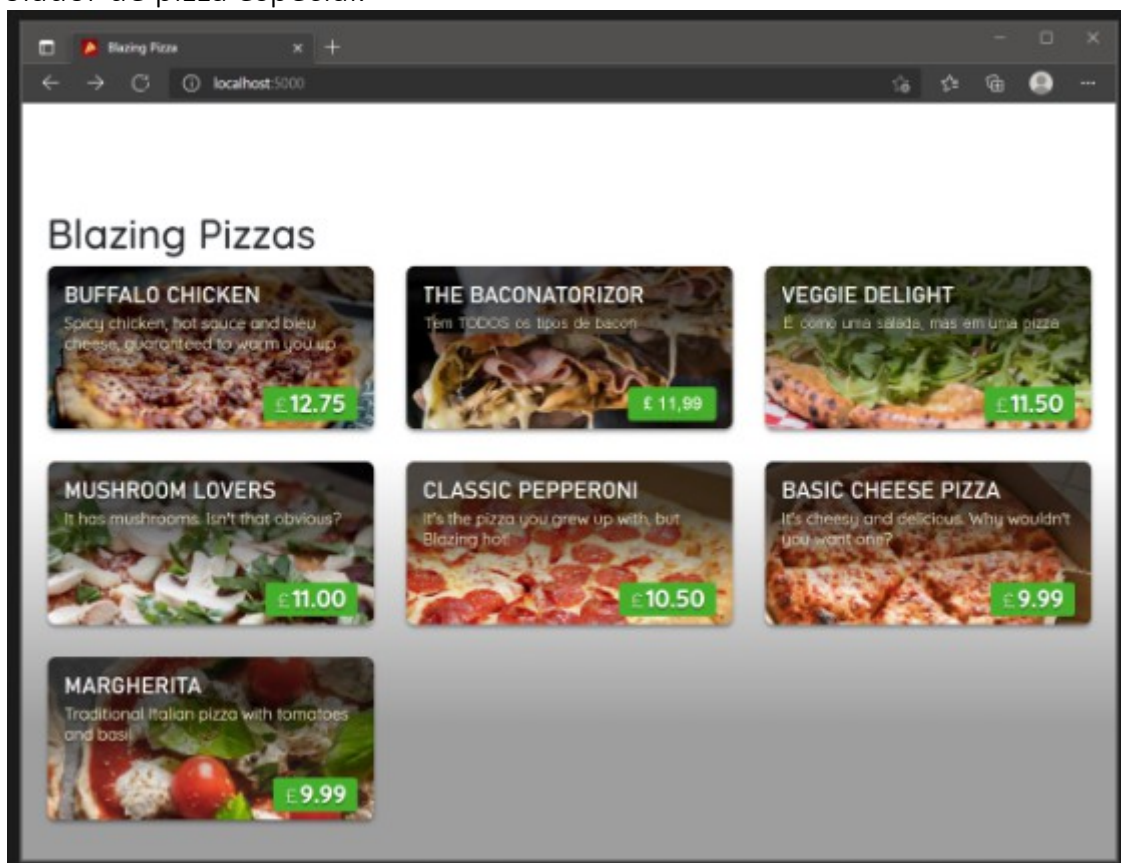
O aplicativo deve funcionar, mas vamos verificar se o JSON está sendo criado corretamente.

1. Vá para (<http://localhost:5000/specials>) para ver:

```
localhost 5000/specials
localhost:5000/specials

[{"id":4,"name":"Buffalo chicken","basePrice":12.75,"description":"Spicy chicken, hot sauce and bleu cheese, guaranteed to warm you up","imageUrl":"img/pizzas/neaty.jpg"}, {"id":3,"name":"The Baconatorizer","basePrice":11.99,"description":"It has EVERY kind of bacon","imageUrl":"img/pizzas/bacon.jpg"}, {"id":7,"name":"Veggie Delight","basePrice":11.5,"description":"It's like salad, but on a pizza","imageUrl":"img/pizzas/salad.jpg"}, {"id":5,"name":"Mushroom Lovers","basePrice":11.0,"description":"It has mushrooms. Isn't that obvious?","imageUrl":"img/pizzas/mushroom.jpg"}, {"id":9,"name":"Classic pepperoni","basePrice":10.5,"description":"It's the pizza you grew up with, but Blazing hot!","imageUrl":"img/pizzas/pepperoni.jpg"}, {"id":1,"name":"Basic Cheese Pizza","basePrice":9.99,"description":"It's cheesy and delicious. Why wouldn't you want one?","imageUrl":"img/pizzas/cheese.jpg"}, {"id":8,"name":"Margherita","basePrice":9.99,"description":"Traditional Italian pizza with tomatoes and basil","imageUrl":"img/pizzas/margherita.jpg"}]
```

O JSON lista as pizzas em ordem decrescente de preço, conforme especificado no controlador de pizza especial.



Compartilhar dados em aplicativos Blazor

O Blazor inclui várias maneiras de compartilhar informações entre componentes. Você pode usar parâmetros de componente ou parâmetros em cascata para enviar valores de um componente pai para um componente filho. O padrão AppState é outra abordagem possível para armazenar valores e acessá-los de qualquer componente no aplicativo.

Suponha que você esteja trabalhando no novo site de entrega de pizza. Várias pizzas devem ser exibidas da mesma maneira na página inicial. Você deseja exibir as pizzas renderizando um componente filho para cada pizza. Agora, você quer passar uma ID para esse componente filho que determina a pizza que será exibida. Você também quer armazenar e exibir um valor em vários componentes que mostra o número total de pizzas que você vendeu hoje.

Nesta unidade, você aprenderá três técnicas diferentes que podem ser usadas para compartilhar valores entre dois ou mais componentes Blazor.

Como compartilhar informações com outros componentes usando parâmetros de componentes

Em um aplicativo Web Blazor, cada componente renderiza uma parte do HTML. Alguns componentes renderizam uma página completa, mas outros renderizam fragmentos menores de marcação, tais como uma tabela, um formulário ou um único controle. Se o seu componente renderizar apenas uma seção de marcação, use-o como um componente filho dentro de um componente pai. Seu componente filho também pode ser pai de outros componentes menores que são renderizados dentro dele. Os componentes filho também são conhecidos como componentes aninhados. Nessa hierarquia de componentes pai e filho, é possível compartilhar informações entre eles usando parâmetros de componente. Defina os parâmetros em componentes filho e defina seus valores no pai. Por exemplo, se você tiver um componente filho que exiba fotos de pizza, poderá usar um parâmetro de componente para passar a ID da pizza. O componente filho pesquisa a pizza a partir da ID e obtém imagens e outros dados. Se você quiser exibir várias pizzas diferentes, poderá usar esse componente filho várias vezes na mesma página pai, passando uma ID diferente para cada filho.

Comece definindo o parâmetro de componente no componente filho. Ele é definido como uma propriedade pública C# e decorado com o atributo `[Parameter]`:

A screenshot of a code editor window titled 'razor' with a 'Copiar' button in the top right corner. The code defines a Blazor component with two parameters: 'PizzaName' and 'PizzaDescription'. The HTML template uses '@PizzaName' and '@PizzaDescription' to render these parameters. The code is as follows:

```
razor
<h2>New Pizza: @PizzaName</h2>
<p>@PizzaDescription</p>
@code {
    [Parameter]
    public string PizzaName { get; set; }

    [Parameter]
    public string PizzaDescription { get; set; } = "The best pizza you've ever tasted."
}
```

Como os parâmetros do componente são membros do componente filho, é possível renderizá-los em seu HTML usando o símbolo `@` reservado do Blazor, seguido de seu nome. Além disso, o código anterior especifica um valor padrão para o parâmetro `PizzaDescription`. Esse valor é renderizado se o componente pai não

transmite um valor. Caso contrário, ele é substituído pelo valor transmitido pelo pai. Você também pode usar classes personalizadas em seu projeto como parâmetros de componente. Considere esta classe que descreve um recheio:

```
C# Copiar

public class PizzaTopping
{
    public string Name { get; set; }
    public string Ingredients { get; set; }
}
```

Você pode usá-la como um parâmetro de componente da mesma maneira que um valor de parâmetro, para acessar propriedades individuais da classe usando a sintaxe de ponto:

```
razor Copiar

<h2>New Topping: @Topping.Name</h2>

<p>Ingredients: @Topping.Ingredients</p>

@code {
    [Parameter]
    public PizzaTopping Topping { get; set; }
}
```

No componente pai, defina os valores de parâmetro usando atributos das tags do componente filho. Você define componentes simples diretamente. Com um parâmetro baseado em uma classe personalizada, use o código C# embutido para criar uma nova instância dessa classe e definir seus valores:

```
razor Copiar

@page "/pizzas-toppings"

<h1>Our Latest Pizzas and Topping</h1>

<Pizza PizzaName="Hawaiian" PizzaDescription="The one with pineapple" />

<PizzaTopping Topping="@{(new PizzaTopping() { Name = "Chilli Sauce", Ingredients = "Three kinds of chilli." })}" />
```

@page "/pizzas-toppings"

<h1>Our Latest Pizzas and Topping</h1>

<Pizza PizzaName="Hawaiian" PizzaDescription="The one with pineapple" />

<PizzaTopping Topping="@{(new PizzaTopping() { Name = "Chilli Sauce", Ingredients = "Three kinds of chilli." })}" />

Compartilhar informações usando parâmetros em cascata

Os parâmetros de componente funcionam bem quando você quer passar um valor para o filho imediato de um componente. As coisas se complicam quando você tem uma hierarquia profunda com filhos de filhos e assim por diante. Os parâmetros do componente não são transmitidos automaticamente para os componentes neto a partir dos componentes ancestrais ou mais para baixo na hierarquia. Para lidar com esse problema de forma elegante, o Blazor inclui parâmetros em cascata. Quando você define o valor de um parâmetro em cascata em um componente, seu valor fica automaticamente disponível a todos os componentes descendentes, em qualquer profundidade.

No componente pai, o uso da tag `<CascadingValue>` especifica as informações que serão transmitidas em cascata para todos os descendentes. Essa tag é implementada como um componente Blazor integrado. Qualquer componente renderizado dentro dessa marca pode acessar o valor.

```
razor Copiar

@page "/specialoffers"

<h1>Special Offers</h1>

<CascadingValue Name="DealName" Value="Throwback Thursday">
    <!-- Any descendant component rendered here will be able to access the cascading value. -->
</CascadingValue>
```

Nos componentes descendentes, é possível acessar o valor em cascata usando membros de componentes e decorando-os com o atributo `[CascadingParameter]`.

```
razor Copiar

<h2>Deal: @DealName</h2>

@code {
    [CascadingParameter(Name="DealName")]
    private string DealName { get; set; }
}
```

Assim, neste exemplo, a tag `<h2>` tem o conteúdo Deal: Throwback Thursday, pois esse valor em cascata foi definido por um componente ancestral.

📌 Observação

Com relação aos parâmetros de componente, será possível passar objetos como parâmetros em cascata se você tiver requisitos mais complexos.

No exemplo anterior, o valor em cascata é identificado pelo atributo `Name` no pai, que corresponde ao valor `Name` no atributo `[CascadingParameter]`. Opcionalmente, você pode omitir esses nomes. Nesse caso, os atributos são correspondidos por tipo. Omitir o nome funciona bem quando você tem apenas um parâmetro desse tipo. Se você quiser passar em cascata dois valores de cadeia de caracteres diferentes, deverá usar nomes de parâmetro para evitar qualquer ambiguidade.

Compartilhar informações usando AppState

Outra abordagem para compartilhar informações entre componentes diferentes é usar o padrão AppState. Você cria uma classe que define as propriedades que deseja armazenar e a registra como um serviço com escopo. Em qualquer componente em que você deseja definir ou usar os valores de AppState, injete o serviço e, em seguida, poderá acessar suas propriedades. Ao contrário dos parâmetros de componente e dos parâmetros em cascata, os valores em AppState estão disponíveis para todos os componentes no aplicativo, até mesmo os componentes que não são filhos do componente que armazenou o valor.

Como exemplo, considere esta classe que armazena um valor sobre vendas:

```
C# Copiar

public class PizzaSalesState
{
    public int PizzasSoldToday { get; set; }
}
```

Você adicionará a classe como um serviço com escopo ao arquivo Program.cs:

```
C# Copiar

...
// Add services to the container
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();

// Add the AppState class
builder.Services.AddScoped<PizzaSalesState>();
...
```

Agora, em qualquer componente em que você queira definir ou recuperar valores de AppState, você pode injetar a classe e acessar as propriedades:

```
razor Copiar

@page "/"
@inject PizzaSalesState SalesState

<h1>Welcome to Blazing Pizzas</h1>

<p>Today, we've sold this many pizzas: @SalesState.PizzasSoldToday</p>

<button @onclick="IncrementSales">Buy a Pizza</button>

@code {
    private void IncrementSales()
    {
        SalesState.PizzasSoldToday++;
    }
}
```

📌 Observação

Esse código implementa um contador que é incrementado quando o usuário seleciona um botão, muito parecido com o exemplo no Tutorial do Blazor – Criar seu primeiro aplicativo Blazor [🔗](#). A diferença é que, nesse caso, como armazenamos o valor do contador em um serviço com escopo AppState, a contagem persiste entre carregamentos de página e pode ser vista por outros usuários.

Verificar seu conhecimento

1.Os componentes Blazor podem receber entrada usando dois tipos de parâmetros. Quais são eles?

☐Parameter e DescendingParameter

☐Parameter e RequiredParameter

☐Parameter e CascadingParameter

2.O AppState pode ser registrado com o localizador de serviço usando quais destas instruções?

☐builder.Services.AddAppState()

☐builder.Services.AddServerSideBlazor()

☐builder.Services.AddScoped<PizzaSalesState>()

Exercícios: compartilhar dados em aplicativos Blazor

Agora que seu aplicativo está conectado a um banco de dados, é hora de adicionar a capacidade de pedir e configurar a pizza de um cliente.

A Blazing Pizza quer que você permita aos clientes a capacidade de alterar o tamanho das pizzas especiais. Você precisa armazenar o pedido, e optou por armazenar o estado do aplicativo em um serviço de contêiner.

Neste exercício, você transmitirá dados para um novo componente de configuração de pedido e verá como armazenar o estado do aplicativo em um serviço com escopo OrderState.

Adicionar uma caixa de diálogo de configuração de novo pedido

1.Interrompa o aplicativo se ele ainda estiver em execução.

2.No Visual Studio Code, clique com o botão direito do mouse na pasta Compartilhado e selecione Novo Arquivo.

3.Insira ConfigurePizzaDialog.razor como o nome do arquivo.

4.Insira este código para a interface do usuário do componente de novo pedido:

```
razor Copiar

@inject HttpClient HttpClient

<div class="dialog-container">
  <div class="dialog">
    <div class="dialog-title">
      <h2>@Pizza.Special.Name</h2>
      @Pizza.Special.Description
    </div>
    <form class="dialog-body">
      <div>
        <label>Size:</label>
        <input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" />
        <span class="size-label">
          @(Pizza.Size) * (€@(Pizza.GetFormattedTotalPrice()))
        </span>
      </div>
    </form>

    <div class="dialog-buttons">
      <button class="btn btn-secondary mr-auto">Cancel</button>
      <span class="mr-center">
        Price: <span class="price">@(Pizza.GetFormattedTotalPrice())</span>
      </span>
      <button class="btn btn-success ml-auto">Order ></button>
    </div>
  </div>
</div>
```

Esse componente é uma caixa de diálogo que mostra a pizza especial selecionada e permite que o cliente selecione o tamanho da pizza.

O componente precisa de uma pizza especial do componente de página de índice para acessar os valores de membro de uma pizza.

1. Adicione o bloco @code do Blazor para permitir que os parâmetros sejam passados para o componente:

```
razor Copiar

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

Pedir uma pizza

Quando um cliente seleciona uma pizza, a caixa de diálogo deve permitir que ele altere o tamanho da pizza. Vamos aprimorar o controle de index.razor para adicionar essa interatividade.

1. No explorador de arquivos, expanda Páginas e selecione Index.razor.

2. Adicione esse código no bloco @code sob a variável List<PizzaSpecial>:

```
C# Copiar

Pizza configuringPizza;
bool showingConfigureDialog;
```

Adicione o código para criar uma pizza sob o método OnInitializedAsync():

```

C# Copiar

void ShowConfigurePizzaDialog(PizzaSpecial special)
{
    configuringPizza = new Pizza()
    {
        Special = special,
        SpecialId = special.Id,
        Size = Pizza.DefaultSize
    };

    showingConfigureDialog = true;
}

```

Permita que a página da Web chame o método ShowConfigurePizzaDialog do lado do servidor, permitindo que os clientes selecionem uma tag de pizza. Substitua a linha por este código:

```

razor Copiar

<li @onclick="@(() => ShowConfigurePizzaDialog(special))" style="background-image: url('@special.ImageUrl')">

```

<li @onclick="@(() => ShowConfigurePizzaDialog(special))" style="background-image: url('@special.ImageUrl')">

Quando um cliente seleciona uma pizza, o servidor executa o método ShowConfigurePizzaDialog que cria uma pizza com os dados da pizza especial e define a variável showingConfigureDialog como true.

1.A página precisa de uma maneira de exibir o novo componente ConfigurePizzaDialog. Adicione este código acima do bloco @code:

```

razor Copiar

@if (showingConfigureDialog)
{
    <ConfigurePizzaDialog Pizza="configuringPizza" />
}

```

Agora, o arquivo index.razor deve ser parecido com este exemplo:

@page "/"

@inject HttpClient HttpClient

@inject NavigationManager NavigationManager

```

<div class="main">
    <h1>Blazing Pizzas</h1>
    <ul class="pizza-cards">

```

```

@if (specials != null)
{
    @foreach (var special in specials)
    {
        <li @onclick="@(() => ShowConfigurePizzaDialog(special))" style="background-
image: url('@special.ImageUrl')">
            <div class="pizza-info">
                <span class="title">@special.Name</span>
                @special.Description
                <span class="price">@special.GetFormattedBasePrice()</span>
            </div>
        </li>
    }
}
</ul>
</div>

```

```

@if (showingConfigureDialog)
{
    <ConfigurePizzaDialog Pizza="configuringPizza" />
}

```

```

@code {
    List<PizzaSpecial> specials = new();
    Pizza configuringPizza;
    bool showingConfigureDialog;

```

```

    protected override async Task OnInitializedAsync()
    {
        specials = await
HttpClient.GetFromJsonAsync<List<PizzaSpecial>>(NavigationManager.BaseUri +
"specials");
    }

```

```

void ShowConfigurePizzaDialog(PizzaSpecial special)
{
    configuringPizza = new Pizza()
    {
        Special = special,
        SpecialId = special.Id,
        Size = Pizza.DefaultSize
    };
}

```

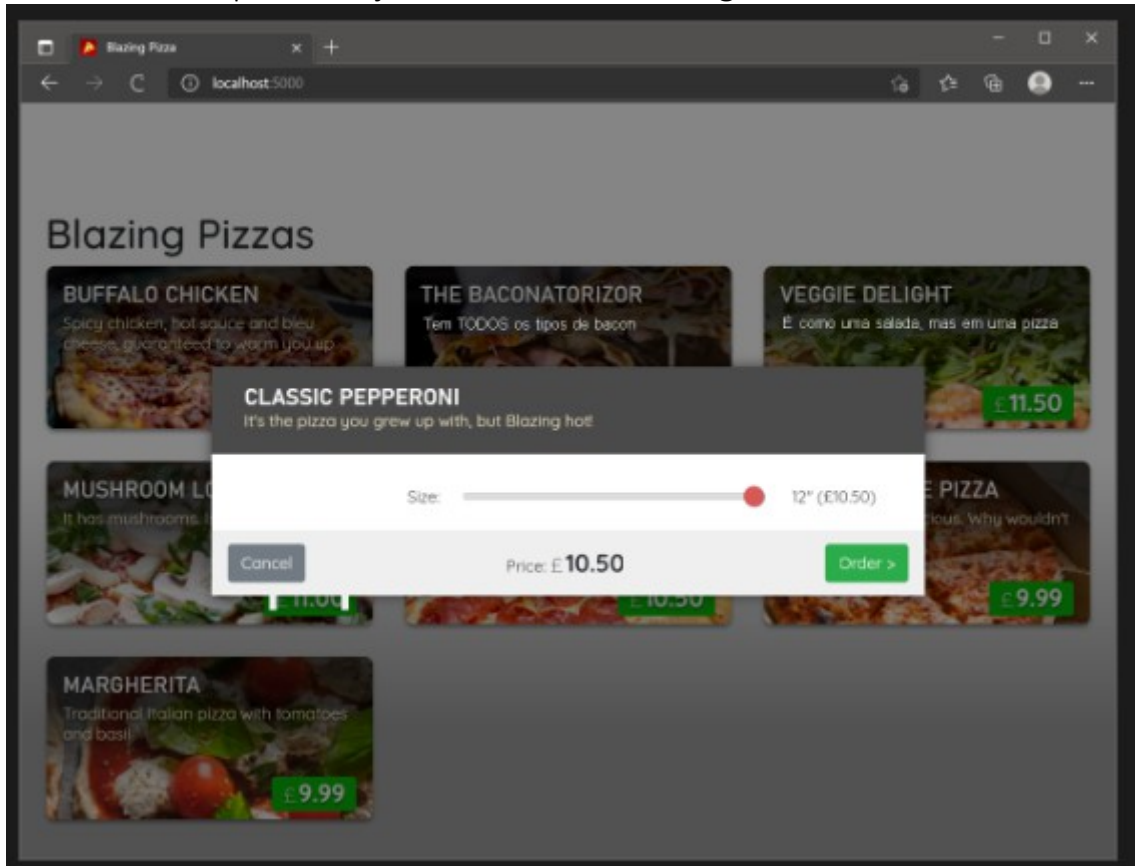
```

    showingConfigureDialog = true;
}
}

```

Selecione F5 ou Executar. Selecione Iniciar Depuração.

1. Escolha uma pizza e veja a nova caixa de diálogo.



Manipular o estado de um pedido

No momento, o aplicativo mostra a caixa de diálogo de configuração, mas não permite que você cancele ou já faça o pedido da pizza. Para gerenciar o estado do pedido, você adicionará um serviço de contêiner de estado de novo pedido.

1. Interrompa o aplicativo se ele ainda estiver em execução.
2. Crie uma nova pasta na pasta BlazingPizza. Nomeie-a como Serviços.
3. Crie um novo arquivo na pasta Serviços. Nomeie-o como OrderState.cs.
4. Insira este código para a classe:

C#


```
C# Copiar

namespace BlazingPizza.Services;

public class OrderState
{
    public bool ShowingConfigureDialog { get; private set; }
    public Pizza ConfiguringPizza { get; private set; }
    public Order Order { get; private set; } = new Order();

    public void ShowConfigurePizzaDialog(PizzaSpecial special)
    {
        ConfiguringPizza = new Pizza()
        {
            Special = special,
            SpecialId = special.Id,
            Size = Pizza.DefaultSize,
            Toppings = new List<PizzaTopping>(),
        };

        ShowingConfigureDialog = true;
    }

    public void CancelConfigurePizzaDialog()
    {
        ConfiguringPizza = null;

        ShowingConfigureDialog = false;
    }

    public void ConfirmConfigurePizzaDialog()
    {
        Order.Pizzas.Add(ConfiguringPizza);
        ConfiguringPizza = null;

        ShowingConfigureDialog = false;
    }
}
```

Você verá que há código atualmente no componente `index.razor` que podemos mover para a nova classe. A próxima etapa é disponibilizar esse serviço no aplicativo.

- 1.No explorador de arquivos, selecione `Program.cs`.
- 2.Na parte do arquivo com as linhas que começam com `builder.Services.`, adicione esta linha:

```
C# Copiar

builder.Services.AddScoped<OrderState>();
```

]
]

No exercício anterior, adicionamos nosso contexto de banco de dados aqui. Esse código adiciona o novo serviço `OrderState`. Com o código aplicado, agora podemos usá-lo no componente `index.razor`.

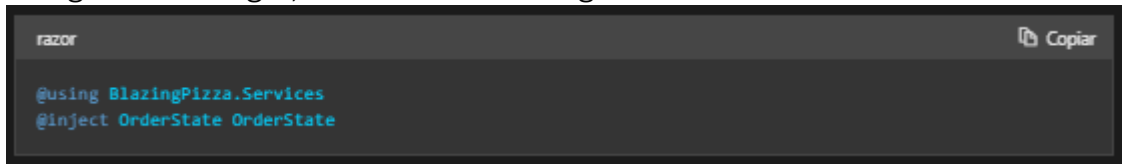
- 1.Adicione a seguinte diretiva `using` ao topo do arquivo para resolver a classe `OrderState`:

```
C# Copiar

using BlazingPizza.Services;
```

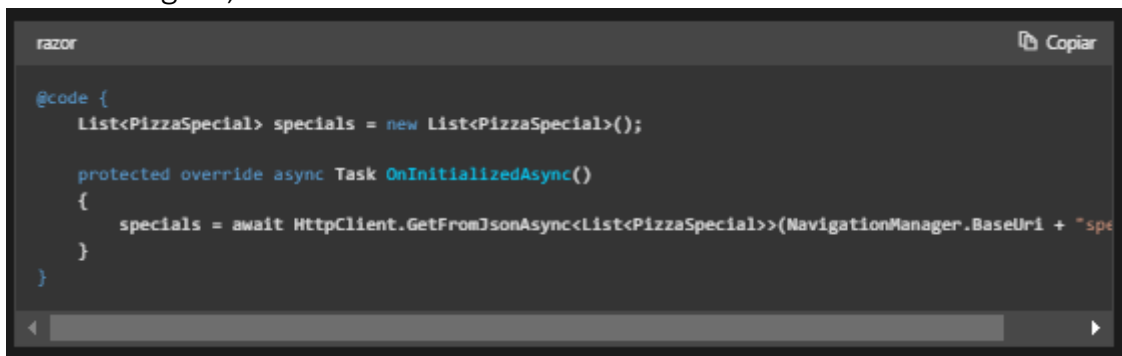
No explorador de arquivos, expanda Páginas e selecione Index.razor.

1. Na parte superior do arquivo, em @inject NavigationManager, adicione este código:



```
razor
@using BlazingPizza.Services
@inject OrderState OrderState
```

Remova configuringPizza, showingConfigureDialog e ShowConfigurePizzaDialog() do @code bloco. Agora, ele deve ser assim:



```
razor
@code {
    List<PizzaSpecial> specials = new List<PizzaSpecial>();

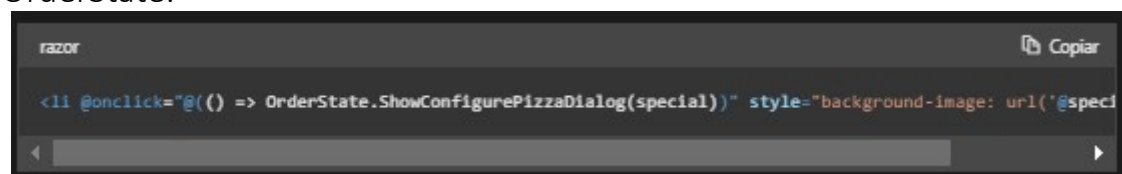
    protected override async Task OnInitializedAsync()
    {
        specials = await HttpClient.GetFromJsonAsync<List<PizzaSpecial>>(NavigationManager.BaseUri + "spe
    }
}
```

```
@code {
    List<PizzaSpecial> specials = new List<PizzaSpecial>();

    protected override async Task OnInitializedAsync()
    {
        specials = await
        HttpClient.GetFromJsonAsync<List<PizzaSpecial>>(NavigationManager.BaseUri +
        "specials");
    }
}
```

Agora, há erros onde todo o código faz referência ao que foi excluído.

1. Altere a chamada a ShowConfigurePizzaDialog(special)) para usar a versão de OrderState:



```
razor
<li @onclick="@(() => OrderState.ShowConfigurePizzaDialog(special))" style="background-image: url('@speci
```

<li @onclick="@(() => OrderState.ShowConfigurePizzaDialog(special))" style="background-image: url('@special.ImageUrl')">Altere a referência para o booleano showingConfigureDialog:

```
razor

@if (OrderState.ShowingConfigureDialog)
```

Altere o parâmetro usando configuringPizza:

```
razor

<ConfigurePizzaDialog Pizza="OrderState.ConfiguringPizza" />
```

Selecione F5 ou Executar. Selecione Iniciar Depuração.

Se tudo estiver correto, você não verá nenhuma diferença. A caixa de diálogo é exibida como antes.

Cancelar e fazer pedidos de pizza

Talvez você tenha observado na classe OrderState dois métodos que ainda não foram usados. Os métodos CancelConfigurePizzaDialog e ConfirmConfigurePizzaDialog fecharão a caixa de diálogo e adicionarão a pizza a um objeto Order se o cliente tiver confirmado o pedido. Vamos conectar esses métodos aos botões da caixa de diálogo de configuração.

1. Interrompa o aplicativo se ele ainda estiver em execução.
 2. No explorador de arquivos, expanda Compartilhado.
- Selecione ConfigurePizzaDialog.razor.
3. No bloco @code, adicione dois novos parâmetros:

```
razor

@code {
    [Parameter] public Pizza Pizza { get; set; }
    [Parameter] public EventCallback OnCancel { get; set; }
    [Parameter] public EventCallback OnConfirm { get; set; }
}
```

Agora, os botões podem ter diretivas @onclick adicionadas. Altere o código atual dos botões da caixa de diálogo para esta marcação:

```
razor

<div class="dialog-buttons">
    <button class="btn btn-secondary mr-auto" @onclick="OnCancel">Cancel</button>
    <span class="mr-center">
        Price: <span class="price">@(Pizza.GetFormattedTotalPrice())</span>
    </span>
    <button class="btn btn-success ml-auto" @onclick="OnConfirm">Order </button>
</div>
```

A última etapa é transmitir os métodos OrderState para cancelar e confirmar pedidos. No explorador de arquivos, expanda Páginas. Depois, selecione Index.razor.

1. Altere o código da chamada para o componente ConfigurePizzaDialog:

```
razor
<ConfigurePizzaDialog
  Pizza="OrderState.ConfiguringPizza"
  OnCancel="OrderState.CancelConfigurePizzaDialog"
  OnConfirm="OrderState.ConfirmConfigurePizzaDialog" />
```

Selecione F5 ou Executar. Selecione Iniciar Depuração.

Agora, o aplicativo deve permitir que os clientes cancelem ou adicionem uma pizza configurada a um pedido. Não temos como mostrar o pedido atual ou atualizar os preços quando o cliente muda o tamanho da pizza. Adicionaremos esses recursos no próximo exercício.

Associar controles aos dados em aplicativos Blazor

O Blazor permite que você associe controles HTML a propriedades, de modo que a alteração de valores seja exibida automaticamente na interface do usuário.

Vamos supor que você esteja desenvolvendo uma página que coleta informações de clientes sobre suas preferências de pizza. Você quer carregar as informações de um banco de dados e permitir que os clientes façam alterações, como gravar seus recheios preferidos. Quando o usuário fizer uma mudança ou houver uma atualização no banco de dados, você quer que os novos valores sejam exibidos na interface do usuário o mais rápido possível.

Nesta unidade, você aprenderá a usar a associação de dados no Blazor para associar elementos de interface do usuário a valores de dados, propriedades ou expressões.

O que é a associação de dados?

Se você quiser que um elemento HTML exiba um valor, poderá escrever código para alterar a exibição. Você precisará escrever código extra para atualizar a exibição quando o valor for atualizado. No Blazor, você pode usar a associação de dados para conectar um elemento HTML a um campo, propriedade ou expressão. Dessa forma, quando o valor for atualizado, o elemento HTML será atualizado automaticamente. A atualização geralmente ocorre rapidamente após a alteração e você não precisa escrever nenhum código de atualização.

Para associar um controle, você usará a diretiva @bind:

```
razor Copiar

@page "/"

<p>
    Your email address is:
    <input @bind="customerEmail" />
</p>

@code {
    private string customerEmail = "john.doe@contoso.com"
}
```

Na página anterior, sempre que a variável `customerEmail` alterar seu valor, o valor `<input>` será atualizado.

🕒 Observação

Controles, como `<input>`, atualizam a exibição somente quando o componente é renderizado e não quando o valor de um campo é alterado. Como os componentes Blazor são renderizados depois que qualquer código do manipulador de eventos é executado, na prática, as atualizações geralmente são exibidas rapidamente.

Associar elementos a eventos específicos

A diretiva `@bind` é inteligente e compreende os controles que usa. Por exemplo, quando você associa um valor a uma caixa de texto `<input>`, ele associa o atributo `value`. Uma caixa de seleção HTML `<input>` tem um atributo `checked` em vez de um atributo `value`. O atributo `@bind` usa automaticamente este atributo `checked`. Por padrão, o controle é associado ao evento DOM `onchange`. Por exemplo, considere esta página:

```
razor Copiar

@page "/"

<h1>My favorite pizza is: @favPizza</h1>

<p>
    Enter your favorite pizza:
    <input @bind="favPizza" />
</p>

@code {
    private string favPizza { get; set; } = "Margherita"
}
```

Quando a página é renderizada, o valor padrão `Margherita` é exibido no elemento `<h1>` e na caixa de texto. Quando você inserir uma nova pizza preferida na caixa de texto, o elemento `<h1>` não será alterado até você sair da caixa de texto ou selecionar `Enter`, pois é nesse momento que o evento DOM `onchange` dispara.

Geralmente, esse é o comportamento desejado. Mas suponha que você queira que o elemento `<h1>` seja atualizado assim que você insere qualquer caractere na caixa de texto. Em vez disso, você pode obter esse resultado associando o evento

DOM oninput. Para associar a esse evento, você deve usar as diretivas @bind-value @bind-value:event:

```
razor
@page "/"

<h1>My favorite pizza is: @favPizza</h1>

<p>
    Enter your favorite pizza:
    <input @bind-value="favPizza" @bind-value:event="oninput" />
</p>

@code {
    private string favPizza { get; set; } = "Margherita"
}
```

Nesse caso, o título é alterado assim que você digita qualquer caractere na caixa de texto.

Formatar valores limitados

Se você exibir datas para o usuário, talvez queira usar um formato de dados localizado. Por exemplo, vamos supor que você escreva uma página especificamente para usuários do Reino Unido, que preferem escrever datas usando o dia primeiro. Você pode usar a diretiva @bind:format para especificar uma cadeia de caracteres de formato de data única:

```
razor
@page "/ukbirthdaypizza"

<h1>Order a pizza for your birthday!</h1>

<p>
    Enter your birth date:
    <input @bind="birthdate" @bind:format="dd-MM-yyyy" />
</p>

@code {
    private DateTime birthdate { get; set; } = new(2000, 1, 1);
}
```

📌 Observação

No momento da gravação, as cadeias de caracteres de formato só têm suporte com valores de data. Formatos de moeda, formatos de número e outros formatos podem ser adicionados posteriormente. Para verificar as informações mais recentes sobre formatos de associação, confira [Formatar cadeias de caracteres](#) na documentação do Blazor.

Como alternativa ao uso da diretiva @bind:format, você pode escrever código em C# para formatar um valor associado. Use os acessadores get e set na definição de

membro, como neste exemplo:

```
razor
@page "/pizzaapproval"
@using System.Globalization

<h1>Pizza: @PizzaName</h1>

<p>Approval rating: @ApprovalRating</p>

<p>
    <label>
        Set a new approval rating:
        <input @bind="ApprovalRating" />
    </label>
</p>

@code {
    private decimal approvalRating = 1.0;
    private NumberStyles style = NumberStyles.AllowDecimalPoint | NumberStyles.AllowLeadingSign;
    private CultureInfo culture = CultureInfo.CreateSpecificCulture("en-US");

    private string ApprovalRating
    {
        get => approvalRating.ToString("0.000", culture);
        set
        {
            if (Decimal.TryParse(value, style, culture, out var number))
            {
                approvalRating = Math.Round(number, 3);
            }
        }
    }
}
```

Na próxima unidade, você aplicará o que aprendeu.

Exercício: associar controles aos dados em aplicativos Blazor

O aplicativo Blazing Pizza precisa atualizar a interface quando os clientes modificarem as pizzas e adicioná-las aos pedidos. O Blazor permite que você associe controles HTML a propriedades em C# para atualizar quando os valores mudarem.

Os clientes devem ver quais pizzas estão pedindo e como o tamanho escolhido afeta o preço que eles pagarão.

Neste exercício, você deixará o aplicativo Blazing Pizza numa posição em que os pedidos podem ser atualizados e editados. Você verá como associar controles às propriedades de uma pizza e recalculer os preços após essas alterações.

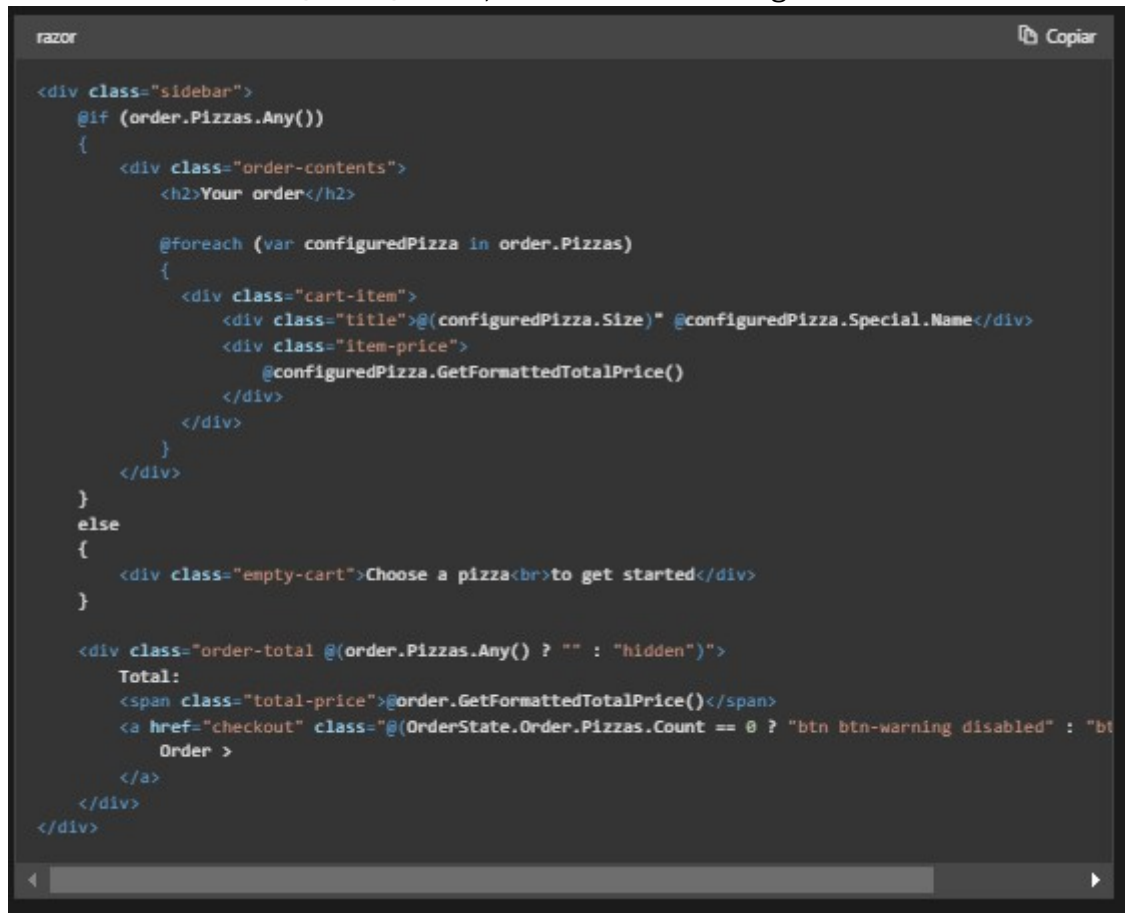
Exibir o pedido de pizza de um cliente

Você adicionará uma barra lateral que exibirá todas as pizzas adicionadas por um cliente ao pedido.

1. Interrompa o aplicativo se ele ainda estiver em execução.

2.No Visual Studio Code, no explorador de arquivos, expanda Páginas e selecione Index.razor.

3.Entre os blocos e @if e @code, adicione este código:

A screenshot of the Visual Studio Code editor with a dark theme. The file explorer on the left shows 'Páginas' expanded and 'Index.razor' selected. The editor displays the following Razor code:

```
<div class="sidebar">
  @if (order.Pizzas.Any())
  {
    <div class="order-contents">
      <h2>Your order</h2>

      @foreach (var configuredPizza in order.Pizzas)
      {
        <div class="cart-item">
          <div class="title">@(configuredPizza.Size)" @configuredPizza.Special.Name</div>
          <div class="item-price">
            @configuredPizza.GetFormattedTotalPrice()
          </div>
        </div>
      }
    </div>
  }
  else
  {
    <div class="empty-cart">Choose a pizza<br>to get started</div>
  }

  <div class="order-total @(order.Pizzas.Any() ? "" : "hidden")">
    Total:
    <span class="total-price">@order.GetFormattedTotalPrice()</span>
    <a href="checkout" class="@(OrderState.Order.Pizzas.Count == 0 ? "btn btn-warning disabled" : "btn btn-primary")">
      Order >
    </a>
  </div>
</div>
```

```
<div class="sidebar">
  @if (order.Pizzas.Any())
  {
    <div class="order-contents">
      <h2>Your order</h2>

      @foreach (var configuredPizza in order.Pizzas)
      {
        <div class="cart-item">
          <div class="title">@(configuredPizza.Size)"
@configuredPizza.Special.Name</div>
          <div class="item-price">
            @configuredPizza.GetFormattedTotalPrice()
          </div>
        </div>
      }
    </div>
  }
  else
```



```
{
  <div class="empty-cart">Choose a pizza<br>to get started</div>
}
```

```
<div class="order-total @(order.Pizzas.Any() ? "" : "hidden")">
```

```
  Total:
```

```
  <span class="total-price">@order.GetFormattedTotalPrice()</span>
```

```
  <a href="checkout" class="@(OrderState.Order.Pizzas.Count == 0 ? "btn btn-
warning disabled" : "btn btn-warning")">
```

```
    Order >
```

```
  </a>
```

```
</div>
```

```
</div>
```

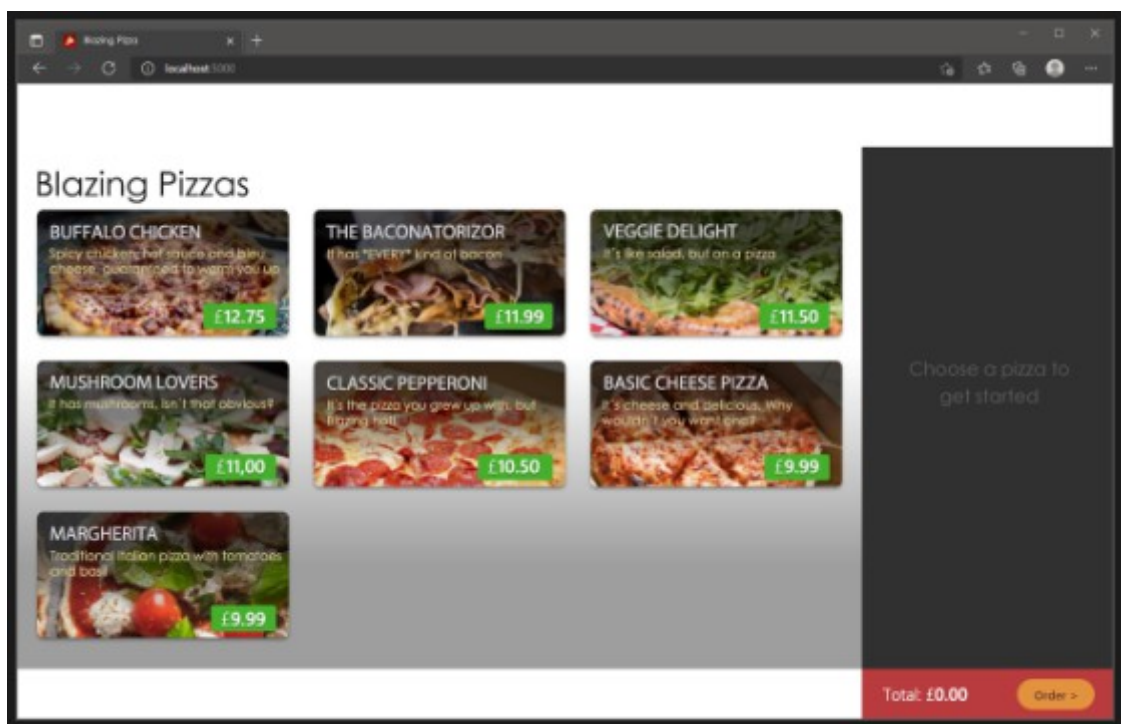
Este HTML adiciona uma barra lateral à página. Se houver pizzas no OrderState.Order, ele as exibirá. Se não houver pedidos, os clientes receberão a solicitação para adicionar alguma.

Você verá alguns erros, pois o componente não sabe quais são os pedidos.

1.No bloco @code em List<PizzaSpecial> specials = new();, adicione este código:

```
Order order => OrderState.Order;
```

Selecione F5 ou Executar. Selecione Iniciar Depuração.



Experimente pedir algumas pizzas e cancelar outras. Você verá que elas são adicionadas ao carrinho e o total do pedido é atualizado.

1. Selecione Shift+F5 ou selecione Parar Depuração.

Remover uma pizza do pedido de um cliente

Talvez você tenha notado que não há como remover uma pizza configurada do carrinho de compras do cliente. Vamos adicionar essa funcionalidade.

A primeira etapa é atualizar o serviço OrderState para que ele possa fornecer um método de remoção de pizzas de um pedido.

1. No explorador de arquivos, selecione Services/OrderState.cs.

2. Na parte inferior da classe, adicione este método:

```
C# Copiar

public void RemoveConfiguredPizza(Pizza pizza)
{
    Order.Pizzas.Remove(pizza);
}
```

No explorador de arquivos, expanda Páginas e selecione Index.razor.

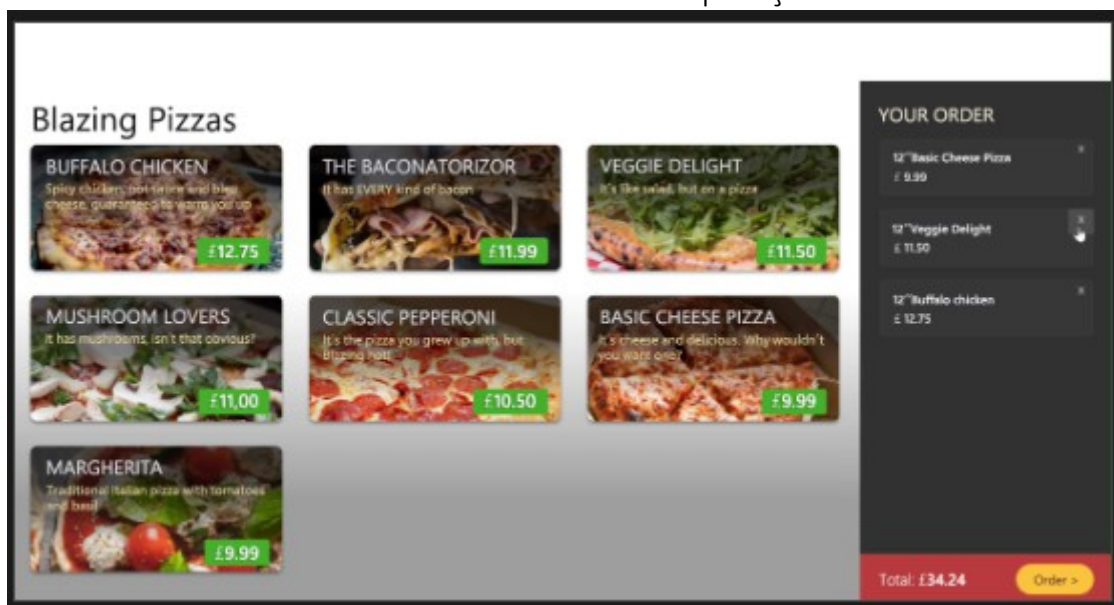
1. Em <div class="cart-item">, adicione uma marca <a> antes do fechamento </div> para criar um botão de remoção:

```
razor Copiar

<a @onclick="@(() => OrderState.RemoveConfiguredPizza(configuredPizza))" class="delete-item">x</a>
```

Essa tag adiciona um X em cada pizza na barra lateral do pedido. Quando é selecionada, ela chama o método RemoveConfiguredPizza no serviço OrderState.

1. Selecione F5 ou Executar. Selecione Iniciar Depuração.

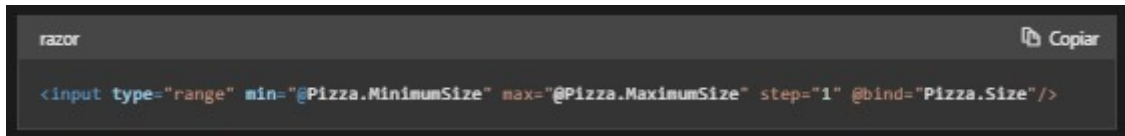


Selecione Shift+F5 ou selecione Parar Depuração.

Configurar um tamanho de uma pizza dinamicamente

A caixa de diálogo de configuração de pizza não é atualizada quando o controle deslizante de tamanho é alterado. O componente precisa de uma maneira de atualizar a pizza e o tamanho e recalculer o preço.

- 1.No explorador de arquivos, expanda Compartilhado e selecione ConfigurePizzaDialog.razor.
- 2.Adicione código ao controle HTML input para associar seu valor ao tamanho de pizza:



```
razor Copiar<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" @bind="Pizza.Size"/>
```

Selecione F5 ou Executar. Selecione Iniciar Depuração.

Use a caixa de diálogo atualizada para adicionar pizzas de tamanho diferente ao seu pedido. Clique na barra de controle deslizante em vez de arrastar. Observe que o tamanho da pizza é atualizado no evento mouse-up do controle. Se você arrastar o controle deslizante, o tamanho não será alterado até liberar o mouse. Vamos corrigir isso.

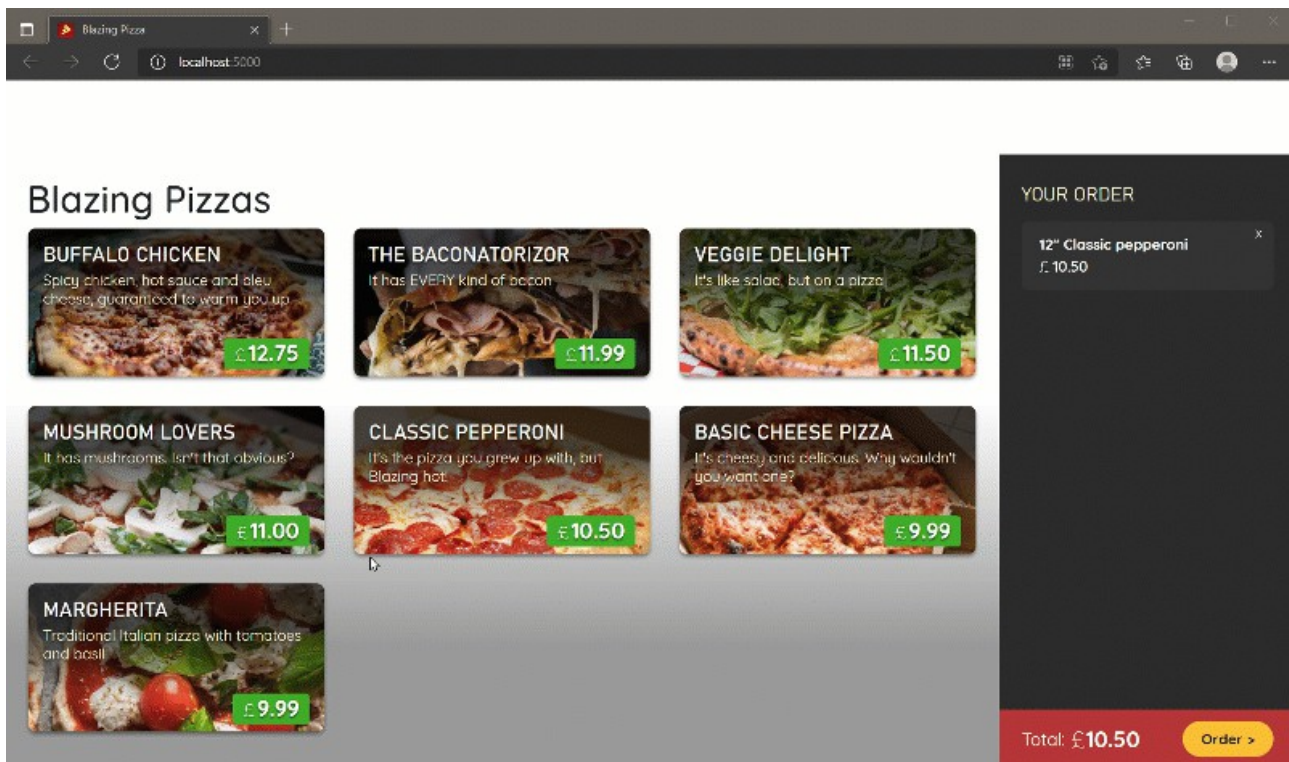
- 1.Selecione Shift+F5 ou selecione Parar Depuração.
- 2.Adicione o evento ao qual o controle deve ser associado:



```
razor Copiar<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" @bind="Pizza.Size" @bind:event="oninput" />
```

```
<input type="range" min="@Pizza.MinimumSize" max="@Pizza.MaximumSize" step="1" @bind="Pizza.Size" @bind:event="oninput" />
```

Selecione F5 ou Executar. Selecione Iniciar Depuração.



Como adicionar o código `@bind="Pizza.Size"` fornece tanta funcionalidade? Se você examinar todo o código de **ConfigurePizzaDialog.razor**, verá que sua equipe já conectou os outros elementos às propriedades da pizza.

Por exemplo, o preço é atualizado por causa desse código:

```

razor
Price: <span class="price">@(Pizza.GetFormattedTotalPrice())</span>
  
```

O preço é atualizado à medida que o tamanho da pizza é alterado, pois o método em `GetFormattedTotalPrice()` da pizza usa o tamanho da pizza para calcular o preço total.

Você progrediu com o aplicativo Blazing Pizza. Se você quer continuar melhorando o aplicativo, conclua o próximo módulo neste roteiro de aprendizagem.

Resumo

Você trabalha para uma empresa de entrega de pizza que deseja um site novo e tem uma equipe com anos de experiência em C#. Com o Blazor, você é capaz de escrever códigos em C# do lado do cliente e do lado do servidor e minimizar o código JavaScript.

Antes do Blazor, você precisava investir no treinamento dos seus desenvolvedores em JavaScript e em outras tecnologias da Web. Esse treinamento não é tão necessário com o Blazor. E o Blazor pode ajudar a acelerar seus projetos de desenvolvimento de aplicativo Web.

Após a conclusão deste módulo, você aprendeu a:

- Montar uma interface do usuário para um aplicativo Web criando componentes Blazor.
- Acessar dados para exibi-los em seu aplicativo Web.
- Compartilhar dados em seu aplicativo Web entre vários componentes Blazor.
- Associar um elemento HTML a uma variável em um componente Blazor.

Saiba mais

- Introdução ao Blazor do ASP.NET Core
- Componentes Razor do ASP.NET Core
- Modelos de hospedagem Blazor do ASP.NET Core
- Valores e parâmetros em cascata Blazor do ASP.NET Core
- Associação de dados Blazor do ASP.NET Core
- Blazor – Workshop de criação de aplicativo