

Usar páginas, roteamento e layouts para melhorar a navegação no Blazor

Saiba como gerenciar o roteamento de solicitações usando a diretiva `@page`, o roteamento do Blazor, bem como os componentes `NavLink` e `NavMenu`. Aumentar a flexibilidade de um aplicativo com a adição de parâmetros de roteamento aos componentes do Blazor. Use layouts para melhorar seu aplicativo com a redução de código duplicado.

Objetivos de aprendizagem

Ao final deste módulo, você saberá como:

- Aprimorar a navegação no aplicativo Blazor usando o componente de roteador e `NavLinks`.
- Aprimorar a funcionalidade com parâmetros de rota.
- Reduzir o código duplicado com o uso de layouts no aplicativo Blazor.

Pré-requisitos

- Conhecimento básico de conceitos de aplicativo Web
- Experiência de nível iniciante no .NET e no C#
- Instalações locais do SDK do .NET e do Visual Studio Code
- Extensão do C# para o Visual Studio Code
- Experiência com o uso da linha de comando
- Conhecimento sobre os componentes do Blazor

Este módulo faz parte destes roteiros de aprendizagem

- Criar aplicativos Web com o Blazor
 - Introdução 3 min
 - Usar o componente de roteador do Blazor para controlar a navegação no aplicativo 5 min
 - Exercício – Alterar a navegação no aplicativo Blazor usando a diretiva `@page` 7 min
 - Explorar como os parâmetros de rota afetam o roteamento do aplicativo Blazor 7 min
 - Exercício: usar parâmetros de rota para melhorar a navegação nos aplicativos 5 min
 - Criar componentes do Blazor reutilizáveis usando layouts 7 min
 - Exercício: adicionar um layout do Blazor para reduzir a duplicação no código 4 min
 - Resumo 3 min

Introdução

O Blazor cria aplicativos Web interativos por meio do .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 já criou páginas que exibem pizzas que permitem aos clientes personalizar as coberturas. Você deseja adicionar páginas de pedido e melhorar a navegação do aplicativo. Você também deseja manter um layout consistente em todo o aplicativo para garantir que os clientes conseguirão encontrar o que estão procurando facilmente.

Neste módulo, você aprenderá a encaminhar os clientes por meio do aplicativo usando a diretiva `@page`, o roteamento do Blazor e o componente `NavLink`. Com a navegação funcionando, você se concentrará em como reduzir o código duplicado com a adição de layouts ao aplicativo.

Objetivos de aprendizagem

Ao final deste módulo, você saberá como:

- Aprimorar a navegação no aplicativo Blazor usando o componente de roteador e `NavLinks`.
- Aprimorar a funcionalidade com parâmetros de rota.
- Reduzir o código duplicado com o uso de layouts no aplicativo Blazor.

Usar o componente de roteador do Blazor para controlar a navegação no aplicativo

O sistema de roteamento do Blazor fornece opções flexíveis que você pode usar para garantir que as solicitações do usuário alcançarão um componente que possa lidar com elas e retornar informações que o usuário deseja.

Suponha que você esteja trabalhando no site da empresa de entrega de pizza. Você deseja configurar o site para que as solicitações de detalhes da pizza e de cobertura personalizada sejam tratadas pelo mesmo componente. Você concluiu essa fase, mas seu teste mostra que as solicitações de cobertura recebem uma mensagem de erro. Você precisa corrigir esse problema.

Aqui, você aprenderá a configurar rotas no Blazor usando a diretiva `@page`.

🔔 Observação

Os blocos de código desta unidade são exemplos ilustrativos. Você escreverá seu código na próxima unidade.

Usar modelos de rota

Quando o usuário faz uma solicitação a uma página do seu aplicativo Web, ele pode especificar o que deseja ver com informações no URI. Por exemplo:

`http://www.contoso.com/pizzas/margherita?extratopping=pineapple`

Após o protocolo e o endereço do site, esse URI indica que o usuário deseja saber mais sobre pizzas Marguerita. Além disso, a cadeia de consulta após o ponto de interrogação mostra que eles estão interessados em uma cobertura extra de abacaxi. No Blazor, você usa o roteamento para garantir que cada solicitação seja enviada para o melhor componente e que o componente tenha todas as informações necessárias para exibir o que o usuário deseja. Nesse caso, o ideal é que a solicitação seja enviada ao componente **Pizzas** e que esse componente exiba uma pizza de marguerita com informações sobre como adicionar abacaxi. O Blazor encaminha solicitações com um componente especializado chamado componente **Router**. Ele está configurado no **App.razor** da seguinte maneira:

```

razor
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
  </Found>
  <NotFound>
    <p>Sorry, we haven't found any pizzas here.</p>
  </NotFound>
</Router>

```

Quando o aplicativo é iniciado, o Blazor verifica o atributo `AppAssembly` para descobrir qual assembly ele deve examinar. Ele verifica se há componentes que tenham **RouteAttribute** presente no assembly. Usando esses valores, o Blazor compila um objeto **RouteData** que especifica como as solicitações são roteadas para os componentes. Ao codificar o aplicativo, você usará a diretiva `@page` em cada componente para corrigir o **RouteAttribute**.

No código anterior, a marca `<Found>` especifica o componente que trata do roteamento em runtime: o componente **RouteView**. Esse componente recebe o objeto **RouteData** e todos os parâmetros do URI ou da cadeia de caracteres de consulta. Em seguida, ele renderiza o componente especificado e seu layout. Você pode usar a marca `<Found>` para especificar um layout padrão, que será usado quando o componente selecionado não especificar um layout com a diretiva `@layout`. Você verá mais sobre esses layouts posteriormente neste módulo.

No componente `<Router>`, você também pode especificar o que é retornado ao usuário quando não há uma rota correspondente usando a marca `<NotFound>`. O exemplo anterior retorna um só parágrafo `<p>`, mas você pode renderizar um HTML mais complexo. Por exemplo, ele pode incluir um link para a home page ou uma página de contato para os administradores do site.

Usar a diretiva `@page`

Em um componente do Blazor, a diretiva `@page` especifica que o componente deve tratar das solicitações diretamente. Você pode especificar um **RouteAttribute** na diretiva `@page` transmitindo-o como uma cadeia de caracteres. Por exemplo, use esse atributo para especificar que a página lida com as solicitações para a rota `/Pizzas`:

```

razor
@page "/Pizzas"

```

Caso você deseje especificar mais de uma rota para o componente, use duas ou mais diretivas `@page`, como neste exemplo:

```

razor
@page "/Pizzas"
@page "/CustomPizzas"

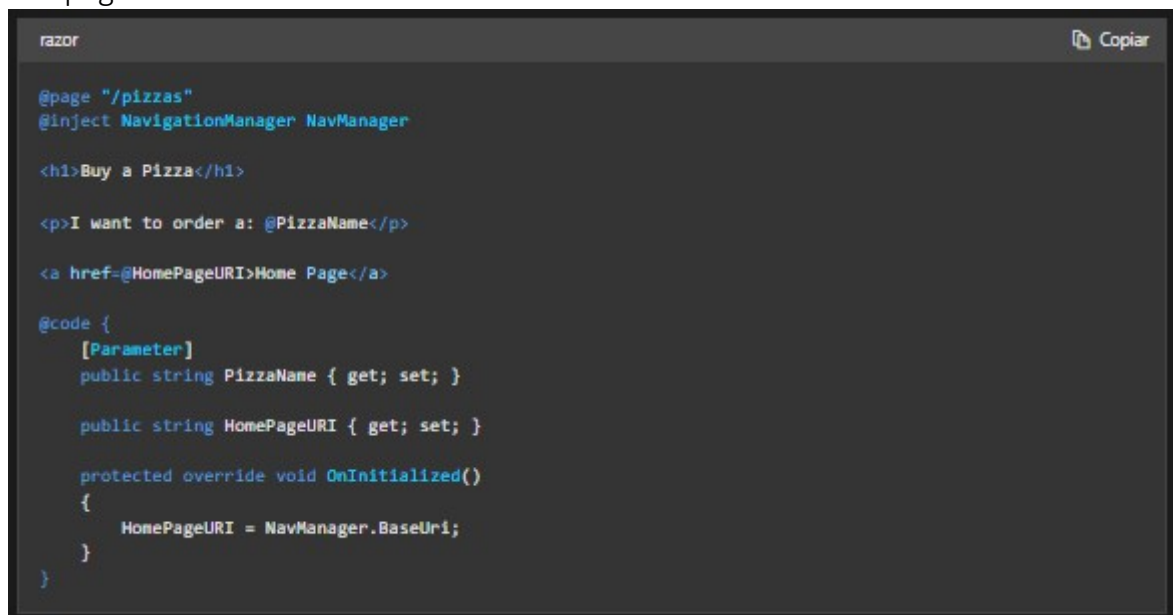
```

Obter informações de localização e navegar com o NavigationManager

Suponha que você esteja escrevendo um componente para tratar os URIs solicitados pelo usuário, como `http://www.contoso.com/pizzas/margherita/?extratopping=pineapple`. Ao escrever um componente, talvez você precise ter acesso a informações de navegação, como:

- O URI completo atual, como `http://www.contoso.com/pizzas/margherita?extratopping=pineapple`.
- O URI base, como `http://www.contoso.com/`.
- O caminho relativo base, como `pizzas/margherita`.
- A cadeia de consulta, como `?extratopping=pineapple`.

Use um objeto `NavigationManager` para obter todos esses valores. Você deve injetar o objeto no componente para, em seguida, poder acessar suas propriedades. Este código usa o objeto `NavigationManager` para obter o URI base do site e usá-lo para definir um link para a home page:



```
razor
@page "/pizzas"
@inject NavigationManager NavManager

<h1>Buy a Pizza</h1>

<p>I want to order a: @PizzaName</p>

<a href=@HomePageURI>Home Page</a>

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

    public string HomePageURI { get; set; }

    protected override void OnInitialized()
    {
        HomePageURI = NavManager.BaseUri;
    }
}
```

Para acessar a cadeia de caracteres de consulta, você precisa analisar o URI completo. Use a classe `QueryHelpers` do assembly `Microsoft.AspNetCore.WebUtilities` para fazer a análise:

```
razor Copiar

@page "/pizzas"
@using Microsoft.AspNetCore.WebUtilities
@inject NavigationManager NavManager

<h1>Buy a Pizza</h1>

<p>I want to order a: @PizzaName</p>

<p>I want to add this topping: @ToppingName</p>

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

    private string ToppingName { get; set; }

    protected override void OnInitialized()
    {
        var uri = NavManager.ToAbsoluteUri(NavManager.Uri);
        if (QueryHelpers.ParseQuery(uri.Query).TryGetValue("extratopping", out var extraTopping))
        {
            ToppingName = System.Convert.ToString(extraTopping);
        }
    }
}
```

Com o componente anterior implantado, se um usuário solicitar o URI `http://www.contoso.com/pizzas?extratopping=Pineapple`, verá a mensagem "Quero adicionar esta cobertura: abacaxi" na página renderizada.

Você também pode usar o objeto `NavigationManager` a fim de enviar seus usuários para outro componente no código chamando o método `NavigationManager.NavigateTo()`:

```
razor Copiar

@page "/pizzas/{pizzaname}"
@inject NavigationManager NavManager

<h1>Buy a Pizza</h1>

<p>I want to order a: @PizzaName</p>

<button class="btn" @onclick="NavigateToPaymentPage">
    Buy this pizza!
</button>

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

    private void NavigateToPaymentPage()
    {
        NavManager.NavigateTo("buypizza");
    }
}
```

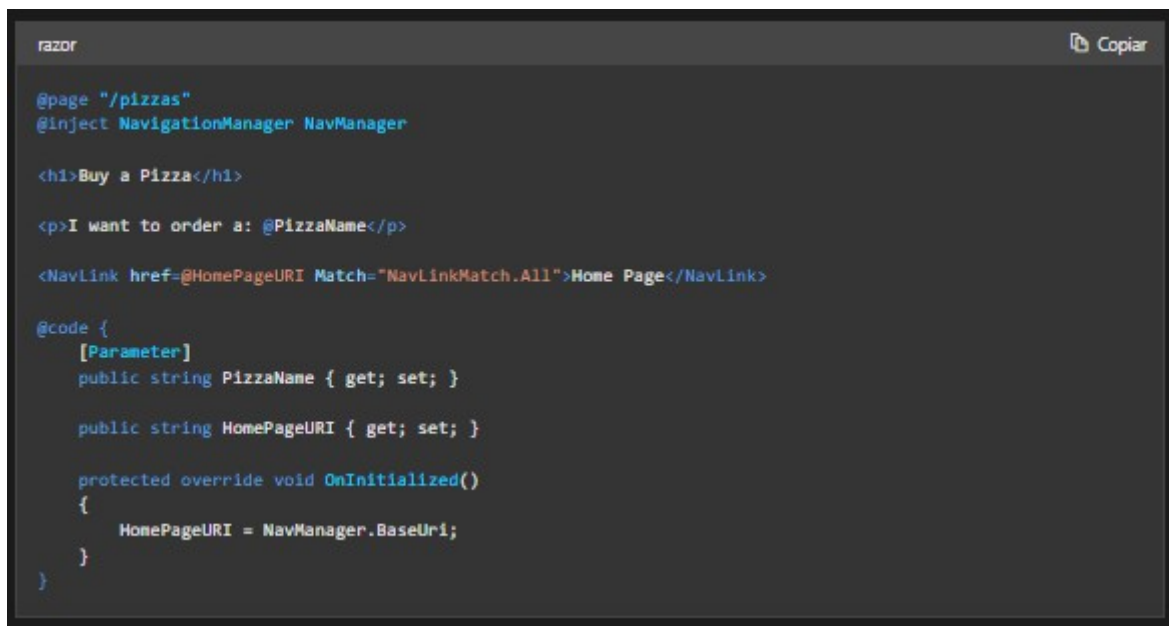
📌 Observação

A cadeia de caracteres transmitida para o método `NavigateTo()` é o URI absoluto ou relativo para o qual você deseja enviar o usuário. Lembre-se de ter um componente configurado nesse endereço. Para o código anterior, um componente com a diretiva `@page "/buypizza"` processará essa rota.

Usar componentes NavLink

Em um dos exemplos anteriores, o código foi usado para obter o valor `NavigationManager.BaseUri` e usá-lo para definir o atributo `href` de uma marca `<a>` como a home page. No Blazor, use o componente **NavLink** para renderizar marcas `<a>` porque ele alterna uma classe CSS `active` quando o atributo `href` do link corresponde à URL atual. Ao estilizar a classe `active`, você poderá deixar claro para o usuário qual link de navegação é para a página atual.

Quando o **NavLink** é usado, o exemplo de link da home page fica parecido com o seguinte código:



```
razor
@page "/pizzas"
@inject NavigationManager NavManager

<h1>Buy a Pizza</h1>

<p>I want to order a: @PizzaName</p>

<NavLink href=@HomePageURI Match="NavLinkMatch.All">Home Page</NavLink>

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

    public string HomePageURI { get; set; }

    protected override void OnInitialized()
    {
        HomePageURI = NavManager.BaseUri;
    }
}
```

O atributo `Match` no componente **NavLink** é usado para gerenciar quando o link é realçado. Há duas opções:

- NavLinkMatch.All**: quando esse valor é usado, o link só é realçado como o link ativo quando o `href` corresponde à URL atual inteira.
- NavLinkMatch.Prefix**: quando esse valor é usado, o link é realçado como ativo quando o `href` corresponde à primeira parte da URL atual. Suponha, por exemplo, que você tenha o link `<NavLink href="pizzas" Match="NavLinkMatch.Prefix">`. Esse link seria realçado como ativo se a URL atual fosse `http://www.contoso.com/pizzas` e para qualquer local dentro dessa URL, como `http://www.contoso.com/pizzas/formaggio`. Esse comportamento pode ajudar o usuário a entender qual seção do site está sendo exibida no momento.

Verificar seu conhecimento

1. Qual componente você deve usar para obter informações de localização de URI dentro de um componente Blazor?

☐ RouteManager

☐ NavigationManager

☐ ServiceLocator

2. Qual é a classe CSS padrão adicionada a uma marca de âncora pelo componente NavLink quando o NavLink referencia a URL atual?

☐ current

☐ live

☐ active

Exercício – Alterar a navegação no aplicativo Blazor usando a diretiva @page

O Blazor tem um auxiliar de estado de navegação que ajuda o código C# a gerenciar URIs de um aplicativo. Há também um componente **NavLink** que é uma substituição drop-in do elemento <a>. Um dos recursos do NavLink é adicionar uma classe ativa a links HTML para os menus de um aplicativo.

Sua equipe começou a trabalhar no aplicativo Blazing Pizza e criou componentes mais elaborados para representar pizzas e pedidos. O aplicativo agora precisa ter check-out e outras páginas relacionadas ao pedido.

Neste exercício, você adicionará uma nova página de check-out, adicionará uma navegação superior ao aplicativo e usará um componente **NavLink** do Blazor para aprimorar seu código.

Clonar o aplicativo existente de sua equipe

📌 Observação

Este módulo usa a CLI (interface de linha de comando) 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 (Windows) ou o Visual Studio para Mac (macOS). Para o desenvolvimento contínuo, use o Visual Studio Code para Windows, Linux e macOS.

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

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](#).

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**.

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, acesse o local em que deseja criar o projeto.
4. Clone o aplicativo do GitHub.

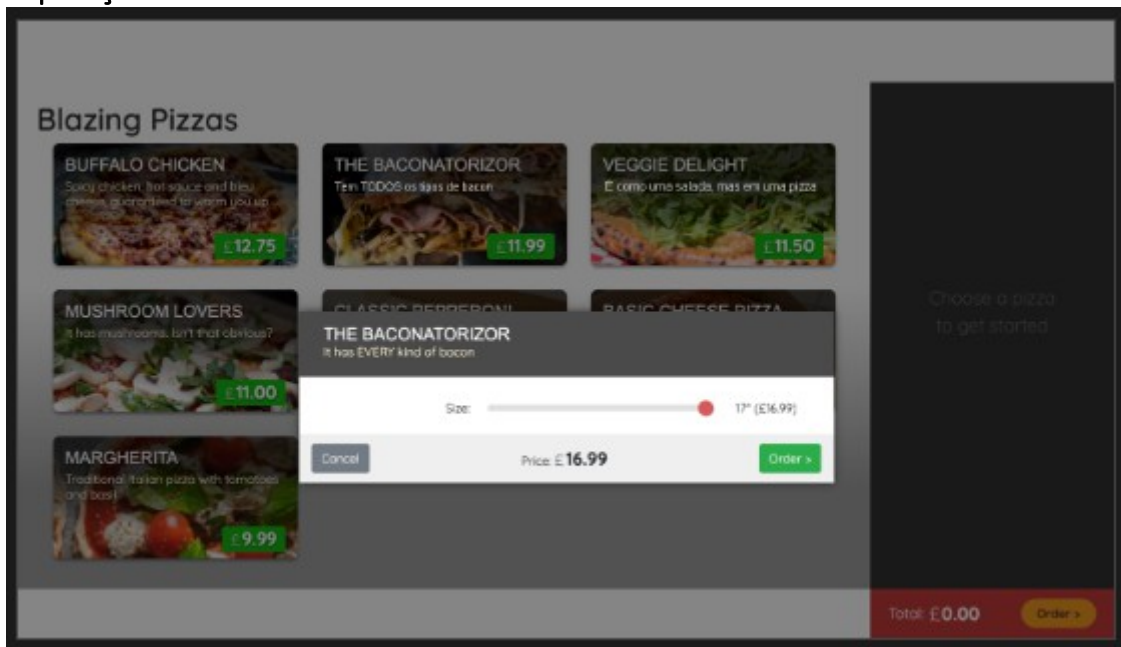
```
PowerShell Copiar  
git clone https://github.com/MicrosoftDocs/mslearn-blazor-navigation.git BlazingPizza
```

Selecione **Arquivo** e **Abrir pasta**.

1. Na caixa de diálogo **Abrir**, acesse a pasta **BlazingPizza** e escolha **Selecionar Pasta**. O Visual Studio Code pode solicitar as dependências não resolvidas.

Selecione **Restaurar**.

2. Execute o aplicativo para verificar se tudo está funcionando corretamente.
3. No Visual Studio Code, selecione F5. Ou, então, no menu **Executar**, selecione **Iniciar Depuração**.



Configure algumas pizzas e adicione-as ao seu pedido. Selecione **Pedido >** na parte inferior da página. Você verá a mensagem padrão "404 não encontrado", pois a equipe ainda não fez uma página de check-out.

1. Selecione Shift + F5 para interromper o aplicativo.

Adicionar uma página de check-out

1. No Visual Studio Code, no gerenciador de arquivos, selecione **App.razor**.

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


O bloco de código <NotFound> é o que os clientes verão se tentarem navegar para uma página que não existe.

- 1.No explorador de arquivos, expanda **Páginas**, clique com o botão direito do mouse na pasta e escolha **Novo Arquivo**.
- 2.Dê ao novo arquivo o nome **Checkout.razor**. Nesse arquivo, escreva o seguinte código:

```
@page "/checkout"
@inject OrderState OrderState
@inject HttpClient HttpClient
@inject NavigationManager NavigationManager

<div class="top-bar">
  <a class="logo" href="">
    
  </a>

  <a href="" class="nav-tab">
    
    <div>Get Pizza</div>
  </a>
</div>

<div class="main">
  <div class="checkout-cols">
    <div class="checkout-order-details">
      <h4>Review order</h4>
      @foreach (var pizza in Order.Pizzas)
      {
        <p>
          <strong>
            @(pizza.Size)
            @pizza.Special.Name
            (€@pizza.GetFormattedTotalPrice())
          </strong>
        </p>
      }

      <p>
        <strong>
          Total price:
          €@Order.GetFormattedTotalPrice()
        </strong>
      </p>
    </div>
    <div>
      <button class="checkout-button btn btn-warning">
        Place order
      </button>
    </div>
  </div>

  @code {
    Order Order => OrderState.Order;
  }
</div>
```

Essa página é criada no aplicativo atual e usa o estado do aplicativo salvo em OrderState. O primeiro div é a nova navegação de cabeçalho do aplicativo. Vamos adicioná-lo à página de índice.

- 1.No explorador de arquivos, expanda **Páginas** e selecione **index.razor**.
- 2.Acima da classe <div class="main">, adicione o HTML top-bar.

```
HTML


<a class="logo" href="">
    
  </a>

  <a href="" class="nav-tab">
    
    <div>Get Pizza</div>
  </a>
</div>


```

Quando estivermos nessa página, será interessante mostrar isso aos clientes realçando o link. A equipe já criou uma classe css active, portanto, adicione active ao atributo class que já contém o estilo nav-tab.

```
HTML

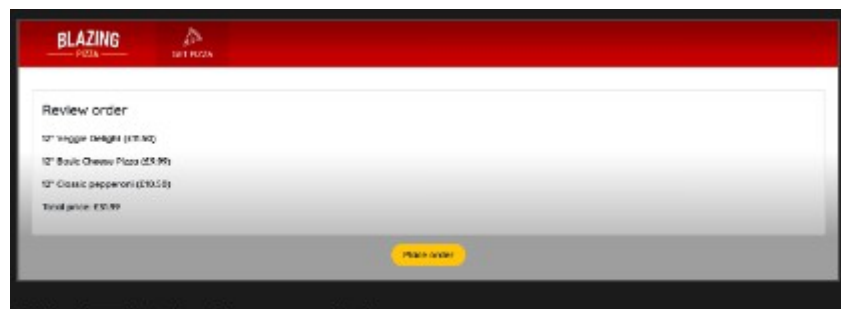

<a class="logo" href="">
    
  </a>

  <a href="" class="nav-tab active">
    
    <div>Get Pizza</div>
  </a>
</div>


```

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

O aplicativo agora tem uma barra de menus interessante na parte superior, que inclui o logotipo da empresa. Adicione algumas pizzas e avance o pedido até a página de check-out. Você verá as pizzas listadas e o indicador ativo ausente no menu.



Selecione Shift + F5 para interromper o aplicativo.

Permitir que os clientes façam um pedido

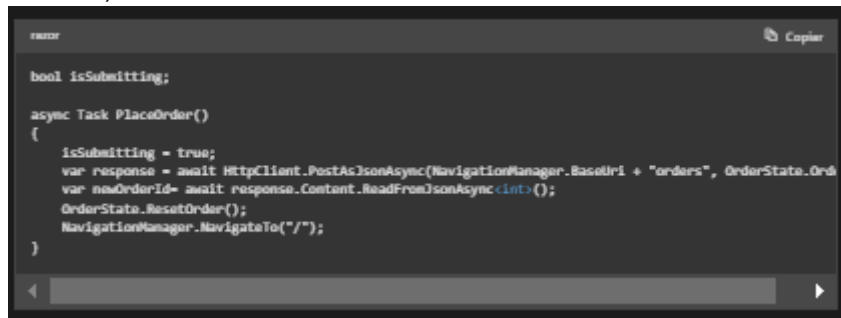
No momento, a página de check-out não permite que os clientes façam pedidos. A lógica do aplicativo precisa armazenar o pedido a fim de enviá-lo para a cozinha. Depois que o pedido é enviado, vamos redirecionar os clientes novamente para a home page.

- 1.No explorador de arquivos, expanda **Páginas** e selecione **Checkout.razor**.
- 2.Modifique o elemento de botão para chamar um método PlaceOrder. Adicione os atributos @onclick e disabled, conforme mostrado abaixo:

```
razor
<button class="checkout-button btn btn-warning" @onclick="PlaceOrder" disabled="@isSubmitting">
  Place order
</button>
```

Para que os clientes não façam pedidos duplicados, o botão **Fazer um pedido** foi desabilitado até a conclusão do processamento do pedido.

1.No bloco @code, adicione este código abaixo do código Order Order => OrderState.Order;.

A screenshot of a code editor with a dark theme. The code is in C# and defines a method named PlaceOrder. It starts with a bool variable isSubmitting. The method is an async Task that sets isSubmitting to true, then uses HttpClient to post a JSON object to the 'orders' endpoint. It then reads the response as a JSON integer, resets the OrderState, and navigates to the root path using the NavigationManager. A 'Copiar' button is visible in the top right corner of the editor.

```
bool isSubmitting;

async Task PlaceOrder()
{
    isSubmitting = true;
    var response = await HttpClient.PostAsJsonAsync(NavigationManager.BaseUri + "orders", OrderState.Order);
    var newOrderId = await response.Content.ReadFromJsonAsync<int>();
    OrderState.ResetOrder();
    NavigationManager.NavigateTo("/");
}
```

O código anterior desabilitará o botão **Fazer um pedido**, postará um JSON que será adicionado a **pizza.db**, limpará o pedido e usará o NavigationManager a fim de redirecionar o cliente para a home page.

Você precisa adicionar o código para tratar da ordem. Você adicionará uma classe **OrderController** para cumprir essa tarefa. Se você examinar **PizzaStoreContext.cs**, verá que há apenas o suporte ao banco de dados do Entity Framework para PizzaSpecials. Vamos corrigir isso primeiro.

Adicionar suporte do Entity Framework a pedidos e pizzas

- 1.No explorador de arquivos, selecione **PizzaStoreContext.cs**.
- 2.Substitua a classe PizzaStoreContext por este código:

C#Copiar

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

    public DbSet<Order> Orders { get; set; }

    public DbSet<Pizza> Pizzas { get; set; }

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

    public DbSet<Topping> Toppings { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```

    {
        base.OnModelCreating(modelBuilder);

        // Configuring a many-to-many special -> topping relationship that is friendly for
        serialization
        modelBuilder.Entity<PizzaTopping>().HasKey(pst => new { pst.PizzaId,
        pst.ToppingId });
        modelBuilder.Entity<PizzaTopping>().HasOne<Pizza>().WithMany(ps =>
        ps.Toppings);
        modelBuilder.Entity<PizzaTopping>().HasOne(pst => pst.Topping).WithMany();
    }

}

```

Esse código adiciona suporte do Entity Framework às classes order e pizza do aplicativo.

3.No menu do Visual Studio Code, escolha **Arquivo>Novo arquivo de texto**.

4.Selecione a linguagem C# e insira este código:

C#Copiar

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace BlazingPizza;

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

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

```

[HttpGet]

```
public async Task<ActionResult<List<OrderWithStatus>>> GetOrders()
{
    var orders = await _db.Orders
        .Include(o => o.Pizzas).ThenInclude(p => p.Special)
        .Include(o => o.Pizzas).ThenInclude(p => p.Toppings).ThenInclude(t => t.Topping)
        .OrderByDescending(o => o.CreatedTime)
        .ToListAsync();

    return orders.Select(o => OrderWithStatus.FromOrder(o)).ToList();
}
```

[HttpPost]

```
public async Task<ActionResult<int>> PlaceOrder(Order order)
{
    order.CreatedTime = DateTime.Now;

    // Enforce existence of Pizza.SpecialId and Topping.ToppingId
    // in the database - prevent the submitter from making up
    // new specials and toppings
    foreach (var pizza in order.Pizzas)
    {
        pizza.SpecialId = pizza.Special.Id;
        pizza.Special = null;
    }

    _db.Orders.Attach(order);
    await _db.SaveChangesAsync();

    return order.OrderId;
}
}
```

O código anterior permite que nosso aplicativo obtenha todos os pedidos atuais e faça um pedido. O atributo [Route("orders")] do Blazor permite que essa classe manipule solicitações HTTP de entrada para **/orders** e **/orders/{orderId}**.

5. Salve as alterações com Ctrl+S.

6. Como nome de arquivo, use **OrderController.cs**. Salve o arquivo no mesmo diretório como **OrderState.cs**.

7. No explorador de arquivos, selecione **OrderState.cs**.

8. Na parte inferior da classe no método `RemoveConfiguredPizza`, modifique `ResetOrder()` para redefinir a ordem:

```
C#Copiar
public void ResetOrder()
{
    Order = new Order();
}
```

Testar a funcionalidade de check-out

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

O aplicativo deve ser compilado, mas se você criar um pedido e tentar fazer check-out, verá um erro de runtime. O erro ocorre porque o banco de dados SQLite **pizza.db** foi criado antes do suporte para pedidos e pizzas. Precisamos excluir o arquivo para que um banco de dados possa ser criado corretamente.

2. Selecione Shift + F5 para interromper o aplicativo.

3. No explorador de arquivos, exclua o arquivo **pizza.db**.

4. Selecione F5. Ou, então, no menu **Executar**, selecione **Iniciar Depuração**.

Como um teste, adicione pizzas, vá para o check-out e faça um pedido. Você será redirecionado para a home page e verá que o pedido agora está vazio.

5. Selecione Shift + F5 para interromper o aplicativo.

O aplicativo está melhorando. Temos a configuração de uma pizza e um check-out.

Queremos permitir que os clientes vejam o status do pedido de pizza após o pedido.

Adicionar uma página de pedidos

1. No explorador de arquivos, expanda **Páginas**, clique com o botão direito do mouse na pasta e escolha **Novo Arquivo**.

2. Dê ao novo arquivo o nome **MyOrders.razor**. Nesse arquivo, escreva o seguinte código:

```
razorCopiar
@page "/myorders"
@inject HttpClient HttpClient
```

@inject NavigationManager NavigationManager

```
<div class="top-bar">
  <a class="logo" href="">
    
  </a>

  <a href="" class="nav-tab">
    
    <div>Get Pizza</div>
  </a>

  <a href="myorders" class="nav-tab active">
    
    <div>My Orders</div>
  </a>
</div>

<div class="main">
  @if (ordersWithStatus == null)
  {
    <text>Loading...</text>
  }
  else if (!ordersWithStatus.Any())
  {
    <h2>No orders placed</h2>
    <a class="btn btn-success" href="">Order some pizza</a>
  }
  else
  {
    <div class="list-group orders-list">
      @foreach (var item in ordersWithStatus)
      {
        <div class="list-group-item">
```



```

<div class="col">
    <h5>@item.Order.CreatedTime.ToLongDateString()</h5>
    Items:
    <strong>@item.Order.Pizzas.Count()</strong>;
    Total price:
    <strong>£@item.Order.GetFormattedTotalPrice()</strong>
</div>
<div class="col">
    Status: <strong>@item.StatusText</strong>
</div>
@if (@item.StatusText != "Delivered")
{
    <div class="col flex-grow-0">
        <a href="myorders/" class="btn btn-success">
            Track &gt;
        </a>
    </div>
}
</div>
}
</div>
}
</div>

```

```

@code {
    List<OrderWithStatus> ordersWithStatus = new List<OrderWithStatus>();

    protected override async Task OnParametersSetAsync()
    {
        ordersWithStatus = await HttpClient.GetFromJsonAsync<List<OrderWithStatus>>(
            $"{NavigationManager.BaseUri}orders");
    }
}

```

A navegação precisa ser alterada em todas as páginas. Agora precisamos incluir um link para a nova página **Meus pedidos**. Abra **Checkout.razor** e **Index.razor** e substitua a navegação por este código:

HTMLCopiar

```
<div class="top-bar">
  <a class="logo" href="">
    
  </a>

  <a href="" class="nav-tab active" >
    
    <div>Get Pizza</div>
  </a>

  <a href="myorders" class="nav-tab" >
    
    <div>My orders</div>
  </a>

</div>
```

Usando elementos `<a>`, precisamos gerenciar qual é a página ativa manualmente adicionando a classe CSS `active`. Vamos atualizar toda a navegação para usar um componente **NavLink**.

3. Nas três páginas com navegação (**Index.razor**, **Checkout.razor** e **MyOrders.razor**), use o mesmo código do Blazor para a navegação:

razorCopiar

```
<div class="top-bar">
  <a class="logo" href="">
    
  </a>

  <NavLink href="" class="nav-tab" Match="NavLinkMatch.All">
    
    <div>Get Pizza</div>
  </NavLink>
```

```

<NavLink href="myorders" class="nav-tab">
    
    <div>My Orders</div>
</NavLink>
</div>

```

A classe CSS **active** agora é adicionada automaticamente às páginas pelo componente **NavLink**. Você não precisa se lembrar de fazer isso em cada página em que há navegação.

4.A última etapa é alterar o **NavigationManager** a fim de redirecionar o usuário para a página **myorders** depois que um pedido for feito. No explorador de arquivos, expanda **Páginas** e selecione **Checkout.razor**.

5.Altere o método **PlaceOrder** a fim de redirecionar para a página correta transmitindo **/myorders** para **NavigationManager.NavigateTo()**:

razorCopiar

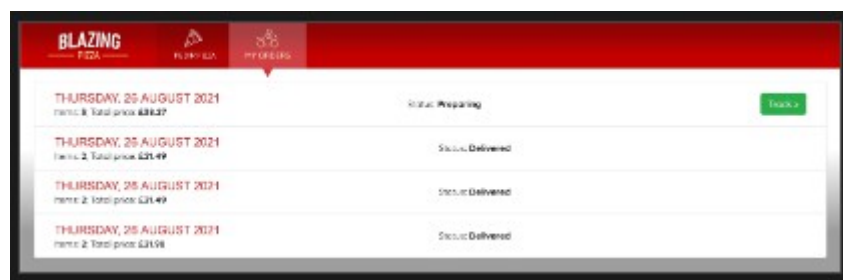
```

async Task PlaceOrder()
{
    isSubmitting = true;
    var response = await
HttpClient.PostAsJsonAsync($"{NavigationManager.BaseUri}orders", OrderState.Order);
    var newOrderId = await response.Content.ReadFromJsonAsync<int>();
    OrderState.ResetOrder();
    NavigationManager.NavigateTo("/myorders");
}

```

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

Você deve conseguir pedir algumas pizzas e ver os pedidos atualmente no banco de dados.



7.

8.Selecione Shift + F5 para interromper o aplicativo.

Explorar como os parâmetros de rota afetam o roteamento do aplicativo Blazor

200 XP

- 7 minutos

Você viu como, no Blazor, é possível usar partes do URI a fim de rotear solicitações para o componente correto. Você também pode interceptar outras partes do URI e acessá-las em seu código usando *parâmetros de rota*.

Suponha que você esteja trabalhando no site da empresa de entrega de pizza e tenha encaminhado pedidos de pizza para o componente **Pizzas.razor**. Agora, você deseja obter a pizza favorita do usuário do URI e usá-la para exibir informações sobre outras pizzas que talvez eles possam gostar.

Aqui, você aprenderá a usar parâmetros de rota para especificar partes da URL a serem processadas em seu código.

Observação

Os blocos de código desta unidade são exemplos ilustrativos. Você escreverá seu código na próxima unidade.

Parâmetros de rota

Anteriormente neste módulo, você aprendeu como partes do URI que o usuário solicita podem ser usadas a fim de rotear a solicitação para o componente certo. Normalmente, você usa outras partes do URI como um valor em sua página renderizada. Por exemplo, suponha que o usuário tenha solicitado:

`http://www.contoso.com/favoritepizza/hawaiian`

Usando a diretiva `@page`, você aprendeu a rotear essa solicitação para, por exemplo, o componente **FavoritePizza.razor**. Agora, você deseja usar o valor **havaiana** no componente. Para obter esse valor, você pode declará-lo como um parâmetro de rota.

Use a diretiva `@page` para especificar partes do URI que serão transmitidas ao componente como parâmetros de rota. No código do componente, você pode obter o valor de um parâmetro de rota da mesma maneira que obteria o valor de um parâmetro de componente:

razorCopiar

```
@page "/FavoritePizzas/{favorite}"
```

```
<h1>Choose a Pizza</h1>
```

```
<p>Your favorite pizza is: @Favorite</p>
```

```
@code {
```

```
    [Parameter]
```

```
    public string Favorite { get; set; }
```

```
}
```

O código anterior usa chaves na diretiva `@page` para especificar o parâmetro de rota e dar um nome a ele.

Observação

Parâmetros de componente são valores enviados de um componente pai a um componente filho. No pai, você especifica o valor do parâmetro de componente como um atributo da marca do componente filho. Os parâmetros de rota são especificados de maneira diferente. Eles são especificados como parte do URI. Nos bastidores, o roteador do Blazor intercepta esses valores e os envia ao componente como valores de componente, e é por isso você pode acessá-los da mesma maneira. Os parâmetros de rota não diferenciam maiúsculas de minúsculas e são encaminhados para um parâmetro de componente com o mesmo nome.

Parâmetros de rota opcionais

No exemplo anterior, o parâmetro `{favorite}` é obrigatório. Para tornar o parâmetro de rota opcional, use um ponto de interrogação:

```
razorCopiar
```

```
@page "/FavoritePizzas/{favorite?}"
```

```
<h1>Choose a Pizza</h1>
```

```
<p>Your favorite pizza is: @Favorite</p>
```

```
@code {  
    [Parameter]  
    public string Favorite { get; set; }  
  
    protected override void OnInitialized()  
    {  
        Favorite ??= "Fiorentina";  
    }  
}
```

É uma boa ideia definir um valor padrão para o parâmetro opcional. No exemplo anterior, o valor padrão do parâmetro `Favorite` é definido no método `OnInitialized`.

Observação

O método `OnInitialized` é executado quando os usuários solicitam a página pela primeira vez. Ele não é executado se eles solicitam a mesma página com um parâmetro de roteamento diferente. Por exemplo, se você espera que os usuários naveguem

de `http://www.contoso.com/favoritepizza/hawaiian` para `http://www.contoso.com/favoritepizza`, defina o valor padrão no método `OnParametersSet()`.

Restrições de rota

Nos exemplos anteriores, a consequência de solicitar o URI `http://www.contoso.com/favoritepizza/2` é uma mensagem estranha: "Sua pizza favorita é: 2". Em outros casos, tipos incompatíveis como esse podem causar uma exceção e exibir um erro para o usuário. Considere especificar um tipo para o parâmetro de rota:

```
razorCopiar
```

```
@page "/FavoritePizza/{preferredsize:int}"
```

```
<h1>Choose a Pizza</h1>
```

```
<p>Your favorite pizza size is: @FavoriteSize inches.</p>
```

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

Neste exemplo, se o usuário solicita `http://www.contoso.com/favoritepizza/margherita`, não há nenhuma correspondência com o componente anterior. Como resultado, a solicitação é roteada para outro lugar. Se o usuário solicitar `http://www.contoso.com/favoritepizza/12`, haverá uma combinação de rotas e o componente exibirá a mensagem "Seu tamanho de pizza favorito é: 30 centímetros". Um tipo específico para o parâmetro de rota como este é chamado de *restrição de rota*. Você pode usar estes outros tipos em uma restrição:

Expandir a tabela

Constraint	Exemplo	Correspondências de exemplo
bool	{vegan:bool}	<code>http://www.contoso.com/pizzas/true</code>
datetime	{birthdate:datetime}	<code>http://www.contoso.com/customers/1995-12-12</code>
decimal	{maxprice:decimal}	<code>http://www.contoso.com/pizzas/15.00</code>
double	{weight:double}	<code>http://www.contoso.com/pizzas/1.234</code>
float	{weight:float}	<code>http://www.contoso.com/pizzas/1.564</code>
guid	{pizza id:guid}	<code>http://www.contoso.com/pizzas/CD2C1638-1638-72D5-1638-DEADBEEF1638</code>
long	{total sales:long}	<code>http://www.contoso.com/pizzas/568192454</code>

Definir um parâmetro de rota catch-all

Considere o seguinte componente visto anteriormente nesta unidade:

razorCopiar

```
@page "/FavoritePizza/{favorite}"
```

```
<h1>Choose a Pizza</h1>
```

```
<p>Your favorite pizza is: @Favorite</p>
```

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

Agora, suponha que o usuário tentou especificar duas pizzas favoritas solicitando o URI `http://www.contoso.com/favoritepizza/margherita/hawaiian`. A página exibe a mensagem "Sua pizza favorita é: marguerita" e ignora a subpasta **havaiana**. Você pode alterar esse comportamento usando um parâmetro de rota *catch-all*, que captura caminhos entre vários limites de pasta de URI (barras `"/`). Coloque um asterisco (`*`) à frente do nome do parâmetro de rota para tornar o parâmetro de rota catch-all:

razorCopiar

```
@page "/FavoritePizza/{*favorites}"
```

```
<h1>Choose a Pizza</h1>
```

```
<p>Your favorite pizzas are: @Favorites</p>
```

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

Com o mesmo URI de solicitação, a página agora exibe a mensagem: "Suas pizzas favoritas são: marguerita/havaiana".

Verificar seu conhecimento

1.Qual é o formato correto para usar um parâmetro de rota a fim de capturar uma parte da URL que define a pizza favorita com a qual trabalhar?

☐/FavoritePizzas/{favorite}

☐/FavoritePizzas/[favorite]

☐/FavoritePizzas?favorite

2.Qual sintaxe demonstra um parâmetro de rota do Blazor com um parâmetro catch-all?

☐/FavoritePizzas/{favorite?}

☐/FavoritePizzas/{favorite:catchall}

☐/FavoritePizzas/{*favorite}

Exercício: usar parâmetros de rota para melhorar a navegação nos aplicativos

100 XP

•5 minutos

Os parâmetros de rota do Blazor permitem que os componentes acessem os dados transmitidos na URL. Os parâmetros de rota permitirão que nosso aplicativo acesse pedidos específicos pelo OrderId.

Os clientes querem poder ver mais informações sobre pedidos específicos. Você atualizará a página de check-out a fim de levar os clientes diretamente para seus pedidos feitos. Em seguida, você atualizará a página de pedidos para permitir que eles acompanhem todos os pedidos abertos no momento.

Neste exercício, você adicionará uma nova página de detalhes do pedido que usa parâmetros de rota. Você verá como adicionar uma restrição ao parâmetro para conferir o tipo de dados correto.

Criar uma página de detalhes do pedido

1.No menu do Visual Studio Code, escolha **Arquivo>Novo arquivo de texto**.

2.Selecione ASP.NET Razor como a linguagem.

3.Crie um componente de página de detalhes do pedido com este código:

```
razorCopiar
```

```
@page "/myorders/{orderId}"
```

```
@inject NavigationManager NavigationManager
```

```
@inject HttpClient HttpClient
```

```
<div class="top-bar">
  <a class="logo" href="">
    
  </a>

  <NavLink href="" class="nav-tab" Match="NavLinkMatch.All">
    
    <div>Get Pizza</div>
  </NavLink>

  <NavLink href="myorders" class="nav-tab">
    
    <div>My Orders</div>
  </NavLink>

</div>

<div class="main">
  @if (invalidOrder)
  {
    <h2>Order not found</h2>
    <p>We're sorry but this order no longer exists.</p>
  }
  else if (orderWithStatus == null)
  {
    <div class="track-order">
      <div class="track-order-title">
        <h2>
          <text>Loading...</text>
        </h2>
        <p class="ml-auto mb-0">
          ...
        </p>
      </div>
    </div>
  }
}
```

```

    </div>
}
else
{
    <div class="track-order">
        <div class="track-order-title">
            <h2>
                Order placed @orderWithStatus.Order.CreatedTime.ToLongDateString()
            </h2>
            <p class="ml-auto mb-0">
                Status: <strong>@orderWithStatus.StatusText</strong>
            </p>
        </div>
        <div class="track-order-body">
            <div class="track-order-details">
                @foreach (var pizza in orderWithStatus.Order.Pizzas)
                {
                    <p>
                        <strong>
                            @(pizza.Size)"
                            @pizza.Special.Name
                            (£@pizza.GetFormattedTotalPrice())
                        </strong>
                    </p>
                }
            </div>
        </div>
    </div>
}
</div>

@code {
    [Parameter] public int OrderId { get; set; }
}

```

```

OrderWithStatus orderWithStatus;
bool invalidOrder = false;

protected override async Task OnParametersSetAsync()
{
    try
    {
        orderWithStatus = await HttpClient.GetFromJsonAsync<OrderWithStatus>(
            $"{NavigationManager.BaseUri}orders/{OrderId}");
    }
    catch (Exception ex)
    {
        invalidOrder = true;
        Console.Error.WriteLine(ex);
    }
}
}

```

A página ficará semelhante ao componente **MyOrders**. Estamos fazendo uma chamada a **OrderController**, mas desta vez estamos fazendo um pedido específico. Queremos aquele que corresponde a OrderId. Vamos adicionar o código para processar essa solicitação.

4.Selecione Ctrl+S para salvar as alterações.

5.Como o nome do arquivo, use **OrderDetail.razor**. Salve o arquivo no diretório **Páginas**.

6.No explorador de arquivos, selecione **OrderController.cs**.

7.No método PlaceOrder, adicione um novo método para retornar pedidos com um status.

razorCopiar

```

[HttpGet("{orderId}")]
public async Task<ActionResult<OrderWithStatus>> GetOrderWithStatus(int orderId)
{
    var order = await _db.Orders
        .Where(o => o.OrderId == orderId)
        .Include(o => o.Pizzas).ThenInclude(p => p.Special)
        .Include(o => o.Pizzas).ThenInclude(p => p.Toppings).ThenInclude(t => t.Topping)
        .SingleOrDefaultAsync();
}

```

```

    if (order == null)
    {
        return NotFound();
    }

    return OrderWithStatus.FromOrder(order);
}

```

Esse código permitiu que o controlador **Order** respondesse a uma solicitação HTTP com a **orderId** na URL. Em seguida, o método usa esse ID para consultar o banco de dados e, se um pedido for encontrado, retornar um objeto **OrderWithStatus**.

Vamos usar essa nova página quando um cliente faz check-out. Você precisa atualizar o componente **Checkout.razor**.

8.No explorador de arquivos, expanda **Páginas**. Em seguida, selecione **Checkout.razor**.

9.Altere a chamada a `NavigationManager.NavigateTo("/myorders")` para usar a ID do pedido feito.

razorCopiar

```
NavigationManager.NavigateTo($"myorders/{newOrderId}");
```

O código existente já estava capturando a `newOrderId` como a resposta ao pedido feito. Agora você pode usá-la para acessar diretamente esse pedido.

Restringir o parâmetro de rota ao tipo de dados correto

O aplicativo só deve responder a solicitações com IDs de pedido numéricas, como (`http://localhost:5000/myorders/6`). Não há nada que impeça alguém de tentar usar pedidos não numéricos. Vamos mudar isso.

1.No explorador de arquivos, expanda **Páginas**. Em seguida, selecione **OrderDetail.razor**.

2.Altere o parâmetro de rota para que o componente aceite apenas inteiros.

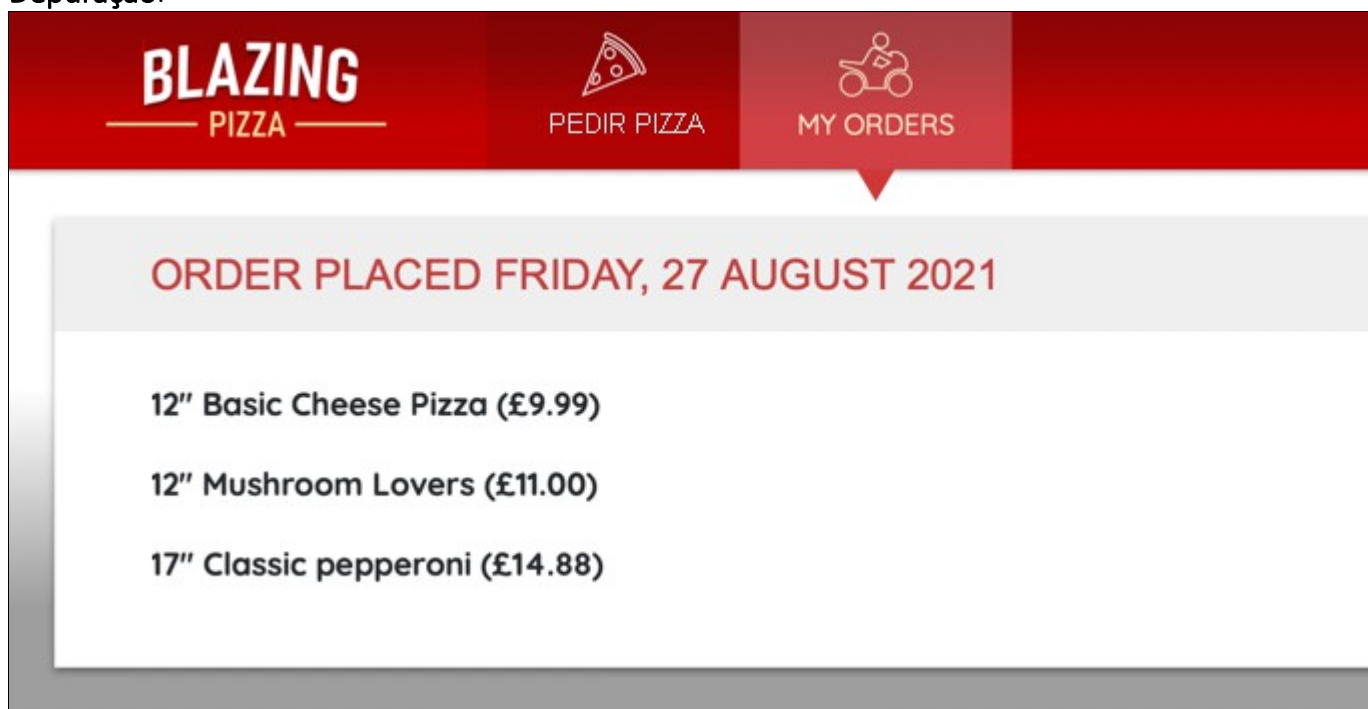
razorCopiar

```
@page "/myorders/{orderId:int}"
```

3.Agora, se alguém tentar acessar (`http://localhost:5000/myorders/non-number`), o roteamento do Blazor não encontrará uma correspondência para a URL e retornará o aviso de página não encontrada.

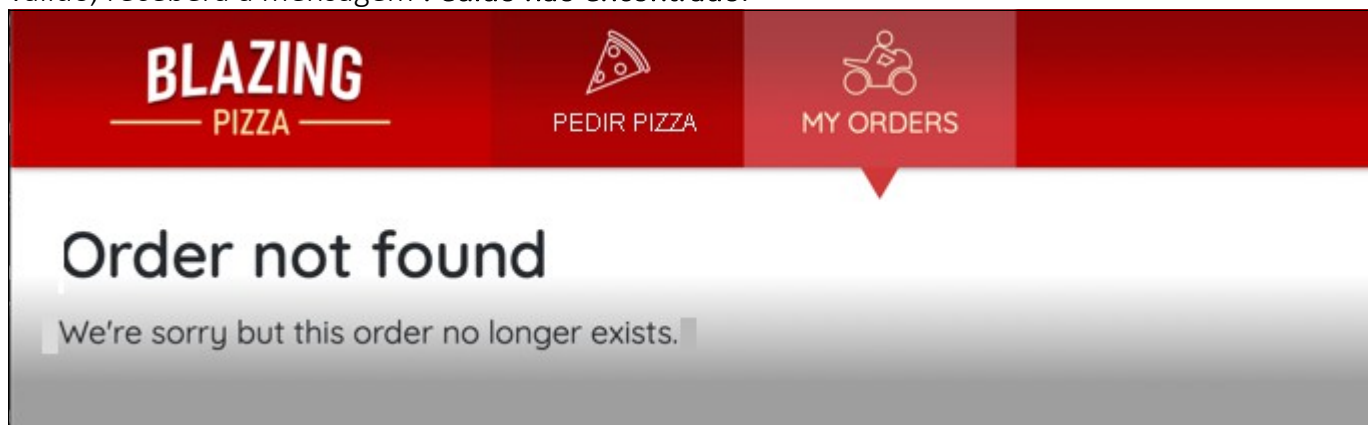
Não há nada neste endereço.

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



Acesse o aplicativo, faça o pedido e finalize a compra. Você será levado para a tela de pedido detalhada e verá o status do pedido.

5.Experimente IDs de pedidos diferentes. Se você usar um inteiro que não é um pedido válido, receberá a mensagem **Pedido não encontrado**.



Se você usar IDs de pedido não inteiras, verá a página Pedido não encontrado. E o mais importante: o aplicativo não terá uma exceção sem tratamento.

6. Selecione Shift + F5 para interromper o aplicativo.

Atualizar a página de pedidos

No momento, a página **Meus Pedidos** tem links para exibir mais detalhes, mas a URL está errada.

1. No explorador de arquivos, expanda **Páginas**. Em seguida, selecione **MyOrders.razor**.
2. Substitua o elemento `` por este código:

razorCopiar

```
<a href="@item.Order.OrderId" class="btn btn-success">
```

Você pode testar como isso funciona fazendo seu último pedido de pizza neste exercício. Selecione **Meus Pedidos** e siga o link **Acompanhar >**.

Criar componentes do Blazor reutilizáveis usando layouts

200 XP

- 7 minutos

O Blazor inclui layouts para facilitar a codificação de elementos comuns da interface do usuário que aparecerão em muitas páginas no aplicativo.

Suponha que você esteja trabalhando no site da empresa de entrega de pizza e tenha criado o conteúdo para a maioria das páginas principais como um conjunto de componentes do Blazor. Você deseja garantir que essas páginas tenham identidade visual, menus de navegação e seção de rodapé iguais, mas não deseja copiar e colar esse código em vários arquivos.

Aqui, você aprenderá a usar componentes de layout no Blazor para renderizar HTML comum em várias páginas.

Observação

Os blocos de código desta unidade são exemplos ilustrativos. Você escreverá seu código na próxima unidade.

O que são layouts do Blazor?

Na maioria dos sites, a organização dos elementos da interface do usuário é compartilhada entre várias páginas. Por exemplo, pode haver uma faixa com identidade visual na parte superior da página, links de navegação do site principal no lado esquerdo e um aviso de isenção de responsabilidade legal na parte inferior. Depois que você codifica esses elementos comuns da interface do usuário em uma página, é entediante copiá-los e colá-los no código de todas as outras páginas. Pior: se houver uma alteração posterior, como uma nova seção principal do site a ser vinculada ou uma reformulação da identidade visual do site, você precisará fazer as mesmas alterações se repetirem em todos os componentes

individuais. Use um *componente de layout* para simplificar e reutilizar elementos comuns da interface do usuário.

Um componente de layout no Blazor é aquele que compartilha sua marcação renderizada com todos os componentes que a referenciam. Você coloca elementos comuns da interface do usuário, como menus de navegação, identidade visual e rodapés no layout. Em seguida, você faz referência a esse layout em vários outros componentes. Quando a página é renderizada, os elementos comuns vêm do layout e os elementos exclusivos, como os detalhes da pizza solicitada, vêm do componente de referência. Você só precisa codificar os elementos comuns da interface do usuário uma vez no layout. Em seguida, se houver uma reformulação da identidade visual ou outra alteração, você só precisará corrigir o layout. A alteração se aplica automaticamente a todos os componentes de referência.

Codificar um layout do Blazor

Um layout do Blazor é um tipo específico de componente, ou seja, escrever um layout do Blazor é uma tarefa semelhante a escrever outros componentes para renderizar a interface do usuário no seu aplicativo. Por exemplo, você usa o bloco `@code` e muitas diretivas da mesma maneira. Os layouts são definidos em arquivos com uma extensão **.razor**. O arquivo geralmente é armazenado na pasta **Compartilhados** no seu aplicativo, mas você pode optar por armazená-lo em qualquer local que possa ser acessado pelos componentes que o usam. Dois requisitos são exclusivos dos componentes de layout:

- Você precisa herdar a classe `LayoutComponentBase`.
- Você precisa incluir a diretiva `@Body` no local em que deseja fazer referência ao conteúdo dos componentes a ser renderizado.

razorCopiar

```
@inherits LayoutComponentBase
```

```
<header>
```

```
    <h1>Blazing Pizza</h1>
```

```
</header>
```

```
<nav>
```

```
    <a href="/Pizzas">Browse Pizzas</a>
```

```
    <a href="/Toppings">Browse Extra Toppings</a>
```

```
    <a href="/FavoritePizzas">Tell us your favorite</a>
```

```
    <a href="/Orders">Track Your Order</a>
```

```
</nav>
```

```
@Body
```

```
<footer>
    @new MarkdownString(TrademarkMessage)
</footer>
```

```
@code {
    public string TrademarkMessage { get; set; } = "All content is &copy; Blazing Pizzas
2021";
}
```

Observação

Os componentes de layout não incluem uma diretiva `@page`, pois não tratam as solicitações diretamente e não devem ter uma rota criada. Em vez disso, os componentes de referência usam a diretiva `@page`.

Se você criou seu aplicativo Blazor com base em um modelo de projeto Blazor, o layout padrão do aplicativo é o componente **Shared/MainLayout.razor**.

Usar um layout em um componente do Blazor

Para usar um layout de outro componente, adicione a diretiva `@layout` com o nome do layout a ser aplicado. O HTML do componente será renderizado na posição da diretiva `@Body`.

```
razorCopiar
@page "/FavoritePizzas/{favorite}"
@layout BlazingPizzasMainLayout
```

```
<h1>Choose a Pizza</h1>
```

```
<p>Your favorite pizza is: @Favorite</p>
```

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

Este diagrama ilustra como um componente e um layout são combinados para renderizar o HTML final:



Caso deseje aplicar um modelo a todos os componentes do Blazor de uma pasta, use o arquivo **_Imports.razor** como atalho. Quando o compilador do Blazor encontra esse arquivo, ele inclui suas diretivas em todos os componentes na pasta automaticamente. Essa técnica remove a necessidade de adicionar a diretiva `@layout` cada componente e se aplica a componentes na mesma pasta que o arquivo **_Imports.razor** e que todas as suas subpastas.

Importante

Não adicione uma diretiva `@layout` ao arquivo **_Imports.razor** na pasta raiz do projeto porque isso resulta em um loop infinito de layouts.

Se você quiser aplicar um layout padrão a cada componente em todas as pastas do aplicativo Web, poderá fazer isso no componente **App.razor**, no qual você configura o componente **Router** como aprendeu na unidade 2. Na marca `<RouteView>`, use o atributo `DefaultLayout`.

razorCopiar

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

Os componentes que têm um layout especificado em sua própria diretiva `@layout` ou em um arquivo **_Imports.Razor** substituirão essa configuração de layout padrão.

Verificar seu conhecimento

1.Qual é a sintaxe correta para especificar um layout chamado MyLayout em uma página do Blazor?

☐ @page layout='MyLayout'

☐ @layout MyLayout

☐ @page MyLayout

Verificar suas respostas

Exercício: adicionar um layout do Blazor para reduzir a duplicação no código

100 XP

4 minutos

c

l

u

í

d

o

Conforme você adicionou páginas ao aplicativo Blazing Pizza, reparou que copiamos o HTML de navegação. O Blazor tem suporte interno à criação desse tipo de scaffolding de página em um só lugar. Eles são chamados de layouts do Blazor.

Agora temos vários HTML duplicados em várias páginas. Em seguida, você criará um layout para todo o aplicativo a fim de adicionar informações de navegação e da empresa em um único local.

Neste exercício, você criará um componente **MainLayout**. Você verá como pode usar esse layout em páginas específicas e torná-lo o layout padrão em todo o aplicativo.

Adicionar um componente MainLayout

- 1.No menu do Visual Studio Code, escolha **Arquivo>Novo arquivo de texto**.
- 2.Selecione ASP.NET Razor como a linguagem.
- 3.Crie um componente de layout e copie o HTML de navegação de qualquer página.

```
razorCopiar
```

```
@inherits LayoutComponentBase
```

```
<div id="app">
```

```
<header class="top-bar">
```

```
<a class="logo" href="">
```

```

```

```
</a>
```

```
<NavLink href="" class="nav-tab" Match="NavLinkMatch.All">
```

```

```

```
<div>Get Pizza</div>
```

```
</NavLink>
```

```
<NavLink href="myorders" class="nav-tab">
```

```

```

```
<div>My Orders</div>
```

```
</NavLink>
```

```
</header>
```

```
<div class="content">
```

```
@Body
```

```
</div>
```

```
<footer>
```

```
&copy; Blazing Pizza @DateTime.UtcNow.Year
```

```
</footer>
```

```
</div>
```

Esse layout usou algumas das marcações de `_Host.cshtml`, ou seja, precisamos removê-las dele.

4. Selecione Ctrl+S para salvar as alterações.

5. Como o nome de arquivo, use **MainLayout.razor**. Salve o arquivo no diretório **Compartilhados**.

6. No explorador de arquivos, expanda **Páginas**. Em seguida, selecione `_Host.cshtml`.

7. Altere os elementos em torno do componente de aplicativo Blazor deste código:

razorCopiar

```
<div id="app">
```

```
<div class="content">
```

```
<component type="typeof(App)" render-mode="ServerPrerendered" />
```

```
</div>
```

```
</div>
```

Para este código:

razorCopiar

```
<component type="typeof(App)" render-mode="ServerPrerendered" />
```

Usar um layout do Blazor em um componente de página

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

2. Exclua o elemento div top-bar e, sob a diretiva `@page`, adicione uma referência ao novo layout.

razorCopiar

```
@layout MainLayout
```

3. Também vamos excluir o elemento `<h1>Blazing Pizzas</h1>` antigo. Ele não é mais necessário porque temos uma barra superior no layout.

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



A nova home page terá um rodapé de direitos autorais deste ano que é atualizado automaticamente. Para ver a aparência de uma página que não está usando esse layout, selecione **Meus Pedidos**. Ou, então, acesse uma página inválida como (<http://localhost:5000/help>). No momento, a página **404 Página não encontrada** não está usando nenhuma das identidades visuais.



5. Selecione Shift + F5 para interromper o aplicativo.

Atualizar todas as páginas para usar o novo layout

Agora você pode adicionar a diretiva `@layout MainLayout` a todas as páginas em nosso aplicativo. O Blazor tem uma solução melhor. Primeiro, exclua a diretiva de layout recém-adicionada a **Index.razor**.

1. No componente **Index.razor**, exclua `@layout MainLayout`.
2. No explorador de arquivos, expanda **Páginas**. Em seguida, selecione **MyOrders.razor**.
3. Exclua o elemento `div top-bar`.
4. No explorador de arquivos, expanda **Páginas**. Em seguida, selecione **OrderDetail.razor**.
5. Exclua o elemento `div top-bar`.
6. No explorador de arquivos, expanda **Páginas**. Em seguida, selecione **Checkout.razor**.
7. Exclua o elemento `div top-bar`.

O componente **App.razor** é o local em que podemos alterar o modo como as páginas são roteadas e instruir o Blazor a usar um layout padrão.

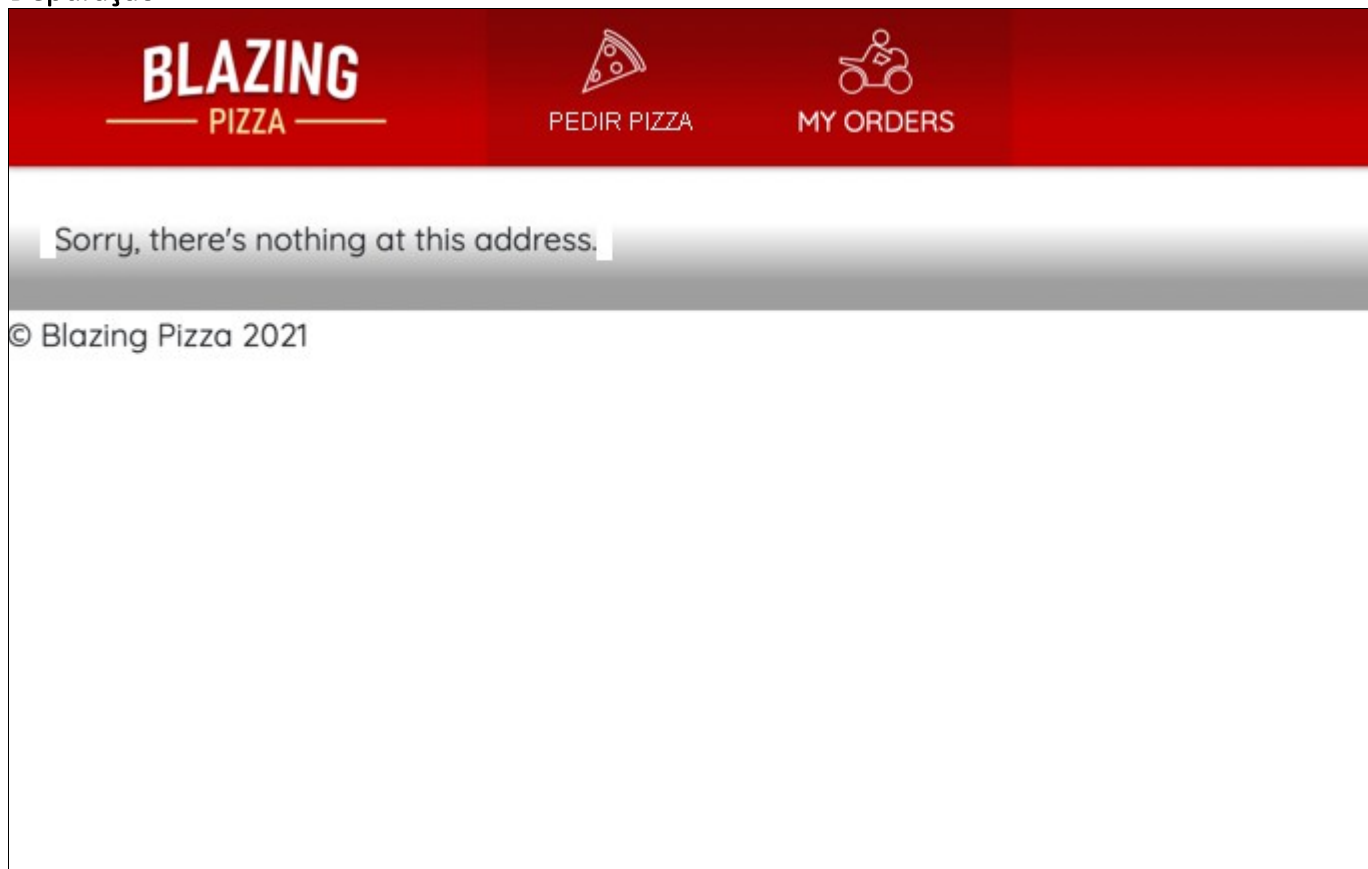
1. No explorador de arquivos, selecione **App.razor**.
2. Adicione a declaração `DefaultLayout="typeof(MainLayout)"` ao elemento **RouteView**.
3. Adicione `Layout="typeof(MainLayout)"` a **LayoutView**.
4. O **App.razor** ficará parecido com este exemplo:

razorCopiar

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

Testar o novo layout

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



O benefício de usar um layout padrão é que você pode aplicá-lo a todas as páginas e usá-lo para a exibição de layout em páginas não encontradas.

2.Selecione Shift + F5 para interromper o aplicativo.

O trabalho que você precisava fazer em relação ao aplicativo foi concluído neste módulo. Caso deseje ver como tratar os formulários e a validação, conclua o próximo módulo deste roteiro de aprendizagem.

Resumo

100 XP

3 minutos

O aplicativo da empresa de pizza agora tem uma navegação melhor. Você simplificou o aplicativo usando layouts para compartilhar as disposições de página comuns em todo o aplicativo. Você viu como é possível compartilhar dados entre parâmetros de página e componentes do Blazor para melhorar a funcionalidade do aplicativo.

Blazor permite que você reutilize sua experiência em C# e traga-a para o desenvolvimento da Web de front-end.

Saiba mais

- Introdução ao Blazor do ASP.NET Core
- Roteamento do Blazor ASP.NET Core
- Layouts do Blazor ASP.NET Core