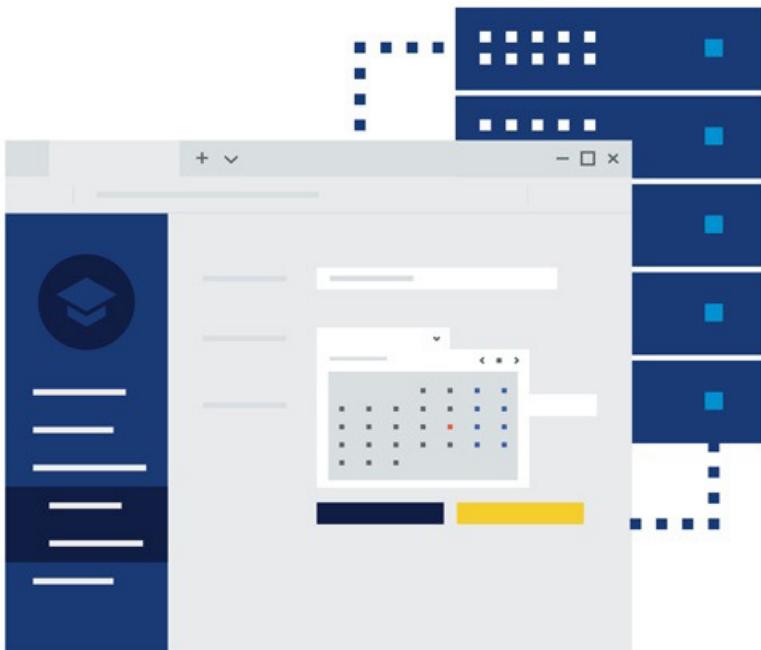


ASP.NET Core MVC

Aplicações modernas em conjunto
com o Entity Framework



Casa do
Código

EVERTON COIMBRA DE ARAÚJO

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-94188-45-8

EPUB: 978-85-94188-46-5

MOBI: 978-85-94188-47-2

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

SOBRE O AUTOR

Everton Coimbra de Araújo atua na área de treinamento e desenvolvimento. É tecnólogo em processamento de dados pelo Centro de Ensino Superior de Foz do Iguaçu, possui mestrado em Ciência da Computação pela UFSC, e doutorado pela UNIOESTE em Engenharia Agrícola.

É professor da Universidade Tecnológica Federal do Paraná (UTFPR), campus Medianeira, onde leciona disciplinas no curso de ciência da computação e em especializações. Já ministrou aulas de algoritmos, técnicas de programação, estrutura de dados, linguagens de programação, Orientação a Objetos, análise de sistemas, UML, Java para web, Java EE, banco de dados e .NET.

Possui experiência na área de Ciência da Computação, com ênfase em Análise e Desenvolvimento de Sistemas, atuando principalmente nos seguintes temas: desenvolvimento web com Java e .NET, e persistência de objetos. O autor é palestrante em seminários de informática voltados ao meio acadêmico e empresarial.



PREFÁCIO

É uma honra e um grande "problema" prefaciar um livro de alguém que admiramos. Everton tem uma larga experiência em criar livros didáticos na área de computação. Seus livros têm sido usados em cursos técnicos e universitários como referência básica em muitas universidades pelo país há bastante tempo.

Neste livro, assim como nos outros, o autor traz as experiências vivenciadas em sala de aula, sanando as principais dúvidas, comuns aos iniciantes, e mostrando o passo a passo de "como se faz". Ao mesmo tempo, atinge não só o público iniciante, mas vai além, auxiliando profissionais da área.

Os capítulos vão ganhando complexidade com o passar do tempo, assim como uma escada: levam o leitor a alcançar novos degraus (conhecimentos), até chegarem a um patamar superior.

Espero que este livro agrade aos leitores, assim como me agradou. Uma boa leitura a todos!

Pedro Luiz de Paula Filho

Universidade Tecnológica Federal do Paraná - Câmpus Medianeira

INTRODUÇÃO

Como sempre digo na abertura de todos os meus livros, ensinar e aprender são tarefas que andam juntas. Para que seja alcançado o objetivo em cada uma delas, é necessário muita dedicação e estudo constante. Não há mágica no aprendizado, mas há muita determinação por parte daquele que quer aprender.

Este livro apresenta o ASP.NET Core MVC, um framework da Microsoft para o desenvolvimento de aplicações web. Durante os capítulos, é feito uso do IDE da Microsoft e o Visual Studio 2017 Community Edition. A linguagem adotada para os exemplos é a C#. Conceitos de técnicas, tanto do framework, da ferramenta e como da linguagem, são introduzidos sempre que utilizados.

O livro traz implementações que poderão auxiliar no desenvolvimento de suas aplicações e apresenta um pouco de JavaScript e jQuery, bem como introduz o Bootstrap. Também faz uso do Entity Framework Core como ferramenta para persistência de dados.

Certamente, este livro pode ser usado como ferramenta em disciplinas que trabalham o desenvolvimento para web, quer seja por acadêmicos ou professores. Isso porque ele é o resultado da experiência que tenho em ministrar aulas dessa disciplina, então, trago para cá anseios e dúvidas dos alunos que estudam comigo.

O objetivo deste trabalho é tornar o livro uma ferramenta para o ensino e o aprendizado no desenvolvimento de aplicações para web. Vamos usar C# de uma forma mais precisa e direcionada às disciplinas ministradas em cursos que envolvem informática e

computação, como Sistemas de Informação, Ciência da Computação, Processamento de Dados, e Análise e Desenvolvimento de Sistemas.

O público-alvo para este livro é vasto. Ele serve para alunos de graduação que tenham disciplinas afins em seus cursos, como também para desenvolvedores que estão começando em ASP.NET, ou até mesmo para os experientes, que buscam por novas informações. Não faz parte do objetivo do livro ensinar a linguagem C# ou Orientação a Objetos.

O livro é desenvolvido em dez capítulos, todos com muita prática e com uma conclusão dos tópicos vistos. Na sequência, são apresentados pequenos resumos do que é trabalhado em cada um deles.

O repositório com todos os códigos-fonte utilizados no livro pode ser encontrado em: <https://github.com/evertonfoz/asp-net-core-mvc-casa-do-codigo>.

Capítulo 1 – A primeira aplicação ASP.NET Core MVC

Este primeiro capítulo traz uma introdução a alguns conceitos relacionados ao ASP.NET Core MVC e ao desenvolvimento de aplicações web. Nele, também já ocorre a implementação da primeira aplicação. Esta realiza as operações básicas em um conjunto de dados, conhecidas como CRUD (*Create, Read, Update e Delete*). Inicialmente, essas operações são realizadas em uma

Collection, pois este capítulo dedica-se à introdução ao framework, e prepara o leitor para o segundo capítulo, que introduz o acesso a uma base de dados.

Capítulo 2 – Acesso a dados com o Entity Framework Core

Com o conhecimento que se espera do leitor após a leitura do capítulo anterior – no qual são apresentadas técnicas específicas do ASP.NET Core MVC para a criação de um CRUD –, neste aplicamos essas técnicas para a manutenção desse CRUD em uma base de dados. Estas implementações são realizadas no SQL Server, por meio do Entity Framework Core, que também é introduzido neste capítulo e usado nos demais.

Capítulo 3 – Layouts, Bootstrap e jQuery DataTable

Em uma aplicação web, o uso de layouts comuns para um número significativo de páginas é normal. Um portal (visto como um sistema, uma aplicação) normalmente é dividido em setores, e cada um pode ter seu padrão de layout. O ASP.NET Core MVC possui recursos para esta característica.

Este capítulo apresenta também o Bootstrap, um componente que possui recursos para facilitar o desenvolvimento de páginas web, por meio de CSS. O capítulo termina com a apresentação de um controle JavaScript, o jQuery DataTable, em que um conjunto de dados é renderizado em uma tabela com recursos para busca e classificação.

Capítulo 4 – Associações no Entity Framework Core

As classes identificadas no modelo de negócio de uma aplicação não são isoladas umas das outras. Muitas vezes, elas associam-se entre si, o que permite a comunicação no sistema. Saber que uma classe se associa a outra quer dizer também que elas podem depender uma da outra.

Em um processo, como uma venda, por exemplo, existe a necessidade de saber quem são o cliente e o vendedor, bem como quais são os produtos adquiridos. Este capítulo trabalha a associação entre duas classes, o que dará subsídios para aplicar esse conhecimento em outras aplicações. Também são trazidos controles que permitem a escolha de objetos para gerar a associação.

Capítulo 5 – Separação da camada de negócios

O ASP.NET Core MVC traz em seu nome o padrão de projeto MVC (*Model-View-Controller*, ou Modelo-Visão-Controlador). Embora os recursos (classes, visões, dentre outros) possam ser armazenados em pastas criadas automaticamente em um novo projeto, a aplicação criada, por si só, não está dividida em camadas. Isso acontece pois não ela está em módulos que propiciem uma independência do modelo. Este capítulo apresenta os conceitos de **coesão** e **acoplamento**, assim como implementa uma estrutura básica que pode ser replicada em seus projetos.

Capítulo 6 – Code First Migrations, Data Annotations e validações

Durante o processo de desenvolvimento de uma aplicação que envolve a persistência em base de dados, qualquer mudança em

uma classe do modelo deve ser refletida nessa base. Quando utilizamos o Entity Framework Core, essas mudanças são identificadas e, dependendo da estratégia de inicialização adotada, pode ocorrer que sua base seja removida e criada novamente, do zero, sem os dados de testes que, porventura, existam.

O Entity Framework Core oferece o Code First Migration, que possibilita o versionamento da estrutura de uma base de dados mapeada por meio de um modelo de classes. Este capítulo apresenta o Code First Migration, que possibilita a atualização dessa base, sem a perda dos dados registrados nela. Também são apresentados os Data Annotations, para definirmos características de propriedades, e algumas regras de validação.

Capítulo 7 – Areas, autenticação e autorização

Quando desenvolvemos uma aplicação com muitas classes, controladores e visões, torna-se difícil administrar a organização pela forma trivial oferecida pelo ASP.NET Core MVC, pois todos os controladores ficam em uma única pasta, assim como os modelos e as visões. Para minimizar este problema, o framework oferece **Areas**.

Elas podem ser vistas como submodelos, onde ficam seus controladores, modelos e visões de maneira mais organizada. O conceito **Modelo para Visões** também é apresentado. O capítulo dedica-se ainda ao processo de autenticação e autorização de usuários – um requisito necessário para o controle de acesso para qualquer aplicação.

Capítulo 8 – Uploads, downloads e erros

Com o surgimento da computação em nuvem, ficaram cada vez mais constantes os portais ou sistemas web possibilitarem uploads e oferecerem downloads. Este capítulo apresenta como é possível: enviar arquivos para uma aplicação, obter arquivos de imagens (para serem renderizadas) e possibilitar o download de arquivos hospedados no servidor. O capítulo termina com uma técnica para tratamento de erros que podem ocorrer na aplicação.

Capítulo 9 – DropDownList com chamadas AJAX e uso de sessões

O uso de controles do tipo `DropDownList` pode ser uma atividade necessária em sua aplicação, assim como o uso de mais de um desses controles. Além disso, um determinado controle pode ter suas opções dependentes de uma seleção em outro. Este capítulo traz o uso desses controles de maneira encadeada, ou seja, um dependendo do dado que está selecionado em outro.

Outro tema abordado neste capítulo é o uso de sessão para armazenamento de dados. Esta prática é comum em aplicações comerciais que precisam persistir dados pequenos para uma rápida recuperação na aplicação.

Capítulo 10 – Os estudos não param por aqui

Com este capítulo, concluímos este livro destacando todos os assuntos que vimos até aqui, junto com estímulos para você continuar a sua jornada de aprendizagem e aplicação do C#.

Sumário

1 A primeira aplicação ASP.NET Core MVC	1
1.1 Criação do projeto no Visual Studio 2017 Community	5
1.2 Criando o controlador para Instituições de Ensino	14
1.3 Criação da classe de domínio para Instituições de Ensino	19
1.4 Implementação da interação da action Index com a visão	20
1.5 O conceito de rotas do ASP.NET Core MVC	30
1.6 Implementação da inserção de dados no controlador	33
1.7 Implementação da alteração de dados no controlador	42
1.8 Implementação da visualização de um único registro	46
1.9 Finalização da aplicação: a operação Delete	49
1.10 Conclusão sobre as atividades realizadas no capítulo	51
2 Acesso a dados com o Entity Framework Core	53
2.1 Começando com o Entity Framework Core	54
2.2 Verificando a base de dados criada no Visual Studio	64
2.3 Implementação do CRUD fazendo uso do Entity Framework Core	65
2.4 Conclusão sobre as atividades realizadas no capítulo	80

3 Layouts, Bootstrap e jQuery DataTable	81
3.1 O Bootstrap	82
3.2 Layouts com o Bootstrap	87
3.3 Primeira página com o layout criado	95
3.4 Adaptando as visões para o uso do Bootstrap	98
3.5 Configuração do menu para acessar as visões criadas	115
3.6 Conclusão sobre as atividades realizadas no capítulo	116
 4 Associações no Entity Framework Core	118
4.1 Associando as classes já criadas	119
4.2 Adaptação para uso das associações	124
4.3 A visão Create para a classe Departamento	128
4.4 A visão Edit para a classe Departamento	132
4.5 A visão Details para a classe Departamento	134
4.6 Criando a visão Delete para a classe Departamento	136
4.7 Inserção de Departamentos na visão Details de Instituições	141
4.8 Conclusão sobre as atividades realizadas no capítulo	145
 5 Separação da camada de negócio	147
5.1 Contextualização sobre as camadas	148
5.2 Criando a camada de negócio: o modelo	149
5.3 Criando a camada de persistência em uma pasta da aplicação	151
5.4 Adaptação da camada de aplicação	153
5.5 Adaptando as visões para minimizar redundâncias	164
5.6 Conclusão sobre as atividades realizadas no capítulo	167
 6 Code First Migrations, Data Annotations e validações	168

Casa do Código	Sumário
6.1 O uso do Code First Migrations	168
6.2 Atualização do modelo de negócio	171
6.3 O uso de validações	175
6.4 Conclusão sobre as atividades realizadas no capítulo	187
7 Areas, autenticação e autorização	188
7.1 Areas	188
7.2 Segurança em aplicações ASP.NET MVC	191
7.3 Criação de um acesso autenticado	195
7.4 Registro de um novo usuário	200
7.5 Usuário autenticado e o seu logout	205
7.6 Conclusão sobre as atividades realizadas no capítulo	210
8 Uploads, downloads e erros	211
8.1 Uploads	211
8.2 Apresentação da imagem na visão Details	215
8.3 Permitindo o download da imagem enviada	218
8.4 Páginas de erro	222
8.5 Conclusão sobre as atividades realizadas no capítulo	224
9 DropDownList com chamadas AJAX e uso de sessões	226
9.1 Criação e adaptação de classes para o registro de professores	227
9.2 O controlador para professores	229
9.3 A visão para o registro de professores	235
9.4 Actions invocadas via AJAX/jQuery para atualização dos Drop Downs	241
9.5 Armazenando valores na sessão	243
9.6 Conclusão sobre as atividades realizadas no capítulo	247

10 Os estudos não param por aqui

248

Versão: 21.7.8

CAPÍTULO 1

A PRIMEIRA APLICAÇÃO ASP.NET CORE MVC

Olá! Seja bem-vindo ao primeiro capítulo deste livro. Vamos começar vendo implementações e exemplos que lhe darão condições para desenvolver uma aplicação com o ASP.NET Core MVC. Embora o foco da obra seja a prática, não há como fugir de determinados conceitos e teorias, assim, quando essa necessidade surgir, estas apresentações ocorrerão.

Os exemplos trabalhados neste livro terão como contexto uma Instituição de Ensino Superior, chamada daqui para a frente de IES. Neste primeiro capítulo, você implementará uma aplicação que permitirá o registro e a manutenção em dados da instituição, mas sem fazer acesso à base de dados. Sendo assim, como o próprio título deste primeiro capítulo diz, vamos começar já criando nossa primeira aplicação.

Como o livro é sobre ASP.NET Core MVC, antes da primeira prática, é importante entender o que é o ASP.NET Core MVC, como também o que ele não é.

Caso tenha interesse, seguem os links para meus trabalhos anteriores.

- <https://www.casadocodigo.com.br/products/livro-csharp>
- <https://www.casadocodigo.com.br/products/livro-aspnet-mvc5>
- <https://www.casadocodigo.com.br/products/livro-xamarin-forms>
- <http://www.visualbooks.com.br/shop/MostraAutor.asp?proc=191>

O que é o ASP.NET Core MVC?

Em rápidas palavras, ele é um framework da Microsoft que possibilita o desenvolvimento de aplicações web, com o padrão arquitetural MVC (*Model-View-Controller* ou, em português, Modelo-Visão-Controlador). Embora o ASP.NET Core MVC faça uso deste padrão, ele não define uma arquitetura de desenvolvimento por si só.

Agora, descrevendo um pouco mais tecnicamente, o ASP.NET Core MVC é um novo framework de código aberto para a construção de aplicações conectadas pela internet. Ele permite o desenvolvimento e a execução de aplicações em Windows, Mac e Linux, e estas podem ser executadas no .NET Core ou no .NET Framework (versão anterior do .NET).

Em sua estrutura, também é possível criar e manter projetos que respeitem a modularidade proposta pelo padrão arquitetural

MVC. Mas é importante ter claro que, se não modularizarmos nosso sistema, criando um projeto básico do ASP.NET Core MVC, o padrão MVC não será aplicado. Veremos essa modularização no capítulo *Separação da camada de negócio*.

O padrão MVC busca dividir a aplicação em responsabilidades relativas à definição de sua sigla. A parte do **Modelo** trata das regras de negócio, o domínio do problema; já a **Visão** busca levar ao usuário final informações acerca do modelo, ou solicitar dados para registros. O **Controlador** busca integrar a visão com o modelo de negócio. Desta maneira, o ASP.NET Core MVC busca estar próximo deste padrão. Ele traz pastas que representam cada camada do MVC em sua estrutura de projeto, mas não de maneira explícita, separadas fisicamente, como você poderá ver ainda neste capítulo.

Um ponto básico – mas que penso ser interessante reforçar – é que, ao desenvolver uma aplicação, tendo a internet como plataforma, independente de se utilizar o ASP.NET Core MVC ou não, tudo é baseado no processo **Requisição-Resposta**. Isso quer dizer que tudo começa com a solicitação, por parte de um cliente, por um serviço ou recurso.

Cliente, aqui, pode ser um usuário, um navegador ou um sistema; serviço pode ser o registro de uma venda; e o recurso pode ser uma imagem, um arquivo ou uma página HTML. Essa solicitação é a requisição (*request*), e a devolução por parte do servidor é a resposta (*response*). Toda esta comunicação é realizada por meio de chamadas a métodos do protocolo conhecido como HTTP (*HyperText Transfer Protocol*).

A figura a seguir representa esse processo. Ela foi retirada do

artigo *Como funcionam as aplicações web*, de Thiago Vinícius, que o aborda mais detalhadamente. Veja em: <http://www.devmedia.com.br/como-funcionam-as-aplicacoes-web/25888>.

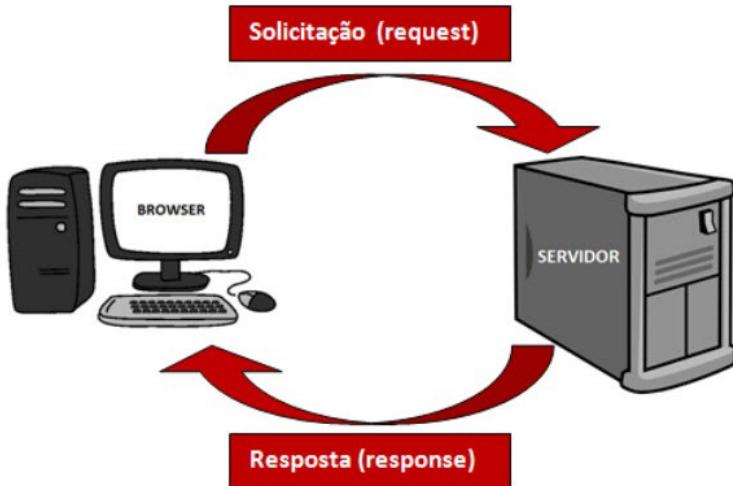


Figura 1.1: Esboço do processo de requisição e resposta de uma aplicação web

Vamos à implementação. Para os projetos usados neste livro, fiz uso do Visual Studio 2017 Community, que você pode obter em: <https://www.visualstudio.com/downloads/>.

Na instalação, selecione as opções como marcadas na figura a seguir. Se você quiser, pode tirar o Desenvolvimento para Desktop e o Desenvolvimento Móvel, pois neste livro não utilizaremos estes recursos. Mão à obra.

	Desenvolvimento com a Plataforma Universal do Windows Crie aplicativos para a Plataforma Universal do Windows com C#, VB, JavaScript ou, como alternativa, C++.	<input checked="" type="checkbox"/>
	ASP.NET e desenvolvimento Web Crie aplicativos Web usando ASP.NET, ASP.NET Core, HTML, JavaScript e CSS.	<input checked="" type="checkbox"/>
	Desenvolvimento móvel com .NET Compile aplicativos de multiplataforma para iOS, Android ou Windows usando Xamarin.	<input checked="" type="checkbox"/>
	Desenvolvimento para desktop com o .NET Compile aplicativos do WPF, do Windows Forms e de console usando o .NET Framework.	<input checked="" type="checkbox"/>
	Processamento e armazenamentos de dados Conecte-se, desenvolva e teste soluções de dados usando SQL Server, Azure Data Lake, Hadoop ou Azure ML.	<input checked="" type="checkbox"/>
	Desenvolvimento de multiplataforma do .NET Core Compile aplicativos multiplataforma usando .NET Core, ASP.NET Core, HTML, JavaScript e ferramentas de...	<input checked="" type="checkbox"/>

Figura 1.2: Opções que devem ser selecionadas para a instalação do Visual Studio 2017 Community Edition

1.1 CRIAÇÃO DO PROJETO NO VISUAL STUDIO 2017 COMMUNITY

Toda aplicação possui alguns domínios que precisam ser persistidos em seu modelo de negócios, e que oferecem algumas funcionalidades. Normalmente, esses domínios também estão

ligados a módulos (ou pacotes) ou ao sistema que está sendo desenvolvido como um todo. Neste início de livro, nosso domínio estará atrelado diretamente ao sistema que está sendo desenvolvido.

Teremos um único módulo, que é a própria aplicação web que criaremos na sequência. Quanto às funcionalidades, a princípio, teremos as conhecidas como básicas, que são: criação, recuperação, atualização e remoção.

Em relação ao sistema proposto neste capítulo, criaremos uma aplicação que permita a execução das funcionalidades anteriormente citadas para fazermos os registros da Instituição de Ensino Superior.

Para a criação de nosso primeiro projeto, com o Visual Studio aberto, selecione no menu a opção Arquivo -> Novo-> Projeto . Na janela que é exibida, na parte esquerda, temos os templates disponíveis (modelos), então, selecione a linguagem Visual C# (1) e, dentro desta categoria, a opção Web (2).

Na área central, marque a opção Aplicativo Web ASP.NET Core (.NET Core) (3). Na parte inferior da janela, informe o nome para o projeto (4), em qual lugar da sua máquina ele deverá ser gravado (5) e, por fim, o nome da solução (6). Clique no botão OK para dar sequência ao processo. A figura a seguir traz a janela em destaque:

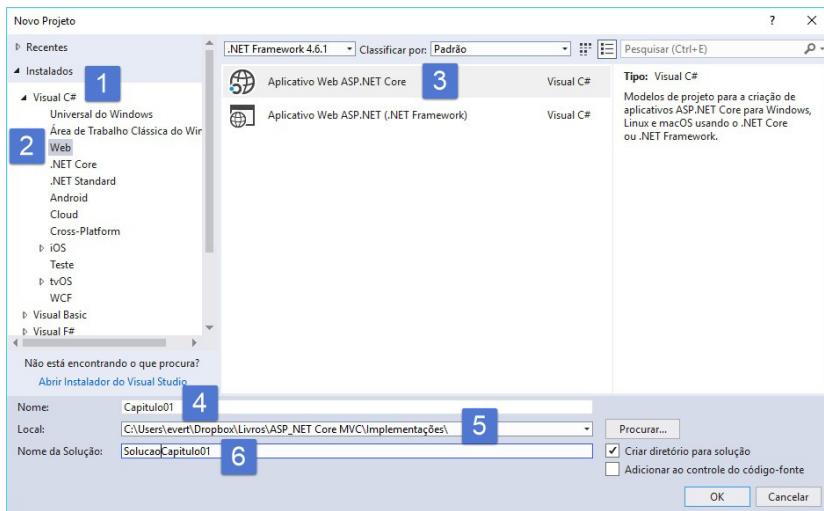


Figura 1.3: Janela do Visual Studio para criação de um projeto web

Na nova janela que se abre, é preciso selecionar qual template de aplicação web deverá ser criado. Selecione Aplicativo Web (1) e deixe a opção de autenticação como Sem autenticação (2). Valide sua janela com a figura a seguir. Clique no botão OK (3) para confirmar a seleção e o projeto a ser criado.

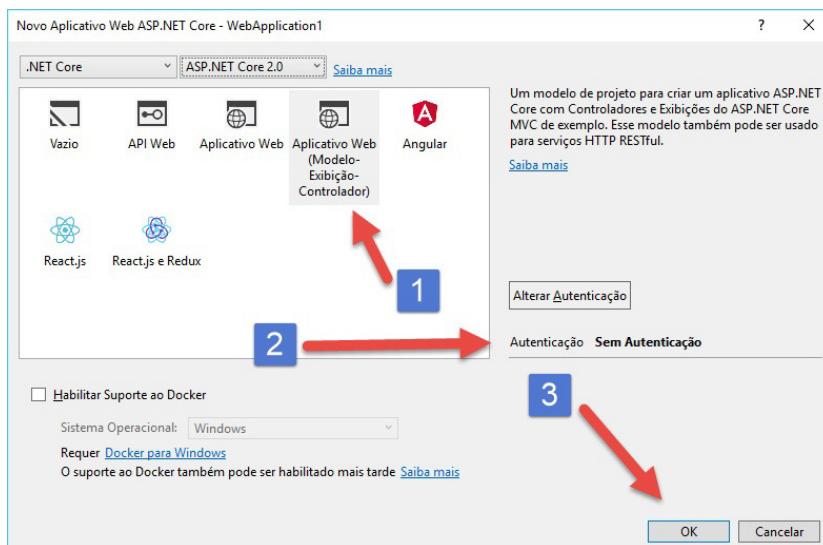


Figura 1.4: Janela de seleção do tipo de projeto web que deverá ser criado

DETALHE IMPORTANTE

Caso a versão que aparece no topo da janela para o ASP.NET Core seja inferior a 2.0, você precisa fechar seu Visual Studio e realizar o download do SDK atualizado, em <https://www.microsoft.com/net/download/core>. Com a versão instalada e disponível, retome a criação do projeto a partir do menu Arquivo -> Novo -> Projeto .

Após a criação, é possível visualizar a estrutura do projeto criada na janela do Gerenciador de Soluções , como mostra a figura adiante. Dependências (1) são as referências que seu projeto faz a componentes e assemblies necessários. Se você

conhece o ASP.NET MVC 5, é algo semelhante às Referências . Se você expandir este item, verá que inicialmente são adicionadas dependências aos Analisadores, Nuget, SDK e Bower; e, dentro destes itens, foram colocados os assemblies específicos.

Caso você ainda não saiba, assemblies tomam a forma de um arquivo executável (.exe) ou arquivo de biblioteca de links dinâmicos (.dll) e são os blocos de construção do .NET Framework. Bower é um gerenciador de dependências mais voltado ao cliente, como CSS, JavaScript, HTML, fontes e até mesmo imagens. Esta ferramenta autointitula-se como "*um gerenciador de pacotes para a web*".

Já em Properties (2), nesta árvore, existe um arquivo chamado `launchSettings.json`, com configurações padrões para o projeto. Recomendo que abra este arquivo (dando um duplo clique) e o investigue-o. Sempre que for utilizar algo sobre ele, apontarei no texto.

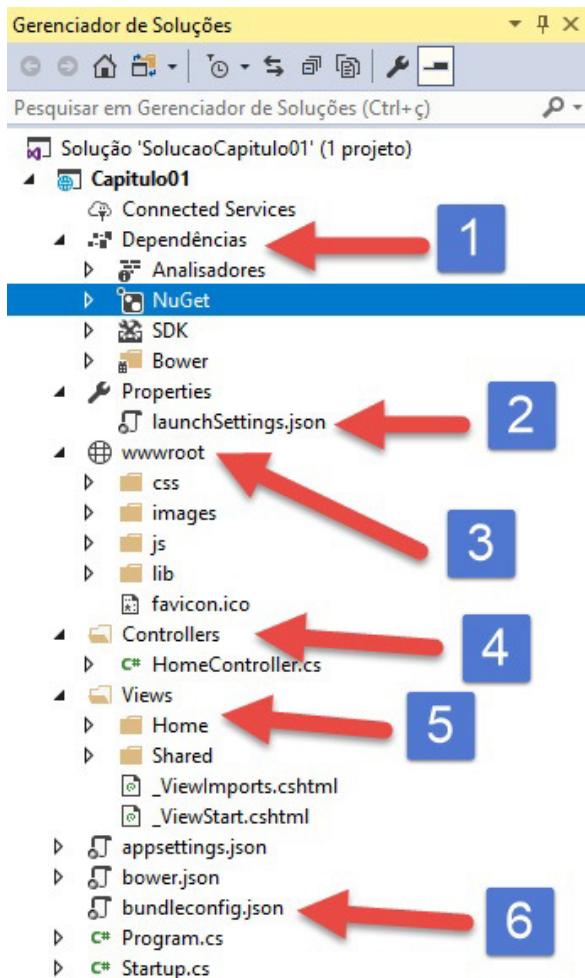


Figura 1.5: Gerenciador de soluções com a estrutura criada para o projeto

A pasta `wwwroot` (3) é o local onde arquivos estáticos (como HTML, CSS, JavaScript e imagens) são armazenados no projeto. A pasta de controladores (4) e a pasta de visões (5) serão detalhadas mais adiante, com uso de prática. Finalizando a explicação da

figura anterior, o template criou o arquivo `Program` (6), que contém o método `Main()`, isto é, o ponto de inicialização da aplicação.

Ao abrir o arquivo, você verá que a classe `Startup` é usada como ponto de inicialização para a aplicação web. Neste ponto, é importante você saber que o ASP.NET Core MVC é um serviço do ASP.NET Core, e a classe `Startup` é um requisito para o ASP.NET Core. É nela que inicializamos serviços, tais como o MVC.

Na figura a seguir, veja o código da classe `Program`. Todo o código apresentado é implementado durante o processo de criação do projeto.

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace Capitulo01
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>() ←
                .Build();
    }
}
```

Figura 1.6: Código da classe `Program`

Vamos testar o projeto criado, com sua execução no navegador. Execute sua aplicação ao clicar no botão `IIS Express`

da barra de tarefas; selecionar o menu Depurar -> Iniciar depuração ; ou, ainda, pressionar F5 . Seu navegador padrão será aberto com uma janela semelhante a apresentada na figura a seguir. Se isso der certo, sua primeira criação de um projeto ASP.NET Core MVC foi concluída com sucesso.

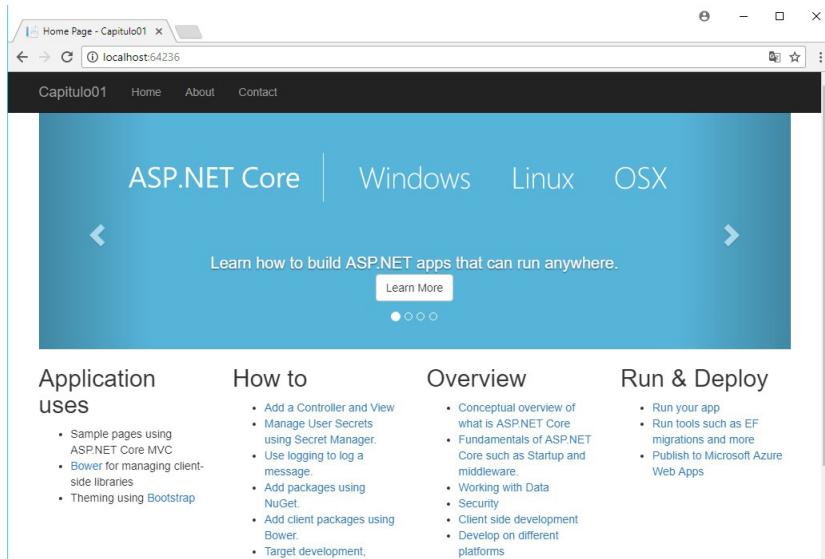


Figura 1.7: Página inicial da aplicação criada

Vamos agora procurar entender como funciona uma aplicação ASP.NET Core MVC. É importante saber que uma requisição é direcionada a uma Action .

Lembre-se de que uma requisição é um pedido do cliente direcionado à action, e esta nada mais é do que um método de uma classe que representa um determinado controlador. Um **Controller** é uma classe que representa a interação com um determinado Modelo de sua aplicação (ou domínio). É no

controlador que são implementados os serviços ofertados, ou seja, as actions.

Como primeiro ponto do domínio a ser trabalhado, temos as *Instituições de Ensino Superior*, que serão representadas pela classe `Instituicao` no modelo. Neste primeiro momento, ela possuirá apenas as propriedades `Nome` e `Endereco`, e precisará de serviços para:

- Obtenção de uma relação com todas as instituições existentes;
- Inserção de uma nova instituição;
- Alteração do nome de uma instituição existente; e
- Remoção de uma instituição.

Em um projeto real, é preciso avaliar a alteração e remoção de dados, pois é possível perder o histórico. Normalmente, para situações assim, recomenda-se uma propriedade que determine se o objeto é ou não ativo no contexto da aplicação.

Por padrão, para a criação dos serviços enumerados anteriormente, são oferecidas as actions do controller para o domínio. Estas podem ser criadas automaticamente pelo Visual Studio, entretanto, para este momento, vamos criá-las nós mesmos. Adotei esta metodologia para que possamos criar nosso projeto do básico, sem os templates oferecidos pelo Visual Studio, para este tipo de funcionalidade.

Particularmente, vejo os templates do Visual Studio como uma ferramenta para investigar alguns dos recursos oferecidos pela plataforma. Portanto, vale apresentar brevemente esses recursos aqui para você.

1.2 CRIANDO O CONTROLADOR PARA INSTITUIÇÕES DE ENSINO

Vamos começar pela criação da camada controladora. No Gerenciador de Soluções , verifique que há uma pasta chamada controllers na estrutura para o seu projeto (que nomeei de Capítulo01). Como a tradução do nome sugere, é nela que ficarão nossos controladores. Clique com o botão direito do mouse sobre ela e, então, na opção Adicionar -> Controlador .

Quando selecionamos o tipo de projeto Aplicação Web (MVC) , as dependências necessárias são inseridas e configuradas em sua criação. Vale a pena você abrir a classe Startup , que contém implementações relacionadas às dependências configuradas para a aplicação. Na sequência, apresento o código dela.

O método Startup() é o construtor da classe, que recebe os parâmetros de configuração para a inicialização do projeto. Veja que há a declaração de um campo que recebe este objeto. O método ConfigureServices() realiza a configuração dos serviços usados na aplicação. Em nosso caso, no momento, temos apenas o MVC.

Finalizando, o método Configure() configura o tratamento de erros – de acordo com o ambiente de execução da aplicação –, determina o uso de arquivos estáticos e configura as rotas para o MVC. Veja:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.Extensions.DependencyInjection;

namespace Capitulo01
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)

        {
            services.AddMvc();
        }

        // Configuração dos serviços que serão utilizados na aplicação
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            // Tratamento de erros
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UseBrowserLink();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
            }

            // Uso de arquivos estáticos
            app.UseStaticFiles();

            // Definição de rotas
            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

```

        });
    }
}
}

```

Com esta etapa concluída, vamos agora criar o controlador para as instituições. Clique com o botão direito na pasta **Controllers** e depois em **Adicionar -> Novo item**. Na janela apresentada, escolha a categoria **web** e, nela, o template **Classe do controlador MVC**. Então, nomeie o controlador de **InstituicaoController** e depois clique em **Adicionar**. Veja a figura a seguir.

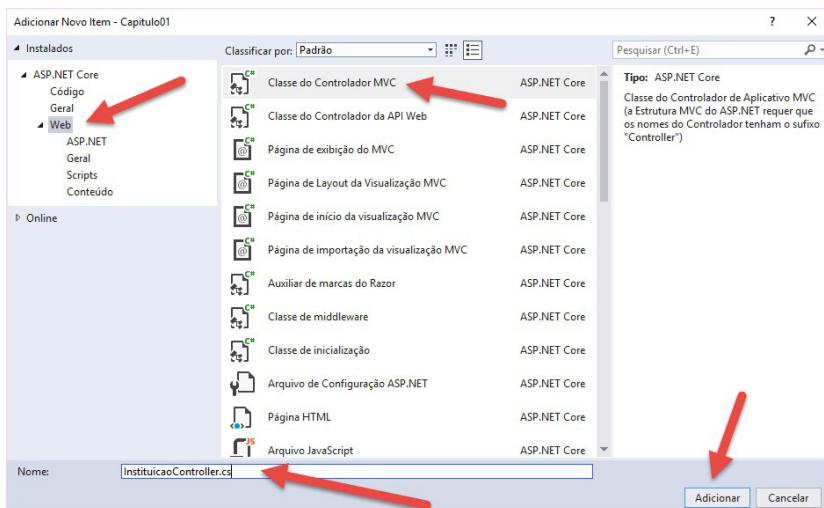


Figura 1.8: Criando um controlador para Instituição

Expliquei a maneira anterior pelo fato de que, em meu ambiente, na criação do primeiro controlador, o Visual Studio não disponibilizou o template para controladores. Após a criação do primeiro, clicando com o botão direito na pasta **Controllers**, foi possível clicar em **Adicionar -> Controlador** e, na janela

apresentada, escolher o template correto para o controlador desejado.

Em nosso exemplo, é o Controlador MVC - Vazio . Assim, clique em Adicionar . Na janela que se abre, confirme o nome e clique em Adicionar .

Se você fez tudo certinho, independente do modo que utilizou, o controlador será criado e exibido. No código a seguir, note que a classe estende Controller , e a action Index() retorna um IActionResult . Perceba um detalhe: na pasta, já existe um controlador, o HomeController , mas deixe-o quietinho lá; depois o veremos.

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo01.Controllers
{
    public class InstituicaoController : Controller
    {
        // Definição de uma action chamada Index
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Controller

Uma classe Controller fornece métodos que respondem a requisições HTTP criadas para uma aplicação ASP.NET Core MVC. Essas respostas são realizadas por métodos action que compõem a classe.

1. Toda classe Controller deve ter seu nome finalizado com

o sufixo `Controller`. Essa obrigatoriedade deve-se ao fato de que o framework busca por esse sufixo.

2. As actions devem ser `public`, para que possam ser invocadas naturalmente pelo framework.
3. As actions não podem ser `static`, pois elas pertencerão a cada controlador instanciado.
4. Por características do framework, as actions não podem ser sobre carregadas com base em parâmetros, mas podem ser sobre carregadas por meio do uso de `Attributes`.

IActionResult

Um objeto da interface `IActionResult` representa o retorno de uma `action`. Existem diversas classes que implementam a interface `IActionResult`, e que representam um nível maior de especialização. Elas serão apresentadas conforme forem usadas.

Para maiores detalhes sobre o retorno que uma action pode dar a uma requisição, recomendo a leitura do artigo *Introduction to formatting response data in ASP.NET Core MVC*, de Steve Smith (<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/formatting>). Se não quiser ler agora, sem problemas, veremos com calma este tipo de retorno durante o livro

View()

O método `View()` (implementado na classe `Controller`) retorna uma visão para o requisitante, e ele possui sobrecargas. A escolha pela versão sem argumentos retorna uma `View` que possui o mesmo nome da `action` requisitada.

Se uma `String` for informada como argumento do método `View()`, ela representará a `View` que deve ser retornada. Existe ainda uma sobrecarga para o método `View()` para o envio dos dados que serão usados na `View` renderizada. Veremos um pouco disso nos exemplos trabalhados.

1.3 CRIAÇÃO DA CLASSE DE DOMÍNIO PARA INSTITUIÇÕES DE ENSINO

Com a estrutura do controlador criada, é preciso definir o modelo que será manipulado por ele. Em uma aplicação ASP.NET Core MVC, as classes de modelo são implementadas na pasta `Models`.

Desta maneira, clique com o botão direito do mouse sobre a pasta `Models` e, então, em `Adicionar -> Classe`. Dê à classe o nome de `Instituicao` e confirme sua criação. Em seu código, verifique que, além das propriedades `Nome` e `Endereco` pertencentes ao domínio, foi implementada a `InstituicaoID`, que terá a funcionalidade de manter a identidade de cada objeto.

No código a seguir, podemos visualizar a classe `Instituicao`, que deve ser criada na pasta `Models`. Caso esta não tenha sido gerada, crie-a clicando com o botão direito do mouse sobre o nome do projeto e em `Adicionar -> Nova Pasta`. Este problema ocorreu comigo em algumas situações – pode ser um

bug do Visual Studio, que certamente será corrigido por futuras atualizações.

```
namespace Capitulo01.Models
{
    public class Instituicao
    {
        public long? InstituicaoID { get; set; }
        public string Nome { get; set; }
        public string Endereco { get; set; }
    }
}
```

1.4 IMPLEMENTAÇÃO DA INTERAÇÃO DA ACTION INDEX COM A VISÃO

Em um projeto real, neste momento teríamos o acesso a dados (via um banco de dados), no qual as instituições registradas seriam devolvidas para o cliente. Entretanto, por ser nosso primeiro projeto, trabalharemos com uma coleção de dados, usando um `List`. Veja no código a seguir essa implementação.

Logo no início da classe, note a declaração `private static IList<Instituicao> instituicoes` com sua inicialização de 5 instituições. O campo é `static` para que possa ser compartilhado entre as requisições. Na action `Index`, na chamada ao método `View()`, agora é enviado como argumento o campo `instituicoes`.

```
using Capitulo01.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace Capitulo01.Controllers
{
    public class InstituicaoController : Controller
    {
```

```

private static IList<Instituicao> instituicoes =
    new List<Instituicao>()
    {
        new Instituicao() {
            InstituicaoID = 1,
            Nome = "UniParaná",
            Endereco = "Paraná"
        },
        new Instituicao() {
            InstituicaoID = 2,
            Nome = "UniSanta",
            Endereco = "Santa Catarina"
        },
        new Instituicao() {
            InstituicaoID = 3,
            Nome = "UniSãoPaulo",
            Endereco = "São Paulo"
        },
        new Instituicao() {
            InstituicaoID = 4,
            Nome = "UniSulgrandense",
            Endereco = "Rio Grande do Sul"
        },
        new Instituicao() {
            InstituicaoID = 5,
            Nome = "UniCarioca",
            Endereco = "Rio de Janeiro"
        }
    };

public IActionResult Index()
{
    return View(instituicoes);
}
}
}

```

Nosso passo seguinte é implementar a visão, ou seja, a página HTML que apresentará ao usuário as instituições registradas na aplicação. Quando criamos o projeto, uma outra pasta criada pelo template (e ainda não trabalhada por nós) foi a `Views`. É nela que criaremos as visões para nossas actions e controladores. Veja-a no

Gerenciador de Soluções .

Para a criação das visões, dentro da pasta `Views`, cada controlador tem uma subpasta com seus nomes por convenção; e nessas subpastas, você encontra as visões que serão requisitadas. Nós poderíamos criar a pasta `Instituicoes` e o arquivo que representará a visão `Index` diretamente, mas faremos uso de recursos do Visual Studio para isso.

A adoção deste critério aqui deve-se ao fato de que escrever códigos HTML é algo muito trabalhoso e, em relação ao template usado pelo Visual Studio, pouca coisa é mudada neste caso. Para garantir que este processo dê certo, dê um `build` em seu projeto (`Ctrl+Shift+B`).

Para iniciar, clique no nome da `action` com o botão direito do mouse e escolha `Adicionar exibição` (particularmente não gosto desta tradução). Na janela que se abre, note que o nome da visão já vem preenchido (1).

No `Modelo` a ser usado, opte pelo `List` (2). Em `Classe de Modelo`, selecione a classe `Instituicao`; caso ela não apareça, o `build` não foi executado de maneira correta e você precisará corrigir os erros apontados pelo Visual Studio (3). Então, desmarque a caixa `Use uma página de layout` (4), mantenha marcado o `Biblioteca de scripts de referência` e clique no botão `Adicionar` para que a visão seja criada.

A figura a seguir apresenta a janela para criação da visão:

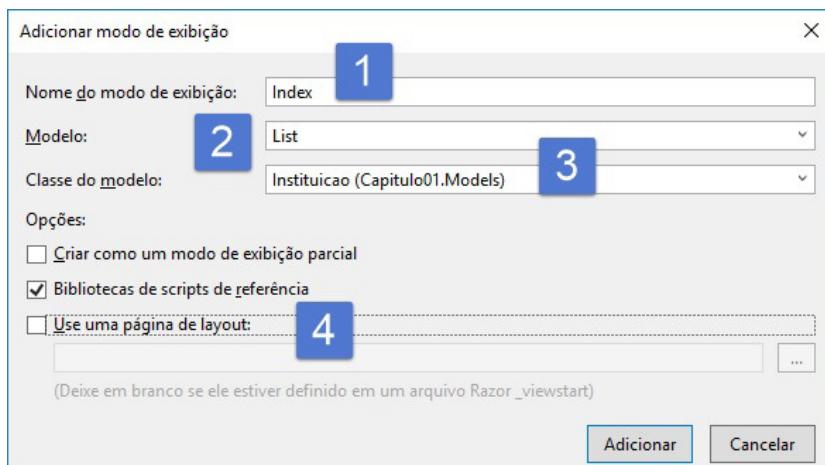


Figura 1.9: Janela para criação da visão

Após a criação da visão, o Visual Studio abre o arquivo que a representa. O que se vê é uma mescla de HTML com código Razor. A primeira implementação Razor , @model IEnumerable<Capitulo01.Models.Instituicao> , define o que a visão recebe e que será definido como modelo para ela.

A implementação <a asp-action="Create">Create New faz uso de um *Tag Helper* (*asp-action*) para que seja renderizado um código HTML que representa um link para uma action – Create , neste caso. Logo à frente, apresento uma explicação específica sobre o que é um Tag Helper, não se preocupe.

A terceira implementação, @Html.DisplayNameFor(model => model.Nome) , já faz uso de um *HTML Helper* (*DisplayNameFor()*) para a definição do título da coluna de dados (da tabela que está sendo renderizada). Note que é feito uso de expressão *lambda* (*model => model.Nome*) para obter um

campo do modelo.

A quarta implementação relevante faz uso de um `foreach` para renderização das linhas da tabela, que representarão as instituições recebidas pela visão. Já a quinta, e última, faz uso de `@Html.DisplayFor()`, que exibirá o conteúdo das propriedades de cada objeto recebido como modelo de dados para a visão.

Para cada linha renderizada, note que três links também são renderizados para: alteração, exclusão e visualização. Estes são direcionados para actions padrões no ASP.NET Core MVC para essas operações. O último argumento enviado para o `@Html.ActionLink()` está comentado, ou seja, não gera saída na visão, pois ele é apenas orientativo para o que você deve inserir.

```
@Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ })
) |
@Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */
y */ }) |
@Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */
*/ })
```

Vamos substituir o comentário por um código válido, mas antes é preciso que você saiba que ele incluirá o texto `/id=valor` na URL gerada pelo ASP.NET Core MVC. Esse valor será usado pela action (`Edit`, `Details` ou `Delete`) para pesquisar o objeto desejado pelo usuário e, assim, fazer processamentos e redirecionamentos.

Veja o código completo da visão gerada, já com essa alteração realizada:

```
@model IEnumerable<Capitulo01.Models.Instituicao>

 @{
    Layout = null;
```

```

}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.InstituicaoID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Nome)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Endereco)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.InstituicaoID)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Nome)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Endereco)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.InstituicaoID }) |
            @Html.ActionLink("Details", "Details", new { id=i

```

```

        tem.InstituicaoID }) |
            @Html.ActionLink("Delete", "Delete", new { id=ite
m.InstituicaoID })
        </td>
    </tr>
}
</tbody>
</table>
</body>
</html>

```

Caso tenha interesse em substituir os HTML Helpers comentados anteriormente por Tag Helpers, poderíamos utilizar o código a seguir:

```

<a asp-action="Edit" asp-route-id="@item.InstituicaoID">Edit</a>
|
<a asp-action="Details" asp-route-id="@item.InstituicaoID">Details</a> |
<a asp-action="Delete" asp-route-id="@item.InstituicaoID">Delete</a>

```

Razor

Razor Pages é uma *view engine*, isto é, uma ferramenta para geração de visões que possibilita a inserção de lógica da aplicação nas visões. Sua sintaxe é simplificada e tem como base o C#.

A inserção de código deve ser precedida do caractere `@`, e blocos de código devem estar delimitados por `{` e `}`. Dentro de um bloco, cada instrução deve ser finalizada com `;` (ponto e vírgula), e é possível fazer uso de variáveis para armazenamento de valores.

Uma boa referência para o Razor pode ser o seguinte link:
<https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor>. Fica a dica para quando você puder acessá-lo.

@model

A declaração de `@model` no início da visão pode ser vista como uma analogia à declaração de um método e de seus parâmetros. Ela habilita o `intellisense` para conhecer qual tipo de dado estamos usando quando o modelo for utilizado pelas *HTML Helpers* e *Tag Helpers*. Além disso, verifica se o tipo passado para a visão pode ser convertido para o tipo esperado, em tempo de execução.

HTML Helpers

Os *HTML Helpers* são métodos que possibilitam a renderização de controles HTML nas visões. Existem diversos deles oferecidos pelo ASP.NET Core MVC e, conforme forem usados, terão sua explicação apresentada. É possível também a implementação de *HTML Helpers* personalizados. Normalmente, o retorno deles é uma String.

Html.ActionLink()

Este método retorna um elemento de âncora do HTML (o `<a href>`), que contém o caminho virtual para uma action em específico. Este método é sobrecarregado, e a versão usada no

exemplo recebe duas Strings e um objeto: a primeira String será com o texto exibido como link, e a segunda receberá o nome da action que será requisitada. O terceiro argumento, um objeto chamado `id`, recebe um valor e é usado para a composição da URL, como parte da rota (veremos sobre isso logo).

Caso precise invocar uma action de outro controlador, é possível usar a versão que recebe três Strings, sendo a terceira o nome do controlador. Existem sobrecargas que permitem enviar atributos HTML, além de valores de rota. Verificaremos isso no decorrer do livro.

Html.DisplayNameFor()

Este método obtém o nome para exibição para o campo do modelo. Normalmente, ele é o nome da propriedade da classe, mas pode ser modificado por `Data Annotations`. Ele também faz uso de expressões *lambda* para obter a propriedade e possui sobrecargas. *Data Annotations* são vistas no capítulo *Code First Migrations, Data Annotations e validações*.

Html.DisplayFor()

Este método obtém o conteúdo da propriedade (do objeto) a ser exibida. A propriedade é informada por meio da expressão *lambda* e possui sobrecargas.

Tag Helpers

Tag Helpers são elementos que podem ser inseridos em tags HTML com habilitação do comportamento no lado do servidor. Em nosso exemplo, fizemos uso do `asp-action="Create"` na tag

HTML `<a>`. Ele é uma alternativa para os HTML Helpers `@Html.ActionLink()` e `@Html.Action()`.

Entretanto, é importante entender que os Tag Helpers não substituem HTML Helpers, e que nem sempre há um Tag Helper para todos os HTML Helpers. Mas é visível que o uso de Tag Helpers deixa o código mais legível e com base no HTML.

Com a implementação da action `Index` e de sua respectiva visão, vamos agora testar e verificar se está tudo certo. Execute sua aplicação, pressionando a tecla `F5` para executar com o Debug ativo, ou `Ctrl-F5` para executar sem o Debug .

Caso apareça uma página de erro, complemente a URL de acordo com o que usei em minha máquina, que foi <http://localhost:64236/Instituicao/Index>. Em sua máquina, lembre-se de que a porta do localhost pode variar.

Então, tente digitar a URL sem o nome da action e veja que você obterá o mesmo resultado. Já vamos falar sobre isso. A figura a seguir exibe a janela do navegador com o resultado fornecido para a visão `Index`, por meio da action `Index`, do controlador `Instituicao` .

Não traduzi os textos exibidos nos links criados, mas, se você julgar necessário para seus testes, fique à vontade para realizar a tradução. Isso fica como uma atividade.

InstituicaoID	Nome	Endereco	
1	UniParaná	Paraná	Edit Details Delete
2	UniSanta	Santa Catarina	Edit Details Delete
3	UniSãoPaulo	São Paulo	Edit Details Delete
4	UniSulgrandense	Rio Grande do Sul	Edit Details Delete
5	UniCarioca	Rio de Janeiro	Edit Details Delete

Figura 1.10: Página HTML que representa a visão Index, retornada pela action Index do controlador Instituicao

Caso você tenha executado sua aplicação com o Debug ativo, é preciso interrompê-la para implementar alterações ou novos códigos no controlador ou nas classes de negócio. Para isso, escolha a opção Stop Debugging no menu Debug , ou clique no botão de atalho para essa opção na barra de tarefas.

1.5 O CONCEITO DE ROTAS DO ASP.NET CORE MVC

Uma URL em uma aplicação ASP.NET Core MVC é formada por segmentos, que compõem rotas. As rotas fazem uso de um recurso chamado Routes Middleware . Middleware é um software **embutido** por meio de injecão de dependências, dentro do pipeline da aplicação, para gerenciar requisições e respostas. Você pode pensar em um pipeline como um túnel por onde passa todo o fluxo de execução da aplicação.

Cada componente de uma rota tem um significado e um papel, executados quando ocorre o processo de requisição. Na URL <http://localhost:64236/Instituicao/Index>, são encontrados os seguintes segmentos:

- `http` , que representa o protocolo a ser utilizado;
- `localhost:64236` , que são o servidor e a porta de comunicação, respectivamente;
- `Instituicao` , que representa o controlador que receberá a requisição; e
- `Index` , que é a action que atenderá à requisição.

Tudo isso é configurável, mas a recomendação é que se mantenha o padrão, salvo necessidade. Essa configuração de rotas pode ser vista na classe `Startup` , no método `Configure` .

Se você acompanhou o desenvolvimento do livro até aqui, deve ter o código apresentado na listagem a seguir ao final de seu método. A chamada ao método de extensão `UseMvc()` adiciona o MVC ao pipeline de execução de requisição.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Observe a chamada ao `routes.MapRoute()` , responsável por adicionar rotas à sua aplicação MVC. Essa rota adicionada recebe o nome `Default` e uma máscara para a URL (`{controller}/{action}/{id}`). Esse nome é o padrão utilizado nas aplicações ASP.NET Core MVC, mas você pode alterar e ter outros.

O terceiro parâmetro para esse método refere-se a valores que serão assumidos para o caso de ausência de valores na requisição. Por exemplo, se requisitarmos <http://localhost:64236>, o controlador usado será o `Home`. Se requisitarmos <http://localhost:64236/Instituicao>, a action utilizada será `Index`; e se não for enviado um terceiro argumento, que seria o `id`, nada é utilizado como padrão, pois ele é configurado como opcional.

O uso desse terceiro argumento pode ser verificado na listagem das instituições. Passe o mouse sobre o link `Edit` da primeira linha de instituições, e você verá <http://localhost:64236/Instituicao/Edit/1> – em que `1` representa o valor do campo `InstituicaoID` do objeto. Verifique os links opcionais dos demais registros. Como atividade, para teste, altere o controlador padrão de `Home` para `Instituicao`, e execute novamente sua aplicação. Viu que agora não apresenta aquela página que vimos quando criamos o projeto?

Vamos retomar o controlador e a visão criada com o projeto, o `HomeController`. Retorne antes a rota para o controlador `Home`. Vamos deixar o padrão.

Abra o controlador `HomeController`. Este traz implementado o método (action) `Index`, além de outros métodos que possuem links na página renderizada pela visão `Index`, que você pode visualizar na pasta `Home`, dentro de `Views`. Tente invocar essas actions no navegador, por exemplo, <http://localhost:64236/Home/About>.

Observe que todas as visões de `Home` exibem o menu superior, com opções. Isso deve-se ao fato de que, na visão `_ViewStart`, está definido o layout `_Layout`, que está na pasta `Shared`.

A visão `_ViewStart` é invocada a cada solicitação de visão. Basicamente, é um código executado a cada requisição de visão efetuada. Mas por que o menu que aparece para as visões Home não aparece na visão `Index` de `Instituicoes`? Porque nós definimos que não vamos utilizar layouts.

Veja o trecho a seguir logo no início da visão `Index` de `Instituicoes`. Trabalharemos, de maneira mais prática, o uso de Layout no capítulo *Layouts, Bootstrap e jQuery DataTable*.

```
@{  
    Layout = null;  
}
```

1.6 IMPLEMENTAÇÃO DA INSERÇÃO DE DADOS NO CONTROLADOR

Na visão `Index`, logo no início, existe a Tag Helper `<a asp-action="Create">Create New`. Traduza-a para `<a asp-action="Create">Criar nova instituição`. O `Create` refere-se à `action` que será alvo do link e que implementaremos conforme o código a seguir.

Deixei o comentário `GET: Create` apenas para poder comentar sobre ele. O método `Index()` e o `Create()` são invocados por meio do `GET` do HTTP, ou seja, pela URL. Quando fizermos uso de formulários HTML, usaremos também chamadas por meio do método `POST` do HTTP.

Na action `Index`, usamos como retorno o `IActionResult`, uma interface, o que permite o uso mais genérico para o retorno. Agora, para a `Create`, utilizamos `ActionResult`, que é uma classe abstrata, na qual diversas outras classes são estendidas.

Quanto mais descer o nível hierárquico das interfaces e classes, mais especializados ficam os tipos de retorno para as actions.

Em nosso caso, o uso desses dois tipos é apenas para exemplificar e apresentar essas possibilidades a você, leitor. Retornos mais especializados serão mostrados no decorrer do livro.

```
// GET: Create  
public ActionResult Create()  
{  
    return View();  
}
```

Precisamos agora criar a visão `Create`. Para isso, seguiremos os passos que tomamos ao criar a visão `Index`. Clique com o botão direito do mouse sobre o nome da action `Create` e clique em `Adicionar exibição`.... Na janela que se apresenta e que já é conhecida, mantenha o nome da visão como `Create`, escolha o template `Create`, selecione `Instituicao` como classe de modelo, e desmarque a opção `Use uma página de layout`.

Clique no botão `Adicionar` para que a visão seja criada. A listagem criada para essa visão é exibida a seguir, em partes, para que explicações possam ser dadas.

Primeira parte da visão `Create`: os elementos iniciais da visão

No código a seguir, é definida a classe para o modelo a ser trabalhado na visão e remetido para o servidor. Como ainda não estamos trabalhando com layout, atribuímos `null` a essa configuração. Há a abertura do documento HTML, definição de seu cabeçalho (`head`) e o início do corpo do HTML (`body`).

```

@model Capitulo01.Models.Instituicao

{@
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Create</title>
</head>
<body>

```

Segunda parte da visão Create: o formulário HTML

No código a seguir, é possível verificar a tag `<form>` , declarando a existência de um formulário HTML. Dentro dessa tag, temos um Tag Helper `asp-action="Create"` , que faz com que, quando o usuário clicar no botão de submissão do formulário, uma action chamada `Create` (agora do método `HTTP POST`) seja requisitada. Ainda não a implementamos, mas logo faremos isso.

Existe um HTML Helper para formulários também, é o `@Html.BeginForm()` . Dentro do elemento `<form>` , existem diversos elementos HTML – como os `<div>` , que geram o layout para o formulário, simulando tabelas com linhas e colunas. Todos os `<div>` possuem um valor para o atributo `class` , referente ao estilo CSS que será utilizado para renderizar cada elemento. Veremos isso em prática quando trabalharemos com o Bootstrap e layouts, no capítulo *Layouts, Bootstrap e jQuery DataTable*.

```

<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Instituicao</h4>
        <hr />

```

Terceira parte da visão Create: os controles de interação com o usuário

Os elementos HTML apresentados no código a seguir são responsáveis pela criação da interface com o usuário. Os controles `<label>` em conjunto com as Tag Helpers `asp-for` renderizarão um texto com o nome da propriedade atribuída à Tag Helper. Isso pode parecer estranho, mas por que não escrevemos diretamente o texto?

É que podemos alterar o nome a ser usado na aplicação. Ele pode ser mais descritivo do que o da propriedade. Veremos isso mais para a frente com Data Annotations, no capítulo *Code First Migrations, Data Annotations e validações*.

Voltando ao código, os elementos HTML `<input>`, em conjunto com as Tag Helpers `<asp-for>`, renderizam um controle de entrada de dados (caixas de textos) para as propriedades que temos. Essas caixas estarão ligadas às propriedades atribuídas à Tag Helper. Desta maneira, quando o objeto da visão for enviado para a action, ele terá os valores informados nessas caixas e saberá a quais propriedades estes se referem. Bem simples, não é?

A primeira instrução do código a seguir possui uma Tag Helper chamada `asp-validation-summary`, que recebe a informação de que as mensagens de validação exibidas deverão ser referentes apenas ao modelo. Veremos sobre validações mais à frente, no capítulo *Code First Migrations, Data Annotations e validações*. Fique tranquilo.

Note que, abaixo dos elementos `<input>`, existe um elemento

 com a Tag Helper `asp-validation-for` , um para cada propriedade que receberá dados por entrada do usuário. Com esta implementação, caso ocorram erros de validação na submissão do formulário – como um campo obrigatório não informado, uma data informada incorretamente, a entrada irregular em um campo (digitar números onde se espera apenas letras) etc. –, mensagens de alerta aparecerão no início do formulário (`summary`) e abaixo de cada controle visual, pelo .

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Nome" class="col-md-2 control-label">
</label>
    <div class="col-md-10">
        <input asp-for="Nome" class="form-control" />
        <span asp-validation-for="Nome" class="text-dange
r"></span>
    </div>
</div>
<div class="form-group">
    <label asp-for="Endereco" class="col-md-2 control-lab
el"></label>
    <div class="col-md-10">
        <input asp-for="Endereco" class="form-control" />
        <span asp-validation-for="Endereco" class="text-d
anger"></span>
    </div>
</div>
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Create" class="btn bt
n-default" />
    </div>
</div>
</div>
</form>
```

Quarta parte da visão Create: incluindo scripts

Antes de falar dos scripts e como são inseridos, note que existe um elemento HTML de link/ancoragem (`<a>`), com um Tag Helper `asp-action` para a visão `Index` que já fizemos. Esta implementação já teve uma explicação quando apresentei a própria visão `Index` .

Quando desenvolvemos aplicações (quer seja web ou não), é comum termos trechos de código que se repetem em diversas partes da aplicação. É assim na Orientação a Objetos, quando criamos classes e métodos. Quando trabalhamos visões no ASP.NET Core MVC, existem também trechos de códigos que são comuns a mais de uma visão, e podemos evitar a redundância ao criar visões parciais e utilizá-las.

No exemplo trabalhado aqui, nossa visão parcial é a que faz referência aos scripts a serem inseridos nas visões. Por isso, sua inserção está na `@section Scripts` .

Devido ao fato de fazermos uso de um método assíncrono, precisamos preceder a chamada a ele com o operador `await` . Dentro da pasta `Views` , existe uma subpasta chamada `Shared` (já falamos dela anteriormente). É nela que as visões parciais são armazenadas, por padrão.

Abra essas pastas e note que lá existe a `_ValidationScriptsPartial.cshtml` . Uma visão com o nome precedido pelo underline indica uma proteção para que ela não seja requisitada diretamente pela URL.

```
<div>
    <a asp-action="Index">Retorno à listagem</a>
</div>
```

```
@section Scripts {
```

```
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
}
</body>
</html>
```

Vamos ver a visão renderizada? Execute seu projeto e, no navegador, requisite a action `Create`. Em minha máquina, usei a URL <http://localhost:64236/Instituicao/Create>. O resultado está na figura a seguir:

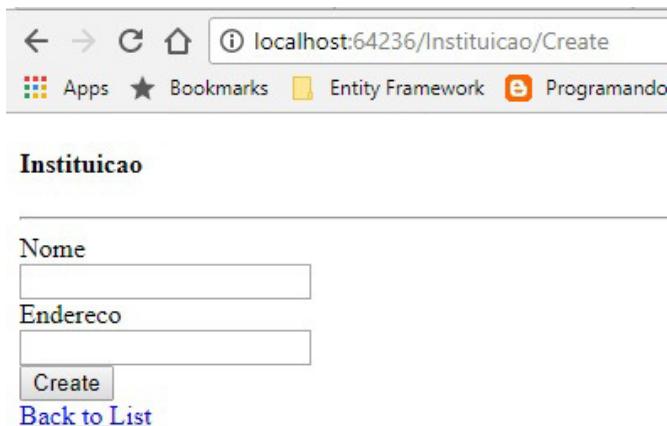


Figura 1.11: Página HTML que representa a visão Create, retornada pela action Create do controlador Instituicao

Implementação da action que recebe o modelo para inserir a visão Create

Como já temos implementadas a action que gera a visão e a visão que receberá os dados para o modelo, precisamos agora implementar uma nova action no controlador `Instituicoes`. Esta será responsável por receber os dados informados na visão e realizar com eles todo o processamento necessário – neste caso, o registro de uma nova instituição.

Ela terá o mesmo nome da criada para retornar a visão de inserção de dados, `create`, pois o `submit` do formulário levará a URL existente no navegador ao servidor. Também precisamos informar que ela será invocada apenas para métodos HTTP `POST`, pois o formulário HTML gerado para a visão foi criado com o método `POST`. Por questões de segurança, ela deve ser decorada para o `AntiForgeryToken`.

A tag `<form asp-action="Create">`, com o Tag Helper, gera o *anti-forgery token* na forma de um campo oculto, o qual é comparado na action. O anti-forgery token insere um código no HTML enviado ao servidor para evitar que sejam enviados dados falsos.

Veja na sequência a listagem da action criada e os atributos, comentados anteriormente, `[HttpPost]` e `[ValidateAntiForgeryToken]`. Observe que o parâmetro do método é um objeto `Instituicao`. Isso é possível graças à declaração do `@model` na visão.

Na inserção, submetemos o nome e o endereço para a categoria, mas o objeto precisa de um identificador para a propriedade `InstituicaoID`. Desta maneira, faremos uso do LINQ para obtermos o valor máximo da coleção para essa propriedade. Vamos incrementá-lo em 1, e atribuímos o resultado à propriedade. Para que possamos trabalhar com o LINQ, precisamos incluir o trecho `using System.Linq;` no início da classe.

LINQ (LANGUAGE INTEGRATED QUERY)

LINQ é uma tecnologia desenvolvida pela Microsoft para fornecer suporte, em nível de linguagem (com recursos oferecidos pelo ambiente), a um mecanismo de consulta de dados para qualquer que seja o tipo desse conjunto, podendo ser matrizes e coleções, documentos XML e base de dados. Existem diversos recursos na web que permitirão uma introdução ao LINQ, mas recomendo a MSDN (<https://msdn.microsoft.com/pt-br/library/bb397906.aspx>).

Neste artigo, existem referências a outros.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Instituicao instituicao)
{
    instituicoes.Add(instituicao);
    instituicao.InstituicaoID =
        instituicoes.Select(i => i.InstituicaoID).Max() + 1;
    return RedirectToAction("Index");
}
```

No momento, nenhum teste foi feito no objeto recebido. Logo trabalharemos com algumas validações. Perceba que o retorno agora é dado pela chamada a `RedirectToAction()`, que recebe o nome da action requisitada antes de a resposta ser gerada.

Desta maneira, após a "gravação" de uma nova instituição, o usuário receberá uma nova visão da action `Index`, que deverá exibir a nova instituição cadastrada, além das já existentes. Realize o teste em sua aplicação, requisitando a action `Create` e registrando uma nova instituição.

1.7 IMPLEMENTAÇÃO DA ALTERAÇÃO DE DADOS NO CONTROLADOR

Como vimos pelas atividades criadas até o momento, estamos tratando a implementação de um CRUD (*Create, Read, Update e Delete* – Criação, Leitura, Atualização e Exclusão) para o modelo `Instituicao`. Já temos a leitura de todos os registros pela action `Index`, e falta a leitura individual, que logo implementaremos. Também implementamos a inclusão (criação) na seção anterior e, agora, trabalharemos para fazer a atualização (update).

Vamos relembrar do exemplo usado para a inserção de dados. Nele, uma operação precisava: de uma action `GET` para gerar a visão de interação com o usuário, e de outra action (HTTP `POST`) para receber os dados inseridos pelo usuário. Agora, implementaremos inicialmente a `GET` action.

Na listagem a seguir, veja que a action recebe um parâmetro, representando o `id` do objeto que se deseja alterar. Também é possível verificar que o método `View()` recebe agora um argumento, ao usar o LINQ novamente. Desta vez, vamos recuperar o primeiro objeto da propriedade `InstituicaoID`, para termos o valor recebido pelo parâmetro `id`.

```
public ActionResult Edit(long id)
{
    return View(instituicoes.Where(
        i => i.InstituicaoID == id).First());
}
```

Com o método que retornará a visão para alteração de dados devidamente implementado e enviando à visão do objeto que se deseja alterar, vamos seguir os mesmos passos já trabalhados. Crie a visão de alteração, mudando apenas o template, que será o

Edit . Na sequência, é possível ver o código gerado. No código da visão Edit , a única mudança em relação à visão Create será a implementação, .

Quando trabalhamos a visão Create , o valor de identidade do objeto (propriedade InstituicaoID) foi inserido pela aplicação, na action POST Create . Agora, na alteração, esse valor será a chave para conseguirmos alterar o objeto correto. Porém, pela implementação do type hidden para o input de InstituicaoID , o id da Instituição não é exibido ao usuário. Então, por que tê-lo?

A resposta é: para que o modelo enviado possua o valor dessa propriedade ao submeter a requisição. A action alvo só recebe valores que estão no modelo da visão.

Depois de tudo implementado e testado, retire esse método da visão e tente alterar um objeto. Fica esta atividade para você. Se optar por um HTML Helper para os controles ocultos, é possível usar o @Html.HiddenFor() .

```
@model Capitulo01.Models.Instituicao

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Edit</title>
</head>
<body>
```

```

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Instituicao</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-dange
r"></div>
        <input type="hidden" asp-for="InstituicaoID" />
        <div class="form-group">
            <label asp-for="Nome" class="col-md-2 control-label">
                <div class="col-md-10">
                    <input asp-for="Nome" class="form-control" />
                    <span asp-validation-for="Nome" class="text-dange
r"></span>
                </div>
            </div>
            <div class="form-group">
                <label asp-for="Endereco" class="col-md-2 control-lab
el"></label>
                <div class="col-md-10">
                    <input asp-for="Endereco" class="form-control" />
                    <span asp-validation-for="Endereco" class="text-d
anger"></span>
                </div>
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <input type="submit" value="Save" class="btn btn-
default" />
                </div>
            </div>
        </div>
    </div>
</form>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
}
</body>
</html>

```

Html.HiddenFor()

Este helper renderiza um elemento HTML `<input type="hidden">` para a propriedade retornada pela expressão lambda. É muito comum o seu uso para passar valores da visão ao controlador, sem a interferência do usuário. Entretanto, note que o uso do Tag Helper facilita muito e deixa o código mais limpo.

Implementação da action que recebe o modelo para alteração da visão Edit

Para finalizar, é preciso criar a action `Edit` que vai responder à requisição HTTP `POST`. Na sequência, apresento o seu código. Execute sua aplicação, e acesse o link `Edit` da visão `Index` de uma das instituições (renomeie o link para `Alterar`).

Depois, altere o nome da instituição, grave e veja a nova listagem. A instituição com o nome atualizado aparecerá no final dela, pois removemos a instituição alterada, para então inseri-la novamente.

É possível alterar a listagem das instituições para que elas fiquem em ordem alfabética. Para isso, na action `Index`, adapte a implementação para: `return View(instituicoes.OrderBy(i => i.Nome));`. Observe que a letra utilizada para o parâmetro foi `i`. Não há uma regra para a definição da letra, mas recomenda-se uma convenção – em nosso caso, foi o tipo do dado recebido pelo método.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(Instituicao instituicao)
{
    instituicoes.Remove(instituicoes.Where(
        i => i.Id == instituicao.Id));
    return RedirectToAction("Index");
}
```

```
i => i.InstituicaoID == instituicao.InstituicaoID)
    .First());
instituicoes.Add(instituicao);
return RedirectToAction("Index");
}
```

Uma maneira alternativa para alterar um item da lista, sem ter de removê-lo e inseri-lo novamente, é usar a implementação `instituicoes[instituicoes.IndexOf(instituicoes.Where(i => i.InstituicaoID == instituicao.InstituicaoID).First())] = instituicao;`.

Aqui o `List` é manipulado como um `array` e, por meio do método `IndexOf()`, sua posição é recuperada, com base na instrução LINQ `Where(i => i.InstituicaoID == instituicao.InstituicaoID).First()`. O correto é realmente alterarmos o objeto existente no momento em que utilizarmos a base de dados; esta é a técnica que vamos adotar.

1.8 IMPLEMENTAÇÃO DA VISUALIZAÇÃO DE UM ÚNICO REGISTRO

A visão `Index` traz ao navegador todos os dados disponíveis no repositório em questão – em nosso caso, em um `List`. Há situações em que os dados exibidos não refletem todas as propriedades da classe que tem os objetos desejados, e que o usuário precisa verificar todos os dados de determinado registro.

Por exemplo, uma listagem de clientes pode trazer apenas os nomes dos clientes, mas em determinado momento, é preciso verificar todos os dados desse cliente. Para isso, existe um link na listagem, chamado `Details` (traduzido para `Mais detalhes`).

Como não haverá interação com o usuário na visão a ser gerada, implementaremos apenas a action HTTP GET , conforme a listagem a seguir. Observe que o código é semelhante ao implementado para a action Edit .

```
public ActionResult Details(long id)
{
    return View(instituicoes.Where(
        i => i.InstituicaoID == id).First());
}
```

A criação de visões já foi aplicada e praticada, logo, você já sabe como criá-las. Mas para gerar a da visão Details , aí vai mais uma ajudinha para o caso de você ter se esquecido: clique com o botão direito do mouse sobre o nome da action Details , confirme a criação da visão, mantenha o nome Details , e selecione o modelo Details e a classe de modelo Instituicao . Agora, desmarque a criação com base em um template e confirme a adição da visão.

A listagem gerada deve ser semelhante à apresentada na sequência:

```
@model Capitulo01.Models.Instituicao

#{@
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Details</title>
</head>
<body>

<div>
```

```

<h4>Instituicao</h4>
<hr />
<dl class="dl-horizontal">
    <dt>
        @Html.DisplayNameFor(model => model.InstituicaoID)
    </dt>
    <dd>
        @Html.DisplayFor(model => model.InstituicaoID)
    </dd>
    <dt>
        @Html.DisplayNameFor(model => model.Nome)
    </dt>
    <dd>
        @Html.DisplayFor(model => model.Nome)
    </dd>
    <dt>
        @Html.DisplayNameFor(model => model.Endereco)
    </dt>
    <dd>
        @Html.DisplayFor(model => model.Endereco)
    </dd>
</dl>
</div>
<div>
    @Html.ActionLink("Edit", "Edit", new { id = Model.InstituicaoID })
    <a asp-action="Index">Back to List</a>
</div>
</body>
</html>

```

Os links oferecidos nessa visão já foram implementados anteriormente, que são o referente à alteração (`Edit`) e o que exibe toda a listagem de `Instituicao` (action `Index`). Não existe nada de novo no código gerado. Entretanto, note o link criado para a visão `Edit` ao final da listagem, e veja que é preciso acertar o parâmetro enviado a ela.

Teste sua aplicação agora e verifique se a visão `Details` é renderizada. Aproveite e teste o link `Alterar` , para verificar se você será redirecionado à visão de alteração de dados (`Edit`).

1.9 FINALIZAÇÃO DA APLICAÇÃO: A OPERAÇÃO DELETE

Quando usamos os templates do Visual Studio para a criação de uma aplicação ASP.NET Core MVC, ele traz a operação de *delete* de uma maneira que os dados são exibidos ao usuário, para que então ele confirme a exclusão – tal qual vimos com o `Details`. O primeiro passo é implementarmos a action `Delete` que capturará a requisição HTTP GET.

Novamente, faremos igual às actions `Edit` e `Details`. Veja o código:

```
public ActionResult Delete(long id)
{
    return View(instituicoes.Where(
        i => i.InstituicaoID == id).First());
}
```

Seguindo o padrão para a criação de visões, escolha o template `Delete` na janela de adição de visões. A visão criada é apresentada no código a seguir. Observe que nele aparece a tag `<form>`, com o Tag Helper para a action `Delete`, que encapsula o elemento HTML `<input type="submit">`. Não existe nada de novo, além desta observação.

```
@model Capitulo01.Models.Instituicao

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
```

```

<title>Delete</title>
</head>
<body>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Instituicao</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.InstituicaoID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.InstituicaoID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Nome)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Nome)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Endereco)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Endereco)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-de
fault" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
</body>
</html>

```

Para finalizar a aplicação, precisamos implementar o método que realmente removerá uma instituição. Essa implementação dá-se pela action `Delete`, capturada por uma requisição HTTP

POST . Veja o código na sequência.

Infelizmente, o objeto que chegará à action estará com as propriedades nulas, pois não existe nenhuma Tag Helper ou HTML Helper que as utilize dentro do formulário HTTP. Tente executar sua aplicação e veja que nada será excluído.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(Instituicao instituicao)
{
    instituicoes.Remove(instituicoes.Where(
        i => i.InstituicaoID == instituicao.InstituicaoID)
        .First());
    return RedirectToAction("Index");
}
```

Como o único valor necessário para a remoção da instituição é o seu `id`, e não é de interesse que ele seja exibido ao usuário, insira o elemento `<input type="hidden" asp-for="InstituicaoID" />` abaixo da tag `<form>`. Assim, a propriedade `InstituicaoID` fará parte do conteúdo enviado à action, mas ficará oculto ao usuário no formulário HTML.

Teste novamente sua aplicação, verificando se a visão que exibe os dados do objeto a ser removido é exibida. Quando ela aparecer, confirme a remoção do objeto e veja se a visão `Index` é renderizada, sem o objeto que foi removido.

1.10 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Parabéns, você chegou ao final do primeiro capítulo do livro, e certamente já tem pronta sua primeira aplicação em ASP.NET Core MVC. É uma aplicação simples, entretanto, o foco não foi a

complexidade, mas, sim, o processo.

Foram apresentados conceitos de controllers, actions, views, razor e roteamento. Ao utilizarmos uma Collection, conseguimos criar uma aplicação CRUD. Na sequência, faremos uso de uma base de dados e trabalharemos com o Entity Framework. Mas estamos apenas começando, temos muita coisa para ver ainda.

CAPÍTULO 2

ACESSO A DADOS COM O ENTITY FRAMEWORK CORE

O Entity Framework Core (ou apenas EF Core) é um framework para mapeamento de objetos para um modelo relacional e de um modelo relacional para objetos (ORM – *Object Relational Mapping*). De maneira mais detalhada, um ORM é responsável por mapear objetos em registros (classes em tabelas) e permitir a recuperação e manutenção dos dados relacionais, seguindo o paradigma orientado a objetos.

Por meio do EF Core, é possível trabalhar com dados relacionais usando objetos da camada de negócio. Desta maneira, eliminamos a codificação de acesso a dados diretamente na aplicação, como o uso explícito de instruções SQL.

Com essas características apresentadas, este capítulo tem por objetivo introduzir o Entity Framework Core como ferramenta para persistência e interação com uma base de dados. Não faz parte do escopo deste capítulo um aprofundamento no tema, que, por si só, já é assunto único de diversos livros.

Além disso, como o EF Core será usado em todos os exemplos

a partir daqui, novas características serão apresentadas conforme forem necessárias. Faremos uso do SQL Server LocalDB, que é uma versão reduzida (porém com muitos recursos) do SQL Server Express; e se você seguiu as recomendações, este foi instalado em conjunto com o Visual Studio Community.

Em relação ao projeto começado no capítulo anterior, concluímos um CRUD para `Instituições`, mas nada foi persistido em uma base de dados. Ou seja, se a aplicação parar, os dados serão perdidos. Com o Entity Framework Core, este problema será resolvido, pois, com ele, será possível persistir nossos objetos em uma base de dados.

2.1 COMEÇANDO COM O ENTITY FRAMEWORK CORE

Para iniciar as atividades relacionadas ao desenvolvimento com o EF Core, o primeiro passo é obter os recursos necessários para seu funcionamento. Para usar o SQL Server e o EF Core, é preciso ter na aplicação o acesso ao pacote `Microsoft.EntityFrameworkCore.SqlServer`, que está dentro do pacote `Microsoft.AspNetCore.All`.

Tudo isso pode ser verificado no Gerenciador de Soluções, em Dependências -> Nuget. Essa dependência já estará registrada se você seguiu a orientação dada no capítulo anterior, na criação do projeto.

Dando sequência ao nosso projeto, vamos implementar funcionalidades para uma nova classe do modelo de negócios, a `Departamento`, responsável pelos dados relacionados aos

departamentos acadêmicos da instituição de ensino. Sendo assim, criaremos essa classe na pasta `Models`, conforme o código apresentado na sequência.

Optei por criar um novo projeto, mas você pode continuar com o mesmo do capítulo anterior. A diferença do seu código para o meu estará nos namespaces.

```
namespace Capitulo02.Models
{
    public class Departamento
    {
        public long? DepartamentoID { get; set; }
        public string Nome { get; set; }
    }
}
```

Com a classe de modelo criada, já é possível começar a preparar o projeto para o uso do Entity Framework Core. O primeiro passo é a criação do contexto, que representará a conexão com a base de dados.

Criação do contexto com a base de dados

Inicialmente, é importante ressaltar que, quando temos classes do modelo de negócio que serão mapeadas para tabelas na base de dados relacional, estas são vistas como **entidades**. Este termo vem do modelo relacional, visto normalmente em disciplinas relacionadas a bancos de dados. Para nós, é importante saber que uma entidade é um conceito mapeado para uma tabela – em nosso caso, a classe.

Se você verificar o código da classe que criamos anteriormente, temos uma propriedade identificadora, que no modelo relacional é conhecida como chave primária. O EF Core mapeia

automaticamente a propriedade que tem o nome da classe e a palavra `ID` em seu próprio nome, como chave primária.

Para que nossa aplicação possa se beneficiar com o Entity Framework Core, é preciso que ele acesse a base de dados por meio de um contexto. Este representará uma sessão de interação da aplicação com a base de dados, seja para consulta ou atualização.

Para o EF Core, um contexto é uma classe que estende `Microsoft.EntityFrameworkCore.DbContext`. Desta maneira, crie em seu projeto uma pasta chamada `Data` e, dentro dela, uma classe `IESContext.cs`, que deverá ter esta extensão implementada, tal qual é mostrado no código a seguir.

```
using Capitulo02.Models;
using Microsoft.EntityFrameworkCore;

namespace Capitulo02.Data
{
    public class IESContext : DbContext
    {
    }
}
```

Devido à estratégia de inicialização que adotei para o EF Core, e que será apresentada adiante, precisamos implementar nesta classe um construtor que vai receber opções de configurações para o contexto e vai remetê-las para o construtor da classe base. Veja o seu código a seguir, que deve estar entre os delimitadores da classe.

```
public IESContext(DbContextOptions<IESContext> options) : base(options)
{
}
```

O próximo passo refere-se ao mapeamento de nossa classe para o modelo relacional. Definimos isso por meio do tipo de dados

`DbSet` , como pode ser visto na sequência. A propriedade apresentada deve estar no mesmo nível do construtor.

```
public DbSet<Departamento> Departamentos { get; set; }  
public DbSet<Instituicao> Instituicoes { get; set; }
```

Se mantivermos nossa classe de contexto da maneira implementada até o momento, uma tabela chamada `Departamentos` será criada na base de dados, pois este é o nome da propriedade que mapeia a classe `Departamento` . Caso você já possua uma base de dados com essa tabela em um nome diferente, ou queira mudar o seu nome, é possível usar a sobrescrita do método `OnModelCreating()` , como a seguir:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)  
{  
    base.OnModelCreating(modelBuilder);  
    modelBuilder.Entity<Departamento>().ToTable("Departamento");  
}
```

Outro método que podemos sobrescrever é o `OnConfiguring()` , que nos permite configurar o acesso ao banco de dados. Apresento-o na sequência, mesmo esta não sendo a opção adotada por mim neste livro, já que tratarei a inicialização do EF no momento da inicialização da aplicação.

Configuraremos o acesso ao banco de dados por meio da classe `Startup` , como veremos após o código. Reforço que a listagem a seguir é apenas para você saber como configurar o acesso ao banco de dados diretamente pela classe de contexto. Você não precisa implementá-la para seguir o livro.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  
{  
    optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;D
```

```
atabase=IESUtfpr;Trusted_Connection=True;MultipleActiveResultSets  
=true");  
}
```

O ASP.NET Core implementa a Injeção de Dependência (*Dependency Injection*, ou DI) por padrão. Injeção de Dependência é um *pattern* que traz um recurso necessário para a aplicação, assim, ela não precisa buscar por este serviço. No código seguinte, a injeção está no argumento do método `ConfigureServices()` e na declaração da propriedade de leitura `Configuration`.

O `services` é injetado no método para ser utilizado. Ao usarmos esse argumento, realizaremos a configuração necessária pelo EF Core para acessar uma base de dados do SQL Server.

Serviços podem ser registrados com DI durante a inicialização da aplicação – que foi a minha escolha aqui –, de acordo com o contexto da base de dados para o EF Core. Os componentes que precisam desses serviços vão recebê-los por meio de parâmetros (do construtor ou métodos), ou pela definição de uma propriedade por seu tipo de dado.

Vamos implementar isso na classe `Startup`, no método `ConfigureServices()`, tal qual o código a seguir. No âmbito do EF Core, o contexto refere-se ao ambiente onde os objetos recuperados da base de dados se encontram na aplicação. Serviços são oferecidos à aplicação – como o uso do SQL Server, em nosso exemplo.

Ao digitar o código da listagem, você precisará inserir os namespaces `Capitulo02.Data` (ou outro que você tenha criado) e `Microsoft.EntityFrameworkCore` no início da classe, nos `usings`.

```
public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<IESContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("IESConnection")));
}

services.AddMvc();
}
```

Com o contexto para o EF Core já implementado, precisamos agora configurar a aplicação para acessar a base de dados por meio da `Connection String`. Esta é obtida pelo método `GetConnectionString()`, de `Configuration`, como visto no código anterior, como `IESConnection`. Essa implementação deve ser realizada no arquivo `appsettings.json`. Seu o código está adiante.

A String de conexão usa especificamente o SQL Server LocalDB, como já comentado anteriormente. Este banco é recomendado para o desenvolvimento, não para a produção. Ele é inicializado por demanda e para cada usuário, não sendo necessária nenhuma configuração adicional.

Os arquivos criados possuem a extensão `.MDF` e ficam na pasta do usuário, dentro de `c:\Usuários` (`c:/Users`). No momento, criaremos todos os recursos no projeto MVC, mas logo começaremos a tratar uma arquitetura que será implementada em camadas.

```
{
  "ConnectionStrings": {
    "IESConnection": "Server=(localdb)\\mssqllocaldb;Database=IES
Utfpr;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
}
```

```
        "Logging": {
            "IncludeScopes": false,
            "LogLevel": {
                "Default": "Warning"
            }
        }
    }
```

Se lembrarmos do projeto criado no capítulo anterior, implementamos primeiro a action e a visão para a listagem das instituições (Index). Agora estamos trabalhando com dados de uma base de dados, não mais como uma coleção, como anteriormente. Para efeito didático, é interessante que nossa aplicação já possua dados quando for executada.

O EF Core oferece recursos para podermos inserir dados quando o modelo for inicializado e a base de dados for criada. Para isso, na pasta `Data`, vamos criar a classe `IESDbInitializer`, assim como o código a seguir.

Observe que a classe é estática. A criação da base de dados é dada quando o método `EnsureCreated()` é invocado. Inicialmente, é verificado se existe algum objeto/registro em `Departamentos` via LINQ.

```
using Capitulo02.Models;
using System.Linq;

namespace Capitulo02.Data
{
    public static class IESDbInitializer
    {
        public static void Initialize(IESContext context)
        {
            context.Database.EnsureCreated();

            if (context.Departamentos.Any())
            {
                return;
            }
        }
    }
}
```

```

        }

        var departamentos = new Departamento[]
        {
            new Departamento { Nome="Ciéncia da Computação"},
            new Departamento { Nome="Ciéncia de Alimentos"}
        };

        foreach (Departamento d in departamentos)
        {
            context.Departamentos.Add(d);
        }
        context.SaveChanges();
    }
}

```

Para que a base de dados seja criada, populada e disponibilizada para a sua aplicação, por meio dos controladores e das visões, precisamos modificar o método `main()` da classe `Program`, tal qual segue a listagem. Nela, foi inserida a cláusula `using` para a criação do escopo de execução da aplicação. Depois, obtemos o contexto do EF Core e realizamos a inicialização da base de dados implementada no código anterior.

```

using Capitulo02.Data;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System;

namespace Capitulo02
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {

```

```

        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<IES
Context>();
            IESDbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILog
ger<Program>>();
            logger.LogError(ex, "Um erro ocorreu ao popul
ar a base de dados.");
        }
    }

    host.Run();
}

// Código omitido
}

```

Execute sua aplicação. Ela pode demorar um pouco mais para inicializar e exibir a visão definida nas rotas como padrão, mas é normal. Lembre-se de que sua base de dados está sendo criada e populada.

DbSet

Propriedades da classe `DbSet` representam entidades (`Entity`), que são utilizadas para as operações de criação, leitura, atualização e remoção de objetos (registros da tabela). Desta maneira, se existe uma tabela que será manipulada (por sua aplicação) na base de dados, pelo EF Core, é importante que haja uma propriedade que a represente na definição do contexto. É importante ressaltar que essa tabela a ser manipulada deve ter uma classe que a mapeie.

Connection String

Uma *Connection String* (ou String de conexão) é uma String que especifica informações sobre uma fonte de dados e sobre como acessá-la. Por meio de código, ela passa informações para um *driver* ou *provider* do que é necessário para iniciar uma conexão.

Normalmente, a conexão é para uma base de dados, mas também pode ser usada para uma planilha eletrônica ou um arquivo de texto, dentre outros. Uma Connection String pode ter atributos, como nome do driver, servidor e base de dados, além de informações de segurança, como nome de usuário e senha.

A escrita de uma Connection String pode variar de acordo com o banco de dados e seu modo de acesso. Para auxiliar nesta atividade, cada produto oferece a informação de como criar uma String de conexão, precisando apenas recorrer à documentação disponibilizada.

É possível minimizar a dificuldade. Existe um site que é referência no assunto, o <http://www.connectionstrings.com>. Ele fornece tutoriais, dicas e artigos relevantes.

Local Data Base – LocalDB

O LocalDB é uma versão simplificada (mas com muitos recursos) do SQL Server Express. Ele tem como foco os desenvolvedores, pois auxilia na redução dos recursos necessários na máquina de desenvolvimento. O LocalDB é instalado com o Visual Studio.

Nosso projeto já está configurado para usar o EF Core e para acessar a base de dados (criada e utilizada por ele). Neste momento, poderíamos adaptar a classe `Instituicao` e seus serviços para utilizar o Entity Framework Core, mas optei por usar a classe `Departamento`, que criamos anteriormente. A adaptação de `Instituicao` ficará como atividade.

Na Connection String criada e apresentada anteriormente, o valor `(localdb)\mssqllocaldb` refere-se ao nome do serviço em minha máquina. Você precisará ver como esse serviço está instalado na sua, para corrigir o valor, se necessário. Você pode recorrer aos Serviços locais do Windows para isso.

2.2 VERIFICANDO A BASE DE DADOS CRIADA NO VISUAL STUDIO

O Visual Studio oferece ferramentas para visualização e manutenção de dados e da estrutura de uma base de dados. Porém, não modificaremos a estrutura das tabelas criadas, pois nossa modelagem está centrada em objetos. Ou seja, qualquer mudança necessária deverá ser realizada nas classes.

Entretanto, em relação aos dados, é possível que você queira inseri-los, visualizá-los, alterá-los ou removê-los diretamente no Visual Studio. Um recurso do IDE simples para visualização da base de dados criada é o `Pesquisador de Objetos do SQL Server`. Você pode abri-lo a partir do menu `Exibir`.

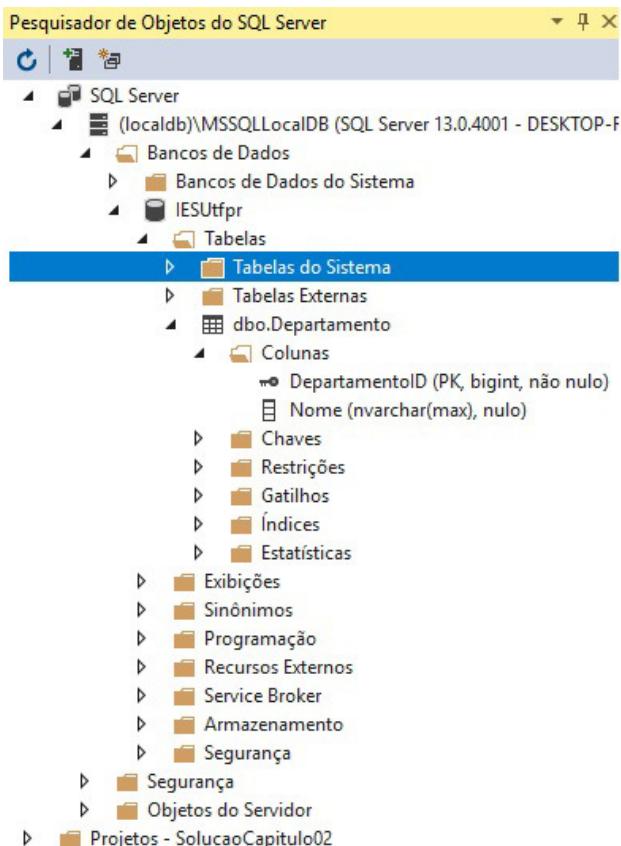


Figura 2.1: Pesquisador de Objetos do SQL Server

2.3 IMPLEMENTAÇÃO DO CRUD FAZENDO USO DO ENTITY FRAMEWORK CORE

Nesta seção, implementaremos as funcionalidades de recuperação e de atualização dos dados relacionados aos departamentos acadêmicos da instituição. Para implementar as funcionalidades que serão oferecidas ao Departamento , precisamos criar um novo controlador, chamado

`DepartamentoController .`

Assim, clique com o botão direito do mouse na pasta `Controllers`, depois em `Adicionar e Controlador`. Então, escolha `Controlador MVC - Vazio`, tal qual fizemos no capítulo *A primeira aplicação ASP.NET Core MVC*.

Visualização dos dados existentes pela action/view Index

Com o controlador criado, precisamos criar as actions que atenderão às requisições do navegador. Seguindo o exemplo do capítulo anterior, implementaremos a action `Index`, de acordo com a listagem a seguir.

No início da classe, observe a declaração do campo `_context`. Este objeto será utilizado por todas as actions. Verifique também a existência de um construtor, que recebe um objeto de `IEntitySetContext`. Este será encaminhado por Injeção de Dependência, assim como configuramos anteriormente, na classe `Startup`.

Na action `Index`, o objeto encaminhado ao método `View()` é agora a coleção de objetos (registros) existentes na coleção de departamentos `DbSet` (tabela), que pertence ao objeto `_context`. Esses departamentos estão classificados por ordem alfabética e pelo nome de cada um.

```
using Capitulo02.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;
```

```
namespace Capitulo02.Controllers
{
    public class DepartamentoController : Controller
    {
        private readonly IESContext _context;

        public DepartamentoController(IESContext context)
        {
            this._context = context;
        }

        public async Task<IActionResult> Index()
        {
            return View(await _context.Departamentos.OrderBy(c =>
c.Nome).ToListAsync());
        }
    }
}
```

Veja com atenção a declaração e o comportamento do método que representa a action `Index`. Ele retorna um `Task<IActionResult>`, de maneira assíncrona, por isso o `async` antecede o tipo de retorno.

Na instrução `return`, o argumento do método `View()` começa com `await`, e a coleção é retornada como `ToListAsync()`. Tudo isso garante que a programação assíncrona esteja implementada em sua action, pois é este o padrão para desenvolvimento com o ASP.NET Core MVC e o Entity Framework Core.

Imagine um servidor web que recebe diversas requisições, existindo a possibilidade de algumas ocorrerem ao mesmo tempo; o que poderá acontecer? Um servidor web tem um limite de threads disponíveis e, em algum momento, todas podem estar ocupadas, indisponíveis. Com isso, o servidor não poderá processar novas requisições, até que alguma volte a ficar livre.

Quando usamos um código síncrono, pode acontecer de algumas threads estarem ocupadas enquanto estão aguardando processamentos de entrada e saída (I/O – E/S) serem concluídos. Já quando usamos o código assíncrono, uma thread aguarda a conclusão de algum processo de E/S, e seu segmento é liberado para o servidor poder utilizar em novas requisições.

Assim, recursos podem ser utilizados de maneira mais eficiente, e o servidor responderá bem quando tiver tráfego intenso. Recomendo que dê uma lida no artigo *Programação assíncrona*, da Microsoft (<https://docs.microsoft.com/pt-br/dotnet/csharp/async>), que fala de uma maneira mais detalhada sobre código assíncrono.

Voltemos ao nosso código. Com a implementação anterior realizada, vamos partir para a criação da visão. Clique com o botão direito do mouse sobre a pasta `Views` e adicione uma nova pasta, chamada `Departamento`. Então, na pasta criada, clique com o botão direito do mouse e em `Adicionar -> Exibição`.

Lembre-se de, antes de criar a visão, realizar um `build` na solução. Você notará que, na janela de criação, um novo dado é solicitado, o `Data context class`, que deverá ser nossa classe `IEntityContext`, conforme vemos na figura a seguir. Com exceção às traduções dos links exibidos ao usuário, nenhuma mudança será necessária. Inclusive, note que essa nova visão é praticamente idêntica à criada para `Instituicao`.

Execute sua aplicação e requisite a nova visão. Em minha máquina, a URL é <http://localhost:64867/Departamento/Index>.

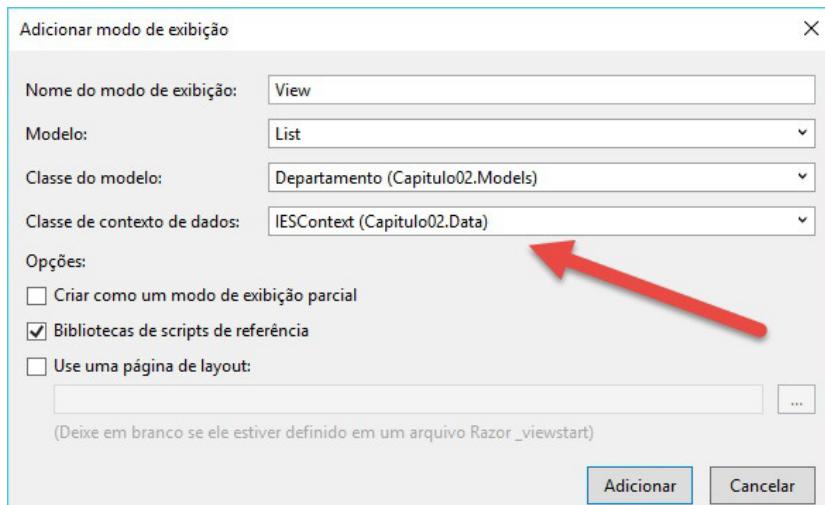
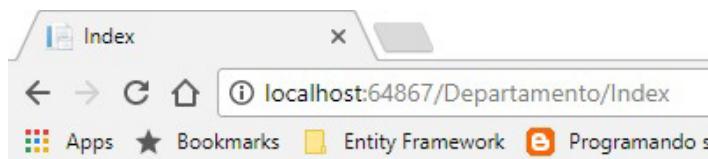


Figura 2.2: Janela de adição de uma visão

É possível que você perceba uma demora maior no processamento da requisição. Isso deve-se ao fato de o Entity Framework Core estar criando o banco de dados. Quando este processamento for concluído, você deverá visualizar uma página semelhante à da figura a seguir.



[Create New](#)

Nome

Ciência da Computação [Edit](#) | [Details](#) | [Delete](#)

Ciência de Alimentos [Edit](#) | [Details](#) | [Delete](#)

Figura 2.3: Visualizando os departamentos existentes

No Pesquisador de Objetos do SQL Server , você pode alterar os dados da tabela ou inserir novos. Para isso, clique com o botão direito do mouse no nome da tabela e, depois, em Exibir dados . Você verá um grid com os dados e uma linha em branco, ao final, para que novos registros sejam inseridos.

Não insira o ID, pois ele é automático. No entanto, lembre-se de que, se a base de dados for criada novamente, isso se perderá e ela retornará aos dados originais, inseridos por sua aplicação. Para o EF Core, por convenção, o valor de uma chave primária é gerado automaticamente, o que torna o campo autoincremento. Mas isso pode ser modificado por meio de atributos.

Inserção de novos dados

Se você fez todos os passos anteriores corretamente, já temos uma listagem de todos os departamentos registrados na base de dados. A inserção dos dados que aparecem foi realizada pelo inicializador do contexto, e não pela aplicação que estamos criando.

Implementaremos agora esta funcionalidade. Para a inserção de um novo registro, precisamos criar as duas actions Create : a que gera a visão (GET), e a que recebe os seus dados (POST).

A primeira action pode ser vista no código a seguir. Observe que ela é semelhante à que implementamos no capítulo anterior para Instituicao .

```
// GET: Departamento/Create
public IActionResult Create()
{
    return View();
}
```

Vamos agora para a action `POST`. Veja o código a seguir. Este adiciona o objeto da classe de departamento na base de dados. Ele usa um modelo conhecido como *binder*. Por meio dele, as propriedades relacionadas no método `Bind()`, que podem ser separadas por vírgulas, são recuperadas de um objeto chamado `Form Collection`, e então são atribuídas ao objeto recebido como argumento.

A propriedade `DepartamentoID` não está relacionada por ser a chave primária e ser criada automaticamente quando um novo objeto é persistido. De mesma maneira, ela não é solicitada na visão, como foi visto no capítulo anterior, ao trabalharmos com instituições.

Esta técnica busca evitar que propriedades (que não têm dados enviados pela visão) sejam atualizadas por ferramentas externas. Mais detalhes sobre este tema podem ser obtidos nos artigos *Preventing mass assignment or over posting in ASP.NET Core* (<https://andrewlock.net/preventing-mass-assignment-or-over-posting-in-asp-net-core/>) e *ASP.NET - Overposting/Mass Assignment Model Binding Security* (<https://www.hanselman.com/blog/ASPNETOverpostingMassAssignmentModelBindingSecurity.aspx>). Se tiver tempo, recomendo a leitura.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("Nome")] Departamento departamento)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(departamento);
```

```
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
}
catch (DbUpdateException)
{
    ModelState.AddModelError("", "Não foi possível inserir os
dados.");
}
return View(departamento);
}
```

Na listagem anterior, note que o modelo é verificado (`ModelState.IsValid`) antes de o objeto ser adicionado ao contexto (`_context.Add(departamento)`). Se nenhum erro de validação existir após a inserção, os dados são efetivamente gravados na base de dados (`await _context.SaveChangesAsync()`), novamente com código assíncrono.

Em caso de sucesso, a visão que será renderizada é a `Index`; em caso de falhas, a `Create` novamente, que exibirá os dados já informados pelo usuário. Note o uso de `nameof()` na chamada ao método `RedirectToAction()`. Isto minimiza o uso de literais no código, possibilitando erros de digitação em nomes de elementos codificados.

É possível realizar várias operações no contexto e, apenas ao final, realizar a atualização na base de dados. Isso garante que, se algo der errado em algum processo de atualização, nada é realizado no banco. Em caso de erro, uma exceção será disparada e exibida pelo Visual Studio, descrevendo-o.

Caso sua aplicação já esteja distribuída em um servidor, o erro capturado pela exceção deverá ser exibido em uma página de erro.

O tratamento de erros será visto mais à frente, no capítulo *Uploads, downloads e erros*.

Com isso posto, precisamos fazer a visão na qual o usuário informará os dados que devem ser inseridos na base de dados. Para isso, é necessário que criemos a visão `Create`. Você pode fazer isso seguindo exatamente os mesmos passos já vistos anteriormente.

Não comentarei nada sobre ela, pois é exatamente igual à visão `Create` que fizemos para `Instituicoes`. Teste sua aplicação, chamando a view `Create` (ou a `Index`), ao clicar no link para adicionar um novo departamento. Insira alguns departamentos para testar.

Alteração de dados já existentes

O método `action`, que gerará a visão para alteração dos dados com o EF Core, é um pouco diferente do que implementamos no capítulo *A primeira aplicação ASP.NET Core MVC*, para `Instituições`. Isso porque precisamos realizar alguns testes. Veja o código na sequência.

Na assinatura do método, note que ele possibilita o envio de um valor nulo (`long? id`). Desta maneira, é preciso verificar se não chegou um `id` ao método. Caso nada tenha sido enviado, caracteriza-se uma requisição inválida e, desta maneira, um erro é retornado ao cliente (navegador).

Em seguida, com um valor válido para o `id`, é realizada a busca de um departamento com esse valor em sua chave primária (o identificador do objeto). Após a busca, é preciso verificar se um

objeto foi encontrado; caso contrário, será retornado um outro tipo de erro.

Se existir um departamento para o `id` recebido pela action, o objeto que o representa será encaminhado para a visão. Veja que a busca pelo departamento utiliza um código assíncrono. Não comentarei mais sobre isso, pois este é o modelo para as aplicações ASP.NET Core MVC com Entity Framework Core.

```
// GET: Departamento/Edit/5
public async Task<IActionResult> Edit(long? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var departamento = await _context.Departamentos.SingleOrDefau
ltAsync(m => m.DepartamentoID == id);
    if (departamento == null)
    {
        return NotFound();
    }
    return View(departamento);
}
```

NotFound()

O método `NotFound()`, que pertence ao controlador, retorna uma instância da classe `HttpNotFoundResult`. Este erro é equivalente ao status HTTP 404.

Agora, é preciso criar a visão para a alteração, mas isto é com você. Ela é idêntica à que fizemos para a instituição. Para finalizarmos, precisamos implementar a action que receberá os dados da visão. Veja no código a seguir esta implementação.

O código começa verificando se os valores recebidos para o `id` e a propriedade `DepartamentoID` do objeto `departamento` são iguais. Caso negativo, um erro é disparado. Após isso, verifica-se se o modelo é válido, ou seja, se não há nenhuma validação de erro – por exemplo, um valor requerido não preenchido.

O EF Core automaticamente cria um flag para as propriedades alteradas no objeto recebido, e mapeia essas alterações para atualizar na base de dados na chamada ao método `_context.Update()`. Com o modelo validado, é preciso dizer que o objeto recebido sofreu uma alteração desde sua recuperação (`context.Entry(fabricante).State = EntityState.Modified`). Essa atualização será persistida pela chamada ao método `SaveChangesAsync()`.

Caso tudo ocorra bem, a aplicação é redirecionada à action `Index`. Caso contrário, o objeto recebido é retornado à visão que requisitou a ação atual, a `Edit`. Veja ainda que existe um catch para `DbUpdateConcurrencyException`, disparado caso ocorra um problema de concorrência no acesso ao departamento em questão.

Para ilustrar problemas de concorrência, vamos a um exemplo trivial: imagine um usuário com desejo de reservar uma passagem de um determinado voo. No momento em que ele acessa o site de reserva, existe um único assento disponível. Ocorre, que neste mesmo momento, outros usuários estão visualizando a mesma situação, e todos querem o último assento. Há então uma concorrência.

Todos confirmam o desejo pelo assento, mas há apenas uma vaga. Garantir que apenas essa vaga seja vendida é um problema

relacionado à concorrência. Em nosso exemplo, caso o EF retorne uma exceção desse tipo, é neste catch que ela deverá ser trabalhada. Dentro do catch , existe a invocação ao método DepartamentoExists() , que está implementado também no código a seguir. Esse método simplesmente verifica se há, na base de dados, algum objeto com o ID do objeto recebido.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(long? id, [Bind("Departamen
toID, Nome")] Departamento departamento)
{
    if (id != departamento.DepartamentoID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(departamento);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!DepartamentoExists(departamento.DepartamentoID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(departamento);
}

private bool DepartamentoExists(long? id)
{
```

```
        return _context.Departamentos.Any(e => e.DepartamentoID == id
);
}
```

Teste sua aplicação. Você pode requisitar diretamente no navegador um determinado `id` ou, por meio da visão `Index`, clicar no link responsável pela edição e alteração dos dados de um determinado objeto. Se você optar por digitar a URL no navegador, tenha como exemplo a minha, que foi <http://localhost:64867/Departamento/Edit/3>.

Visualização dos detalhes de um objeto

A visualização dos dados de um determinado departamento é possível quando optamos por editá-los. Porém, pode acontecer que queiramos apenas ver, e não editar. Para isso, existe a visão `Details`. Esta implementação segue os mesmos princípios da que fizemos para instituições, no capítulo *A primeira aplicação ASP.NET Core MVC*. Vamos implementá-la agora.

Lembre-se de que, para a action `Details`, precisamos apenas da action que atende à requisição `GET` do HTTP, pois o usuário não modificará os dados na visão. Na sequência, veja o código que representa essa action. Observe que a implementação é exatamente igual à que fizemos para a action `Edit`, que gera a visão.

Mais adiante, trabalharemos a arquitetura e reduziremos a redundância de códigos. Crie agora a visão para a action e teste sua aplicação. As orientações para o teste são as mesmas fornecidas para o teste da action `Edit`.

```
public async Task<IActionResult> Details(long? id)
{
    if (id == null)
```

```

    {
        return NotFound();
    }

    var departamento = await _context.Departamentos.SingleOrDefaultAsync(m => m.DepartamentoID == id);
    if (departamento == null)
    {
        return NotFound();
    }

    return View(departamento);
}

```

Remoção de um departamento

Para a conclusão do CRUD para Departamentos , resta-nos apenas implementar a funcionalidade de remoção de um registro/objeto. Para isso, recorde do que fizemos no capítulo anterior para Instituicoes . Precisamos implementar uma action que gerará a visão (GET) e uma que receberá a confirmação da exclusão do objeto que está em exibição para o usuário.

Vamos começar com a action que gerará a visão. Veja sua listagem a seguir. Observe, uma vez mais, a redundância de código comum. Novamente temos as mesmas instruções utilizadas para o Edit e o Details . Veremos algumas simplificações e mais sobre reúso a partir do capítulo *Separação da camada de negócio*.

```

// GET: Departamento/Delete/5
public async Task<IActionResult> Delete(long? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var departamento = await _context.Departamentos.SingleOrDefaultAsync(m => m.DepartamentoID == id);

```

```

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

    return View(departamento);
}

```

Agora, é preciso criar a visão da mesma maneira que fazemos desde o primeiro capítulo. Com ela criada, precisamos criar a segunda action para a remoção de departamentos. Esta terá o mesmo nome que a anterior, entretanto, será capturada por uma requisição HTTP POST . Veja o código na sequência.

A única instrução que ainda não foi trabalhada neste código é a responsável por remover o objeto recuperado da coleção de objetos, no contexto

`_context.Departamentos.Remove(departamento)` . Note que, após remover o objeto, o método `SaveChangesAsync()` é invocado.

Você observou o nome que foi dado ao método da action? Foi `DeleteConfirmed` . Entretanto, antes da assinatura do método, existe um atributo, o `ActionName` , que nomeará esse método para ser visto como `Delete` .

```

// POST: Departamento/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(long? id)
{
    var departamento = await _context.Departamentos.SingleOrDefaultAsync(m => m.DepartamentoID == id);
    _context.Departamentos.Remove(departamento);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}

```

2.4 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Mais um capítulo concluído, parabéns! Estamos apenas começando, *hein?* Acabamos de implementar um CRUD com acesso à base de dados, usando o Entity Framework Core. Foi possível ver que as mudanças nas actions foram específicas para o acesso a dados, e que as visões praticamente não sofreram alterações.

Agora, o que acha de adaptar todo o trabalho que fizemos no capítulo anterior, com `Instituições`, para acessar a base de dados, tal qual fizemos neste capítulo? Já temos dois domínios sendo atendidos (`Instituicao` e `Departamento`) e usando o EF Core para persistir uma base de dados.

Apenas relembrando: você pode obter este e os demais capítulos em <https://github.com/evertonfoz/asp-net-core-mvc-casa-do-codigo>. Assim, você pode comparar a sua implementação com a que fiz.

Na sequência, trabalharemos o uso de layouts, que podem servir e atender diversas visões. Apresentarei também o Bootstrap e um pouco do jQuery.

CAPÍTULO 3

LAYOUTS, BOOTSTRAP E JQUERY DATATABLE

Aplicações web são aplicações complexas, até certo ponto. Normalmente, elas são desenvolvidas em camadas, e pode ocorrer que cada camada seja desenvolvida por um profissional diferente, ou por um profissional que desempenhe papéis diferentes de acordo com a camada que está implementando.

A camada que "resolve" o problema - como já vimos brevemente nos capítulos anteriores - está no servidor, e essa resolução está implementada por classes em C# (em nosso caso). Mas não é essa camada que seu usuário visualizará e, em alguns casos, nem saberá da sua existência. Então, o que o usuário vê e em que se interessa inicialmente?

A resposta para essa pergunta é: a camada de apresentação – a interface com o usuário, ou seja, a página (ou conjunto de páginas) disponibilizada para ser vista no navegador. Desenvolver essa camada requer, em meu ponto de vista, um dom e não apenas conhecimento técnico, pois trabalhamos com imagens, cores, tipos de fontes (letras) e diversas configurações para cada tipo de recurso a ser manipulado.

Reforço que, em meu ponto de vista, o Web Designer precisa ser um artista. Como não sou designer, e muito menos artista, recorro ao Bootstrap e seus templates que podem ser baixados na web, alguns inclusive de maneira gratuita. E é sobre isso que este capítulo trata.

Até aqui, nossa aplicação possui dois controladores para os dois domínios que já trabalhamos, `Instituicao` e `Departamento`. Neste capítulo, não trago nada de novo em relação ao modelo de negócio. O foco é a camada de apresentação.

Vamos verificar um pouco o que o Bootstrap pode fazer por nós, adaptando as visões de `Instituicao`. Criamos essas visões no primeiro capítulo e, no segundo, lembro de ter pedido a você para adaptá-las para acessar a base de dados, por meio do EF Core.

3.1 O BOOTSTRAP

Ao acessar o site do Bootstrap (<http://getbootstrap.com/>), deparamo-nos com uma definição: "*O Bootstrap é o conjunto de ferramentas de código aberto para desenvolvimento com HTML, CSS e JS. Permite criar rapidamente protótipos de ideias ou construção de uma aplicação completa, sistema de grid (grade) responsivo e um grande conjunto de plugins desenvolvidos com jQuery*". E ele é gratuito! Mas, o que seria o "design responsivo"?

Vamos falar sobre o problema que esta técnica aborda. Atualmente, temos vários dispositivos que permitem acesso a web. Temos computadores desktop, notebooks, tablets, celulares e outros que certamente surgirão.

Para cada um deles, existe uma série de especificações técnicas

para a tela que apresentará a interface de sua aplicação ao usuário. E, para nosso desespero total, existe ainda uma série de navegadores.

Desta maneira, como fazer para que seu website possa "responder" de maneira adequada às requisições de diversos dispositivos e navegadores, de uma maneira que sua aplicação seja sempre visível de forma organizada em seu layout? Inclusive, não esqueça que o ideal é você não precisar se preocupar com estas especificações.

Uma resposta para esta inquietação está no uso do Bootstrap. Ele trata tudo isso para nós.

Atualização dos controles de dependência com o Bower

Em relação ao Bootstrap, quando criamos nossa solução e projeto ASP.NET Core MVC, ele já veio instalado. Entretanto, no momento de escrita deste livro, existe uma versão beta, que já vem sendo comentada e utilizada, então, será ela que usaremos. Na figura a seguir, veja que o Gerenciador de Soluções apresenta tanto o Bootstrap como o jQuery e as extensões que usaremos, nas dependências do Bower.

O Bower é um projeto que possibilita o gerenciamento de pacotes conhecidos como *client-side*, que são utilizados pela parte cliente de uma aplicação. Em vez de acessar o site de cada projeto necessário para sua aplicação, baixar e disponibilizar os arquivos de maneira manual, ele permite a automatização destes downloads diretamente no seu projeto. Você pode até especificar uma determinada versão do projeto desejado.

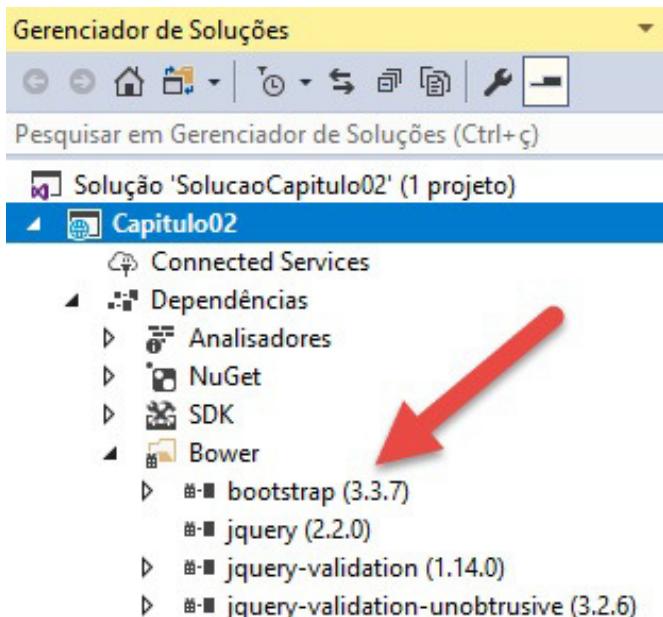


Figura 3.1: Verificação do Bootstrap e jQuery disponíveis nas dependências da aplicação

Precisamos atualizar a versão desses controles para as mais recentes. Clique com o botão direito na pasta `Bower` e, em seguida, em `Gerenciar Pacotes do Bower`. Veja a figura a seguir:

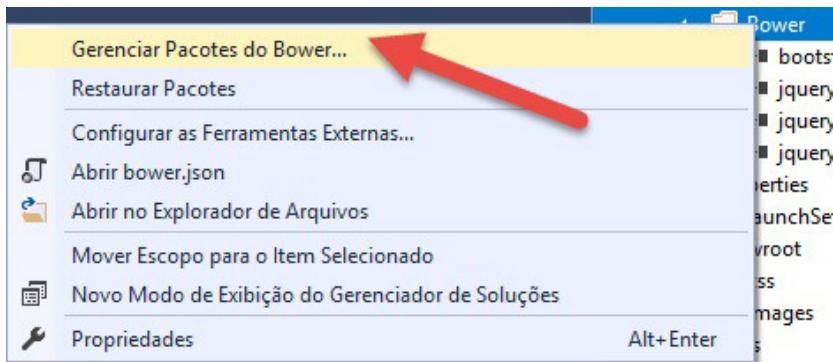


Figura 3.2: Acesso ao gerenciador de pacotes do Bower

Uma janela será exibida no Visual Studio, precisamos clicar na opção Atualização Disponível , que trará a relação do que precisamos atualizar. É para aparecer os três componentes apresentados na figura a seguir: Bootstrap , jQuery e jQuery Validation . Vamos atualizar um de cada vez; basta selecionar o componente, e depois o botão de atualizar.

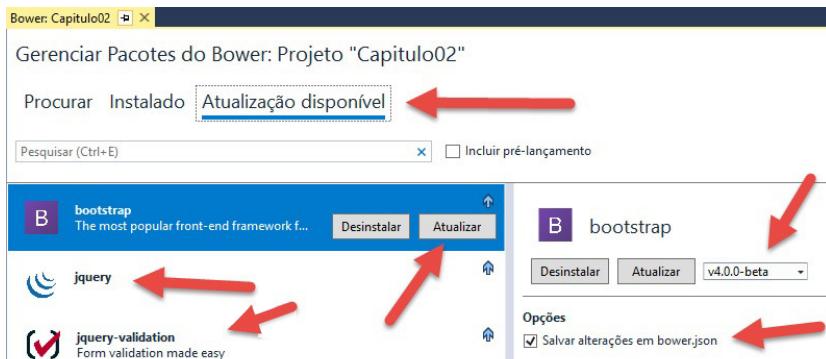


Figura 3.3: Janela para atualização dos componentes do Bower

Após a atualização, sua pasta de dependência do Bower deve

estar semelhante à apresentada na figura a seguir. Mas lembre-se de que essas versões podem evoluir, de acordo com o período em que você adquiriu este livro. Caso isso ocorra, os exemplos trabalhados no livro devem funcionar também de maneira correta, sem problemas.

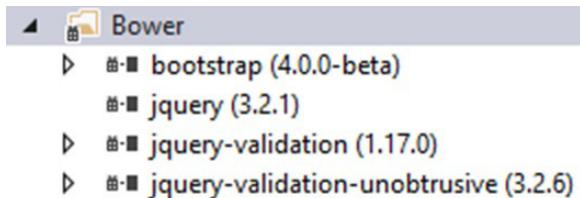


Figura 3.4: Controles do Bower atualizados

Se você estiver em uma rede que possua um proxy para acesso à internet, pode ser que tenha problemas durante o processo de atualização. Isso ocorre por conta de configurações realizadas na rede pela equipe de infraestrutura, proibindo o acesso a determinados serviços, sem uso do proxy.

Para contornar isso, na pasta de seu usuário, no gerenciador de arquivos (Windows Explorer), crie um arquivo chamado `.bowerrc`. No conteúdo dele, digite o que está na sequência, mas alterando para seus valores de servidor de proxy – em meu caso, a pasta foi `C:\Usuários\Everton`.

```
{  
  "registry": "http://bower.herokuapp.com",  
  "proxy": "http://proxyuser:proxypwd@proxy.server.com:8080",  
  "https-proxy": "http://proxyuser:proxypwd@proxy.server.com:8080"  
}
```

De volta ao Bootstrap

O Bootstrap fornece um grande conjunto de elementos HTML, configurações CSS e um excelente sistema de grades para ajudar no design de páginas web. Com o advento dos dispositivos móveis, surgiu o termo *Mobile-first*, que em suma significa: "*Pense em dispositivos móveis antes dos desktops*".

Esta técnica, combinada aos recursos do Bootstrap, habilita os desenvolvedores a construírem interfaces web intuitivas de uma maneira mais simples e rápida. Trabalharemos os recursos do Bootstrap conforme formos utilizando-os.

3.2 LAYOUTS COM O BOOTSTRAP

Quando desenvolvemos uma aplicação web, é muito comum que diversas páginas de sua aplicação possuam áreas em comum, como:

1. Um cabeçalho com a logomarca da empresa, área de busca e informações sobre usuário, dentre outras;
2. Uma área de rodapé, com informações de contato, por exemplo;
3. Uma área com links para páginas específicas do site;
4. Uma área chamada como "área de conteúdo", local onde as informações específicas de cada página são exibidas.

Não é produtivo (nem racional) implementarmos essas áreas comuns em cada página a ser desenvolvida. Para isso, fazemos uso de layouts. Sendo assim, vamos para nossa primeira implementação deste capítulo, que será a criação de um layout, usando o Bootstrap. Legal, não é? Vamos lá.

O padrão ASP.NET Core MVC diz que os arquivos de layout

(isso mesmo, pode haver mais de um) devem ficar dentro de uma pasta chamada `Shared`, dentro da pasta `Views`. Esta já existe e traz um layout pré-definido. Caso se lembre, ele foi utilizado nas visões do controlador `Home`.

Entretanto, criaremos um layout específico para nossos exemplos. Clique com o botão direito do mouse na pasta `Shared` (dentro de `Views`) e, então, na opção `Adicionar -> Novo Item`. Na categoria `Web`, busque por `Página de Layout da Visualização MVC`. Nomeie este layout de `_LayoutIES.cshtml`. Isso mesmo, começando com um sublinhado, para que a visão não seja requisitada.

No padrão ASP.NET Core MVC, uma visão com um nome iniciado com sublinhado não poderá ser requisitada diretamente pelo navegador. O código partionado, que veremos a seguir, representa a implementação para o arquivo `_LayoutIES.cshtml`.

Início do layout

No código, é definida a `viewport` na primeira instrução, dentro da tag `<head>`. A `viewport` é a área visível para o usuário de uma página web. Ela varia de acordo com o dispositivo usado, podendo ser pequena para celulares e grandes para desktops.

Em `content`, o valor `width=device-width` define como tamanho da `viewport` o tamanho da tela do dispositivo; o valor `initial-scale=1.0` define o valor inicial para zoom; e `shrink-to-fit=no` impede que o conteúdo configurado e exibido em uma área fora da janela visível tenha seu tamanho reduzido para ser apresentado no dispositivo em execução.

A instrução `@ViewBag.Title` representa o título que aparece na barra de títulos do navegador e que pode ser definido em cada visão que usar o layout. Terminando esta parte do código, na tag `<head>` ainda, temos a inclusão do arquivo de CSS do Bootstrap. Veja o código. Aqui temos conteúdo novo e é preciso explicá-lo.

```
<!DOCTYPE html>

<html lang="pt-br">
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
    <title>@ViewBag.Title</title>
    <environment include="Development">
        <link rel="stylesheet" href("~/lib/bootstrap/dist/css/bootstrap.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css"
              asp-fallback-href "~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
    </environment>
    @RenderSection("Styles", required: false)
</head>
```

O ASP.NET Core MVC possibilita que implementemos nossas aplicações para serem executadas em diversos ambientes (*environments*), tais como Desenvolvimento (Development) , Pré-produção (Staging) e Produção (Production) . Desta maneira, em nosso código anterior, na primeira tag `<environment>` , seus elementos serão usados sempre que o ambiente for Development , por isso o atributo `include` . Note que o caminho para o CSS é local.

Já na segunda tag `<environment>`, quando a aplicação estiver executando em um ambiente que não seja `Development`, o CSS vem de um CDN (*Content Delivery/Distribution Network*, ou Serviço de Entrega em Rede). Por isso temos o atributo `exclude`.

A adoção de um CDN para uso de CSS e JavaScripts é uma prática comum que visa melhor disponibilidade de recursos caso sua aplicação precise. É claro que você pode pensar que é possível o servidor CDN ficar fora de serviço em algum momento, e isso não pode causar uma pausa ou um problema em sua aplicação, mas é por isso que a tag `<link>` faz uso da Tag Helper `asp-fallback-href`. Esta aponta onde, em seu servidor, o recurso estará disponibilizado.

O código anterior termina com a instrução `@RenderSection()`, apontando que as seções chamadas `Styles`, nas páginas que usarem esse layout, deverão ser renderizadas neste ponto (quando existirem).

Definição do layout

Com a chegada do HTML 5, algumas tags semânticas começaram a ser usadas, e uma delas é a `<nav>`. Ela possibilita a criação de uma lista de elementos que servem de ligação para outros recursos.

O Bootstrap permite, além de formatações, a configuração de diversas modalidades para essa tag ao usar classes específicas. A primeira tag do código a seguir define a barra de navegação como fixa no topo da página, por meio da classe `fixed-top`. As classes `navbar-light` e `bg-primary` configuram as cores dos textos e da barra de navegação.

A segunda tag `<nav>` é responsável pela criação de uma barra de navegação na base da página. O objetivo aqui é apenas verificar os recursos do Bootstrap para a tag. Se você quiser uma referência completa sobre os possíveis usos para ela tag, acesse: <http://getbootstrap.com/docs/4.0/components/navs/>.

Entre as tags `<nav>`, está o container para o layout de nossas páginas e o local onde será exibido o conteúdo das páginas que farão uso desse layout. Veja que sempre é utilizada a tag `<div>`, configurada com classes do Bootstrap.

A classe `container-fluid` define uma área na qual podemos inserir e organizar conteúdos que o Bootstrap precisa para fazer uso de seu sistema de grades e organizar elementos nele. Essa classe usa toda a área disponibilizada como *view-port*. Existe outra, o `container`, que acaba deixando espaços entre as margens. Caso queira, troque e teste de maneira alternada.

Dentro do `container`, existem duas linhas, cada uma definida com um `<div>` que utiliza a classe `row`. O primeiro define quatro outros `<div>`, cada um com a classe `col-12`, que é a quantidade máxima de células no sistema de grade. Em cada um desses elementos, é renderizado um espaço em branco entre a barra de navegação superior e o conteúdo a ser exibido pelas páginas que usam o layout, por meio da instrução .

A segunda linha define duas colunas, uma com opções de acesso a conteúdos (algo como um menu) e outra com a diretiva `@RenderBody()`, dizendo que é neste local que deve ser renderizado o conteúdo da página que utiliza o layout.

Nesta segunda linha, veja o uso das classes `col-2` e `col-10`,

definindo os tamanhos das colunas. Note também que fiz dois usos diferentes para a tag `<nav>` neste exemplo: um para menu, que é a ideia desta tag do HTML 5; e outro para um efeito visual, no topo e na base da página.

Para um aprofundamento maior sobre `@RenderBody()`, acesse <http://www.codeproject.com/Articles/383145/RenderBody-RenderPage-and-RenderSection-methods-in>.

```
<body>
    <nav class="navbar fixed-top navbar-light bg-primary">
        <a class="navbar-brand" href="#">
            <img src "~/images/casa-do-codigo_2x.png" width="30"
height="30" alt="">
                ASP.NET Core MVC
        </a>
    </nav>

    <div class="container-fluid">
        <div class="row">
            <div class="col-12">&ampnbsp</div>
            <div class="col-12">&ampnbsp</div>
            <div class="col-12">&ampnbsp</div>
            <div class="col-12">&ampnbsp</div>
        </div>

        <div class="row">
            <div class="col-2">
                <nav class="nav flex-column bg-transparent">
                    <a class="nav-link active" href="#">
                        <img src "~/images/university.png" width=
"30" height="30" alt="">
                            Instituições
                    </a>
                    <a class="nav-link" href="#">
                        <img src "~/images/department.png" width=
"30" height="30" alt="">
                            Departamentos
                    </a>
                </nav>
            </div>
            <div class="col-10">
```

```
<div class="row">
    <div class="col-12">
        @RenderBody()
    </div>
</div>
</div>

<nav class="navbar" style="background-color: #e3f2fd;">
    <a class="navbar-brand" href="#">Everton Coimbra de Araújo
</a>
</nav>
```

Término do layout

Para fechar a implementação do código da visão que representará o layout de nossas páginas, teremos novamente duas tags `<environment>` , agora com a adição dos scripts para o jQuery e o Bootstrap. A ideia é a mesma apresentada anteriormente.

Ao final do código, antes do fechamento da tag `<body>` , existe uma instrução razor `@RenderSection()` . Esta renderiza os scripts usados na visão que utiliza o layout, caso eles existam.

Nas tags `<script>` , existem dois atributos, `crossorigin` e `integrity` . Estes referem-se à Subresource Integrity , uma característica de segurança que instrui os navegadores a verificar que os recursos enviados – por parte de terceiros (como CDNs) – estão distribuídos sem manipulação.

O atributo `Integrity` permite ao navegador checar o arquivo-fonte para garantir que o código não seja carregado caso tenha sido manipulado. O atributo `crossorigin` é utilizado quando uma requisição é carregada com `CORS` , o qual é agora um

requisito da checagem SRI, quando não é carregado da mesma origem.

O código termina com a renderização da seção `ScriptPage`. Logo aplicaremos esse conceito de seção nas visões.

Entre as tags `<environment>`, inseri um script para o componente `Popper`, que é um pré-requisito para o Bootstrap. Preferi seguir a orientação da documentação e utilizar o CDN diretamente, evitando assim ter de baixar e instalá-lo.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/popper.min.js" integrity="sha384-b/U6ypiBEHpOf/4+1nzFpr5rXe4B1qQ2ogWcLJY/5TcvuI/Wsp6FkSSZB+
```

```
<environment include="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
</environment>

<environment exclude="Development">
    <script src="https://code.jquery.com/jquery-3.2.1.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCKRr/rE9PQpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN">
    </script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/bootstrap.min.js"
        asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
        crossorigin="anonymous"
        integrity="sha384-h0AbiXch4ZDo7tp9hKZ4TsHbi047NrKGLO3SEJAg45jXxnGIfYzk4Si90RDIqNm1">
    </script>
```

```
</environment>

@RenderSection("ScriptPage", required: false)
</body>
</html>
```

CORS E SRI

CORS (*Cross-origin Resource Sharing*) é um padrão para acessar recursos web em domínios diferentes. Ele permite que scripts web interajam de maneira mais aberta com conteúdos externos ao domínio original, melhorando a integração de *web services*.

Já o SRI (*Subresource Integrity*) é uma característica de segurança que habilita navegadores a verificar quais arquivos recuperados (de um CDN, por exemplo) são entregues sem manipulações inesperadas. Ele funciona por meio de uma permissão de fornecimento de um hash criptografado, sendo necessário o arquivo desejado ter um igual.

Já temos o layout definido, então, vamos utilizá-lo.

3.3 PRIMEIRA PÁGINA COM O LAYOUT CRIADO

Vamos criar nossa primeira visão que fará uso do layout criado anteriormente. Elimine a visão `Index` da pasta `\Views\Home` e crie-a novamente. Porém, na janela que pré-configura a visão, marque a opção de utilizar um layout, e depois clique no botão que

apresenta reticências (. . .) para abrir a janela de localização e seleção da visão de layout – semelhante à apresentada na figura a seguir.

Escolha seu layout e clique no botão `OK`. Ao retornar para a janela de criação da visão, configure: o modelo da visão para `List`, que é o modelo para listar o conteúdo de alguma fonte de dados; a classe de modelo para `Instituicao`, que representa os objetos que pretendemos listar; e o `IESContext` como contexto, que tem o mapeamento de nosso modelo de objetos para o relacional.

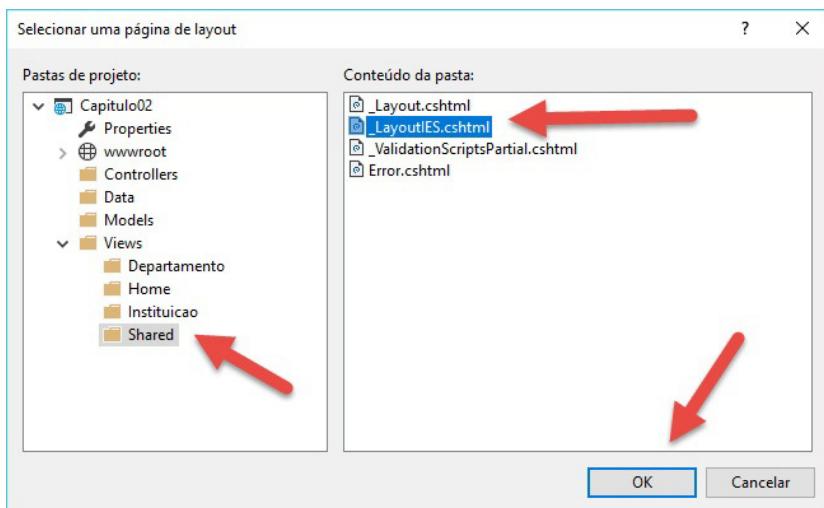


Figura 3.5: Selecionando uma visão de layout para a visão a ser criada

Após a criação da visão, execute sua aplicação e requisite a visão `Index` do controlador `Home`. Veja o código criado na listagem a seguir. A definição do layout está no início. Também alterei o título para a página. A visão em si é o que está dentro do container `Bootstrap`.

O Bootstrap oferece um componente que possibilita o efeito de rolagem lateral de imagens, conhecido como `Carousel`, e é ele que usaremos agora. No terceiro `<div>` do código, veja o uso das classes `carousel` e `slide`. Note também o atributo `data-ride`. São estes valores que configuram o elemento HTML para se comportar como um `Carousel`.

Nas tags ``, aponto para figuras que baixei para a pasta `images`, dentro de `wwwroot`. Fica a seu critério escolher qualquer figura. O `Carousel` é um componente que faz analogia a uma apresentação de slides: ele exibirá uma imagem após a outra e, quando chegar ao final, voltará a exibir a primeira.

```
@{  
    ViewData["Title"] = "Casa do Código - ASP.NET Core MVC";  
    Layout = "~/Views/Shared/_LayoutIES.cshtml";  
}  
  
<div class="container">  
    <div class="row justify-content-center align-items-center">  
        <div id="carouselExampleSlidesOnly" class="carousel slide  
        data-ride="carousel">  
            <div class="carousel-inner" role="listbox">  
                <div class="carousel-item active">  
                      
                </div>  
                <div class="carousel-item">  
                      
                </div>  
                <div class="carousel-item">  
                      
                </div>  
            </div>  
        </div>  
    </div>  
</div>
```

Execute sua aplicação. Se você manteve a rota inicialmente criada pelo Visual Studio, a nova visão será exibida por padrão. Veja-a na figura a seguir. Note a parte comum, definida no layout. Experimente mudar o tamanho da janela do navegador para ver o comportamento da página.

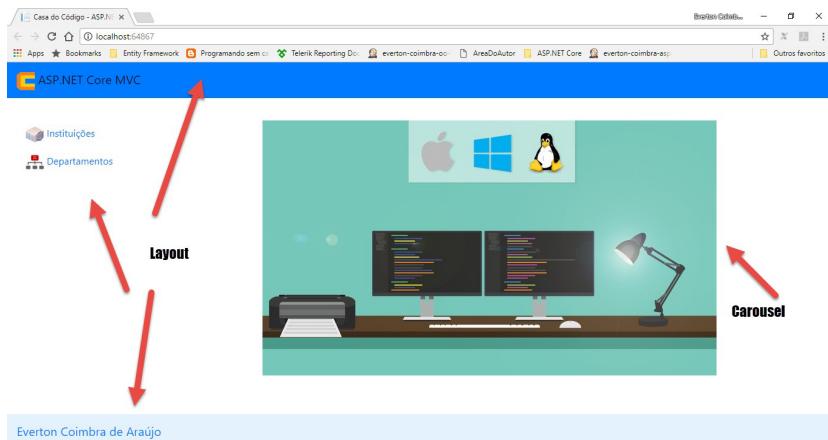


Figura 3.6: Página inicial da aplicação, usando o layout criado

3.4 ADAPTANDO AS VISÕES PARA O USO DO BOOTSTRAP

Nesta seção, focaremos nas visões renderizadas pelo controlador `Instituicao`. Para começar, a primeira visão que adaptaremos é a `Index`. Nela, é apresentada uma relação de instituições registradas. Para essa implementação, além de usarmos o Bootstrap, também utilizaremos um novo componente, o jQuery DataTables (<https://www.datatables.net/>).

O DataTables é um plugin para o jQuery, uma ferramenta que traz enormes ganhos para listagens em forma de tabelas. Para

disponibilizá-lo, faremos uso do Bower. Desta maneira, tal qual fizemos para atualizar os pacotes do Bower, vamos instalar um novo pacote.

Em **Dependências**, clique com o botão direito do mouse sobre **Dependências/Bower** e escolha **Gerenciar Pacotes Bower**. Na janela aberta, clique em **Procurar** e digite **jQuery DataTables**. Escolha a versão 1.10.15 (utilizada neste livro) e clique no botão para instalar. Veja a figura a seguir.

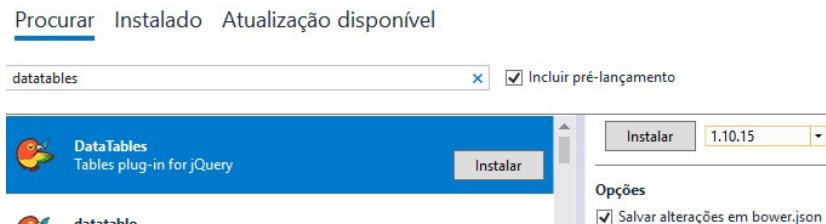


Figura 3.7: Instalando o DataTables no projeto

A visão Index de Instituições

Após a instalação do DataTables, é criada uma pasta chamada `datatables` dentro de `lib`, contendo os recursos necessários para a nossa implementação. Você verá que é bem simples. Na sequência, vamos à implementação da visão `Index` de `Instituição`, que implementa o `DataTables`.

Note no início e no final da listagem a declaração das seções (`@section`) `styles` e `ScriptPage`, para conseguirmos inserir arquivos de CSS e JavaScript do DataTables na visão e utilizar o componente. Veja a definição do layout logo no início do código.

Todo elemento HTML foi tirado, deixando exclusivamente o

que pertence à visão que será renderizada. Nas `sections`, usamos diretamente a referência aos recursos. Também foram utilizados os recursos visuais e deixado os de CDN para você adaptar como fizemos no layout. A ideia é também apresentar essa técnica.

O primeiro elemento HTML do código a seguir é um `<div>` que define um `card` do Bootstrap. A ideia desse componente é organizar o conteúdo em um tipo de agrupamento, que possa ser configurado visualmente de maneira independente.

A tabela representada pela tag `<table>` tem a definição de seu `id`. É por esse identificador que configuraremos o DataTables. Ao final do código, na seção de scripts, há um código jQuery que busca o controle por seu `id` e, então, configura o DataTables.

```
@model IEnumerable<Capitulo02.Models.Instituicao>
{@
    Layout = "_LayoutIES";
}
@section styles {
    <link rel="stylesheet" href("~/lib/datatables/media/css/jquery.dataTables.min.css" />
}

<div class="card-block">
    <div class="card-header text-white bg-primary text-center h1">
        Instituições Registradas</div>
    <div class="card-body">
        <table id="tabela_instituicoes">
            <thead>
                <tr>
                    @Html.DisplayNameFor(model => model.InstituicaoID)
                    </th>
                    <th>
                        @Html.DisplayNameFor(model => model.Nome)
                    </th>
                    <th>
```

```

        @Html.DisplayNameFor(model => model.Ender
eco)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.In
stituicaoID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.No
me)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.En
dereco)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@i
tem.InstituicaoID">Edit</a> |
                    <a asp-action="Details" asp-route-id=
"@item.InstituicaoID">Details</a> |
                    <a asp-action="Delete" asp-route-id="
@item.InstituicaoID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
</div>
<div class="card-footer bg-success text-center">
    <a asp-action="Create" class="btn-success">Criar uma nova
instituição</a>
</div>
</div>

@section ScriptPage {
    <script type="text/javascript" src="~/lib/datatables/media/js
/jquery.dataTables.min.js"></script>

```

```
<script type="text/javascript">
    $(document).ready(function () {
        $('#tabela_instituicoes').DataTable();
    });
</script>
}
```

Para que o jQuery DataTables funcione em uma tabela HTML (`<table>`), é preciso que ela tenha o cabeçalho de suas colunas definido por um `<thead>`, e um `<th>` para cada coluna. O conteúdo da tabela também precisa estar envolvido por um `<tbody>`, tal qual demonstra o código anterior.

No código, você poderá visualizar dois blocos Razor, que definem duas seções: `@section styles` e `@section ScriptPage`. Em `styles`, definimos quais arquivos de CSS devem ser incluídos na visão (além dos definidos no layout); e em `ScriptPage`, definimos quais arquivos JavaScript devem ser inseridos também, além dos definidos no layout.

Mas como esses arquivos serão inseridos na página renderizada, que é a união do layout com a visão? Para a seção `Style`, antes de fechar a tag `<head>`, insira a instrução `@RenderSection("styles", false)`. Já para o JS, antes de fechar a tag `<body>`, insira a instrução `@RenderSection("ScriptPage", false);`. Já temos isso implementado em nosso layout, mas achei importante reforçar aqui.

E como fazemos com que a tabela HTML seja configurada para exibir os recursos do jQuery DataTables? Isso é feito com apenas uma instrução, como pode ser verificado ao final da listagem, a chamada à `$('.table').DataTable();`.

O símbolo `$` denota uma instrução jQuery. Com o argumento `('#tabela_instituicoes')`, selecionamos os elementos HTML que possuem o `id tabela_instituicoes`. Com o elemento recuperado, invocamos o método `DataTable()`. Esta única instrução está envolvida pelo bloco `$(document).ready(function () {})`. O bloco é o que está entre as chaves `({})`. Essa instrução significa que, assim que o documento (`document`) estiver pronto (`ready`), a função anônima (`function() {}`) será executada.

Com isso, chegamos ao final da primeira implementação com Bootstrap, layouts e DataTable. A figura a seguir apresenta a página gerada para a visão `Index` de `Instituições`. Em destaque, estão os elementos oferecidos pelo DataTable em conjunto com o `card` do Bootstrap.

No início da tabela gerada, existe a configuração de quantos elementos por página se deseja exibir, sendo o padrão 10. Ao lado, existe um controle para pesquisar o conteúdo renderizado na tabela. Ao lado do título de cada coluna, há um botão para classificação dos dados com base na coluna desejada. Ao final, ao lado esquerdo, está a informação de quantos elementos estão sendo exibidos e o seu total. Por fim, à direita, vemos uma opção de navegação entre páginas. Tudo isso pronto.

Figura 3.8: A visão Index de Instituições em conjunto com o layout, Bootstrap e DataTables

Com a adição do DataTables, ele traz um comportamento padrão para os dados, que é a classificação default com base na primeira coluna. No capítulo anterior, nosso exemplo não apresentava o ID da Instituição. Neste capítulo, nós o inserimos como primeira coluna, sendo que o DataTables define que a classificação padrão se dá pela primeira coluna.

Desta maneira, os dados não são classificados por ordem alfabética com base no nome, tal qual gerado na action. Para corrigir isso, precisamos enviar ao método `DataTable()` a coluna pela qual a listagem será classificada, de acordo com o código seguinte.

O valor `1` refere-se à segunda coluna, e o `"asc"` significa ascendente/crescente . Para ordem decrescente, use `desc` .

```
$('#tabela_instituicoes').DataTable({
    "order": [[1, "asc"]]
});
```

A visão Create

Espero que você tenha gostado da formatação da visão `Index`. Você pode melhorar e mudar as cores e a formatação com as diversas classes de estilo que o Bootstrap oferece e que são fáceis de aprender.

Agora, vamos adaptar nosso formulário de registro de uma nova instituição para fazer uso do Bootstrap. O código a seguir traz a nova implementação para a visão `Create`.

```
@model Capitulo02.Models.Instituicao
 @{
    Layout = "_LayoutIES";
}

<div class="card-block">
    <div class="card-header text-white bg-danger text-center h1">
        Registrando uma nova instituição</div>
    <div class="card-body">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Nome" class="control-label"></label>
                <input asp-for="Nome" class="form-control" />
                <span asp-validation-for="Nome" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Endereco" class="control-label"></label>
                <input asp-for="Endereco" class="form-control" />
                <span asp-validation-for="Endereco" class="text-danger"></span>
            </div>
            <div class="form-group text-center h3">
                <input type="submit" value="Registrar Instituição" class="btn btn-light" />
                <a asp-action="Index" class="btn btn-info">Retorn
```

```
ar à listagem de instituições</a>
        </div>
    </form>
</div>
<div class="card-footer bg-dark text-center text-white">
    Informe os dados acima e/ou clique em um dos botões de ação
</div>
</div>
@section ScriptPage {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
```

Se você notar o código da visão `Create`, algumas tags HTML já continham classes do Bootstrap, como: a classe `form-group` (`<div>`), que registra em seu interior um grupo de elementos do formulário; e `control-label` e `form-control`, que formatam os controles renderizados com o CSS do Bootstrap. Para receber os controles renderizados, fiz novamente uso de `cards`.

Ao final, temos os scripts jQuery que deverão ser inseridos na renderização da visão. Abra a visão parcial indicada no código e verifique as bibliotecas que serão inseridas nela. Já comentei sobre o método `@Html.RenderPartialAsync()` anteriormente. Logo, usaremos alguns dos recursos disponibilizados por essas bibliotecas.

Mudei as cores dos botões do rodapé apenas para que você veja esta variedade. Note que também retiramos as tags `<html>`, `<head>` e `<body>`, pois são definidas no layout.

Teste sua aplicação e requisite a visão `Create`. Ela deverá ser semelhante à apresentada na figura a seguir:

Registrando uma nova instituição

Nome

Endereço

[Registrar Instituição](#) [Retornar à listagem de instituições](#)

Informe os dados acima e/ou clique em um dos botões de ação

Figura 3.9: A visão Create de Instituição

A visão Edit

Na alteração para a visão `Edit` utilizar o layout e o Bootstrap, não há nada de novo para se apresentar, pois é uma visão semelhante à `Create`. Comprove no código a seguir. Após a implementação, execute sua aplicação, requisite a visão `Edit` e verifique se o layout está sendo usado de maneira correta.

```
@model Capitulo02.Models.Instituicao
@{
    Layout = "_LayoutIES";
}

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h
1">Alterando uma instituição existente</div>
    <div class="card-body">
        <form asp-action="Edit">
            <input type="hidden" asp-for="InstituicaoID" />
            <div asp-validation-summary="ModelOnly" class="text-d
anger"></div>
            <div class="form-group">
                <label asp-for="Nome" class="control-label"></lab
el>
                <input asp-for="Nome" class="form-control" />
                <span asp-validation-for="Nome" class="text-dange
r"></span>
            </div>
        </form>
    </div>
</div>
```

```

        <div class="form-group">
            <label asp-for="Endereco" class="control-label"><label>
label>
            <input asp-for="Endereco" class="form-control" />
            <span asp-validation-for="Endereco" class="text-d
anger"></span>
        </div>
        <div class="form-group text-center h3">
            <input type="submit" value="Atualizar Instituição
class="btn btn-primary" />
            <a asp-action="Index" class="btn btn-warning">Ret
ornar à listagem de instituições</a>
        </div>
    </form>
</div>
<div class="card-footer bg-info text-center text-white">
    Informe os dados acima e/ou clique em um dos botões de aç
ão
</div>
</div>
@section ScriptPage {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
}

```

A visão Details

Nesta visão, há algo novo: uma maneira de visualizar os dados, diferente da que vimos no capítulo anterior. O Bootstrap tem um componente, chamado `Input Group`, que partitiona uma linha em duas, dando um efeito especial no controle. Há também o uso de fontes especiais, que renderizam ícones – os quais usaremos nos `input groups`.

O componente usado foi o `Font Awesome`, na versão 4.7.0. Neste exemplo, como optei por mostrar os dados em controles que recebem dados, precisei desabilitá-los, pelo atributo `disabled="disabled"`.

O Font Awesome (<http://fontawesome.io>) contém as fontes com imagens que vamos usar. Para instalar esse componente, utilize o Bower, tal qual fizemos para as dependências anteriores.

Caso você se registre no site, você pode receber orientações para usar CDN com o componente. O uso sem registro está limitado a usar o arquivo disponibilizado em sua aplicação. Veja na listagem adiante a implementação final para a visão Details .

No código, a implementação do Input Group acontece pela classe input-group nas `<div>`, e a inserção de um `` com a classe input-group-addon . É dentro do `` que inserimos o ícone do Font Awesome.

Veja que, para a propriedade InstituicaoID , são utilizadas as classes fa e fa-key . O atributo aria-hidden="true" informa que, caso não haja espaço no dispositivo em que a visão será renderizada, o ícone pode ser omitido. Você pode obter a relação dos ícones disponíveis em <http://fontawesome.io/icons/>.

```
@model Capitulo02.Models.Instituicao
 @{
    Layout = "_LayoutIES";
}

@section styles {
    <link rel="stylesheet" href="~/lib/font-awesome/css/font-awesome.min.css" />
}

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h-1">Exibindo uma instituição existente</div>
    <div class="card-body">
        <div class="form-group">
            <label asp-for="InstituicaoID" class="control-label">
        </label>
        <br />
```

```

<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-key" aria-hidden="true"></i>
    </span>
    <input asp-for="InstituicaoID" class="form-control" disabled="disabled" />
</div>
<label asp-for="Nome" class="control-label"></label>
<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-user-circle-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Nome" class="form-control" disabled="disabled" />
</div>
<label asp-for="Endereco" class="control-label"></label>
<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-address-card-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Endereco" class="form-control" disabled="disabled" />
</div>
</div>
<div class="card-footer bg-info text-center text-white">
    <a asp-action="Edit" class="btn btn-warning" asp-route-id="@Model.InstituicaoID">Alterar</a> |
    <a asp-action="Index" class="btn btn-warning">Retornar à listagem de instituições</a>
</div>
</div>

```

Teste sua aplicação e faça a requisição dos detalhes de alguma instituição. A figura a seguir apresenta o recorte da página renderizada:

Exibindo uma instituição existente

InstituicaoID

Nome
 UniAcre

Endereco

[Retornar à listagem de instituições](#)

Figura 3.10: A visão Details de Instituição

A visão Delete

Na implementação desta última visão do CRUD, com o Bootstrap, veremos um novo recurso: a exibição de mensagens. Para essa implementação, também teremos a passagem de valores entre a visão e o controlador, com `TempData`.

Para começarmos, na sequência, veja a visão gerada pela action `Delete` (GET).

```
@model Capitulo02.Models.Instituicao
@{
    Layout = "_LayoutIES";
}

@section styles {
    <link rel="stylesheet" href="~/lib/font-awesome/css/font-awesome.min.css" />
}

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h-100">Exibindo uma instituição existente</div>
    <div class="card-body">
        <div class="form-group">
            <label asp-for="InstituicaoID" class="control-label">
        </label>
```

```

<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-key" aria-hidden="true"></i>
    </span>
    <input asp-for="InstituicaoID" class="form-control" disabled="disabled" />
</div>
<label asp-for="Nome" class="control-label"></label>
<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-user-circle-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Nome" class="form-control" disabled="disabled" />
</div>
<label asp-for="Endereco" class="control-label"></label>
else
    <br />
    <div class="input-group">
        <span class="input-group-addon">
            <i class="fa fa-address-card-o" aria-hidden="true"></i>
        </span>
        <input asp-for="Endereco" class="form-control" disabled="disabled" />
    </div>
</div>
<div class="card-footer bg-info text-center text-white">
    <form asp-action="Delete">
        <input type="hidden" asp-for="InstituicaoID" />
        <input type="submit" value="Remover Instituição" class="btn btn-light" />
        <a asp-action="Index" class="btn btn-info">Retornar à listagem de instituições</a>
    </form>
</div>
</div>
@section ScriptPage {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}

```

Nessa listagem, note que não há nada de novo. No código a seguir, que se refere à action `Delete` (`POST`), veja a inclusão da instrução `TempData["Message"] = "Instituição " + instituicao.Nome.ToUpper() + " foi removida";` . Com ela, criamos um valor associado à chave `Message` . Na visão, será possível recuperar este valor.

```
// POST: Instituicao/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(long? id)
{
    var instituicao = await _context.Instituicoes.SingleOrDefaultAsync(m => m.InstituicaoID == id);
    _context.Instituicoes.Remove(instituicao);
    TempData["Message"] = "Instituição " + instituicao.Nome.ToUpper() + " foi removida";
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

TEMPDATA

O TempData é um recurso útil quando se deseja armazenar um valor em uma curta sessão de tempo, entre requisições. A princípio, cria-se uma chave e armazena-se nela um valor. Este ficará disponível até que ele seja recuperado.

No momento de recuperá-lo, ele fica marcado para ser removido ao final da requisição. Você pode dar uma olhada nos métodos Peek() e Keep() de TempData para ver como evitar este comportamento.

Em nosso exemplo, criamos a chave Message , e nela armazenamos um valor. Este será recuperado na visão que renderiza os dados da Instituição a ser removida, como pode ser visto na listagem a seguir (visão Index).

Uma boa leitura sobre este recurso pode ser realizada nos artigos:

What is ViewData, ViewBag and TempData? – MVC Options for Passing Data Between Current and Subsequent Request –
<http://www.codeproject.com/Articles/476967/What-is-ViewData-ViewBag-and-TempData-MVC-Option>

ASP.Net MVC – ViewData, ViewBag e TempData –
<http://eduardopires.net.br/2013/06/asp-net-mvc-viewdata-viewbag-tempdata/>

```
@model IEnumerable<Capitulo02.Models.Instituicao>
@{
```

```

        Layout = "_LayoutIES";
    }

@section styles {
    <link rel="stylesheet" href("~/lib/datatables/media/css/jquery.dataTables.min.css" />
}

@if (@ TempData["Message"] != null) {
    <div class="alert alert-success" role="alert">
        @ TempData["Message"]
    </div>
}

/*
    Demais instruções foram ocultadas
*/

```

No código anterior, note a inclusão da instrução `if`, que verifica se há algum valor declarado e ainda não recuperado na chave `Message`. Caso haja, um elemento `<div>` é inserido. Nesse elemento, note as classes do Bootstrap e o papel (`role`). Esta configuração faz com que o conteúdo no corpo do elemento `<div>` seja exibido em uma caixa de mensagens:



The screenshot shows a web page titled "Instituições Registradas". At the top, there is a green header bar with the text "Instituição UNIACRE foi removida". Below this, a red arrow points to a blue header bar with the title "Instituições Registradas". The main content area contains a table with one row of data. The table has columns for "InstituicaoID", "Nome", and "Endereço". The first row shows "1", "UniParaná", and "Paraná". To the right of the table are links for "Edit | Details | Delete". At the bottom of the table, it says "Showing 1 to 1 of 1 entries". On the right side of the page, there are navigation links for "Previous", "1", and "Next". A green footer bar at the bottom has the text "Criar uma nova instituição".

Figura 3.11: A visão Index de Instituições exibindo uma Mensagem Bootstrap

3.5 CONFIGURAÇÃO DO MENU PARA ACESSAR AS VISÕES CRIADAS

Concluímos todas as visões para o CRUD de Instituições . Entretanto, estamos acessando as visões de testes ao digitar diretamente a URL no navegador. Precisamos alterar o menu de opções que está no layout compartilhado, o _LayoutIES .

No código da visão, localize o trecho que apresenta as opções oferecidas ao usuário, e deixe-o como mostra o código a seguir. Veja que inseri dois Tag Helpers, o `asp-controller` e o `asp-action` , que já utilizamos em códigos anteriores.

```
<div class="col-2">
    <nav class="nav flex-column">
        <a class="nav-item nav-link" asp-controller="Instituicao"
asp-action="Index">
            <img src "~/images/university.png" width="30" height=
"30" alt="">
                Instituições
            </a>
        <a class="nav-item nav-link" href="#">
            <img src "~/images/department.png" width="30" height=
"30" alt="">
                Departamentos
            </a>
        </nav>
    </div>
```

3.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Este capítulo foi interessante para a camada de visão. O uso do Bootstrap foi iniciado e foram apresentados bons recursos dele. Também introduzi o uso de layouts, que permite definir uma estrutura básica para um conjunto de visões.

O uso do jQuery foi mais uma novidade. E sempre que ele for necessário, uma explicação será fornecida. Na parte do ASP.NET

Core MVC em si, apenas foi mostrado o uso do `TempData` para o envio de valores às visões. Em relação ao Entity Framework Core, nada de novo foi apresentado, mas, no próximo capítulo, isso será compensado. Veremos mais sobre as associações.

Agora, o que acha de aplicar o layout criado nas visões de Departamentos ? Fica aí o desafio.

CAPÍTULO 4

ASSOCIAÇÕES NO ENTITY FRAMEWORK CORE

Quando desenvolvemos nosso modelo de negócio, muitas de nossas classes estão associadas a outras. Em Orientação a Objetos (OO), uma associação representa um vínculo entre objetos de uma ou mais classes, de maneira que estes se relacionem. É possível que, por meio desses vínculos (associações), um objeto invoque (ou requisite) serviços de outro, ou que acesse ou atribua valores às propriedades desse objeto.

Associações são importantes, pois, por meio delas, os objetos podem se relacionar, quer seja em um cadastro complexo ou em um processo que envolva diversas classes que o componham. Essas associações, no modelo relacional, são conhecidas como relacionamentos.

Tanto para o modelo relacional como o de objetos, existem algumas regras quando implementamos uma associação/relacionamento. Duas delas são: a multiplicidade (cardinalidade no modelo relacional) e a navegabilidade. Na primeira, é preciso especificar quantos objetos/registros são possíveis para cada lado da associação/relacionamento. Já a segunda é se é possível recuperar de um objeto/registro um outro

objeto associado.

Até o momento, tivemos apenas duas classes, que geraram duas tabelas, mas nenhuma está associada. Implementar a associação de classes usando o Entity Framework Core é um dos objetivos deste capítulo.

4.1 ASSOCIANDO AS CLASSES JÁ CRIADAS

Em nosso contexto de problema, implementamos as funcionalidades de instituições e departamentos para que pudéssemos ter estes dados registrados em nossas aplicações. O que precisamos agora é associar essas duas classes.

A princípio, trabalharei a situação de que um departamento pertence a uma instituição. Então, quando instanciarmos um objeto de departamento, ele precisará saber a qual objeto de instituição ele pertence. Na listagem a seguir, veja o novo código para a classe Departamento :

```
namespace Capitulo02.Models
{
    public class Departamento
    {
        public long? DepartamentoID { get; set; }
        public string Nome { get; set; }

        public long? InstituicaoID { get; set; }
        public Instituicao Instituicao { get; set; }
    }
}
```

Nesse código, note que as propriedades DepartamentoID (chave primária) e InstituicaoID (chaves estrangeiras) são do tipo `long?`, ou seja, aceitam valores nulos. Verifique também a

propriedade `Instituicao`, que representa as associações com objetos de seus respectivos tipos.

Você pode se perguntar: *"Por que manter o ID da associação se já possuímos a associação com seu respectivo atributo?"*. Este problema refere-se ao carregamento (ou recuperação) do objeto da base de dados. O Entity Framework Core oferece três tipos de carregamento de objetos: *Eagerly Loading* (carregamento forçado), *Lazy Loading* (carregamento tardio) e *Explicitly Loading* (carregamento explícito).

No carregamento de um objeto `Departamento`, a propriedade que representa a chave estrangeira (`InstituicaoID`), mapeada diretamente para uma coluna na tabela da base de dados, é carregada imediatamente. Já a propriedade `Instituicao` possui dados relativos às suas propriedades que, dependendo do tipo de carregamento usado, podem ou não ser carregados em conjunto.

Quando o carregamento ocorrer em conjunto com um objeto `Departamento` (*Eagerly Loading*), o SQL mapeado terá um *join* (junção) com a tabela `Instituicao`, para que todos os dados sejam carregados por meio de um único `select`. Já se o carregamento for tardio (*Lazy Loading*), quando alguma propriedade do objeto `Instituicao` (do objeto `Departamento`) for requisitada, um novo `select` (SQL) será executado. Perceba que isso refere-se diretamente à performance das consultas realizadas na base de dados.

Perfeito, já temos uma associação implementada e pronta para que o EF Core aplique-a na base de dados. Ela não precisa obrigatoriamente de nenhuma implementação adicional, pois tudo será feito por convenção. A associação implementada possui

navegabilidade da classe Departamento para Instituicao . Ou seja, é possível saber qual é a Instituicao de um Departamento .

Já em relação à sua multiplicidade, ela é um, pois cada Departamento está associado a apenas uma Instituicao . Mas e como fica a associação de Instituicao para Departamento ? Uma Instituicao pode ter vários Departamento s.

Com isso, já identificamos a multiplicidade de *muitos* (*ou um*) para *muitos*, mas não temos ainda a navegabilidade. Isto é, não é possível identificar quais departamentos uma Instituicao possui. A implementação da multiplicidade e da navegabilidade está na listagem a seguir.

```
using System.Collections.Generic;

namespace Capitulo02.Models
{
    public class Instituicao
    {
        public long? InstituicaoID { get; set; }
        public string Nome { get; set; }
        public string Endereco { get; set; }

        public virtual ICollection<Departamento> Departamentos {
            get; set; }
    }
}
```

Nessa listagem, a propriedade Departamentos é uma ICollection<Departamento> , e é definida como virtual . Definir elementos como virtual possibilita a sua sobrescrita, o que, para o EF Core, é necessário. Assim, ele poderá fazer o *Lazy Load* (carregamento tardio), por meio de um padrão de projeto conhecido como Proxy. Não entrarei em detalhes sobre isso, pois

não faz parte de nosso escopo, mas é importante você ter este conhecimento.

VIRTUAL

A `virtual` é uma palavra-chave usada para modificar uma declaração de método, propriedade, indexador ou evento, e permitir que ele seja sobreescrito em uma classe derivada. Para mais informações, recomendo a leitura do artigo *Virtual vs Override vs New Keyword in C#*, de Abhishek Jaiswall, em <http://www.codeproject.com/Articles/816448/Virtual-vs-Override-vs-New-Keyword-in-Csharp>.

ICOLLECTION

A escolha da interface `ICollection` para uma propriedade deve-se ao fato de que, com ela, é possível iterar (navegar) nos objetos recuperados e modificá-los. Existe ainda a possibilidade de utilizar `IEnumerable` apenas para navegar, e `IList` quando precisar de recursos a mais, como uma classificação dos elementos.

Para mais informações, veja o interessante artigo *List vs IEnumerable vs IQueryable vs ICollection vs IDictionary*, de Mahsa Hassankashi, em <http://www.codeproject.com/Articles/832189/List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v>.

Quando a tabela na base de dados for atualizada para ter a associação mapeada (criada na classe Departamento), uma coluna chamada InstituicaoID será adicionada. Não é preciso que você modifique nada sozinho, pois instruiremos nossa aplicação a eliminar a base de dados que já temos criada, para então criá-la novamente – agora com a associação.

No capítulo *Code First Migrations, Data Annotations e validações*, trabalharemos com Migrations, que permitirá que esse processo de atualização da base de dados seja realizado sem que ela precise ser eliminada.

Na classe IESDbInitializer, antes da chamada ao método context.Database.EnsureCreated(), invoque o método context.Database.EnsureDeleted(), responsável pela remoção da base de dados. Em relação à inserção de dados de teste para as classes, implemente o código a seguir, após o EnsureCreated() – atualizando o código que você já tem codificado.

```
if (context.Departamentos.Any())
{
    return;
}

var instituicoes = new Instituicao[]
{
    new Instituicao { Nome="UniParaná", Endereco="Paraná"}, 
    new Instituicao { Nome="UniAcre", Endereco="Acre"} 
};

foreach (Instituicao i in instituicoes)
{
    context.Instituicoes.Add(i);
}
context.SaveChanges();
```

```

var departamentos = new Departamento[]
{
    new Departamento { Nome="Ciéncia da Computação", InstituicaoID=1 },
    new Departamento { Nome="Ciéncia de Alimentos", InstituicaoID=2 }
};

foreach (Departamento d in departamentos)
{
    context.Departamentos.Add(d);
}
context.SaveChanges();

```

4.2 ADAPTAÇÃO PARA USO DAS ASSOCIAÇÕES

Nossa primeira adaptação será realizada na visão Index de Departamentos , para exibirmos na listagem o nome da instituição à qual o departamento faz parte. Entretanto, como estamos removendo a base de dados e criando-a a cada execução – lembre do context.Database.EnsureDeleted() e do context.Database.EnsureCreated() no IESDbInitializer –, primeiro vamos adaptar os dados que são alimentados na inicialização da aplicação.

Na sequência, veja o trecho do código atualizado, que insere os departamentos. É preciso colocar a chamada ao context.SaveChanges() antes da inicialização, para que os objetos de instituição sejam efetivamente persistidos. Agora, na inicialização dos objetos de departamento, utilizamos a propriedade InstituicaoID .

```

context.SaveChanges();

var departamentos = new Departamento[]

```

```

{
    new Departamento { Nome="Ciéncia da Computação", InstituicaoID=1 },
    new Departamento { Nome="Ciéncia de Alimentos", InstituicaoID=2 }
};

```

A visão Index de Departamento

Quando implementamos a visão Index para Instituicao , os dados exibidos no DataTable vinham todos de uma única tabela da base de dados, que mapeamos para uma classe com o EF Core. Agora, vamos apresentar ao usuário dados de duas tabelas, pois mostraremos o nome da instituição a que um departamento pertence.

Na tabela de departamentos, temos apenas o ID, mapeado para a propriedade InstituicaoID . Veja o código da visão Index de Departamentos na sequência.

```

@model IEnumerable<Capitulo02.Models.Departamento>
 @{
     Layout = "_LayoutIES";
 }

@section styles {
    <link rel="stylesheet" href("~/lib/datatables/media/css/jquery.dataTables.min.css" />
}

@if (@ TempData["Message"] != null)
{
    <div class="alert alert-success" role="alert">
        @ TempData["Message"]
    </div>
}

<div class="card-block">
    <div class="card-header text-white bg-primary text-center h1">
        Departamentos Registrados</div>

```

```

<div class="card-body">
    <table id="tabela_departamentos">
        <thead>
            <tr>
                <th>
                    @Html.DisplayNameFor(model => model.Nome)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Insti
tuicao.Nome)
                </th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model)
            {
                <tr>
                    <td>
                        @Html.DisplayFor(modelItem => item.No
me)
                    </td>
                    <td>
                        @Html.DisplayFor(modelItem => item.In
stituicao.Nome)
                    </td>
                    <td>
                        <a asp-action="Edit" asp-route-id="@i
tem.DepartamentoID">Edit</a> | 
                        <a asp-action="Details" asp-route-id=
"@item.DepartamentoID">Details</a> | 
                        <a asp-action="Delete" asp-route-id="
@item.DepartamentoID">Delete</a>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
<div class="card-footer bg-success text-center">
    <a asp-action="Create" class="btn-success">Criar um novo
departamento</a>
</div>
</div>

```

```

@section ScriptPage {
    <script type="text/javascript" src "~/lib/datatables/media/js/jquery.dataTables.min.js"></script>

    <script type="text/javascript">
        $(document).ready(function () {
            $('#tabela_departamentos').DataTable({
                "order": [[1, "asc"]]
            });
        });
    </script>
}

```

Note que a única diferença que temos é a invocação de uma propriedade e, como resultado dessa propriedade, uma outra é chamada. Estou falando da instrução `@Html.DisplayFor(modelItem => item.Instituicao.Nome)`.

Execute sua aplicação e veja que nada é exibido na segunda coluna, chamada `Nome` e referente ao nome da instituição. Essa situação ocorre por conta do tipo de carregamento padrão que o EF Core faz quando ele recupera um objeto de departamentos, que é a tardia. Precisamos mudar isso.

Queremos que o objeto do departamento recuperado já traga os dados da instituição relacionada a ele. Ou seja, precisamos de um carregamento forçado. Para isso, mudaremos nossa action `Index` para que fique semelhante ao apresentado na sequência:

```

public async Task<IActionResult> Index()
{
    return View(await _context.Departamentos.Include(i => i.Instituicao)
        .OrderBy(c => c.Nome).ToListAsync());
}

```

A instrução anterior seleciona todas as instituições relacionadas aos departamentos recuperados, por meio da inserção

do método `Include()`. Existem outros métodos que auxiliam neste processo, como o `ThenInclude()` e o `Load()`.

No caso do primeiro, o uso é recomendado quando temos mais de uma tabela vinculada à que está sendo recuperada. Já o `Load()` pode ser utilizado para recuperar objetos relacionados a uma determinada condição – em vez de usar o `ToListAsync()`, que usamos anteriormente.

Existe também o método `Single()`, que recupera um único objeto, com base em uma condição. O `Load()` ainda pode ser usado no carregamento explícito de uma propriedade que representa uma coleção de dados – como o caso da propriedade `Departamentos` de `Instituicao`. Conforme esses recursos forem necessários, as explicações e os exemplos serão trazidos.

4.3 A VISÃO CREATE PARA A CLASSE DEPARTAMENTO

Com a visão `Index` de `Departamentos` concluída, vamos criar agora a visão `Create`. Deixe a action `GET` de `Create` tal qual o código na sequência.

Observe que, uma vez mais, utilizamos a `ViewBag`, agora para armazenar objetos de instituições. Note também que primeiro os objetos são recuperados da base de dados e classificados por nome, para então serem atribuídos a uma variável. Depois, um objeto é inserido na coleção, logo no início, realocando os demais.

Isso é feito para que uma mensagem orientativa seja exibida ao usuário no `DropDownList` (que será criado). Para que essa coleção de dados possa ser usada na visão, ela é atribuída a uma

chave na ViewBag .

```
// GET: Departamento/Create
public IActionResult Create()
{
    var instituicoes = _context.Instituicoes.OrderBy(i => i.Nome)
        .ToList();
    instituicoes.Insert(0, new Instituicao() { InstituicaoID = 0,
        Nome = "Selecione a instituição" });
    ViewBag.Instituicoes = instituicoes;
    return View();
}
```

Com os dados recuperados e transferidos para a visão, vamos criá-la. Veja o seu código na listagem a seguir.

No segundo <div> da classe de estilo form-group , em vez de termos um elemento <input> , temos um elemento <select> do HTML. Ele é o responsável pela renderização no estilo DropDownList para a coleção de dados informada na Tag Helper asp-items .

Para popular esse elemento, é instanciado um novo objeto de SelectList() , que informa a coleção, o campo chave e o campo com o texto a ser informado ao usuário, em seu construtor.

```
@model Capitulo02.Models.Departamento
 @{
    Layout = "_LayoutIES";
}

<div class="card-block">
    <div class="card-header text-white bg-danger text-center h1">
        Registrando um novo departamento</div>
    <div class="card-body">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Nome" class="control-label"></label>
```

```

el>
    <input asp-for="Nome" class="form-control" />
    <span asp-validation-for="Nome" class="text-dange
r"></span>
</div>
<div class="form-group">
    <label asp-for="InstituicaoID" class="control-lab
el"></label>
    <select asp-for="InstituicaoID" class="form-contr
ol" asp-items="@{new SelectList(@ViewBag.Instituicoes, "Instituic
aoID", "Nome"))}" ></select>
</div>
<div class="form-group text-center h3">
    <input type="submit" value="Registrar Departament
o" class="btn btn-light" />
    <a asp-action="Index" class="btn btn-info">Retorn
ar à listagem de departamentos</a>
</div>
</form>
</div>
<div class="card-footer bg-dark text-center text-white">
    Informe os dados acima e/ou clique em um dos botões de aç
ão
</div>
</div>

@section ScriptPage {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}

```

SELECTLIST()

O `SelectList()` representa uma lista de itens da qual o usuário pode selecionar um item. Em nosso caso, os itens serão expostos em um `DropDownList`. A classe possui diversos construtores, mas o usado no código anterior recebe:

1. A coleção de itens que popularão o `DropDownList`;
2. A propriedade que representa o valor que será armazenado no controle;
3. A propriedade que possui o valor a ser exibido pelo controle.

Então, precisamos mudar o `Binding` do método que representa a action `POST` da visão `Create`. Veja a assinatura nova para o método e a inserção da propriedade `InstituicaoID` no código:

```
public async Task<IActionResult> Create([Bind("Nome, InstituicaoID")] Departamento departamento)
```

Com a implementação realizada, resta testar a aplicação. Acesse a visão `Create` de `Departamentos`, e você receberá uma visão semelhante ao recorte da figura a seguir:

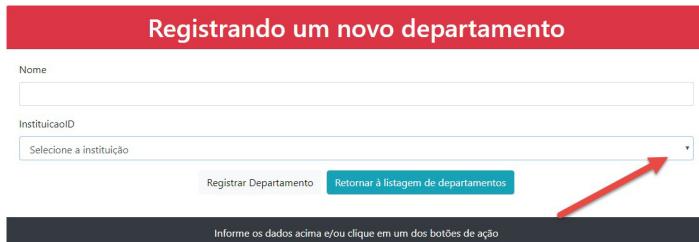


Figura 4.1: Visão Create de Departamentos com o DropDownList para a classe associada

4.4 A VISÃO EDIT PARA A CLASSE DEPARTAMENTO

Com a inserção dos novos departamentos concluída, é necessário implementar a funcionalidade para a alteração de dados de um departamento, a action `Edit`. O código adiante apresenta o método que representa essa action para a geração da visão.

Observe que usaremos `SelectList` nela, pois vamos apenas referenciá-la como fonte de dados na visão. Um quarto argumento é utilizado na invocação, já que queremos mostrar qual é a instituição que está associada ao objeto em consulta.

```
// GET: Departamento/Edit/5
public async Task<IActionResult> Edit(long? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var departamento = await _context.Departamentos.SingleOrDefault(m => m.DepartamentoID == id);
    if (departamento == null)
    {
        return NotFound();
    }
    ViewBag.Instituicoes = new SelectList(_context.Instituicoes.O
```

```

rderBy(b => b.Nome), "InstituicaoID", "Nome", departamento.Instit
uicaoID);

        return View(departamento);
}

```

A visão a ser gerada pelo código anterior precisa ser implementada de acordo com o código seguinte (visão `Edit`). Veja no elemento `<select>` que é apenas feita referência ao elemento da `ViewBag`.

```

@model Capitulo02.Models.Departamento
 @{
    Layout = "_LayoutIES";
}
<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h
1">Alterando um departamento existente</div>
    <div class="card-body">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-d
anger"></div>
            <input type="hidden" asp-for="DepartamentoID" />
            <div class="form-group">
                <label asp-for="Nome" class="control-label"></lab
el>
                    <input asp-for="Nome" class="form-control" />
                    <span asp-validation-for="Nome" class="text-dange
r"></span>
                </div>
                <div class="form-group">
                    <label asp-for="InstituicaoID" class="control-lab
el"></label>
                    <select asp-for="InstituicaoID" class="form-contr
ol" asp-items=@ViewBag.Instituicoes></select>
                </div>
                <div class="form-group text-center h3">
                    <input type="submit" value="Atualizar Departament
o" class="btn btn-primary" />
                    <a asp-action="Index" class="btn btn-warning">Ret
ornar à listagem de departamentos</a>
                </div>
            </form>

```

```
        </div>
    </div>

@section ScriptPage {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}

}
```

Após a alteração dos dados, na visão `Edit`, o usuário requisitará a action `Edit` (`POST`) para persistir suas alterações. O código para essa action – a exemplo do que foi feito para o `Create/POST` – precisa ser alterado apenas no cabeçalho, como segue:

```
public async Task<IActionResult> Edit(long? id, [Bind("Departamen
toID, Nome, InstituicaoID")] Departamento departamento)
```

Ao final, antes do retorno que renderiza a visão `Edit` novamente, é preciso popular a lista de instituições uma vez mais, em caso de erro. Veja o trecho a seguir:

```
ViewBag.Instituicoes = new SelectList(_context.Instituicoes.Order
By(b => b.Nome), "InstituicaoID", "Nome", departamento.Instituica
oID);
```

4.5 A VISÃO DETAILS PARA A CLASSE DEPARTAMENTO

A visão `Details` terá interação apenas com uma action `Details`, responsável por renderizá-la. Após isso, o usuário poderá retornar à listagem (action `Index`), ou alterar os dados (action `Edit`).

O código a seguir traz a action `Details` que renderiza essa visão. Vamos usar: o `SingleOrDefaultAsync()`, que retorna o primeiro registro que satisfaça a condição; e o `Load()`, que

carrega o objeto desejado no contexto, ou seja, a instituição do departamento procurado.

```
public async Task<IActionResult> Details(long? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var departamento = await _context.Departamentos
        .SingleOrDefaultAsync(m => m.DepartamentoID == id);
    _context.Instituicoes.Where(i => departamento.InstituicaoID =
= i.InstituicaoID).Load(); ;
    if (departamento == null)
    {
        return NotFound();
    }
    return View(departamento);
}
```

O passo seguinte é a criação da visão `Details`. Siga os passos já vistos anteriormente para isso. O código final deverá ser o apresentado na sequência; não há nada de novo nele.

```
@model Capitulo02.Models.Departamento

 @{
    Layout = "_LayoutIES";
}

@section styles {
    <link rel="stylesheet" href "~/lib/font-awesome/css/font-awesome.min.css" />
}

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h-1">Exibindo uma departamento existente</div>
    <div class="card-body">
        <div class="form-group">
            <label asp-for="DepartamentoID" class="control-label">
        </label>
```

```

<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-key" aria-hidden="true"></i>
    </span>
    <input asp-for="DepartamentoID" class="form-control" disabled="disabled" />
</div>
<label asp-for="Nome" class="control-label"></label>
<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-user-circle-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Nome" class="form-control" disabled="disabled" />
</div>
<label asp-for="Instituicao.Nome" class="control-label"></label>
<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-user-circle-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Instituicao.Nome" class="form-control" disabled="disabled" />
</div>
</div>
<div class="card-footer bg-info text-center text-white">
    <a asp-action="Edit" class="btn btn-warning" asp-route-id="@Model.DepartamentoID">Alterar</a> |
    <a asp-action="Index" class="btn btn-warning">Retornar à listagem de departamentos</a>
</div>
</div>

```

4.6 CRIANDO A VISÃO DELETE PARA A CLASSE DEPARTAMENTO

Para finalizar o CRUD, como nos casos anteriores, deixamos a action e a view `Delete` por último. Veja no código a seguir a semelhança com a action `Details` e `Edit`.

```
// GET: Departamento/Delete/5
public async Task<IActionResult> Delete(long? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var departamento = await _context.Departamentos
        .SingleOrDefaultAsync(m => m.DepartamentoID == id);
    _context.Instituicoes.Where(i => departamento.InstituicaoID =
= i.InstituicaoID).Load();
    if (departamento == null)
    {
        return NotFound();
    }

    return View(departamento);
}
```

Para a visão, teremos uma inovação: o uso de caixas de diálogo modal do Bootstrap. Veja o código da visão na sequência. Na `<div>` que representa o `card-footer`, verifique a presença de um `<button>` com os atributos `data-toggle="modal"` e `data-target="#modalConfirmationDelete"`. O primeiro especifica para o Bootstrap que o link gerará uma janela "modal", e o segundo faz referência à `<div>` com o ID `modalConfirmationDelete`, definido ao final do código.

Abaixo do fechamento da `<div>` do `card-block`, está a `<div> modalConfirmationDelete`, que representa o modal. Veja que existem partes para o modal (*header*, *body* e *footer*). Dentro do *body*, existe a declaração de um formulário e, dentro

dele, encontramos a declaração do `id` do departamento como campo oculto (já vimos isso anteriormente).

No rodapé, existem dois *buttons*, sendo o primeiro responsável pela submissão do formulário, que é o declarado no footer do modal. No segundo botão, verifique a existência do atributo `data-dismiss="modal"` , que define o botão como de fechamento do modal.

```
@model Capitulo02.Models.Departamento
#{@
    Layout = "_LayoutIES";
}

@section styles {
    <link rel="stylesheet" href="~/lib/font-awesome/css/font-awesome.min.css" />
}

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h1">Exibindo uma departamento para remoção</div>
    <div class="card-body">
        <div class="form-group">
            <label asp-for="DepartamentoID" class="control-label">
        </label>
        <br />
        <div class="input-group">
            <span class="input-group-addon">
                <i class="fa fa-key" aria-hidden="true"></i>
            </span>
            <input asp-for="DepartamentoID" class="form-control" disabled="disabled" />
        </div>
        <label asp-for="Nome" class="control-label"></label>
        <br />
        <div class="input-group">
            <span class="input-group-addon">
                <i class="fa fa-user-circle-o" aria-hidden="true"></i>
            </span>
            <input asp-for="Nome" class="form-control" disabled="disabled" />
        </div>
    </div>
</div>
```

```

        ed="disabled" />
    </div>
    <label asp-for="Instituicao.Nome" class="control-labe
l"></label>
    <br />
    <div class="input-group">
        <span class="input-group-addon">
            <i class="fa fa-user-circle-o" aria-hidden="t
rue"></i>
        </span>
        <input asp-for="Instituicao.Nome" class="form-con
trol" disabled="disabled" />
    </div>
</div>
<div class="card-footer bg-info text-center text-white">
    <a asp-action="Index" class="btn btn-danger">Retornar
    à listagem de Departamentos</a>
    <button type="button" class="btn btn-dark" data-toggler
="modal" data-target="#modalConfirmationDelete">
        Remover Departamento
    </button>
</div>
</div>

<div class="modal fade" id="modalConfirmationDelete" tabindex="-1"
role="dialog" aria-labelledby="deleteModal" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="deleteModal">Remoção
                de Departamento</h5>
                <button type="button" class="close" data-dismiss=
"modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <p>Confirma a exclusão do departamento @Model.Nom
e.ToUpper() ??</p>
            </div>
            <div class="modal-footer">
                <form asp-action="Delete">
                    <input type="hidden" asp-for="DepartamentoID"
/>

```

```

        <input type="submit" value="Remover Departamento" class="btn btn-primary" />
        <button type="button" class="btn btn-secondary" data-dismiss="modal">Fechar</button>
    </form>
</div>
</div>
</div>

```

Para concluir a operação de remoção de Departamentos , precisamos implementar a action Delete para o POST . Seu código pode ser verificado na sequência:

```

// POST: Departamento/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task DeleteConfirmed(long? id)
{
    var departamento = await _context.Departamentos.SingleOrDefaultAsync(m => m.DepartamentoID == id);
    _context.Departamentos.Remove(departamento);
    TempData["Message"] = "Departamento " + departamento.Nome.ToUpper() + " foi removido";
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}

```

Execute agora sua aplicação, e teste a remoção de um departamento. A exibição do modal é representada pela figura a seguir:

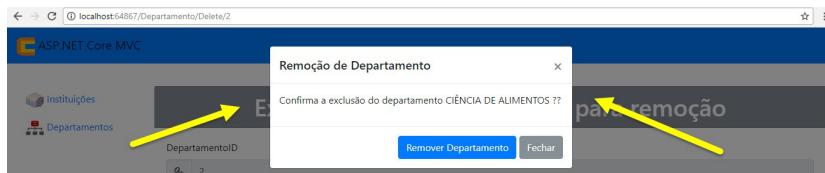


Figura 4.2: Visão DELETE com destaque para o modal de confirmação

4.7 INSERÇÃO DE DEPARTAMENTOS NA VISÃO DETAILS DE INSTITUIÇÕES

No início do capítulo, implementamos a associação *um para muitos* na classe `Instituicao`, em que cada instituição pode ter muitos departamentos. Precisamos agora fazer com que essa associação possa ser visualizada.

Inicialmente, faremos isso na visão `Details`. Com esta implementação, quando o usuário visualizar os detalhes de uma `Instituicao`, ele terá acesso à listagem de `Departamentos` da instituição em questão.

Entretanto, para que a visão possa exibir os departamentos associados às instituições, é preciso inserir a chamada ao método `Include(d => d.Departamentos)` logo antes do `SingleOrDefaultAsync()`. Sua instrução deverá ficar semelhante ao código a seguir. Lembre-se de que estamos falando da action `Details` do `DepartamentoController`.

```
var instituicao = await _context.Instituicoes.Include(d => d.Departamentos).SingleOrDefaultAsync(m => m.InstituicaoID == id);
```

Já para a visão, teremos o conceito de Partial View. Para isso, é preciso criar um novo arquivo na pasta `Views` de `Instituicao` – e recomendo o nome de `_ComDepartamentos.cshtml`. Esse arquivo precisará ter o conteúdo apresentado no código seguinte. Na janela de criação da visão, desmarque todas as caixas de checagem e também deixe sem modelo.

PARTIAL VIEW

Partial Views são visões que contêm código (HTML e/ou Razor) e são projetadas para serem renderizadas como parte de uma visão. Elas não possuem layouts, como as visões, e podem ser "inseridas" dentro de diversas visões, como se fossem um componente ou controle.

Recomendo a leitura do artigo *Work with Partial view in MVC framework*, de Steve Smith, Maher Jendoubi e Rick Anderson, em <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/partial>.

```
@model IEnumerable<Projeto01.Models.Departamento>

<div class="panel panel-primary">
    <div class="panel-heading">
        Relação de DEPARTAMENTOS registrados para a instituição
    </div>
    <div class="panel-body">
        <table class="table table-striped table-hover">
            <thead>
                <tr>
                    <th>
                        @Html.DisplayNameFor(model => model.
                            DepartamentoID)
                    </th>
                    <th>
                        @Html.DisplayNameFor(model => model.Nome)
                    </th>
                    <th>Instituicao</th>
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Model)
                {

```

```

<tr>
    <td>
        @Html.DisplayFor(modelItem => item.
            DepartamentoID)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.
            Nome)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.
            Instituicao.Nome)
    </td>
</tr>
}
</tbody>
</table>
</div>
<div class="panel-footer panel-info">
</div>
</div>

```

Agora, precisamos alterar a visão `Details`, para o código a seguir. Vamos utilizar o artifício de colunas do Bootstrap, para que os departamentos sejam exibidos ao lado dos dados da instituição.

```

@model Capitulo02.Models.Instituicao
 @{
     Layout = "_LayoutIES";
 }

@section styles {
    <link rel="stylesheet" href("~/lib/font-awesome/css/font-awesome.min.css" />
}

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h-1">Exibindo uma instituição existente</div>
    <div class="card-body">
        <div class="row">
            <div class="col-9">
                <div class="form-group">

```

```

        <label asp-for="InstituicaoID" class="control-
-label"></label>
        <br />
        <div class="input-group">
            <span class="input-group-addon">
                <i class="fa fa-key" aria-hidden="true"></i>
            </span>
            <input asp-for="InstituicaoID" class="form-
control" disabled="disabled" />
        </div>
        <label asp-for="Nome" class="control-label"><i
hidden="true"></i>
        <br />
        <div class="input-group">
            <span class="input-group-addon">
                <i class="fa fa-user-circle-o" aria-h
idden="true"></i>
            </span>
            <input asp-for="Nome" class="form-control"
disabled="disabled" />
        </div>
        <label asp-for="Endereco" class="control-labe
l"><i
hidden="true"></i>
        <br />
        <div class="input-group">
            <span class="input-group-addon">
                <i class="fa fa-address-card-o" aria-
hidden="true"></i>
            </span>
            <input asp-for="Endereco" class="form-con
trol" disabled="disabled" />
        </div>
    </div>
    <div class="col-3">
        @Html.Partial("~/Views/Instituicao/_ComDepartamen
tos.cshtml", Model.Departamentos.ToList())
    </div>
</div>
<div class="card-footer bg-info text-center text-white">
    <a asp-action="Edit" class="btn btn-warning" asp-route-id:
"@Model.InstituicaoID">Alterar</a> |
    <a asp-action="Index" class="btn btn-warning">Retornar à

```

```
listagem de instituições</a>
    </div>
</div>
```

Na segunda coluna, existe a inclusão da instrução Razor que insere a Partial View no local desejado. Veja a seguir a instrução isolada.

```
@Html.Partial("~/Views/Instituicao/_ComDepartamentos.cshtml", Model.Departamentos.ToList())
```

Com isso, criamos uma visão master-detail entre `Instituicao` e `Departamentos`. Teste sua aplicação. A figura a seguir traz o recorte da página renderizada:

The screenshot shows a web page titled "Exibindo uma instituição existente". On the left, there is a form with fields for "InstituicaoID" (set to 2), "Nome" (set to "UniAcre"), and "Endereço" (set to "Acre"). On the right, there is a yellow box titled "Departamentos Registrados na Instituição" containing a table with one row: "Nome" (set to "Ciência de Alimentos"). A red arrow points from the text "listagem de Departamentos para uma Instituição" in the caption below to the "Departamentos Registrados na Instituição" box. At the bottom, there are buttons for "Alterar" and "Retornar à listagem de instituições".

Figura 4.3: Visão Details com destaque para a listagem de Departamentos para uma Instituição

4.8 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Associações entre classes é um tema interessante e que ocorre em praticamente todas as situações. Este capítulo buscou introduzir como essas associações – tanto na multiplicidade *um para muitos* como em *muitos para um* – são mapeadas pelo Entity

Framework Core.

Na aplicação das associações pelo ASP.NET Core MVC, foram apresentadas as estratégias para o carregamento de objetos e a passagem de valores do controlador para a visão (por meio de ViewBag). Aprendemos também mais sobre a criação de um DropDownList com uma coleção de objetos – para que o usuário selecione um –, e a criação e uso de Partial Views. Já com o Bootstrap, vimos o uso de janelas de diálogo modal.

Ufa, vimos bastante coisa. Agora, no próximo capítulo, trabalharemos algumas técnicas em relação à separação de camadas em nosso projeto. Ele será bem interessante também!

CAPÍTULO 5

SEPARAÇÃO DA CAMADA DE NEGÓCIO

Por mais que exista a separação entre as pastas `Models`, `Controllers` e `Views` em nosso projeto ASP.NET Core MVC, o conteúdo delas não está componentizado, pois pertencem a um único assembly. Pode-se dizer que um sistema componentizado é um sistema no qual cada camada (ou parte) responsável está desenvolvida em módulos consumidos pelo sistema em si. Normalmente, esses módulos fazem parte de assemblies (aplicativos e/ou DLLs).

Quando o Visual Studio criou nosso template de projeto ASP.NET Core MVC, ele já o trouxe configurado para ter a separação de camadas por meio de pastas. No capítulo *Acesso a dados com o Entity Framework Core*, inserimos uma nova camada, a de persistência, mesclada com as actions nos métodos dos controladores.

Neste capítulo, separaremos a camada de negócio, o conteúdo da pasta `Models`. Vamos criar métodos de acesso aos dados, vislumbrando uma separação também da camada de persistência. Também retiraremos algumas redundâncias de código, que já foram comentadas anteriormente.

5.1 CONTEXTUALIZAÇÃO SOBRE AS CAMADAS

Tudo o que fizemos até o momento foi feito em apenas uma camada, embora usássemos o conceito de MVC e o ASP.NET Core MVC, pois tudo está em um único projeto. Além disso, nossas classes controladoras estão desempenhando atividades além do que seria de sua obrigação.

Estes dois pontos estão diretamente ligados a **acoplamento** e **coesão**. O acoplamento trata da independência dos componentes interligados e, em nosso caso, a independência é zero, pois está tudo em um único projeto. Desta maneira, temos um forte acoplamento.

Já a coesão busca medir um componente individualmente. Temos as classes controladoras, que, em nosso caso, desempenham muitas funções, uma vez que validam os dados, trabalham a persistência e ainda geram a comunicação com a visão. Logo, temos uma baixa coesão em nossa aplicação.

Com isso posto, teremos dois projetos em nossa aplicação/solução: um projeto de biblioteca para o modelo de negócio; e o que já temos para a aplicação, que continuará com a camada de persistência, de visão e a controladora.

Só com essa mudança já é possível verificar um ganho, pois nosso projeto de modelo pode ser utilizado em uma aplicação para dispositivos móveis, por exemplo. No projeto de aplicação , que possui os demais componentes, criaremos uma pasta para a persistência, com classes oferecendo serviços de acesso aos dados. Estes serão utilizados pelas actions dos controladores.

Isso facilitaria uma futura separação da camada de persistência, caso você opte por isso no futuro, nessa ou em outra aplicação. Porém, neste livro, optei por não separar a camada de persistência da camada de aplicação, pois a Injeção de Dependências para o contexto é bem simples e funcional. Vamos ao trabalho.

5.2 CRIANDO A CAMADA DE NEGÓCIO: O MODELO

Como primeira atividade deste capítulo, criaremos um projeto do tipo `Biblioteca de Classes (.NET Core)`, chamado `Modelo`. Para isso, clique com o botão direito sobre o nome de sua solução, no `Gerenciador de Soluções`, e selecione `Adicionar -> Novo Projeto`.

Na janela que se apresenta, dentro da categoria `Visual C#`, selecione `.NET Core (1)`. Na área central, selecione o template `Biblioteca de Classes (.NET Core) (2)`, e nomeie o projeto como `Modelo (3)`. Para finalizar, confirme a criação do projeto clicando no botão `Ok (4)`. O assembly gerado pelo projeto será uma DLL.

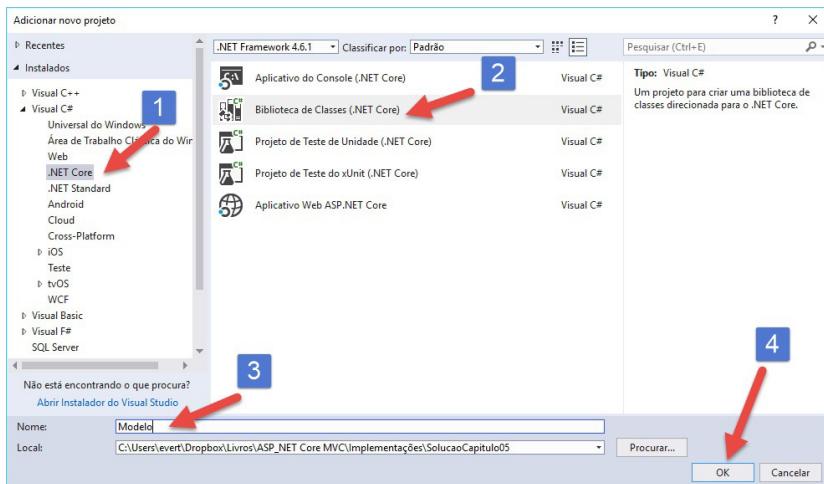


Figura 5.1: Janela para criação de projeto library para o modelo

No novo projeto, é criada uma classe de nome `Class1.cs`. Como criaremos as classes dentro de pastas (namespaces), apague o arquivo e crie uma pasta chamada `Cadastros`.

Como dito, particionaremos nosso modelo em áreas (namespaces para o C#). Como já temos classes criadas que realocaremos em pastas, mova as classes `Instituicao` e `Departamento` para a pasta `Cadastros` do novo projeto, e altere o seu namespace para `Modelo.Cadastros`. Se não conseguir movê-las, copie-as para os novos destinos, e depois as apague na origem.

Precisamos agora referenciar o novo projeto que representa o modelo de negócio da nossa aplicação, em nosso projeto web. Nele, clique com o botão direito em `Dependências` e, depois, em `Adicionar Referência...`. Na janela aberta, clique na categoria `Projetos` e, no lado central, marque `Modelo` e clique no botão

OK para concluir (figura a seguir).

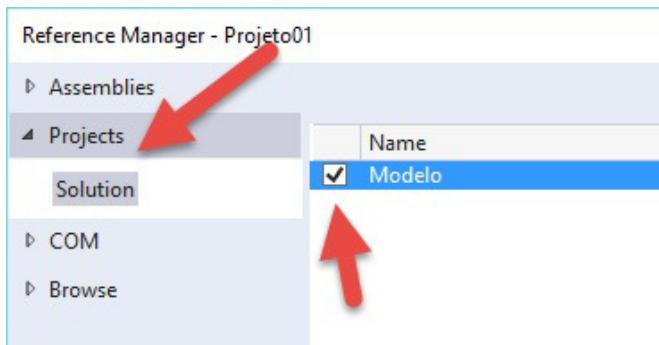


Figura 5.2: Adicionando a referência a um projeto da própria solução

Realize o build da solução. Diversos erros surgirão, pois nossas classes e visões faziam referência às classes que estavam na pasta `Models` do projeto da aplicação ASP.NET Core MVC. Mas agora estas não existem mais, já que foram movidas para o projeto `Modelo`. Vamos corrigir esses problemas informando o novo namespace para as classes, que é `Modelo.Cadastros`.

É preciso arrumar esse problema também nas visões. Esses erros só serão apontados quando sua aplicação for executada. Antecipe a correção, abrindo todas as visões e mudando a instrução `@model`.

Com estas poucas alterações, já podemos dizer que temos uma arquitetura – modesta, mas temos. Com a separação do modelo de negócio da aplicação, agora é possível criar uma aplicação mobile, desktop ou de serviços para o mesmo modelo.

5.3 CRIANDO A CAMADA DE PERSISTÊNCIA

EM UMA PASTA DA APLICAÇÃO

Criaremos agora as classes relacionadas à persistência dos objetos na base de dados. Para isso, na pasta `Data`, crie uma chamada `DAL` e, dentro dela, uma chamada `Cadastros`. Vamos retirar da classe `IESContext` o método sobreescrito que muda o nome da tabela, pois manteremos as tabelas mapeadas no plural.

Veja na sequência o código atualizado:

```
using Microsoft.EntityFrameworkCore;
using Modelo.Cadastros;

namespace Capitulo02.Data
{
    public class IESContext : DbContext
    {
        public IESContext(DbContextOptions<IESContext> options) : 
base(options)
        {

        }

        public DbSet<Departamento> Departamentos { get; set; }
        public DbSet<Instituicao> Instituicoes { get; set; }
    }
}
```

Inicialmente, vamos criar uma classe `DAL` para a classe `Instituicao` e `Departamento`. Todo trabalho relacionado à persistência se concentrará nelas. Na pasta `Cadastros`, crie a classe `InstituicaoDAL`. Veja no código a seguir como ela deve ficar.

Observe que o contexto é declarado no início da classe e, por enquanto, implementamos apenas o método que retorna todas as instituições classificadas (ordenadas) pelo nome. Note que o contexto será entregue por meio do construtor.

```

using Microsoft.EntityFrameworkCore;
using Modelo.Cadastros;
using System.Linq;
using System.Threading.Tasks;

namespace Capitulo02.Data.DAL.Cadastros
{
    public class InstituicaoDAL
    {
        private IESContext _context;

        public InstituicaoDAL(IISContext context)
        {
            _context = context;
        }

        public IQueryable<Instituicao> ObterInstituicoesClassificadasPorNome()
        {
            return _context.Instituicoes.OrderBy(b => b.Nome);
        }
    }
}

```

5.4 ADAPTAÇÃO DA CAMADA DE APLICAÇÃO

Com a criação da nossa arquitetura, é preciso agora adaptar os controladores. Vamos então ver a alteração do `InstituicaoController`, que deverá ter acesso a um objeto da classe `InstituicaoDAL`. Veja o código a seguir, já adaptado.

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Modelo.Cadastros;

namespace Capitulo02.Controllers
{
    public class InstituicaoController : Controller
    {

```

```
private readonly IESContext _context;
private readonly InstituicaoDAL instituicaoDAL;

public InstituicaoController(IESContext context)
{
    _context = context;
    instituicaoDAL = new InstituicaoDAL(context);
}
// Código omitido
```

A primeira action a ser alterada será a `Index`, mostrada a seguir. Veja o uso do método definido na classe de `DAL` para instituições.

```
public async Task<IActionResult> Index()
{
    return View(await instituicaoDAL.ObterInstituicoesClassificadasPorNome().ToListAsync());
}
```

A segunda implementação, que altera o controlador, resolverá um problema de redundância de código. Se observarmos as actions `GET Details`, `GET Edit` e `GET Delete`, a recuperação da instituição a ser retornada à visão é semelhante. Desta maneira, criaremos um método privado na própria classe para apenas usá-lo nas actions, resolvendo esse problema.

Entretanto, antes disso, precisamos criar um novo método para a classe `DAL`, que é a obtenção de uma instituição por meio de seu `id`. Veja o código do método para a classe `InstituicaoDAL` na sequência:

```
public async Task<Instituicao> ObterInstituicaoPorId(long id)
{
    return await _context.Instituicoes.Include(d => d.Departamentos).SingleOrDefaultAsync(m => m.InstituicaoID == id);
}
```

Com essa implementação realizada, podemos finalmente

implementar o método que será reutilizado pelas três actions. Veja-o na listagem a seguir. Observe a chamada ao método – que retornará a instituição solicitada pelo usuário – no corpo do action:

```
private async Task<IActionResult> ObterVisaoInstituicaoPorId(long? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instituicao = await instituicaoDAL.ObterInstituicaoPorId(
long) id);
    if (instituicao == null)
    {
        return NotFound();
    }

    return View(instituicao);
}
```

Agora, com o método criado, podemos usar as actions que precisam recuperar uma instituição e retorná-la à visão. Veja os seus códigos a seguir.

```
public async Task<IActionResult> Details(long? id)
{
    return await ObterVisaoInstituicaoPorId(id);
}

public async Task<IActionResult> Edit(long? id)
{
    return await ObterVisaoInstituicaoPorId(id);
}

public async Task<IActionResult> Delete(long? id)
{
    return await ObterVisaoInstituicaoPorId(id);
}
```

Com a implementação das actions `GET`, restam-nos agora as que respondem ao `HTTP POST`. A primeira que trabalharemos será a `Create`, referente à inserção de uma nova instituição. Para que ela seja atendida, precisamos implementar a funcionalidade no `DAL` e, depois, no controlador. Veja o código a seguir para a classe `InstituicaoDAL`.

```
public async Task<Instituicao> GravarInstituicao(Instituicao instituicao)
{
    if (instituicao.InstituicaoID == null)
    {
        _context.Instituicoes.Add(instituicao);
    }
    else
    {
        _context.Update(instituicao);
    }
    await _context.SaveChangesAsync();
    return instituicao;
}
```

Com essa implementação, precisamos alterar as actions, como veremos na sequência. Se você verificar o código, notará que a sua implementação é muito particular ao problema que resolve, não restando muita possibilidade de generalização. Por isso, optei por mantê-los como estão no código.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("Nome,Endereco")] Instituicao instituicao)
{
    try
    {
        if (ModelState.IsValid)
        {
            await instituicaoDAL.GravarInstituicao(instituicao);
            return RedirectToAction(nameof(Index));
        }
    }
```

```

        }
        catch (DbUpdateException)
        {
            ModelState.AddModelError("", "Não foi possível inserir os
dados.");
        }
        return View(instituicao);
    }

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(long? id, [Bind("Instituica
oID, Nome, Endereco")] Instituicao instituicao)
{
    if (id != instituicao.InstituicaoID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            await instituicaoDAL.GravarInstituicao(instituicao);
        }
        catch (DbUpdateConcurrencyException)
        {
            if (! await InstituicaoExists(instituicao.Instituicao
ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instituicao);
}

```

Para finalizar, vamos implementar a action POST para a remoção de um registro. Seguiremos a mesma lógica das actions

anteriores. Na sequência, está a implementação do método de remoção na classe `InstituicaoDAL` :

```
public async Task<Instituicao> EliminarInstituicaoPorId(long id)
{
    Instituicao instituicao = await ObterInstituicaoPorId(id);
    _context.Instituicoes.Remove(instituicao);
    await _context.SaveChangesAsync();
    return instituicao;
}
```

Seguindo o fluxo, vamos para a implementação da action. Observe que a única alteração é a retirada da remoção da instituição e a invocação do método da classe DAL :

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(long? id)
{
    var instituicao = await instituicaoDAL.EliminarInstituicaoPorId((long) id);
    TempData["Message"] = "Instituição " + instituicao.Nome.ToUpper() + " foi removida";
    return RedirectToAction(nameof(Index));
}
```

Para finalizar o controlador, precisamos implementar o método `InstituicaoExists()` para o código a seguir, que utiliza o método `ObterInstituicaoPorId()` do DAL para verificar a existência de uma determinada instituição. Veja o corpo do método e perceba que ele recebe um `id` para comprovar essa presença, retornando então um valor lógico (booleano) à tarefa.

```
private async Task<bool> InstituicaoExists(long? id)
{
    return await instituicaoDAL.ObterInstituicaoPorId((long) id)
!= null;
```

Para iniciar a sessão seguinte, você precisa realizar o build na

solução; certamente, erros aparecerão. Um deles refere-se ao controlador de departamentos, que tem o acesso a dados. Nele, será preciso realizar também as mudanças que fizemos no controlador de instituições.

Para facilitar, vamos conferir os códigos das classes alteradas, a começar pela `DepartamentoDAL`, e finalizando com a `DepartamentoController`. Não é necessário comentarmos sobre esses códigos, pois são semelhantes a tudo que fizemos para as instituições.

Veja a classe `DepartamentoDAL`:

```
using Microsoft.EntityFrameworkCore;
using Modelo.Cadastros;
using System.Linq;
using System.Threading.Tasks;

namespace Capitulo02.Data.DAL.Cadastros
{
    public class DepartamentoDAL
    {
        private IESContext _context = new IESContext();

        public IQueryable<Departamento> ObterDepartamentosClassificadosPorNome()
        {
            return _context.Departamentos.Include(i => i.Instituicao).OrderBy(b => b.Nome);
        }

        public async Task<Departamento> ObterDepartamentoPorId(long id)
        {
            var departamento = await _context.Departamentos.SingleOrDefaultAsync(m => m.DepartamentoID == id);
            _context.Instituicoes.Where(i => departamento.InstituicaoID == i.InstituicaoID).Load();
            return departamento;
        }
    }
}
```

```

        public async Task<Departamento> GravarDepartamento(Departamento departamento)
    {
        if (departamento.DepartamentoID == null)
        {
            _context.Departamentos.Add(departamento);
        }
        else
        {
            _context.Update(departamento);
        }
        await _context.SaveChangesAsync();
        return departamento;
    }

        public async Task<Departamento> EliminarDepartamentoPorId
(long id)
    {
        Departamento departamento = await ObterDepartamentoPorId(id);
        _context.Departamentos.Remove(departamento);
        await _context.SaveChangesAsync();
        return departamento;
    }
}
}

```

Veja a classe DepartamentoController :

```

using Capitulo02.Data;
using Capitulo02.Data.DAL.Cadastros;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using Modelo.Cadastros;
using System.Linq;
using System.Threading.Tasks;

namespace Capitulo02.Controllers
{
    public class DepartamentoController : Controller
    {
        private readonly IESContext _context;

```

```

private readonly DepartamentoDAL departamentoDAL;
private readonly InstituicaoDAL instituicaoDAL;

public DepartamentoController(IESContext context)
{
    _context = context;
    instituicaoDAL = new InstituicaoDAL(context);
    departamentoDAL = new DepartamentoDAL(context);
}

public async Task<IActionResult> Index()
{
    return View(await departamentoDAL.ObterDepartamentosC
lassificadosPorNome().ToListAsync());
}

public IActionResult Create()
{
    var instituicoes = instituicaoDAL.ObterInstituicoesCl
assificadasPorNome().ToList();
    instituicoes.Insert(0, new Instituicao() { Instituica
oID = 0, Nome = "Selecione a instituição" });
    ViewBag.Instituicoes = instituicoes;
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("Nome, Inst
ituicaoID")] Departamento departamento)
{
    try
    {
        if (ModelState.IsValid)
        {
            await departamentoDAL.GravarDepartamento(depa
rtamento);
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException)
    {
        ModelState.AddModelError("", "Não foi possível in
serir os dados.");
    }
}

```

```

        return View(departamento);
    }

    public async Task<IActionResult> Edit(long? id)
    {
        ViewResult visaoDepartamento = (ViewResult) await Obt
erVisaoDepartamentoPorId(id);
        Departamento departamento = (Departamento)visaoDepart
amento.Model;
        ViewBag.Instituicoes = new SelectList(instituicaoDAL.
ObterInstituicoesClassificadasPorNome(), "InstituicaoID", "Nome",
departamento.InstituicaoID);
        return visaoDepartamento;
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Edit(long? id, [Bind("De
partamentoID, Nome, InstituicaoID")] Departamento departamento)
    {
        if (id != departamento.DepartamentoID)
        {
            return NotFound();
        }

        if (ModelState.IsValid)
        {
            try
            {
                await departamentoDAL.GravarDepartamento(depa
rtamento);
            }
            catch (DbUpdateConcurrencyException)
            {
                if (! await DepartamentoExists(departamento.D
epartamentoID))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }
        }
    }
}

```

```

        return RedirectToAction(nameof(Index));
    }
    ViewBag.Instituicoes = new SelectList(instituicaoDAL.
ObterInstituicoesClassificadasPorNome(), "InstituicaoID", "Nome",
departamento.InstituicaoID);
    return View(departamento);
}

private async Task<bool> DepartamentoExists(long? id)
{
    return await departamentoDAL.ObterDepartamentoPorId((
long)id) != null;
}

public async Task<IActionResult> Details(long? id)
{
    return await ObterVisaoDepartamentoPorId(id);
}

public async Task<IActionResult> Delete(long? id)
{
    return await ObterVisaoDepartamentoPorId(id);
}

// POST: Instituicao/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(long? id
)
{
    var departamento = await departamentoDAL.EliminarDepa
rtamentoPorId((long) id);
    TempData["Message"] = "Departamento " + departamento.
Nome.ToUpper() + " foi removido";
    return RedirectToAction(nameof(Index));
}

private async Task<IActionResult> ObterVisaoDepartamentoP
orId(long? id)
{
    if (id == null)
    {
        return NotFound();
    }
}

```

```

        var departamento = await departamentoDAL.ObterDepartamentoPorId((long)id);
        if (departamento == null)
        {
            return NotFound();
        }

        return View(departamento);
    }
}
}

```

5.5 ADAPTANDO AS VISÕES PARA MINIMIZAR REDUNDÂNCIAS

Veremos que não é só o código C# que pode ser melhorado, mas as visões também. Se você olhar as `Details` e `Delete`, existe muito código redundante. Será nosso trabalho agora minimizar isso, usando o Partial Views, que vimos no capítulo anterior.

Para começar, vamos separar o conteúdo referente ao conteúdo dos Cards . Na pasta `Views` de `Instituicao` , crie uma visão chamada `_PartialDetailsContentCard.cshtml` . Nela, implemente o código a seguir. Este é o mesmo processo usado na visão `Details` de `Instituição` . Nele são codificados os controles que serão renderizados para a apresentação dos dados que serão visualizados.

```

@model Modelo.Cadastros.Instituicao

<div class="card-body">
    <div class="row">
        <div class="col-9">
            <div class="form-group">
                <label asp-for="InstituicaoID" class="control-label"></label>

```

```

<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-key" aria-hidden="true"></i>
    </span>
    <input asp-for="InstituicaoID" class="form-control" disabled="disabled" />
</div>
<label asp-for="Nome" class="control-label"></label>
<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-user-circle-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Nome" class="form-control" disabled="disabled" />
</div>
<label asp-for="Endereco" class="control-label"></label>
<br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-address-card-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Endereco" class="form-control" disabled="disabled" />
</div>
</div>
<div class="col-3">
    @Html.Partial("~/Views/Instituicao/_ComDepartamentos.cshtml",
    Model.Departamentos.ToList())
</div>
</div>
</div>

```

Agora, para utilizar essa Partial View na visão Details , substitua a <div> do corpo do card pelo código apresentado a seguir. Veja que todo o código que existia e que foi levado à Partial

View apresentada anteriormente foi substituída pela invocação da instrução `@Html.Partial()`. Esta recebe o arquivo da Partial View que será trazida para a visão.

```
@Html.Partial("~/Views/Instituicao/_PartialDetailsContentCard.cshtml", Model)
```

Para reutilizarmos essa Partial view, inclua-a na visão `Delete`, da mesma maneira como em `Details`. Para praticar, adapte as mesmas visões (`Details` e `Delete`) para `Departamentos`. Implemente a visão `_PartialDetailsContentCard.cshtml` e compare sua implementação com o código apresentado na sequência.

```
@model Modelo.Cadastros.Departamento

<div class="card-body">
    <div class="form-group">
        <label asp-for="DepartamentoID" class="control-label"></label>
        <br />
        <div class="input-group">
            <span class="input-group-addon">
                <i class="fa fa-key" aria-hidden="true"></i>
            </span>
            <input asp-for="DepartamentoID" class="form-control" disabled="disabled" />
        </div>
        <label asp-for="Nome" class="control-label"></label>
        <br />
        <div class="input-group">
            <span class="input-group-addon">
                <i class="fa fa-user-circle-o" aria-hidden="true"></i>
            </span>
            <input asp-for="Nome" class="form-control" disabled="disabled" />
        </div>
        <label asp-for="Instituicao.Nome" class="control-label"></label>
        <br />
```

```
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-user-circle-o" aria-hidden="true">
</i>
    </span>
    <input asp-for="Instituicao.Nome" class="form-control" disabled="disabled" />
</div>
</div>
</div>
```

Por fim, insira a chamada à Partial View criada anteriormente nas visões Details e Delete , na <div> do Card Body . O código está na sequência.

```
@Html.Partial("~/Views/Departamento/_PartialDetailsContentCard.cshtml", Model)
```

5.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

A separação de uma aplicação em camadas é uma necessidade cada vez mais constante nos projetos. Ela pode favorecer a modularização e o reúso, pois trabalha os pontos relacionados à coesão e ao acoplamento.

Este capítulo demonstrou como separar seu projeto em camadas, criando assim uma arquitetura. Foi um capítulo com mais código do que teoria, mas foi bom. No próximo, trabalharemos validações. Será muito legal.

CAPÍTULO 6

CODE FIRST MIGRATIONS, DATA ANNOTATIONS E VALIDAÇÕES

Quando implementamos o modelo de negócios, desde o capítulo *Acesso a dados com o Entity Framework Core*, trabalhamos com base de dados. Criamos uma tabela depois de outra, criamos associações e, então, para facilitar o mapeamento entre o modelo e o banco, apagávamos, criávamos e populávamos todo o banco a cada execução da aplicação.

Neste capítulo, a apresentação do Code First Migrations permitirá que mudanças realizadas no modelo sejam refletidas na base de dados, sem perda dos dados e sem ter de remover o banco.

Desde a criação da primeira visão, surgiu o tema *validação*, pois ela já estava preparada para exibir mensagens de erro relativas às regras de validação dos dados. O ASP.NET Core MVC oferece diversos atributos que podem marcar propriedades para diversos tipos de validações. Estes são implementados neste capítulo e, em conjunto com o Code First Migrations, aplicados na base de dados.

6.1 O USO DO CODE FIRST MIGRATIONS

A técnica que utilizamos até agora faz com que o banco seja eliminado e criado novamente, ao realizarmos mudanças nas classes que representam o modelo de negócio. Isso causa a perda de todos os dados que tínhamos registrados.

Este processo é ruim, pois precisamos sempre inserir elementos para nossos testes. Uma forma usada e subsidiada pelo Entity Framework Core é por meio do uso do *Code First Migrations*.

Desta maneira, precisamos habilitá-lo para o projeto. Para isso, selecione com o botão direito no nome do projeto da aplicação (o Core MVC) na janela de Gerenciador de Soluções e, em seguida, clique em Editar o projeto. No arquivo XML que aparece, verifique se os componentes da listagem a seguir se encontram nele; se não estiverem, insira-os.

```
<ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
</ItemGroup>
```

Para partirmos para um banco novo, abra a janela Pesquisador de Objetos do SQL Server (SQL Server Object Explorer), localize o banco e remova-o. Uma vez que o arquivo da base de dados esteja removido, clique com o botão direito do mouse sobre o nome do projeto e, então, em Abrir pasta no Gerenciador de Arquivos .

Depois, na caixa de endereço, digite cmd para acessar o prompt do console de comandos. Nesta janela, digite a instrução

`dotnet ef migrations add InitialCreate`. Se tudo correr bem, o console exibirá algo semelhante ao apresentado na sequência.

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\evert\AppData\Local\ASP.NET\DataProtection-Keys' as key repository and Windows DP API to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'IESContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

Com a execução da instrução anterior bem-sucedida, você poderá verificar no seu Gerenciador de Soluções a existência de uma nova pasta, chamada `Migrations`. Inicialmente, esta possui os arquivos necessários para a configuração da base de dados. Você pode abri-los e estudá-los, se quiser.

Precisamos agora retirar o código que executa a classe de inicialização do contexto. Esse código está na classe `Program`, apresentado na sequência. Comente-o ou retire-o. Fica a seu critério.

```
using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    try
    {
        var context = services.GetRequiredService<IESContext>();
        IESDbInitializer.Initialize(context);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "Um erro ocorreu ao popular a base de
```

```
dados.");
    }
}
```

Na pasta `Migrations`, no arquivo com o nome composto pela data e hora (*timestamp*) da configuração do `Migrations` e terminado com `InitialCreate`, existe uma classe com dois métodos: `Up()` e `Down()`. Estes criam a estrutura da base de dados e remove-a, respectivamente.

O segundo arquivo é uma representação do esquema da base de dados (*snapshot*) no momento da habilitação do `Migrations`. O EF Core faz uso desse arquivo para saber o que deve ser alterado na base de dados. Não elimine o arquivo do *timestamp* nem o de *snapshot*, pois eles trabalham em sincronismo.

Vamos usar uma dessas instruções para que a base possa ser criada e atualizada a partir desses arquivos gerados pelo `Migrations`. Na janela do console, execute `dotnet ef database update`. Se tudo ocorrer bem, você verá a execução das instruções relacionadas à criação das tabelas, bem como suas regras e índices. Verifique novamente no `Pesquisador de Objetos do SQL Server` a existência da base de dados.

6.2 ATUALIZAÇÃO DO MODELO DE NEGÓCIO

Agora, vamos popular nossa base de dados. Execute sua aplicação e insira algumas instituições e departamentos. Com alguns registros, vamos implementar novas classes para nosso modelo. Sendo assim, insira as classes `Curso` e `Disciplina` no projeto `modelo`, na pasta `Cadastros` – ambas com código

apresentado na sequência.

Conceitualmente, teríamos uma associação *muitos para muitos* entre essas duas novas classes, pois um curso pode ter várias disciplinas, e cada disciplina pode estar associada a vários cursos. Entretanto, no Entity Framework Core, quando temos essa situação (permitida em OO), ao mapear a associação para uma base relacional, o framework precisa de uma classe da associação.

Desta maneira, nas duas classes a seguir, observe que temos uma coleção para essa classe associativa. Esta é composta apenas pelas propriedades que mapeiam a relação. Na classe `Curso`, existe também o mapeamento para a associação com `Departamento`, pois cada curso está associado a um único departamento.

Veja a classe `Curso`:

```
using System.Collections.Generic;

namespace Modelo.Cadastros
{
    public class Curso
    {
        public long? CursoID { get; set; }
        public string Nome { get; set; }

        public long? DepartamentoID { get; set; }
        public Departamento Departamento { get; set; }

        public virtual ICollection<CursoDisciplina> CursosDisciplinas { get; set; }
    }
}
```

Veja a classe `Disciplina`:

```
using System.Collections.Generic;
```

```
namespace Modelo.Cadastros
{
    public class Disciplina
    {
        public long? DisciplinaID { get; set; }
        public string Nome { get; set; }

        public virtual ICollection<CursoDisciplina> CursosDisciplinas { get; set; }
    }
}
```

Veja a `CursoDisciplina`:

```
namespace Modelo.Cadastros
{
    public class CursoDisciplina
    {
        public long? CursoID { get; set; }
        public Curso Curso { get; set; }
        public long? DisciplinaID { get; set; }
        public Disciplina Disciplina { get; set; }
    }
}
```

Além desta, precisamos definir o relacionamento via código usando *Fluent API*. Isso será feito na classe `IEntityContext`, por meio da sobrescrita do método `OnModelCreating()`. Veja o código na sequência.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<CursoDisciplina>()
        .HasKey(cd => new { cd.CursoID, cd.DisciplinaID });

    modelBuilder.Entity<CursoDisciplina>()
        .HasOne(c => c.Curso)
        .WithMany(cd => cd.CursosDisciplinas)
        .HasForeignKey(c => c.CursoID);
```

```
modelBuilder.Entity<CursoDisciplina>()
    .HasOne(d => d.Disciplina)
    .WithMany(cd => cd.CursosDisciplinas)
    .HasForeignKey(d => d.DisciplinaID);
}
```

Como dito e visto anteriormente, um curso está ligado a um departamento. Logo, precisamos adaptar nossa classe de departamentos para que ela tenha a coleção de cursos associados a cada departamento. Veja o novo código para a classe Departamento :

```
using System.Collections.Generic;

namespace Modelo.Cadastros
{
    public class Departamento
    {
        public long? DepartamentoID { get; set; }
        public string Nome { get; set; }

        public long? InstituicaoID { get; set; }
        public Instituicao Instituicao { get; set; }

        public virtual ICollection<Curso> Cursos { get; set; }
    }
}
```

Na classe IESContext , precisamos informar o mapeamento dessas novas classes ao contexto. Para isso, insira as instruções a seguir na classe, logo após a declaração dos mapeamentos já existentes.

```
public DbSet<Curso> Cursos { get; set; }
public DbSet<Disciplina> Disciplinas { get; set; }
```

Agora, realize um build em sua solução e, no prompt do console, digite dotnet ef migrations add CursoDisciplina . Após a execução, verifique a criação do arquivo com instruções

que serão executadas para atualizar a base de dados, na sua pasta `Migrations`.

Para que essa atualização seja efetivada, execute a instrução `dotnet ef database update` no console. Então, verifique a criação das novas tabelas em sua base de dados.

No Entity Framework 6 (versão anterior à Core), o mapeamento desse tipo de associação era mais simples. Acredito que, no futuro, a tendência é melhorar essa situação em uma nova versão.

6.3 O USO DE VALIDAÇÕES

Na seção anterior, fizemos uso da Fluent API para configurar a chave primária e os relacionamentos na base de dados de uma classe mapeada para o banco relacional. Existem outras regras que podem ser aplicadas ao nosso modelo e mapeadas para a base de dados relacional.

Algumas delas são referentes a: tamanho máximo do campo, se o campo é de preenchimento obrigatório; e verificação, se os valores digitados fazem parte do domínio de caracteres possíveis para o campo em questão.

A aplicação dessas regras pode ser realizada ao utilizarmos a Fluent API, mas outra maneira de fazer isso é pelo uso de Data Annotations. Data Annotations são características aplicadas antes da definição de uma propriedade – como atributos (com o nome entre colchetes), que marcam o elemento com essas características.

Para a aplicação desse novo recurso, criaremos uma nova

classe, chamada `Academico`. Crie uma nova pasta chamada `Discente` no projeto `Modelo`, e nela crie a classe de acordo ao código a seguir:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace Modelo.Discente
{
    public class Academico
    {
        public long? AcademicoID { get; set; }

        [StringLength(10, MinimumLength = 10)]
        [RegularExpression("[0-9]{10}")]
        [Required]
        public string RegistroAcademico { get; set; }

        [Required]
        public string Nome { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:dd-MM-yyyy}")]
        [Required]
        public DateTime? Nascimento { get; set; }
    }
}
```

Observe que todas as propriedades possuem a anotação `[Required]`, tornando-as obrigatórias. No caso da propriedade `AcademicoID`, pelo fato de o seu nome seguir a convenção para definição de chave primária, não é necessário o uso da anotação, já que a chave será gerada automaticamente a cada novo objeto registrado na base de dados.

A propriedade `RegistroAcademico` tem ainda mais duas anotações, conhecidas também como atributos. Uma delas é a `[StringLength]`, que define o tamanho que o campo deverá ter na tabela; e ainda com ela, definimos também o comprimento

mínimo para a entrada de dados, por meio do `MinimumLength`. Já a anotação `[RegularExpression]` determina uma máscara que será atribuída no valor informado (no caso, apenas 10 números).

Por fim, as anotações `[DataType]` e `[DisplayFormat]` definem a propriedade como do tipo `Data`, bem como sua formatação para exibição, respectivamente. Elas também são uma informação obrigatória.

Vamos atualizar nossa base de dados. Para isso, insira o mapeamento da classe `Academico` na classe `IESContext`, por meio da declaração `public DbSet<Academico> Academicos { get; set; }`. No console, execute a instrução `dotnet ef migrations add Academico` e, em seguida, `dotnet ef database update`.

Precisamos agora criar a classe `DAL`, o controlador e as visões para a manipulação dos dados. Começamos pela classe `AcademicoDAL`, que tem seu código na sequência. Crie-a em uma pasta nova, chamada `Discente`. Todo o seu conteúdo já foi explicado quando trabalhamos com os `DALs` `Instituicao` e `Departamento`, portanto, não são necessárias explicações adicionais para o código.

```
using Modelo.Discente;
using System.Linq;
using System.Threading.Tasks;

namespace Capitulo02.Data.DAL.Discente
{
    public class AcademicoDAL
    {
        private IESContext _context;

        public AcademicoDAL(IESContext context)
        {
```

```

        _context = context;
    }

    public IQueryable<Academico> ObterAcademicosClassificados
PorNome()
{
    return _context.Academicos.OrderBy(b => b.Nome);
}

public async Task<Academico> ObterAcademicoPorId(long id)
{
    return await _context.Academicos.FindAsync(id);
}

public async Task<Academico> GravarAcademico(Academico academico)
{
    if (academico.AcademicoID == null)
    {
        _context.Academicos.Add(academico);
    }
    else
    {
        _context.Update(academico);
    }
    await _context.SaveChangesAsync();
    return academico;
}

public async Task<Academico> EliminarAcademicoPorId(long id)
{
    Academico academico = await ObterAcademicoPorId(id);
    _context.Academicos.Remove(academico);
    await _context.SaveChangesAsync();
    return academico;
}
}
}

```

Na sequência, precisamos da classe controladora e suas actions. Novamente, o código é semelhante ao que já fizemos. Seu código para o controlador de acadêmicos deve ficar como ao que segue:

```

using Capitulo02.Data;
using Capitulo02.Data.DAL.Discente;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Modelo.Discente;
using System.Threading.Tasks;

namespace Capitulo02.Controllers
{
    public class AcademicoController : Controller
    {
        private readonly IESContext _context;
        private readonly AcademicoDAL academicoDAL;

        public AcademicoController(IESContext context)
        {
            _context = context;
            academicoDAL = new AcademicoDAL(context);
        }

        public async Task<IActionResult> Index()
        {
            return View(await academicoDAL.ObterAcademicosClassificadosPorNome().ToListAsync());
        }

        private async Task<IActionResult> ObterVisaoAcademicoPorId(long? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var academico = await academicoDAL.ObterAcademicoPorId((long)id);
            if (academico == null)
            {
                return NotFound();
            }

            return View(academico);
        }

        public async Task<IActionResult> Details(long? id)

```

```

    {
        return await ObterVisaoAcademicoPorId(id);
    }

    public async Task<IActionResult> Edit(long? id)
    {
        return await ObterVisaoAcademicoPorId(id);
    }

    public async Task<IActionResult> Delete(long? id)
    {
        return await ObterVisaoAcademicoPorId(id);
    }

    // GET: Academico/Create
    public IActionResult Create()
    {
        return View();
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create([Bind("Nome,Regis
troAcademico,Nascimento")] Academico academico)
    {
        try
        {
            if (ModelState.IsValid)
            {
                await academicoDAL.GravarAcademico(academico)
;
                return RedirectToAction(nameof(Index));
            }
        }
        catch (DbUpdateException)
        {
            ModelState.AddModelError("", "Não foi possível in
serir os dados.");
        }
        return View(academico);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Edit(long? id, [Bind("Ac

```

```

        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> DeleteConfirmed(long? id)
    )
    {
        var academico = await academicoDAL.EliminarAcademicoPorId((long)id);
        TempData["Message"] = "Acadêmico " + academico.Nome.ToUpper() + " foi removida";
        return RedirectToAction(nameof(Index));
    }

    private async Task<bool> AcademicoExists(long? id)

```

```
        {
            return await academicoDAL.ObterAcademicoPorId((long)i
d) != null;
        }
    }
}
```

Precisamos agora adaptar nosso menu de opções, que está no layout das páginas, para que ofereça ao usuário a opção de acesso às funcionalidades referentes aos acadêmicos. Para isso, lá insira a opção tal qual mostra o código a seguir:

```
<a class="nav-item nav-link" asp-controller="Academico" asp-action="Index">
    
    Acadêmicos
</a>
```

Enfim, criaremos a visão Index para os acadêmicos, como o código na sequência. Siga os passos já trabalhados anteriormente para a criação de visões: clicar no nome da action no controlador com o botão direito do mouse, e escolher a opção de criar visão/exibição.

```
@model IEnumerable<Modelo.Discente.Academico>
 @{
    Layout = "_LayoutIES";
}

@section styles {
    <link rel="stylesheet" href="~/lib/datatables/media/css/jquery.dataTables.min.css" />
}

@if (@ TempData["Message"] != null)
{
    <div class="alert alert-success" role="alert">
        @ TempData["Message"]
    </div>
}
```

```

<div class="card-block">
    <div class="card-header text-white bg-primary text-center h1": Acadêmicos Registrados</div>
    <div class="card-body">
        <table id="tabela_academicos">
            <thead>
                <tr>
                    <th>
                        @Html.DisplayNameFor(model => model.Regis
troAcademico)
                    </th>
                    <th>
                        @Html.DisplayNameFor(model => model.Nome)
                    </th>
                    <th>
                        @Html.DisplayNameFor(model => model.Nasci
mento)
                    </th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Model)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Re
gistroAcademico)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.No
me)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Na
scimento)
                        </td>
                        <td>
                            <a asp-action="Edit" asp-route-id="@i
tem.AcademicoID">Edit</a> | <a asp-action="Details" asp-route-id=
"@item.AcademicoID">Details</a> | <a asp-action="Delete" asp-route-id=
"@item.AcademicoID">Delete</a>
                
```

```

        </td>
    </tr>
}
</tbody>
</table>
</div>
<div class="card-footer bg-success text-center">
    <a asp-action="Create" class="btn-success">Criar um novo
acadêmico</a>
</div>
</div>

@section ScriptPage {
    <script type="text/javascript" src "~/lib/datatables/media/js
/jquery.dataTables.min.js"></script>

    <script type="text/javascript">
        $(document).ready(function () {
            $('#tabela_academicos').DataTable({
                "order": [[1, "asc"]]
            });
        });
    </script>
}

```

Pronto, agora vamos criar a visão para testar as Data Annotations que utilizamos. Crie a visão Create tal qual o código seguinte. Observe que ela é semelhante às visões Create , criadas anteriormente. Não há necessidade de implementações específicas para isso, pois elas já foram feitas na classe de negócio.

```

@model Modelo.Discente.Academico
 @{
    Layout = "_LayoutIES";
}

<div class="card-block">
    <div class="card-header text-white bg-danger text-center h1">
        Registrando um novo acadêmico</div>
    <div class="card-body">
        <form asp-action="Create">

```

```

        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="RegistroAcademico" class="control-label"></label>
                    <input asp-for="RegistroAcademico" class="form-control" />
                    <span asp-validation-for="RegistroAcademico" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Nome" class="control-label"></label>
                <input asp-for="Nome" class="form-control" />
                <span asp-validation-for="Nome" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Nascimento" class="control-label"></label>
                <input asp-for="Nascimento" class="form-control" />
                <span asp-validation-for="Nascimento" class="text-danger"></span>
            </div>
            <div class="form-group text-center h3">
                <input type="submit" value="Registrar Acadêmico" class="btn btn-light" />
                <a asp-action="Index" class="btn btn-info">Retornar à listagem de acadêmicos</a>
            </div>
        </form>
    </div>
    <div class="card-footer bg-dark text-center text-white">
        Informe os dados acima e/ou clique em um dos botões de ação
    </div>
</div>

@section ScriptPage {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}

```

Então, vamos ao teste. Execute sua aplicação e acesse a visão de criação de um novo acadêmico. Quando ela for exibida, clique diretamente no botão que deverá fazer um novo registro. Como as informações não foram fornecidas, erros aparecerão, conforme pode ser comprovado pela figura:

Registrando um novo acadêmico

RegistroAcademico

The RegistroAcademico field is required.

Nome

The Nome field is required.

Nascimento

dd/mm/aaaa

The Nascimento field is required.

[Registrar Acadêmico](#) [Retornar à listagem de acadêmicos](#)

Figura 6.1: Erros gerados pelo uso de Data Annotations

Nas visões que implementamos a funcionalidade de acadêmicos, o nome da propriedade `RegistroAcademico` sempre é exibido: são duas palavras, que aparecem juntas e sem acento. Poderíamos melhorar isso e fazer com que seja exibido `RA`, que é mais comum.

O Data Annotations também permite a configuração desta funcionalidade. Veja na sequência o novo código para as propriedades `AcademicoID` e `RegistroAcademico` da classe `Acadêmico`. Observe os atributos `[DisplayName()]`. Quando a visão fizer uso do nome da propriedade, será utilizado o texto informado no atributo, e não mais o nome da propriedade.

```
[DisplayName("Id")]
public long? AcademicoID { get; set; }
```

```
[StringLength(10, MinimumLength = 10)]
[RegularExpression("([0-9]{10})")]
[Required]
[DisplayName("RA")]
public string RegistroAcademico { get; set; }
```

Não implementei aqui neste capítulo todas as visões relacionadas ao CRUD, pois elas são semelhantes às que fizemos nos capítulos anteriores. Deixo isso como desafio para você.

6.4 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Durante o processo de desenvolvimento de uma aplicação, a atualização no modelo de negócio pode ser necessária em algumas etapas. Fazendo uso do Entity Framework Core, essa atualização é refletida na base de dados. Ela precisa acontecer sem que perdemos os dados existentes na base, e vimos isso neste capítulo, por meio do Code First Migrations.

Ainda em relação ao modelo de negócio, existem certas propriedades que precisam ser validadas. Com os atributos do Data Annotations, foi possível realizar isso.

Foi um curto capítulo, mas interessante. No próximo, trabalharemos o controle de usuários e seus acessos à aplicação.

CAPÍTULO 7

AREAS, AUTENTICAÇÃO E AUTORIZAÇÃO

Em uma aplicação grande, é possível existirem diversos controladores, e não apenas os poucos que apresentei neste livro. Estes, por sua vez, podem ter diversas visões associadas a eles. Desta maneira, a estrutura básica oferecida pelo framework pode ser ineficaz no que se diz respeito à organização.

Para estes projetos, o framework traz o conceito de *Areas*, que podem ser vistas como unidades (ou módulos) de um projeto. Trabalhar com elas é extremamente simples, como veremos neste capítulo.

Sempre que uma aplicação é desenvolvida, é preciso se preocupar com a segurança relacionada ao acesso de usuários a ela. Este controle pode ser implementado pelo processo de autenticação e de autorização, que também serão apresentados aqui.

7.1 AREAS

A criação de áreas no ASP.NET MVC 5 (anterior ao Core) era um processo simples. O IDE fazia todo o trabalho burocrático,

mas, na versão Core, nós mesmos temos de fazer parte da criação. No projeto da nossa aplicação, vamos criar uma pasta chamada `Areas` . Clique com o botão direito nessa pasta e, então, em `Adicionar -> Area` .

Em nosso projeto, implementaremos duas Areas, chamadas de: `Cadastros` e `Discente` . Normalmente, são criadas quatro pastas dentro de cada área: `Controllers` , `Data` , `Models` e `Views` ; entretanto, em nosso caso, criaremos apenas as `Controllers` e `Views` . Um arquivo com orientações será aberto após a criação de cada área. Já falaremos sobre ele.

Agora, traga seus controladores e suas visões para cada área. Nos controladores, será preciso acertar os nomes dos namespaces para: `Capitulo02.Areas.Cadastros.Controllers` e `Capitulo02.Areas.Discente.Controllers` . Isso porque mudamos a organização física e, neste livro, adotei manter uma organização igual à organização lógica de namespaces. Mas você pode ficar à vontade se quiser aplicar uma diferente.

Por característica do ASP.NET Core MVC, precisamos também fazer uso de atributos nas classes dos controladores. Veja na sequência o código para cada situação:

```
namespace Capitulo02.Areas.Cadastros.Controllers  
{  
    [Area("Cadastros")]  
    public class InstituicaoController : Controller
```

```
[Area("Cadastros")]  
public class DepartamentoController : Controller
```

```
[Area("Discente")]  
public class AcademicoController : Controller
```

No início do livro, apresentei as rotas e seu uso na invocação de recursos da aplicação. Agora, com o uso de áreas, precisamos adicionar a informação de que faremos uso delas na classe `Startup`, do método `Configure()`. Para isso, antes da rota já definida, insira a do código seguinte:

```
routes.MapRoute(  
    name: "areaRoute",  
    template: "{area:exists}/{controller}/{action=Index}/{id?}");
```

Então, para acessarmos os serviços com essa nova configuração, precisamos adicionar, em nosso template, as áreas que estamos utilizando nas tags que geram os links: `asp-area="Cadastros"` e `asp-area="Discente"`.

Para minha aplicação funcionar, precisei copiar o arquivo `_ViewImports.cshtml` – que está na pasta `Views` da aplicação – na pasta `Views` de cada Area. Faça essa cópia e teste sua aplicação para ver se ela funcionará sem problemas nos acessos às visões. Veja a seguir o trecho de código que foi alterado.

```
<nav class="nav flex-column">  
    <a class="nav-item nav-link" asp-area="Cadastros" asp-control  
    ler="Instituicao" asp-action="Index">  
        <img src "~/images/university.png" width="30" height="30"  
        alt="">  
            Instituições  
    </a>  
    <a class="nav-item nav-link" asp-area="Cadastros" asp-control  
    ler="Departamento" asp-action="Index">  
        <img src "~/images/department.png" width="30" height="30"  
        alt="">  
            Departamentos  
    </a>  
    <a class="nav-item nav-link" asp-area="Discente" asp-controll  
    er="Academico" asp-action="Index">  
        <img src "~/images/academic.png" width="30" height="30" a  
        lt="">  
            Acadêmicos
```

```
</a>
</nav>
```

Para finalizar as alterações, precisamos alterar as visões que usam Partial View. Por exemplo, a visão Details de Instituição deverá estar igual a `@Html.Partial("~/Areas/Cadastros/Views/Instituicao/_PartialEditContentPanel.cshtml", Model)`. Note a inserção da Area na URL. Esta alteração é necessária também na visão Delete. Lembre-se de fazer o mesmo para Departamento.

Você precisará atualizar o caminho de inserção da Partial View `_ComDepartamentos.cshtml` para o novo caminho, agora com as áreas. O caminho correto para ela é `@Html.Partial("~/Areas/Cadastros/Views/Instituicao/_ComDepartamentos.cshtml", Model.Departamentos.ToList())`. Essa atualização deve ser realizada na Partial View `_PartialDetailsContentCard.cshtml`.

Se você precisar usar um controlador de outra Area em algum link, além de informar o `asp-area`, será preciso informar o `asp-controller`. Lembre-se disso. Como estamos usando actions de controladores de Areas iguais, isso não se faz necessário.

Com todas as alterações para que as Areas funcionem, restamos testar nossa aplicação. Execute-a e invoque a visão Create de `Instituicao`. Em minha máquina, a URL é <http://localhost:64867/Cadastros/Instituicao/Create>. Verifique se a renderização da visão ocorre de maneira correta.

7.2 SEGURANÇA EM APLICAÇÕES ASP.NET MVC

Toda aplicação precisa estar segura, principalmente quando se diz respeito ao controle de acesso de usuários a ela. Quando um usuário acessa uma aplicação, esta precisa saber quem está se conectando e o que esse usuário tem direito a acessar.

Um recurso pode estar disponível apenas para um grupo de usuários, e os que não fazem parte desse grupo não podem acessá-lo. Um exemplo seria o **Lançamento de Notas**, em que o acesso seria dado apenas aos professores. Desta maneira, apresentarei aqui os conceitos necessários para a implementação dessa segurança.

Autenticação e autorização

De maneira simplista, pode-se dizer que um sistema está seguro se ele garante o acesso apenas a usuários autenticados a recursos autorizados. Com esta frase anterior, busquei definir *autenticação*, que é o processo de validar que um usuário possui direitos ao acesso de uma aplicação. Já *autorização* é o processo de verificar se ele **já autenticado** possui direitos ao recurso requisitado, como registrar um novo departamento, por exemplo.

ASP.NET Core Identity

O ASP.NET Core Identity é um sistema de registro de usuários que permite a adição de funcionalidades relacionadas ao login em suas aplicações. Ao usuário, é permitido acessar o sistema por meio do par usuário/senha, ou usar provedores externos de autenticação, como Facebook, Google e Conta Microsoft, dentre outros. É possível configurar para que seja utilizado o SQL Server como armazenamento dos dados relativos a usuários, ou ainda

mecanismos de persistência próprios.

Configuração para o uso

Nosso primeiro passo é criar uma classe que estenda `IdentityUser`, pois, conforme dito no início da seção, é preciso saber quem está acessando a aplicação. Essa classe terá a responsabilidade de informar isso.

Assim, na pasta `Models` da aplicação, crie outra chamada `Infra` e, dentro dela, crie uma classe chamada `UsuarioDaAplicacao`, com o código que segue. Veja que não sobrescrevemos nada.

```
using Microsoft.AspNetCore.Identity;

namespace Capitulo02.Models.Infra
{
    public class UsuarioDaAplicacao : IdentityUser
    {
    }
}
```

Com a criação deste modelo de dados específico do Identity, precisamos mudar nossa classe de contexto, que está estendendo `DbContext`. Agora, para o uso do ASP.NET Core Identity, ela precisa estender `IdentityDbContext<UsuarioDaAplicacao>`. Veja o cabeçalho da classe na sequência.

```
public class IESContext : IdentityDbContext<UsuarioDaAplicacao>
```

Precisamos configurar nossa classe `Startup` para registrar o ASP.NET Core Identity e também receber o serviço por Injeção de Dependência (DI). Sendo assim, no método `ConfigureServices`, insira o código a seguir. O método `ConfigureApplicationCookie()` registra um cookie,

transmitido entre requisições, para a recuperação de usuários autenticados.

```
services.AddIdentity<UsuarioDaAplicacao, IdentityRole>()
    .AddEntityFrameworkStores<IESContext>()
    .AddDefaultTokenProviders();

services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Infra/Acessar";
    options.AccessDeniedPath = "/Infra/AcessoNegado";
});
```

O ASP.NET Core é disponibilizado para a aplicação por meio da chamada ao método `UseAuthentication()`, implementado antes do `app.UseMvc()`, no método `Configure()` da classe `Startup`. Implemente-a, tal qual mostra o código a seguir.

```
app.UseAuthentication();
```

Precisamos agora determinar o que os usuários podem acessar, isto é, o nosso controle de acesso a recursos. Isso pode ser implementado diretamente nos controladores e também em actions. Para este exemplo, escolhi a implementação diretamente no controlador de instituição.

Desta maneira, antes da definição da classe do controlador, insira o atributo `[Authorize]`, como é mostrado no código seguinte. O uso desse atributo determina que todas as actions só podem ser acessadas por um usuário que tenha se autenticado na aplicação.

É possível que você anote apenas determinadas actions com o `[Authorize]`, e não todo o controlador. Também é possível aplicar políticas específicas para autorização, como no uso de `[Authorize(Roles = "Docentes")]`, que restringe o acesso

apenas a usuários que têm o papel Docentes . Ainda, uma vez anotado o controlador com regras de autorização, caso você tenha uma action que pode ser acessada por qualquer usuário (mesmo sem ser autenticado), é possível anotá-la com [AllowAnonymous] .

```
[Area("Cadastros")]
[Authorize]
public class InstituicaoController : Controller
```

Vamos testar. Execute sua aplicação e tente acessar a action Index de Instituicao . Você verá que nada é exibido e, na URL, aparece algo semelhante a <http://localhost:64867/Infra/Acessar?ReturnUrl=%2FCadastros%2FInstituicao>. Nada foi renderizado.

Na URL, perceba que a aplicação foi redirecionada ao endereço que definimos na configuração, /Infra/Acessar , mas não temos essa action implementada ainda, então, está tudo bem. Ainda nela, vemos que o endereço que tentamos acessar foi enviado por meio de uma QueryString , para que aconteça o redirecionamento ao recurso desejado, após a autenticação ocorrer. Uma QueryString refere-se aos valores da URL que estão informados após o ponto de interrogação.

7.3 CRIAÇÃO DE UM ACESSO AUTENTICADO

Precisamos agora oferecer ao usuário recursos para que ele se autentique na aplicação, pois estamos limitando o acesso a usuários autenticados. Então, vamos implementar a action Acessar no controlador InfraController , um novo controlador para a aplicação que você precisa criar, tal qual

criamos os demais durante o livro.

Veja no código seguinte que já são trazidas algumas declarações que serão utilizadas pelas actions. Também é implementado o construtor para esse controller, que recebe argumentos por Injeção de Dependência.

O tipo de dados `UserManager` será responsável por ações relacionadas ao gerenciamento de usuário, como a sua criação. O `SignInManager` será o responsável por registrar o acesso do usuário à aplicação, e o `ILogger`, responsável por registrar mensagens de Log e exibi-las no console. Veja que todos estes objetos são recebidos no construtor.

Por segurança, o método da action `Acessar` realiza a operação de saída de algum usuário que possa estar autenticado. Após isso, registra a URL recebida no `ViewData` como argumento, por meio da `Querystring`, para então a visão `Acessar` ser renderizada.

Veja o atributo `[AllowAnonymous]` na action. Isso é necessário pelo fato de o usuário poder fazer login, sem estar autenticado.

```
using Capitulo02.Models.Infra;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace Capitulo02.Controllers
{
    [Authorize]
    public class InfraController : Controller
```

```

    {
        private readonly UserManager<UsuarioDaAplicacao> _userManager;
        private readonly SignInManager<UsuarioDaAplicacao> _signInManager;
        private readonly ILogger _logger;

        public InfraController(
            UserManager<UsuarioDaAplicacao> userManager,
            SignInManager<UsuarioDaAplicacao> signInManager,
            ILogger<InfraController> logger)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _logger = logger;
        }

        [HttpGet]
        [AllowAnonymous]
        public async Task<IActionResult> Acessar(string returnUrl
= null)
        {
            await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);

            ViewData["ReturnUrl"] = returnUrl;
            return View();
        }
    }
}

```

Precisamos criar a visão para a action `Acessar`. Mas, antes disso, é necessário criar o modelo que será usado para a visão. Note que este não é de negócio, logo, vamos criá-lo na pasta `Models` da aplicação, na pasta `Infra`. Este tipo de modelo é conhecido por modelos de visão (ou `View Model`), e vamos utilizar isso como sufixo para o nome da classe. Veja o seu código na sequência.

```

using System.ComponentModel.DataAnnotations;
namespace Capitulo02.Models.Infra
{

```

```
public class AcessarViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Senha { get; set; }

    [Display(Name = "Lembrar de mim?")]
    public bool LembrarDeMim { get; set; }
}
```

No código, observe as anotações novas, como a `[EmailAddress]`, que validará se o valor informado obedece à máscara básica para um endereço de e-mail. Também há um novo `DataType`, o `Password`. As propriedades definidas para o usuário são as básicas exigidas pelo ASP.NET Core Identity.

O e-mail será a chave usada para localizar o usuário, e a senha seguirá o padrão básico de exigir ao menos uma letra maiúscula e, ao menos, um número. Como teste, é comum usar a senha `P@ssw0rd`, e será ela que usaremos sempre que necessário.

Agora, vamos implementar a visão `Acessar`. Na sequência, veja o código para essa visão. Verifique na tag `<form>` a definição do elemento `asp-route-returnurl`. Ela conterá o endereço que o usuário tentou utilizar e que foi negado, por não estar autenticado.

Com essa informação, após o sucesso na autenticação, o usuário será redirecionado para esse endereço. Também é utilizado no código um `CheckBox`, no qual o usuário marcará (ou não) se deseja ser lembrado pela aplicação. Essa visão deverá ser criada em

uma pasta chamada `Infra`, dentro da pasta `Views` da aplicação.

```
@model Capitulo02.Models.Infra.AcessarViewModel

{@
    Layout = "_LayoutIES";
}

<div class="card-header text-white bg-secondary text-center h1">Utilize uma conta local para acessar</div>

<div class="row">
    <div class="col-md-4">
    </div>
    <div class="col-md-4">
        <section>
            <form asp-route-returnurl="@ViewData["ReturnUrl"]" method="post">
                <div asp-validation-summary="All" class="text-danger"></div>
                <div class="form-group">
                    <label asp-for="Email"></label>
                    <input asp-for="Email" class="form-control" />
                    <span asp-validation-for="Email" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Senha"></label>
                    <input asp-for="Senha" class="form-control" />
                    <span asp-validation-for="Senha" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <div class="checkbox">
                        <label asp-for="LembrarDeMim">
                            <input asp-for="LembrarDeMim" />
                            @Html.DisplayNameFor(m => m.LembrarDeMim)
                        </label>
                    </div>
                </div>
                <div class="form-group">
```

```

        <button type="submit" class="btn btn-default">Acessar</button>
    </div>
    <div class="form-group">
        <p>
            <a asp-action="RegistrarNovoUsuario"
asp-route-returnurl="@ ViewData["ReturnUrl"]">Registrar um novo usuário?</a>
        </p>
    </div>
    </form>
</section>
</div>
<div class="col-md-4">
</div>
</div>

@section ScriptPage {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}

```

Fico devendo aqui o método efetivo que registrará o acesso do usuário. Explico-o mais à frente no capítulo, pois agora precisaremos criar um usuário, uma vez que a base de dados está vazia.

7.4 REGISTRO DE UM NOVO USUÁRIO

Execute sua aplicação e procure acessar o menu de `Instituições`. Agora, é para aparecer uma página solicitando o nome do usuário e sua senha. Porém, ainda não temos nenhum. Por isso, colocamos opções para registrar um novo usuário na visão, e é isso que faremos aqui. Depois, retornaremos para testar o acesso.

No controlador de `Infra`, vamos criar a action `RegistrarNovoUsuario`, tal qual o código a seguir. Ele é simples

e dispensa explicação, mas note que a action recebe a URL que o usuário tentou acessar. Caso ela não venha, ele recebe `null`.

```
[HttpGet]
[AllowAnonymous]
public IActionResult RegistrarNovoUsuario(string returnUrl = null
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}
```

Assim como foi feito para a visão `Acessar`, a `RegistrarNovoUsuario` (que pedirá dados ao registro) fará uso de um modelo de visão. Desta maneira, na pasta `Infra`, crie a classe `RegistrarNovoUsuarioViewModel`, tal qual o código apresentado na sequência. Observe o uso de mensagens de erro personalizadas nos atributos.

```
using System.ComponentModel.DataAnnotations;

namespace Capitulo02.Models.Infra
{
    public class RegistrarNovoUsuarioViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "A {0} precisa ter ao m
enos {2} e no máximo {1} caracteres de cumprimento.", MinimumLeng
th = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Senha")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirmar senha")]
        [Compare("Password", ErrorMessage = "Os valores informado
os não são iguais")]
    }
}
```

```

    s para SENHA e CONFIRMAÇÃO não são iguais."]
        public string ConfirmPassword { get; set; }
    }
}

```

Agora podemos criar a visão para o registro de usuário. Veja seu código e observe que não há nada de novo nele:

```

@model Capitulo02.Models.Infra.RegistrarNovoUsuarioViewModel
 @{
     Layout = "_LayoutIES";
 }

<div class="card-header text-white bg-secondary text-center h1">R
egistrar uma nova conta de usuário</div>

<div class="row">
    <div class="col-md-4">
        </div>

        <div class="col-md-4">
            <form asp-route-returnUrl="@ViewData["ReturnUrl"]" method:
"post">
                <div asp-validation-summary="All" class="text-danger">
                    <div class="form-group">
                        <label asp-for="Email"></label>
                        <input asp-for="Email" class="form-control" />
                        <span asp-validation-for="Email" class="text-dang
er"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="Password"></label>
                        <input asp-for="Password" class="form-control" />
                        <span asp-validation-for="Password" class="text-d
anger"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="ConfirmPassword"></label>
                        <input asp-for="ConfirmPassword" class="form-cont
rol" />
                        <span asp-validation-for="ConfirmPassword" class=
"text-danger"></span>
                    </div>
    </div>
</div>

```

```

        <button type="submit" class="btn btn-default">Registrar</button>
    </form>
</div>

<div class="col-md-4">
</div>
</div>

@section ScriptPage {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}

```

Para que o usuário seja registrado, é necessário que implementemos a action (agora POST) para essa visão. Veja o código na sequência.

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> RegistrarNovoUsuario(RegistrarNovoUsuarioViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new UsuarioDaAplicacao { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("Usuário criou uma nova conta com senha.");
            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);

            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation("Usuário acesso com a conta criada.");
            return RedirectToAction(returnUrl);
        }
    }
}

```

```
        AddErrors(result);
    }
    return View(model);
}
```

Dois métodos são requisitados no código anterior, o `RedirectToLocal()` e o `AddErrors()`. Estes são exclusivos para esse controlador. Desta maneira, serão privados e estarão definidos na listagem a seguir.

O método `RedirectToLocal()` redireciona a requisição do usuário para uma determinada URL, que ele recebe como argumento. Já o `AddErrors()` adiciona um erro que poderá ser apresentado ao usuário na visão. Implemente-os em seu controlador também.

```
private void AddErrors(IdentityResult result)
{
    foreach (var error in result.Errors)
    {
        ModelState.AddModelError(string.Empty, error.Description)
    }
}

private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Ho
me");
    }
}
```

Agora, já que a ação referente ao registro de um novo usuário será realizada em tabelas na base de dados, e essas tabelas são

referentes ao ASP.NET Identity Core, precisamos executar o Migrations para que elas sejam criadas na base de dados. Desta maneira, acesse o console na pasta do projeto e execute as instruções: `dotnet ef migrations add Identity` e `dotnet ef database update`. Seja curioso e veja as tabelas criadas na base de dados.

Com as tabelas do Identity feitas, na página de acesso que você testou anteriormente, clique no link para registrar um novo usuário. Informe os dados solicitados (lembrando da regra de senha que falei), e confirme o registro do usuário desejado. Se tudo der certo, você agora consegue visualizar a visão `Index` de `Instituições`, pois o método que registra o novo usuário já realiza a sua autenticação.

7.5 USUÁRIO AUTENTICADO E O SEU LOGOUT

Ao realizar o login na aplicação, é interessante que as visões apresentem qual é o nome do usuário que está autenticado. Da mesma maneira e importância, o usuário precisa ter acesso a uma opção que registre sua saída da aplicação (logout) e, quando não estiver autenticado, seja oferecido um link para a página de autenticação.

Optei por ter estas informações definidas no `_LayoutIES.cshtml`, antes das tags de navegação. Veja na sequência o código.

Vamos usar um método chamado `GetUserName()`, pertencente à classe `UserManager`, que já será apresentada (assim

como a `SignInManager`). Esse método é chamado quando o `IsSignedIn()` retornar verdadeiro na expressão `if()` , ou seja, quando o usuário estiver autenticado corretamente.

Note que faremos uso da HTML Helper `ActionLink()` , pois a action `Logout` estará definida no controlador `Infra` , que está fora do contexto de áreas que criamos.

```
<div class="row" style="text-align:right;">
    <div class="col-6"></div>
    <div class="col-6">
        @if (SignInManager.IsSignedIn(User))
        {
            <div>Olá @UserManager.GetUserName(User), seja bem-vindo(a) / @Html.ActionLink("Logout", "Sair", "Infra", new { area = "" })</div>
        }
        else
        {
            <a class="nav-item nav-link" asp-area="" asp-controller="Infra" asp-action="Acessar">
                Informar usuário/senha para acessar
            </a>
        }
    </div>
</div>
```

Como dito na explicação do código anterior, temos duas classes: `SignInManager` e `UserManager` . A `SignInManager` é responsável pelo gerenciamento de acesso autenticado. O método `IsSignedIn()` retorna o status da verificação se o usuário enviado como parâmetro está ou não autenticado.

A classe `UserManager` é responsável pelo gerenciamento de usuários, e o método `GetUserName()` retorna o nome do usuário informado como parâmetro. A variável `User` , utilizada nas duas situações, contém informações do usuário autenticado na aplicação. Para o uso dessas classes na visão, precisei inserir duas

instruções de injeção em seu início, apresentadas na sequência:

```
@inject Microsoft.AspNetCore.Identity.SignInManager<Capitulo02.Models.Infra.UsuarioDaAplicacao> SignInManager  
@inject Microsoft.AspNetCore.Identity.UserManager<Capitulo02.Models.Infra.UsuarioDaAplicacao> UserManager
```

Execute sua aplicação. O ideal é sua página de instituições apresentar os dados do usuário conectado e um link para logout, tal qual mostra a figura a seguir.



Figura 7.1: Visualização de usuário autenticado

Primeiramente, é preciso implementar a action para o Logout . Veja-a na sequência. Note que ela chama Sair . O método executa a operação do Sing out para o usuário autenticado, imprime uma informação no log da aplicação e redireciona a aplicação para a sua action inicial.

```
[HttpGet]  
public async Task<IActionResult> Sair()  
{  
    await _signInManager.SignOutAsync();  
    _logger.LogInformation("Usuário realizou logout.");  
    return RedirectToAction(nameof(HomeController.Index), "Home")  
};  
}
```

Já temos a action GET para esse link, que é a Acessar , como também a visão. Falta a action POST e, como foi prometido anteriormente, ela está apresentada no código a seguir. Você pode ver que a parte de acesso é semelhante a que utilizamos na action de registrar um novo usuário.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Acessar(AcessarViewModel model,
string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Senha, model.LembrarDeMim, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation("Usuário Autenticado.");
            return RedirectToAction(returnUrl);
        }
    }
    ModelState.AddModelError(string.Empty, "Faha na tentativa de login.");
    return View(model);
}
```

Execute sua aplicação, autentique-se nela, clique no link Logout e, depois, veja que o texto e o link apresentados ao usuário são diferentes, tal qual pode ser visto na instrução anteriormente apresentada para a visão _LayoutIES .

Com o texto, você já tem o controle de acesso implementado. O que você poderia pensar agora é em uma evolução para essa situação, fazendo uso de papéis e realizando manutenções nos dados do usuário. Para isso, deixo na sequência alguns links que poderão auxiliá-lo nesta melhora.

RECOMENDAÇÕES DE LEITURA

Artigo *ASP.NET Core MVC: Authentication And Role Based Authorization With ASP.NET Core Identity*, de Sandeep Shekhawat

(<https://social.technet.microsoft.com/wiki/contents/articles/36804.asp-net-core-mvc-authentication-and-role-based-authorization-with-asp-net-core-identity.aspx>) – Trabalha um exemplo de gerenciamento de usuários e seus papéis.

Artigo *Create an ASP.NET Core app with user data protected by authorization*, de Rick Anderson e Joe Audette (<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/secure-data>) –

Apresenta um controle de acesso com autorização baseada em três níveis de acesso.

Artigo *Authorization in ASP.NET Core: Simple, role, claims-based, and custom* (<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/>) – Exibe uma relação de links com informações referentes ao processo de autenticação/ autorização.

Artigo *Role based Authorization* (<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles>) – Traz o uso de papéis para implementar a política de autorização a funcionalidades.

7.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Neste capítulo, foi possível conhecer uma maneira de organizar a lógica e a física de seu projeto, por meio de Areas. Em relação ao controle de acesso, foram apresentados os conceitos de autenticação e de autorização.

Inserimos implementações para criação e manutenção de usuários. Também foi exemplificado o controle de quais recursos estão disponíveis a um usuário autenticado. O capítulo apresentou ainda recursos para direcionar o usuário para a página de login e para a realização de logout.

O próximo capítulo trabalhará funcionalidades como uploads de arquivos binários, como imagens, downloads de arquivos e uma introdução ao tratamento de erros.

CAPÍTULO 8

UPLOADS, DOWNLOADS E ERROS

Em aplicações desenvolvidas para a internet, é comum ver o envio de imagens, que podem ou não ser exibidas aos usuários. Com o processo de digitalização de documentos cada vez mais utilizado, o envio de arquivos diferentes de imagem também é algo corriqueiro.

Este curto capítulo traz o envio de imagens e demais arquivos, assim como exibição de imagens e downloads de dados enviados. Durante o desenvolvimento dos exemplos, algumas vezes vamos nos deparar com páginas de erro, que traziam um visual ruim e fora do padrão da aplicação. Então, também veremos como trabalhar com a exibição de erros.

8.1 UPLOADS

Em nossa aplicação, no modelo `Academico`, seria interessante ligá-lo a uma imagem, como a foto do acadêmico. Para que isso seja possível, primeiro é preciso ter no modelo propriedades que possam representar essa imagem.

A listagem a seguir traz as propriedades que precisam ser

inseridas na classe `Academico`: a primeira representa o nome do tipo do arquivo armazenado, e a segunda manterá a representação binária do arquivo enviado. Também é definida uma terceira propriedade, que não será mapeada para a tabela. Ela será usada para receber o arquivo que o usuário enviará pela visão.

```
public string FotoMimeType { get; set; }
public byte[] Foto { get; set; }

[NotMapped]
public IFormFile formFile { get; set; }
```

Com esta mudança em nosso modelo, é preciso atualizar nossa base de dados. Execute as instruções `dotnet ef migrations add FotoAcademico` e `dotnet ef database update` no console, tal qual fizemos nos capítulos anteriores.

Para que nossa visão possa enviar arquivos, precisamos inserir o atributo `enctype="multipart/form-data"` na action `<form>`. Como ainda não implementamos a visão `Edit`, trago para cá todo o código dela.

Antes dos links, note a declaração de um `<input type="file">`, que é o controle pelo qual será possível enviar a foto do acadêmico:

```
@model Modelo.Discente.Academico
 @{
     Layout = "_LayoutIES";
 }

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h-1">Alterando um acadêmico existente</div>
    <div class="card-body">
        <form enctype="multipart/form-data" asp-action="Edit">
            <input type="hidden" asp-for="AcademicoID" />
            <div asp-validation-summary="ModelOnly" class="text-d
```

```

anger"></div>
        <div class="form-group">
            <label asp-for="RegistroAcademico" class="control-label"></label>
                <input asp-for="RegistroAcademico" class="form-control" />
                <span asp-validation-for="RegistroAcademico" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label asp-for="Nome" class="control-label"></label>
            <input asp-for="Nome" class="form-control" />
            <span asp-validation-for="Nome" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label asp-for="Nascimento" class="control-label"></label>
            <input asp-for="Nascimento" class="form-control" />
            <span asp-validation-for="Nascimento" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label class="control-label">Foto</label>
            <input type="file" name="foto" class="form-control" />
        </div>
        <div class="form-group text-center h3">
            <input type="submit" value="Atualizar Acadêmico" class="btn btn-primary" />
            <a asp-action="Index" class="btn btn-warning">Retornar à listagem de acadêmicos</a>
        </div>
    </form>
</div>
<div class="card-footer bg-info text-center text-white">
    Informe os dados acima e/ou clique em um dos botões de ação
</div>
</div>
@section ScriptPage {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

```
}
```

Na sequência, precisamos adaptar a action que receberá a requisição. No exemplo, trabalharemos a action `Edit`. Depois de pronto, você pode implementar a mesma lógica na action `Create`. Veja a listagem da action com a nova implementação a seguir.

As mudanças estão em sua assinatura, que, além do objeto do modelo, recebe um `IFormFile` na variável `foto`. Depois, no bloco `try`, esse objeto é copiado para um stream de memória e, então, copiado como array para a propriedade do nosso objeto em alteração.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(long? id, [Bind("AcademicoID, Nome, RegistroAcademico, Nascimento")] Academico academico, IFormFile foto)
{
    if (id != academico.AcademicoID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            var stream = new MemoryStream();
            await foto.CopyToAsync(stream);
            academico.Foto = stream.ToArray();
            academico.FotoMimeType = foto.ContentType;

            await academicoDAL.GravarAcademico(academico);
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!await AcademicoExists(academico.AcademicoID))
            {
```

```

        return NotFound();
    }
    else
    {
        throw;
    }
}
return RedirectToAction(nameof(Index));
}
return View(academico);
}

```

Este foi todo o processo necessário para enviar um arquivo (em nosso caso, uma imagem) do cliente para o servidor. Optei em persistir a imagem recebida na base de dados, mas existem trabalhos que recomendam que, na base de dados, seja gravado o caminho físico desse arquivo no servidor. Isso levaria a gravar o arquivo fora da base, no sistema de arquivos do servidor.

8.2 APRESENTAÇÃO DA IMAGEM NA VISÃO DETAILS

Com a imagem armazenada, é interessante que o usuário possa ver sua foto ao visualizar os detalhes do professor. Para isso, vamos implementar nossa visão `Details`, que ainda não foi feita. O código para a visão está todo apresentado na sequência.

Ao final do código, veja uma HTML Helper `Url.Action()`, que invoca uma action chamada `GetFoto`. Já veremos mais sobre isso. Com exceção a esta observação, todo o resto já é de nosso conhecimento.

```

@model Modelo.Discente.Academico
 @{
     Layout = "_LayoutIES";
 }

```

```

@section styles {
    <link rel="stylesheet" href="~/lib/font-awesome/css/font-awesome.min.css" />
}

<div class="card-block">
    <div class="card-header text-white bg-secondary text-center h-1">Exibindo um acadêmico existente</div>

    <div class="card-body">
        <div class="row">
            <div class="col-9">
                <div class="form-group">
                    <label asp-for="AcademicoID" class="control-label"></label>
                    <br />
                    <div class="input-group">
                        <span class="input-group-addon">
                            <i class="fa fa-key" aria-hidden="true"></i>
                        </span>
                        <input asp-for="AcademicoID" class="form-control" disabled="disabled" />
                    </div>
                    <label asp-for="RegistroAcademico" class="control-label"></label>
                    <br />
                    <div class="input-group">
                        <span class="input-group-addon">
                            <i class="fa fa-address-card-o" aria-hidden="true"></i>
                        </span>
                        <input asp-for="RegistroAcademico" class="form-control" disabled="disabled" />
                    </div>
                    <label asp-for="Nome" class="control-label"></label>
                    <br />
                    <div class="input-group">
                        <span class="input-group-addon">
                            <i class="fa fa-user-circle-o" aria-hidden="true"></i>
                        </span>
                        <input asp-for="Nome" class="form-control" />
                    </div>
                </div>
            </div>
        </div>
    </div>

```

```

        disabled="disabled" />
    </div>
    <label asp-for="Nascimento" class="control-label"><br />
<div class="input-group">
    <span class="input-group-addon">
        <i class="fa fa-address-card-o" aria-hidden="true"></i>
    </span>
    <input asp-for="Nascimento" class="form-control" disabled="disabled" />
</div>
</div>
<div class="col-3">
    
</div>
</div>
<div class="card-footer bg-info text-center text-white">
    <a asp-action="Edit" class="btn btn-warning" asp-route-id="@Model.AcademicoID">Alterar</a> |
    <a asp-action="Index" class="btn btn-warning">Retornar à listagem de acadêmicos</a>
</div>
</div>

```

Se você executar sua aplicação e requisitar a visão Details de algum acadêmico, a sua parte direita (onde a foto deve ser exibida) está sem nada, pois não implementamos o serviço ainda. Como comentado antes do código anterior, esse serviço será realizado pela action GetFoto , a ser implementada no controlador de acadêmicos, com o código apresentado na sequência:

```

public async Task<FileContentResult> GetFoto(long id)
{
    Academico academico = await academicoDAL.ObterAcademicoPorId(
        id);
    if (academico != null)

```

```
    {
        return File(academico.Foto, academico.FotoMimeType);
    }
    return null;
}
```

Execute sua aplicação e veja novamente os detalhes de um acadêmico que você enviou a imagem. Veja que ela aparecerá agora. A figura a seguir destaca isso.

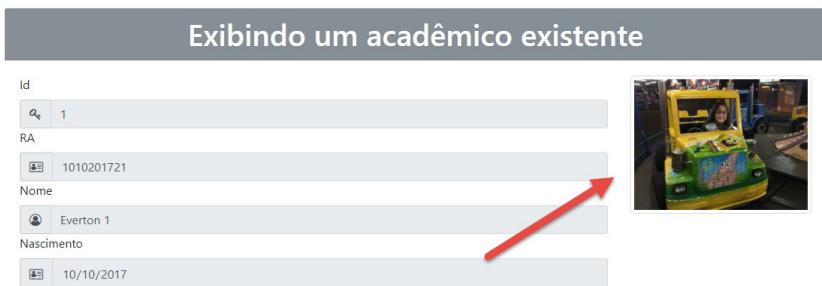


Figura 8.1: Visualização da foto do professor

8.3 PERMITINDO O DOWNLOAD DA IMAGEM ENVIADA

Após o envio e a visualização da imagem, vamos fornecer ao usuário a possibilidade de realizar o download do arquivo que representa a foto. Lembre-se de que poderia ser qualquer tipo de arquivo.

Vamos fornecer o link para o usuário, novamente fazendo uso de uma HTML Helper. Veja o código na sequência. Coloque-o abaixo da URL que exibe a foto, que fizemos anteriormente.

```
@Html.ActionLink("Download", "DownloadFoto", "Academico", new { id = Model.AcademicoID }, null)
```

Para implementarmos a action `DownloadFoto`, serão necessárias algumas alterações em nosso controlador. A primeira é definirmos um campo em nosso controlador para acessar recursos do ambiente em que nossa aplicação está executando. Isso é possível com um objeto da interface `IHostingEnvironment`. Depois, por injeção de dependência, definimos o construtor para receber esse objeto.

```
[Area("Discente")]
public class AcademicoController : Controller
{
    private readonly IESContext _context;
    private IHostingEnvironment _env;
    private readonly AcademicoDAL academicoDAL;

    public AcademicoController(IESContext context, IHostingEnviro
nment env)
    {
        _context = context;
        _env = env;
        academicoDAL = new AcademicoDAL(context);
    }
}
```

Com o objeto disponibilizado para o controlador, podemos implementar a action `DownloadFoto`. Veja o seu código na sequência. Como temos a foto do acadêmico persistida na tabela da base de dados, precisamos agora recuperar o objeto de acordo com o `Id` recebido.

Depois disso, é realizada a criação do arquivo em disco, para que seja possível recuperá-lo. Com a criação realizada, precisamos ler esse arquivo ao usar a classe `PhysicalFileProvider()`, para que então ele seja enviado ao cliente que o solicitou.

Na definição do `FileStream`, utilizamos o `_env.WebRootPath`, que retornará o diretório físico para a pasta

`wwwroot` da aplicação. Veja que, na composição do nome do arquivo, colocamos a extensão `JPG`.

```
public async Task<FileResult> DownloadFoto(long id)
{
    Academico academico = await academicoDAL.ObterAcademicoPorId(
        id);
    string nomeArquivo = "Foto" + academico.AcademicoID.ToString(
        ).Trim() + ".jpg";
    FileStream fileStream = new FileStream(System.IO.Path.Combine(
        _env.WebRootPath, nomeArquivo), FileMode.Create, FileAccess.Writ
        e);
    fileStream.Write(academico.Foto, 0, academico.Foto.Length);
    fileStream.Close();

    IFileProvider provider = new PhysicalFileProvider(_env.WebRoo
        tPath);
    IFileInfo fileInfo = provider.GetFileInfo(nomeArquivo);
    var readStream = fileInfo.CreateReadStream();
    return File(readStream, academico.FotoMimeType, nomeArquivo);
}
```

Um CheckBox para remover a foto do acadêmico

Vamos adaptar nossa visão `Edit` para que também exiba a imagem. Você já tem o código para isso na visão `Details`. Abaixo da foto, vamos inserir um checkbox para que, quando marcado, remova a foto do objeto, deixando o acadêmico sem uma imagem persistida.

Assim, abaixo da tag responsável por exibir a foto, insira o código seguinte:

```
<div class="checkbox">
    <label>
        <input type="checkbox" name="chkRemoverFoto" value="Sim">
        Remover Foto
    </label>
</div>
```

Em nossa action `POST` para o `Edit`, temos de verificar o recebimento do valor para esse controle HTML que será renderizado. Se ele for nulo, quer dizer que não estava marcado; caso contrário, precisamos atribuir `null` à propriedade `Foto`, já que a intenção do usuário é remover a foto do cadastro.

Veja primeiramente o código para a assinatura da action, mantendo a atenção ao último argumento:

```
public async Task<IActionResult> Edit(long? id, [Bind("AcademicoID,Nome,RegistroAcademico,Nascimento")] Academico academico, IFormFile foto, string chkRemoverFoto)
```

Com esta informação, vamos aplicar a lógica comentada anteriormente. Seu código está na sequência. Parte dele foi omitida para facilitar a visualização. Faça a implementação seguinte e teste a sua aplicação.

\\\ Código omitido

```
if (ModelState.IsValid)
{
    var stream = new MemoryStream();
    if (chkRemoverFoto != null)
    {
        academico.Foto = null;
    } else
    {
        await foto.CopyToAsync(stream);
        academico.Foto = stream.ToArray();
        academico.FotoMimeType = foto.ContentType;
    }
    try
    {
        await academicoDAL.GravarAcademico(academico);
    }
}
```

\\\ Código omitido

8.4 PÁGINAS DE ERRO

Na aplicação que temos até agora, podem ocorrer erros em alguns momentos, como ao inserir um login errado, deixar de informar algum dado, ou requisitar um recurso que não existe. Nesses casos, é importante termos uma página de erro para cada situação, mostrando ao usuário uma mensagem que seja orientativa.

O ASP.NET MVC Core oferece alguns recursos para a manipulação de erros. Para começarmos a usá-los, é preciso mudar a variável de ambiente que informa o estágio da aplicação. Se você não mexeu em nada, ela deve estar como `Development`; e se você acessar uma página que não exista, por exemplo, tentar editar um acadêmico com ID inexistente, aparecerá para você uma página em branco, e não um erro de página não encontrada (erro HTTP 404).

Assim, clique com o botão direito do mouse sobre o nome do projeto de nossa aplicação, vá em `Propriedades` e depois em `Depuração`. Em variáveis de ambiente, substitua o valor `Development` para `Production`. Veja a figura a seguir para lhe auxiliar.

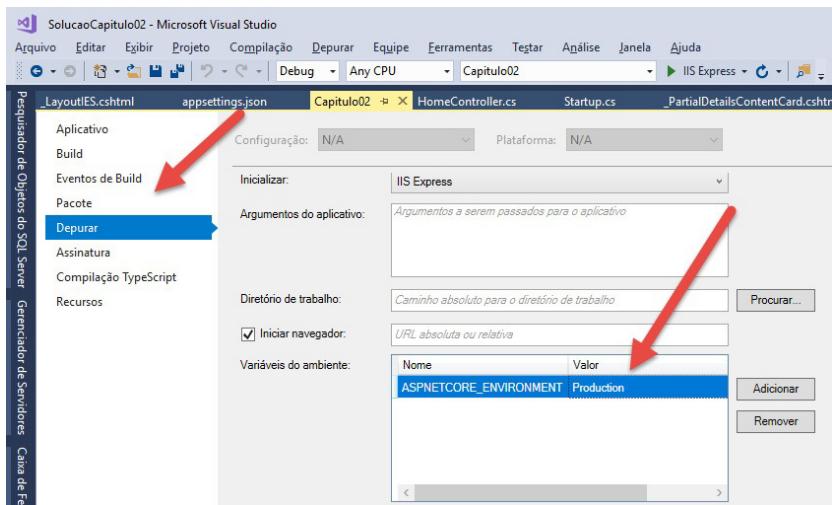


Figura 8.2: Alteração da variável do ambiente de execução da aplicação

Com essa alteração, se você requisitar uma página inexistente, como uma action que não existe no controlador desejado, o navegador mostrará a sua página de erro padrão. Em meu caso, estou usando o Chrome, e a página é a apresentada na figura a seguir.



Figura 8.3: Página de endereço inválido do navegador

A questão é como fazermos para mudar isso. A primeira tentativa será não mostrar a página de erro do navegador, mas sim uma página simples, informando o status do erro. Para isso, na classe `Startup`, no método `Configure()`, adicione a instrução
`app.UseStatusCodePages();` após
`app.UseExceptionHandler("/Home/Error");`, e execute novamente sua aplicação, requisitando uma página inexistente.

Você deverá receber uma página semelhante a apresentada na figura a seguir:

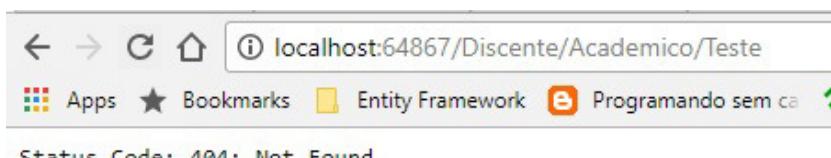


Figura 8.4: Página de endereço inválido do ASP.NET Core MVC

Podemos usar a opção de redirecionamento, substituindo a última instrução inserida pela
`app.UseStatusCodePagesWithRedirects("/Home/Error/{0}")`; . Teste sua aplicação e veja que agora é exibida uma visão dela, na qual é possível personalizar.

Na URL informada para o erro, `Home` representa o controlador; `Error`, a action; e `{0}`, o nome do erro ocorrido. Existe ainda a possibilidade de capturar os erros disparados por exceção, mas isso está fora do escopo deste livro.

8.5 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Neste capítulo, vimos o processo de upload e download de arquivos. Trabalhamos com uma imagem para conseguirmos renderizá-la, porém, este aprendizado serve para qualquer tipo de arquivo. Finalizamos o capítulo apresentando uma técnica para tratamento de erros.

O próximo ensinará a seleção de disciplinas que o aluno se matriculará. Para isso, trabalharemos também o uso de `DropDownLists` aninhados.

CAPÍTULO 9

DROPDOWNLIST COM CHAMADAS AJAX E USO DE SESSÕES

Quando trabalhamos com e-commerce, muitas aplicações referem-se à comercialização de algum produto e/ou serviço como o famoso "*Carrinho de compra*". Existem algumas técnicas que podem ser aplicadas para isso. Em nosso modelo de negócio, não temos a figura desse carrinho, mas, no registro de professores, podemos aplicar o conceito relacionado ao uso de sessão.

Seguindo o problema que trabalhamos no livro, possuímos o cadastro de professores; cada professor pode estar associado a um ou mais cursos; e cada curso pode ter um ou mais professores. Nesta leitura, precisamos registrar os professores nos cursos em que estão associados. Em nosso modelo, ocorre que os cursos estão associados a departamentos, que, por sua vez, estão associados a instituições.

Assim, vamos oferecer ao usuário quatro `DropDownLists` : uma para instituição, outra para departamentos, uma terceira para cursos e uma última para professores. Ao final, após registrarmos os educadores, vamos armazená-los em uma variável de sessão,

para efeito de conhecimento da técnica.

9.1 CRIAÇÃO E ADAPTAÇÃO DE CLASSES PARA O REGISTRO DE PROFESSORES

Começamos a prática criando a classe `Professor`, associada à classe `CursoProfessor` – ambas no namespace `Docente`. Vamos criá-las em uma pasta com este nome, no projeto `Modelo`.

Veja o código para `Professor` na sequência:

```
using System.Collections.Generic;

namespace Modelo.Docente
{
    public class Professor
    {
        public long? ProfessorID { get; set; }
        public string Nome { get; set; }

        public virtual ICollection<CursoProfessor> CursosProfessores { get; set; }
    }
}
```

No código anterior, note que fazemos uso de uma classe que representará a associação entre as classes `Curso` e `Professor`, que terá uma associação de *muitos para muitos*. Como visto no capítulo *Code First Migrations, Data Annotations e validações*, o EF Core não mapeia automaticamente esse tipo de associação para a base de dados.

Logo, precisamos criar essa classe e associá-la às desta associação. Veja na sequência o código para a classe `CursoProfessor`, que representa a associação *um para muitos*. Ela deverá ser criada na pasta `Docente`.

```
using Modelo.Cadastros;

namespace Modelo.Docente
{
    public class CursoProfessor
    {
        public long? CursoID { get; set; }
        public Curso Curso { get; set; }
        public long? ProfessorID { get; set; }
        public Professor Professor { get; set; }
    }
}
```

Agora, precisamos atualizar nossa classe `Curso`, para que ela mapeie a coleção relacionada à classe da associação, apresentada anteriormente. Veja na sequência o código que deve ser adicionado após as propriedades já existentes na classe.

```
public virtual ICollection<CursoProfessor> CursosProfessores { get;
; set; }
```

Resta atualizar a classe de contexto, que precisa ter o mapeamento para professores, assim como o método `OnModelCreating()` atualizado para a nova classe associativa, a `CursoProfessor`. Veja esses códigos na sequência.

```
public DbSet<Professor> Professores { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder
{
    base.OnModelCreating(modelBuilder);

    // Código omitido

    modelBuilder.Entity<CursoProfessor>()
        .HasKey(cd => new { cd.CursoID, cd.ProfessorID });

    modelBuilder.Entity<CursoProfessor>()
        .HasOne(c => c.Curso)
        .WithMany(cd => cd.CursosProfessores)
```

```
        .HasForeignKey(c => c.CursoID);

modelBuilder.Entity<CursoProfessor>()
    .HasOne(d => d.Professor)
    .WithMany(cd => cd.CursosProfessores)
    .HasForeignKey(d => d.ProfessorID);
}
```

9.2 O CONTROLADOR PARA PROFESSORES

Com o modelo e o contexto atualizados, precisamos criar os serviços que serão fornecidos ao usuário. Como o serviço (foco deste capítulo) é o de registrar o professor, precisamos criar um controlador para eles.

Já que estamos trabalhando com Areas, crie uma nova em sua aplicação, chamada Docente e, dentro da pasta controllers , crie um controlador chamado ProfessorController . O código inicial para esse controlador pode ser verificado na sequência.

No código, observe a declaração de campos privados para o contexto e para os DALs que manipularemos pelas actions que implementarmos nesse controlador. O construtor, que recebe o contexto por injeção de dependências, é responsável pela inicialização dos DALs. Note o atributo [Area("Docente")] antes do nome do controlador:

```
using Capitulo02.Data;
using Capitulo02.Data.DAL.Cadastros;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using Modelo.Cadastros;
using Capitulo02.Data.DAL.Docente;
using System.Collections.Generic;
using Modelo.Docente;

namespace Capitulo02.Areas.Docente.Controllers
```

```

{
    [Area("Docente")]
    public class ProfessorController : Controller
    {
        private readonly IESContext _context;
        private readonly InstituicaoDAL instituicaoDAL;
        private readonly DepartamentoDAL departamentoDAL;
        private readonly CursoDAL cursoDAL;
        private readonly ProfessorDAL professorDAL;

        public ProfessorController(IESContext context)
        {
            _context = context;
            instituicaoDAL = new InstituicaoDAL(context);
            departamentoDAL = new DepartamentoDAL(context);
            cursoDAL = new CursoDAL(context);
            professorDAL = new ProfessorDAL(context);
        }
    }
}

```

Precisamos agora criar a action responsável por renderizar a visão pela qual o usuário registrará os professores nos cursos em que ministram disciplinas. Para esta atividade, você precisará ter dados nas tabelas de cursos e professores. Não fizemos isso aqui no livro, mas você já consegue implementar essas funcionalidades sozinho, com tudo o que vimos.

Crie os controladores e, neles, as actions do CRUD e suas respectivas visões, tal qual fizemos para a instituição e o departamento. Os modelos para essas classes são vistos na sequência.

Veja a classe `Curso` :

```

using Modelo.Docente;
using System.Collections.Generic;

namespace Modelo.Cadastros
{

```

```

public class Curso
{
    public long? CursoID { get; set; }
    public string Nome { get; set; }

    public long? DepartamentoID { get; set; }
    public Departamento Departamento { get; set; }

    public virtual ICollection<CursoDisciplina> CursosDisciplinas { get; set; }
    public virtual ICollection<CursoProfessor> CursosProfessores { get; set; }
}

```

Veja a classe Professor :

```

using System.Collections.Generic;

namespace Modelo.Docente
{
    public class Professor
    {
        public long? ProfessorID { get; set; }
        public string Nome { get; set; }

        public virtual ICollection<CursoProfessor> CursosProfessores { get; set; }
    }
}

```

Para o que desejo, precisarei de um método que atenderá a action HTTP GET e a action de confirmação do professor a ser registrado, a HTTP POST . Na listagem a seguir, note que o método recebe quatro coleções e, em cada uma delas, no corpo do método, é inserido um elemento com índice zero, que servirá de orientação. Então, são criados os valores para o ViewBag , que serão passados para a visão.

```

public void PrepararViewBags(List<Instituicao> instituicoes, List<Departamento> departamentos, List<Curso> cursos, List<Professor>

```

```

professores)
{
    instituicoes.Insert(0, new Instituicao() { InstituicaoID = 0,
Nome = "Selecione a instituição" });
    ViewBag.Instituicoes = instituicoes;

    departamentos.Insert(0, new Departamento() { DepartamentoID = 0,
Nome = "Selecione o departamento" });
    ViewBag.Departamentos = departamentos;

    cursos.Insert(0, new Curso() { CursoID = 0, Nome = "Selecione
o curso" });
    ViewBag.Cursos = cursos;

    professores.Insert(0, new Professor() { ProfessorID = 0, Nome =
"Selecione o professor" });
    ViewBag.Professores = professores;
}

```

Agora vamos à action HTTP GET , que o usuário requisitará via menu ou qualquer outro link que você julgar necessário. No código a seguir, perceba que o método anteriormente criado é invocado e enviamos as listas que deverão ser usadas na visão para ele. O primeiro parâmetro recupera todas as instituições registradas, e os demais são criados sem valores, pois dependerão uns dos outros em relação aos dados que disponibilizarão.

A visão deverá funcionar da seguinte maneira: o usuário escolhe uma instituição, para os departamentos da instituição serem exibidos; ao selecionar o departamento, os cursos dele também são mostrados. Por último, ao selecionar o curso, são selecionados os professores ainda não atribuídos a ele. Ou seja, implementaremos um aninhamento de DropDownList .

```

[HttpGet]
public IActionResult AdicionarProfessor()
{
    PrepararViewBags(instituicaoDAL.ObterInstituicoesClassificada
sPorNome().ToList(),

```

```

        new List<Departamento>().ToList(), new List<Curso
>().ToList(), new List<Professor>().ToList());
    return View();
}

```

O método que representa a action `POST` da visão renderizada pelo método já apresentado pode ser visto a seguir:

```

[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult AdicionarProfessor([Bind("InstituicaoID, Dep
artamentoID, CursoID, ProfessorID")] AdicionarProfessorViewModel
model)
{
    if (model.InstituicaoID == 0 || model.DepartamentoID == 0 || 
model.CursoID == 0 || model.ProfessorID == 0)
    {
        ModelState.AddModelError("", "É preciso selecionar todos
os dados");
    } else
    {
        cursoDAL.RegistrarProfessor((long)model.CursoID, (long)mo
del.ProfessorID);

        PrepararViewBags(instituicaoDAL.ObterInstituicoesClassifi
cadasPorNome().ToList(),
                    departamentoDAL.ObterDepartamentosPorInstituicao((lon
g)model.InstituicaoID).ToList(),
                    cursoDAL.ObterCursosPorDepartamento((long)model.Depar
tamentoID).ToList(),
                    cursoDAL.ObterProfessoresForaDoCurso((long)model.Curs
oID).ToList());
    }
    return View(model);
}

```

Na implementação anterior, temos a invocação de dois métodos novos: o `RegistrarProfessor()`, que está logo no início do `else`; e o `ObterProfessoresForaDoCurso()` – ambos pertencentes à classe `CursoDAL`. O método apresentado anteriormente tem em si a funcionalidade semelhante aos métodos

Create , que criamos nos exemplos do livro. Assim, não são necessárias maiores explicações.

Veja o método RegistrarProfessor() :

```
public void RegistrarProfessor(long cursoID, long professorID)
{
    var curso = _context.Cursos.Where(c => c.CursoID == cursoID).
Include(cp => cp.CursosProfessores).First();
    var professor = _context.Professores.Find(professorID);
    curso.CursosProfessores.Add(new CursoProfessor() { Curso = cu
rso, Professor = professor });
    _context.SaveChanges();
}
```

No código anterior, o método recebe dois argumentos do tipo long , referentes às chaves que deverão ser procuradas na base de dados. Fazemos uso dos métodos de extensão do LINQ em associação com o EF Core. Após recuperarmos os dois objetos, é realizada a sua inserção na coleção CursosProfessores , que é carregada em conjunto com o objeto do curso desejado por usarmos o Include() na primeira instrução.

Agora, vamos ao método responsável por selecionar os professores que ainda não fazem parte do curso em que se deseja registrá-los. Veja o método na sequência.

Buscamos o resultado de um SQL conhecido como NOT IN , ou seja, um conjunto de dados A que não pertença ao conjunto de dados B. Para isso, recuperamos o curso recebido como argumento, selecionamos os professores desse curso e transformamos o resultado em um array.

Para finalizar, no objeto Professores , é realizado um Where() , negando o resultado de Contains() em relação aos professores que já estão no curso:

```
public IQueryable<Professor> ObterProfessoresForaDoCurso(long cursoID)
{
    var curso = _context.Cursos.Where(c => c.CursoID == cursoID).
Include(cp => cp.CursosProfessores).First();
    var professoresDoCurso = curso.CursosProfessores.Select(cp =>
cp.ProfessorID).ToArray();
    var professoresForaDoCurso = _context.Professores.Where(p =>
!professoresDoCurso.Contains(p.ProfessorID));
    return professoresForaDoCurso;
}
```

9.3 A VISÃO PARA O REGISTRO DE PROFESSORES

Como já foi visto nas visões já criadas, podemos ter associado a elas modelos responsáveis por manter o valor informado na visão e enviado para o controlador (e também do controlador para a visão). O modelo que vamos utilizar aqui faz parte apenas da camada de visão.

Desta maneira, na pasta `Models` da área `Docente`, crie uma classe chamada `AdicionarProfessorViewModel`, representada na listagem a seguir:

```
namespace Capitulo02.Areas.Docente.Models
{
    public class AdicionarProfessorViewModel
    {
        public long? InstituicaoID { get; set; }
        public long? DepartamentoID { get; set; }
        public long? CursoID { get; set; }
        public long? ProfessorID { get; set; }
    }
}
```

Vamos partir para a visão. Crie-a da mesma maneira como fizemos com todas as anteriores: clicando com o botão direito

sobre o nome da action e em Adicionar exibição . O modelo será vazio, pois criaremos a visão do zero.

Na sequência, está a listagem completa da visão e, após ela, veremos as partes relevantes ao tópico.

```
@model Capitulo02.Areas.Docente.Models.AdicionarProfessorViewMode
l
#{@
    Layout = "_LayoutIES";
}

<div class="card-block">
    <div class="card-header text-white bg-danger text-center h1">
        Registrando um professor em um curso</div>
    <div class="card-body">
        <form asp-action="AdicionarProfessor">
            <div asp-validation-summary="ModelOnly" class="text-d
anger"></div>
            <div class="form-group">
                <label asp-for="InstituicaoID" class="control-lab
el"></label>
                <select asp-for="InstituicaoID" class="form-contr
ol" asp-items="@((new SelectList(@ ViewBag.Instituicoes, "Instituic
aoID", "Nome")))"></select>
            </div>
            <div class="form-group">
                <label asp-for="DepartamentoID" class="control-la
bel"></label>
                <select asp-for="DepartamentoID" class="form-cont
rol" asp-items="@((new SelectList(@ ViewBag.Depa
rtamentos, "DepartamentoID", "Nome")))" data-url =
"@Url.Action("ObterDepartament
osPorInstituicao", "Professor", new { area = "Docente" })">
            </select>
        </div>
        <div class="form-group">
            <label asp-for="CursoID" class="control-label"></
label>
            <select asp-for="CursoID" class="form-control"
                asp-items="@((new SelectList(@ ViewBag.Curs
os, "CursoID", "Nome")))">
```

```

        data-url = "@Url.Action(\"ObterCursosPorDepartamento\", \"Professor\", new { area = \"Docente\" })">
            </select>
        </div>
        <div class="form-group">
            <label asp-for="ProfessorID" class="control-label">
        <></label>
            <select asp-for="ProfessorID" class="form-control">
                asp-items="@((new SelectList(@ViewBag.Professores, "ProfessorID", "Nome")))">
                    data-url = "@Url.Action(\"ObterProfessoresForaDoCurso\", \"Professor\", new { area = \"Docente\" })">
                        </select>
                    </div>
                    <div class="form-group text-center h3">
                        <input type="submit" value="Registrar Professor" class="btn btn-light" />
                        <a asp-action="VerificarUltimosRegistros" class="btn btn-info">Verificar últimos registros</a>
                    </div>
                </form>
            </div>
            <div class="card-footer bg-dark text-center text-white">
                Informe os dados acima e/ou clique em um dos botões de ação
            </div>
        </div>
    </div>

@section ScriptPage {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
<script type="text/javascript">
    $(document).on("change", '#InstituicaoID', function (e) {
        var instituicaoID = $(this).find(":selected").val();
        GetDadosParaDropDownList(instituicaoID, '#DepartamentoID', 'DEPARTAMENTO');
    });

    $(document).on("change", '#DepartamentoID', function (e) {
        var departamentoID = $(this).find(":selected").val();
        GetDadosParaDropDownList(departamentoID, '#CursoID', 'CURSO');
    });
}

```

```

$(document).on("change", '#CursoID', function (e) {
    var cursoID = $(this).find(":selected").val();
    GetDadosParaDropDownList(cursoID, '#ProfessorID', 'PROFESSOR');
});

function GetDadosParaDropDownList(valorBuscar, controleAtualizar, nomeDado) {
    var optionControleAtualizar = controleAtualizar + ' option';
    if (valorBuscar.length > 0) {
        var url = $(controleAtualizar).data('url');
        $.getJSON(url, { actionID: valorBuscar }, function(dadosRecebidos) {
            $(optionControleAtualizar).remove();
            $(controleAtualizar).append('<option value="0">Selecione o ' + nomeDado + '</option>');
            for (i = 0; i < dadosRecebidos.length; i++) {
                $(controleAtualizar).append('<option value=' + dadosRecebidos[i].value + '>' + dadosRecebidos[i].text + '</option>');
            }
        }).fail(function (jqXHR, textStatus, errorThrown) {
            debugger;
            alert('Erro de conexão', 'Erro obtendo dados');
        });
    } else {
        $(optionControleAtualizar).remove();
        $(controleAtualizar).append('<option value=""></option>');
    }
}
</script>
}

```

Busque no código e verifique que temos quatro tags `<select>` , para: instituições, departamentos, cursos e professores. Esse elemento será renderizado no navegador como controles `DropDown` . Já utilizamos isso quando atribuímos uma

instituição a um departamento.

Com exceção do primeiro (instituições), todos têm um atributo dentro desta tag: data-url = "@Url.Action("ObterDepartamentosPorInstituicao", "Professor", new { area = "Docente" })". Há uma variação nos elementos quando informamos a URL no HtmlHelper Url.Action(), então, fique atento a isso. Cada elemento possui configurações que definem o comportamento a ser capturado por instruções em JavaScript, que estão ao final do código anterior e logo serão explicadas.

Este valor de data-url será usado pelo código JavaScript e jQuery para renderizar sempre que necessário os controles que exibem os departamentos, cursos e professores, sem a necessidade de renderizar toda a página.

Isso é uma implementação do conceito de AJAX, que não é mais tão recente. Se você ainda não o conhece, uma breve introdução pode ser vista em:
https://www.w3schools.com/xml/ajax_intro.asp.

Agora, vá ao final do código apresentado anteriormente. Temos a definição da seção @ScriptPage e, dentro dela, temos o elemento <script>. Este implementa o código JavaScript que capturará o evento responsável por identificar mudanças de valores nos DropDownList .

Temos código para os controles de instituições, departamentos

e cursos (três funções anônimas). Veja-os em destaque na sequência. Observe que uma função anônima (uma para cada controle) é invocada quando ocorre a alteração (change) do valor de cada controle que desejamos (`InstituicaoID`, `DepartamentoID` e `CursoID`).

Nesta função, o valor selecionado é pesquisado e obtido no controle, e atribuído a uma variável (`instituicaoID`, `departamentoID` e `cursoID`). Em seguida, é invocada a função

`GetDadosParaDropDownList()`, enviando a ela como argumentos: o valor do ID do controle selecionado e o nome do controle capturado pelo evento que a função manipulará.

No nosso caso, os nomes desses controles são atribuídos de acordo com as propriedades definidas na classe de modelo. Veja:

```
$("document").on("change", "#InstituicaoID", function (e) {
    var instituicaoID = $(this).find(":selected").val();
    GetDadosParaDropDownList(instituicaoID, '#DepartamentoID', 'DEPARTAMENTO');
});

$("document").on("change", "#DepartamentoID", function (e) {
    var departamentoID = $(this).find(":selected").val();
    GetDadosParaDropDownList(departamentoID, '#CursoID', 'CURSO')
};

$("document").on("change", "#CursoID", function (e) {
    var cursoID = $(this).find(":selected").val();
    GetDadosParaDropDownList(cursoID, '#ProfessorID', 'PROFESSOR')
});
```

Vamos agora à função `GetDadosParaDropDownList()`, que está na sequência. Seu corpo começa com a concatenação do ID do

controle que deverá ter os dados atualizados, com a palavra `option`. Com isso, podemos remover as opções disponibilizadas no controle antes da alteração dos seus dados.

Então, é obtida a URL definida para execução quando o controle sofrer alteração – aquela que definimos no elemento `data-url`. Essas URLs (uma em cada controle atualizável) apontam para actions que ainda não criamos. Caso essas actions retornem valores, eles são adicionados ao controle em questão. Não detalhei a execução do `fail()` por ter seu código semanticamente compreendido.

```
function GetDadosParaDropDownList(valorBuscar, controleAtualizar,
nomeDado) {
    var optionControleAtualizar = controleAtualizar + ' option';
    var url = $(controleAtualizar).data('url');
    $.getJSON(url, { actionID: valorBuscar }, function (dadosRecebidos) {
        $(optionControleAtualizar).remove();
        $(controleAtualizar).append('<option value="0">Selecione
o ' + nomeDado + '</option>');
        for (i = 0; i < dadosRecebidos.length; i++) {
            $(controleAtualizar).append('<option value="' + dados
Recebidos[i].value + '">' + dadosRecebidos[i].text + '</option>');
        }
    }).fail(function (jqXHR, textStatus, errorThrown) {
        debugger;
        alert('Erro de conexão', 'Erro obtendo dados');
    });
}
```

9.4 ACTIONS INVOCADAS VIA AJAX/JQUERY PARA ATUALIZAÇÃO DOS DROPPDOWNNS

Como comentado anteriormente, os atributos `data-url` (definidos nos elementos `<select>`) referem-se a uma action que deverá ser invocada quando os controles `DropDown` sofrerem

alterações. Desta maneira, vamos implementá-las no controlador.

Veja seus códigos na sequência. Observe nos métodos que o retorno agora é JsonResult .

```
public JsonResult ObterDepartamentosPorInstituicao(long actionID)
{
    var departamentos = departamentoDAL.ObterDepartamentosPorInst
ituicao(actionID).ToList();
    return Json(new SelectList(departamentos, "DepartamentoID", "Nome"));
}

public JsonResult ObterCursosPorDepartamento(long actionID)
{
    var cursos = cursoDAL.ObterCursosPorDepartamento(actionID).To
List();
    return Json(new SelectList(cursos, "CursoID", "Nome"));
}

public JsonResult ObterProfessoresForaDoCurso(long actionID)
{
    var professores = cursoDAL.ObterProfessoresForaDoCurso(action
ID).ToList();
    return Json(new SelectList(professores, "ProfessorID", "Nome"
));
}
```

Muito bem, agora você pode testar a aplicação. Requisite a action AdicionarProfessor do controlador Professor (da Area Docente), e selecione os dados, começando por Instituição, depois Departamento, seguindo para Curso e finalizando com Professor.

Clique no botão responsável por registrar o professor. Tente registrar mais educadores em instituições, departamentos e cursos diferentes. A minha URL para teste ficou assim:
<http://localhost:64867/Docente/Professor/AdicionarProfessor>.

9.5 ARMAZENANDO VALORES NA SESSÃO

Neste livro, vimos como armazenar dados entre requisições, por meio de `ViewBags` e `TempData`. Também estudamos a persistência de objetos em uma base de dados SQL Server ao utilizar o Entity Framework Core.

Outro mecanismo que possibilita armazenamento de dados, que vale a pena conhecermos, é a sessão do cliente com a aplicação. Para conseguirmos usar sessões em nosso projeto, precisamos adicionar a ele as dependências Nuget `Microsoft.AspNetCore.Session`. Lembre-se de que, para isso, basta clicar com o botão direito em `Dependências` e, então, em `Gerenciar Pacotes do Nuget...`.

Após a instalação do pacote na classe `Startup`, insira as chamadas da sequência no método `ConfigureServices()`, antes da chamada à `services.AddMvc()`. Elas adicionam o uso de sessão à aplicação.

```
services.AddSession();
services.AddDistributedMemoryCache();
```

Agora, ainda na classe `Startup`, mas no método `Configure()` e antes da chamada à `app.UseMvc()`, insira a chamada à `app.UseSession();`.

Na sequência, vamos criar um método no controlador de Professores com a implementação a seguir. Na primeira instrução, observe a instanciação de um objeto com os valores recebidos como argumentos no método (ainda não o invocamos).

Em seguida, uma coleção é declarada, e uma chave é buscada na sessão (`cursosProfessores`). Caso a chave exista, ela é

desserializada para o objeto que manterá a coleção de professores registrados. Ao final, o novo objeto é adicionado à coleção, que então é serializada e armazenada na sessão, com a chave já comentada.

```
public void RegistrarProfessorNaSessao(long cursoID, long professorID)
{
    var cursoProfessor = new CursoProfessor() { ProfessorID = professorID, CursoID = cursoID };
    List<CursoProfessor> cursosProfessor = new List<CursoProfessor>();
    string cursosProfessoresSession = HttpContext.Session.GetString("cursosProfessores");
    if (cursosProfessoresSession != null)
    {
        cursosProfessor = JsonConvert.DeserializeObject<List<CursoProfessor>>(cursosProfessoresSession);
    }
    cursosProfessor.Add(cursoProfessor);
    HttpContext.Session.SetString("cursosProfessores", JsonConvert.SerializeObject(cursosProfessor));
}
```

Para nosso teste, vamos invocar o método anteriormente criado na action POST de registro de professor. Logo após o registro do educador, inclua a instrução

```
RegistrarProfessorNaSessao((long)model.CursoID,
(long)model.ProfessorID);
```

pela chamada ao respectivo método DAL . Com isso feito, podemos criar agora o método que representará a action que vai recuperar esses dados armazenados na sessão e passá-los para uma visão.

Veja este código na sequência:

```
public IActionResult VerificarUltimosRegistros()
{
    List<CursoProfessor> cursosProfessor = new List<CursoProfessor>();
```

```

    string cursosProfessoresSession = HttpContext.Session.GetString("cursosProfessores");
    if (cursosProfessoresSession != null)
    {
        cursosProfessor = JsonConvert.DeserializeObject<List<CursoProfessor>>(cursosProfessoresSession);
    }
    return View(cursosProfessor);
}

```

Note que a segunda instrução do método é buscada por uma chave na sessão, `cursosProfessores`. Se a chave já existir, ela é desserializada para o tipo de dado da variável `cursosProfessor`, declarada na primeira instrução do método. A action enviará esses dados para a visão. Caso a chave não retorne nada, uma coleção vazia é enviada.

Para testarmos essa implementação, precisamos criar a visão. Crie-a clicando no nome do método com o botão direito do mouse, tal qual fizemos para todas as visões criadas no livro. Seu código é apresentado na sequência.

Observe que apresento apenas os IDs de curso e professor. Fica como uma atividade a recuperação dos objetos utilizando o EF Core e a exibição dos nomes dos cursos e dos professores registrados.

```

@model IEnumerable<Modelo.Docente.CursoProfessor>

 @{
     Layout = null;
 }

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>VerificarUltimosRegistros</title>

```

```

</head>
<body>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Curso)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Professor)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CursoID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ProfessorID)
                </td>
            </tr>
        }
        </tbody>
    </table>
    <p>
        <a asp-action="AdicionarProfessor">Voltar para Adicionar Professor</a>
    </p>
</body>
</html>

```

Teste sua aplicação, registre um professor, e acesse a action criada anteriormente para visualizar os educadores registrados. Registre um novo professor e retorne à visualização dos dados recuperados da sessão. Tente verificar os dados da sessão em outro navegador. Você verá que nada aparece, pois a sessão está atrelada ao navegador, ou seja, o cliente da conexão com a aplicação.

9.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Neste capítulo, vimos o processo relacionado à interação do usuário com um conjunto de controles `DropDownLists` para selecionar dados de maneira encadeada. Utilizamos o JavaScript com jQuery nesse processo.

Concluímos o capítulo apresentando o uso de sessão como mecanismo de persistência para dados temporários. Com isso, concluímos as atividades previstas neste livro.

CAPÍTULO 10

OS ESTUDOS NÃO PARAM POR AQUI

O .NET Core é uma evolução do .NET Framework, e isso foi muito bem-visto pela comunidade, pois abriu mais ainda a entrada da Microsoft em ambientes multiplataforma. Por ser estável, vem ganhando cada vez mais adeptos, e detém uma poderosa linguagem, o C#, com uma curva de aprendizado relativamente boa. Além do framework, também foi possível conhecer o Visual Studio, o Entity Framework Core, o Bootstrap e um pouquinho de jQuery e JavaScript.

O livro teve seus três capítulos iniciais como introdutórios – uma apresentação do ASP.NET Core MVC 5, do Entity Framework Core e do Bootstrap –, dando a você subsídios para a criação de pequenas aplicações. Os três seguintes capítulos trouxeram recursos adicionais, como associações, uma arquitetura para as suas futuras aplicações e personalização de propriedades das classes de modelo.

Em seus três capítulos seguintes, foram apresentados técnicas e recursos para o controle de acesso de usuários à aplicação, uploads, downloads e tratamento de erros. O livro finalizou com a apresentação de controles `DropDownList` aninhados e o uso da

sessão para o armazenamento de dados.

Espera-se que os recursos e as técnicas exibidos tenham provocado em você uma curiosidade ou interesse em um aprofundamento, o que certamente agora se tornará mais fácil. Programar com C# é muito bom, e criar aplicações web usando .NET Core e o ASP.NET Core MVC não é difícil.

Comece agora a criar suas próprias aplicações. Quando surgirem dificuldades, você verá que a comunidade existente na internet é bem grande e está disposta a auxiliá-lo. Bons estudos e sucesso!