

ASP.NET MVC 5

Crie aplicações web na
plataforma Microsoft®



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: 978-85-5519-189-3

EPUB: 978-85-5519-190-9

MOBI: 978-85-5519-191-6

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

INTRODUÇÃO

Como sempre digo na abertura de todos os meus livros, ensinar e aprender são tarefas que andam juntas e, para que seja alcançado o objetivo em cada uma delas, são necessários 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 MVC 5, que é um framework da Microsoft para o desenvolvimento de aplicações Web. Durante os capítulos, é feito uso do IDE da Microsoft, o Visual Studio 2015 Community Edition, e linguagem adotada para os exemplos é a C#. Conceitos de técnicas, do framework, da ferramenta e da linguagem, sempre que utilizados, são introduzidos.

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 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 torná-lo uma ferramenta no ensino e aprendizado no desenvolvimento de aplicações para web, fazendo uso de C#, de uma forma mais precisa e direcionada às disciplinas ministradas em cursos que envolvam informática e

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

O repositório com todos os códigos-fonte usados no livro podem ser encontrados em: <https://github.com/evertonfoz/asp-net-mvc-casa-do-codigo>.

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

Capítulo 1 – A primeira aplicação ASP.NET MVC 5

Este primeiro capítulo traz a introdução a alguns conceitos relacionados ao ASP.NET MVC e o desenvolvimento de aplicações web. Também já ocorre a implementação da primeira aplicação. Esta realiza as operações básicas em um conjunto de dados, conhecidas como CRUD (C reate, R ead, U pdate e D elete). Inicialmente, essas operações são realizadas em uma collection, pois este capítulo se dedica à introdução ao framework, preparando para o segundo capítulo, que introduz acesso a uma base de dados.

Capítulo 2 – Realizando acesso a dados na aplicação ASP.NET MVC com o Entity Framework

Com o conhecimento que se espera do leitor após a leitura do primeiro capítulo, no qual são apresentadas técnicas específicas do ASP.NET MVC para a criação de um CRUD, neste segundo capítulo aplica-se estas técnicas para a manutenção do CRUD em

uma base de dados. Essas implementações são realizadas no SQL Server, por meio do Entity Framework, 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, e o ASP.NET possui recursos para esta característica. Este capítulo apresenta também o Bootstrap, que é 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, que é o jQuery DataTable, onde um conjunto de dados é renderizado em uma tabela com recursos para busca e classificação.

Capítulo 4 – Associações no Entity Framework

As classes identificadas no modelo de negócio de uma aplicação não são isoladas umas das outras. Muitas vezes elas se associam 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 é o cliente, o vendedor e os produtos que são adquiridos. Este capítulo trabalha a associação entre duas classes, o que dará subsídios para aplicar estes conhecimentos em outras aplicações. Também são trazidos controles que permitem a escolha de objetos para gerar a associação.

Capítulo 5 – Separando a aplicação em camadas

O ASP.NET MVC traz em seu nome a o padrão de projeto MVC (*Mode-View-Controller*, ou Modelo-Visão-Controlador). Embora os recursos (classes e visões, dentre outros) possam ser armazenados em pastas que são criadas automaticamente em um novo projeto (Model, Views e Controllers), a aplicação criada, por si só, não está dividida em camadas, pois não estão em módulos que propiciem uma independência do modelo. Este capítulo apresenta os conceitos de Coesão e Acoplamento. Implementa uma estrutura básica que pode ser replicada para seus projetos.

Capítulo 6 — Code First Migrations, Data Annotations, validações e jQueryUI

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 na base de dados. Quando se utiliza o Entity Framework, estas 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 oferece o Code First Migration, que possibilita o versionamento da estrutura de uma base de dados que é 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 nela registrados. Também são apresentados os Data Annotations para definir características de propriedades e algumas regras de validação. O jQuery novamente surge, agora com a exibição de mensagens de erro e com validações no lado cliente.

Capítulo 7 — Areas, autenticação e autorização

Quando se desenvolve uma aplicação com muitas classes, controladores e visões, torna-se difícil administrar a organização pela forma trivial oferecida pelo ASP.NET 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 em submodelos, onde ficam, de maneira mais organizada, seus controladores, modelos e visões. O conceito Modelo para Visões também é apresentado. O capítulo se dedica 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 enviar arquivos para uma aplicação, como obter arquivos de imagens (para que possam ser renderizadas) e como 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 — Registro de compras em um carrinho fazendo uso de sessão

Quando trabalhamos aplicações comerciais na web, muitas delas se referem a comercialização de algum produto e/ou serviço, o que ficou conhecido como "Carrinho de compra". Uma das

técnicas comuns para isso é a aplicação fornecer uma listagem de produtos, com possibilidade de pesquisa e, por meio desta listagem, adicionar o item desejado ao seu carrinho de compras. E ao final, visualizar seu carrinho e informar seus dados para cobrança e recebimento dos itens adquiridos.

O armazenamento dos produtos selecionados em uma variável de sessão, que pode ter seu tempo de vida definido pela aplicação, é um mecanismo comum nesta funcionalidade. O capítulo traz algo mais sobre o jQuery, um controle que funcionará como autocomplemento e finaliza com a apresentação de conteúdo fazendo uso de AJAX.

Capítulo 10 — Utilização de DropDownList aninhado, RadioButton e CheckBox

O uso de controles do tipo `DropDownList` pode ser uma atividade necessária em sua aplicação, e pode ser necessário o uso de mais que um destes controles, além de um determinado controle ter suas opções dependentes de uma seleção em outro. Também é comum o uso de controles do tipo `RadioButton` e `CheckBox`, em que opções também são apresentadas, permitindo seleção por parte do usuário. No caso do `CheckBox`, existe a possibilidade da marcação ou não de uma destas opções. Este capítulo traz estas três implementações.

Capítulo 11 — Os estudos não param por aqui

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

Caso tenha alguma dúvida ou sugestão, procure a comunidade do livro para tirar dúvidas. Ela está disponível em <http://forum.casadocodigo.com.br/>. Lá podemos discutir mais sobre os temas tratados aqui. Você será muito bem-vindo!

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 para o meio acadêmico e empresarial.



Sumário

1 A primeira aplicação ASP.NET MVC 5	1
1.1 Criando o projeto no Visual Studio 2015 Community	4
1.2 Criando o controlador para Categorias de produtos	8
1.3 Criando a classe de domínio para Categorias de produtos	11
1.4 Implementando a interação da action Index com a visão	12
1.5 O conceito de rotas do ASP.NET MVC	19
1.6 Implementando a inserção de dados no controlador	21
1.7 Implementando a alteração de dados no controlador	30
1.8 Implementando a visualização de um único registro	34
1.9 Finalizando a aplicação por meio da implementação da operação Delete do CRUD	36
1.10 Conclusão sobre as atividades realizadas no capítulo	38
2 Realizando acesso a dados na aplicação ASP.NET MVC com o Entity Framework	40
2.1 Começando com o Entity Framework	41
2.2 Implementando o CRUD fazendo uso do Entity Framework	48
2.3 Conclusão sobre as atividades realizadas no capítulo	59

3 Layouts, Bootstrap e jQuery DataTable	61
3.1 O Bootstrap	62
3.2 Layouts	65
3.3 Adaptando as visões para o uso do Bootstrap	70
3.4 Configurando o menu de acesso para destacar a página atual	86
3.5 Conclusão sobre as atividades realizadas no capítulo	90
4 Associações no Entity Framework	91
4.1 Associando as classes já criadas a uma nova classe	92
4.2 Criando a visão Index para a classe associada	96
4.3 Inicializadores de contexto do Entity Framework	100
4.4 Criando a visão Create para a classe Produto	103
4.5 Criando a visão Edit para a classe Produto	108
4.6 Criando a visão Details para a classe Produto	111
4.7 Criando a visão Delete para a classe Produto	114
4.8 Adaptando a visão Details de Fabricantes	118
4.9 Conclusão sobre as atividades realizadas no capítulo	122
5 Separando a aplicação em camadas	124
5.1 Contextualização sobre as camadas	124
5.2 Criando a camada de negócio — O modelo	126
5.3 Criando a camada de persistência	128
5.4 Criando a camada de serviço	132
5.5 Adaptando a camada de aplicação	135
5.6 Adaptando as visões para minimizar redundâncias	140
5.7 Conclusão sobre as atividades realizadas no capítulo	142
6 Code First Migrations, Data Annotations, validações e	

Casa do Código	Sumário
jQueryUI	143
6.1 Fazendo uso do Code First Migrations	144
6.2 Adaptando a classe Produto para as validações	145
6.3 Testando as alterações implementadas	148
6.4 Implementando o controle de data	150
6.5 Fazendo uso do jQueryUI para controles de data	152
6.6 Validação no lado cliente	156
6.7 Conclusão sobre as atividades realizadas no capítulo	158
7 Areas, autenticação e autorização	160
7.1 Areas	160
7.2 Segurança em aplicações ASP.NET MVC	163
7.3 Listando os usuários registrados	170
7.4 Criando usuários	175
7.5 Alterando usuários já cadastrados	180
7.6 Removendo um usuário existente	185
7.7 Adaptando o menu da aplicação para as funcionalidades de usuários	189
7.8 Restringindo o acesso a actions	189
7.9 Implementando a autenticação	190
7.10 Utilizando papéis (roles) na autorização	196
7.11 Gerenciando membros de papéis	205
7.12 Criando os acessos para login e logout	210
7.13 Alterando o processo de autorização para utilizar papéis	211
7.14 Conclusão sobre as atividades realizadas no capítulo	213
8 Uploads, downloads e erros	215

Sumário	Casa do Código
8.1 Uploads	215
8.2 Download	220
8.3 Páginas de erro	222
8.4 Conclusão sobre as atividades realizadas no capítulo	227
9 Um carrinho de compras	229
9.1 Adição do carrinho de compra ao modelo de negócio	230
9.2 O controlador para o carrinho de compra	231
9.3 A visão para o carrinho de compra	232
9.4 Registrando o produto no carrinho de compra	239
9.5 O que falta para terminar o carrinho?	240
9.6 Conclusão sobre as atividades realizadas no capítulo	241
10 Uso de DropDownList aninhado, RadioButton e CheckBox	242
10.1 Implementações necessárias para as classes fornecedoras dos DropDownLists	243
10.2 Adaptações nas implementações existentes para o DropDownList	246
10.3 Inserção de um controle RadioButton em Fabricantes	255
10.4 Inserção de um controle CheckBox em Fabricantes	256
10.5 Conclusão sobre as atividades realizadas no capítulo	257
11 Os estudos não param por aqui	258

Versão: 21.7.7

CAPÍTULO 1

A PRIMEIRA APLICAÇÃO ASP.NET MVC 5

Olá! Seja bem-vindo ao primeiro capítulo deste livro. Nele buscarei ser prático, apresentando logo no início implementações e exemplos que lhe darão o subsídio inicial para o desenvolvimento de uma aplicação ASP.NET MVC.

Embora o foco da obra seja a prática, não há como fugir de determinados conceitos e teorias e, quando surgirem essas necessidades, estas apresentações ocorrerão. Neste primeiro capítulo, você implementará uma aplicação que permitirá o registro e manutenção em dados de Categorias de um produto, mas sem fazer acesso a base de dados. Sendo assim, como o próprio título deste primeiro capítulo diz, vamos começar já criando nossa primeira aplicação. Caso tenha interesse, seguem os links para meus trabalhos anteriores.

- <http://www.casadocodigo.com.br/products/livro-c-sharp>
- [http://www.visualbooks.com.br/shop/MostraAutor.asp?
proc=191](http://www.visualbooks.com.br/shop/MostraAutor.asp?proc=191)

Como o livro é sobre ASP.NET MVC e nossa primeira aplicação fará uso, obviamente, do ASP.NET MVC. Neste

momento, é importante entender o que o ASP.NET MVC é e o que ele não é, antes da primeira prática.

O que é o ASP.NET MVC?

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

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

Um ponto básico, mas penso ser interessante reforçar, é que ao desenvolver uma aplicação tendo a internet como plataforma, independente de se utilizar o ASP.NET 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 o usuário, navegador ou um sistema; serviço pode ser o registro de uma venda; e recurso pode ser uma imagem, arquivo ou uma página HTML. Esta 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 este processo. Ela foi retirada de um artigo que o aborda mais detalhadamente. Acesse-o pelo link <http://www.devmedia.com.br/como-funcionam-as-aplicacoes-web/25888>.

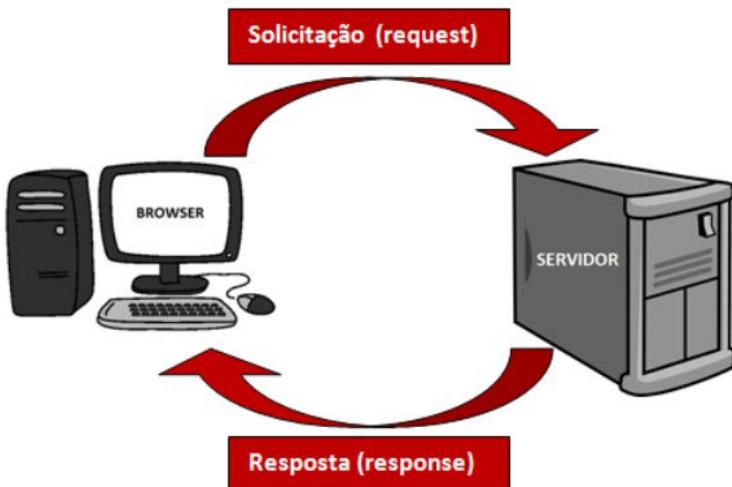


Figura 1.1: Esboço do processo de requisição e resposta de uma aplicação web

Podemos definir o ASP.NET MVC como um framework que possibilita, em sua estrutura, 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 MVC, o padrão MVC não será aplicado. Veremos esta modularização no capítulo 5. *Separando a aplicação em camadas.*

Vamos à implementação. Para os projetos usados neste livro, fiz uso do Visual Studio 2015 Community, que você pode obter em <https://www.visualstudio.com/pt-br/downloads/download-visual-studio-vs.aspx>. Na instalação, marque a opção **SQL Server Data Tools, pois precisaremos desta ferramenta.** :-) Então, mãos à obra.

1.1 CRIANDO O PROJETO NO VISUAL STUDIO 2015 COMMUNITY

Toda aplicação possui, em seu modelo de negócios, alguns domínios que precisam ser persistidos e que ofereçam algumas funcionalidades. Estes domínios, normalmente, também estão ligados a módulos (ou pacotes) ou ao sistema como um todo, que está sendo desenvolvido.

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 Categorias de um produto.

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

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

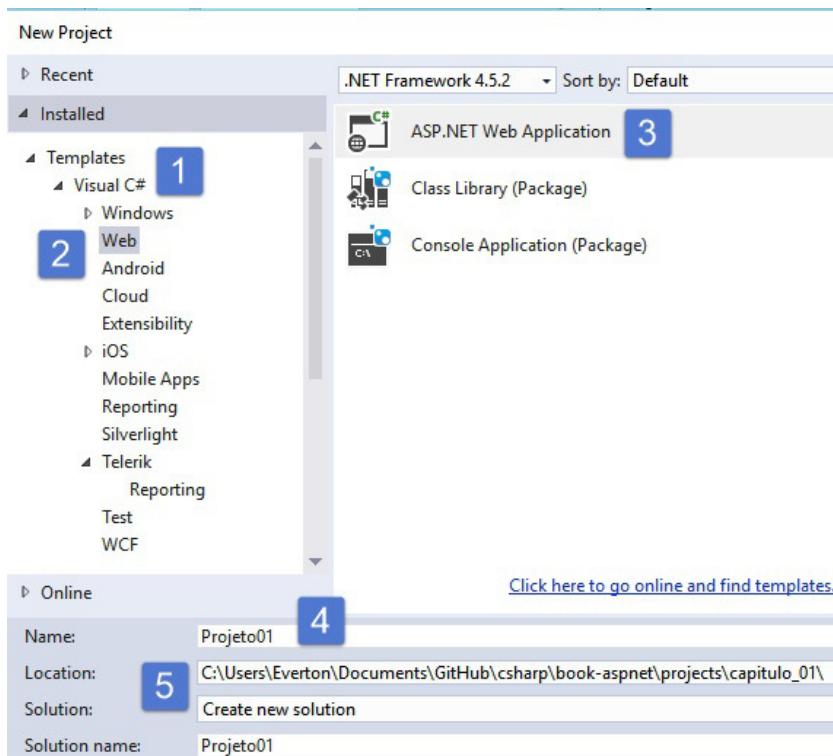


Figura 1.2: Janela do Visual Studio para criação de um projeto web

Na nova janela que se abre é preciso selecionar qual template de uma aplicação web deverá ser criado. Selecione Empty (1) e marque a opção MVC (2). Valide sua janela com a figura a seguir. Clique no botão OK para confirmar a seleção e o projeto poder ser

criado.

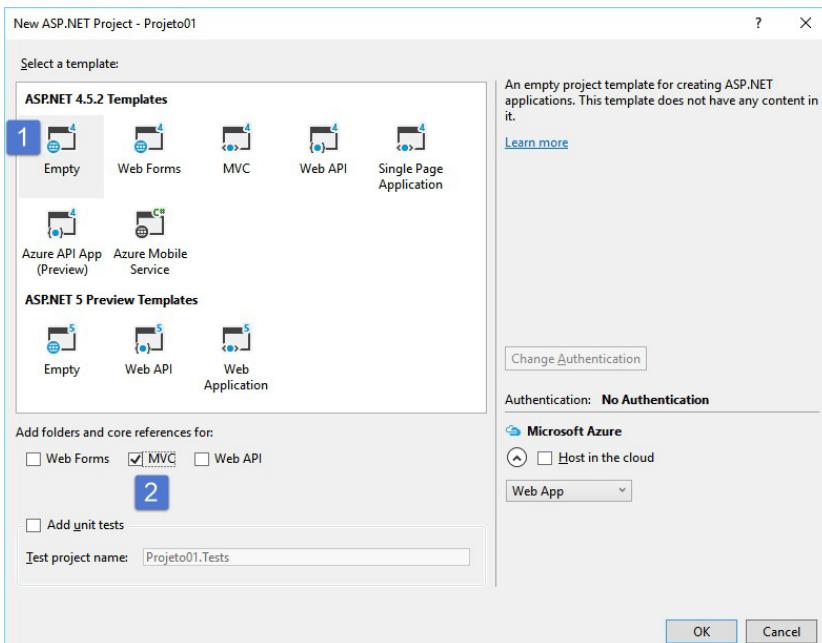


Figura 1.3: Janela de seleção do tipo de projeto web que deverá ser criado

Após a criação ter sido realizada, é possível visualizar a estrutura do projeto criado na janela do Solution Explorer , como mostra a figura a seguir. Veja a criação das pastas **Controllers** , **Models** e **Views** .

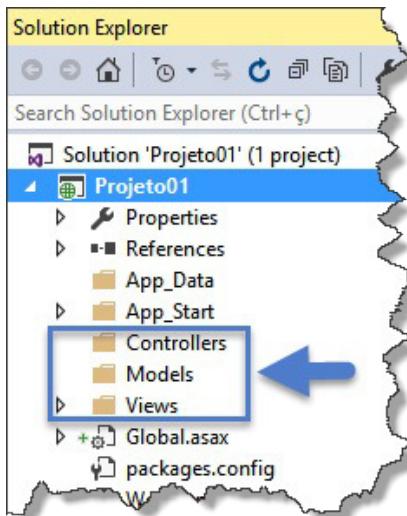


Figura 1.4: Solution Explorer com a estrutura criada para o projeto

No ASP.NET MVC, uma requisição é direcionada a uma Action . Esta nada mais é do que um método de uma classe que representa um determinado controlador. Um Controller é uma classe que representa um determinado Modelo de sua aplicação (ou domínio). É no controlador que são implementados os serviços ofertados (não no contexto de web service), ou seja, as actions.

O objetivo deste livro não é a criação de uma aplicação completa, mas sim apresentar recursos e técnicas que poderão ser utilizados no desenvolvimento de qualquer aplicação.

Como primeiro ponto do domínio a ser trabalhado, temos as Categorias dos produtos , que será representada pela classe Categoria no modelo. Neste primeiro momento, ela possuirá apenas a propriedade Nome , e precisará de serviços para:

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

Em um projeto real, é preciso avaliar a alteração e remoção de dados, pois pode-se 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, estes serviços são oferecidos por actions do controller para o domínio. Estas actions 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. Particularmente, vejo os templates do Visual Studio como uma ferramenta para investigar alguns dos recursos oferecidos pela plataforma. Mas eu apresentarei estes recursos aqui para você. :-)

1.2 CRIANDO O CONTROLADOR PARA CATEGORIAS DE PRODUTOS

Na Solution Explorer, clique com o botão direito do mouse sobre a pasta `Controllers` e clique na opção `Add->Controller`. Na janela que se abre, selecione o template `MVC 5 Controller - Empty`, conforme pode ser verificado pela figura a seguir. Para iniciar o processo de criação do controlador, clique no botão `Add`. Na janela que se apresenta, solicitando o nome do controlador, digite `Categorias` e mantenha o sufixo

`Controller`, pois faz parte da convenção do ASP.NET MVC.



Figura 1.5: Janela de seleção do tipo de controlador a ser criado

Após a conclusão da criação do controller, o Visual Studio o exibe. Note, no código a seguir, que a classe estende `Controller` e a action `Index()` retorna um `ActionResult`.

```
using System.Web.Mvc;

namespace Projeto01.Controllers
{
    public class CategoriasController : Controller
    {
        // GET: Categorias
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Controller

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

1. Toda classe `Controller` deve ter seu nome finalizado com

o sufixo `Controller`. Esta obrigatoriedade se deve ao fato de que o framework busca por este sufixo. O artigo "*But I don't want to call Web Api controllers "Controller"!*" (<http://www.strathweb.com/2013/02/but-i-dont-want-to-call-web-api-controllers-controller/>) traz possibilidades de alteração do sufixo, mas não vejo ganhos com isso.

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 apenas com uso de `Attributes`, que causem a desambiguidade. Se quiser maiores detalhes sobre como sobre carregar métodos de um controlador, sendo `action` ou não, veja o artigo *Overloading Controller Actions In ASP.NET MVC* (<http://www.binaryintellect.net/articles/8f9d9a8f-7abf-4df6-be8a-9895882ab562.aspx>).

ActionResult

Um objeto da classe `ActionResult` representa o retorno de uma `action`. Existem diversas classes que estendem a `ActionResult`, e que representam um nível maior de especialização. Elas serão apresentadas conforme forem sendo usadas. No exemplo do código anterior, como o retorno é uma `View`, a `ActionResult` poderia ser substituída por `ViewResult`.

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 envio dos dados que serão utilizados na `view` que será renderizada.

1.3 CRIANDO A CLASSE DE DOMÍNIO PARA CATEGORIAS DE PRODUTOS

Com a estrutura do controlador criada, é preciso definir o modelo que será manipulado por ele. No código a seguir, podemos visualizar a classe `Categoria`, que deve ser criada na pasta `Models`.

Para a criação, clique com o botão direito do mouse sobre a pasta, e então em `Add->Class`. Dê à classe o nome de `Categoria` e confirme a criação. Verifique que no código da classe, além da propriedade `Nome` que pertence ao domínio, foi implementada a `CategoriaId`, que terá a funcionalidade de manter a identidade de cada objeto.

```
namespace Projeto01.Models
{
    public class Categoria
    {
        public long CategoriaId { get; set; }
        public string Nome { get; set; }
    }
}
```

1.4 IMPLEMENTANDO A INTERAÇÃO DA ACTION INDEX COM A VISÃO

Em um projeto real, teríamos neste momento o acesso a dados (via um banco de dados), no qual as categorias 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 esta implementação.

Note, logo no início da classe, a declaração `private static IList<Categoria> categorias` com sua inicialização com 5 categorias. 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 `categorias`.

```
using Projeto01.Models;
using System.Collections.Generic;
using System.Web.Mvc;

namespace Projeto01.Controllers
{
    public class CategoriasController : Controller
    {
        private static IList<Categoria> categorias =
            new List<Categoria>()
        {
            new Categoria() {
                CategoriaId = 1,
                Nome = "Notebooks"
            },
            new Categoria() {
                CategoriaId = 2,
                Nome = "Monitores"
            },
            new Categoria() {
                CategoriaId = 3,
                Nome = "Impressoras"
            }
        }
    }
}
```

```

    },
    new Categoria() {
        CategoriaId = 4,
        Nome = "Mouses"
    },
    new Categoria() {
        CategoriaId = 5,
        Nome = "Desktops"
    }
};

// GET: Categorias
public ActionResult Index()
{
    return View(categorias);
}
}
}

```

Muito bem, nosso passo seguinte agora é implementar a visão, ou seja, a página HTML que apresentará ao usuário as categorias registradas na aplicação. Talvez você tenha notado que, ao criar o controlador, foi criada uma pasta chamada `Categorias` na pasta `Views`, que é o nome dado ao controlador. Se você não notou, verifique agora.

Nós poderíamos criar o arquivo que representará a visão `Index` diretamente nesta pasta, mas faremos uso de recursos do Visual Studio para isso. A adoção deste critério aqui se deve ao fato de que escrever códigos HTML é algo muito trabalhoso e, em relação ao template utilizado 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 `Add View`. Na janela que se abre, note que o nome da visão já vem preenchido

(1); no Template a ser usado, opte pelo List (2); em Model class , selecione a classe Categoria (caso ela não apareça, o build não foi executado de maneira correta) (3); desmarque a caixa Use a layout page (4); e clique no botão Add para que a visão seja criada.

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

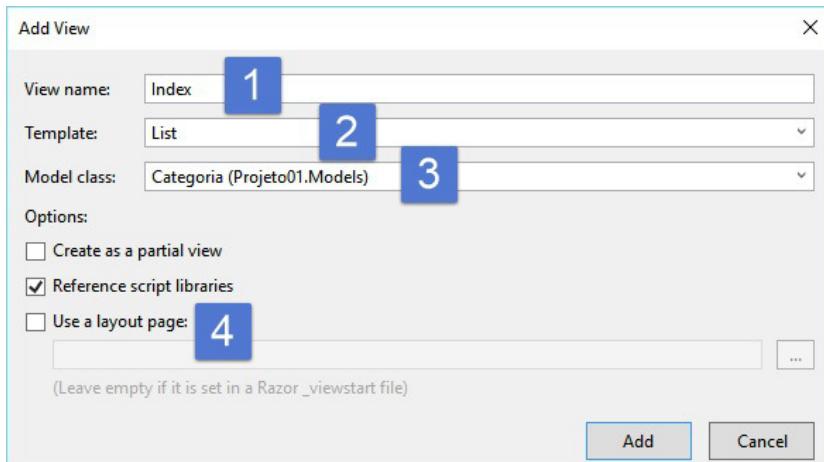


Figura 1.6: Janela para criação da visão

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<Projeto01.Models.Categoria> , define o que a visão recebe e que será definido como Modelo para ela.

A implementação @Html.ActionLink("Create New", "Create") faz uso de um *HTML Helper* (ActionLink()) para que seja renderizado um código HTML que representa um link para uma action. A terceira implementação,

`@Html.DisplayNameFor(model => model.Nome)`, faz uso de outro *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 e última implementação relevante faz uso de um `foreach` para renderização das linhas da tabela, que representarão as categorias recebidas pela visão.

Veja que, para cada linha renderizada, três links são também renderizados para: alteração, exclusão e visualização. Estes são direcionados para actions padrões no ASP.NET MVC para estas operações. Para a exibição do nome de cada categoria, é usado o `Html.DisplayFor()`. Na sequência, veja o código completo da visão gerada.

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

{@
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport"
        content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <p>
        @Html.ActionLink("Create New", "Create")
    </p>
    <table class="table">
        <tr>
            <th>
                @Html.DisplayNameFor(
```

```

        model => model.Nome)
    </th>
    <th></th>
</tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Nome)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit",
                new { id=item.CategoriaId }) |
            @Html.ActionLink("Details", "Details",
                new { id=item.CategoriaId }) |
            @Html.ActionLink("Delete", "Delete",
                new { id=item.CategoriaId })
        </td>
    </tr>
}
</table>
</body>
</html>
```

Razor

O ASP.NET Razor é uma *view engine* (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 de código, cada instrução deve ser finalizada com `;` (ponto e vírgula), e é possível fazer uso de variáveis para armazenamento de valores.

`@model`

A declaração de `@model` no início da visão pode ser vista

como uma analogia a declaração de um método e seus parâmetros. Ela habilita o intellisense a conhecer qual tipo de dados estamos usando quando o modelo for utilizado por *HTML Helpers*. Também verifica, em tempo de execução, se o tipo passado para a visão pode ser convertido para o tipo esperado.

HTML Helpers

Os *HTML Helpers* são métodos que possibilitam a renderização de controles HTML nas visões. Existem diversos deles oferecidos pelo ASP.NET MVC e, conforme forem sendo usados, terão sua explicação apresentada.

É possível também a implementação de *HTML Helpers* personalizados. Normalmente, o retorno destes é uma String. No capítulo 7. *Areas, autenticação e autorização*, criaremos métodos que serão *HTML Helpers*.

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 utilizada no exemplo recebe duas Strings: a primeira com o texto que será exibido como link, e a segunda que recebe o nome da action que será requisitada. Essa action precisa estar no mesmo controlador da URL.

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 valores de rota e atributos HTML. Uma delas pode ser verificada

nos `ActionLinks` do `foreach`, que enviam o `id` de cada categoria para compor a URL do link (mais conhecido como rota).

Html.DisplayNameFor()

Este método obtém o nome para exibição para o campo do modelo. Normalmente, este nome é o nome da propriedade, 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 6. *Code First Migrations, Data Annotations, validações e jQueryUI*.

Html.DisplayFor()

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

Com a implementação da action `Index` e de sua respectiva visão, vamos agora testar e verificar se está tudo correto. 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:50827/Categorias/Index> — podendo, em sua máquina, variar a porta do `localhost`.

Tente em sua máquina digitar a URL sem o nome da action e 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 `Categorias`. Eu não traduzi os textos exibidos nos links criados,

mas é interessante que você o faça. Fica como uma atividade. :-)

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 esta opção na barra de tarefas.

[Create New](#)

Nome
Notebooks Edit Details Delete
Monitores Edit Details Delete
Impressoras Edit Details Delete
Mouses Edit Details Delete
Desktops Edit Details Delete

Figura 1.7: Página HTML que representa a visão Index, retornada pela action Index do controlador Categorias

1.5 O CONCEITO DE ROTAS DO ASP.NET MVC

Uma URL em uma aplicação ASP.NET MVC é formada por segmentos, que compõem rotas. Cada componente tem um significado e um papel que é executado quando ocorre o processo de requisição. Na URL <http://localhost:50827/Categorias/Index>, são encontrados os seguintes segmentos:

- `http` , que representa o protocolo a ser utilizado;
- `localhost:50827` , que são o servidor e porta de comunicação, respectivamente;
- `Categorias` , que representa o controlador que receberá a

requisição; e

- `Index` , que é a action que atenderá a requisição.

Tudo isso é configurável, mas a recomendação é que se mantenha o padrão, salvo necessidade. Esta configuração de rotas pode ser vista na classe `RouteConfig` , que está implementada em um arquivo de mesmo nome, na pasta `App_Start` . Seu código pode ser verificado na listagem a seguir.

```
using System.Web.Mvc;
using System.Web.Routing;

namespace Projeto01
{
    public class RouteConfig
    {
        public static void RegisterRoutes(
            RouteCollection routes)
        {
            routes.IgnoreRoute(
                "{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional })
        }
    }
}
```

Observe no código anterior, na implementação `routes.MapRoute()` , que é dado um nome para a rota (`Default`) e uma máscara para a URL (`{controller}/{action}/{id}`). O terceiro parâmetro para este método refere-se a valores que serão assumidos para o caso de ausência de valores na requisição.

Por exemplo, se requisitarmos <http://localhost:50827>, o controlador que será usado será o `Home`. Se requisitarmos <http://localhost:50827/Categorias>, a action que será 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 deste terceiro argumento pode ser verificado na listagem das categorias. Passe o mouse sobre o link `Edit` da primeira linha de categorias, e você verá <http://localhost:50827/Categorias/Edit/1>, onde `1` representa o valor do campo `CategoriaId` do objeto. Verifique o link dos demais registros e links.

Como atividade, para teste, altere o controlador padrão de `Home` para `Categorias`, e execute novamente sua aplicação.

1.6 IMPLEMENTANDO A INSERÇÃO DE DADOS NO CONTROLADOR

Na visão `Index`, logo no início, existe o HTML Helper `@Html.ActionLink("Create New", "Create")`, traduza-o para `@Html.ActionLink("Criar nova categoria", "Create")`. 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 método `GET` do HTTP, ou seja, por meio da URL. Quando fizermos uso de formulários HTML, faremos uso também de chamadas por meio do método `POST` do HTTP.

```
// 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 `Add View...`. Na janela que se apresenta e que já é conhecida, mantenha o nome da visão como `Create`, escolha o template `Create`, selecione `Categoria` como classe de modelo, e desmarque a opção `Use a layout page`. Clique no botão `Add` para que a visão seja criada.

A listagem criada para a visão `Create` é exibida a seguir:

```
@model Projeto01.Models.Categoria

{@
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport"
        content="width=device-width" />
    <title>Create</title>
</head>
<body>
    <script src="~/Scripts/jquery-1.10.2.min.js">
        </script>
    <script src="~/Scripts/jquery.validate.min.js">
        </script>
    <script src="~/Scripts/jquery.validate.unobtrusive.min.js">
        </script>

    @using (Html.BeginForm())
    {
        @Html.AntiForgeryToken()
```

```

<div class="form-horizontal">
    <h4>Categoria</h4>
    <hr />
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(model => model.Nome,
                      htmlAttributes: new { @class =
                        "control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.Nome,
                            new { htmlAttributes =
                            new { @class = "form-control" }
                            })
            @Html.ValidationMessageFor
                (model => model.Nome, "", new { @class = "text-danger" })
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit"
                  value="Adicionar categoria"
                  class="btn btn-default" />
        </div>
    </div>
</div>
<div>
    @Html.ActionLink("Retornar para a listagem",
                    "Index")
</div>
</body>
</html>

```

As implementações dos `<script>` referente ao jquery possibilitarão a validação dos dados no navegador (conhecida como validação no cliente). Neste momento, não trabalharemos com isso e, por este motivo, não detalharei agora, mas sim quando

realmente formos utilizar.

A implementação `@using (Html.BeginForm()) {}` delimita o formulário HTML, que terá como destino, na submissão, uma `action` de nome `Create`, igual à visão. Porém, marcada para ser invocada quando o método `POST` do HTTP for utilizado.

Em algumas tags HTML, podemos verificar a referência à classes de CSS. Estas são definidas pelo Bootstrap, que também detalharei quando formos usá-lo (adiante, no capítulo 3. *Layouts, Bootstrap e jQuery DataTable*). No momento, elas não terão efeito algum, pois não estamos nos referenciando a nenhum CSS na visão. Logo veremos isso também.

A implementação `@Html.ValidationSummary(true, "", new { @class = "text-danger" })` renderizará na visão todos os erros de validação, do modelo, que ocorrerem no momento da submissão do formulário para o servidor. Se a validação no cliente estiver ativa, também exibirá os erros, sem que a submissão precise ser feita.

A implementação `@Html.ValidationMessageFor(model => model.Nome, "", new { @class = "text-danger" })` também apresenta uma mensagem de erro, porém de forma individualizada, abaixo do controle que acusou problemas referentes à validação.

O HTML Helper `@Html.LabelFor()` renderizará o nome da propriedade retornada pela expressão lambda que, a princípio, é o nome da propriedade, caso não tenha sido feito uso de Data Annotations. Veremos isso em breve, no capítulo 6. *Code First Migrations, Data Annotations, validações e jQueryUI*. Já o HTML

Helper `@Html.EditorFor()` gerará um controle de edição para a propriedade em questão, e o controle será de acordo ao tipo de dado da propriedade. Neste nosso exemplo, um `<input type="text"/>`.

Por fim, temos o HTML `<input type="submit" value="Create" class="btn btn-default" />` que renderizará o botão de submissão e, no final do arquivo, um HTML Helper que gerará um link que remeterá o usuário à action `Index`. A figura a seguir apresenta a visão gerada.

Categoria

Nome

[Adicionar categoria](#)

[Retornar para a listagem](#)

Figura 1.8: Página HTML que representa a visão Create, retornada pela action Create do controlador Categorias

Na sequência, veja as definições dos HTML Helpers trabalhados pela primeira vez.

Html.BeginForm()

Este helper escreve a tag de abertura `<form>`, configurando a tag `action` para uma action específica de um controlador específico. Na visão `Create` que apresentei anteriormente, foi feito uso da assinatura padrão, na qual a `action` possui o mesmo nome da visão renderizada, para o mesmo controlador também. Quando se utiliza este helper com o bloco `using`, também é

renderizada a tag de fechamento `</form>` . O método do HTTP padrão é `POST` .

Html.AntiForgeryToken()

Da mesma maneira que estamos criando uma requisição HTTP fazendo uso do método `POST` , outros sites e aplicações também podem fazê-lo, inclusive direcionando suas requisições para nossas actions, o que, certamente, causaria um problema de segurança, conhecido como *Cross Site Request Forgery*. Para evitar este problema, fazemos uso do `Html.AntiForgeryToken()` , pois ele cria um `cookie` com os códigos de AntiForgery e insere o código mais recente como campo oculto na visão.

Para que isso funcione, precisamos fazer uso do atributo `[ValidateAntiForgeryToken]` na action que receberá a requisição. Caso seja de seu interesse, acesse o link <http://www.devcurry.com/2013/01/what-is-antiforgerytoken-and-why-do-i.html> para um detalhamento sobre este assunto.

Html.ValidationSummary()

Este helper renderiza uma lista não ordenada (elemento HTML ``) para as mensagens que estejam armazenadas em um objeto `ModelStateDictionary` . Nós ainda não vimos como essas mensagens de erro são geradas, pois é preciso escrever algumas regras de validação para as propriedades do objeto que é o modelo da visão, mas logo veremos tudo isso.

Este método possui sobrecargas e, na versão que estamos utilizando, o primeiro argumento refere-se à exibição apenas de erros do modelo. O segundo, que está representado por uma string

vazia (""), renderiza uma mensagem a ser exibida antes do erro. Já o terceiro insere configurações específicas para o HTML que será renderizado.

Html.LabelFor()

Este helper renderiza um rótulo para a propriedade pela expressão lambda fornecida. Normalmente, como em nosso exemplo, é uma propriedade do modelo da visão. Como pode ser visto em nosso exemplo, é também possível enviar para o método configurações para o elemento HTML que será renderizado.

@Html.EditorFor()

Este helper renderiza um controle de interação com o usuário, onde ele informará valores solicitados. O tipo de controle a ser renderizado varia de acordo ao tipo da propriedade especificada na expressão lambda. Também é possível configurar os atributos HTML para o controle renderizado.

@Html.ValidationMessageFor()

Este helper renderiza a mensagem de erro específica para o controle representado pela expressão lambda, caso haja erro. Também permite a configuração do elemento HTML.

Implementando a action que recebe o modelo para inserção

Como já temos implementadas a action que gera a visão e a visão que receberá os dados para o modelo, precisamos agora, no controlador `Categorias`, implementar uma nova action. Esta

será responsável por receber os dados informados na visão.

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

Veja na sequência a listagem da action criada. Note os atributos `[HttpPost]` e `[ValidateAntiForgeryToken]`. Observe que o parâmetro do método é um objeto `Categoria`. Isso é possível graças à declaração do `@model` na visão. Na inserção, submetemos apenas o nome para a categoria, mas o objeto precisa de um identificador para a propriedade `CategoriaId`.

Desta maneira, fazemos uso do LINQ para obtermos o valor máximo da coleção para esta propriedade, o incrementamos em um, e atribuímos o resultado à propriedade. Para que possamos trabalhar com o LINQ, precisamos incluir no início da classe o trecho `using System.Linq;`.

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 deste conjunto de dados, 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.

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 que será requisitada antes da resposta ser gerada.

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

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Categoria categoria)
{
    categorias.Add(categoria);
    categoria.CategoriaId =
        categorias.Select(m => m.CategoriaId).Max() + 1;
    return RedirectToAction("Index");
}
```

1.7 IMPLEMENTANDO A ALTERAÇÃO DE DADOS NO CONTROLADOR

Como pôde ser notado por meio das atividades criadas até o momento, estamos tratando a implementação de um CRUD (_C_reate, _R_ead, _U_pdate e _D_elete - Criação, Leitura, Atualização e Exclusão) para o modelo `Categoria`. Já temos a leitura de todos os registros pela action `Index`, faltando a leitura individual, que logo implementaremos. Também implementamos, na seção anterior, a inclusão (criação) e, agora, trabalharemos para que a atualização (update) possa ser implementada.

Seguindo o exemplo usado para a inserção de dados, no qual uma operação precisa de uma action `GET` para ser gerada a visão de interação com o usuário, e de outra action (`HTTP POST`) que recebe os dados inseridos pelo usuário, implementaremos inicialmente a `GET` action. Veja a listagem a seguir.

Note que a action recebe um parâmetro, que representa o `id` do objeto que se deseja alterar. Também é possível verificar que o método `View()` recebe agora um argumento, mais uma vez fazendo uso de LINQ. Desta vez, recuperando o primeiro objeto que tenha na propriedade `CategoriaId` o valor recebido pelo parâmetro `id`.

```
public ActionResult Edit(long id)
{
    return View(categorias.Where(
        m => m.CategoriaId == id).First());
}
```

Com o método que retornará a visão para alteração de dados devidamente implementado e enviando para a visão o objeto que se deseja alterar, seguindo os mesmos passos anteriormente 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` é a implementação `@Html.HiddenFor(model => model.CategoriaId)`.

Quando trabalhamos a visão `Create`, o valor de identidade do objeto (propriedade `CategoriaId`) foi inserido pela aplicação, na action `POST Create`. Agora, na alteração, este valor será a chave para que possamos alterar o objeto correto. Mas, pela implementação do Helper `HiddenFor()`, o Id da Categoria não é exibido para o usuário. Então, por que o ter?

A resposta é: para que, ao submeter a requisição, o modelo enviado possua o valor desta propriedade. A action alvo só recebe valores que estejam no modelo da visão. Depois de tudo implementado e testado, retire este método da visão e tente alterar um objeto. Fica esta atividade para você. :-)

```
@model Projeto01.Models.Categoria

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport"
          content="width=device-width" />
    <title>Edit</title>
</head>
<body>
    <script src="~/Scripts/jquery-1.10.2.min.js">
        </script>
    <script src="~/Scripts/jquery.validate.min.js">
```

```

        </script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js">
</script>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Categoria</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.CategoriaId)

        <div class="form-group">
            @Html.LabelFor(model => model.Nome,
                htmlAttributes: new { @class =
                    "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(
                    model => model.Nome, new
                    { htmlAttributes = new {
                        @class = "form-control" } })
                @Html.ValidationMessageFor(
                    model => model.Nome, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit"
                    value="Gravar dados alterados"
                    class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Retornar para a listagem",
        "Index")
</div>
</body>

```

```
</html>
```

Html.HiddenFor()

Este helper renderiza um elemento HTML `<input type="hidden">` para a propriedade retornada pela expressão lambda. É muito comum o uso deste helper para que valores possam ser passados da visão para o controlador, sem a interferência do usuário.

Para finalizar, é preciso criar a action `Edit` que responda à requisição HTTP `POST`. Na sequência, está apresentado o código. Execute sua aplicação, acesse o link `Edit` da visão `Index` de um dos produtos (renomeie o link para `Alterar`). Altere o nome do produto, grave e veja a nova listagem. O produto com o nome atualizado aparecerá no final dela.

É possível alterar a listagem dos produtos para que sejam listados em ordem alfabética. Para isso, na action `Index`, adapte a implementação para: `return View(categorias.OrderBy(c => c.Nome));`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(Categoria categoria)
{
    categorias.Remove(categorias.Where(
        c => c.CategoriaId == categoria.CategoriaId)
        .First());
    categorias.Add(categoria);
    return RedirectToAction("Index");
}
```

Uma maneira alternativa para alterar um item da lista, sem ter de removê-lo e inseri-lo novamente, é fazer uso da implementação `categorias[categorias.IndexOf(categorias.Where(c =>`

```
c.CategoriaId == categoria.CategoriaId).First())] = categoria; Aqui o List é manipulado como um array e, por meio do método IndexOf() , sua posição é recuperada, com base na instrução LINQ Where(c => c.CategoriaId == categoria.CategoriaId).First()) .
```

1.8 IMPLEMENTANDO A VISUALIZAÇÃO DE UM ÚNICO REGISTRO

A visão Index traz para o 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 em questão, 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 de determinado cliente.

Para isso, há um link na listagem, chamado Details (traduzida 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(categorias.Where(
        m => m.CategoriaId == id).First());
}
```

A criação de visões já foi aplicada e praticada, logo, você já sabe como criá-las. Mas para a criação da visão Details , vai mais uma ajudinha: clique com o botão direito do mouse sobre o nome da

action `Details` , confirme a criação da visão, mantenha o nome `Details` , selecione o modelo `Details` e a classe de modelo `Categoria` . 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 Projeto01.Models.Categoria

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport"
          content="width=device-width" />
    <title>Details</title>
</head>
<body>
    <div>
        <h4>Categoria</h4>
        <hr />
        <dl class="dl-horizontal">
            <dt>
                @Html.DisplayNameFor(
                    model => model.Nome)
            </dt>
            <dd>
                @Html.DisplayFor(model => model.Nome)
            </dd>
        </dl>
    </div>
    <p>
        @Html.ActionLink("Alterar", "Edit",
                         new { id = Model.CategoriaId }) |
        @Html.ActionLink("Retornar para a listagem",
                        "Index")
    </p>
</body>
</html>
```

Note que os links oferecidos nesta visão já foram implementados anteriormente, que são o referente à alteração (`Edit`) e o que exibe toda a listagem de `Categoria`, que é a action `Index`. Não existe nada de novo no código gerado.

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

1.9 FINALIZANDO A APLICAÇÃO POR MEIO DA IMPLEMENTAÇÃO DA OPERAÇÃO DELETE DO CRUD

Quando se utiliza os templates do Visual Studio para a criação de uma aplicação ASP.NET MVC, ele traz a operação de *delete* de uma maneira que os dados sejam 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, igual às actions `Edit` e `Details`. Veja o código na sequência:

```
public ActionResult Delete(long id)
{
    return View(categorias.Where(
        m => m.CategoriaId == 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 aparece neste código o HTML Helper `BeginForm()`, porém, diferentemente das visões `Create` e `Edit`, ele encapsula apenas o elemento HTML

<input type="submit"> . Não existe neste código nada de novo, além desta observação.

```
@model Projeto01.Models.Categoria

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport"
          content="width=device-width" />
    <title>Delete</title>
</head>
<body>
    <h3>Confirma o desejo de excluir a categoria abaixo?</h3>
    <div>
        <h4>Categoria</h4>
        <hr />
        <dl class="dl-horizontal">
            <dt>
                @Html.DisplayNameFor(
                    model => model.Nome)
            </dt>

            <dd>
                @Html.DisplayFor(model => model.Nome)
            </dd>
        </dl>

        @using (Html.BeginForm()) {
            @Html.AntiForgeryToken()

            <div class="form-actions no-color">
                <input type="submit"
                      value="Excluir categoria"
                      class="btn btn-default" /> |
                @Html.ActionLink(
                    "Retornar para a listagem",
                    "Index")
            </div>
        }
    </div>
```

```
        }
    </div>
</body>
</html>
```

Para finalizar a aplicação, precisamos implementar o método que realmente removerá uma categoria. Esta implementação se dá por meio da action `Delete`, capturada por uma requisição HTTP `POST`. Veja o código na sequência. Infelizmente, o objeto que chegará para a action estará com as propriedades nulas, pois não existe nenhum HTML Helper que as utilize dentro do formulário HTTP (tente executar sua aplicação e veja que nada será excluído).

Como o único valor necessário para a remoção da categoria é o `id` dela, e não é de interesse que ele seja exibido ao usuário, insira, abaixo do `Html.AntiForgeryToken()` a implementação `@Html.HiddenFor(model => model.CategoriaId)`. Teste novamente sua aplicação, verificando se a visão que exibe os dados do objeto a ser removido é exibida. Ela aparecendo, confirme a remoção do objeto e veja se a visão `Index` é renderizada e sem o objeto que foi removido.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(Categoria categoria)
{
    categorias.Remove(categorias.Where(
        c => c.CategoriaId == categoria.CategoriaId)
        .First());
    return RedirectToAction("Index");
}
```

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 MVC. É uma aplicação simples, entretanto o foco não foi na complexidade, mas sim no processo.

Foram apresentados conceitos de controllers, actions, views, razor e roteamento. Fazendo uso de 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

REALIZANDO ACESSO A DADOS NA APLICAÇÃO ASP.NET MVC COM O ENTITY FRAMEWORK

O Entity Framework, ou apenas EF, é um framework para mapeamento de objetos para um modelo relacional e de um modelo relacional para objetos (ORM — *Object Relational Mapping*). Por meio do EF, é possível trabalhar com dados relacionais fazendo uso de objetos da camada de negócio, eliminando, desta maneira, a codificação de acesso a dados diretamente na aplicação, como, por exemplo, o uso explícito de instruções SQL.

Com estas características apresentadas, este capítulo tem por objetivo introduzir o Entity Framework 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 tema único de diversos livros.

Além disso, como o EF será usado em todos os exemplos, a partir deste capítulo, novas características serão apresentadas

conforme forem sendo necessárias. Faremos uso do SQL Server LocalDB, que é uma versão reduzida (porém com muitos recursos) do SQL SERVER EXPRESS, e que é instalado em conjunto com o Visual Studio Community.

Em relação ao projeto começado no capítulo anterior, concluímos um CRUD para Categorias, mas nada foi persistido em uma base de dados. Ou seja, se a aplicação for parada, os dados serão perdidos. Com o Entity Framework, este problema será resolvido, pois poderemos persistir nossos objetos em uma base de dados por meio dele.

2.1 COMEÇANDO COM O ENTITY FRAMEWORK

Para iniciar as atividades relacionadas ao desenvolvimento fazendo uso do EF, o primeiro passo é obter os recursos necessários para utilizá-lo. A primeira ferramenta a ser disponibilizada no Visual Studio é o EF Tools for Visual Studio, que já vem instalada na versão 2015.

Dando sequência ao nosso projeto, iniciado no capítulo anterior, crie uma nova classe, chamada `Fabricante`, conforme código apresentado na sequência. Lembre-se de que, a princípio, esta classe deverá ser criada na pasta `Models`.

```
namespace Projeto01.Models
{
    public class Fabricante
    {
        public long FabricanteId { get; set; }
        public string Nome { get; set; }
    }
}
```

A segunda ferramenta necessária é a EF Runtime, que deve ser disponibilizada nas referências (References) do projeto, como um Pacote/Assembly (em uma DLL). Para adicionar a referência ao EF Runtime, o meio mais simples é fazer uso do NuGet, que instalará no projeto selecionado o EF NuGet Package.

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

Como a criação deste contexto faz uso da classe `System.Data.Entity.DbContext`, e dispõe uma propriedade do tipo `DbSet< TEntity >` para cada classe que deverá ser representada na base de dados, é preciso adicionar a referência ao NuGet Package do Entity Framework.

Desta maneira, para que o projeto em desenvolvimento possa fazer uso do EF, clique, com o botão direito do mouse sobre o nome do projeto, e então na opção `Manage NuGet Packages`. Na janela exibida, é preciso pesquisar pelo Entity Framework, caso ele não apareça de imediato. Tendo selecionado o EF, ao clicar nele, basta pressionar o botão `Install` para que a instalação seja realizada. Siga as instruções que aparecerão.

A figura a seguir apresenta a janela do NuGet:

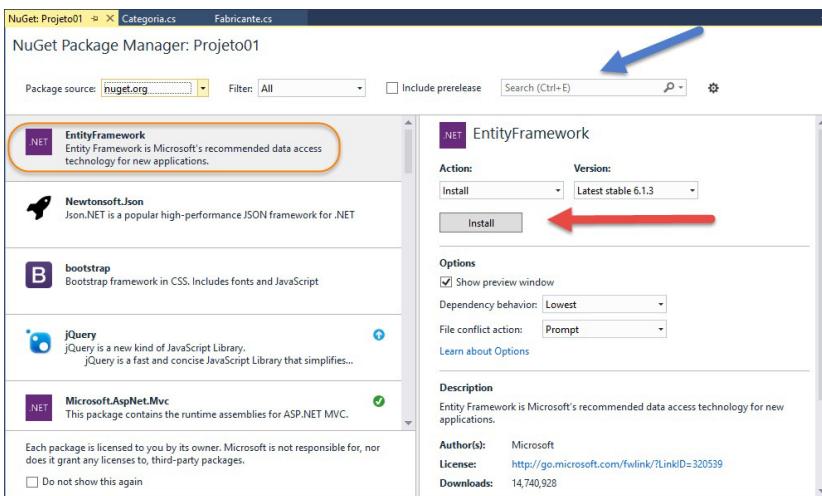


Figura 2.1: Janela do NuGet para instalação do Entity Framework

Após a conclusão da instalação, verifique a adição às referências do projeto dos Assemblies do Entity Framework, conforme exibido na figura a seguir.

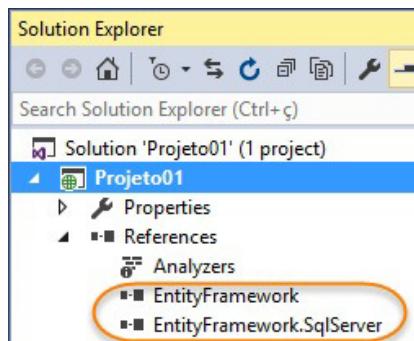


Figura 2.2: Visualização das referências do Entity Framework instaladas no projeto

NuGet

O NuGet é um gerenciador de pacotes para desenvolvimento na plataforma Microsoft, o que inclui o desenvolvimento de aplicações ASP.NET MVC. Como é uma ferramenta que dispõe os pacotes a serem utilizados na aplicação em um servidor remoto, é oferecido também ferramentas clientes do NuGet, que permitem produzir e consumir pacotes NuGets.

Criando o contexto com a base de dados

Para que nossa aplicação possa se beneficiar do Entity Framework, é preciso que ele acesse a base de dados por meio de um contexto, que representará uma sessão de interação da aplicação com a base de dados, seja para consulta ou atualização. Para o EF, um contexto é uma classe que estende `System.Data.Entity.DbContext`.

Desta maneira, crie em seu projeto uma pasta chamada `Contexts` e, dentro dela, uma classe `EFContext.cs`, que deverá ter esta extensão implementada. Veja o código a seguir. Note que o construtor estende a execução do construtor da classe base, invocando `base("Asp_Net_MVC_CS")`. A string enviada como argumento refere-se ao nome de uma `Connection String`, que deve ser configurada no arquivo `Web.config`.

```
using Projeto01.Models;
using System.Data.Entity;

namespace Projeto01.Contexts
{
```

```
public class EFContext : DbContext
{
    public EFContext() : base("Asp_Net_MVC_CS") {}

    public DbSet<Categoria> Categorias { get; set; }
    public DbSet<Fabricante> Fabricantes
    { get; set; }
}
```

Com o contexto para o EF já implementado, precisamos agora configurar a aplicação para acessar a base de dados por meio da Connection String enviada ao construtor base da classe EFContext , o Asp_Net_MVC_CS . Esta implementação deve ser realizada no arquivo Web.config , e deve ser implementada abaixo do elemento <appSettings> .

No momento, criaremos todos os recursos no projeto MVC, mas logo começaremos a tratar uma arquitetura para ser implementada em camadas. O elemento <add> é escrito em uma única linha e não separado, como mostra o código na sequência.

```
<connectionStrings>
    <add name="Asp_Net_MVC_CS" connectionString="Data Source=(localdb)\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\Asp_Net_MVC_DB.mdf;Integrated Security=True; MultipleActiveResultSets=True;" providerName="System.Data.SqlClient" />
</connectionStrings>
```

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 há na base de dados uma tabela que será manipulada por sua aplicação, por meio do EF, é importante que haja uma propriedade que a represente na definição do contexto. É

importante ressaltar que esta 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 de como acessá-la. Ela passa, por meio de código, 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.

Dentro do elemento `<connectionString>`, há um elemento `<add>`, que adiciona ao contexto da aplicação uma nova connection string, e alguns atributos são definidos:

1. `name` — Define o nome para a conexão a ser adicionada, neste caso `Asp_Net_MVC_CS` ;
2. `connectionString` — Um atributo complexo, em que:
 - `Data Source` é o servidor onde o banco de dados está e, neste caso, é apontado o Local DataBase. O valor `(localdb)\MSSQLLocalDB` refere-se ao banco local instalado junto com o Visual Studio.
 - `AttachDbFileName` refere-se ao caminho físico para o arquivo que representa a base de dados. No exemplo, o `|DataDirectory|` refere-se à pasta `App_Data` do projeto.

- Integrated Security define como a autenticação será utilizada na conexão. Quando recebe `True`, assume-se a autenticação do sistema operacional (Windows no caso) e, se o valor atribuído for `False`, será necessário informar o nome de usuário e senha.
 - `MultipleActiveResultSets` permite a execução de múltiplos lotes de instrução em uma única conexão.
3. `providerName` — Fornece para a conexão o nome do Data Provider responsável por realizar a conexão com a base de dados.

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

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

Local Data Base — LocalDB

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

Nosso projeto já está configurado para fazer uso do EF, e para acessar a base de dados que será criada e utilizada por ele. Neste

momento, poderíamos adaptar a classe `Categoria` e seus serviços para fazer uso do Entity Framework, mas optei por usar a classe `Fabricante`, que criamos anteriormente. A adaptação para `Categoria` ficará como atividade.:-)

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

2.2 IMPLEMENTANDO O CRUD FAZENDO USO DO ENTITY FRAMEWORK

Para que possamos implementar as funcionalidades que serão oferecidas para `Fabricante`, precisamos criar um novo controlador, que será chamado `FabricantesController`. Para isso, clique com o botão direito do mouse na pasta `Controllers`, depois em `Add` e `Controller`, e então escolha `MVC 5 Controller - Empty`, tal qual fizemos no capítulo 1. *A primeira aplicação ASP.NET MVC 5.*

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.

Observe, no início da classe, a declaração do campo `context`. Este objeto será utilizado por todas as actions. Na action `Index`, o objeto encaminhado para o método `View()` é agora a coleção de objetos (registros) existentes na coleção `DbSet` (tabela), que

pertence ao objeto context .

```
using Projeto01.Contexts;
using System.Linq;
using System.Web.Mvc;

namespace Projeto01.Controllers
{
    public class FabricantesController : Controller
    {
        private EFContext context = new EFContext();

        // GET: Fabricantes
        public ActionResult Index()
        {
            return View(context.Fabricantes.OrderBy(
                c => c.Nome));
        }
    }
}
```

Com a implementação anterior realizada, partiremos agora para a criação da visão. Crie-a de acordo com as orientações já recebidas quando criamos o controlador para Categorias . 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 EFContext , conforme pode ser visto na figura a seguir. Com exceção às traduções dos links que serão exibidos ao usuário, nenhuma mudança se fará necessária. Você pode, inclusive, notar que esta nova visão é praticamente idêntica à criada para Categorias . Execute sua aplicação e requisite a nova visão. Em minha máquina, a URL é <http://localhost:50827/Fabricantes/Index>.

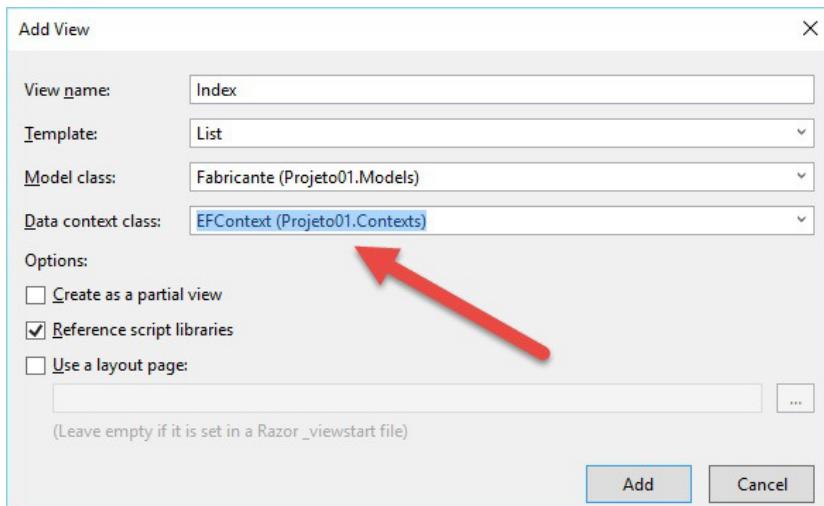


Figura 2.3: Janela de adição de uma visão

É possível que você note uma demora maior no processamento da requisição. Isso se deve ao fato de o Entity Framework estar criando o banco de dados, por ser a primeira vez que a aplicação é executada com o EF instalado e fazendo uso dele. Quando a resposta retornar, nada será exibido, pois a tabela não existia. Encerre a execução de sua aplicação e vamos ver o banco criado.

Após a aplicação ter sido encerrada, na janela Solution Explorer clique no ícone responsável por exibir todos os arquivos, inclusive os que não estão incluídos no projeto. Esse ícone está apontado na figura a seguir. Após clicar nele, expanda a pasta App_Data e veja o arquivo destacado também na figura a seguir. Ele foi criado pelo EF, em uma execução da aplicação, e não pelo Visual Studio. Para inserir este arquivo no projeto, clique nele com o botão direito do mouse e na opção **Include in project**.

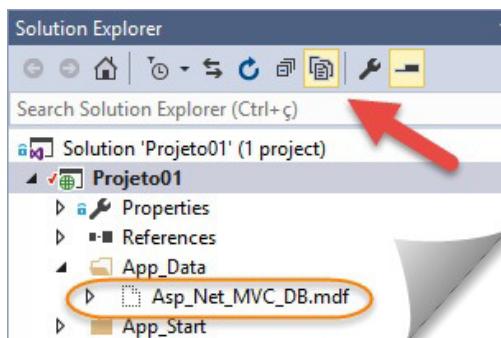


Figura 2.4: Solution Explorer exibindo o arquivo de dados criado pelo EF

Agora, com o arquivo de dados inserido no projeto, dê um duplo-clique nele. A janela do Server Explorer será exibida com a conexão para este arquivo estabelecida. Veja na figura a seguir como deverá estar sua janela.

Clique com o botão direito em sua tabela `Fabricantes` e na opção `Show Table Data`. Na janela que se abre, insira alguns registros. Não precisa do `Id`, pois ele é autoincremento. O EF identifica como chaves as propriedades que possuam `Id` em seu nome. Em nosso caso, temos a `FabricanteId`, que é composta pelo nome da classe e a palavra `Id`. Desta maneira, essa propriedade será a chave primária na tabela a ser criada.

Para o EF, por convenção, o valor para uma chave primária é gerado automaticamente, o que torna o campo autoincremento. Mas isso pode ser modificado por meio de atributos. Execute novamente sua aplicação e requisite a action `Index` de `Fabricantes`. Seus dados agora aparecem na listagem. :-)



Figura 2.5: Server Explorer com a conexão para a base de dados criada

Implementando a inserção de novos dados com o Entity Framework

Se você fez todos os passos anteriores corretamente, já temos uma listagem de todos os fabricantes que estão registrados na base de dados. A inserção dos dados que aparecem foi realizada diretamente na tabela, 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`). Essas actions podem ser vistas nos códigos apresentados na sequência.

Observe que a action que gera a visão (a primeira listagem) é semelhante a que implementamos no capítulo anterior para `Categorias` . A segunda listagem já traz duas instruções diferentes. A primeira insere o objeto recebido no `DbSet Fabricantes` , em seguida é realizada a atualização na base de dados. Essa atualização se dá por meio da chamada ao método `SaveChanges()` do contexto.

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 descrevendo-o será disparada e exibida pelo Visual Studio.

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. A terceira e última instrução redireciona para a action que renderizará a visão de listagem dos dados cadastrados, a `Index`.

```
public ActionResult Create()
{
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Fabricante fabricante)
{
    context.Fabricantes.Add(fabricante);
    context.SaveChanges();
    return RedirectToAction("Index");
}
```

Crie a visão `Create`, seguindo exatamente os mesmos passos já vistos anteriormente. Não comentarei nada sobre ela, pois é exatamente igual à visão `Create` para `Categorias`. Teste sua aplicação, chamando a view `Create`, ou a `Index`, clicando no link para adicionar um novo fabricante. Insira alguns fabricantes para testar. :-)

Alterando dados já existentes com o Entity Framework

O método `action` que gerará a visão para alteração dos dados, fazendo uso do EF, é um pouco diferente do que implementamos no capítulo 1. A *primeira aplicação ASP.NET MVC 5*, para Categorias, pois precisamos realizar alguns testes. Veja o código na sequência.

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

Em seguida, com um valor válido para o `id`, é realizada a busca de um fabricante com este valor em sua chave primária (o identificador do objeto). Após a busca, é preciso verificar se um objeto foi encontrado e, caso não, um outro tipo de erro é retornado. Caso exista um fabricante para o `id` recebido pela `action`, o objeto que o representa é encaminhado para a visão.

```
// GET: Fabricantes/Edit/5
public ActionResult Edit(long? id)
{
    if (id == null)
    {
        return new
            HttpStatusCodeResult(
                HttpStatusCode.BadRequest);
    }
    Fabricante fabricante = context.Fabricantes.Find(id);
    if (fabricante == null)
    {
        return HttpNotFound();
    }
    return View(fabricante);
}
```

HttpStatusCodeResult(HttpStatusCode.BadRequest)

A classe `HttpStatusCodeResult` fornece uma maneira de retornar um `ActionResult` com uma descrição e código de status de uma resposta HTTP. A enumeração `HttpStatusCode` contém os valores de status definidos para o HTTP. Vale a pena acessar a MSDN e ver na documentação estes valores possíveis.

O `BadRequest` é um dos valores possíveis, e é equivalente ao status 400 HTTP. Este valor indica que a solicitação não pôde ser atendida pelo servidor. Normalmente, ele é enviado quando nenhum outro erro é aplicável, ou se o erro exato é conhecido ou não tem seu próprio código de erro.

HttpNotFound()

O método `HttpNotFound()`, que pertence ao controlador, retorna uma instância da classe `HttpNotFoundResult`. Este erro é equivalente ao status HTTP 404 e é interessante que a aplicação possua uma página para indicar o erro ao usuário. Logo veremos isso.

Agora, é preciso criar a visão para a alteração e isto é com você. :-) Ela é idêntica ao que fizemos para categoria. Para finalizarmos, precisamos implementar a action que receberá os dados da visão. Veja no código a seguir esta implementação.

O código começa verificando se o modelo é válido, ou seja, se não há nenhuma validação de erro, como por exemplo, um valor requerido não preenchido. Com o modelo validado, é preciso dizer que o objeto recebido sofreu uma alteração desde sua recuperação
(`context.Entry(fabricante).State` =

`EntityState.Modified`). Isso é preciso para que uma atualização ocorra na base de dados, e esta é realizada pelo método já conhecido, o `SaveChanges()` .

Caso tudo ocorra bem, a aplicação é redirecionada para a action `Index` . Caso contrário, o objeto recebido é retornado para a visão que requisitou a ação atual, a `Edit` .

```
// POST: Fabricantes/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(Fabricante fabricante)
{
    if (ModelState.IsValid)
    {
        context.Entry(fabricante).State =
            EntityState.Modified;
        context.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(fabricante);
}
```

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, como exemplo, a minha foi <http://localhost:50827/Fabricantes/Edit/2>.

Implementando a visualização dos detalhes de um objeto com o Entity Framework

A visualização dos dados de um determinado fabricante está sendo possível quando optamos por editar os de um determinado fabricante, mas pode ser que queiramos apenas ver, e não editar.

Para isso, existe a visão `Details`. Esta implementação segue os mesmos princípios da que fizemos para categorias, no capítulo 1. A *primeira aplicação ASP.NET MVC 5*. Vamos implementá-la agora.

Lembre-se de que, para a action `Details`, precisamos apenas da action que atende a requisição `GET` do HTTP, pois o usuário não modificará os dados na visão. Na sequência, veja o código que representa esta action. Observe que a implementação é exatamente igual a 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 esta action e teste sua aplicação. As orientações para o teste são as mesmas fornecidas para o teste da action `Edit`.

```
// GET: Testes/Details/5
public ActionResult Details(long? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(
            HttpStatusCode.BadRequest);
    }
    Fabricante fabricante = context.Fabricantes.
        Find(id);
    if (fabricante == null)
    {
        return HttpNotFound();
    }
    return View(fabricante);
}
```

Implementando a remoção de um fabricante da base de dados com o Entity Framework

Para a conclusão do CRUD para `Fabricantes`, nos resta apenas implementar a funcionalidade de remoção de um

registro/objeto. Para isso, recordando o que fizemos no capítulo anterior para `Categorias`, precisamos implementar uma action que gerará a visão (`GET`) e uma que receberá a confirmação de exclusão do objeto em exibição ao usuário.

Começamos então com a action que gerará a visão e tem seu código exibido na sequência. Observe, uma vez mais, a redundância de código comum. São as mesmas instruções que as utilizadas para o `Edit` e `Details`. Calma, logo trabalharemos uma simplificação para isso. :-)

```
// GET: Fabricantes/Delete/5
public ActionResult Delete(long? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(
            HttpStatusCode.BadRequest);
    }
    Fabricante fabricante = context.Fabricantes.
        Find(id);
    if (fabricante == null)
    {
        return HttpNotFound();
    }
    return View(fabricante);
}
```

Agora você precisa criar a visão, da mesma maneira que já estamos fazendo desde o primeiro capítulo. Com ela criada, precisamos criar a segunda action para a remoção de fabricantes. Agora, uma que capture a requisição HTTP `POST`. Veja na sequência o código.

A única instrução que ainda não foi trabalhada neste código é a responsável por remover o objeto recuperado da coleção de objetos no contexto: a `context.Fabricantes.Remove(fabricante)`.

Note que, após remover o objeto, o método `SaveChanges()` é invocado.

```
// POST: Fabricantes/Delete/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(long id)
{
    Fabricante fabricante = context.Fabricantes.
        Find(id);
    context.Fabricantes.Remove(fabricante);
    context.SaveChanges();
    return RedirectToAction("Index");
}
```

2.3 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, fazendo uso do Entity Framework. 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 `Categorias`, para acessar a base de dados, tal qual fizemos neste segundo capítulo? Com este finalizado, já teremos dois domínios sendo atendidos (`Categoria` e `Fabricante`) e fazendo uso do EF para serem persistidos em uma base de dados.

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

legal. :-)

CAPÍTULO 3

LAYOUTS, BOOTSTRAP E JQUERY DATATABLE

Aplicações web são aplicações, até certo ponto, complexas. Elas normalmente 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 dois capítulos anteriores, está no servidor, e esta resolução está implementada por classes em C# (em nosso caso). Mas não é esta camada que seu usuário visualizará e, em alguns casos, ele nem sabe da sua existência. Então, o que o usuário vê e se interessa inicialmente?

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 se trabalha 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. :-) Bem, como não sou designer e muito menos artista, recorro ao Bootstrap e templates que o utilizam e que podem ser baixados na web, alguns inclusive de maneira gratuita. E é sobre isso que este capítulo trabalha.

Nossa aplicação até o início deste capítulo possui dois controladores para os dois domínios que já trabalhamos, `Categoria` e `Fabricante`. 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 `Fabricante`, que criamos no capítulo anterior.

3.1 O BOOTSTRAP

Ao acessar o site do Bootstrap (<http://getbootstrap.com/>), nos deparamos com uma definição para ele: "*O Bootstrap é o framework HTML, CSS e JS mais popular para o desenvolvimento responsivo na web*". E ele é gratuito :-). Mas, o que é o "Design Responsivo"?

Vamos falar do 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 destes dispositivos, existe uma série de especificações técnicas para a tela que apresentará a interface de sua aplicação para o usuário. E, para 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 uma forma organizada em seu layout? Isso, sem que você tenha de se preocupar com estas especificações. Uma resposta para esta inquietação está no uso do Bootstrap. Ele trata tudo isso para nós.

Em relação ao Bootstrap, é possível obtê-lo baixando um arquivo ZIP pelo endereço <http://getbootstrap.com>. Nele existem 3 pastas: `css` , `fonts` e `js` . A versão utilizada neste livro é a 3.3.6. Outra maneira, e a que inicialmente usaremos, é o uso do NuGet para baixar e instalar o Bootstrap em nosso projeto.

Na Solution Explorer, clique com o botão direito do mouse sobre o nome de nosso projeto, e depois em `Manage NuGet Packages` . Na guia que se abre, pesquise por `Bootstrap` e, quando ele for exibido, selecione-o e clique no botão `Install` , conforme a figura a seguir.

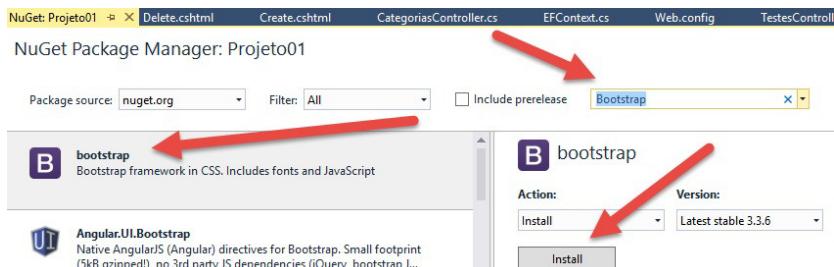


Figura 3.1: Janela do NuGet para instalação do Bootstrap

Durante a instalação, janelas que apresentam a licença de uso surgirão para sua confirmação. Após a instalação ser concluída, duas novas pastas estarão disponíveis em seu projeto: `Content` e `fonts` . A pasta `Scripts` , que já existia e não havíamos comentado sobre ela ainda, receberá os arquivos JavaScript do

Bootstrap, que serão somados aos já existentes do jQuery (figura a seguir). Esta pasta e arquivos foram criados quando criamos a primeira visão, especificando que deveriam ser referenciadas, na visão, as bibliotecas de script.

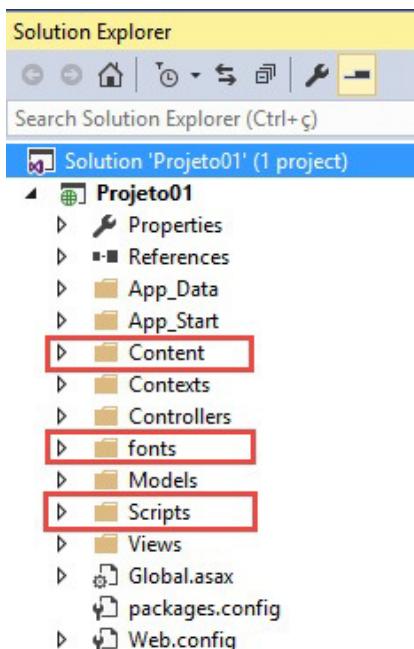


Figura 3.2: Solution Explorer apresentando as pastas criadas com a instalação do Bootstrap

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 construir interfaces web intuitivas de uma maneira mais simples e rápida.

A figura seguinte traz as classes e indicações sobre o sistema de grades. Logo veremos na prática. Os asteriscos devem ser substituídos por valores entre 1 e 12 (cada linha, para o Bootstrap, possui 12 colunas).

Nome da classe	Tipo de dispositivo	Resolução	Tamanho do container	Tamanho da coluna
col-xs-*	Celulares	< 768px	Automático	Automático
col-sm-*	Tablets	≥ 768px	750px	~62px
col-md-*	Desktops	≥ 992px	970px	~81px
col-lg-*	Desktops de alta resolução	≥ 1200px	1170px	~97px

Figura 3.3: Classes de estilo a serem utilizadas pelo sistema de grades do Bootstrap

3.2 LAYOUTS

Quando se desenvolve 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 estas 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, já fazendo uso do Bootstrap. Legal, não é? :-) Vamos lá.

O padrão ASP.NET MVC diz que os arquivos de layout (isso mesmo, pode existir mais de um), devem ficar dentro de uma pasta chamada `Shared`, dentro da pasta `Views`. Sendo assim, crie esta pasta, pois ela ainda não existe. Com ela criada, clique com o botão direito do mouse sobre ela e, em seguida, na opção `Add->MVC 5 Layout Page (Razor)`.

Na janela que se abre, informe o nome `_Layout.cshtml`. Isso mesmo, começando com um sublinhado, para que a visão não seja requisitada. No padrão ASP.NET MVC, uma visão com o nome iniciado com sublinhado não poderá ser requisitada diretamente pelo navegador. O código da sequência representa a implementação para o arquivo `_Layout.cshtml`.

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width,
        initial-scale=1.0" charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/content/bootstrap.css")"
        rel="stylesheet"/>
</head>
<body>
    <div class="container">
        <div class="row">
            <div class="page-header bg-danger">
                <h1>Desenvolvimento em ASP.NET MVC
                    <small class="text-danger">
                        Utilizando o Bootstrap</small></h1>
            </div>
        </div>
        <div class="row">
            <div class="col-xs-3 col-md-3 bg-success">
                <ul class="nav nav-pills nav-stacked">
                    <li role="presentation" class=
                        "active"><a href="#">Home</a></li>
```

```

<li role="presentation">
    @Html.ActionLink("Categorias",
    "Index", "Categorias")</li>
<li role="presentation">
    @Html.ActionLink("Fabricantes",
    "Index", "Fabricantes")</li>
</ul>
</div>
<div class="col-xs-9 col-md-9">
    <br />
    @RenderBody()
</div>
</div>

<br/>
<div class="row">
    <div class="col-xs-12 col-md-12 bg-danger">
        <small>Layout com Bootstrap</small>
    </div>
</div>
</div>
<script src="@Url.Content(
    "~/scripts/jquery-1.10.2.js")"></script>
<script src="@Url.Content(
    "~/scripts/bootstrap.js")"></script>
</body>
</html>

```

No código anterior, na primeira instrução dentro da tag `<head>` é definida a `Viewport`. 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, e o valor `initial-scale=1.0` define o valor inicial para zoom.

A instrução `@ViewBag.Title` representa o título que aparece na barra de títulos do navegador e que pode ser definido em cada visão que utilizar o layout. Terminando a tag `<head>`, temos a

inclusão do arquivo de CSS do Bootstrap. Sempre que o início for ~ em uma URL do ASP.NET MVC, significa que o endereço informado tem seu início na raiz da aplicação.

Dentro da tag `<body>`, já é possível identificar o uso do Bootstrap pelas classes das tags `<div>`. O Bootstrap precisa de um elemento container para encapsular os conteúdos do site e manter o sistema de grades nele. A classe `.container` define um container responsivo de tamanho fixo.

A classe `.row` define uma linha, que poderá conter no máximo 12 colunas; `page-header` define um estilo específico para o cabeçalho da página; `bg-danger` define uma cor de fundo (background); `text-danger` define uma cor para a fonte de texto; e `col-xs-3` define uma coluna com 3 posições, o mesmo valendo para `col-md-3`. Lembre-se de que o `xs` é para celulares e `md` para desktops.

Definindo estes dois elementos em conjunto, seguimos a premissa do `Mobile-first` e atendemos também requisições de um ambiente desktop. Para a definição do menu, fazemos uso das classes `nav`, `nav-pills` e `nav-stacked` para formatar uma lista não ordenada (tag ``). Observe que na tag ``, responsável pelo link `Home`, há a definição da classe `active`, que dará um visual diferenciado para esta opção.

Temos a instrução Razor `@RenderBody()` que renderizará o conteúdo da visão que utiliza este layout. Essa instrução, usada em páginas de layout, renderiza a porção da página de conteúdo que não está dentro de uma seção nomeada. Para um aprofundamento maior neste conteúdo, acesse <http://www.codeproject.com/Articles/383145/RenderBody->

[RenderPage-and-RenderSection-methods-in.](#)

Ao final, temos a inclusão de arquivos JavaScript pela instrução `@Url.Content()`, que converte um caminho relativo (ou virtual) para um caminho absoluto na aplicação. É importante ressaltar que tanto os arquivos de CSS quanto JavaScript importados no layout serão importados para todas as visões que façam uso do layout. Se existir algum recurso (CSS ou JS) específico para uma visão, ele deve ser importado na visão.

Já temos o layout definido, agora, vamos utilizá-lo. Abra a visão `Index` de `Fabricantes` e, antes da tag `<!DOCTYPE html>`, insira o código a seguir. Este renderizará a visão dentro do layout especificado. A visão é inserida no layout por meio da instrução `@RenderBody()`.

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

Agora, para testarmos esta alteração, é preciso que você execute sua aplicação e requisite a visão alterada. A figura adiante apresenta a página renderizada. Vamos fazer alguns testes?

1. Mude o tamanho da janela do navegador e veja o comportamento da página;
2. Retire o `col-xs` e mude o tamanho da janela que exibe o navegador. Legal o comportamento, não é? :-)

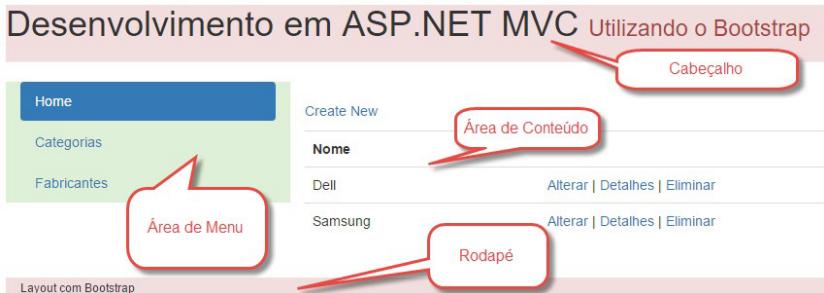


Figura 3.4: Visão Index de Fabricantes utilizando o layout criado

Com isso, terminamos uma breve introdução ao Bootstrap e a layouts, mas vamos continuar com isso já já. É importante um aprofundamento nos recursos oferecidos pelo Bootstrap, o que foge do contexto deste livro. Ao final deste capítulo, você poderá encontrar recomendações de links que poderá acessar e fazer uma leitura atenciosa sobre todos os recursos oferecidos pelo Bootstrap.

Agora, se você quer fazer uso de layouts profissionais em seus projetos, você pode adquirir alguns templates prontos. Alguns são inclusive gratuitos e de fácil localização na web. Veja um link com exemplos para a área de administração de sites: <https://wrapbootstrap.com/themes/admin>.

3.3 ADAPTANDO AS VISÕES PARA O USO DO BOOTSTRAP

Nesta seção, terei o foco nas visões renderizadas pelo controlador `Fabricantes`. Desta maneira, para começar, a primeira visão que adaptaremos é a `Index`. Nela, é apresentada uma relação de fabricantes registrados. Para essa implementação, além de usarmos o Bootstrap, faremos uso de um novo

componente, o jQuery DataTables (<https://www.datatables.net/>).

O DataTables é um plugin para o jQuery, e uma ferramenta que traz enormes ganhos para listagens em forma de tabelas. Para disponibilizar este plugin, faremos uso do NuGet. Desta maneira, clique com o botão direito do mouse sobre o nome do projeto e acesse Manage NuGet Packages... . Na janela que se abre, pesquise por DataTables e o instale em seu projeto (figura a seguir).

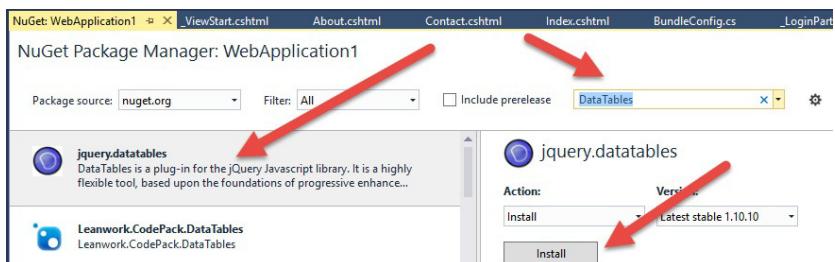


Figura 3.5: Instalando o DataTables no projeto

Após a instalação do DataTables, uma pasta chamada DataTables é criada dentro das pastas Content e Scripts , contendo os recursos que serão necessários para nossa implementação. Você verá que é bem simples. Na sequência a implementação da visão Index , que implementa o DataTables .

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

{@
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Listagem de FABRICANTES";
}

<div class="panel panel-primary">
    <div class="panel-heading">
        Relação de FABRICANTES registrados
    </div>
```

```

</div>
<div class="panel-body">
    <table class="table table-striped table-hover">
        <thead>
            <tr>
                <th>
                    @Html.DisplayNameFor(
                        model => model.FabricanteId)
                </th>
                <th>
                    @Html.DisplayNameFor(
                        model => model.Nome)
                </th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model)
            {
                <tr>
                    <td>
                        @Html.DisplayFor(modelItem
                            => item.FabricanteId)
                    </td>
                    <td>
                        @Html.DisplayFor(modelItem
                            => item.Nome)
                    </td>
                    <td>
                        @Html.ActionLink("Alterar",
                            "Edit", new { id =
                                item.FabricanteId }) | 
                        @Html.ActionLink("Detalhes",
                            "Details", new { id =
                                item.FabricanteId }) | 
                        @Html.ActionLink("Eliminar",
                            "Delete", new { id =
                                item.FabricanteId })
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
<div class="panel-footer panel-info">

```

```

        @Html.ActionLink("Registrar um novo FABRICANTE",
            "Create", null, new { @class =
            "btn btn-success" })
    </div>
</div>

@section styles{
    <link href="@Url.Content(
        "~/content/DataTables/css/dataTables.bootstrap
.css")" rel="stylesheet"/>
}

@section ScriptPage {
<script src="@Url.Content("~/scripts/DataTables/jquery.
dataTables.js")"></script>
<script src="@Url.Content("~/scripts/DataTables/
dataTables.bootstrap.js")"></script>
<script type="text/javascript">
    $(document).ready(function () {
        $('.table').dataTable();
    });
</script>
}

```

Na visão `Index`, implementada pela listagem anterior, a instrução `ViewBag.Title` define o título para a janela do navegador quando ela for renderizada. Note que não temos mais as tags `<html>`, `<head>` e `<body>`, pois elas não são mais necessárias na visão. Entretanto, lembre-se de que fazemos uso de um layout, e nele essas tags estão definidas. Vamos tratar as particularidades desta visão.

Faremos uso de painéis do Bootstrap. Um painel pode ter até três componentes: cabeçalho (`panel-heading`), corpo (`panel-body`) e rodapé (`panel-footer`). Estes (cada um em uma `<div>`) estão encapsulados por outra `<div>`, que define o `panel`. Verifique que, na definição do painel, definimos também a cor para ele (`panel-primary`), que refletirá no cabeçalho e, para

o rodapé, temos a cor `panel-info`. Tudo por meio de classes CSS. Verifique, no rodapé do painel, que o link para registrar um novo fabricante será gerado com um estilo CSS, que dá a ele a aparência de um botão (`new { @class = "btn btn-success" }`).

Para que o jQuery DataTables possa funcionar em uma tabela HTML (`<table>`), é preciso que essa tabela 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 da sequência.

Ao final do 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 estes arquivos serão inseridos na página que será renderizada, que é a união do layout com a visão? Para a seção `Style`, no layout antes de fechar a tag `<head>`, insira a instrução `@RenderSection("styles", false)` e, para o JS, antes de fechar a tag `<body>`, insira a instrução `@RenderSection("ScriptPage", false)`.

E para fazer 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 no final da listagem, a chamada a `$('.table').dataTable();`. O símbolo `$` denota uma instrução jQuery.

Com o argumento `('table')`, selecionamos os elementos HTML que possuem a classe `table` de CSS. Com o elemento recuperado, invoca-se 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.

Ufa! Chegamos ao fim da primeira implementação fazendo uso de Boostrap, layouts e DataTable. A figura a seguir apresenta a página gerada para a visão `Index` de `Fabricantes`. Veja nela, em destaque, os elementos oferecidos pelo DataTable em conjunto com o Painel 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 pesquisa no conteúdo renderizado na tabela. Ao lado do título de cada coluna, existe um botão para classificação dos dados com base na coluna desejada. Ao final, do lado esquerdo, a informação de quantos elementos estão sendo exibidos e o seu total. Por fim, à direita, uma opção de navegação entre páginas. Legal? Tudo isso pronto. :-)

Desenvolvimento em ASP.NET MVC Utilizando o Bootstrap

The screenshot shows a web application interface. On the left, there is a sidebar with a blue header 'Home' and two green items: 'Categorias' and 'Fabricantes'. The main content area has a blue header 'Relação de FABRICANTES registrados'. Below it is a table with two rows of data:

FabricanteId	Nome	Ações
1	Dell	Alterar Detalhes Eliminar
2	Samsung	Alterar Detalhes Eliminar

Red arrows highlight several UI elements: one arrow points to the 'Show 10 entries' dropdown in the top-left; another points to the 'Search:' input field; a third points to the 'FabricanteId' column header; a fourth points to the 'Nome' column header; a fifth points to the 'Showing 1 to 2 of 2 entries' message; and a sixth points to the 'Next' button at the bottom-right.

Registrar um novo FABRICANTE

Layout com Bootstrap

Figura 3.6: A visão Index de Fabricantes 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 do Fabricante, e neste capítulo nós o inserimos como primeira coluna, sendo o DataTables que 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 para o 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` .

```
$('.table').dataTable({
    "order": [[1, "asc"]]
});
```

Adaptando a visão Create

Espero que você tenha gostado da formatação dada à visão Index . Você pode melhorar e mudar as cores e formatação com as diversas classes de estilo que o Bootstrap oferece e que são fáceis de se conhecer. Fica como atividade para você visitar os links informados ao final do capítulo, e testar novas classes de estilo que encontrará.

Agora, vamos adaptar nosso formulário de registro de um novo fabricante para fazer uso do Bootstrap. O código a seguir traz a nova implementação para a visão Create .

```
@model Projeto01.Models.Fabricante

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Registrando um NOVO FABRICANTE";
}

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Registro de um NOVO FABRICANTE
        </div>
        <div class="panel-body">
            <div class="form-horizontal">
                @Html.ValidationSummary(true, "", new { @class = "text-danger" })
                <div class="form-group">
                    @Html.LabelFor(model => model.Nome,
                        htmlAttributes: new { @class =
                            "control-label col-md-2" })
                    <div class="col-md-10">
                        @Html.EditorFor(model =>
                            model.Nome, new {
                                htmlAttributes = new {
                                    @class = "form-control" } })
                        @Html.ValidationMessageFor(
                            model => model.Nome, "",
```

```

        new { @class = "text-danger" })
    </div>
</div>
</div>
</div>
<div class="panel-footer panel-info">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit"
            value="Adicionar Fabricante"
            class="btn btn-danger" />
        @Html.ActionLink(
            "Retornar para a listagem de FABRICANTES",
            "Index", null, new { @class =
            "btn btn-warning" })
    </div>
    <br />
    <br />
</div>
</div>
}
}

@section ScriptPage {
    <script src="~/Scripts/jquery.validate.min.js">
    </script>
    <script src="~/Scripts/jquery.validate.unobtrusive.min.js">
    </script>
}

```

Se você notar o código da visão `Create`, algumas tag `<div>` já continham classes do Bootstrap, como:

- `form-horizontal` , que define que as tags que ela contém devem ser inseridas de maneira horizontal;
- `form-group` , que registra em seu interior um grupo de elementos do formulário.

Alguns controles também já possuíam algumas classes, como: `control-label` e `form-control` , que formatam os controles que serão renderizados com os CSSs do Bootstrap. No rodapé do painel (que já conhecemos), temos uma classe nova: `col-md-`

`offset-2`, que deixa em branco duas colunas da linha Bootstrap (lembre-se de que cada linha pode ter no máximo 12 colunas). Com ela, os elementos dentro da `<div>` que a utiliza são exibidos a partir da terceira coluna.

Ao final, temos os scripts jQuery que deverão ser inseridos na renderização da visão. Logo usaremos estes recursos. 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 a apresentada na figura a seguir, na sequência, que apresenta um recorte da visão `Create`.

The screenshot shows a web form titled "Registro de um NOVO FABRICANTE". It has a single input field labeled "Nome" with a placeholder "Digite o nome...". Below the input field are two buttons: a red one labeled "Adicionar Fabricante" and an orange one labeled "Retornar para a listagem de FABRICANTES".

Figura 3.7: A visão Create de Fabricantes

Adaptando a visão Edit

Na alteração para que a visão `Edit` faça uso do layout e do 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 Projeto01.Models.Fabricante
```

```

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title =
        "Alterando os dados de um FABRICANTE";
}

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Alteração dos dados de um FABRICANTE
        </div>
        <div class="panel-body">
            <div class="form-horizontal">
                @Html.ValidationSummary(true, "", new
                    { @class = "text-danger" })
                @Html.HiddenFor(model => model.FabricanteId)

                <div class="form-group">
                    @Html.LabelFor(model => model.Nome,
                        htmlAttributes: new { @class =
                            "control-label col-md-2" })
                    <div class="col-md-10">
                        @Html.EditorFor(model => model.Nome,
                            new { htmlAttributes = new {
                                @class = "form-control" } })
                        @Html.ValidationMessageFor(model =>
                            model.Nome, "", new { @class =
                                "text-danger" })
                    </div>
                </div>
            </div>
            <div class="panel-footer panel-info">
                <div class="col-md-offset-2 col-md-10">
                    <input type="submit" value=
                        "Gravar alterações" class=
                        "btn btn-info" />
                    @Html.ActionLink(
                        "Retornar para a listagem de FABRICANTES"
                    ,
                        "Index", null, new { @class =
                            "btn btn-warning" })
                </div>
            </div>
        </div>
    </div>
}

```

```

        <br />
        <br />
    </div>
</div>
</div>
}

@section ScriptPage {
<script src("~/Scripts/jquery.validate.min.js">
</script>
<script src "~/Scripts/jquery.validate.unobtrusive.min.js">
</script>
}

```

Adaptando a visão Details

Nesta visão, trago algo novo: uma maneira de visualizar os dados diferente da que vimos no capítulo anterior. Trago dois novos recursos do Bootstrap e a visualização do dado em um controle de entrada, desabilitado. Veja o código na sequência.

Veja, na `<div>` após o `@Html.LabelFor()`, que faço uso da classe `input-group`. Esta permite um efeito visual diferente, como se tivéssemos um controle dividido em dois. Na primeira parte, temos um ``, com a classe `input-group-addon` e, dentro deste, temos um elemento `<i>`, com as classes `glyphicon` e `glyphicon-user`. Ao usarmos estas classes, teremos renderizada uma figura dentro do ``.

O link <http://getbootstrap.com/components/#glyphicons-glyphs> traz o conjunto de figuras disponíveis para este uso. Na instrução `@Html.EditorFor()`, veja os elementos para o `htmlAttributes`. O `disabled` proíbe a interação do usuário com o controle.

```
@model Projeto01.Models.Fabricante
```

```

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title =
        "Visualizando detalhes de um FABRICANTE";
}

<div class="panel panel-primary">
    <div class="panel-heading">
        Visualizando detalhes de um FABRICANTE
    </div>
    <div class="panel-body">
        <div class="form-group">
            @Html.LabelFor(model => model.Nome)
            <br />
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="glyphicon glyphicon-user"></i>
                </span>
                @Html.EditorFor(model => model.Nome,
                    new { htmlAttributes = new {
                        @class = "form-control",
                        disabled = "disabled" } })
            </div>
        </div>
    </div>
    <div class="panel-footer panel-info">
        @Html.ActionLink("Alterar", "Edit", new { id =
            Model.FabricanteId }, new { @class =
            "btn btn-info" })
        @Html.ActionLink("Retornar para a listagem",
            "Index", null, new { @class =
            "btn btn-info" })
    </div>
</div>

```

Teste sua aplicação e requisite os detalhes de algum fabricante.
A figura a seguir apresenta o recorte da página renderizada.



Figura 3.8: A visão Details de Fabricantes

Adaptando a visão Delete

Na implementação desta última visão do CRUD, fazendo uso do Bootstrap, trago um novo recurso: a exibição de mensagens. Para essa implementação, também trago a passagem de valores entre visão e controlador, fazendo uso de `TempData`. Para começarmos, na sequência, veja a visão gerada pela action `Delete` (`GET`).

```
@model Projeto01.Models.Fabricante

 @{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title =
        "Visualizando detalhes de um FABRICANTE";
}

<div class="panel panel-primary">
    <div class="panel-heading">
        Dados do fabricante a ser removido
    </div>
    <div class="panel-body">
        <div class="form-group">
            @Html.LabelFor(model => model.Nome)<br />
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="glyphicon glyphicon-user">
                        </i>
                </span>
                @Html.EditorFor(model => model.Nome, new
                { htmlAttributes = new { @class =

```

```

        "form-control", disabled = "disabled"
    } })

```

```

    </div>
</div>
<div class="panel-footer panel-info">
    @using (Html.BeginForm())
    {
        @Html.AntiForgeryToken()
        <input type="submit" value=
            "Remover FABRICANTE" class=
            "btn btn-danger" />
        @Html.ActionLink(
            "Retornar para a listagem",
            "Index", null, new { @class =
            "btn btn-info" })
    }
</div>
</div>

```

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

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(long id)
{
    Fabricante fabricante = context.Fabricantes
        .Find(id);
    context.Fabricantes.Remove(fabricante);
    context.SaveChanges();
    TempData["Message"] = "Fabricante " +
        fabricante.Nome.ToUpper() + " foi removido";
    return RedirectToAction("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 estará disponível até que ele seja recuperado. No momento de recuperação deste valor, ele deixa de existir e retornará nulo em um novo processo de recuperação.

Em nosso exemplo, criamos a chave `Message`, e nela armazenamos um valor. Este será recuperado na visão que renderiza os dados do Fabricante a ser removido, como pode ser visto na listagem a seguir (visão `Index`). Uma boa leitura sobre este recurso pode ser realizada nos links <http://www.codeproject.com/Articles/476967/What-is-ViewData-ViewBag-and-TempData-MVC-Option> e <http://eduardopires.net.br/2013/06/asp-net-mvc-viewdata-viewbag-tempdata/>.

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

 @{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Listagem de FABRICANTES";
}

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

Demais instruções foram ocultadas
*@

Observe, no código anterior, 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. Neste 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, como pode ser verificado na figura:



Figura 3.9: A visão Index de Fabricantes exibindo uma Mensagem Bootstrap

3.4 CONFIGURANDO O MENU DE ACESSO PARA DESTACAR A PÁGINA ATUAL

Concluímos todas as visões para o CRUD de `Fabricantes`. Entretanto, o menu sempre mostra em destaque o link `Home`, pois é ele que está definido com a classe `active`. Isso pode ser comprovado na listagem a seguir, que apresenta um trecho de nosso `_Layout.cshtml`.

```
<div class="col-xs-3 col-md-3 bg-success">
    <ul class="nav nav-pills nav-stacked">
```

```

<li role="presentation" class="active">
    <a href="#">Home</a></li>
<li role="presentation">@Html.ActionLink(
    "Categorias", "Index", "Categorias")</li>
<li role="presentation">@Html.ActionLink(
    "Fabricantes", "Index", "Fabricantes")</li>
</ul>
</div>

```

Temos que realizar duas implementações para que o menu destaque corretamente qual visão está sendo renderizada no navegador. A primeira é atribuir um `id` para cada elemento ``, como podemos ver na listagem a seguir.

```

<div class="col-xs-3 col-md-3 bg-success">
    <ul class="nav nav-pills nav-stacked">
        <li id="liHome" role="presentation" class=
            "active"><a href="#">Home</a></li>
        <li id="liCategorias" role="presentation">@Html.
            ActionLink( "Categorias", "Index",
            "Categorias")</li>
        <li id="liFabricantes" role="presentation">@Html.
            ActionLink( "Fabricantes", "Index",
            "Fabricantes")</li>
    </ul>
</div>

```

A segunda implementação fará uso do jQuery. Ela precisará retirar a classe `active` do elemento que a possui e inseri-la no elemento correto. Veja o código a seguir, que deve ser inserido na seção de scripts, no final de cada visão.

Na visão `Index`, apenas as duas instruções referente à classe `active` devem ser inseridas, pois lá já existe um código de script para o DataTables. A primeira instrução, que requisita o método `removeClass()`, removerá a classe `active` de todo elemento ``. Já a segunda instrução, por estar usando um elemento nomeado (veja o `#` antes do nome `liFabricantes`), adicionará

a classe `active` ao elemento `liFabricantes`.

```
<script type="text/javascript">
$(document).ready(function () {
    $('li').removeClass("active");
    $('#liFabricantes').addClass("active");
});
</script>
```

Execute sua aplicação, requisitando qualquer visão, e verifique essa alteração em funcionamento. Lembre-se de que, para ela funcionar, o item que representa a visão renderizada precisa estar marcado como ativo. Aproveite para adaptar o CRUD de Categorias para fazer uso dos recursos apresentados neste capítulo.

RECOMENDAÇÕES DE LEITURA

Algumas leituras adicionais são sempre bem-vindas. Desta maneira, seguem algumas recomendações.

Sobre design responsivo, para um aprofundamento no assunto, indico o link <https://responsivedesign.is/> e o livro <http://www.casadocodigo.com.br/products/livro-web-design-responsivo>. Sobre Bootstrap para ASP.NET MVC, recomendo o livro <https://www.packtpub.com/web-development/bootstrap-aspnet-mvc>.

Como é importante o conhecimento, mesmo que básico de HTML, CSS e JavaScript, recomendo para HTML e CSS o livro <http://www.casadocodigo.com.br/products/livro-html-css>, e para JavaScript, <http://www.casadocodigo.com.br/products/livro-programacao>. Como faremos uso de jQuery, seria interessante ler também: <http://www.casadocodigo.com.br/products/livro-javascript-jquery>.

LINKS SOBRE O BOOTSTRAP

Uma relação completa dos componentes disponibilizados pelo Bootstrap pode ser obtida em <http://getbootstrap.com/components/>. Informações sobre as classes de CSS, incluindo o sistema de grade (*grid system*) podem ser verificadas em <http://getbootstrap.com/css/>. Já exemplos do sistema de grades podem ser vistos em <https://getbootstrap.com/examples/grid/>.

3.5 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 que você defina uma estrutura básica para um conjunto de visões.

O uso do jQuery foi mais uma novidade. E sempre que for necessário seu uso, uma explicação será fornecida. Na parte do ASP.NET MVC em si, apenas o uso do TempData para o envio de valores para as visões foi apresentado. Em relação ao Entity Framework, nada de novo foi apresentado, mas no próximo capítulo isso será compensado. Veremos associações.

CAPÍTULO 4

ASSOCIAÇÕES NO ENTITY FRAMEWORK

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 destes vínculos (associações), um objeto invoque (ou requisite) serviços de outro, ou que acesse ou atribua valores às propriedades deste 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. Estas associações, no modelo relacional, são conhecidas como relacionamentos.

Tanto para o modelo relacional como o de objetos, existem algumas regras quando se implementa 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 e, a segunda, se de um objeto/registo é possível recuperar o objeto

associado.

Até o momento, tivemos apenas duas classes, que geraram duas tabelas e nenhuma delas estão associadas. Implementar a associação de classes fazendo uso do Entity Framework é um dos objetivos deste capítulo.

4.1 ASSOCIANDO AS CLASSES JÁ CRIADAS A UMA NOVA CLASSE

Em nosso contexto de problema, implementamos as funcionalidades de categorias e fabricantes para que pudéssemos ter estes dados registrados em nossos produtos. Assim, como temos as classes `Categoria` e `Fabricante`, precisamos agora utilizá-las na classe `Produto`, que tem seu código exposto na listagem a seguir. Lembre-se de que esta classe precisa ser criada na pasta `Models`.

Observe, no código a seguir, que as propriedades `ProdutoId` (chave primária), `CategoriaId` e `FabricanteId` (chaves estrangeiras) são do tipo `long?`, ou seja, aceitam valores nulos. Verifique também as propriedades `Categoria` e `Fabricante`, que representam as associações com objetos de seus respectivos tipos.

Você pode se perguntar: "*Por que manter os Ids das associações se já possuímos a associação com seus respectivos atributos?*". Este problema refere-se ao carregamento (ou recuperação) do objeto da base de dados. O Entity Framework oferece três tipos de carregamento de objetos: *Eagerly Loading* (carregamento forçado), *Lazy Loading* (carregamento tardio) e

Explicit Loading (carregamento explícito).

No carregamento de um objeto `Produto`, as propriedades que representam chaves estrangeiras (`CategoriaId`, `FabricanteId`), como são mapeadas diretamente para colunas na tabela da base de dados, são carregadas imediatamente. Já as propriedades `Categoria` e `Fabricante` possuem dados relativos às suas propriedades que, dependendo do tipo de carregamento a ser usado, podem ou não ser carregados em conjunto.

Quando o carregamento ocorrer em conjunto com o objeto `produto` (*Eagerly Loading*), o SQL mapeado terá o join (junção) com as tabelas `Categoria` e `Fabricante` para que todos os dados sejam carregados por meio de um único `select`. Já se o carregamento for tardio (*Lazy Loading*), quando alguma propriedade dos objetos `Categoria` e/ou `Fabricante` do objeto `produto` for requisitada, um novo `select` (SQL) será executado. Perceba que isso se refere diretamente à performance das consultas realizadas na base de dados.

```
namespace Projeto01.Models
{
    public class Produto
    {
        public long? ProdutoId { get; set; }
        public string Nome { get; set; }

        public long? CategoriaId { get; set; }
        public long? FabricanteId { get; set; }

        public Categoria Categoria { get; set; }
        public Fabricante Fabricante { get; set; }
    }
}
```

Perfeito, já temos uma associação implementada e pronta para que o EF a aplique na base de dados. Não precisa, obrigatoriamente, de nenhuma implementação adicional, pois tudo será feito por convenção. A associação implementada possui navegabilidade da classe `Produto` para `Categoria` e para `Fabricante`. Ou seja, é possível saber quem é o `Fabricante` de um `Produto` e a que `Categoria` ele pertence.

Já em relação à sua multiplicidade, ela é um, pois cada `Produto` está associado a apenas um `Fabricante` e a apenas uma `Categoria`. Mas e como fica a associação de `Categoria` e `Fabricante` para `Produto`? Uma `Categoria` pode ter vários `Produto`s e um `Fabricante` produz diversos `Produto`s.

Com isso, já identificamos a multiplicidade de muitos (ou um para muitos), mas não temos ainda a navegabilidade. Ou seja, de uma `Categoria` ou `Fabricante`, não é possível identificar quais produtos pertencem a uma `Categoria` ou quais `Produtos` um `Fabricante` produz. A implementação da multiplicidade e navegabilidade está nas duas listagens a seguir.

```
namespace Projeto01.Models
{
    public class Categoria
    {
        public long CategoriaId { get; set; }
        public string Nome { get; set; }

        public virtual ICollection<Produto> Produtos { get; set;
    }
}
}

namespace Projeto01.Models
{
    public class Fabricante
    {
```

```
    public long FabricanteId { get; set; }
    public string Nome { get; set; }

    public virtual ICollection<Produto> Produtos { get; set;
}
}
}
```

Observe nas listagens anteriores que a propriedade `Produtos` são uma `ICollection<Produto>` nas duas classes, e são definidas como `virtual`. Definir elementos como `virtual` possibilita a sua sobreescrita, o que, para o EF, é necessário para que ele possa 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 ter este conhecimento.

VIRTUAL

`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 maiores informações, recomendo a leitura do artigo *Virtual vs Override vs New Keyword in C#*, em <http://www.codeproject.com/Articles/816448/Virtual-vs-Override-vs-New-Keyword-in-Csharp>.

ICOLLECTION

A escolha da interface `ICollection` para uma propriedade se deve ao fato de que, com esta interface, seja 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. Veja o artigo *List vs I Enumerable vs IQueryable vs ICollection vs IDictionary*, em <http://www.codeproject.com/Articles/832189/List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v>, pois é interessante este conhecimento.

Para terminar a etapa de implementação e configuração do EF, na classe `EFContext` implemente a propriedade `DbSet` para `Produtos`, como no código:

```
public DbSet<Produto> Produtos { get; set; }
```

4.2 CRIANDO A VISÃO INDEX PARA A CLASSE ASSOCIADA

Como ainda não temos o controlador para `Produtos`, vamos criá-lo. Faremos uso dos recursos do Visual Studio para criar este controlador. Clique com o botão direito do mouse sobre a pasta `Controllers`, clique em `Add` e depois em `Controller`. Na janela que se exibe, escolha a opção `MVC 5 Controller with read/write actions`, e clique no botão `Add`.

No nome do controller, informe `ProdutosController` . Após a criação, verifique o arquivo gerado e note que todas as actions para o CRUD foram criadas, mas ainda precisamos implementar o comportamento para elas. Neste momento, trabalharemos apenas a `Index` , que deve estar de acordo ao código apresentado na sequência. Veja que, antes dele, está declarado o contexto do EF.

```
public class ProdutosController : Controller
{
    private EFContext context = new EFContext();

    // GET: Produtos
    public ActionResult Index()
    {
        return View(context.Produtos.OrderBy(c => c.Nome));
    }
}
```

Com o controlador implementado, vamos criar a visão. Para isso, clique com o botão direito do mouse sobre o nome da action `Index` , e então em `Add View` . Minha sugestão é que você se baseie na visão criada para `Fabricantes` . Desta maneira, escolha o template `Empty` e copie o conteúdo da visão `Index` de `Fabricantes` para o novo arquivo criado, que deverá ser semelhante ao apresentado na sequência.

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

 @{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Listagem de PRODUTOS";
}

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

```

<div class="panel panel-primary">
    <div class="panel-heading">
        Relação de PRODUTOS registrados
    </div>
    <div class="panel-body">
        <table class="table table-striped table-hover">
            <thead>
                <tr>
                    <th>
                        @Html.DisplayNameFor(model =>
                            model.ProdutoId)
                    </th>
                    <th>
                        @Html.DisplayNameFor(model => model.Nome)
                    </th>
                    <th>Categoria</th>
                    <th>Fabricante</th>
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Model)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem =>
                                item.ProdutoId)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem =>
                                item.Nome)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem =>
                                item.Categoria.Nome)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem =>
                                item.Fabricante.Nome)
                        </td>
                        <td>
                            @Html.ActionLink("Alterar",
                                "Edit", new { id = item.
                                ProdutoId }) |
                            @Html.ActionLink("Detalhes",
                                "Details", new { id = item.

```

```

        ProdutoId }) |
    @Html.ActionLink("Eliminar",
        "Delete", new { id = item.
        ProdutoId })
    </td>
</tr>
}
</tbody>
</table>
</div>
<div class="panel-footer panel-info">
    @Html.ActionLink("Registrar um novo PRODUTO", "Create",
        null, new { @class = "btn btn-success" })
</div>
</div>

@section styles{
    <link href="@Url.Content("~/content/DataTables/css/
        dataTables.bootstrap.css")" rel="stylesheet">
}

@section ScriptPage {
    <script src="@Url.Content("~/scripts/DataTables/jquery.
        dataTables.js")"></script>
    <script src="@Url.Content("~/scripts/DataTables/dataTables.
        bootstrap.js")"></script>
    <script type="text/javascript">
        $(document).ready(function () {
            $('li').removeClass("active");
            $('#liProdutos').addClass("active");
            $('.table').dataTable({
                "order": [[1, "asc"]]
            });
        });
    </script>
}

```

Veja na listagem anterior, nos `@Html.DisplayFor()`, a chamada às propriedades de `Categoria` e `Fabricante`. Estas são as únicas alterações em relação à visão `Index` de `Fabricantes`. Insira no menu do layout a opção de acesso a `Produtos`, tal qual é mostrado no código a seguir. Feito isso, teste

sua aplicação.

```
<li id="liProdutos" role="presentation">@Html.ActionLink("Produtos", "Index", "Produtos")</li>
```

Se tudo der certo, você receberá uma mensagem de erro antes que a nova visão possa ser renderizada. Este erro pode ser verificado na figura a seguir. A mensagem apresentada diz que o modelo e a base de dados estão diferentes. E é verdade. Nós inserimos uma nova classe e, na classe de contexto, criamos a propriedade `DbSet` para ela, e ela não existe na base de dados e o EF não está configurado para criá-la. Vamos tratar isso na sequência.

The screenshot shows a Visual Studio error window titled 'InvalidOperationException was unhandled by user code'. The message says: 'An exception of type 'System.InvalidOperationException' occurred in EntityFramework.dll but was not handled in user code'. Below it, additional information states: 'Additional information: The model backing the 'EFContext' context has changed since the database was created. Consider using Code First Migrations to update the database (<http://go.microsoft.com/fwlink/?LinkId=230269>).'. Under 'Troubleshooting tips:', there is a link 'Get general help for this exception.'

```
using Projeto01.Contexts;
using System.Linq;
using System.Web.Mvc;

namespace Projeto01.Controllers
{
    public class ProdutosController : Controller
    {
        private EFContext context = new EFContext();

        // GET: Produtos
        public ActionResult Index()
        {
            return View(context.Produtos.OrderBy(c => c.Nome));
        }

        // GET: Produtos/Details/5
        public ActionResult Details(int id)
        {
            return View();
        }
    }
}
```

Figura 4.1: Erro de base de dados desatualizada

4.3 INICIALIZADORES DE CONTEXTO DO ENTITY FRAMEWORK

Como pôde ser notado, o EF identifica mudanças nas classes que fazem parte do contexto e a inclusão ou exclusão dos conjuntos de entidades na classe de contexto. Mas como resolver? A mensagem de erro recomenda o uso do Code First Migrations, e nós faremos uso dele, mas mais para a frente. No momento,

precisamos entender as estratégias de inicialização do EF, e faremos isso conforme for sendo necessário em nosso processo de aprendizado.

Como o erro identificado aponta mudanças no modelo, estas precisam ser aplicadas na base de dados. Existe uma estratégia específica para este problema, e é ela que usaremos agora. Altere sua classe de contexto para o código na sequência.

Veja que agora o construtor foi implementado, e nele foi definida qual estratégia de inicialização para o contexto deve ser utilizada, a `DropCreateDataBaseIfModelChanges`. Veja que é mandado o tipo do contexto, `EFContext`, a qual se refere o inicializador, mesmo estando na classe dele.

Execute sua aplicação agora e requisite a action `Index` de `Produtos`. Funcionou, certo? Agora, acesse o `Index` de `Fabricantes`. Seus dados não aparecem mais, certo? Pois é, o `Drop` matou TODAS as tabelas da base de dados e as criou novamente, sem dados. Para evitar isso, o recomendado é realmente o *Code First Migrations*, mas o veremos apenas no capítulo 6. *Code First Migrations, Data Annotations, validações e jQueryUI*.:-)

```
public class EFContext : DbContext
{
    public EFContext() : base("Asp_Net_MVC_CS") {
        Database.SetInitializer<EFContext>(
            new DropCreateDatabaseIfModelChanges<EFContext>());
    }

    public DbSet<Categoria> Categorias { get; set; }
    public DbSet<Fabricante> Fabricantes { get; set; }
    public DbSet<Produto> Produtos { get; set; }
}
```

Alimentando as tabelas para testar a visão Index de Produtos

Agora que já temos a tabela de `Produtos` criada na base de dados, precisamos inserir registros nela para vermos como fica a visão `Index`. Insira primeiro novas `Categorias` e `Fabricantes`. Você pode fazer isso pela janela do Server Explorer ou pelas visões `Create` que já estão implementadas.

Para `Produtos`, faremos uso do Server Explorer, pois ainda não temos a visão `Create`. Selecione o menu `View|Server Explorer` e, na janela que se abre, expanda o banco da aplicação (`Asp_Net_MVC_CS`), depois expanda `Tables` e clique com o botão direito do mouse na tabela `Produtoes`, e em seguida em `Show Table Data`. Insira um novo produto e informe os `Ids` das chaves estrangeiras.

Sim, a tabela de produtos foi chamada de `Produtoes`, pois o EF trabalha a pluralização no idioma inglês. Logo veremos como configurar isso. Teste agora a visão `Index` de `Produtos`. A figura a seguir apresenta o recorte da visão.

Relação de PRODUTOS registrados			
ProdutoId	Nome	Categoria	Fabricante
1	Laserjet V		
Alterar Detalhes Eliminar			
Registrar um novo PRODUTO			

Figura 4.2: Visão index de Produtos sem exibir as associações

Observe, ainda nessa figura, que os nomes de `Categoria` e `Fabricante` não apareceram. Isso se deve ao fato de que o

carregamento dos objetos associados não ocorreu, pois o tipo de carregamento padrão é o tardio (Lazy loader).

Vamos forçar o carregamento destas associações. Veja o código a seguir para a action `Index` do controlador de produtos. Para implementar o código seguinte, inclua no topo da classe o código `using System.Data.Entity;`.

```
public ActionResult Index()
{
    var produtos = context.Produtos.Include(c => c.Categoria).
        Include(f => f.Fabricante).OrderBy(n => n.Nome);
    return View(produtos);
}
```

Teste novamente a visão `Index`. Legal agora, não é? Pois é. O método `Include()` permite a inclusão de associações nos objetos que serão retornados do `DbSet` em questão.

4.4 CRIANDO A VISÃO CREATE PARA A CLASSE PRODUTO

Com a visão `Index` de `Produtos` 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 categorias e de fabricantes. Os nomes `CategoriaId` e `FabricanteId` foram usados para estarem ligados com os controles `DropDownList` que serão usados na visão `Create`.

```
// GET: Produtos/Create
public ActionResult Create()
{
    ViewBag.CategoriaId = new SelectList(context.Categorias.
        OrderBy(b => b.Nome), "CategoriaId", "Nome");
```

```
ViewBag.FabricanteId = new SelectList(context.Fabricantes.  
    OrderBy(b => b.Nome), "FabricanteId", "Nome");  
return View();  
}
```

SELECTLIST()

O método `SelectList()` representa uma lista de itens da qual o usuário pode selecionar um item. Em nosso caso, os itens serão expostos em um `DropDownList`. O método 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.

Agora nos resta criar a visão `Create` . Veja na sequência o código implementado para ela.

Verifique o segundo e terceiro `<div>` , que contém a classe `form-group` . Note neles o helper `@Html.DropDownList()` . Veja o primeiro argumento, que se refere ao nome da propriedade na qual ele está "ligado". É por meio deste nome que ele buscará no `ViewBag` o conjunto de dados a serem utilizados.

Isso tudo é convenção. Você não precisa se preocupar em configurar nada, mas se você quiser, isso também é possível. Mas vamos lá, usar a convenção é muito melhor, não acha? Recomendo o artigo *Uma técnica simples para utilizar DropDownList no*

ASP.NET MVC, em <http://eduardopires.net.br/2014/08/tecnica-simples-dropdownlist-asp-net-mvc/>. Ele traz esta situação referente à criação de DropDownList no ASP.NET MVC.

```
@model Projeto01.Models.Produto

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Registrando um NOVO PRODUTO";
}

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Registro de um NOVO PRODUTO
        </div>
        <div class="panel-body">
            <div class="form-horizontal">
                @Html.ValidationSummary(true, "", new { @class = "text-danger" })
                <div class="form-group">
                    @Html.LabelFor(model => model.Nome,
                        htmlAttributes: new { @class =
                            "control-label col-md-2" })
                    <div class="col-md-10">
                        @Html.EditorFor(model => model.Nome,
                            new { htmlAttributes = new { @class =
                                "form-control" } })
                        @Html.ValidationMessageFor(model =>
                            model.Nome, "", new { @class =
                                "text-danger" })
                    </div>
                </div>
                <div class="form-group">
                    @Html.LabelFor(model => model.CategoriaId,
                        "CategoriaId", htmlAttributes: new {
                            @class = "control-label col-md-2" })
                    <div class="col-md-10">
```

```

        @Html.DropDownList("CategoriaId", null,
            htmlAttributes: new { @class =
                "form-control" })
        @Html.ValidationMessageFor(model =>
            model.CategoriaId, "", new { @class =
                "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.FabricanteId,
        "FabricanteId", htmlAttributes: new {
            @class = "control-label col-md-2"})
    <div class="col-md-10">
        @Html.DropDownList("FabricanteId",
            null, htmlAttributes: new { @class =
                "form-control" })
        @Html.ValidationMessageFor(model =>
            model.FabricanteId, "", new {
                @class = "text-danger" })
    </div>
</div>
</div>
<div class="panel-footer panel-info">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value=
            "Adicionar Produto" class=
            "btn btn-danger" />
        @Html.ActionLink("Retornar para a listagem
            de PRODUTOS", "Index", null, new {
                @class = "btn btn-warning" })
    </div>
    <br />
    <br />
    </div>
</div>
</div>
}
@section ScriptPage {
<script type="text/javascript">
    $(document).ready(function () {
        $('li').removeClass("active");
        $('#liProdutos').addClass("active");
    });
</script>

```

}

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

The screenshot shows a web form titled "Registro de um NOVO PRODUTO". It has three input fields: "Nome" (Name), "Categoriald" (Category ID), and "Fabricanteld" (Manufacturer ID). The "Nome" field is empty. The "Categoriald" field contains "Desktops" with a dropdown arrow. The "Fabricanteld" field contains "Acer" with a dropdown arrow. At the bottom, there are two buttons: a red "Adicionar Produto" (Add Product) button and an orange "Retornar para a listagem de PRODUTOS" (Return to the PRODUCTS list) button.

Figura 4.3: Visão Create de Produtos com os DropDownList para as classes associadas

Para finalizar este processo, é preciso implementar a action `Create` para o `POST`, que persistirá os dados informados na visão. Veja o código na sequência. Com exceção do uso do bloco `try..catch`, o código é semelhante ao utilizado para `Fabricantes`.

```
// POST: Produtos/Create
[HttpPost]
public ActionResult Create(Produto produto)
{
    try
    {
        context.Produtos.Add(produto);
        context.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

```
        catch
    {
        return View(produto);
    }
}
```

4.5 CRIANDO A VISÃO EDIT PARA A CLASSE PRODUTO

Com a inserção de novos produtos concluída, precisamos agora implementar a funcionalidade para alteração de dados de um produto. Falo da action `Edit`. O código seguinte apresenta o método que representa a action `Edit` para a geração da visão.

Observe que, na declaração dos `ViewBags`, existe agora um quarto parâmetro. Ele definirá o valor que deve ser considerado como selecionado no `DropDownList` e, em nosso caso, refere-se a `Categoria` e `Fabricante` do `Produto` a ser alterado.

```
// GET: Produtos/Edit/5
public ActionResult Edit(long? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.
            BadRequest);
    }
    Produto produto = context.Produtos.Find(id);
    if (produto == null)
    {
        return HttpNotFound();
    }
    ViewBag.CategoriaId = new SelectList(context.Categorias.
        OrderBy(b => b.Nome), "CategoriaId", "Nome", produto.
        CategoriaId);
    ViewBag.FabricanteId = new SelectList(context.Fabricantes.
        OrderBy(b => b.Nome), "FabricanteId", "Nome", produto.
        FabricanteId);
    return View(produto);
}
```

```
}
```

A visão a ser gerada pelo código anterior precisa ser implementada de acordo com o código seguinte (visão Edit). Comprove, fazendo uma leitura do código, que ele é muito (mas muito mesmo) semelhante ao da visão Create.

```
@model Projeto01.Models.Produto

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Visualizando os dados de um PRODUTO";
}

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Alteração dos dados de um PRODUTO
        </div>
        <div class="panel-body">
            <div class="form-horizontal">
                <h4>Produto</h4>
                <hr />
                @Html.ValidationSummary(true, "", new { @class = "text-danger" })
                @Html.HiddenFor(model => model.ProdutoId)

                <div class="form-group">
                    @Html.LabelFor(model => model.Nome,
                        htmlAttributes: new { @class =
                            "control-label col-md-2" })
                    <div class="col-md-10">
                        @Html.EditorFor(model => model.Nome,
                            new { htmlAttributes = new { @class =
                                "form-control" } })
                        @Html.ValidationMessageFor(model =>
                            model.Nome, "", new { @class =
                                "text-danger" })
                    </div>
                </div>
            </div>
        </div>
    </div>
```

```

</div>

<div class="form-group">
    @Html.LabelFor(model => model.CategoriaId,
        "CategoriaId", htmlAttributes: new {
            @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownList("CategoriaId", null,
            htmlAttributes: new { @class =
                "form-control" })
        @Html.ValidationMessageFor(model =>
            model.CategoriaId, "", new { @class =
                "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.FabricanteId,
        "FabricanteId", htmlAttributes: new {
            @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownList("FabricanteId",
            null, htmlAttributes: new { @class =
                "form-control" })
        @Html.ValidationMessageFor(model =>
            model.FabricanteId, "", new {
                @class = "text-danger" })
    </div>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value=
            "Gravar alterações" class=
            "btn btn-default" />
    </div>
</div>
</div>
</div>
</div>
}

@section ScriptPage {
    <script type="text/javascript">
        $(document).ready(function () {

```

```

        $( 'li' ).removeClass("active");
        $( '#liProdutos' ).addClass("active");
    });
</script>
}

```

Após a alteração dos dados, na visão `Edit`, o usuário requisitará a action `Edit` (`POST`) para persistir suas alterações. O código para esta action pode ser verificado na sequência. Novamente, com exceção da cláusula `try...catch`, o código é semelhante ao apresentado para `Fabricantes`.

```

[HttpPost]
public ActionResult Edit(Produto produto)
{
    try
    {
        if (ModelState.IsValid)
        {
            context.Entry(produto).State = EntityState.Modified;
            context.SaveChanges();
            return RedirectToAction("Index");
        }
        return View(produto);
    }
    catch
    {
        return View(produto);
    }
}

```

4.6 CRIANDO A VISÃO DETAILS PARA A CLASSE PRODUTO

A visão `Details` terá interação apenas com uma action `Details`, que será a responsável por renderizá-la. Após sua renderização, o usuário poderá retornar para a listagem (action `Index`) ou alterar os dados (action `Edit`). O código a seguir traz

a action `Details` que renderiza a visão. Note que, diferentemente do que foi feito em `Fabricantes` (que usava o método `Find()`), fiz agora uso do `Where()`, pois precisaremos dos dados das classes associadas, semelhante ao que foi feito na action `Index`.

```
// GET: Produtos/Details/5
public ActionResult Details(long? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.
            BadRequest);
    }
    Produto produto = context.Produtos.Where(p => p.ProdutoId == 
        id).Include(c => c.Categoria).Include(f => f.Fabricante).
        First();
    if (produto == null)
    {
        return HttpNotFound();
    }
    return View(produto);
}
```

O passo seguinte é a criação da visão `Details`. Siga os passos já vistos anteriormente para esta criação. O código final deverá ser o apresentado na sequência.

```
@model Projeto01.Models.Produto

 @{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Alterando os dados de um FABRICANTE";
}

<div class="panel panel-primary">
    <div class="panel-heading">
        Visualizando detalhes de um FABRICANTE
    </div>
    <div class="panel-body">
        <div class="form-group">
            @Html.LabelFor(model => model.Nome)<br />
            <div class="input-group">
```

```

        <span class="input-group-addon">
            <i class="glyphicon glyphicon-user"></i>
        </span>
        @Html.EditorFor(model => model.Nome,
            new { htmlAttributes = new { @class =
                "form-control", disabled = "disabled" } })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.Categoria.Nome)<br />
    <div class="input-group">
        <span class="input-group-addon">
            <i class="glyphicon glyphicon-user"></i>
        </span>
        @Html.EditorFor(model => model.Categoria.Nome,
            new { htmlAttributes = new { @class =
                "form-control", disabled = "disabled" } })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.Fabricante.Nome)<br />
    <div class="input-group">
        <span class="input-group-addon">
            <i class="glyphicon glyphicon-user"></i>
        </span>
        @Html.EditorFor(model => model.Fabricante.Nome,
            new { htmlAttributes = new { @class =
                "form-control", disabled = "disabled" } })
    </div>
</div>
<div class="panel-footer panel-info">
    @Html.ActionLink("Alterar", "Edit", new { id = Model.
        ProdutoId }, new { @class = "btn btn-info" })
    @Html.ActionLink("Retornar para a listagem", "Index",
        null, new { @class = "btn btn-info" })
</div>
</div>
@section ScriptPage {
    <script type="text/javascript">
        $(document).ready(function () {
            $('li').removeClass("active");
            $('#liProdutos').addClass("active");
        });
    </script>
}

```

```
}
```

4.7 CRIANDO A VISÃO DELETE PARA A CLASSE PRODUTO

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

```
// GET: Produtos/Delete/5
public ActionResult Delete(long? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.
            BadRequest);
    }
    Produto produto = context.Produtos.Where(p => p.ProdutoId ==
        id).Include(c => c.Categoria).Include(f => f.Fabricante).
        First();
    if (produto == null)
    {
        return HttpNotFound();
    }
    return View(produto);
}
```

Para a visão, trago uma inovação: o uso de caixas de diálogo modal do Bootstrap. Veja o código da visão na sequência. Verifique na <div> que representa o rodapé do panel a existência de um <a href> . Neste elemento, estão definidos dois atributos: data-toggle e data-target . O primeiro especifica para o Bootstrap que o link gerará uma janela "modal". O segundo atributo indica qual é a <div> que contém o modal a ser exibido.

Abaixo do fechamento da <div> do panel, está a <div> deleteConfirmationModal , que representa o modal. Veja que

existem partes para o modal tal qual para o panel (*header*, *body* e *footer*). Veja que, dentro do *body*, existe a declaração de um formulário e, dentro dele, está a declaração do *id* do produto como campo oculto (já vimos isso anteriormente). No rodapé, existem dois *buttons*, sendo o primeiro responsável pela submissão do formulário `delete-form`, que é o declarado no *body* do modal.

```
@model Projeto01.Models.Produto

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Visualizando detalhes do PRODUTO a ser removido";
}

<div class="panel panel-primary">
    <div class="panel-heading">
        Visualizando detalhes do PRODUTO a ser removido
    </div>
    <div class="panel-body">
        <div class="form-group">
            @Html.LabelFor(model => model.Nome)<br />
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="glyphicon glyphicon-user"></i>
                </span>
                @Html.EditorFor(model => model.Nome,
                    new { htmlAttributes = new { @class =
                        "form-control", disabled = "disabled" } })
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.Categoria.Nome)<br />
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="glyphicon glyphicon-user"></i>
                </span>
                @Html.EditorFor(model => model.Categoria.Nome,
                    new { htmlAttributes = new { @class =
                        "form-control", disabled = "disabled" } })
            </div>
        </div>
    </div>
</div>
```

```

        </div>
    </div>
<div class="form-group">
    @Html.LabelFor(model => model.Fabricante.Nome)<br />
    <div class="input-group">
        <span class="input-group-addon">
            <i class="glyphicon glyphicon-user"></i>
        </span>
    @Html.EditorFor(model => model.Fabricante.Nome,
        new { htmlAttributes = new { @class =
            "form-control", disabled = "disabled" } })
    </div>
    </div>
</div>
<div class="panel-footer panel-info">
    <a href="#" class = "btn btn-info" data-toggle="modal"
        data-target="#deleteConfirmationModal">Remover</a>
    @Html.ActionLink("Retornar para a listagem", "Index",
        null, new { @class = "btn btn-info" })
</div>
</div>

<div class="modal fade" id="deleteConfirmationModal" tabindex="-1'

    role="dialog" aria-hidden="true">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss=
                    "modal" aria-hidden="true">
                    &times;
                </button>
                <h4 class="modal-title">Confirmação de exclusão d
e
                    PRODUTO</h4>
            </div>
            <div class="modal-body">
                <p>
                    Você está prestes a remover o produto @Model.
                    Nome.ToUpper() do cadastro.
                </p>
                <p>
                    <strong>
                        Você está certo que deseja prosseguir?

```

```

        </strong>
    </p>
    @using (Html.BeginForm("Delete", "Produtos",
        FormMethod.Post, new { @id = "delete-form",
        role = "form" }))
    {
        @Html.HiddenFor(m => m.ProdutoId)
        @Html.AntiForgeryToken()
    }
</div>
<div class="modal-footer">
    <button type="button" class="btn btn-default"
            onclick="#('delete-form').submit();">
        Sim, exclua este produto.
    </button>
    <button type="button" class="btn btn-primary"
            data-dismiss="modal">
        Não, não exclua este produto.
    </button>
</div>
</div>
</div>
</div>

@section ScriptPage {
    <script type="text/javascript">
        $(document).ready(function () {
            $('li').removeClass("active");
            $('#liProdutos').addClass("active");
        });
    </script>
}

```

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

```

// POST: Produtos/Delete/5
[HttpPost]
public ActionResult Delete(long id)
{
    try
    {

```

```

        Produto produto = context.Produtos.Find(id);
        context.Produtos.Remove(produto);
        context.SaveChanges();
        TempData["Message"] = "Produto " + produto.Nome.ToUpper()
            + " foi removido";
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

```

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

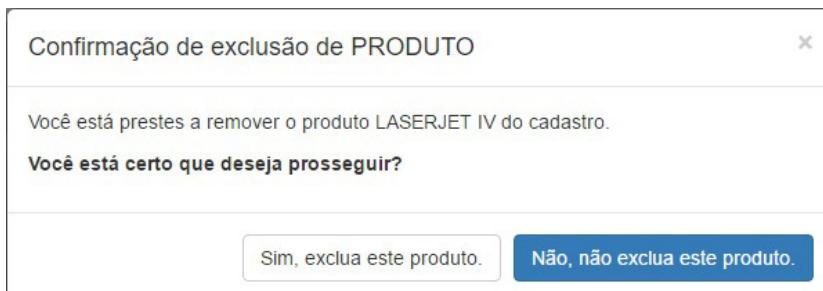


Figura 4.4: Visão DELETE com destaque para o MODAL de confirmação

4.8 ADAPTANDO A VISÃO DETAILS DE FABRICANTES

No início deste capítulo, implementamos a associação *um para muitos* nas classes `Categoria` e `Fabricante`. Precisamos agora fazer com que esta associação possa ser visualizada. Faremos isso, inicialmente, na visão `Details` de `Fabricante`. Com esta implementação, quando o usuário visualizar os detalhes de um `Fabricante`, ele terá acesso à listagem de `Produtos` do fabricante em questão.

Entretanto, para que a visão possa exibir os dados associados aos produtos, como `Categoria`, é preciso substituir o método `Find()` que existe na action `Details`, pelo seguinte:

```
Fabricante fabricante = context.Fabricantes.Where(f =>
f.FabricanteId == id).Include("Produtos.Categoria").First();
```

Já para a visão trarei o conceito de `Partial View`. Crie, na pasta `Views` de `Produtos`, um novo arquivo chamado `_PorFabricante.cshtml`, e nele insira o código seguinte.

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

<div class="panel panel-primary">
    <div class="panel-heading">
        Relação de PRODUTOS registrados para o fabricante
    </div>
    <div class="panel-body">
        <table class="table table-striped table-hover">
            <thead>
                <tr>
                    <th>
                        @Html.DisplayNameFor(model => model.
                            ProdutoId)
                    </th>
                    <th>
                        @Html.DisplayNameFor(model => model.Nome)
                    </th>
                    <th>Categoria</th>
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Model)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.
                                ProdutoId)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.
```

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

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 fosse um componente ou controle. Recomendo a leitura do artigo *Work with Partial view in MVC framework*, em <http://www.codeproject.com/Articles/821330/Work-with-Partial-view-in-MVC-framework>.

Com a implementação da Partial View concluída, precisamos fazer uso dela. Na visão Details de Fabricante , após o fechamento da <div> do Panel e antes de @section ScriptPage , insira:

```
@Html.Partial("~/Views/Produtos/_PorFabricante.cshtml"
, Model.Produtos.ToList()) .
```

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

The screenshot shows two views. The top view is titled "Visualizando detalhes de um FABRICANTE" and displays a form with a "Nome" field containing "Epson". Below the form are two buttons: "Alterar" and "Retornar para a listagem". A red arrow points downwards from the bottom of this view to the top of the second view. The second view is titled "Relação de PRODUTOS registrados para o fabricante" and contains a table with one row. The table has columns: "ProdutoId", "Nome", and "Categoria". The data in the table is: 1, Laserjet V, Impressoras. A red arrow points to the right side of the "Categoria" column for the single row.

ProdutoId	Nome	Categoria
1	Laserjet V	Impressoras

Figura 4.5: Visão Details com destaque para a listagem de Produtos para um Fabricante

Agora, o que acha de fazer o mesmo para a visão Categoria ? E o que acha de usar a Partial View criada para as visões Edit e Delete ?

RECOMENDAÇÕES DE LEITURA

Algumas leituras adicionais são sempre bem-vindas. Assim, seguem algumas recomendações.

Sobre o carregamento de objetos pelo Entity Framework, para maiores detalhes, veja os links <https://msdn.microsoft.com/en-us/data/jj574232.aspx> e <http://www.codeproject.com/Articles/788559>Loading-Related-Entities-with-Entity-Framework-A-B>.

Para um aprofundamento sobre estratégias de inicialização do Entity Framework, recomendo a leitura do artigo *Various Strategies to Initialize Database in Entity Framework*, de Sourav Kayal (<http://www.codeproject.com/Articles/794187/Various-Strategies-to-Initialize-Database-in-Entit>).

O uso de ViewBag tem também um excelente exemplo no artigo *What is ViewData, ViewBag and TempData? – MVC Options for Passing Data Between Current and Subsequent Request*, de Monjurul Habib (<http://www.codeproject.com/Articles/476967/What-is-ViewData-ViewBag-and-TempData-MVC-Option>), que vai um pouco além do proposto neste livro.

4.9 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, nas multiplicidades um para muitos e muitos para um, são mapeadas pelo Entity Framework.

Na aplicação destas associações pelo ASP.NET MVC, foram apresentadas as estratégias para carregamento de objetos, a passagem de valores do controlador para a visão (por meio de ViewBag), a criação de um `DropDownList` com uma coleção de objetos para que o usuário possa selecionar um, e a criação e utilização de Partial Views. Já com o Bootstrap foi trazida a utilização de janelas de diálogo modal.

Ufa, vimos bastante coisa. Agora, no próximo capítulo, trabalharemos algumas técnicas em relação a separação de camadas em nosso projeto. Será bem interessante também.

CAPÍTULO 5

SEPARANDO A APLICAÇÃO EM CAMADAS

Por mais que em nosso projeto ASP.NET MVC exista a separação entre as pastas `Models`, `Controllers` e `Views`, o conteúdo delas não está componentizado, pois pertencem a um único assembly. Pode-se dizer que um sistema componentizado é um sistema onde, em uma arquitetura, cada camada ou parte responsável está desenvolvida em módulos que são consumidos pelo sistema em si. Normalmente, estes módulos fazem parte de assemblies (DLLs).

Neste capítulo, separaremos partes deste projeto em outros, aplicando assim uma arquitetura para a solução. Também retiraremos algumas redundâncias de código, que foram comentadas no capítulo anterior.

5.1 CONTEXTUALIZAÇÃO SOBRE AS CAMADAS

Tudo o que fizemos até o momento, embora tenhamos usado o conceito de MVC e estarmos utilizando o ASP.NET MVC, foi feito

em apenas uma camada, pois tudo está em um único projeto. Além disso, nossas classes controladoras estão desempenhando muitas 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 na coesão, que busca medir um componente individualmente, temos as classes controladoras, que, em nosso caso, desempenham muitas funções, pois validam os dados, trabalham a persistência e ainda geram a comunicação com a visão. Com isso, temos uma baixa coesão em nossa aplicação.

A figura a seguir apresenta a estrutura de como nosso projeto ficará após este capítulo.

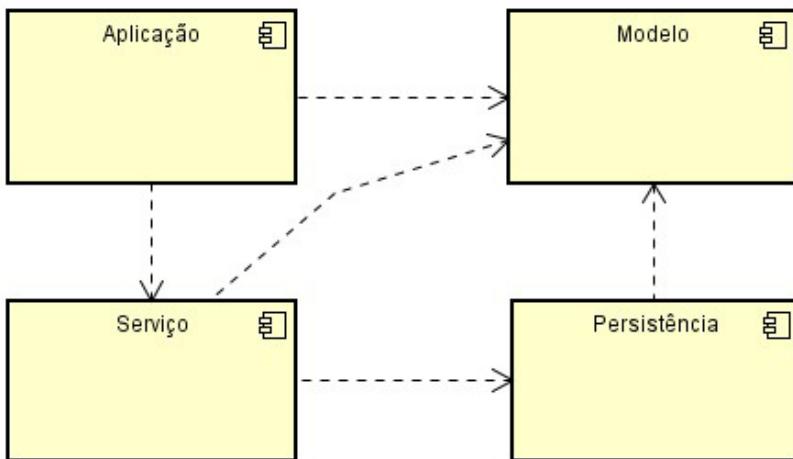


Figura 5.1: Diagrama de componentes do que se espera ao final deste capítulo

Cada componente do diagrama apresentado nessa figura será um projeto em nossa solução. Verifique que o `Modelo` não depende de ninguém, e que `Persistência` só depende do `Modelo`. Só com esta mudança já é possível verificar um ganho, pois nosso projeto de `Modelo` pode ser utilizado em uma aplicação para dispositivos móveis, por exemplo.

O projeto de `Aplicação`, que tem os controladores, agora não tem a responsabilidade de persistir os dados, pois você verá no código que os controladores devem apenas delegar as requisições submetidas às suas actions ao projeto `Serviço`. Desta maneira, a camada `Serviço` se tornará responsável em requisitar persistências e recuperações de objetos.

Vamos ao trabalho. :-)

5.2 CRIANDO A CAMADA DE NEGÓCIO – O MODELO

Como primeira atividade deste capítulo, criaremos um projeto do tipo `Library`, chamado `Modelo`. Para isso, clique com o botão direito sobre o nome da solução e selecione `Add->New Project`. Na janela que se apresenta, dentro da categoria `Visual C#` selecione `Windows (1)`. Na área central, selecione o template `Class Library (2)` e, para finalizar, nomeie o projeto como `Modelo (3)`. Confirme a criação do projeto clicando no botão `ok`. O assembly a ser gerado por este projeto será uma DLL.

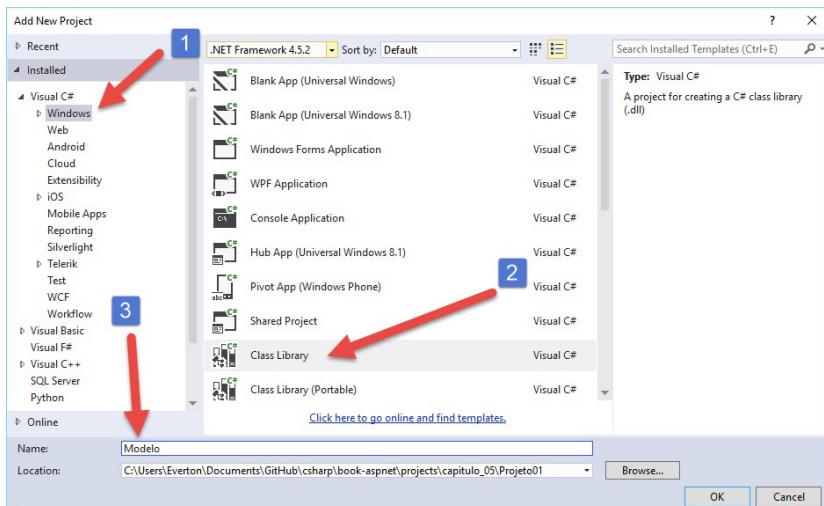


Figura 5.2: Janela para criação de projeto library para o modelo

No novo projeto, apague a classe `Class1.cs` criada pelo template, e crie duas pastas: `Tabelas` e `Cadastros`. Particionaremos nosso modelo em áreas (namespaces para o C#). Mova a classe `Categoria` para a pasta `Tabelas` do novo projeto, e altere o namespace dela para `Modelo.Tabelas`. Mova `Fabricante` e `Produto` para a pasta `Cadastros` no novo projeto, mudando o namespace para `Modelo.Cadastros`. Se não conseguir mover, copie as classes para os novos destinos e depois as apague na origem.

Precisamos agora referenciar, no projeto web, o novo projeto que representa o modelo de negócio de nossa aplicação. No projeto web, clique com o botão direito em `References` e depois em `Add Reference...`. Na janela que se abre, clique na categoria `Projects` e, no lado central, marque `Modelo`, depois clique no botão `OK` para concluir (figura a seguir).

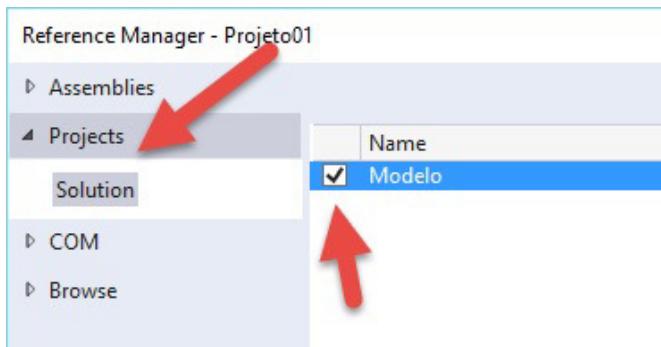


Figura 5.3: Adicionando a referência a um projeto da própria solução

Realize o build da solução. Diversos erros surgirão, pois faziam referência às classes que estavam na pasta `Models` do projeto da aplicação ASP.NET MVC e agora estas classes não existem mais, pois foram movidas para o projeto `Modelo`. Corrija estes problemas informando os novos namespaces para as classes, que são: `Modelo.Tabelas` e `Modelo.Cadastros`.

É preciso arrumar este problema também nas visões. Estes erros só serão apontados quando sua aplicação for executada. Antecipe este problema abrindo todas as visões e corrigindo 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, podemos agora, para o mesmo modelo, criar uma aplicação mobile, desktop ou de serviços.

5.3 CRIANDO A CAMADA DE PERSISTÊNCIA

Da mesma maneira que foi criado o projeto `Modelo`, crie

agora o `Persistencia`. Adicione ao projeto a referência ao projeto `Modelo` e instale nele o Entity Framework. Crie uma pasta chamada `Contexts` e mova para lá a classe `EFContext`, que existe no projeto da aplicação MVC. Corrija o nome do namespace da classe. Crie uma pasta chamada `DAL` (de *Data Access Layer* - Camada de Acesso a Dados) e, dentro desta pasta, crie outras duas: `Cadastros` e `Tabelas`.

Primeiramente, adaptaremos a classe `EFContext` para resolver o problema de pluralização, que causou com que a tabela que mapeasse o `DbSet<Produto>()` se chamassem `Produtos`. Para isso, sobrescreveremos o método `OnModelCreating()`, como segue na listagem a seguir.

```
protected override void OnModelCreating(
    DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Conventions.
        Remove<PluralizingTableNameConvention>();
}
```

Criaremos três classes DAL. É nelas que todo trabalho relacionado à persistência se concentrará. Vamos começar criando, na pasta `Tabelas`, a classe `CategoriaDAL`. Veja no código a seguir como deve ficar esta classe neste momento. Observe que o contexto é declarado no início da classe que criamos e, por enquanto, apenas o método que retorna todas as categorias classificadas (ordenadas) pelo nome está implementado.

```
using Modelo.Cadastros;
using Persistencia.Contexts;
using System.Linq;

namespace Persistencia.DAL.Tabelas
{
```

```

public class CategoriaDAL
{
    private EFContext context = new EFContext();

    public IQueryable<Categoria>
        ObterCategoriasClassificadasPorNome()
    {
        return context.Categorias.OrderBy(b => b.Nome);
    }
}
}

```

Precisamos também criar a classe `FabricanteDAL` na pasta `Cadastros`. Por enquanto, esta classe tem o mesmo método que a `CategoriaDAL`, pois precisaremos destes dados para implementar o CRUD de `Produtos` em nossa arquitetura. Veja o código a seguir.

Note que, logo no início da classe, é definido um campo privado responsável pelo contexto com EF. Este campo será usado por todos os métodos da classe. O método

`ObterFabricantesClassificadosPorNome()` retorna o conteúdo do `DbSet Fabricantes`, classificados pelo nome. Essa classificação é realizada pela chamada ao método LINQ `OrderBy()`, que recebe um lambda, que aponta para a propriedade `Nome`, sendo esta a que será utilizada para a classificação.

```

using Modelo.Cadastros;
using Persistencia.Contexts;
using System.Linq;

namespace Persistencia.DAL.Cadastros
{
    public class FabricanteDAL
    {
        private EFContext context = new EFContext();

```

```

        public IQueryable<Fabricante>
            ObterFabricantesClassificadosPorNome()
        {
            return context.Fabricantes.OrderBy(b => b.Nome);
        }
    }
}

```

Finalizando a implementação da camada de persistência, precisamos implementar, na pasta `Cadastros`, a classe `ProdutoDAL`, conforme listagem a seguir. Observe, no método `GravarProduto()`, que nele estão encapsuladas a inserção de um novo produto e a alteração de um já existente. É importante que você note nestas três classes que elas não resolvem nada que não seja relativo à persistência de dados, e dependem apenas do modelo de negócio, favorecendo assim o acoplamento e a coesão.

O método `ObterProdutosClassificadosPorNome()` faz uso da chamada ao método `Include()` para que as classes das propriedades `Categoria` e `Fabricante` sejam incluídas no mapeamento para a seleção dos registros. Esta inclusão também se repete no método `ObterProdutoPorId()`. Com estes procedimentos, forçamos a carga dos objetos (Eager).

```

using Persistencia.Contexts;
using System.Linq;
using System.Data.Entity;
using Modelo.Cadastros;

namespace Persistencia.DAL.Cadastros
{
    public class ProdutoDAL
    {
        private EFContext context = new EFContext();

        public IQueryable<Produto> ObterProdutosClassificadosPorNome()
        {

```

```

        return context.Produtos.Include(c => c.Categoria).
            Include(f => f.Fabricante).OrderBy(n => n.Nome);
    }

    public Produto ObterProdutoPorId(long id)
    {
        return context.Produtos.Where(p => p.ProdutoId == id)

            Include(c => c.Categoria).Include(f =>
                f.Fabricante).First();
    }

    public void GravarProduto(Produto produto)
    {
        if (produto.ProdutoId == null)
        {
            context.Produtos.Add(produto);
        } else
        {
            context.Entry(produto).State =
                EntityState.Modified;
        }
        context.SaveChanges();
    }

    public Produto EliminarProdutoPorId(long id)
    {
        Produto produto = ObterProdutoPorId(id);
        context.Produtos.Remove(produto);
        context.SaveChanges();
        return produto;
    }
}
}

```

5.4 CRIANDO A CAMADA DE SERVIÇO

Precisamos agora garantir que a camada da aplicação, que é o projeto ASP.NET MVC, possa persistir ou recuperar dados da base de dados. Porém, não podemos expor a camada de persistência para a aplicação. Devemos ter uma camada intermediária, que é a

de serviços.

Da mesma maneira que criamos os dois projetos anteriores, vamos criar agora um que se chame `Servicos`. Adicione a ele as referências para os projetos `Persistencia` e `Modelo`. Para mantermos o padrão, crie duas pastas neste projeto: a `Tabelas` e `Cadastros`. Crie, na pasta `Tabelas`, a classe `CategoriaServico`, com o código apresentado a seguir.

```
using Modelo.Cadastros;
using Persistencia.DAL.Tabelas;
using System.Linq;

namespace Servicos.Tabelas
{
    public class CategoriaServico
    {
        private CategoriaDAL categoriaDAL = new CategoriaDAL();

        public IQueryable<Categoria>
            ObterCategoriasClassificadasPorNome()
        {
            return categoriaDAL.
                ObterCategoriasClassificadasPorNome();
        }
    }
}
```

Note no código anterior a definição do DAL para uso em toda a classe. Verifique também que o nome do método para o serviço é o mesmo que utilizamos na classe DAL. Na listagem a seguir, com as mesmas observações, segue o código para a classe `FabricanteServico`, que deve ser criada na pasta `Cadastros`.

```
using Modelo.Cadastros;
using Persistencia.DAL.Cadastros;
using System.Linq;

namespace Servicos.Cadastros
```

```

{
    public class FabricanteServico
    {
        private FabricanteDAL fabricanteDAL = new FabricanteDAL();

        public IQueryable<Fabricante>
            ObterFabricantesClassificadosPorNome()
        {
            return fabricanteDAL.
                ObterFabricantesClassificadosPorNome();
        }
    }
}

```

Finalizando a camada de serviços, vamos implementar a classe `ProdutoServico` na pasta `Cadastros`. O código deve ser o apresentado na listagem a seguir. Nesta classe, definimos os métodos que poderão ser consumidos pela aplicação. Note, no início da classe, que declaramos o campo `produtoDAL`, que será usado por todos os métodos. Veja que em cada método este campo é utilizado para atender ao serviço desejado, apenas chamando por ele, não executando nenhuma lógica adicional.

```

using Modelo.Cadastros;
using Persistencia.DAL.Cadastros;
using System.Linq;

namespace Servicos.Cadastros
{
    public class ProdutoServico
    {
        private ProdutoDAL produtoDAL = new ProdutoDAL();

        public IQueryable<Produto> ObterProdutosClassificadosPorNome()
        {
            return produtoDAL.ObterProdutosClassificadosPorNome();
        }
    }
}

```

```

public Produto ObterProdutoPorId(long id)
{
    return produtoDAL.ObterProdutoPorId(id);
}

public void GravarProduto(Produto produto)
{
    produtoDAL.GravarProduto(produto);
}

public Produto EliminarProdutoPorId(long id)
{
    return produtoDAL.EliminarProdutoPorId(id);
}
}
}

```

5.5 ADAPTANDO A CAMADA DE APLICAÇÃO

Com a criação de nossa arquitetura, é preciso agora adaptar os controladores. Veremos a adaptação do `ProdutosController`. A primeira mudança é retirar do código que existe a declaração do `EFContext`, pois agora nossa aplicação não conhece nada de persistência. Quem terá este conhecimento é a camada `Servico`. Para isso, precisamos declarar os serviços que serão usados neste controlador no início da classe. Veja o código a seguir.

```

namespace Projeto01.Controllers
{
    public class ProdutosController : Controller
    {
        private ProdutoServico produtoServico =
            new ProdutoServico();
        private CategoriaServico categoriaServico =
            new CategoriaServico();
        private FabricanteServico fabricanteServico =
            new FabricanteServico();
    }
}

```

Como primeira action a ser alterada, segue o código para a

`Index` . Veja o uso do método definido na classe de serviços para produtos. Já não existe nada referente ao contexto do Entity Framework.

```
public ActionResult Index()
{
    return View(produtoServico.
        ObterProdutosClassificadosPorNome());
}
```

A segunda implementação alterando o controlador resolverá um problema de redundância de código. Se observar as actions `GET Details` , `Edit` e `Delete` , a recuperação do produto a ser retornado para a visão é semelhante. Desta maneira, buscando eliminar este problema, criaremos um método privado na própria classe e apenas vamos usá-lo nas actions. Veja o método na listagem a seguir. Observe, no corpo do método, a chamada ao serviço que retornará o produto solicitado pelo usuário.

```
private ActionResult ObterVisaoProdutoPorId(long? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(
            HttpStatusCode.BadRequest);
    }
    Produto produto = produtoServico.ObterProdutoPorId((long) id);
;
    if (produto == null)
    {
        return HttpNotFound();
    }
    return View(produto);
}
```

Agora, com o método criado, podemos fazer o uso nas actions que precisam recuperar um produto e retorná-lo para a visão. Veja os códigos delas a seguir.

```

public ActionResult Details(long? id)
{
    return ObterVisaoProdutoPorId(id);
}

public ActionResult Delete(long? id)
{
    return ObterVisaoProdutoPorId(id);
}

```

A action `Edit` precisa de uma mudança a mais, antes de a trazermos para cá. Veja em seu código, ainda não modificado, que existe o fato de passar as categorias e fabricantes para a `ViewBag`, que popularão os `DropDownLists` da visão. Esta mesma situação ocorre para a action `GET Create`. Sendo assim, vamos criar um método privado que resolverá este problema. Ele pode ser verificado na sequência.

Veja que, na assinatura do método, o parâmetro `produto` é opcional. E quando ele não existir, é atribuído `null` a ele. Isso foi adotado para podermos diferenciar quando o quarto parâmetro do `SelectList()` será informado.

```

private void PopularViewBag(Produto produto = null)
{
    if (produto == null)
    {
        ViewBag.CategoriaId = new SelectList(categoriaServico.
            ObterCategoriasClassificadasPorNome(),
            "CategoriaId", "Nome");
        ViewBag.FabricanteId = new SelectList(fabricanteServico.
            ObterFabricantesClassificadosPorNome(),
            "FabricanteId", "Nome");
    }
    else
    {
        ViewBag.CategoriaId = new SelectList(categoriaServico.
            ObterCategoriasClassificadasPorNome(),
            "CategoriaId", "Nome", produto.CategoriaId);
        ViewBag.FabricanteId = new SelectList(fabricanteServico.

```

```
        ObterFabricantesClassificadosPorNome(),
        "FabricanteId", "Nome", produto.FabricanteId);
    }
}
```

Com a criação do método, já podemos trabalhar nas actions `Edit` e `Create`. Veja na sequência como elas devem ficar.

```
public ActionResult Edit(long? id)
{
    PopularViewBag(produtoServico.ObterProdutoPorId((long)id));
    return ObterVisaoProdutoPorId(id);
}

public ActionResult Create()
{
    PopularViewBag();
    return View();
}
```

Com a implementação das action `GET`, nos restam agora as que respondem ao `HTTP POST`. Duas delas referem-se à atualização ou inserção de um produto. Se você notar em seu código, elas são semelhantes. Desta maneira, mais uma vez vamos criar um método privado para responder a estas requisições. Veja este código:

```
private ActionResult GravarProduto(Produto produto)
{
    try
    {
        if (ModelState.IsValid)
        {
            produtoServico.GravarProduto(produto);
            return RedirectToAction("Index");
        }
        return View(produto);
    }
    catch
    {
        return View(produto);
    }
}
```

```
    }  
}
```

Veja na implementação anterior que há a invocação ao método de serviço para gravar o produto, independente de ser atualização ou inserção. Na sequência, verifique a implementação das actions POST Edit e Create .

```
[HttpPost]  
public ActionResult Create(Produto produto)  
{  
    return GravarProduto(produto);  
}  
  
[HttpPost]  
public ActionResult Edit(Produto produto)  
{  
    return GravarProduto(produto);  
}
```

Para finalizar, vamos implementar a action POST para a remoção de um registro. Veja o código na sequência. A primeira instrução, dentro do bloco try , invoca o método EliminarProdutoId() , que retornará o produto removido. A segunda instrução armazena no TempData a mensagem que será exibida na visão, após a remoção ser concluída.

```
[HttpPost]  
public ActionResult Delete(long id)  
{  
    try  
    {  
        Produto produto = produtoServico.EliminarProdutoPorId(id)  
;  
        TempData["Message"] = "Produto " + produto.Nome.ToUpper()  
            + " foi removido";  
        return RedirectToAction("Index");  
    }  
    catch  
    {
```

```
        return View();
    }
}
```

5.6 ADAPTANDO AS VISÕES PARA MINIMIZAR REDUNDÂNCIAS

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

Para começar, vamos separar o conteúdo referente ao conteúdo dos Panels. Na pasta `Views` de `Produtos`, crie uma visão chamada `_PartialDetailsContentPanel.cshtml` e implemente nela o código a seguir. Este é o mesmo processo que já foi utilizado na visão `Details` de `Produto`. Nele são codificados os controles que serão renderizados para apresentação dos dados a serem visualizados.

```
@model Modelo.Cadastros.Produto

<div class="form-group">
    @Html.LabelFor(model => model.Nome)<br />
    <div class="input-group">
        <span class="input-group-addon">
            <i class="glyphicon glyphicon-user"></i>
        </span>
        @Html.EditorFor(model => model.Nome, new { htmlAttributes =
            new { @class = "form-control", disabled =
                "disabled" } })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.Categoria.Nome)<br />
    <div class="input-group">
```

```

<span class="input-group-addon">
    <i class="glyphicon glyphicon-user"></i>
</span>
@Html.EditorFor(model => model.Categoria.Nome, new {
    htmlAttributes = new { @class = "form-control",
        disabled = "disabled" } })
</div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.Fabricante.Nome)<br />
    <div class="input-group">
        <span class="input-group-addon">
            <i class="glyphicon glyphicon-user"></i>
        </span>
        @Html.EditorFor(model => model.Fabricante.Nome, new {
            htmlAttributes = new { @class = "form-control",
                disabled = "disabled" } })
    </div>
</div>

```

Agora, para utilizar esta Partial View, na visão Details , adapte a <div> panel-body para ficar de acordo com o apresentado na sequência. Veja que todo o código que existia nela e que foi levado para a partial view apresentada na listagem anterior foi substituído pela invocação da instrução @Html.Partial() , que recebe o arquivo da partial view que será trazida para a visão.

```

<div class="panel-body">
    @Html.Partial("~/Views/Produtos/_PartialDetailsContentPanel.
        cshtml", Model)
</div>

```

Agora uma atividade. Adapte Categorias e Fabricantes para a nossa arquitetura, e teste sua aplicação.

RECOMENDAÇÕES DE LEITURA

Algumas leituras adicionais são sempre bem-vindas. Desta maneira, seguem algumas recomendações.

Leia o artigo *Coesão e Acoplamento em Sistemas OO* (https://www.researchgate.net/publication/261026207_Coesao_e_Acoplamento_em_Sistemas_OO) para um aprofundamento sobre coesão e acoplamento.

5.7 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 reutilização, pois trabalha os pontos relacionados a coesão e acoplamento.

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

CAPÍTULO 6

CODE FIRST MIGRATIONS, DATA ANNOTATIONS, VALIDAÇÕES E JQUERYUI

As últimas atualizações que vínhamos implementando em nosso modelo de negócio causavam a exclusão das tabelas e, consequentemente, todos os dados eram também removidos. A apresentação do Code First Migration neste capítulo permitirá que mudanças que foram realizadas no modelo e na execução do projeto sejam refletidas na base de dados, sem perda de dados.

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

Finalizando este capítulo, será apresentado o jQueryUI para que seja possível exibir um calendário em um campo que seja do tipo `Data`. Bom estudo. :-)

6.1 FAZENDO USO DO CODE FIRST MIGRATIONS

Quando realizamos mudanças nas classes que representam o modelo de negócio e são mapeadas para tabelas na base de dados, o banco é eliminado e criado novamente, o que causa a perda de todos os dados que tínhamos registrados. Este processo é ruim, pois precisamos sempre inserir elementos para nossos testes.

Existem algumas formas para fugir deste problema. A mais indicada é por meio do uso do *Code First Migrations*. Desta maneira, precisamos habilitá-lo para o projeto `Persistencia`. Para isso, selecione o menu `Tools>Nuget Package Manager>Package Manager Console`.

A janela semelhante à figura a seguir será aberta para você. Confirme se o projeto em `Default Project` é o projeto `Persistencia`, e digite `Enable-Migrations` no prompt do Package Manager Console.

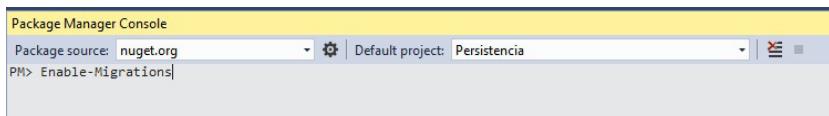


Figura 6.1: Habilitando o Code First Migrations

Após a instalação do Code First Migration no projeto, uma pasta chamada `Migrations` é inserida em sua estrutura. Dentro dela, são criados dois arquivos: um com um número grande, terminando com `_InitialCreate` e outro, chamado `Configuration`. O primeiro traz o código que mapeia a atual estrutura de sua base de dados, com base na classe `EFContext`. O

segundo é responsável pela configuração pertinente ao Migrations. Abra estes arquivos e leia-os.

No arquivo Configuration.cs , no construtor mude a instrução AutomaticMigrationsEnabled = false; para AutomaticMigrationsEnabled = true; . No arquivo EFContext.cs , que está na pasta Contexts , mude o construtor para Database.SetInitializer<EFContext>(new MigrateDatabaseToLatestVersion<EFContext>, Configuration>()); . Atenção para o Configuration() , ele é o que está na classe Configuration , dentro da pasta Migrations .

Com estas alterações, o Code First Migrations está instalado e configurado. Para testarmos, precisamos mudar nosso modelo e o faremos a seguir.

6.2 ADAPTANDO A CLASSE PRODUTO PARA AS VALIDAÇÕES

Para que possamos fazer uso do recurso referente ao Data Annotations, precisamos inserir no projeto Modelo o assembly System.ComponentModel.DataAnnotations . Para isso, clique com o botão direito em References no projeto Modelo e, em seguida, em Add Reference.... Na janela que se abre, clique em Assemblies e localize o System.ComponentModel.DataAnnotations . Marque-o, e então clique no botão OK.

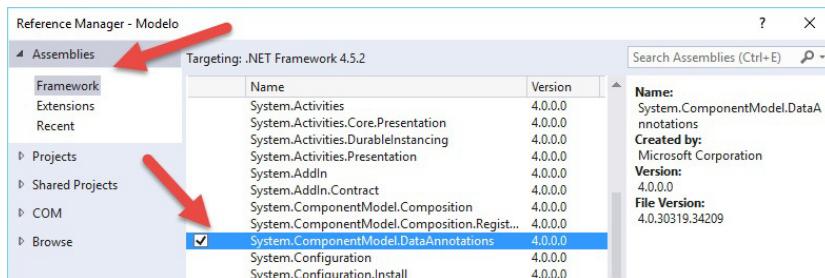


Figura 6.2: Inserindo assembly para Data Annotations

Além de podermos aplicar regras de validação por meio de atributos, é possível também aplicar algumas características para as propriedades. Veja na sequência o novo código para a classe `Produto`. Adapte sua classe para ficar igual a este código.

Observe os atributos `[DisplayName()]`. Quando a visão for fazer uso do nome da propriedade, por meio de HTML Helpers, será usado o texto informado no atributo, e não mais o nome da propriedade. O atributo `[StringLength()]` determina o tamanho máximo e mínimo para uma string. Se este não for utilizado, toda string terá o mapeamento para uma coluna na tabela com o tamanho máximo. O terceiro atributo usado no código é o `[Required()]` que torna o preenchimento do controle mapeado para a propriedade obrigatório.

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace Modelo.Cadastros
{
    public class Produto
    {
        [DisplayName("Id")]
        public long? ProdutoId { get; set; }
        [StringLength(100, ErrorMessage = "O nome do produto")

```

```

        precisa ter no mínimo 10 caracteres",
        MinimumLength = 10)]
[Required(ErrorMessage = "Informe o nome do produto")]
public string Nome { get; set; }
[DisplayName("Data de Cadastro")]
[Required(ErrorMessage = "Informe a data de cadastro
    do produto")]
public DateTime? DataCadastro { get; set; }

[DisplayName("Categoria")]
public long? CategoriaId { get; set; }
[DisplayName("Fabricante")]
public long? FabricanteId { get; set; }

public Categoria Categoria { get; set; }
public Fabricante Fabricante { get; set; }
}
}

```

Agora, com a mudança no modelo implementada, podemos testar o Code First Migrations. Execute sua aplicação. Caso, nesta execução, ocorra o erro apresentado na figura a seguir, insira no construtor da classe Configuration o código AutomaticMigrationDataLossAllowed = true;. Esta configuração faz com que os dados possam ser perdidos durante a atualização.

Sei que é isso que estamos buscando, mas acredite, nenhum dado será perdido. Execute novamente sua aplicação. Agora vá ao Server Explorer e verifique a alteração na estrutura da tabela Produto , e verifique os dados em todas as tabelas. Eles estão lá. Agora, altere esta implementação que fez para atribuir false .

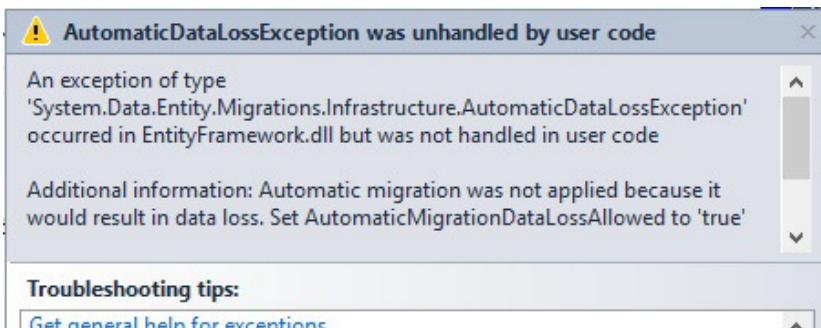


Figura 6.3: Erro ao executar aplicação com atualizações no Migrations

6.3 TESTANDO AS ALTERAÇÕES IMPLEMENTADAS

Execute sua aplicação e acesse a listagem de produtos. Note que os títulos das colunas mudaram. Para a propriedade `ProdutoId`, agora é exibido apenas `Id`; para `CategoriaId`, apenas `Categoria`; e para `FabricanteId`, apenas `Fabricante`. Já na página que permite a inserção de um novo produto, esta mudança não ocorreu. Mas é por causa da maneira como o Visual Studio cria os templates.

Na visão `Create`, localize o HTML Helper `@Html.LabelFor()`, e retire dele o argumento que tem o nome da propriedade (`CategoriaId` e `FabricanteId`). Inclusive, você não precisa parar a aplicação para realizar mudanças na visão. Pode mudar e atualizar a página no navegador.

Com as alterações relacionadas à camada de visão validada, precisamos agora testar as validações. Na visão `Create`, tente inserir um produto sem nome, e depois gravar. Possivelmente, você receberá o erro apontado pela figura a seguir. Este ocorre

justamente pela ocorrência de erros na validação do modelo.

Na execução do método privado, no controlador de produtos `GravarProduto()`, há a validação do modelo pelo código `if (ModelState.IsValid)`. Até aí, tudo bem. Há erros e a visão será novamente renderizada, com o objeto `produto` como modelo. O problema que ocorre é que o `ViewBag` requisitado na visão para popular os `DropDownList`s não existe mais. Precisamos inseri-lo.

Insira a instrução `PopularViewBag(produto);` antes dos dois `return View(produto);` que existem no método `GravarProduto()`. Execute novamente sua aplicação e realize o teste outra vez.

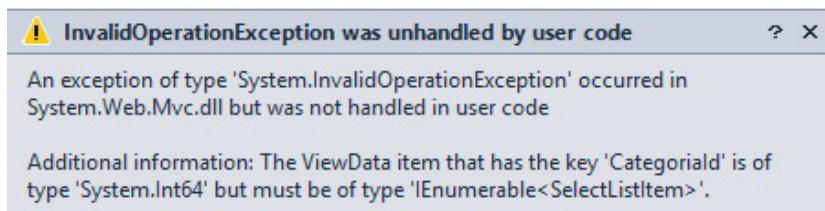


Figura 6.4: Erro ao inserir um produto com o nome em branco

Agora, no teste que se resume em tentar registrar um novo produto (sem informar o nome), se tudo der certo, sua página deve estar semelhante a apresentada pela figura a seguir. Caso os erros do `ValidationSummary()` não apareçam, é preciso alterar um parâmetro da chamada para que os erros possam aparecer. Sendo assim, se seu teste não resultou na semelhança da figura a seguir, troque o `true` por `false`.

Pelos erros exibidos no `Summary`, é possível ver que precisamos preencher a propriedade `DataCadastro`, mas não

temos controle para ela. Vamos implementar isso na sequência.

The screenshot shows a registration form titled 'Registro de um NOVO PRODUTO'. It includes fields for 'Nome' (Name), 'Categoria' (Category), and 'Fabricante' (Manufacturer). A red arrow points to the 'Nome' field, which has a validation error message: 'Informe o nome do produto'. Another red arrow points to the 'Categoria' field, which also has a validation error message: 'Informe a data de cadastro'. At the bottom, there are buttons for 'Adicionar Produto' (Add Product) and 'Retornar para a listagem de PRODUTOS' (Return to PRODUCT LIST).

Figura 6.5: Erros gerados pela validação

6.4 IMPLEMENTANDO O CONTROLE DE DATA

Como visto anteriormente, precisamos agora implementar um controle para receber a data de cadastro de um produto. Podemos aproveitar este momento e refatorar as visões `Edit` e `Create`, que possuem os mesmos controles. Vamos fazer a adaptação tal qual fizemos no capítulo anterior para as visões `Details` e `Delete`.

Crie uma visão como `Partial View`, nomeie-a de `_PartialEditContentPanel.cshtml` e a deixe tal qual o código apresentado na sequência. Neste código, é possível verificar que ele implementa os controles que deverão ser renderizados, tanto para criação de um novo produto como para a alteração de um já existente, uma vez que fazem parte do mesmo modelo.

```
@model Modelo.Cadastros.Produto
```

```

<div class="form-horizontal">
    @Html.ValidationSummary(false, "Verifique os seguintes
        erros", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(model => model.Nome, htmlAttributes:
            new { @class = "control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.Nome, new {
                htmlAttributes = new { @class = "form-control" } })
        )
        @Html.ValidationMessageFor(model => model.Nome,
            "", new { @class = "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.DataCadastro,
        htmlAttributes: new { @class = "control-label
            col-md-2" })
    <div class="col-md-3">
        @Html.EditorFor(model => model.DataCadastro, new {
            htmlAttributes = new { @class = "form-control" } })
    )
    @Html.ValidationMessageFor(model =>
        model.DataCadastro, "", new { @class =
            "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.CategoriaId,
        htmlAttributes: new { @class = "control-label
            col-md-2" })
    <div class="col-md-10">
        @Html.DropDownList("CategoriaId", null,
            htmlAttributes: new { @class = "form-control" })
        @Html.ValidationMessageFor(model =>
            model.CategoriaId, "", new { @class =
            "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.FabricanteId,
        htmlAttributes: new { @class = "control-label
            col-md-2" })

```

```
    col-md-2" })
<div class="col-md-10">
    @Html.DropDownList("FabricanteId", null,
        htmlAttributes: new { @class = "form-control" })
    @Html.ValidationMessageFor(model =>
        model.FabricanteId, "", new { @class =
        "text-danger" })
</div>
</div>
</div>
```

Da maneira que implementamos, a data precisa ser informada com a respectiva máscara. A seguir, faremos uso do jQueryUI para trazer alguns recursos interessantes para este tipo de propriedade. É preciso agora alterar a visão `Create` para usar essa Partial View. Veja na sequência a alteração realizada. Adapte a visão `Edit` também. :-)

```
<div class="panel-body">
    @Html.Partial("~/Views/Produtos/
        _PartialEditContentPanel.cshtml", Model)
</div>
```

6.5 FAZENDO USO DO JQUERYUI PARA CONTROLES DE DATA

Para utilizar o jQueryUI, precisamos realizar o seu download. Você pode fazer isso diretamente pelo link <https://jqueryui.com/>, onde também pode ver exemplos dos recursos disponibilizados. É possível também fazer o uso do NuGet. Desta vez, faremos uso do download.

Veja na figura a seguir o destaque da página de download. Você deverá clicar em `Custom Download`, conforme o destaque.



Figura 6.6: Página para download do jQueryUI

Na página que se abre, desmarque o Checkbox `Toggle All` em `Components`. Você pode optar por deixar ele marcado, pois assim o jQueryUI que baixar estará completo, com todos os componentes. Entretanto, em nosso caso, faremos uso apenas do `DateTimePicker`, que você encontrará em `Widgets`. Todas as dependências para este controle serão automaticamente selecionadas.

No final da página, clique em `Download`. Descompacte a pasta baixada dentro da pasta `Scripts` de seu projeto. Você pode, se quiser, criar uma subpasta para organizar melhor. Eu criei. :-)

Vamos ao uso do controle agora. Na visão `Create`, na sessão de scripts, é preciso importar o JavaScript do jQueryUI. Também é preciso ativar o controle da data. Veja que o jQuery faz a invocação pelo ID do controle (que mostraremos já esta mudança também). Veja no código da sequência a nova implementação para a sessão de JavaScript.

```
@section ScriptPage {
    <script src("~/Scripts/jquery-ui-1.11.4.custom/
        jquery-ui.js"></script>
    <script type="text/javascript">
        $(document).ready(function () {
            $('li').removeClass("active");
            $('#liProdutos').addClass("active");
            $("#datepicker").datepicker();
        });
    </script>
}
```

Para o uso dos controles disponibilizados pelo jQueryUI, é preciso trazermos também os estilos disponibilizados para a visão. Desta maneira, é necessário inserir uma nova sessão na visão. Inclua o código a seguir antes da sessão de scripts.

Vale aqui uma análise. Se os controles do jQueryUI forem usados em muitas visões, eles podem ser inseridos na visão de layout, e não nas visões em que utilizamos o layout.

```
@section styles{
    <link href("~/Scripts/jquery-ui-1.11.4.custom/
        jquery-ui.css" rel="stylesheet" />
}
```

Para concluir o uso do controle `DateTimePicker`, precisamos fazer mais duas coisas:

1. Altere o HTML Helper do controle, como mostrado na

sequência. O que foi alterado diz respeito apenas ao ID para o controle;

2. Acesse o link <https://github.com/jquery/jquery-ui/tree/master/ui/i18n>, e baixe o arquivo datepicker-pt-BR.js para que os textos do controle apareçam em português. Isso se chama *internacionalização* (Internationalization ou i18n).

```
@Html.EditorFor(model => model.DataCadastro, new {  
    htmlAttributes = new { @class = "form-control",  
        id="datepicker" } })
```

A figura a seguir apresenta o controle do jQueryUI renderizado.

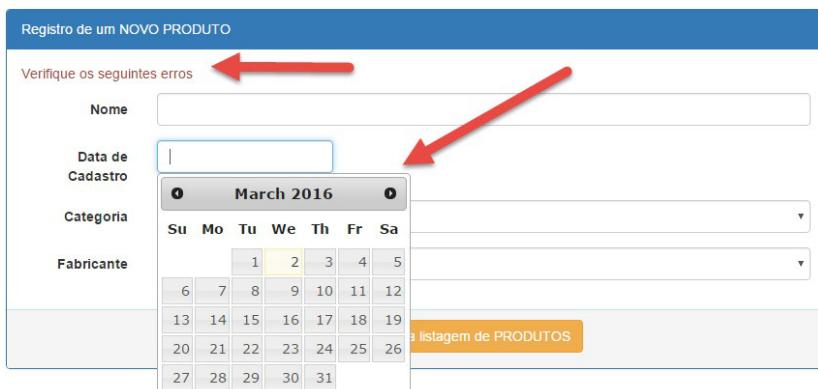


Figura 6.7: DateTimePicker do jQueryUI

Caso você não queira fazer uso do controle do jQueryUI que foi introduzido neste capítulo, com vistas à apresentação desta importante biblioteca de recursos, é possível fazer uso dos recursos do HTML 5 para campos do tipo `Data`. Para isso, coloque o atributo `[DataType(DataType.Date)]` antes da propriedade `DataCadastro`, comente o script que habilita o

`DateTimePicker`, e execute novamente sua aplicação. A figura a seguir apresenta este novo visual. E já vem internacionalizado.

The screenshot shows a web form titled "Registro de um NOVO PRODUTO". It includes fields for "Nome" (Name), "Data de Cadastro" (Registration Date), "Categoria" (Category), and "Fabricante" (Manufacturer). Below the "Nome" field is a red arrow pointing to a validation summary message: "Verifique os seguintes erros". The "Data de Cadastro" field contains a date picker with the value "02/03/2016". A red arrow points from the validation message to the date picker's calendar overlay, which displays the month "março de 2016" and a grid of dates from 28 to 26. A large orange button at the bottom right of the form says "A listagem de PRODUTOS".

Figura 6.8: DateTimePicker do HTML 5

Você pode ter percebido que a mensagem de erros do `ValidationSummary` está aparecendo o tempo todo na página. Vamos corrigir isso. Substitua a implementação pelo código da sequência. Ele verifica se existem erros no modelo e, apenas em caso verdadeiro, renderiza o sumário de erros, com um estilo diferenciado.

```
@if(ViewData.ModelState.Keys.Any(k => ViewData.ModelState[k].Errors.Count() > 0))
{
    <div class="alert alert-danger alert-dismissible">
        @Html.ValidationSummary(false, "Verifique os seguintes
        erros:")
    </div>
}
```

6.6 VALIDAÇÃO NO LADO CLIENTE

Segundo a documentação da Microsoft, a validação no lado

cliente (navegador) é para funcionar automaticamente quando a aplicação é criada pelo Visual Studio 2015, pois ele configura automaticamente o `Web.config`. Entretanto, caso não esteja funcionando em sua aplicação, verifique no `Web.Config` se a implementação a seguir está idêntica em seu arquivo. Na realidade, o que importa são as duas últimas instruções do `<appSettings>`.

```
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

Após esta verificação, é preciso também verificar se os arquivos JavaScript referentes à validação foram importados. Veja no código seguinte quais arquivos são estes. Você pode optar por importá-los no layout, ou nas visões `Create` e `Edit`.

```
<script src="@Url.Content("~/Scripts/jquery.validate.js")">
</script>
<script src="@Url.Content("~/Scripts/jquery.validate.
unobtrusive.js")"></script>
```

RECOMENDAÇÕES DE LEITURA

Algumas leituras adicionais são sempre bem-vindas. Desta maneira, seguem algumas recomendações.

A leitura do artigo *Code First Migrations With Entity Framework*, em

<http://www.codeproject.com/Articles/801545/Code-First-Migrations-With-Entity-Framework>, pode ser interessante para expandir o conhecimento sobre o Code First Migrations.

Embora o artigo *The Complete Guide to Validation in ASP.NET MVC 3 - Part 1*

(<http://www.devrends.co.uk/blog/the-complete-guide-to-validation-in-asp.net-mvc-3-part-1>) traga o conteúdo para o ASP.NET MVC 3 e este livro seja para o ASP.NET MVC 5, a leitura é importante, pois você poderá ter uma visão ampla sobre as possibilidades de validações.

6.7 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, essa atualização é refletida na base de dados. Ela precisa acontecer sem que se percam os dados que existem na base, e isso foi visto neste capítulo, por meio do Code First Migrations.

Ainda em relação ao modelo de negócio, existem certas propriedades que precisam ser validadas, e isso foi possível neste capítulo por meio de atributos do Data Annotations. Finalizando o capítulo, vimos como fazer uso do jQuery UI para usarmos um controle do tipo `DateTimePicker`.

Foi um curto capítulo, mas interessante. No próximo, trabalharemos o controle de usuários e seus acessos na aplicação. Será um capítulo longo, mas muito significativo.

CAPÍTULO 7

AREAS, AUTENTICAÇÃO E AUTORIZAÇÃO

A organização da estrutura de um projeto é algo muito importante. Muitas vezes, pode acontecer que classes e controladores tenham o mesmo nome, mas pertençam a um contexto diferente da aplicação. O ASP.NET MVC permite esta organização física e lógica por meio de *Areas*, e isso será visto 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 autorização, que também serão apresentados aqui.

7.1 AREAS

Em uma aplicação grande, podem existir 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 na prática a seguir.

Para criar uma *Area* em seu projeto, na *Solution Explorer*, clique com o botão direito sobre o nome dele, e então em *Add->Area*. Vamos nomear as *Areas* seguindo o padrão com que já estamos trabalhando. Desta maneira, seus nomes deverão ser: *Tabelas* e *Cadastros*. Após a criação das duas áreas, sua *Solution Explorer* deve estar semelhante ao apresentado na figura a seguir.

Observe que uma classe é criada para cada *Area*. Esta classe é responsável por registrar as rotas para as *Areas*. Abra os arquivos e verifique a semelhança com a classe *RouteConfig*.

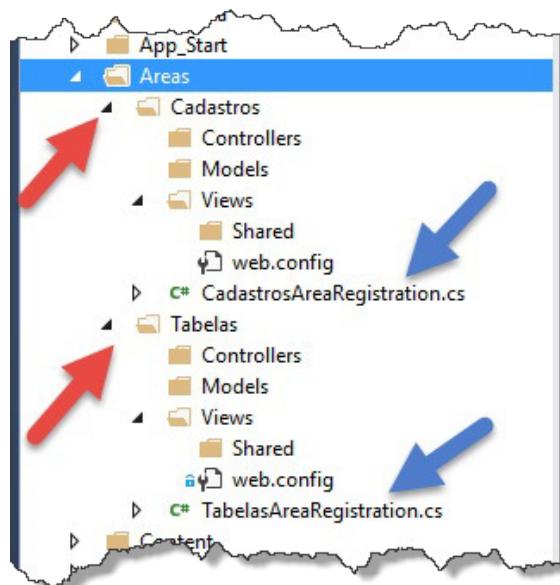


Figura 7.1: Solution Explorer com áreas criadas

Precisamos agora levar os controladores já existentes para a pasta `Controllers` de cada Área. Para a pasta `Controllers` de `Cadastros`, copie os controladores `FabricantesController` e `ProdutosController`. Para a pasta `Controllers` de `Tabelas`, copie o controlador `CategoriasController`.

Com esta mudança física realizada, precisamos acertar a organização lógica destas classes, e isso torna-se possível por meio dos namespaces. Em minha máquina, para `Fabricantes` e `Produtos`, o namespace ficou `Projeto01.Areas.Cadastros.Controllers` e para `Categorias`, ficou `Projeto01.Areas.Tabelas.Controllers`. Abra seus controladores e acerte estas linhas de código. Elas estão logo no início, antes da declaração da classe.

Além dos controladores, precisamos transferir as visões para as Áreas. Copie todos os arquivos de visão existentes na pasta `Produtos` que existe na `Views` da raiz do projeto, para a pasta `Views`, dentro de `Produtos` da Área `Cadastros`. Altere os nomes dos `Models` nas visões, faça o mesmo para `Fabricantes` e `Categorias`, lembrando que `Categorias` pertence à Área `Tabelas`.

Para finalizar as alterações, precisamos alterar as visões que usam `Partial View`. Por exemplo, a visão `Create` deverá estar igual a
`@Html.Partial("~/Areas/Cadastros/Views/Produtos/_PartialEditContentPanel.cshtml", Model)`. Note a inserção da Área na URL. Esta alteração se faz necessária também nas visões `Details`, `Delete` e `Edit`.

Com todas as alterações para que as Áreas funcionem, nos

resta testar nossa aplicação. Execute-a e invoque a visão Create de Produtos . Em minha máquina, a URL é <http://localhost:50827/Cadastros/Produtos/Create>.

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. Desta maneira, apresentarei aqui os conceitos necessários para a implementação desta 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, já 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 o usuário **já autenticado** possui direitos ao recurso requisitado, como por exemplo, registrar um novo produto.

ASP.NET Identity e OWIN

O OWIN (*Open Web Interface for .NET*) é uma especificação de código aberto que descreve uma camada de abstração entre os servidores web e componentes de aplicação. Para implementar a segurança em uma aplicação ASP.NET MVC, é oferecido o framework ASP.NET Identity. O ASP.NET Identity pode ser visto como um componente distribuído como um middleware do OWIN.

O ASP.NET Identity precisa de uma base de dados para criação de suas tabelas, tais como papéis para usuários (*roles*) e usuários. Esta base de dados pode ser uma exclusiva para o Identity ou pode ser a mesma que é utilizada pelo projeto da aplicação. Em nosso caso, para simplificar, usaremos a base de dados que já estamos utilizando em nossa aplicação e que está configurada para usar o EF Migrations.

Como nós criamos nosso projeto usando o template `Empty`, nada referente ao Identity foi preparado para ele. Desta maneira, precisamos adicionar alguns pacotes Nuggets ao projeto. Usaremos o *Package Manage Explorer*, o mesmo que foi utilizado para habilitar o Migrations. Garanta que o projeto padrão seja a aplicação Web, e digite as três instruções na sequência.

```
Install-Package Microsoft.AspNet.Identity.EntityFramework  
Install-Package Microsoft.AspNet.Identity.OWIN  
Install-Package Microsoft.Owin.Host.SystemWeb
```

Após a instalação dos pacotes anteriormente informados, é preciso inserir duas novas configurações. Dentro de `<appSettings>` em seu `Web.Config`, a primeira configuração é:

```
<add key="owin:AppStartup"  
value="Projeto01.IdentityConfig" />. Seu atributo value  
faz referência a uma classe ainda não criada. Vamos criá-la na  
pasta App_Start. Ela será invocada pelo OWIN no momento da  
inicialização da aplicação. Trabalharemos em sua implementação  
mais adiante.
```

A segunda configuração é a `Connection String` para a base de dados do ASP.NET Identity. Veja a seguir como está a minha. Optei por criar a base de dados de maneira independente no projeto da aplicação para simplificar as atividades.

```
<add name="IdentityDb" providerName="System.Data.SqlClient" connectionstring="Data Source=(localdb)\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\IdentityDB.mdf;Integrated Security=True;Connect Timeout=15;Encrypt=False;TrustServerCertificate=False;MultipleActiveResultSets=True"/>
```

Criando a classe Usuario

Com o ASP.NET Identity instalado e as configurações iniciais realizadas, precisamos pensar no modelo de negócio para a parte de segurança da aplicação. A primeira classe a ser criada é a que representa o usuário. O ASP.NET Identity oferece uma classe chamada `IdentityUser`, com diversas propriedades, que serão apresentadas conforme a necessidade de uso.

Em nossa aplicação, criaremos uma classe que estende `IdentityUser`. A listagem a seguir apresenta o código para ela, que se resume a extensão da `Microsoft.AspNet.Identity.EntityFramework.IdentityUser`. Inclusive, esta classe deve ser criada na pasta `Models`, dentro da `Area Segurança`, que você deverá criar agora. Veja na listagem o namespace para ela.

```
using Microsoft.AspNet.Identity.EntityFramework;

namespace Projeto01.Areas.Seguranca.Models
{
    public class Usuario : IdentityUser
    {
    }
}
```

Criando o contexto EF para o ASP.NET Identity

Como você deve ter notado, apenas para simplicidade, não estamos separando a implementação da segurança em um projeto

isolado. Desta maneira, para acessar a base de dados que será responsável pelo ASP.NET Identity, precisamos de um contexto Entity Framework específico para o Identity.

Sendo assim, crie no projeto uma pasta chamada `DAL`, e nela crie um arquivo chamado `IdentityDbContextAplicacao`, conforme código a seguir. Observe que o código possui a definição da classes `IdentityDbContextAplicacao`, que é o contexto EF específico para o Identity. O método estático `Create()` é o meio pelo qual instâncias da classe serão fornecidas, quando necessárias, pelo OWIN, fazendo uso da classe declarada como inicializadora do ASP.NET Identity no `Web.Config` (`IdentityConfig`).

```
using Microsoft.AspNet.Identity.EntityFramework;
using Projeto01.Areas.Seguranca.Models;
using System.Data.Entity;

namespace Projeto01.DAL
{
    public class IdentityDbContextAplicacao :
        IdentityDbContext<Usuario>
    {
        public IdentityDbContextAplicacao() : base("IdentityDb")
    }

    static IdentityDbContextAplicacao()
    {
        Database.SetInitializer<IdentityDbContextAplicacao>
            (new IdentityDbInit());
    }

    public static IdentityDbContextAplicacao Create()
    {
        return new IdentityDbContextAplicacao();
    }
}

public class IdentityDbInit : DropCreateDatabaseIfModelChanges<IdentityDbContextAplicacao>
```

```
{  
}  
}
```

Criando a classe para gerenciar usuários

Com o contexto criado e configurado, e com a classe de negócio que representará cada usuário registrado na aplicação também implementada, precisamos agora criar a classe que permitirá a gerência destes usuários. Para esta finalidade, o ASP.NET Identity dispõe da classe `UserManager`, que já oferece diversos métodos que serão explicados conforme formos utilizando.

Na listagem a seguir está a classe. Para registrá-la, na raiz de seu projeto, crie uma pasta chamada `Infraestrutura` e, em seguida, crie-a nesta pasta. O método estático `Create()` será chamado quando o ASP.NET Identity precisar de uma instância de `GerenciadorUsuario`, o que ocorrerá quando forem realizadas operações em dados de usuário. Assim que terminarmos a configuração, isso será trabalhado. A classe `UserStore` possui as implementações específicas para os métodos definidas em `UserManager`.

```
using Microsoft.AspNet.Identity;  
using Microsoft.AspNet.Identity.EntityFramework;  
using Microsoft.AspNet.Identity.Owin;  
using Microsoft.Owin;  
using Projeto01.DAL;  
  
namespace Projeto01.Infraestrutura  
{  
    public class GerenciadorUsuario : UserManager<Usuario>  
    {  
        public GerenciadorUsuario(IUserStore<Usuario> store) :  
            base(store)  
        {}  
    }  
}
```

```
public static GerenciadorUsuario Create(
    IdentityFactoryOptions<GerenciadorUsuario> options,
    IOwinContext context)
{
    IdentityDbContextAplicacao db = context.Get
        <IdentityDbContextAplicacao>();
    GerenciadorUsuario manager = new GerenciadorUsuario(
        new UserStore<Usuario>(db));
    return manager;
}
}
```

Implementando a classe inicializadora: IdentityConfig

O OWIN surgiu independentemente do ASP.NET e tem suas próprias convenções. Uma delas é a existência de uma classe que é instanciada para carregar e configurar o middleware, e executar qualquer outro trabalho de configuração que é necessário. Por padrão, essa classe é chamada `Start`, e é definida no namespace global.

Ela contém um método chamado `Configuration`, que é chamado pela infraestrutura do OWIN e passa uma implementação da interface `Owin.IAppBuilder`, suportando a configuração do middleware que uma aplicação requer. A classe de início é geralmente definida como uma classe parcial, com os seus outros arquivos de classe dedicados a cada tipo de middleware que está sendo usado.

Como optei por registrar a classe de inicialização no `Web.Config`, e a inseri na pasta `App_Start`, a implementação comentada no parágrafo anterior é dispensada. Na sequência, veja o código para a classe `IdentityConfig`.

A interface `IAppBuilder` é suportada por um certo número de métodos de extensão definidos em classes no namespace `Owin`. O método `CreatePerOwinContext()` cria uma nova instância de `GerenciadorUsuario` e `IdentityDbContextAplicacao` para cada solicitação. Isso garante que cada pedido tenha acesso aos dados do ASP.NET Identity, sem precisar se preocupar com sincronização de dados ou cache.

O método `UseCookieAuthentication()` diz ao ASP.NET Identity como usar um cookie para identificar usuários autenticados, onde as opções são especificadas por meio da classe `CookieAuthenticationOptions`. A parte importante aqui é a propriedade `LoginPath`, que especifica uma URL que os clientes devem ser redirecionados quando eles solicitam conteúdo sem estarem autenticados. O controlador `Account` e action `Login` serão ainda implementados.

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
using Projeto01.DAL;
using Projeto01.Infraestrutura;

namespace Projeto01
{
    public class IdentityConfig
    {
        public void Configuration(IAppBuilder app)
        {
            app.CreatePerOwinContext<IdentityDbContextAplicacao>(
                IdentityDbContextAplicacao.Create);
            app.CreatePerOwinContext<GerenciadorUsuario>(
                GerenciadorUsuario.Create);
            app.UseCookieAuthentication(new
                CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.
```

```

        ApplicationCookie,
        LoginPath = new PathString(
            "/Seguranca/Account/Login"),
        });
    }
}
}

```

7.3 LISTANDO OS USUÁRIOS REGISTRADOS

Mantendo o padrão adotado nos exemplos anteriores, começaremos o CRUD de usuários pelas action e visão relativas à recuperação e exibição de todos os usuários registrados, mesmo não tendo ainda nenhum usuário armazenado. A listagem a seguir traz o controlador que será responsável por administrar usuários. Dê a ele o nome `AdminController`, e crie-o na pasta `Controllers` da Area `Seguranca`.

Como diversas actions farão uso do `GerenciadorUsuario`, foi optado por criar uma propriedade privada apenas de leitura para obter este objeto, que é utilizado na action `Index`, e retorna todos os usuários registrados. O método `GetOwinContext()` é um método de extensão, adicionado ao `HttpContext`, por meio do assembly `Microsoft.Owin.Host.SystemWeb`.

```

using Microsoft.AspNet.Identity.Owin;
using Projeto01.Infraestrutura;
using System.Web;
using System.Web.Mvc;

namespace Projeto01.Areas.Seguranca.Controllers
{
    public class AdminController : Controller
    {
        public ActionResult Index()
        {
            return View(GerenciadorUsuario.Users);
        }
    }
}

```

```

        }
        private GerenciadorUsuario GerenciadorUsuario
        {
            get
            {
                return HttpContext.GetOwinContext().
                    GetUserManager<GerenciadorUsuario>();
            }
        }
    }
}

```

Vamos agora partir para a criação da visão `Index`. Clique com o botão direito do mouse sobre o nome do método `Index` e, na janela que se apresenta, opte pelo template `Empty`. Teremos como base o código da visão `Index` de `Produtos`. Na sequência, está o código para a visão.

```

@model IEnumerable<Projeto01.Areas.Seguranca.Models.Usuario>

 @{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Listagem de USUÁRIOS";
}

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

<div class="panel panel-primary">
    <div class="panel-heading">
        Relação de USUÁRIOS registrados
    </div>
    <div class="panel-body">
        <table class="table table-striped table-hover">
            <thead>
                <tr>
                    <th>
                        @Html.DisplayNameFor(model => model.Id)

```

```

        </th>
        <th>
            @Html.DisplayNameFor(model =>
                model.UserName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Email
)
        </th>
    </tr>
</thead>
<tbody>
@if (Model.Count() == 0)
{
    <tr><td colspan="3" class="text-center">
        Sem usuários registrados</td></tr>
}
else
{
    foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem =>
                    item.Id)
            </td>
            <td>
                @Html.DisplayFor(modelItem =>
                    item.UserName)
            </td>
            <td>
                @Html.DisplayFor(modelItem =>
                    item.Email)
            </td>
            <td>
                @Html.ActionLink("Alterar",
                    "Edit", new { id = item.Id })
            | @Html.ActionLink("Detalhes",
                    "Details", new { id = item.Id
})
        }) |
            @Html.ActionLink("Eliminar",
                    "Delete", new { id = item.Id
})
    }
}
</td>

```

```

        </tr>
    }
}
</tbody>
</table>
</div>
<div class="panel-footer panel-info">
    @Html.ActionLink("Registrar um novo USUÁRIO", "Create",
        null, new { @class = "btn btn-success" })
</div>
</div>

@section styles{
    <link href="@Url.Content("~/content/DataTables/css/dataTables
        bootstrap.css")" rel="stylesheet">
}

@section ScriptPage {
    <script src="@Url.Content("~/scripts/DataTables/jquery.
        dataTables.js")"></script>
    <script src="@Url.Content("~/scripts/DataTables/dataTables.
        bootstrap.js")"></script>
    <script type="text/javascript">
        $(document).ready(function () {
            $('li').removeClass("active");
            $('#liUsuarios').addClass("active");
            $('.table').dataTable({
                "order": [[1, "asc"]]
            });
        });
    </script>
}

```

Após implementar o código, execute sua aplicação e requisite a visão `Index`. Veja o resultado no corte da visão renderizada na figura a seguir. Observe no destaque que agora é informado quando não se possui registros relacionados à requisição. Em minha máquina, a URL para esta visão é: <http://localhost:50827/Seguranca/Admin>. Note que o nome da

Área faz parte da URL.

The screenshot shows a web application interface titled "Relação de USUÁRIOS registrados". It has three columns: "Id", "UserName", and "Email". Below the table, a green button says "Registrar um novo USUÁRIO". A red arrow points to the text "Sem usuários registrados" (No users registered).

Figura 7.2: Visualização de usuário registrados

Vamos verificar a base de dados criada? Pare a execução de sua aplicação, vá ao Solution Explorer, e clique no botão responsável por exibir todos os arquivos existentes para que possamos ver a base criada pela aplicação. Veja a figura:

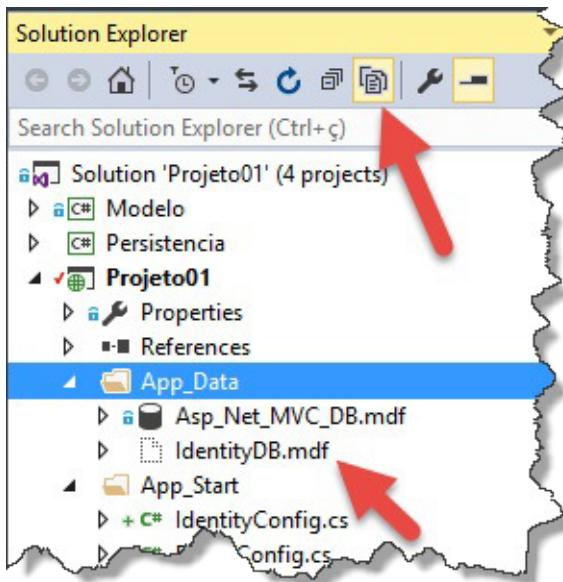


Figura 7.3: Visualizando o arquivo da base de dados criado

Clique com o botão direito do mouse sobre o nome do arquivo

(`IdentityDb`), e depois em `Include in Project`. Dê um duplo clique no nome do arquivo para ser direcionado ao `Server Explorer`. Verifique as tabelas criadas na base de dados. A estrutura pode ser comparada com a figura a seguir.

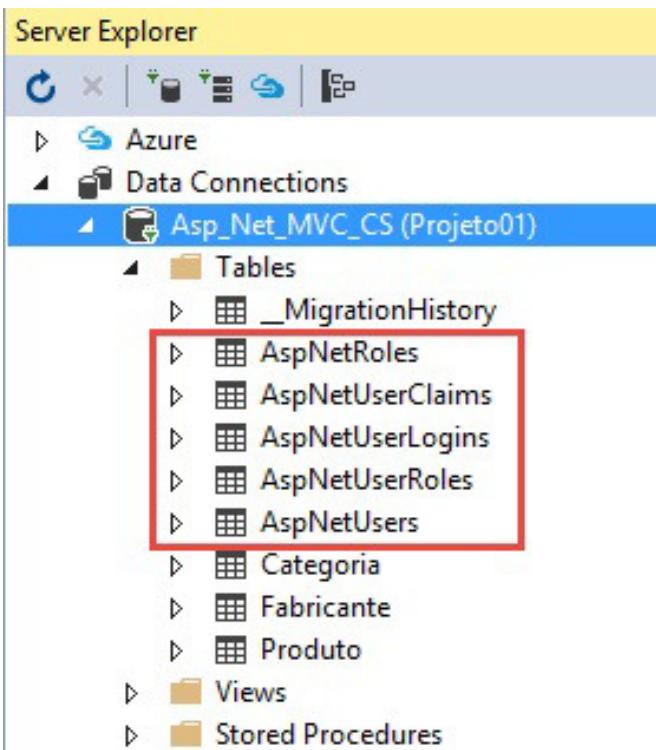


Figura 7.4: Visualizando as tabelas criadas

7.4 CRIANDO USUÁRIOS

Para a criação de usuários, faremos uso de uma classe com um novo papel, que ainda não vimos. Ela será armazenada na pasta `Models de Segurança`, mas não fará parte do modelo de negócio. Sua funcionalidade é servir de modelo exclusivamente

para a visão.

Normalmente, este tipo de classe é chamado de *modelo de visão*. Sendo assim, crie um arquivo de classe com o nome SeguracaViewModelos . Neste arquivo, teremos várias classes que servirão de modelo para a visão. Na sequência, observe o código da primeira classe, a UsuarioViewModelo , que representará a visão para registro de usuário.

```
using System.ComponentModel.DataAnnotations;

namespace Projeto01.Areas.Seguranca.Models
{
    public class UsuarioViewModel
    {
        public string Id { get; set; }
        [Required]
        public string Nome { get; set; }
        [Required]
        public string Email { get; set; }
        [Required]
        public string Senha { get; set; }
    }
}
```

Precisamos agora implementar as actions Create no AdminController . Lembre-se de que são duas: uma para o GET e outra para o POST . Também faz parte desta implementação um método específico (e privado) para o registro de possíveis erros ocorridos durante a tentativa de inserção de um usuário. Estes erros, quando ocorrerem, serão exibidos na visão. Veja na sequência o código inserido no controlador.

```
public ActionResult Create()
{
    return View();
}
```

Na sequência do trabalho, precisamos implementar a visão `Create`, fazendo uso da classe `UsuarioViewModel`. Essa implementação pode ser verificada na listagem a seguir. Ela segue o padrão já aplicado em visões anteriores. Uma diferença está no uso do helper `Html.PasswordFor()`, para a senha. O controle que será renderizado por meio do helper omite a exibição dos caracteres informados.

```
@model Projeto01.Areas.Seguranca.Models.UsuarioViewModel

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Registrando um NOVO USUÁRIO";
}

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Registro de um NOVO USUÁRIO
        </div>
        <div class="panel-body">
            <div class="form-horizontal">
                @if (ViewData.ModelState.Keys.Any(k => ViewData.
                    ModelState[k].Errors.Count() > 0))
                {
                    <div class="alert alert-danger
                        alert-dismissible">
                        @Html.ValidationSummary(false, "Verifique
                            os seguintes erros:")
                    </div>
                }
                <div class="form-group">
                    @Html.LabelFor(model => model.Nome,
                        htmlAttributes: new { @class =
                            "control-label col-md-2" })
                    <div class="col-md-10">
                        @Html.EditorFor(model => model.Nome,
                            new { htmlAttributes = new {
```

```

        @class = "form-control" } })
@Html.ValidationMessageFor(model => model

        Nome, "", new { @class =
"text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.Email,
    htmlAttributes: new { @class =
"control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Email,
        new { htmlAttributes = new {
            @class = "form-control" } })
    @Html.ValidationMessageFor(model => model

        Email, "", new { @class =
"text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.Senha,
    htmlAttributes: new { @class =
"control-label col-md-2" })
    <div class="col-md-10">
        @Html.PasswordFor(model => model.Senha,
        new { htmlAttributes = new {
            @class = "form-control" } })
    @Html.ValidationMessageFor(model => model

        Senha, "", new { @class =
"text-danger" })
    </div>
</div>
</div>
<div class="panel-footer panel-info">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Adicionar Usuário"
        class="btn btn-danger" />
    @Html.ActionLink("Retornar para a listagem de
    USUÁRIOS", "Index", null, new { @class =

```

```

        "btn btn-warning" })
    </div>
    <br />
    <br />
</div>
</div>
}

@section ScriptPage {
<script type="text/javascript">
$(document).ready(function () {
    $('li').removeClass("active");
    $('#liUsuarios').addClass("active");
});
</script>
}

```

Muito bem, em relação à criação de um usuário, até aqui temos a action que renderiza a visão (Create HTTP GET) e a própria visão, na qual serão informados os dados referentes ao usuário que se deseja cadastrar. Precisamos agora gerar a action que receberá estes dados, que enfim realizará a persistência deles na base de dados. Veja na sequência o código para esta action e, após ele, o método AddErrorsFromResult() que é invocado pela action.

```

[HttpPost]
public ActionResult Create(UsuarioViewModel model)
{
    if (ModelState.IsValid)
    {
        Usuario user = new Usuario { UserName = model.Nome,
            Email = model.Email };
        IdentityResult result = GerenciadorUsuario.Create(user,
            model.Senha);
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            AddErrorsFromResult(result);
        }
    }
}

```

```

    }
    return View(model);
}

private void AddErrorsFromResult(IdentityResult result)
{
    foreach (string error in result.Errors)
    {
        ModelState.AddModelError("", error);
    }
}

```

Após esta última implementação, nós precisamos testar a aplicação. Execute-a e requisite a action `Create`. Em minha máquina, a URL é <http://localhost:50827/Seguranca/Admin/Create>.

Tente registrar um novo usuário. Teste deixar os campos em branco para verificar as validações. Tente também gravar um novo usuário com o nome de um já existente. Ao conseguir registrar um novo, você será redirecionado para a visão `Index`, que exibirá o usuário cadastrado.

7.5 ALTERANDO USUÁRIOS JÁ CADASTRADOS

Para que um usuário seja alterado, precisamos recuperá-lo da base de dados e exibir seus dados para que sejam verificados e a alteração do que for necessário possa ocorrer. Na sequência, note o código responsável pela recuperação do usuário. Veja a diferença em relação ao que vínhamos implementando.

Fazemos uso da classe `GerenciadorUsuario`, que estende `Microsoft.AspNet.Identity.UserManager`. Com esta extensão, a classe passa a possuir um método próprio, `FindByID()`. Por

meio da invocação deste método, informando como argumento o `id`, recupera-se um usuário, que na realidade é um objeto `Microsoft.AspNet.Identity.EntityFramework.Identity`.

No código da sequência, para fins de exemplo, fazemos uso da classe `UsuarioModelView` como objeto de retorno para a visão. Veja o mapeamento.

```
public ActionResult Edit(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(
            HttpStatusCode.BadRequest);
    }
    Usuario usuario = GerenciadorUsuario.FindById(id);
    if (usuario == null)
    {
        return HttpNotFound();
    }
    var uvm = new UsuarioViewModel();
    uvm.Id = usuario.Id;
    uvm.Nome = usuario.UserName;
    uvm.Email = usuario.Email;
    return View(uvm);
}
```

Com o objeto recuperado, a sua visão como modelo precisa ser renderizada. Na sequência, é possível verificar esta implementação. Note as propriedades que exibem os dados. Por questão de segurança, a senha não é recuperada com o objeto e não aparece na visão, mas forneceremos mecanismos para que ela seja alterada. Verifique que todo o código da visão é semelhante ao apresentado anteriormente, na criação de um usuário.

```
@model Projeto01.Areas.Seguranca.Models.UsuarioViewModel
 @{
    Layout = "~/Views/Shared/_Layout.cshtml";
```

```

        ViewBag.Title = "Alterando dados de um NOVO USUÁRIO";
    }

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Alterando o USUÁRIO @Html.DisplayFor(model =>
                model.Id)
        </div>
        <div class="panel-body">
            <div class="form-horizontal">
                @if (ViewData.ModelState.Keys.Any(k => ViewData.
                    ModelState[k].Errors.Count() > 0))
                {
                    <div class="alert alert-danger alert-dismissible">
                        @Html.ValidationSummary(false, "Verifique
                            os seguintes erros:")
                    </div>
                }
                <div class="form-group">
                    @Html.LabelFor(model => model.Nome,
                        htmlAttributes: new { @class = "control-l
abel
                           col-md-2" })
                    <div class="col-md-10">
                        @Html.EditorFor(model => model.Nome, new
                            { htmlAttributes = new { @class =
                                "form-control" } })
                        @Html.ValidationMessageFor(model => model
                            Nome, "", new { @class = "text-danger
" })
                    </div>
                </div>

                <div class="form-group">
                    @Html.LabelFor(model => model.Email,
                        htmlAttributes: new { @class = "control-l
abel
                           col-md-2" })
                    <div class="col-md-10">

```

```

        @Html.EditorFor(model => model.Email, new
            { htmlAttributes = new { @class =
                "form-control" } })
        @Html.ValidationMessageFor(model => model

        Email, "", new { @class = "text-danger" }

    )
        </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.Senha,
        htmlAttributes: new { @class =
            "control-label col-md-2" })
    <div class="col-md-10">
        @Html.PasswordFor(model => model.Senha,
            new { htmlAttributes = new { @class =
                "form-control" } })
        @Html.ValidationMessageFor(model => model

        Senha, "", new { @class = "text-dange
r" })
    </div>
    </div>
</div>
</div>
<div class="panel-footer panel-info">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Alterar Usuário"
            class="btn btn-danger" />
        @Html.ActionLink("Retornar para a listagem de
            USUÁRIOS", "Index", null, new { @class =
                "btn btn-warning" })
    </div>
    <br />
    <br />
</div>
</div>
}

@section ScriptPage {
    <script type="text/javascript">
        $(document).ready(function () {
            $('li').removeClass("active");

```

```

        $( '#liUsuarios' ).addClass("active");
    });
</script>
}

```

Agora, com a visão que permitirá a informação dos dados que precisam ser alterados implementada, precisamos criar a action que recebe estes dados e atualiza-os na base de dados. Veja o código desta action na sequência.

Observe a recuperação do objeto e sua atualização com os dados recebidos pela action. Em seguida, é feita a invocação ao método `HashPassword()` da propriedade `PasswordHasher` da classe `GerenciadorUsuário`, que estende a `Microsoft.AspNet.Identity.UserManager`. Após a preparação da senha, é chamado o método `Update()`, também de `GerenciadorUsuário`.

```

[HttpPost]
public ActionResult Edit(UsuarioViewModel uvm)
{
    if (ModelState.IsValid)
    {
        Usuario usuario = GerenciadorUsuario.FindById(uvm.Id);
        usuario.UserName = uvm.Nome;
        usuario.Email = uvm.Email;
        usuario.PasswordHash = GerenciadorUsuario.PasswordHasher.
            HashPassword(uvm.Senha);
        IdentityResult result = GerenciadorUsuario.Update(usuario
    );
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            AddErrorsFromResult(result);
        }
    }
    return View(uvm);
}

```

```
}
```

7.6 REMOVENDO UM USUÁRIO EXISTENTE

Seguindo a metodologia que estamos adotando nos domínios já implementados, para a remoção dos clientes, mostraremos uma visão semelhante à `Details`, para que os dados do usuário possam ser apresentados, e então a remoção possa ser confirmada. Entretanto, para que a visão seja renderizada, precisamos de uma action que faça isso. A listagem da sequência é responsável por isso. A estratégia é a mesma adotada para a action `Edit`.

```
public ActionResult Delete(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(
            HttpStatusCode.BadRequest);
    }
    Usuario usuario = GerenciadorUsuario.FindById(id);
    if (usuario == null)
    {
        return HttpNotFound();
    }
    return View(usuario);
}
```

Agora é preciso implementar a visão que será renderizada pela action anterior. Ela está na sequência. Veja que este código, assim como nas visões anteriores de exclusão, existe a definição do `<div>` que representará a Modal Dialog.

```
@model Projeto01.Areas.Seguranca.Models.Usuario

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Visualizando detalhes de um USUÁRIO";
}
```

```

<div class="panel panel-primary">
    <div class="panel-heading">
        Visualizando detalhes do USUÁRIO a ser removido
    </div>
    <div class="panel-body">
        <div class="form-group">
            @Html.LabelFor(model => model.Id)<br />
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="glyphicon glyphicon-user"></i>
                </span>
            @Html.EditorFor(model => model.Id, new
                { htmlAttributes = new { @class =
                    "form-control", disabled = "disabled" } })
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.UserName)<br />
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="glyphicon glyphicon-user"></i>
                </span>
            @Html.EditorFor(model => model.UserName, new
                { htmlAttributes = new { @class =
                    "form-control", disabled = "disabled" } })
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.Email)<br />
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="glyphicon glyphicon-user"></i>
                </span>
            @Html.EditorFor(model => model.Email, new
                { htmlAttributes = new { @class =
                    "form-control", disabled = "disabled" } })
            </div>
        </div>
        <div class="panel-footer panel-info">
            <a href="#" class="btn btn-info" data-toggle="modal"
               data-target="#deleteConfirmationModal">Remover</a>
            @Html.ActionLink("Retornar para a listagem", "Index",
                null, new { @class = "btn btn-info" })
        </div>
    </div>

```

```

        </div>
    </div>

<div class="modal fade" id="deleteConfirmationModal" tabindex="-1"
     role="dialog" aria-hidden="true">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss=
                    "modal" aria-hidden="true">
                    &times;
                </button>
                <h4 class="modal-title">Confirmação de exclusão d
e
                    usuário</h4>
            </div>
            <div class="modal-body">
                <p>
                    Você está prestes a remover o usuário @Model.
                    Id.ToUpper() do cadastro.
                </p>
                <p>
                    <strong>
                        Você está certo que deseja prosseguir?
                    </strong>
                </p>
                @using (Html.BeginForm("Delete", "Admin",
                    FormMethod.Post, new { @id = "delete-form",
                    role = "form" }))
                {
                    @Html.HiddenFor(m => m.Id)
                    @Html.AntiForgeryToken()
                }
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-default"
                    onclick="#delete-form').submit();">
                    Sim, exclua este usuário.
                </button>
                <button type="button" class="btn btn-primary"
                    data-dismiss="modal">
                    Não, não exclua este usuário.
                </button>
            </div>
        </div>
    </div>

```

```

        </div>
    </div>
</div>
</div>

@section ScriptPage {
    <script type="text/javascript">
        $(document).ready(function () {
            $('li').removeClass("active");
            $('#liUsuarios').addClass("active");
        });
    </script>
}

```

Para finalizar a funcionalidade de remoção, precisamos implementar a action que realmente removerá o usuário da base de dados, que está sendo exibida a seguir. Note, no código, a chamada ao método `Delete()` da classe `GerenciadorUsuario`.

```

[HttpPost]
public ActionResult Delete(Usuario usuario)
{
    Usuario user = GerenciadorUsuario.FindById(usuario.Id);
    if (user != null)
    {
        IdentityResult result = GerenciadorUsuario.Delete(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            return new HttpStatusCodeResult(
                HttpStatusCode.BadRequest);
        }
    }
    else
    {
        return HttpNotFound();
    }
}

```

Precisamos finalizar o CRUD para usuários. Resta apenas

implementar a funcionalidade relativa ao `Details`. Mas isso fica para você fazer, como tarefa. É só adaptar um pouco o trabalho que fizemos para a remoção. :-)

7.7 ADAPTANDO O MENU DA APLICAÇÃO PARA AS FUNCIONALIDADES DE USUÁRIOS

Com o término da implementação das funcionalidades relacionadas a usuários, é importante agora inserir o acesso a elas no menu. Na visão do layout, a `_Layout.cshtml`, insira a opção de usuários, como mostra o código a seguir. Faça a inserção logo após o item de `Produtos`.

Veja, na chamada ao método `@Html.ActionLink()`, o penúltimo parâmetro, que especifica a área para o controlador. Veja também o último parâmetro, `null`, para atributos do HTML.

```
<li id="liUsuarios" role="presentation">@Html.ActionLink(  
    "Usuários", "Index", "Admin", new { area = "Seguranca" },  
    null)</li>
```

7.8 RESTRINGINDO O ACESSO A ACTIONS

Até este momento, neste capítulo, apresentei recursos relativos à manutenção de usuários no sistema, nada relacionado ao controle de acesso deles a recursos da aplicação. E é isso que vamos trabalhar agora. O primeiro passo é restringirmos o acesso às actions que renderizam visões, pois, em nosso caso, apenas usuários que estejam devidamente autenticados podem ter acesso a essas visões.

Esta é uma atividade simples. Na linha que antecede a declaração dos métodos das actions, adicione o atributo [Authorize]. Uma vez implementada esta linha em suas actions, execute sua aplicação e, por meio do menu, acesse a listagem de usuários. Se tudo der certo, você deve ter renderizado em seu navegador um resultado semelhante ao apresentado pela figura a seguir.



Figura 7.5: Erro ao tentar acessar action que exige autenticação

Observe nessa figura os destaques na URL e no recurso solicitado que não foi encontrado. Na URL, o endereço `http://localhost:50827/Seguranca/Account/Login?ReturnUrl=%2FSeguranca%2FAdmin` traz a chamada ao controlador Account e à action Login. Esta URL ainda é composta pela Querystring que contém o nome da Area e Controlador que foram requisitados sem sucesso.

Este erro ocorreu pelo fato de a action solicitada estar com acesso restrito a usuários que estejam autenticados. Já a chamada ao controlador Account e à action Login está registrada na classe `IdentityConfig`, que está na pasta `App_Start`.

7.9 IMPLEMENTANDO A AUTENTICAÇÃO

Já que agora bloqueamos o acesso do usuário às actions, precisamos permitir que ele acesse a aplicação por meio de uma visão de login, para que então possa acessar os recursos desejados. Nosso primeiro passo será a criação da classe `LoginViewModel`, dentro do arquivo `SegurancaViewModels` que está na pasta `Models`. O código para esta classe é apresentado na sequência. Veja que se refere apenas à declaração de duas propriedades.

```
using System.ComponentModel.DataAnnotations;

namespace Projeto01.Areas.Seguranca.Models
{
    public class LoginViewModel
    {
        [Required]
        public string Nome { get; set; }
        [Required]
        public string Senha { get; set; }
    }
}
```

Para fazermos uso da classe anteriormente criada, precisamos, antes da visão, das actions que a renderizarão e validarão o modelo desta visão. Para criá-las, na pasta `Controllers`, na Area `Seguranca`, crie o controlador chamado `AccountController` e o codifique de acordo com o código apresentado na listagem a seguir.

Veja na primeira action que o argumento que chega é uma string, de mesmo nome da variável da Querystring apontada anteriormente. Este parâmetro faz parte da segunda action, que recebe também um objeto `LoginViewModel`, com os dados referentes ao usuário e senha informados. Observe, no final da classe, a existência de duas propriedades de leitura. Estas foram criadas apenas para facilitar a invocação dos métodos usados por

elas.

Verifique ainda que, caso a autenticação tenha ocorrido bem, a variável `returnUrl` é testada para ver se não é nula. Se for, é preciso atribuir a ela o retorno para a visão `Home`, que chamará a action padrão, a `Index`. Não temos nada disso criado até aqui. Mas fica como tarefa para você. Siga os passos: crie um controlador `HomeController` na pasta `Controllers` na raiz do projeto, com uma action simples chamada `Index`. Finalize criando a visão `Index` para `Home`, informando que ela faz uso do layout que estamos utilizando.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Projeto01.Areas.Seguranca.Models;
using Projeto01.Infraestrutura;
using System.Security.Claims;
using System.Web;
using System.Web.Mvc;

namespace Projeto01.Areas.Seguranca.Controllers
{
    public class AccountController : Controller
    {
        public ActionResult Login(string returnUrl)
        {
            ViewBag.returnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Login(LoginViewModel details, string
returnUrl)
        {
            if (ModelState.IsValid)
            {
                Usuario user = UserManager.Find(details.Nome,
details.Senha);
```

```

        if (user == null)
        {
            ModelState.AddModelError("", "Nome ou senha
                inválido(s).");
        }
        else
        {
            ClaimsIdentity ident = UserManager.
                CreateIdentity(user,
                    DefaultAuthenticationTypes.
                    ApplicationCookie);
            AuthManager.SignOut();
            AuthManager.SignIn(new
                AuthenticationProperties
            {
                IsPersistent = false
            }, ident);
            if (returnUrl == null)
                returnUrl = "/Home";
            return Redirect(returnUrl);
        }
    }
    return View(details);
}

private IAuthenticationManager AuthManager
{
    get
    {
        return HttpContext.GetOwinContext().
            Authentication;
    }
}

private GerenciadorUsuario UserManager
{
    get
    {
        return HttpContext.GetOwinContext().GetUserManage
r
        <GerenciadorUsuario>();
    }
}
}
}

```

A action POST Login , apresentada no código anterior, verifica em um primeiro momento se o modelo de dados está cumprindo as validações existentes. Em caso positivo, uma busca por um usuário que tenha o nome e senha informados na visão será realizada. Caso esta existência seja comprovada, é preciso criar uma identidade para ele. Inicialmente, isso é feito pela chamada ao método CreateIdentity() .

Após esta criação, a aplicação realiza a saída do usuário atualmente conectado e, em seguida, realiza o acesso para o usuário atual, criando um cookie na máquina cliente. Este cookie será enviado em todas as requisições para comprovar que o usuário está autenticado.

Agora, após a criação das actions, precisamos criar a visão para que o usuário possa se autenticar. Para isso, crie a visão Login e deixe-a tal qual o código apresentado na sequência. Note que serão renderizados os controles que solicitarão o nome e senha do usuário que deseja acessar a aplicação e um botão para confirmar os dados solicitados.

```
@model Projeto01.Models.Login

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Acessando o sistema";
}

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Acessando o sistema
        </div>
```

```

<div class="panel-body">
    <div class="form-horizontal">
        @if (ViewData.ModelState.Keys.Any(k => ViewData.
            ModelState[k].Errors.Count() > 0))
        {
            <div class="alert alert-danger
                alert-dismissible">
                @Html.ValidationSummary(false, "Verifique
                    os seguintes erros:")
            </div>
        }
        <div class="form-group">
            @Html.LabelFor(model => model.Nome,
                htmlAttributes: new { @class =
                    "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Nome, new
                    { htmlAttributes = new { @class =
                        "form-control" } })
                @Html.ValidationMessageFor(model => model

                    Nome, "", new { @class =
                        "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Senha,
                htmlAttributes: new { @class =
                    "control-label col-md-2" })
            <div class="col-md-10">
                @Html.PasswordFor(model => model.Senha,
                    new { htmlAttributes = new { @class =
                        "form-control" } })
                @Html.ValidationMessageFor(model => model

                    Senha, "", new { @class = "text-dange
r" })
            </div>
        </div>
    </div>
    <div class="panel-footer panel-info">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Acessar"

```

```
        class="btn btn-danger" />
    </div>
    <br />
    <br />
</div>
</div>
}
```

Após a implementação da visão estar concluída, execute sua aplicação. Tente acessar a lista de usuários registrados. Você deve ser redirecionado para a visão de `Login`. Informe o nome e a senha de um usuário que você criou anteriormente, e confirme o acesso. Agora, com a autenticação concluída, você terá acesso à listagem de usuários.

7.10 UTILIZANDO PAPÉIS (ROLES) NA AUTORIZAÇÃO

O processo de autorização visto anteriormente é básico, pois basta que o usuário esteja autenticado para ter direito de acesso ao recurso controlado. O que precisamos agora é refinar esta autorização: limitar o acesso para um grupo de usuários. Essa limitação pode ser feita por meio de papéis (roles), que criamos e inserimos neles um grupo de usuários.

O primeiro passo é a criação da classe que representará um papel. Esta é uma extensão de `IdentityRole`, e vamos criá-la dentro da pasta `Models` da Área Segurança, de acordo com o código a seguir. Note que apenas são criados dois métodos como construtores, que invocam, por sua vez, os construtores da classe base.

```
using Microsoft.AspNet.Identity.EntityFramework;
```

```

namespace Projeto01.Areas.Seguranca.Models
{
    public class Papel : IdentityRole
    {
        public Papel() : base() { }
        public Papel(string name) : base(name) { }
    }
}

```

Da mesma maneira que, para usuário, temos a classe `GerenciadorUsuario`, precisamos de uma classe para gerenciar os papéis. Sendo assim, crie na mesma pasta (a Infraestrutura na raiz do projeto) a classe `GerenciadorPapel`, de acordo com o código seguinte. Veja a extensão para a classe `RoleManager`.

Note a semelhança funcional com a classe `GerenciadorUsuario`. São definidos dois métodos no código a seguir: um construtor que invoca o construtor da classe base, e um método estático que cria uma instância de `AppRoleManager`.

```

using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Projeto01.Areas.Seguranca.Models;
using Projeto01.DAL;
using System;

namespace Projeto01.Areas.Seguranca.Models
{
    public class GerenciadorPapel : RoleManager<Papel>,
        IDisposable
    {
        public GerenciadorPapel(RoleStore<Papel> store) :
            base(store)
        {
        }

        public static GerenciadorPapel Create(
            IdentityFactoryOptions<GerenciadorPapel> options,
            IOWinContext context)
        {

```

```

        return new GerenciadorPapel(new RoleStore<Papel>
            (context.Get<IdentityDbContextAplicacao>())));
    }
}
}

```

Outra vez, seguindo o exemplo da implementação para a classe `GerenciadorUsuario`, precisamos criar a instância `Owin` na classe `IdentityConfig`. Insira a linha a seguir abaixo da que registra a `GerenciadorUsuario`.

```
app.CreatePerOwinContext<GerenciadorPapel>(GerenciadorPapel.Create);
```

Implementando a listagem de papéis registrados

Iniciando os trabalhos com as visões a serem renderizadas para que haja interação com o usuário, começaremos criando a visão relacionada à listagem de papéis cadastrados, a `Index`. O primeiro passo é a criação do controlador (que ainda não existe) e a respectiva action. Crie este controlador, chamado de `PapelAdminController`, na pasta `Controllers` da Área Segurança. O código está na sequência. Veja que, para auxiliar, foi criada uma propriedade de leitura chamada `RoleManager`.

```

using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Projeto01.Areas.Seguranca.Models;
using System.ComponentModel.DataAnnotations;
using System.Web;
using System.Web.Mvc;

namespace Projeto01.Areas.Seguranca.Controllers
{
    public class PapelAdminController : Controller
    {
        // GET: Seguranca/RoleAdmin
        public ActionResult Index()
        {

```

```

        return View(RoleManager.Roles);
    }

    private GerenciadorPapel RoleManager
    {
        get
        {
            return HttpContext.GetOwinContext().
                GetUserManager<GerenciadorPapel>();
        }
    }
}

```

Esse código por si só retorna a coleção de papéis registrados na base para a visão. Entretanto, para efeitos didáticos, é interessante exibir quais usuários fazem parte de cada papel. Para isso, criaremos um método de extensão (Extension Method) como um novo método helper para a instrução Razor `@Html`.

Crie na pasta `Infraestrutura` uma classe chamada `IdentityHelpers` e atribua a ela o código da sequência. Note nesse código que o método busca uma instância da classe `GerenciadorUsuario`, para que então possa buscar pelo usuário, por meio do `id` que chega como parâmetro, e retornar o seu nome.

```

using Microsoft.AspNet.Identity.Owin;
using System.Web;
using System.Web.Mvc;

namespace Projeto01.Infraestrutura
{
    public static class IdentityHelpers
    {
        public static MvcHtmlString GetUserName(
            this HtmlHelper html, string id)
        {
            GerenciadorUsuario mgr = HttpContext.Current.
                GetOwinContext().GetUserManager<GerenciadorPapel>();
        }
    }
}

```

```

        <GerenciadorUsuario>());
        return new MvcHtmlString(mgr.FindByIdAsync(id).Result
            .UserName);
    }
}
}

```

Vamos agora criar a visão `Index`, que fará uso do método criado anteriormente. O código para esta visão está na listagem a seguir. Veja que, para a exibição dos papéis, existe um `foreach` no `Model` e, para que os usuários de cada papel sejam exibidos, é feito uso do LINQ.

```

@using Projeto01.Infraestrutura
@using Projeto01.Areas.Seguranca.Models
@model IEnumerable<Papel>

 @{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Listando os papéis registrados";
}

<div class="panel panel-primary">
    <div class="panel-heading">Papéis</div>
    <table class="table table-striped">
        <tr><th>ID</th><th>Nome</th><th>Usuários</th><th></th>
        </tr>
        @if (Model.Count() == 0)
        {
            <tr><td colspan="4" class="text-center">Sem papéis
                registrados</td></tr>
        }
        else
        {
            foreach (Papel role in Model)
            {
                <tr>
                    <td>@role.Id</td>
                    <td>@role.Name</td>
                    <td>
                        @if (role.Users == null || role.Users.

```

```

        Count == 0)
    {
        @: Sem usuários no papel
    }
else
{
    <p>
        @string.Join(", ", role.Users.
            Select(x => Html.
                GetUserName(x.UserId)))
    </p>
}
</td>
<td>
@using (Html.BeginForm("Delete",
    "PapelAdmin", new { id = role.Id }))
{
    @Html.ActionLink("Alterar", "Edit",
        new { id = role.Id }, new { @class =
            "btn btn-primary btn-xs" })
    <button class="btn btn-danger btn-xs"
        type="submit">
        Delete
    </button>
}
</td>
</tr>
}
</table>
</div>
@Html.ActionLink("Criar um novo papel", "Create", null, new
{ @class = "btn btn-primary" })

```

Agora, após toda implementação relacionada à visão Index , vamos testar e ver o resultado. Execute sua aplicação e requisite a visão Index . Em meu navegador, a URL é <http://localhost:50827/Seguranca/RoleAdmin/Index> .

É para aparecer a estrutura da tabela, sem nenhum papel. Fica como exercício você criar o menu para papéis e as funcionalidades

relacionadas a mostrá-lo como ativo, quando for o caso.

Inserindo novos papéis

A segunda etapa relacionada a papéis refere-se a inserção deles. Para isso, precisamos de duas action no controlador `PapelAdminController`. O primeiro, que se refere à action que gerará a visão para entrada de dados, tem seu código exibido na sequência. Veja que o comportamento do método `Create()` apenas retorna uma visão com mesmo nome do método.

```
public ActionResult Create()
{
    return View();
}
```

Para que a visão retornada pelo método anterior possa ser renderizada, precisamos implementá-la. Desta maneira, crie a visão `Create` e atribua a ela o código a seguir.

```
@model string

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "Registrando um NOVO PAPEL";
}

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
            Registro de um NOVO PAPEL
        </div>
        <div class="panel-body">
            <div class="form-horizontal">
                <div class="form-group">
                    <div class="col-md-10">
```

```

        <label>Nome</label>
        <input name="nome" value="@Model"
               class="form-control" />
    </div>
</div>
<div class="panel-footer panel-info">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Adicionar Papel"
               class="btn btn-danger" />
        @Html.ActionLink("Retornar para a listagem de
                         PAPÉIS", "Index", null, new { @class =
                         "btn btn-warning" })
    </div>
    <br />
    <br />
</div>
</div>
}

```

Para a criação da action que receberá o nome do papel a ser inserido, implementaremos no controlador um método privado e mais uma propriedade, tal qual segue no código a seguir.

```

private void AddErrorsFromResult(IdentityResult result)
{
    foreach (string error in result.Errors)
    {
        ModelState.AddModelError("", error);
    }
}

private GerenciadorUsuario UserManager
{
    get
    {
        return HttpContext.GetOwinContext().
            GetUserManager<GerenciadorUsuario>();
    }
}

```

Agora, com os pré-requisitos implementados, podemos

implementar a action `POST` que recebe o nome do papel a ser inserido. Veja o código na sequência. Veja, na assinatura do método, que o parâmetro `nome` é definido como obrigatório.

```
[HttpPost]
public ActionResult Create([Required]string nome)
{
    if (ModelState.IsValid)
    {
        IdentityResult result = RoleManager.Create(
            new Papel(nome));
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            AddErrorsFromResult(result);
        }
    }
    return View(nome);
}
```

Agora, vamos testar nossa aplicação. Para isso, execute-a e requisite a action `Create`. Na visão renderizada, informe o nome `Administradores` e clique no botão para confirmar a inclusão. Se tudo ocorrer bem, você será redirecionado para a visão que exibe os papéis registrados. A figura a seguir exibe o recorte das duas visões.

Figura 7.6: Visões Create e Index para papéis

7.11 GERENCIANDO MEMBROS DE PAPÉIS

Uma vez criado os papéis para gerenciamento de usuários, é preciso agora gerenciar os usuários dentro deles. Isso não é uma tarefa difícil. O processo se resume a recuperação de papéis e a invocação de métodos dos usuários para associação deles aos papéis.

Para desenvolver este trabalho, começaremos com a criação de modelos de visão na classe `SegurancaModelViews`. Insira neste arquivo as classes a seguir. A classe `PapelEditModel` é a responsável por hospedar, para um papel, quem é membro e quem não é. Essa classe será utilizada na action `GET Edit`.

```
public class PapelEditModel
{
    public Papel Papel { get; set; }
    public IEnumerable<Usuario> Membros { get; set; }
    public IEnumerable<Usuario> NaoMembros { get; set; }
}

public class PapelModificationModel
{
```

```

[Required]
public string NomePapel { get; set; }
public string[] IdsParaAdicionar { get; set; }
public string[] IdsParaRemover { get; set; }
}

```

Na sequência, precisamos criar a action no controlador `PapelAdminController`, que gerará a visão para a edição dos usuários em um papel. O código está na sequência. Observe que a action receberá o `id` do papel que se deseja alterar. Com este valor, logo no início, é recuperado o papel e, em seguida, todos os usuários que são membros dele.

O LINQ é usado para esta seleção, e o resultado é convertido para uma matriz pelo método `ToArray()`. Depois, uma coleção para membros e outra para não membros é populada, também fazendo uso de LINQ, tendo como base todos os usuários registrados. Ao final, a action retorna para a visão um objeto `PapelEditModel`.

```

public ActionResult Edit(string id)
{
    Papel papel = RoleManager.FindById(id);
    string[] memberIDs = papel.Users.Select(x => x.UserId).
        ToArray();
    IEnumerable<Usuario> membros = UserManager.Users.Where(x =>
        memberIDs.Any(y => y == x.Id));
    IEnumerable<Usuario> naoMembros = UserManager.Users.Except(
        membros);
    return View(new PapelEditModel
    {
        Role = papel,
        Membros = membros,
        NaoMembros = naoMembros
    });
}

```

Agora, precisamos implementar a visão que será responsável pela manutenção de usuários em papéis existentes. Note o nome

dos controles HTML, pois eles são os mesmos das propriedades da classe `PapelModificationModel`, que será o parâmetro recebido na action `POST Edit`. O código da visão pode ser visto na sequência.

Veja que existem duas tabelas: uma para os membros do papel, e outra para os usuários que não são membros. No início do código, existe um campo oculto, com o nome do papel que está em edição.

```
@using Projeto01.Areas.Seguranca.Models  
@model PapelEditModel  
  
{@  
    Layout = "~/Views/Shared/_Layout.cshtml";  
    ViewBag.Title = "Alteração de usuários em um PAPEL";  
}  
  
@Html.ValidationSummary()  
@using (Html.BeginForm())  
{  
    <input type="hidden" name="nomePapel" value="@Model.Papel.  
        Name" />  
    <div class="panel panel-primary">  
        <div class="panel-heading">Adicionar para @Model.Papel.  
            Name</div>  
        <table class="table table-striped">  
            @if (Model.NaoMembros.Count() == 0)  
            {  
                <tr><td colspan="2">Todos os usuários são membros  
                    </td></tr>  
            }  
            else  
            {  
                <tr><td>Usuários</td><td>Adicionar ao Papel  
                    </td></tr>  
                foreach (Usuario usuario in Model.NaoMembros)  
                {  
                    <tr>  
                        <td>@usuario.UserName</td>  
                        <td>
```

```

                <input type="checkbox" name=
                    "IdsParaAdicionar" value=
                    "@usuario.Id">
            </td>
        </tr>
    }
}
</table>
</div>

<div class="panel panel-primary">
    <div class="panel-heading">Remover de @Model.Papel.Name
    </div>
    <table class="table table-striped">
        @if (Model.Membros.Count() == 0)
        {
            <tr><td colspan="2">Sem usuários membros</td></tr>
        }
        else
        {
            <tr><td>Usuários</td><td>Remover do Papel</td>
            </tr>
            foreach (Usuario usuario in Model.Membros)
            {
                <tr>
                    <td>@usuario.UserName</td>
                    <td>
                        <input type="checkbox"
                            name="IdsParaRemover" value=
                            "@usuario.Id">
                    </td>
                </tr>
            }
        }
    </table>
</div>
<button type="submit" class="btn btn-primary">Gravar
    alterações</button>
@Html.ActionLink("Cancelar", "Index", null, new { @class =
    "btn btn-default" })
}

```

Agora, para finalizar, vamos à action POST Edit , que

receberá os dados da visão e atualizará o papel na base de dados. O código está na sequência. Note a existência de dois `foreach`: um para os membros que serão adicionados, e outro para retirar membros do papel.

```
[HttpPost]
public ActionResult Edit(PapelModificationModel model)
{
    IdentityResult result;
    if (ModelState.IsValid)
    {
        foreach (string userId in model.IdsParaAdicionar ??
            new string[] { })
        {
            result = UserManager.AddToRole(userId, model.
                NomePapel);
            if (!result.Succeeded)
            {
                return new HttpStatusCodeResult(HttpStatusCode.
                    BadRequest);
            }
        }
        foreach (string userId in model.IdsParaRemover ??
            new string[] { })
        {
            result = UserManager.RemoveFromRole(userId, model.
                NomePapel);
            if (!result.Succeeded)
            {
                return new HttpStatusCodeResult(HttpStatusCode.
                    BadRequest);
            }
        }
        return RedirectToAction("Index");
    }
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
}
```

A figura a seguir apresenta um recorte da visão que possibilita a alteração de um papel.

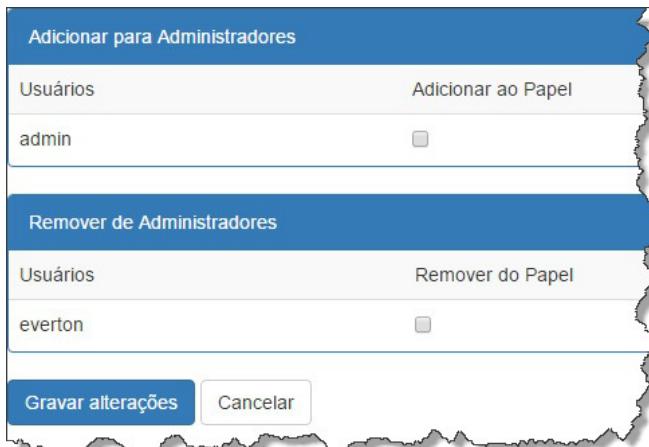


Figura 7.7: Visão para alteração de um papel

7.12 CRIANDO OS ACESSOS PARA LOGIN E LOGOUT

Vamos inserir no topo da página (na visão `_Layout.cshtml~`) o nome do usuário que está autenticado e a possibilidade de sair da aplicação, por meio de um `log out`. Também inseriremos a possibilidade de acessar a aplicação, caso não esteja autenticado (`log in`).

Como precisaremos do nome do usuário que está autenticado, será necessário implementar mais um método na classe `IdentityHelper`, que está na pasta `Infraestrutura`. Veja a listagem a seguir.

```
public static MvcHtmlString GetAuthenticatedUser(
    this HtmlHelper html)
{
    return new MvcHtmlString(HttpContext.Current.User.Identity.
        Name);
}
```

Com o método anterior implementado, precisamos alterar o nosso layout padrão. Para isso, insira a instrução `@using Projeto01.Infraestrutura` como sendo a primeira linha da visão `_Layout.cshtml`. Depois, na `<div>` do cabeçalho, altere-a para que fique semelhante à apresentada na listagem seguinte.

Observe que, caso não exista usuário autenticado, é oferecido ao usuário um link para acessar a aplicação. Caso contrário, o nome do usuário é exibido, com o link para sair.

```
<div class="page-header bg-danger">
    <div>
        <h1>Desenvolvimento em ASP.NET MVC <small class="text-danger">Utilizando o Bootstrap</small></h1>
    </div>
    <div style="text-align:right;">
        @{
            if (Html.GetAuthenticatedUser().ToString() != "") {
                @Html.GetAuthenticatedUser();
                @Html.ActionLink(" - Sair", "Logout", "Account");
            } else
            {
                @Html.ActionLink("Acessar", "Login", "Account",
                    new { area = "Seguranca" }, null);
            }
        }
    </div>
</div>
```

Para finalizar este processo, é preciso criar a action `Logout`, no controlador `Account`. Veja o código dela na listagem a seguir.

```
public ActionResult Logout()
{
    AuthManager.SignOut();
    return RedirectToAction("Index", "Home", new { area = "" });
}
```

7.13 ALTERANDO O PROCESSO DE

AUTORIZAÇÃO PARA UTILIZAR PAPÉIS

Precisamos agora alterar o processo de autorização, que se baseia apenas na autenticação do usuário. Fazer isso na action é também um processo simples, tal qual é para autorizar usuários que estejam apenas autenticados. Para termos a autorização para determinados papéis (em nosso caso, o papel Administradores), basta inserir a instrução [Authorize(Roles="Administradores")] antes da assinatura da action.

RECOMENDAÇÕES DE LEITURA

Algumas leituras adicionais são sempre bem-vindas. Desta maneira, seguem algumas recomendações.

Um pouco mais de teoria sobre Areas pode ser buscada nos artigos *Walkthrough: Organizando um ASP.NET aplicativo de MVC usando áreas* ([https://msdn.microsoft.com/pt-br/library/ee671793\(v=vs.100\).aspx](https://msdn.microsoft.com/pt-br/library/ee671793(v=vs.100).aspx)) e *Areas in ASP.NET MVC 4* (<http://www.codeproject.com/Articles/714356/Areas-in-ASP-NET-MVC>).

O artigo *Autenticação, Autorização e Accounting: Conceitos Fundamentais*, em <https://alexandremspmoraes.wordpress.com/2013/02/15/autenticacao-autorizacao-e-accounting-conceitos-fundamentais/>, apresenta um pouco mais de literatura acerca de autenticação e autorização.

Para um aprofundamento em OWIN, recomendo a leitura do

artigo ASP.NET: *Understanding OWIN, Katana, and the Middleware Pipeline*, em <http://johnatten.com/2015/01/04/asp-net-understanding-owin-katana-and-the-middleware-pipeline/#What-is-OWIN-and-Why-Do-I-Care->.

Já em relação específica ao ASP.NET Identity, três leituras complementares são recomendadas:
<http://www.codeproject.com/Articles/762428/ASP-NET-MVC-and-Identity-Understanding-the-Basics>,
<http://www.devmedia.com.br/trabalhando-com-o-projeto-owin/32848> e <http://www.asp.net/identity>.

Mais sobre Modelo de Visão (ou View Model) pode ser verificado no artigo <http://rachelappel.com/use-viewmodels-to-manage-data-amp-organize-code-in-asp-net-mvc-applications/>.

7.14 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Neste capítulo, foi possível conhecer uma maneira de organização lógica e física de seu projeto, por meio de Areas. Em relação ao controle de acesso, foram apresentados os conceitos de autenticação e autorização.

Implementações foram inseridas para a criação e manutenção de usuários e papéis. O controle de quais recursos estão disponíveis a um papel ou para usuários autenticados também foi exemplificado. O capítulo apresentou ainda recursos para

direcionar o usuário para a página de login e para a realização de logout. Foi um capítulo longo, mas não tinha como não ser. Foi muito conteúdo, mas também aprendizado!

CAPÍTULO 8

UPLOADS, DOWNLOADS E ERROS

Em aplicações desenvolvidas para a internet, é cada vez mais comum ver o envio de imagens, que podem ou não ser exibidas aos usuários. O envio de arquivos diferentes de imagem também é algo comum, com o processo de digitalização de documentos cada vez mais utilizado.

Este curto capítulo traz o envio de imagens e demais arquivos, assim como exibição de imagens e downloads dos arquivos enviados. Durante o desenvolvimento dos exemplos do livro, algumas vezes nos deparamos com páginas de erro. E elas traziam um visual ruim e fora do padrão da aplicação. Também veremos neste capítulo como trabalhar com a exibição de erros.

8.1 UPLOADS

Em nossa aplicação, no modelo `Produto`, seria interessante ligar a ele uma imagem para exibi-lo. Para que isso possa ocorrer, primeiro se faz necessário ter no modelo propriedades que possam representar esta imagem. A listagem a seguir traz as propriedades que precisam ser inseridas na classe `Produto`, a primeira que representa o nome do tipo do arquivo armazenado e a segunda

manterá a representação binária do arquivo enviado.

```
public string LogotipoMimeType { get; set; }
public byte[] Logotipo { get; set; }
```

Na sequência, precisamos adaptar a action que receberá a requisição. No exemplo, trabalharei a action `Edit`. Depois de pronto, você pode implementar a mesma lógica na action `Create`. Na sequência, veja a listagem da action com a nova implementação.

O parâmetro `logotipo` receberá o arquivo enviado pelo usuário, com toda a sua informação, como tipo e tamanho, por exemplo. Já o parâmetro `chkRemoverImagem`, quando for enviado, determinará que a imagem atualmente armazenada no produto deve ser removida. Note que a chamada ao método privado `GravarProduto()` também insere agora os novos dois argumentos, que são opcionais na action.

```
// POST: Produtos/Edit/5
[HttpPost]
public ActionResult Edit(Produto produto, HttpPostedFileBase logo
tipo = null, string chkRemoverImagem = null)
{
    return GravarProduto(produto, logotipo, chkRemoverImagem);
}
```

O método responsável por gravar o produto precisa ser adaptado, pois agora recebe os argumentos referentes a imagem a ser enviada ou removida. A implementação na sequência traz o método com estas adaptações. As alterações estão logo após a verificação da validade do modelo.

Se for para remover a imagem, `null` é atribuído à propriedade. Caso uma imagem seja enviada, as novas propriedades recebem seu tipo e a representação binária dela, que

é retornada por um novo método, o `SetLogotipo()`.

Para este exemplo, qualquer arquivo poderá ser enviado. Não validamos que apenas arquivos de imagens sejam enviados. Esta validação pode ser realizada com base no `ContentType`. Isso fica como tarefa. :-) O resto do método se manteve como já era.

```
private ActionResult GravarProduto(Produto produto, HttpPostedFileBase logotipo, string chkRemoverImagem)
{
    try
    {
        if (ModelState.IsValid)
        {
            if (chkRemoverImagem != null) {
                produto.Logotipo = null;
            }
            if (logotipo != null)
            {
                produto.LogotipoMimeType = logotipo.ContentType;
                produto.Logotipo = SetLogotipo(logotipo);
            }
            produtoServico.GravarProduto(produto);
            return RedirectToAction("Index");
        }
        PopularViewBag(produto);
        return View(produto);
    }
    catch
    {
        PopularViewBag(produto);
        return View(produto);
    }
}
```

Como visto na listagem anterior, há a chamada ao método `SetLogotipo()`. É ele que fará a leitura do arquivo recebido e transformará este arquivo em valores binários para serem armazenados na propriedade `e`, consequentemente, no campo que será criado na tabela. Veja na sequência a listagem deste novo

método.

Observe que um array de byte é criado, tendo como tamanho o tamanho da imagem recebida. Depois, uma leitura é realizada e seu resultado armazenado na variável bytesLogotipo , a qual é retornada pelo método.

```
private byte[] SetLogotipo(HttpPostedFileBase logotipo)
{
    var bytesLogotipo = new byte[logotipo.ContentLength];
    logotipo.InputStream.Read(bytesLogotipo, 0, logotipo.ContentLength);
    return bytesLogotipo;
}
```

Com a implementação do lado do servidor concluída, precisamos implementar agora o comportamento para o lado do cliente, as visões. A primeira implementação necessária é no HTML Helper BeginForm() , pois precisamos informar que um arquivo será enviado. Veja esta instrução na sequência. Para que esta informação fosse informada, precisamos informar qual a action que será requisitada, o controlador que a possui, o método do formulário e, por fim, o argumento new { enctype = "multipart/form-data" } .

```
@using (Html.BeginForm("Edit", "Produtos", FormMethod.Post, new {
    enctype = "multipart/form-data" }))
```

Agora, precisamos implementar a adaptação necessária para que o usuário possa enviar uma imagem, e que esta também possa ser exibida quando o produto que estiver sendo alterado possuí-la. Ao terminar, implemente também a visão Details . A implementação necessária para a visão está no código da listagem a seguir.

Logo no início, é definido o controle <input

`type="file"/>` . É ele que permitirá a seleção do arquivo que será enviado para a action. Na sequência, caso exista uma imagem (logotipo) no produto do modelo, ela é exibida, juntamente com um controle checkbox para o usuário selecionar se deseja remover a imagem atual do produto.

```
<div class="col-md-offset-7 col-md-5">
    <div class="form-group">
        <input type="file" name="logotipo" id="logotipo" onchange:='$("#upload-fileinfo").html($(this).val());'>
        <span id="upload-file-info"></span>
    </div>

    <div class="form-group">
        @if (Model.Logotipo == null)
        {
            <div class="form-control-static">Nem uma imagem adicionada</div>
        }
        else
        {
            <div class="panel-body">
                
            </div>
            <div class="panel-footer">
                <div class="checkbox">
                    <label>
                        <input type="checkbox" name="chkRemoverImagem" value="Sim">Remover imagem
                    </label>
                </div>
            </div>
        }
    </div>
</div>
```

Na listagem anterior, na parte em que a imagem é renderizada, está a chamada à action `GetLogotipo()` , que retornará a imagem para exibição na visão. Sua implementação pode ser verificada a

seguir.

```
public FileContentResult GetLogotipo(long id)
{
    Produto produto = produtoServico.ObterProdutoPorId(id);
    if (produto != null)
    {
        return File(produto.Logotipo, produto.LogotipoMimeType);
    }
    return null;
}
```

Com tudo implementado, nos resta testar a aplicação. Execute-a e, na listagem de produtos, escolha um para alterar. A figura a seguir apresenta um recorte da visão já com uma imagem inserida. Teste enviar uma imagem e depois removê-la.



Figura 8.1: Exibindo uma imagem enviada por Upload

8.2 DOWNLOAD

Para oferecermos ao usuário a possibilidade de fazer o download de um arquivo, independente de ele ser uma imagem ou

um PDF, por exemplo, precisaremos de mais duas propriedades na classe `Produto`, de nosso modelo. Na sequência, estão estas duas propriedades.

A primeira refere-se ao nome original do arquivo recebido e que será realizado o download; e a segunda ao tamanho do arquivo, pois precisaremos, em nosso exemplo, criá-lo em disco para poder ser realizado o download.

```
public string NomeArquivo { get; set; }
public long TamanhoArquivo { get; set; }
```

Agora, na sequência, precisamos adaptar nosso método no controlador para que o nome do arquivo e seu tamanho possam ser gravados junto com o objeto. A listagem seguinte traz as duas instruções que devem ser inseridas dentro do bloco que atribui a imagem ao objeto produto, no método `GravarProduto()`. A primeira instrução armazena o nome do arquivo recebido, e a segunda seu tamanho.

```
produto.NomeArquivo = logotipo.FileName;
produto.TamanhoArquivo = logotipo.ContentLength;
```

Vamos implementar a visão para exibir o arquivo de maneira que seja um link, para que o download seja possível. A instrução a seguir deverá ser inserida embaixo do `<div>` do `checkbox`.

```
@Html.ActionLink("Baixar o arquivo : " + Model.NomeArquivo, "DownloadArquivo", "Produtos", new { area = "Cadastros", id = Model.ProdutoId }, null)
```

Veja na instrução anterior que será chamada a action `DownloadArquivo` quando o usuário clicar no link que será renderizado. Na sequência, veja a implementação dessa action. Observe, na segunda instrução da action, que o arquivo será

gravado em uma pasta chamada `TempData` , que deverá existir na raiz do projeto. Esta ainda não existe, então crie-a.

```
public ActionResult DownloadArquivo(long id)
{
    Produto produto = produtoServico.ObterProdutoPorId(id);
    FileStream fileStream = new FileStream(Server.MapPath("~/Temp
Data/") + produto.NomeArquivo, FileMode.Create, FileAccess.Write)
    ;
    fileStream.Write(produto.Logotipo, 0, Convert.ToInt32(produto
.TamanhoArquivo));
    fileStream.Close();
    return File(fileStream.Name, produto.LogotipoMimeType, produ
t.NomeArquivo);
}
```

Com tudo implementado, nos resta testar a aplicação. Altere outro produto. Não pode ser o mesmo, pois a visão necessita de dados que não foram gravados anteriormente, pois isso geraria uma exceção. Insira uma imagem e depois solicite nova alteração. Veja que, abaixo do `checkbox` , aparecerá um link para a imagem. Clique nele e o download começará.

8.3 PÁGINAS DE ERRO

Para finalizarmos este capítulo, vou apresentar uma maneira para trabalhar com exibição de erros. Um erro padrão é o 404, gerado pelo `HttpNotFound()` . Há diversos templates para página deste erro. Na internet, há vários templates gratuitos, e um deles é o <https://www.freshdesignweb.com/free-404-error-page-template/>.

Existem diversas técnicas para captura e manipulação de erros, e eu adotarei uma delas, que é capturar os erros globalmente na aplicação. A primeira etapa é implementarmos, no arquivo `Global.asax` , a lógica para captura do erro disparado e definir

uma action para gerar a visão relativa ao erro. O código a ser inserido deve ser feito no método `Application_Error()`, como pode ser visto na listagem adiante.

Na implementação, é recuperado o erro que causou a captura do evento. É criado um objeto `RouteData` que vai compor a URL a ser requisitada após a determinação de qual tipo de erro ocorreu. Verifica-se qual erro HTTP que foi disparado para a atribuição do nome da action ao `RouteData`. Armazena-se na sessão a exceção capturada, para usá-la na camada de apresentação, e instancia-se o controlador (ainda não criado) que será responsável pelas actions dos erros. Por fim, cria-se o `RequestContext` para os dados da rota criado e executa-o.

```
protected void Application_Error()
{
    var exception = Server.GetLastError();
    var httpException = exception as HttpException;
    Response.Clear();
    Server.ClearError();
    var routeData = new RouteData();
    routeData.Values["controller"] = "Errors";
    routeData.Values["action"] = "General";
    routeData.Values["exception"] = exception;
    Response.StatusCode = 500;
    if (httpException != null)
    {
        Response.StatusCode = httpException.GetHttpCode();
        switch (Response.StatusCode)
        {
            case 400:
                routeData.Values["action"] = "Http400";
                break;
            case 403:
                routeData.Values["action"] = "Http403";
                break;
            case 404:
                routeData.Values["action"] = "Http404";
                break;
        }
    }
}
```

```

        }
    }
    Session["ErrorException"] = exception;
    IController errorsController = new ErrorsController();
    var rc = new RequestContext(new HttpContextWrapper(Context),
routeData);
    errorsController.Execute(rc);
}

```

Na listagem anterior, na antepenúltima instrução, verifica-se o controlador `ErrorsController`. Nós ainda não o temos, então vamos criá-lo na pasta `Controllers` da aplicação, junto com a `HomeController`. O código para este controlador pode ser verificado na listagem a seguir. Não há nada complicado neste controlador, apenas as actions, que retornam visões relativas ao erro HTTP.

```

using System;
using System.Web.Mvc;

namespace Projeto01.Controllers
{
    public class ErrorsController : Controller
    {
        public ActionResult General()
        {
            return View();
        }
        public ActionResult Http400()
        {
            return View();
        }
        public ActionResult Http404()
        {
            return View();
        }
    }
}

```

Para concluir, precisamos criar as três visões. A primeira, `General`, pode ser vista na listagem a seguir. Veja que não há

nenhuma formatação de estilo, e isso fica a seu critério. Note a recuperação da exceção armazenada na Session .

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>General</title>
</head>
<body>
    <div>
        <h1>500</h1>
        <h2>Houston, temos um problema.</h2>
        <p>
            Infelizmente um erro interno ocorreu. Por favor, tent
e mais tarde ou entre em contato com o suporte.
            <br /><br />
            @HttpContext.Current.Session["ErrorException"].ToStri
ng().Substring(0, 400)
        </p>
        <p>
            <a href='@Url.Action("Index", "Home")'>
                Retornar ao início
            </a>
            <br>
        </p>
    </div>
</body>
</html>
```

A visão Http400 é semelhante à anterior, e fica para você implementar como tarefa. A Http404 é exibida na sequência. Note que não há recuperação da exceção.

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Http404</title>
</head>
<body>
```

```
<div>
    <h1>404</h1>
    <h2>Houston, temos um problema.</h2>
    <p>
        Infelizmente, a página que você solicitou não existe.
    </p>
    <p>
        <a href='@Url.Action("Index", "Home")'>
            Retornar ao início
        </a>
        <br>
    </p>
</div>
</body>
</html>
```

Agora, para testar, requisite uma URL que não existe ou tente visualizar os dados de um produto com `id` inexistente. É para você obter uma página de erro, semelhante à apresentada na figura:

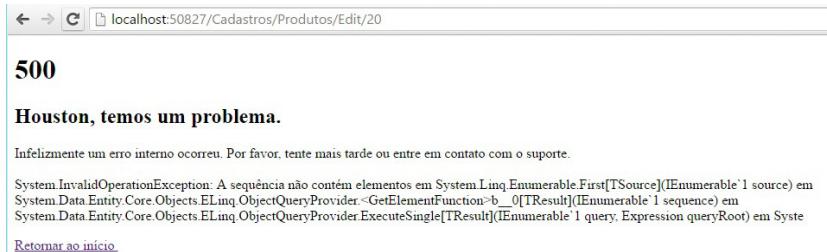


Figura 8.2: Página exibindo um erro identificado pela aplicação

RECOMENDAÇÕES DE LEITURA

Algumas leituras adicionais são sempre bem-vindas. Desta maneira, seguem algumas recomendações.

Um pouco mais sobre upload e download pode ser visto em <http://www.codeproject.com/Articles/1001348/Uploading-and-Downloading-in-MVC-Step-by-Step>.

O artigo *Exception handling in ASP.NET MVC (6 methods explained)*, em

<http://www.codeproject.com/Articles/850062/Exception-handling-in-ASP-NET-MVC-methods-explained>, traz um pouco mais sobre o tratamento de erros no ASP.NET MVC.

Mais sobre session pode ser verificado no artigo *ASP.NET – Como trabalhar com Session de forma elegante*, em <http://rafaelzaccanini.net.br/2014/11/11/asp-net-como-trabalhar-com-session-de-uma-forma-mais-elegante/>.

8.4 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Vimos neste capítulo o processo de upload e download de arquivos. Trabalhamos com uma imagem, para ela também poder ser renderizada, mas o aprendizado é para qualquer tipo de arquivo e finalizamos o capítulo apresentando uma técnica para tratamento de erros. No próximo capítulo, será apresentado o processo relacionado a um carrinho de compras.

CAPÍTULO 9

UM CARRINHO DE COMPRAS

Quando trabalhamos aplicações comerciais na web, muitas delas referem-se a comercialização de algum produto e/ou serviço, o que ficou conhecido como "Carrinho de compra". Existem algumas técnicas que podem ser aplicadas para isso.

Uma delas é a aplicação fornecer uma listagem de produtos, com possibilidade de pesquisa. E por meio desta listagem, adicionar o item desejado ao seu carrinho de compras e, ao final, visualizar seu carrinho e informar seus dados para cobrança e recebimento dos itens adquiridos.

A questão é como armazenar os itens desejados entre uma pesquisa de produtos e outra. Persistir na base de dados? Mas e se o cliente desistir da compra apenas fechando o navegador? Os dados ficarão persistidos e precisarão depois serem eliminados.

Uma boa técnica para isso é o armazenamento dos dados em uma variável de sessão, que pode ter seu tempo de vida definido pela aplicação. É justamente este o objetivo deste capítulo, implementar um carrinho de compra, com seu armazenamento em uma variável de sessão. De carona, veremos algo mais sobre o

jQuery, um controle que funcionará como autocomplemento e finalizaremos com a apresentação de conteúdo fazendo uso de AJAX. Promete ser um bom capítulo.

9.1 ADIÇÃO DO CARRINHO DE COMPRA AO MODELO DE NEGÓCIO

Para que possamos iniciar o trabalho com o carrinho de compra na metodologia que pretendo adotar, uma classe precisa ser adicionada ao nosso modelo de negócio, a `ItemCarrinho`. Esta classe conterá o item registrado como possível compra pelo usuário. Como estamos separando nosso modelo de negócio em pastas, vamos criar uma pasta chamada `Carrinho`, que conterá as classes de modelo para este contexto de negócio. Na sequência está o código para esta classe.

```
using Modelo.Cadastros;

namespace Modelo.Carrinho
{
    public class ItemCarrinho
    {
        public long? ItemCarrinhoId { get; set; }
        public Produto Produto { get; set; }
        public int Quantidade { get; set; }
        public double ValorUnitario { get; set; }
        public double SubTotal { get { return Quantidade * ValorU
nitario; } }
    }
}
```

Objetos da classe anterior serão adicionados a uma coleção, que será persistida em uma variável de sessão. Entretanto, já seguimos nela a convenção do EF, pois quando (e se) o cliente fechar o pedido, eles deverão ser armazenados em uma base de

dados, associados, é claro, a um objeto referente ao carrinho de compra, com dados do cliente e data da compra.

9.2 O CONTROLADOR PARA O CARRINHO DE COMPRA

Se fossemos armazenar imediatamente o carrinho em uma tabela, teríamos agora de implementar a camada de persistência e serviço para o carrinho. Mas, como faremos uso da sessão para armazenar as compras, inicialmente partiremos para o controlador, criando uma action para o método HTTP GET , que gerará a visão de seleção de produtos para compras.

Crie uma Area chamada Carrinho e nela o controlador CarrinhosController . Veja o código dessa action na listagem a seguir. Observe que o comportamento do método inicia recuperando uma variável chamada carrinho da sessão existente no contexto da requisição HTTP. Caso a variável seja nula, significa que está ocorrendo a primeira requisição do cliente para esta action/visão, o que leva, então, a criação da variável na sessão, antes de retorná-la.

```
public ActionResult Create()
{
    IEnumerable<ItemCarrinho> carrinho = HttpContext.Session["carrinho"] as IEnumerable<ItemCarrinho>;
    if (carrinho == null)
    {
        carrinho = new List<ItemCarrinho>();
        HttpContext.Session["carrinho"] = carrinho;
    }
    return View(carrinho);
}
```

Session é uma das possibilidades oferecidas pelo ASP.NET MVC para gerenciamento de estados. Uma aplicação visualiza uma sessão por cada cliente conectado a ela. E quando falo em cliente aqui, me refiro a navegadores.

Desta maneira, cada navegador poderá ter uma sessão diferente com a aplicação e ter seus dados de sessão compartilhados entre requisições. É importante não haver um exagero no uso de sessões. Uma discussão sobre gerenciamento de estados pode ser verificada em <http://www.codeproject.com/Articles/808839/State-Management-in-ASP-NET>.

9.3 A VISÃO PARA O CARRINHO DE COMPRA

Vamos agora criar a visão para esta action. Crie-a e veja se fica semelhante ao apresentado na sequência. Esta visão terá um único campo de interação com o usuário, que receberá o nome do produto a ser inserido no carrinho de compra.

Optei por solicitar a digitação do nome para que pudesse apresentar um novo controle do jQuery UI para você, o autocomplete . Desta maneira, dirija-se ao site do jQuery (como fizemos anteriormente com o DateTimePicker) e selecione o novo controle.

Lembre-se de selecionar também o DateTimePicker , pois o arquivo baixado terá o mesmo nome do que você realizou o download anteriormente. Na sequência, comentarei mais sobre

este controle.

Observe, antes do `Label` *Produto para pesquisar*, a existência de um campo oculto, o `idproduto`. É neste controle que armazenaremos o id para o nome selecionado na listagem que será gerada pelo autocomplete.

Na listagem, deixei dois comentários. O primeiro se refere ao controle que terá a funcionalidade de autocomplete, e o segundo a area onde os itens inseridos no carrinho serão exibidos, que você já pode notar o uso de `Partial View`. Na definição do controle `nomeproduto`, nas classes de estilo para o controle, existe a `autocomplete-with-update-field`, que é por meio dela que o `jQuery UI` identificará que este controle fará uso do autocomplete.

Existe também o atributo `data_updatefield` que configura o controle oculto `idproduto` como recebedor do valor quando uma seleção for feita. Já o atributo `data_url` define a URL (action) que será invocada quando valores forem digitados no controle de nome do produto. Logo apresento o código JavaScript para o uso deste controle.

```
@model IEnumerable<Modelo.Carrinho.ItemCarrinho>

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@using (Html.BeginForm("AddProduto", "Carrinhos", new { area = "Carrinho" }, FormMethod.Post, new { id = "formCarrinho" }))
{
    @Html.AntiForgeryToken()

    <div class="panel panel-primary">
        <div class="panel-heading">
```

```

        Seleção de itens para o carrinho de compra
    </div>
    <div class="panel-body">
        <div class="form-horizontal">
            <div class="col-md-12">
                <div class="form-group">
                    Digite o nome do produto e confirme a ins
                erção
                </div>
            </div>

            <div class="col-lg-12">
                <div class="form-group">
                    @Html.Hidden("idproduto")
                    @Html.Label("Produto para pesquisar ")
                    <!-- COMENTÁRIO 1
                        O ELEMENTO A SEGUIR SERÁ DE AUTOCOMP
                    LEMENTO, VIA AJAX, POR MEIO DE UM CONTROLE jQUERY -->
                    @Html.TextBox("nomedoproduto", "", new
                    {
                        @class = "form-control autocomplete-with-update-field",
                        data_updatefield = "idproduto",
                        data_url = Url.Action("GetProdut
osPorNome", "Produtos", new { Area = "Cadastros" })
                    })
                </div>
            </div>
            <div class="col-md-12">
                <div class="form-group" style="text-align:center;">
                    >
                        <input id="SubmitForm" type="submit" value="A
dicionar produto ao carrinho" class="btn btn-success" />
                    </div>
                </div>
            </div>
        </div>

        <div class="panel panel-primary">
            <div class="panel-body">
                <div class="col-md-12">
                    <!-- COMENTÁRIO 2
                        O DIV A SEGUIR TERÁ SEU CONTEÚDO ATUALIZADO VIA

```

```

AJAX A CADA INSERÇÃO DE ITEM NO CARRINHO -->
    <div id="itenscarrinho">
        @Html.Partial("~/Areas/Carrinho.Views/Carrinhos/_ItensCarrinho.cshtml", Model)
    </div>
</div>
</div>
}

```

Verifique, no código anterior, no **COMENTÁRIO 2**, que dei um nome ao `<div>`. Isso porque, por meio desta referência, a **Partial View** será renderizada a cada produto inserido no carrinho.

Na primeira requisição à action/visão, nada aparecerá na listagem. Porém, tente inserir um produto, ir para outra visão e retornar para ela. Você verá que os dados serão exibidos, pois pertenciam à sessão e ela só finaliza-se quando o navegador é fechado, ou quando o seu término é forçado, por meio de código.

Veja, na sequência, o código desta **Partial View** (arquivo `_ItensCarrinho.cshtml`). Lembre-se de que você pode inovar sempre as visões.

```

@model IEnumerable<Modelo.Carrinho.ItemCarrinho>

<table class="table table-striped table-hover">
    <thead>
        <tr>
            <th>
                Produto
            </th>
            <th>
                Quantidade
            </th>
            <th>Valor</th>
            <th>SubTotal</th>
        </tr>
    
```

```

</thead>
<tbody>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Produto.Nome)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Quantidade)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ValorUnitario)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.SubTotal)
            </td>
        </tr>
    }
</tbody>
</table>
Total Carrinho:
@{ var totalCarrinho = Model.Sum(s => s.SubTotal);}
@totalCarrinho

```

O código para o autocomplete

Inicialmente, precisamos configurar as sessões Style e ScriptPage para utilizarem os recursos necessários para o autocomplemento. Veja a seguir o código que deve ser inserido após o fechamento da tag <html> da visão Create .

```

@section styles{
    <link href="~/Scripts/jquery-ui-1.11.4.custom/jquery-ui.css"
rel="stylesheet" />
}

@section ScriptPage {
    <script src="~/Scripts/jquery-ui-1.11.4.custom/jquery-ui.js">

```

```
</script>
}
```

Dando sequência, antes das chaves que fecham a seção `ScriptPage`, insira o código que se segue. Verifique que a seleção do controle em que será aplicado o autocomplete se dá por meio da classe que atribuímos ao controle. A configuração acontece por quatro parâmetros:

1. `minLength`, que representa o comprimento mínimo da string digitada para que a chamada a action possa ser realizada;
2. `source`, que é a fonte de dados para popular o combobox com as opções disponíveis para seleção; e
3. `select`, que representa o processo de atribuição do valor a ser passado para o controle que deverá ser atualizado após a seleção do produto desejado.

```
<script type="text/javascript">
$(document).ready(function (response, status, xhr) {
    $('.autocomplete-with-update-field').autocomplete({
        minLength: 5,
        source: function (request, response) {
            var url = $(this.element).data('url');
            $.getJSON(url, { param: request.term }, function
(data) {
                response(data);
            });
        },
        select: function (event, ui) {
            var updatefield = '#' + $("#" + event.target.id).
data('updatefield');
            $(updatefield).val(ui.item.id);
            setTimeout(function () {
                $(updatefield).focus();
            }, 200);
        }
    });
}</script>
```

No código anterior, no parâmetro `source`, você pode verificar a chamada à função `getJSON()`. Ela invocará a URL que atribuímos na visão e que está representada pelo código a seguir. Após a execução da action, seu resultado é atribuído à variável `data`, que, por meio da função `response()`, gera o combobox. Lembre-se de que esta action pertence ao `ProdutosController`.

```
public JsonResult GetProdutosPorNome(string param)
{
    var r = produtoServico.ObterProdutosPorNome(param);
    return Json(r, JsonRequestBehavior.AllowGet);
}
```

Como o método `ObterProdutosPorNome()` não existe ainda na classe `ProdutoServico`, precisamos inseri-lo. Veja como no código seguinte.

```
public IList ObterProdutosPorNome(string param)
{
    return produtoDAL.ObterProdutosPorNome(param);
}
```

Igualmente ao caso anterior, não temos ainda o método `ObterProdutosPorNome()` na classe `ProdutoDAL`. Vamos criá-lo? Veja o código da sequência.

É neste método que está o *segredo* de como os dados devem estar estruturados para popular o combobox de autocomplemento. Os nomes para as propriedades que serão criadas para a classe anônima são obrigatórios para que tudo funcione perfeitamente.

```
public IList ObterProdutosPorNome(string param)
{
    var r = from produto in context.Produtos
            where produto.Nome.ToUpper().StartsWith(param.ToUpper())
            orderby (produto.Nome)
            select new
```

```

    {
        id = produto.ProdutoId,
        label = produto.Nome,
        value = produto.Nome
    };
    return r.ToList();
}

```

9.4 REGISTRANDO O PRODUTO NO CARRINHO DE COMPRA

Com toda a implementação que fizemos até agora, já é possível pensarmos na inserção do produto no carrinho. Ela se dará por meio de uma action no `CarrinhosController`. Veja o código na sequência.

Observe que obtemos o carrinho da sessão, buscamos pelo produto (adicone a variável `produtoServico` antes do construtor), instanciamos um objeto `ItemCarrinho` e atribuímos os valores para suas propriedades. Após isso, adicionamos o item ao carrinho e atualizamos a variável de sessão, para então retornarmos uma `Partial View`. Veja, na listagem a seguir, que faço uso de uma propriedade chamada `ValorUnitario`, que você precisa inserir em seu modelo (a classe `Produto`) e, consequentemente, nas visões que trabalham esta classe.

```

[HttpPost]
[ValidateAntiForgeryToken]
public PartialViewResult AddProduto/FormCollection collection)
{
    List<ItemCarrinho> carrinho = HttpContext.Session["carrinho"]
as List<ItemCarrinho>;
    var produto = produtoServico.ObterProdutoPorId(Convert.ToInt3
2(collection.Get("idproduto")));
    var itemCarrinho = new ItemCarrinho()
    {
        Produto = produto,

```

```
        Quantidade = 1,  
        ValorUnitario = produto.ValorUnitario  
    };  
    carrinho.Add(itemCarrinho);  
    HttpContext.Session["carrinho"] = carrinho;  
    return PartialView("_ItensCarrinho",carrinho);  
}
```

Como pode ser verificado no código anterior, se inserirmos mais de uma vez o mesmo produto na coleção, ele ficará duplicado. Acho que você já tem condições de evitar isso, incrementando em um a quantidade de produtos comprada, a cada inserção repetida de um produto. Fica como atividade. :-)

9.5 O QUE FALTA PARA TERMINAR O CARRINHO?

O objetivo deste capítulo era trazer uma luz ao tema de variáveis de sessão, e isso foi bem abordado aqui. Agora, para finalizar, basta você criar um link, ao lado do botão de submissão, que redirecione o usuário para uma segunda etapa do carrinho de compras: uma visão que solicite os dados do cliente e de entrega.

Nesta visão, composta de um formulário HTTP, a submissão será para uma action que criará um carrinho de compra, associará a ele o cliente e demais dados, e então adicionará os itens de carrinho também à base de dados. A princípio, além do comentado anteriormente, você precisará criar a classe `Carrinho` e, se for trabalhar com registro de clientes, a classe `Cliente` também.

A figura a seguir apresenta como ficou o que foi implementado neste capítulo. Lembre-se também de inserir o carrinho no menu.

Você pode pensar também em uma listagem das compras já

registradas, com alteração e exclusão. Todo o conhecimento para gerar isso já foi passado. Agora, é pôr a mão na massa para fechar bem a leitura do livro.

Seleção de itens para o carrinho de compra

Digite o nome do produto e confirme a inserção

Produto para pesquisar

- DeskJet V Color
- DeskJet V Color
- DeskJet V Color abc

Produto	Quantidade	Valor	SubTotal
Total Carrinho: 0			

Figura 9.1: Registro de produtos ao carrinho de compra

9.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Vimos neste capítulo o processo relacionado a um carrinho de compra, fazendo uso de variáveis de sessão. Também foi possível verificar e aplicar um novo controle da biblioteca jQuery UI, além de realizarmos uma atualização AJAX em uma visão. O próximo capítulo apresentará a implementação de controles DropDownList aninhados, e o uso de RadioButton e CheckBox .

CAPÍTULO 10

USO DE DROPODOWNLIST ANINHADO, RADIOBUTTON E CHECKBOX

Em uma aplicação, na qual você faça uso de controles DropDownList , pode ocorrer de você precisar popular um DropDownList com opções selecionadas com base em outro DropDownList . Isso é conhecido como aninhamento.

Um exemplo clássico e que vamos implementar neste capítulo é o preenchimento do cadastro, em nosso caso, do cadastro de fabricantes. Todo fabricante precisa de um endereço e este tem em sua composição o estado e a cidade. A exibição das cidades dependerá de qual estado foi selecionado, ou seja, o nosso usuário vai precisar selecionar um estado, para nossa aplicação fazer um filtro nas cidades para popular um segundo controle, com base na seleção do primeiro. Faremos isso utilizando JavaScript, jQuery e AJAX.

Outro ponto comum também em aplicações é a existência de controles RadioButton e CheckBox . O RadioButton permite,

dentre um conjunto de opções, selecionar uma delas como válida. Em nosso cadastro de fabricantes, poderíamos ter o dado que represente o tipo de pessoa (física ou jurídica). Já no CheckBox , o trabalho refere-se ao estado de um dado, por exemplo, o estado do fabricante, Ativo ou Inativo. Trabalharemos nossa aplicação, neste capítulo, para que estas alterações possam ser implementadas no cadastro de fabricantes.

10.1 IMPLEMENTAÇÕES NECESSÁRIAS PARA AS CLASSES FORNECEDORAS DOS DROPODOWNLISTS

Esta seção traz as implementações necessárias para que possamos ver a execução de um DropDownList aninhado.

Classes de negócio

Para implementarmos o DropDownList aninhado, conforme explanado na introdução deste capítulo, precisaremos criar nossas classes. Para isso, no projeto Modelo , dentro da pasta Tabelas , crie as classes Estado e Cidade , tal qual ilustra o código a seguir.

```
using System.Collections.Generic;

namespace Modelo.Tabelas
{
    public class Estado
    {
        public long? EstadoID { get; set; }
        public string UF { get; set; }
        public string Nome { get; set; }

        public virtual ICollection<Cidade> Cidades { get; set; }
    }
}
```

```
    }
}

namespace Modelo.Tabelas
{
    public class Cidade
    {
        public long? CidadeID { get; set; }
        public long? EstadoID { get; set; }
        public string Nome { get; set; }

        public Estado Estado { get; set; }
    }
}
```

Veja, no código anterior, a implementação das propriedades das classes `Estado` e `Cidade`, em que a associação entre elas também é implementada. A definição dos IDs, seguindo o padrão do EF, já permitirá a criação das chaves primárias e estrangeiras.

Coleções no contexto do Entity Framework

Com as tabelas que serão mapeadas para o contexto da base de dados implementadas, precisamos inseri-las no contexto do Entity Framework, que fará o mapeamento para o banco de dados. Desta maneira, na classe `EFContext`, que está na pasta `Contexts` do projeto de persistência, insira o código a seguir.

```
public DbSet<Estado> Estados { get; set; }
public DbSet<Cidade> Cidades { get; set; }
```

Veja nesse código que o acesso às tabelas `Estados` e `Cidades` foi mapeado para os conjuntos de seus respectivos nomes.

Classes de acesso a dados

Você precisará implementar estas classes de acordo aos

serviços que precisem ser fornecidos, relacionados ao acesso à base de dados. Como meu objetivo neste capítulo é focalizado em determinadas implementações, apresentarei apenas códigos pertinentes a estas características. A criação básica de suas classes DAL devem estar semelhantes às apresentadas na sequência.

```
using Persistencia.Contexts;

namespace Persistencia.DAL.Tabelas
{
    public class EstadoDAL
    {
        private EFContext context = new EFContext();
    }
}

using Persistencia.Contexts;

namespace Persistencia.DAL.Tabelas
{
    public class CidadeDAL
    {
        private EFContext context = new EFContext();
    }
}
```

Nos códigos anteriores, a preocupação foi apenas implementar a base para as classes DAL, ou seja, a definição do contexto com o Entity Framework. Os métodos de acesso aos dados e atualizações serão apresentados conforme forem necessários.

Classes de serviço

Tal qual como explanado para as classes DAL, apresentarei a estrutura básica para as classes fornecedoras de serviços para os controladores. A implementação de métodos que não cumpram a proposta deste capítulo ficará a seu cargo. :-)

Como visto no capítulo *Separando a aplicação em camadas*, nosso projeto está separado em camadas. Desta maneira, precisamos criar as classes de serviço para as classes Estado e Cidade . Para isso, no projeto de Serviços, na pasta Tabelas , crie as duas classes a seguir:

```
using Persistencia.DAL.Tabelas;

namespace Servicos.Tabelas
{
    public class EstadoServico
    {
        private EstadoDAL estadoDAL = new EstadoDAL();
    }
}

using Persistencia.DAL.Tabelas;

namespace Servicos.Tabelas
{
    public class CidadeServico
    {
        private CidadeDAL cidadeDAL = new CidadeDAL();
    }
}
```

Tal qual foi a implementação básica para as classes DAL, os códigos anteriores implementam, de maneira básica, as classes de serviço para estados e cidades. Os serviços serão implementados conforme as necessidades forem surgindo.

10.2 ADAPTAÇÕES NAS IMPLEMENTAÇÕES EXISTENTES PARA O DROPODOWNLIST

Classe de Modelo do Fabricante

Precisamos inserir na classe Fabricante , em nosso modelo,

as propriedades para Estado , Cidade e Endereço , para que possamos aplicar na interface com o usuário, a visão, os DropDownList s aninhados. Veja na sequência como fica a implementação para a classe Fabricante .

```
using Modelo.Tabelas;
using System.Collections.Generic;

namespace Modelo.Cadastros
{
    public class Fabricante
    {
        public long? FabricanteID { get; set; }
        public string Nome { get; set; }

        public long? EstadoID { get; set; }
        public long? CidadeID { get; set; }

        public virtual Cidade Cidade { get; set; }
        public virtual Estado Estado { get; set; }

        public virtual ICollection<Produto> Produtos { get; set;
    }
}
}
```

Verifique, no código anterior, a implementação das propriedades que definem as associações com Estado e Cidade . Com elas, o EF cria automaticamente o relacionamento nas tabelas.

A visão e as actions Create (GET e POST)

Com a estrutura necessária já criada e adaptada, precisamos agora adaptar as actions e visões para a criação de um fabricante. Veja, no código a seguir, as instruções inseridas na visão Create.cshtml , logo após o bloco que define o controle para o nome do fabricante.

```

<div class="form-group">
    @Html.LabelFor(model => model.Estado, htmlAttributes: new { @
    class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownListFor(model => model.EstadoID, ViewBag.Es
tadoID as IEnumerable<SelectListItem>, "Selecione um estado", ne
w { @class = "form-control input-sm" })
        @Html.ValidationMessageFor(model => model.EstadoID, "", n
ew { @class = "text-danger" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.Cidade, htmlAttributes: new { @
    class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownListFor(model => model.CidadeID, ViewBag.Ci
dadeID as IEnumerable<SelectListItem>, "Selecione uma cidade", ne
w { @class = "form-control input-sm", data_url = @Url.Action("Get
CidadesDoEstado", "Cidades", new { area = "Tabelas" }) })
        @Html.ValidationMessageFor(model => model.CidadeID, "", n
ew { @class = "text-danger" })
    </div>
</div>

```

Observe, no código anterior, na utilização do Helper `DropDownListFor`, os parâmetros usados. O primeiro obtém a propriedade que será utilizada para receber o valor selecionado; o segundo define de maneira explícita o uso do `ViewBag` como um `IEnumerable<SelectedListItem>`; o terceiro define o primeiro texto a ser exibido para o usuário, para orientá-lo a selecionar um item da lista; e o quarto e último define algumas propriedades para o controle.

No caso do `DropDownList` de Cidades, existe a definição do atributo HTML `data_url`, que recebe uma `Url.Action`. Esta será invocada via JavaScript quando um estado for selecionado.

Quando o usuário requisitar a visão `Create` antes de sua renderização, a action de mesmo nome será executada. Desta

maneira, é preciso obter os dados que popularão os dois controles (estados e cidades) e enviá-los do controlador para a visão. Isso será feito com o uso de `ViewBags`, que foram apresentadas no capítulo *Associações no Entity Framework*. Veja, no código a seguir, a implementação para a action `Create` (GET).

```
public ActionResult Create()
{
    ViewBag.EstadoID = new SelectList(estadoServico.ObterEstadosClassificadosPorNome(), "EstadoID", "Nome");
    ViewBag.CidadeID = new SelectList(cidadeServico.ObterCidadesPorEstado(null), "CidadeID", "Nome");
    return View();
}
```

Os métodos utilizados anteriormente para a obtenção dos dados foram definidos nas classes `EstadoServico` e `CidadeServico`, e podem ser verificados nas implementações a seguir:

```
public IQueryable<Estado> ObterEstadosClassificadosPorNome()
{
    return estadoDAL.ObterEstadosClassificadosPorNome();
}

public IQueryable<Cidade> ObterCidadesPorEstado(long? estadoID)
{
    return cidadeDAL.ObterCidadesPorEstado(estadoID);
}
```

Por meio da leitura dos códigos anteriores, observa-se que métodos de mesmo nome, pertencentes às classes DAL, são invocados. Estes métodos podem também serem visualizados na implementação a seguir (classes `EstadoDAL` e `CidadeDAL`).

```
public IQueryable<Estado> ObterEstadosClassificadosPorNome()
{
    return context.Estados.OrderBy(b => b.Nome);
}
```

```
public IQueryable<Cidade> ObterCidadesPorEstado(long? estadoID)
{
    return context.Cidades.
        Where(c => c.EstadoID == estadoID).
        OrderBy(c => c.Nome);
}
```

Veja no código a seguir a implementação do `CidadesController`, que consome o serviço criado anteriormente.

```
using Servicos.Tabelas;
using System;
using System.Web.Mvc;

namespace Projeto01.Areas.Tabelas.Controllers
{
    public class CidadesController : Controller
    {
        private CidadeServico cidadeServico = new CidadeServico();

        public JsonResult GetCidadesDoEstado(string estadoID)
        {
            var cidades = cidadeServico.ObterCidadesPorEstado(Convert.ToInt32(estadoID));
            return Json(cidades, JsonRequestBehavior.AllowGet);
        }
    }
}
```

Com a visão apresentando nos `DropDownList`s todos os estados e todas as cidades, precisamos aplicar um filtro de cidades de acordo com o estado selecionado. Para isso, faremos uso de JavaScript.

Veja o código da sequência inserido na visão `create` de `Fabricantes`. Este código, para lembrar, faz parte da seção `@section ScriptPage`. Se você vem seguindo os capítulos, essa seção já está criada, assim como o elemento `<script>`.

```

$(document).on("change", '#EstadoID', function (e) {
    var estadoID = $(this).find(":selected").val();
    GetCidades(estadoID, '#CidadeID');
});

```

Observe, no código anterior, que uma função anônima é invocada quando ocorre a alteração (`change`) do valor do controle `EstadoID` . Nesta função, é pesquisado e obtido o valor selecionado no controle e atribuído à variável `estadoID` . Em seguida, é invocada a função `GetCidades()` (que ainda não criamos), enviando a ele como argumentos: o valor do ID do estado selecionado e o nome do controle de cidades, que a função manipulará.

Estes nomes, `EstadoID` e `CidadeID` , são atribuídos em nosso exemplo para os controles de acordo com a propriedade deles. Na sequência, veja o código para a função `GetCidades()` .

```

function GetCidades(estado, campoCidade) {
    var optionCampoCidade = campoCidade + ' option';
    if (estado.length > 0) {
        var url = $(campoCidade).data('url');
        $.getJSON(url, { estadoID: estado }, function (cidade) {
            $(optionCampoCidade).remove();
            $(campoCidade).append('<option value="">Selecione uma
cidade</option>');
            for (i = 0; i < cidade.length; i++) {
                $(campoCidade).append('<option value="' + cidade[
i].CidadeID + '">' + cidade[i].Nome + '</option>');
            }
        }).fail(function (jqXHR, textStatus, errorThrown) {
            debugger;
            alert('Erro de conexão', 'Erro obtendo cidades');
        });
    } else {
        $(optionCampoCidade).remove();
        $(campoCidade).append('<option value=""></option>');
    }
}

```

O corpo da função anterior começa com a concatenação do ID do controle para cidades com a palavra `option`, para que seja possível a remoção das opções disponibilizadas no controle antes da alteração do estado. Depois, caso acha um valor informado no controle de estado, é extraída a URL informada no controle `cidade`, no atributo `data-url`. Em seguida, por meio do jQuery, esta URL é requisitada, enviando a ela o ID do estado selecionado e retornando as cidades filtradas para este estado.

O código seguinte, dentro da função anônima de retorno da URL, são inseridas as cidades como opções para o `DropDownList`. Não detalhei a execução do `fail()` e do `else`, por terem seus códigos semânticamente compreendidos.

Com estas implementações, já é possível você executar e testar a inserção de um novo fabricante, podendo selecionar o seu estado e sua respectiva cidade. No entanto, a gravação do novo fabricante depende ainda da implementação da action `Create` (`POST`), que tem seu código apresentado na sequência.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Fabricante fabricante)
{
    if (ModelState.IsValid)
    {
        fabricanteServico.GravarFabricante(fabricante);
        return RedirectToAction("Index");
    }
    return View(fabricante);
}
```

O código apresentado anteriormente já é conhecido, o que precisamos verificar é a implementação do método `GravarFabricante()`, que deve ser implementada na classe

`FabricanteServico` , de acordo com o código apresentado a seguir. Este método será também invocado pela action `Edit` (`POST`), que implementaremos mais à frente.

```
public void GravarFabricante(Fabricante fabricante)
{
    fabricanteDAL.GravarFabricante(fabricante);
}
```

Como pode ser verificado no código anterior, o método de serviço invoca um método, de mesmo nome, da classe `FabricanteDAL` , que tem sua implementação apresentada na sequência, que você já conhece de implementações anteriores.

```
public void GravarFabricante(Fabricante fabricante)
{
    if (fabricante.FabricanteId == null)
    {
        context.Fabricantes.Add(fabricante);
    }
    else
    {
        context.Entry(fabricante).State = EntityState.Modified;
    }
    context.SaveChanges();
}
```

Visão Edit e as actions Edit (GET e POST)

Para que seja possível ao usuário realizar a alteração dos dados de um fabricante já cadastrado, também se faz necessário implementações semelhantes (com algumas particularidades) ao que foi implementado para a criação de um fabricante. A primeira alteração é a maneira que os dados são enviados do controlador para a visão, e isso pode ser visto no código a seguir, que traz a implementação da action `Edit` (`GET`).

```
// GET: Fabricantes/Edit/5
```

```
public ActionResult Edit(long? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Fabricante fabricante = fabricanteServico.ObterFabricantesPorId(id);
    if (fabricante == null)
    {
        return HttpNotFound();
    }
    ViewBag.EstadoID = new SelectList(estadoServico.ObterEstadosClassificadosPorNome(), "EstadoID", "Nome", fabricante.EstadoID);
    ViewBag.CidadeID = new SelectList(cidadeServico.ObterCidadesPorEstado(fabricante.EstadoID), "CidadeID", "Nome", fabricante.CidadeID);
    return View(fabricante);
}
```

Observe, no código anterior, que já é conhecido por você de implementações anteriores, que na definição dos ViewBags um quarto parâmetro foi inserido. Este argumento, como visto também em aplicação no capítulo *Associações no Entity Framework*, atribui aos DropDownList's os valores que deverão aparecer como selecionados. Verifique também que o método `ObterCidadesPorEstado()` agora recebe um parâmetro, que é o estado atual do fabricante.

Para a visão `Edit`, uma única alteração é necessária. Veja o código na sequência. Observe que o segundo argumento do helper `DropDownListFor` agora recebe `null`. Desta maneira, o valor passado para os `ViewBag`'s, na action, é exibido corretamente nos controles.

Implemente nesta visão o mesmo código JavaScript implementado na visão `Create`. Você pode pensar em

reutilização, criando um arquivo de código JavaScript. e nele colocar as funções. :-) Para finalizar, basta implementar a action Edit (POST), que deve ser semelhante ao implementado para Create .

```
<div class="form-group">
    @Html.LabelFor(model => model.Estado, htmlAttributes: new { @
class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownListFor(model => model.EstadoID, null, "Sel
ecione um estado", new { @class = "form-control input-sm" })
        @Html.ValidationMessageFor(model => model.EstadoID, "", n
ew { @class = "text-danger" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.Cidade, htmlAttributes: new { @
class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownListFor(model => model.CidadeID, null, "Sel
ecione uma cidade", new { @class = "form-control input-sm", data_
url = @Url.Action("GetCidadesDoEstado", "Cidades", new { area =
"Fabricantes" }) })
        @Html.ValidationMessageFor(model => model.CidadeID, "", n
ew { @class = "text-danger" })
    </div>
</div>
```

10.3 INSERÇÃO DE UM CONTROLE RADIOPUTTON EM FABRICANTES

Como exemplificado na introdução deste capítulo, implementaremos o controle RadioButton por meio da propriedade TipoPessoa , que informará se o fabricante é uma pessoa física ou jurídica. Na classe de modelo Fabricante , insira o código a seguir.

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

Na visão `Create`, após as `<div>` referentes a estado e cidade, implemente o seguinte código. Este mesmo código deverá ser implementado na visão `Edit`. Após isso, pode testar sua aplicação.

```
<div class="form-group">
    @Html.LabelFor(model => model.TipoPessoa, htmlAttributes: new
    { @class = "control-label col-md-2" })
    <div class="btn-group col-md-10">
        @Html.RadioButtonFor(model => model.TipoPessoa, "Física")
        Física<br/>
        @Html.RadioButtonFor(model => model.TipoPessoa, "Jurídica")
        Jurídica
    </div>
</div>
```

10.4 INSERÇÃO DE UM CONTROLE CHECKBOX EM FABRICANTES

Para finalizar este capítulo, vamos trabalhar a situação de uma propriedade ter um valor ou outro, reforçando que o `RadioGroup` pode ter várias opções e o `CheckBox` apenas duas (em nosso exemplo, verdadeiro ou falso). Para esta implementação, altere sua classe de negócio `Fabricante` para ter a propriedade a seguir.

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

Para concluir a implementação e poder testá-la, nas visões `Create` e `Edit`, após os `RadioButton`s, insira o código da sequência e então teste sua aplicação.

```
<div class="form-group">
    @Html.LabelFor(model => model.EstaAtivo, htmlAttributes: new
    { @class = "control-label col-md-2" })
    <div class="btn-group col-md-10">
        @Html.CheckBoxFor(model => model.EstaAtivo, "EstaAtivo")
        Está Ativo<br />
```

```
</div>  
</div>
```

10.5 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Vimos neste último capítulo a implementação de controles DropDownList aninhados, com atualização por meio de chamadas AJAX. Foram apresentadas também o uso de controles RadioButton e CheckBox . Com isso, terminamos o livro. Espero que o conteúdo trabalhado tenha sido de utilidade para você. Desejo sucesso com o ASP.NET MVC. :-)

CAPÍTULO 11

OS ESTUDOS NÃO PARAM POR AQUI

O .NET já não é uma novidade, é uma plataforma estável que vem ganhando cada vez mais adeptos e que trouxe uma nova e poderosa linguagem, o C#, com uma curva de aprendizado relativamente boa. O ASP.NET MVC 5 é um framework oferecido pelo .NET e este livro cuja leitura você concluiu foi sobre ele. Além do framework, foi possível conhecer o Visual Studio, Entity Framework, Bootstrap e um pouquinho de jQuery e JavaScript.

O livro teve seus três capítulos iniciais como introdutórios, apresentação do ASP.NET MVC 5, do Entity Framework 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 suas futuras aplicações e personalização de propriedades das classes de modelo. Em seus dois capítulos seguintes foram apresentadas técnicas e recursos para o controle de acesso de usuários à aplicação, uploads, downloads e tratamento de erros. O livro finalizou, com seus dois últimos capítulos, apresentando o processo de carrinho de compra, DropDownList aninhados, RadioButton e CheckBox .

Espera-se que, com os recursos e técnicas exibidos, alguns

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 e o ASP.NET MVC 5 não é difícil.

Comece agora a criar suas próprias aplicações. Quando surgirem dificuldades, você verá que a comunidade existente na internet, disposta a lhe auxiliar, é muito grande. Vale relembrar do grupo de discussão deste livro em específico:
<http://forum.casadocodigo.com.br/>.

Sucesso, e que a força esteja com você! ;-)