Blazor Revelado

Construindo Aplicativos da Web em .NET

-

Peter Himschoot

Blazor Revelado

Construindo Aplicativos da Web em .NET

Peter Himschoot

Página 3

Blazor Revelado: Construindo Aplicativos da Web em .NET

Peter Himschoot
Melle, Bélgica

ISBN-13 (pbk): 978-1-4842-4342-8 ISBN-13 (eletrônico): 978-1-4842-4343-5
https://doi.org/10.1007/978-1-4842-4343-5

Número de controle da Biblioteca do Congresso: 2019932722

Diretor administrativo, Apress Media LLC: Welmoed Spahr
Editor de aquisições: Jonathan Gennick
Editor de Desenvolvimento: Laura Berendson
Editor Coordenador: Jill Balzano

Capa desenhada por eStudioCalamar

Imagem da capa desenhada por Freepik (www.freepik.com)

Distribuído para o mercado mundial de livros pela Springer Science + Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Telefone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny @ springer-sbm.com ou visite www.springeronline.com. Apress Media, LLC é uma California LLC e o único membro (proprietário) é Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc é uma Corporação **Delaware** .

Página 4

Índice

Página 5

Página 6

Página 7

Página 8

Página 9

Página 10

Sobre o autor

Peter Himschoot trabalha como treinador líder, arquiteto e estrategista da U2U Training. Peter tem um grande interesse em desenvolvimento de software, que inclui aplicativos para a Web, Windows e dispositivos móveis. Peter treinou milhares de desenvolvedores, é um palestrante regular em conferências, e esteve envolvido em muitas web e dispositivos móveis projetos de desenvolvimento como arquiteto de software. Peter também é um Diretor Regional da Microsoft, um grupo de consultores confiáveis para o públicos de desenvolvedores e profissionais de TI e para a Microsoft.

Página 11

Sobre o Revisor Técnico

Gerald Versluis é desenvolvedor e Microsoft MVP da Holanda, com anos de experiência trabalhando com Xamarin, Azure, ASP.NET e outras tecnologias .NET. Ele tem esteve envolvido em inúmeros projetos, em várias funções. UMA grande número de seus projetos são aplicativos Xamarin. Não somente Gerald gosta de codificar, mas está interessado em divulgar seu conhecimento, bem como ganhar algum na barganha. Ele fala, oferece sessões de treinamento e escreve blogs e artigos nas horas vagas.

Página 12

Agradecimentos

Quando Jonathan Gennick da Apress me perguntou se eu estaria interessado em escrever um livro em Blazor, me senti honrado e, claro, concordei que Blazor merece um livro. Escrevendo um livro é um esforço de grupo, então agradeço Jonathan Gennick e Jill Balzano por me darem dicas sobre como estilizar e escrever este livro, e agradeço a Gerald Versluis por fazer a parte técnica reveja e aponte as seções que precisavam de um pouco mais de explicação. Eu também agradeço a magda Thielman e Lieven Iliano do U2U Training, meu empregador, por me encorajar a escreva este livro.

Gostei muito de escrever este livro e espero que você goste de ler e aprender com isso.

Página 13

Introdução ao WebAssembly e Blazor

Eu estava participando do *Encontro de Diretores Regionais e Profissionais Mais Valiosos* da *Microsoft* quando fomos apresentados ao Blazor pela primeira vez por *Steve Sanderson* e *Daniel Roth* . E devo admitir que fiquei super empolgado com Blazor! Blazor é uma estrutura que permite que você crie aplicativos de página única (SPAs) usando C # e permite que você execute qualquer biblioteca .NET padrão no navegador. Antes do Blazor, suas opções para construir um SPA eram JavaScript ou uma das outras linguagens de nível superior, como TypeScript, que são compiladas em JavaScript de qualquer maneira. Nesta introdução, veremos como os navegadores agora são capazes de executar assemblies .NET no navegador usando WebAssembly, Mono e Blazor.

> blazor é, no momento em que este artigo foi escrito, uma estrutura eXPeRIMenTal. Espero que com o tempo você está lendo este livro que foi oficializado pela Microsoft.

A Tale of Two Wars

Pense nisso. O navegador é um dos principais aplicativos do computador. Vocês use-o todos os dias. As empresas que criam navegadores sabem disso muito bem e estão fazendo lances para você usar o navegador deles. No início da Internet mainstream, todo mundo estava usando *Netscape* . A Microsoft queria uma fatia do mercado, então em 1995 ela criou o *Internet Explorer 1.0* , lançado como parte do Windows 95 Plus! pacote. Versões mais recentes foram lançadas rapidamente e os navegadores começaram a adicionar novos recursos, como os elementos <blink> e <marquee>. Este foi o início da primeira guerra de navegadores, dando às pessoas (especialmente designers) dores de cabeça porque alguns desenvolvedores estavam construindo páginas com controles de marcação piscando ☺. Mas os desenvolvedores também estavam ficando doloridos por causa das incompatibilidades entre os navegadores. *A primeira guerra dos navegadores foi sobre ter mais recursos HTML do que os concorrentes.*

Página 14

Mas tudo isso ficou para trás com a introdução do HTML5 e do moderno navegadores como Google Chrome, Microsoft Edge, Firefox e Opera. HTML5 não só define uma série de elementos HTML padrão, mas também regras sobre como eles devem ser renderizados, tornando muito mais fácil construir um site que tenha a mesma aparência em todos os navegadores modernos.

Mas vamos voltar a 1995, quando *Brendan Eich* escreveu uma pequena linguagem de programação conhecido como *JavaScript* (inicialmente chamado de *LiveScript*) em 10 dias (O quê !?). Era Chamado JavaScript porque sua sintaxe era muito semelhante a Java.

> Javascript e Java não estão relacionados. Java e Javascript têm tanto em comum como presunto e hamster (não sei quem formulou isso primeiro, mas eu adoro esta frase).

Mal sabia o Sr. Eich como essa linguagem impactaria a Web moderna e até mesmo o desenvolvimento de aplicativos de desktop. Em 1995, *Jesse James Garett* escreveu um white paper chamado *Ajax (Asynchronous JavaScript and XML)* , que descreve um conjunto de tecnologias onde JavaScript é usado para carregar dados do servidor e esses dados são usados para atualizar o HTML do navegador, evitando, assim, recarregamentos completos da página e permitindo a web do lado do cliente aplicativos (aplicativos escritos em JavaScript que são executados completamente no navegador). Uma das primeiras empresas a aplicar o Ajax foi a Microsoft, quando criou o *Outlook Web Access (OWA)* . OWA é um aplicativo da web quase idêntico ao aplicativo Outlook para desktop mas fornecendo o poder do Ajax. Logo outros aplicativos Ajax começaram a aparecer, com O Google Maps ficou na minha memória como um dos outros aplicativos fundamentais. Google O Maps baixaria mapas de forma assíncrona e com algumas interações simples do mouse permitiu que você ampliasse e movesse o mapa. Antes do Google Maps, o servidor faria a renderização do mapa e um navegador exibiria o mapa como qualquer outra imagem por baixar um bitmap de um servidor.

Construir um site Ajax foi uma tarefa importante, que apenas grandes empresas como a Microsoft e o Google poderiam pagar. Isso logo mudou com a introdução de Bibliotecas JavaScript como jQuery e knockout.js. Hoje podemos construir aplicativos da web ricos com Angular, React e Vue.js. Todos eles usam JavaScript ou linguagens de nível superior, como TypeScript, que é compatível com JavaScript. O que nos traz de volta ao JavaScript e a segunda guerra do navegador. O desempenho do JavaScript é fundamental em navegadores modernos. Chrome, Edge, Firefox e Safari estão competindo entre si, tentando convencer os usuários de que seu navegador é o mais rápido, com nomes interessantes para seus Motor JavaScript como *V8* e *Chakra* . Esses mecanismos usam os truques de otimização mais recentes

Página 15

como a compilação Just-in-Time (JIT), onde o JavaScript é convertido em código nativo, como ilustrado pela figura 1 .

Figura 1. *O processo de execução de JavaScript*

Este processo exige muito esforço porque o JavaScript precisa ser baixado no navegador, onde é analisado, compilado em bytecode e JIT convertido em Código nativo. Então, como podemos tornar esse processo ainda mais rápido?

A segunda guerra do navegador é toda sobre o desempenho do JavaScript.

Apresentando WebAssembly

O WebAssembly permite que você analise e compile para o servidor. Com WebAssembly você compila seu código em um formato chamado WASM (uma abreviação de WebASseMbly), que é baixado pelo navegador onde é compilado o JIT em código nativo, conforme mostrado na Figura 2. Abra seu navegador e google " *webassembly demo zen garden.* " Um dos os links são https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html onde você pode ver uma demonstração impressionante de ray-trace de um jardim Zen japonês, mostrado em Figura 3 .

Página 16

Figura 2. *O processo de execução do WebAssembly*

Figura 3. *Jardim Zen Japonês*

Página 17

No site oficial, www.webassembly.org :

WebAssembly (abreviado Wasm) é um formato de instrução binário para um sistema baseado em pilha máquina virtual. Wasm é projetado como um alvo portátil para compilação de alto nível linguagens como C / C ++ / Rust, permitindo implantação na web para cliente e servidor formulários.

Portanto, WebAssembly é um novo formato binário otimizado para execução em navegador; não é JavaScript. Existem compiladores para linguagens como C ++ e Rust que compilam para WASM.

Quais navegadores suportam WebAssembly?

WebAssembly é compatível com todos os principais navegadores: Chrome, Edge, Safari e Firefox, incluindo suas versões móveis. À medida que WebAssembly se torna cada vez mais importante, veremos outros navegadores modernos seguirem o exemplo, mas não espere que o Internet Explorer suporte WASM.

WebAssembly e Mono

Mono é uma implementação de código aberto da especificação .NET CLI, o que significa que Mono é uma plataforma para executar assemblies .NET. Mono é usado em *Xamarin* para construção aplicativos móveis executados em dispositivos móveis Windows, Android e iOS sistemas. Mono também permite que você execute .NET no Linux (sua finalidade original) e é escrito em C ++. Esta última parte é importante porque você viu que pode compilar C ++ para WebAssembly. Então, o que aconteceu é que a equipe do Mono decidiu tentar compilar o Mono para WebAssembly, o que eles fizeram com sucesso. Existem duas abordagens. Um é onde você pega seu código .NET e o compila junto com o tempo de execução Mono em um grande aplicativo WASM. No entanto, essa abordagem leva muito tempo porque você precisa tome várias etapas para compilar tudo em WASM, o que não é tão prático para o dia-a-dia desenvolvimento do dia. A outra abordagem pega o tempo de execução Mono, compila-o no WASM, e isso é executado no navegador onde executará a linguagem intermediária .NET apenas como o .NET normal faz. A grande vantagem é que você pode simplesmente executar assemblies .NET sem ter que compilá-los primeiro no WASM. Esta é a abordagem atualmente adotada por Blazor. Mas Blazor não é o único que está adotando essa abordagem. Por exemplo, o projeto *Ooui* permite que você execute aplicativos *Xamarin.Forms* no navegador. A desvantagem disso é que ele precisa baixar muitos assemblies .NET. Isso pode ser resolvido usando a *árvore* Algoritmos de *agitação* , que removem todo o código não utilizado dos assemblies. Essas ferramentas não são ainda disponíveis, mas eles estão no pipeline.

Página 18

Interagindo com o navegador com o Blazor

WebAssembly com Mono permite que você execute código .NET no navegador. *Steve Sanderson* usei isso para construir o Blazor. Blazor usa a abordagem popular ASP.NET MVC para a construção aplicativos executados no navegador. Com o Blazor, você constrói arquivos Razor (Blazor = Navegador + Razor) que executam dentro do navegador para construir dinamicamente uma página web. Com Blazor, você não precisa de JavaScript para construir um aplicativo da web, o que é uma boa notícia para milhares de .NET

desenvolvedores que desejam continuar usando C # (ou F #).

Como funciona?

Vamos começar com um arquivo Razor simples. Veja a lista1 , que você pode encontrar ao criar um novo projeto Blazor.

Listagem 1. O arquivo Counter Razor

```
@page "/ counter"

<h1> Contador </h1>

<p> Contagem atual: @currentCount </p>

<button class = "btn btn-primary" onclick = "@ IncrementCount"> Clique aqui </button>

@funções {
    int currentCount = 0;

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

Este arquivo é compilado em código .NET (você descobrirá como mais tarde neste livro), que é então executado pelo motor Blazor. O resultado desta execução é uma estrutura semelhante a uma árvore chamada de *árvore de renderização* . A árvore de renderização é então enviada para JavaScript, que atualiza o DOM para refletir a árvore de renderização (criando, atualizando e removendo elementos HTML e atributos). Listagem1 resultará em h1, p (com o valor de currentCount) e botão Elementos HTML. Quando você interage com a página, por exemplo, quando você clica no

xx

Página 19

botão, isso irá acionar o evento de clique do botão, que invocará o IncrementCount método da lista 1 . A árvore de renderização é então regenerada e todas as alterações são enviadas novamente para JavaScript, que atualizará o DOM. Este processo é ilustrado na Figura 4 .

Este modelo é muito flexível. Ele permite que você crie *aplicativos da web progressivos* e também pode ser incorporado em aplicativos de desktop *Electron* , dos quais o Visual Studio Code é um dos principais exemplo.

Figura 4. *O processo de geração do Blazor DOM*

Blazor do lado do servidor

Em 7 de agosto de 2018, Daniel Roth apresentou um novo modelo de execução para Blazor chamado Blazor do lado do servidor no *standup da comunidade ASP.NET* . Neste modelo, seu site Blazor é executado no servidor, resultando em um download muito menor para o navegador.

O modelo do lado do servidor

Você acabou de ver que o Blazor do lado do cliente constrói uma *árvore de renderização* usando o tempo de execução Mono, que em seguida, é enviado ao JavaScript para atualizar o DOM. Com o Blazor do lado do servidor, a árvore de renderização é construído no servidor e, em seguida, serializado para o navegador usando *SignalR* . JavaScript no navegador, em seguida, desserializa a árvore de renderização para atualizar o DOM, o que é bastante semelhante ao modelo Blazor do lado do cliente. Quando você interage com o site, os eventos obtêm

Página 20

serializado de volta para o servidor, que então executa o código .NET, atualizando a renderização árvore, que então é serializada de volta para o navegador. Você pode ver esse processo em Figura 5 . A grande diferença é que não há necessidade de enviar o runtime Mono e seu Blazor assemblies para o navegador. E o modelo de programação permanece o mesmo!

Figura 5. *Blazor do lado do servidor*

Prós e contras do modelo do lado do servidor

O modelo do lado do servidor tem alguns benefícios, mas também algumas desvantagens. Vamos discutir eles aqui para que você possa decidir qual modelo se adapta às necessidades do seu aplicativo.

Downloads menores

Com o Blazor do lado do servidor, seu aplicativo não precisa baixar mono.wasm nem todos seus assemblies .NET. Isso significa que o aplicativo será iniciado muito mais rápido.

Página 21

Processo de desenvolvimento

O lado do cliente do Blazor tem recursos de depuração limitados, resultando em registro adicional. Porque

seu código .NET está sendo executado no servidor, você pode usar o depurador .NET normal. Vocês poderia começar a construir seu aplicativo Blazor usando o modelo do lado do servidor e quando for Concluiu a mudança para o modelo do lado do cliente fazendo uma pequena alteração em seu código.

APIs .NET

Como você está executando seu código .NET no servidor, pode usar todas as APIs .NET que você usaria com aplicativos MVC regulares, por exemplo, acessar o banco de dados diretamente. Observe que fazer isso impedirá que você seja capaz de convertê-lo rapidamente para um cliente aplicativo.

Online somente

Executar o aplicativo Blazor no servidor significa que seus usuários sempre precisa de acesso ao servidor. Isso impedirá que o aplicativo seja executado no Electron; vocês também não pode executá-lo como um aplicativo da web progressivo (PWA). E se a conexão cair entre o navegador e o servidor, o usuário pode perder algum trabalho porque o aplicativo irá parar de funcionar.

Escalabilidade do servidor

Todo o seu código .NET é executado no servidor, portanto, se você tiver milhares de clientes, seu (s) servidor (es) terá que lidar com todo o trabalho. Além disso, o Blazor usa um modelo state-full, o que significa que você deve acompanhar o estado de cada usuário no servidor.

Resumo

Nesta introdução, você olhou para a história da guerra dos navegadores e como eles resultaram na criação do WebAssembly. Mono permite que você execute assemblies .NET; Porque Mono pode ser executado no WebAssembly, agora você pode executar assemblies .NET no navegador! Tudo de isso resultou na criação do Blazor, onde você pode construir arquivos Razor contendo .NET código, que atualiza o DOM do navegador, oferecendo a capacidade de construir uma única página aplicativos em .NET.

CAPÍTULO 1

Seu primeiro projeto Blazor

Ter uma experiência prática é a melhor maneira de aprender. Neste capítulo, você instalará o pré-requisitos para desenvolver com o Blazor, que inclui o Visual Studio junto com alguns extensões necessárias. Em seguida, você criará seu primeiro projeto Blazor no Visual Studio, execute o projeto para vê-lo funcionar e inspecionar os diferentes aspectos do projeto para obter uma "configuração do land ”para ver como os aplicativos Blazor são desenvolvidos.

Pré-requisitos de instalação do Blazor

Trabalhar com o Blazor requer que você instale alguns pré-requisitos, então vamos começar.

.NET Core

O Blazor é executado em cima do .NET Core, fornecendo o servidor web para o seu projeto, que irá servir os arquivos do cliente que são executados no navegador e executar quaisquer APIs do lado do servidor que seu Blazor

necessidades do projeto. .NET Core é a solução de plataforma cruzada da Microsoft para trabalhar com .NET em Windows, Linux e OSX.

Você pode encontrar os arquivos de instalação em www.microsoft.com/net/download. Olhe para a versão mais recente do .NET Core SDK. Baixe o instalador, execute-o e aceite o padrões.

Verifique a instalação quando o instalador terminar abrindo um novo prompt de comando e digitando o seguinte comando:

```
dotnet –version
```

Procure a seguinte saída para indicar que você tem a versão correta instalada. O número da versão deve ser pelo menos 2.1.300.

Se a saída do comando mostrar uma versão mais antiga (por exemplo 2.1.200), você deve baixe e instale uma versão mais recente do .NET Core SDK.

1

P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_1

Página 23

Visual Studio 2017

O Visual Studio 2017 (a partir de agora me referirei ao Visual Studio como VS) é um dos ambientes de desenvolvimento integrado (IDEs) que você usará ao longo deste livro. O outro IDE é o Visual Studio Code. Com qualquer um, você pode editar seu código, compilá-lo e execute tudo a partir do mesmo aplicativo. Os exemplos de código também são os mesmos. No entanto, VS só funciona no Windows, portanto, se você estiver usando outro sistema operacional, continue na seção sobre Código do Visual Studio.

Baixe a versão mais recente do Visual Studio 2017 em www.visualstudio.com/Transferências/.

Execute o instalador e certifique-se de instalar o ASP.NET e o desenvolvimento web função, conforme mostrado na Figura 1-1 .

Figura 1-1. *A seleção de cargas de trabalho do instalador do Visual Studio*

Após a instalação, execute o Visual Studio a partir do menu Iniciar. Em seguida, abra o menu Ajuda e selecione Sobre o Microsoft Visual Studio. A caixa de diálogo Sobre o Microsoft Visual Studio janela deve especificar pelo menos a versão 15.7.3, conforme ilustrado na Figura 1-2.

2

Página 24

Figura 1-2. *Sobre o Microsoft Visual Studio*

Serviços de linguagem ASP.NET Core Blazor

O plugin Blazor Language Services para Visual Studio irá ajudá-lo a digitar Blazor arquivos e instalará os modelos de projeto do Blazor VS. A instalação do plugin está concluída diretamente do Visual Studio. Ferramentas abertas ➤ Extensões e atualizações. Clique na guia Online e digite Blazor na caixa de pesquisa. Você deve ver a linguagem ASP.NET Core Blazor Serviços listados conforme mostrado na Figura 1-3. Selecione-o e clique no botão Download para instalar.

Figura 1-3. *Instalando o Blazor Language Services a partir das extensões e atualizações cardápio*

Código Visual Studio

O Visual Studio Code é um ambiente de desenvolvimento gratuito, moderno e de plataforma cruzada com editor integrado, controle de origem git e depurador. O ambiente tem uma gama enorme de extensões disponíveis, permitindo que você use todos os tipos de linguagens e ferramentas diretamente do Código. Portanto, se você não tiver acesso ao Visual Studio 2017 (porque você está executando um Sistema operacional Windows ou se você não quiser usá-lo), use o Code.

Página 25

Baixe o instalador de www.visualstudio.com/ . Execute-o e escolha o padrões.

Após a instalação, aconselho você a instalar algumas extensões para o Code, especialmente as extensões C #. Código de início e, no lado esquerdo, selecione a guia Extensões, conforme mostrado na figura 1-4 .

Figura 1-4. *Guia Extensões de código do Visual Studio*

Você pode pesquisar extensões, então comece com C #, que é a primeira extensão de Figura 1-4. Esta extensão fornecerá IntelliSense para a linguagem de programação C # e assemblies .NET. Você provavelmente obterá uma versão mais recente listada, então pegue a mais recente.

Clique em Instalar.

Outra extensão que você deseja pesquisar é Razor +, conforme mostrado na Figura 1-5 . Esta a extensão lhe dará uma boa coloração de sintaxe para o tipo de arquivos do Razor que você usará em Blazor.

Figura 1-5. *Razor + para Visual Studio Code*

Instalando os modelos do Blazor para VS / Code

Ao longo deste livro, você criará vários projetos Blazor diferentes. Nem todos eles pode ser criado a partir do Visual Studio ou Code, o que significa que você precisará instalar os modelos para projetos Blazor. O exemplo desta seção mostra como instalar esses modelos a partir do

Página 26

Interface de linha de comando do .NET Core, também conhecida como CLI do .NET Core. Você devia ter essa interface de linha de comando como parte da instalação do .NET Core.

Abra uma linha de comando em seu sistema operacional e digite o seguinte para instalar os modelos do NuGet:

```
dotnet new -i Microsoft.AspNetCore.Blazor.Templates
```

Esses modelos permitirão que você gere projetos e itens rapidamente. Verifique o instalação digitando o seguinte comando:

```
dotnet new –-help
```

Este comando irá listar todos os modelos que foram instalados pelo comando-interface de linha. Você verá quatro colunas. O primeiro mostra a descrição do modelo, o a segunda coluna exibe o nome, a terceira lista os idiomas para os quais o modelo está disponível, e o último mostra as tags, uma espécie de nome de grupo para o modelo. Entre os listados são os seguintes:

Blazor (hospedado em servidor ASP.NET)	blazorhosted
Biblioteca Blazor	blazorlib
Blazor (lado do servidor no ASP.NET Core)	blazorserverside
Blazor (autônomo)	blazor

Gerando Seu Projeto com Visual Studio

Com os projetos do Blazor, você tem algumas opções. Você pode criar um Blazor independente (usando o modelo blazor) que não precisa de código do lado do servidor. Este tipo de projeto tem a vantagem de que você pode simplesmente implantá-lo em qualquer servidor web, que irá funcionar como um servidor de arquivos, permitindo que os navegadores façam download do seu site como qualquer outro site. Ou você pode criar um projeto hospedado (usando o modelo blazorhosted) com cliente, servidor, e código compartilhado. Este tipo de projeto exigirá que você hospede-o onde houver um núcleo .NET 2.1 suporte porque você executará código no servidor também. A terceira opção é execute todo o código Blazor no servidor (usando o modelo blazorserverside). Nesse caso, o navegador usará uma conexão SignalR para receber atualizações de interface do usuário do servidor e para enviar a interação do usuário de volta ao servidor para processamento. Neste livro, você usará a segunda opção, mas os conceitos que você aprenderá neste livro são os mesmos para todos três opções.

Página 27

Criando um projeto com o Visual Studio

Para seu primeiro projeto, inicie o Visual Studio e selecione File ➤ New ➤ Project. À esquerda lado da caixa de diálogo Novo projeto, selecione C # ➤ Web e, em seguida, selecione ASP.NET Core Web Aplicação, conforme ilustrado na Figura 1-6 .

Figura 1-6. *Caixa de diálogo Novo Projeto do Visual Studio*

Nomeie seu projeto como *MyFirstBlazor* , deixe o resto com os padrões predefinidos e clique em OK. Na próxima tela, você pode selecionar que tipo de projeto ASP.NET Core deseja gerar. Nos menus suspensos superiores, selecione .NET Core e ASP.NET Core 2.1 (ou superior), como mostrado na figura 1-7. Em seguida, selecione Blazor (hospedado em ASP.NET) e clique em OK.

Página 28

Figura 1-7. *Novo aplicativo da web ASP.NET Core*

Aguarde a conclusão do Visual Studio. Em seguida, construa sua solução.

no momento em que este livro foi escrito, o Blazor tinha depuração do lado do cliente muito limitada (e apenas no Chrome), portanto, executar um projeto Blazor com um depurador só vai demorar mais hora de mostrar o navegador. De agora em diante, direi para você executar sem o depurador.

Gerando o projeto com dotnet cli

Para gerar o projeto com dotnet cli, abra a linha de comando e altere o atual para onde você deseja criar o projeto. Agora execute este comando para crie um novo projeto a partir do modelo blazorhosted no diretório MyFirstBlazor:

```
dotnet new blazorhosted -o MyFirstBlazor
```


Página 29

Este comando vai demorar um pouco porque vai baixar um monte de NuGet pacotes da Internet. Quando o comando estiver pronto, você pode construir seu projeto usando

```
cd MyFirstBlazor
compilação dotnet
```

Agora abra a pasta do seu projeto com Code. Quando o Code carregar tudo, ele irá faça uma pequena pergunta, como mostrado na Figura 1-8.

Figura 1-8. *Código pedindo para adicionar recursos de compilação e depuração*

Responda sim. Isso adicionará uma pasta chamada .vscode com a adição de arquivos de configuração suporte para construir e executar o projeto do Code.

Executando o Projeto

Pressione Ctrl-F5 para executar (isso deve funcionar para Visual Studio e Code). Seu (padrão) o navegador deve abrir e exibir a página inicial, conforme mostrado na Figura 1-9 .

Figura 1-9. *A página inicial do seu primeiro aplicativo*

Página 30

Este aplicativo gerado de página única tem no lado esquerdo um menu de navegação permitindo que você pule entre páginas diferentes. No lado direito você verá o selecionado tela mostrada na Figura 1-9 : a página inicial. E no canto superior direito há um Sobre link para https://blazor.net/ , que é o site de documentação "oficial" do Blazor.

A página inicial

A página inicial mostra o obrigatório "Hello, world!" demo, e também contém uma pesquisa componente em que você pode clicar para preencher uma pesquisa (informe à Microsoft que você gosta do Blazor!).

A página contrária

No menu de navegação, clique na guia Contador. Isso abre uma tela simples com um número e um botão, conforme ilustrado na Figura 1-10 . Clicar no botão aumentará o contador. Tente!

Figura 1-10. *A tela do contador*

A página de busca de dados

No menu de navegação, clique na guia Buscar dados. Aqui você pode assistir a um (aleatório e falsa) previsão do tempo, conforme mostrado na Figura 1-11. Esta previsão é gerada no servidor quando solicitado pelo cliente. Isso é muito importante porque o cliente (que está sendo executado em o navegador) não pode acessar dados de um banco de dados diretamente, então você precisa de um servidor que possa acessar bancos de dados e outro armazenamento de dados.

Página 31

Figura 1-11. *A tela de busca de dados*

Examinando as partes do projeto

Poder brincar com essas páginas é bom, mas vamos dar uma olhada em como tudo isso funciona. Vocês começará com o projeto do servidor, que hospeda o seu site Blazor. Então você vai olhar para o projeto compartilhado, que contém classes usadas pelo servidor e pelo cliente. Enfim tu examinará o projeto do cliente, que é a implementação real do Blazor.

A solução

O Visual Studio e o Code usam arquivos de solução para agrupar projetos que formarão um aplicativo. Portanto, um projeto Blazor típico consiste em um servidor, um cliente e um projeto compartilhado agrupado em uma única solução. Isso simplifica a construção de tudo, pois a solução permite ferramentas para descobrir em que ordem compilar tudo. Ei, você pode até alternar entre Visual Studio e Code porque ambos usam os mesmos arquivos de projeto e solução!

O servidor

Os aplicativos da Web são, na verdade, um monte de arquivos que são baixados pelo navegador de um servidor. É função do servidor fornecer os arquivos ao navegador mediante solicitação. Existe um toda a gama de servidores existentes para escolher, por exemplo IIS no Windows ou Apache

Página 32

no Linux. ASP.NET Core tem seu próprio servidor embutido que você gerou com o modelo blazorhosted, que pode ser executado no Windows, Linux ou OSX.

O tópico deste livro é Blazor, então não vou discutir todos os detalhes do servidor projeto que foi gerado usando o modelo blazorhosted, mas eu quero mostrar a vocês uma coisa importante. No projeto do servidor, procure Startup.cs. Abra este arquivo e role até o método Configure mostrado na Listagem 1-1.

Listagem 1-1. O Método de Configuração do Projeto de Servidor

```
// Este método é chamado pelo tempo de execução.
// Use este método para configurar o pipeline de solicitação HTTP.
public void Configure (aplicativo IApplicationBuilder,
                       IHostingEnvironment env)
{
    app.UseResponseCompression ();

    if (env.IsDevelopment ())
    {
        app.UseDeveloperExceptionPage ();
    }

    app.UseMvc (routes =>
    {
        routes.MapRoute (name: "default",
                         modelo: "{controlador} / {ação} / {id?}");
    });

    app.UseBlazor <Client.Program> ();
}
```

O método Configure é responsável por instalar o middleware. Middleware são pequenos componentes .NET, cada um com uma responsabilidade clara. Quando você digita um URI, o navegador envia uma solicitação ao servidor, que então a repassa ao middleware

componentes. Alguns deles aceitarão a solicitação e retornarão uma resposta; alguns deles pegue a resposta e faça algo com ela. Olhe para a primeira linha no Configure método, mostrado na Listagem 1-2 .

Página 33

Listagem 1-2. O Middleware UseResponseCompression

```
app.UseResponseCompression ();
```

Seu cliente Blazor baixará muitos arquivos do servidor, incluindo .NET assemblies, portanto, compactar esses arquivos resultará em um download mais rápido. O O middleware UseResponseCompression cuida disso.

Você gostaria de ver uma página de erro detalhada quando o servidor tiver um erro não capturado exceção? O UseDeveloperExceptionPage cuida disso. Claro que voce nao precisa dele na produção (você deve lidar com todas as exceções corretamente) para que este middleware é usado apenas quando executado em um ambiente de desenvolvimento. Como o servidor sabe se você está executando em desenvolvimento ou lançamento? A instrução if que você vê aqui verifica um variável de ambiente chamada ASPNETCORE_ENVIRONMENT, e se a variável de ambiente é definido como Desenvolvimento, ele sabe que você está executando no modo de desenvolvimento.

A tela Buscar dados baixa informações meteorológicas do servidor. Esses tipos de solicitações serão tratadas pelo middleware MVC. Vou discutir isso com mais detalhes em Capítulo 5.

O processo de bootstrap do Blazor requer vários arquivos especiais, especialmente mono.wasm. Eles são servidos pelo middleware Blazor, que pode ser encontrado no final do Configure o método.

O Projeto Compartilhado

Quando você clica na guia Buscar dados, seu projeto Blazor busca alguns dados do servidor. A forma desses dados precisa ser descrita em detalhes (os computadores são exigentes coisas); em projetos clássicos, você descreve a forma deste modelo duas vezes, uma para o cliente e novamente para o servidor porque eles usam idiomas diferentes. Não com Blazor! Em Blazor, ambos cliente e servidor usam C #, então você pode descrever o modelo uma vez e compartilhá-lo entre o cliente e servidor, conforme mostrado na listagem 1-3.

Listagem 1-3. A classe WeatherForecast compartilhada

```
public class WeatherForecast
{
    public DateTime Date {get; definir; }
    public int TemperatureC {get; definir; }
    public string Resumo {get; definir; }
```


Página 34


```
    public int TemperatureF
                => 32 + (int) (TemperaturaC / 0,5556);
}
```

O Projeto Blazor do Cliente

Abra a pasta wwwroot do projeto do cliente e procure index.html. O conteúdo disso arquivo deve aparecer como mostrado na Listagem 1-4. Para ser honesto, isso parece mais um normal Página HTML. Mas, em uma inspeção mais próxima, você verá que há uma tag <app> html estranha lá:

```
<app> Carregando ... </app>
```

O elemento <app> html não existe! É um exemplo de componente Blazor. Vocês também verá um elemento <script>:

```
<script src = "_ framework / blazor.webassembly.js"> </script>
```

Este script irá instalar o Blazor baixando mono.wasm e seus assemblies.

Listagem 1-4. index.html

```
<! DOCTYPE html>
<html>
<head>
    <meta charset = "utf-8" />
    <meta name = "viewport" content = "width = device-width">
    <title> MyFirstBlazor </title>
    <base href = "/" />
    <link href = "css / bootstrap / bootstrap.min.css"
          rel = "stylesheet" />
    <link href = "css / site.css" rel = "stylesheet" />
</head>
<body>
    <app> Carregando ... </app>

    <script src = "_ framework / blazor.webassembly.js"> </script>
</body>
</html>
```


Página 35

Encaminhamento

O que é esse elemento <app>? Abra Startup.cs a partir do projeto MyFirstBlazor.Client e procure o método Configure, conforme mostrado na Listagem 1-5 . Aqui você pode ver o App componente sendo associado à tag de aplicativo de index.html. Um componente Blazor usa uma tag personalizada como <app>, e o tempo de execução do Blazor substitui a tag pelo marcação do componente, que é um HTML normal reconhecido pelo navegador. Eu discutirei Componentes do Blazor no Capítulo 3 .

Listagem 1-5. O método de configuração associando o elemento app ao app Componente

```
public void Configure (aplicativo IBlazorApplicationBuilder)
{
    app.AddComponent <App> ("app");
}
```

A principal coisa que o componente App faz é instalar o roteador, conforme mostrado na Listagem 1-6. O roteador é responsável por carregar um componente Blazor dependendo do URI no navegador. Por exemplo, se você navegar para / URI, o roteador irá procurar por um componente com uma diretiva @page correspondente.

Listagem 1-6. O componente do aplicativo

```
<Router AppAssembly = typeof (Program) .Assembly />
```

Em seu projeto MyFirstBlazor atual, isso corresponderá ao componente Índice, que você pode encontrar no arquivo Index.cshtml, que você pode encontrar na pasta Pages. O índice

O componente exibe uma mensagem Hello World e o link de pesquisa, conforme mostrado na Listagem 1-7 .

Listagem 1-7. O Componente de Índice

```
@página "/"

<h1> Olá, mundo! </h1>

Bem-vindo ao seu novo aplicativo.

<SurveyPrompt Title = "Como o Blazor está funcionando para você?" />
```


Página 36

Componentes de Layout

Veja a figura 1-9 e Figura 1-10. Ambos têm o mesmo menu. Este menu é compartilhado entre todos os seus componentes do Blazor e é conhecido como um componente de layout. Eu discutirei componentes de layout no Capítulo 7 . Mas como o Blazor sabe qual componente é o componente de layout? Abra a pasta Pages do projeto MyFirstBlazor.Client e procure o arquivo _ViewImports.cshtml. No Razor você usa um arquivo _ViewImports.cshtml para definir a marcação comum entre todos os arquivos do Razor na mesma pasta. Se você está familiarizado com o .NET Core, _ViewImports.cshtml do .NET Core é muito semelhante. A pasta Pages contém esse arquivo e especifica que todos os arquivos usam o mesmo componente MainLayout, como mostrado na Listagem 1-8 .

Listagem 1-8. Especificando o componente de layout em _ViewImports.cshtml

```
@layout MainLayout
```

Em seu projeto, o componente de layout pode ser encontrado em MainLayout.cshtml do Pasta compartilhada, que é mostrada na Listagem 1-9 .

Listagem 1-9. O componente MainLayout

```
@inherits BlazorLayoutComponent

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

<div class = "main">
    <div class = "linha superior px-4">
        <a href = "http://blazor.net" target = "_ blank"
            class = "ml-md-auto"> Sobre </a>
    </div>

    <div class = "content px-4">
        @Corpo
    </div>
</div>
```


Página 37

O primeiro div com a barra lateral de classe contém um único componente: NavMenu. Aqui é onde

seu menu de navegação é definido. Você verá com mais detalhes a navegação e o roteamento no Capítulo 7 .

O próximo div com a classe main tem duas partes. O primeiro é o link Sobre que você vê no cada página. A segunda parte contém o @Body; este é o lugar onde a página selecionada estará mostrando. Por exemplo, quando você clica no link Contador no menu de navegação, este é onde o componente Counter.cshtml Blazor irá.

O processo de bootstrap do Blazor

Examine a Listagem 1-4 novamente. Na parte inferior você encontrará o elemento <script> responsável para inicializar o Blazor no navegador. Vejamos esse processo.

Volte para o seu navegador e abra as ferramentas de desenvolvedor. (A maioria dos navegadores abrirá o ferramentas de desenvolvedor quando você pressiona F12.) Vejamos o que acontece na camada de rede.

> todas as capturas de tela neste livro usam o navegador Chrome, principalmente porque ele está disponível em todas as plataformas (Windows, Linux e osX) e porque é muito popular com um muitos desenvolvedores da web. se você gosta mais de outro navegador, vá em frente!

Atualize seu navegador para ver o que é baixado do servidor, conforme mostrado em Figura 1-12. Se a Figura 1-12 não corresponder ao que você vê, limpe o cache do navegador. Os navegadores usam um cache para evitar recarregar arquivos do servidor, mas quando você está desenvolvimento, você deve limpar o cache para garantir que está obtendo as alterações mais recentes o servidor. Primeiro, você verá index.html sendo baixado, que por sua vez baixa bootstrap.css e site.css e, em seguida, blazor.webassembly.js. Um pouco mais baixo você vai veja que mono.js é baixado, que por sua vez baixará mono.wasm. Isto é o tempo de execução mono compilado para rodar em WebAssembly!

Página 38

Figura 1-12. *Examinando o processo de bootstrap usando o log de rede*

Agora que o runtime .NET está em execução, você verá que MyFirstBlazor.Client.dll é baixado, seguido por todas as suas dependências, incluindo mscorlib.dll e sistema. dll. Esses arquivos contêm as bibliotecas .NET contendo classes como string, usadas para executam todos os tipos de coisas e são as mesmas bibliotecas que você usa no servidor. Isso é muito poderoso porque você pode reutilizar bibliotecas .NET existentes no Blazor que você ou outros construído antes!

Resumo

Neste capítulo, você instalou os pré-requisitos necessários para desenvolver e executar Aplicativos Blazor. Em seguida, você criou seu primeiro projeto Blazor. Este projeto será usado ao longo deste livro para explicar todos os conceitos do Blazor que você precisa conhecer. Finalmente, você examinou esta solução, olhando para o projeto do lado do servidor, o projeto compartilhado e o projeto Blazor do lado do cliente.

CAPÍTULO 2

Ligação de dados

Imagine qualquer aplicativo que precise exibir dados para o usuário e capturar as alterações feito por aquele usuário para salvar os dados modificados. Uma maneira de construir um aplicativo como isso serve para, assim que você tiver os dados, iterar sobre cada item de dados. Por exemplo, para cada membro de uma lista, você geraria o mesmo elemento de repetição e, em seguida, dentro desse elemento, você geraria caixas de texto, menus suspensos e outros elementos da IU que apresentam dados. Mais tarde, depois que o usuário fez algumas alterações, você iria iterar sobre o elementos, e para cada um você inspecionaria os elementos filho se seus dados fossem mudado. Nesse caso, você deve copiar os dados de volta para os objetos usados para salvá-los.

Este é um processo sujeito a erros e muito trabalhoso se você quiser fazer isso com algo como jQuery (jQuery é uma estrutura JavaScript muito popular que permite manipular o Document Object Model (DOM) do navegador.

Estruturas modernas como Angular e React se tornaram populares porque simplifique muito esse processo por meio da *vinculação de dados* . Com vinculação de dados, a maior parte deste trabalho para gerar a IU e copiar dados de volta em objetos é feito pela estrutura.

Uma rápida olhada no Razor

Blazor é a combinação de *Browser* + *Razor* (com muita liberdade artística). Então, para entender o Blazor, você precisa entender os navegadores e o Razor. Eu vou assumir você entender o que é um navegador, já que a Internet é muito popular há mais de uma década. Mas o Razor (como linguagem de computador) pode não ser tão claro (ainda). Navalha é uma sintaxe de marcação que permite incorporar código em uma página da web. No ASP.NET Core MVC o código é executado no lado do servidor para gerar HTML que é enviado ao navegador. Mas no Blazor este código é executado dentro do seu navegador e irá atualizar dinamicamente a web página sem ter que voltar ao servidor.

Lembre-se da solução MyFirstBlazor que você gerou a partir do modelo em o capítulo anterior? Abra-o novamente com Visual Studio ou Code e dê uma olhada em SurveyPrompt.cshtml, conforme mostrado na Listagem 2-1 .

P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_2

Listagem 2-1. Examining SurveyPrompt.cshtml

```
<div class = "alert alert-secondary mt-4" role = "alert">
    <span class = "oi oi-pencil mr-2" aria-hidden = "true"> </span>
    <strong> @Title </strong>
    <span class = "text-nowrap">
        Por favor, pegue nosso
        <a target = "_ blank" class = "font-weight-bold"
            href = "https://go.microsoft.com/fwlink/?linkid=873042">
            breve pesquisa
        </a>
    </span>
    e diga-nos o que você pensa.
</div>
@funções {
[Parâmetro]
string Título {get; definir; } // Demonstra como um pai
                                componente pode fornecer parâmetros
}
```

Como você pode ver, o Razor consiste principalmente em marcação HTML. Mas se você quiser ter algum Propriedades ou métodos C #, você pode incorporá-los na seção @functions de um arquivo Razor. Isso funciona porque o arquivo Razor é usado para gerar uma classe .NET e tudo em @functions está embutido nessa classe. Por exemplo, o componente SurveyPrompt permite você deve definir a propriedade Title, que é definida em Index.cshtml, conforme mostrado na Listagem 2-2.

Listagem 2-2. Definição do título do SurveyPrompt (trecho de index.cshtml)

```
<SurveyPrompt Title = "Como o Blazor está funcionando para você?" />
```

Como a propriedade Title pode ser definida em outro componente, a propriedade torna-se um parâmetro, e por isso você precisa aplicar o atributo [Parameter], conforme mostrado na Listagem 2-1 . SurveyPrompt pode, então, incorporar o conteúdo da propriedade Title em seu Marcação HTML usando a sintaxe @. Esta sintaxe diz ao Razor para mudar para C #, e isso irá obter a propriedade e incorpore seu valor na marcação.

Página 41

Vinculação de dados unilateral

A vinculação de dados unilateral é onde os dados fluem do componente para o DOM, ou vice-versa vice-versa, mas apenas em uma direção. A vinculação de dados do componente ao DOM é onde alguns dados, como o nome do cliente, precisam ser exibidos. Vinculação de dados do DOM para o componente é onde algum evento DOM ocorreu, como o usuário clicar em um botão, e você deseja que algum código seja executado.

Sintaxe de vinculação de dados unilateral

Vejamos um exemplo de vinculação de dados unilateral no Razor. Abra a solução que você criou Capítulo 1 (MyFirstBlazor.sln) e abra Counter.cshtml, repetido aqui na Listagem 2-3 .

Listagem 2-3. Examinando a ligação de dados unilateral com Counter.cshtml

```
@page "/ counter"

<h1> Contador </h1>
```

```
<p> Contagem atual: @currentCount </p>

<button class = "btn btn-primary" onclick = "@ IncrementCount">
    Clique em mim
</button>
@funções {
    int currentCount = 0;
    void IncrementCount ()
    {
        currentCount ++;
    }
}
```

Nesta página, você obtém um contador simples, que pode incrementar clicando no botão, conforme ilustrado pela Figura 2-1.

Página 42

Figura 2-1. *A página do contador*

Vejamos o funcionamento desta página. O campo do contador é definido nas funções @ seção em Counter.cshtml. Este não é um campo que pode ser definido de fora, então não há necessidade de [ParameterAttribute].

Para exibir o valor do contador no Razor, você usa a sintaxe @currentCount Razor mostrado na Listagem 2-4 .

Listagem 2-4. Vinculação de dados do componente ao DOM

```
<p> Contagem atual: @currentCount </p>
```

Sempre que Blazor vê que currentCount pode ter sido atualizado, ele irá automaticamente atualize o DOM com o valor mais recente de currentCount.

Atributos condicionais

Às vezes, você pode controlar o navegador adicionando alguns atributos aos elementos DOM. Por exemplo, para desabilitar um botão, você pode simplesmente usar o atributo disabled. Olhe para a Listagem 2-5.

Listagem 2-5. Desativando um botão usando o atributo desativado

```
<button disabled> On Strike </button>
```

Com o Blazor vocêpoderá associar dados de um atributo a uma expressão Booleana (por exemplo, uma propriedade ou método do tipo bool) e o Blazor ocultará o atributo se a expressão for avaliada como falso (ou nulo) e mostrará o atributo se for avaliado como verdadeiro. Volte para o balcão. cshtml e adicione o código da Listagem 2-6 .

Página 43

Listagem 2-6. Desativando o botão Click Me

```
<button onclick = "@ IncrementCount"
        disabled = "@ (currentCount> = 10)"> Clique aqui </button>
```

Tente. Clicar no botão até que currentCount se torne 10 desabilitará o botão. Assim que currentCount cair abaixo de 10, o botão será habilitado novamente.

Tratamento de eventos e vinculação de dados

Você atualiza currentCount usando o método IncrementCount () da Listagem 2-3. Esta método é chamado clicando no botão Click Me. Isso, novamente, é um dado unilateral ligação, mas na outra direção, do botão para o seu componente.

Sintaxe de associação de eventos

Olhe para a lista 2-7 . Agora você está usando a sintaxe on <event>; neste caso, você quer vincular ao evento DOM de clique do botão, então você usa o atributo onclick no botão elemento e você passa o método que deseja chamar.

Listagem 2-7. Vinculação de dados do DOM ao componente

```
<button class = "btn btn-primary" onclick = "@ IncrementCount">
    Clique em mim
</button>
```

Clicar no botão fará com que a IU seja atualizada com o novo valor do contador. Sempre que o usuário interage com o site, por exemplo, clicando em um botão, o Blazor assume que o evento terá algum efeito colateral porque um método é chamado, então atualizará a IU com os valores mais recentes. Simplesmente chamar um método não fará com que o Blazor atualizar a IU. Discutirei isso mais tarde neste capítulo.

Argumentos do Evento

No .NET regular, os manipuladores de eventos do tipo EventHandler podem encontrar mais informações sobre o evento usando os argumentos sender e EventArgs. No Blazor, manipuladores de eventos não siga o padrão de evento estrito do .NET, mas você pode declarar o manipulador de eventos método para obter um argumento de algum tipo derivado de EventArgs, por exemplo UIMouseEventArgs, conforme mostrado na Listagem 2-8 .

Página 44

Listagem 2-8. Um manipulador de eventos Blazor considerando argumentos

```
void IncrementCount (UIMouseEventArgs e)
```

Usando funções C # Lambda

A vinculação de dados a um evento nem sempre exige que você escreva um método. Você também pode usar Sintaxe da função lambda C #; veja o exemplo mostrado na Listagem 2-9 .

Listagem 2-9. Ligação de dados de eventos com sintaxe lambda

```
<button class = "btn btn-primary"
        onclick = "@ (() => currentCount + = incremento)">
    Clique em mim
```

</button>

Se você quiser usar uma função lambda, será necessário colocá-la entre colchetes.

Vinculação de dados bidirecional

Às vezes, você deseja exibir alguns dados para o usuário e permite que ele faça alterações nesses dados. Isso é comum em formulários de entrada de dados. Vamos explorar os dois forma de sintaxe de ligação de dados.

Sintaxe de vinculação de dados bidirecional

Com a vinculação de dados bidirecional, você tem a atualização do DOM sempre que o componente muda, mas o componente também será atualizado devido às modificações no DOM. O o exemplo mais simples é com um elemento HTML <input>.

Vamos tentar algo. Modifique Counter.cshtml adicionando um campo de incremento e um entrada usando o atributo bind, conforme mostrado na Listagem 2-10 .

Listagem 2-10. Adicionando um incremento e uma entrada

```
@page "/ counter"

<h1> Contador </h1>

<p> Contagem atual: @currentCount </p>
```


Página 45


```
<button class = "btn btn-primary"
        onclick = "@ IncrementCount"> Clique aqui </button>

<input type = "number" bind = "@ increment" />

@funções {
int currentCount = 0;

incremento interno = 1;

void IncrementCount ()
{
   currentCount + = incremento;
}
}
```

Construa e execute.

Agora você deve ser capaz de incrementar o contador com outros valores, conforme mostrado em Figura 2-2.

Figura 2-2. *Adicionar um incremento com vinculação de dados bidirecional*

Observe o elemento de entrada que você acabou de adicionar, repetido aqui na Listagem 2-11 .

Listagem 2-11. Ligação de dados bidirecional com a sintaxe de ligação

```
<input type = "number" bind = "@ increment" />
```

Aqui você está usando a sintaxe de ligação, que é o equivalente a dois diferentes ligações, como mostrado na listagem 2-12 .

Página 46

Listagem 2-12. Vinculação de dados em ambas as direções

```
<input type = "número"
        valor = "@ incremento"
        onchange = "@ ((UIChangeEventArgs e) =>
                        incremento = int.Parse ($ "{e.Value}")) "/>
```

Essa sintaxe alternativa é muito detalhada e não é muito prática de usar. Usar bind é muito mais prático. No entanto, não se esqueça dessa técnica; usando o mais prolixo a sintaxe às vezes pode ser uma solução mais elegante!

Formatar datas

A vinculação de dados a um valor DateTime pode ser formatada com o atributo format-value, como mostrado na Listagem 2-13.

Listagem 2-13. Formatando uma Data

```
<input type = "text" bind = "@ someDate"
                        format-value = "dd-MM-aaaa" />

@funções {

    private DateTime someDate = DateTime.Now;

}
```

Nesse caso, a data usará o formato de data europeu. Valores de DateTime atuais são os únicos que suportam o atributo format-value.

Mudanças de relatório

O Blazor atualizará o DOM sempre que achar que foram feitas alterações em seus dados. Um exemplo é quando um evento executa parte do seu código, ele assume que você modificou alguns valores como um efeito colateral e renderiza a IU. No entanto, Blazor nem sempre é capaz de detectar todas as mudanças, e neste caso, você terá que dizer ao Blazor para aplicar as mudanças para o DOM. Um exemplo típico é com threads de fundo. Vejamos um exemplo.

Página 47

Abra Counter.cshtml e adicione outro botão que irá incrementar automaticamente o contador quando pressionado, como mostrado na Listagem 2-14. O método AutoIncrement usa um .NET Instância de cronômetro para incrementar currentCount a cada segundo.

Listagem 2-14. Adicionando outro botão

```
@page "/ counter"
@using Microsoft.AspNetCore.Blazor.Components

<h1> Contador </h1>
```

```
<p> Contagem atual: @currentCount </p>

<button class = "btn btn-primary" onclick = "@ IncrementCount">
    Clique em mim
</button>

<input type = "number" bind = "@ increment" />

<button class = "btn btn-success" onclick = "@ AutoIncrement">
    Incremento automático
</button>

@funções {
    int currentCount = 0;

    incremento interno = 1;

    void IncrementCount ()
    {
        currentCount + = incremento;
    }

    void AutoIncrement ()
    {
        var timer = new System.Threading.Timer ((_) =>
        {
            IncrementCount ();
        }, nulo, TimeSpan.FromSeconds (1),
                    TimeSpan.FromSeconds (1));
    }
}
```

27

Página 48

Você pode encontrar o argumento da função lambda no construtor do Timer um pouco estranho. Eu uso um sublinhado quando preciso nomear um argumento que não é usado no corpo da função lambda. Chame do que quiser, por exemplo, ignore; isso não matéria. Eu simplesmente gosto de usar o sublinhado porque então não tenho que pensar em um bom nome do argumento.

Execute esta página. Clicar no botão de incremento automático iniciará o cronômetro, mas o contador não será atualizado na tela. Por quê? Experimente clicar no botão Incrementar. O contador foi atualizado, portanto, é um problema de interface do usuário.

O Blazor irá renderizar a página sempre que ocorrer um evento. Também irá renderizar novamente a página no caso de operações assíncronas. No entanto, algumas mudanças não podem ser detectado automaticamente. Neste caso, você precisa dizer ao Blazor para atualizar a página chamando o método StateHasChanged, do qual cada componente do Blazor herda sua classe base.

Volte para o método AutoIncrement e adicione uma chamada para StateHasChanged, como em Listagem 2-15. StateHasChanged diz ao Blazor que algum estado mudou (quem iria pensei!) e que ele precisa renderizar a página novamente.

Listagem 2-15. Adicionando StateHasChanged

```
void AutoIncrement ()
{
    var timer = new System.Threading.Timer ((_) =>
    {
        IncrementCount ();
        StateHasChanged ();
    }, nulo, TimeSpan.FromSeconds (1), TimeSpan.FromSeconds (1));
}
```

Corra novamente. Pressionar o botão de incremento automático agora funcionará.

Como você pode ver, às vezes você precisa dizer ao Blazor manualmente para atualizar o DOM.

Aplicativo de página única do Pizza Place

Vamos aplicar esse novo conhecimento e construir um bom site para pedidos de pizza.
No restante deste livro, você aprimorará este site com todos os tipos de recursos.

Página 49

Criando o Projeto PizzaPlace

Crie um novo projeto hospedado no Blazor, usando Visual Studio ou dotnet cli.
Consulte a explicação no primeiro capítulo se não se lembrar de como. Ligue para o projeto Pizzaria. Você obtém um projeto semelhante ao projeto MyFirstBlazor. Agora vamos aplicar algumas mudanças!

> Fora da caixa, o Blazor usa a popular estrutura de layout Bootstrap 4. Contudo, você pode usar qualquer outra estrutura de layout, porque o Blazor usa htML padrão e CSS. este livro é sobre o Blazor, não layouts sofisticados, então não vamos gastar muito de tempo escolhendo cores agradáveis e fazendo o site parecer ótimo. Foco!

No projeto do servidor, jogue fora SampleDataController.cs. Você não precisa previsões do tempo para pedir pizzas. No projeto compartilhado, jogue fora WeatherForecast.cs. Mesma coisa. No projeto do cliente, jogue fora o Counter.cshtml e arquivos FetchData.cshtml da pasta Pages, e SurveyPrompt.cshtml de a pasta compartilhada.

Sua solução deve ser semelhante à Figura 2-3.

Página 50

Figura 2-3. *A solução depois de remover arquivos desnecessários*

Adicionando classes compartilhadas para representar os dados

No Blazor, é melhor adicionar classes contendo dados ao projeto compartilhado. Essas aulas são usado para enviar os dados do servidor para o cliente e, posteriormente, para enviar os dados de volta. O que você precisa? Comece com aulas que representam uma pizza e como ela é picante, conforme mostrado na Listagem 2-16 .

Página 51

Listagem 2-16. As aulas de especiarias e pizza

```
using System;
usando System.Linq;

namespace PizzaPlace.Shared
{
    public enum Spiciness
    {
        Nenhum,
        Picante,
        Quente
    }
    pizza de classe pública
    {
        Pizza pública (int id, nome da string, preço decimal,
                       Picante picante)
        {
            this.Id = id;
            this.Name = nome
                ?? lance novo ArgumentNullException (nameof (name),
                    "Uma pizza precisa de um nome!");
            this.Price = price;
            this.Spiciness = picante;
        }
```

```
        public int Id {get; }
        public string Name {get; }
        Preço decimal público {get; }
        Picante público Picante {obter; }
    }
}
```

Seu aplicativo NÃO é sobre edição de pizzas, então tornei esta aula *imutável* , então nada pode ser alterado após a criação de um objeto pizza. Em C #, isso é feito facilmente por criando propriedades com apenas um getter.

Em seguida, você precisa de uma classe que representa o menu que você oferece. Adicionar uma nova turma ao compartilhado projeto denominado Menu com a implementação da Listagem 2-17 .

Página 52

Listagem 2-17. A Classe do Menu

```
using System.Collections.Generic;
usando System.Linq;
namespace PizzaPlace.Shared
{
    menu de aula pública
    {
        Lista pública <Pizza> Pizzas {get; definir; }
        = nova lista <Pizza> ();
        public Pizza GetPizza (int id)
        => Pizzas.SingleOrDefault (pizza => pizza.Id == id);
    }
}
```

Como na vida real, o menu de um restaurante é uma lista de refeições, neste caso uma pizza.

Você também precisa de uma classe Customer no projeto compartilhado com implementação de Listagem 2-18.

Listagem 2-18. A Classe do Cliente

```
using System;
using System.Collections;
using System.ComponentModel;
namespace PizzaPlace.Shared
{
    cliente de classe pública
    {
        public int Id {get; definir; }

        public string Name {get; definir; }

        public string Street {get; definir; }

        public string City {get; definir; }

    }
}
```


Página 53

Cada cliente tem uma cesta de compras, então adicione a classe Basket ao projeto compartilhado, como mostrado na lista 2-19 .

Listagem 2-19. A classe da cesta, que representa o pedido do cliente

```
using System.Collections.Generic;

namespace PizzaPlace.Shared
{
    cesta de classe pública
    {
        Cliente público Cliente {get; definir; } = novo cliente ();

        Lista pública <int> Pedidos {get; definir; } = nova lista <int> ();

        public bool HasPaid {get; definir; } = falso;

    }
}
```

Observe que você apenas mantém o ID da pizza na coleção Pedidos. Você vai aprender porque mais tarde.

Mais uma aula antes de agrupá-los todos. Você usará uma classe de IU para acompanhar de algumas opções de IU, portanto, adicione esta classe ao projeto compartilhado, conforme mostrado na Listagem 2-20.

Listagem 2-20. A classe de opções de interface do usuário

```
namespace PizzaPlace.Shared
{
    IU de classe pública
    {
        public bool ShowBasket {get; definir; } = verdadeiro;
    }
}
```

Finalmente, você agrupa todas essas classes em uma única classe de estado, novamente na classe compartilhada projeto com implementação da Listagem 2-21 .

Página 54

Listagem 2-21. A classe estadual

```
usando System.Linq;

namespace PizzaPlace.Shared
{
    classe pública estado
    {
        Menu público Menu {get; definir; } = novo Menu ();

        public Basket Basket {get; definir; } = novo cesto ();

        IU da IU pública {get; definir; } = nova IU ();

    }
}
```

há outro bom motivo para colocar todas essas classes no projeto compartilhado. lá é a depuração limitada para Blazor e não há estrutura de teste de unidade. Colocando essas classes no projeto compartilhado, você pode aplicar as práticas recomendadas de teste de unidade nas classes compartilhadas porque é um projeto normal .net Core, e até mesmo usar o depurador para examinar o comportamento estranho.

Construindo a IU para mostrar o menu

Com as classes estabelecidas para representar os dados, a próxima etapa é construir a interface do usuário que mostra o menu. Você começará exibindo o menu para o usuário e, em seguida, aprimore a IU para permitir que o usuário peça uma ou mais pizzas.

Exibindo o Menu

O problema de exibição do menu é duplo. Primeiro, você precisa exibir uma lista de dados. O menu pode ser considerado uma lista, como qualquer outra lista. Em segundo lugar, em seu aplicativo, você precisam converter as opções picantes de seus valores numéricos em URLs que levam ao ícones usados para indicar diferentes níveis de gostosura.

Página 55

Exibindo uma lista de dados

Abra Index.cshtml. Adicione a seção @functions para manter o seu restaurante (limitado) menu com código da lista 2-22 inicializando a instância State com um Menu.

Listagem 2-22. Construindo o menu do seu aplicativo

```
@funções {

    estado privado Estado {get; } = novo estado ()
    {
        Menu = novo Menu
        {
            Pizzas = nova lista <Pizza>
            {
                Pizza nova (1, "Pepperoni", 8,99M, Spiciness.Spicy),
                pizza nova (2, "Margarita", 7,99 milhões, tempero.Nenhuma),
                pizza nova (3, "Diabolo", 9,99 milhões, especiarias.quente)
            }
        }
    };
}
```

O menu Pizza Place é uma lista como qualquer outra lista. Você pode exibi-lo adicionando alguns Marcação do Razor para gerar o menu como HTML, conforme mostrado na Listagem 2-23.

Listagem 2-23. Gerando o HTML com o Razor

```
@página "/"
@using PizzaPlace070.Shared
<! - Menu ->

<h1> Nossa seleção de pizzas </h1>

@foreach (var pizza in State.Menu.Pizzas)
{
    <div class = "row">
        <div class = "col">
            @ pizza.Name
        </div>
```

Página 56


```
        <div class = "col">
            @ pizza.Price
        </div>
        <div class = "col">
            <img src = "@ SpicinessImage (pizza.Spiciness)"
                    alt = "@ pizza.Spiciness" />
        </div>
        <div class = "col">
            <button class = "btn btn-success"
                        onclick = "@ (() => AddToBasket (pizza))">
                Adicionar
            </button>
        </div>
    </div>
}

<! - Menu final ->
```

Gosto de usar comentários para mostrar o início e o fim de cada seção da minha página. esta torna mais fácil encontrar uma determinada parte da minha página quando eu volto a ela mais tarde. no próximo capítulo, você converterá cada seção em seu próprio componente Blazor, tornando a manutenção futura é muito mais fácil de fazer.

O que você está fazendo aqui é iterar cada pizza no menu e gerar uma linha com quatro colunas: uma para o nome, preço, sabor picante e, finalmente, uma para o botão de pedido.

Convertendo Valores

Você ainda tem um pequeno problema. Você precisa converter o valor picante em um URL, que é feito pelo método SpicinessImage mostrado na listagem 2-24. Adicione este método ao área @functions do arquivo Index.cshtml.

Listagem 2-24. Convertendo um valor com uma função de conversor

```
string privada SpicinessImage (Spiciness spiciness)
    => $ "images / {spiciness.ToString (). ToLower ()}. png";
```


Página 57

Esta função de conversão simplesmente converte o nome do valor da enumeração de Listagem 2-14 no URL de um arquivo de imagem, que pode ser encontrado na seção do projeto Blazor pasta de imagens, conforme mostrado na Figura 2-4 (imagens cortesia de https://openclipart.org). Adicione esta pasta (que pode ser encontrada no download deste livro) à pasta wwwroot.

Figura 2-4. *A pasta de imagens*

Adicionando Pizzas ao Carrinho de Compras

O funcionamento do menu leva naturalmente à adição de pizzas ao carrinho de compras. Ao clicar no botão Adicionar, o método AddToBasket será executado com o pizza. Você pode encontrar a implementação do método AddToBasket na Listagem2-25 . Para tornar a depuração mais fácil, você adiciona um Console.WriteLine, que aparecerá no console do navegador.

Listagem 2-25. Pedindo uma pizza

```
private void AddToBasket (Pizza pizza)
{
    Console.WriteLine ($ "Adicionou pizza {pizza.Name}");
    State.Basket.Add (pizza.Id);
}
```

Sua classe Basket agora precisa de um método Add, mostrado na Listagem 2-26.

Listagem 2-26. O método de adição da cesta

```
public void Add (int pizzaId)
{
    Orders.Add (pizzaId);
}
```


Página 58

Veja o manipulador de eventos onclick para o botão da Listagem 2-23. Por que é isso manipulador de eventos usando um lambda? Quando você pede uma pizza, é claro que você quer ter seu pizza escolhida adicionada à cesta. Então, como você pode passar a pizza para AddToBasket de Listagem 2-25? Usando uma função lambda, você pode simplesmente passar a variável pizza usada em o loop @foreach para ele. Usar um método normal não funcionaria porque não é fácil forma de enviar a pizza selecionada. Isso também é conhecido como *encerramento* (muito semelhante ao JavaScript fechamentos) e pode ser muito prático!

Execute o aplicativo. Você deve ver a Figura 2-5 .

Figura 2-5. *O menu Pizza Place*

Ao clicar no botão Adicionar, você está adicionando uma pizza à cesta de compras. Mas como você pode ter certeza (já que você não está exibindo a cesta de compras ainda)?

Abra as ferramentas de depuração do navegador e observe o Console. Cada vez que você clica Adicione, você deve ver alguma saída do Console.WriteLine no AddToBasket método, conforme mostrado na Figura 2-6.

Figura 2-6. *Olhando para a saída de Console.WriteLine*

Página 59

Mostrando a cesta de compras

A próxima coisa no menu (algum trocadilho intencional) é exibir a cesta de compras. Vocês usarão um novo recurso do C # 7 chamado tuplas. Explicarei as tuplas em um momento. Isso requer a adição do pacote System.ValueTuple NuGet.

Adicionando um pacote com o Visual Studio

Para adicionar este pacote NuGet com Visual Studio, clique com o botão direito do mouse no projeto do cliente e selecione Gerenciar pacote NuGet, conforme ilustrado pela Figura 2-7. Procure o pacote e instale-o.

Figura 2-7. *Instalando o pacote System.ValueTuple com NuGet*

Adicionando um pacote com o código do Visual Studio

Para adicionar o pacote com o Visual Studio Code, selecione o arquivo PizzaPlace.Client.csproj e adicione uma nova referência de pacote:

```
<PackageReference Include = "System.ValueTuple"
                  Versão = "4.5.0" />
```

Exibindo o carrinho de compras

Agora você está pronto para exibir a cesta de compras. Adicione a Listagem 2-27 após o menu de Listagem 2-21.

Listagem 2-27. Exibindo o carrinho de compras

```
<! - Menu final ->
<! - Carrinho de compras ->
@if (State.Basket.Orders.Any ())
{
    <h1> Seu pedido atual </h1>
```


Página 60


```
    @foreach (var (pizza, pos) em
        State.Basket.Orders.Select (
          (id, pos) => (State.Menu.GetPizza (id), pos)))
    {
      <div class = "row">
        <div class = "col">
          @ pizza.Name
        </div>
        <div class = "col">
          @ pizza.Price
        </div>
```

```
            <div class = "col">
                <button class = "btn btn-perigo"
                        onclick = "@ (() => RemoveFromBasket (pos))">
                    Remover
                </button>
            </div>
        </div>
    }
    <div class = "row">
        <div class = "col"> Total: </div>
        <div class = "col"> @ State.TotalPrice </div>
        <div class = "col"> </div>
    </div>
}

<! - Fim da cesta de compras ->
```

A maioria dessas coisas é muito semelhante, mas agora você está iterando sobre uma lista de tuplas (uma novo recurso útil em C # 7). Vejamos este código com um pouco mais de detalhes na Listagem2-28 .

Listagem 2-28. Convertendo o carrinho de compras para fácil exibição

```
@foreach (var (pizza, pos) em
    State.Basket.Orders.Select (
      (id, pos) => (State.Menu.GetPizza (id), pos)))
```

Página 61

Você está usando o Select do LINQ para iterar a lista de pedidos (que contém pizza ids). Para exibir a pizza na cesta de compras, você precisa de uma pizza, então você converte o id para uma pizza com o método GetPizza do Menu. Por favor, adicione este método de Listagem 2-29 para a classe Menu (e usando System.Linq).

Listagem 2-29. O método GetPizza

```
public Pizza GetPizza (int id)
    => Pizzas.SingleOrDefault (pizza => pizza.Id == id);
```

Este método converte um id de pizza em uma pizza usando LINQ.

Vejamos a função lambda usada no Select mostrado na Listagem 2-30 .

Listagem 2-30. Criação de tuplas

```
(id, pos) => (State.Menu.GetPizza (id), pos)
```

O método LINQ Select tem duas sobrecargas, e você está usando a sobrecarga, levando um elemento da coleção (id) e a posição na coleção (pos). Você usa para criar tuplas. Cada tupla representa uma pizza da cesta e sua posição na O cesto!

A pizza é usada para exibir seu nome e preço, enquanto a posição é usada no Botão Excluir. Este botão invoca o método RemoveFromBasket da Listagem2-31 .

Listagem 2-31. Removendo itens da cesta de compras

```
private void RemoveFromBasket (int pos)
{
    Console.WriteLine ($ "Removendo pizza na pos {pos}");
    State.Basket.RemoveAt (pos);
}
```

E, claro, você precisa adicionar o método RemoveAt à classe Basket, como mostrado na Listagem 2-32 .

Listagem 2-32. O método RemoveAt da classe Basket

```
public void RemoveAt (int index)
{
    Orders.RemoveAt (index);
}
```


Página 62

Na parte inferior da cesta de compras, o valor total do pedido é mostrado. Isso é calculado pela classe State. Adicione o método TotalPrice da Listagem 2-33 ao Estado aula. Não se esqueça de adicionar uma instrução using System.Linq ao início.

Listagem 2-33. Calculando o Preço Total na Classe Estadual

```
TotalPrice decimal público
=> Basket.Orders.Sum (id => Menu.GetPizza (id) .Price);
```

Execute o aplicativo e peça algumas pizzas. Você deve ver um pedido atual semelhante a Figura 2-8.

Figura 2-8. *Sua cesta de compras com algumas pizzas*

Insira o cliente

Claro, para concluir o pedido, você precisa saber algumas coisas sobre o cliente, principalmente o endereço, porque você precisa entregar o pedido.

Comece adicionando o Razor da Listagem 2-34 à sua página Index.cshtml.

Listagem 2-34. Adicionando elementos de formulário para entrada de dados

```
<! - Fim da cesta de compras ->
<! - Entrada do cliente ->
<h1> Insira seus dados abaixo </h1>
<fieldset>
  <p>
    <label for = "name"> Nome: </label>
    <input id = "name" bind = "@ State.Basket.Customer.Name" />
  </p>
  <p>
```


Página 63


```
    <label for = "street"> Rua: </label>
    <input id = "street" bind = "@ State.Basket.Customer.Street" />
  </p>
  <p>
    <label for = "city"> Cidade: </label>
    <input id = "city" bind = "@ State.Basket.Customer.City" />
  </p>
```

```
    <button onclick = "@ PlaceOrder"> Check-out </button>

</fieldset>

<! - Finalizar entrada do cliente ->
```

Isso adiciona três rótulos e suas respectivas entradas para nome, rua e cidade.

Você também precisa adicionar o método PlaceOrder às suas funções, conforme mostrado em Listagem 2-35.

Listagem 2-35. O Método PlaceOrder

```
@funções {

...

private void PlaceOrder ()
{
    Console.WriteLine ("Ordem de colocação");
}

}
```

O método PlaceOrder não faz nada ainda; você enviará o pedido para o servidor mais tarde.

Execute o aplicativo e insira seus detalhes, como na Figura 2-9 .

Página 64

Figura 2-9. *Preenchendo os detalhes do cliente*

Dica de depuração

Blazor tem depuração limitada, e você deseja ver o objeto State porque ele contém os detalhes e o pedido do cliente. Você enviará as informações corretas para o servidor quando você pressiona o botão Checkout? Para isso, você usará um truque simples, exibindo o estado em sua página para que você possa revisá-lo a qualquer momento. Comece adicionando uma nova classe chamada DebuggingExtensions para seu projeto Blazor, conforme mostrado na Listagem 2-36.

Listagem 2-36. A classe DebuggingExtensions

```
usando Microsoft.AspNetCore.Blazor;

namespace PizzaPlace.Client
{
    public static class DebuggingExtensions
    {
        public static string ToJson (este objeto obj)
```

```
    => Microsoft.JSInterop.Json.Serialize (obj); }
}
```

E na parte inferior de Index.cshtml adicione um parágrafo simples, conforme mostrado na Listagem 2- 37.

Listagem 2-37. Mostrando estado

```
<! - Finalizar entrada do cliente ->

<p> @ State.ToJson () </p>
```


Página 65

Execute seu projeto. Conforme você interage com a página, você verá a mudança de estado, com um exemplo mostrado na Figura 2-10 .

Figura 2-10. *Assistindo a mudança de estado*

deve ser óbvio que você deve remover este recurso de depuração quando a página está pronto. ☺

Validando as informações do cliente

Mas espere! Clicar no botão Checkout funciona, mesmo quando não há nome de cliente, endereço ou cidade! Você precisa fazer alguma validação! Então, vamos começar com uma introdução a Validação do .NET.

Permitindo que as entidades se validem

Classes como Customer devem se validar porque têm o melhor conhecimento sobre a validade de suas propriedades. .NET tem algumas validações integradas mecanismos, e aqui você usará o System.ComponentModel padrão. Interface INotifyDataErrorInfo mostrada na Listagem 2-38.

Listagem 2-38. A interface System.ComponentModel.INotifyDataErrorInfo

```
interface pública INotifyDataErrorInfo
{
    bool HasErrors {get; }

    event EventHandler <DataErrorsChangedEventArgs> ErrorsChanged;

    IEnumerable GetErrors (string propertyName);
}
```


Página 66

Sua principal característica é o método GetErrors, que retorna quaisquer erros de validação para um propriedade como um IEnumerable. Para atualizar, IEnumerable e IEnumerable <T> são usados pela palavra-chave C # foreach para iterar. A propriedade HasErrors verifica se há

quaisquer erros; ele pode ser usado para desativar o botão Checkout.

Vamos fazer o Cliente implementar a interface INotifyDataErrorInfo. Lá é mais uma coisa que você deve fazer primeiro, no entanto. A propriedade HasErrors do A interface INotifyDataErrorInfo deve retornar um booleano. Uma maneira fácil de fazer isso é simplesmente chamar o método de extensão Any do LINQ no método GetErrors. Infelizmente, este método retorna um IEnumerable, e LINQ só funciona com o interface IEnumerable <T> genérica mais recente. Não tem problema: você mesmo pode construir facilmente! Adicione uma nova classe chamada IEnumerableExtensions ao projeto compartilhado com o Any método de extensão da Listagem 2-39.

Listagem 2-39. Adicionando o método Any Extension a IEnumerable

```
using System.Collections;

namespace PizzaPlace.Shared
{
    public static class IEnumerableExtensions
    {
        public static bool Any (este IEnumerable enumerable)
            => enumerable.GetEnumerator (). MoveNext () == true;
    }
}
```

Agora implemente a interface INotifyDataErrorInfo para a classe Customer, como mostrado na Listagem 2-40.

Listagem 2-40. A classe do cliente com INotifyDataErrorInfo

```
using System;
using System.Collections;
using System.ComponentModel;
namespace PizzaPlace.Shared
{
public class Customer: INotifyDataErrorInfo
{
    ...
```


Página 67


```
        public bool HasErrors => GetErrors (string.Empty) .Any ();

        evento público EventHandler <DataErrorsChangedEventArgs> ErrorsChanged;

        public IEnumerable GetErrors (string propertyName)
        {
            if (string.IsNullOrEmpty (propertyName)
                    || propertyName == nameof (Name))
            {
                if (string.IsNullOrEmpty (Nome))
                {
                    yield return $ "O nome de um cliente é obrigatório";
                }
                else if (Name.Contains ("Pizza"))
                {
                    yield return $ "O nome não deve conter \" Pizza \ "";
                }
            }
            if (string.IsNullOrEmpty (propertyName)
                    || propertyName == nameof (Street))
            {
                if (string.IsNullOrEmpty (Street))
                {
                    retorno de rendimento $ "{propertyName} é obrigatório";
```

```
            }
        }
        if (string.IsNullOrEmpty (propertyName)
            || propertyName == nameof (City))
        {
            if (string.IsNullOrEmpty (City))
            {
                retorno de rendimento $ "{propertyName} é obrigatório";
            }
        }
    }
}
}
```

47

Página 68

Eu implementei para tornar cada propriedade obrigatória e, como um exemplo extra, que O nome não deve conter “Pizza” (substitua por qualquer validação que você achar adequada).

> Certifique-se de que a segunda validação esteja dentro de uma cláusula else; caso contrário você pode obtenha uma NullReferenceException em tempo de execução!

Observe que o método GetErrors retorna TODOS os erros de validação quando você o chama com um propertyName vazio. A propriedade HasErrors simplesmente chama GetErrors com um vazio propertyName e então usa seus métodos de extensão Any para retornar true se a coleção não está vazio.

Mostrando Erros de Validação

Agora que o cliente tem validação, você pode adicionar alguma IU para mostrar os erros de validação como feedback para o usuário. Não acho que preciso explicar que isso é simplesmente uma boa (e obrigatório) prática! Comece adicionando IU de validação para Nome, conforme mostrado na Listagem2-41 .

Listagem 2-41. IU de validação para o nome de um cliente

```
<p>
    <label for = "name"> Nome: </label>
    <input id = "name" bind = "@ State.Basket.Customer.Name" />
    @if (State.Basket.Customer
                                .GetErrors (nameof (Customer.Name))
                                .Algum())
    {
        <ul class = "validation-error">
            @foreach (erro de string em State.Basket.Customer
                                .GetErrors (nameof (Customer.Name)))
            {
                <li> @error </li>
            }
        </ul>
    }
</p>
```

48

Página 69

Vamos discutir essa lógica. Primeiro, você não precisa mostrar nenhuma IU de validação se não houver erros de validação. Portanto, você começa verificando se há algum erro em Nome. Eu repeti esta lógica na listagem 2-42.

Listagem 2-42. Verificando se há algum erro de validação para o nome de um cliente

```
@if (State.Basket.Customer
                    .GetErrors (nameof (Customer.Name))
                    .Algum())
```

Você chama GetErrors para a propriedade Name e usa o método de extensão Any para transformá-lo em um booleano. Se houver erros, você usa uma lista não ordenada para mostrá-los, como em Listagem 2-43.

Listagem 2-43. Usando uma lista não ordenada para mostrar erros de validação

```
<ul class = "validation-error">
   @foreach (erro de string em State.Basket.Customer
                    .GetErrors (nameof (Customer.Name)))
   {
      <li> @error </li>
   }
</ul>
```

O elemento ul tem uma classe CSS de erro de validação para estilização. Procure no wwwroot pasta para a pasta css e adicione (por exemplo) o estilo simples da Listagem 2-44.

Listagem 2-44. Adicionando alguns estilos para exibir erros de validação

```
.validation-error li {
   cor vermelha;
}
```

Lista de repetição 2-41 para as propriedades Street e City, então você obtém a Listagem 2-45.

Página 70

Listagem 2-45. Concluindo a validação para rua e cidade

```
<fieldset>
   <p>
      <label for = "name"> Nome: </label>
      <input id = "name" bind = "@ State.Basket.Customer.Name" />
      @if (State.Basket.Customer
                    .GetErrors (nameof (Customer.Name))
                    .Algum())
      {
      <ul class = "validation-error">
         @foreach (erro de string em State.Basket.Customer
                    .GetErrors (nameof (Customer.Name)))
         {
            <li> @error </li>
         }
      </ul>
      }
   </p>
```

```
<p>
  <label for = "street"> Rua: </label>
  <input id = "street" bind = "@ State.Basket.Customer.Street" />
  @if (State.Basket.Customer
                        .GetErrors (nameof (Customer.Street))
                        .Algum())
  {
  <ul class = "validation-error">
    @foreach (erro de string em State.Basket.Customer
                        .GetErrors (nameof (Customer.Street)))
    {
      <li> @error </li>
    }
  </ul>
  }
</p>
```

50

Página 71


```
<p>
  <label for = "city"> Cidade: </label>
  <input id = "city" bind = "@ State.Basket.Customer.City" />
  @if (State.Basket.Customer
                        .GetErrors (nameof (Customer.City))
                        .Algum())
  {
  <ul class = "validation-error">
    @foreach (erro de string em State.Basket.Customer
                        .GetErrors (nameof (Customer.City)))
    {
      <li> @error </li>
    }
  </ul>
  }
</p>

<button onclick = "@ PlaceOrder" disabled = "@ State.Basket.Customer.
HasErrors "> Check-out </button>

</fieldset>
```

Desativando o botão de check-out

Finalmente, você não deseja permitir que o usuário clique no botão Checkout quando houver quaisquer erros de validação. Uma maneira fácil é desabilitar o botão Checkout. Você vai usar um atributo condicional para definir o atributo de desativação, conforme mostrado na Listagem 2-46.

Listagem 2-46. Usando Vinculação de Atributo para Habilitar / Desabilitar o Botão Check-out

```
<button onclick = "@ PlaceOrder"
                disabled = "@ State.Basket.Customer.HasErrors">
  Confira
</button>
```

Portanto, assim que houver um erro de validação, este botão será desabilitado, interrompendo o cliente de colocar o pedido.

Administre o site. Seu cliente deve ver os erros de validação em vermelho, conforme mostrado na Figura 2-11 .

51

Página 72

Figura 2-11. *Mostrando erros de validação*

Resumo

Neste capítulo, você examinou a vinculação de dados no Blazor. Você começou com dados unilaterais vinculação onde você incorpora o valor de uma propriedade de campo na IU usando o Sintaxe @SomeProperty. Você então olhou para a associação de evento, onde você vincula um elemento evento para um método usando a sintaxe on <event> = "@ SomeMethod". Blazor também suporta ligação de dados bidirecional em que você atualiza a IU com o valor de uma propriedade e vice versa usando a sintaxe bind = "@ SomeProperty". Finalmente, você examinou a validação onde você pode usar técnicas de validação padrão do .NET como o INotifyDataErrorInfo interface.

Página 73

CAPÍTULO 3

Componentes e Estrutura para Blazor Formulários

No capítulo anterior sobre vinculação de dados, você construiu um único aplicativo monolítico com

Blazor. Depois de um tempo, ficará cada vez mais difícil de manter.

No desenvolvimento da web moderno, construímos aplicativos a partir de componentes, que normalmente são construídos a partir de componentes menores. Um componente Blazor é um pedaço autocontido de interface do usuário. Os componentes do Blazor são classes construídas a partir do Razor e C # com um propósito específico (também conhecido como *o princípio da responsabilidade única*) e são mais fáceis de entender, depurar e manter. E, claro, você pode usar o mesmo componente em páginas diferentes.

O que é um componente Blazor?

Para simplificar, cada arquivo CSHTML no Blazor é um componente. É isso simples! Um arquivo Razor no Blazor contém marcação e tem código na seção @functions. Cada página que você no projeto MyFirstBlazor é um componente! E os componentes podem ser construído adicionando outros componentes como filhos.

Abra o projeto MyFirstBlazor no Visual Studio (ou Code) e vamos dar uma olhada em alguns dos componentes lá.

Abra index.cshtml (Listagem 3-1).

P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_3

Página 74

Listagem 3-1. A página de índice

```
@página "/"

<h1> Olá, mundo! </h1>

Bem-vindo ao seu novo aplicativo.

<SurveyPrompt Title = "Como o Blazor está funcionando para você?" />
```

Veja SurveyPrompt? É um dos componentes do modelo Blazor. É preciso um parâmetro, Título, que você pode definir onde deseja usar o componente. Vamos ter um dê uma boa olhada no componente SurveyPrompt.

Examinando o componente SurveyPrompt

Abra SurveyPrompt.cshtml (consulte a lista 3-2), que pode ser encontrado na pasta Compartilhada de o projeto do cliente.

Listagem 3-2. O componente SurveyPrompt

```
<div class = "alert alert-secondary mt-4" role = "alert">
    <span class = "oi oi-pencil mr-2" aria-hidden = "true"> </span>
    <strong> @Title </strong>

    <span class = "text-nowrap">
        Por favor, pegue nosso
        <a target = "_ blank" class = "font-weight-bold"
        href = "https://go.microsoft.com/fwlink/?linkid=874928">
            breve pesquisa
        </a>
    </span>
    e diga-nos o que você pensa.
</div>

@funções {
[Parâmetro]
string Título {get; definir; } // Demonstra como um componente pai pode
fornecer parâmetros
```

}

54

Página 75

Veja a marcação do Razor. Este componente simples exibe um ícone na frente do Título, conforme mostrado na Figura 3-1e, em seguida, exibe um link para a pesquisa (que você deve (pegue ☺ porque mostrará à Microsoft que você está interessado no Blazor).

Figura 3-1. *O componente SurveyPrompt*

A seção de código @functions simplesmente contém a propriedade Title, que usa um forma de ligação de dados para renderização no componente. Observe o atributo [Parâmetro]. Isto é necessário para componentes que desejam expor suas propriedades ao componente pai. *Os parâmetros não podem ser propriedades públicas* , e o compilador apresentará um erro quando você tente fazer assim.

> Você pode se perguntar por que as propriedades [Parameter] não podem ser públicas. eu perguntei ao Daniel roth, que está na equipe Blazor, e esta é sua resposta: “pense em parâmetros como como parâmetros para um método ou construtor. eles não são algo que você deveria geralmente podem sofrer mutação externamente ao componente após terem sido Transmitido." Steve Sanderson, que é o principal autor de Blazor, explica que alterar o valor de um parâmetro do código não se comportará como esperado porque a detecção de mudanças não verá a mudança. Alterar o valor por meio de vinculação de dados mostra a mudança.

Construindo um Componente de Alerta Simples com o Razor

Vamos construir um componente Blazor simples que mostrará um alerta simples. Alertas são usados para chamar a atenção do usuário para alguma mensagem, por exemplo, um aviso.

Criando um Novo Componente com Visual Studio

Abra a solução MyFirstBlazor. Clique com o botão direito na pasta Páginas e selecione Adicionar> Novo Item. A janela Adicionar Novo Item deve abrir, como na Figura 3-2 .

55

Página 76

Figura 3-2. *A janela Adicionar Novo Item*

Selecione Razor View e nomeie-o Alert.cshtml. Clique no botão Adicionar.

Criando um Novo Componente com Código

Clique com o botão direito na pasta Páginas do projeto cliente e selecione Novo arquivo. Nomeie-o Alert.cshtml.

Implementar o componente de alerta

Remova todo o conteúdo existente de Alert.cshtml e substitua por Listagem 3-3.

Listagem 3-3. O Componente de Alerta

```
@if (Mostrar)
{
   <div class = "alert alert-secondary mt-4" role = "alert">
      @ChildContent
   </div>
}

@funções {
[Parâmetro]
bool Show {get; definir; }
```

56

Página 77


```
[Parâmetro]
RenderFragment ChildContent {get; definir; }
}
```

O componente Alerta irá mostrar qualquer conteúdo que você aninha nele (usando bootstrap estilo).

os modelos padrão do Blazor usam o Bootstrap 4 para estilização. Bootstrap (http: // getbootstrap.com) é uma estrutura Css muito popular, construída originalmente para o Twitter, fornecendo layout fácil para páginas da web. no entanto, o Blazor não exige que você use Bootstrap, para que você possa usar o estilo de sua preferência. se assim for, você deve atualizar todos os arquivos de navalha na solução para usar os outros estilos, assim como na web normal desenvolvimento. neste livro, usaremos o Bootstrap.

O @ChildContent conterá esse conteúdo e precisa ser do tipo RenderFragment porque é assim que o motor do Blazor o transmite (você verá isso mais adiante neste capítulo).

Volte para Index.cshtml e adicione o elemento Alert. Visual Studio é inteligente o suficiente para fornecer o IntelliSense (consulte a Figura 3-3) para o componente Alerta e seu parâmetros! O código do Visual Studio, infelizmente (no momento em que este capítulo foi escrito), não oferece IntelliSense ainda.

Figura 3-3. *Suporte do Visual Studio IntelliSense para componentes personalizados do Blazor*

Complete o Alerta e adicione um botão como na Listagem 3-4 .

Listagem 3-4. Usando o Componente de Alerta

```
<Alert Show = "@ ShowAlert">
    <span class = "oi oi-check mr-2" aria-hidden = "true"> </span>
    <strong> Blazor é tão legal! </strong>
</Alert>
```


Página 78


```
<button class = "btn btn-default" onclick = "@ ToggleAlert">
    Alternancia
</button>

@funções {

public bool ShowAlert {get; definir; } = verdadeiro;

public void ToggleAlert ()
{
    ShowAlert =! ShowAlert;
}
}
```

Dentro da tag <Alert> está um exibindo um ícone de marca de seleção e um elemento que exibe uma mensagem simples. Eles serão definidos como a propriedade @ChildContent de o componente Alerta. Construa e execute seu projeto. Quando você clica no <button>, ele chama o método ToggleAlert, que irá ocultar e mostrar o Alerta, conforme mostrado na Figura 3-4 .

Figura 3-4. *O componente de alerta simples antes de clicar no botão Alternar*

Separando a vista e o modelo de vista

Você pode não gostar dessa mistura de marcação (visualização) e código (modelo de visualização). Se você gosta, você pode usar dois arquivos separados, um para a visão usando o Razor e outro para o modelo de visão usando C #. A visualização exibirá os dados do modelo de visualização e manipuladores de eventos no view irá invocar métodos do modelo de view. Algumas pessoas preferem esta forma de trabalhar porque é mais parecido com o padrão MVVM. Vamos tentar isso!

Página 79

Criação de um componente DismissableAlert

Se você ainda não fez isso, abra a solução MyFirstBlazor. Com o Visual Studio, clique com o botão direito na pasta Pages e selecione Add ➤ New Item. A janela para Adicionar Novo Item deve abrir conforme mostrado na Figura 3-2. Desta vez, selecione Razor Page e nomeie-a DismissableAlert. Com o Visual Studio Code, clique com o botão direito na pasta Pages, selecione New

Arquivo e nomeie-o como DismissableAlert.cshtml. Faça isso novamente para criar um novo arquivo chamado DismissableAlert.cshtml.cs.

Um DismissableAlert é um alerta com um pequeno botão x que o usuário pode clicar para dispensar o alerta. Substitua a marcação no arquivo CSHTML com a Listagem 3-5.

Listagem 3-5. A marcação para DismissableAlert.cshtml

```
@if (Mostrar)
{
<div class = "alert alert-warning alert-dispensable fade show"
     role = "alert">
   @ChildContent
   <button type = "button" class = "close" data-despedir = "alerta"
                aria-label = "Fechar" onclick = "@ Dismiss">
     <span aria-hidden = "true"> & times; </span>
   </button>
</div>
}
```

Substitua o código C # em DismissableAlert.cshtml.cs pela Listagem 3-6 .

Listagem 3-6. O código para DismissableAlert.cshtml.cs

```
using System;
usando Microsoft.AspNetCore.Blazor.Components;
usando Microsoft.AspNetCore.Blazor;

namespace MyFirstBlazor.Client.Pages
{
      public class DismissableAlertViewModel: BlazorComponent
      {
```


Página 80


```
            [Parâmetro]
            protegido bool Show {get; definir; } = verdadeiro;

            [Parâmetro]
            protegido RenderFragment ChildContent {get; definir; }

            public void Dismiss ()
            {
                  Console.WriteLine ("Alerta de dispensa");
                  Mostrar = falso;
            }
      }
}
```

Observe que as propriedades Show e ChildContent agora são propriedades protegidas. Caso contrário, você não poderá referenciá-los no arquivo Razor. Também é importante herde aqui de BlazorComponent. Voltaremos ao BlazorComponent posteriormente neste capítulo.

A classe DismissableAlertViewModel servirá como a classe base para o arquivo Razor, que você precisa indicar com um @inherits no topo da marcação, que você pode encontrar na lista 3-7.

Listagem 3-7. Fazendo o CSHTML herdar do modelo de visualização

```
@inherits DismissableAlertViewModel

@if (Mostrar)
{
<div class = "alert alert-warning alert-dispensable fade show"
```

```
        role = "alert">
    @ChildContent
    <button type = "button" class = "close" data-despedir = "alerta"
                aria-label = "Fechar" onclick = "@ Dismiss">
      <span aria-hidden = "true"> & times; </span>
    </button>
</div>
}
```

60

Página 81

Então, em vez de colocar seu código na seção @functions de um arquivo Razor, você pode colocar o código em uma classe base e então herda dele no arquivo Razor.

Qual modelo é o melhor? Não acho que um seja melhor do que o outro; é mais um questão de gosto. Escolha o que você gosta.

Referindo-se a um componente filho

Os componentes pai e filho normalmente se comunicam por meio de vinculação de dados. Para exemplo, na Listagem 3-8 você usa DismissableAlert, que se comunica com o pai componente através da propriedade ShowAlert do pai. Clicar no botão Alternar irá ocultar e mostrar o alerta. Você pode tentar isso substituindo o conteúdo de Index.cshtml por Listagem 3-8.

Listagem 3-8. Usando DismissableAlert

```
<DismissableAlert Show = "@ ShowAlert">
      <span class = "oi oi-check mr-2" aria-hidden = "true"> </span>
      <strong> Blazor é tão legal! </strong>
</DismissableAlert>

<button class = "btn btn-default" onclick = "@ ToggleAlert"> Alternar </button>

@funções {

public bool ShowAlert {get; definir; } = verdadeiro;

public void ToggleAlert ()
{
     ShowAlert =! ShowAlert;
}
```

Em vez de usar vinculação de dados na interação entre pai e filho componente, você também pode interagir diretamente com o componente filho. Vejamos um exemplo. Digamos que você queira que o alerta desapareça automaticamente após 5 segundos.

61

Página 82

Adicionando um Componente de Cronômetro

Comece adicionando uma nova classe chamada Timer à pasta Pages, conforme mostrado na Listagem 3-9 (o cronômetro não terá nenhuma parte visual, então você nem precisa do CSHTML para construir o visualizar). Esta classe Timer irá invocar um delegado (Tick) após um certo número de segundos (TimeInSeconds) expirou. O parâmetro Tick é do tipo Action, que é um dos tipos de delegados integrados de .NET. Uma ação é simplesmente um método que retorna um vazio sem parâmetros. Existem outros tipos de ação genéricos, como Action <T>, que é um método retornando um vazio com um parâmetro do tipo T.

Listagem 3-9. The Timer Class

```
using System;
usando Microsoft.AspNetCore.Blazor.Components;

namespace MyFirstBlazor.Client.Pages
{
    public class Timer: BlazorComponent
    {
        [Parâmetro]
        protegido duplo TimeInSeconds {get; definir; }

        [Parâmetro]
        ação protegida Assinale {get; definir; }

        sobrescrito protegido void OnInit ()
        {
            base.OnInit ();

            var timer = new System.Threading.Timer (
                (_) => Tick.Invoke (),
                nulo,
                TimeSpan.FromSeconds (TimeInSeconds),
                System.Threading.Timeout.InfiniteTimeSpan);
        }
    }
}
```

62

Página 83

Agora adicione o componente Timer à página de índice, conforme mostrado na Listagem 3-10 . Vamos olhar em algumas coisas. Primeiro, você adiciona uma referência ao componente dispensávelAlert usando a sintaxe ref. Isso permitirá que você faça referência ao componente em seu código.

Listagem 3-10. Adicionando o componente Timer para ignorar o alerta

```
<DismissableAlert ref = "dismissableAlert"
                  Show = "@ ShowAlert">
    <span class = "oi oi-check mr-2" aria-hidden = "true"> </span>
    <strong> Blazor é tão legal! </strong>
</DismissableAlert>

<Timer TimeInSeconds = "5" Tick = "@ DismissAlert" />
```

Tenha cuidado ao usar <Timer> </Timer>. qualquer conteúdo, mesmo espaços em branco, será visto como ChildContent, e como o Timer não suporta nenhum, você pode obter erros do compilador. é melhor usar um único elemento <Timer />.

Isso requer que você adicione um campo denominado Alerta dispensável do tipo DismissableAlert para o pai, que conterá a referência ao filho componente, como você pode ver na Listagem 3-11.

Listagem 3-11. Usando um campo para se referir ao componente filho

```
@funções {
    public DismissableAlert dispensableAlert;
    public bool ShowAlert {get; definir; } = verdadeiro;
    public void ToggleAlert ()
    {
        ShowAlert =! ShowAlert;
    }
    public void DismissAlert ()
    {
        dispensávelAlert.Dismiss ();
    }
}
```

63

Página 84

Agora, quando o tempo acaba, ele invoca seu método Tick, que chama DismissAlert. DismissAlert chama o método Dismiss no dismissableAlert referência, que deve então ocultar o alerta.

Execute o aplicativo e aguarde pelo menos 5 segundos. O alerta não se esconde! Por que?!

Usando vinculação de dados componente a componente

Então, por que o DismissableComponent não se esconde após 5 segundos?

Observe a marcação, que está na Listagem 3-10 , para DismissibleAlert novamente. Isto mostra o componente com base no parâmetro Show e é definido por meio de vinculação de dados. O O problema é que o ShowAlert do componente pai Index permanece verdadeiro. Alterando a valor do campo de exibição local DismissableAlert não atualizará o índice do componente Propriedade ShowAlert. O que você precisa é de ligação de dados bidirecional entre os componentes e Blazor tem isso.

Com a vinculação de dados bidirecional, alterar o valor do parâmetro Show atualizará o valor da propriedade ShowAlert do pai e vice-versa.

Abra a classe DismissableAlertViewModel e altere a propriedade Show implementação, conforme mostrado na Listagem 3-12. Aqui você adiciona um parâmetro extra que deve ser chamado de <<yourproperty>> Alterado e deve ser do tipo Ação <<typeofyourproperty>>.

Listagem 3-12. A classe DismissableAlertViewModel com ligação bidirecional Apoiar

```
public class DismissableAlertViewModel: BlazorComponent
{
    show bool privado = verdadeiro;

    [Parâmetro]
    protegido bool Show
    {
        get => show;
        definir
        {
            if (mostrar! = valor)
            {
                show = valor;
```

64

Página 85

```
                ShowChanged? .Invoke (show);
            }
        }
    }

    [Parâmetro]
    protegido Ação <bool> ShowChanged {get; definir; }

    [Parâmetro]
    protegido RenderFragment ChildContent {get; definir; }

    public void Dismiss ()
    {
        Mostrar = falso;
    }
}
```

Agora, sempre que alguém ou algo muda o valor da propriedade Show, o O setter da propriedade aciona o delegado ShowChanged. Isso significa que o componente pai pode injetar algum código na propriedade do delegado ShowChanged, que invocará quando a propriedade é alterada (interna ou externamente).

> lembre-se de verificar se o valor mudou. isso irá ajudá-lo a evitar um desagradável bug onde a propriedade filho atualiza a propriedade pai, que aciona o filho propriedade a ser atualizada e assim por diante ad infinitum.

Corra novamente. Mesmo assim, o alerta não desaparece. Pense sobre isso. Você invoca um método de forma assíncrona usando um Timer. Quando o cronômetro dispara, você define a propriedade ShowAlert como falso. Mas você ainda precisa atualizar a IU. Você pode fazer isso chamando StateHasChanged no método DismissAlert da Listagem 3-11 . Mas existe uma maneira melhor, que é mostrada na Listagem 3-13 . Aqui você chama StateHasChanged sempre que a propriedade ShowAlert obtém um novo valor.

Página 86

Listagem 3-13. Atualizando a IU quando ShowAlert muda o valor

```
public bool ShowAlert
{
    get => showAlert;
    definir
    {
        if (showAlert! = valor)
        {
            showAlert = valor;
            this.StateHasChanged ();
        }
    }
}
```

Corre. Aguarde 5 segundos.
O alerta deve se ocultar automaticamente, conforme ilustrado pela Figura 3-5 e Figura 3-6.

Figura 3-5. *O alerta sendo mostrado*

Figura 3-6. *O alerta é ocultado automaticamente após 5 segundos*

Construindo uma Biblioteca de Componentes

Os componentes devem ser reutilizáveis. Mas você não quer reutilizar um componente entre projetos copiando e colando o componente entre eles. Neste caso é muito melhor para construir uma *biblioteca de componentes* e, como você verá, isso não é nada difícil! O que você vai fazer aqui está movendo o componente DismissableAlert and Timer para uma biblioteca e, em seguida, você use esta biblioteca em seu projeto Blazor.

Página 87

Criando o Projeto de Biblioteca de Componentes

No momento, você não pode criar bibliotecas de componentes do Blazor a partir do Visual Studio, então você terá que usar o prompt da linha de comando.

Abra um prompt de comando ou use o terminal integrado do Visual Studio Code (você pode usar Ctrl-`como um atalho para alternar o terminal em Código). Mudar o atual diretório para a pasta da solução. Digite o seguinte comando:

```
dotnet new blazorlib -o MyFirstBlazor.Components
```

O novo comando dotnet criará um novo projeto baseado em um modelo. O template que você quer é o template blazorlib. Se você deseja que o projeto seja criado em um subdiretório, você pode especificá-lo usando o parâmetro -o subdiretório.

A execução deste comando deve mostrar uma saída como:

```
O template "Biblioteca Blazor" foi criado com sucesso.
```

Adicione-o à sua solução digitando o próximo comando:

```
dotnet sln add MyFirstBlazor.Components \ MyFirstBlazor.Components.csproj
```

Desta vez, você deseja alterar a solução, e dotnet sln add permite que você adicione um projeto (que é o último argumento) para a solução. Quando você volta ao Visual Studio, irá informá-lo sobre uma modificação de arquivo, como mostrado na Figura 3-7.

Basta pressionar Reload para continuar trabalhando.

Figura 3-7. *O Visual Studio detectou alterações feitas na solução*

Página 88

Adicionando componentes à biblioteca

Anteriormente, você construía alguns componentes. Alguns deles são muito reutilizáveis, então você irá movê-los para o seu projeto de biblioteca. Comece com Timer.

Arraste e solte o arquivo Timer.cs de seu projeto cliente para o projeto de componentes. Você deve ver um novo arquivo Timer.cs, conforme ilustrado pela Figura 3-8.

Figura 3-8. *Copiando o arquivo Timer.cs para o projeto Componentes*

O Visual Studio cria uma cópia do arquivo, portanto, remova o arquivo Timer.cs do cliente projeto (não há necessidade de fazer isso com o Código). Clique com o botão direito do mouse no arquivo Timer.cs no projeto do cliente e selecione Excluir, como na Figura 3-9.

Página 89

Figura 3-9. *Excluir um arquivo de um projeto*

Faça o mesmo para DismissableAlert.cshtml. Ambos os componentes ainda estão usando o namespace do cliente, então atualize seu namespace para MyFirstBlazor.Components, conforme mostrado na Listagem 3-14 .

Listagem 3-14. Dispensando o Alerta

```
@inherits DismissableAlertViewModel

@if (Mostrar)
{
<div class = "alert alert-warning alert-dispensable fade show"
        role = "alert">
    @ChildContent
    <button type = "button" class = "close" data-despedir = "alerta"
                aria-label = "Fechar"
                onclick = "@ Dismiss">
        <span aria-hidden = "true"> & times; </span>
    </button>
</div>
}
```


Página 90

Construir a solução ainda irá disparar erros de compilador do projeto do cliente porque você precisa adicionar uma referência do projeto cliente à biblioteca de componentes, que você irá corrigir na próxima parte.

Referindo-se à Biblioteca do Seu Projeto

Agora que sua biblioteca está pronta, você vai usá-la em seu projeto. A forma como a biblioteca funciona é que você pode usá-lo em outros projetos. Ei, você poderia até mesmo torná-lo um NuGet pacote e deixe o resto do mundo desfrutar do seu trabalho!

Referindo-se a outro projeto com Visual Studio

Comece clicando com o botão direito do mouse em seu projeto cliente e selecionando Add ➤ Reference. Estúdio visual vai mostrar a figura 3-10 .

Figura 3-10. *Adicionando uma referência a outro projeto*

Certifique-se de verificar MyFirstBlazor.Components e clique em OK.

70

Página 91

Referindo-se a outro projeto com código

Abra o arquivo MyFirstBlazor.Client.csproj e adicione outro <ProjectReference>
elemento a ele, conforme mostrado na Listagem 3-15. É a última <ProjectReference> da Listagem 3-15
você precisa adicionar.

Listagem 3-15. Adicionando uma Referência a Outro Projeto

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

  <PropertyGroup>
    <TargetFramework> netstandard2.0 </TargetFramework>
    <OutputType> Exe </OutputType>
    <LangVersion> 7.3 </LangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Incluir = "Microsoft.AspNetCore.Blazor.Browser"
      Versão = "0.5.1" />
    <PackageReference
      Incluir = "Microsoft.AspNetCore.Blazor.Build"
      Versão = "0.5.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include = ".. \ MyFirstBlazor.Shared \ MyFirstBlazor.
    Shared.csproj "/>
    <ProjectReference Include = ".. \ MyFirstBlazor.Components \ MyFirstBlazor.
    Components.csproj "/>
  </ItemGroup>

</Project>
```

Agora você adicionou a biblioteca de componentes ao seu projeto, mas se quiser usar
os componentes em seus próprios arquivos CSHTML, você deve consultar sua biblioteca de componentes em
seus arquivos CSHTML.

71

Página 92

Compreendendo os ajudantes de tag

ASP.NET Core introduziu *ajudantes de marca* . Os ajudantes de tag são elementos personalizados que obtêm
convertido em elementos HTML padrão em tempo de execução. Se você usa ASP.NET Core normal
MVC, o servidor irá converter os ajudantes de tag em HTML, e no Blazor o cliente irá
converter ajudantes de tag. Na verdade, qualquer componente do Blazor se torna automaticamente um
ajudante de tag. O Visual Studio reconhece automaticamente os ajudantes de tag do projeto atual,
mas você precisa dar uma mão para as bibliotecas de componentes, e você faz isso com
@addTagHelper.

Abra _ViewImports.cshtml e adicione @addTagHelper como na Listagem 3-16 .

Listagem 3-16. Adicionando componentes de uma biblioteca Blazor

```
@layout MainLayout

@addTagHelper *, MyFirstBlazor.Components
```

Aqui você inclui todos os componentes personalizados usando um caractere curinga (*) do Biblioteca MyFirstBlazor.Components. De agora em diante, você pode usar qualquer componente deste biblioteca como uma tag HTML, por exemplo <Timer />.

Ao construir, você ainda obterá um erro de compilação, e isso é porque você está usando o tipo DismissableAlert em suas funções. E, assim como qualquer outro tipo, você pode refira-se a ele usando seu nome completo, incluindo o namespace, ou você pode adicionar uma instrução using para Index.cshtml como na Listagem 3-17.

Listagem 3-17. Adicionando uma instrução using para referir-se aos tipos do namespace

```
@página "/"
@using MyFirstBlazor.Components
```

Construa e execute sua solução. Deve ser semelhante à Figura 3-5 . Parabéns. Você tem acabou de construir e consumir sua primeira biblioteca de componentes do Blazor!

Refatorando PizzaPlace em componentes

No capítulo anterior sobre vinculação de dados, você construiu um site para pedir pizzas. Costumava apenas um componente com três seções diferentes. Vamos dividir este componente em componentes menores e mais fáceis de entender e tente maximizar a reutilização.

Página 93

Criação de um componente para exibir uma lista de pizzas

Abra o projeto PizzaPlace Blazor do capítulo anterior. Comece revisando index.cshtml. Este é o seu componente principal e tem três seções principais: um menu, um cesta de compras e informações do cliente.

O menu lista as pizzas e exibe cada uma com um botão para pedir. As compras cesta também exibe uma lista de pizzas (mas agora da cesta de compras) com um botão para remova-os do pedido. Parece que ambos têm algo em comum: eles precisam exibir pizzas com uma ação que você escolher clicando no botão.

Adicione um novo componente à pasta Pages chamado PizzaItem com conteúdo de Listagem 3-18. Você pode copiar a maior parte da marcação do componente Índice com alguns alterar.

Listagem 3-18. O componente PizzaItem

```
@using PizzaPlace.Shared

<div class = "row">
    <div class = "col">
        @ Pizza.Name
    </div>
    <div class = "col">
        @ Pizza.Price
    </div>
    <div class = "col">
        <img src = "@ SpicinessImage (Pizza.Spiciness)"
                alt = "@ Pizza.Spiciness" />
    </div>
    <div class = "col">
        <button class = "@ ButtonClass"
```

```
    </div>          onclick = "@ (() => Selecionado (Pizza))"> Adicionar </button>
</div>

@funções {
```

73

Página 94


```
[Parâmetro]
Pizza Pizza protegida {get; definir; }

[Parâmetro]
string protegida ButtonTitle {get; definir; }

[Parâmetro]
string protegida ButtonClass {get; definir; }

[Parâmetro]
ação protegida <Pizza> selecionada {obter; definir; }

string privada SpicinessImage (Spiciness spiciness)
=> $ "images / {spiciness.ToString (). ToLower ()}. png";

}
```

O componente PizzaItem exibirá uma pizza, então não deve ser uma surpresa que tem um parâmetro Pizza. Este componente também exibe um botão, mas como este a aparência e o comportamento do botão variam dependendo de onde você o usa. E é por isso que tem um parâmetro ButtonTitle e ButtonClass para mudar a aparência do botão, e também tem uma ação Selecionada que é chamada quando você clica no botão.

Agora você pode usar este componente para exibir o menu. Adicione um novo componente ao Pasta Pages chamada PizzaList.cshtml como na Listagem 3-19.

Listagem 3-19. O componente PizzaList

```
@using PizzaPlace.Shared

<h1> @Title </h1>

@foreach (var pizza em Menu.Pizzas)
{
   <PizzaItem Pizza = "@ pizza" ButtonTitle = "Pedido"
                       ButtonClass = "btn btn-sucesso"
                       Selecionado = "@ ((p) => Selecionado (p))" />
}

@funções {
```

74

Página 95


```
    [Parâmetro]
    string protegida Título {get; definir; }

    [Parâmetro]
    protegido Menu Menu {get; definir; }
```

```
[Parâmetro]
ação protegida <Pizza> selecionada {obter; definir; }
}
```

O componente PizzaList exibe um Título e todas as pizzas do Menu, então toma-os como parâmetros. Ele também executa uma ação Selecionada que você invoca clicando em o botão ao lado de uma pizza. Observe que o componente PizzaList usa o componente PizzaItem para exibir cada pizza, e se a ação PizzaList Selecionada é passada diretamente para a ação PizzaItem Selecionado. O componente Índice definirá esta ação, e será executado pelo componente PizzaItem.

Com PizzaItem e PizzaList prontos, você pode usá-los no Índice, que você pode encontrar na lista 3-20 .

Listagem 3-20. Usando o componente PizzaList no índice

```
<! - Menu ->

<PizzaList Title = "Nossa lista selecionada de pizzas"
           Menu = "@ State.Menu"
           Selecionado = "@ ((pizza) => AddToBasket (pizza))" />

<! - Menu final ->
```

Execute o aplicativo e tente pedir uma pizza. A cesta de compras não exibe quando você clica nos botões de pedido! Por quê? Porque a IU não é atualizada. Você precisa para consertar isso. Você já viu como fazer isso. Pense nisso.

Atualizar a IU após alterar o objeto de estado

Comece alterando o método AddToBasket de Index, como na listagem 3-21 . Depois de adicionar um item para a cesta de compras, você chama de StateHasChanged. Este método diz ao Blazor que ele deve atualizar a IU com novos dados.

Página 96

Listagem 3-21. Chamando StateHasChanged em AddToBasket

```
private void AddToBasket (Pizza pizza)
{
   Console.WriteLine ($ "Adicionou pizza {pizza.Name}");
   State.Basket.Add (pizza.Id);
   StateHasChanged ();
}
```

Corra e peça algumas pizzas. Funciona!

Pense sobre isso. Os componentes se processam novamente após os eventos, mas apenas eles mesmos. Quando um componente faz uma alteração que afeta outros componentes, você precisa para chamar StateHasChanged nos componentes afetados.

Mostrando o componente ShoppingBasket

Adicione um novo componente chamado ShoppingBasket à pasta Pages e mude seu conteúdo da Listagem 3-22.

Listagem 3-22. O componente ShoppingBasket

```
@using PizzaPlace.Shared

@if (Basket.Orders.Any ())
{
   <h1> @Title </h1>
```

```
@foreach (var (pizza, pos) em Pizzas)
{
    <PizzaItem Pizza = "@ pizza" ButtonClass = "btn btn-perigo"
               ButtonTitle = "Remover"
               Selecionado = "@ (p => Selecionado (pos))" />
}

<div class = "row">
    <div class = "col"> Total: </div>
    <div class = "col"> @TotalPrice </div>
    <div class = "col"> </div>
```


Página 97


```
        <div class = "col"> </div>
    </div>
}

@funções {

[Parâmetro]
string protegida Título {get; definir; }

[Parâmetro]
Cesta protegida Cesta {get; definir; }

[Parâmetro]
protegido Func <int, Pizza> GetPizzaFromId {get; definir; }

[Parâmetro]
ação protegida <int> Selecionada {get; definir; }

protegido IEnumerable <(Pizza pizza, int pos)> Pizzas {get; definir; }

TotalPrice decimal protegido {get; definir; }

override protegido void OnParametersSet ()
{
    base.OnParametersSet ();
    Pizzas = Basket.Orders.Select ((id, pos) => (pizza: GetPizzaFromId (id),
    pos: pos));
    TotalPrice = Pizzas.Select (tuple => tuple.pizza.Price) .Soma ();
}
}
```

O componente ShoppingBasket é semelhante ao componente PizzaList, mas existem algumas grandes diferenças. A classe basket mantém o controle do pedido usando apenas ids de pizzas, então você precisa de algo para obter o objeto pizza. Isso é feito por meio de o delegado GetPizzaFromId. Outra mudança é o método OnParametersSet. O O método OnParametersSet é chamado quando os parâmetros do componente são definidos. Aqui você o substitui para construir uma lista de tuplas (pizza, posição) que você precisa durante os dados ligação e para calcular o preço total da encomenda.

Página 98

Tuplas são apenas outro tipo em C #. Mas o C # moderno oferece uma sintaxe muito conveniente; por exemplo, IEnumerable <(Pizza pizza, int post)> significa que você tem um tipo que é um lista de pizza e pares de posição.

Usar o componente ShoppingBasket no Index é fácil, como você pode ver na Listagem 3-23 .

Listagem 3-23. Usando o componente ShoppingBasket

```
<! - Carrinho de compras ->

<ShoppingBasket Title = "Seu pedido atual"
                Basket = "@ State.Basket"
                GetPizzaFromId = "@ State.Menu.GetPizza"
                Selecionado = "@ (pos => RemoveFromBasket (pos))" />

<! - Fim da cesta de compras ->
```

Criação de uma biblioteca de componentes de validação

A terceira seção do componente Índice trata de inserir e validar detalhes sobre o consumidor. Você poderia dizer que a validação é uma coisa muito comum, mas não há na validação no Blazor, então você criará uma biblioteca de componentes para validação e, em seguida, usará para construir o componente CustomerEntry. A classe Customer já implementa a interface INotifyDataErrorInfo, portanto, essa parte não precisa ser alterada.

Abra um prompt de comando para a pasta que contém sua solução. Modelo

```
dotnet new blazorlib -o PizzaPlace.Extensions.Validation
```

Isso cria um novo projeto de biblioteca Blazor.

Então digite

```
dotnet sln add PizzaPlace.Extensions.Validation \
PizzaPlace.Extensions.Validation.csproj
```

Isso adiciona o novo projeto à sua solução.

Volte para o Visual Studio e clique em Recarregar. (Não há necessidade de fazer isso com o Código.) clique no projeto PizzaPlace.Extentions.Validation e adicione um novo Razor View chamado ValidationError e completo é como na Listagem 3-24.

Página 99

Listagem 3-24. O componente ValidationError

```
@using Microsoft.AspNetCore.Blazor
@using System.ComponentModel

@if (Errors.Any ())
{
    <ul class = "validation-error">
        @foreach (erro de string em erros)
        {
            <li> @error </li>
        }
    </ul>
}

@funções {

[Parâmetro]
objeto protegido Assunto {get; definir; }

[Parâmetro]
string protegida Propriedade {get; definir; }
```

```
    public IEnumerable <string> Erros
    {
        obter
        {
            switch (assunto)
            {
                caso nulo:
                    yield return $ "{nameof (Subject)} não foi definido!";
                    quebra de rendimento;
                case INotifyDataErrorInfo ine:
                    if (Propriedade == null)
                    {
                        yield return $ "{nameof (Property)} não foi definido!";
                        quebra de rendimento;
                    }
```


Página 100


```
                foreach (var err in ine.GetErrors (Property))
                {
                    rendimento return (string) err;
                }
                pausa;
            case IDataErrorInfo ide:
                if (Propriedade == null)
                {
                    yield return $ "{nameof (Property)} não foi definido!";
                    quebra de rendimento;
                }
                string error = ide [propriedade];
                if (erro! = nulo)
                {
                    erro de retorno de rendimento;
                }
                senão
                {
                    quebra de rendimento;
                }
                pausa;
        }
    }
}
}
```

Você espera que os parâmetros Assunto e Propriedade sejam definidos para um objeto implementar a interface IDataErrorInfo ou INotifyDataErrorInfo.
Você usa isso para construir dinamicamente a coleção Error, que é então usada para listar quaisquer erros de validação.

Lembre-se do estilo de erro de validação que você adicionou no capítulo anterior para mudar a cor dos erros de validação? Mova este CSS para o arquivo /content/styles.css do biblioteca de componentes. Isso conclui a biblioteca de componentes de validação.

Página 101

Adicionando o Componente CustomerEntry

Adicione uma referência à biblioteca PizzaPlace.Extensions.Validation, como você viu anteriormente neste capítulo. Agora adicione uma nova visão do Razor chamada CustomerEntry para a pasta Pages, como mostrado na Listagem 3-25.

Listagem 3-25. O Componente CustomerEntry

```
@using PizzaPlace.Shared

@addTagHelper *, PizzaPlace.Extensions.Validation

<h1> @Title </h1>

<fieldset>
    <p>
        <label for = "name"> Nome: </label>
        <input id = "name" bind = "@ Customer.Name" />
        <ValidationError Subject = "@ Customer"
                         Property = "@ nameof (Customer.Name)" />
    </p>
    <p>
        <label for = "street"> Rua: </label>
        <input id = "street" bind = "@ Customer.Street" />
        <ValidationError Subject = "@ Customer"
                         Property = "@ nameof (Customer.Street)" />
    </p>
    <p>
        <label for = "city"> Cidade: </label>
        <input id = "city" bind = "@ Customer.City" />
        <ValidationError Subject = "@ Customer"
                         Propriedade = "@ nameof (Customer.City)" />
    </p>

    <button onclick = "@ (() => Enviar (cliente))"
            disabled = "@ Customer.HasErrors"> Check-out </button>
</fieldset>

@funções {
```


Página 102


```
[Parâmetro]
string protegida Título {get; definir; }

[Parâmetro]
string protegida Título {get; definir; }

[Parâmetro]
Cliente protegido Cliente {get; definir; }

[Parâmetro]
ação protegida <Customer> Enviar {get; definir; }

[Parâmetro]
ação protegida <Customer> Enviar {get; definir; }
}
```

O componente CustomerEntry usa um rótulo e um elemento de entrada para cada cliente propriedade. Você também usa um componente ValidationError de sua biblioteca recém-construída para exibir quaisquer erros de validação. Agora você está pronto para completar o índice com este último

componente. Listagem3-26 mostra todo o Index.cshtml

Listagem 3-26. O Componente de Índice

```
@página "/"
<! - Menu ->
<PizzaList Title = "Nossa lista selecionada de pizzas"
                    Menu = "@ State.Menu"
                    Selecionado = "@ ((pizza) => AddToBasket (pizza))" />
<! - Menu final ->
<! - Carrinho de compras ->
<ShoppingBasket Title = "Seu pedido atual"
                            Basket = "@ State.Basket"
                            GetPizzaFromId = "@ State.Menu.GetPizza"
                            Selecionado = "@ (pos => RemoveFromBasket (pos))" />
<! - Fim da cesta de compras ->
<! - Entrada do cliente ->
<CustomerEntry Title = "Por favor, insira seus dados abaixo"
                           bind-Customer = "@ State.Basket.Customer"
```


Página 103


```
                            Enviar = "@ ((_) => PlaceOrder ())" />
<! - Finalizar entrada do cliente ->

@funções {

estado privado Estado {get; } = novo estado ()
{
    Menu = novo Menu
    {
        Pizzas = nova lista <Pizza>
{
Pizza nova (1, "Pepperoni", 8,99M, Spiciness.Spicy),
pizza nova (2, "Margarita", 7,99 milhões, tempero.Nenhuma),
pizza nova (3, "Diabolo", 9,99 milhões, especiarias.quente)
}
    }
};

private void AddToBasket (Pizza pizza)
{
    Console.WriteLine ($ "Adicionou pizza {pizza.Name}");
    State.Basket.Add (pizza.Id);
    StateHasChanged ();
}

private void RemoveFromBasket (int pos)
{
    Console.WriteLine ($ "Removendo pizza na pos {pos}");
    State.Basket.RemoveAt (pos);
    StateHasChanged ();
}

private void PlaceOrder ()
{
    Console.WriteLine ($ "Ordem de colocação {State.Basket.Customer.ToJson ()}");
}
}
```

Página 104

Crie e execute o aplicativo PizzaPlace. As coisas devem funcionar como antes, exceto por uma Coisa. Lembra da dica de depuração do capítulo anterior? Quando você muda o nome do cliente, essa dica não atualiza corretamente. Vamos consertar isso. O problema é da seguinte forma: quando você altera o nome do cliente, o componente CustomerEntry muda o nome do cliente, mas o componente Índice não vê essa mudança.
Você pode corrigir isso registrando-se para alterações no cliente. Para se registrar para mudanças em objetos .NET tem a interface INotifyPropertyChanged, que faz parte do .NET desde o .NET 2.0, então você deve estar familiarizado com ele. Esta interface é mostrada na Listagem3-27 e tem apenas o evento PropertyChanged.

Listagem 3-27. A interface INotifyPropertyChanged

```
namespace System.ComponentModel
{
    interface pública INotifyPropertyChanged
    {
        evento PropertyChangedEventHandler PropertyChanged;
    }
}
```

Sempre que uma propriedade do cliente muda, ele deve acionar este evento. Faça o A classe do cliente implementa esta interface, como na Listagem 3-28.

Listagem 3-28. A classe Customer implementa INotifyPropertyChanged

```
public class Customer: INotifyDataErrorInfo,
                                INotifyPropertyChanged
```

Agora altere a propriedade Nome da classe do cliente, conforme mostrado na Listagem 3-29 .

Listagem 3-29. A propriedade do nome da classe do cliente

```
usando System.Runtime.CompilerServices;
...
nome da string privada;
```


Página 105


```
public string Name
{
    get {return name; }
    definir {nome = valor; OnPropertyChanged (); }
}
```

Sempre que a propriedade é modificada, você dispara o evento PropertyChanged usando o Método OnPropertyChanged da Listagem 3-30.

Listagem 3-30. O método OnPropertyChanged

```
private void OnPropertyChanged (
                [CallerMemberName] string propertyName = "")
```

```
{
    PropertyChanged? .Invoke (this,
        novo PropertyChangedEventArgs (propertyName));
}
```

Vou explicar um pouco a implementação. Você deve passar o nome do imóvel no Evento PropertyChanged. Você pode passar esse nome como uma string para o OnPropertyChanged método, mas quando você altera o nome da propriedade, há uma grande chance de você esqueça de atualizar esta string. É melhor não passar nada e fazer com que o compilador descubra O nome da propriedade. Isso pode ser feito usando o atributo CallerMemberName, que fará com que o compilador descubra o nome do chamador, neste caso o nome do propriedade!

> Que tipo de código é o código mais fácil de manter e sem bugs que você pode escrever? Código que você não escreveu!

Implemente as propriedades Street e City da mesma maneira.

Quase lá. Abra Index.cshtml. Adicione um método OnInit como na Listagem 3-31. Esta método registra mudanças no cliente e chama StateHasChanged, que irá atualizar a IU. Dessa forma, você não precisa se preocupar em chamar StateHasChanged quando um A propriedade do cliente é modificada.

Página 106

Listagem 3-31. O Método OnInit

```
override protegido void OnInit () {
    this.State.Basket.Customer.PropertyChanged + =
      (remetente, e) => this.StateHasChanged ();
}
```

Construa e execute. Agora, quando você faz uma alteração para o cliente, a dica de depuração atualizará. Você pode pensar: "E daí?" O cliente também pode ser usado por outro componente que precisa ver mudanças.

Ganchos de ciclo de vida de componentes

Cada componente tem alguns métodos que você pode substituir para capturar o ciclo de vida de o componente. Nesta seção, você verá esses ganchos de ciclo de vida porque é muito importante entendê-los muito bem. Colocar o código no gancho de ciclo de vida errado irá provavelmente quebrará seu componente.

OnInit e OnInitAsync

Quando seu componente tiver sido completamente inicializado, o OnInit e OnInitAsync métodos são chamados. Implemente um desses métodos se quiser fazer algo extra inicialização após o componente ter sido criado, como buscar alguns dados de um servidor, como o componente FetchData do projeto MyFirstBlazor.

Use OnInit para código síncrono, conforme mostrado na Listagem 3-32.

Listagem 3-32. O gancho do ciclo de vida OnInit

```
sobrescrito protegido void OnInit ()
{

}
```

Use OnOnitAsync (Listagem 3-33) para chamar métodos assíncronos, por exemplo, tornando

Chamadas REST (você verá como fazer chamadas REST nos próximos dois capítulos).

Página 107

Listagem 3-33. O gancho de ciclo de vida OnInitAsync

```
substituição protegida assíncrona Tarefa OnInitAsync ()
{

}
```

OnParametersSet e OnParametersSetAsync

Quando você precisar de um ou mais parâmetros para inicialização, use OnParametersSet ou OnParametersSetAsync em vez dos métodos OnInit / OnInitAsync. Esses métodos obtêm chamado depois que o componente foi inicializado e depois que os parâmetros foram dados-limite. Por exemplo, você pode ter um componente DepartmentSelector que permite o usuário para selecionar um departamento de uma empresa, e outro componente EmployeeList que toma o departamento selecionado como parâmetro. O componente EmployeeList pode então buscar os funcionários desse departamento em seu método OnParametersSetAsync.

Use OnParametersSet (Listagem 3-34) se você estiver apenas chamando métodos síncronos.

Listagem 3-34. O método OnParametersSet

```
override protegido void OnParametersSet ()
{

}
```

Use OnParametersSetAsync (Listagem 3-35) se você precisar chamar métodos assíncronos.

Listagem 3-35. O método OnParametersSetAsync

```
sobrescrita protegida assíncrona Tarefa OnParametersSetAsync ()
{

}
```

OnAfterRender e OnAfterRenderAsync

Os métodos OnAfterRender e OnAfterRenderAsync são chamados após o Blazor ter renderizou completamente o componente. Isso significa que o DOM do navegador foi atualizado com as alterações feitas em seu componente Blazor. Você pode usar esses métodos para invocar o código JavaScript que precisa de acesso a elementos do DOM (que abordaremos no capítulo sobre JavaScript).

Página 108

Use OnAfterRender (Listagem 3-36) para chamar métodos síncronos, por exemplo em JavaScript.

Listagem 3-36. O gancho de ciclo de vida OnAfterRender

```
override protegido void OnAfterRender ()
{

}
```

Use OnAfterRenderAsync (Listagem 3-37) para chamar métodos assíncronos, por exemplo Métodos JavaScript que retornam promessas.

Listagem 3-37. O gancho de ciclo de vida OnAfterRenderAsync

```
sobrescrita protegida assíncrona Tarefa OnAfterRenderAsync ()
{

}
```

Descartável

Se você precisar executar algum código de limpeza quando seu componente for removido da IU, implementar IDisposable. Você pode implementar esta interface no Razor usando @implements, como mostrado na lista 3-38 . Normalmente você coloca @implements no topo do arquivo CSHTML.

> na maioria das vezes, a injeção de dependência cuidará de chamar Dispose, então geralmente você não precisa implementar IDisposable se você só precisa descartar suas dependências.

Listagem 3-38. Implementando a interface IDisposable em um componente

```
@implements IDisposable
```

> a interface IDisposable requer que você implemente um método Dispose, que você coloca em funções @, como na listagem 3-39.

Página 109

Listagem 3-39. Implementando o Método de Dispose

```
@funções {

    public void Dispose ()
    {
        // Recursos de limpeza aqui
    }
}
```

Se você separou a visualização e o modelo de visualização, você implementa esta interface no modelo de visualização.

Usando componentes modelados

Os componentes são os blocos de construção do Blazor para reutilização. Blazor também suporta *modelos componentes* onde você pode especificar um ou mais modelos de IU como parâmetros, tornando componentes modelados ainda mais reutilizáveis! Por exemplo, seu aplicativo pode ser usando grades em todo o lugar. Agora você pode construir um componente com modelo para uma grade tomando o tipo usado na grade como um parâmetro (da mesma forma que você pode construir um digite .NET) e especifique a IU usada para cada item separadamente! Vejamos um exemplo.

Criando o Componente com Modelo de Grade

Abra o projeto MyFirstBlazor que você está usando. Agora adicione um novo componente (um Visualização do Razor) para a pasta Pages do projeto MyFirstBlazor.Client e denomine Grid como na Listagem 3-40 . Este é um componente modelado porque declara o TItem como um tipo parâmetro usando a sintaxe @typeparam TItem. É como um tipo genérico declarado em C # com a classe pública List <T> onde T é um parâmetro de tipo.

Você pode ter mais de um parâmetro de tipo. simplesmente liste cada parâmetro de tipo usando a sintaxe @typeparam.

89

Página 110

Capítulo 3 Componentes e estrutura para aplicativos Blazor

Listagem 3-40. O Componente com Modelo de Grade

```
@typeparam TItem

<table border = "1">
    <thead>
        <tr> @Header </tr>
    </thead>
    <tbody>
        @foreach (item var em itens)
        {
            <tr> @Row (item) </tr>
        }
    </tbody>
    <tfoot>
        <tr> @Footer </tr>
    </tfoot>
</table>

@funções {
[Parâmetro]
RenderFragment Header {get; definir; }

[Parâmetro]
RenderFragment <TItem> Row {get; definir; }

[Parâmetro]
RenderFragment Footer {get; definir; }

[Parâmetro]
IReadOnlyList <TItem> Itens {get; definir; }
}
```

90

Página 111

Capítulo 3 Componentes e estrutura para aplicativos Blazor

O componente Grid possui quatro parâmetros. Os parâmetros Cabeçalho e Rodapé são do tipo RenderFragment, que representa algum HTML que você pode especificar quando você usa o componente Grid (você verá um exemplo logo após explorar o

Componente de grade mais adiante). Procure o elemento <thead> na Listagem 3-40 no Grid componente. Aqui você usa a sintaxe do @Header razor para dizer ao componente Grid para colocar o HTML para o parâmetro Header aqui (a mesma coisa para o rodapé).

O parâmetro Row é do tipo RenderFragment <TItem>, que é uma versão genérica do RenderFragment. Neste caso, você pode especificar HTML com acesso ao TItem permitindo você acessa as propriedades e métodos do TItem. O parâmetro Items é um IReadOnlyList <TItem> que pode ser vinculado por dados a qualquer classe com IReadOnlyList <TItem> interface. Procure o elemento <tbody> na Listagem3-40 . Você itera sobre todos os itens (de digite TItem) do <TItem> IReadOnlyList e use a sintaxe do Razor @Row (elemento) para aplicar o parâmetro Row, passando o item atual como um argumento.

Usando o componente de modelo de grade

Agora vamos dar uma olhada em um exemplo de uso do componente modelado Grid. Abra o Componente FetchData.cshtml no projeto MyFirstBlazor.Client. Substitua o <table> (comente a <table> porque você voltará a ela em capítulos posteriores) com o componente de grade na listagem 3-41 .

> o componente FetchData usa algumas coisas como @page e @inject.
> Vou discuti-los em capítulos posteriores, portanto, continue com o exemplo.

O componente FetchData usa o componente Grid, especificando o parâmetro Items como a matriz de previsões de instâncias do WeatherForecast. O compilador é inteligente o suficiente para inferir disso que o parâmetro de tipo de Grid (TItem) é o tipo WeatherForecast.

Listagem 3-41. O componente FetchData

```
@using MyFirstBlazor.Shared
@page "/ fetchdata"
@inject HttpClient Http
```


Página 112


```
<h1> Previsão do tempo </h1>

<p> Este componente demonstra a obtenção de dados do servidor. </p>

@if (previsões == null)
{
    <p><em>Carregando...</em> </p>
}
senão
{
    <Grid Items = "@ Forecast">
        <Header>
            <th> Data </th>
            <th> Temp (Celsius) </th>
            <th> Resumo </th>
        </Header>
        <Row Context = "previsão">
            <! - por padrão, chamado de contexto ->
            <td> @ forecast.Date </td>
            <td> @ forecast.TemperatureC </td>
            <td> @ forecast.Summary </td>
        </Row>
        <Footer>
            <td colspan = "3"> A primavera está no ar! </td>
        </Footer>
```

```
    </Grid>

    @ * <table class = "table">
    ...
    </table> * @
}

@funções {
Previsões do WeatherForecast [];
```

92

Página 113


```
substituição protegida assíncrona Tarefa OnInitAsync ()
{
    previsões = aguardar Http.GetJsonAsync <WeatherForecast []> ("api / SampleData /
Previsões do tempo");
}
}
```

Agora olhe para o parâmetro <Header> do componente Grid na Listagem 3-41 . Esta A sintaxe vincula tudo o que está dentro de <Header> ao parâmetro Grid's Header. Nisso exemplo, você especifica alguns cabeçalhos de tabela. O Grid os coloca dentro do elemento <tr> da lista 3-40 . Novamente, o <Footer> é semelhante.

Examine o parâmetro <Row> na Listagem 3-41 . Dentro da <Row> você deseja usar o item atual da iteração na Listagem 3-40. Mas como você deve acessar o atual item? Por padrão, o Blazor passará o item como o argumento de contexto (do tipo TItem), então você acessa a data da instância de previsão como @ context.Date. Mas você pode substituir o nome do argumento, e isso é o que você faz com os parâmetros de contexto (fornecidos por Blazor) usando <Row Context = "forecast">. Agora, o item da iteração pode ser acessado usando o argumento de previsão.

Execute sua solução e selecione o link Buscar dados no menu de navegação. Admirar seu novo componente com modelo, mostrado na Figura 3-11!

Figura 3-11. *Mostrando previsões com o componente modelado de grade*

93

Página 114

Agora você tem um componente Grid reutilizável que pode ser usado para mostrar qualquer lista de itens passando a lista para os parâmetros de itens e especificando o que deve ser mostrado no Parâmetros de cabeçalho, linha e rodapé! Mas tem mais!

Especificando o tipo do parâmetro de tipo explicitamente

Normalmente, o compilador pode inferir o tipo do parâmetro de tipo, mas se isso não funcionar como você espera, você pode especificar o tipo explicitamente. Basta especificar o tipo do seu tipo parâmetro especificando-o ao usar o componente, conforme mostrado na Listagem 3-42.

Listagem 3-42. Especificando explicitamente o parâmetro de tipo

```
<Grid Items = "@ Forecast" TItem = "WeatherForecast">
```

Modelos de navalha

Você também pode especificar um RenderFragment ou RenderFragment <TItem> usando a sintaxe do Razor. Um *modelo Razor* é uma forma de definir um snippet de IU, por exemplo @ <Row> ... </Row>. Nisso caso, você especifica um RenderFragment sem nenhum argumento. Mas se você precisar passar no argumento para o modelo Razor, você usa uma função lambda. Vejamos um exemplo.
Comece adicionando um novo componente chamado ListView, conforme mostrado na Listagem 3-43. Isso vai mostra uma lista não ordenada de itens (do tipo TItem) usando <ul> e <li> elementos HTML.

Listagem 3-43. O Componente ListView Templated

```
@typeparam TItem

<ul>
    @foreach (item var em itens)
    {
        @ItemTemplate (item)
    }
</ul>

@funções {
[Parâmetro] RenderFragment <TItem> ItemTemplate {get; definir; }
[Parâmetro] IReadOnlyList <TItem> Itens {get; definir; }
}
```


Página 115

Agora atualize o componente FetchData como na Listagem 3-44. Aqui você especifica o <ListView> do <ItemTemplate>, que é do tipo RenderFragment <TItem>, usando um Razor modelo. Olhe para o forecastTemplate na listagem3-44 . Ele usa uma função lambda, tomando a previsão como um argumento, que retorna um RenderFragment <TItem> usando a sintaxe @ ... Razor. No <ItemTemplate> do componente <ListView>, você simplesmente invoque a função lambda.

Listagem 3-44. Usando modelos do Razor para especificar o fragmento de renderização

```
@using MyFirstBlazor.Shared
@page "/ fetchdata"
@inject HttpClient Http

@ {
    RenderFragment <WeatherForecast> forecastTemplate =
        (previsão) => @ <li> @ forecast.Date.ToLongDateString () - @forecast.
        Resumo </li>;
}

<h1> Previsão do tempo </h1>
```

```
<p> Este componente demonstra a obtenção de dados do servidor. </p>
@if (previsões == null)
{
    <p><em>Carregando...</em> </p>
}
senão
{
    <Grid Items = "@ Forecast" TItem = "WeatherForecast">
    ...
    </Grid>

    <ListView Items = "@ Forecast" TItem = "WeatherForecast">
        <ItemTemplate>
            @forecastTemplate (contexto)
        </ItemTemplate>
    </ListView>

}
```

95

Página 116


```
@funções {
Previsões do WeatherForecast [];

substituição protegida assíncrona Tarefa OnInitAsync ()
{
    previsões = aguardar Http.GetJsonAsync <WeatherForecast []> ("api / SampleData /
Previsões do tempo");
}
}
```

Os modelos do Razor são uma ótima maneira de reutilizar um trecho de IU porque você pode invocá-lo em componentes diferentes.

Você também pode chamar um modelo Razor diretamente em seu componente como na Listagem 3-45.

Listagem 3-45. Invocando um modelo do Razor em seu componente

```
@forecastTemplate (novo WeatherForecast {
                        Date = DateTime.Now,
                        TemperaturaC = 26,
                        Resumo = "Legal!"
                    })
```

O modelo de compilação do Blazor

Cada arquivo Razor (CSHTML) é compilado em código C # e é muito interessante ter um olhe para eles. Esses arquivos são gerados na subpasta obj do seu projeto, e você pode observe esses arquivos gerados no Visual Studio. Selecione o projeto PizzaPlace.Client em Solution Explorer e clique no botão Show All Files mostrado na Figura 3-12 .

Figura 3-12. *O botão Mostrar Todos os Arquivos*

Agora abra a pasta obj / Debug / netstandard2.0 / Pages no Solution Explorer. Abrir PizzaItem.g.cs, que você pode encontrar na Listagem 3-46 . (Eu deixei de fora alguns dos menos detalhes importantes.)

96

Página 117

Listagem 3-46. O arquivo gerado PizzaItem.g.cs

```
namespace PizzaPlace.Client.Pages
{
    public class PizzaItem: BlazorComponent
    {
        override protegido void BuildRenderTree (RenderTreeBuilder builder)
        {
            base.BuildRenderTree (construtor);
            builder.OpenElement (0, "div");
            builder.AddAttribute (1, "classe", "linha");
            builder.AddContent (2, "\ n");
            builder.OpenElement (3, "div");
            builder.AddAttribute (4, "classe", "col");
            builder.AddContent (5, "\ n");
            builder.AddContent (6, Pizza.Name);
            builder.AddContent (7, "\ n");
            builder.CloseElement ();
            builder.AddContent (8, "\ n");
            builder.OpenElement (9, "div");
            builder.AddAttribute (10, "classe", "col");
            builder.AddContent (11, "\ n");
            builder.AddContent (12, Pizza.Price);
            builder.AddContent (13, "\ n");
            builder.CloseElement ();
            builder.AddContent (14, "\ n");
            builder.OpenElement (15, "div");
            builder.AddAttribute (16, "classe", "col");
            builder.AddContent (17, "\ n");
            builder.OpenElement (18, "img");
            builder.AddAttribute (19, "src", SpicinessImage (Pizza.Spicyness));
            builder.AddAttribute (20, "alt", Pizza.Spicyness);
            builder.CloseElement ();
            builder.AddContent (21, "\ n");
            builder.CloseElement ();
```


Página 118


```
            builder.AddContent (22, "\ n");
            builder.OpenElement (23, "div");
            builder.AddAttribute (24, "classe", "col");
            builder.AddContent (25, "\ n");
            builder.OpenElement (26, "botão");
            builder.AddAttribute (27, "classe", ButtonClass);
            builder.AddAttribute (28, "onclick", Microsoft.AspNetCore.Blazor.
            Components.BindMethods.GetEventHandlerValue <Microsoft.AspNetCore.
            Blazor.UIMouseEventArgs> (() => Selecionado (Pizza)));
            builder.AddContent (29, ButtonTitle);
            builder.CloseElement ();
            builder.AddContent (30, "\ n");
            builder.CloseElement ();
            builder.AddContent (31, "\ n");
            builder.CloseElement ();
```

```
        }

    [Parâmetro]
    Pizza Pizza protegida {get; definir; }

    [Parâmetro]
    string protegida ButtonTitle {get; definir; }

    [Parâmetro]
    string protegida ButtonClass {get; definir; }

    [Parâmetro]
    ação protegida <Pizza> selecionada {obter; definir; }

    string privada SpicinessImage (Spiciness spiciness)
    => $ "images / {spiciness.ToString (). ToLower ()}. png";
    }
}
```

Como você pode ver, a maior parte do código gerado é o método BuildRenderTree. Este método cria elementos, atributos, conteúdo e manipuladores de eventos. Por exemplo, o o arquivo CSHTML original contém a Listagem 3-47, que é gerado como Listagem 3-48 .

98

Página 119

Listagem 3-47. The Original Razor

```
<div class = "col">
    @ Pizza.Name
</div>
```

Listagem 3-48. O código gerado pelo Razor

```
builder.OpenElement (3, "div");
builder.AddAttribute (4, "classe", "col");
builder.AddContent (5, "\ n");
builder.AddContent (6, Pizza.Name);
builder.AddContent (7, "\ n");
builder.CloseElement ();
```

Se você realmente quiser, pode herdar diretamente do BlazorComponent e substituir o BuildRenderTree método e gerar seu HTML personalizado diretamente aqui. Este é apenas interessante em alguns cenários muito avançados que não abordo neste livro.

Resumo

Neste capítulo, você explorou a construção de componentes do Blazor e bibliotecas de componentes. Vocês também aprendeu como os componentes podem se comunicar uns com os outros por meio de parâmetros e vinculação de dados. Você aplicou esse aprendizado dividindo o componente de índice monolítico do aplicativo PizzaPlace em componentes menores. Você também viu que no Blazor você pode construir componentes modelados, que se assemelham a classes genéricas. Estes modelos componentes podem ser parametrizados para renderizar interfaces de usuário diferentes, o que os torna bastante reutilizável! Finalmente, você deu uma olhada nos ganchos do ciclo de vida do componente (que você vai precisar em capítulos adicionais) e como os componentes do Razor são compilados no bom e velho código C #.

Página 120

CAPÍTULO 4

Serviços e Dependência Injeção

A inversão de dependência é um dos princípios básicos de um *bom design orientado a objetos* . O o grande facilitador é a *injeção de dependência* . Neste capítulo, você examinará a dependência inversão e injeção e porque são peças fundamentais da Blazer. Você vai explorar construindo um serviço que encapsula onde os dados são recuperados e armazenados.

O que é inversão de dependência?

Atualmente, seu aplicativo Blazor PizzaPlace recupera seus dados de dados de amostra embutidos em código. Mas em uma situação da vida real, esses dados serão armazenados em um banco de dados no servidor. Recuperando e o armazenamento desses dados pode ser feito no próprio componente, mas isso é uma má ideia. Por quê? Porque a tecnologia muda com bastante frequência, e diferentes clientes de seu aplicativo podem querer usar sua própria tecnologia específica, exigindo que você atualize seu aplicativo para cada cliente.

Em vez disso, você colocará essa lógica em um *objeto de serviço* . A função de um objeto de serviço é encapsular regras de negócios específicas, especialmente como os dados são comunicados entre o cliente e o servidor. Um objeto de serviço também é muito mais fácil de testar, pois você pode escrever unidades testes que são executados por conta própria, sem exigir que o usuário interaja com o aplicativo para testando.

Mas primeiro, vamos falar sobre o princípio de inversão de dependência e como a dependência injeção nos permite aplicar este princípio.

P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_4

Página 121

Compreendendo a inversão de dependência

Imagine um componente que usa um serviço. O componente cria o serviço usando o novo operador, como na Listagem 4-1 .

Listagem 4-1. Um componente que usa um ProductsService

```
@using MyFirstBlazor.Client.Services

<div>
    @foreach (var product in productsService.GetAllProducts ())
    {
```

```
        <div> @ product.Name </div>
        <div> @ product.Description </div>
        <div> @ product.UnitPrice </div>
    }
</div>

@funções {

    ProductsService productsService = new ProductsService ();

}
```

Este componente agora é totalmente dependente do ProductsService! Você não pode substitua o ProductsService sem percorrer cada linha de código em seu aplicativo onde o ProductsService é usado e substituindo-o por outra classe. Isto é também conhecido como *acoplamento apertado* (ver Figura 4-1)

Lista de produtos ProductService

Figura 4-1. *Abordagem tradicional em camadas com forte acoplamento*

Agora você deseja testar o componente ProductList. ProductsService requer um servidor na rede para conversar. Neste caso, você deve configurar um servidor apenas para executar o teste. E se o servidor ainda não estiver pronto (o desenvolvedor encarregado do servidor não veio em torno dele), você não pode testar seu componente! Ou digamos que você esteja usando ProductsService em

Página 122

vários lugares em sua localização, e você precisa substituir o ProductsService por outro aula. Agora você precisa encontrar todos os usos do ProductsService e substituir a classe. Pesadelo de manutenção!

Usando o Princípio de Inversão de Dependência

Os estados do *Princípio de Inversão de Dependência*

A. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

B. As abstrações não devem depender de detalhes. Os detalhes devem depender em abstrações.

O que isso significa é que o componente ProductsList (o módulo de nível superior) não deve depender diretamente do ProductsService (o módulo de nível inferior). Em vez de, deve contar com uma abstração. Deve contar com uma *interface* que descreve o que O ProductsService deve ser capaz de fazer, não uma classe que descreve como deve funcionar.

A interface IProductsService se parece com a Listagem 4-2.

Listagem 4-2. A abstração conforme descrita em uma interface

```
interface pública IProductsService
{
    Listar <Product> GetAllProducts ();
}
```

Portanto, altere o componente ProductsList para confiar nesta abstração mostrada na Listagem 4-3 .

Listagem 4-3. O componente ProductList usando a interface IProductsService

```
@using MyFirstBlazor.Client.Services

<div>
    @foreach (var product in productsService.GetAllProducts ())
```

```
{
    <div> @ product.Name </div>
    <div> @ product.Description </div>
    <div> @ product.UnitPrice </div>
}
```


Página 123


```
</div>

@funções {

    IProductsService productsService;

}
```

Agora, o componente ProductList depende apenas da interface IProductsService, um abstração. Claro, agora você faz o ProductsService implementar a interface como na Listagem 4-4 .

Listagem 4-4. O ProductsService que implementa a interface IProductsService

```
public class ProductsService: IProductsService
{
    Lista pública <Product> GetAllProducts ()
    {
        // alguma implementação
    }

}
```

Se você quiser testar o componente ProductList com inversão de dependência no lugar, você pode simplesmente construir uma versão codificada do ProductsService e executar o teste sem precisando de um servidor, como na Listagem 4-5. E se você usa ProductsService em lugares diferentes em seu aplicativo, tudo que você precisa fazer para substituir sua implementação é construir outro classe que implementa a interface IProductsService e informa a injeção de dependência para use a outra classe! Isso também é conhecido como o princípio aberto / fechado de SOLID.

Listagem 4-5. Um serviço de produtos codificados permanentemente usado para teste

```
public class HardCodedProductsService: IProductsService
{
    Lista pública estática <Produto> produtos = nova Lista <Produto>
    {
        novo produto {
            Id = 1,
            Nome = "Isabelle's Caseiro Marmelade",
```


Página 124


```
            Descrição = "...",
            Preço Unitário = 1,99M}
    };

    Lista pública <Product> GetAllProducts ()
    {
        devolver produtos;
```

```
    }
}
```

Aplicando o *Princípio de Inversão de Dependência* (ver Figura 4-2), você ganhou muito mais flexibilidade.

Figura 4-2. *Objetos fracamente acoplados por meio de inversão de dependência*

Adicionando injeção de dependência

Se você executar seu aplicativo, agora obterá uma NullReferenceException. Por quê? Porque o componente ProductsList ainda precisa de uma instância de uma implementação de classe IProductsService! Você poderia passar o ProductsService no construtor do Componente ProductList, por exemplo na Listagem 4-6 .

Página 125

Listagem 4-6. Passando pelo ProductService no Construtor

```
new ProductList (new ProductService ())
```

Mas se o ProductsService também depende de outra classe, rapidamente se torna como Listagem 4-7.

Listagem 4-7. Criando uma Cadeia Profunda de Dependências Manualmente

```
new ProductList (new ProductService (nova Dependência ()))
```

Obviamente, esta não é uma forma prática de trabalhar! Por causa disso, você usará *um Recipiente de inversão de controle* (não fui eu que inventei esse nome!).

Aplicando um Container de Inversão de Controle

Um Container de Inversão de Controle (IoCC) é apenas outro objeto especializado em criando objetos para você. Você simplesmente pede a ele para criar uma instância de uma classe e levará cuidado ao criar quaisquer dependências necessárias.

É um pouco como em um filme quando um cirurgião, no meio de uma operação, precisa um bisturi. O cirurgião estende a mão e pergunta "bisturi número 5". O enfermeira (o Recipiente de Inversão de Controle) que está ajudando simplesmente entrega o cirurgião o bisturi. O cirurgião não se importa de onde vem o bisturi ou como foi construído.

Portanto, como o IoCC pode saber de quais dependências seu componente precisa? Existem dois caminhos.

Injeção de dependência de construtor

As classes que precisam de uma dependência podem simplesmente declarar suas dependências em seu construtor.

O IoCC examinará o construtor e instanciará as dependências antes de chamar o construtor. E se essas dependências tiverem suas próprias dependências, o IoCC também irá construí-los! Por exemplo, se o ProductsService tem um construtor que leva um argumento do tipo Dependency, como na Listagem 4-8, então o IoCC criará uma instância do tipo Dependency e, em seguida, chamará o construtor do ProductsService com esse instância. O construtor ProductsService então armazena uma referência à dependência em algum campo, como na Listagem 4-8 . O construtor do ProductsService deve levar vários argumentos, o IoCC criará uma instância para cada argumento. Construtor injeção é normalmente usada para dependências *necessárias* .

Página 126

Listagem 4-8. O construtor do ProductsService com argumentos

```
public class ProductsService {
    dependência privada dep;

    public ProductsService (Dependency dep) {
        this.dep = dep;
    }
}
```

Injeção de Dependência de Propriedade

Se a classe que o IoCC precisa construir tem propriedades que indicam uma dependência, então essas propriedades são preenchidas pelo IoCC. A maneira como uma propriedade faz isso depende do IoCC (no .NET existem alguns frameworks IoCC diferentes), mas no Blazor você pode faça com que o IoCC injete uma instância com a diretiva @inject em seu arquivo Razor, como no terceira linha de código na listagem 4-9.

Listagem 4-9. Injetando uma dependência com a diretiva @inject

```
@using MyFirstBlazor.Client.Services

@inject IProductsService productsService

<div>
    @foreach (var product in productsService.GetAllProducts ())
    {
        <div> @ product.Name </div>
        <div> @ product.Description </div>
        <div> @ product.UnitPrice </div>
    }
</div>

@funções {


}
```

Se estiver usando separação de código, você pode adicionar uma propriedade à sua classe e aplicar o Atributo [Injetar] como na Listagem 4-10.

Página 127

Listagem 4-10. Usando o atributo de injeção para injeção de propriedade

```
using System;
usando Microsoft.AspNetCore.Blazor.Components;
using MyFirstBlazor.Client.Services;

namespace MyFirstBlazor.Client.Pages
{
    public class ProductListViewModel: BlazorComponent
    {
        [Injetar]
        public IProductsService ProductsService {get; definir; }
    }
}
```

Você pode então usar esta propriedade diretamente em seu arquivo Razor, como na Listagem 4-11.

Listagem 4-11. Usando a propriedade ProductsService que foi injetada pela dependência

```
@inherits ProductListViewModel

<div>
@foreach (var product in ProductsService.GetAllProducts ())
{
    <div> @ product.Name </div>
    <div> @ product.Description </div>
    <div> @ product.UnitPrice </div>
}
</div>
```

Configurando injeção de dependência

Há mais uma coisa que preciso discutir. Quando sua dependência é uma classe, o IoCC pode saber facilmente que precisa criar uma instância da classe com o construtor da classe. Mas se sua dependência é uma interface, o que geralmente precisa ser se você estiver aplicando o Princípio de Inversão de Dependência, então qual classe ele usa para criar o instância? Sem a sua ajuda, ele não pode saber.

Página 128

Um IoCC possui um mapeamento entre interfaces e classes, e é seu trabalho configure este mapeamento. Você configura o mapeamento no Startup do seu projeto Blazor classe, assim como no ASP.NET Core. Portanto, abra Startup.cs, como na Listagem 4-12.

Listagem 4-12. The Startup Class

```
usando Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace MyFirstBlazor.Client
{
    public class Startup
    {
        public void ConfigureServices (serviços IServiceCollection)
        {
            // Configure dependências aqui
        }

        public void Configure (aplicativo IBlazorApplicationBuilder)
        {
            app.AddComponent <App> ("app");
        }
    }
}
```

Veja o comentário? A ideia é que você configure o mapeamento a partir da interface para a classe aqui, e você usa métodos de extensão no serviceProvider. Que método de extensão que você chama da Figura 4-3 depende da vida útil que você deseja dar a dependência. Existem três opções para o tempo de vida de uma instância, que irei discutir a seguir.

Página 129

Figura 4-3. *Configurando injeção de dependência*

Dependências Singleton

Classes singleton são classes que possuem apenas uma instância. Eles são normalmente usados para gerenciar algum estado global; por exemplo, você pode ter uma classe que monitora como muitas vezes as pessoas clicaram em um determinado produto. Ter várias instâncias deste classe complicaria as coisas porque eles teriam que começar a se comunicar com uns aos outros para manter o controle dos cliques. As classes singleton também podem ser classes que não tem qualquer estado, que só tem comportamento (classes de utilidade, como aquela que faz conversões entre unidades imperiais e métricas). Você configura a injeção de dependência para reutilizar o mesma instância o tempo todo com o método de extensão AddSingleton, como na Listagem 4-13 .

Listagem 4-13. Adicionando um Singleton à injeção de dependência

```
public void ConfigureServices (serviços IServiceCollection)
{
    services.AddSingleton <IProductsService, ProductsService> ();
}
```


Página 130

Por que não usar métodos estáticos em vez de singletons? Métodos estáticos e propriedades são muito difíceis de substituir por implementações falsas durante os testes. (você já tentou testar um método que usa uma data com DateTime.Now, e você deseja testar com 29 de fevereiro de algum ano bissexto quântico?) No entanto, durante o teste, você pode substitua facilmente a classe real por uma classe falsa porque implementa uma interface!

Dependências Transientes

Quando você configura injeção de dependência para usar uma classe transitória, cada vez que uma instância precisa ser criado pelo IoCC, ele criará uma nova instância. O IoCC também irá descartar da instância (quando sua classe implementa a interface IDisposable) quando não há mais necessário. A maioria das classes do lado do servidor deve ser transitória porque cada solicitação em um servidor não deve depender de solicitações anteriores.

No entanto, no Blazor você está trabalhando no lado do cliente e, nesse caso, a interface do usuário permanece por toda a interação. Isso significa que você terá componentes que possuem apenas um instância criada e apenas uma instância da dependência. Você pode pensar neste caso transiente e único farão a mesma coisa. Mas pode haver outro componente que precisa do mesmo tipo de dependência. Se você estiver usando um singleton, ambos os componentes irá compartilhar a mesma instância da dependência, enquanto com transiente cada um obtém seu próprio instância! Você deve estar ciente disso.

Você configura a injeção de dependência para usar instâncias temporárias com o Método de extensão AddTransient, como na Listagem 4-14.

Listagem 4-14. Adicionando uma classe transitória à injeção de dependência

```
public void ConfigureServices (serviços IServiceCollection)
{
   services.AddTransient <IProductsService, ProductsService> ();
}
```

Dependências com escopo

Quando você configura injeção de dependência para usar uma dependência com escopo, o IoCC irá reutilizar a mesma instância por solicitação, mas usará novas instâncias entre diferentes solicitações de. Isso é especialmente útil se você usar objetos de repositório. Objetos de repositório

Página 131

acompanhar todas as alterações feitas em seus objetos e, em seguida, permitir que você salve (ou descarte) todas as alterações no final da solicitação. Se você usar instanciação temporária para repositórios, um uma única solicitação pode perder algumas alterações, o que resultaria em erros sutis. Vamos olhar para um exemplo. Imagine que você tenha um DebitService e outro CreditService. Ambos fazem muda para uma conta bancária e ambos usam um objeto BankRepository como uma dependência. Um TransferService usa um DebitService para debitar uma conta, e o CreditService credita uma conta, tudo usando o BankRepository. Olhe para a lista4-15.

Listagem 4-15. Implementando um TransferService

```
public class TransferService {

   private DebitService ds;
   private CreditService cs;
   private BankRepository br;

   public TransferService (
      DebitService ds, CreditService cs, BankRepository br)
   {
```

```
        this.ds = ds;
        this.cs = cs;
        this.br = br;
    }

    Transferência pública (valor decimal, conta de, conta para) {
        ds.Debit (de, montante);
        cs.Credit (para, montante);
        br.Commit ();
    }
}
```

Se todos os três serviços usam a mesma instância de BankRepository, então isso deve funcionar bem, como na Figura 4-4.

112

Página 132

Figura 4-4. *Usando um repositório com escopo*

Mas se cada um recebe sua própria nova instância de BankRepository, o método Commit não fará nada porque nenhuma alteração foi feita na instância BankRepository do TransferService, como na Figura 4-5.

113

Página 133

Figura 4-5. *Usando um repositório temporário*

Usar dependências com escopo no Blazor geralmente não terá uso prático, mas no
no próximo capítulo, você usará uma instância com escopo para implementar o microsserviço.

nunca use dependências com escopo dentro de singletons. a dependência com escopo provavelmente terá um estado incorreto após a primeira solicitação.

Eliminando Dependências

Um dos extras interessantes que você obtém com injeção de dependência é que ela cuida de chamar o método Dispose de instâncias que implementam IDisposable. Se o BankRepository classe do exemplo anterior implementa IDisposable, a limpeza ocorrerá no final da vida útil da instância. No caso de um singleton, isso seria no final do programa; para instâncias com escopo definido, isso seria no final da solicitação; e para transiente instâncias, isso normalmente seria quando seu componente é removido da IU. Dentro geral, se suas classes implementam IDisposable corretamente, você não precisa se preocupar com algo mais.

Página 134

Construindo Serviços Blazor

Vamos voltar ao seu projeto PizzaPlace e apresentá-lo a alguns serviços. Eu posso imaginar pelo menos dois serviços: um para recuperar o menu e outro para fazer o pedido quando o usuário clica no botão Order.

Comece revisando o componente Índice, que é mostrado na Lista 4-16 com o métodos deixados de fora por causa da concisão.

Listagem 4-16. O Componente de Índice

```
@página "/"

<! - Menu ->
<PizzaList Title = "Nossa lista selecionada de pizzas"
                    Menu = "@ State.Menu"
```

```
                    Selecionado = "@ ((pizza) => AddToBasket (pizza))" />
    <! - Menu final ->
    <! - Carrinho de compras ->
    <ShoppingBasket Title = "Seu pedido atual"
                    Basket = "@ State.Basket"
                    GetPizzaFromId = "@ State.Menu.GetPizza"
                    Selecionado = "@ (pos => RemoveFromBasket (pos))" />
    <! - Fim da cesta de compras ->
    <! - Entrada do cliente ->
    <CustomerEntry Title = "Por favor, insira seus dados abaixo"
                   Customer = "@ State.Basket.Customer"
                   Enviar = "@ ((_) => PlaceOrder ())" />
    <! - Finalizar entrada do cliente ->

    <p> @ State.ToJson () </p>

    @funções {

    estado privado Estado {get; } = novo estado ()
    {
        Menu = novo Menu
        {
            Pizzas = nova lista <Pizza> {
```


Página 135


```
            Pizza nova (1, "Pepperoni", 8,99M, Spiciness.Spicy),
            pizza nova (2, "Margarita", 7,99 milhões, tempero.Nenhuma),
            pizza nova (3, "Diabolo", 9,99 milhões, especiarias.quente)
        }
    }
};

...
}
```

Preste atenção especial ao patrimônio do Estado. Você irá inicializar o State.Menu propriedade de um serviço e você usará a injeção de dependência para passar o serviço.

Adicionando a abstração MenuService e IMenuService

Se você estiver usando o Visual Studio, clique com o botão direito do mouse no projeto PizzaPlace.Shared e selecione Adicionar ➤ Novo Item. Se você estiver usando Code, clique com o botão direito do mouse no projeto PizzaPlace.Shared e selecione Adicionar arquivo. Adicione uma nova classe de interface chamada IMenuService e complete-a conforme mostrado na Listagem 4-17 .

Listagem 4-17. A interface IMenuService

```
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
    interface pública IMenuService
    {
        Tarefa <Menu> GetMenu ();
    }
}
```

Esta interface permite que você recupere um menu. Observe que o método GetMenu retorna uma Tarefa <Menu>; isso ocorre porque você espera que o serviço recupere seu menu de um servidor (você vai construir isso nos próximos capítulos) e você quer que o método suporte um chamada assíncrona.

116

Página 136

Vamos elaborar sobre isso. Dê uma olhada no método OnInitAsync da Listagem 4-20 . É um método assíncrono que usa a palavra-chave async em sua declaração. Dentro de Método OnInitAsync, você chama o método GetMenu usando a palavra-chave await, que requer GetMenu para retornar um Task <Menu>. Graças à sintaxe async / await, isso é fácil fazer, mas requer que você retorne uma tarefa.

Agora adicione a classe HardCodedMenuService ao projeto PizzaPlace.Shared, como em Listagem 4-18.

Listagem 4-18. A classe HardCodedMenuService

```
using System.Collections.Generic;
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
    public class HardCodedMenuService: IMenuService
    {
        public Task <Menu> GetMenu ()
        {
            return Task.FromResult <Menu> (novo Menu {
                Pizzas = nova lista <Pizza> {
                    Pizza nova (1, "Pepperoni", 8,99M, Spiciness.Spicy),
                    pizza nova (2, "Margarita", 7,99 milhões, tempero.Nenhuma),
                    pizza nova (3, "Diabolo", 9,99 milhões, especiarias.quente)
                }
            });
        }
    }
}
```

Agora você está pronto para usar o IMenuService em seu componente de índice. Começar por adicionar a dependência em IMenuService usando a sintaxe @inject, como na Listagem 4-19 .

117

Página 137

Listagem 4-19. Afirmando que o componente de índice depende de um IMenuService

```
@página "/"
@using PizzaPlace.Shared;
@inject IMenuService menuService

<! - Menu ->
...
```

Você inicializa a propriedade State.Menu no método de ciclo de vida OnInitAsync, como em Listagem 4-20. Você já tem um método OnInit do capítulo anterior que você

não precisa mais, então não se esqueça de removê-lo.

Listagem 4-20. Inicializando o menu do componente de índice

```
@funções {

    estado privado Estado {get; } = novo estado ();

    substituição protegida assíncrona Tarefa OnInitAsync ()
    {
        State.Menu = espera menuService.GetMenu ();
        this.State.Basket.Customer.PropertyChanged + =
            (remetente, e) => this.StateHasChanged (); }

...
}
```

nunca chame serviços assíncronos no construtor de seu componente Blazor; sempre use OnInitAsync ou OnParametersSetAsync.

Agora você está pronto para configurar a injeção de dependência, então abra Startup.cs do projeto do cliente. Você usará um objeto transiente, conforme declarado na Listagem 4-21.

Listagem 4-21. Configurando injeção de dependência para o MenuService

```
usando Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Shared;
```


Página 138


```
namespace PizzaPlace.Client
{
    public class Startup
    {
        public void ConfigureServices (serviços IServiceCollection)
        {
            services.AddTransient <IMenuService,
                                   HardCodedMenuService> ();
        }

        public void Configure (aplicativo IBlazorApplicationBuilder)
        {
            app.AddComponent <App> ("app");
        }
    }
}
```

Execute seu projeto Blazor. Tudo ainda deve funcionar!

Pedindo pizzas com um serviço

Quando o usuário faz uma seleção de pizzas e preenche as informações do cliente, você quero enviar o pedido ao servidor para que eles possam aquecer o forno e enviar alguns bons pizzas para o endereço do cliente. Comece adicionando uma interface IOrderService ao Projeto PizzaPlace.Shared como na Listagem 4-22 .

Listagem 4-22. A abstração IOrderService como uma interface C #

```
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
```

```
    interface pública IOrderService
    {
        Tarefa PlaceOrder (cesta da cesta);
    }
}
```

119

Página 139

Para fazer um pedido, basta enviar a cesta para o servidor. No próximo capítulo, você irá construir o código real do lado do servidor para fazer um pedido; por enquanto, você vai usar um falso implementação que simplesmente grava o pedido no console do navegador. Adicione uma classe chamada ConsoleOrderService para o projeto PizzaPlace.Shared como na listagem 4-23 .

Listagem 4-23. O ConsoleOrderService

```
using System;
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
    public class ConsoleOrderService: IOrderService
    {
        Public Task PlaceOrder (Basket basket)
        {
            Console.WriteLine ($ "Encomenda para {basket.Customer.Name}");
            return Task.CompletedTask;
        }
    }
}
```

O método PlaceOrder simplesmente grava a cesta no console. No entanto, este método implementa o padrão assíncrono do .NET, então você precisa retornar um Instância da tarefa. Isso é feito facilmente usando a propriedade Task.CompletedTask. Tarefa. CompletedTask é simplesmente uma tarefa "sem nada" e é muito útil se você precisar implementar um método que precisa retornar uma instância de Task.

Injete o IOrderService no componente Índice como na Listagem 4-24.

Listagem 4-24. Injetando o IOrderService

```
@página "/"
@using PizzaPlace.Shared;
@inject IMenuService menuService
@inject IOrderService orderService
```

120

Página 140

Use o serviço de pedido quando o usuário clicar no botão Pedido, substituindo o implementação do método PlaceOrder no componente Index. Desde o orderService é assíncrono, você precisa invocá-lo de forma assíncrona, como na Listagem 4-25.

Listagem 4-25. O método Asynchronous PlaceOrder

```
Private assíncrono Task PlaceOrder ()
{
    esperar orderService.PlaceOrder (State.Basket);
}
```

Como etapa final, configure a injeção de dependência. Novamente, faça o orderService transitório como na listagem 4-26 .

Listagem 4-26. Configurando injeção de dependência para o orderService

```
usando Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Shared;

namespace PizzaPlace.Client
{
    public class Startup
    {
        public void ConfigureServices (serviços IServiceCollection)
        {
            services.AddTransient <IMenuService,
                                   HardCodedMenuService> ();
            services.AddTransient <IOrderService,
                                   ConsoleOrderService> ();
        }

        public void Configure (aplicativo IBlazorApplicationBuilder)
        {
            app.AddComponent <App> ("app");
        }
    }
}
```


Página 141

Pense sobre isso. Será difícil substituir a implementação de um dos Serviços? Há apenas um lugar que diz qual classe você usará, e esse lugar está em Listagem 4-26. No próximo capítulo, você construirá o código do lado do servidor necessário para armazenar o menu e os pedidos, e no capítulo seguinte você substituirá esses serviços por o verdadeiro negócio!

Construa seu projeto. Você receberá um aviso sobre como fazer uma chamada para um sistema assíncrono método. Isso ocorre porque o método PlaceOrder de IOrderService agora é assíncrono. Corrija-o alterando a propriedade Submit de CustomerEntry para usar um lambda assíncrono funcionar como na listagem 4-27 .

Listagem 4-27. Mudando para uma Função Lambda Assíncrona

```
<! - Entrada do cliente ->
<CustomerEntry Title = "Por favor, insira seus dados abaixo"
               Customer = "@ State.Basket.Customer"
               Enviar = "@ (async (_) => esperar PlaceOrder ())" />
<! - Finalizar entrada do cliente ->
```

Construa e execute seu projeto novamente, abra o depurador do seu navegador e abra o guia do console. Peça algumas pizzas e clique no botão Encomendar. Você deveria ver alguns feedback, conforme mostrado na Figura 4-6.

Figura 4-6. *O console do navegador mostra que um pedido foi feito*

Página 142

Resumo

Neste capítulo, você aprendeu sobre a inversão de dependência, que é uma prática recomendada para construir aplicativos orientados a objetos de fácil manutenção e teste. Você também viu que injeção de dependência torna muito fácil criar objetos com dependências, especialmente objetos que usam inversão de dependência. Ao configurar a injeção de dependência, você precisa ter cuidado com o tempo de vida de suas instâncias, então vamos repetir isso novamente:

- Objetos *transitórios* são sempre diferentes; uma nova instância é fornecida para cada componente e cada serviço.
- Os objetos *com escopo* são os mesmos em uma solicitação, mas diferentes entre pedidos diferentes.
- Os objetos *singleton* são iguais para todos os objetos e para todas as solicitações.

Página 143

CAPÍTULO 5

Armazenamento de dados e Microsserviços

Em geral, os aplicativos de navegador do lado do cliente precisam armazenar alguns de seus dados. Em alguns casos, como jogos, o aplicativo pode armazenar seus dados no próprio navegador, utilizando armazenamento local do navegador. Mas, na maioria dos casos, o armazenamento acontecerá no servidor, que tem acesso a mecanismos de banco de dados como o SQL Server. Neste capítulo, você aprenderá o básico de armazenar dados usando o Entity Framework Core e expor esses dados usando REST e microsserviços desenvolvidos com base no ASP.NET Core.

O que é REST?

O armazenamento de dados na Web é onipresente. Mas como os aplicativos podem se comunicar com um outro? *Representational State Transfer* (REST) é um protocolo construído em cima do HTTP protocolo para invocar funcionalidade em servidores, como recuperar e armazenar dados de / em um banco de dados.

Entendendo HTTP

Antes de falar sobre REST, você deve ter um bom entendimento do *hipertexto Protocolo de transferência* , mais conhecido como HTTP. HTTP foi criado por *Tim Berners-Lee* no CERN em 1989. O CERN é um centro de pesquisa em física elementar, e o que os pesquisadores fazem quando eles concluíram sua pesquisa? Eles publicam artigos com suas pesquisas descobertas. Antes da Internet, a publicação de um artigo era feita literalmente no papel (daí o nome) e demorou muito entre escrever o artigo e publicá-lo em um revista de pesquisa. Em vez disso, Tim Berners-Lee criou uma maneira de colocar papéis em um servidor e permitir que os usuários leiam esses documentos usando um navegador.

P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_5

Página 144

Além disso, os artigos científicos contêm muitas referências, e quando você deseja ler um papel como este ajuda a poder acessar os papéis referenciados. A Internet facilita leitura de artigos através do uso de *Hypertext Markup Language* (HTML). O hipertexto é um formato de documento eletrônico que pode conter links para outros documentos. Você simplesmente clica o link para ler o outro artigo e você pode voltar ao primeiro artigo simplesmente clicando o botão Voltar do seu navegador.

Identificadores de recursos universais e verbos

Navegadores são aplicativos que sabem falar HTTP, e a primeira coisa que você faz depois abrir um navegador é digitar um URI (*Universal Resource Identifier*). Um URI permite um navegador para falar com um servidor, mas é necessário mais. Como o nome sugere, um URI identifica algum recurso universalmente, mas você também precisa usar um *verbo* para instruir o servidor a fazer algo com o URI. O verbo mais comum é GET. Como figura 5-1 shows, quando você digite um URI no navegador, ele fará um GET no servidor.

Figura 5-1. *O navegador usa o verbo GET para recuperar um documento*

Cada vez que você clica em um hiperlink no documento HTML, o navegador repete isso processo com outro URI.

Mas existem outros verbos. Se você deseja publicar um novo artigo, você pode usar o POST verbo para enviar o papel ao servidor, fornecendo-lhe um URI. Neste caso, o servidor irá armazene o papel no URI solicitado. Se você quiser fazer uma alteração em seu jornal, por exemplo para corrigir um erro ortográfico, você pode usar o verbo PUT. Agora o servidor irá sobrescrever o conteúdo do URI. E, finalmente, você pode deletar o papel usando o verbo DELETE e seu URI.

Página 145

Códigos de status HTTP

O que acontece quando você pergunta a um servidor sobre algo que ele não tem? O que deveria o servidor retorna? Os servidores não retornam apenas HTML, eles também retornam um código de status sobre o resultado. Quando o servidor pode processar a solicitação com sucesso, em geral código de status de retorno 200 (existem outros códigos de status bem-sucedidos). Quando o servidor não consegue encontrar o recurso, ele retornará um código de status 404. O código de status 404 significa simplesmente não encontrado. O cliente receberá este código de status e pode reagir de forma adequada. Quando o navegador recebe um código de status 200, exibe o HTML; quando recebe um 404, exibe um not tela encontrada, etc.

Invocando a funcionalidade do servidor usando REST

Pense nesses verbos de que acabamos de falar. Com o POST você pode CRIAR algo em um servidor; com GET você pode LER de volta; com PUT você pode ATUALIZAR algo no servidor; e com DELETE você pode DELETE algo no servidor. Eles são também conhecidas como operações CRUD (CREATE-READ-UPDATE-DELETE). *Roy Fielding* , o inventor do REST, percebeu que usando o protocolo HTTP você também pode usar o HTTP para trabalhar com dados armazenados em um banco de dados. Por exemplo, se você usar o verbo GET com um URI http: // alguns servidores / categorias, o servidor pode executar algum código para recuperar dados do tabela relacional de categorias e devolvê-lo. Claro, o servidor usaria um formato mais apropriado para a transferência de dados, como XML ou JSON. Porque existem muitos formatos de dados, o servidor também precisa de uma forma de transmitir o formato que está enviando. (No início da Web, apenas HTML foi usado como formato.) Isso é feito por meio de HTTP cabeçalhos.

Cabeçalhos HTTP

Os cabeçalhos HTTP são instruções trocadas entre o cliente e o servidor. Cabeçalhos são pares de valor-chave, onde cliente e servidor concordam com a chave. Muitos cabeçalhos HTTP padrão existir. Por exemplo, um servidor pode usar o *cabeçalho Content-Type* para dizer ao cliente para esperar um formato específico. Outro cabeçalho é o *cabeçalho Aceitar* , que é enviado pelo cliente ao servidor para pedir educadamente ao servidor para enviar o conteúdo nesse formato; isso também é conhecido como *negociação de conteúdo* . Atualmente, o formato mais popular é *JavaScript Object Notation* (JSON). E este é o formato de troca que você usará com o Blazor.

Página 146

JavaScript Object Notation

JSON é um formato compacto para transferência de dados. Veja o exemplo na Listagem 5-1 .

Listagem 5-1. Um exemplo de JSON

```
{ "livro" : {
   "title": "Blazor Revealed",
   "capítulos": ["Seu primeiro projeto Blazor", "Data Binding"]
   }
}
```

Este formato JSON descreve um livro, um objeto na memória. Os objetos são denotados com chaves. Dentro do livro existem duas propriedades; cada propriedade usa uma chave: valor notação. O título do livro é "Blazor Revealed". Observe que o nome da propriedade também é transferido como uma string. E, finalmente, a propriedade chapters é uma matriz de strings, onde você usa colchetes para indicar uma matriz.

O formato JSON é usado para transferir dados entre duas máquinas, mas hoje é também muito usado para configurar ferramentas como ASP.NET Core. JSON hoje é muito mais mais popular na Web do que XML, provavelmente por causa de sua simplicidade.

Alguns exemplos de chamadas REST

Você precisa de uma lista de pizzas de um servidor, e o servidor expõe as pizzas em URI http: // someserver / pizzas. Para obter uma lista de pizzas, você usa o verbo GET, e você usa o Aceitar cabeçalho com o valor application / json para solicitar o formato JSON. Observe a Figura 5-2 para este exemplo.

Figura 5-2. *Usando REST para recuperar uma lista de pizzas*

Página 147

Talvez o seu cliente queira exibir os detalhes de uma pizza com o número de identificação 5. Neste caso, ele pode anexar o id ao URI e executar um GET. O servidor não deve ter nenhum pizza com esse id, ele pode retornar um código de status 404, conforme ilustrado na Figura 5-3 .

Figura 5-3. *Usando REST para recuperar uma pizza específica por meio de seu id único*

Como último exemplo, vamos enviar alguns dados do cliente para o servidor. Imagine isso

o cliente preencheu todos os detalhes do pedido e clica no botão Pedido. Vocês
em seguida, envie o pedido como JSON para o servidor usando o verbo POST (lembre-se de POST significa
inserir). O servidor pode então processar o pedido da maneira que desejar; por exemplo, pode
insira o pedido em seu banco de dados e retorne um 201: código de status criado, como na Figura 5-4 .
REST recomenda retornar um código de status 201 com o cabeçalho Location definido para o URI
para o recurso recém-criado.

Figura 5-4. *POSTANDO um pedido para o servidor*

Página 148

Construindo um microsserviço simples usando ASP.NET Core

Então, como você constrói um serviço REST? Seu projeto Blazor usa ASP.NET Core para hospedagem
o cliente Blazor e adicionar um serviço ao seu projeto é fácil. Mas primeiro, vamos fazer uma pequena introdução
para microsserviços.

Serviços e responsabilidade única

Um serviço é um software que escuta as solicitações; quando recebe um pedido, o
serviço lida com a solicitação e retorna com uma resposta. Na vida real, você também encontra
serviços e eles são muito semelhantes. Considere um banco. Você entra em um banco e dá
ao caixa, o número da sua conta, algum ID e solicite $ 100. O caixa irá verificar o seu
conta; se você tiver dinheiro suficiente em sua conta, o caixa irá deduzir o dinheiro
e te dar o dinheiro. Se sua conta estiver muito baixa, o caixa recusará. Em ambos
casos, você obteve uma resposta.

Os serviços também devem seguir o princípio da responsabilidade única. Eles deviam
faça uma coisa muito bem e pronto. Por exemplo, o serviço de pizza permitirá que os clientes
recuperar pizzas, adicionar, atualizar e excluir pizzas. É isso. Uma única responsabilidade, neste
case PIZZAS.

Você também pode ter outros serviços, cada um com sua própria responsabilidade. Serviços que levam
cuidar de uma coisa são conhecidos como *microsserviços* .

The Pizza Service

Abra a solução PizzaPlace na qual você trabalhou nos capítulos anteriores. Neste capítulo, você
se concentrará no projeto PizzaPlace.Server, mostrado na Figura 5-5.

Página 149

Figura 5-5. *O projeto PizzaPlace.Server*

A única função deste projeto atualmente é hospedar seu aplicativo cliente Blazor, mas agora você vai aprimorar essa função adicionando alguns microsserviços.

Abra Startup.cs e observe o método Configure, como na Listagem 5-2.

Listagem 5-2. Método de configuração da classe de inicialização

```
public void Configure (aplicativo IApplicationBuilder,
                       IHostingEnvironment env)
{
    app.UseResponseCompression ();

    if (env.IsDevelopment ())
    {
        app.UseDeveloperExceptionPage ();
    }

    app.UseMvc (routes =>
    {
```


Página 150


```
        routes.MapRoute (name: "default",
                         modelo: "{controlador} / {ação} / {id?}");
    });

    app.UseBlazor <Client.Program> ();
}
```

A última linha com o método UseBlazor cuida do seu projeto do cliente Blazor. Mas logo antes, você vê o método UseMvc que ele usava para hospedar seus serviços.

Como o método UseMvc funciona não é o tópico deste livro, mas vou cobrir o que você precisa saber. Se você quiser aprender mais sobre ASP.NET Core, existem muitos bons livros sobre esse tópico, como *Pro ASP.NET Core MVC* de Adam Freeman.

Em seguida está a pasta Controllers. Na Figura 5-5, esta pasta está vazia e o

ideia é que você coloque suas classes de serviço aqui. No ASP.NET, as classes de serviço são conhecidas como controladores. Se você estiver usando o Visual Studio, clique com o botão direito nesta pasta e selecione Adicionar ➤ Controlador. Selecione o controlador de API - vazio da figura 5-6 e clique em Adicionar.

Figura 5-6. *Adicionando um novo controlador*

Página 151

Digite PizzasController como na Figura 5-7 e clique em Adicionar novamente.

Figura 5-7. *Nomeando o controlador*

Se você estiver usando Code, simplesmente clique com o botão direito na pasta Controllers e selecione Add File. Nomeie-o como PizzasController.cs.

Isso adicionará uma nova classe chamada PizzasController, herdada de ControllerBase, que você pode ver na Listagem 5-3. Observe que o atributo Route indica que o O URI que você deve usar é api / pizzas. A parte [do controlador] da rota é um marcador para o nome do controlador, mas sem a parte do controlador.

Listagem 5-3. The Empty PizzasController

```
namespace PizzaPlace.Server.Controllers
{
    [Route ("api / [controlador]")]
    [ApiController]
    public class PizzasController: ControllerBase
    {
    }
}
```

Vamos adicionar um método para recuperar uma lista de pizzas. No momento você vai difícil-codifique a lista, mas na próxima seção você irá recuperá-la de um banco de dados. Modifique o PizzasController conforme mostrado na Listagem 5-4 .

Listagem 5-4. Adicionando um método ao PizzaController para recuperar uma lista de pizzas

```
using System.Collections.Generic;
usando System.Linq;
usando Microsoft.AspNetCore.Mvc;
using PizzaPlace.Shared;
```

133

Página 152

Capítulo 5 Armazenamento de dados e miCroServiços

```
namespace PizzaPlace.Server.Controllers
{
    [ApiController]
    public class PizzasController: ControllerBase
    {
        Lista estática privada <Pizza> pizzas = nova Lista <Pizza>
        {
            Pizza nova (1, "Pepperoni", 8,99M, Spiciness.Spicy),
            pizza nova (2, "Margarita", 7,99 milhões, tempero.Nenhuma),
            pizza nova (3, "Diabolo", 9,99 milhões, especiarias.quente)
        };

        [HttpGet ("pizzas")]
        public IQueryable <Pizza> GetPizzas ()
        {
            retornar pizzas.AsQueryable ();
        }
    }
}
```

Vamos examinar essa implementação. Primeiro, você declara uma lista estática codificada de pizzas. Em seguida, está o método GetPizzas, que possui um atributo HttpGet ("pizzas"). Este atributo diz que quando você realiza um GET no servidor com o URI de pizzas o o servidor deve chamar o método GetPizzas.

O método GetPizzas retorna um IQueryable <Pizza> e o ASP.NET Core enviará este resultado volta para o cliente com o formato solicitado. A interface IQueryable <Pizza> é usado em .NET para representar dados que podem ser consultados, como dados de banco de dados, e é retornado por consultas LINQ.

Observe que o método GetPizzas não contém nada sobre como os dados serão transferido para o cliente. tudo isso é cuidado para você pelo aSp.net Core! Por padrão, sua implementação no aSp.net Core usará JSon, que é o que você deseja.

134

Página 153

Capítulo 5 Armazenamento de dados e miCroServiços

É hora de ver se funciona. Primeiro, certifique-se de que o projeto PizzaPlace.Server seja a inicialização projeto. Clique com o botão direito do mouse no projeto PizzaPlace.Server e selecione Definir como projeto de inicialização de o menu suspenso. O projeto PizzaPlace.Server deve ser mostrado em negrito, como em Figura 5-5.

Agora execute seu projeto e espere o navegador abrir porque você executará um GET; você pode usar o navegador, mas para outros verbos, você usará mais tarde uma boa ferramenta chamada Carteiro.

Altere o URI no navegador para http: // localhost: xxxx / pizzas em que xxxx é o

número da porta original em seu navegador (o número da porta é selecionado pelo Visual Studio e será diferente do meu). Você deve ver o resultado mostrado na Figura 5-8.

Uma lista de pizzas codificada em JSON! Funciona!

Figura 5-8. *Os resultados de obter uma lista de pizzas do serviço de pizza*

Agora você está pronto para recuperar os dados de um banco de dados real usando o Entity Framework Testemunho.

O que é o Entity Framework Core?

Entity Framework Core é a estrutura que a Microsoft recomenda para trabalhar bancos de dados. Ele permite que você escreva classes como classes C # normais e, em seguida, armazene e recupere Objetos .NET de um banco de dados sem precisar ser um especialista em SQL. Vai cuidar disso para você. Isso também é conhecido como *ignorância de persistência* , em que seu código não precisa saiba como e onde os dados são armazenados!

Página 154

Usando a abordagem Code First

Mas é claro, você precisa explicar ao Entity Framework Core que tipo de dados você deseja armazenar. Entity Framework Core usa uma técnica chamada *código primeiro* , onde você escreve código para descrever os dados e como eles devem ser armazenados no banco de dados. Então você pode use isso para gerar o banco de dados, as tabelas e as restrições. Se você quiser fazer alterações para o banco de dados, você pode atualizar o esquema do banco de dados com *migrações* iniciais de *código* . Se você já tem um banco de dados, você também pode gerar o código do banco de dados, mas isso não é o alvo deste livro.

Na abordagem do primeiro código, você descreve as classes (também conhecidas como entidades) que irá mapear para tabelas de banco de dados. Você já tem a classe Pizza (que pode encontrar no Projeto PizzaPlace.Shared) para descrever a tabela Pizza no banco de dados. Mas você precisa faça mais.

> nesta parte, você usará o SQL Server. se você instalou o Visual Studio em seu Máquina Windows, SQL Server também foi instalado. se você não tem o SQL Server ligado sua máquina, você pode instalar uma versão gratuita do SQL Server ou usar um SQL Server instância na nuvem, por exemplo SQL Server no azure (https: // azure. microsoft.com/en-us/get-started/)

Você precisa adicionar o Entity Framework Core ao projeto PizzaPlace.Server. Se você é usando o Visual Studio, clique com o botão direito do mouse no projeto do servidor e selecione Gerenciar Pacotes NuGet, como mostrado na Figura 5-9 .

Figura 5-9. *Adicionando pacotes NuGet ao seu projeto*

136

Página 155

A janela NuGet será aberta no Visual Studio. NuGet é uma maneira muito prática de instalando dependências como Entity Framework Core em seu projeto. Não vai instale apenas a biblioteca Microsoft.EntityFrameworkCore.SqlServer, mas também todos os seus dependências.

Selecione a guia Procurar e digite Microsoft.EntityFrameworkCore.SqlServer no caixa de pesquisa. Você deve ver esta biblioteca como o principal resultado da pesquisa. Selecione-o e, em seguida, selecione o Última versão estável na lista suspensa Versão e clique no botão Instalar, conforme mostrado na Figura 5-10 .

No momento em que você lê este livro, a Microsoft pode ter implantado um mais recente versão, embora Figura 5-10 mostra a versão 2.1.1, você deve selecionar o mais recente Versão estável.

Figura 5-10. *Adicionando Entity Framework Core usando NuGet*

Com o Código, você abre o prompt de comando e digita o seguinte comando após alterando o diretório atual do seu projeto PizzaPlace.Server:

```
dotnet adicionar pacote Microsoft.EntityFrameworkCore.SqlServer
```

Com essa dependência instalada, você está pronto para fazer algumas alterações no código. Entidade O Framework Core requer que as classes de entidade tenham um construtor padrão e que propriedades são de leitura e gravação. Atualize a classe Pizza adicionando um construtor padrão e adicionando setters para as propriedades, conforme mostrado na Listagem 5-5.

137

Página 156

Listagem 5-5. Modificando a classe Pizza para Entity Framework Core

```
pizza de classe pública
{
```

```
    pizza pública () {}

    pizza pública (int id, nome da string,
                   preço decimal, picante picante)
    {
      this.Id = id;
      this.Name = nome ?? lance novo ArgumentNullException (
                      nameof (name), "Uma pizza precisa de um nome!");
      this.Price = price;
      este. Picante = picante;
    }

    public int Id {get; definir; }
    public string Name {get; definir; }
    Preço decimal público {get; definir; }
    picante público Picante {obter; definir; }
}
```

Adicione uma nova classe chamada PizzaPlaceDbContext ao projeto PizzaPlace.Server, como mostrado na lista 5-6. Esta classe representa o banco de dados, e você precisa fornecer um algumas dicas sobre como você deseja que seus dados sejam armazenados no SQL Server (ou algum outro mecanismo de banco de dados; isso usa o mesmo código).

Listagem 5-6. A classe PizzaPlaceDbContext

```
usando Microsoft.EntityFrameworkCore;
using PizzaPlace.Shared;

namespace PizzaPlace.Server
{
    public class PizzaPlaceDbContext: DbContext
    {
```


Página 157


```
        public PizzaPlaceDbContext (
            Opções DbContextOptions <PizzaPlaceDbContext>)
            : base (opções)
        {}

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

        protected override void OnModelCreating (
            ModelBuilder modelBuilder)
        {
            base.OnModelCreating (modelBuilder);

            var pizzaEntity = modelBuilder.Entity <Pizza> ();
            pizzaEntity.HasKey (pizza => pizza.Id);
            pizzaEntity.Property (pizza => pizza.Price)
                       .HasColumnType ("dinheiro");

        }
    }
}
```

Primeiro, você precisa criar um construtor para a classe PizzaPlaceDbContext tomando um argumento DbContextOptions <PizzaPlaceDbContext>. Isso é usado para passar o conexão com o servidor, o que você fará posteriormente nesta seção.

Em seguida, você adiciona uma tabela ao banco de dados para representar suas pizzas usando uma propriedade pública do tipo DbSet <Pizza>. DbSet <T> é a classe de coleção usada pelo Entity Framework Core,

mas você pode pensar nisso como uma Lista <T>. O Entity Framework Core usará o DbSet <T> para mapear esta coleção para uma mesa, neste caso a mesa de Pizzas.

Por fim, você substitui o método OnModelCreating, que usa um modelBuilder argumento. No método OnModelCreating, você pode descrever como cada DbSet <T> deve ser mapeado para o banco de dados; por exemplo, você pode dizer qual tabela usar, como cada coluna deve ser chamada, que tipo usar, etc. Este modelBuilder tem um monte de métodos que permitem descrever como as classes devem ser mapeadas para seu banco de dados. Neste caso, você diz ao modelBuilder que a mesa Pizza deve ter uma chave primária, a propriedade Id da classe Pizza. Você também precisa dizer como a propriedade Pizza.Price deve ser mapeado para o SQL Server. Você usará o tipo MONEY do SQL Server para isso. Para o momento, isso é o suficiente para sua implementação atual.

Página 158

Preparando Seu Projeto para Migrações Code First

Agora você está pronto para dizer ao projeto PizzaPlaze.Server para usar o SQL Server como o base de dados. Você faz isso com injeção de dependência. No ASP.NET Core, você configura injeção de dependência no método ConfigureServices da classe Startup. Vamos ter um observe este método mostrado na Listagem 5-7 .

Listagem 5-7. O método Startup.ConfigureServices

```
public void ConfigureServices (serviços IServiceCollection)
{
   services.AddMvc (). AddJsonOptions (options =>
   {
      options.SerializerSettings.ContractResolver =
         new DefaultContractResolver ();
   });

   services.AddResponseCompression (options =>
   {
      options.MimeTypes = ResponseCompressionDefaults.MimeTypes
         .Concat (novo [] {
            MediaTypeNames.Application.Octet,
            WasmMediaTypeNames.Application.Wasm,
         });
   });
}
```

Lembra-se de IServiceCollection do capítulo sobre injeção de dependência? Aqui, dependências para ASP.NET Core são adicionadas, como dependências para Mvc e ResponseCompression, que são necessários para o seu serviço.

Comece adicionando um construtor à classe Startup como na Listagem 5-8 .

Listagem 5-8. O construtor da classe de inicialização

```
usando Microsoft.Extensions.Configuration;

...

Inicialização pública (configuração IConfiguration)
```


Página 159

```
{
    Configuração = configuração;
}

Configuração de IConfiguration pública {get; }
```

Você precisa desse construtor para ter acesso ao arquivo de configuração de projetos. O a configuração conterá a string de conexão para o banco de dados se comunicar.

No ConfigureServices, você precisa adicionar quaisquer dependências adicionais de seu a implementação requer. Adicione o seguinte código da lista5-9 no final do método.

Listagem 5-9. Adicionando Dependências do Entity Framework

```
services.AddDbContext <PizzaPlaceDbContext> (opções
    => options.UseSqlServer (
            Configuration.GetConnectionString ("PizzaDb")));
```

Esta única instrução informa ao ASP.NET Core que você usará o PizzaPlaceDbContext e que você o armazenará no SQL Server. Este código também parece até a string de conexão para o banco de dados na configuração, que você ainda precisa adicionar.

Clique com o botão direito do mouse no projeto PizzaPlace.Server e selecione Add ➤ New Item. Digite json na caixa de pesquisa e selecione Arquivo de configurações do aplicativo, conforme mostrado na Figura 5-11 . Mantenha o padrão nome de appsettings.json e clique em Adicionar.

Figura 5-11. *Adicionando o arquivo de configuração do aplicativo*

Página 160

Com o Code, basta adicionar um novo arquivo chamado appsettings.json. Clique duas vezes no novo arquivo appsettings.json para abri-lo. ASP.NET Core usa um arquivo JSON para configuração e você precisa adicionar uma string de conexão ao banco de dados. Uma string de conexão de banco de dados diz ao seu código onde encontrar o servidor de banco de dados, qual banco de dados usar e qual credenciais devem ser usadas para fazer login. O Visual Studio adicionou um arquivo de configuração, como em Listagem 5-10. Esta string de conexão usa o servidor (localdb) \\ MSSQLLocalDB, que é o servidor instalado com o Visual Studio. As únicas coisas que você precisa fazer são definir o nome do banco de dados substituindo _CHANGE_ME por um nome mais adequado para seu banco de dados e para alterar o nome da conexão. Claro, se você estiver usando outro banco de dados servidor, você também terá que alterar o nome do servidor. Ou continue lendo para descobrir como obter a string de conexão com o Visual Studio.

Listagem 5-10. O arquivo de configuração appsettings.json

```
{
  "ConnectionStrings": {
    "PizzaDb": "Servidor = (localdb) \\ MSSQLLocalDB; Banco de dados = _CHANGE_ME; Confiável_
    Connection = True; MultipleActiveResultSets = true "
  }
}
```

Encontrando a string de conexão do servidor de banco de dados

Se você não tiver certeza de qual string de conexão usar, você pode encontrar a string de conexão em Visual Studio selecionando View ➤ SQL Server Object Explorer.

Você pode se conectar a um banco de dados clicando no ícone do servidor com o pequeno sinal + verde, mostrado na Figura 5-12.

Figura 5-12. *SQL Server Object Explorer*

Página 161

Você pode procurar servidores de banco de dados disponíveis expandindo o local, rede ou Azure como na Figura 5-13. Eu recomendo que você tente encontrar o banco de dados MSSQLLocalDB servidor. Se você usar outro servidor de banco de dados, pode ser necessário alterar a forma de fazer login no seu banco de dados. Quando estiver pronto, clique em Conectar.

Figura 5-13. *Encontrar a string de conexão para um banco de dados*

Página 162

Em seguida, expanda SQL Server da Figura 5-13 e selecione seu servidor. Clique com o botão direito e selecione Propriedades. Agora copie a string de conexão da janela de propriedades e mude o nome do banco de dados para PizzaDb.

Criando Sua Primeira Migração do Code First

Você está quase pronto para gerar o banco de dados a partir do código. Mas primeiro você precisa criar uma migração. Uma migração é uma classe C # que contém as mudanças que precisam ser feitas para o banco de dados para ativá-lo (ou desativá-lo) para o esquema de que seu aplicativo precisa. Isso está feito através de uma ferramenta.

Comece selecionando no menu do Visual Studio Exibir ➤ Outras janelas ➤ Pacote Console do gerenciador, que você pode ver na Figura 5-14.

Figura 5-14. *O console do gerenciador de pacotes*

Certifique-se de que o projeto padrão esteja definido como PizzaPlace.Server. Isso fará com que o seu os comandos têm como alvo o projeto selecionado.

Se você estiver usando Code, use o terminal integrado ou abra um prompt de comando.

Você deve executar o próximo comando no diretório PizzaPlace.Server, então certifique-se você está no diretório correto. Opcionalmente, digite o seguinte comando para alterar o diretório atual para o diretório do projeto PizzaPlace.Server:

```
cd PizzaPlace.Server
```

Agora execute o seguinte comando para criar a migração:

```
dotnet ef migations add CreatingPizzaDb
```


Página 163

Aqui você usa o comando dotnet para executar a ferramenta ef (Entity Framework) para adicionar um nova migração chamada CreatingPizzaDb. Deverá ver o seguinte resultado (por favor ignore quaisquer diferenças nas versões mostradas):

```
informações: Microsoft.EntityFrameworkCore.Infrastructure [10403]
      Entity Framework Core 2.1.0-rtm-30799 inicializado
'PizzaPlaceDbContext' usando o provedor 'Microsoft.EntityFrameworkCore.
SqlServer 'com opções: Nenhum
Feito. Para desfazer esta ação, use 'ef migrations remove'
```

Se você receber um erro ou avisos, revise o código para Pizza e o PizzaPlaceDbContext e tente novamente.

Esta ferramenta criou uma nova pasta de Migrações no projeto PizzaPlace.Server com dois arquivos semelhantes à Figura 5-15, mas com um carimbo de data / hora diferente.

Figura 5-15. *O resultado da adição da primeira migração*

Abra o arquivo CreatingPizzaDb.cs da Listagem 5-11 e veja o que a ferramenta fez.

Listagem 5-11. O arquivo CreatingPizzaDb.cs

```
classe pública parcial CreatingPizzaDb: migração
{
    override protegido void Up (
                                        MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable (
                nome: "Pizzas",
                colunas: tabela => novo
                {
                    Id = tabela.Coluna <int> (anulável: falso)
                    .Annotation ("SqlServer: ValueGenerationStrategy",
```


Página 164


```
                    SqlServerValueGenerationStrategy.IdentityColumn),
                    Nome = tabela.Coluna <string> (anulável: verdadeiro),
                    Preço = tabela.Coluna <decimal> (tipo: "dinheiro",
                                                        anulável: falso),
                    Spiciness = table.Column <int> (nullable: false)
                },
                restrições: tabela =>
                {
                    table.PrimaryKey ("PK_Pizzas", x => x.Id);
                });
    }

    override protegido void Down (
                                    MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable (
                nome: "Pizzas");
    }
}
```

Uma classe de migração possui dois métodos: Up e Down. O método Up irá atualizar o esquema de banco de dados. Neste caso, será criada uma nova tabela chamada Pizzas com Id, Nome, Colunas de preço e tempero.

O método Down faz o downgrade do esquema do banco de dados, neste caso eliminando o coluna.

Gerando o Banco de Dados

Agora você está pronto para gerar o banco de dados de suas migrações. Com o Visual Studio, volte para a janela do Console do gerenciador de pacotes (Exibir ➤ Outras janelas ➤ Pacote Console do gerenciador), ou com Código abra o terminal integrado (Exibir ➤ Terminal) e digite o seguinte comando:

```
atualização do banco de dados dotnet ef --verbose
```

146

Página 165

Como você pediu que a ferramenta fosse detalhada, isso gerará muitos resultados, entre onde você encontrará as instruções DDL executadas, como na Listagem 5-12 .

Listagem 5-12. Um extrato da saída da ferramenta de geração de banco de dados

```
CRIAR MESA [Pizzas] (
        [Id] int NOT NULL IDENTITY,
        [Nome] nvarchar (max) NULL,
        [Preço] dinheiro NÃO NULO,
        [Spicyness] int NOT NULL,
        CONSTRAINT [PK_Pizzas] PRIMARY KEY ([Id])
);
```

Isso acabou de criar o banco de dados para você!

Vamos dar uma olhada no banco de dados. No Visual Studio, abra View ➤ SQL Server Object Explorer e expanda a árvore para o banco de dados PizzaDb como na Figura 5-16 (no meu sistema Eu tenho alguns outros bancos de dados; apenas ignore-os).

147

Página 166

Figura 5-16. *SQL Server Object Explorer mostrando o banco de dados PizzaDb*

Se você não tiver o Visual Studio, pode baixar o *SQL Operations Studio* em www.microsoft.com/en-us/sql-server/developer-tools. Após o término da instalação, SQL O Operations Studio será iniciado. Digite o nome do seu servidor e selecione PizzaDb no lista suspensa, conforme mostrado na Figura 5-17 .

Página 167

Figura 5-17. *Conexão com SQL Operations Studio*

Aprimorando o microsserviço de pizza

Vamos adicionar algumas funcionalidades ao microsserviço Pizza para que ele use o banco de dados em vez de dados embutidos em código e adicione um método para inserir uma pizza em seu banco de dados.

Abra a classe PizzaController, que fica na pasta Controllers do o projeto PizzaPlace.Server. Comece adicionando um construtor que leva o

PizzaPlaceDbContext como um argumento, como na Listagem 5-13.

Página 168

Listagem 5-13. Injetando uma instância PizzaPlaceDbContext no controlador

```
public class PizzasController: ControllerBase
{
   private PizzaPlaceDbContext db;

   public PizzasController (PizzaPlaceDbContext db)
   {
      this.db = db;
   }
```

Para se comunicar com o banco de dados, o PizzasController precisa de um PizzaPlaceDbContext instância, e como você aprendeu no capítulo sobre *injeção de dependência* , você pode usar um construtor para fazer isso. O construtor só precisa salvar a referência em um campo local (por enquanto).

Você não precisa da lista de pizzas codificada, então remova o campo estático e atualize o método GetPizza para usar o PizzaPlaceDbContext em vez disso, como na Listagem 5-14. Para obter todas as pizzas você pode simplesmente usar a propriedade Pizzas do PizzaPlaceDbContext. O O Entity Framework acessará o banco de dados ao acessar a propriedade Pizzas.

Listagem 5-14. Recuperando as pizzas do banco de dados

```
[HttpGet ("pizzas")]
public IQueryable <Pizza> GetPizzas ()
{
   return db.Pizzas;
}
```

Agora vamos adicionar um método para inserir uma nova pizza no banco de dados. Adicione o InsertPizza método da lista 5-15 para a classe PizzasController. Este método receberá um instância de pizza do cliente como parte do corpo da solicitação POST, então você adiciona o HttpPost atributo com o URI que você deve postar. O objeto pizza será postado no corpo da solicitação, e é por isso que o argumento pizza do método InsertPizza tem o Atributo FromBody para informar à ASP.NET MVC Core para converter o corpo em uma instância de pizza. O método adiciona a pizza à tabela PizzaPlaceDbContext Pizzas e a salva para o banco de dados. O método InsertPizza retorna um código de status 201 Created com o URI da pizza como a resposta. Você examinará esta resposta com Postman no próxima parte deste capítulo.

Página 169

Listagem 5-15. O método InsertPizza

```
[HttpPost ("pizzas")]
public IActionResult InsertPizza ([FromBody] Pizza pizza)
{
   db.Pizzas.Add (pizza);
   db.SaveChanges ();
```

```
    retorno Criado ($ "pizzas / {pizza.Id}", pizza);
}
```

esta é uma introdução aos serviços de reSt. Construindo serviços reais com todos os diferentes abordagens e melhores práticas podem ocupar um livro inteiro. a ideia disso capítulo é para colocá-lo em funcionamento.

Testando seu microsserviço usando o Postman

Agora você tem seu primeiro microsserviço. Mas como você testa isso? Anteriormente você costumava navegador para testar o método GetPizzas, mas para outros verbos você precisa de uma ferramenta melhor. Aqui você usará o Postman, que é uma ferramenta específica para testar serviços REST.

Instalando Postman

Abra seu navegador favorito e vá para www.getpostman.com . Baixe o aplicativo (clique no botão Baixar o aplicativo da Figura 5-18 e escolha sua plataforma) e instale-o.

Página 170

Figura 5-18. *A página do Postman*

No momento em que você lê este livro, o procedimento de instalação pode ter mudado um pouco, portanto, siga as instruções do instalador.

Após a instalação, execute o Postman.

Fazendo chamadas REST com Postman

O Postman será aberto e perguntará o que você deseja fazer. Selecione Solicitar, como mostrado em Figura 5-19.

152

Página 171

Figura 5-19. *Selecione Solicitar para começar a usar o Postman*

Em seguida, ele perguntará onde salvar a solicitação, então escolha um nome e uma pasta, conforme mostrado na Figura 5-20 .

153

Página 172

Figura 5-20. *Salvando a solicitação*

Página 173

Fazendo uma solicitação GET

Agora execute a solução PizzaPlace e copie a URL do navegador. Cole em
O campo “Enter the Request URL” do Postman e anexe / pizzas como na Figura 5-21 . Não
esqueça que provavelmente você terá um número de porta diferente!

Figura 5-21. *Fazendo uma solicitação GET com Postman*

Antes de clicar em ENVIAR, vamos adicionar o cabeçalho Aceitar. Clique na guia Cabeçalhos e digite
Aceite como a chave e application / json como o valor. Por favor, consulte a Figura 5-22 para
referência.

Figura 5-22. *Adicionando cabeçalhos à solicitação no Postman*

Página 174

Agora você pode clicar em Enviar. Você deve receber uma lista vazia como na Figura 5-23 (que é normal porque você ainda não adicionou nenhuma linha à mesa de pizza).

Figura 5-23. *Recebendo uma lista vazia de pizzas do servidor*

Inserindo Pizzas com POST

Vamos adicionar algumas pizzas ao banco de dados. Na parte superior do Postman, você encontrará uma guia com um sinal de mais. Clique nele para adicionar outra guia. Selecione POST como o verbo e copie o URI da guia anterior, conforme mostrado na Figura 5-24 .

Figura 5-24. *Começando com a solicitação POST*

Agora selecione a seção Cabeçalhos e adicione um novo cabeçalho com a chave Content-Type e value application / json como na Figura 5-25 .

Página 175

Figura 5-25. *Adicionando o cabeçalho Content-Type para a solicitação POST*

Agora selecione a seção Corpo, clique no botão de opção de formato bruto e insira uma pizza objeto usando JSON. Consulte a Figura 5-26 . Observe que esta string bruta contém o propriedades de pizza serializadas como JSON, e que você não precisa enviar a propriedade Id porque o servidor irá gerar o id quando for inserido no banco de dados.

Figura 5-26. *Entrando em uma pizza usando JSON*

Página 176

Clique no botão Enviar. Se tudo estiver bem, você deve receber um 201 criado positivo resposta. Na área de resposta do Postman, selecione a guia Cabeçalhos como na Figura 5-27. Procure o cabeçalho Location. Ele mostrará o novo URI dado a esta pizza. Este local o cabeçalho é retornado pelo método Created que você chamou como a última linha da Listagem 5-15 .

Figura 5-27. *A resposta POST no Postman*

Clique na primeira guia em que você criou a solicitação GET e clique em Enviar novamente. Agora você deve ter uma lista de pizzas (uma lista de uma). Experimente criar algumas outras pizzas. Figura 5-28 é o meu resultado depois de adicionar três pizzas.

Página 177

Figura 5-28. *Uma lista de pizzas armazenadas no banco de dados*

Resumo

Neste capítulo, você deu uma olhada em como armazenar dados no servidor usando o Entity Framework Core e como expor esses dados usando REST e microsserviços. Você adicionou uma pizza serviço para o aplicativo PizzaPlace e depois continuei testando-o com o Postman.

Página 178

CAPÍTULO 6

Comunicação com microsserviços

No capítulo anterior, você construiu um microsserviço usando ASP.NET Core e Entity Framework Core para recuperar o cardápio de pizzas do servidor. Neste capítulo, você vai adicione suporte ao cliente Blazor para conversar com esse microsserviço. Você também completará o projeto adicionando suporte para completar o pedido.

Usando a classe HttpClient

Comece abrindo a solução MyFirstBlazor que você criou no primeiro capítulo. Você vai usar este projeto para examinar o modelo que foi criado para você. Você vai começar olhando para o lado do servidor da solução, depois o código do projeto compartilhado e, a seguir, o lado do cliente.

Examinando o Projeto de Servidor

Olhe para o projeto MyFirstBlazor.Server e procure o SampleDataController classe, que está na lista 6-1.

Listagem 6-1. A classe SampleDataController

```
using MyFirstBlazor.Shared;
usando Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
usando System.Linq;
using System.Threading.Tasks;
namespace MyFirstBlazor.Server.Controllers
```


P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_6

Página 179


```
{
   [Route ("api / [controlador]")]
   public class SampleDataController: Controller
   {
      string estática privada [] Resumos = novo []
      {
            "Congelando", "Preparando-se", "Frio", "Frio", "Suave",
            "Quente", "Balmy", "Quente", "Sufocante", "Ardente"
      };
      [HttpGet ("[ação]")]
      public IEnumerable <WeatherForecast> WeatherForecasts ()
      {
         var rng = novo Random ();
         retornar Enumerable.Range (1, 5)
```

```
                .Selecione (índice => novo WeatherForecast
            {
                Date = DateTime.Now.AddDays (índice),
                TemperaturaC = rng.Next (-20, 55),
                Resumo = Resumos [rng.Next (Summaries.Length)]
            });
        }
    }
}
```

A classe SampleDataController expõe um ponto de extremidade REST em / api / SampleData / WeatherForecasts para recuperar uma lista de objetos WeatherForecast. Desta vez o SampleDataController usa o atributo [Route ("api / [controller]")] para configurar o endpoint para ouvir genericamente um URI que contém o nome do controlador (sem o sufixo "Controlador") e, em seguida, usa o atributo [HttpGet ("[ação]")] para esperar o nome do método como a terceira parte do URI.

Para invocar este método, você deve usar um GET no api / SampleData / URI do WeatherForecasts, que você pode tentar com seu navegador (ou, se preferir, Postman). Execute a solução e digite o URI em seu navegador (não se esqueça que você terá um diferente número da porta), que resultará na Figura 6-1 (espere um clima diferente).

Página 180

Figura 6-1. *Invocar o serviço usando o navegador*

O método WeatherForecasts da Listagem 6-1 usa uma escolha aleatória de temperaturas e resumos para gerar essas previsões, o que é ótimo para uma demonstração.

Por que usar um projeto compartilhado?

Agora abra a classe WeatherForecast do projeto MyFirstBlazor.Shared, que é na Listagem 6-2 .

Listagem 6-2. A classe WeatherForecast compartilhada

```
using System;

namespace MyFirstBlazor.Shared
{
    public class WeatherForecast
    {
        public DateTime Date {get; definir; }
        public int TemperatureC {get; definir; }
        public string Resumo {get; definir; }
        public int TemperatureF
        => 32 + (int) (TemperaturaC / 0,5556);
    }
}
```

Esta classe WeatherForecast é direta, contendo a data da previsão, a temperatura em Celsius e Fahrenheit, e um Resumo, mas quero desenhar o seu atenção ao fato dessa turma residir no projeto Compartilhado. Este projeto compartilhado é usado tanto pelo servidor quanto pelo projeto do cliente.

Página 181

Se você já criou um aplicativo da web com JavaScript, deve estar familiarizado com o experiência de construir uma classe para o projeto de servidor, por exemplo em C #, e construir outra classe em JavaScript (ou Typescript) para o cliente. Você deve ter certeza de que ambos as classes são serializadas no mesmo formato JSON; caso contrário, você obterá erros de tempo de execução ou mesmo pior, perder dados! Se o modelo crescer, você deve atualizar ambas as classes novamente. Este é um ENORME problema de manutenção neste tipo de projeto, porque você corre o risco de atualizar apenas um lado em um dia de trabalho agitado.

Com o Blazor, você não sofre com isso porque o servidor e o cliente usam C #. E é por isso que existe um projeto compartilhado. Você coloca suas aulas aqui e elas são compartilhadas entre o servidor e o cliente, e então você os usa simplesmente adicionando uma referência a o projeto compartilhado. Adicionar outro dado significa atualizar uma classe compartilhada, que funciona facilmente! Você não precisa mais atualizar duas partes do código.

Olhando para o projeto do cliente

Agora olhe para o projeto MyFirstBlazor.Client. Dentro da pasta Pages você encontrará o Componente FetchData da listagem 6-3.

Listagem 6-3. O componente FetchData

```
@using MyFirstBlazor.Shared
@page "/ fetchdata"
@inject HttpClient Http

<h1> Previsão do tempo </h1>

<p> Este componente demonstra a obtenção de dados do servidor. </p>

@if (previsões == null)
{
<p><em>Carregando...</em> </p>
}
senão
{
<table class = "table">
    <thead>
        <tr>
            <th> Data </th>
```


Página 182


```
            <th> Temp. (C) </th>
            <th> Temp. (F) </th>
            <th> Resumo </th>
        </tr>
    </thead>
    <tbody>
        @foreach (previsão var nas previsões)
        {
            <tr>
```

```
            <td> @ forecast.Date.ToShortDateString () </td>
            <td> @ forecast.TemperatureC </td>
            <td> @ forecast.TemperatureF </td>
            <td> @ forecast.Summary </td>
        </tr>
    }
    </tbody>
</table>
}
@funções {
Previsões do WeatherForecast [];
substituição protegida assíncrona Tarefa OnInitAsync ()
{
    previsões = aguardar Http.GetJsonAsync <WeatherForecast []>
                        ("api / SampleData / WeatherForecasts");
}
}
```

Vamos examinar isso linha por linha. A primeira linha na lista6-3 adiciona um Razor @using declaração para o namespace do projeto compartilhado para o componente. Você precisa disso porque você usa a classe WeatherForecast do projeto compartilhado. Assim como em C #, você usa usando instruções no Razor para se referir a classes de outro namespace.

A segunda linha adiciona o caminho para o roteamento. Você verá o roteamento no próximo capítulo. No momento, você deve saber que quando o URI é / fetchdata o FetchData componente será mostrado no navegador.

Página 183

Na terceira linha, você injeta a instância HttpClient usando a sintaxe @inject de Navalha. A classe HttpClient é aquela que você usará para se comunicar com o servidor. Você vai aprender sobre a classe HttpClient em detalhes posteriormente neste capítulo.

> eu quero salientar que você nunca deve instanciar uma instância do Classe HttpClient você mesmo. Blazor configura a classe HttpClient de uma forma especial forma, e se você mesmo criar uma instância, ela simplesmente não funcionará como esperado! outra razão para não criar uma instância sozinho é que esta é uma dependência de o componente FetchData e os componentes nunca devem criar dependências eles mesmos!

Um pouco mais abaixo na Listagem 6-3, você encontrará uma instrução @if. Porque você busca os dados do servidor de forma assíncrona, o campo de previsões inicialmente mantenha uma referência nula. Portanto, se o campo de previsões não foi definido, você diz ao usuário para esperar. Se você tem uma rede lenta, pode ver isso acontecendo. Quando você testa seu Blazor aplicativo em sua própria máquina, a rede é rápida, mas você pode emular um rede usando o navegador (neste caso, usando o Google Chrome).

Como emular uma rede lenta no Chrome

Inicie seu projeto Blazor para que o navegador abra a página inicial. Agora abra o depurador ferramentas do navegador (no Windows, você faz isso pressionando F12) e selecione a rede guia como na Figura 6-2 . No lado direito, você deve ver uma lista suspensa que permite que você selecione o tipo de rede para emular. Selecione 3G lento.

Figura 6-2. *Usando o depurador do navegador Chrome para emular uma rede lenta*

166

Página 184

Em seguida, selecione a guia Buscar dados em seu site Blazor (caso você já esteja neste guia, selecione outra guia e, em seguida, a guia Buscar dados). Porque agora você está usando um lento rede, o feedback Carregando… aparecerá, conforme mostrado na Figura 6-3.

Figura 6-3. *O Carregando ... feedback com uma rede lenta*

depois de testar seu site Blazor com uma rede lenta, não se esqueça de selecionar Online no menu suspenso da Figura 6-2 para restaurar sua rede para o seu velocidade normal.

Se o campo de previsões contiver dados, seu arquivo Razor mostrará uma tabela com as previsões iterando sobre eles, como você pode ver na outra parte da Listagem 6-3 .

Na seção @functions do arquivo FetchData Razor. Primeiro, você declara um campo denominado previsões para conter uma matriz de instâncias do WeatherForecast. Inicialmente o O campo de previsões terá um valor nulo. Em seguida, você substitui o método OnInitAsync. Os componentes do Blazor têm dois métodos que são chamados quando o componente foi inicializado: OnInit e OnInitAsync. Porque você busca os dados do servidor usando um API assíncrona, você precisa colocar seu código em OnInitAsync. O método OnInitAsync é prefixado com a palavra-chave assíncrona do C #, o que torna mais fácil chamar APIs assíncronas com o aguarde a palavra-chave.

comunicação assíncrona significa que o cliente precisa esperar um bom tempo para que o resultado seja retornado. em vez de usar uma chamada que impedirá o Blazor de completando outra solicitação (congelando a interface do usuário), você usa o OnInitAsync método, que irá esperar em segundo plano pelo resultado.

167

Página 185

Use o Http.GetJsonAsync <WeatherForecast []> ("ALGUM URI") para invocar o ponto de extremidade GET do servidor no URI e você informa o método GetJsonAsync (usando genéricos) esperar uma série de objetos WeatherForecast. Quando o resultado volta do servidor, você coloca o resultado no campo de previsões e o Blazor cuidará do re-renderização a IU com seus novos dados, conforme mostrado na Figura 6-4.

Figura 6-4. *Exibindo os objetos WeatherForecast*

Compreendendo a classe HttpClient

Toda a comunicação entre o cliente e o servidor passa pela classe HttpClient.
Esta é a mesma classe que outros .NET frameworks usam e sua função é tornar o HTTP
pedido ao servidor e expor o resultado do servidor. Também permite que você
trocar dados binários ou outros dados formatados, mas no Blazor normalmente usamos JSON.

O Google definiu um protocolo mais eficiente chamado buffers de protocolo, que também é apoiado pelo Blazor. se você precisa enviar muitos dados, você pode querer olhar buffers de protocolo.

Os métodos HttpClientJsonExtensions

Para tornar muito mais fácil falar com microsserviços JSON, o Blazor oferece vários
métodos de extensão úteis que cuidam da conversão entre objetos .NET e JSON,
que você pode encontrar na classe HttpClientJsonExtensions. Eu aconselho você usar estes
métodos, então você não precisa se preocupar em serializar e desserializar JSON.

Página 186

GetJsonAsync

O método de extensão GetJsonAsync faz uma solicitação GET assíncrona para o
URI especificado. Sua assinatura está na Listagem6-4.

Listagem 6-4. A assinatura do método de extensão GetJsonAsync

```
public static Task <T> GetJsonAsync <T> (
                                        este HttpClient httpClient,
                                        string requestUri)
```

Por ser um método de extensão, você o chama como um método de instância normal no
Classe HttpClient, conforme mostrado na listagem 6-5.

isso também é verdadeiro para os outros métodos de extensão.

Listagem 6-5. Usando o método de extensão GetJsonAsync

```
previsões = aguardar Http.GetJsonAsync <WeatherForecast []>
                                ("api / SampleData / WeatherForecasts");
```

GetJsonAsync <T> espera que a resposta contenha JSON conforme especificado pelo
argumento genérico. Por exemplo, na Listagem 6-5, ele espera uma matriz de WeatherForecast
instâncias. Você invoca o método GetJsonAsync prefixando-o com a palavra-chave await,
o que o torna assíncrono. Não se esqueça de que você só pode usar a palavra-chave await em
métodos e funções lambda que são assíncronos.

Você sempre pode inspecionar a solicitação e a resposta usando o depurador do seu navegador. Corre
seu projeto Blazor e abra o depurador do navegador na guia Rede. Agora selecione o
Busque a guia de dados em seu site do Blazor para fazê-lo carregar os dados e olhar para o navegador do
Guia Rede, conforme mostrado na Figura 6-5.

169

Página 187

Capítulo 6 Comunicação com miCroserviços

Figura 6-5. *Inspecionar a rede usando o depurador do navegador*

Você sempre pode limpar a guia de rede de solicitações anteriores antes de fazer o
solicitação usando o botão limpar, que no Chrome se parece com um círculo com uma barra
através dele (o sinal de proibido).

Veja a entrada WeatherForecasts na Figura 6-5 ? Agora você pode clicar nessa entrada para olhar
a pedido e resposta. Vamos começar com a visualização da solicitação mostrada na Figura 6-6 .

Figura 6-6. *Usando a guia Visualizar para ver a resposta*

Usando a guia Visualização, você pode ver a resposta do servidor.
Se você quiser ver os cabeçalhos de solicitação e resposta, clique em Cabeçalhos
guia, conforme mostrado na Figura 6-7 .

170

Página 188

Capítulo 6 Comunicação com miCroserviços

Figura 6-7. *Usando a guia Cabeçalhos para ver a solicitação e a solicitação / resposta cabeçalhos*

Aqui você pode ver a URL da solicitação e o verbo GET (o método da solicitação). Isso também mostra o código de status HTTP 200 OK. Role para baixo para ver os cabeçalhos. Um dos cabeçalhos de resposta é Content-Type com um valor de application / json, que foi definido pelo servidor informando ao cliente para esperar JSON.

PostJsonAsync

O método de extensão PostJsonAsync faz uma solicitação POST com o argumento de conteúdo serializado no corpo da solicitação como JSON para o URI especificado. Sua assinatura está na Listagem6-6 .

Listagem 6-6. A assinatura do método PostJsonAsync

```
public static Task PostJsonAsync (este HttpClient httpClient,
                                  string requestUri,
                                  conteúdo do objeto)
```

Use este método se você não espera nenhum dado de volta do servidor. Há também uma versão genérica deste método que espera uma resposta JSON. Sua assinatura está em Listagem 6-7 . Este método pegará a resposta JSON e a desserializará como um T.

Listagem 6-7. A assinatura do método PostJsonAsync <T>

```
public static Task <T> PostJsonAsync <T> (
                                  este HttpClient httpClient,
                                  string requestUri,
                                  conteúdo do objeto)
```

171

Página 189

PutJsonAsync

O método de extensão PutJsonAsync faz uma solicitação PUT com o argumento de conteúdo serializado como JSON no corpo da solicitação para o URI especificado. Sua assinatura está na Listagem6-8 . Seu uso é muito semelhante ao PostJsonAsync; a única diferença é que usa o verbo PUT.

Listagem 6-8. A assinatura do método PutJsonAsync

```
public static Task PutJsonAsync (este HttpClient httpClient,
                                 string requestUri,
                                 conteúdo do objeto)
```

Use este método se você não espera nenhum dado de volta do servidor. Há também uma versão genérica deste método que espera uma resposta JSON. Sua assinatura está em Listagem 6-9 . Este método pegará a resposta JSON e a desserializará como um T.

Listagem 6-9. A assinatura do método PutJsonAsync <T>

```
public static Task <T> PutJsonAsync <T> (
                                 este HttpClient httpClient,
                                 string requestUri,
                                 conteúdo do objeto)
```

SendJsonAsync

Com SendJsonAsync, você pode usar qualquer outro verbo compatível com HTTP para fazer uma solicitação. Sua assinatura está na Listagem 6-10 . A ideia é que você passe o verbo como parâmetro do método.

Listagem 6-10. A assinatura do método SendJsonAsync

```
public static Task SendJsonAsync (este HttpClient httpClient,
                                  Método HttpMethod,
                                  string requestUri,
                                  conteúdo do objeto)
```

Use este método se você não espera nenhum dado de volta do servidor. Há também uma versão genérica deste método que espera uma resposta JSON. Sua assinatura está em Listagem 6-11. Este método pegará a resposta JSON e a desserializará como um T.

Página 190

Listagem 6-11. A assinatura do método SendJsonAsync <T>

```
public static Task <T> SendJsonAsync <T> (
                                  este HttpClient httpClient,
                                  Método HttpMethod,
                                  string requestUri,
                                  conteúdo do objeto)
```

Recuperando Dados do Servidor

Agora você está pronto para implementar os serviços apresentados anteriormente. Abra o Solução PizzaPlace e procure no projeto Blazor.Client por Startup.cs, que é mostrado na Listagem 6-12.

Listagem 6-12. Aula de inicialização do seu projeto Blazor

```
usando Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Shared;
namespace PizzaPlace.Client
{
   public class Startup
   {
      public void ConfigureServices (serviços IServiceCollection)
      {
         services.AddTransient <IMenuService,
                                HardCodedMenuService> ();
         services.AddTransient <IOrderService,
                                ConsoleOrderService> ();
      }
      public void Configure (aplicativo IBlazorApplicationBuilder)
      {
         app.AddComponent <App> ("app");
      }
   }
}
```

Página 191

No método ConfigureServices, você adicionou dois serviços, HardCodedMenuService e ConsoleOrderService. Vamos substituir essas implementações falsas por serviços reais que falam com o servidor.

Com o Visual Studio, clique com o botão direito do mouse no projeto PizzaPlace.Client e selecione Add ➤ New Pasta no menu suspenso. Com o Code, clique com o botão direito do mouse no projeto PizzaPlace.Client e selecione Nova pasta. Nomeie essa pasta como Serviços. Agora adicione uma nova classe a esta pasta chamado MenuService, que pode ser encontrado na Listagem 6-13 .

novamente, você está aplicando o princípio da responsabilidade única, onde encapsula como você fala com o servidor em um serviço. desta forma, você pode facilmente substituir este implementação com outro em caso de necessidade.

Listagem 6-13. A classe MenuService

```
usando Microsoft.AspNetCore.Blazor;
using PizzaPlace.Shared;
usando System.Linq;
usando System.Net.Http;
using System.Threading.Tasks;
namespace PizzaPlace.Client.Services
{
    public class MenuService: IMenuService
    {
        private HttpClient httpClient;
        public MenuService (HttpClient httpClient)
        {
            this.httpClient = httpClient;
        }
        public async Task <Menu> GetMenu ()
        {
            var pizzas =
                aguarde httpClient.GetJsonAsync <Pizza []> ("/ pizzas");
            retornar novo Menu {Pizzas = pizzas.ToList ()};
        }
    }
}
```

174

Página 192

Você começa adicionando um construtor a esta classe levando a dependência do MenuService no HttpClient e você o armazena em um campo denominado httpClient. Então você implementa o método GetMenu da interface IMenuService, onde você fala com o servidor chamando o GetJsonAsync no endpoint do servidor / pizza. Observe que o endpoint / pizza é relativo à base do site (<base href = "/" />), que pode ser encontrada no arquivo index.html. Como o serviço MenuService retorna um menu, e não uma lista de pizzas, você envolve o lista de pizzas que você obteve do servidor em um objeto Menu. É isso!

usando o princípio da responsabilidade única resulta em muitas classes pequenas, que são mais fáceis de entender, manter e testar.

Você tem o serviço; agora você precisa dizer à injeção de dependência para usar o MenuService. No método ConfigureServices da classe Startup, substitua-o conforme mostrado na Listagem 6-14 .

Listagem 6-14. Substituindo o HardCodedMenuService pelo MenuService

```
usando Microsoft.AspNetCore.Blazor.Builder;
```

```
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Client.Services;
using PizzaPlace.Shared;
namespace PizzaPlace.Client
{
    public class Startup
    {
        public void ConfigureServices (serviços IServiceCollection)
        {
            services.AddTransient <IMenuService, MenuService> ();
            services.AddTransient <IOrderService,
                                        ConsoleOrderService> ();
        }
        public void Configure (aplicativo IBlazorApplicationBuilder)
        {
            app.AddComponent <App> ("app");
        }
    }
}
```

175

Página 193

Execute seu projeto. Você deve ver a lista de pizzas (recuperada de seu banco de dados) como na Figura 6-8 !

Figura 6-8. *O aplicativo PizzaPlace mostrando as pizzas do banco de dados*

Você provavelmente verá primeiro um menu vazio, especialmente em uma rede lenta. Isso pode confundir alguns clientes, então vamos adicionar alguma IU para dizer ao cliente para esperar um pouco. Atualizar pizzalist.cshtml para se parecer com a Listagem 6-15 .

Listagem 6-15. Adicionando uma IU de carregamento ao componente PizzaList

```
<h1> @Title </h1>

@if (Menu == null || Menu.Pizzas == null
                        || Menu.Pizzas.Count == 0)
{
    <div style = "altura: 20vh;" classe = "pt-3">
        <div class = "mx-left pt-3" style = "width: 200px">
            <div class = "progress">
                <div class = "progress-bar bg-perigo
                                barra de progresso listrada
                                barra de progresso animada w-100 "
                        role = "progressbar"
                        aria-valuenow = "100" aria-valuen = "0"
                        aria-valuemax = "100"> </div>
            </div>
        </div>
```

176

Página 194


```
    </div>
}
senão
{
    @foreach (var pizza em Menu.Pizzas)
    {
        <PizzaItem Pizza = "@ pizza" ButtonTitle = "Pedido"
            ButtonClass = "btn btn-success" Selecionado = "@ ((p) => Selecionado (p))" />
    }
}

@funções {

[Parâmetro]
string protegida Título {get; definir; }
[Parâmetro]
protegido Menu Menu {get; definir; }
[Parâmetro]
ação protegida <Pizza> selecionada {obter; definir; }
}
```

Se o menu ainda não foi carregado, ele exibirá uma barra de progresso como na Figura 6-9 .

Figura 6-9. *Mostrando uma barra de progresso de carregamento ao carregar o menu*

Armazenamento de alterações

Agora vamos armazenar o pedido do cliente. Porque você não tem um microsserviço
ainda para armazenar o pedido, você vai construir isso primeiro e, em seguida, vai implementar o cliente
serviço para enviar o pedido ao servidor.

Página 195

Atualizando o Banco de Dados com Pedidos

O que é um pedido? Cada cliente tem um pedido e cada pedido tem uma ou mais pizzas.
Uma pizza pode pertencer a mais de um pedido, o que pode resultar em um problema específico: você
precisa de uma relação muitos para muitos entre pizzas e pedidos. Em bancos de dados relacionais,
isso é feito adicionando uma tabela entre pedidos e pizzas, que você mapeará usando um
Classe PizzaOrder, conforme mostrado na Figura 6-10 .

Figura 6-10. *Modelando os relacionamentos*

entidade Framework Core 2.1 não tem suporte para ocultar esta tabela extra, então você precisa fazer isso manualmente. em versões futuras, a Microsoft irá (esperançosamente) adicionar esse recurso.

Adicione uma nova classe ao projeto PizzaPlace.Shared chamada PizzaOrder, conforme mostrado em Listagem 6-16.

Listagem 6-16. A classe PizzaOrder

```
namespace PizzaPlace.Shared
{
    classe pública PizzaOrder
    {
        public int Id {get; definir; }
        Ordem pública Ordem {get; definir; }
        Pizza pública Pizza {get; definir; }
    }
}
```

Em seguida, adicione uma nova classe chamada Order ao projeto PizzaPlace.Shared, conforme mostrado em Listagem 6-17.

Página 196

Listagem 6-17. The Order Class

```
using System.Collections.Generic;

namespace PizzaPlace.Shared
{
    Ordem de classe pública
    {
        public int Id {get; definir; }

        Cliente público Cliente {get; definir; }

        public int CustomerId {get; definir; }

        Lista pública <PizzaOrder> PizzaOrders {get; definir; }

        Preço total decimal público {obter; definir; }
    }
}
```

Atualize a classe Customer do projeto PizzaPlace.Shared adicionando um Order a ele, como na Listagem 6-18 .

Listagem 6-18. A Classe do Cliente

```
using System;
using System.Collections;
using System.ComponentModel;
usando System.Runtime.CompilerServices;
namespace PizzaPlace.Shared
{
    public class Customer: INotifyDataErrorInfo,
                                        INotifyPropertyChanged
    {
        public int Id {get; definir; }

        nome da string privada;
```

```
        public string Name
        {
```

179

Página 197


```
            get {return name; }
            definir {nome = valor; OnPropertyChanged (); }
        }

        rua string privada;

        rua string pública
        {
            get {return street; }
            definir {rua = valor; OnPropertyChanged (); }
        }

        cidade privada string;

        string pública cidade
        {
            get {return city; }
            definir {cidade = valor; OnPropertyChanged (); }
        }

        Ordem pública Ordem {get; definir; }

    // O resto da classe omitido para maior clareza

}
```

E você precisa adicionar uma nova propriedade PizzaOrders à classe Pizza como na Listagem 6-19 .

Listagem 6-19. The Pizza Class

```
pizza de classe pública
{
    pizza pública () {}
    Pizza pública (int id, nome da string, preço decimal,
                   Picante picante)
    {
        ...
    }
    public int Id {get; definir; }
    public string Name {get; definir; }
```

180

Página 198


```
        Preço decimal público {get; definir; }
        picante público Picante {obter; definir; }
        Lista pública <PizzaOrder> PizzaOrders {get; definir; }
    }
```

Agora você pode adicionar essas tabelas à classe PizzaPlaceDbContext, que pode ser encontrada na Listagem 6-20 .

Listagem 6-20. A classe PizzaPlaceDbContext atualizada

```
usando Microsoft.EntityFrameworkCore;
using PizzaPlace.Shared;
namespace PizzaPlace.Server
{
   public class PizzaPlaceDbContext: DbContext
   {
      public PizzaPlaceDbContext (
         Opções DbContextOptions <PizzaPlaceDbContext>)
         : base (opções)
      {}

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

      public DbSet <Customer> Clientes {get; definir; }

      public DbSet <Order> Pedidos {get; definir; }

      public DbSet <PizzaOrder> PizzaOrders {get; definir; }

      protected override void OnModelCreating (
                                          ModelBuilder modelBuilder)
      {
         base.OnModelCreating (modelBuilder);
         var pizzaEntity = modelBuilder.Entity <Pizza> ();
         pizzaEntity.HasKey (pizza => pizza.Id);
         pizzaEntity.Property (pizza => pizza.Price)
                        .HasColumnType ("dinheiro");
```


Página 199


```
      var customerEntity = modelBuilder.Entity <Customer> ();
      customerEntity.HasKey (cliente => cliente.Id);
      customerEntity.HasOne (customer => customer.Order)
                        .WithOne (pedido => pedido.Cliente)
                        .HasForeignKey <Order> (
                           pedido => pedido.CustomerId);
      var orderEntity = modelBuilder.Entity <Order> ();
      orderEntity.HasKey (pedido => pedido.Id);
      orderEntity.HasMany (order => order.PizzaOrders)
                     .WithOne (pizzaOrder => pizzaOrder.Order);
      pizzaEntity.HasMany (pizza => pizza.PizzaOrders)
                     .WithOne (pizzaOrder => pizzaOrder.Pizza);
   }
  }
}
```

Aqui você adicionou as tabelas Clientes, Pedidos e PizzaOrders, e no Método OnModelCreating, você explica ao Entity Framework Core como as coisas deveriam ser mapeado.

Um cliente tem um ID de chave primária e uma relação um-para-um com um pedido. Quando usando uma relação um-para-um, o Entity Framework Core precisa saber de que lado está o *mestre* na relação, e é por isso que você precisa adicionar uma chave estrangeira à classe Order com o método HasForeighKey <Order>.

Um pedido tem um ID de chave primária e um relacionamento muitos-para-um com um PizzaOrder (um pedido pode ter muitos PizzaOrders, e cada PizzaOrder pertence a um pedido).

Finalmente, você indica que uma Pizza pode pertencer a muitos PizzaOrders, e um PizzaOrder tem uma Pizza. Desta forma, cada Order pode ter muitas instâncias de Pizza, e cada Pizza pode ter várias instâncias de Order.

Construa seu projeto e corrija quaisquer erros do compilador que você possa ter.

Agora é hora de criar outra migração. Esta migração irá atualizar seu banco de dados com suas novas tabelas. No Visual Studio, abra o Console do Gerenciador de Pacotes (que você pode encontrar em Exibir ➤ Outras janelas ➤ Console do gerenciador de pacotes). Com o Código, abra o terminal integrado.

Mude o diretório para o projeto PizzaPlace.Server

Página 200

Agora digite o seguinte comando:

```
migrações dotnet ef add HandlingOrders
```

Isso criará uma migração para seu novo esquema de banco de dados.

Aplique a migração ao seu banco de dados digitando o seguinte comando:

```
atualização do banco de dados dotnet ef
```

Isso conclui a parte do banco de dados.

Criação do microsserviço do pedido

É hora de criar o microsserviço para receber pedidos. Com o Visual Studio, clique com o botão direito do mouse no Pasta Controllers do projeto PizzaPlace.Server e selecione New ➤ Controller.
Selecione um Empty API Controller e nomeie-o OrdersController. Com o Código, clique com o botão direito a pasta Controllers do projeto PizzaPlace.Shared e selecione New File, nomeando-o OrdersController. Esta classe pode ser encontrada na Listagem6-21 .

Listagem 6-21. A classe OrdersController

```
using System.Collections.Generic;
usando System.Linq;
usando Microsoft.AspNetCore.Mvc;
using PizzaPlace.Shared;

namespace PizzaPlace.Server.Controllers
{
    [ApiController]
    public class OrdersController: ControllerBase
    {
        private PizzaPlaceDbContext db;
        public OrdersController (PizzaPlaceDbContext db)
        {
            this.db = db;
        }
        [HttpPost ("/ pedidos")]
        public IActionResult CreateOrder ([FromBody] Basket basket)
        {
```


Página 201


```
            var cliente = basket.Customer;
            var pedido = novo pedido () {
                PizzaOrders = nova lista <PizzaOrder> ()
            };
```

```
            pedido do cliente = pedido;
            foreach (var pizzaId in basket.Orders)
            {
                var pizza = db.Pizzas.Single (p => p.Id == pizzaId);
                order.PizzaOrders.Add (new PizzaOrder {
                    Pizza = pizza, Pedido = pedido
                });
            }
            order.TotalPrice =
                order.PizzaOrders.Sum (po => po.Pizza.Price);
            db.Customers.Add (cliente);
            db.SaveChanges ();
            retornar Ok ();
        }
    }
}
```

Seu OrdersController precisa de um PizzaPlaceContextDb, então você adiciona um construtor tomar a instância e você deixa a injeção de dependência cuidar do resto. Para criar um novo pedido, você usa o verbo POST para o método CreateOrder tomando uma instância de Basket no corpo da solicitação. Após o recebimento de uma instância de cesta, você cria o cliente e pedido. Em seguida, você define o pedido do cliente. Não há necessidade de definir o cliente do pedido propriedade; O Entity Framework Core cuidará da relação inversa para você.
Em seguida, você preenche a coleção PizzaOrders do pedido com pizzas. Então você calcula o preço total do pedido e você salva todo o Cliente ➤ Pedido ➤ PizzaOrdens ➤ Rede de pizzarias adicionando a entidade raiz Customer a PizzaPlaceDbContext e chamando SaveChanges. É isso. O Entity Framework Core faz todo o trabalho de armazenamento dos dados!

Conversando com o microsserviço do pedido

Adicione uma nova classe chamada OrderService à pasta Services do PizzaPlace.Client projeto. Este OrderService usa uma solicitação POST para o servidor, conforme mostrado na Listagem 6-22.

Página 202

Listagem 6-22. A classe OrderService

```
using PizzaPlace.Shared;
using System.Threading.Tasks;
usando Microsoft.AspNetCore.Blazor;
usando System.Net.Http;

namespace PizzaPlace.Client.Services
{
    public class OrderService: IOrderService
    {
        private HttpClient httpClient;
        public OrderService (HttpClient httpClient)
        {
            this.httpClient = httpClient;
        }
        Public assíncrono Task PlaceOrder (Basket basket)
        {
            aguarde httpClient.PostJsonAsync ("/ pedidos", cesta);
        }
    }
}
```

Primeiro, você adiciona um construtor à classe OrderService, tomando o HttpClient dependência, que você armazena no campo httpClient da classe OrderService. Próximo, você implementa a interface IOrderService adicionando o método PlaceOrder, tomando

uma cesta como parâmetro. Finalmente, você invoca o método assíncrono PostJsonAsync usando a palavra-chave await.

Agora abra a classe Startup do projeto PizzaPlace.Client e substitua o Classe ConsoleOrderService com sua nova classe OrderService, conforme mostrado na listagem 6-23 .

Listagem 6-23. Configurando injeção de dependência para usar a classe OrderService

```
usando Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Client.Services;
using PizzaPlace.Shared;
namespace PizzaPlace.Client
```

185

Página 203


```
{
   public class Startup
   {
      public void ConfigureServices (serviços IServiceCollection)
      {
         services.AddTransient <IMenuService, MenuService> ();
         services.AddTransient <IOrderService, OrderService> ();
      }
      public void Configure (aplicativo IBlazorApplicationBuilder)
      {
         app.AddComponent <App> ("app");
      }
   }
}
```

Execute o aplicativo PizzaPlace e faça um pedido de algumas pizzas. Agora abra o SQL Server Object Explorer no Visual Studio (ou SQL Operations Studio) e examine as tabelas Clientes, Pedidos e PizzaOrders. Eles devem conter o seu nova ordem.

Resumo

Neste capítulo, você aprendeu que no Blazor você fala com o servidor usando o HttpClient , chamando os métodos de extensão GetJsonAsync e PostJsonAsync. Você também aprendeu que você deve encapsular a chamada do servidor usando uma classe de serviço do lado do cliente para que você pode facilmente mudar a implementação mudando o tipo de serviço usando dependência injeção.

186

Página 204

CAPÍTULO 7

Aplicativos de página única e roteamento

Blazor é uma estrutura .NET que você usa para criar aplicativos de página única, assim como você pode usar estruturas JavaScript populares, como Angular, React e VueJs. Mas o que é um SPA? Neste capítulo, você usará o roteamento para pular entre as diferentes seções de um SPA e enviar dados entre diferentes componentes.

O que é um aplicativo de página única?

No início da Web havia apenas páginas estáticas. Uma *página estática* é um arquivo HTML em algum lugar no servidor que é enviado de volta ao navegador mediante solicitação. Mais tarde veio o ascensão de páginas dinâmicas. Quando um navegador solicita uma *página dinâmica* , o servidor executa um programa para construir o HTML na memória e enviar o HTML de volta para o navegador (este HTML nunca é armazenado em disco; claro, o servidor pode armazenar o HTML gerado em seu cache para rápido recuperação posterior). As páginas dinâmicas são flexíveis na forma como o mesmo código pode gerar milhares de páginas diferentes, recuperando dados de um banco de dados e usando-os para construir a página. Mas ainda há um problema de usabilidade. Cada vez que seu usuário clica em um link, o servidor deve gerar a próxima página do zero e enviá-la ao navegador para renderização. Esta resulta em um período de espera perceptível e, claro, o navegador processa a página inteira novamente.

Então as páginas da web começaram a usar JavaScript para recuperar partes da página quando o usuário interage com a IU. Um dos primeiros exemplos dessa técnica foi o Outlook da Microsoft Aplicativo da web. Este aplicativo da web se parece com o Outlook, um aplicativo de desktop, com suporte para todas as interações do usuário que você espera de um aplicativo de desktop. Do Google Gmail é outro exemplo. Eles agora são conhecidos como *aplicativos de página única* . Com SPAs certas seções da página da web são substituídas em tempo de execução, dependendo do usuário interação. Se você clicar em um e-mail, a seção principal da página é substituída pelo e-mail visualizar. Se você clicar em sua caixa de entrada, a seção principal será substituída por uma lista de e-mails, etc.

P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_7

Página 205

Um SPA é um aplicativo da web que substitui certas partes da UI sem recarregar a página completa. SPAs usam JavaScript para implementar essa manipulação do navegador árvore de controle (também conhecida como DOM) e a maioria deles consiste em uma IU fixa e um elemento placeholder onde o conteúdo é sobrescrito dependendo de onde o usuário cliques. Uma das principais vantagens de usar um SPA é que você pode fazer um SPA state-full. Isso significa que você pode manter as informações carregadas pelo aplicativo na memória. Você irá veja um exemplo neste capítulo.

Usando componentes de layout

Vamos começar com a parte fixa de um SPA. Cada aplicativo da web contém elementos de interface do usuário que você pode encontrar em todas as páginas, como cabeçalho, rodapé, direitos autorais, menu, etc. Copiar e colar esses elementos para cada página dariam muito trabalho e exigiriam a atualização de cada página se um desses elementos precisava ser alterado. Os desenvolvedores não gostam de fazer isso, todo framework para construção de sites tem uma solução para isso. Por exemplo, ASP. NET WebForms usa páginas mestras, ASP.NET MVC tem páginas de layout. Blazor também tem um mecanismo para este chamado componentes de layout.

Componentes de layout do Blazor

Os componentes do layout são componentes do Blazor. Qualquer coisa que você possa fazer com um regular componente que você pode fazer com um componente de layout, como injeção de dependência, dados ligação e aninhamento de outros componentes. A única diferença é que eles devem herdar da classe BlazorLayoutComponent.

A classe BlazorLayoutComponent define uma propriedade Body como na Listagem 7-1.

Listagem 7-1. A classe BlazorLayoutComponent

```
namespace Microsoft.AspNetCore.Blazor.Layouts
{
   classe abstrata pública BlazorLayoutComponent
                                    : BlazorComponent
   {
      BlazorLayoutComponent protegido ();
```


Página 206


```
      [Parâmetro]
      Protegido RenderFragment Body {get; }
   }
}
```

Como você pode ver na Listagem 7-1 , a classe BlazorLayoutComponent herda de a classe BlazorComponent. É por isso que você pode fazer as mesmas coisas que com o normal componentes. Vejamos um exemplo. Abra a solução MyFirstBlazor anterior capítulos. Agora veja o componente MainLayout.cshtml no MyFirstBlazor. Pasta compartilhada do cliente, que você encontrará na listagem 7-2.

Listagem 7-2. MainLayout.cshtml

```
@inherits BlazorLayoutComponent

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

<div class = "main">
      <div class = "linha superior px-4">
            <a href = "http://blazor.net" target = "_ blank"
                  class = "ml-md-auto"> Sobre </a>
      </div>

      <div class = "content px-4">
            @Corpo
      </div>
</div>
```

Na primeira linha, o componente MainLayout declara que herda de BlazorLayoutComponent. Então você vê uma barra lateral e o elemento principal, com o principal vinculação de dados do elemento à propriedade Body herdada.

Na figura 7-1 você pode ver a barra lateral no lado esquerdo (contendo os links para o diferentes componentes) e a área principal no lado direito com o @Body enfatizado com um retângulo preto (que acrescentei à figura). Clicar em Home, Counter ou Fetch O link de dados na barra lateral substituirá a propriedade Corpo pelo componente selecionado, atualizar a IU sem recarregar a página inteira.

Página 207

Figura 7-1. *O componente* MainLayout

Selecionando um Componente @layout

Cada componente pode selecionar qual layout usar, informando o nome do layout componente com a diretiva @layout. Por exemplo, comece copiando o MainLayout. arquivo cshtml para MainLayout2.cshtml. Isso irá gerar um novo componente de layout chamado MainLayout2, inferido a partir do nome do arquivo. Altere o texto do link Sobre para Layout como em Listagem 7-3.

Listagem 7-3. Um segundo componente de layout

```
@inherits BlazorLayoutComponent

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

<div class = "main">
    <div class = "linha superior px-4">
        <a href = "http://blazor.net" target = "_ blank"
          class = "ml-md-auto"> Layout </a>
    </div>
```


Página 208


```
    <div class = "content px-4">
        @Corpo
    </div>
</div>
```

Agora abra o componente Counter e adicione um @layout como na Listagem 7-4 .

Listagem 7-4. Escolhendo um layout diferente com @layout

```
@page "/ counter"
@layout MainLayout2

<h1> Contador </h1>

<p> Contagem atual: @currentCount </p>

<button class = "btn btn-primary" onclick = "@ IncrementCount"> Clique aqui </button>

@funções {
```

```
    int currentCount = 0;

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

Execute o aplicativo e observe a mudança de layout (o texto do link no canto superior direito canto) ao alternar entre Home e Counter.

Você também pode usar o LayoutAttribute se estiver construindo seu componente completamente em código.

_ViewImports.cshtml

A maioria dos componentes usará o mesmo layout. Em vez de copiar o mesmo @layout a cada página, você também pode adicionar um arquivo _ViewImports.cshtml à mesma pasta como seus componentes. Abra a pasta Pages do projeto MyFirstBlazor.Client e veja o arquivo _ViewImports.cshtml, que pode ser encontrado na Listagem 7-5.

Página 209

Listagem 7-5. _ViewImports.cshtml

```
@layout MainLayout
```

Qualquer componente que não declare explicitamente um componente @layout usará o Componente MainLayout. Qualquer coisa que é compartilhada entre todos os seus componentes pode ser coloque em _ViewImports.cshtml, especialmente instruções @using. Um componente sempre pode sobrescrever @layout adicionando explicitamente o layout como na Listagem 7-4 .

Layouts aninhados

Os componentes de layout também podem ser aninhados. Você pode definir o MainLayout para conter toda a IU que é compartilhada entre todos os componentes e, em seguida, defina um layout aninhado para ser usado por um subconjunto desses componentes. Por exemplo, adicione um novo Razor View chamado NestedLayout.cshtml para a pasta Compartilhada e substitua seu conteúdo por Listagem 7-6.

Listagem 7-6. Um layout aninhado simples

```
@inherits BlazorLayoutComponent
@layout MainLayout

<div class = "container-fluid">
    <div class = "row bg-primary text-white">
        <div class = "col-sm-12">
            <h2> Este é um layout aninhado </h2>
        </div>
    </div>
    <div class = "row">
        <div class = "col-sm-12">
            @Corpo
        </div>
    </div>
</div>
```

Página 210

Para construir um layout aninhado você @inherit from BlazorLayoutComponent e definir seu @layout para outro layout, por exemplo MainLayout. Agora faça o componente Contador use este layout aninhado como na Listagem 7-7 .

Listagem 7-7. O componente do contador está usando o layout aninhado.

```
@page "/ counter"
@layout NestedLayout

<h1> Contador </h1>

<p> Contagem atual: @currentCount </p>

<button class = "btn btn-primary" onclick = "@ IncrementCount"> Clique aqui </button>

@funções {
int currentCount = 0;

void IncrementCount ()
{
    currentCount ++;
}

sobrescrito protegido void OnInit ()
{
    base.OnInit ();
}
}
```

Execute seu aplicativo e selecione o componente Contador, conforme mostrado na Figura 7-2 .

Página 211

Figura 7-2. *O componente Contador usando o layout aninhado*

Compreendendo o roteamento

Os aplicativos de página única usam o roteamento para selecionar qual componente é escolhido para preencher o propriedade Body do componente de layout. O roteamento é o processo de correspondência do URI do navegador a uma coleção de *modelos* de *rota* e é usado para selecionar o componente a ser mostrado em tela. É por isso que cada componente usa uma diretiva @page para definir o modelo de rota para informar ao roteador qual componente escolher.

Instalando o Roteador

Quando você cria uma solução Blazor do zero, o roteador já está instalado, mas vamos dê uma olhada em como isso é feito. Abra App.cshtml. Este componente de aplicativo tem apenas um componente, o componente Roteador, conforme mostrado na listagem 7-8.

Listagem 7-8. O componente do aplicativo que contém o roteador

```
<Router AppAssembly = typeof (Program) .Assembly />
```

O roteador irá procurar por todos os componentes que possuem o RouteAttribute (a @page diretiva é compilada em um RouteAttribute) e escolha o componente que corresponde o URI do navegador atual. Você verá como definir este RouteAttribute um pouco mais tarde neste capítulo, mas primeiro você precisa olhar para o componente NavMenu.

Página 212

O componente NavMenu

Revise o componente MasterLayout da Listagem 7-2 . Na quarta linha você vai consulte o componente NavMenu. Este componente contém os links para navegar entre componentes. Abra a solução MyFirstBlazor e procure o componente NavMenu em a pasta compartilhada, que é repetida na listagem 7-9.

Listagem 7-9. O componente NavMenu

```
<div class = "linha superior pl-4 navbar navbar-dark">
    <a class="navbar-brand" href=""> MyFirstBlazor </a>
    <button class = "navbar-toggler" onclick = @ ToggleNavMenu>
        <span class = "navbar-toggler-icon"> </span>
    </button>
</div>

<div class = @ (collapseNavMenu? "collapse": null) onclick = @ ToggleNavMenu>
    <ul class = "nav flex-column">
        <li class = "nav-item px-3">
            <NavLink class = "nav-link" href = ""
                Match = NavLinkMatch.All>
                <span class = "oi oi-home" aria-hidden = "true"> </span>
                Casa
            </NavLink>
        </li>
        <li class = "nav-item px-3">
            <NavLink class = "nav-link" href = "counter">
                <span class = "oi oi-plus" aria-hidden = "true"> </span>
                Balcão
            </NavLink>
        </li>
        <li class = "nav-item px-3">
            <NavLink class = "nav-link" href = "fetchdata">
```

```
<span class = "oi oi-list-rich"
        aria-hidden = "true"> </span>
Obter dados
</NavLink>
```

195

Página 213


```
        </li>
    </ul>
</div>

@funções {
bool collapseNavMenu = true;

void ToggleNavMenu ()
{
    collapseNavMenu =! collapseNavMenu;
}
}
```

A primeira parte da listagem 7-9 contém um botão de alternância que permite ocultar e mostrar o menu de navegação. Este botão só é visível em monitores com largura estreita (por exemplo, monitores móveis). Se você quiser dar uma olhada nele, execute seu aplicativo e defina a largura do navegador menor até ver o *botão de hambúrguer* no canto superior direito, como na Figura 7-3 . Clique no para mostrar o menu de navegação e clique nele novamente para ocultar o menu novamente.

Figura 7-3. *Seu aplicativo em uma tela estreita mostra o botão de alternância*

A marcação restante contém o menu de navegação, que consiste no NavLink componentes. Vejamos o componente NavLink.

196

Página 214

O componente NavLink

O componente NavLink é uma versão especializada de um elemento âncora <a/> usado para criando links de navegação. Quando o URI do navegador corresponde à propriedade href do O NavLink aplica um estilo CSS (a classe CSS ativa se você quiser personalizá-lo) a si mesmo para informá-lo de que é a rota atual. Por exemplo, veja a Listagem 7-10.

Listagem 7-10. O NavLink do Contador de Rota

```
<NavLink class = "nav-link" href = "counter">
    <span class = "oi oi-plus" aria-hidden = "true"> </span> Contador
</NavLink>
```

Quando o URI do navegador termina com / counter (ignorando coisas como strings de consulta) isso O NavLink aplicará o estilo ativo. Vejamos outro na Listagem7-11.

Listagem 7-11. O NavLink da rota padrão

```
<NavLink class = "nav-link" href = "" Match = NavLinkMatch.All>
    <span class = "oi oi-home" aria-hidden = "true"> </span> Página inicial
</NavLink>
```

Quando o URI do navegador está vazio (exceto para o URL do site), o NavLink de A Listagem 7-11 estará ativa. Mas aqui você tem um caso especial. Normalmente NavLink os componentes correspondem apenas ao final do URI. Por exemplo, / counter / 55 corresponde ao NavLink da Listagem 7-10. Mas com um URI vazio, isso corresponderia a tudo! Isso é por que, no caso especial de um URI vazio, você precisa dizer ao NavLink para corresponder ao todo URI. Você faz isso com a propriedade Match, que por padrão é definida como NavLinkMatch. Prefixo. Se você deseja corresponder a todo o URI, use NavLinkMatch.All como na Listagem7-11 .

Configurando o modelo de rota

O componente de roteamento do Blazor examina o URI do navegador e procura por o modelo de rota de um componente para corresponder. Mas como você define a rota de um componente modelo? Abra o componente do contador mostrado na listagem7-4 . No topo deste arquivo está o @page "/ counter" diretiva. Ele define o modelo de rota. Um modelo de rota é uma string que pode conter parâmetros, que você pode usar em seu componente.

Página 215

Usando parâmetros de rota

Você também pode especificar parâmetros no modelo de rota. Passando parâmetros no rota, você pode alterar o que é exibido no componente. Você poderia passar o id de um produto, procure os detalhes do produto com o id e use-o para exibir o produto detalhes. Vejamos um exemplo. Altere o componente do contador para se parecer com a Listagem7-12 .

Listagem 7-12. Definindo um modelo de rota com um parâmetro

```
@page "/ counter"
@page "/ counter / {CurrentCount: int}"

@layout NestedLayout

<h1> Contador </h1>

<p> Contagem atual: @CurrentCount </p>

<button class = "btn btn-primary"
            onclick = "@ IncrementCount"> Clique aqui </button>

@funções {

    [Parâmetro]
    protegido int CurrentCount {get; definir; }

    void IncrementCount ()
    {
        CurrentCount ++;
    }
}
```

A Listagem 7-12 adiciona o modelo de rota @page "/ counter / {CurrentCount: int}". Isso diz o componente do roteador para corresponder a um URI como / counter / 55 e colocar o número no Parâmetro CurrentCount do seu componente Contador. Você envolve os parâmetros em colchetes. O componente Roteador colocará o valor da rota na propriedade com o mesmo nome. Você também pode especificar vários parâmetros. Blazor não permite que você especifique parâmetros padrão, e é por isso que você precisa especificar dois modelos de rota. A primeira rota modelo escolherá o componente Contador, com CurrentCount definido com seu valor padrão de 0. O segundo modelo de rota escolherá o componente Contador e definirá o CurrentCount parâmetro para um valor int. Deve ser um int por causa da restrição de rota int.

Página 216

Filtrar URIs com restrições de rota

Assim como as rotas no ASP.NET MVC Core, você pode usar *restrições de rota* para limitar o tipo de parâmetro para combinar. Por exemplo, se você fosse usar o URI / counter / Blazor, a rota modelo não corresponderia porque o parâmetro não contém um valor inteiro e o o roteador não encontrou nenhum componente compatível.

As restrições são obrigatórias mesmo se você não estiver usando parâmetros de string; por outro lado o roteador não converte o parâmetro para o tipo apropriado. Você especifica a restrição por anexando-o usando dois pontos, por exemplo "/ counter / CurrentCount: int".

Uma lista de outras restrições pode ser encontrada na Tabela 7-1.

Tabela 7-1. *Restrições de roteamento*

Restrições de rota
Bool
Data hora
Decimal
Dobro
Flutuador
Guid
Int
Longo

Se você estiver construindo seus componentes como componentes C # puros, aplique o RouteAttribute para sua classe com o modelo de rota como um argumento. Isso é em que a diretiva @page foi compilada.

Página 217

Adicionando um Modelo de Rota Catchall

A maioria das estruturas permitem que você defina um modelo de rota que captura todos os URIs que não correspondem

qualquer um dos outros modelos de rota. No momento, o Blazor não suporta isso, mas o
A equipe do Blazor pretende adicionar esse recurso. Apenas seja paciente.

Redirecionando para outras páginas

Como você navega para outro componente usando roteamento? Você tem três opções: use um elemento âncora padrão, use o componente NavLink e use o código. Vamos começar com o marca de âncora normal.

Navegando usando uma âncora

Usar uma âncora (o elemento <a/>) é fácil se você usar um href relativo. Por exemplo, adicionar lista 7-13 abaixo do botão da Listagem 7-12.

Listagem 7-13. Navegação usando uma etiqueta âncora

```
<a class="btn btn-primary" href="/"> Página inicial </a>
```

Este link foi estilizado como um botão usando Bootstrap 4. Execute seu aplicativo e navegue até o componente Contador. Clique no botão Home para navegar para o Índice componente cujo modelo de rota corresponde a "/".

Navegando usando o componente NavLink

O componente NavLink usa uma âncora subjacente, portanto, seu uso é semelhante. A única diferença é que um componente NavLink aplica a classe ativa quando ela corresponde ao rota. Geralmente, você só usa um NavLink no componente NavMenu, mas está livre para use-o em vez de âncoras.

Navegando com Código

Navegar no código também é possível, mas você precisará de uma instância da classe IUriHelper por meio de injeção de dependência. Esta instância permite que você examine o URI da página e tem o método NavigateTo útil. Este método usa uma string que se tornará o novo URI do navegador.

200

Página 218

Vamos tentar um exemplo. Modifique o componente do contador para se parecer com a Listagem 7-14.

Listagem 7-14. Usando o IUriHelper

```
@using Microsoft.AspNetCore.Blazor.Services
@page "/ counter"
@page "/ counter / {CurrentCount: int}"
@layout NestedLayout
@inject IUriHelper uriHelper

<h1> Contador </h1>

<p> Contagem atual: @CurrentCount </p>

<button class = "btn btn-primary" onclick = "@ IncrementCount"> Clique aqui </button>
<a class="btn btn-primary" href="/"> Página inicial </a>
<button class = "btn btn-primary"
        onclick = "@ StartFrom50"> Comece a partir de 50 </button>

@funções {

[Parâmetro]
protegido int CurrentCount {get; definir; }

void IncrementCount ()
{
    CurrentCount ++;
```

```
}
void StartFrom50 ()
{
    uriHelper.NavigateTo ("/ contador / 50");
}
}
```

Para usar o IUriHelper, você precisa adicionar uma diretiva @using para a Microsoft. Namespace AspNetCore.Blazor.Services. Então você diz injeção de dependência com o @inject diretiva para fornecer a você uma instância do IUriHelper e colocá-lo no uriHelper campo. Em seguida, você adiciona um botão que chama o método StartFrom50 quando clicado. Este método usa o uriHelper para navegar para outro URI chamando o método NavigateTo. Execute o seu aplicativo e clique no botão “Iniciar de 50”. Você deve navegar para / counter / 50.

Página 219

Compreendendo a Base Tag

Não use URIs absolutos ao navegar. Por quê? Porque quando você implanta seu aplicativo na Internet, o URI básico será alterado. Em vez disso, o Blazor usa o <base /> elemento e todos os URIs relativos serão combinados com esta tag <base />. Onde está tag <base />? Abra a pasta wwwroot do seu projeto Blazor e abra index.html, mostrado na Listagem 7-15.

Listagem 7-15. Index.html

```
<! DOCTYPE html>
<html>
<head>
    <meta charset = "utf-8" />
    <meta name = "viewport" content = "width = device-width">
    <title> MyFirstBlazor </title>
    <base href = "/" />
    <link href = "css / bootstrap / bootstrap.min.css"
        rel = "stylesheet" />
    <link href = "css / site.css" rel = "stylesheet" />
</head>
<body>
    <app> Carregando ... </app>

    <script src = "_ framework / blazor.webassembly.js"> </script>
</body>
</html>
```

Quando você implanta na produção, tudo que você precisa fazer é atualizar a tag base. Por exemplo, você pode implantar seu aplicativo para https://online.u2u.be/auto-avaliação. Neste caso, você atualizaria o elemento base para <base href = "/ autoavaliação "/>. Então, por que você precisa fazer isso? Se você implantar em https: // online.u2u.be/selfassement, o URI do componente do contador se torna https: // online.u2u.be/selfassessment/counter. O roteamento irá ignorar o URI básico, então coincidir com o contador conforme o esperado. Você só precisa especificar o URI de base uma vez, conforme mostrado na lista 7 a 15.

Página 220

Compartilhando estado entre componentes

Quando você navegar entre os diferentes componentes do Blazor com roteamento, você provavelmente encontrar a necessidade de enviar informações de um componente para outro. Uma maneira de fazer isso é definindo um parâmetro no componente de destino, passando-o no URI. Por exemplo, você pode navegar para / pizzadetail / 5 para informar o destino componente para exibir informações sobre a pizza com id 5. O componente de destino pode então usar um serviço para carregar as informações sobre a pizza nº 5 e exibir este em formação. Mas no Blazor existe outra maneira. Você pode construir uma classe State (a maioria os desenvolvedores chamam isso de estado, mas isso é apenas uma convenção e você pode chamá-lo de qualquer coisa você quer; O estado faz sentido) e, em seguida, use injeção de dependência para dar a cada componente a mesma instância desta classe. Isso também é conhecido como Padrão Singleton. Seu aplicativo Pizza Place já está usando uma classe State, então não deve ser muito trabalhar para usar este padrão.

Comece abrindo a solução Pizza Place dos capítulos anteriores. Abra o índice componente da pasta Pages (no projeto PizzaPlace.Client) e procure o campo de estado privado. Remova este campo (eu fiz um comentário) e substitua-o por um Diretiva @inject como na Listagem 7-16.

Listagem 7-16. Usando injeção de dependência para obter a instância singleton de estado

```
@página "/"
@inject IMenuService menuService
@inject IOrderService orderService
@inject State State

<! - Menu ->
<PizzaList Title = "Nossa lista selecionada de pizzas"
           Menu = "@ State.Menu"
           Selecionado = "@ ((pizza) => AddToBasket (pizza))" />
<! - Menu final ->
<! - Carrinho de compras ->
<ShoppingBasket Title = "Seu pedido atual"
                Basket = "@ State.Basket"
                GetPizzaFromId = "@ State.Menu.GetPizza"
                Selecionado = "@ (pos => RemoveFromBasket (pos))" />
<! - Fim da cesta de compras ->
```


Página 221


```
<! - Entrada do cliente ->
<CustomerEntry Title = "Por favor, insira seus dados abaixo"
               bind-Customer = "@ State.Basket.Customer"
               Enviar = "@ ((_) => PlaceOrder ())" />
<! - Finalizar entrada do cliente ->

<p> @ State.ToJson () </p>

@funções {

// estado privado Estado {get; } = novo estado ();

substituição protegida assíncrona Tarefa OnInitAsync ()
{
    State.Menu = espera menuService.GetMenu ();
}

private void AddToBasket (Pizza pizza)
{
    Console.WriteLine ($ "Adicionou pizza {pizza.Name}");
    State.Basket.Add (pizza.Id);
```

```
    StateHasChanged ();
}

private void RemoveFromBasket (int pos)
{
    Console.WriteLine ($ "Removendo pizza na pos {pos}");
    State.Basket.RemoveAt (pos);
    StateHasChanged ();
}

Private assíncrono Task PlaceOrder ()
{
    esperar orderService.PlaceOrder (State.Basket);
}

}
```

Agora configure a injeção de dependência em Startup.cs para injetar a instância de State como um singleton, como na Listagem 7-17.

Página 222

Listagem 7-17. Configurando injeção de dependência para o singleton de estado

```
usando Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Client.Services;
using PizzaPlace.Shared;

namespace PizzaPlace.Client
{
    public class Startup
    {
        public void ConfigureServices (serviços IServiceCollection)
        {
            services.AddTransient <IMenuService, MenuService> ();
            services.AddTransient <IOrderService, OrderService> ();
            services.AddSingleton <State> ();
        }

        public void Configure (aplicativo IBlazorApplicationBuilder)
        {
            app.AddComponent <App> ("app");
        }
    }
}
```

Execute o aplicativo. Tudo ainda deve funcionar! O que você fez é usar o *Singleton Pattern* para injetar o singleton State no componente Index. Vamos adicionar outro componente que usará a mesma instância de estado.

Você deseja exibir mais informações sobre uma pizza usando um novo componente, mas antes de fazer isso, você precisa atualizar a classe State. Adicione uma nova propriedade chamada CurrentPizza para a classe State, conforme mostrado na Listagem 7-18.

Listagem 7-18. Adicionando uma propriedade CurrentPizza à classe de estado

```
using System;
using System.Collections.Generic;
using System.Text;
usando System.Linq;
```

Página 223


```
namespace PizzaPlace.Shared
{
    classe pública estado
    {
        Menu público Menu {get; definir; } = novo Menu ();

        public Basket Basket {get; definir; } = novo cesto ();

        IU da IU pública {get; definir; } = nova IU ();

        TotalPrice decimal público
            => Basket.Orders.Sum (id => Menu.GetPizza (id) .Price);

        public Pizza CurrentPizza {get; definir; }
    }
}
```

Agora, quando alguém clica em uma pizza do menu, ele exibe a em formação. Atualize o componente PizzaItem envolvendo o nome da pizza em um âncora, como na listagem 7-19 .

Listagem 7-19. Adicionando uma âncora para exibir as informações da pizza

```
@using Microsoft.AspNetCore.Blazor.Services
@using PizzaPlace.Shared

<div class = "row">
    <div class = "col">
        <a href = ""
            onclick = "@ (() => ShowPizzaInformation (Pizza))">
            @ Pizza.Name </a>
    </div>
    <div class = "col">
        @ Pizza.Price
    </div>
    <div class = "col">
        <img src = "@ SpicinessImage (Pizza.Spiciness)"
                alt = "@ Pizza.Spiciness" />
    </div>
```


Página 224


```
        <div class = "col">
            <button class = "@ ButtonClass"
                    onclick = "@ (() => Selecionado (Pizza))">
                    @ButtonTitle </button>
        </div>
    </div>

    @funções {

        [Parâmetro]
        Pizza Pizza protegida {get; definir; }

        [Parâmetro]
        string protegida ButtonTitle {get; definir; }

        [Parâmetro]
```

```
        string protegida ButtonClass {get; definir; }

        [Parâmetro]
        ação protegida <Pizza> selecionada {obter; definir; }

        [Parâmetro]
        protegida Ação <Pizza> ShowPizzaInformation {get; definir; }

        string privada SpicinessImage (Spiciness spiciness)
        => $ "images / {spiciness.ToString (). ToLower ()}. png";
    }
```

Quando alguém clica neste link, ele define a CurrentPizza da instância de estado propriedade. Mas você não tem acesso ao objeto State. Uma maneira de resolver isso seria ser injetando a instância State no componente PizzaItem. Mas você não quer sobrecarregar este componente, então você adiciona um delegado de retorno de chamada ShowPizzaInformation para diga ao componente que contém PizzaList que você deseja exibir mais informações sobre a pizza. Clicar no link do nome da pizza simplesmente invoca este retorno de chamada sem sabendo o que deve acontecer.

Página 225

Você está aplicando um padrão aqui conhecido como "Componentes burros e inteligentes". um idiota componente é um componente que nada sabe sobre a imagem global do aplicativo. Porque ele não sabe nada sobre o resto do aplicativo a componente burro é mais fácil de reutilizar. um componente inteligente conhece o outro partes do aplicativo e usará componentes burros para exibir suas informações. em seu exemplo, PizzaList e PizzaItem são componentes burros, enquanto o O componente de índice é um componente inteligente.

Atualize o componente PizzaList para definir o componente PizzaItem Parâmetro ShowPizzaInformation como na listagem 7-20 .

Listagem 7-20. Adicionando um retorno de chamada PizzaInformation ao componente PizzaList

```
@using PizzaPlace.Shared

<h1> @Title </h1>

@if (Menu == null || Menu.Pizzas == null
        || Menu.Pizzas.Count == 0)
{
    <div style = "altura: 20vh;" classe = "pt-3">
        <div class = "mx-left pt-3" style = "width: 200px">
            <div class = "progress">
                <div class = "progress-bar bg-perigo progress-bar-stripes
                progress- bar- animated w-100 "role =" progressbar "aria-valuenow =" 100 "
                aria- valuemin = "0" aria-valuemax = "100"> </div>
            </div>
        </div>
    </div>

}
senão
{
    @foreach (var pizza em Menu.Pizzas)
    {
```

<PizzaItem Pizza = "@ pizza" ButtonTitle = "Pedido"

208

Página 226

ButtonClass = "btn btn-sucesso"
Selecionado = "@ ((p) => Selecionado (p))"
ShowPizzaInformation = "@ ShowPizzaInformation" />
}
}

@funções {

[Parâmetro]
string protegida Título {get; definir; }

[Parâmetro]
protegido Menu Menu {get; definir; }

[Parâmetro]
ação protegida <Pizza> selecionada {obter; definir; }

[Parâmetro]
protegida Ação <Pizza> ShowPizzaInformation {get; definir; }

}

Você adicionou um retorno de chamada ShowPizzaInformation ao componente PizzaList e simplesmente passe-o para o componente PizzaItem. O componente Índice definirá este retorno de chamada e o PizzaList o passará para o componente PizzaItem.

Atualize o componente Index para definir a CurrentPizza da instância State e navegue ao componente PizzaInfo, conforme mostrado na listagem 7-21 .

Listagem 7-21. O componente de índice navega até o componente PizzaInfo

@página "/"
@using Microsoft.AspNetCore.Blazor.Services
@inject IMenuService menuService
@inject IOrderService orderService
@inject State State
@inject IUriHelper UriHelper

<! - Menu ->
<PizzaList Title = "Nossa lista selecionada de pizzas"
Menu = "@ State.Menu"

209

Página 227

Selecionado = "@ ((pizza) => AddToBasket (pizza))"
ShowPizzaInformation = "@ ((pizza) => ShowPizzaInformation (pizza))" />
<! - Menu final ->
<! - Carrinho de compras ->
<ShoppingBasket Title = "Seu pedido atual"
Basket = "@ State.Basket"
GetPizzaFromId = "@ State.Menu.GetPizza"
Selecionado = "@ (pos => RemoveFromBasket (pos))" />
<! - Fim da cesta de compras ->
<! - Entrada do cliente ->

```
<CustomerEntry Title = "Por favor, insira seus dados abaixo"
                        bind-Customer = "@ State.Basket.Customer"
                        Enviar = "@ (async (_) => esperar PlaceOrder ())" />
<! - Finalizar entrada do cliente ->

<p> @ State.ToJson () </p>

@funções {

// estado privado Estado {get; } = novo estado ();

substituição protegida assíncrona Tarefa OnInitAsync ()
{
    State.Menu = espera menuService.GetMenu ();
}

private void AddToBasket (Pizza pizza)
{
    Console.WriteLine ($ "Adicionou pizza {pizza.Name}");
    State.Basket.Add (pizza.Id);
    StateHasChanged ();
}

private void RemoveFromBasket (int pos)
{
    Console.WriteLine ($ "Removendo pizza na pos {pos}");
    State.Basket.RemoveAt (pos);
    StateHasChanged ();
}
```

210

Página 228


```
Private assíncrono Task PlaceOrder ()
{
    esperar orderService.PlaceOrder (State.Basket);
}

private void ShowPizzaInformation (Pizza pizza)
{
    State.CurrentPizza = pizza;
    UriHelper.NavigateTo ("/ PizzaInfo");
}

}
```

O componente Index diz ao componente PizzaList para chamar ShowPizzaInformation método quando alguém clica no link de informações do componente PizzaItem. O método ShowPizzaInformation então define a propriedade CurrentPizza do estado e navega usando o método UriHelper.NavigateTo para a rota / PizzaInfo.

Clique com o botão direito na pasta Pages e adicione um novo Razor View chamado PizzaInfo, como mostrado na Listagem 7-22 (para economizar algum tempo e manter as coisas simples, você pode copiar a maior parte o componente PizzaItem). O componente PizzaInfo mostra informações sobre o CurrentPizza do estado. Isso funciona porque você compartilha a mesma instância de estado entre esses componentes.

Listagem 7-22. Adicionando um componente PizzaInfo

```
@using PizzaPlace.Shared
@page "/ PizzaInfo"
@inject State State

<h2> Detalhes da pizza </h2>

<div class = "row">
    <div class = "col">
        @ State.CurrentPizza.Name
```

```
        </div>
    </div>
    <div class = "row">
        <div class = "col">
            @ State.CurrentPizza.Price
```

211

Página 229


```
    </div>
</div>
<div class = "row">
    <div class = "col">
        <img src = "@ SpicinessImage (State.CurrentPizza.Spiciness)"
                alt = "@ State.CurrentPizza.Spiciness" />
    </div>
</div>
<div class = "row">
    <div class = "col">
        <a class="btn btn-primary" href="/"> Menu </a>
    </div>
</div>

@funções {

string privada SpicinessImage (Spiciness spiciness)
=> $ "images / {spiciness.ToString (). ToLower ()}. png";
}
```

Na parte inferior da marcação você adiciona uma âncora (e faz com que pareça um botão usando o estilo do Bootstrap) para retornar ao menu. É um exemplo de mudança de rota com âncoras. Claro, em um aplicativo da vida real, você mostraria os ingredientes da pizza, uma bela foto e outras informações. Deixo isso como um exercício para você.

Resumo

Neste capítulo, você examinou duas coisas, layouts e roteamento.

Layouts permitem que você evite replicar marcação em seu aplicativo e ajudam a manter a aparência de seu aplicativo é consistente. Você também viu que os layouts podem ser aninhados.

O roteamento é uma parte importante da construção de aplicativos de página única e cuida de escolher o componente a ser mostrado com base no URI do navegador. Você define modelos de rota usando a sintaxe @page onde você usa parâmetros de rota e restrições. Navegação em seu aplicativo de página única pode ser feito usando tags âncora e a partir do código usando o Classe IUriHelper. Em seguida, você modificou o aplicativo Pizza Place para mostrar como compartilhar informações entre diferentes rotas em um aplicativo Blazor.

212

Página 230

CAPÍTULO 8

Interoperabilidade de JavaScript

Às vezes, simplesmente não há como escapar do uso de JavaScript ☺. Por exemplo, o próprio Blazor usa JavaScript para atualizar o DOM do navegador a partir dos componentes do Blazor. Você pode, também. Neste capítulo, você verá a interoperabilidade com JavaScript e, como exemplo, você construirá uma biblioteca de componentes do Blazor para exibir um gráfico de linha usando um popular biblioteca JavaScript de código aberto para gráficos. Este capítulo exige que você tenha alguns conhecimento básico de JavaScript.

Chamando JavaScript de C

Os navegadores têm muitos recursos que você pode usar em seu site do Blazor. Para Por exemplo, você pode querer usar o *armazenamento local* do navegador para controlar alguns dados. Graças à interoperabilidade de JavaScript do Blazor, isso é fácil.

Fornecendo uma função de cola

Para chamar a funcionalidade JavaScript, você começa construindo uma *função cola* em JavaScript. Eu gosto de chamo essas funções de funções de colagem (minha própria convenção de nomenclatura) porque elas se tornam a cola entre .NET e JavaScript.

As funções de colagem são funções regulares do JavaScript. Uma função de colagem de JavaScript pode levar qualquer número de argumentos, com a condição de que sejam serializáveis em JSON (o que significa que você só pode usar tipos que são conversíveis em JSON, incluindo classes cujo propriedades são serializáveis em JSON). Isso é necessário porque os argumentos e retornam tipo são enviados como JSON entre os tempos de execução .NET e JavaScript.

Em seguida, você adiciona esta função ao objeto de *escopo global* , que no navegador é o objeto de janela. Você verá um exemplo um pouco mais tarde, então continue lendo. Então você pode chame esta função de colagem de JavaScript do seu componente Blazor, como você verá no próxima seção.

P. Himschoot, *Blazor Revelado* ,https://doi.org/10.1007/978-1-4842-4343-5_8

Página 231

Usando JSRuntime para chamar a função Glue

De volta à terra do .NET. Para invocar a função de colagem do JavaScript a partir do C #, você usa o .NET Instância IJSRuntime fornecida por meio da propriedade estática JSRuntime.Current. Esta instância tem o método genérico InvokeAsync <T>, que leva o nome da cola função e seus argumentos e retorna um valor do tipo T, que é o tipo de retorno .NET de a função de cola. Se isso parece confuso, você verá um exemplo imediatamente ...

O método InvokeAsync é assíncrono para oferecer suporte a todos os cenários assíncronos, e esta é a maneira recomendada de chamar JavaScript. Se você precisar ligar para o função cola de forma síncrona, você pode reduzir a instância IJSRuntime para IJSInProcessRuntime e chame seu método Invoke <T> síncrono. Este método leva os mesmos argumentos que InvokeAsync <T> com as mesmas restrições.

Armazenamento de dados no navegador com interoperabilidade

É hora de olhar um exemplo e você começará com a função de colagem do JavaScript. Abrir a solução MyFirstBlazor que você usou nos capítulos anteriores. Abra a pasta wwwroot do projeto MyFirstBlazor.Client e adicione uma nova subpasta chamada scripts. Adicionar um novo arquivo JavaScript para a pasta de scripts chamado interop.js e adicione as funções de colagem da lista 8-1. Essas funções de cola permitem que você acesse o objeto localStorage a partir de o navegador, que permite armazenar dados no computador do cliente para que você possa acessá-lo mais tarde, mesmo depois que o usuário reiniciou o navegador.

Listagem 8-1. As funções getProperty e setProperty Glue

```
window.interop = {
    setProperty: function (name, value) {
```

```
        window.localStorage [nome] = valor;
        valor de retorno;
    },
    getProperty: function (name) {
        return window.localStorage [nome];
    }
};
```

214

Página 232

Seu site do Blazor precisa incluir este script, então abra o arquivo index.html do pasta wwwroot e adicione uma referência de script após o script Blazor, conforme mostrado na Listagem 8-2.

Listagem 8-2. Incluindo a Referência do Script em sua Página HTML

```
<! DOCTYPE html>
<html>
<head>
    <meta charset = "utf-8" />
    <meta name = "viewport" content = "width = device-width">
    <title> MyFirstBlazor5 </title>
    <base href = "/" />
    <link href = "css / bootstrap / bootstrap.min.css"
          rel = "stylesheet" />
    <link href = "css / site.css" rel = "stylesheet" />
</head>
<body>
    <app> Carregando ... </app>

    <script src = "_ framework / blazor.webassembly.js"> </script>
    <script src = "scripts / interop.js"> </script>
</body>
</html>
```

Quando você usa uma biblioteca de componentes blazor, você não precisa incluir o script no a página index.html. você verá um exemplo disso posteriormente neste capítulo.

Agora vamos ver como chamar essas funções glue setProperty / getProperty. Abra o componente Index.cshtml Blazor e modifique-o para se parecer com a Listagem 8-3. Para manter as coisas simples, você chamará as funções de cola de forma síncrona, o que requer o Instância IJSInProcessRuntime, que você armazenará no IPR (em tempo de execução do processo) variável.

215

Página 233

Listagem 8-3. Invocando as funções do Glue de um componente do Blazor

```
@página "/"

<h1> Olá, mundo! </h1>

Bem-vindo ao seu novo aplicativo.

<input type = "number" bind = "@ Counter" />

<SurveyPrompt Title = "Como o Blazor está funcionando para você?" />

@funções {

    IPR IJSInProcessRuntime privado
    => (IJSInProcessRuntime) JSRuntime.Current;

    public int Counter
    {
        obter
        {
            valor da string =
                IPR.Invoke <string> ("interop.getProperty",
                                     nameof (contador));
            if (value! = null && int.ExperimenteParse (value, out var v))
            {
                return v;
            }
            return 0;
        }
        definir
        {
            IPR.Invoke <string> ("interop.setProperty",
                                 nameof (contador), $ "{valor}");
        }
    }
}
```


Página 234

Isso se parece um pouco com o componente Contador, mas agora o Contador armazena seu valor na janela do navegador.localstorage. Para fazer isso, você usa uma propriedade Counter, que invoca suas funções glue no setter e getter de propriedade. Essas funções de cola são síncrona, então você primeiro cria uma propriedade IPR privada para armazenar o IJSInProcessRuntime instância (porque você não quer se repetir, e copiar e colar foi a causa de tantos bugs sutis). Em seguida, no getter da propriedade Counter, você chama a janela. função de colagem interop.getProperty. Mas, como o armazenamento local usa strings, você precisa converter entre o contador do tipo int e string. Há mais uma advertência: inicialmente localstorage não terá um valor ainda, então, se isso retornar uma referência nula, você simplesmente retorna 0. Semelhante ao setter da propriedade Counter, você invoca o window.interop. função de colagem setProperty, certificando-se de converter o int em uma string. Esse último a conversão é absolutamente essencial; caso contrário, você verá erros no console do navegador.

Execute a solução e modifique o valor do Contador. Agora, quando você atualizar seu navegador, você verá o último valor do contador. O contador agora é persistido entre sessões! Você pode sair do navegador, abri-lo novamente e verá o Contador novamente com o último valor.

Passando uma referência para JavaScript

Às vezes, seu JavaScript precisa acessar um de seus elementos HTML. Você pode fazer isso armazenando o elemento em um ElementRef e, em seguida, passando este ElementRef para o

função de cola.

> nunca use a interoperabilidade do JavaScript para modificar o DoM porque isso interferirá no processo de renderização do blazor! se você precisar modificar o DoM do navegador, use um blazor componente.

Você deve usar este ElementRef como um identificador opaco, o que significa que você só pode passar para uma função de colagem de JavaScript, que irá recebê-lo como uma referência de JavaScript para o elemento.

Vejamos um exemplo definindo o foco no elemento de entrada Contador usando interoperabilidade. Comece adicionando uma propriedade do tipo ElementRef à área @functions no Índice. html como na listagem 8-4 .

Página 235

Listagem 8-4. Adicionando uma propriedade ElementRef

```
Private ElementRef inputElement {get; definir; }
```

Em seguida, modifique o elemento de entrada para definir a propriedade inputElement como na Listagem 8-5 .

Listagem 8-5. Configurando o inputElement

```
<input ref = "@ inputElement" type = "number" bind = "@ Counter" />
```

Agora adicione outra função glue a interop.js como na Listagem 8-6.

Listagem 8-6. Adicionando a função de cola setFocus

```
window.interop = {
    setProperty: function (name, value) {
        window.localStorage [nome] = valor;
        valor de retorno;
    },
    getProperty: function (name) {
        return window.localStorage [nome];
    },
    setFocus: function (element) {
        element.focus ();
    }
};
```

Agora vem a parte complicada. O Blazor criará seu componente e, em seguida, chamará o métodos de ciclo de vida, como OnInit. Se você invocar a função setFocus glue em OnInit o DOM não foi atualizado com o elemento de entrada, então isso resultará em um tempo de execução erro porque a função glue receberá uma referência nula. Você precisa esperar pelo DOM a ser atualizado, o que significa que você só deve passar o ElementRef para sua cola função no método OnAfterRender / OnAfterRenderAsync!

Adicione o método OnAfterRender à seção @functions como na Listagem 8-7 .

Listagem 8-7. Passando o ElementRef em OnAfterRender

```
override protegido void OnAfterRender ()
{
    IPR.Invoke <string> ("interop.setFocus", inputElement);
}
```


Página 236

Execute sua solução e você verá que o elemento de entrada recebe o foco automaticamente, como na Figura 8-1 .

Figura 8-1. *O contador recebe o foco automaticamente*

Chamando métodos .NET a partir de JavaScript

Você também pode chamar métodos .NET de JavaScript. Por exemplo, seu JavaScript pode quer dizer ao seu componente que algo interessante aconteceu, como o usuário clicando em algo no navegador. Ou seu JavaScript pode querer perguntar ao Blazor componente sobre alguns dados de que necessita. Você pode chamar um método .NET, mas com alguns das condições. Primeiro, os argumentos e o valor de retorno do seu método .NET precisam ser JSON serializável, o método deve ser público e você precisa adicionar o atributo JSInvokable para o método. O método pode ser estático ou um método de instância.

Para invocar um método estático, use o JavaScript DotNet.invokeMethodAsync ou função DotNet.invokeMethod, passando o nome do assembly, o nome do o método e seus argumentos. Para chamar um método de instância, você passa a instância embrulhado como um DotNetObjectRef para uma função de colagem JavaScript, que pode então invocar o método .NET usando o invokeMethodAsync ou invokeMethod do DotNetObjectRef , passando o nome do método .NET e seus argumentos. Vamos continuar com o exemplo anterior. Quando você faz uma alteração no armazenamento local, o armazenamento aciona um Evento de armazenamento JavaScript, passando o valor antigo e o novo (e mais). Isso permite que você registre-se para alterações em outras guias ou janelas do navegador e use-o para atualizar a página.

Página 237

Adicionando uma função Glue obtendo uma instância .NET

Abra interop.js do exemplo anterior e adicione uma função de observação, como na Listagem 8-8 .

Listagem 8-8. A função watch permite que você se registre para alterações de armazenamento local

```
window.interop = {
    setProperty: function (name, value) {
        window.localStorage [nome] = valor;
        valor de retorno;
    },
    getProperty: function (name) {
        return window.localStorage [nome];
    },
    setFocus: function (element) {
        element.focus ();
    },
    assistir: função (instância) {
```

```
        window.addEventListener ('armazenamento', função (e) {
            console.log ('evento de armazenamento');
            instance.invokeMethodAsync ('UpdateCounter');
        });
    }
};
```

A função de observação leva uma referência a uma instância DotNetObjectRef e invoca o Método UpdateCounter quando o armazenamento muda.

Adicionando um método JSInvokable para invocar

Abra Index.cshtml e adicione o método UpdateCounter à área @functions, como mostrado na Listagem 8-9 .

Página 238

Listagem 8-9. O Método UpdateCounter

```
[JSInvokable]
public Task UpdateCounter ()
{
    this.StateHasChanged ();
    return Task.CompletedTask;
}
```

Este método aciona a IU para atualizar com o valor mais recente de Counter. Observe que esse método segue o padrão assíncrono .NET, retornando uma instância de Task. Porque você não estão realmente chamando nenhuma API assíncrona, você retorna o Task.CompetedTask. Completar No exemplo, adicione o método de ciclo de vida OnInit mostrado na Listagem 8-10 .

Listagem 8-10. O Método OnInit

```
sobrescrito protegido void OnInit ()
{
    IPR.Invoke <string> ("interop.watch",
                                novo DotNetObjectRef (this));
}
```

O método OnInit envolve esta referência do componente Index em um DotNetObjectRef e o passa para a função interop.watch.

Para ver isso em ação, abra duas guias do navegador em seu site. Quando você muda o valor em uma guia você deve ver a outra atualização da guia para o mesmo valor, conforme mostrado em Figura 8-2.

Página 239

Figura 8-2. *Atualizar o contador em uma guia atualiza a outra guia*

Construindo uma biblioteca de componentes gráficos Blazor

Nesta seção, você construirá uma *biblioteca de componentes do Blazor* para exibir gráficos usando um popular biblioteca JavaScript de código aberto chamada Chart.js (www.chartjs.org). Contudo, envolver toda a biblioteca tornaria este capítulo muito longo, então você apenas usará um componente de gráfico de linha simples.

Página 240

Criando a biblioteca de componentes do Blazor

Abra o Visual Studio e comece criando um novo projeto Blazor chamado ChartTestProject, como mostrado na figura 8-3. Este projeto será usado apenas para testar o componente gráfico.

Figura 8-3. *Criação de um novo projeto Blazor*

Se você estiver usando o código, abra um prompt de comando e digite

dotnet new blazorhosted -o ChartTestProject

Com o Visual Studio e o Code, abra um prompt de comando no diretório contendo a solução ChartTestProject (a pasta onde está o arquivo .sln) e digite

dotnet new blazorlib -o U2U.Components.Chart

Isso criará uma nova *biblioteca de componentes do Blazor* . Infelizmente, você não pode criar esse tipo de projeto com Visual Studio (ainda). Volte para o Visual Studio, clique com o botão direito do mouse no solução e selecione Add ➤ Existing Project. Selecione o projeto U2U.Components.Chart.
Sua solução deve ser semelhante à Figura 8-4.

Página 241

Figura 8-4. *A solução que contém a biblioteca de componentes*

Se você estiver usando o código, basta digitar este comando para adicionar a biblioteca de componentes ao solução:

dotnet sln add U2U.Components.Chart / U2U.Components.Chart.csproj

Adicionando a Biblioteca de Componentes ao Seu Projeto

Agora você tem o projeto de biblioteca de componentes do Blazor. Vamos usá-lo no projeto de teste. Procure Component1.cshtml no projeto U2U.Components.Chart e renomeie-o para LineChart.cshtml. Adicione uma referência à biblioteca de componentes no projeto do cliente. Dentro Visual Studio, clique com o botão direito em ChartTestProject e selecione Add Add Reference. Verifica a Projeto U2U.Components.Chart, mostrado na Figura 8-5e clique em OK.

Página 242

Figura 8-5. *Adicionando uma referência à biblioteca de componentes*

Com o Código, use o terminal integrado, altere o diretório atual para ChartTestProject.Client e digite este comando:

```
dotnet add reference ../U2U.Components.Chart/U2U.Components.Chart.csproj
```

Isso adicionará uma referência à biblioteca de componentes U2U.Components.Chart.

Procure o arquivo _ViewImports.cshtml (aquele ao lado de App.cshtml) no ChartTestProject e abra-o no editor. Lembre-se do Capítulo 3 que usar um componente de uma biblioteca, você precisa adicioná-lo como um auxiliar de tag MVC. Insira o Diretiva @addTagHelper mostrada na Listagem 8-11.

Listagem 8-11. Adicionando o LineChart tagHelper ao Projeto Blazor

```
@using System.Net.Http
@using Microsoft.AspNetCore.Blazor.Layouts
@using Microsoft.AspNetCore.Blazor.Routing
@using Microsoft.JSInterop
@using ChartTestProject
@using ChartTestProject.Shared

@addTagHelper *, U2U.Components.Chart
```


Página 243

Abra o arquivo Index.cshtml da pasta Pages e adicione o LineChart componente mostrado na Listagem 8-12.

Listagem 8-12. Adicionando o Componente LineChart

```
@página "/"

<h1> Olá, mundo! </h1>

Bem-vindo ao seu novo aplicativo.
```

```
<LineChart />

<SurveyPrompt Title = "Como o Blazor está funcionando para você?" />
```

Crie e execute seu aplicativo. Deve ser semelhante à Figura 8-6 .

Figura 8-6. *Testando se a biblioteca de componentes foi adicionada corretamente*

Adicionando Chart.js à Biblioteca de Componentes

O componente LineChart não se parece com um gráfico, então é hora de consertar isso! Primeiro você precisa adicionar a biblioteca JavaScript Chart.js ao projeto de biblioteca de componentes. Vamos para www.chartjs.org/ . Esta é a página principal do Chart.js. Agora clique no botão GitHub, mostrado na Figura 8-7 , para abrir a página GitHub do projeto.

Figura 8-7. *A página principal do Chart.js*

Role para baixo nesta página procurando o link de lançamentos do GitHub. Pressione este link com o seu mouse e a página de lançamento será aberta, conforme mostrado na Figura 8-8 .

> Uma vez que leva algum tempo entre escrever um livro e lê-lo, há uma grande chance de que a versão tenha aumentado. Certifique-se de selecionar uma versão começando com 2, já que a versão 3 conterá alterações importantes.

Clique em Chart.bundle.min.js para baixá-lo, conforme mostrado na Figura 8-8.

Página 245

Figura 8-8. *Página de lançamentos do GitHub para Chart.js*

Após o download, copie este arquivo para a pasta de conteúdo do Projeto U2U.Components.Chart, conforme mostrado na Figura 8-9 . Todos os arquivos nesta pasta são baixado automaticamente no navegador pelo Blazor para que você não precise adicioná-los ao index.html.

Figura 8-9. *Copiando Chart.bundle.min.js para a pasta de conteúdo*

Página 246

Verificando se a biblioteca JavaScript foi carregada corretamente

Você sabe sobre a Lei de Murphy? Ele afirma: "Tudo o que pode dar errado, dá."
Vamos nos certificar de que a biblioteca Chart.js seja carregada pelo navegador. Execute o seu Blazor

projeto e abra o depurador do navegador. Verifique se Chart.bundle.min.js foi carregado corretamente. A maneira mais fácil de fazer isso é ver se window.Chart foi definido (Chart.js adiciona uma função construtora chamada Gráfico para o objeto global da janela). Você pode fazer isso de a guia Console do depurador digitando window.Chart, conforme mostrado na Figura 8-10.

Figura 8-10. *Usando o console do navegador para verificar o valor de window.Chart*

Se isso retornar indefinido, reconstrua o projeto U2U.Components.Chart. Então você pode tentar atualizar o navegador após esvaziar o cache do navegador. Quando o navegador depurador for mostrado, clique com o botão direito no botão de atualização e você obterá um menu suspenso, como mostrado na Figura 8-11. Selecione o item de menu Empty Cache and Hard Reload.

Figura 8-11. *Recarregando a página após limpar o cache*

Página 247

Adicionando dados Chart.js e classes de opções

Abra seu navegador e digite www.chartjs.org/docs/latest/ . Aqui você pode ver um amostra do uso de Chart.js em JavaScript. Esta biblioteca requer duas estruturas de dados para ser passado para ele: um contendo os dados do gráfico e outro contendo as opções. Esta seção irá adicionar essas classes à biblioteca de componentes do Blazor, mas agora usando C #. Novamente, eu não sou indo para uma cobertura completa de todos os recursos do Chart.js para manter as coisas nítidas.

A classe ChartOptions

Vamos começar com a aula de opções. Clique com o botão direito na biblioteca U2U.Components.Chart e adicione uma nova classe chamada ChartOptions como na Listagem 8-13.

esta é uma boa quantidade de código. você pode considerar copiá-lo das fontes fornecido com este livro. Eu também deixei de fora comentários que descrevem cada propriedade para concisão.

Listagem 8-13. A classe ChartOptions

```
public class ChartOptions
{
    public class TitleOptions
    {
        public static readonly TitleOptions Default
```

```
    = novo TitleOptions ();

    public bool Display {get; definir; } = falso;
}

public class ScalesOptions
{
    padrão público estático somente leitura ScalesOptions
    = new ScalesOptions ();

    public class ScaleOptions
    {
```

230

Página 248


```
        public static readonly ScaleOptions Default
        = novo ScaleOptions ();

        public class TickOptions
        {
            public static readonly TickOptions Default
            = novo TickOptions ();

            public bool BeginAtZero {get; definir; } = verdadeiro;

            public int Max {get; definir; } = 100;
        }

        public bool Display {get; definir; } = verdadeiro;

        public TickOptions Ticks {get; definir; }
        = TickOptions.Default;
    }

    public ScaleOptions [] YAxes {get; definir; }
    = novo ScaleOptions [] {ScaleOptions.Default};
  }

  public static readonly ChartOptions Default
  = novo ChartOptions {};

  public TitleOptions Title {get; definir; }
  = TitleOptions.Default;

  public bool Responsive {get; definir; } = verdadeiro;

  public bool MaintainAspectRatio {get; definir; } = verdadeiro;

  public ScalesOptions Scales {get; definir; }
  = ScalesOptions.Default;
}
```

esta classe C #, com classes aninhadas, reflete o objeto de opções do JavaScript (parcialmente) de Chart.js. observe que adicionei propriedades estáticas padrão a cada classe para torna mais fácil para os desenvolvedores construir a hierarquia de opções.

231

Página 249

A classe LineChartData

Chart.js espera que você forneça os dados que ele renderizará. Para isso, é preciso conhecer alguns de coisas, como a cor da linha, a cor do preenchimento abaixo da linha e, é claro, o números para traçar o gráfico. Então, como você representará cores e pontos em seu Blazor componente? Acontece que existem classes no .NET para representar cores e pontos: System.Drawing.Color e System.Drawing.Point. Infelizmente, você não pode usar Cor porque não se converte em uma cor JavaScript, mas você pode permitir que os usuários usem em seu código. Discutirei como fazer isso um pouco mais tarde. Adicione uma nova classe LineChartData para a biblioteca de componentes chamada LineChartData, conforme mostrado na Listagem 8-14.

Listagem 8-14. A classe LineChartData

```
using System;
using System.Collections.Generic;
using System.Drawing;

namespace U2U.Components.Chart
{
   public class LineChartData
   {
      public class DataSet
      {
         public string Label {get; definir; }

         Public List <Point> Data {get; definir; } = nulo;

         public string BackgroundColor {get; definir; }

         public string BorderColor {get; definir; }

         public int BorderWidth {get; definir; } = 2;
      }

      public string [] Labels {get; definir; }
      = Array.Empty <string> ();

      conjunto de dados público [] Conjuntos de dados {obter; definir; }
   }
}
```


Página 250

A maior parte dessa classe deve ser clara, exceto talvez para Array.Empty <string> (). Esta método retorna uma matriz vazia do argumento genérico. Mas por que isso é melhor? Vocês não pode modificar uma matriz vazia, então você pode usar a mesma instância em qualquer lugar (isto também é conhecido como *Padrão Flyweight*). Isso é como uma corda. Vazio e usá-lo coloca menos tensão no coletor de lixo.

Registrando a função JavaScript Glue

Para invocar a biblioteca Chart.js, você precisa adicionar um pouco de JavaScript próprio. Abrir a pasta de conteúdo do projeto de biblioteca de componentes e comece renomeando o arquivo exampleJsInterop.js para JsInterop.js e substituindo o código por Listagem 8-15 .

Listagem 8-15. Registrando a classe JavaScript Glue

```
window.components = (function () {
   Retorna {
      gráfico: função (id, dados, opções) {
         var context = document.getElementById (id)
                                    .getContext ('2d');
         var chart = new Chart (contexto, {
            tipo: 'linha',
```

```
                dados: dados,
                opções: opções
            });
        }
    };
}) ();
```

Isso adiciona uma função de colagem window.components.chart que, quando invocada, chama o Função de gráfico (de Chart.js), passando o contexto gráfico para a tela, dados, e opções. É muito importante que você passe o id do canvas porque alguém pode querer usar o componente LineChart várias vezes na mesma página. Usando um id único para cada componente LineChart você acaba com telas com ids únicos.

Página 251

Fornecendo o serviço de interoperabilidade JavaScript

Seu componente LineChart precisará chamar a biblioteca Chart.js usando sua janela. função de cola components.chart. Mas colocar toda essa lógica no componente LineChart diretamente é algo que você deseja evitar. Em vez disso, você construirá um serviço encapsulando esta lógica e injeta o serviço no componente LineChart. A equipe Blazor deve na Microsoft decidiu mudar a forma como a interoperabilidade JavaScript funciona (eles fizeram que antes), então você só precisará mudar uma classe (novamente, a *responsabilidade única Princípio*). Comece adicionando uma nova interface ao projeto de biblioteca U2U.Component.Chart chamado IChartInterop com o código da Listagem 8-16.

Listagem 8-16. A interface IChartInterop

```
namespace U2U.Components.Chart
{
    interface pública IChartInterop
    {
        void CreateLineChart (string id, LineChartData data,
                                                Opções ChartOptions);
    }
}
```

Como você pode ver, o método CreateLineChart desta interface é muito parecido com o função de colagem window.components.chart. Vamos implementar este serviço. Adicionar um novo classe chamada ChartInterop para o projeto e implementação da biblioteca de componentes é como em Listagem 8-17.

Listagem 8-17. Implementando a classe ChartInterop

```
usando Microsoft.JSInterop;

namespace U2U.Components.Chart
{

    /// <resumo>
    /// É sempre uma boa ideia ocultar uma implementação específica
    /// detalhes por trás de uma classe de serviço
    /// </summary>
    public class ChartInterop: IChartInterop
```

Página 252


```
    {
        public void CreateLineChart (string id, LineChartData data,
                                                Opções ChartOptions)
        {
            JSRuntime.Current
                        .InvokeAsync <string> ("components.chart",
                                                    id, dados, opções);
        }
    }
}
```

Este método CreateLineChart invoca a função JavaScript components.chart você adicionou na Listagem 8-15.

É hora de configurar a injeção de dependência. Você pode pedir ao usuário da biblioteca para adicione a dependência IChartInterop diretamente, mas você não quer colocar muito responsabilidade nas mãos do usuário. Em vez disso, você fornecerá ao usuário um prático *C #* *método de extensão* que esconde todos os detalhes sangrentos do usuário. Adicione a nova classe chamada DependencyInjection para o projeto de biblioteca de componentes com o código da Listagem 8-18 .

Listagem 8-18. O método de extensão AddCharts

```
using Microsoft.Extensions.DependencyInjection;

namespace U2U.Components.Chart
{
    public static class DependencyInjection
    {
        public static IServiceCollection AddCharts (
            este IServiceCollection services)
        => services.AddSingleton <IChartInterop, ChartInterop> ();
    }
}
```

Esta classe fornece o método de extensão AddCharts do qual o usuário o componente LineChart agora pode ser adicionado ao projeto do cliente. Vamos fazer isso. Certificar-se de que tudo é criado primeiro e, em seguida, abra Startup.cs no ChartTestProject e adicione um ligue para AddCharts como na listagem 8-19 .

Página 253

Listagem 8-19. Injeção de dependência conveniente com AddCharts

```
usando Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;
using U2U.Components.Chart;

namespace ChartTestProject
{
    public class Startup
    {
        public void ConfigureServices (serviços IServiceCollection)
        {
            services.AddCharts ();
        }

        public void Configure (aplicativo IBlazorApplicationBuilder)
        {
```

```
            app.AddComponent <App> ("app");
        }
    }
}
```

O usuário do componente não precisa saber nenhum detalhe de implementação para usar o componente LineChart. Missão cumprida!

Implementando o Componente LineChart

Agora você está pronto para implementar o componente LineChart. Chart.js faz todo o seu desenho usando um elemento de tela HTML5, e esta será a marcação do LineChart componente. Atualize LineChart.cshtml para corresponder à Listagem 8-20.

Listagem 8-20. O Componente LineChart

```
@inject IChartInterop JsInterop

<canvas id = "@ Id" class = "@ Class">
</canvas>
```


Página 254


```
@funções {

    [Parâmetro]
    string protegida Id {get; definir; }

    [Parâmetro]
    string protegida Class {get; definir; }

    [Parâmetro]
    LineChartData Data {get; definir; }

    [Parâmetro]
    Opções de ChartOptions {get; definir; } = ChartOptions.Default;

    override protegido void OnAfterRender ()
    {
        string id = Id;
        JsInterop.CreateLineChart (Id, Dados, Opções);
    }
}
```

O componente LineChart possui alguns parâmetros. O parâmetro Id é usado para dê a cada tela do LineChart um identificador exclusivo; desta forma, você pode usar LineChart vários vezes na mesma página. O parâmetro Class pode ser usado para dar à tela um ou mais Classes CSS para adicionar algum estilo (e você nunca terá estilo suficiente). Finalmente, os dados e os parâmetros de opções são passados para o JavaScript para configurar o gráfico.

Agora vem a parte complicada (é como a seção anterior onde você queria definir o foco na entrada). Para chamar a função de gráfico JavaScript, a tela precisa estar no DOM do navegador. Quando isso acontece? Blazor cria a hierarquia de componentes, chama OnInit, OnInitAsync, OnParameterSet e OnParameterSetAsync de cada componente métodos e, em seguida, usa a hierarquia de componentes para construir sua árvore interna, que então é usado para atualizar o DOM do navegador. Em seguida, o Blazor chama OnAfterRender de cada componente e métodos OnAfterRenderAsync. Porque o elemento canvas já deve fazer parte do DOM, você precisa esperar pelo método OnAfterRender antes de chamar JsInterop. CreateLineChart.

Página 255

Usando o componente LineChart

Com tudo no lugar, agora você pode concluir o componente LineChart do
Página de índice em seu ChartTestProject. Atualize o arquivo Index.cshtml para corresponder à Listagem8-21 .
Você adicionará o método de extensão toJS () mais tarde, então ignore quaisquer erros até então.

Listagem 8-21. Completando o Componente de Índice

```
@página "/"
@using U2U.Components.Chart
@using System.Drawing

<h1> Olá, mundo! </h1>

Bem-vindo ao seu novo aplicativo.

<LineChart Id = "test" Class = "linechart"
                    Data = "@ Data" Options = "@ Options" />

<SurveyPrompt Title = "Como o Blazor está funcionando para você?" />

@funções {

Dados LineChartData privados {get; definir; }

Private ChartOptions Options {get; definir; }

sobrescrito protegido void OnInit ()
{
    this.Options = ChartOptions.Default;

    this.Data = new LineChartData
    {
        Rótulos = nova string [] {"", "A", "B", "C"},
        Conjuntos de dados = novo LineChartData.DataSet []
        {
            new LineChartData.DataSet
            {
                Label = "Teste",
                BackgroundColor = Color.Transparent.ToJs (),
                BorderColor = Color.FromArgb (10, 96, 157, 219)
                                            .ToJs (),
```


Página 256


```
                BorderWidth = 5,
                Dados = nova lista <ponto>
                {
                    novo ponto (0,0),
                    novo ponto (1, 11),
                    novo ponto (2, 76),
                    novo ponto (3,13)
                }
            }
        }
    };
```

```
}
}
```

Você começa adicionando duas diretivas @using para U2U.Components.Chart e System. Desenho de namespaces. Em seguida, você adiciona os parâmetros Id, Class, Data e Options. Você dá esses valores de parâmetros no método OnInit (você deve obter esses dados do servidor você usaria o método OnInitAsync). Mais uma coisa antes de construir e executar o projeto e admire seu trabalho: adicione uma nova classe chamada ColorExtensions ao U2U. Projeto Component.Chart. Implemente-o conforme mostrado na listagem8-22 .

Listagem 8-22. A classe ColorExtensions com o método de extensão toJS

```
using System.Drawing;

namespace U2U.Components.Chart
{
   public static class ColorExtensions
   {
      ToJs de string estáticas públicas (esta cor c)
      => $ "rgba ({cR}, {cG}, {cB}, {cA})";
   }
}
```

Construa e execute seu projeto. Se tudo estiver bem, você deve ver a Figura 8-12.

Página 257

Figura 8-12. *O exemplo de gráfico acabado*

Resumo

Neste capítulo, você viu como pode chamar JavaScript de seus componentes do Blazor usando o método JSRuntime.Current.InvokeAsync <T>. Isso requer que você registre um Função de colagem JavaScript adicionando esta função ao objeto global da janela do navegador.

Você também pode chamar seu .NET estático ou método de instância de JavaScript. Começar por adicionando o atributo JSInvokable ao método .NET. Se o método for estático, você usa a função DotNet.invokeMethodAsync do JavaScript (ou DotNet.invokeMethod se a chamada é síncrono), passando o nome do assembly, o nome do método e seu argumentos. Se o método for um método de instância, você passa a instância .NET empacotada em um DotNetObjectRef para a função glue, que pode então usar o invokeMethodAsync função para chamar o método, passando o nome do método e seus argumentos.

Finalmente, você aplicou esse conhecimento envolvendo a biblioteca de código aberto Chart.js para

desenhe um bom gráfico de linhas. Você construiu *uma biblioteca de componentes do Blazor* , adicionou algumas classes para passar os dados para a função Gráfico e, em seguida, use uma função de colagem para desenhar o gráfico.

Página 258

Índice

UMA

B

P. Himschoot, *Blazor Revealed* , https://doi.org/10.1007/978-1-4842-4343-5

Página 259

C

Página 260

D

E

Página 261

F

G

H

eu

J, K

eu

Página 262

Página 263

Página 264