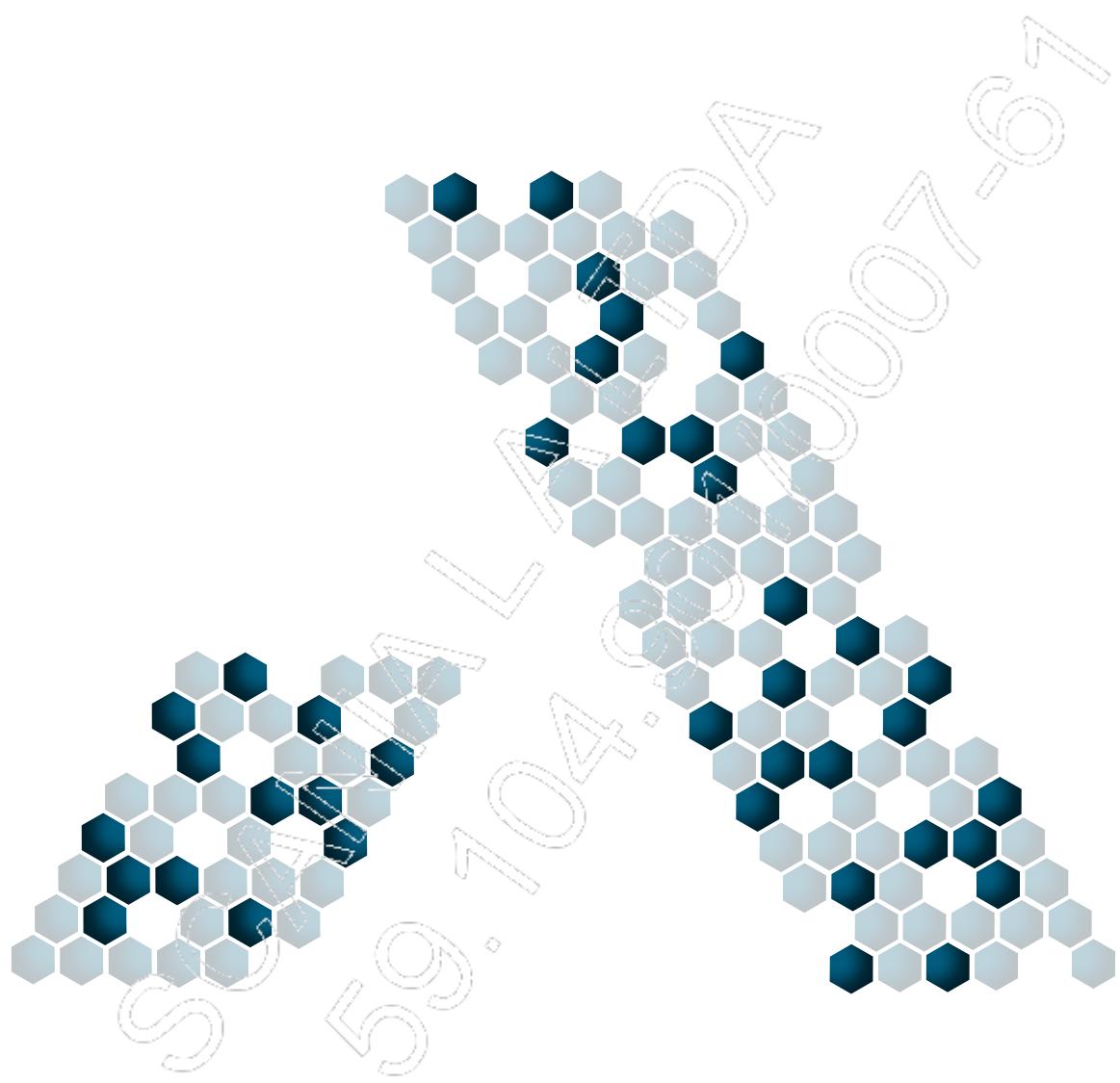




Desenvolvimento Web com ASP.NET MVC e ASP.NET Core



Editora
IMPACTA





Desenvolvimento Web com ASP. NET MVC e ASP.NET Core



Editora
IMPACTA



Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Monte Everest Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

Desenvolvimento Web com ASP.NET MVC e ASP.NET Core

Coordenação Geral

Henrique Thomaz Bruscagin

Autoria

Emilio Celso de Souza

Revisão Ortográfica e Gramatical

Fernanda Monteiro Laneri

Marcos César dos Santos Silva

Diagramação

Bruno de Oliveira Santos

Edição nº 1 | 1891_0

Junho/ 2020

Sumário

Capítulo 1 - Fundamentos de aplicações Web	09
1.1. Introdução	10
1.2. Arquitetura cliente-servidor	10
1.3. O servidor Web	10
1.4. O protocolo HTTP	11
1.4.1. Requisições (Request)	11
1.4.2. Respostas (Response)	12
1.5. Tecnologias utilizadas nas páginas Web	13
1.5.1. Exemplo de uma página retornada do servidor contendo HTML, CSS e JavaScript ..	14
1.6. Tecnologias utilizadas no servidor	15
1.7. IIS - Internet Information Services	16
1.7.1. Habilitando o IIS no Windows	16
1.7.2. IIS - Criando uma aplicação ASP.NET	19
1.8. O IIS Express e o Visual Studio	21
1.9. Criando um projeto web	22
Pontos principais	29
Teste seus conhecimentos.....	31
Mãos à obra!.....	35
Capítulo 2 - Web Pages com Razor	41
2.1. Introdução	42
2.2. O mecanismo Razor	42
2.2.1. Criando um projeto com uma Web Page usando Razor	43
2.2.2. Estruturas de repetição	47
2.2.3. Estruturas condicionais	49
2.2.4. Render e layout	51
2.2.4.1. RenderPage()	51
2.2.4.2. RenderBody()	53
2.2.4.3. RenderSection()	54
2.2.5. Request e Form	56
2.2.6. HTML Helpers	60
Pontos principais	61
Teste seus conhecimentos.....	63
Mãos à obra!.....	67
Capítulo 3 - ASP.NET MVC – Controllers, Models e Views	89
3.1. Introdução	90
3.2. Criando um projeto MVC	90
3.2.1. Conhecendo as partes de um projeto ASP.NET MVC	92
3.3. Rotas	93
3.3.1. Iniciando a rota	95
3.4. Controllers e Actions	95
3.4.1. Arquitetura MVC (Model – View – Controller)	96
3.4.2. Criando o controller	96
3.4.3. Criando um action	99
3.4.4. Retornos comuns de actions	101
3.4.5. Criando Views	103
3.5. Os objetos ViewBag e ViewData	106
3.6. Models - Views fortemente tipadas	108
3.6.1. Definindo o Model	108
3.6.2. Consumindo o Model no Controller	109
3.6.3. Consumindo o Model na View	110
3.7. Elaboração de formulários	112
3.7.1. HTML Helpers	112

Desenvolvimento Web com ASP.NET MVC e ASP.NET Core

3.7.2.	Formulário não tipado	113
3.7.3.	Formulário fortemente tipado	116
3.8.	Validação e filtros	119
3.8.1.	Validação.....	119
3.8.2.	Data Annotation	119
3.8.2.1.	ModelState	120
3.8.2.2.	Required.....	120
3.8.2.3.	StringLength.....	124
3.8.3.	Regular Expression	125
3.8.4.	Range.....	125
3.8.5.	Remote.....	126
3.8.6.	Compare	126
3.8.6.1.	Display	127
3.8.6.2.	DisplayFormat	131
3.9.	Layout	132
3.9.1.	Fluxo de execução do MVC com o layout	137
	Pontos principais	138
	Teste seus conhecimentos.....	139
	Mãos à obra!	143

Capítulo 4 - Entity Framework..... 153

4.1.	Introdução.....	154
4.2.	Preparando o ambiente.....	155
4.3.	Database First	157
4.3.1.	Usando o modelo criado	166
4.4.	Model First	167
4.4.1.	Adicionando uma Entity (entidade).....	169
4.4.2.	Adicionando propriedades	171
4.4.3.	Adicionando associações	173
4.4.4.	Criando o banco de dados	175
4.4.5.	Usando o modelo criado	177
4.5.	Code First.....	178
4.5.1.	Modelo de domínio.....	179
4.5.2.	Contexto da conexão.....	182
4.5.3.	Mapeamento.....	185
4.5.4.	Mapeamento por código	191
4.5.4.1.	Método OnModelCreating	191
4.5.5.	Database Initializer	195
	Pontos principais	196
	Teste seus conhecimentos.....	197
	Mãos à obra!	201

Capítulo 5 - Segurança – ASP.NET Identity..... 269

5.1.	Introdução.....	270
5.2.	ASP.NET Identity	270
5.3.	OWIN	274
5.4.	Implementações do Visual Studio	279
5.4.1.	Incluindo um menu de acesso.....	279
5.4.2.	Criando os models e os actions	282
5.4.3.	Escrevendo o código para criar um novo usuário	285
5.4.3.1.	Definindo a string de conexão	286
5.4.3.2.	Implementação do cadastro de usuários	287
5.4.3.3.	Implementação do Login	289

Sumário

5.4.3.4. Implementação do Logout	290
5.4.4. Executando a aplicação.....	290
Pontos principais	294
Teste seus conhecimentos.....	295
Mãos à obra!.....	299

Capítulo 6 - Serviços - Web API..... 313

6.1. Introdução à Web API.....	314
6.2. REST na prática.....	315
6.3. Usando Web API.....	316
6.3.1. Estrutura Web API	317
6.3.2. Criando um serviço REST	320
6.3.3. Definindo actions para o verbo HTTP GET	322
6.3.4. Testando o método GET com parâmetros	327
6.3.5. Definindo actions para os verbos HTTP POST, PUT e DELETE	328
6.4. Criando um projeto para consumir o serviço.....	331
Pontos principais	349
Teste seus conhecimentos.....	351
Mãos à obra!.....	355

Capítulo 7 - ASP.NET Core 379

7.1. Introdução	380
7.2. .NET Core e seus projetos.....	381
7.3. Comandos do .NET Core	381
7.3.1. Criando aplicativos por linhas de comandos	381
7.3.2. Usando o Visual Studio Code	383
7.3.3. Usando o Visual Studio	386
7.3.4. Aplicação ASP.NET Core.....	388
7.4. Fluxo de execução de uma aplicação ASP.NET Core	390
7.5. Controllers e Actions	392
7.5.1. Adicionando um controller ao projeto.....	393
7.6. Microsoft.AspNetCore.Mvc.TagHelpers.....	394
Pontos principais	398
Teste seus conhecimentos.....	399
Mãos à obra!.....	403

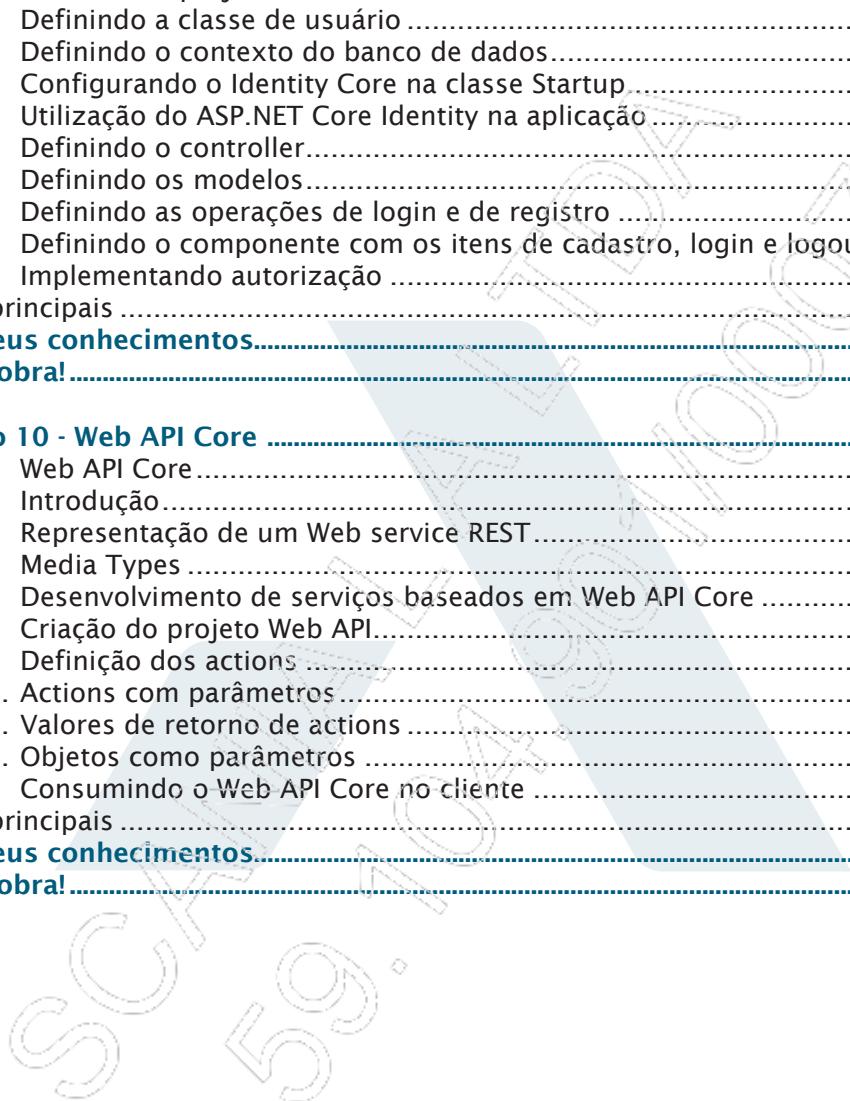
Capítulo 8 - Entity Framework Core..... 411

8.1. Introdução.....	412
8.2. Entity Framework Core (EF Core).....	412
8.3. Database Providers	412
8.4. Usando o Entity Framework Core	413
8.4.1. Definindo a entidade	414
8.4.2. Definindo o contexto	415
8.4.3. Adicionando o DbSet ao contexto	415
8.4.4. Conectando o EF Core ao Provider SQL Server	416
8.4.5. Definindo o componente responsável por criar o banco de dados	418
8.4.6. Configurando a aplicação para a injeção de dependência.....	418
8.4.7. Preparando o controller para incluir e listar registros.....	420
Pontos principais	426
Teste seus conhecimentos.....	427
Mãos à obra!.....	431



Desenvolvimento Web com ASP.NET MVC e ASP.NET Core

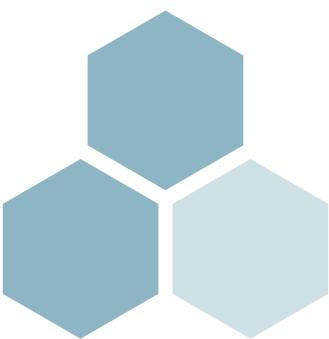
Capítulo 9 - ASP.NET Core Identity	465
9.1. Introdução.....	466
9.2. Configuração do ASP.NET Core Identity.....	466
9.2.1. Definindo o projeto	467
9.2.2. Definindo a classe de usuário	468
9.2.3. Definindo o contexto do banco de dados.....	469
9.2.4. Configurando o Identity Core na classe Startup.....	469
9.3. Utilização do ASP.NET Core Identity na aplicação.....	471
9.3.1. Definindo o controller.....	471
9.3.2. Definindo os modelos.....	472
9.3.3. Definindo as operações de login e de registro	473
9.3.4. Definindo o componente com os itens de cadastro, login e logout.....	479
9.4. Implementando autorização	481
Pontos principais	484
Teste seus conhecimentos.....	485
Mãos à obra!.....	489
Capítulo 10 - Web API Core	505
10. Web API Core.....	506
10.1. Introdução.....	506
10.2. Representação de um Web service REST.....	506
10.3. Media Types	506
10.4. Desenvolvimento de serviços baseados em Web API Core	507
10.5. Criação do projeto Web API.....	508
10.5.1. Definição dos actions	509
10.5.1.1. Actions com parâmetros	512
10.5.1.2. Valores de retorno de actions	513
10.5.1.3. Objetos como parâmetros	515
10.5.2. Consumindo o Web API Core no cliente	516
Pontos principais	521
Teste seus conhecimentos.....	523
Mãos à obra!.....	527



1

Fundamentos de aplicações Web

- Arquitetura cliente-servidor;
- O servidor Web;
- O protocolo HTTP;
- Tecnologias utilizadas nas páginas Web e no servidor;
- IIS – Internet Information Services;
- O IIS Express e o Visual Studio;
- Criando um projeto Web.



1.1. Introdução

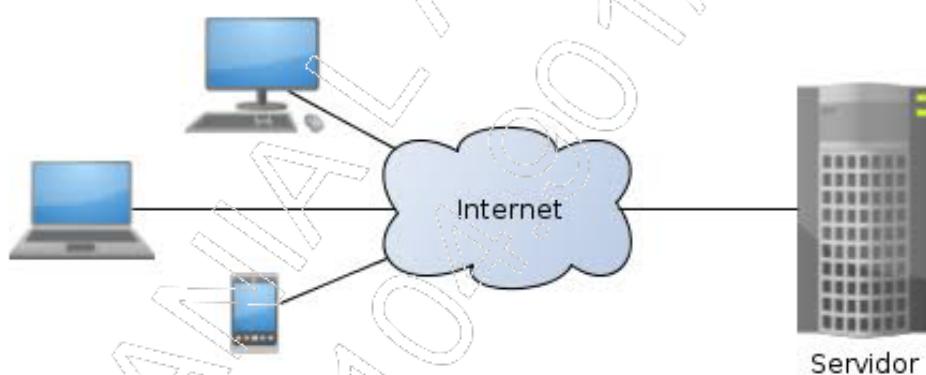
Uma aplicação Web consiste essencialmente em duas partes: um cliente e um servidor. O cliente é o browser, onde as páginas da aplicação são visualizadas, e por meio do qual ocorrem todas as interações com o usuário. O servidor é a parte do sistema responsável por receber e processar as requisições realizadas na aplicação (por intermédio do browser).

Neste capítulo, serão apresentados os conceitos de cliente e servidor, como uma aplicação Web é criada e executada, e como habilitar o servidor IIS no Windows.

1.2. Arquitetura cliente-servidor

Uma arquitetura em que um aplicativo solicita processamento de outro aplicativo ou serviço é chamada **cliente-servidor**.

A Internet é um exemplo dessa arquitetura. O **cliente** é o navegador Web e o **servidor** é o serviço que disponibiliza recursos para receber os dados do cliente e responde por essa requisição.



Observe que o cliente, sendo um browser, pode ser executado em diferentes dispositivos, e não apenas em um computador.

1.3. O servidor Web

Para desempenhar o papel de servidor, um computador necessita de um programa ou serviço que seja capaz de receber solicitações do cliente e de responder a essas solicitações. No Windows, existe um pacote de serviços conhecido como **IIS (Internet Information Services)**. Esses serviços incluem, entre outras coisas, disponibilização de páginas no formato HTML usando o protocolo HTTP, disponibilização de arquivos usando o protocolo FTP, gerenciamento de e-mails, módulos de aplicativos, certificados digitais, entre outros.

1.4. O protocolo HTTP

O **HTTP** (**H**yper**T**ext **T**ransfer **P**rotocol) é um protocolo de comunicação que estabelece transferência de dados entre os navegadores Web e os servidores de aplicação.

O protocolo HTTP se baseia em comandos e dados enviados ao servidor. Os comandos são chamados de **verbos** (**verbs**) e são os seguintes:

- **GET**: Este comando obtém informações do servidor. Usado para pesquisa de dados;
- **POST**: Este comando é utilizado para enviar ao servidor solicitações com dados encapsulados dentro da requisição. É usado para processar informações;
- **HEAD**: Igual ao GET, mas sem o corpo da página. Apenas os cabeçalhos de identificação são retornados. Os cabeçalhos são informações, enviadas junto com a página, que definem, entre outras coisas, o tipo de conteúdo que será transmitido, o conjunto de caracteres utilizado (encoding), os dados armazenados no servidor (cookies) e o tamanho, em bytes, desses dados;
- **PUT**: Comando semelhante ao POST, usado para incluir ou criar novos dados;
- **DELETE**: Comando para excluir dados;
- **TRACE**: Comando para exibir a requisição que foi enviada ao servidor. Usado para investigar processos;
- **OPTIONS**: Comando que retorna os comandos suportados pelo servidor;
- **CONNECT**: Usado para criar uma conexão criptografada usando o protocolo TCP/IP. Esse processo é chamado **tunnel TCP/IP**;
- **PATCH**: Usado para realizar modificações parciais no servidor.

É importante lembrar que os comandos em si não efetuam nenhum processamento no servidor. O aplicativo instalado no servidor é responsável por interpretar e executar os comandos necessários.

1.4.1. Requisições (Request)

O formato básico de uma requisição HTTP (**Request**) é o seguinte:

```
GET /Home/Contact HTTP/1.1
Host: localhost:51021
Accept: text/html

Cookie: usuario=3WrUX
```

A linha 1 informa o método (**GET**), a URL relativa ao servidor (**/Home/Contact**) e o protocolo (**HTTP/1.1**). As informações são separadas por espaço.

A linha 2 informa o endereço do servidor (**Host**).

A linha 3 adiciona uma informação à requisição. Nesse exemplo, apenas um item está definido (**Accept**), mas múltiplas informações podem ser inseridas nessa parte. Nesse caso, o item **Accept** diz ao servidor o formato de dados esperado pelo cliente nessa requisição.

A linha 5 informa os cookies armazenados pelo navegador. Um cookie é um texto armazenado pelo navegador e transmitido para o servidor a cada requisição.

No exemplo anterior, o método GET solicita dados do servidor. Geralmente, esse método é utilizado para solicitar uma página. Para enviar dados digitados em um formulário, o método POST pode ser utilizado. Nesse caso, a informação digitada pelo usuário é incluída dentro da requisição:

```
POST / Usuario/Cadastro HTTP/1.1
Host: localhost:51021
Content-Length: 37

nome=Maria&email=maria@teste.com.br
```

No exemplo anterior, a linha 3 informa o tamanho da informação; a linha 4 está em branco (deve ser assim); e a linha 5 contém os dados enviados ao servidor.

1.4.2. Respostas (Response)

O formato da resposta (**Response**) no protocolo HTTP é o seguinte:

```
HTTP/1.1 200 OK
Content-Length: 12260
Content-Type: text/html;

<html><body><h1>Teste</h1></body></html>
```

A linha 1 informa o protocolo (**HTTP/1.1**), o status da resposta (**200**) e o texto do status da resposta (**OK**). Os status mais comuns são os seguintes:

- 200 – OK;
- 204 – Nenhum conteúdo;
- 404 – Não encontrado;
- 401 – Não autorizado;
- 500 – Erro interno do servidor.

A linha 2 informa o tamanho das informações (**Content-Length**) que serão transmitidas. O navegador pode criar uma barra de progresso para informar ao usuário quando a página foi completamente carregada.

A linha 3 define o tipo de conteúdo (**Content-Type**). Nesse caso, é uma página HTML. O servidor pode definir muitos outros dados e inseri-los a partir dessa linha.

A linha 4 está em branco.

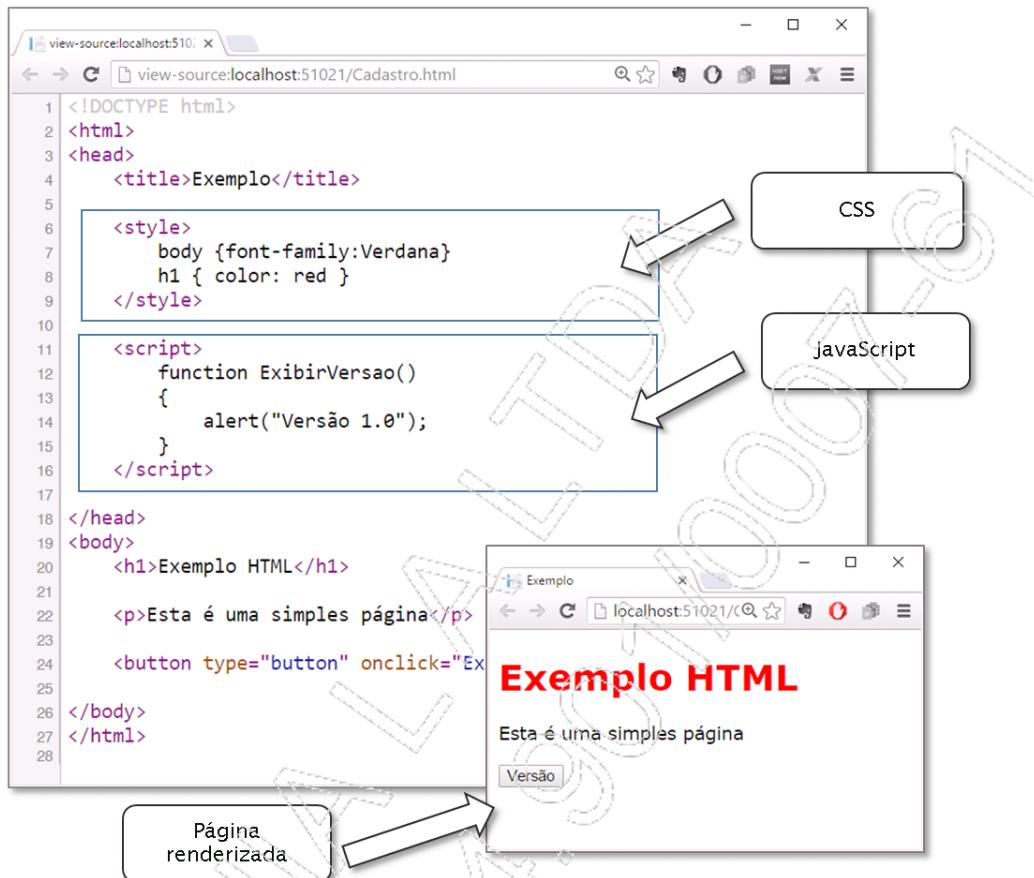
Da linha 5 em diante é o conteúdo da página.

1.5. Tecnologias utilizadas nas páginas Web

O **servidor** pode enviar ao **cliente** qualquer tipo de conteúdo: textos, imagens ou arquivos binários. Cabe ao **navegador** entender e gerar uma página a partir dos arquivos recebidos. Todo navegador, por padrão, consegue manipular três tipos de arquivos: HTML, JavaScript e CSS.

- **HTML (HyperText Markup Language)**, ou linguagem de marcação de hipertexto, é uma linguagem criada para construir documentos. O objetivo desta linguagem é definir, em um documento, onde ficam o título, o subtítulo, um parágrafo, uma seção, um vínculo com outro documento, uma área para navegação, entre outras partes. Essa definição é feita usando tags (marcas) no texto;
- **CSS (Cascading Style Sheets)**, ou folhas de estilo em cascata, é uma linguagem criada para definir a aparência de um documento HTML. As folhas de estilo utilizam seletores – para selecionar qual parte do documento HTML vai ser formatada – e palavras reservadas – para definir itens como tamanho, cor, margem, espaçamento e outras propriedades relacionadas à maneira como as informações serão exibidas;
- **JavaScript** é uma linguagem de programação executada pelo navegador, ou seja, do lado do cliente. A linguagem JavaScript pode interagir com os elementos de uma página HTML usando um modelo de objetos chamado **DOM (Document Object Model)** e pode interceptar eventos como movimentos do mouse, gestos (no caso de dispositivos touch) ou carregamento de imagens.

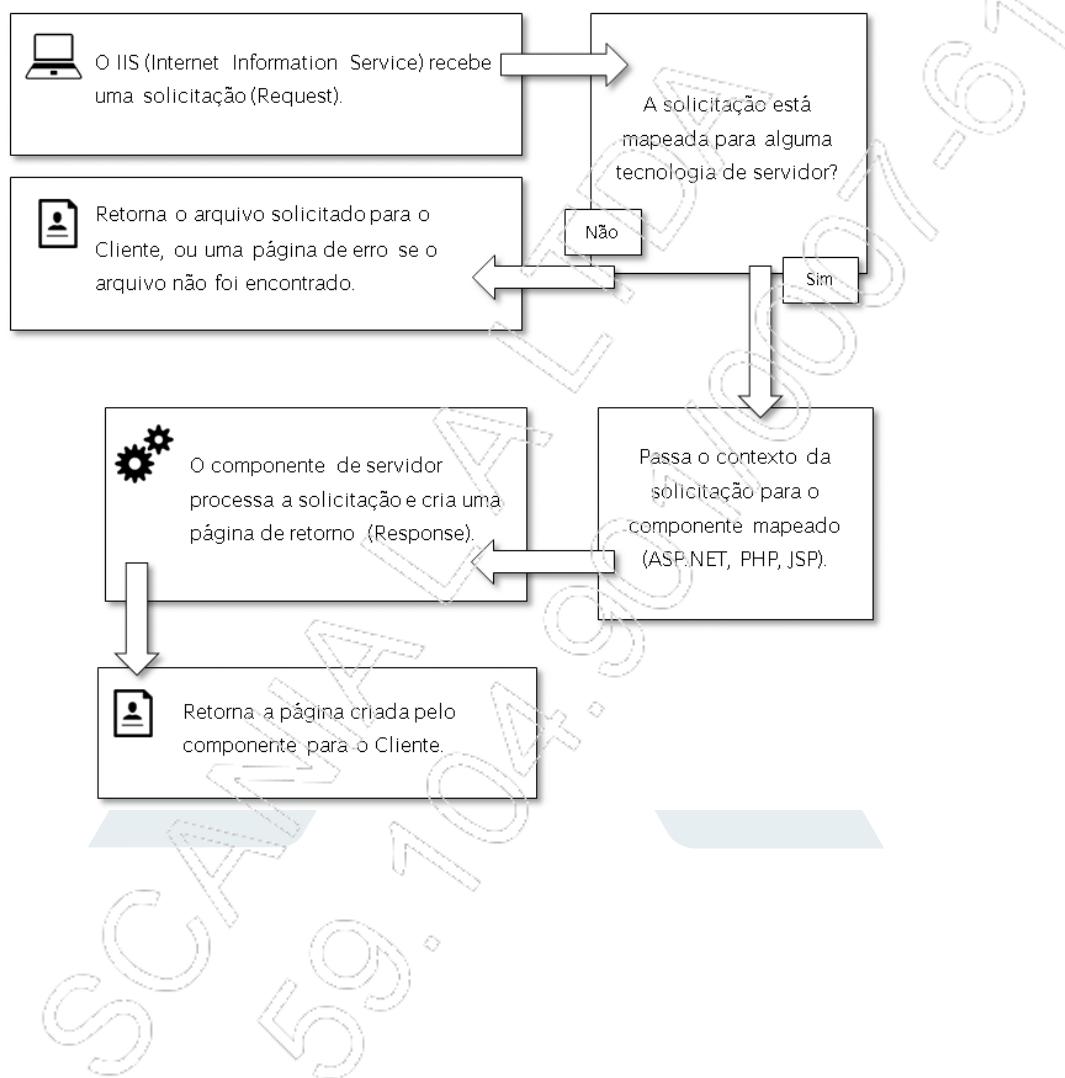
1.5.1. Exemplo de uma página retornada do servidor contendo HTML, CSS e JavaScript



Uma página real pode parecer bem mais complexa do que a do exemplo. Isso ocorre porque, além de o conteúdo ser maior, as folhas de estilo e o JavaScript são normalmente colocados em arquivos separados, existem referências a imagens, bibliotecas de terceiros, metadados e objetos incorporados.

1.6. Tecnologias utilizadas no servidor

O servidor Web é o responsável por atender a solicitações do cliente, localizar o conteúdo solicitado, preparar o documento HTML e retorná-lo. O ASP.NET entra nesse processo depois que o IIS recebeu a solicitação e antes de retornar o conteúdo para o cliente. Esse caminho é chamado de **pipeline** (encanamento) e diversos componentes podem ser colocados nesse processo.



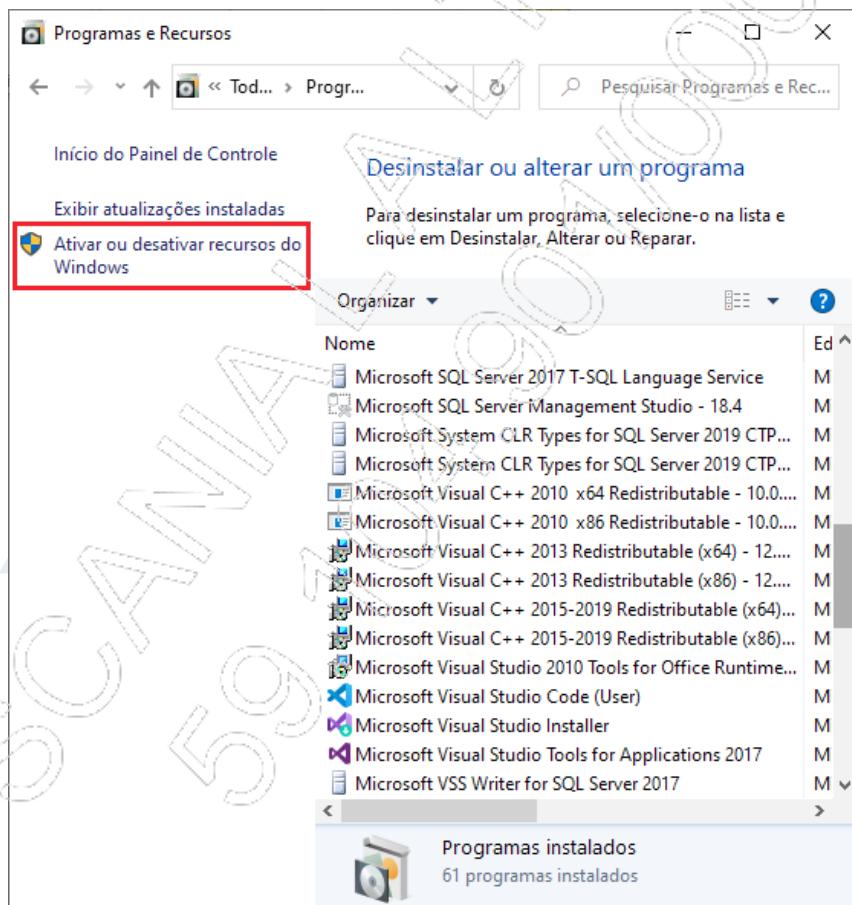
1.7.IIS – Internet Information Services

O IIS (Internet Information Services) é o serviço do sistema operacional Windows responsável por receber e responder a solicitações via protocolo HTTP.

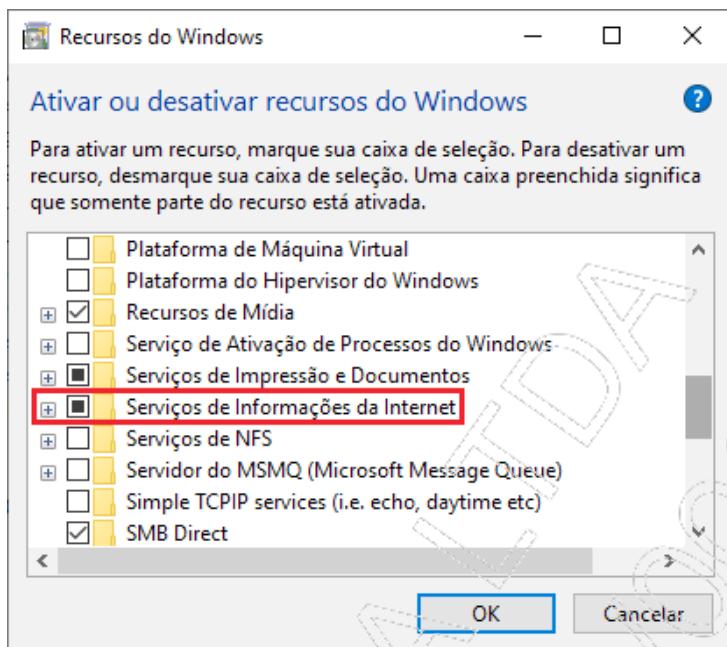
1.7.1. Habilitando o IIS no Windows

O IIS já está disponível no Windows, desde que seja uma versão Professional ou superior. Existem poucos passos que devem ser seguidos para habilitar os serviços do IIS:

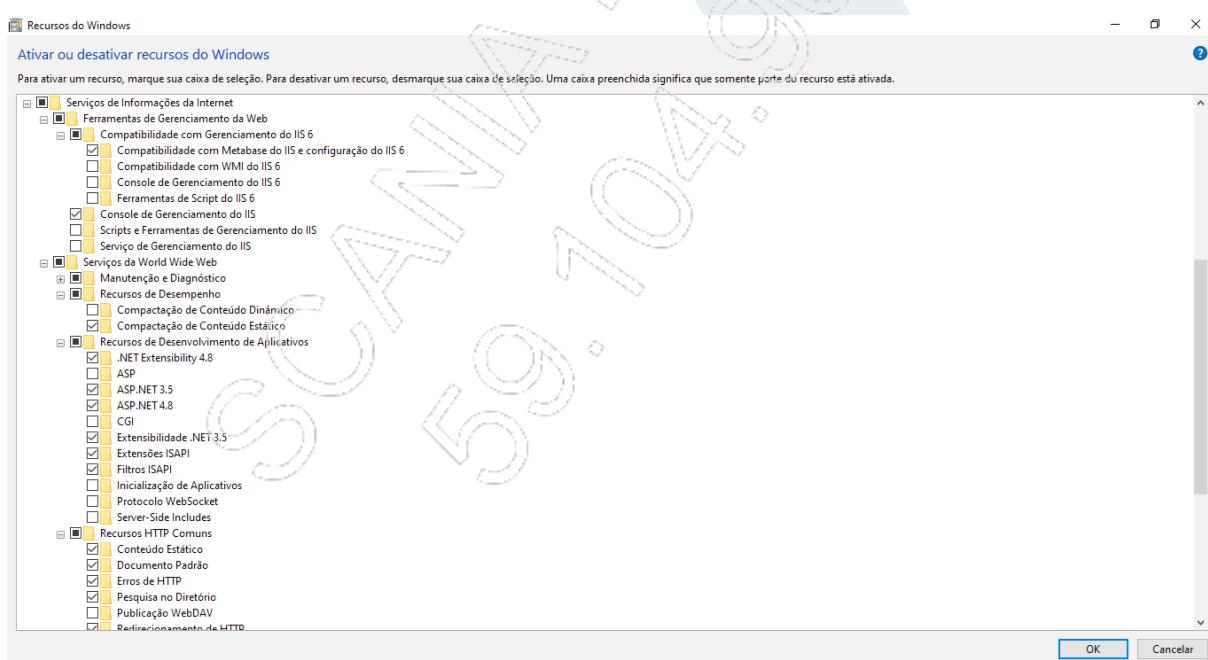
- No Painel de Controle, selecione a opção **Programas e Recursos**;
- Em seguida, selecione a opção **Ativar ou Desativar Recursos do Windows**:



- Localize o item **Serviços de Informação da Internet** (Internet Information Services).



- Para habilitar os recursos de publicação de aplicações desenvolvidas em **Asp. Net MVC, WebAPI e Asp.Net Core**, selecione as opções listadas:



O mais importante é o item relacionado a **Recurso de Desenvolvimento de Aplicativos**. A habilitação do ASP.NET 4.8 é fundamental para a publicação dos recursos dessa plataforma.

- Após esta etapa, basta aguardar o processo de instalação e atualização.

A partir deste momento, estamos prontos para publicar aplicações no IIS.

Para administrar este serviço, é utilizado o **Gerenciador de Serviços de Informações da Internet (IIS)**, dentro do **Painel de Controle**, em **Ferramentas Administrativas**.

Em linhas gerais, o que esse serviço faz é mapear uma pasta física para uma pasta virtual. A pasta virtual pode ser isolada em uma área de memória protegida e preparada para executar scripts. Quando uma pasta é preparada dessa forma, é chamada de **pasta de aplicação**.

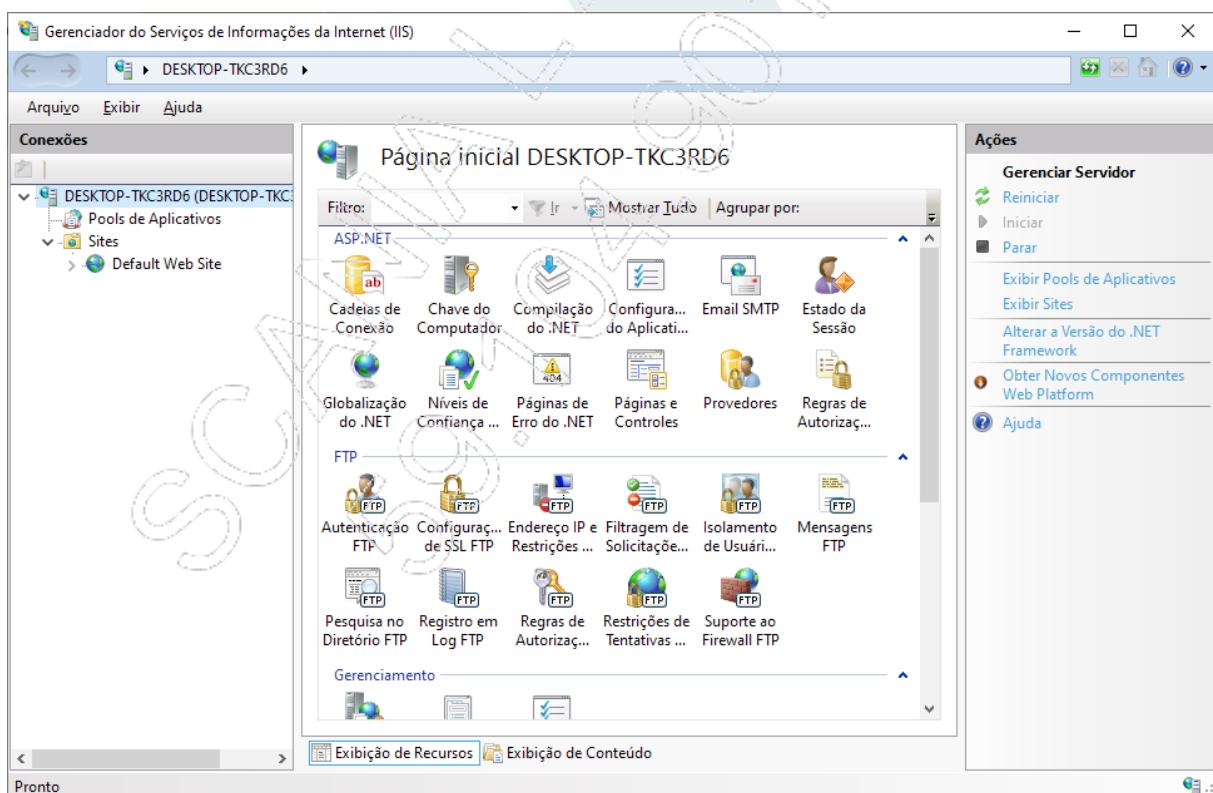
A pasta física é o local real onde está um aplicação Web. Por exemplo:

C:\inetpub\wwwroot\MinhaAppWeb

A pasta virtual é a URL (Uniform Resource Location) que dá acesso ao servidor e à aplicação Web. Por exemplo:

http://MeuServidor/AppExemplo

Executando o gerenciador do IIS, obtém-se a seguinte tela:



Na janela **Conexões**, existe o nome do servidor e o Web Site padrão, chamado **Default Web Site**. Cada aplicação Web pode compartilhar dados de segurança com outras aplicações. O item **Pool de Aplicativos** define esses grupos.

1.7.2. IIS – Criando uma aplicação ASP.NET

Para criar uma aplicação ASP.NET, siga os passos descritos a seguir:

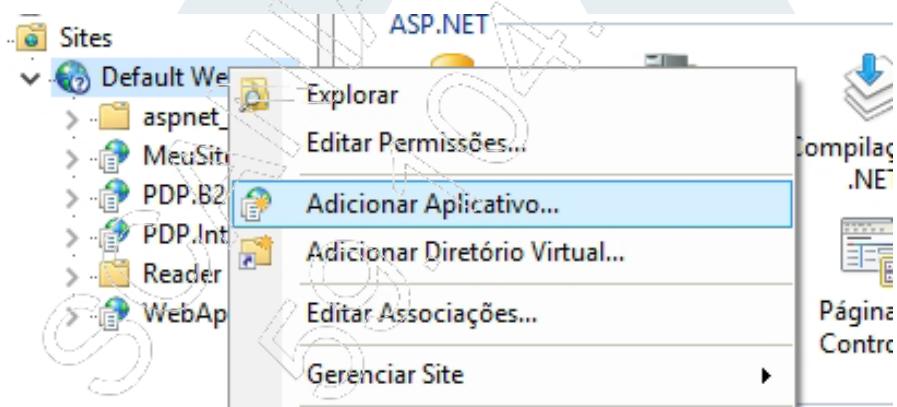
1. Crie uma pasta em qualquer drive. Se quiser usar a "pasta oficial" do ASP.NET, o endereço é **C:\inetpub\wwwroot\NomeDoSeuApp**;
2. Crie um arquivo HTML (usando o Notepad, o Notepad++ ou o editor de sua preferência) chamado **Index.htm** dentro dessa pasta. Use o template mínimo para um arquivo HMTL:

```
<!DOCTYPE html>

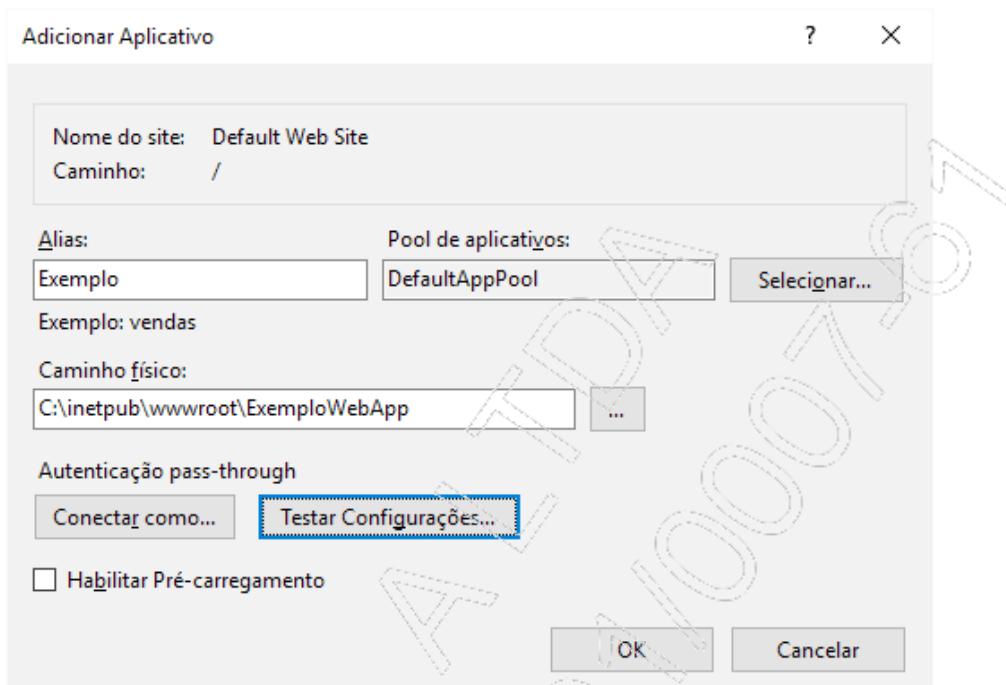
<html>
<head>
    <title></title>
</head>
<body>
    <h1>Exemplo de Arquivo HTML</h1>

    <p>Isto é um parágrafo</p>
</body>
</html>
```

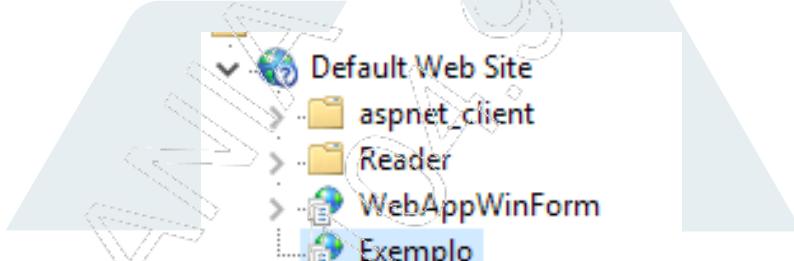
3. No gerenciador do IIS, no menu de contexto de **Default Web Site**, escolha **Adicionar Aplicativo**:



4. Informe o apelido do site (**Alias**) e o caminho físico. O alias pode ser qualquer nome sem espaços ou acentos. É por meio desse nome que o aplicativo será executado. O caminho físico deve ser exatamente aquele onde a pasta foi criada;



5. Verifique se foi criado corretamente:



6. Usando qualquer navegador, entre no endereço usando o seguinte modelo: **http://NomeDoServidor/Alias/Index.htm**.

Por exemplo, se o seu servidor se chama **Servidor01** e o apelido do seu aplicativo é **MeuApp**, a URL será **http://Servidor01/MeuApp/Index.htm**.

Quando o servidor é o mesmo computador onde está o navegador, é possível trocar o nome do servidor por **localhost** ou pelo número de IP.

O arquivo **Index.html** está na lista dos arquivos que abrem automaticamente, por isso pode ser omitido.



O exemplo anterior executa e o browser consegue exibir o arquivo, mas não há nenhum processamento no servidor além do IIS, porque a extensão **htm** não está mapeada para nenhum componente. Nesse caso, o IIS apenas entrega o arquivo solicitado.

1.8. O IIS Express e o Visual Studio

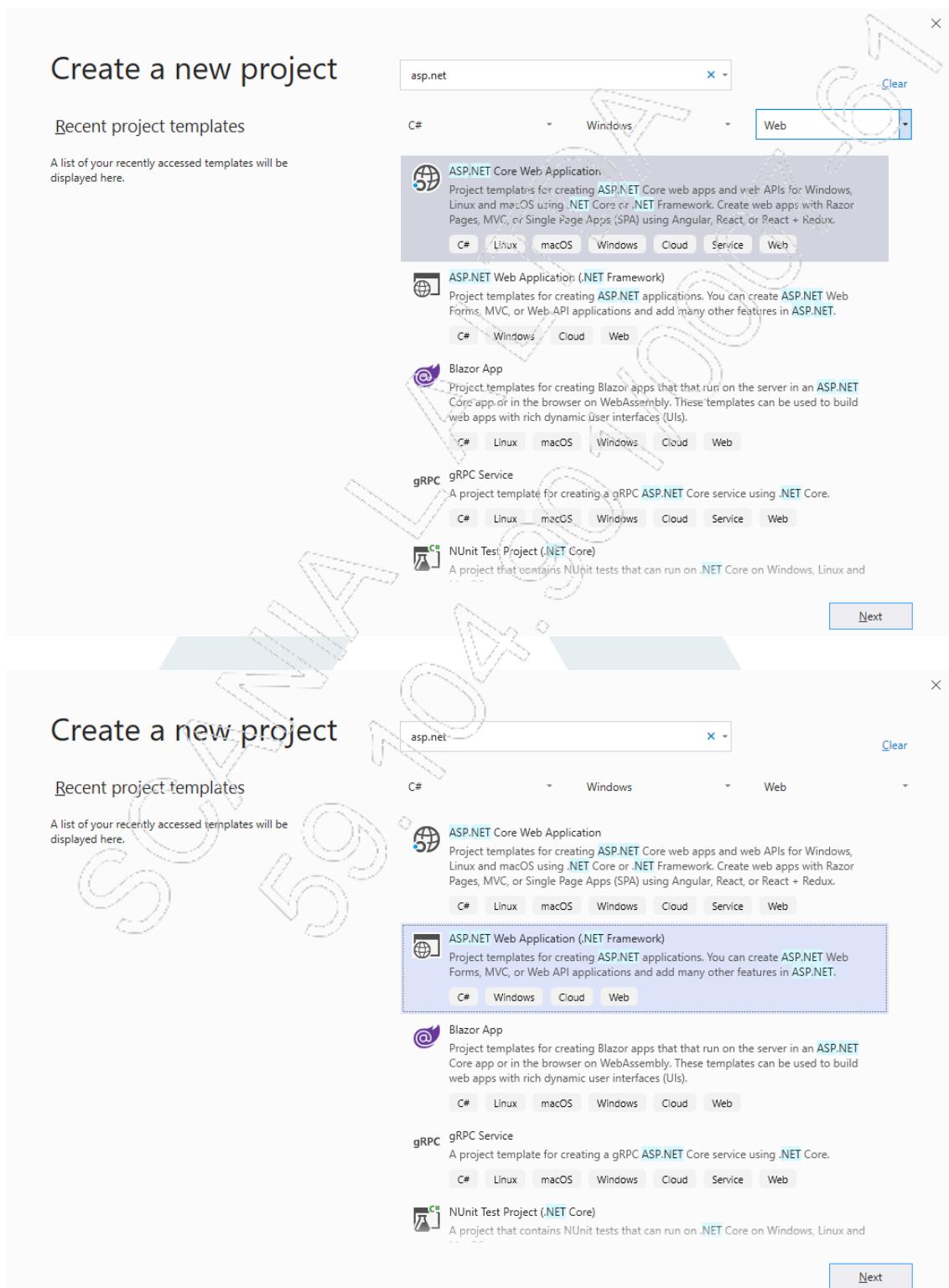
Apesar de ser bastante eficiente usarmos o servidor IIS e uma aplicação devidamente publicada, nem sempre é produtivo publicar todas as aplicações que estejam em fase de desenvolvimento.

O Visual Studio disponibiliza uma versão mais simplificada do servidor IIS, que permite ao desenvolvedor poupar esse tempo de publicações constantes. Trata-se do **IIS Express**. Ele age como um serviço temporário, ou seja, um ambiente em **Run Time**, que é executado quando um projeto Web é executado.

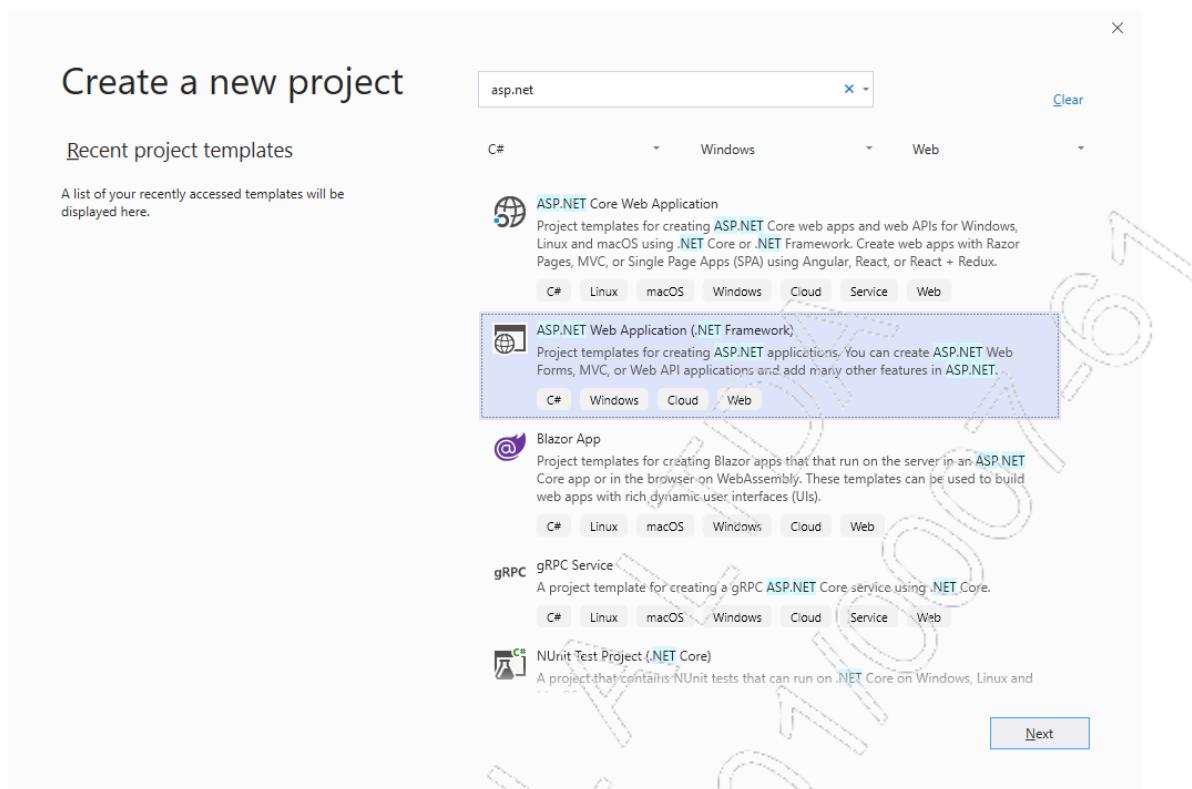
Em todos os projetos que elaborarmos ao longo do curso, usaremos o IIS Express, já que realizaremos diversos testes.

1.9. Criando um projeto web

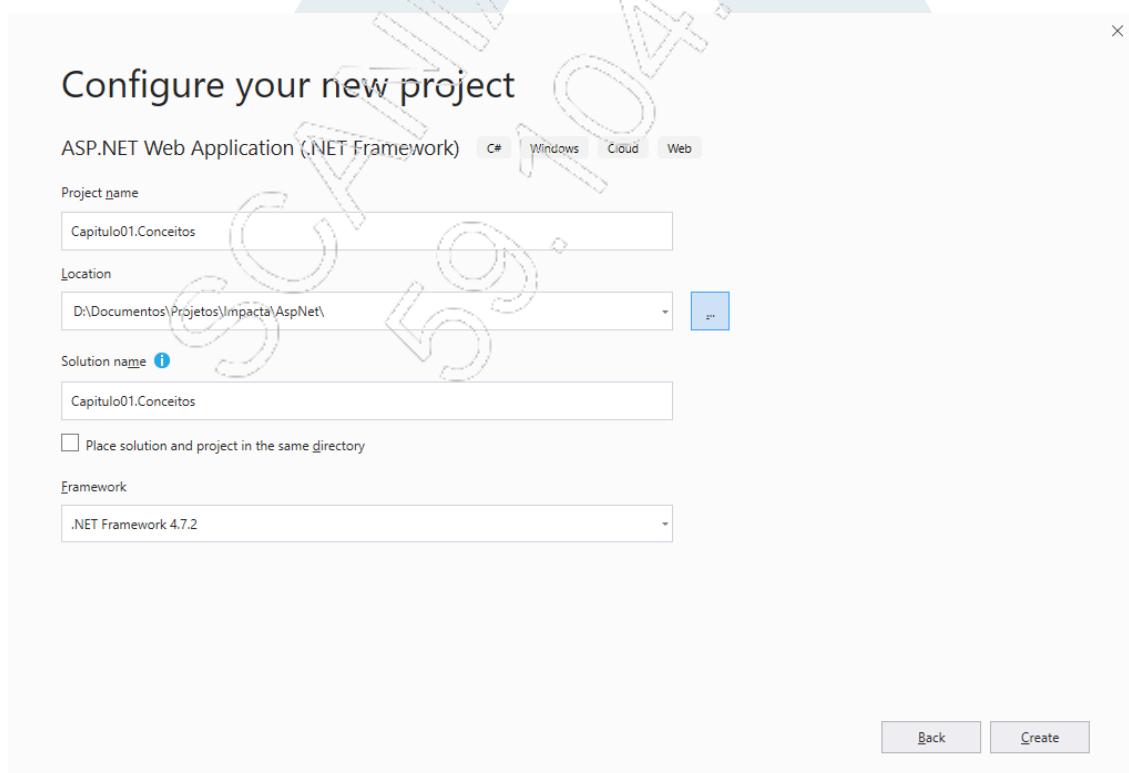
Um projeto Web é composto por uma série de arquivos devidamente organizados, que são convertidos em uma aplicação Web quando publicados no IIS. Existem vários templates para a criação de aplicações Web no Visual Studio. Usando o comando de menu **File / New Project**, podemos escolher o tipo de projeto. Veja algumas opções:



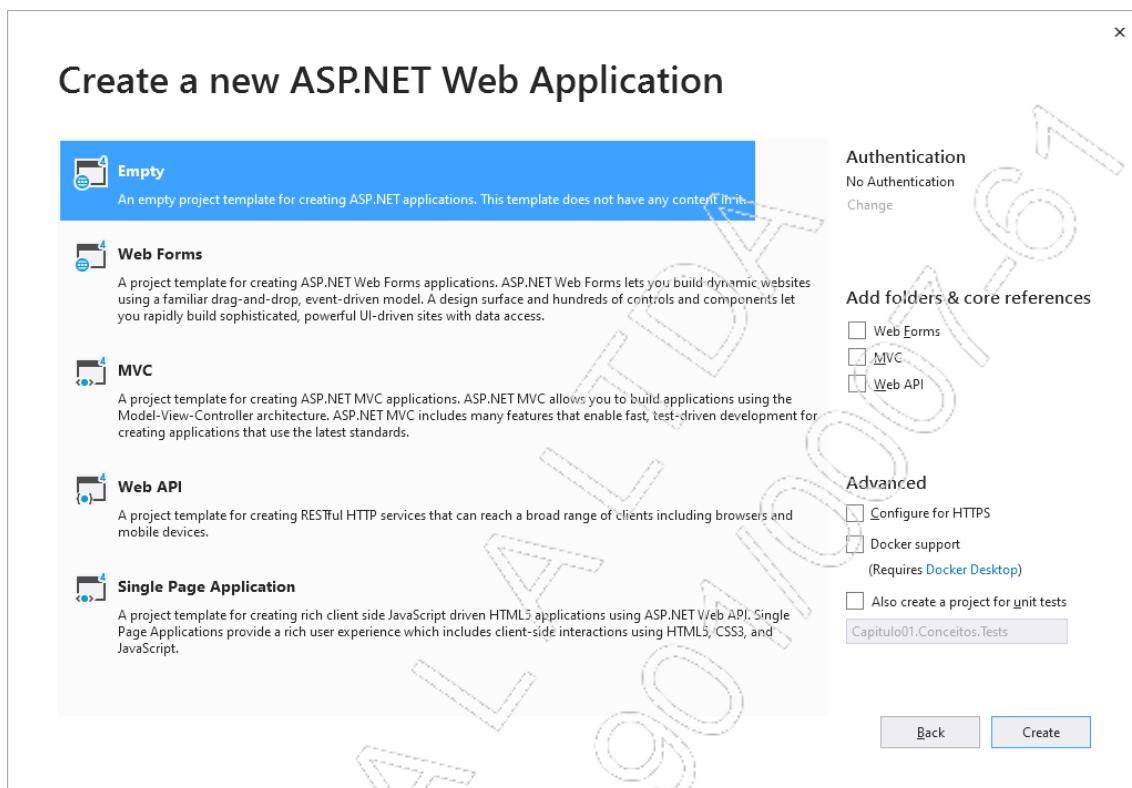
Para exemplificar, vamos criar um projeto **Asp.Net Web Application (.NET Framework)**:



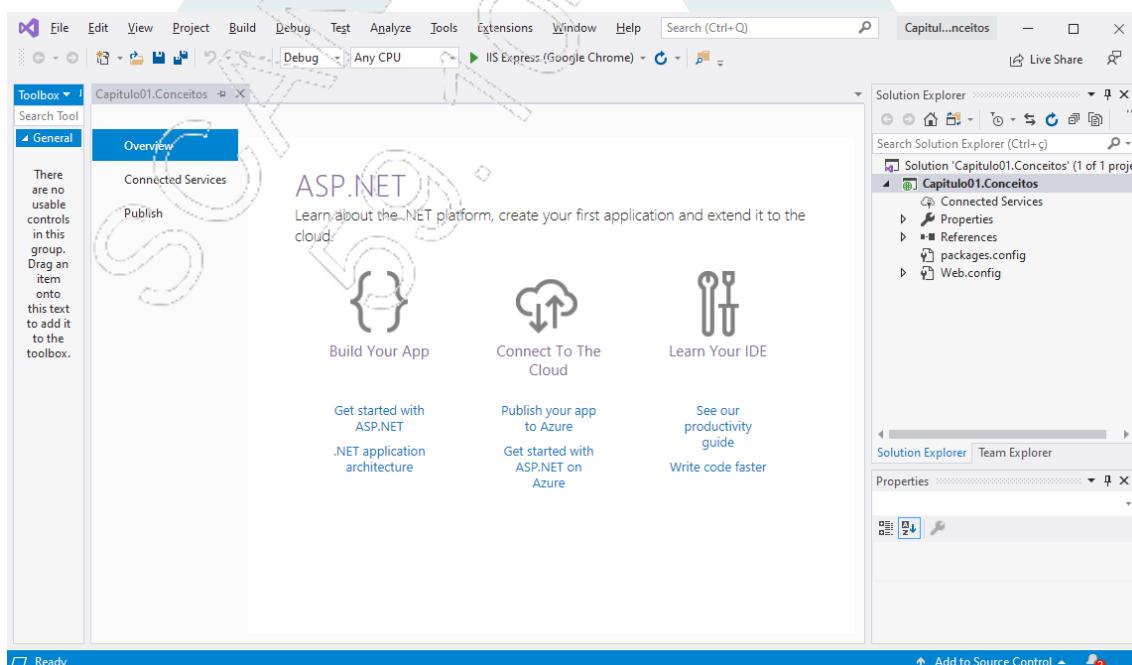
No próximo passo, definimos: o nome do projeto, a localização, o nome do Solution e a versão do .NET Framework:



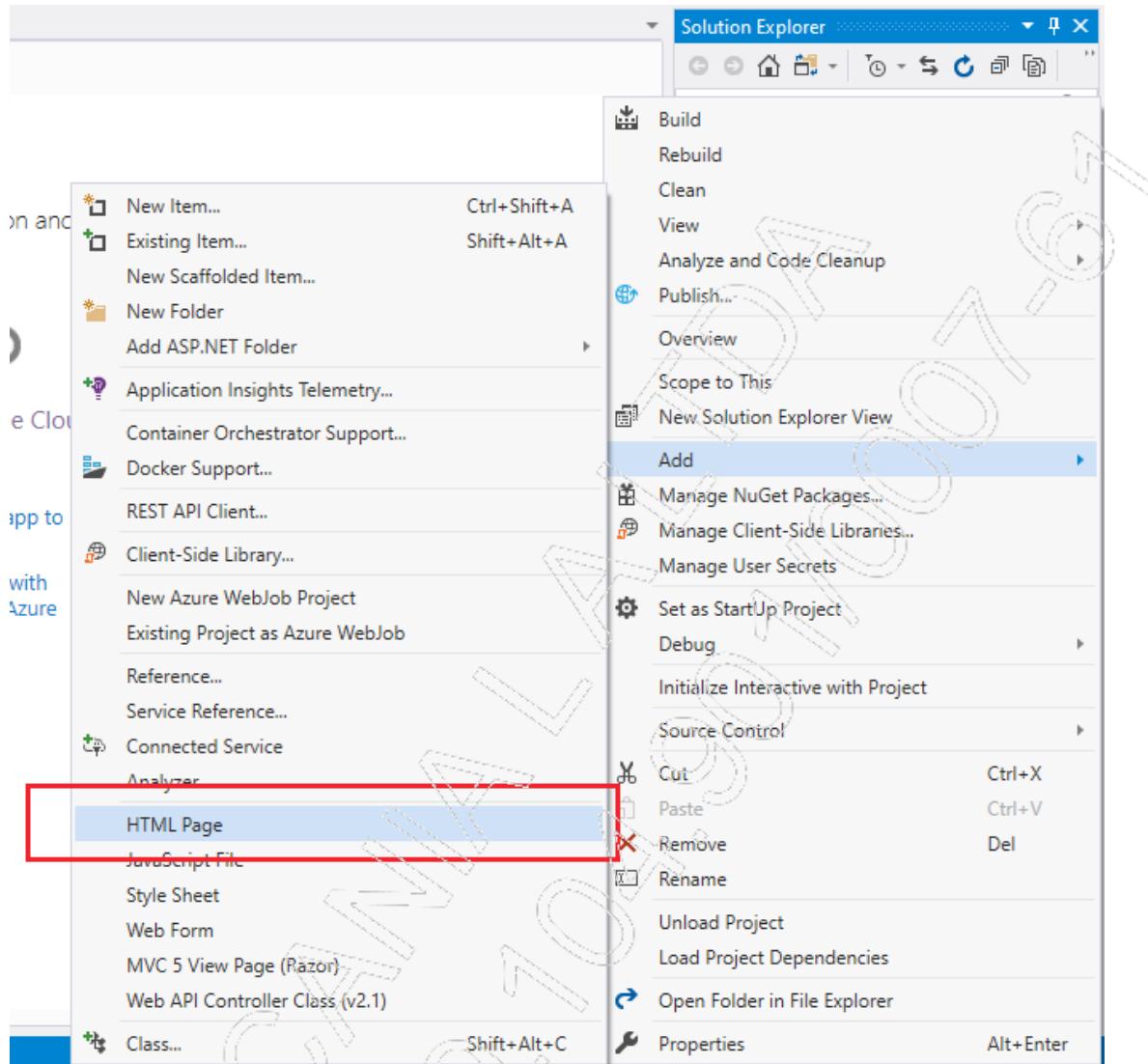
Na etapa final, selecionamos o template para o modelo de projeto que escolhemos. Neste caso, temos os modelos apresentados para a aplicação ASP.NET Web Application (.NET Framework) que escolhemos. Vamos definir um projeto vazio, deixando as demais opções como no exemplo:



Após alguns instantes, o projeto é criado. Temos o projeto com a seguinte estrutura:



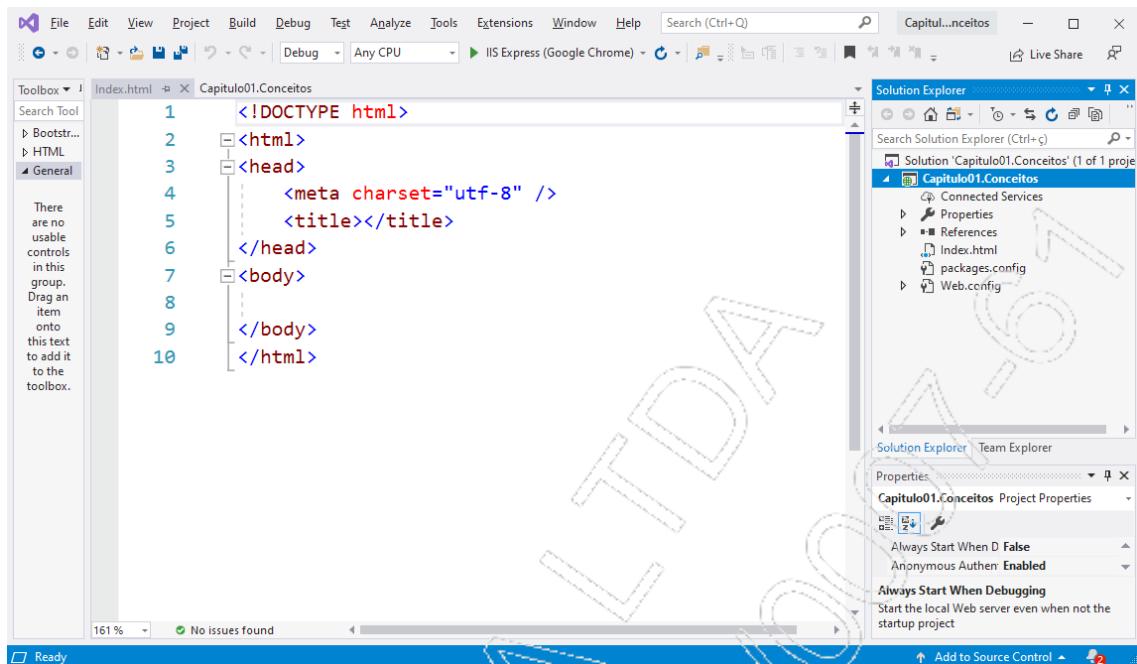
Para iniciarmos o desenvolvimento, vamos incluir uma página HTML ao projeto. Um dos caminhos para esse processo é clicando com o botão direito do mouse sobre o projeto e escolhendo adicionar uma página HTML (**HTML Page**):



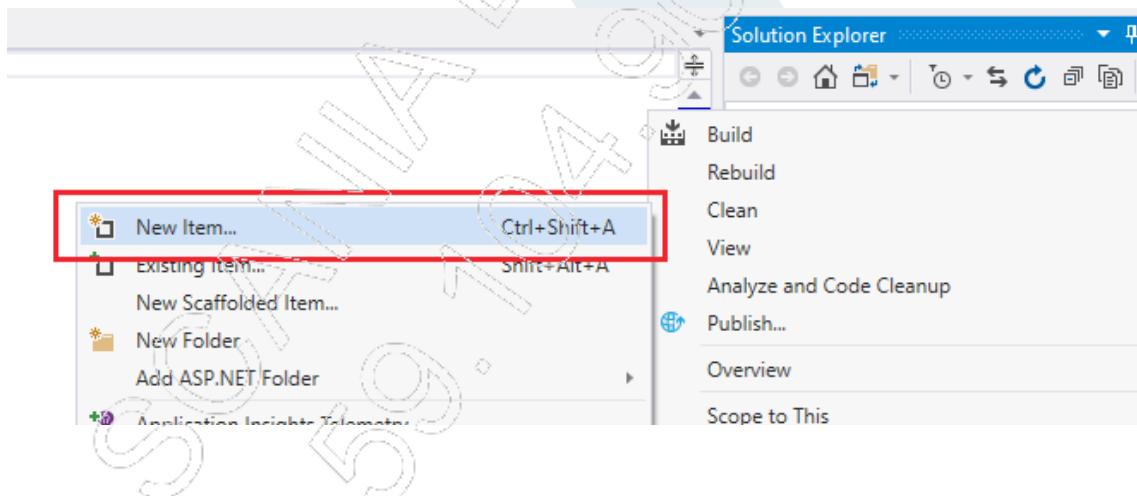
No campo a seguir, informe o nome do arquivo. Vamos chamá-lo de **Index.html**:



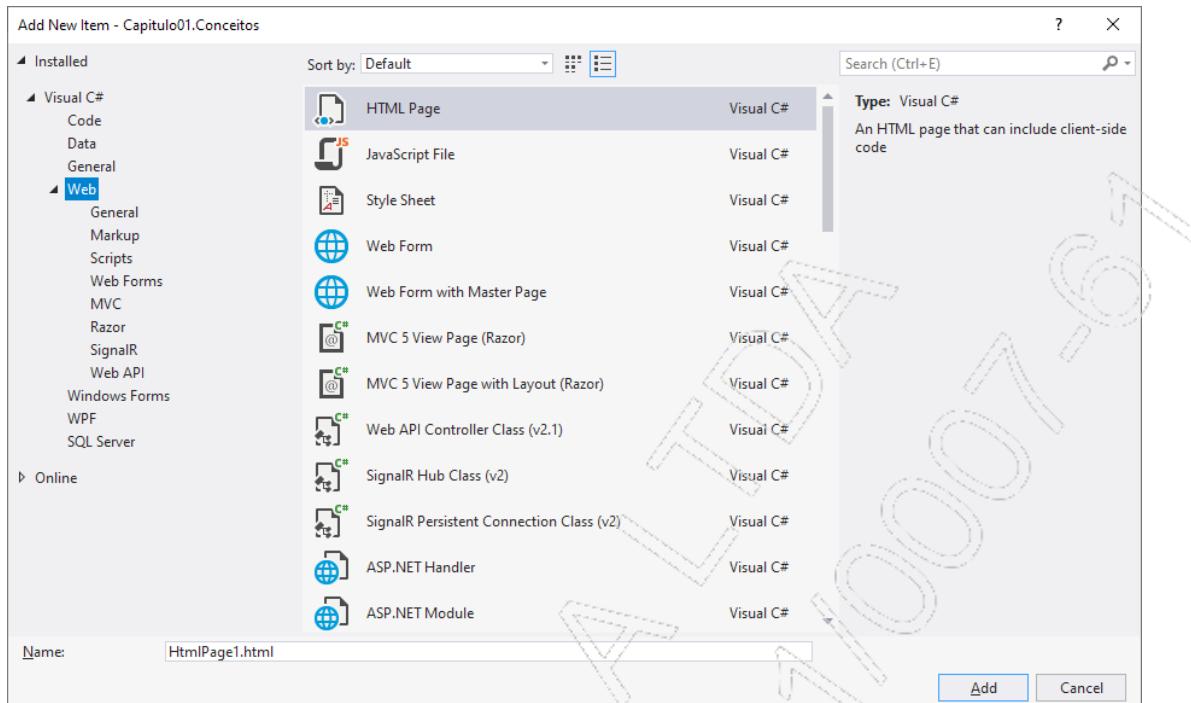
O arquivo **Index.html** é criado no projeto. Temos o seguinte ambiente:



Outro atalho que poderíamos usar é clicar com o botão direito do mouse sobre o projeto e selecionar a opção **New Item...**:



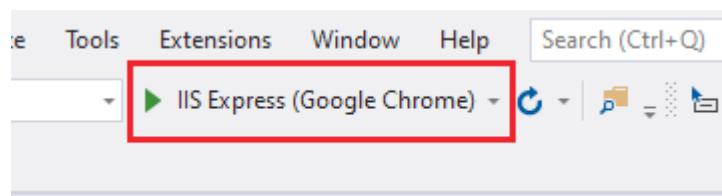
A vantagem dessa opção é que temos acesso a várias outras opções, incluindo a opção anterior (**HTML Page**):



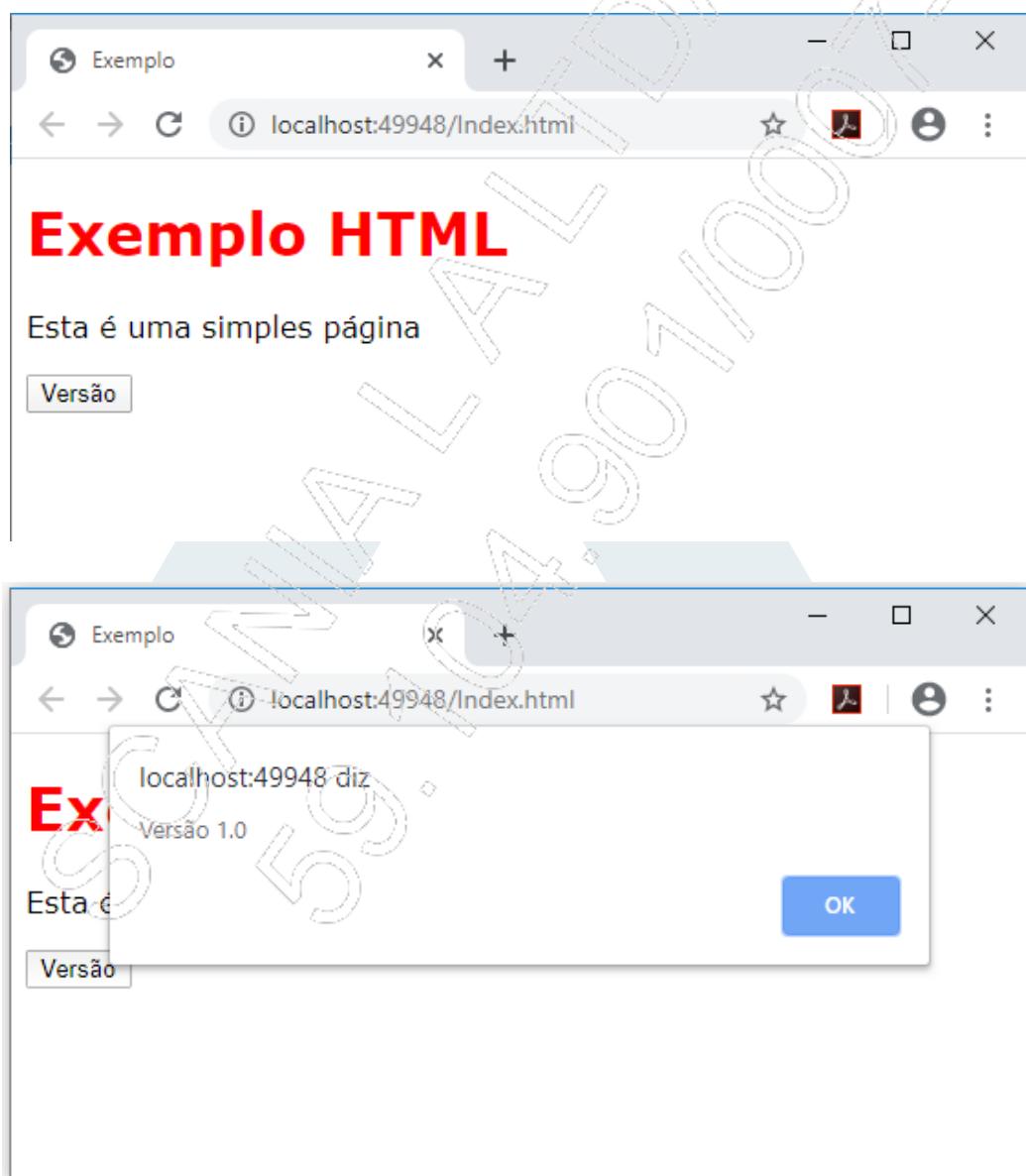
Inclua o seguinte conteúdo no arquivo **Index.html**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Exemplo</title>
    <style>
        body{ font-family: Verdana; }
        h1{ color:red; }
    </style>
    <script>
        function ExibirVersao() {
            alert("Versão 1,0");
        }
    </script>
</head>
<body>
    <h1>Exemplo HTML</h1>
    <p>Esta é uma simples página</p>
    <button type="button" onclick="ExibirVersao();">
        Versão
    </button>
</body>
</html>
```

Execute a aplicação, pressionando F5 ou clicando no botão adiante:



Esse processo ativa o IIS Express, que permitirá que a aplicação seja executada no browser selecionado. A seguir, temos o resultado da execução antes e depois do clique no botão:



Uma aplicação Web pode utilizar diversas arquiteturas de desenvolvimento e incluir diferentes componentes. Esses componentes serão apresentados ao longo deste curso.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

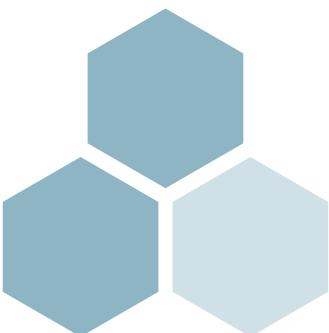
- A arquitetura cliente-servidor se baseia no conceito de **requisição** de um aplicativo **cliente** e **resposta** de um aplicativo **servidor**. O ambiente Web é um exemplo de arquitetura cliente-servidor;
- O **HTTP** (HyperText Transfer Protocol) consiste em um protocolo que define a comunicação entre os navegadores Web e os Websites;
- O navegador pode receber, inicialmente, três tipos de informação do servidor quando solicita uma página Web: o código **HTML**, que é responsável pela estrutura do documento; o código **CSS**, que determina a aparência da página; e o código **JavaScript**, que permite manipular eventos e elementos da página;
- **ASP.NET** é a parte da plataforma **.NET** responsável pela criação de aplicações Web e serviços Web;
- Nos sistemas operacionais da Microsoft, o **IIS** é o servidor mais utilizado para interceptar solicitações HTTP;
- Um projeto Web é um conjunto de arquivos que precisa ser compilado e publicado em um servidor.



1

Fundamentos de aplicações Web

Teste seus conhecimentos



1. Como se chamam a solicitação de uma aplicação cliente para o servidor e a resposta do servidor?

- a) HTTP e FTP.
- b) Response e Request.
- c) Request e Response.
- d) HTML e CSS.
- e) HTTP e IIS.

2. Qual é o comando (verb) frequentemente utilizado quando dados de um formulário são enviados ao servidor com o objetivo de alterar informações?

- a) GET
- b) DELETE
- c) ALT
- d) HEADER
- e) POST

3. Em qual pasta, por padrão, aplicações Web devem ser colocadas para que o IIS possa executá-las?

- a) C:\Program Files
- b) C:\Windows
- c) C:\wwwroot
- d) C:\inetpub
- e) C:\inetpub\wwwroot

4. Das tecnologias abaixo, qual não é executada pelo browser?

- a) CSS
- b) JavaScript
- c) HTML
- d) C#
- e) Imagem

5. Quando uma aplicação Web é executada pelo IIS no computador local, o objeto usado na URL é chamado de:

- a) localserver
- b) server
- c) localhost
- d) inetpub
- e) www



1

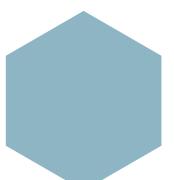
Fundamentos de aplicações Web



Mãos à obra!

Editora

IMPACTA

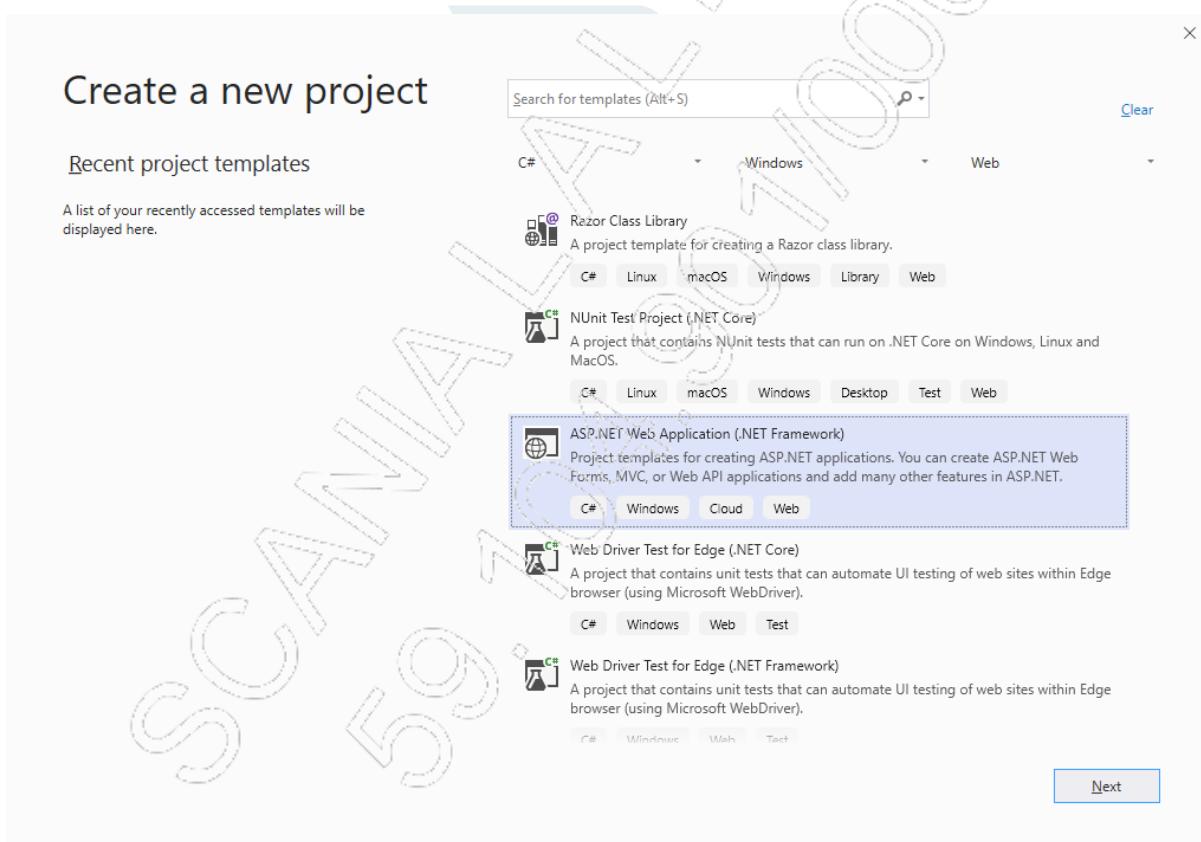


Laboratório 1

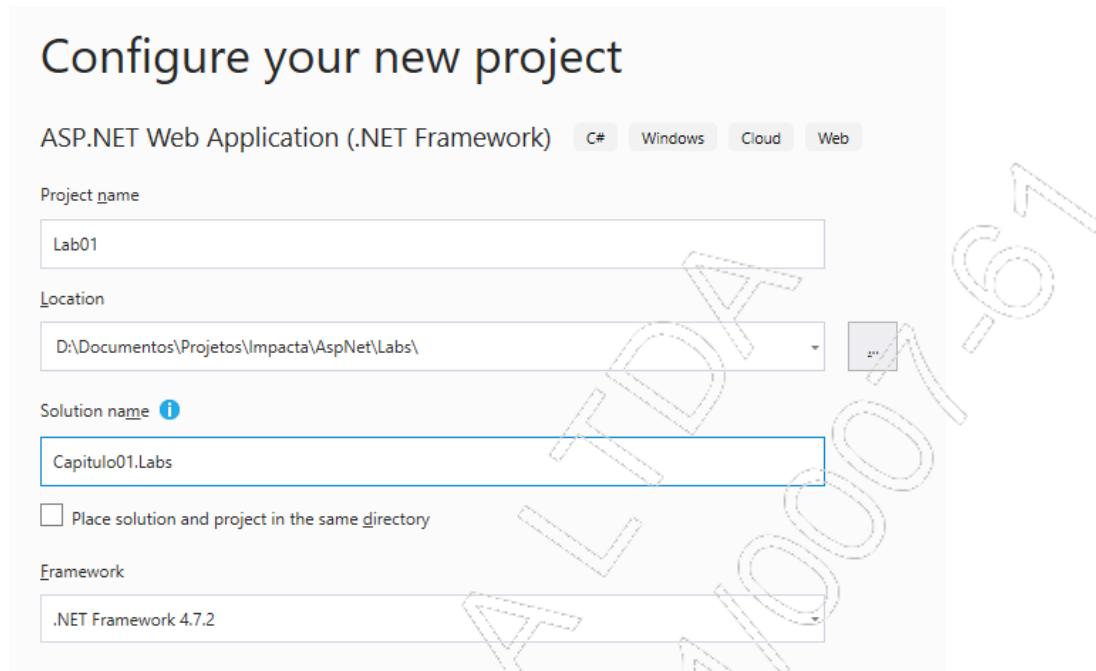
Objetivos:

O objetivo deste laboratório é demonstrar o processo de criação de uma aplicação Web e executá-la usando o servidor IIS.

1. Escolha uma pasta no seu computador. Nessa pasta, defina uma nova pasta chamada **Labs**;
2. Abra o **Visual Studio** e defina um projeto **ASP.NET Web Application (.NET Framework)** em branco, apontando para esta pasta:



3. Defina como nome do projeto **Lab01**. Como nome do solution, defina **Capítulo01.Labs**:



Neste e em laboratórios futuros, quando necessário, mostraremos partes da tela.

4. O projeto deve ser vazio:

The screenshot shows the 'Create a new ASP.NET Web Application' dialog. It lists five project templates: 'Empty', 'Web Forms', 'MVC', 'Web API', and 'Single Page Application'. The 'Empty' template is selected. On the right, there are sections for 'Authentication' (No Authentication, Change), 'Add folders & core references' (Web Forms, MVC, Web API), and 'Advanced' options (Configure for HTTPS, Docker support, Also create a project for unit tests).

5. Clique com o botão direito do mouse sobre o projeto **Lab01** e adicione uma nova página HTML. Coloque o nome de **Index.html**:

```
Index.html ✘ X
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8" />
5      <title></title>
6  </head>
7  <body>
8
9  </body>
10 </html>
```

6. Escreva o seguinte conteúdo no arquivo **Index.html**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Capítulo 01</title>
    <style>
        .margem{
            margin: 50px;
        }

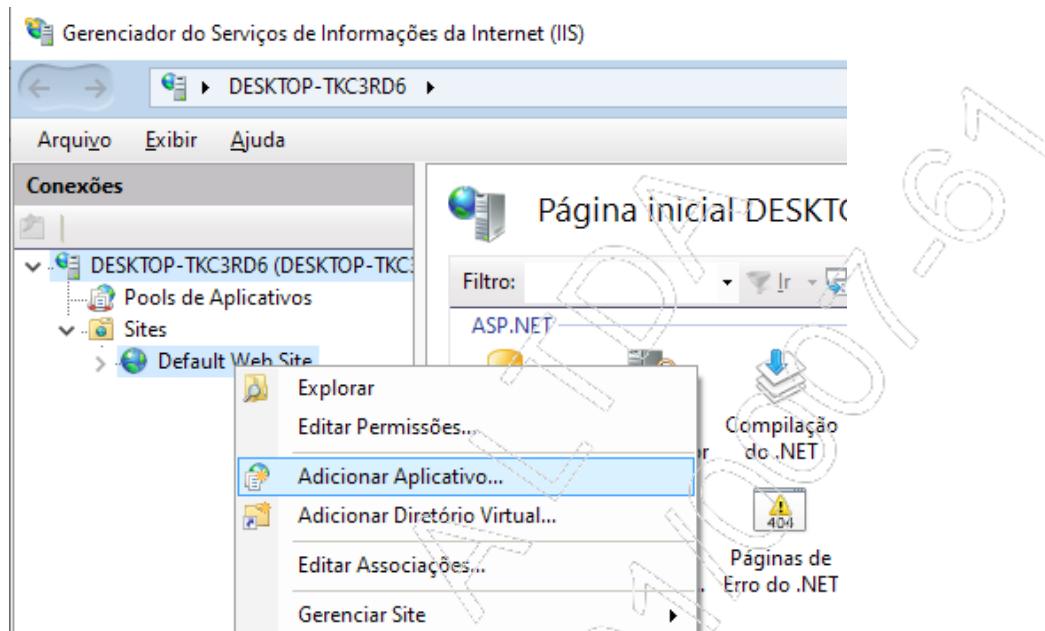
        .borda {
            border: 1px solid #a9a9a9;
            border-radius: 15px;
            padding: 20px;
            background-color: #d3d3d3;
        }

        .centraliza {
            text-align: center;
        }
    </style>
</head>
<body class="margem">
    <div class="borda centraliza">
        <h3>Bem vindo ao curso de Asp.Net</h3>
    </div>

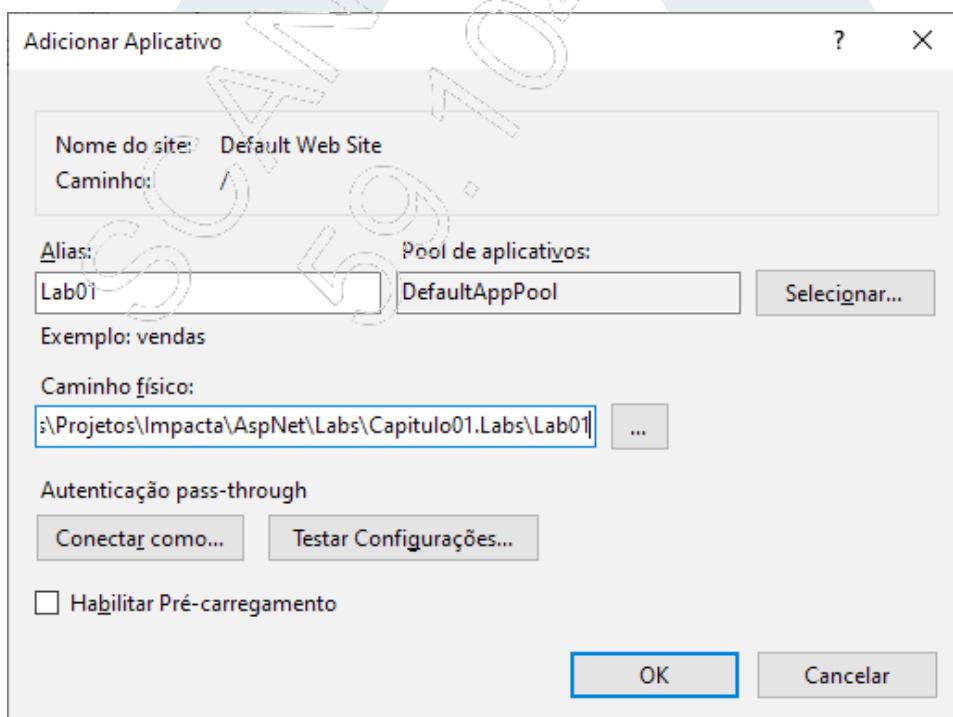
    <h4>Aproveite todo o conteúdo:</h4>
    <ul>
        <li>Asp.Net MVC</li>
        <li>Entity Framework</li>
        <li>Asp.Net Core</li>
        <li>e muito mais...</li>
    </ul>

</body>
</html>
```

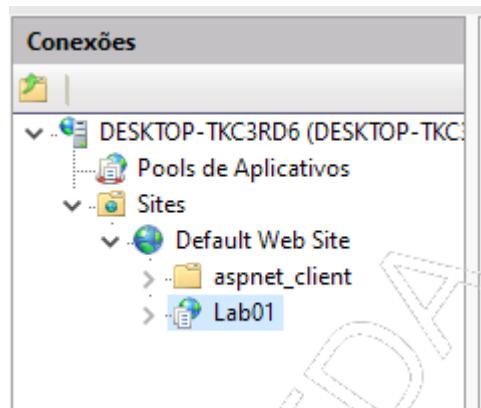
7. Abra o Painel de Controle, o grupo **Ferramentas Administrativas** e o **Gerenciador de Serviços de Informações da Internet (IIS)**. No painel **Conexões**, localize **Sites**, **Default Web Site**, abra o menu de contexto e escolha a opção **Adicionar Aplicativo...**:



8. Defina o nome virtual do aplicativo (**Alias**) como **Lab01**, o **Caminho físico (Physical path)** da pasta como sendo o local onde o arquivo **Index.html** está localizado:



9. O gerenciador do IIS deve ter criado uma pasta com o nome da aplicação Web;



10. Abra o navegador e digite o endereço <http://localhost/Lab01/> para visualizar a página.

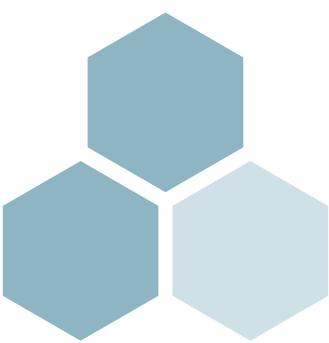




2

Web Pages com Razor

- ◆ O mecanismo Razor.



2.1. Introdução

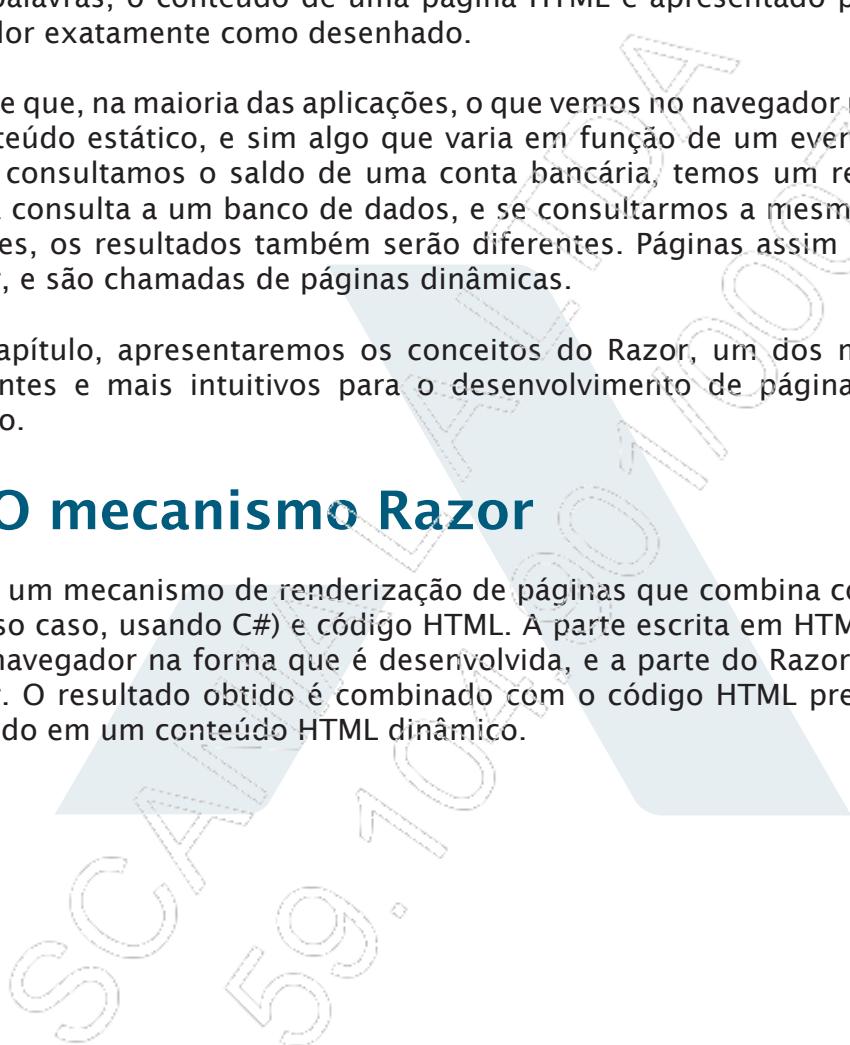
Em uma aplicação Web, somente páginas HTML não são suficientes. As páginas HTML são denominadas estáticas por não terem nenhuma interação com o servidor. Em outras palavras, o conteúdo de uma página HTML é apresentado para o usuário no navegador exatamente como desenhado.

Acontece que, na maioria das aplicações, o que vemos no navegador não é exatamente um conteúdo estático, e sim algo que varia em função de um evento. Por exemplo, quando consultamos o saldo de uma conta bancária, temos um resultado baseado em uma consulta a um banco de dados, e se consultarmos a mesma página em dias diferentes, os resultados também serão diferentes. Páginas assim interagem com o servidor, e são chamadas de páginas dinâmicas.

Neste capítulo, apresentaremos os conceitos do Razor, um dos mecanismos mais importantes e mais intuitivos para o desenvolvimento de páginas com conteúdo dinâmico.

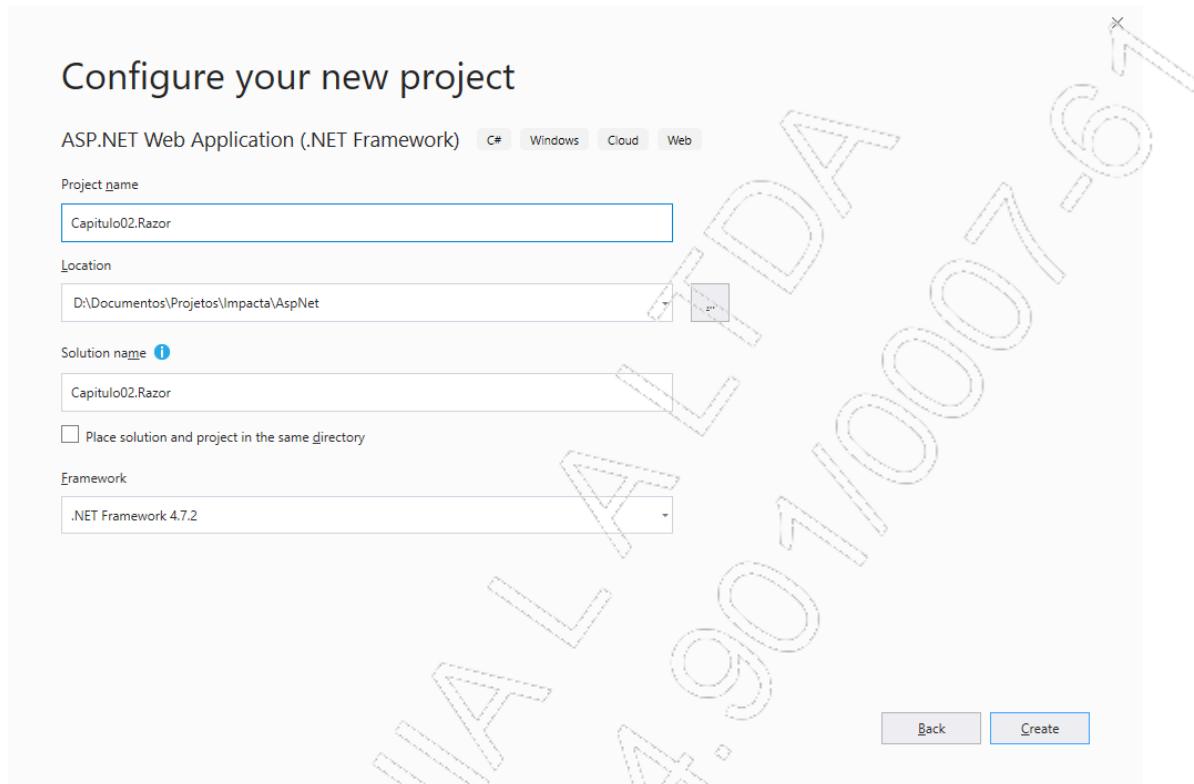
2.2. O mecanismo Razor

Razor é um mecanismo de renderização de páginas que combina código de servidor (no nosso caso, usando C#) e código HTML. A parte escrita em HTML puro é enviada para o navegador na forma que é desenvolvida, e a parte do Razor é processada no servidor. O resultado obtido é combinado com o código HTML presente na página, resultando em um conteúdo HTML dinâmico.

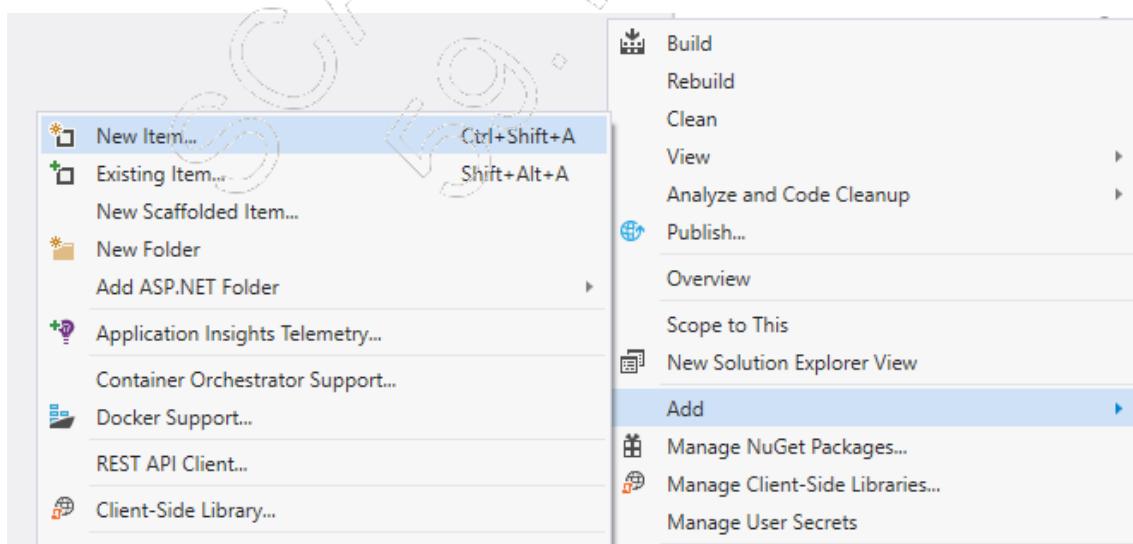


2.2.1. Criando um projeto com uma Web Page usando Razor

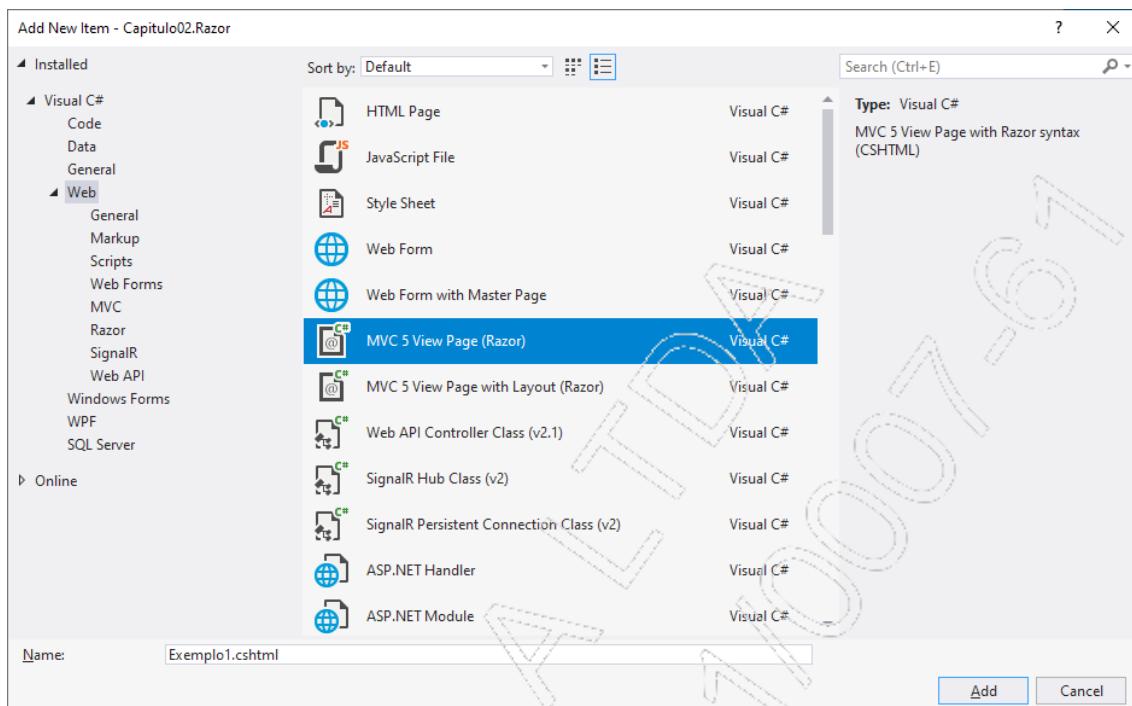
Para ilustrar o processo de desenvolvimento com Razor, vamos criar um projeto **ASP.NET Web Application (.NET Framework)** vazio chamado **Capítulo02.Razor**:



Uma vez que o projeto tenha sido criado, devemos clicar com o botão direito do mouse sobre ele e selecionar a opção **Add... / New Item...**:



Selecione a opção **MVC 5 View Page (Razor)**. Nomeie o arquivo como **Exemplo1.cshtml**:



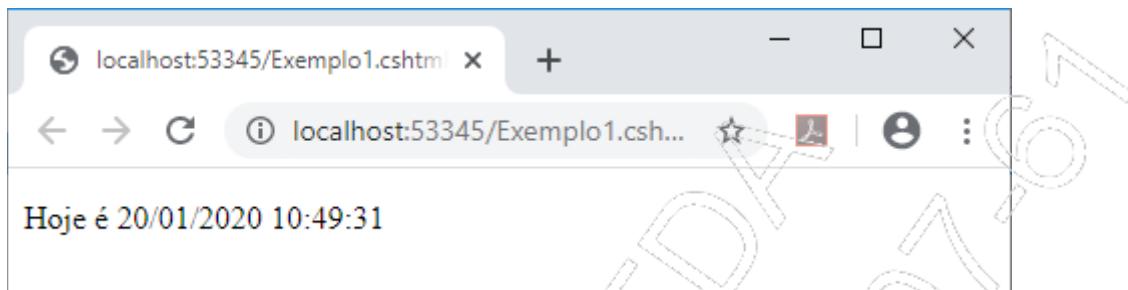
Usando C#, a extensão de uma Web Page com Razor é .cshtml.

Observe que o código gerado é semelhante a uma página HTML, com alguns itens adicionais. Elaboremos o seguinte exemplo (observe o destaque):

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
</head>  
<body>  
    <div>  
        <p>Hoje é @DateTime.Now</p>  
    </div>  
</body>  
</html>
```

O símbolo usado para abrir uma parte do script para ser executada no servidor é o caractere arroba (@). Não existe tag de fechamento. Quando um caractere que não deveria estar no local é encontrado, o mecanismo do Razor entende que terminou a expressão.

O resultado enviado para o navegador é semelhante a:



Na mesma Web Page, vamos adicionar outro exemplo. Agora, um exemplo de bloco de código usando chaves ({}):

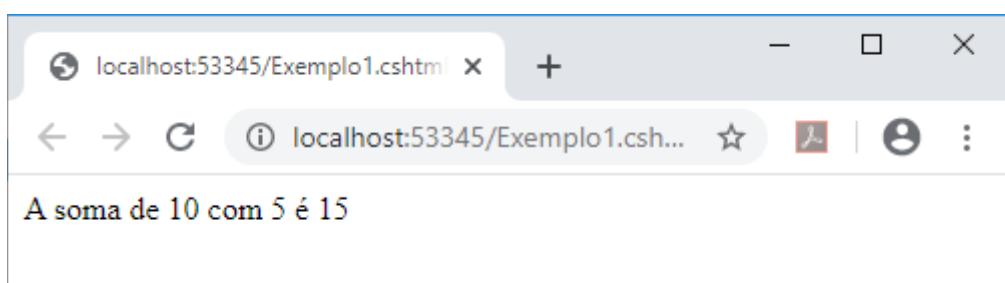
```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
</head>
<body>
    <div>
        @*<p>Hoje é @DateTime.Now</p>*@

        @{
            var x = 10;
            var y = 5;
        }

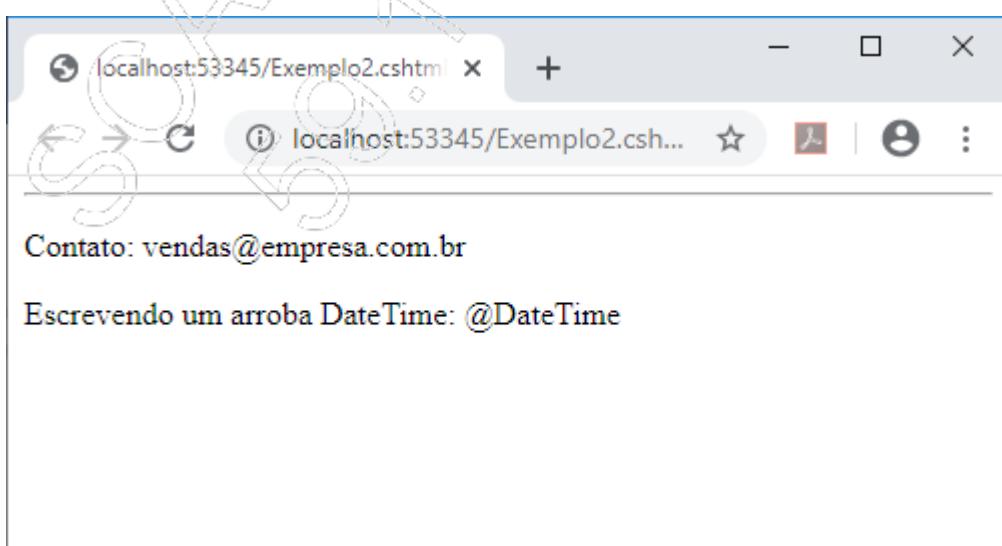
        A soma de @x com @y é @(x + y)
    </div>
</body>
</html>
```



Em alguns casos, o próprio mecanismo de interpretação do Razor percebe quando um sinal de arroba não é um comando, por exemplo, quando fizer parte de um e-mail. Esse recurso não é infalível, mas funciona em quase todos os casos. Se quiser, porém, escrever o sinal de arroba e, em seguida, uma palavra que seja um comando válido e não quiser que o Razor interprete como comando, basta inserir dois sinais de arroba.

O exemplo adiante mostra um sinal de arroba dentro de um e-mail, portanto, automaticamente não será interpretado como comando, e um conjunto de dois sinais de arroba que será escrito literalmente na página, também não sendo interpretado:

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
</head>  
<body>  
    <div>  
        <hr />  
        <p>  
            Contato: vendas@empresa.com.br  
        </p>  
  
        Escrevendo um arroba DateTime: @@DateTime  
    </div>  
</body>  
</html>
```



2.2.2. Estruturas de repetição

Razor não é uma linguagem de programação e sim uma View Engine que utiliza qualquer linguagem, contanto que exista um plugin disponível. Usando C#, a sintaxe para criar loops é a seguinte:

- Loop **for**

```
@for (int i = 1; i <= 10; i++)  
{  
    <p>  
        O valor de i é  
        <span>@i</span>  
    </p>  
}
```

- Loop **foreach**

```
@{  
    var numeros = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
    foreach (int i in numeros)  
    {  
        <p>  
            O valor de i é  
            <span>@i</span>  
        </p>  
    }  
}
```

- Loop **while**

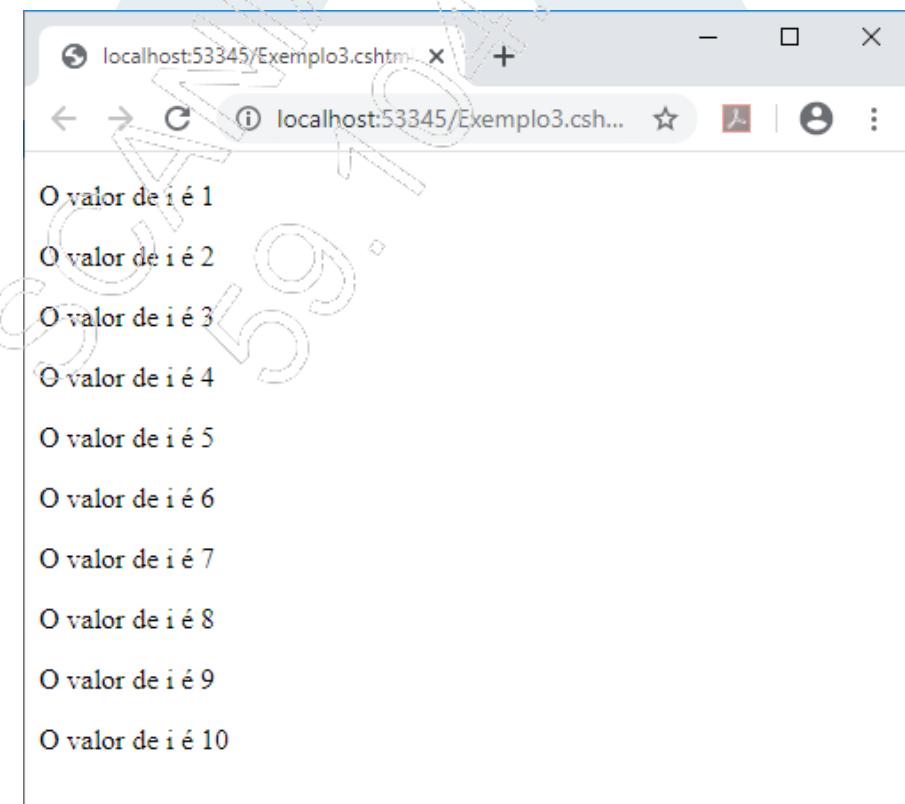
```
@{ int i = 1;  
    while (i <= 10)  
    {  
        <p>  
            O valor de i é  
            <span>@i</span>  
        </p>  
        i = i + 1;  
    }  
}
```

Qualquer um desses códigos incluídos em uma Web Page produzirá no browser o mesmo resultado. Como exemplo, incluiremos o código usando a estrutura **while**:

```
@{ Layout = null; }

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
</head>
<body>
    <div>
        @{ int i = 1;
            while (i <= 10)
            {
                <p>
                    O valor de i é
                    <span>@i</span>
                </p>
                i = i + 1;
            }
        </div>
    </body>
</html>
```



É interessante notar que não existe uma regra para terminar um trecho de código e iniciar um em HTML. O Razor é inteligente o suficiente para deduzir quando é um código de servidor e quando é HTML. No exemplo anterior, o fato de existir um parágrafo (`<p>`) no meio do código indica que é HTML e, ao encontrar chaves (`{ }`) ou uma expressão, ele retorna para o código. É claro que, às vezes, é necessário intervir nessa interpretação, como o caractere arroba (@) colocado para exibir a variável `i`, já que a sua chamada ocorre dentro de elementos HTML. Se não tivéssemos colocado, a saída seria 10 vezes a letra "i".

2.2.3. Estruturas condicionais

As estruturas condicionais, quando usadas, devem ser precedidas pelo caractere arroba (@) e seu conteúdo, em um par de chaves (`{ }`).

Vejamos, a seguir, suas estruturas condicionais:

- Estrutura `if`

```
@if (DateTime.Now.DayOfWeek == DayOfWeek.Friday)
{
    <span>
        Hoje é sexta-feira.
        Agora são @DateTime.Now.ToShortTimeString()
    </span>
}
else
{
    <span>
        Hoje não é sexta
    </span>
}
```

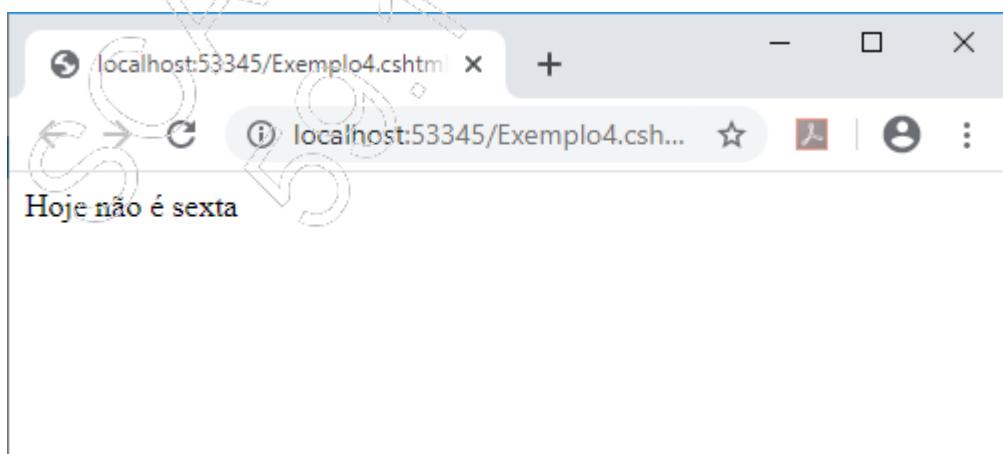
Observe que o comando `else` não foi precedido por arroba (@) porque ele faz parte do comando `if`.

- Estrutura `switch`

```
@switch (DateTime.Now.Hour)
{
    case 12:
    case 13:
        <p>Hora do Almoço</p>
        break;
    case 18:
    case 19:
        <p>Hora do Jantar</p>
        break;
}
```

Vamos apresentar o exemplo usando o comando **if**:

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
</head>  
<body>  
    <div>  
        @if (DateTime.Now.DayOfWeek == DayOfWeek.Friday)  
        {  
            <span>  
                Hoje é sexta-feira.  
                Agora são @DateTime.Now.ToShortTimeString()  
            </span>  
        }  
        else  
        {  
            <span>  
                Hoje não é sexta  
            </span>  
        }  
    </div>  
</body>  
</html>
```



2.2.4. Render e layout

Normalmente, em uma aplicação Web, grande parte do código HTML é compartilhada entre todas as páginas: cabeçalho, menus, rodapés. Esses elementos podem ser isolados em páginas separadas e inseridos em outra página por meio do método `RenderPage()`.

2.2.4.1. RenderPage()

O método `RenderPage()` insere um arquivo externo na posição do comando.

- Arquivo: **Exemplo.cshtml**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo</title>
  </head>
  <body>

    @RenderPage("Titulo.cshtml")

    <p>Bem-vindo ao site</p>

    @RenderPage("Rodape.cshtml")

  </body>
</html>
```

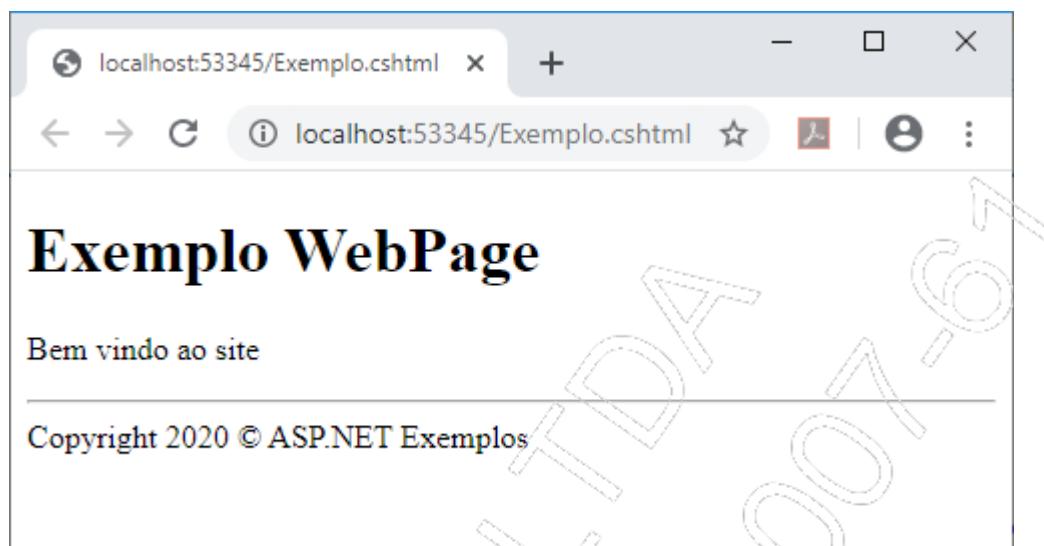
- Arquivo: **Titulo.cshtml**

```
<h1>Exemplo WebPage</h1>
```

- Arquivo: **Rodape.cshtml**

```
<hr/>
Copyright @DateTime.Now.Year © ASP.NET Exemplos
```

Ao executar a página, o Razor View Engine monta os três arquivos em um só, que é renderizado para ser enviado ao browser. Veja:



Olhando o código-fonte que foi enviado ao navegador, é possível perceber que não foi inserido nada além do conteúdo dos arquivos. Veja:

A screenshot of a Microsoft Edge browser window. The address bar shows 'view-source:localhost:53345/Exemplo.cshtml'. The main content area displays the raw HTML code of the page, including the DOCTYPE declaration, HTML structure, head section with meta viewport and title tags, body section with a heading, a paragraph, a horizontal line, and the copyright notice. The code is numbered from 1 to 23.

O método **RenderPage()** não é a melhor opção para garantir um layout consistente em um aplicativo Web com muitas páginas. Em todas as páginas do aplicativo, seria necessário incluir uma chamada ao método para renderizar o título e outra para renderizar o rodapé. Se a estrutura padrão mudasse, por exemplo, se fosse incluído um navegador, todas as páginas teriam que incluir uma renderização para o arquivo do menu.

O conceito de layout resolve o problema de padronização em Web Pages.

2.2.4.2. RenderBody()

O método **RenderBody()** renderiza uma página configurada como página de conteúdo dentro de uma página configurada como página de layout.

Para ser uma página que sirva de modelo para as outras, não tem nenhuma configuração especial, basta ter uma chamada ao método **RenderBody()**. A página que será renderizada deve indicar esta página de layout como primeira declaração.

- Página modelo: **_Layout.cshtml**

```
<!DOCTYPE html>
<html>
    <head><title>Minha App</title></head>
    <body>
        <h1>Tituto Minha App</h1>

        @RenderBody()

        <hr/>
        <p>Desenvolvido para o curso de ASP.NET&copy; @DateTime.Now.
Year </p>
    </body>
</html>
```

- Página que utiliza o modelo: **Conteudo.cshtml**

```
@{
    Layout = "~/__Layout.cshtml";
}
<p>
Aqui é o corpo da página que será renderizado
pelo método RenderBody() da página de Layout...
</p>
```

Não é obrigatório utilizar o arquivo iniciando com o caractere sublinhado (_), mas é uma prática comum, porque os arquivos que iniciam dessa forma não podem ser visualizados pelo usuário, e o arquivo que serve de modelo geralmente é usado apenas para esse propósito, portanto não teria sentido isolá-lo.



2.2.4.3. RenderSection()

O método **RenderSection** renderiza o conteúdo de uma seção nomeada. Este método deve estar dentro de um arquivo de layout. O primeiro parâmetro deste método é o nome da área nomeada, e o segundo é se essa área é opcional. Se no segundo parâmetro for marcado **false**, a página não precisará ter uma área nomeada. Alteramos o arquivo de layout para incluir o método **RenderSection**:

- Página de layout alterada: `_Layout.cshtml`

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Minha App</title>
</head>
<body>
    <div>
        <h1>Tituto Minha App</h1>

        @RenderBody()

        <p>Mais texto do arquivo de Layout....</p>

        @RenderSection("Mensagem", false)

        <hr />
        <p>Desenvolvido para o curso de ASP.NET&copy; @DateTime.Now.Year </p>

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

- Página que usa o layout, com o conteúdo para a section: `Conteudo2.cshtml`

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

<p>Esta é a página principal (que usa o layout)</p>

@section Mensagem{
    <p>Esta é a área "mensagem" da página principal</p>
}
```

- Veja o resultado ao ser renderizado:



O fato de podermos definir se a renderização é obrigatória ou não é muito importante, porque, se fosse sempre obrigatório, todas as páginas teriam de ter uma seção nomeada. Dessa forma, se a página não tem uma seção com o nome definido, o método não é executado e não provoca nenhum erro.

2.2.5. Request e Form

Grande parte do sucesso do protocolo HTTP e da linguagem HTML é a versatilidade na maneira de transmitir e receber informações pela Internet. As Web Pages disponibilizam propriedades para manipular e/ou consultar informações vindas do navegador.

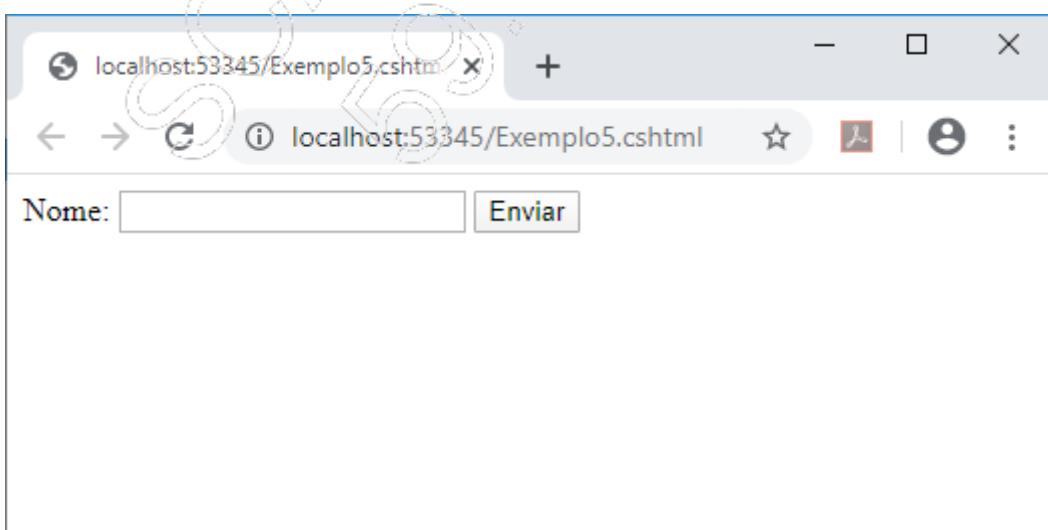
- **IsPost e Request**

IsPost é a propriedade herdada da classe `WebPage` que retorna se a requisição for de um formulário com o método **POST**.

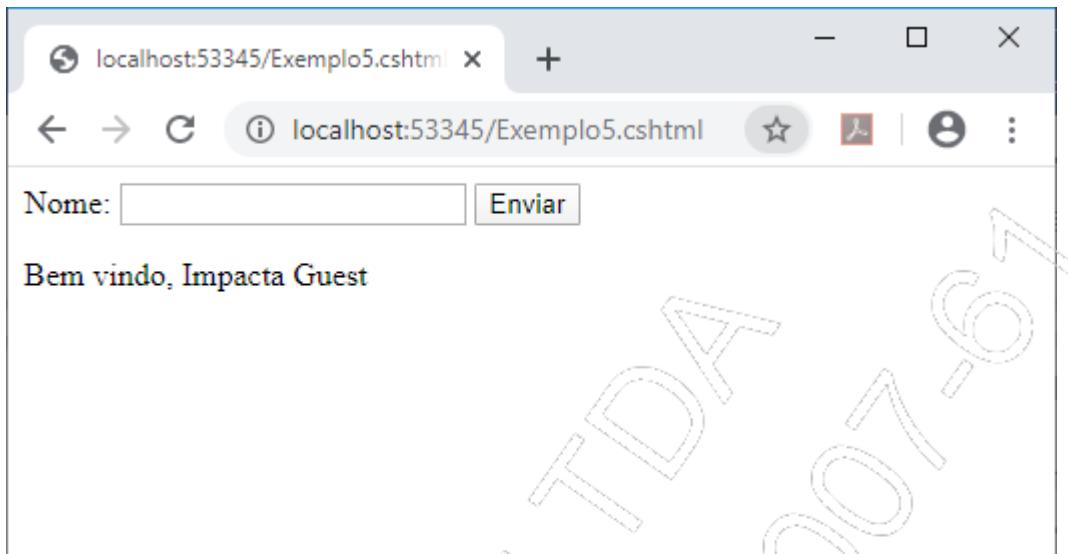
A propriedade **Request** retorna os detalhes da requisição, e os dados do formulário podem ser obtidos por meio da propriedade **Request.Form**.

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
</head>  
<body>  
    <div>  
        <form action="" method="post">  
            <label>Nome:</label>  
            <input name="nome" />  
            <button type="submit">Enviar</button>  
        </form>  
  
        @if (IsPost)  
        {  
            var nome = Request.Form["nome"];  
            <p>Bem vindo, @nome</p>  
        }  
    </div>  
</body>  
</html>
```

Veja o resultado da execução desta página, antes de clicarmos no botão:



Após fornecer um nome no campo de entrada e clicar no botão, teremos:



No exemplo, fornecemos o nome **Impacta Guest** como dado de entrada.

Os dados obtidos pela coleção **Form** podem ser obtidos pela coleção indexadora da classe **Request**:

```
var nome=Request.Form["nome"];
```

Pode ser:

```
var nome=Request["nome"];
```

Quando é usada a propriedade indexadora, o item **nome** não é procurado apenas no **Form**, mas também no **QueryString** e **Cookies**. É importante ter cuidado com essa característica, principalmente quando existe a alteração de dados.

Procurar em todas as coleções torna o programa mais vulnerável, pois fica muito fácil passar um parâmetro errado pela **QueryString**, por exemplo. Para alterações, o melhor é utilizar **POST** e **Request.Form**.

- `IsInt()`, `IsDate()`, `IsBool()`, `IsDecimal()`, `IsFloat()`

Estes métodos permitem verificar se o tipo dentro de uma requisição pode ser convertido em um tipo primitivo como `int`, `DateTime`, `Decimal` ou `Boolean`. Podem ser usados em conjunto com o formulário, para validações:

```
<form action="" method="post">

    <label for="numero">Digite um Número</label>
    <input type="text" id="numero" name="numero" />

    <label for="data">Digite uma Data</label>
    <input type="text" id="data" name="data" />

    <br />
    <button type="submit">Enviar</button>

    @if (IsPost)
    {
        if (Request["Numero"].IsInt())
        {
            <p>Número OK</p>
        }
        else
        {
            <p>Número Inválido</p>
        }

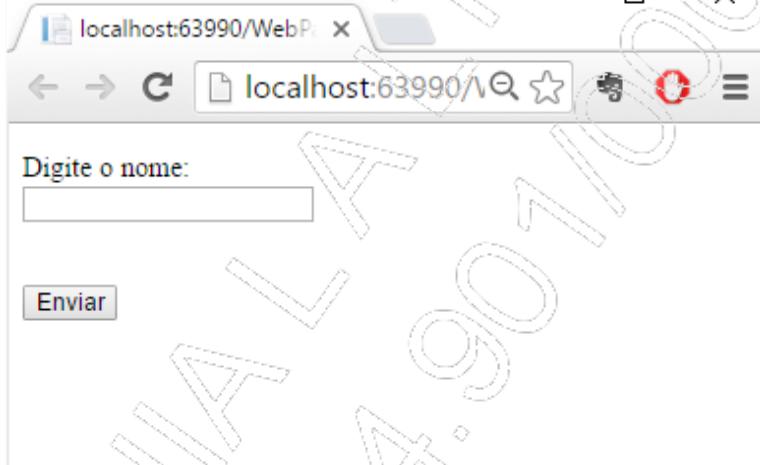
        if (Request["Data"].IsDateTime())
        {
            <p>Data OK</p>
        }
        else
        {
            <p>Data Inválida!</p>
        }
    }
</form>
```

2.2.6. HTML Helpers

Helpers são pequenos métodos que geram código HTML. Isso facilita a construção de uma página HTML, principalmente quando há muito código com a mesma estrutura. Por exemplo, um formulário sempre vai ser constituído de diversos controles **Label** seguidos de controles **Input**. Essas tags HTML podem ser criadas automaticamente por meio dos métodos fornecidos pelo Helper. Veja o exemplo, usando o HTML:

```
@Html.Label("Digite o nome:", "nome") <br />
@Html.TextBox("nome") <br />
```

O resultado da página renderizada é este:



O código HTML gerado pelo servidor é este:

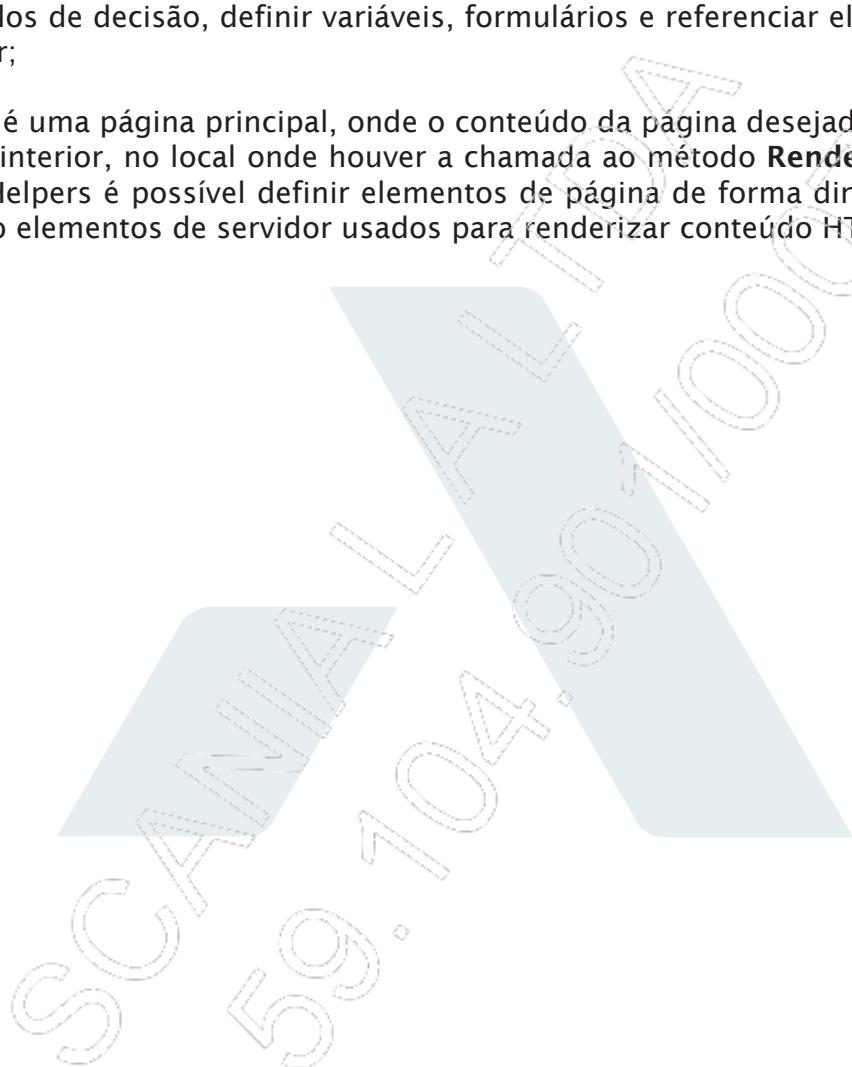
```
<label for="nome">Digite o nome:</label> <br />
<input id="nome" name="nome" type="text" value="" /> <br />
```

Os seguintes métodos geram código HTML: **CheckBox**, **DropDownList**, **Hidden**, **ListBox**, **Password**, **RadioButton**, **TextArea** e **ValidationMessage**, **ValidationSummary**. Teremos oportunidade de usarmos esses controles quando estudarmos o MVC.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- O **Razor** combina conteúdo HTML com código executado no servidor. A renderização da página é o resultado da execução do código no servidor com o código HTML fornecido; Com Razor é possível criar estruturas de repetição, comandos de decisão, definir variáveis, formulários e referenciar elementos do servidor;
- **Layout** é uma página principal, onde o conteúdo da página desejada é inserido no seu interior, no local onde houver a chamada ao método **RenderBody**; Com HTML Helpers é possível definir elementos de página de forma dinâmica, pois eles são elementos de servidor usados para renderizar conteúdo HTML.

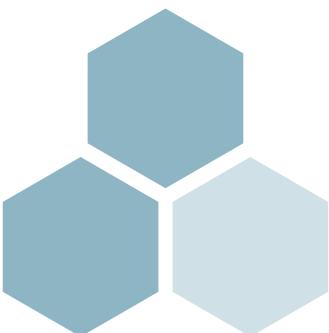




2

Web Pages com Razor

Teste seus conhecimentos



1. Qual será o resultado do comando Razor adiante?

A soma de 10 + 5 é @10+5

- a) A soma de 10 + 5 é 10+5.
- b) A soma de 10 + 5 é 15.
- c) Vai ocorrer um erro.
- d) A soma de 10 + 5 é 105.
- e) A soma de 10 + 5 é @15.

2. Qual método é usado para inserir uma página dentro de outra?

- a) InsertPage
- b) IncludePage
- c) AddPage
- d) RenderPage
- e) GetPage

3. Para que o comando RenderBody funcione, qual comando deve estar na página que será renderizada?

- a) RenderMe
- b) DOCTYPE
- c) Layout
- d) FlowLayout
- e) _Layout

4. O que faz o método RenderSection?

- a) Inclui uma variável de sessão do servidor no cliente.
- b) Substitui o layout por outra página.
- c) Insere um trecho de código no local onde o método é chamado.
- d) Desabilita o Razor.
- e) Permite que um código JavaScript seja inserido no servidor.

5. Qual dos métodos abaixo NÃO faz parte do Razor?

- a) IsDate
- b) IsModel
- c) IsPost
- d) IsDecimal
- e) IsFloat



2

Web Pages com Razor



Mãos à obra!



Objetivos:

Nestes laboratórios, apresentaremos os conceitos do Razor, com o propósito de iniciarmos o desenvolvimento de um projeto, a partir do próximo capítulo.

Laboratório 1

1. No Visual Studio, crie um novo projeto chamado **Lab01**, em um solution chamado **Capítulo02.Labs**. O projeto deve ser do tipo **ASP.NET Web Application**, vazio:

Configure your new project

ASP.NET Web Application (.NET Framework) C# Windows Cloud Web

Project name

Lab01

Location

D:\Documentos\Projetos\Impacta\AspNet\Labs

Solution

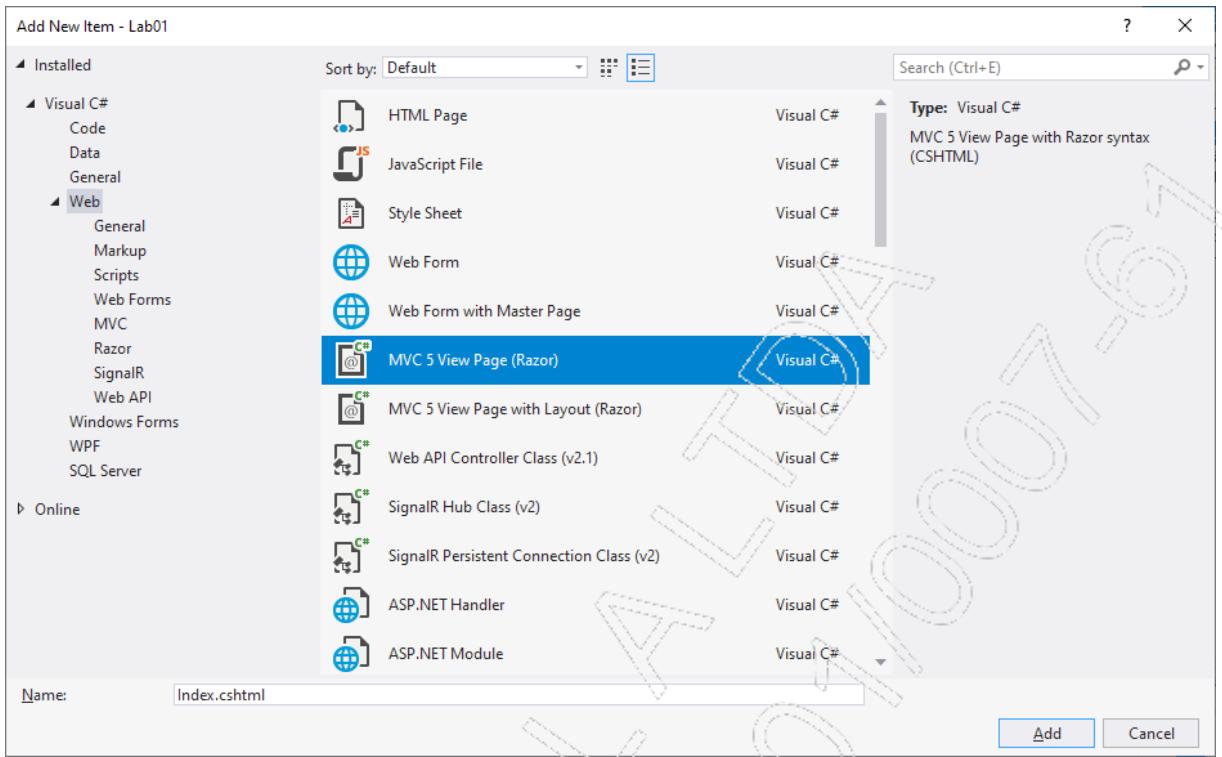
Create new solution

Solution name 

Capítulo02.Labs

Place solution and project in the same directory

2. Com o projeto criado, clique com o botão direito do mouse e adicione, como novo item, a opção **MVC 5 View Page (Razor)**. Coloque o nome de **Index**:



3. No código HTML gerado, insira o título e a informação de data e hora:

```

@{
    Layout = null;
}

<!DOCTYPE html>

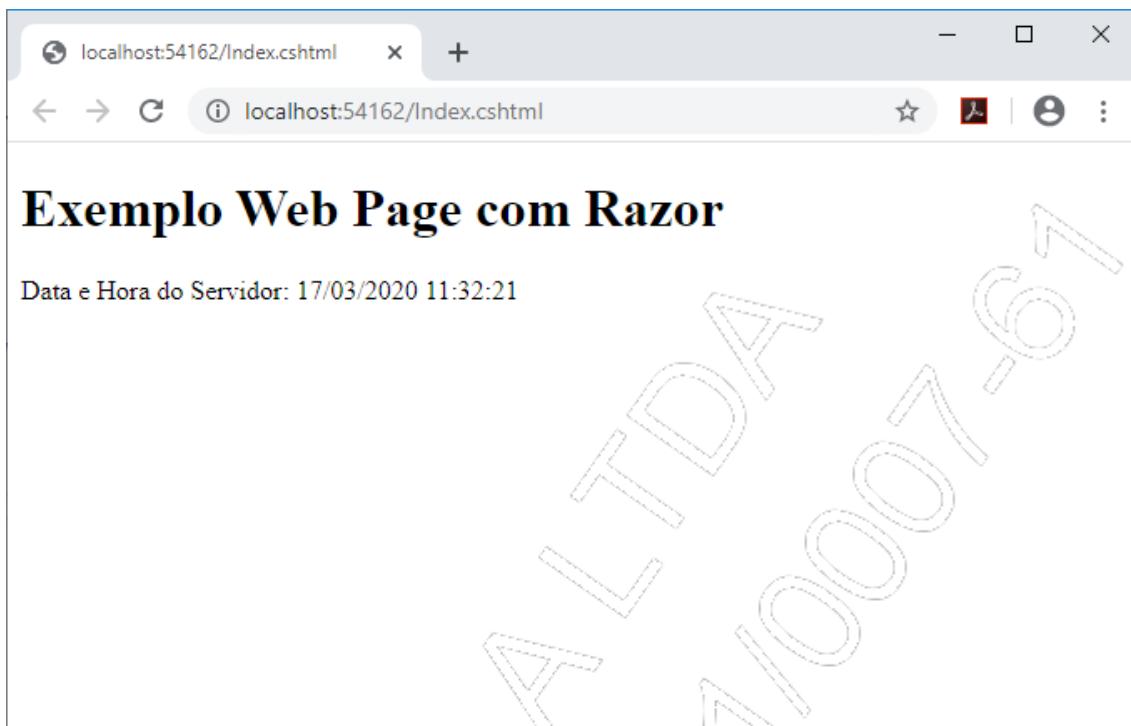
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
</head>
<body>
    <div>

        <h1>Exemplo Web Page com Razor</h1>
        Data e Hora do Servidor: @DateTime.Now

    </div>
</body>
</html>

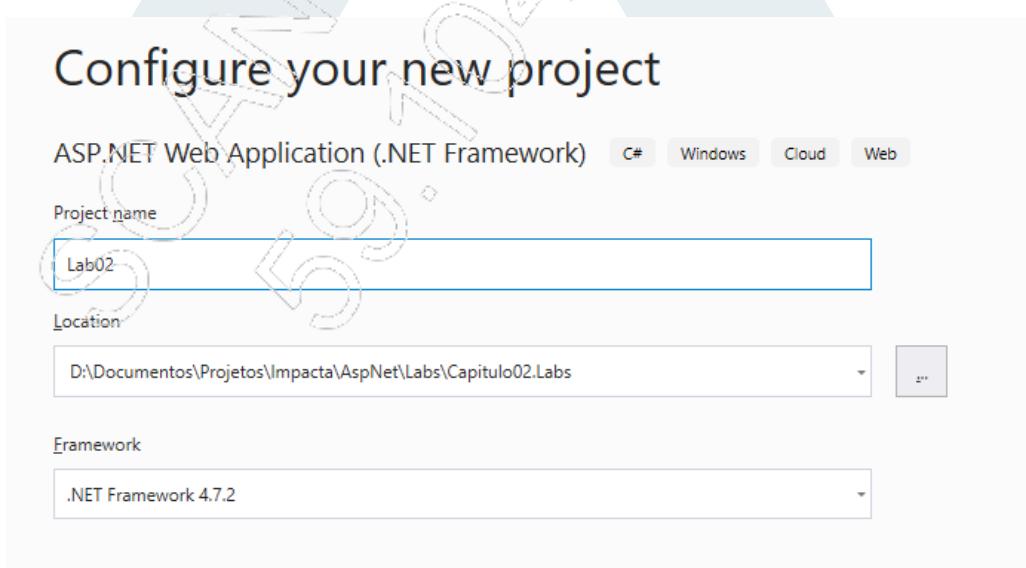
```

4. Teste o código pressionando CTRL + F5.



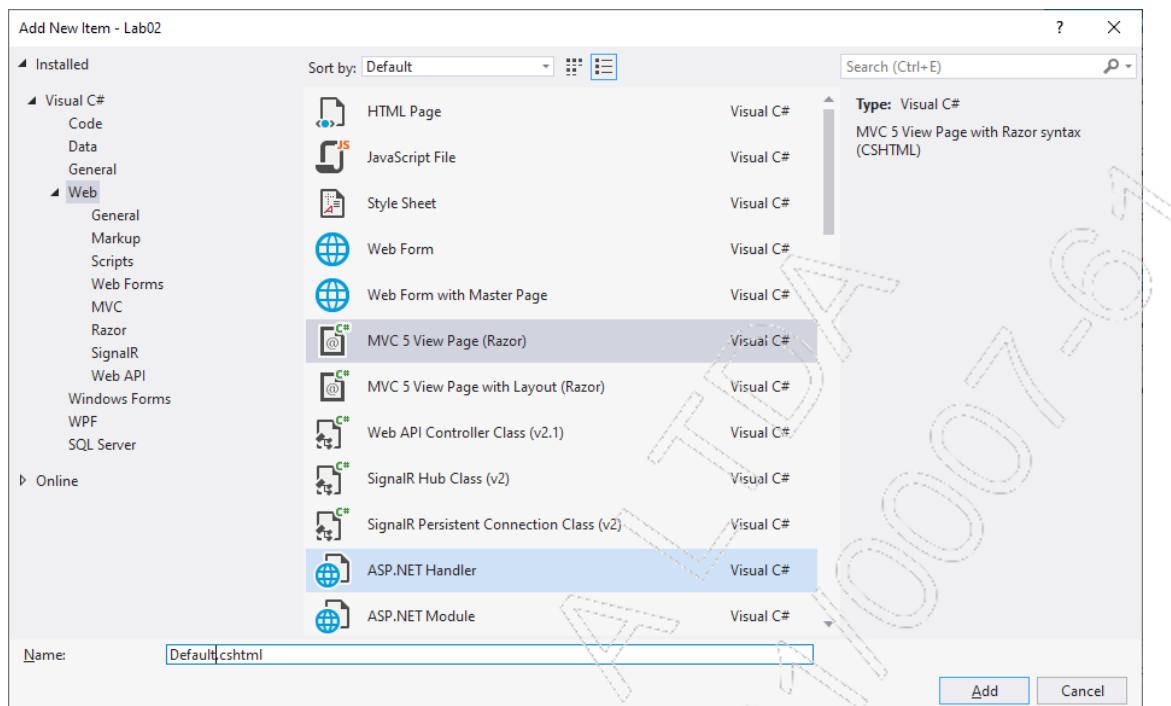
Laboratório 2

1. Adicione ao solution **Capítulo02.Labs** um novo projeto ASP.NET Web Application vazio chamado **Lab02**:



2. Torne o projeto **Lab02** como projeto Default. Para isso, clique com o botão direito do mouse sobre o projeto e selecione a opção **Set as Startup Project**;

3. No projeto Lab02, adicione uma Web Page com Razor (MVC 5 View Page (Razor)) chamada Default.cshtml:



4. Usando as tags do HTML5, defina três regiões para o documento: **Header**, **Section** e **Footer**. Essas serão as áreas do cabeçalho, conteúdo e rodapé, respectivamente;

```

@{
    Layout = null;
}

<!DOCTYPE html>

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

        <section>
        </section>

        <footer>
        </footer>

    </div>
</body>
</html>

```

5. Insira o título da página e o rodapé:

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
</head>  
<body>  
    <div>  
        <header>  
            <h1>Exemplo Web Page</h1>  
            <h2>Tecnologias Web - Html, Javascript, Css, Razor e  
Web Pages</h2>  
  
        </header>  
  
        <section>  
        </section>  
  
        <footer>  
            <hr />  
            <p>  
                Desenvolvido para o curso de ASP.NET  
                © @DateTime.Now.Year - Impacta Tecnologia  
            </p>  
        </footer>  
    </div>  
</body>  
</html>
```

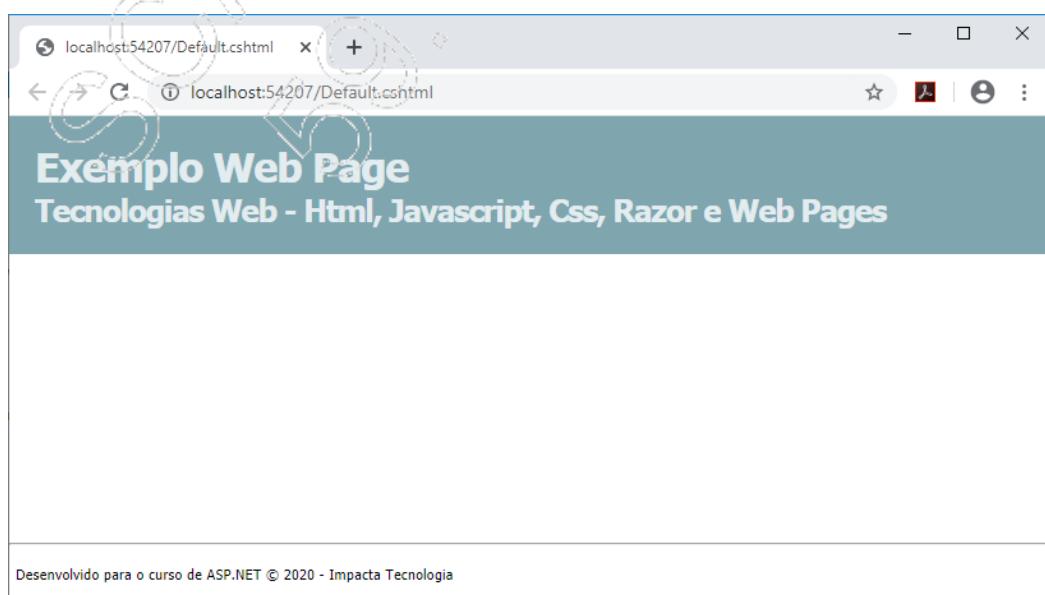
6. Adicione ao projeto uma folha de estilo chamada **Estilos.css**. Para isso, use o menu **Project, Add New Item, Style Sheet**;

7. Defina os seguintes estilos no arquivo **Estilos.css**:

```
body {  
    font-family: Tahoma;  
    margin: 0px;  
}  
  
header {  
    padding: 20px;  
    background-color: #749ca7;  
    color: #e0eaed;  
    letter-spacing: -1px;  
}  
  
h1, h2 {  
    margin: 0px;  
}  
  
footer {  
    position: fixed;  
    bottom: 0px;  
    width: 100%;  
}  
  
footer p {  
    padding: 5px;  
    font-size: 70%;  
}  
  
section {  
    max-width: 960px;  
    padding: 20px;  
}  
  
section ul {  
    padding: 0px;  
}
```

8. Arraste o arquivo **Estilos.css** da janela **Solution Explorer** para dentro da página **Default.cshtml**, abaixo da tag **<title>** (ou digite a referência diretamente). Visualize o resultado até o momento:

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
    <link href="~/Estilos.css" rel="stylesheet" />  
</head>  
<body>  
    <div>  
        <header>  
            <h1>Exemplo Web Page</h1>  
            <h2>Tecnologias Web - Html, Javascript, Css, Razor e Web  
Pages</h2>  
        </header>  
  
        <section>  
        </section>  
  
        <footer>  
            <hr />  
            <p>  
                Desenvolvido para o curso de ASP.NET  
                &copy; @DateTime.Now.Year - Impacta Tecnologia  
            </p>  
        </footer>  
    </div>  
</body>  
</html>
```



9. Na página **Default.cshtml**, insira o conteúdo do item **section** e visualize a página:

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
    <link href="~/Estilos.css" rel="stylesheet" />  
</head>  
<body>  
    <div>  
        <header>  
            <h1>Exemplo Web Page</h1>  
            <h2>Tecnologias Web - Html, Javascript, Css, Razor e Web  
Pages</h2>  
        </header>  
  
        <section>  
            <p>  
                No desenvolvimento Web, diversas tecnologias são  
utilizadas.  
outras  
                Algumas são executadas do lado do Servidor(Server) e  
                são executadas do lado do Cliente(Client).  
            </p>  
            <dl>  
                <dt>  
                    <a href="#">Tecnologias: Client</a>  
                </dt>  
                <dd>  
                    <ul>  
                        <li>Html</li>  
                        <li>Css</li>  
                        <li>Javascript</li>  
                    </ul>  
                </dd>  
            </dl>  
        </section>  
    </div>  
</body>
```

```
<dt>
    <a href="#">Tecnologias: Server</a>
</dt>
<dd>
    <ul>
        <li>C#</li>
        <li>ASP.NET</li>
        <li>Web Pages</li>
        <li>MVC</li>
        <li>Razor</li>
    </ul>
</dd>

</dl>

</section>

<footer>
    <hr />
    <p>
        Desenvolvido para o curso de ASP.NET
        &copy; @DateTime.Now.Year - Impacta Tecnologia
    </p>
</footer>

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

10. O **JavaScript** é útil para realizar processamento na página em que não há a necessidade de obter dados do servidor. As descrições das tecnologias vão ficar escondidas e serão exibidas quando o usuário clicar no link. Insira o código que esconde ou exibe um elemento HTML. O script deve ser incluído como última instrução antes do fechamento do elemento **<body>**. Altere também os links da página para executar a função em JavaScript e defina o atributo **Id** apropriado:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
    <link href="~/Estilos.css" rel="stylesheet" />
</head>
<body>
    <div>
        <header>
            <h1>Exemplo Web Page</h1>
            <h2>Tecnologias Web - Html, Javascript, Css, Razor e Web
Pages</h2>
        </header>
        <div id="content">
            <ul>
                <li>C#</li>
                <li>ASP.NET</li>
                <li>Web Pages</li>
                <li>MVC</li>
                <li>Razor</li>
            </ul>
        </div>
    </div>
</body>
```

```
</header>

<section>
    <p>
        No desenvolvimento Web, diversas tecnologias são utilizadas.
        Algumas são executadas do lado do Servidor(Server) e outras
        são executadas do lado do Cliente(Client).
    </p>

    <dl>
        <dt>
            <a href="javascript:exibirEsconderElemento('ddClient')">
                Tecnologias: Client
            </a>
        </dt>
        <dd id="ddClient" style="display:none">
            <ul>
                <li>Html</li>
                <li>Css</li>
                <li>Javascript</li>
            </ul>
        </dd>

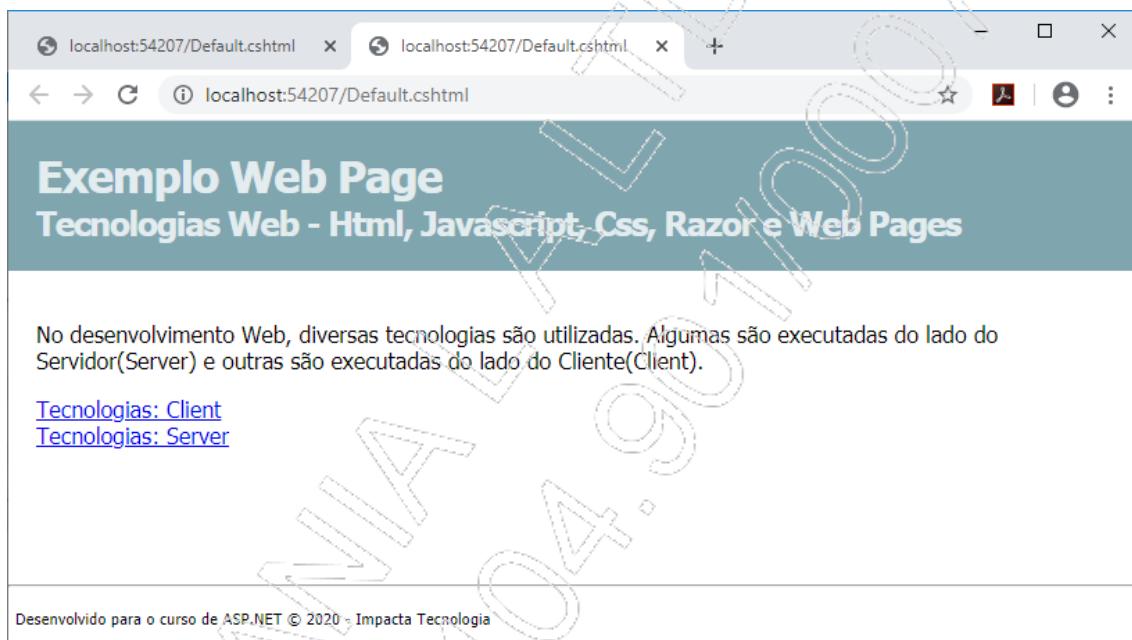
        <dt>
            <a href="javascript:exibirEsconderElemento('ddServer')">
                Tecnologias: Server
            </a>
        </dt>
        <dd id="ddServer" style="display:none">
            <ul>
                <li>C#</li>
                <li>ASP.NET</li>
                <li>Web Pages</li>
                <li>MVC</li>
                <li>Razor</li>
            </ul>
        </dd>
    </dl>
</section>

<footer>
    <hr />
    <p>
        Desenvolvido para o curso de ASP.NET
        © @DateTime.Now.Year - Impacta Tecnologia
    </p>
</footer>

</div>
```

```
<script>
    function exibirEsconderElemento(elementoId) {
        var elemento = document.getElementById(elementoId);
        if (elemento.style.display == "none") {
            elemento.style.display = "block";
        }
        else {
            elemento.style.display = "none";
        }
    }
</script>
</body>
</html>
```

11. Visualize e teste a página, clicando em cada link:



The screenshot shows a web browser window with the URL `localhost:54207/Default.cshtml#`. The page has a dark blue header with the title "Exemplo Web Page" and subtitle "Tecnologias Web - Html, Javascript, Css, Razor e Web Pages". The main content area contains a text paragraph about web development technologies and two links: "Tecnologias: Client" and "Tecnologias: Server". A decorative background pattern of large, semi-transparent letters (HTML, CSS, JS, etc.) is visible. At the bottom, a footer bar reads "Desenvolvido para o curso de ASP.NET © 2020 - Impacta Tecnologia".

The screenshot shows a web browser window with the URL `localhost:54207/Default.cshtml#`. The page has a dark blue header with the title "Exemplo Web Page" and subtitle "Tecnologias Web - Html, Javascript, Css, Razor e Web Pages". The main content area contains a text paragraph about web development technologies and two links: "Tecnologias: Client" and "Tecnologias: Server". A decorative background pattern of large, semi-transparent letters (HTML, CSS, JS, etc.) is visible. At the bottom, a footer bar reads "Desenvolvido para o curso de ASP.NET © 2020 - Impacta Tecnologia".

12. Adicione um parágrafo depois da lista de tecnologias, na página **Default.cshtml**:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
    <link href="~/Estilos.css" rel="stylesheet" />
</head>
<body>
    <div>
        <header>
            <h1>Exemplo Web Page</h1>
            <h2>Tecnologias Web - Html, Javascript, Css, Razor e Web
Pages</h2>
        </header>

        <section>
            <p>
                No desenvolvimento Web, diversas tecnologias são
utilizadas.
                Algumas são executadas do lado do Servidor(Server) e outras
são executadas do lado do Cliente(Client).
            </p>
            <dl>
                <dt>
                    <a href="javascript:exibirEsconderElemento('ddClient')">
                        Tecnologias: Client
                    </a>
                </dt>
                <dd id="ddClient" style="display:none">
                    <ul>
                        <li>Html</li>
                        <li>Css</li>
                        <li>Javascript</li>
                    </ul>
                </dd>
            <dt>
```

```

        <a href="javascript:exibirEsconderElemento('ddServer')">
            Tecnologias: Server
        </a>
    </dt>
    <dd id="ddServer" style="display:none">
        <ul>
            <li>C#</li>
            <li>ASP.NET</li>
            <li>Web Pages</li>
            <li>MVC</li>
            <li>Razor</li>
        </ul>
    </dd>
</dl>

<p>
    <a href="ComentarioForm.cshtml">Deixe seu comentário</a>
</p>

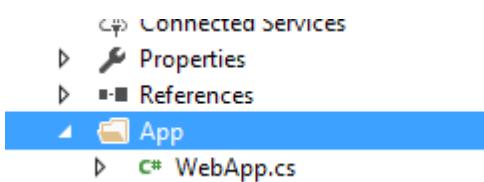
</section>

<footer>
    <hr />
    <p>
        Desenvolvido para o curso de ASP.NET
        © @DateTime.Now.Year - Impacta Tecnologia
    </p>
</footer>

</div>
<script>
    function exibirEsconderElemento(elementoId) {
        var elemento = document.getElementById(elementoId);
        if (elemento.style.display == "none") {
            elemento.style.display = "block";
        }
        else {
            elemento.style.display = "none";
        }
    }
</script>
</body>
</html>

```

13. Os comentários serão gravados em um arquivo de texto. Adicione uma pasta ao projeto chamada **App** e, dentro dessa pasta, uma classe chamada **WebApp**:



14. A classe **WebApp** é estática e terá um método para gravar um comentário e outro para ler os comentários. Insira as declarações dos métodos e compile o projeto:

```
namespace Lab02.App
{
    public static class WebApp
    {
        public static void ComentarioIncluir(
            string nome, string comentario)
        {

        }

        public static string ComentariosObter()
        {
            return null;
        }
    }
}
```

15. Na classe **WebApp**, insira uma propriedade interna para retornar o nome do arquivo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace Lab02.App
{
    public static class WebApp
    {
        private static string comentarioArquivo
        {
            get
            {
                return HttpContext.Current.Server.MapPath("~/comentarios.txt");
            }
        }

        public static void ComentarioIncluir(
            string nome, string comentario)
        {

        }

        public static string ComentariosObter()
        {
            return null;
        }
    }
}
```

16. Na classe **WebApp**, crie o corpo dos métodos. Utilize a diretiva de compilação **using System.IO** para acesso às classes **StreamWriter** e **StreamReader** e a diretiva **using System.Text** para acesso à classe **Encoding**;

```
using System;
using System.IO;
using System.Text;
using System.Web;

namespace Lab02.App
{
    public static class WebApp
    {
        private static string comentarioArquivo
        {
            get
            {
                return HttpContext.Current.Server.MapPath("~/comentarios.
txt");
            }
        }

        public static void ComentarioIncluir(
            string nome, string comentario)
        {
            using (var writer =
                new StreamWriter(comentarioArquivo, true, Encoding.UTF8))
            {
                writer.WriteLine(
                    $"{DateTime.Now:dd/MM/yyyy} - {DateTime.
Now:HH:mm:ss}");
                writer.WriteLine($"{nome}: {comentario}\r\n");
            }
        }

        public static string ComentariosObter()
        {
            string texto = string.Empty;
            if (!File.Exists(comentarioArquivo))
            {
                return texto;
            }

            using (var reader = new StreamReader(comentarioArquivo))
            {
                texto = reader.ReadToEnd();
            }

            return texto;
        }
    }
}
```

17. Adicione, na raiz do projeto, a página **ComentarioForm.cshtml**, que contém, além dos elementos padrão, um formulário para enviar o comentário do usuário;

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title></title>  
    <link href="~/Estilos.css" rel="stylesheet" />  
</head>  
<body>  
    <div>  
        <header>  
            <h1>Exemplo Web Page</h1>  
            <h2>Html, Javascript, Css, Razor e Web Pages</h2>  
  
            <a href="~/Default.cshtml">Voltar</a>  
        </header>  
        <section>  
  
            <p>Deixe seu comentário:</p>  
  
            <form action="~/ComentarioProcessar.cshtml" method="post">  
  
                <div class="form-grupo">  
                    <label for="nome">Nome:</label>  
                    <input type="text" name="nome" id="nome" />  
                </div>  
  
                <div class="form-grupo">  
                    <label for="comentario">Comentário</label>  
                    <textarea id="comentario" name="comentario"></textarea>  
                </div>  
  
                <div class="form-botoes">  
                    <input type="submit" value="Enviar" />  
                </div>  
            </form>  
  
        </section>  
  
    </div>  
    </body>  
</html>
```

18. Adicione a página **ComentarioProcessar.cshtml**. Essa página processa os dados enviados pelo formulário;

```
@{
    Layout = null;
}
@using Lab02.App;

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
</head>
<body>
    <div>
        <header>
            <h1>Exemplo Web Page</h1>
            <h2>Html, Javascript, Css, Razor e Web Pages</h2>
            <a href="~/Default.cshtml">Voltar</a>
        </header>

        <section>
            @if (!IsPost)
            {
                Response.Redirect("~/comentarioForm.cshtml");
            }
            else
            {

                string nome = Request.Form["nome"];
                string comentario = Request.Form["comentario"];

                if (string.IsNullOrWhiteSpace(nome) ||
                    string.IsNullOrWhiteSpace(comentario))
                {
                    Response.Redirect("~/comentarioForm.cshtml");
                }
                else
                {
                    WebApp.ComentarioIncluir(nome, comentario);
                }
            }

            <p>Obrigado por seu comentário</p>

            <p>
                <a href="~/Default.cshtml">Voltar</a>
                <a href="~/ComentarioLista.cshtml">Exibir Comentários</a>
            </p>
        </section>
    </div>
</body>
</html>
```

19. Adicione a página **ComentarioLista.cshtml**. Essa página exibe os comentários gravados;

```
@{
    Layout = null;
}
@using Lab02.App

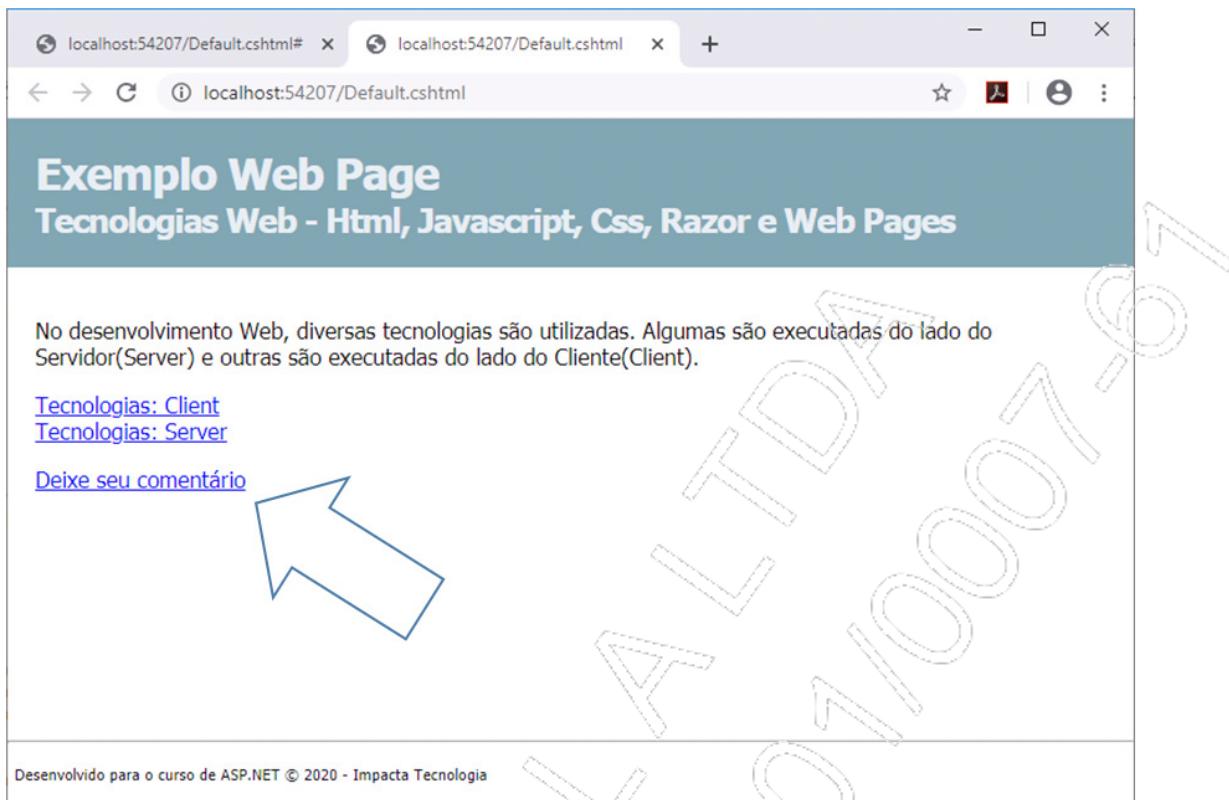
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
</head>
<body>
    <div>
        <header>
            <h1>Exemplo Web Page</h1>
            <h2>Html, Javascript, Css, Razor e Web Pages</h2>
            <a href="~/Default.cshtml">Voltar</a>
        </header>

        <section>
            <p>Comentários:</p>
            <pre>
                @WebApp.ComentariosObter()
            </pre>
        </section>
    </div>
</body>
</html>
```



20. Teste o projeto completo:

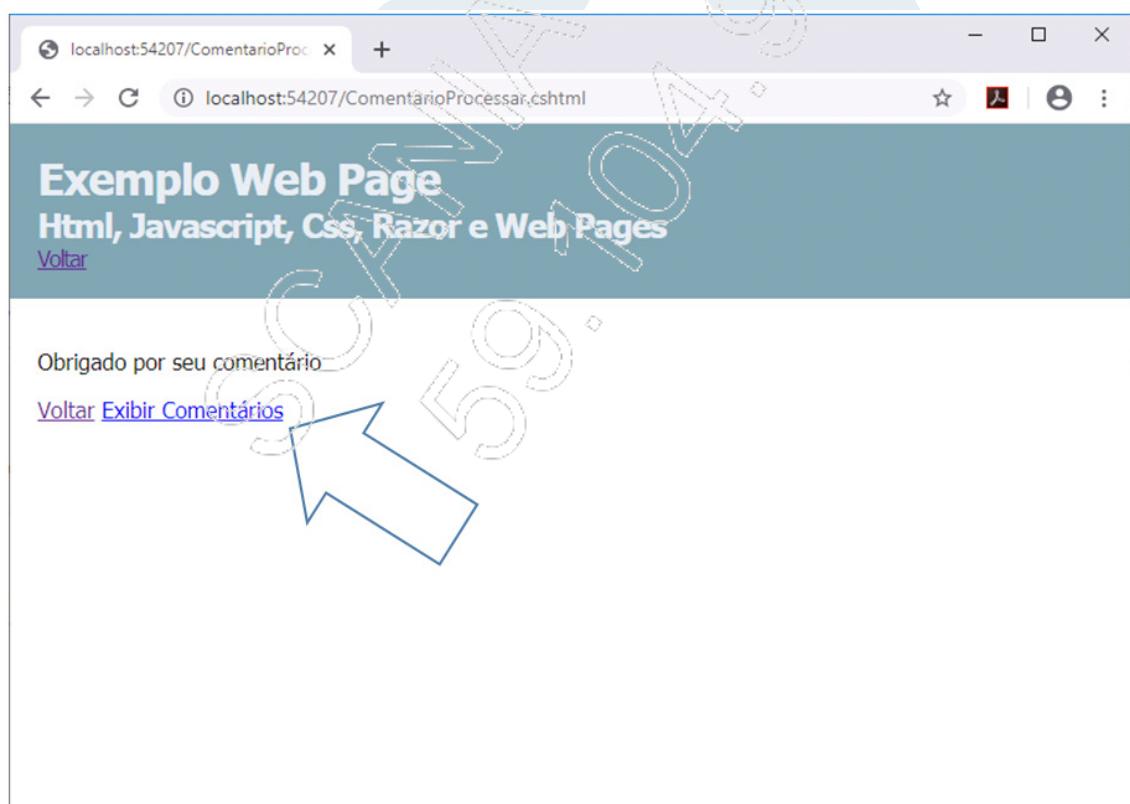


No desenvolvimento Web, diversas tecnologias são utilizadas. Algumas são executadas do lado do Servidor(Server) e outras são executadas do lado do Cliente(Client).

[Tecnologias: Client](#)
[Tecnologias: Server](#)

[Deixe seu comentário](#)

Desenvolvido para o curso de ASP.NET © 2020 - Impacta Tecnologia



Obrigado por seu comentário

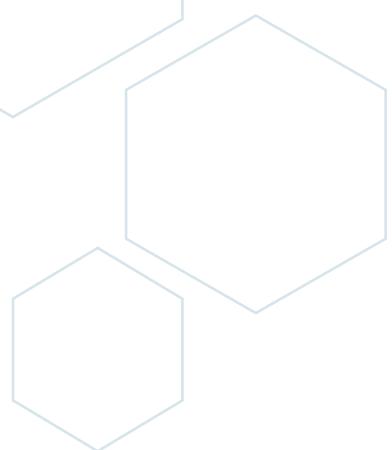
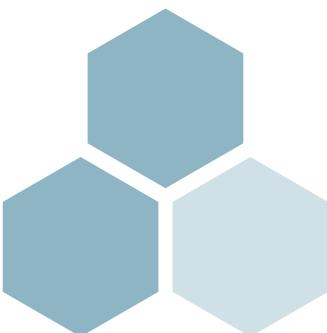
[Voltar](#) [Exibir Comentários](#)



3

ASP.NET MVC – Controllers, Models e Views

- Criando um projeto MVC;
- Rotas;
- Controllers e Actions;
- Os objetos ViewBag e ViewData;
- Models – Views fortemente tipadas;
- Elaboração de formulários;
- Validação e filtros;
- Layout.



3.1. Introdução

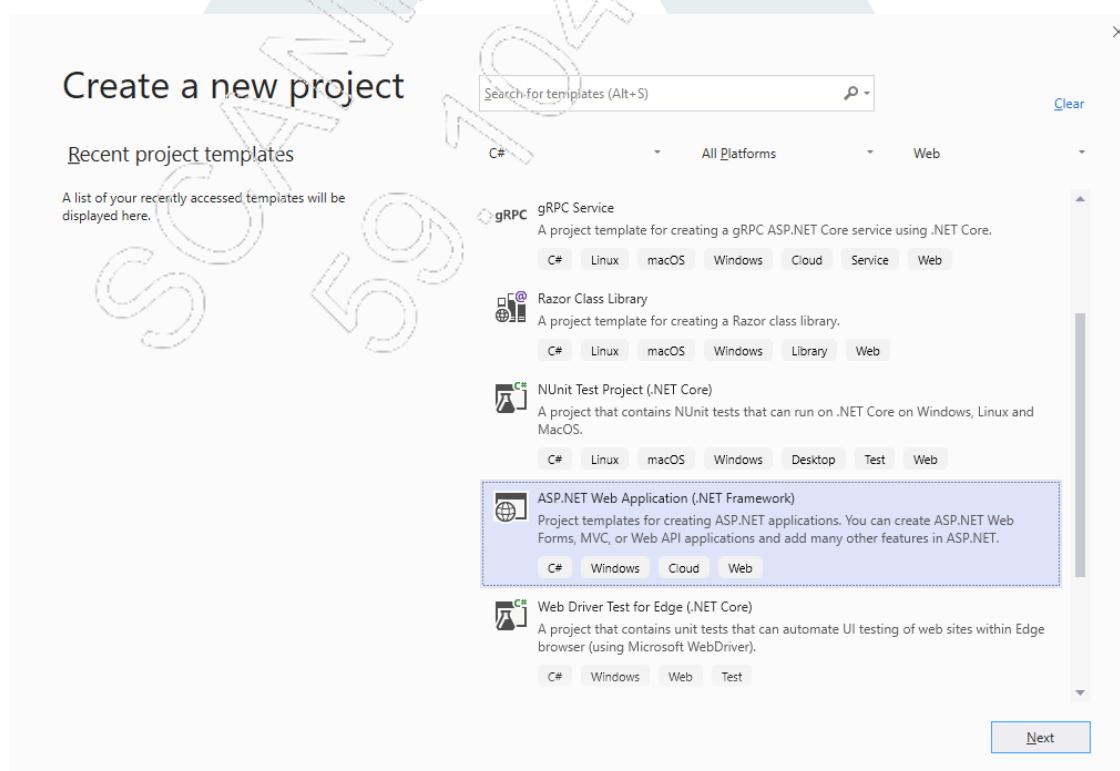
A arquitetura MVC é bem diferente da arquitetura baseada em Web Pages, vista no Capítulo 2. O mecanismo usado pelo MVC usa as Web Pages não como páginas executadas no browser, mas como **views**, ou seja, como componentes que recebem informações provenientes de algum processamento no servidor. Em outras palavras, um componente central (o **controller**) interage com as outras partes do sistema, como acesso a dados, acesso a arquivos, imagens, etc. e o resultado dessa interação é apresentado para o usuário na view, através da execução de um método do controller.

Um controller é uma classe como outra qualquer, mas com características especiais que a definem como tal. Os métodos nessa classe (controller) também recebem outra denominação: **actions**. Esses, quando executados, geram algum tipo de resposta para o usuário, e se essa resposta for um conteúdo HTML, temos as views para apresentá-la.

Neste capítulo, apresentaremos todo o processo para o desenvolvimento de uma aplicação ASP.NET MVC, ou simplesmente MVC. Este processo inclui o desenvolvimento de controllers, de views, e de outro grupo especial de classes, os **Models**. Vamos iniciar com a criação de um projeto e, em seguida, apresentaremos as etapas e padrões necessários para que a aplicação seja funcional.

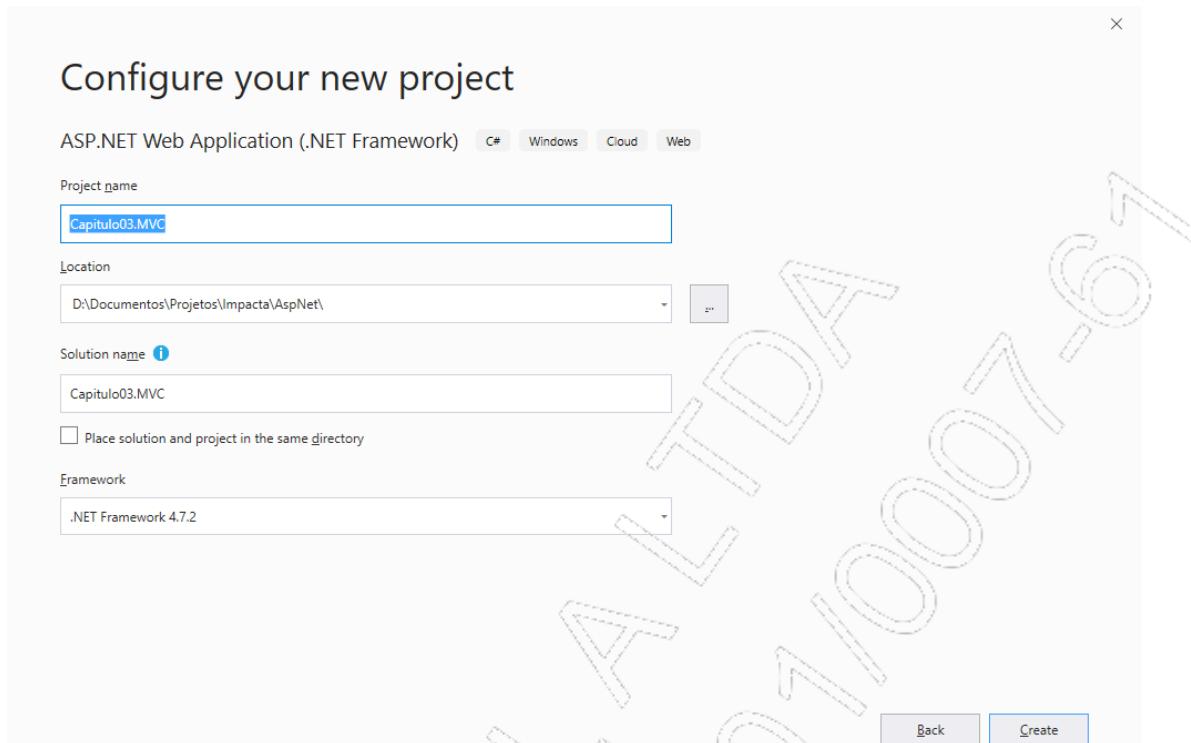
3.2. Criando um projeto MVC

Vamos iniciar pela criação de um projeto do tipo MVC. No Visual Studio, selecione a opção **File / New / Project...**. Em seguida, selecione o item **ASP.NET Web Application (.NET Framework)** (lembre-se de manter a opção com a linguagem C#):

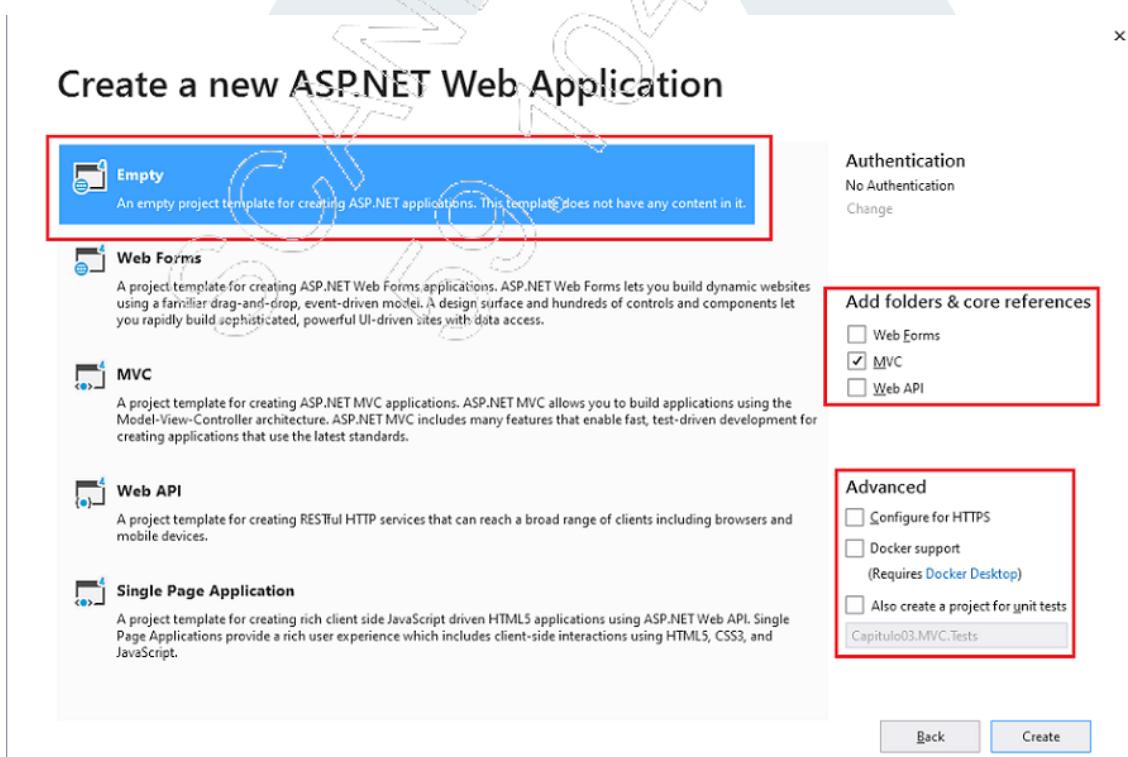


ASP.NET MVC - Controllers, Models e Views

No próximo passo, atribua um nome para o projeto e para o solution. Neste exemplo, definimos ambos como **Capítulo03.MVC**. Escolha uma pasta conveniente.



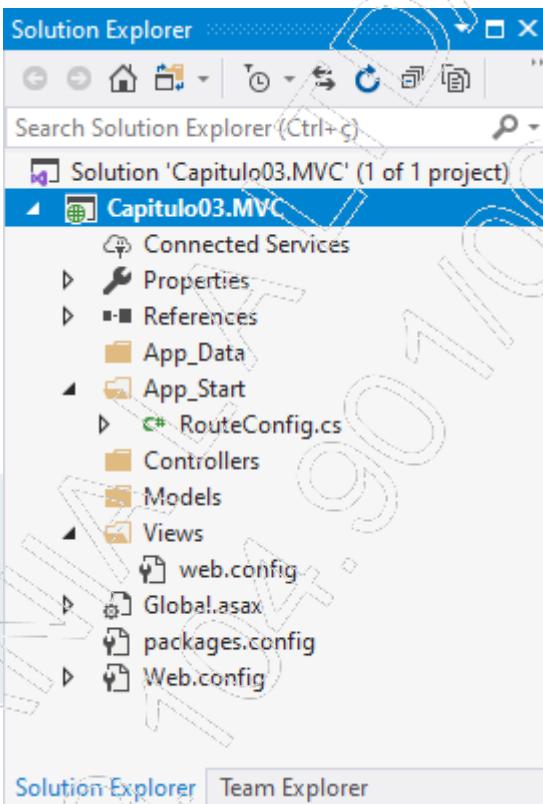
Ao clicar em **Create**, selecione a opção **Empty**, mas marque à direita a caixa de seleção **MVC**, no grupo nomeado como **Add folders & core references**. Desmarque a opção **Configure for HTTPS**, no grupo **Advanced**:



Quando marcamos a opção MVC, o Visual Studio configurou as dependências necessárias para esse tipo de projeto. Algumas utilizaram o NuGet, e outras já estavam presentes no framework. O objetivo da utilização do NuGet é que, se o projeto for aberto em outra versão do Visual Studio, o NuGet permite que seja feito o download das dependências necessárias para o bom funcionamento do projeto.

3.2.1. Conhecendo as partes de um projeto ASP.NET MVC

Vamos apresentar as pastas criadas durante o processo de criação do projeto.



- **Pasta App_Data:** Nesta pasta ficam os arquivos de bancos de dados, quando estes forem criados localmente, e na modalidade "Arquivo de banco de dados";
- **Pasta App_Start:** Nesta pasta ficam, principalmente, os arquivos de configurações de rotas, tanto para aplicações MVC como WebAPI. É possível que um projeto conte com ambas as tecnologias e, neste caso, teremos dois arquivos de configurações;
- **Pasta Controllers:** É nesta pasta que as classes representando os Controllers da aplicação são incluídas. É importante que seja usada esta pasta, pois ela é parte integrante da convenção do MVC;

- **Pasta Models:** Esta pasta não faz parte da convenção do MVC, mas recomenda-se que todas as classes representando os models sejam criadas nesta pasta, por questões de padronização;
- **Pasta Views:** Esta é outra pasta que participa da convenção usada no projeto MVC. Quando criarmos as views, veremos que deverá haver uma subpasta com o nome do controller, para conter as Web Pages representando as views para o controller em questão. Outros arquivos do ciclo de vida da aplicação também devem estar nesta pasta. Uma subpasta importante (ainda não criada) é a subpasta **Shared**. Nela ficam as views comuns a todos os controllers;
- Arquivo **Global.asax**: Neste arquivo são escritos os métodos representando os eventos da aplicação, como: inicialização da aplicação, criação de uma sessão, finalização da aplicação, entre outros. O evento usado em qualquer aplicação é o **Application_Start()**, responsável principalmente pela definição da rota definida no arquivo **App_Start/RouteConfig.cs**.

O **padrão MVC** utiliza o conceito **COC (Convention Over Configuration)**. Isso significa que as convenções apresentadas fazem parte do padrão de configuração adotado para projetos desse tipo.

Cada uma das partes apresentadas neste tópico será detalhada na medida em que evoluirmos no desenvolvimento do projeto de exemplo.

3.3. Rotas

Como já apresentado na introdução deste capítulo, em aplicações MVC não executamos uma página com extensão .cshtml como Web Page. Elas são chamadas pelos actions dos controllers. Os actions são os métodos desenvolvidos no controller e, quando executados, produzem uma resposta.

Sendo assim, para que uma resposta seja apresentada para o usuário, devemos executar, direta ou indiretamente, o action. A chamada ao action acontece por meio do sistema de **rotas**.

Quando um projeto é criado com o modelo MVC, é gerado o arquivo **RouteConfig.cs** na pasta **App_Start**. Observe seu conteúdo:

```
namespace Capitulo03.MVC
{
    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 });
        }
    }
}
```

Vemos a classe **RouteConfig** e o método estático **RegisterRoutes**. Esse método recebe como parâmetro uma coleção do tipo **RouteCollection** chamado **routes**. A coleção **RouteCollection** expõe, dentre outros, o método **MapRoute**, responsável por adicionar um padrão de rota para a aplicação.

Como padrão, o Visual Studio inclui o padrão **{controller}/{action}/{id}** definido como valor do parâmetro **url** do método **MapRoute**. Observe que esse método especifica outros dois parâmetros: **defaults** e **name**. **name** permite especificar um nome para a rota, e **defaults** permite especificar os valores padrão para os elementos da rota:

- O parâmetro **id** é opcional, e está definido assim através do valor **UrlParameter.Optional**;
- O parâmetro **action**, quando não especificado, assume o valor **Index**. Nesse caso, deve haver o método (**action**) **Index** definido no controller;
- O parâmetro **controller**, quando não especificado, assume o valor **Home**. Analogamente, devemos ter esse controller definido na aplicação.

É possível alterá-lo, ou mesmo adicionar novas rotas a esse método. É possível também alterar os valores padrão especificados na rota default.

3.3.1. Iniciando a rota

Vimos que o arquivo **Global.asax** contém métodos que pertencem ao ciclo de vida da aplicação. Sendo assim, o método **Application_Start()** é chamado pelo ASP.NET quando a aplicação é iniciada.

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

namespace Capitulo03.MVC
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

Neste método é realizada uma chamada, a **RouteConfig.RegisterRoutes**. Este é o método estático da classe **RouteConfig**, definida no arquivo de mesmo nome.

O objeto **RouteTable.Routes** é configurado pelo método **RegisterRoutes**, através de uma alteração da referência. É este objeto que, no final, o MVC utiliza para armazenar as rotas especificadas.

Na sequência, apresentaremos o processo de criação de controllers e actions, onde poderemos aplicar os conceitos de rotas apresentados aqui.

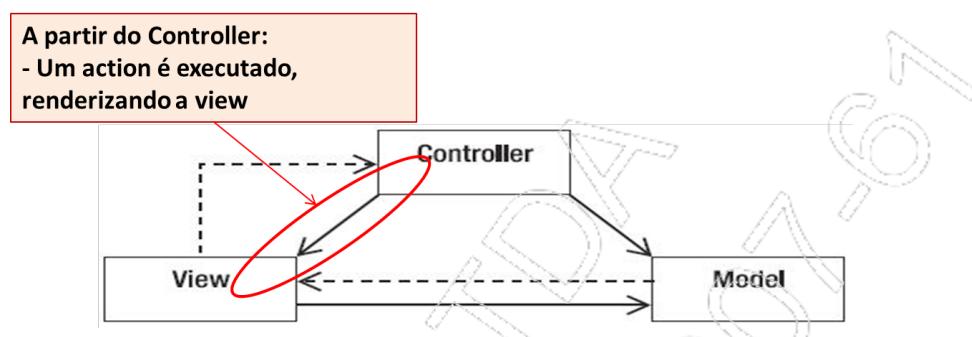
3.4. Controllers e Actions

Os controllers compõem a parte fundamental de uma aplicação MVC, principalmente por conta da arquitetura de mesmo nome.

Como já mencionado, um **controller** é uma classe contendo métodos capazes de gerar a camada de interação com o usuário. Estes métodos são chamados de **actions**.

3.4.1. Arquitetura MVC (Model – View – Controller)

Esta arquitetura pode ser representada pelo esquema:

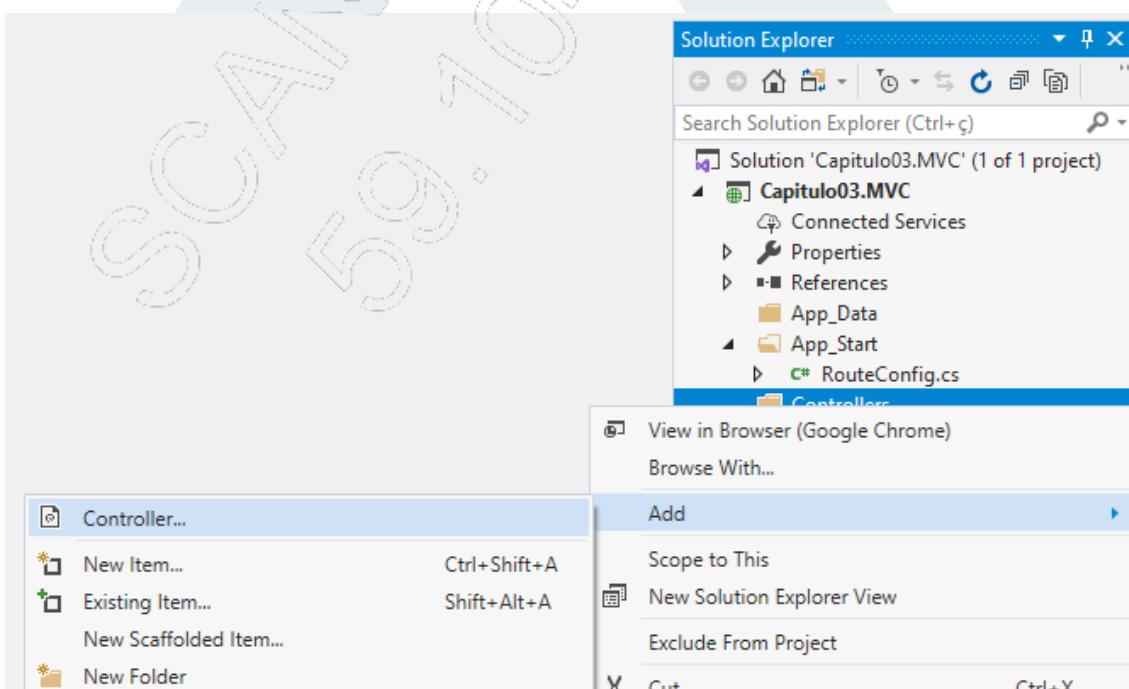


O controller é responsável por produzir as informações a serem apresentadas na view, que é a camada de apresentação.

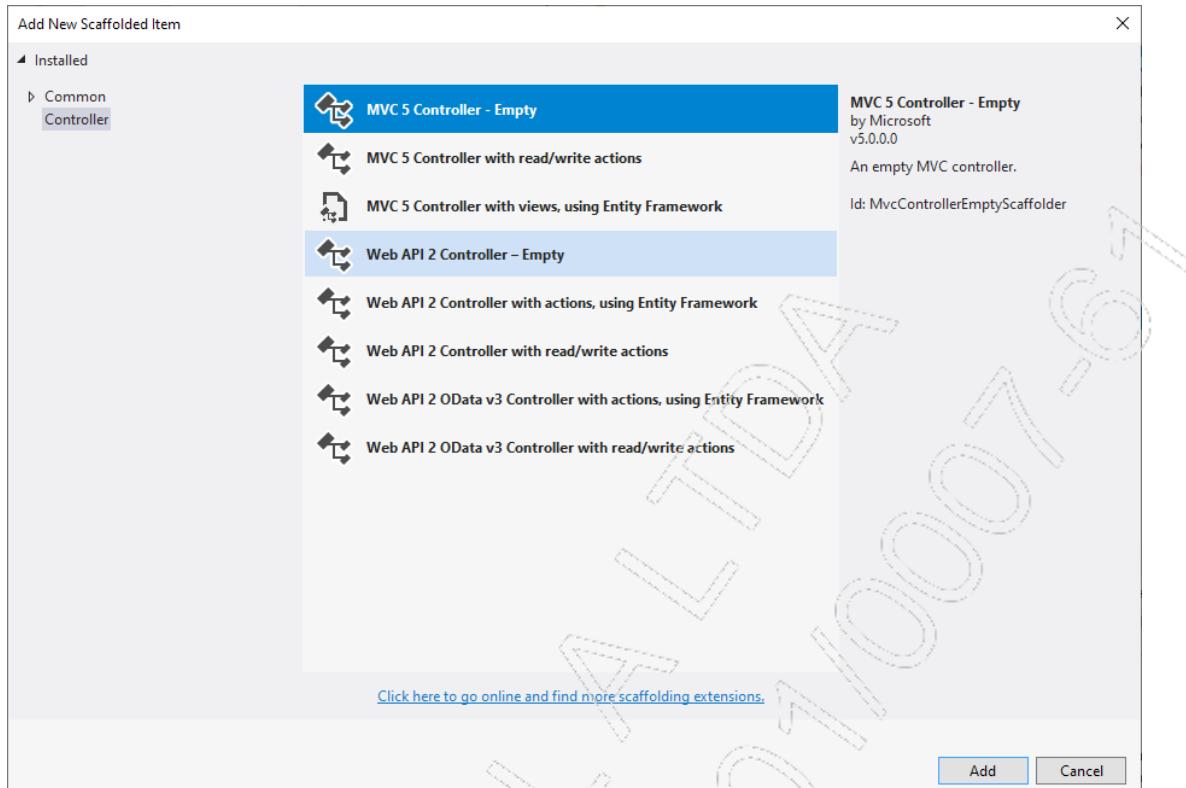
O controller é o componente responsável por receber e processar as requisições, e enviar a resposta para o usuário.

3.4.2. Criando o controller

No nosso projeto **Capítulo03.MVC**, vamos adicionar o controller **Home**. Vamos utilizar a rota da forma como definida pelo Visual Studio. Para adicionar um controller, clique com o botão direito do mouse sobre a pasta **Controllers**:

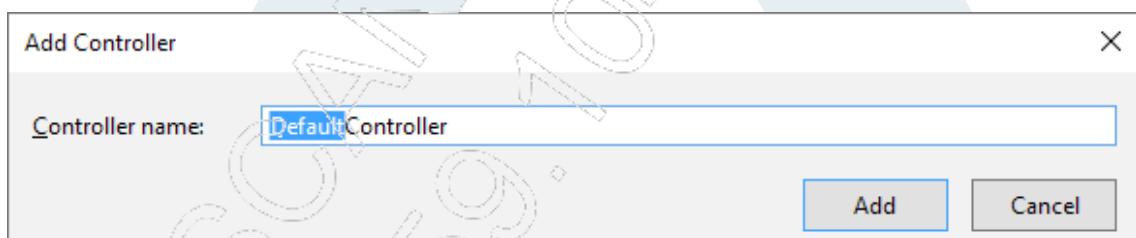


Quando a opção **Controller** for selecionada, aparecerá esta nova janela:



Como estamos desenvolvendo uma aplicação MVC do zero, utilizaremos a opção **MVC 5 Controller - Empty**.

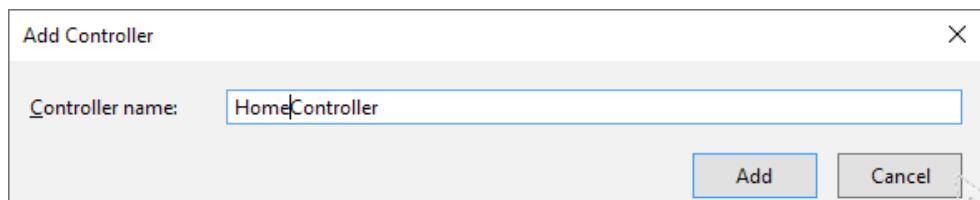
Ao adicionar (**Add**), teremos a próxima janela:



Aqui convém uma observação importante: o nome da classe que representa o controller deve terminar com a palavra **Controller**. No nosso caso, como pretendemos criar o controller **Home**, a classe deve se chamar **HomeController**.

A convenção do MVC especifica que todas as classes representando o controller seja formada pelo nome do controller, seguido do sufixo **Controller. Assim, o controller **Home** deve ser definido na classe **HomeController**.**

Sendo assim, informaremos a palavra **Home** na parte pré-selecionada pelo Visual Studio:



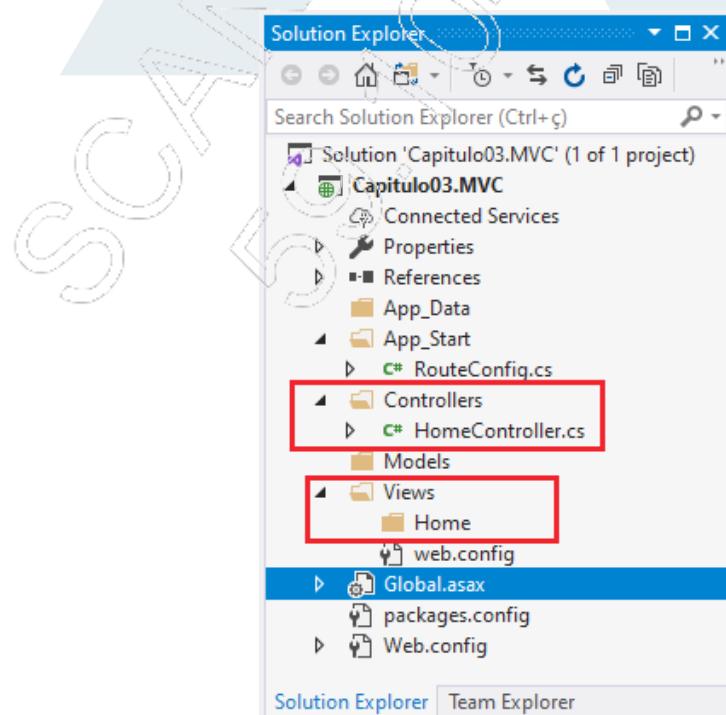
A classe criada já vem com um action predefinido: **Index**.

```
using System.Web.Mvc;

namespace Capitulo03.MVC.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

A classe, além de ter o sufixo **Controller**, deve ser subclasse de **Controller**, presente no namespace **System.Web.MVC**.

Observe que o Visual Studio criou uma nova pasta chamada **Home**, abaixo da pasta **Views**:



É na pasta **Home** que ficarão todas as Web Pages que representarão as views produzidas pelos actions deste controller. Cada controller adicionado ao projeto fará com que uma pasta com seu nome seja criada como subpasta de **Views**. Se não fosse pelo template do Visual Studio, esta pasta deveria ser criada manualmente.

Na classe **HomeController**, já foi adicionado um action chamado **Index**:

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

Ele ainda não será usado neste exemplo, pois teremos uma abordagem separada sobre seu tipo de retorno.

3.4.3. Criando um action

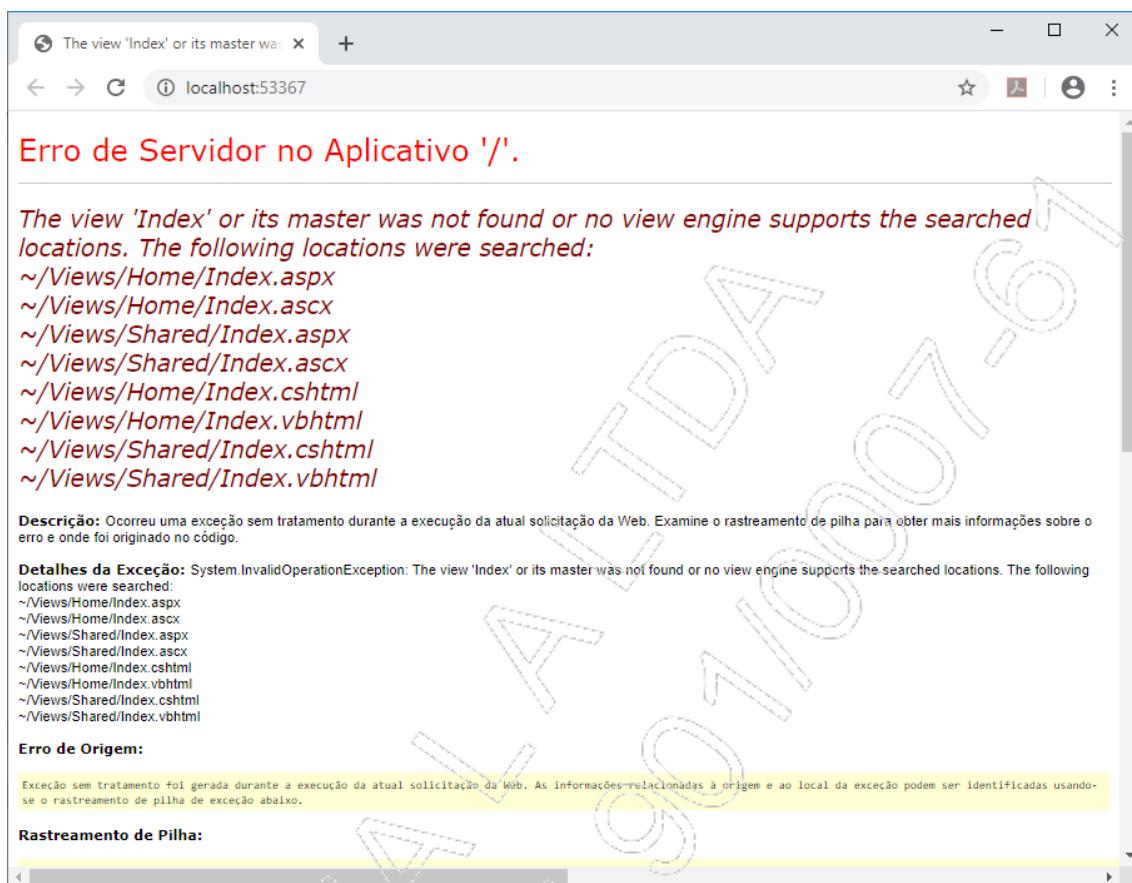
Vamos adicionar nosso primeiro action ao controller **Home**.

Na classe **HomeController**, adicione o método **MostrarTexto**. Este será nosso primeiro action:

```
namespace Capitulo03.MVC.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            return View();
        }

        public string MostrarTexto()
        {
            return "<h1>Primeiro exemplo de action</h1>";
        }
    }
}
```

Após a inclusão deste método, execute a aplicação. Aparecerá um erro semelhante a este:



Vamos entender o que aconteceu aqui.

A rota padrão especifica valores default para o controller e para o action, respectivamente, **Home** e **Index**. A URL executada foi:

http://localhost:53367/

Como nenhum valor foi especificado na rota, a requisição anterior é equivalente a:

http://localhost:53367/home/index

No caso, a rota **/home/index** foi admitida como padrão. O erro ocorreu porque não foi encontrada nenhuma Web Page representando o action **Index**. Observe que o erro mostrou o caminho de busca, até que o erro foi gerado, por não ter sido encontrada nenhuma página para o action mencionado. Observe também que o nome da Web Page foi buscada com o mesmo nome do action: **Index.aspx**, **Index.ascx**, **Index.cshtml** ou **Index.vbhtml**.

No caminho de busca, os arquivos com extensão **aspx** e **ascx** foram incluídos por questões de retrocompatibilidade, já que as versões 1 e 2 do MVC as usavam como extensão para suas views.

Para executar o action que criamos – **MostrarTexto** – vamos escrevê-lo na URL, seguido do nome do controller:

<http://localhost:53367/home/mostrartexto>



O retorno especificado como string para este action gerou um conteúdo HTML, enviado como resposta para o browser.

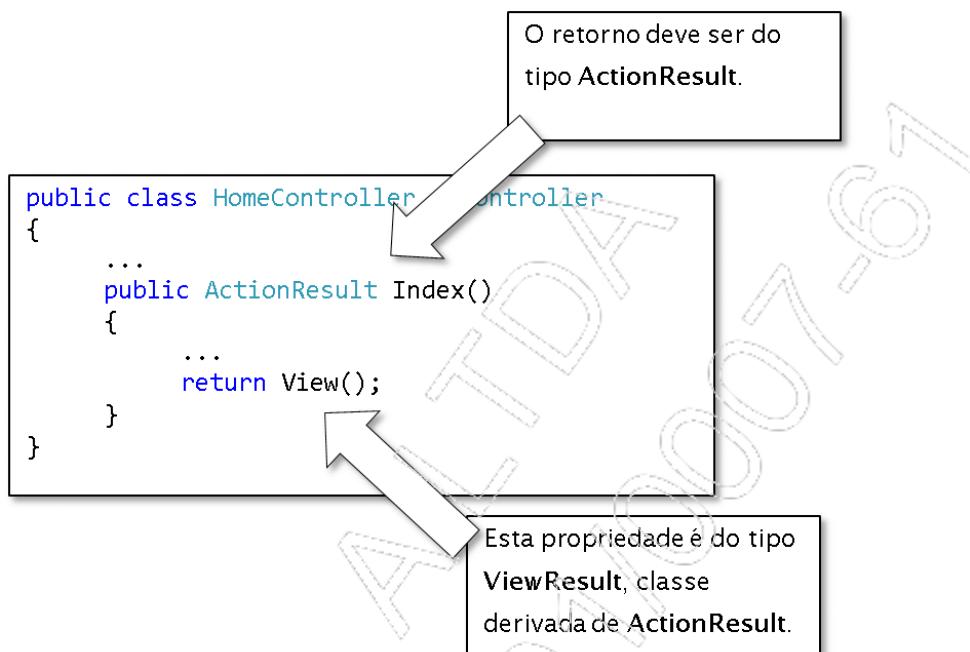
Este foi um exemplo ilustrativo, e não constitui uma prática comum especificarmos actions desta forma. A seguir, apresentaremos os retornos de actions mais comuns.

3.4.4. Retornos comuns de actions

Voltemos ao action **Index**:

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

O retorno deste action é uma chamada ao método `View()`, que está definido na classe base **Controller**, herdado pela classe **HomeController**. Esse retorno é do tipo **ViewResult**, um dos tipos derivados de **ActionResult**. Essa classe (**ActionResult**) representa o resultado de um método disparado por uma URL.



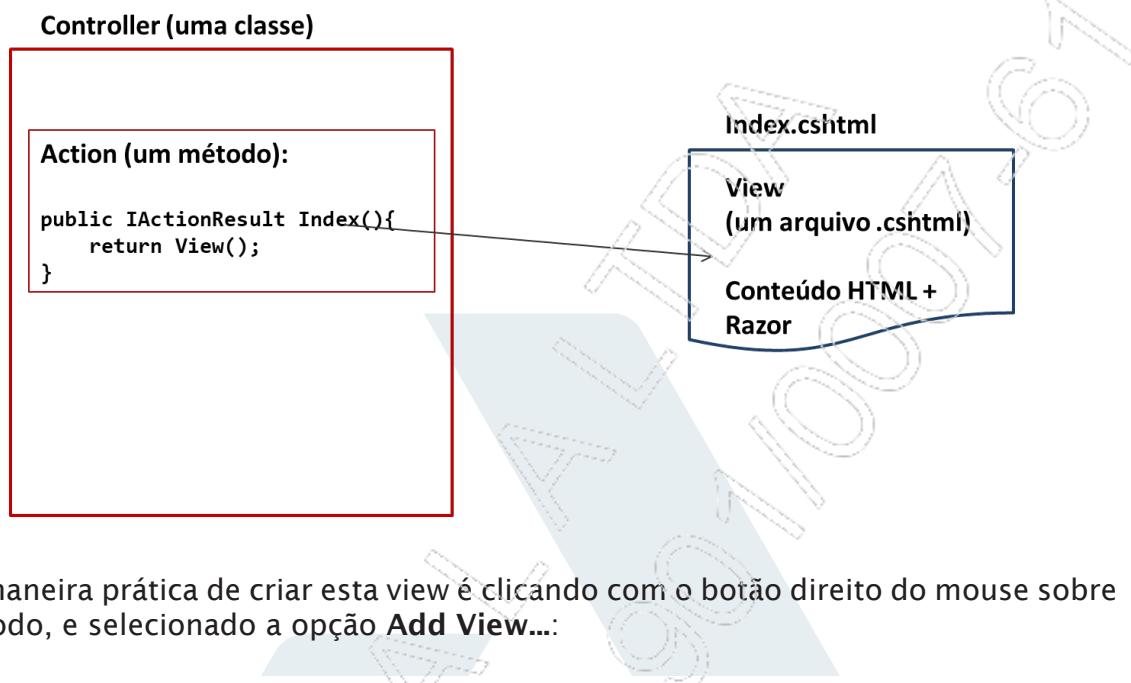
A classe **ActionResult** é muito importante no ASP.NET MVC, pois é a classe base para muitas outras classes que retornam dados:

- **EmptyResult**: Uma resposta vazia, sem conteúdo, é retornada;
- **PartialViewResult**: Retorna uma renderização parcial de uma view;
- **ViewResult**: O retorno mais comum. Uma view é enviada na resposta da solicitação HTTP;
- **RedirectResult**: Realiza um redirecionamento padrão do protocolo HTTP;
- **JsonResult**: Retorna dados no formato JSON;
- **JavascriptResult**: Retorna dados no formato JavaScript;
- **ContentResult**: Escreve o conteúdo diretamente no retorno da solicitação sem que o mecanismo MVC procure por uma view;
- **FileContentResult**: Retorna um arquivo para a solicitação HTTP;
- **FileStreamResult**: Retorna um fluxo de dados (stream) para a solicitação (cliente);
- **FilePathResult**: Retorna um arquivo para a solicitação recebida.

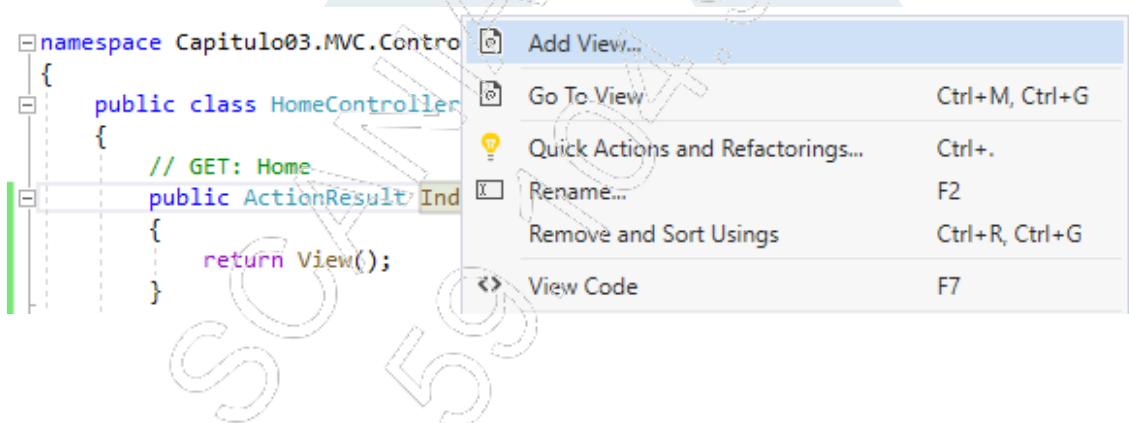
3.4.5. Criando Views

O método **View()**, retornado pelo action **Index**, por ser do tipo **ViewResult**, requer a presença de uma view na pasta **Views/Home**.

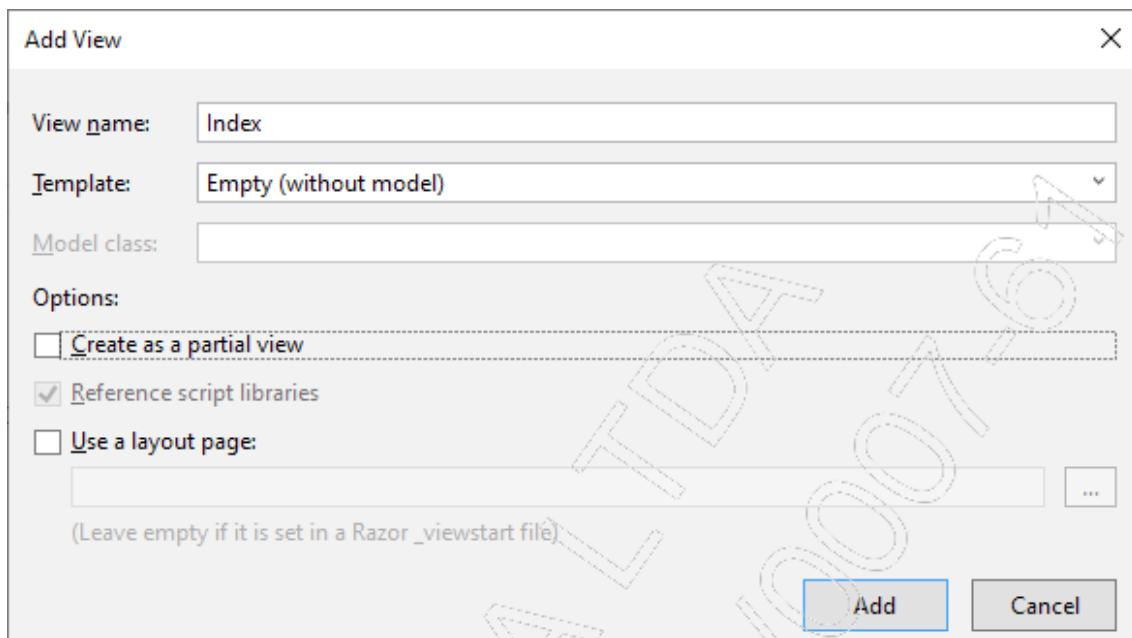
O processo de execução de uma view a partir do action pode ser visto na ilustração a seguir:



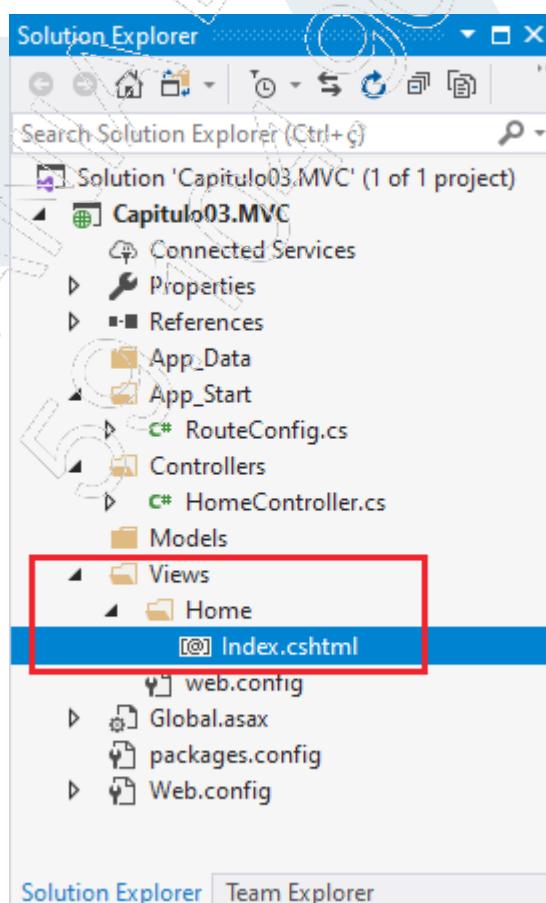
Uma maneira prática de criar esta view é clicando com o botão direito do mouse sobre o método, e selecionado a opção **Add View...**:



Aparecerá a janela a seguir. Deixe DESMARCADAS as opções **Create as Partial View** e **Use a layout page**:



Veja que o nome da view foi considerada como sendo o nome do action (**Index**). Clicando em **Add**, temos o novo arquivo **Index.cshtml** na pasta **Views/Home**:



Index.cshtml

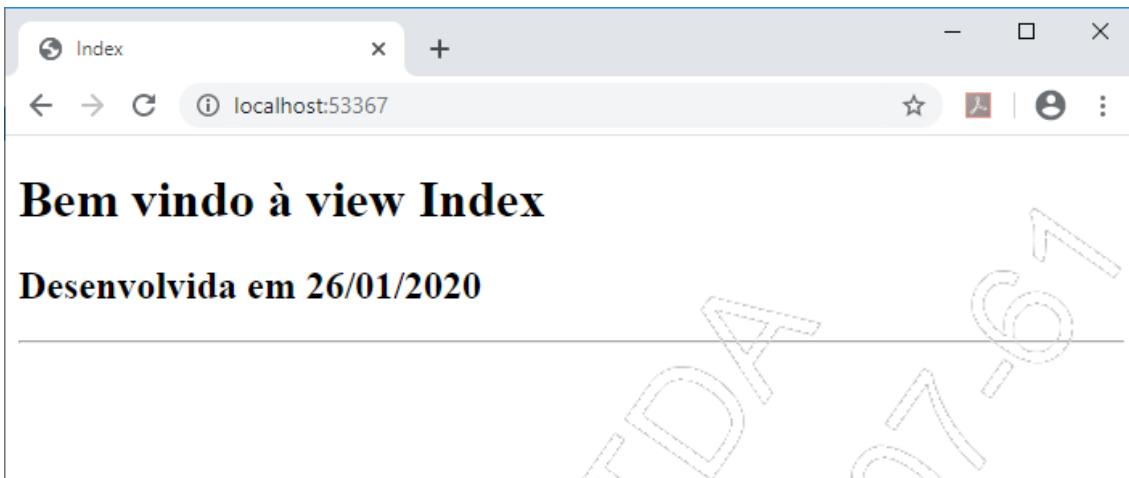
```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Index</title>  
</head>  
<body>  
    <div>  
    </div>  
</body>  
</html>
```

É um arquivo com conteúdo HTML, e por ter a extensão `.cshtml`, admite elementos Razor.

Para incluir uma funcionalidade a esta view, escreva o seguinte conteúdo:

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Index</title>  
</head>  
<body>  
    <div>  
        <h1>Bem vindo à view Index</h1>  
        <h2>Desenvolvida em @DateTime.Now.ToShortDateString()</h2>  
        <hr/>  
  
    </div>  
</body>  
</html>
```

Executando a aplicação, não teremos mais problemas em executar a rota padrão:



3.5. Os objetos ViewBag e ViewData

As páginas **Razor** podem receber, de diversas formas, informações do controller. Uma dessas maneiras é usar os objetos **ViewBag** e **ViewData**. Na verdade, estes objetos são propriedades definidas na classe **ControllerBase**, que é a superclasse de **Controller**.

Os objetivos destas duas propriedades são semelhantes, diferindo na sua definição. **ViewBag** é uma propriedade dinâmica, o que nos permite anexar propriedades em tempo de execução, ao passo que **ViewData** é uma coleção, e requer a definição de uma chave para cada elemento.

Para demonstrar sua utilização, vamos alterar o action **Index**, enviando um **ViewBag** e um **ViewData**:

```
namespace Capitulo03.MVC.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            ViewBag.Empresa = "Impacta Tecnologia";
            ViewData["endereco"] = "Avenida Paulista, 1009";

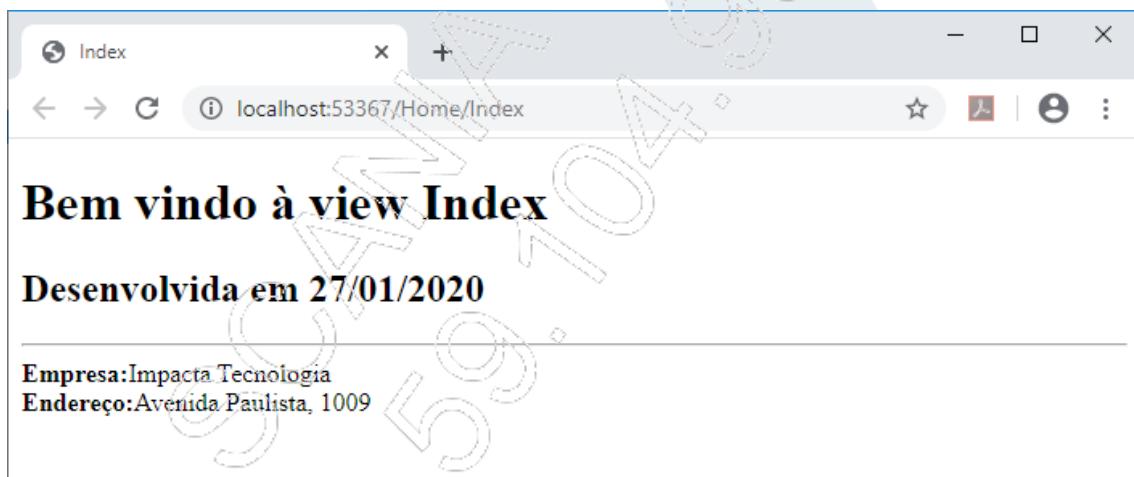
            return View();
        }

        public string MostrarTexto()
        {
            return "<h1>Primeiro exemplo de action</h1>";
        }
    }
}
```

A view recebe estes dados através dos nomes que lhes atribuímos:

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Index</title>  
</head>  
<body>  
    <div>  
        <h1>Bem vindo à view Index</h1>  
        <h2>Desenvolvida em @DateTime.Now.ToShortDateString()</h2>  
        <hr />  
        <span><strong>Empresa:</strong>@ViewBag.Empresa</span> <br/>  
        <span><strong>Endereço:</strong>@ ViewData["endereco"]</span>  
    </div>  
</body>  
</html>
```

Veja o resultado:



É importante verificar que tanto ViewBag como ViewData são criados no controller e passados para a view, e não o oposto. A passagem destes valores é unidirecional.

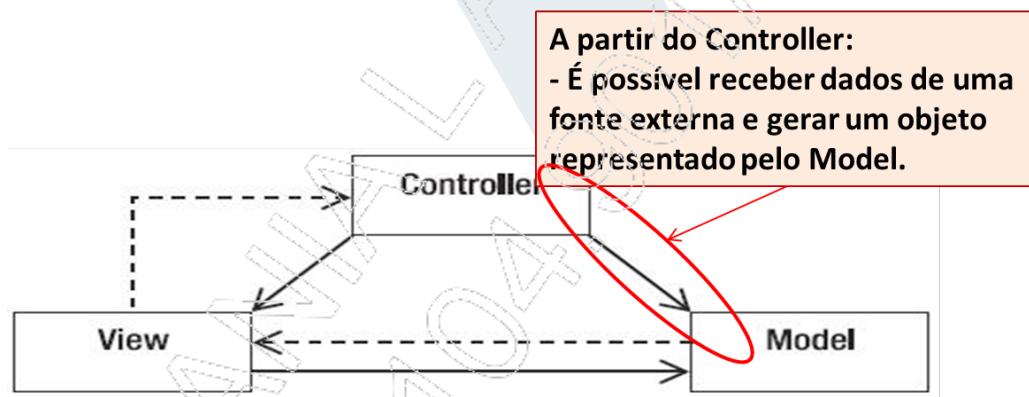
3.6. Models - Views fortemente tipadas

Enviar informações para a view através dos objetos ViewBag e ViewData é uma boa opção, mas somente quando a informação possui um propósito geral. Na maioria dos cenários, é importante fazer com que a view represente um objeto de uma classe pertencente ao modelo usado no projeto. Por exemplo, se a aplicação desenvolvida fosse para o cadastro e consulta de produtos, teríamos que ter uma classe adequada para conter propriedades para o produto. Neste caso, o action e a view deveriam representar um produto – um objeto da classe Produto.

Isso nos leva a definir uma view servindo como um template para o objeto desejado. Esta view, por ter um tipo específico definido, é chamada de **View Fortemente Tipada**.

3.6.1. Definindo o Model

A camada **Model** é constituída por classes que fornecem informações e funcionalidades, de acordo com a regra de negócios requerida na aplicação. As classes que constituem esta camada são colocadas, preferencialmente, na pasta **Models** do projeto. O esquema a seguir ilustra sua utilização:



No projeto, vamos incluir uma classe chamada **Produto**. O processo para a inclusão de um model é o mesmo usado para incluir uma classe. Basta selecionar a pasta **Models** e, com o botão direito do mouse, adicionar uma nova classe.

```

namespace Capitulo03.MVC.Models
{
    public class Produto
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
    }
}

```

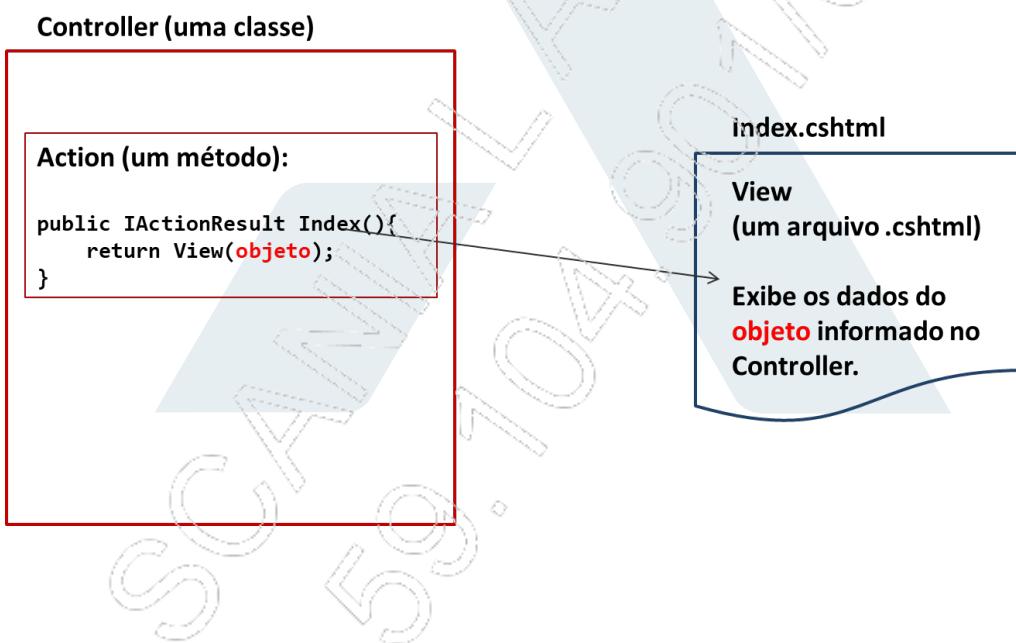
3.6.2. Consumindo o Model no Controller

Para usar o model no controller, devemos decidir como usá-lo. Podemos ter uma lista de objetos, uma simples instância ou outras estruturas. No nosso projeto, vamos adicionar um novo action ao controller **Home**, o **MostrarProduto**.

```
public ActionResult MostrarProduto()  
{  
    Produto produto = new Produto()  
    {  
        Id = 100,  
        Descricao = "Bicicleta",  
        Preco = 1000  
    };  
  
    return View(produto);  
}
```

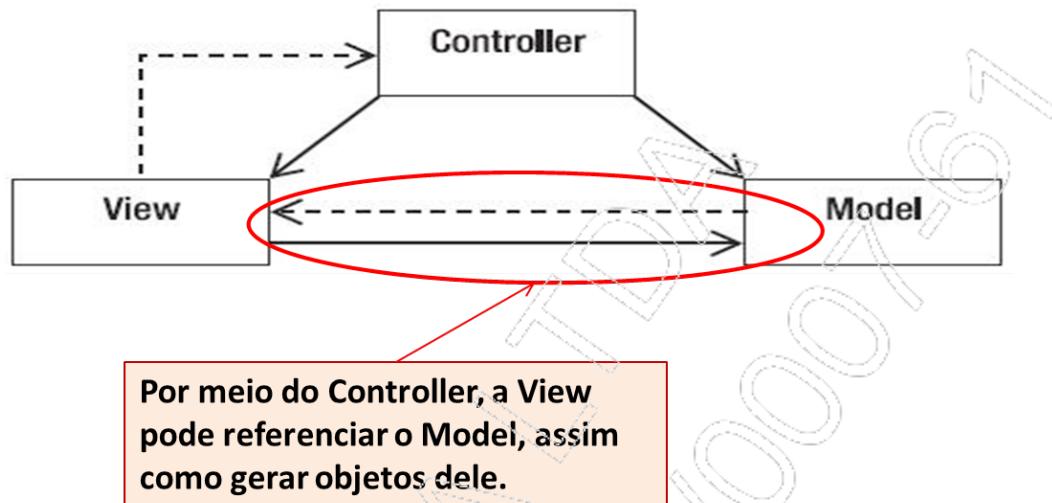
Repare na instância da classe **Produto** sendo passada para a view.

Usamos uma versão sobrecarregada do método **View** para passar o objeto para a view – a versão que recebe um objecto como parâmetro. Veja o esquema a seguir:

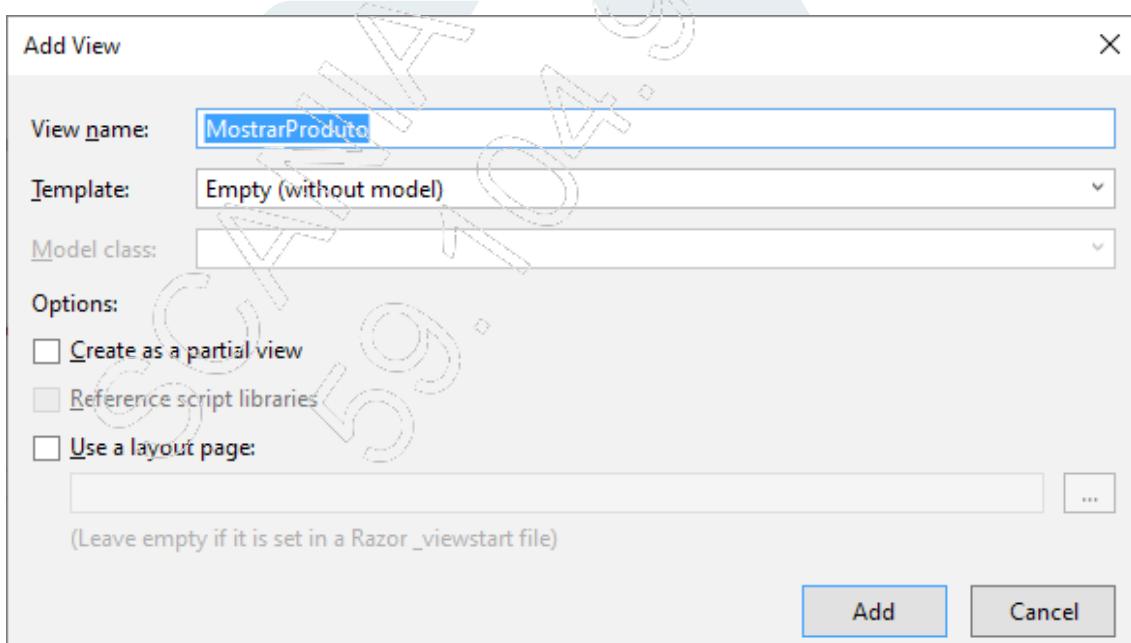


3.6.3. Consumindo o Model na View

Assim como o Controller foi usado para instanciar a classe Produto, a view receberá esta instância e apresentará seus dados para o usuário. O esquema adiante mostra a interação do **Model** com a **View**:



Vamos adicionar uma view para o action **MostrarProduto**. Clique com o botão direito do mouse sobre o método e selecione a opção **Add View** (exatamente como fizemos para a view **Index**):



Para que a view seja tipada, devemos adicionar a seguinte instrução no início da página:

```
@model Capitulo03.MVC.Models.Produto
```

Todas as views, independentemente de serem ou não fortemente tipadas, expõem uma propriedade chamada **Model**. Seu tipo depende da classe usada na tipagem da view (no nosso exemplo, a classe **Produto**).

Sendo assim, nossa view ficará assim:

```
@model Capitulo03.MVC.Models.Produto

{@
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>MostrarProduto</title>
</head>
<body>
    <div>
        <h1>Dados do produto recebido:</h1>
        <div>
            <span><strong>Código:</strong><span>@Model.Id</span><br />
            <span><strong>Descrição:</strong><span>@Model.Descricao</span><br />
            <span><strong>Preço:</strong><span>@Model.Preco.ToString("c")</span><br />
        </div>
    </div>
</body>
</html>
```

A rota executada para esta view, seguindo a convenção do projeto, deverá ser:

<http://localhost:53367/Home/MostrarProduto>

E o resultado da execução:



3.7. Elaboração de formulários

Existem situações em que é necessário o usuário fornecer informações em formulários. Neste caso, estas informações são enviadas também para o Controller.

Na grande maioria das vezes, é utilizado o método GET de uma requisição HTTP, para exibir dados ou enviar parâmetros de consulta, e o método POST, para enviar dados, principalmente quando a finalidade do envio é a inclusão ou alteração de informações.

3.7.1. HTML Helpers

A classe **HtmlHelper** disponibiliza uma série de métodos que podem ser usados para auxiliar a criação dos elementos do formulário:

- **Html.BeginForm**: Inicia um formulário;
- **@Html.AntiForgeryToken**: Cria um campo oculto para validar o request e evitar ataques;
- **@Html.ValidationSummary**: Cria uma tag `` para exibir mensagens de validação;
- **@Html.Label, @Html.LabelFor**: Cria uma legenda para um campo;
- **@Html.TextBox, @Html.TextBoxFor**: Cria uma caixa de texto;
- **@Html.Password, @Html.PasswordFor**: Cria uma caixa de texto do tipo senha;
- **@Html.ValidationMessage, @Html.ValidationMessageFor**: Cria um span para exibir mensagens de validação;
- **@Html.CheckBox, @Html.CheckBoxFor**: Cria uma caixa do tipo CheckBox;
- **@Html.ActionLink**: Cria links para métodos definidos nas classes do tipo Controller;
- **@Html.Hidden, @Html.HiddenFor**: Cria um campo de formulário oculto;
- **@Html.DropDownList, @Html.DropDownListFor**: Cria um DropDownList.

Esses métodos não são de uso obrigatório e servem apenas para facilitar a escrita do código HTML e para evitar a revisão do código quando algum modelo mudar.

Os formulários podem ser escritos com tags HTML puras, mas vale a pena nos beneficiarmos dos HTML Helpers.

Na lista de componentes que apresentamos, alguns foram escritos em duplicidade, sendo o segundo terminado com **For**, como foi o caso de **@Html.Label** e **@Html.LabelFor**.

Visualmente, os componentes duplicados desta forma apresentam o mesmo resultado, mas são bastante diferentes na sua utilização. Os componentes SEM o sufixo **For** não utilizam os **Models** e, portanto, não constituem uma view fortemente tipada. Já os componentes COM o prefixo **For**, quando aplicados, utilizam as propriedades do Model através de expressões lambda, e são portanto aplicados a views fortemente tipadas.

Vamos apresentar um exemplo de cada tipo.

3.7.2. Formulário não tipado

Para ilustrar o uso de formulários, vamos incluir um action chamado **IncluirProduto** no controller **Home**. Este controller deverá ter apenas a chamada à **View()**:

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

Quando executado, este action apenas apresentará a view com os dados do formulário. Vamos gerar a view para este action, da mesma forma que fizemos nos actions anteriores. Veja o conteúdo do formulário, usando os **HTML Helpers**:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>IncluirProduto</title>
</head>
<body>
    <div>
        <h1>Cadastro de Produtos (simulação)</h1>
        @using (Html.BeginForm())
        {
            @Html.Label("Código")<br />
            @Html.TextBox("id")<br />

            @Html.Label("Descrição")<br />
            @Html.TextBox("descricao")<br />

            @Html.Label("Preço")<br />
            @Html.TextBox("preco")<br />

            <input type="submit" value="Enviar" />
        }
    </div>
</body>
</html>
```

Os valores passados como primeiro parâmetro para os elementos `Html.TextBox` representam seu **nome**. Por exemplo, o elemento `Html.TextBox("codigo")` será renderizado como:

```
<input type="text" name="descricao" id="descricao" />
```

Por padrão, o elemento `Html.BeginForm()` renderiza um elemento **form**, e envia seus dados usando o método **POST**.

Para que o controller receba os dados da view através de uma requisição POST, deve haver um action apropriado. Vamos escrever este action:

```
[HttpPost]
public ActionResult IncluirProduto/FormCollection form)
{
    //conteúdo do action
    return View();
}
```

Temos duas informações importantes:

- O atributo **[HttpPost]**: Deve ser incluído em actions que recebe requisições de formulários através do método POST. Quando não informado, o padrão é **[HttpGet]**;
- O parâmetro **FormCollection**: Este parâmetro define uma coleção que conterá os elementos do formulário, acessíveis através do seu nome. Independentemente do tipo desejado para o campo, o retorno através do **FormCollection** será sempre uma string e, se outro tipo for desejado, será necessária uma conversão.

Sendo assim, continuaremos com a implementação do action:

```
[HttpPost]
public ActionResult IncluirProduto/FormCollection form)
{
    //recebendo os dados do formulário
    var id = Convert.ToInt32(form["id"]);
    var descricao = form["descricao"];
    var preco = Convert.ToDouble(form["preco"]);

    //criando um objeto Produto com estes dados
    var produto = new Produto()
    {
        Id = id,
        Descricao = descricao,
        Preco = preco
    };

    return View("MostrarProduto", produto);
}
```

O parâmetro **form** usa como índice o nome do componente recebido do formulário.

Criamos uma instância da classe **Produto** com os dados lidos do formulário e, em seguida, usamos uma outra versão do método **View**, que recebe não só o objeto, mas o nome da view que exibirá estes dados.

Vale mencionar que, quando usamos a versão do método **View** que recebe o nome do action como primeiro parâmetro (de acordo com nosso exemplo), a requisição é a mesma, só o conteúdo que foi transferido para outra view:

```
return View("MostrarProduto", produto);
```

A execução desta aplicação consiste em executar o action **IncluirProduto** na rota, o que chamará o método GET. Quando o formulário estiver disponível, deverá ser preenchido e, ao ser enviado de volta para o controller, este executará o action **IncluirProduto**, com o parâmetro **FormCollection**, que apresentará seus dados para o usuário:

The figure consists of two side-by-side screenshots of a web browser window. Both windows have a title bar 'IncluirProduto' and a URL 'localhost:53367/Home/IncluirProduto'.
The top screenshot shows a blank form with three input fields labeled 'Código', 'Descrição', and 'Preço', each with a corresponding empty text input box. Below the inputs is a blue 'Enviar' button.
The bottom screenshot shows the same form but with data entered: 'Código' is '120', 'Descrição' is 'Mochila', and 'Preço' is '199'. The 'Enviar' button is also visible. Both screenshots are overlaid with a large, faint watermark containing the text 'SCAMILLA 704.907'.



3.7.3. Formulário fortemente tipado

O formulário apresentado no tópico anterior não possui nenhum vínculo com nenhuma classe em específico. Nós criamos uma instância da classe **Produto** com os dados do formulário de forma arbitrária, ou seja, poderíamos usar quaisquer informações para criar este objeto, como poderíamos criar objetos de outras classes, ou mesmo nem criar objetos.

Um formulário fortemente tipado utiliza uma classe como modelo, e os componentes representando os campos de entrada possuem um vínculo com as propriedades desta classe, através de expressões lambda.

Vamos reproduzir o mesmo exemplo, desta vez definindo um modelo. Para tanto, adicione o action **IncluirProdutoModel** no controller **Home**:

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

Vamos gerar a view, usando os mesmos procedimentos. Veja o conteúdo da view:

```
@model Capitulo03.MVC.Models.Produto
@{
    Layout = null;
}
```

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>IncluirProdutoModel</title>
</head>
<body>
    <div>
        <h1>Cadastro de Produtos - model (simulação)</h1>
        @using (Html.BeginForm())
        {
            @Html.LabelFor(m => m.Id)<br />
            @Html.TextBoxFor(m => m.Id)<br />

            @Html.LabelFor(m => m.Descricao)<br />
            @Html.TextBoxFor(m => m.Descricao)<br />

            @Html.LabelFor(m => m.Preco)<br />
            @Html.TextBoxFor(m => m.Preco)<br />

            <input type="submit" value="Enviar" />
        }
    </div>
</body>
</html>
```

O componente **Html.LabelFor** apresentará como label a própria descrição da propriedade usada na expressão lambda. É possível mudar este valor adicionando **Data Annotations**. Mais adiante neste capítulo apresentaremos os **Data Annotations**.

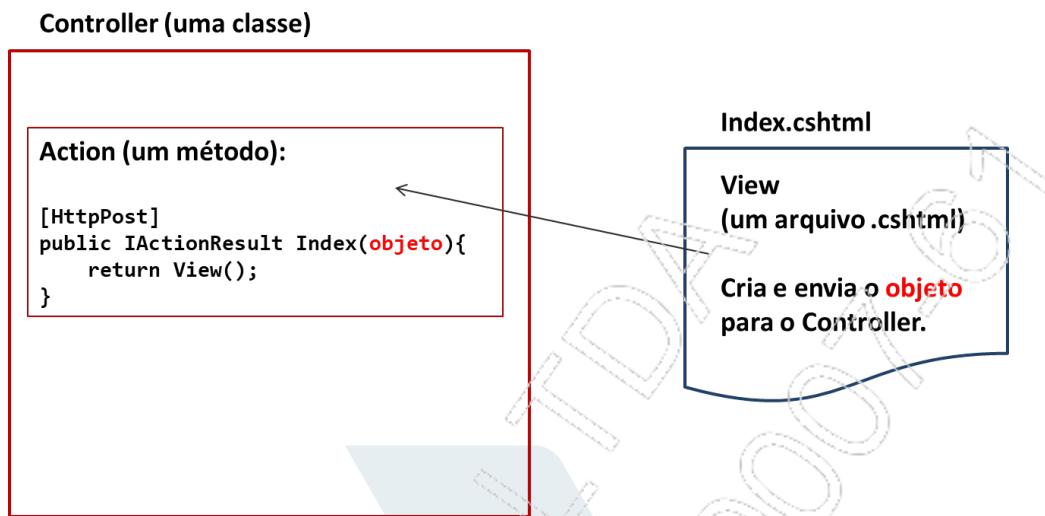
Já o componente **Html.TextBoxFor** permite atribuir o conteúdo do campo de entrada à propriedade definida como expressão lambda.

Quando o conteúdo do formulário é enviado para o controller, o MVC usa os campos de entrada como valores das propriedades referenciadas para criar a instância da classe especificada no model (no nosso exemplo, a classe **Produto**).

Vamos definir a versão POST do action:

```
[HttpPost]
public ActionResult IncluirProdutoModel(Produto produto)
{
    return View("MostrarProduto", produto);
}
```

Observe que não foi necessário criar um objeto, pois ele já existe! O action recebe um produto como parâmetro, e este é gerado na view. O processo é ilustrado no esquema a seguir:



O resultado da execução produz o mesmo resultado que o formulário definido anteriormente, usando **FormCollection**. A diferença está na estrutura usada: uma estrutura usando uma **view fortemente tipada**. O único detalhe é quanto ao texto apresentado, proveniente dos componentes Label:



Como já mencionado, apresentaremos a solução para este e outros problemas no próximo tópico.

3.8. Validação e filtros

Validar dados é uma tarefa que exige uma atenção especial em qualquer aplicação. O processo de verificar e manter a integridade dos dados engloba diversas áreas, como verificação de tipos de dados, limites, credenciais, criptografia, certificados, meios de transporte de informação e formatos. Uma parte desse processo é a validação dos dados que o usuário digita.

3.8.1. Validação

O processo de validação de campos consiste em verificar os seguintes critérios:

- **Tipo de dados:** Verificar se o campo é do tipo **Data**, **Inteiro**, **Booleano**, **Texto** ou **Moeda** e se o conteúdo do campo é compatível, de acordo com a **CultureInfo** da aplicação;
- **Faixa:** Verificar se o valor está dentro de uma faixa aceitável. Por exemplo, o dia do mês deve ser um número entre 1 e 31, a data final de um relatório deve ser maior ou igual à data inicial, e assim por diante;
- **Formato:** Alguns itens seguem convenções de formato. Por exemplo, a sigla do estado deve ser duas letras maiúsculas.

A validação deve ser feita no cliente, se possível, e sempre no servidor. A validação no cliente geralmente utiliza JavaScript, que é facilmente interceptado e manipulado, sendo, portanto, muito pouco seguro. Uma vez que os dados chegaram no servidor, devem ser verificados quanto à sua integridade e aplicadas as regras de negócio necessárias.

3.8.2. Data Annotation

O MVC utiliza uma tecnologia chamada **Data Annotation**, que se resume em declarar as regras que devem ser aplicadas ao Model. Considere a classe **Produto** que usamos no nosso projeto (ela será usada para validação):

```
namespace Capitulo03.MVC.Models
{
    public class Produto
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
    }
}
```

Os validadores são aplicados quando os dados do formulário são enviados para o servidor. Antes de apresentar os principais validadores, apresentaremos uma propriedade muito importante: **ModelState**.

3.8.2.1. ModelState

Ao clicar no botão **Enviar**, os dados do formulário são enviados para a URL informada. O método que receberá os dados do formulário deverá ter as seguintes características:

1. Deve ser marcado com o atributo **HttpPost**;

2. Deve receber como parâmetro:

- uma instância do modelo,
- parâmetros iguais aos campos do formulário. ou
- uma coleção do tipo **FormCollection**.

A melhor estratégia é receber uma instância do modelo;

3. Deve ser verificado se o modelo é válido por meio da propriedade **IsValid** da classe **ModelStateDictionary**, disponibilizada por meio da propriedade **ModelState** da classe **Controller**.

A classe **ModelStateDictionary** (disponibilizada por meio da propriedade **ModelState**) contém todos os dados de validação de um formulário, incluindo cada campo, se é válido ou não e, se não for, a mensagem de erro correspondente.

Nos subtópicos seguintes, veremos os principais validadores.

3.8.2.2. Required

Para tornar os campos obrigatórios, é necessário incluir o atributo **Required** neles. O namespace em que residem as classes de anotações usadas pelo MVC é o **System.ComponentModel.DataAnnotations**.

```
using System.ComponentModel.DataAnnotations;

namespace Capitulo03.MVC.Models
{
    public class Produto
    {
        [Required]
        public int Id { get; set; }

        [Required]
        public string Descricao { get; set; }

        [Required]
        public double Preco { get; set; }
    }
}
```

Na view, adicionamos o elemento `@Html.ValidationSummary()`:

```
@model Capitulo03.MVC.Models.Produto
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>IncluirProdutoModel</title>
</head>
<body>
    <div>
        <h1>Cadastro de Produtos - model (simulação)</h1>

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

            @Html.LabelFor(m => m.Id)<br />
            @Html.TextBoxFor(m => m.Id)<br />

            @Html.LabelFor(m => m.Descricao)<br />
            @Html.TextBoxFor(m => m.Descricao)<br />

            @Html.LabelFor(m => m.Preco)<br />
            @Html.TextBoxFor(m => m.Preco)<br />

            <input type="submit" value="Enviar" />
        }
    </div>
</body>
</html>
```

Como já apresentado, não basta incluir este método na view. É necessário verificar se o modelo é valido quando for enviado para o controller. Usamos para esta tarefa a propriedade **ModelState** e, através dela, a propriedade **IsValid**. A propriedade **IsValid** retorna **true** se o modelo for válido, de acordo com os **Data Annotations** configurados no Model, e **false** quando alguma especificação falhar.

```
[HttpPost]
public ActionResult IncluirProdutoModel(Produto produto)
{
    if (!ModelState.IsValid)
    {
        return View();
    }
    return View("MostrarProduto", produto);
}
```

Para testar, executaremos a aplicação e enviaremos o formulário, sem nenhum dado preenchido. O resultado é mostrado a seguir:



É possível mostrar a mensagem de validação junto do TextBox usando o método **ValidationMessageFor**, disponibilizado pelo helper HTML.

```
@model Capitulo03.MVC.Models.Produto
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>IncluirProdutoModel</title>
</head>
<body>
    <div>
        <h1>Cadastro de Produtos - model (simulação)</h1>

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

            @Html.LabelFor(m => m.Id)<br />
            @Html.TextBoxFor(m => m.Id)
            @Html.ValidationMessageFor(m => m.Id)<br />

            @Html.LabelFor(m => m.Descricao)<br />
            @Html.TextBoxFor(m => m.Descricao)
            @Html.ValidationMessageFor(m => m.Descricao)<br />

            @Html.LabelFor(m => m.Preco)<br />
            @Html.TextBoxFor(m => m.Preco)
            @Html.ValidationMessageFor(m => m.Preco)<br />

            <input type="submit" value="Enviar" />
        }
    </div>
</body>
</html>
```

A execução da aplicação e o envio do formulário sem nada preenchido produz o resultado:

The screenshot shows a browser window titled "IncluirProdutoModel" with the URL "localhost:53367/Home/IncluirProdutoModel". The page displays an error message: "Cadastro de Produtos - model (simulação)". It lists three validation errors: "O campo Id é obrigatório.", "O campo Descricao é obrigatório.", and "O campo Preco é obrigatório.". Below the errors, there is a form with three input fields: "Id", "Descricao", and "Preco", each containing the error message. A blue "Enviar" button is at the bottom.

Vamos apresentar outros atributos.

3.8.2.3. StringLength

O atributo **StringLength** define o tamanho total permitido para um texto. É muito útil para não truncar informações ao gravar no banco de dados.

```
namespace Capitulo03.MVC.Models
{
    public class Produto
    {
        [Required]
        public int Id { get; set; }

        [Required]
        [StringLength(30)]
        public string Descricao { get; set; }

        [Required]
        public double Preco { get; set; }
    }
}
```

É possível alterar a mensagem padrão, definindo o atributo **ErrorMessage**:

```
[StringLength(30, ErrorMessage="Tamanho máximo: 30 caracteres")]
```

3.8.3. Regular Expression

Usar expressões regulares para validar formatos de entrada é uma das maneiras mais versáteis que existe. Pode ser usada para qualquer informação que tenha um formato definido: e-mail, CPF, CNPJ, IP, placa de carro, sigla de estado etc. No exemplo adiante, a classe **Veículo** aceita no campo **Placa** apenas os formatos corretos:

```
namespace Capitulo03.MVC.Models
{
    public class Veiculo
    {
        [RegularExpression("[A-Z]{3}[0-9]{4}")]
        [Required]
        public string Placa { get; set; }

        [Required]
        public int Ano { get; set; }

        [Required]
        public string Modelo { get; set; }
    }
}
```

3.8.4. Range

O atributo **Range** define os limites de campos numéricos. Os parâmetros a serem passados são o valor mínimo e o máximo. A faixa válida inclui esses valores. O exemplo a seguir exibe um uso típico: o valor mínimo e máximo para a entrada de um dia do mês.

```
namespace Capitulo03.MVC.Models
{
    public class Pagamento
    {
        [Range(1, 31)]
        public int Mes { get; set; }

        public int Ano { get; set; }

        public decimal Valor { get; set; }
    }
}
```

3.8.5. Remote

O atributo **Remote** permite realizar validações personalizadas no servidor. Por exemplo, em um programa de vendas, talvez seja necessário consultar o estoque ao informar uma quantidade. Isso só é possível fazendo uma chamada ao servidor.

```
[Remote("VerificarEstoque", ErrorMessage="Estoque Insuficiente")]
public int Quantidade { get; set; }
```

O atributo **Remote** espera passar o nome de um método que retorne um valor booleano, encapsulado em um objeto JSON.

```
public JsonResult VerificarEstoque(int quantidade)
{
    bool = ObterInfoEstoque(quantidade, produtoId);
    return Json(false, JsonRequestBehavior.AllowGet);
}
```

3.8.6. Compare

O atributo **Compare** verifica se duas propriedades de um modelo são iguais. É usado tipicamente para confirmação de dados:

```
[Compare("Senha")]
public string RepetirSenha { get; set; }
```

Observe que, em uma das versões no MVC, existe um conflito entre **Remote** e **Compare**. O atributo **Compare** estava definido nos namespaces **System.Web.Mvc** e **System.ComponentModel.DataAnnotations**. Não é um problema sério, pois, se isso acontecer, basta referenciar uma das classes pelo nome completo:

```
[System.Web.Mvc.Compare("Senha")]
public string RepetirSenha { get; set; }
```

3.8.6.1. Display

Alguns atributos interagem com o método **LabelFor** e **TextBoxFor** do helper HTML. O atributo **Display** determina qual texto será exibido no lugar do nome da propriedade:

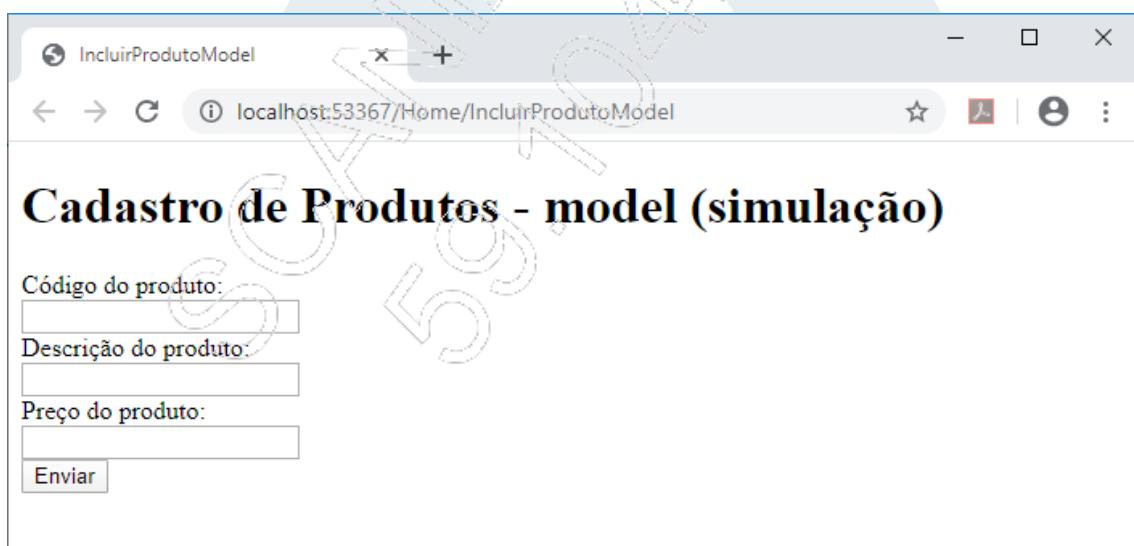
- **Model**

```
namespace Capitulo03.MVC.Models
{
    public class Produto
    {
        [Required]
        [Display(Name = "Código do produto")]
        public int Id { get; set; }

        [Required]
        [Display(Name = "Descrição do produto")]
        [StringLength(30)]
        public string Descricao { get; set; }

        [Required]
        [Display(Name = "Preço do produto")]
        public double Preco { get; set; }
    }
}
```

- **Resultado**



A classe de atributo **Display** define as seguintes propriedades:

- **AutoGeneratedField**

Indica se o campo deve ser gerado automaticamente. Isso faz com que o campo fique somente leitura.

- **Description**

É a descrição do campo. Esta propriedade não interage automaticamente com os helpers **TextBoxFor** ou **LabelFor**, mas é fácil construir uma extensão do helper HTML para exibir a descrição. É uma das vantagens de se construir um sistema aberto, que pode ser expandido:

```
using System;
using System.Linq.Expressions;
using System.Web;
using System.Web.Mvc;

namespace Capitulo03.MVC
{
    public static class HtmlExtensions
    {
        public static IHtmlString DescriptionFor<TModel, TValue>(
            this HtmlHelper<TModel> html,
            Expression<Func<TModel, TValue>> expression)
        {
            var metadata = ModelMetadata
                .FromLambdaExpression(expression, html.ViewData);
            var description = metadata.Description;

            return new HtmlString(description);
        }
    }
}
```

Essa classe cria o método de extensão **DescriptionFor**, que pode ser usado para exibir uma descrição do campo, por exemplo, usando a propriedade **Title**.

ASP.NET MVC - Controllers, Models e Views

No caso do helper para exibir a descrição como um **toolTip**, pode ser usado este modelo:

```
@model Capitulo03.MVC.Models.Produto
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>IncluirProdutoModel</title>
</head>
<body>
    <div>
        <h1>Cadastro de Produtos - model (simulação)</h1>

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

            @Html.LabelFor(m => m.Id)<br />
            @Html.TextBoxFor(m => m.Id)
            @Html.ValidationMessageFor(m => m.Id)<br />

            @Html.LabelFor(m => m.Descricao)<br />
            @Html.TextBoxFor(
                m => m.Descricao,
                new { title = Html.DescriptionFor(m => m.Descricao) })

            @Html.ValidationMessageFor(m => m.Descricao)<br />

            @Html.LabelFor(m => m.Preco)<br />
            @Html.TextBoxFor(m => m.Preco)
            @Html.ValidationMessageFor(m => m.Preco)<br />

            <input type="submit" value="Enviar" />
        }
    </div>
</body>
</html>
```

No modelo, se a descrição estiver definida, será exibida como **toolTip** (propriedade **Title** do HTML), conforme mostrado adiante:

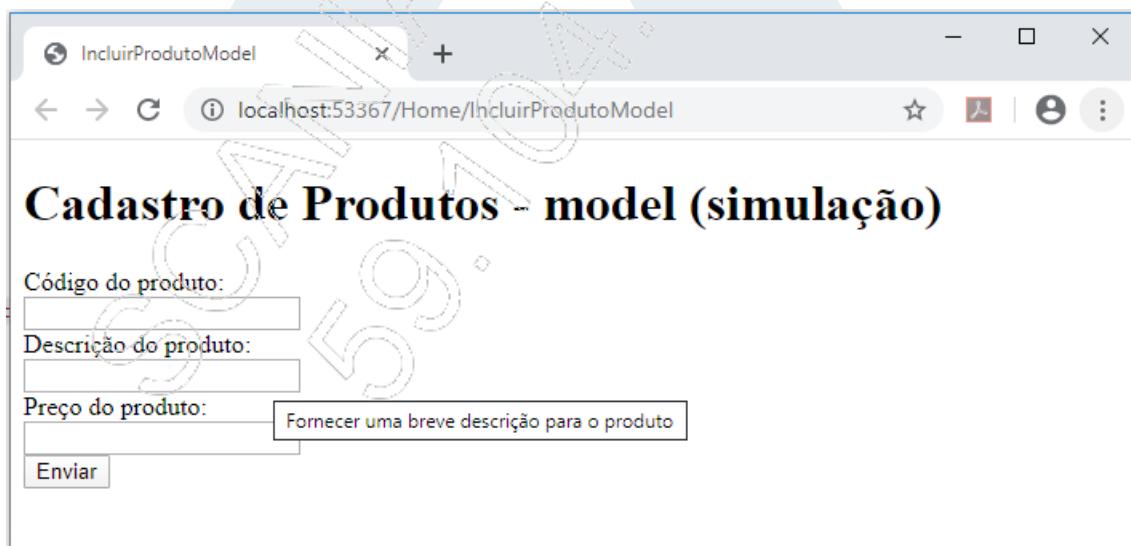
```
using System.ComponentModel.DataAnnotations;

namespace Capitulo03.MVC.Models
{
    public class Produto
    {
        [Required]
        [Display(Name = "Código do produto")]
        public int Id { get; set; }

        [Required]
        [Display(
            Name = "Descrição do produto:",
            Description = "Fornecer uma breve descrição para o produto")]
        [StringLength(30)]
        public string Descricao { get; set; }

        [Required]
        [Display(Name = "Preço do produto")]
        public double Preco { get; set; }
    }
}
```

Ao mover o mouse sobre o campo de entrada **Descrição**, temos o **toolTip**:



- **GroupName**

Uma string usada para agrupar campos na interface.

- **Name**

O nome que aparece na legenda do campo. Se não for informado, o nome da propriedade é exibido.

- **Order**

A ordem em que esse campo aparece. Não precisa ser numerado exatamente. Cada número atribuído tem um peso. Esta propriedade é utilizada quando o formulário de um Model é criado automaticamente.

- **Prompt**

Texto usado como marca d'água nas caixas de texto. Quando o usuário começa a digitar, esse texto some. Alguns componentes de interface implementam essa funcionalidade.

- **ResourceType**

Retorna o tipo (**Type**) do modelo usado nas propriedades **Name**, **Description** e **Display**.

- **ShortName**

O nome usado quando a propriedade é exibida em uma tabela, nos títulos das colunas. Por padrão, o nome da propriedade é exibido ou o texto definido no parâmetro **Name** do atributo **Display**.

- **ReadOnly**

O atributo **ReadOnly** bloqueia o campo para edição nos métodos **EditorFor** e **TextBoxFor** do helper HTML.

3.8.6.2. DisplayFormat

Este atributo define o formato que será usado para exibir os dados de um campo.

```
namespace Capitulo03.MVC.Models
{
    public class Pagamento
    {
        [Range(1, 31)]
        public int Mes { get; set; }

        public int Ano { get; set; }

        [DisplayFormat(DataFormatString = "{0:c}")]
        public decimal Valor { get; set; }
    }
}
```

Em uma view, ao utilizar um método para exibir dados do produto, o formato é aplicado automaticamente.

3.9. Layout

No Capítulo 2, apresentamos o conceito de layout quando abordamos o Razor. Para recordar, um layout é uma página comum a todas as views da aplicação, ou de parte dela. Quando uma view é chamada, seu conteúdo é inserido no layout na parte que contém a chamada a `@RenderBody()`.

No ASP.NET MVC o processo é o mesmo, porém de forma mais otimizada.

Vamos apresentar o processo de criação de uma view utilizando layout. Para isso, vamos adicionar um novo action no controller **Home** chamado **IncluirProdutoLayout**. Sua funcionalidade será idêntica a **IncluirProdutoModel**:

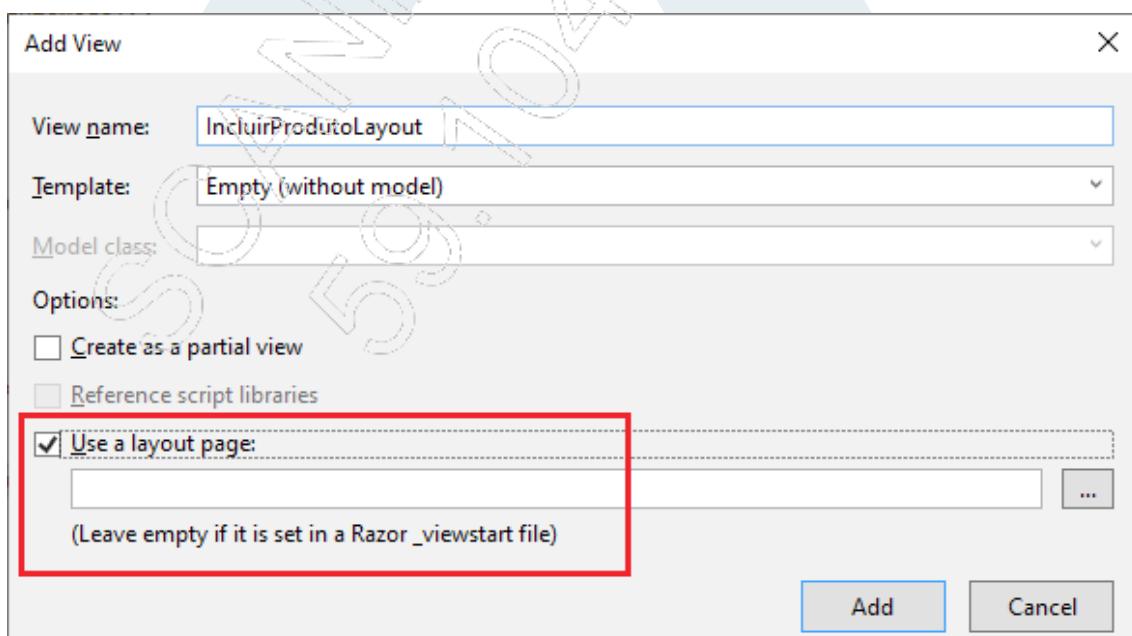
- **No controller**

Primeiro, vamos adicionar o action no controller, que será responsável por apresentar a view:

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

- **Na View**

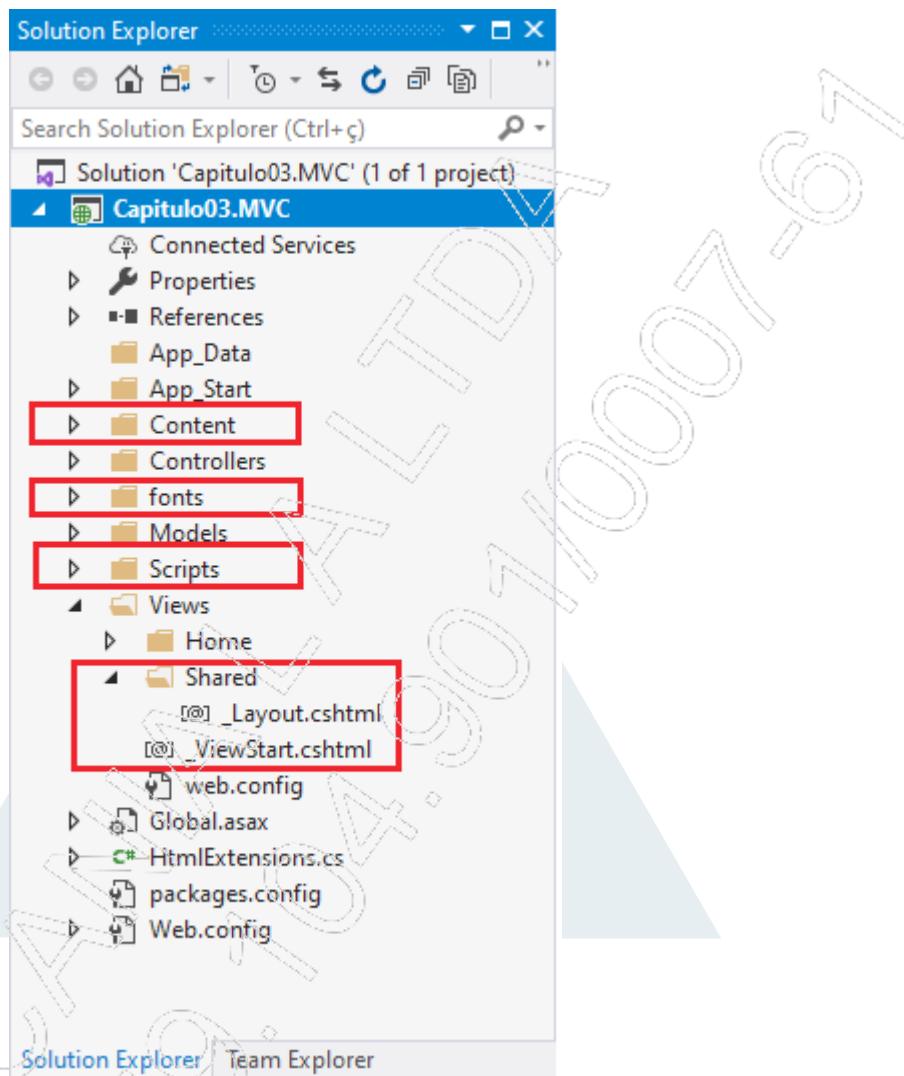
Quando criarmos a view (clicando com o botão direito do mouse sobre o método), deixaremos marcada a opção para incluir uma página de layout:



ASP.NET MVC - Controllers, Models e Views

Observe que não definimos nenhum arquivo de layout, embora possamos selecionar um arquivo que tenhamos definido.

Quando fizemos esta seleção pela primeira vez, o Visual Studio adicionou diversos arquivos e pastas no projeto:



Foi adicionado o arquivo `_ViewStart.cshtml` na pasta `Views`, e o arquivo `_Layout.cshtml` na pasta `Views/Shared`.

O arquivo `_ViewStart.cshtml` contém o caminho do layout:

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

E o conteúdo do arquivo `_Layout.cshtml` é:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link href("~/Content/Site.css" rel="stylesheet" type="text/css" />
    <link href("~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />
    <script src "~/Scripts/modernizr-2.8.3.js"></script>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle"
                    data-toggle="collapse"
                    data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home",
                    new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                </ul>
            </div>
        </div>
        <div class="container body-content">
            @RenderBody()
            <hr />
            <footer>
                <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
            </footer>
        </div>

        <script src "~/Scripts/jquery-3.3.1.min.js"></script>
        <script src "~/Scripts/bootstrap.min.js"></script>
    </body>
</html>
```

Na view **IncluirProdutoLayout.cshtml**, temos apenas o conteúdo necessário para ser inserido no layout:

```
@{  
    ViewBag.Title = "IncluirProdutoLayout";  
}  
  
<h2>IncluirProdutoLayout</h2>
```

Após incluir o conteúdo do formulário, teremos:

```
@model Capitulo03.MVC.Models.Produto  
  
{@  
    ViewBag.Title = "IncluirProdutoLayout";  
}  
  
<h1>Cadastro de Produtos - model (simulação)</h1>  
  
@using (Html.BeginForm())  
{  
    @Html.ValidationSummary()  
  
    @Html.LabelFor(m => m.Id)  
    <br />  
    @Html.TextBoxFor(m => m.Id)  
    @Html.ValidationMessageFor(m => m.Id)  
    <br />  
  
    @Html.LabelFor(m => m.Descricao)  
    <br />  
    @Html.TextBoxFor(  
        m => m.Descricao,  
        new { title = Html.DescriptionFor(m => m.Descricao) })  
  
    @Html.ValidationMessageFor(m => m.Descricao)  
    <br />  
  
    @Html.LabelFor(m => m.Preco)  
    <br />  
    @Html.TextBoxFor(m => m.Preco)  
    @Html.ValidationMessageFor(m => m.Preco)  
    <br />  
  
    <input type="submit" value="Enviar" />  
}
```

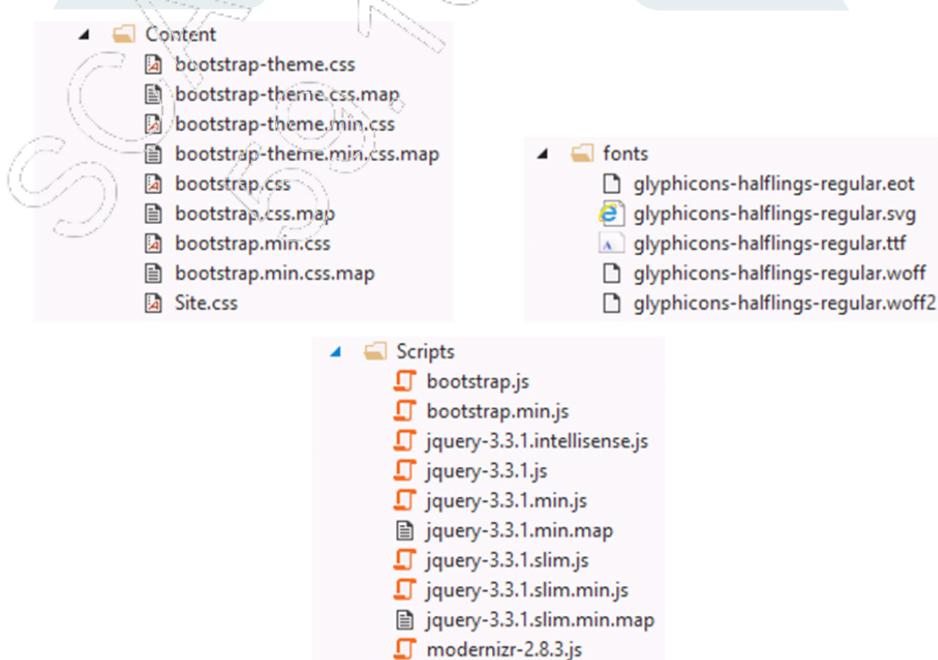
A execução desta view apresenta o seguinte resultado:



O layout utiliza o bootstrap como framework para a camada de visualização, além das funcionalidades JavaScript agregadas ao bootstrap. Os arquivos do bootstrap estão na pasta criada **Content**, e os arquivos JavaScript, incluindo o JQuery e o próprio arquivo de script do bootstrap, estão na pasta **Scripts**. Na pasta **fonts** estão os ícones do bootstrap.

É importante verificar a versão do bootstrap usada pelo ASP.NET MVC, que é 3.0. Neste caso, se houver a necessidade de usar uma versão mais atual, será necessário refatorar o layout, para não haver perdas de formatação.

Expandindo as pastas **Content**, **fonts** e **Scripts**, temos:



3.9.1. Fluxo de execução do MVC com o layout

Na nossa Web Page `IncluirProdutoLayout.cshtml`, não fizemos nenhuma referência ao arquivo de layout. Este arquivo já está referenciado em `_ViewStart.cshtml`.

Isso se deve ao fluxo de execução do MVC:

- Para cada view chamada, ocorre primeiro a execução de `_ViewStart.cshtml`;
- `_ViewStart.cshtml` faz referência a `_Layout.cshtml` que, por sua vez, tem a chamada a `@RenderBody()`, `@RenderSection()`, entre outros;
- `_Layout.cshtml` inclui o conteúdo da view chamada no início do fluxo;
- O resultado é apresentado para o usuário.

Com exceção de `IncluirProdutoLayout.cshtml`, todas as views do projeto não utilizaram o layout. Para não usá-lo, basta sobrescrever, na view desejada, a propriedade `Layout` como:

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

Se quisermos usar um layout diferente na view, basta atribuirmos o caminho do layout desejado a esta propriedade.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

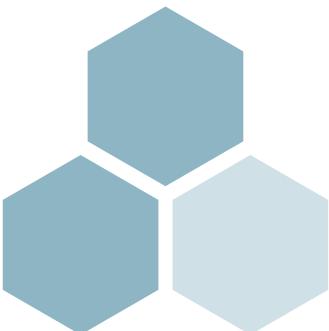
- Usamos as rotas para combinar controllers e actions de modo a produzir uma URL capaz de executar a aplicação;
- A execução de um action no controller produz o resultado visual para o usuário. É por meio do action que definimos o que será apresentado para o usuário, de forma dinâmica;
- Para definirmos um controller, devemos criar uma classe com o nome do controller desejado, mas terminando com o sufixo **Controller**;
- As views devem ficar na pasta **Views**, ou em alguma subpasta. De acordo com a convenção do MVC, as views definidas nos controllers devem ficar em uma subpasta com o mesmo nome do controller, e as views de propósito geral, na subpasta **Shared**;
- Usamos os **Models** para especificar o tipo de objeto que uma view irá representar. Podemos passar objetos do model do controller para a view, bem como criá-los na view e enviá-los para o controller realizar seu processamento;
- Formulários não tipados são aqueles cujo conteúdo não possui nenhum vínculo com algum model; já os formulários tipados possuem seus componentes vinculados a uma determinada classe, que é o model;
- Podemos efetuar validações de campos de entrada e formatações de saída por meio dos validadores e dos filtros disponíveis no MVC.



3

ASP.NET MVC – Controllers, Models e Views

Teste seus conhecimentos



1. Em uma aplicação MVC, em qual pasta normalmente ficam os arquivos de banco de dados?

- a) AppData
- b) AppDataBase
- c) App_Code
- d) App_Data
- e) AppCode

2. No arquivo Global.asax, uma rota é iniciada pelo método:

- a) Application_Start
- b) RegisterRoutes
- c) RouteConfig
- d) RouteTable
- e) Roputes

3. Quando um objeto é enviado do controller para uma view fortemente tipada, dizemos que este objeto faz parte da camada:

- a) Model
- b) View
- c) ViewBag
- d) Rota
- e) Controller

4. Para que uma aplicação MVC siga as convenções usadas neste tipo de aplicação, o controller deve iniciar com que nome?

- a) Controller
- b) ControllerBase
- c) Action
- d) O nome da classe
- e) O nome do controller

5. Em qual namespace se encontra a superclasse Controller?

- a) System.Net
- b) System.Mvc
- c) System.Web.Mvc
- d) System.Web.Http
- e) System.Http



3

ASP.NET MVC – Controllers, Models e Views



Mãos à obra!



Objetivos:

A partir deste laboratório, iniciaremos o desenvolvimento de um projeto ASP.NET MVC.

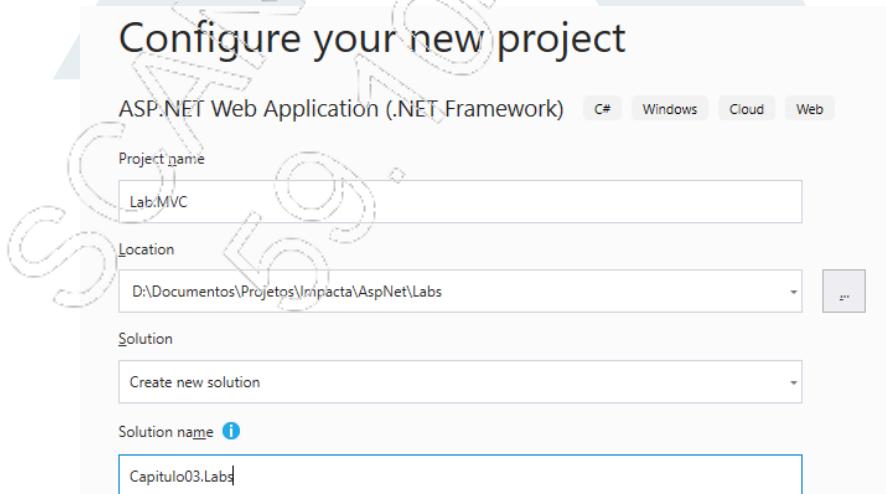
O projeto consistirá em um sistema de cadastro de clientes e de produtos, além da venda de produtos para clientes. Como recursos, será possível:

- Listar, alterar e remover clientes;
- Listar alterar e remover produtos;
- Listar pedidos por cliente;
- Listar itens de pedido;
- Efetuar o pagamento de um pedido via cartão de crédito (simulação) através de um Webservice.

Neste laboratório, iniciaremos pela criação do projeto, com a estrutura necessária para que ele se torne funcional e com as rotas para as demais operações.

Laboratório 1

1. Na pasta **Labs**, crie um novo projeto chamado **Lab.MVC**. Use como solution o nome **Capítulo03.Labs**. O projeto deverá ser do tipo **ASP.NET Web Application**, vazio, com as referencias do **MVC**:



Create a new ASP.NET Web Application

Empty
An empty project template for creating ASP.NET applications. This template does not have any content in it.

Web Forms
A project template for creating ASP.NET Web Forms applications. ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access.

MVC
A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Web API
A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

Authentication
No Authentication
Change

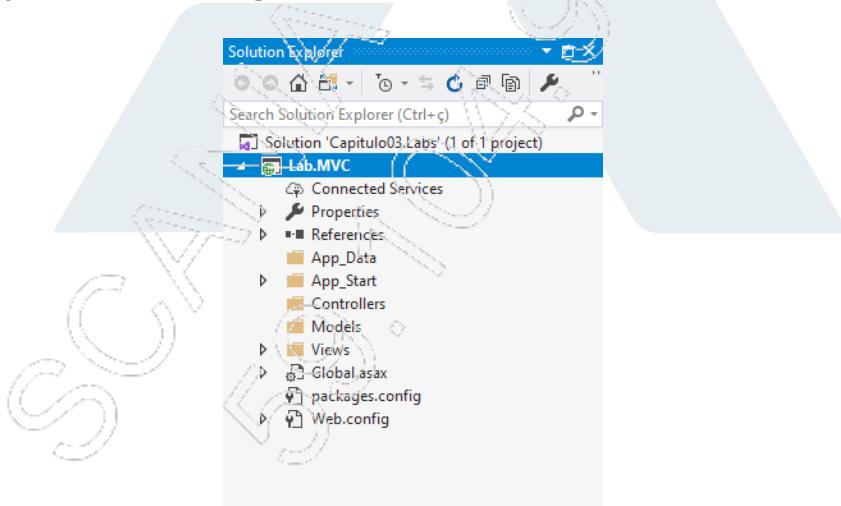
Add folders & core references

- Web Forms
- MVC
- Web API

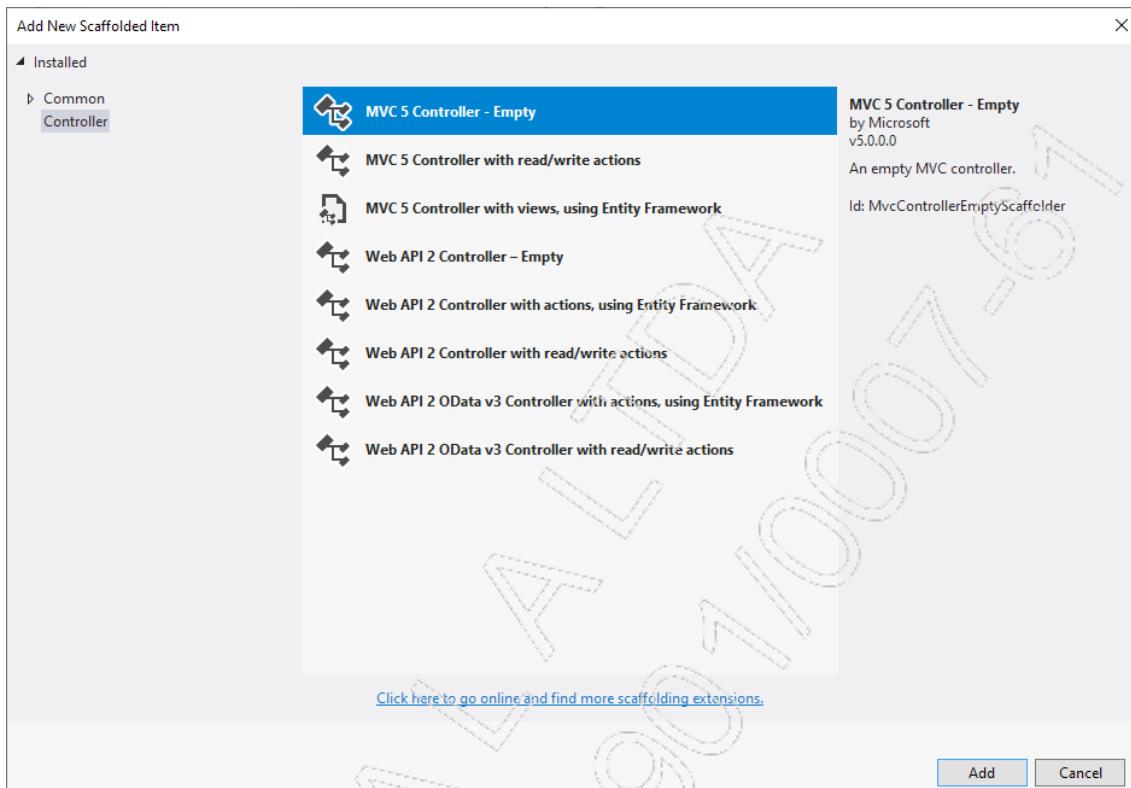
Advanced

- Configure for HTTPS
- Docker support

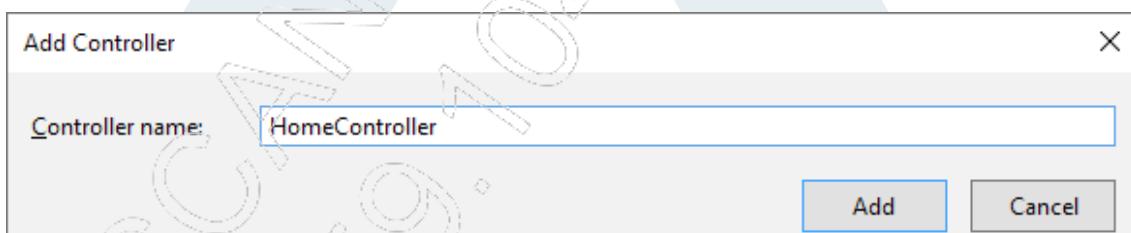
2. Nosso projeto possui a seguinte estrutura:



3. Vamos definir o controller que representará a página inicial da aplicação. Na pasta **Controllers**, clique com o botão direito do mouse, selecione a opção **Add / Controller...** e escolha o item **MVC 5 Controller - Empty**:



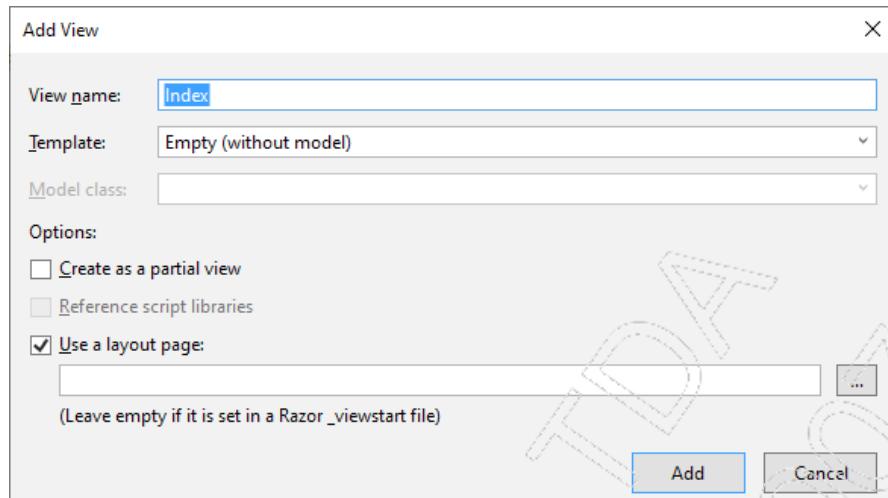
4. Nomeie o controller como **Home**. Será criada a classe **HomeController**:



```
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

5. Clique com o botão direito do mouse sobre o action **Index**. Selecione a opção **Add View...**:



6. Será criada a página **Index.cshtml** na pasta **Views/Home**. Na pasta **Views** são incluídas novas pastas com os nomes dos controllers e, em cada pasta, as views correspondentes. A criação destas pastas é realizada pelo próprio Visual Studio. Escreva o seguinte conteúdo nesta página:

```
@{  
    ViewBag.Title = "Home";  
}  
  
<h2>Seja bem vindo ao sistema de Gestão de Vendas</h2>  
<h3>Escolha no menu as opções desejadas</h3>
```

7. Execute a aplicação. Observe que a rota padrão, definida no arquivo **RouteConfig.cs**, foi mantida. A rota padrão considera como controller padrão o **Home** e, como action padrão, o **Index**. Quando a aplicação for executada pela primeira vez, o Visual Studio criará o arquivo **Views/Shared/_Layout.cshtml**, o arquivo **Views/_ViewStart.cshtml** e as pastas contendo os arquivos css do **Bootstrap** e os arquivos JavaScript contendo os scripts do próprio **Bootstrap** e do **JQuery**. O resultado da execução contempla este resultado:



8. Vamos realizar algumas alterações de adequação ao nosso layout. Abra o arquivo **_Layout.cshtml** e faça as alterações sugeridas:

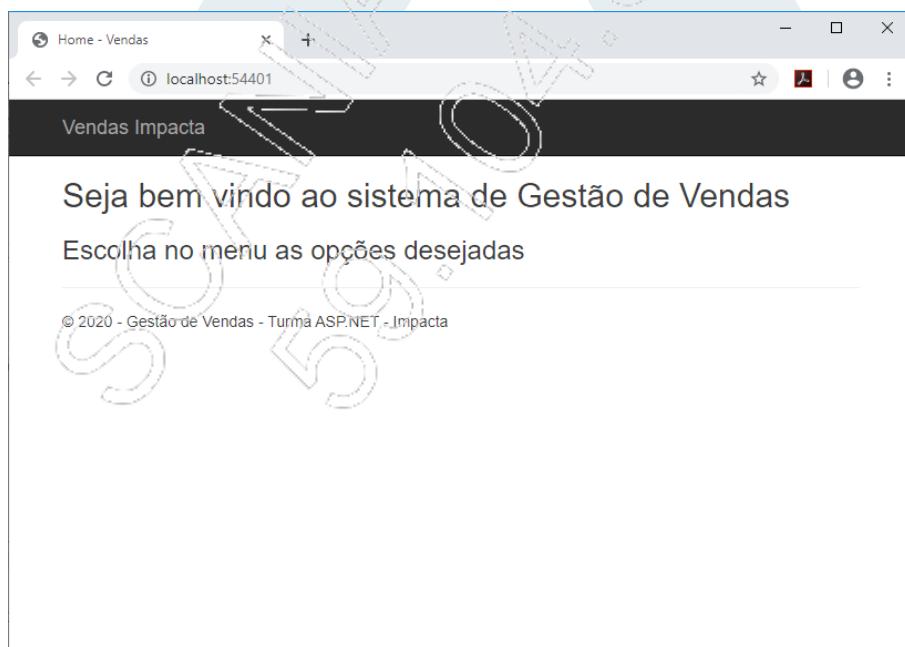
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Vendas</title>
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />
    <script src="~/Scripts/modernizr-2.8.3.js"></script>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle"
                    data-toggle="collapse" data-target=".navbar-collapse">
```

```
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
    </button>
    @Html.ActionLink("Vendas Impacta", "Index", "Home",
    new { area = "" }, new { @class = "navbar-brand" })
</div>
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
    </ul>
</div>
</div>
</div>

<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Gestão de Vendas - Turma ASP.
    NET - Impacta</p>
    </footer>
</div>

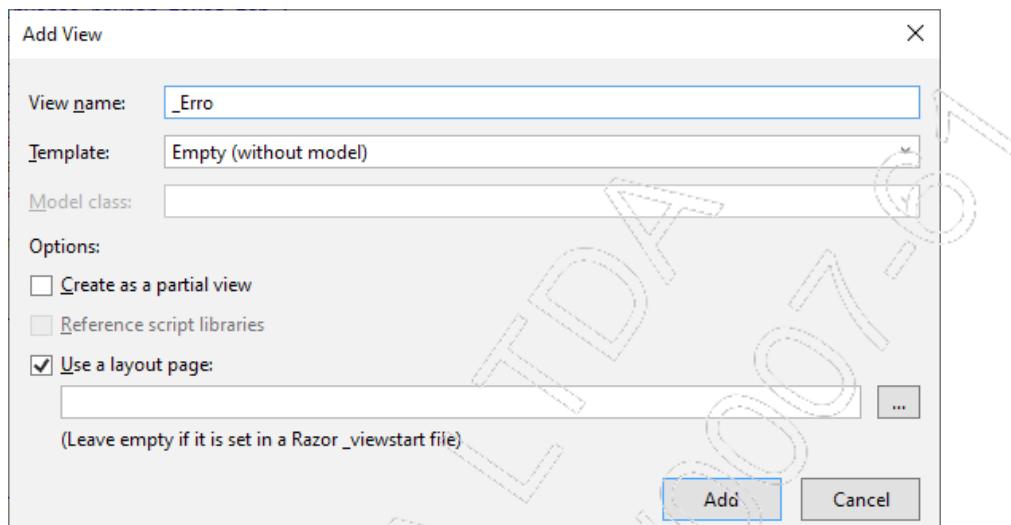
<script src="~/Scripts/jquery-3.3.1.min.js"></script>
<script src="~/Scripts/bootstrap.min.js"></script>
</body>
</html>
```

9. Execute novamente para contemplar as diferenças:



Sinta-se à vontade para incluir imagens, ou outros elementos para tornar a página inicial mais atraente.

10. Em diversas partes da aplicação, estaremos sujeitos a erros de requisição ou de processamento. Vamos criar uma view que será usada por toda a aplicação quando um erro ocorrer. Clique com o botão direito do mouse na pasta **Views/Shared**. Adicione uma view chamada **_Erro** (arquivo **_Erro.cshtml**):



11. Inclua o seguinte conteúdo a esta view:

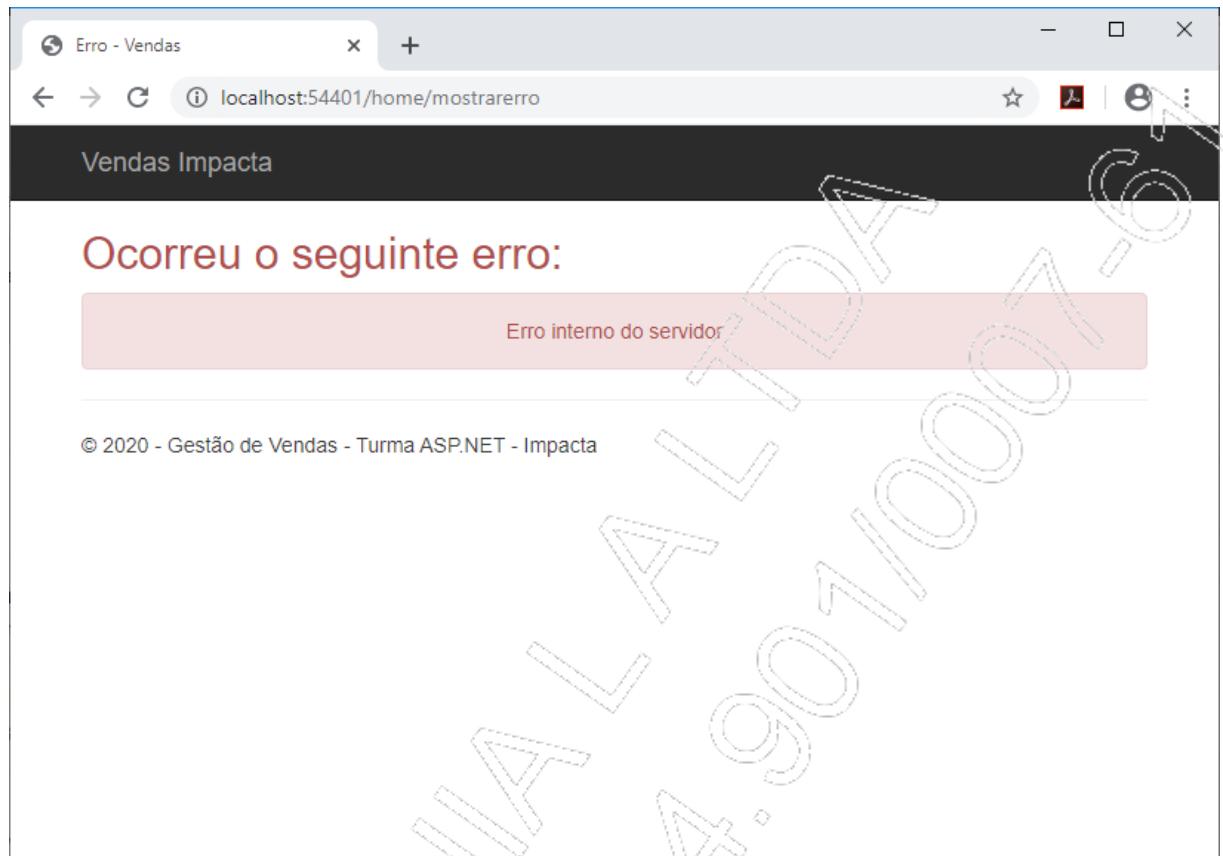
```
@{  
    ViewBag.Title = "Erro";  
}  
  
<h2 class="text-danger">Ocorreu o seguinte erro:</h2>  
  
<div class="alert alert-danger text-center">  
    @ViewBag.MensagemErro  
</div>
```

12. Para testar esta nova view, acrescente o seguinte action no controller **Home**:

```
using System.Web.Mvc;  
  
namespace Lab.MVC.Controllers  
{  
    public class HomeController : Controller  
    {  
        // GET: Home  
        public ActionResult Index()  
        {  
            return View();  
        }  
  
        public ActionResult MostrarErro()  
        {  
            ViewBag.MensagemErro = "Erro interno do servidor";  
            return View("_Erro");  
        }  
    }  
}
```

13. Execute a aplicação, e chame na URL a seguinte rota (não esqueça de copiar a sua URL, pois o numero da porta será diferente!):

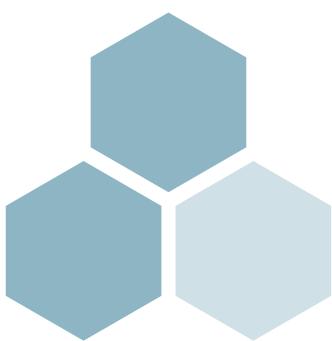
http://localhost:54401/home/mostrarerro



4

Entity Framework

- ◆ Preparando o ambiente;
- ◆ Database First;
- ◆ Model First;
- ◆ Code First.



4.1. Introdução

O **Entity Framework** é um componente que facilita o processo de obter informações de um ou mais bancos de dados e de transferir essas informações para um conjunto de classes que serão manipuladas por uma aplicação.

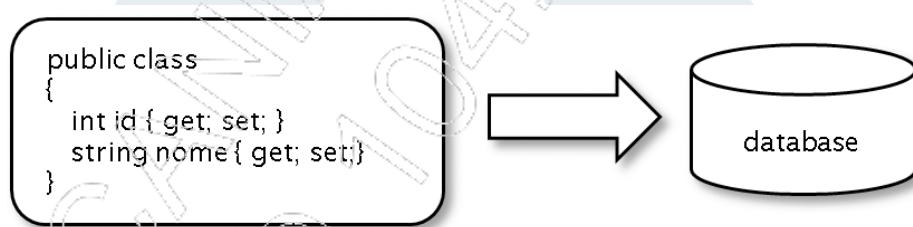
Esse tipo de programa é conhecido como **ORM (Object-Relational Mapping)**. Grande parte do trabalho de escrever expressões SQL, manipular os objetos ADO.NET e criar código para transferir esses dados é feita pelo programa, deixando o programador concentrado no aplicativo em si, nas regras de negócio e nas funcionalidades do sistema.

É importante considerar que nem sempre os objetos da aplicação e o banco de dados obedecem a uma relação de um-para-um (uma tabela para cada classe e vice-versa). Existem relacionamentos, agrupamentos de informações, validações, regras de negócio etc. que podem exigir configurações diferentes dos objetos usados no aplicativo e das tabelas do banco. Por isso, é importante entender como o Entity Framework funciona e como são mapeados os dados do aplicativo e as tabelas do banco de dados.

O Entity Framework nos permite trabalhar de três maneiras:

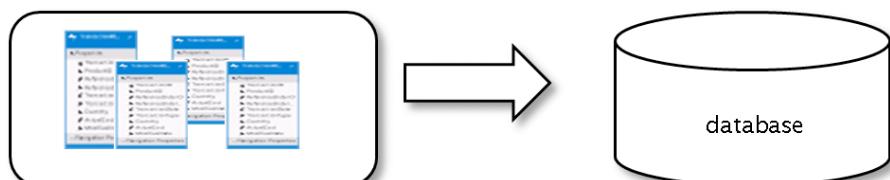
- **Code First**

O programador cria as classes de dados manualmente, e o Entity Framework cria as tabelas no banco de dados ou usa as tabelas de um banco existente. Este modo foi incluído definitivamente na versão 4.0.



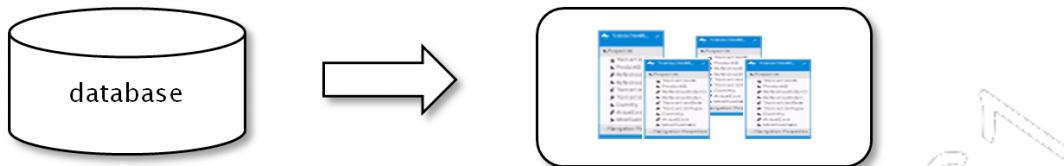
- **Model First**

O programador define as classes de dados usando um ambiente Visual (designer), e o Entity Framework cria as tabelas necessárias no banco de dados ou usa tabelas de um banco já existente.



- **Database First**

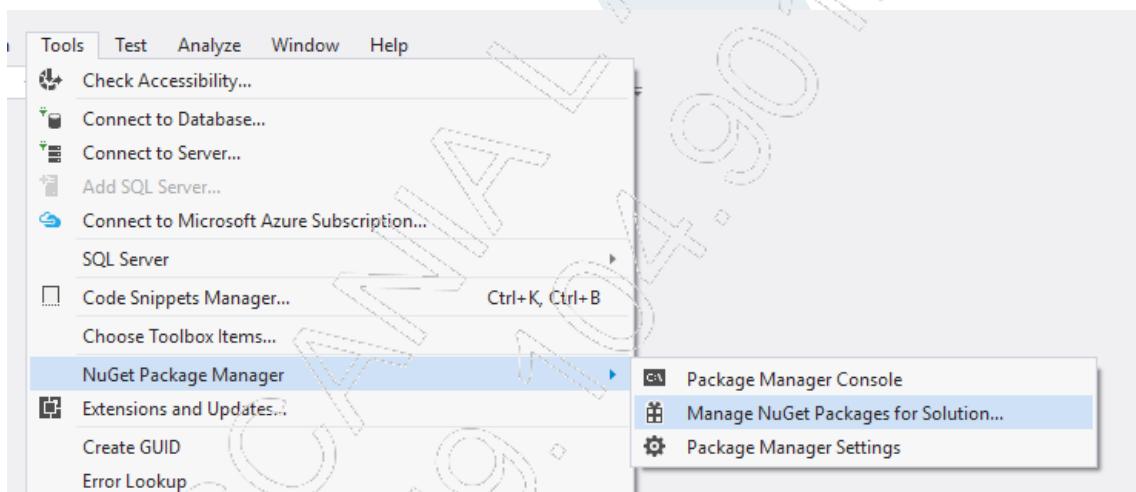
O programador usa um banco de dados existente, e o Entity Framework cria o modelo de dados baseado em suas tabelas e relacionamentos.



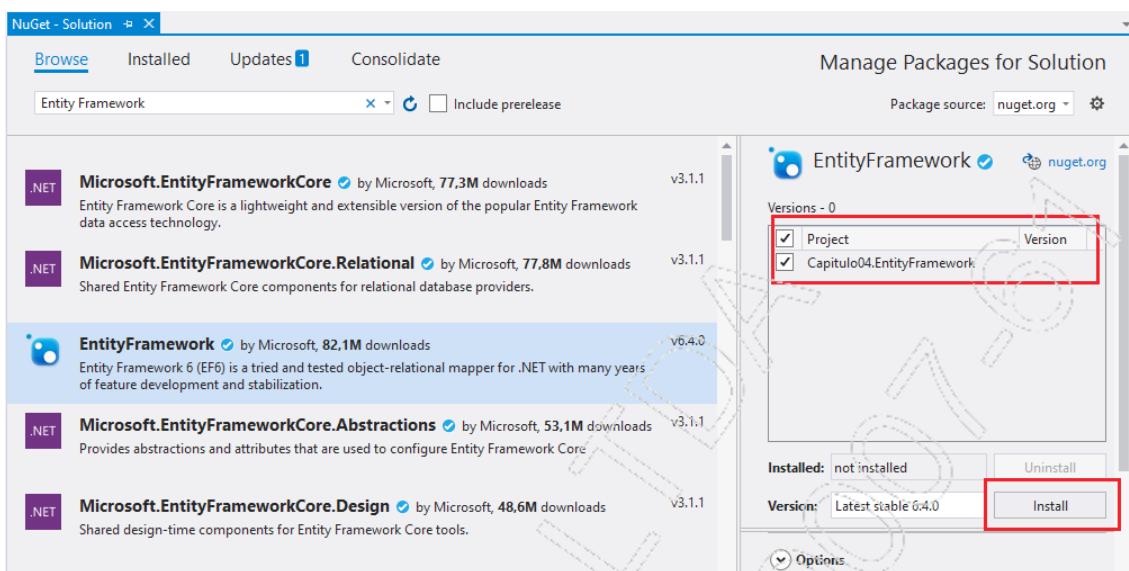
Neste Capítulo, apresentaremos estas três maneiras de trabalho com o Entity Framework.

4.2. Preparando o ambiente

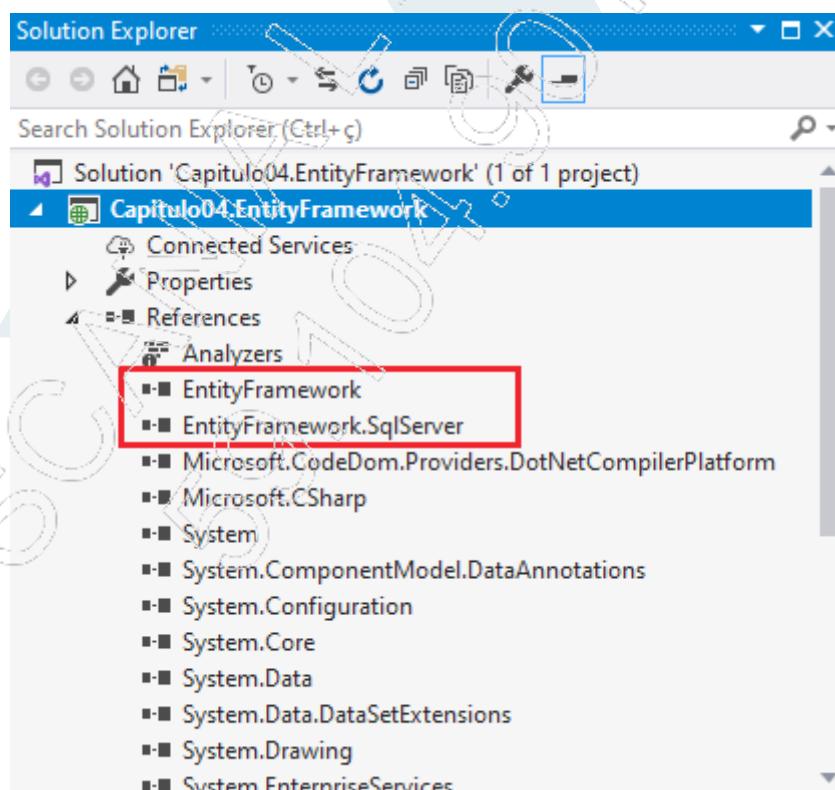
A melhor maneira de preparar um projeto para usar o Entity Framework é usar o gerenciador de pacotes **NuGet**. Em um novo projeto chamado **Capitulo04. EntityFramework** (Asp.Net MVC), selecione o menu **Tools**, escolha a opção **NuGet Package Manager** e, depois, **Manage NuGet Packages for Solution**.



Na janela do **NuGet**, na caixa de pesquisa, digite **Entity Framework** e, depois de encontrada a opção, marque os projetos em que deseja instalá-lo e escolha **Install**.



Algumas referências serão acrescentadas ao seu projeto. Verifique na janela **Solution Explorer**:



Ainda no Solution Explorer, o item **packages.config** contém a lista de componentes instalados pelo NuGet.

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
    <package id="EntityFramework" version="6.4.0" targetFramework="net472"
    />
    <package id="Microsoft.CodeDom.Providers.DotNetCompilerPlatform"
    version="2.0.0" targetFramework="net472" />
</packages>
```

4.3. Database First

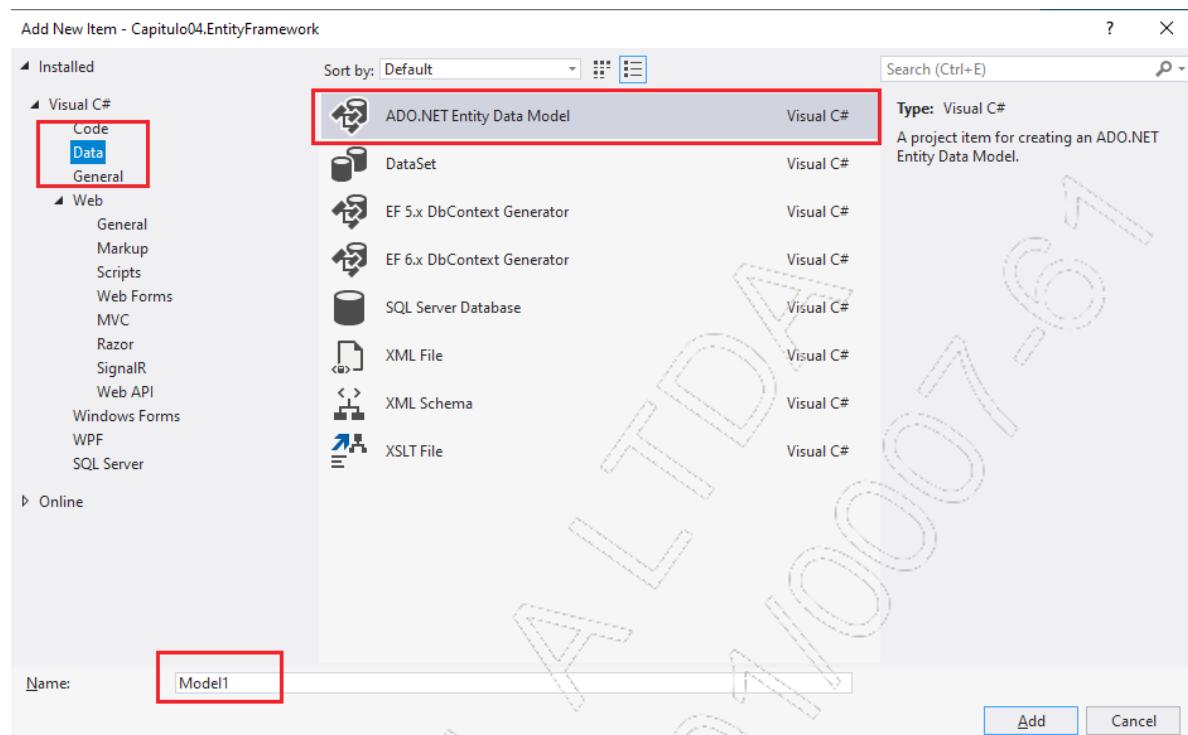
Neste modo, um banco de dados existente é usado, e o Entity Framework gera as classes. Automaticamente, tabelas, campos, relacionamentos e restrições são mapeados e se tornam propriedades das classes geradas.

O exemplo que apresentaremos para ilustrar o mecanismo **Database First** utilizará o banco de dados **Northwind**, disponibilizado pela Microsoft para estudos. O link usado para obter seu script é <https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs>

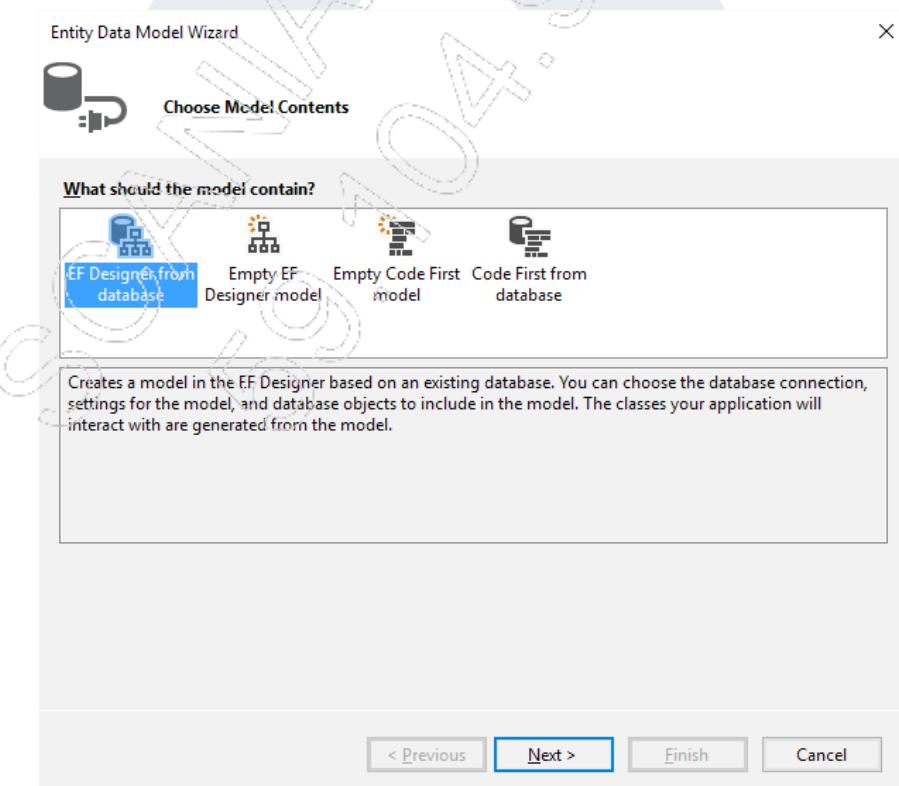
Para aplicar este modo, siga os passos adiante:

1. No projeto, com o botão direito do mouse sobre ele, escolha a opção **Project / Add New Item**;

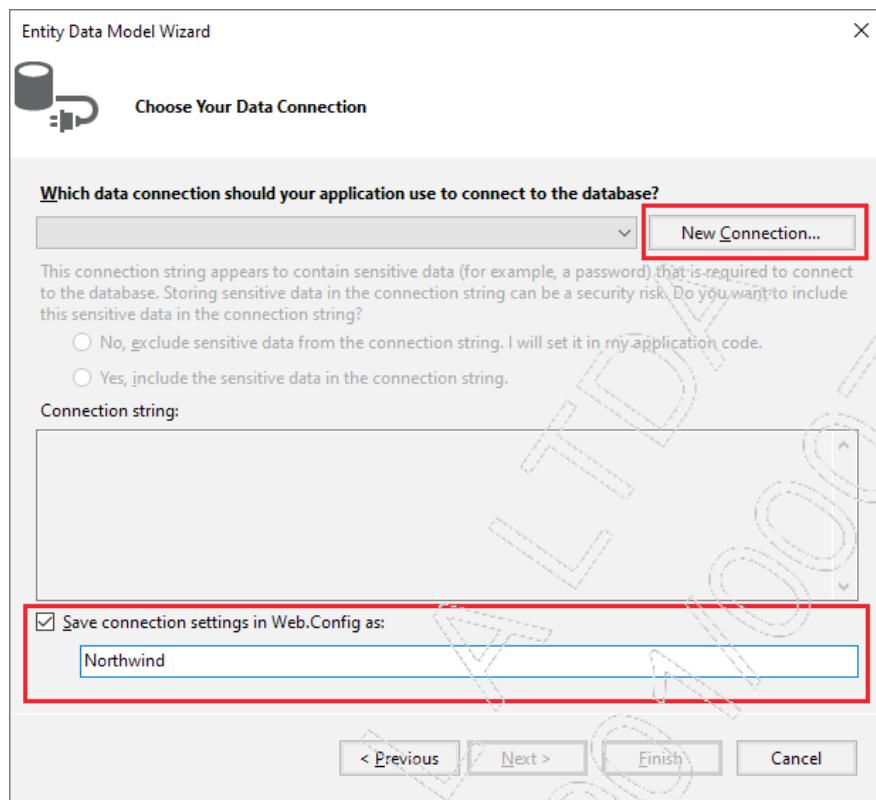
2. Na caixa de diálogo, escolha, na coluna à esquerda, o item **Data**, em seguida o modelo **ADO.NET Entity Data Model** e insira um nome para o modelo:



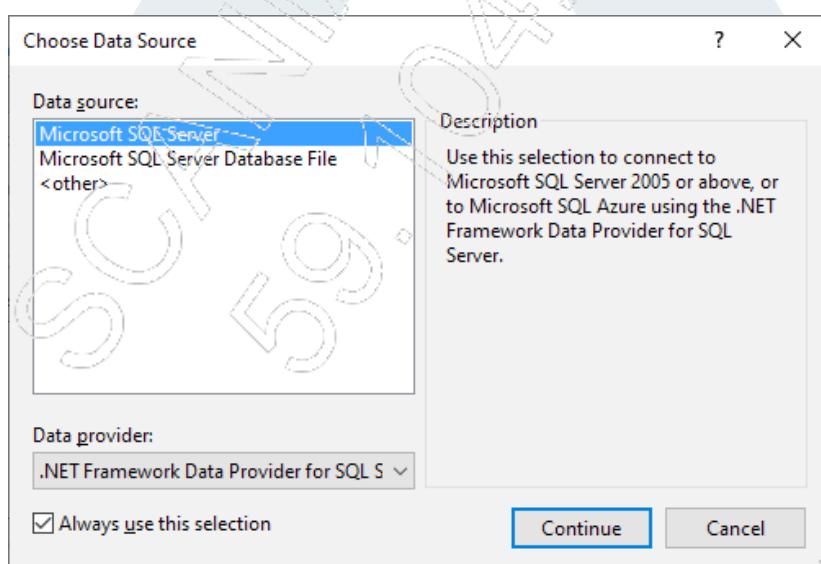
3. Na janela seguinte, escolha **EF Designer from database**:



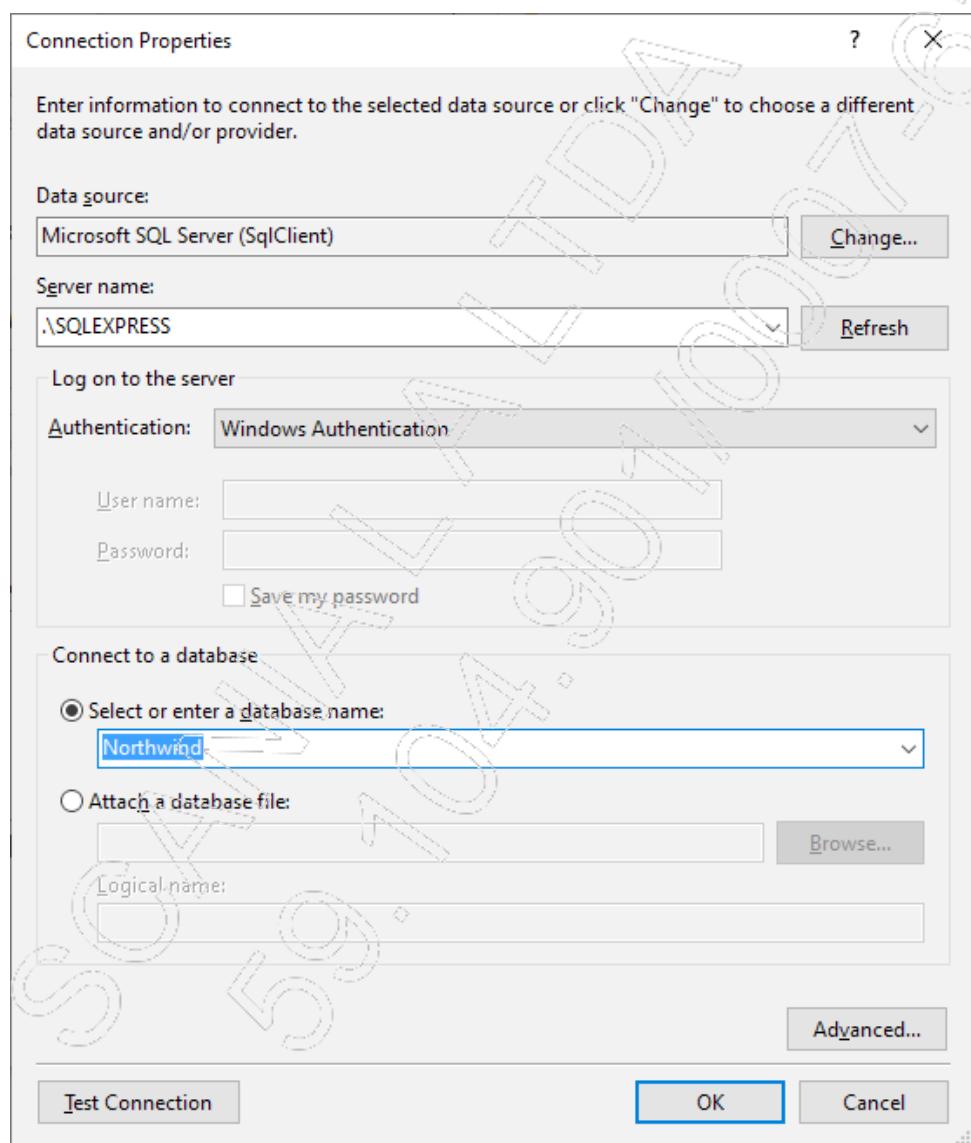
4. Na janela de conexão, crie uma nova conexão, mantendo a opção para salvar as configurações no **Web.Config** marcada:



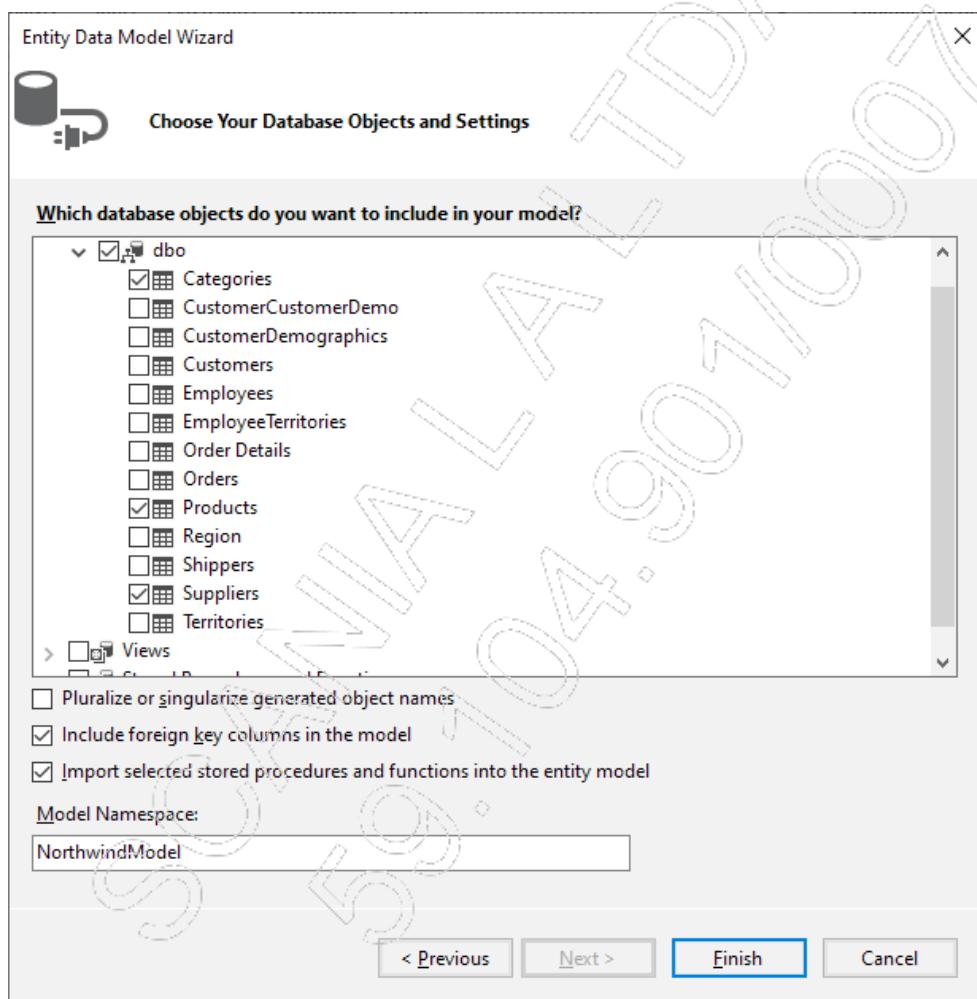
5. Escolha o banco de dados:



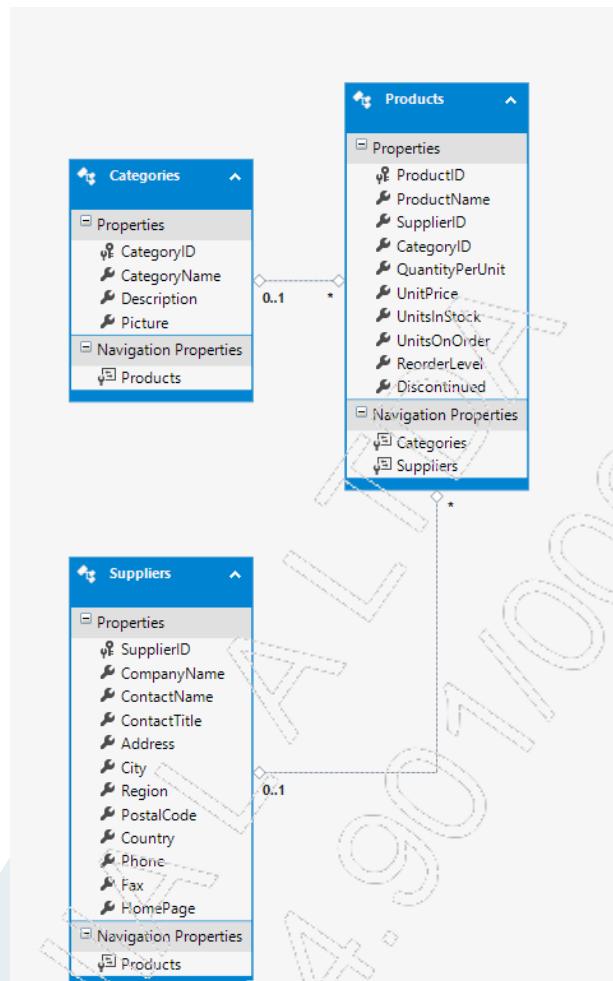
6. É possível selecionar um banco de dados no SQL Server, ou um arquivo local (este será adicionado na pasta **App_Data**). Selecione o banco de dados **Northwind**:



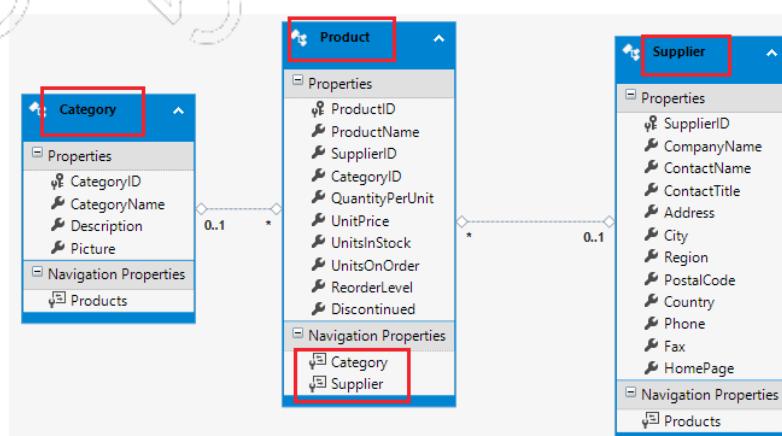
7. A próxima janela permite selecionar quais objetos serão inseridos no modelo. Um modelo de domínio não precisa ter todo o banco de dados, mas apenas as tabelas que fazem sentido juntas. É possível criar diversos modelos para um banco de dados. No banco de dados **Northwind**, vamos selecionar as tabelas **Products**, **Suppliers** e **Categories**:



8. Após a confirmação, todo o modelo é criado automaticamente:



9. É possível alterar o modo como são criadas as classes que representam as tabelas do banco. É importante que toda alteração seja realizada no próprio modelo, pois não só as classes mas o mapeamento entre estas e o banco de dados também serão afetados. Vamos alterar os nomes das entidades no modelo e as propriedades de navegação (aqueles mapeadas como chave primária / estrangeira no banco de dados):



A seguir, a listagem das classes **Product**, **Category** e **Supplier** que foi gerada pelo EF:

```
namespace Capitulo04.EntityFramework
{
    using System;
    using System.Collections.Generic;

    public partial class Product
    {
        public int ProductID { get; set; }
        public string ProductName { get; set; }
        public Nullable<int> SupplierID { get; set; }
        public Nullable<int> CategoryID { get; set; }
        public string QuantityPerUnit { get; set; }
        public Nullable<decimal> UnitPrice { get; set; }
        public Nullable<short> UnitsInStock { get; set; }
        public Nullable<short> UnitsOnOrder { get; set; }
        public Nullable<short> ReorderLevel { get; set; }
        public bool Discontinued { get; set; }

        public virtual Category Category { get; set; }
        public virtual Supplier Supplier { get; set; }
    }
}
```

Na classe **Product**, todo campo que não é obrigatório está marcado como **Nullable**. Isso é necessário para não ter um valor imposto como padrão. O banco de dados, nesse caso, retorna **DBNull.Value** para um registro não preenchido e o EF converte para **Null**. No campo **ProductName** não há essa abordagem porque o tipo **string** aceita **Null** como um valor válido. **String** é inherentemente **Nullable** por ser uma classe.

```
namespace Capitulo04.EntityFramework
{
    using System;
    using System.Collections.Generic;

    public partial class Category
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Category()
        {
            this.Products = new HashSet<Product>();
        }

        public int CategoryID { get; set; }
        public string CategoryName { get; set; }
        public string Description { get; set; }
        public byte[] Picture { get; set; }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Product> Products { get; set; }
    }
}
```

Na tabela **Category**, a lista de produtos (**Products**) é inicializada no construtor, mas só será lida no banco de dados quando for solicitado (**Lazy Load**).

A classe **HashSet** é a maneira mais rápida de armazenar uma coleção, pois se utiliza de um **Hash** (dados criptografados que identificam um usuário). O único problema é que essa classe não expõe a coleção por acesso com um índice ou chave. Não existe acesso individual aos elementos.

Outro detalhe da tabela **Category** é o fato de que a imagem é representada por um array de bytes (propriedade **Picture**) e não por uma propriedade do tipo **System.Drawing.Image**. O mapeamento de imagens para array de bytes evita a criação de objetos **Bitmap** em memória que definitivamente serão usados apenas quando a imagem for exibida. Além disso, tendo um array de bytes, fica a cargo do aplicativo que vai exibir os dados decidir a melhor maneira de renderizar a imagem.

```
namespace Capitulo04.EntityFramework
{
    using System;
    using System.Collections.Generic;

    public partial class Supplier
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Supplier()
        {
            this.Products = new HashSet<Product>();
        }

        public int SupplierID { get; set; }
        public string CompanyName { get; set; }
        public string ContactName { get; set; }
        public string ContactTitle { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Region { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
        public string Phone { get; set; }
        public string Fax { get; set; }
        public string HomePage { get; set; }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Product> Products { get; set; }
    }
}
```

A classe fornecedores (**Supplier**) é uma classe simples com uma única associação (neste diagrama) à outra classe (**Products**). Um fornecedor pode fornecer diversos produtos, e cada produto pertence a um único fornecedor. É uma associação do tipo **um-para-muitos**.

A seguir, a listagem gerada pelo EF para a classe derivada de **DbContext**, que centraliza todas as operações com o banco de dados:

```
namespace Capitulo04.EntityFramework
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class NorthwindEntities : DbContext
    {
        public NorthwindEntities()
            : base("name=NorthwindEntities")
        {
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<Category> Categories { get; set; }
        public virtual DbSet<Product> Products { get; set; }
        public virtual DbSet<Supplier> Suppliers { get; set; }
    }
}
```

Repare que o construtor chama o construtor da classe base passando uma string, que, nesse caso, é uma string de conexão gravada no Web.Config.

Como esse construtor pode ser usado tanto para passar uma string de conexão quanto o nome de uma string de conexão armazenada, a forma de diferenciar uma da outra é passar, quando for o nome de uma conexão, a sintaxe **name=xxxxx**.

- **Na classe DbContext**

```
public NorthwindEntities()
    : base("name=NorthwindEntities") { }
```

- No Web.Config

```
<connectionStrings>
    <add name="NorthwindEntities" connectionString="metadata=res://*/Model1.csdl|res://*/Model1.ssdl|res://*/Model1.msl;provider=System.Data.SqlClient;provider connection string="data source=.\SQLEXPRESS;initial catalog=Northwind;integrated security=True;Multiple ActiveResultSets=True;App=EntityFramework"" providerName="System.Data.EntityClient" />
</connectionStrings>
```

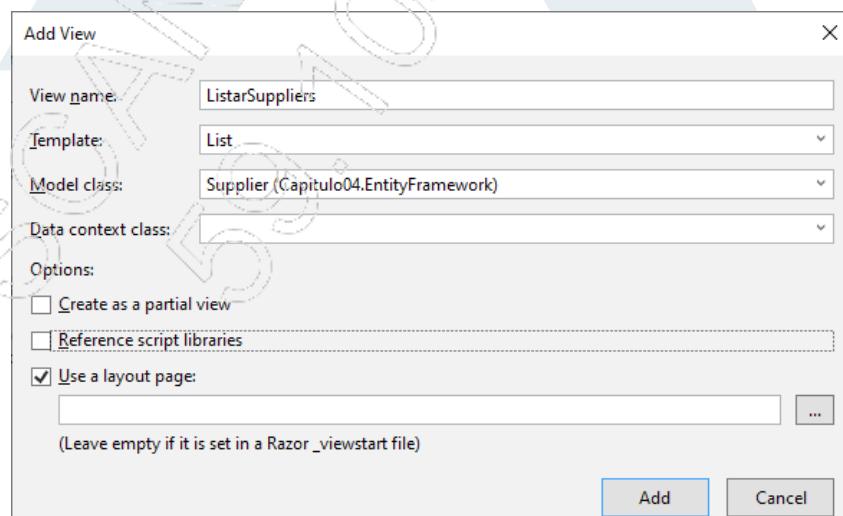
4.3.1. Usando o modelo criado

Uma vez que o modelo esteja criado, podemos usá-lo na aplicação em uma view. Vamos seguir os passos para visualizarmos o resultado da coleção **Suppliers**:

1. No projeto, inclua o controller **Home** (classe **HomeController**);
2. Adicione o action **ListarSuppliers()**;
3. Nesse action, escreva o código:

```
public ActionResult ListarSuppliers()
{
    var db = new NorthwindEntities();
    return View(db.Suppliers.ToList());
}
```

4. Inclua a view **ListarSuppliers**, com o template **List**. Selecione o modelo **Supplier**:

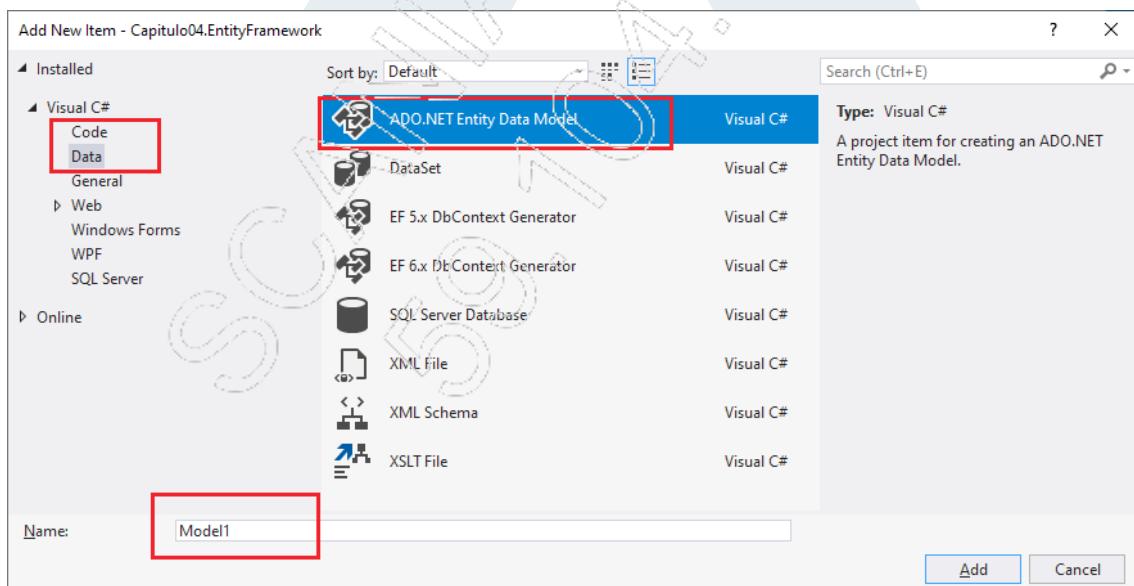


Vejamos o resultado da execução:

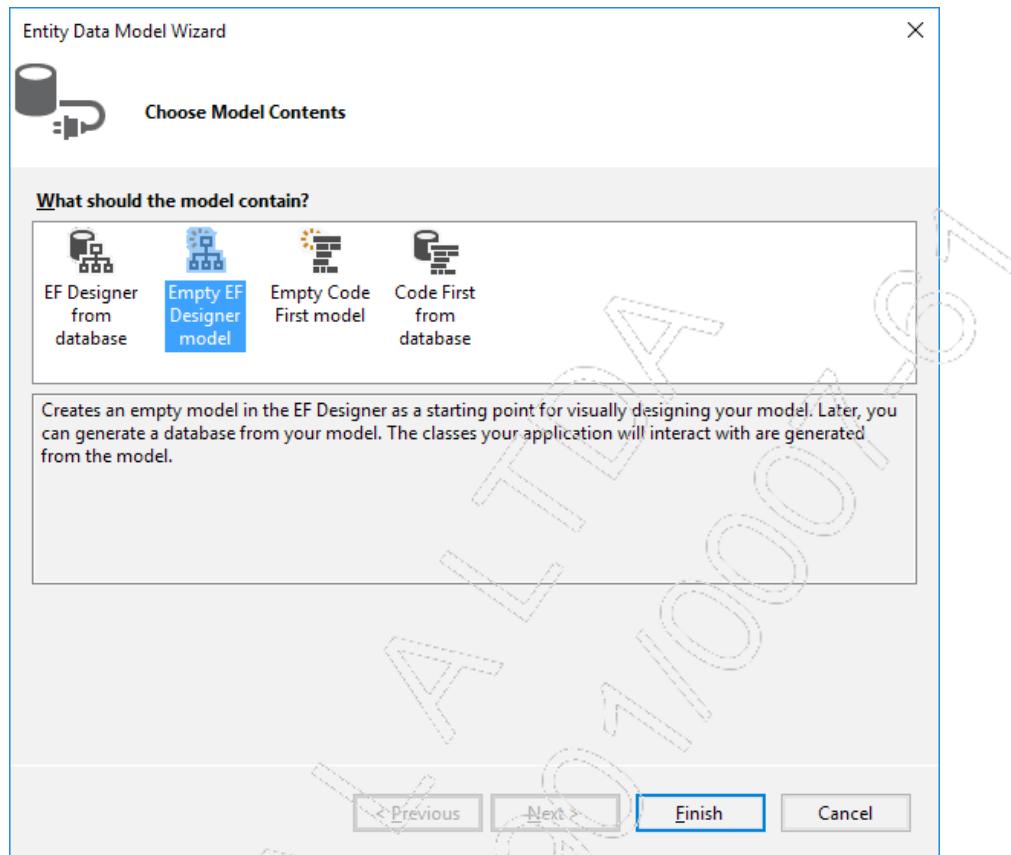
CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
Exotic Liquids	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London		EC1 4SD	UK	(171) 555-2222	
New Orleans Cajun Delights	Shelley Burke	Order Administrator	P.O. Box 78934	New Orleans	LA	70117	USA	(100) 555-4822	
Grandma Kelly's Homestead	Regina Murphy	Sales Representative	707 Oxford Rd.	Ann Arbor	MI	48104	USA	(313) 555-5735	(313) 555-3349
Tokyo Traders	Yoshi Nagase	Marketing Manager	9-8 Sekimai Musashino-shi	Tokyo		100	Japan	(03) 3555-5011	

4.4. Model First

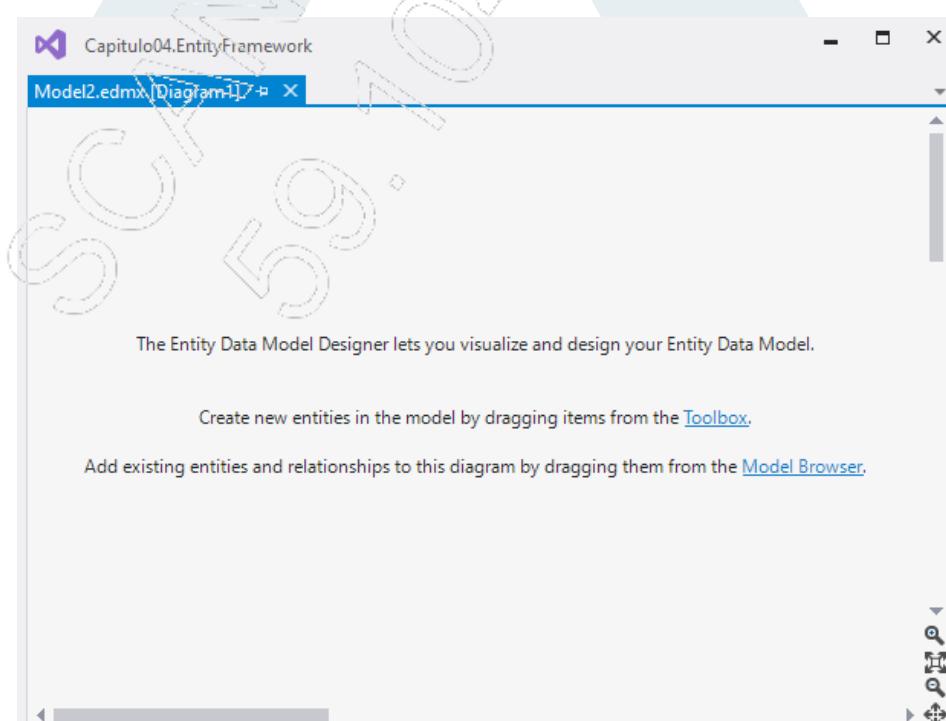
Para criar um modelo de domínio em um projeto do Visual Studio, escolha, no menu, a opção **Project / Add New Item**. Na caixa de diálogo, escolha, na coluna à esquerda, o item **Data**, em seguida o modelo **ADO.NET Entity Data Model** e insira um nome para o modelo:



Escolha a opção **Empty EF Designer model** mostrada a seguir:

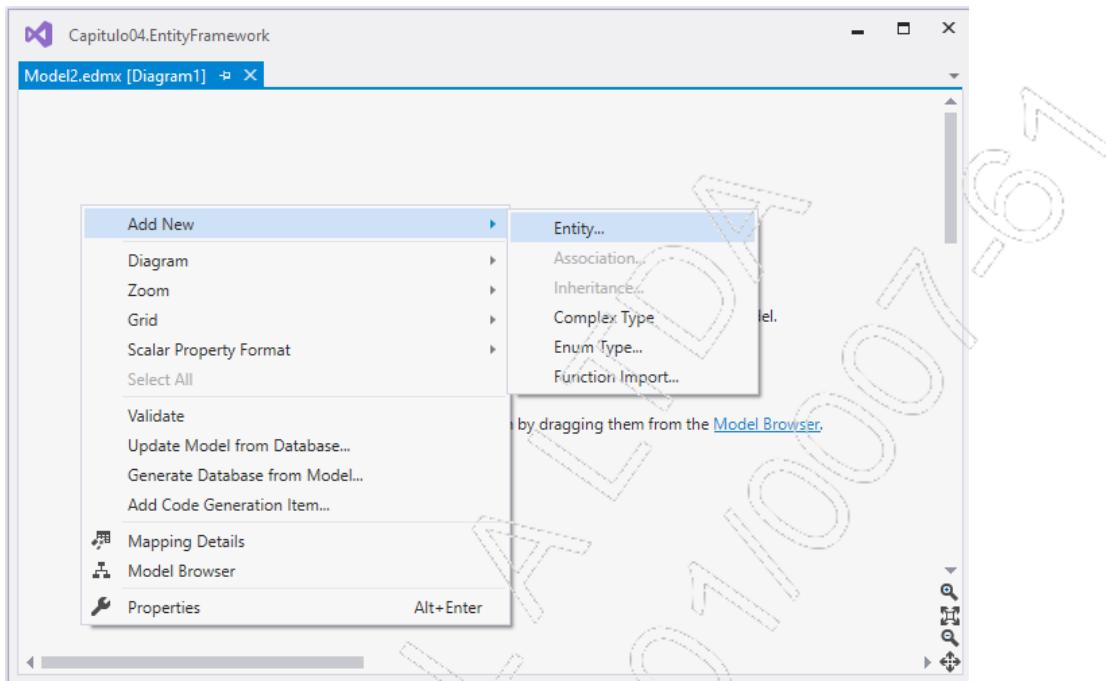


Nesse modo, é possível criar entidades e relacionamentos para depois criar ou utilizar um banco de dados.

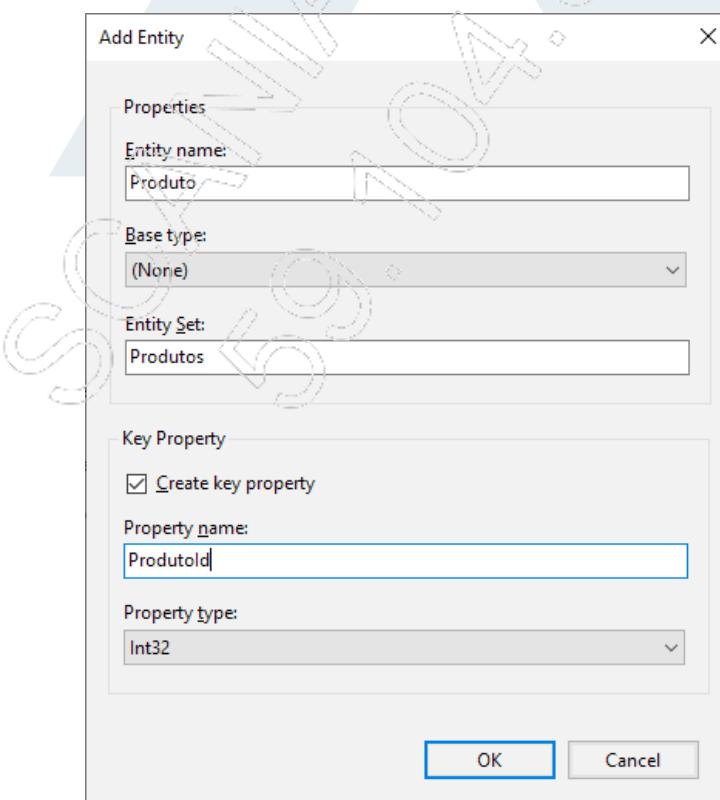


4.4.1. Adicionando uma Entity (entidade)

A principal estrutura de dados é a Entity (entidade). Usando o menu de contexto do EF Designer, escolha Add New / Entity:



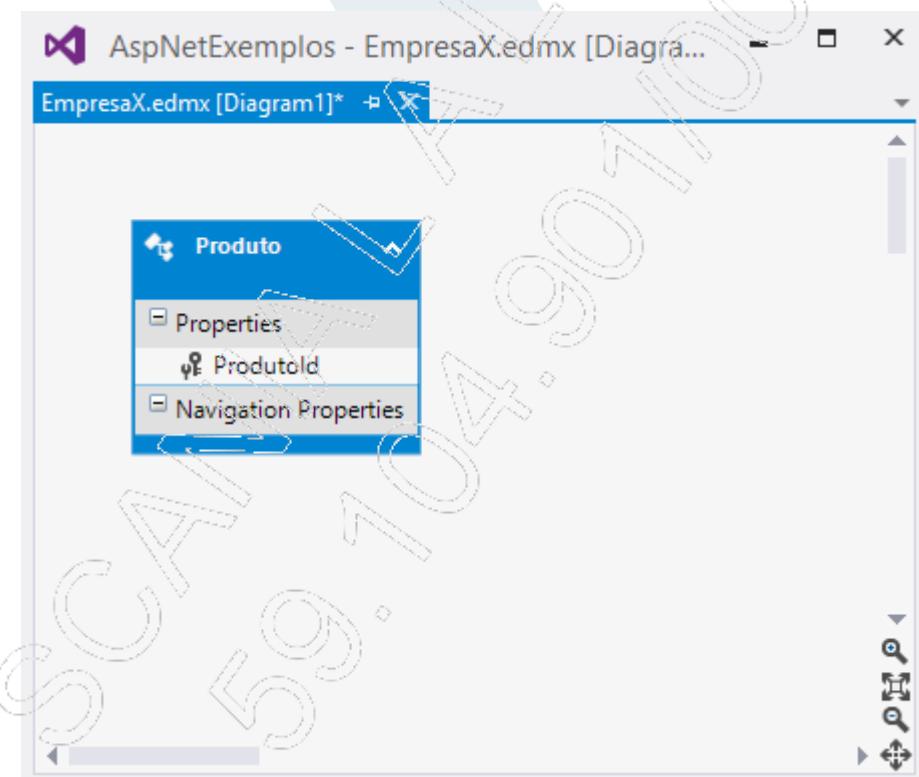
A caixa de diálogo usada para definir uma Entity é exibida:



Os seguintes campos podem ser preenchidos:

- **Entity name:** O nome da entidade, por exemplo: **cliente, produto, cidade, funcionario, notafiscal, boleto, veiculo, aluguel** etc.;
- **Base type:** A classe base para esta classe;
- **Entity Set:** O nome do conjunto de elementos para essa entidade. Por padrão, é o nome do objeto no plural: **Produtos, Clientes, Fornecedores**;
- **Create key property:** Marque esta opção para criar a chave primária;
- **Property name:** O nome da chave primária;
- **Property type:** O tipo da chave primária.

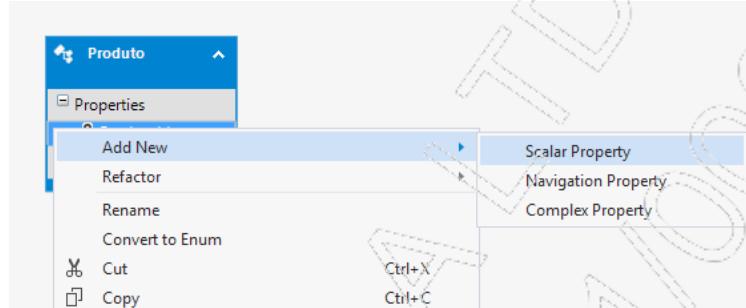
Ao término do preenchimento, o modelo aparecerá no EF Designer:



4.4.2. Adicionando propriedades

Clicando com o botão direito sobre o diagrama da classe **Produto** e, em seguida, em **Add New**, existem três tipos de elementos que podem ser adicionados no modelo:

- **Scalar Property**: Propriedades com tipos primitivos como **string**, **int32**, **DateTime**, **decimal** ou **double**;
- **Navigation Property**: Propriedades que se relacionam a outra Entity;
- **Complex Property**: Propriedades compostas, que são o agrupamento de dois ou mais campos simples.

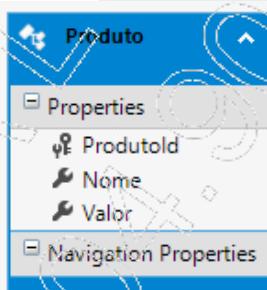


Apontando para uma propriedade e abrindo o menu de contexto, existe a opção **Properties**, em que é possível definir detalhes sobre esse elemento. Os itens que podem ser definidos são os seguintes:

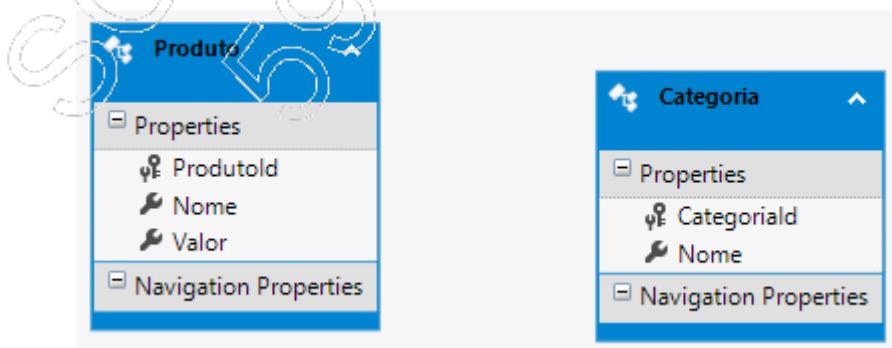
- **Concurrency Mode**: Os valores podem ser **Fixed** ou **None**. Se definido para **Fixed**, o valor original do campo é enviado para a cláusula **Where** quando um registro é atualizado. Se o valor foi alterado enquanto o registro estava sendo editado, a atualização falhará, porque não será encontrado o registro. Esse modo é chamado de **concorrência otimista**. Se o valor estiver definido para **None**, o valor original não será enviado para a cláusula **Where**, o que resultará na falta da verificação de alteração do registro. A atualização, nesse caso, irá sobrepor o conteúdo alterado por outro processo. Esse modo é chamado de **concorrência pessimista**;
- **Default Value**: O valor padrão do campo. Este valor será inserido automaticamente nos novos registros, se nenhum valor for fornecido para este campo;
- **Documentation**: Estrutura com os campos **Long Description** para armazenar uma descrição detalhada do campo e **Summary** para uma descrição simplificada;
- **Entity Key**: Determina se é a chave primária da Entity;
- **Fixed Length**: Determina se a propriedade tem um tamanho fixo em bytes;
- **Getter**: Modificador de acesso para leitura. Pode ser **Private**, **Public**, **Internal** e **Protected**;

- **Max Length:** O número máximo de caracteres;
- **Name:** O nome do campo;
- **Nullable:** Se o campo é do tipo `Nullable<T>`, ou seja, se pode receber valores nulos;
- **Setter:** Modificador de acesso para gravação. Pode ser **Private**, **Public**, **Internal** e **Protected**;
- **StoreGeneratedPattern:** Define se uma coluna é calculada automaticamente. O tipo de geração automática mais comum é o campo do tipo **Identity**, no qual um valor numérico é incrementado automaticamente quando um registro novo é inserido. Existe o tipo **Computed**, que define um campo como calculado, e o tipo **None**, indicando que o campo não é calculado no servidor;
- **Type:** O tipo da propriedade. Pode ser um tipo primitivo, como **Binary**, **Byte**, **Boolean**, **DateTime**, **Double**, **Decimal**, **Int32**, **Int64**, **String**, tipos que representam coordenadas, como **Geography**, **Geometry**, e tipos especiais, como **Guid**.

Vamos adicionar as propriedades **Nome (string)** e **Valor (decimal)**:



Vejamos, a seguir, um exemplo simples com duas entidades (**Produto** e **Categoria** de produto):

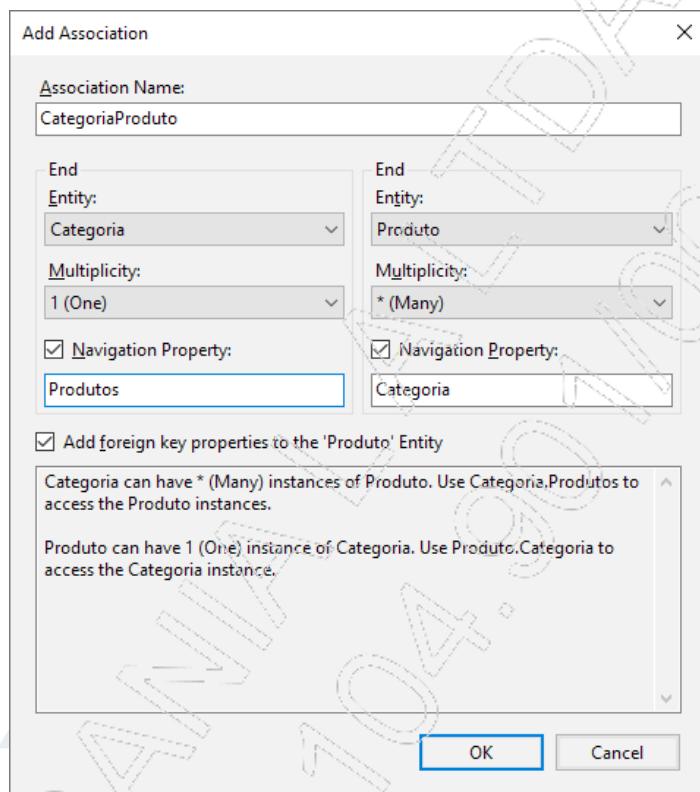


4.4.3. Adicionando associações

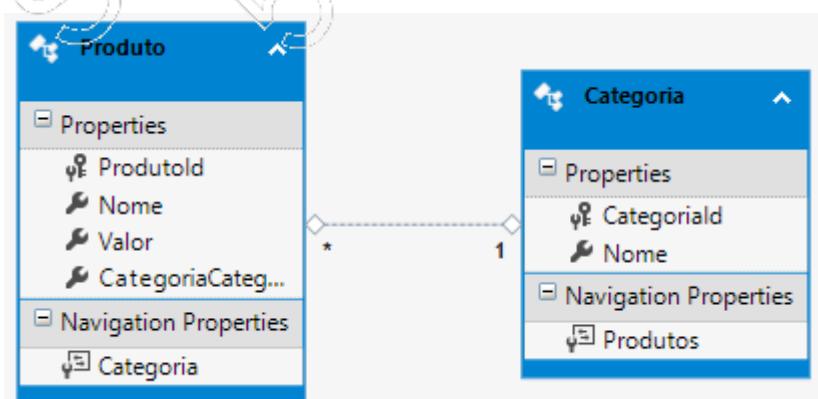
Uma associação define o relacionamento entre duas entidades. No exemplo da estrutura anterior, cada **Produto** pertence a uma **Categoria**, e cada **Categoria** pode estar associada a infinitos **Produtos**. Esse tipo de relação é chamado **um-para-muitos**.

- **Associação um-para-muitos**

Clicando com o botão direito do mouse sobre o design, escolha **Add Association** para ter acesso à janela de criação de associação:



A propriedade **Categoria.Produtos** retorna a lista de produtos de uma categoria, e a propriedade **Produto.Categoria** retorna a categoria de um produto.



- **Associação um-para-um**

O tipo de associação na qual uma instância de uma classe se associa a outra instância de outra classe é chamado **um-para-um**.

- **Associação muitos-para-muitos**

Neste tipo de associação, uma instância de uma classe pode se referir a diversas instâncias de outra classe, e essas outras instâncias podem se referir a diversas instâncias da primeira classe.

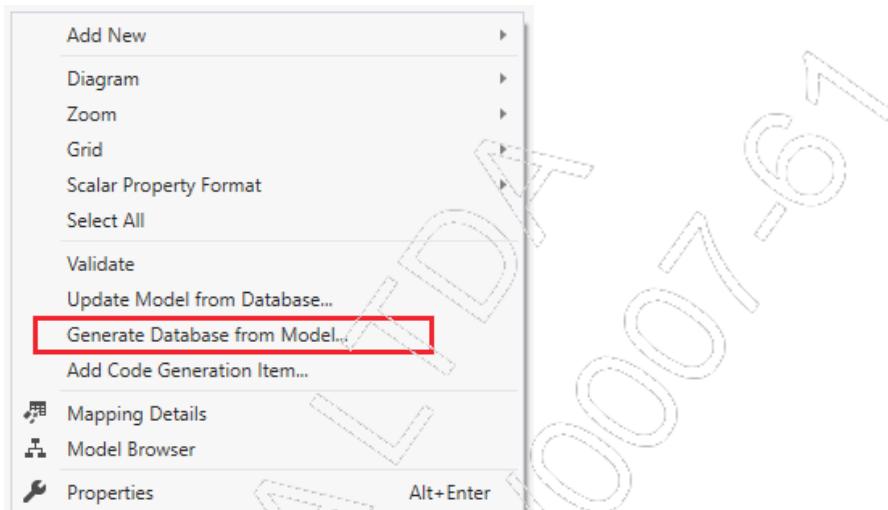
Pense em uma relação na qual um fornecedor pode fornecer diversas categorias de produtos e uma categoria pode ser referenciada por diversos fornecedores. Por exemplo:

- O fornecedor **EmpresaX** fornece produtos das categorias **Computadores** e **Impressoras**;
- O fornecedor **EmpresaY** fornece produtos das categorias **Impressoras** e **Monitores**;
- A categoria **Computadores** é fornecida pelo fornecedor **EmpresaX**;
- A categoria **Impressoras** é fornecida pelos fornecedores **EmpresaX** e **EmpresaY**;
- A categoria **Monitores** é fornecida pelo fornecedor **EmpresaY**.

Para criar uma relação de **muitos-para-muitos** é necessária uma tabela (ou entidade) intermediária (existem outras formas, mas por enquanto o objetivo é ficar o mais próximo possível da estrutura de um banco de dados).

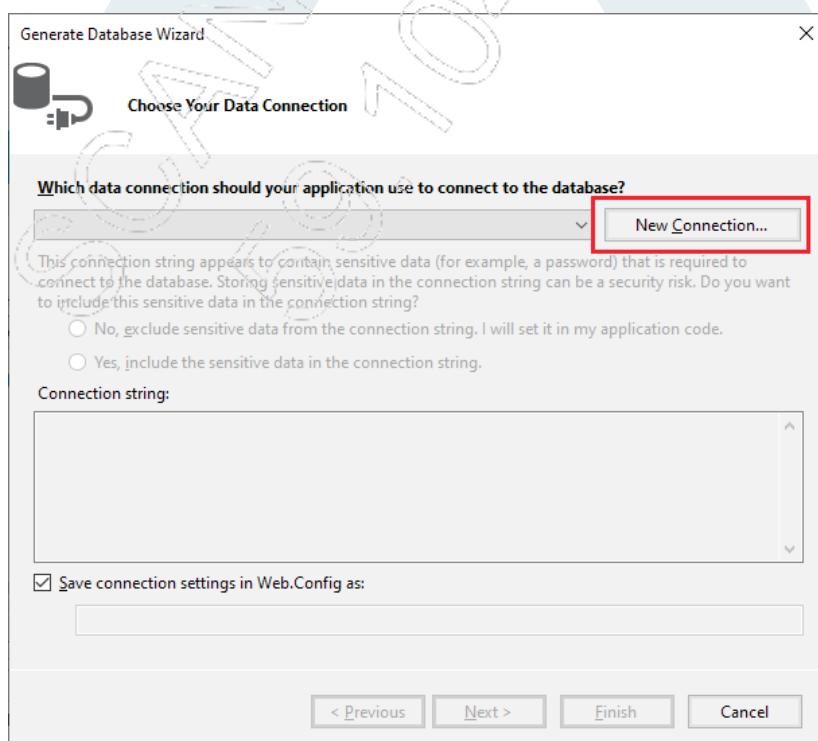
4.4.4. Criando o banco de dados

Uma vez criado o modelo, é hora de associá-lo a um banco de dados. Para criar o banco de dados, é necessário acionar o menu de contexto do EF Designer e escolher **Generate Database from Model**.

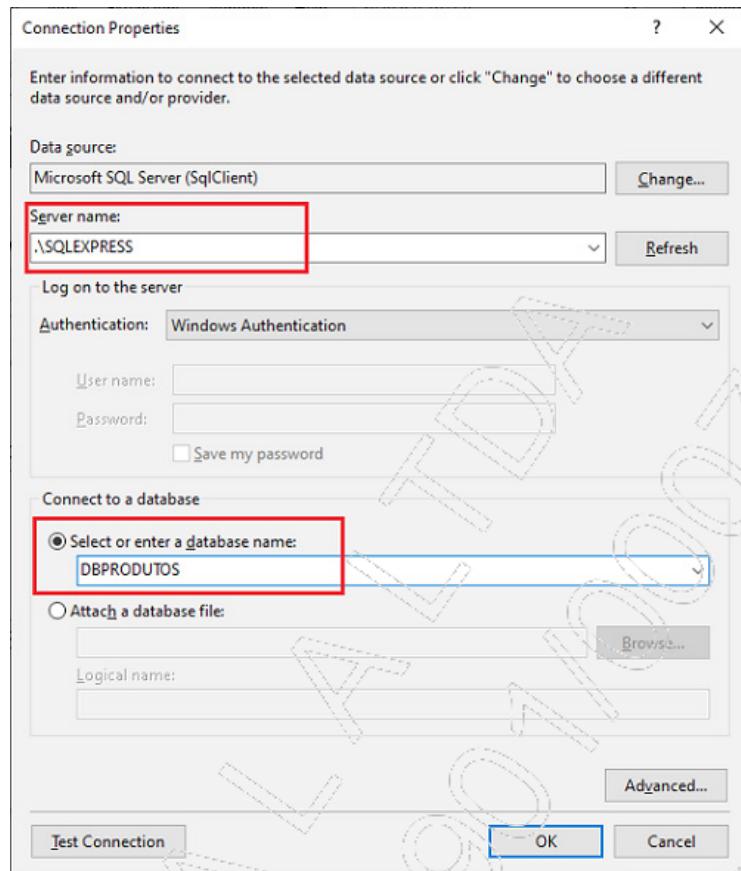


É possível conectar-se a um banco de dados existente ou criar um novo. Para bancos de dados SQL Server, todo o suporte para manipular o banco está pronto. Para outros bancos de dados, como MySQL, é necessário fazer o download do Provider do Entity Framework.

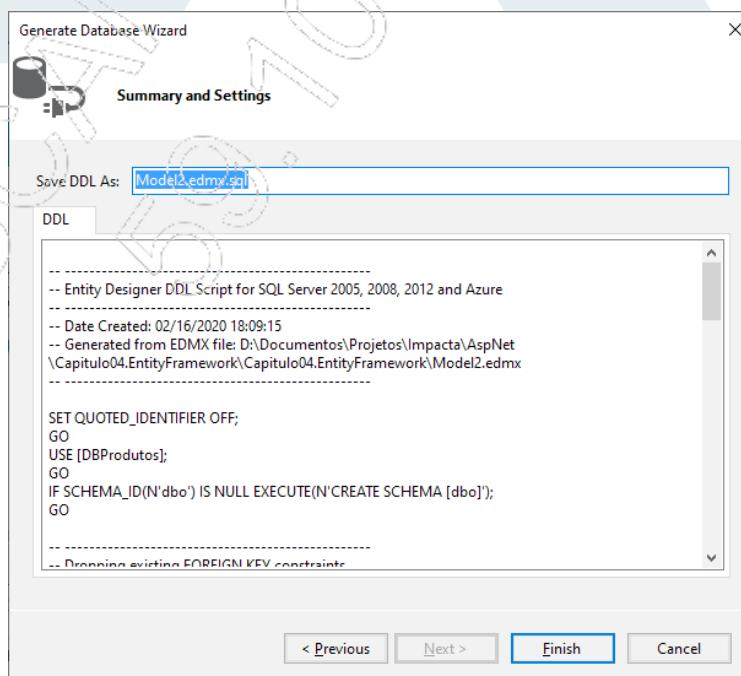
A tela de obter conexão aparecerá. Pode ser um banco existente ou um novo banco. Clicando em **New Connection**, a tela de criação e conexão com o SQL Server aparecerá.



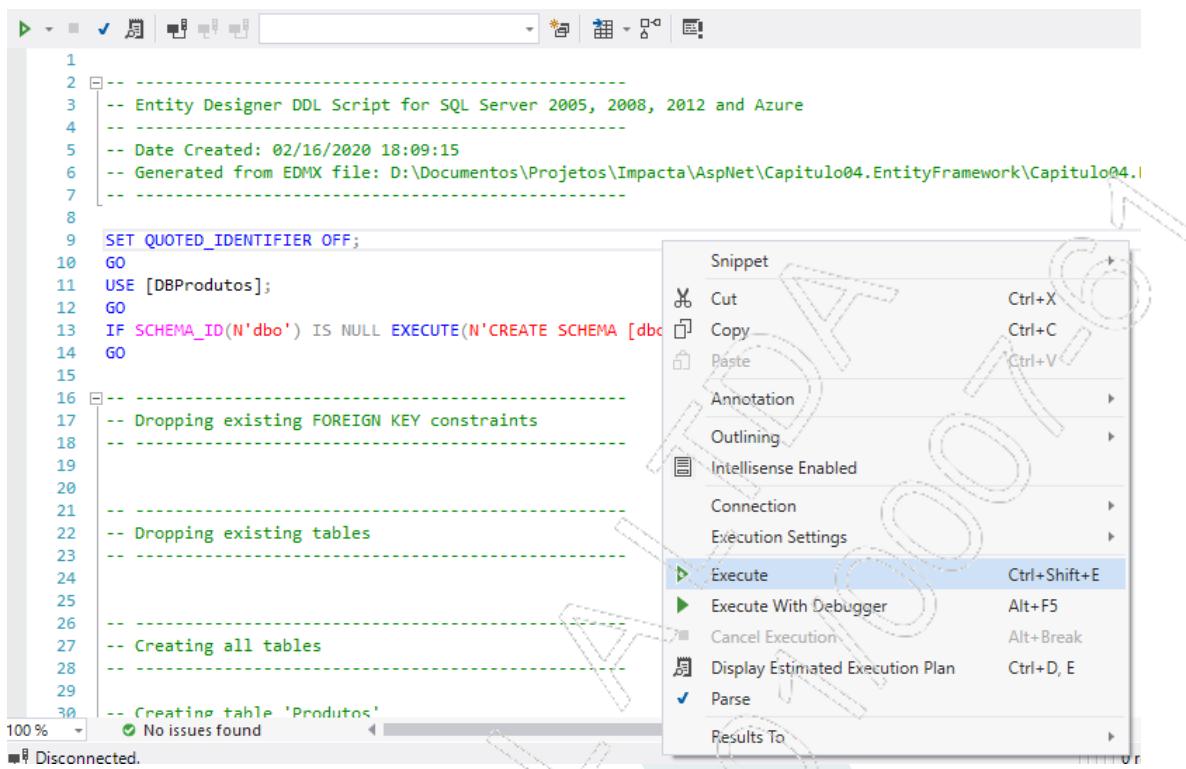
Para criar um novo banco, basta digitar o nome.



O script de criação do banco é gerado. Neste ponto, o script foi gerado em uma nova janela.



Clicando em **Finish**, o script aparece no Visual Studio e pode ser executado, usando o menu de contexto:



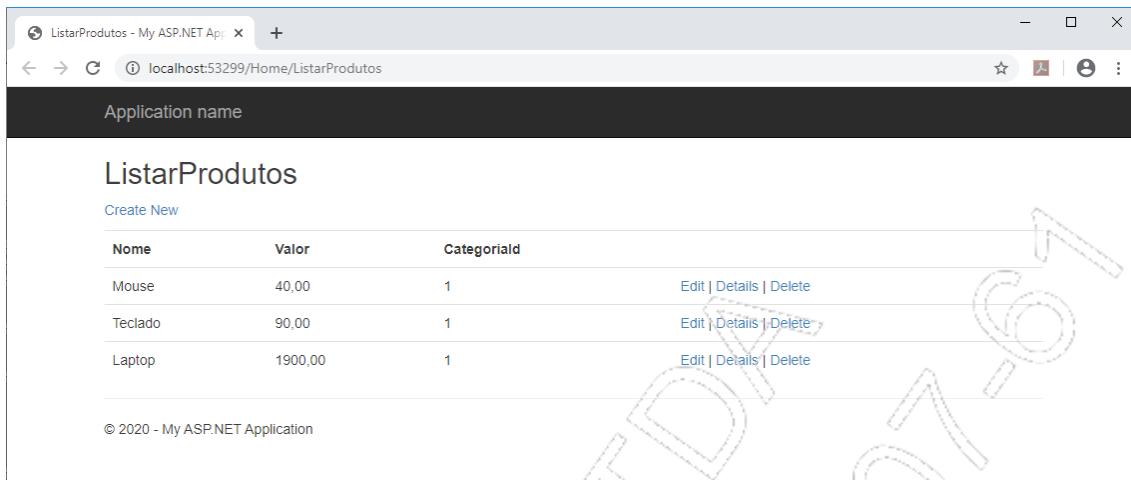
É possível também copiar o conteúdo do script no SQL Server e executar separadamente do Visual Studio.

4.4.5. Usando o modelo criado

Uma vez tendo o modelo criado e os dados mapeados para um banco de dados, já é possível visualizar, incluir, alterar e excluir dados usando as classes geradas pelo Entity Framework. O exemplo a seguir mostra dados de um produto em uma view. Para executá-lo, é necessária uma inclusão prévia de alguns produtos no banco de dados. Vamos adicionar um action no controller Home chamado **ListarProdutos**.

```
public ActionResult ListarProdutos()
{
    var db = new Model2Container();
    return View(db.Produtos.ToList());
}
```

Assim como no exemplo anterior, geramos uma view e a executamos:



4.5. Code First

Em linhas gerais, o Entity Framework, usando o modo **Code First**, trabalha da seguinte maneira:

- **Modelo de domínio:** São criadas classes que representam os dados;
- **Contexto da conexão:** Cria-se uma classe derivada da classe **DbContext**, que é relacionada a uma conexão de um ou mais bancos;
- **Mapeamento:** Dentro da classe derivada de **DbContext**, são criadas coleções do tipo **DbSet<T>**, sendo que **T** é o tipo da classe de modelo de domínio;
- **Operações CRUD (Create, Read, Update, Delete):** Usando expressões **Linq** ou acessando as coleções e objetos individuais, é possível ler, inserir, alterar e excluir informações gravadas nas tabelas do banco de dados mapeado.

Cada etapa segue algumas convenções, mas fornece, também, todos os recursos necessários para personalizar a maneira como o banco é criado ou conectado, o modo como os dados são validados e transferidos para objetos de memória e como estes são usados para atualizar as informações originais ou criar novas informações. A seguir, veremos cada uma dessas etapas.

4.5.1. Modelo de domínio

Um **modelo de domínio** é o detalhamento organizado das informações que fazem parte de um assunto. Esse modelo determina como os dados se relacionam entre si, a terminologia usada e as características inerentes a cada informação.

O ponto central do modelo de domínio é o conceito de **entidade**, que é a representação conceitual de algo existente. Em um modelo de domínio que represente uma loja, as entidades podem ser, entre outras, **Cliente**, **Fornecedor**, **Venda**, **Compra** e **Produto**. Em um modelo que represente uma locadora de carros, as entidades podem ser elementos como **Carro**, **Cliente**, **Vendedor**, **Aluguel** e **Seguro**. Em um modelo de uma clínica médica, as entidades podem ser, entre outras, **Paciente**, **Médico**, **Conta**, **Quarto**, **Diagnóstico**, **Recepcionista**, **Enfermeira**, **Parente** e **Receita**. Em programação orientada a objetos, essas entidades são classes com **Propriedades**, **Métodos** e **Eventos**. O exemplo a seguir define a entidade **Produto**:

```
namespace Capitulo04.EntityFramework.Models
{
    public class Produto
    {
        public int ProdutoId { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
        public int Estoque { get; set; }
    }
}
```

Classes desse tipo, apenas com propriedades, são conhecidas como **POCO (Plain Old CLR Object)**. Esse tipo de classe é muito usada em arquiteturas que utilizam serviços, porque é leve e não depende de outras classes, sendo facilmente serializada, ou seja, transformada em string, XML, stream ou quaisquer dados em série.

Uma classe de modelo de domínio pode incluir funcionalidades:

```
namespace Capitulo04.EntityFramework.Models
{
    public class Produto
    {
        public int ProdutoId { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
        public int Estoque { get; set; }

        public int IncrementarEstoque(int quantidade)
        {
            this.Estoque += quantidade;
            return this.Estoque;
        }
    }
}
```

Os relacionamentos entre as entidades, diferente do modelo relacional, são definidos usando objetos. Por exemplo, considere a classe **Categoria**, que representa a categoria de um produto:

```
namespace Capitulo04.EntityFramework.Models
{
    public class Categoria
    {
        public int CategoriaId { get; set; }
        public string Nome { get; set; }
    }
}
```

Cada produto pertence a uma categoria, e cada categoria pode conter diversos produtos. No modelo baseado em objetos, a categoria à qual um produto pertence é definida pelo campo do tipo **Categoria** e não pelo campo **CategoriaId**. Vejamos o exemplo:

```
namespace Capitulo04.EntityFramework.Models
{
    public class Produto
    {
        public int ProdutoId { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
        public int Estoque { get; set; }

        public Categoria Categoria { get; set; }

        public int IncrementarEstoque(int quantidade)
        {
            this.Estoque += quantidade;
            return this.Estoque;
        }
    }
}
```

Esses campos são chamados **Campos de Navegação**. Na entidade **Categoria**, é possível navegar pelos produtos daquela categoria:

```
namespace Capitulo04.EntityFramework.Models
{
    public class Categoria
    {
        public int CategoriaId { get; set; }
        public string Nome { get; set; }

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

A coleção de objetos relacionada deve ser uma classe que implementa a interface **ICollection<T>** e deve ser sempre inicializada no construtor, conforme o exemplo:

```
namespace Capitulo04.EntityFramework.Models
{
    public class Categoria
    {
        public Categoria()
        {
            this.Produtos = new List<Produto>();
        }

        public int CategoriaId { get; set; }
        public string Nome { get; set; }

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

A interface pode ser declarada no campo relacionado. Isso possibilita criar uma lista de qualquer classe que implemente aquela interface:

```
namespace Capitulo04.EntityFramework.Models
{
    public class Categoria
    {
        public Categoria()
        {
            this.Produtos = new List<Produto>();
        }

        public int CategoriaId { get; set; }
        public string Nome { get; set; }

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

4.5.2. Contexto da conexão

Uma vez criado o modelo de domínio, o próximo passo é criar uma classe derivada de **DbContext** e propriedades que representem coleções das entidades. Isso equivale a criar, no banco de dados, tabelas e relacionamentos.

```
namespace Capitulo04.EntityFramework.Models
{
    public class LojaContext : DbContext
    {
        }
}
```

Cada propriedade que será mapeada para uma tabela deve ser declarada como uma coleção do tipo **DbSet**.

```
namespace Capitulo04.EntityFramework.Models
{
    public class LojaContext : DbContext
    {
        public DbSet<Categoria> Categorias { get; set; }
        public DbSet<Produto> Produtos { get; set; }
    }
}
```

O programa está pronto para ser executado. Para incluir uma categoria de produto na tabela de produtos, é necessário instanciar a classe derivada **DbContext**, adicionar uma instância de uma Entidade (**Produto**, **Categoria**) em uma coleção (instância da classe **DbSet<T>**) e chamar o método **SaveChanges** da classe **DbContext**. O exemplo a seguir inclui uma categoria:

```
var db = new LojaContext();
var c = new Categoria() { Nome = "Eletrodomésticos" };
db.Categorias.Add(c);
db.SaveChanges();
```

Várias perguntas vêm à tona quando o código exibido anteriormente é executado:

- Em qual banco de dados o Entity Framework gravou os dados?
- Como foi definida a chave primária?
- Foram definidas chaves primárias (PK) e chaves estrangeiras (FK)?
- A chave primária é do tipo **Identity**?

A resposta a todas essas perguntas é a mesma: O Entity Framework usou uma série de convenções. Vejamos:

- Por padrão, o EF (Entity Framework) cria um banco de dados no SQL Server Express, se este estiver instalado;
- O EF usa o campo do tipo inteiro que se chame **ID** ou **NomeDaClasseID** como chave primária do tipo **Identity**;
- O nome das tabelas é o nome das classes no plural. Isso nem sempre funciona em português. A classe **Cliente** vira a tabela **Clientes**, mas a classe **Produto** vira a tabela **Produto** e não **Produtos**;
- O nome do banco de dados é o nome da classe derivada de **DbContext**, incluindo o namespace.

O EF dispõe de recursos para personalizar cada decisão tomada pelas convenções internas de nomenclatura e estrutura criada no banco.

O primeiro nível de personalização é passar um parâmetro para o construtor da classe **DbContext**. Esse construtor pode ser:

- **Construtor sem parâmetros**

Um banco de dados é criado no SQL Server com o mesmo nome da classe derivada de **DbContext**.

```
namespace Capitulo04.EntityFramework.Models
{
    public class LojaContext : DbContext
    {
        public LojaContext()
        {

        }

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

- Um parâmetro do tipo string (string de conexão)

Se um parâmetro do tipo string é passado, pode significar uma **string de conexão** ou o nome de uma string de conexão definida no **Web.Config**, na seção **ConnectionStrings**. O exemplo a seguir mostra uma string de conexão sendo passada:

```
namespace Capitulo04.EntityFramework.Models
{
    public class LojaContext : DbContext
    {
        public LojaContext(string conexao) : base(conexao)
        {

        }

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

No exemplo anterior, ao criar uma instância da classe derivada de **DbContext**, deve-se passar a string de conexão:

```
var db = new LojaContext(@"Data Source=localhost\sqlexpress;
    Initial Catalog=EmpresaX;
    Integrated Security=true");

var c = new Categoria() { Nome = "Eletrodomésticos" };
db.Categorias.Add(c);
db.SaveChanges();
```

No caso apresentado anteriormente, a classe derivada recebe um parâmetro do tipo string, mas isso é uma opção.

- Um parâmetro do tipo string (nome de uma conexão)

Usamos a mesma classe **DbContext** anterior, mas com uma string de conexão definida no **Web.Config**:

- Arquivo **Web.Config**:

```
<connectionStrings>
<add name="minhaConexao" providerName="System.Data.SqlClient"
      connectionString="Data Source =.\SQLEXPRESS;Initial
      Catalog=EmpresaX;Integrated Security=true"/>
</connectionStrings>
```

- Durante a execução do programa:

```
var db = new LojaContext("minhaConexao");
```

- Um parâmetro do tipo `DbConnection`

Se um parâmetro do tipo `string` é passado, pode significar uma string de conexão ou o nome de uma string de conexão definida no `Web.Config`, na seção `ConnectionStrings`.

O exemplo a seguir mostra uma string de conexão sendo passada:

```
namespace Capitulo04.EntityFramework.Models
{
    public class LojaContext : DbContext
    {
        public LojaContext(DbConnection cn) : base(cn, true)
        {

        }

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

Esse construtor aceita dois parâmetros: o primeiro é uma instância de uma classe derivada de `DbConnection`, e o segundo é um parâmetro booleano que indica se a conexão deve ser liberada da memória (método `Dispose()` da interface `IDisposable`).

A não ser que haja algum motivo para deixar a conexão aberta, é melhor que o EF feche a conexão e libere a memória assim que acabar de usar a conexão. Passar o valor `True` para o segundo parâmetro deste construtor garante este comportamento.

4.5.3. Mapeamento

É possível definir com exatidão como o .NET Framework vai criar o banco de dados e quais as definições que existem nos campos. Isso pode ser feito usando atributos definidos para as classes e propriedades ou sobrepondo o método `OnModelCreating` da classe `DbContext`.

O namespace `System.ComponentModel.DataAnnotations.Schema` contém classes derivadas da classe `Attribute`, cujo objetivo é inserir metadados nas classes e propriedades que fornecem informações adicionais que podem ser utilizadas pelo Entity Framework ao criar ou atualizar um modelo de dados. As classes frequentemente utilizadas são as seguintes:

- **Table**

Define o nome da tabela no banco de dados. Se este atributo não foi definido, o nome da tabela é o nome da classe de entidade no plural.

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

namespace Capitulo04.EntityFramework.Models
{
    [Table("Produtos")]
    public class Produto
    {
        public int ProdutoId { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
        public int Estoque { get; set; }

        public Categoria Categoria { get; set; }

        public int IncrementarEstoque(int quantidade)
        {
            this.Estoque += quantidade;
            return this.Estoque;
        }
    }
}
```

- **Column**

Representa as informações de um campo de uma tabela no banco de dados. Este atributo permite definir o nome, a ordem e o tipo de uma coluna.

```
namespace Capitulo04.EntityFramework.Models
{
    [Table("Produtos")]
    public class Produto
    {
        [Column(name: "CodigoDoProduto")]
        public int ProdutoId { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
        public int Estoque { get; set; }

        public Categoria Categoria { get; set; }

        public int IncrementarEstoque(int quantidade)
        {
            this.Estoque += quantidade;
            return this.Estoque;
        }
    }
}
```

- Key

Este atributo define um campo da tabela como chave primária. Toda classe no EF precisa de uma chave primária, mesmo que não exista no banco de dados.

É importante lembrar que, na maioria das vezes, não é preciso indicá-la explicitamente. Caso o campo seja numérico, tenha o nome **Id** ou o nome da classe seguido por **Id**, o Entity Framework automaticamente o define como chave primária.

```
namespace Capitulo04.EntityFramework.Models
{
    public class Categoria
    {
        public Categoria()
        {
            this.Produtos = new List<Produto>();
        }

        [Key]
        public int CategoriaId { get; set; }
        public string Nome { get; set; }

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

O atributo **Key** é obrigatório quando a chave primária é composta. Neste caso, é necessário, também, indicar a ordem dos campos.

```
public class ProdutoCategoria
{
    [Key]
    [Column(Order=1)]
    public int ProdutoId { get; set; }

    [Key]
    [Column(Order = 2)]
    public int CategoriaId { get; set; }
}
```

- **Required**

O atributo **Required** indica que o campo deve ser preenchido com um valor diferente de **Null** e de **String.Empty** (no caso do tipo do campo ser string).

```
namespace Capitulo04.EntityFramework.Models
{
    public class Categoria
    {
        public Categoria()
        {
            this.Produtos = new List<Produto>();
        }

        [Key]
        public int CategoriaId { get; set; }

        [Required]
        public string Nome { get; set; }

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

Os campos que não são requeridos devem ser declarados como **Nullable**, pois o Entity Framework tentará inserir um valor nulo na propriedade:

```
public Nullable<int> Estoque { get; set; }
```

- **MaxLength**

Define o número máximo de caracteres permitidos em um campo. Esta é uma opção interessante, pois não há maneira nativa de limitar o tamanho de uma string no .NET Framework. Se for usado o valor -1 como argumento em um campo do tipo **string** ou **Array de Bytes**, o SQL Server usará **VARCHAR(MAX)** e **VARBINARY(MAX)** respectivamente.

```
public class Cliente
{
    public int ClienteId { get; set; }

    [Required]
    [MaxLength(30)]
    public string Nome { get; set; }

    [MaxLength(-1)]
    public byte[] Foto { get; set; }

}
```

- **NotMapped**

Por padrão, todas as propriedades públicas são mapeadas pelo Entity Framework. Para que uma propriedade não seja gravada ou lida do banco de dados, usa-se o atributo **NotMapped**.

```
public class Cliente
{
    public int ClienteId { get; set; }

    [Required]
    public string Nome { get; set; }

    [NotMapped]
    public string NumeroRevisaoProg { get; set; }
}
```

- **ForeignKey**

O atributo **ForeignKey** define uma chave estrangeira, relacionando duas tabelas (ou duas entidades). Esse atributo somente é necessário caso não tenha uma propriedade do tipo da classe relacionada na classe principal.

Por exemplo, no banco de dados **Northwind**, cada **Produto** pertence a uma **Categoria**, e uma categoria pode pertencer a diversos produtos. Esse relacionamento é gerado automaticamente ao criarmos, na tabela **Produtos**, um campo do tipo **Categoria**:

```
public class Produto
{
    public int ProdutoId { get; set; }

    public string Nome { get; set; }

    public decimal Preco { get; set; }

    public Categoria Categoria { get; set; }
}

public class Categoria
{
    public int CategoriaId { get; set; }

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

Propriedades do tipo descrito anteriormente são chamadas de **propriedades de navegação**, pois são elas que permitem navegar pela estrutura do banco de dados.

Em alguns casos, é interessante ter um campo relacional típico representando a chave primária de outra tabela.

No caso da necessidade de usar apenas o **Id** da tabela, não faz sentido criar um objeto para cada registro lido. Nesse caso, uma propriedade relacionada a outra tabela também pode ser mapeada.

```
public class Produto
{
    public int ProdutoId { get; set; }
    public string Nome { get; set; }
    public decimal Preco { get; set; }
    public int Estoque { get; set; }

    [ForeignKey("Categorias")]
    public int CategoriaId { get; set; }

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

- **DatabaseGenerated**

Campos calculados pelo servidor de dados devem ter o atributo **DatabaseGeneratedAttribute** definido na propriedade. Isso faz com que o Entity Framework não tente alterar ou incluir informações nesse campo.

```
public class Produto
{
    public int ProdutoId { get; set; }
    public string Nome { get; set; }
    public decimal Preco { get; set; }
    public int Estoque { get; set; }

    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public decimal PrecoPromocao { get; protected set; }
}
```

As opções do enumerador **DatabaseGeneratedOption** são as seguintes:

- None = 0;
- Identity = 1;
- Computed = 2.

O Entity Framework não aceita campos calculados (até a versão 6.0), mas isso está previsto para as futuras versões.

4.5.4. Mapeamento por código

Tudo que é possível definir aplicando atributos pode ser definido via código. Isso abre algumas possibilidades interessantes, como criar o mapeamento dinamicamente em tempo de execução. Uma tabela pode ter um campo requerido para um tipo de usuário e não obrigatório para outro tipo. Ao fazer login, o sistema pode analisar as permissões do usuário e definir as regras de negócio que devem ser aplicadas. Usando atributos, seria necessário duplicar as classes em áreas diferentes ou escrever o código para cada situação.

4.5.4.1. Método OnModelCreating

O método **OnModelCreating** da classe **DbContext** pode ser sobreescrito para alterar a maneira como o modelo de domínio é criado. Este método espera receber um parâmetro do tipo **DbModelBuilder**.

```
public class LojaContext : DbContext
{
    private static string conexao = @"Data ...";
    public LojaContext(): base(conexao) {}

    protected override void
        OnModelCreating(DbModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
    }

    public DbSet<Categoria> Categorias { get; set; }
    public DbSet<Produto> Produtos { get; set; }
    public DbSet<Cliente> Clientes { get; set; }
}
```

O exemplo a seguir mostra como configurar para definir o campo **Nome** da tabela **Produto** como **Required** (requerido):

```
public class LojaContext : DbContext
{

    protected override void
        OnModelCreating(DbModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Produto>()
            .Property(d => d.Nome)
            .IsRequired();
    }

}
```

- O método **Entity(Produto)** retorna uma instância da classe de configuração **EntityTypeConfiguration** para a classe **Produto**;
- O método **Property()** retorna uma instância de **System.Data.Entity.ModelConfiguration**;
- O método **IsRequired** define que o campo **Nome** é requerido.

Adiante está um resumo das principais classes e métodos envolvidos no exemplo anterior:

- **Classe DbModelBuilder**

Esta classe é o ponto central do Entity Framework, quando é usado o modo **Code First**. Por meio de uma instância desta classe enviada ao método **OnModelCreating**, é possível configurar todos os mapeamentos.

Vejamos, a seguir, seus principais métodos:

- **Entity<T>**: Registra um tipo como parte do modelo e retorna uma instância da classe **EntityTypeConfiguration**. É usado para modificar o comportamento das propriedades, assim como as classes de atributos;
 - **Ignore<T>**: Exclui o tipo do modelo de dados. Equivalente ao atributo **[ignore]**;
 - **ComplexType<T>**: Registra um tipo como sendo um **ComplexType** para ser usado dentro de uma classe mapeada.
-
- **Classe EntityTypeConfiguration<T>**

Permite realizar configurações em um modelo de entidade. É obtido por meio do método **Entity<T>** da classe **DbModelBuilder**.

Vejamos, a seguir, seus principais métodos:

- **ToTable<T>(string NomeDaTabela, string NomeDoSchema)**: Relaciona uma classe a uma tabela e schema. É equivalente a usar o atributo **Table**;
- **Ignore<T>**: Ignora uma classe e todas as suas propriedades em um mapeamento ou uma propriedade em particular;
- **Property**: Retorna uma referência a uma propriedade de uma classe, permitindo configurar seu comportamento.

O exemplo a seguir define o nome de uma tabela e a chave primária do tipo **identity**:

```
protected override
void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    //Define que a classe Produto será
    //mapeada para a tabela Products
    modelBuilder.Entity<Produto>().ToTable("Products", "dbo");

    //Define a chave primária da tabela Products
    modelBuilder.Entity<Produto>().HasKey(x => x.ProdutoId);

    //Define que a chave primária é do tipo Identity
    modelBuilder
        .Entity<Produto>()
        .Property(x => x.ProdutoId)
        .HasDatabaseGeneratedOption(
            DatabaseGeneratedOption.Identity);
}
```

Neste outro exemplo, um relacionamento é definido entre a tabela **Produtos** e **Categorias**:

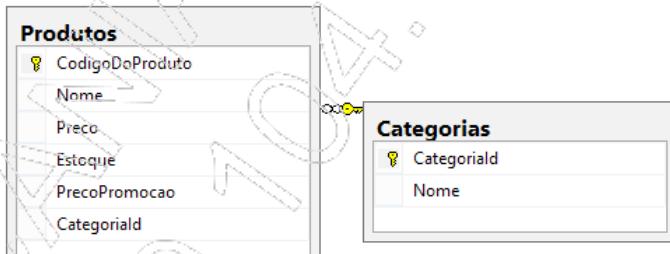
```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    //Define que a classe Produto
    //será mapeada para a tabela Products
    modelBuilder.Entity<Produto>().ToTable("Produtos", "dbo");

    //Define a chave primária da tabela Products
    modelBuilder.Entity<Produto>().HasKey(x => x.ProdutoId);

    //Define que a chave primária é do tipo Identity
    modelBuilder.Entity<Produto>()
        .Property(x => x.ProdutoId)
        .HasDatabaseGeneratedOption(
            DatabaseGeneratedOption.Identity);

    //Define que um produto tem muitas categorias
    modelBuilder.Entity<Categoria>()
        .HasMany(x => x.Produtos)
        .WithRequired(x => x.Categoria);
}
```



4.5.5. Database Initializer

Quando uma classe mapeada é utilizada, o Entity Framework inicia o processo de criação ou conexão com o banco de dados existente. Este processo inicial pode ser definido de diversas maneiras: o banco de dados pode ser excluído e criado a cada mudança de estrutura, pode ter um processo de migração controlado e pode reverte uma migração que tenha apresentado alguma falha.

Por padrão, o Entity Framework vai criar o banco de dados apenas se este não existir no servidor informado. Esse processo todo é feito usando classes que implementam a interface **IDatabaseInitializer**. As seguintes classes (Providers) fazem parte do Entity Framework:

- **DropCreateDatabaseAlways<contexto>**: O banco de dados é excluído e gerado novamente cada vez que é usado;
- **DropCreateDatabaseIfModelChanges<contexto>**: O banco de dados somente será criado se o modelo de dados for alterado. O Entity Framework sempre verifica alterações no modelo e não no banco de dados;
- **CreateDatabaseIfNotExists<contexto>**: O banco de dados é criado apenas se não existir. Este é o padrão.

Para alterar o padrão, basta declarar no construtor da classe derivada de **DbContext**:

```
static LojaContext()
{
    Database.SetInitializer(
        new DropCreateDatabaseAlways<LojaContext>());
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

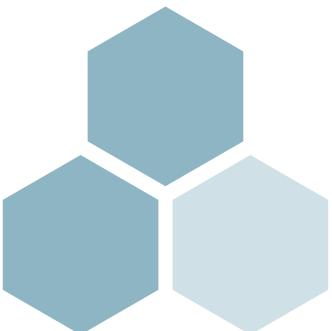
- O Entity Framework pode gerar as classes para armazenar as informações de um banco de dados automaticamente. Podemos criar o modelo usando o EF Designer (**Model First**) ou usando um banco de dados existente e extraíndo o modelo a partir da estrutura das tabelas (**Database First**);
- Para criar um modelo vazio ou a partir de um banco de dados, é necessário adicionar ao projeto um **ADO.NET Entity Data Model**;
- A string de conexão pode ficar armazenada no arquivo Web.Config, na seção **<connectionstrings>**. Quando instanciar uma classe derivada de DbContext usando o construtor que aceita uma string, esta deve ser no formato **name=xxx**, em que **xxx** é o nome da string de conexão armazenada;
- O **Entity Framework** é uma ferramenta que permite converter dados em formato de tabelas relacionadas para objetos de memória e vice-versa;
- **DbContext** é a classe principal do Entity Framework;
- **Code First** é a técnica de escrever primeiramente o modelo de domínio e, então, criar o banco de dados a partir deste modelo, usando o Entity Framework;
- **DbSet** é a classe do .NET Framework que permite definir um conjunto de objetos que serão lidos ou gravados em um banco de dados.



4

Entity Framework

Teste seus conhecimentos



1. Em um projeto MVC, as classes foram geradas a partir do banco de dados. Trata-se de:

- a) Code First
- b) Database After
- c) Database First
- d) Code Database
- e) Database Code

2. A classe que representa o mapeamento de uma classe com uma tabela no banco de dados se chama:

- a) IEnumerable
- b) DbContext
- c) DbSet
- d) Controller
- e) HashSet

3. Para adicionar um objeto na coleção de objetos, usamos qual método?

- a) Add
- b) Insert
- c) SaveChanges
- d) Update
- e) Include

4. Para sincronizarmos um objeto com o banco de dados, usamos qual método?

- a) Add
- b) Insert
- c) SaveChanges
- d) Update
- e) Include

5. Qual atributo podemos incluir em uma propriedade do modelo para indicar que o campo a ser mapeado é obrigatório?

- a) Required
- b) Mapped
- c) Key
- d) NotNullable
- e) NotEmpty



4

Entity Framework

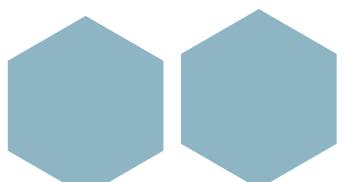


Mãos à obra!

SCALING
50.
ALTDAS
10007-67



Editora
IMPACTA



Objetivos:

Neste laboratório, nós definiremos o banco de dados a ser usado na aplicação, e aplicaremos o procedimento **Database First**, para geração das entidades a serem usadas ao longo do projeto. Apesar dos procedimentos **Model First** e **Code First** serem apresentados neste capítulo, nós os aplicaremos em capítulos posteriores, o que fará mais sentido para manter a integridade do projeto.

Laboratório 1

A - Preparação do projeto, do banco de dados e das entidades

1. Na pasta **Labs**, crie uma solution vazia (**Blank solution**) chamada **Capitulo04.Labs**;
2. Copie o projeto **Lab.MVC** do solution **Capitulo03.Labs** para a pasta do seu novo solution, **Capitulo04.Labs**;
3. Adicione este projeto ao novo solution. Execute a aplicação para testar se está tudo ok;
4. Neste capítulo, necessitaremos do SQL Server. O script a seguir deve ser usado para a criação do banco de dados e das tabelas que usaremos. **Não se esqueça de alterar o caminho dos arquivos do banco de dados para corresponder à sua pasta**:

```
USE MASTER;
GO

CREATE DATABASE DB_VENDAS
ON PRIMARY (NAME=N'DB_VENDAS', FILENAME=N'D:\Documentos\Projetos\Dados\DB_VENDAS.MDF')
LOG ON (NAME=N'DB_VENDAS_LOG', FILENAME=N'D:\Documentos\Projetos\Dados\DB_VENDAS_LOG.LDF')
GO

USE DB_VENDAS;
GO

CREATE TABLE TBClientes
(
    Documento      varchar(14) not null,
    Nome           varchar(60) not null,
    Telefone        varchar(20) not null,
    Email          varchar(60) not null
    PRIMARY KEY (Documento),
    CHECK(LEN(Documento) = 11 OR LEN(Documento) = 14)
);
GO
```

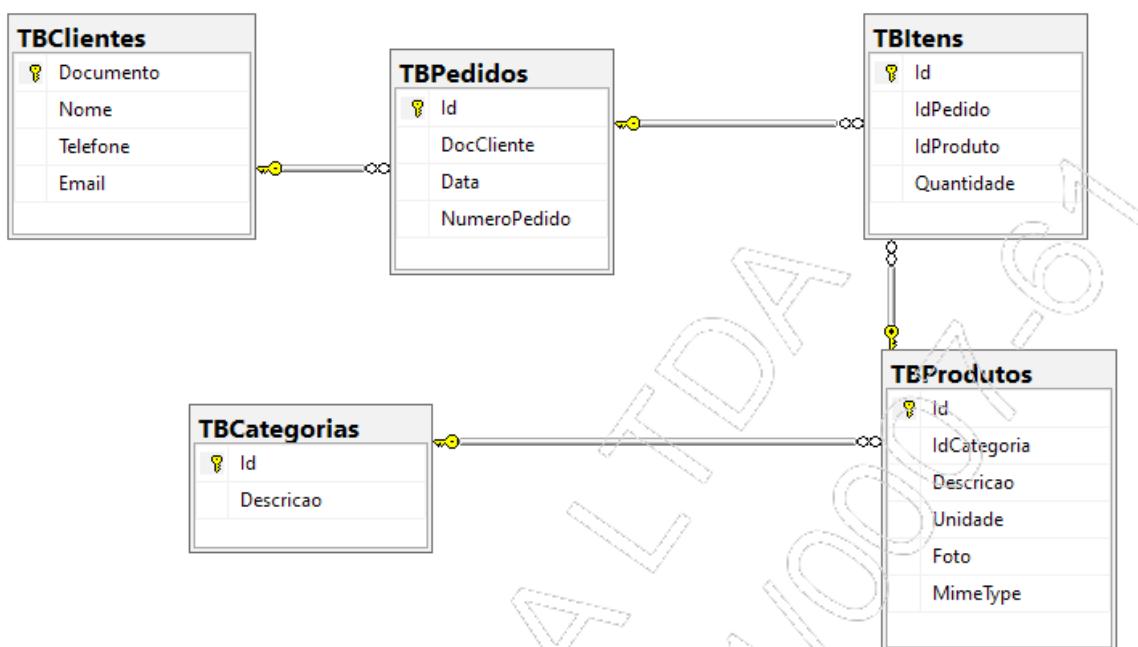
```
CREATE TABLE TBPedidos
(
    Id          int IDENTITY(1,1) not null,
    DocCliente  varchar(14) not null,
    Data        datetime not null,
    NumeroPedido  varchar(20) not null,
    PRIMARY KEY (Id),
    FOREIGN KEY (DocCliente) REFERENCES TBClientes (Documento)
);
GO

CREATE TABLE TBCategorias
(
    Id          int IDENTITY(1,1) not null,
    Descricao   varchar(20) not null,
    PRIMARY KEY (Id)
);
GO

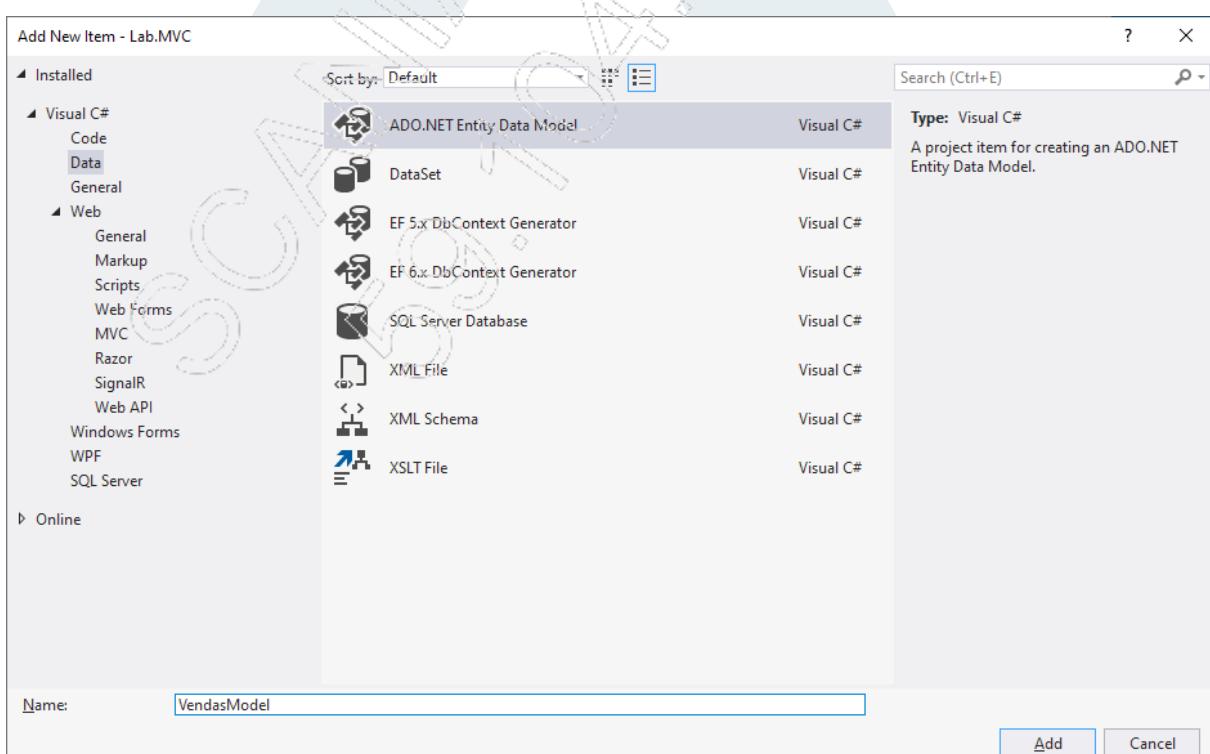
CREATE TABLE TBProdutos
(
    Id          int IDENTITY(1,1) not null,
    IdCategoria int not null,
    Descricao   varchar(50) not null,
    Unidade     varchar(10) not null,
    Foto        varbinary(MAX),
    MimeType    varchar(20),
    Preco       float not null,
    PRIMARY KEY (Id),
    FOREIGN KEY (IdCategoria) REFERENCES TBCategorias (Id)
);
GO

CREATE TABLE TBItems
(
    Id          int IDENTITY(1,1) not null,
    IdPedido    int not null,
    IdProduto   int not null,
    Quantidade  float not null,
    PRIMARY KEY (Id),
    FOREIGN KEY (IdPedido) REFERENCES TBPedidos (Id),
    FOREIGN KEY (IDProduto) REFERENCES TBProdutos (Id)
);
GO
```

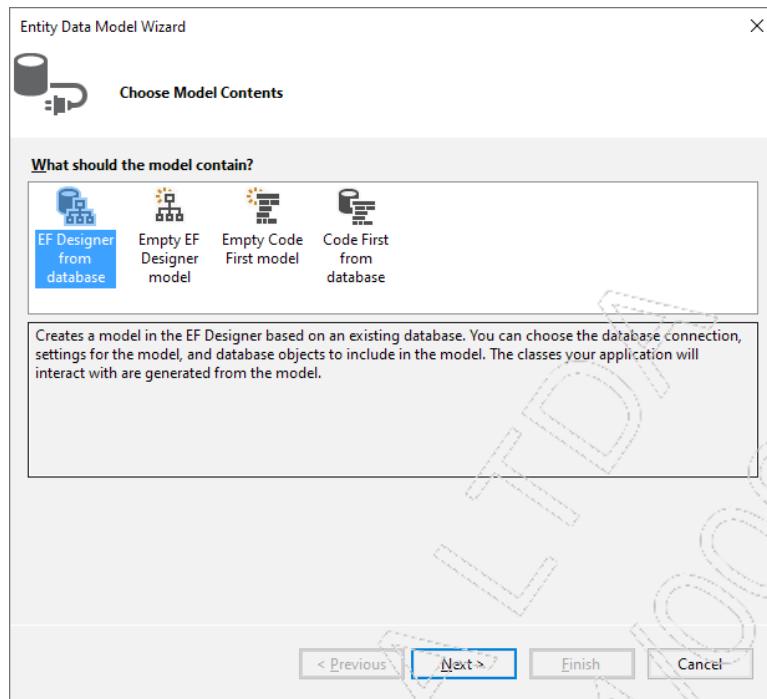
Após executar o script, o banco de dados gerado deve ter um diagrama semelhante ao mostrado adiante:



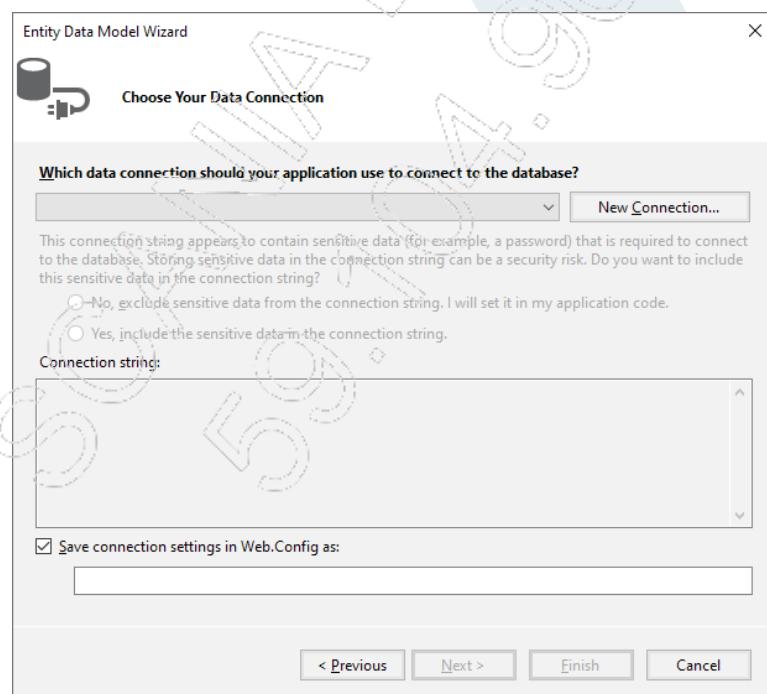
5. Uma vez que o banco de dados tenha sido criado, vamos definir os recursos do **Entity Framework** na aplicação. Na pasta **Models**, clique com o botão direito do mouse e selecione **Add / New Item / ADO.NET Entity Data Model**. Atribua o nome **VendasModel**:



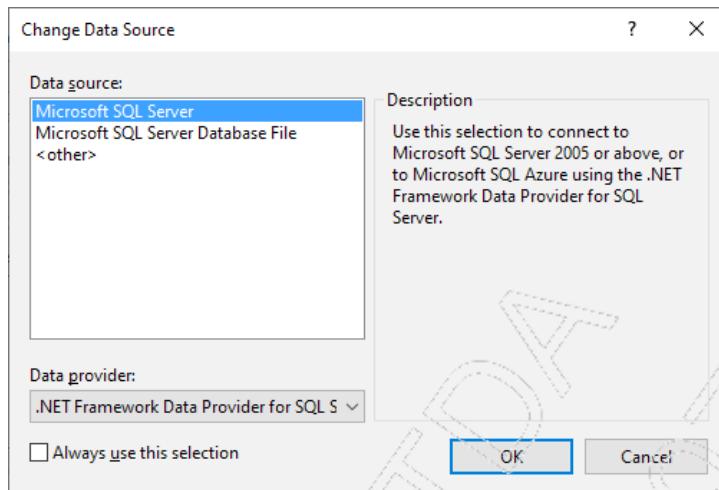
6. Na próxima etapa, selecione a opção EF Designer From Database:



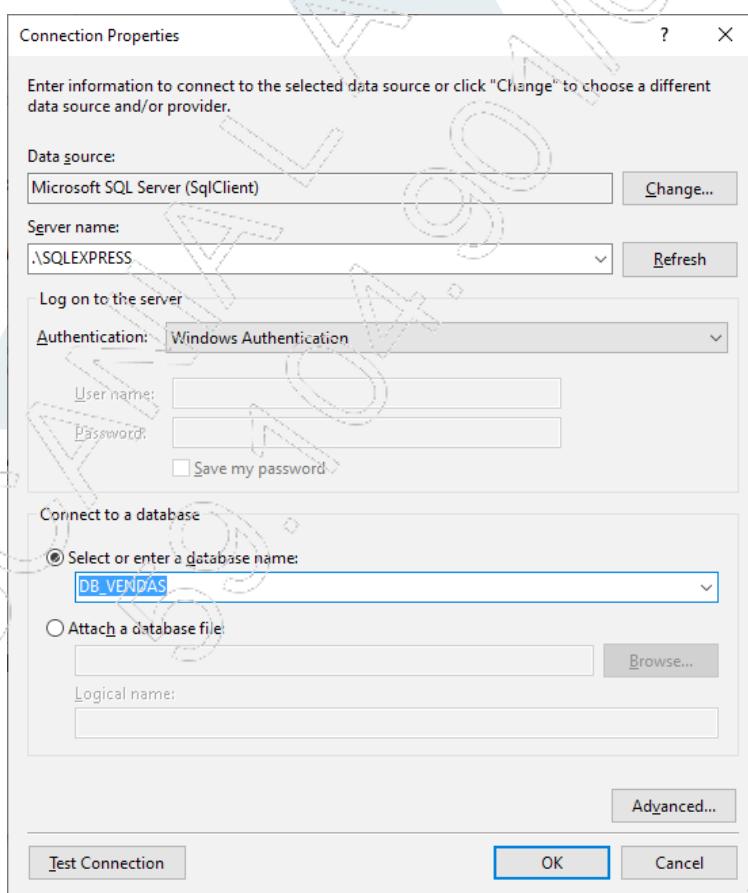
7. Na próxima etapa, clique em New Connection...:



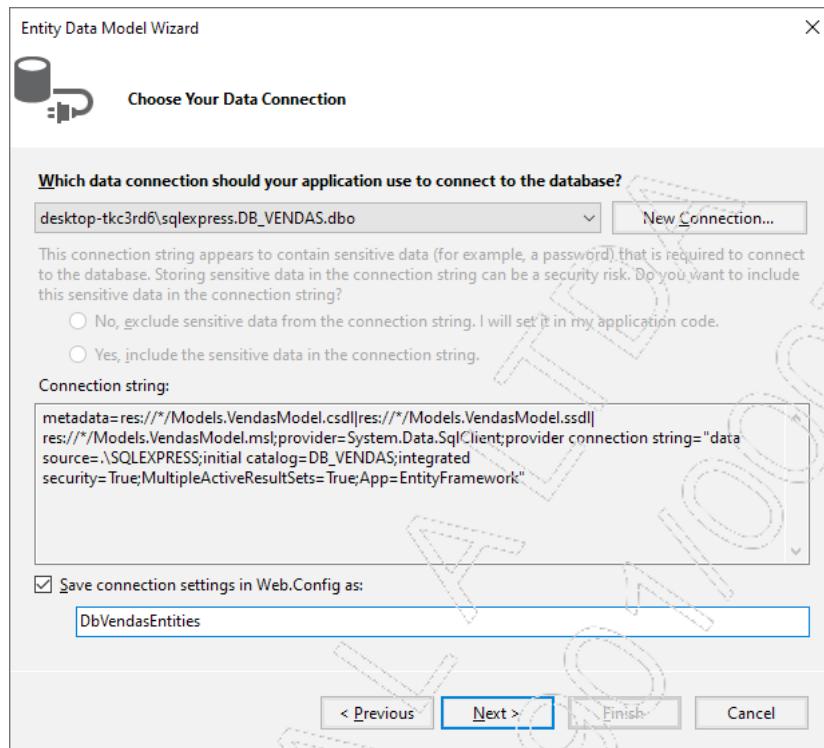
8. Escolha a opção Microsoft SQL Server:



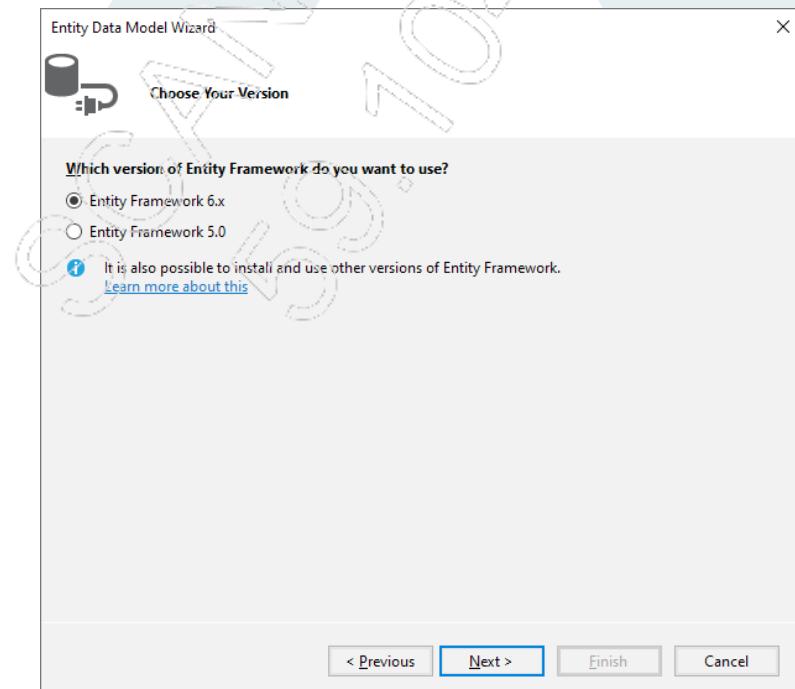
9. No próximo passo, forneça as informações do banco de dados (novamente, é importante fornecer as informações do seu banco de dados):



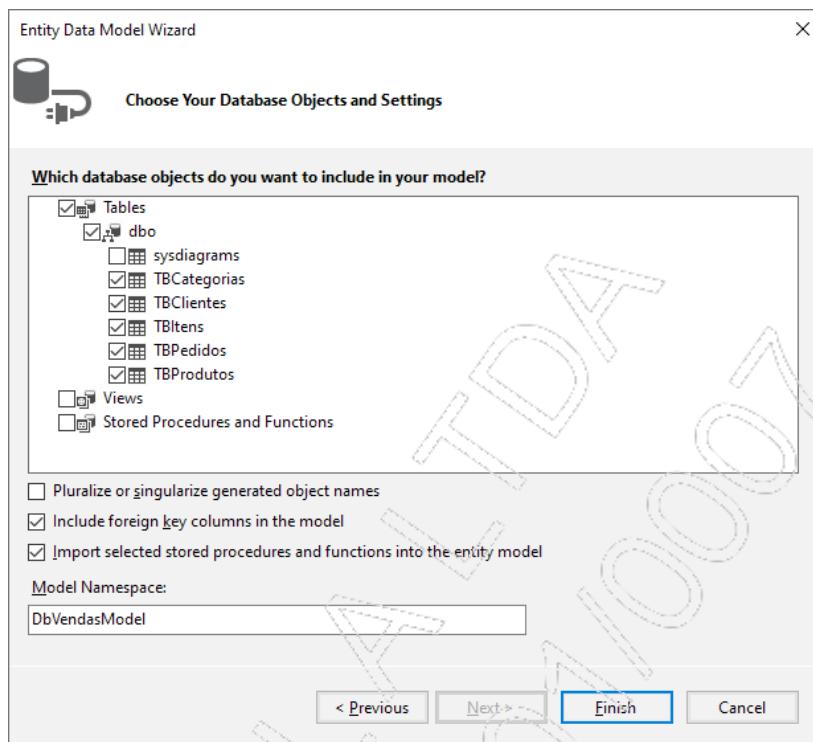
10. Vale lembrar que, na elaboração deste material, a conexão usada não utilizou usuário e senha. Caso sejam utilizados, deixe a opção "Yes, include the sensitive data in the connection string". Esta opção pode ser vista na tela a seguir. Inclua o nome **DbVendasEntities** para a conexão:



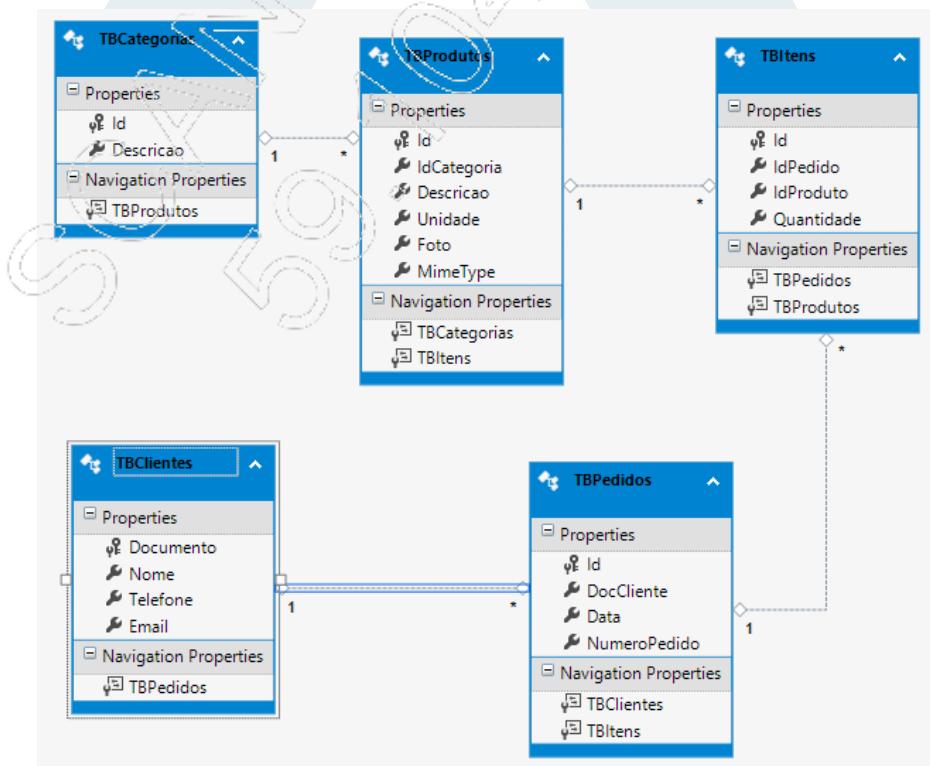
11. Escolha a opção Entity Framework 6.x:



12. Na próxima etapa, selecione as tabelas que foram criadas no banco de dados. O nome do model será **DbVendasModel**:



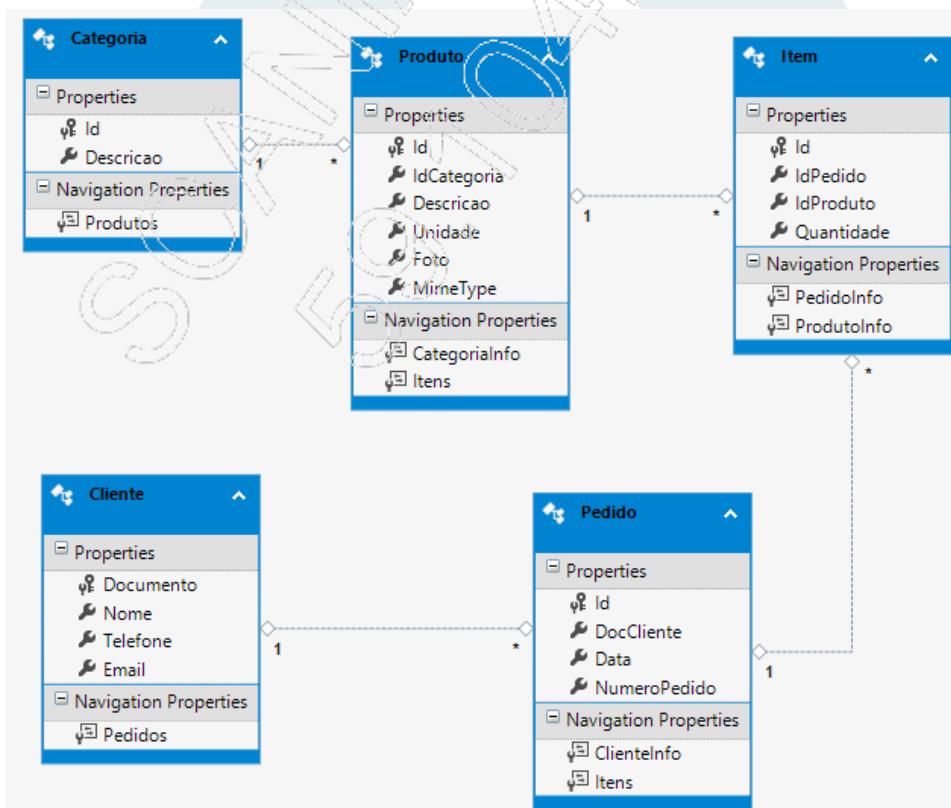
13. A finalização desta etapa fará com que as classes representando as entidades e os contextos sejam criados. Num primeiro momento, teremos o modelo mostrado a seguir:



14. Por questões de regras de nomenclatura, atualizaremos os nomes das classes, das propriedades de navegação e dos Entity Sets. Para esta tarefa, é imprescindível que as alterações sejam realizadas no modelo, pois o mapeamento entre as classes e o banco de dados são alterados pelo Visual Studio. Jamais devemos realizar alterações diretamente nas classes, sob o risco de perdermos o benefício da atualização do mapeamento. Faremos as seguintes alterações:

Tabela	Classe	Propriedades de Navegação	Entity Set
TBClientes	Cliente	Pedidos	Clientes
TBPedidos	Pedido	ClientelInfo Itens	Pedidos
TBItens	Item	PedidoInfo ProdutoInfo	Itens
TBProdutos	Produto	Itens CategorialInfo	Produtos
TBCategorias	Categoria	Produtos	Categorias

Com estas informações, o modelo passa a ser:



É importante salvar o modelo a cada alteração realizada;

15. Observe que as classes também foram alteradas. Analise os seguintes arquivos no projeto:

Categoria.cs, Cliente.cs, Item.cs, Pedido.cs e Produto.cs

E o arquivo contendo a classe do contexto:

VendasModel.Context.cs

B - Cadastro de Clientes

16. A partir de agora, iniciaremos o desenvolvimento da aplicação considerando a persistência no banco de dados. Iniciaremos pelo cadastro de clientes. No projeto **Lab.MVC**, crie a pasta **Data**. Nesta pasta, inclua uma classe chamada **ClientesDao**. Esta classe conterá métodos estáticos responsáveis pela manipulação dos dados. Escreveremos métodos para incluir, buscar, listar, remover e alterar clientes:

```
using Lab.MVC.Models;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace Lab.MVC.Data
{
    public class ClientesDao
    {
        //método para incluir um novo cliente
        public static void IncluirCliente(Cliente cliente)
        {
            using(var ctx = new DbVendasEntities())
            {
                ctx.Clientes.Add(cliente);
                ctx.SaveChanges();
            }
        }

        //método para buscar um cliente pelo número do documento
        public static Cliente BuscarCliente(string documento)
        {
            using (var ctx = new DbVendasEntities())
            {
                return ctx.Clientes.FirstOrDefault(p =>
                    p.Documento.Equals(documento));
            }
        }
    }
}
```

```

//método para listar todos os clientes
public static IEnumerable<Cliente> ListarClientes()
{
    using (var ctx = new DbVendasEntities())
    {
        return ctx.Clientes.ToList();
    }
}

//método para alterar um cliente
public static void AlterarCliente (Cliente cliente)
{
    using (var ctx = new DbVendasEntities())
    {
        ctx.Entry<Cliente>(cliente).State = EntityState.Modified;
        ctx.SaveChanges();
    }
}

//método para remover um cliente
public static void RemoverCliente(Cliente cliente)
{
    using (var ctx = new DbVendasEntities())
    {
        ctx.Entry<Cliente>(cliente).State = EntityState.Deleted;
        ctx.SaveChanges();
    }
}
}

```

17. Para que o cadastro seja consistente, vamos adicionar alguns validadores na classe **Cliente**. Abra a classe e adicione os atributos indicados:

```

namespace Lab.MVC.Models
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;

    public partial class Cliente
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Cliente()
        {
            this.Pedidos = new HashSet<Pedido>();
        }

        [Required]
        [StringLength(14)]
        public string Documento { get; set; }
    }
}

```

```
[Required]
public string Nome { get; set; }

[Required]
public string Telefone { get; set; }

[Required]
[DataType(DataType.EmailAddress)]
public string Email { get; set; }

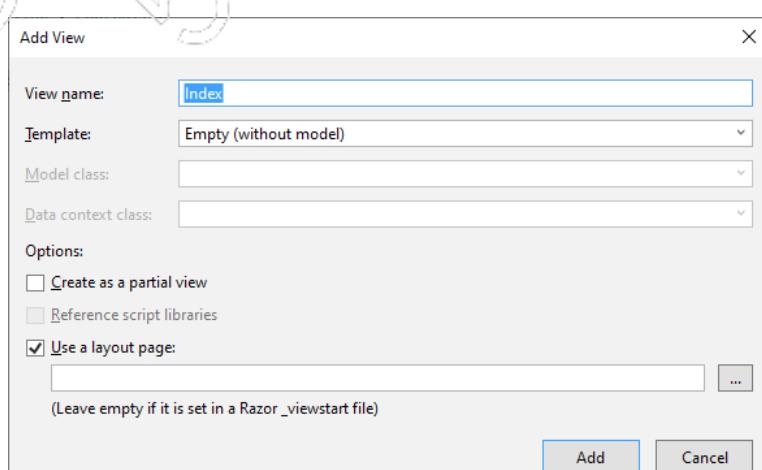
[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2227:CollectionPropertiesShouldBeReadOnly")]
public virtual ICollection<Pedido> Pedidos { get; set; }
}
```

18. Com estes validadores, e com a classe **ClientsDao** criada, estamos prontos para definir os recursos para cadastro de clientes. Na pasta **Controllers**, adicione um novo controller chamado **ClientsController**. O código da classe gerada deve ser semelhante ao mostrado a seguir:

```
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ClientsController : Controller
    {
        // GET: Clients
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

19. A partir do action **Index**, adicione uma view sem modelo. Esta view representará o menu de opções acerca do cadastro de clientes. Nela, teremos as opções para incluir um novo cliente e para listar os clientes. Clique com o botão direito do mouse sobre o action e selecione a opção **Add View**:



20. Na view gerada, inclua as seguintes instruções:

```
@{  
    ViewBag.Title = "Clientes";  
}  
  
<h2>Gestão de clientes - Escolha uma opção</h2>  
  
<ul>  
    <li>@Html.ActionLink("Incluir novo cliente", "Incluir")</li>  
    <li>@Html.ActionLink("Listar Clientes", "Listar")</li>  
</ul>
```

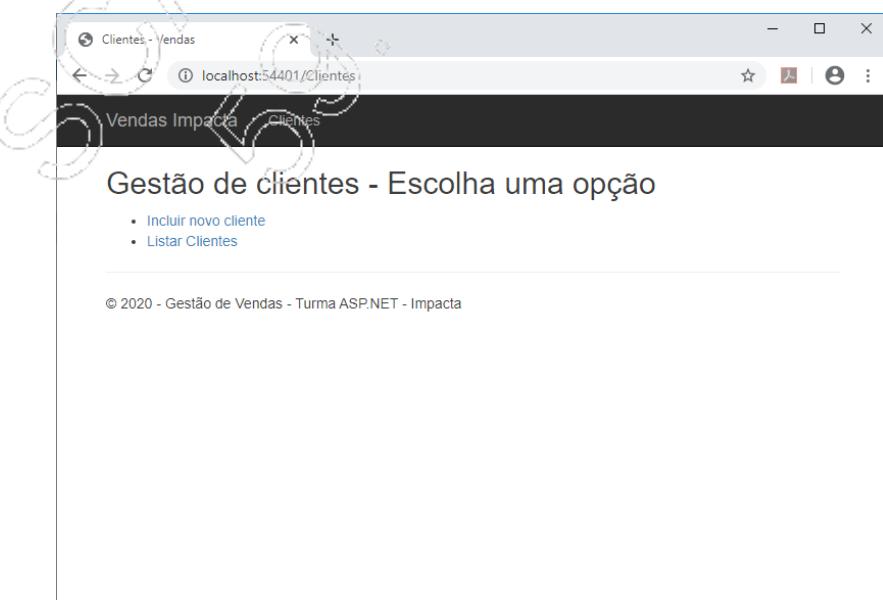
21. Altere o layout para acessar esta view:

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <title>@ViewBag.Title - Vendas</title>  
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />  
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/  
css" />  
    <script src="~/Scripts/modernizr-2.8.3.js"></script>  
</head>  
<body>  
    <div class="navbar navbar-inverse navbar-fixed-top">  
        <div class="container">  
            <div class="navbar-header">  
                <button type="button" class="navbar-toggle"  
                    data-toggle="collapse" data-target=".navbar-collapse">  
                    <span class="icon-bar"></span>  
                    <span class="icon-bar"></span>  
                    <span class="icon-bar"></span>  
                </button>  
                @Html.ActionLink("Vendas Impacta", "Index", "Home",  
                    new { area = "" }, new { @class = "navbar-brand" })  
            </div>  
            <div class="navbar-collapse collapse">  
                <ul class="nav navbar-nav">  
                    <li>  
                        @Html.ActionLink("Clientes", "Index", "Clientes")  
                    </li>  
                </ul>  
            </div>  
        </div>  
    </div>
```

```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Gestão de Vendas -
        Turma ASP.NET - Impacta</p>
    </footer>
</div>

<script src="~/Scripts/jquery-3.3.1.min.js"></script>
<script src="~/Scripts/bootstrap.min.js"></script>
</body>
</html>
```

22. Execute a aplicação para verificar se o novo action está sendo chamado corretamente:



23. Agora vamos preparar para incluir um novo cliente. Adicione o action **Incluir** no controller **Clientes**. A versão GET do método acessará o formulário, e a versão POST receberá os dados do formulário e os enviará para o banco de dados. Como usaremos um formulário tipado, nossa view terá a referência à classe Cliente. Neste caso, ao criar a view, selecionaremos o template **Create**, com a classe **Cliente** como modelo. Primeiro, escreveremos o método no controller:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Web.Mvc;

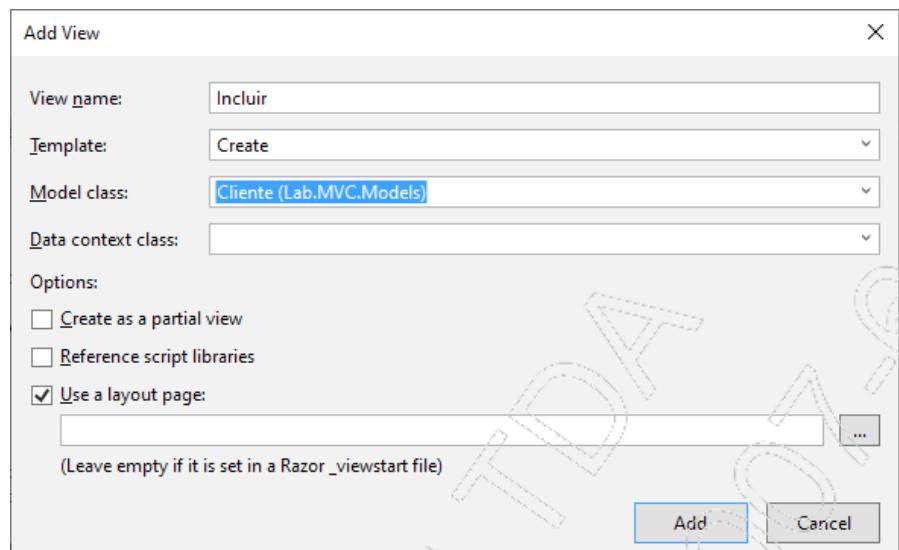
namespace Lab.MVC.Controllers
{
    public class ClientesController : Controller
    {
        // GET: Clientes
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Incluir(Cliente cliente)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            try
            {
                ClientesDao.IncluirCliente(cliente);
                return RedirectToAction("Listar");
            }
            catch (Exception ex)
            {
                ViewBag.MensagemErro = ex.Message;
                return View("_Erro");
            }
        }
    }
}
```

Observe que escrevemos o action **Incluir** na versão POST com um bloco **try / catch**, transferindo o usuário para a página de erro, caso algum ocorra. Em caso de inclusão com sucesso, transferimos o usuário para a lista de clientes. Esta tarefa será executada corretamente quando tivermos o action responsável pela listagem devidamente implementado;

24. Com o action **Incluir** definido, vamos adicionar a view:

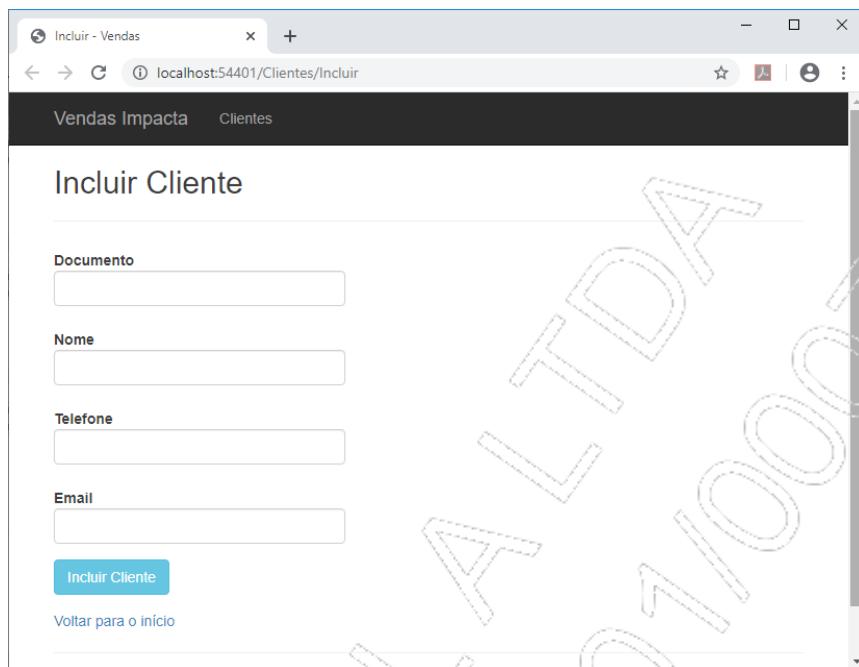


25. Na view gerada, realizaremos alguns ajustes no título, no botão e no link adicionado ao final da página. O restante do código foi omitido:

```
@model Lab.MVC.Models.Cliente  
  
{@  
    ViewBag.Title = "Incluir";  
}  
  
<h2>Incluir Cliente</h2>  
  
@using (Html.BeginForm())  
{  
    @Html.AntiForgeryToken()  
  
    <div class="form-horizontal">  
        <hr />  
        //código omitido  
        <div class="form-group">  
            <div class="col-md-offset-2 col-md-10">  
                <input type="submit" value="Incluir Cliente"  
                      class="btn btn-info" />  
            </div>  
        </div>  
    </div>  
}  
  
<div>  
    @Html.ActionLink("Voltar para o início", "Index")  
</div>
```

Faremos estas mesmas alterações nas demais views;

26. Execute a aplicação e, no menu de opções, chame esta view para verificar se está tudo correto:



27. Inclua, no controller, o action para listar os clientes;

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ClientesController : Controller
    {
        // GET: Clientes
        public ActionResult Index()
        {
            return View();
        }

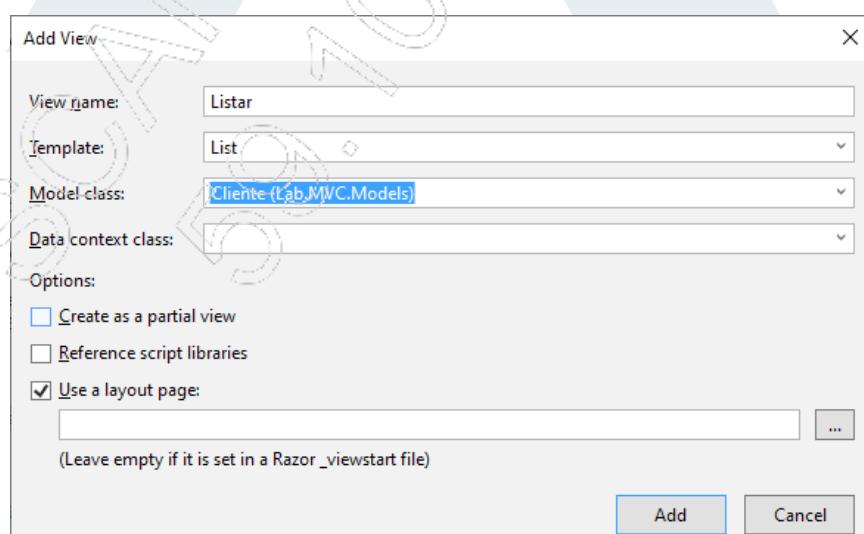
        [HttpGet]
        public ActionResult Incluir()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Incluir(Cliente cliente)
        {
```

```
if (!ModelState.IsValid)
{
    return View();
}
try
{
    ClientesDao.IncluirCliente(cliente);
    return RedirectToAction("Listar");
}
catch (Exception ex)
{
    ViewBag.MensagemErro = ex.Message;
    return View("_Erro");
}

public ActionResult Listar()
{
    try
    {
        return View(ClientesDao.ListarClientes());
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
}
```

28. Adicione a view para este controller. Use o template **List**:



29. Na view produzida, também realizaremos alguns ajustes funcionais:

```
@model IEnumerable<Lab.MVC.Models.Cliente>

@{
    ViewBag.Title = "Listar";
}



## Listar Clientes



@Html.ActionLink("Novo Cliente", "Incluir")



| @Html.DisplayNameFor(model => model.Documento) | @Html.DisplayNameFor(model => model.Nome) | @Html.DisplayNameFor(model => model.Telefone) | @Html.DisplayNameFor(model => model.Email) |  |
|------------------------------------------------|-------------------------------------------|-----------------------------------------------|--------------------------------------------|--|
|------------------------------------------------|-------------------------------------------|-----------------------------------------------|--------------------------------------------|--|


```

30. Com a implementação do action **Listar**, já podemos adicionar um cliente, sabendo que ocorrerá o direcionamento para esta listagem. Vamos então executar a aplicação e incluir um usuário para visualizar todo o fluxo:

The image contains two screenshots of a web application interface. The top screenshot shows the 'Incluir - Vendas' (Create) page. It has fields for 'Documento' (56435746087), 'Nome' (Ernesto Souza), 'Telefone' (3254-2200), and 'Email' (ernesto@mail.com). A blue 'Incluir Cliente' button is at the bottom. The bottom screenshot shows the 'Listar - Vendas' (List) page, displaying a single client entry with the same details: Documento 56435746087, Nome Ernesto Souza, Telefone 3254-2200, and Email ernesto@mail.com. There are links for 'Alterar', 'Ver Detalhes', and 'Remover'.

31. Na sequência, implementaremos os actions para alterar, mostrar os detalhes e remover um cliente. Com exceção da exibição dos detalhes, os outros dois terão versão GET e POST. Na versão GET, os três actions terão exatamente o mesmo comportamento: receberão o documento do cliente como o parâmetro **id**, buscarão pelo cliente com este parâmetro e, encontrando o cliente, enviarão seus dados para a respectiva view. Como o código é o mesmo, escreveremos apenas uma view e faremos com que os demais actions a chamem. Sendo assim, no controller **Clientes**, escreva o seguinte action generalizado:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ClientesController : Controller
    {
        // GET: Clientes
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir()
        {
            return View();
        }

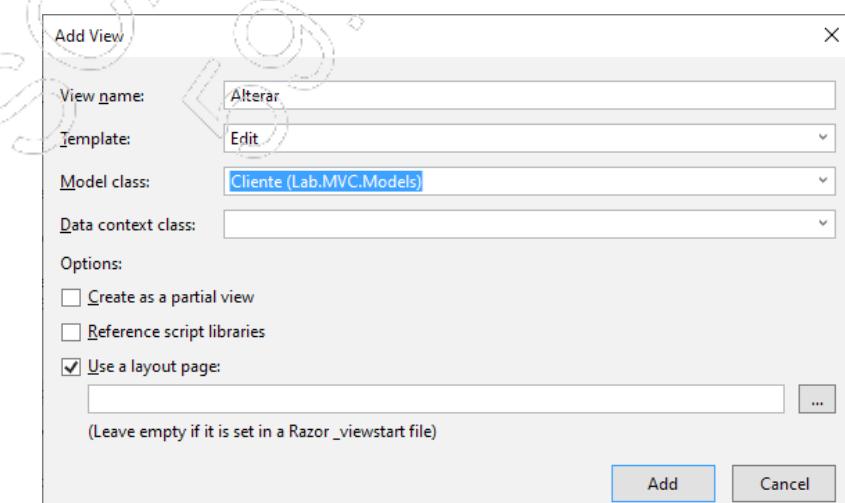
        [HttpPost]
        public ActionResult Incluir(Cliente cliente)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            try
            {
                ClientesDao.IncluirCliente(cliente);
                return RedirectToAction("Listar");
            }
            catch (Exception ex)
            {
                ViewBag.MensagemErro = ex.Message;
                return View("_Erro");
            }
        }

        public ActionResult Listar()
        {
            try
            {
                return View(ClientesDao.ListarClientes());
            }
            catch (Exception ex)
            {
                ViewBag.MensagemErro = ex.Message;
                return View("_Erro");
            }
        }
    }
}
```

```
private ActionResult Buscar(string id, string viewName)
{
    try
    {
        if(id == null)
        {
            throw new Exception("O documento não foi informado
corretamente");
        }
        var cliente = ClientesDao.BuscarCliente(id);
        if(cliente == null)
        {
            throw new Exception("Nenhum cliente encontrado");
        }
        return View(viewName, cliente);
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

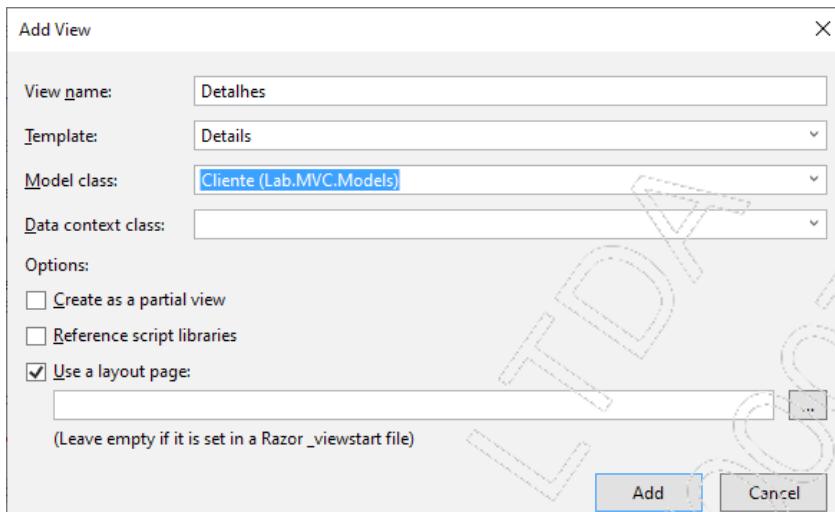
public ActionResult Alterar(string id)
{
    return Buscar(id, "Alterar");
}
public ActionResult Detalhes(string id)
{
    return Buscar(id, "Detalhes");
}
public ActionResult Remover(string id)
{
    return Buscar(id, "Remover");
}
}
```

32. Defina a view Alterar:



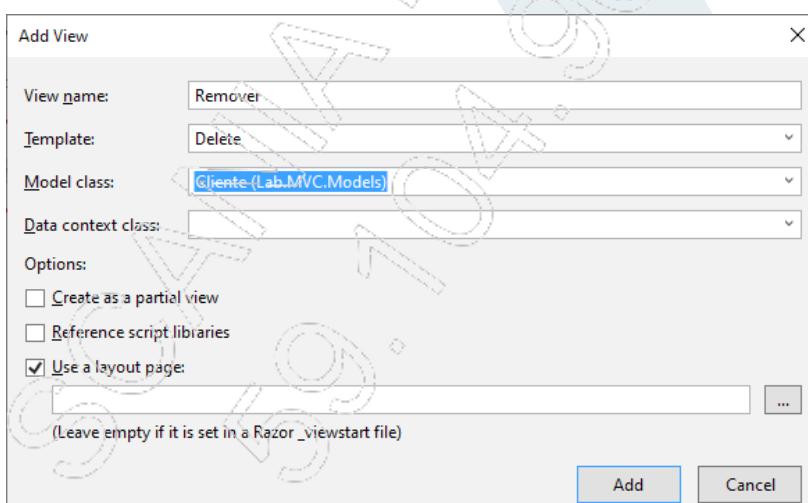
Realizamos as mesmas alterações no botão **submit** e no link que havíamos feito na ocasião da inclusão do cliente;

33. Defina a view **Detalhes**:



Nesta view, eliminamos os links adicionados ao final da página;

34. Defina a view **Remover**:



A view apresenta uma confirmação de exclusão. Se o usuário decidir remover, ele deverá clicar no botão e efetivar a operação. Caso contrário, ele poderá retornar para a lista de clientes. O código é apresentado a seguir:

```
@model Lab.MVC.Models.Cliente

 @{
    ViewBag.Title = "Remover";
}

<h2>Remover</h2>

<h3 class="text-danger">
    Tem certeza que deseja remover este cliente?
</h3>
<div>

    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Documento)
        </dt>

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

        <dt>
            @Html.DisplayNameFor(model => model.Nome)
        </dt>

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

        <dt>
            @Html.DisplayNameFor(model => model.Telefone)
        </dt>

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

        <dt>
            @Html.DisplayNameFor(model => model.Email)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Email)
        </dd>
    </dl>
</div>
```

```

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

    <div class="form-actions no-color">
        @Html.HiddenFor(model => model.Documento)
        <input type="submit" value="Sim, remover" class="btn btn-danger" />
        | @Html.ActionLink("Não, voltar para a lista", "Listar")
    </div>
}
</div>

```

Uma observação importante a respeito da chave primária: tanto para alterar como para remover, a chave primária deve ser enviada para os métodos da classe **ClientesDao** que processarão estas operações. Como é o conteúdo do formulário que é encaminhado para o controller, precisamos garantir que a chave primária seja enviada. Para isso, incluímos a propriedade **Documento** em um campo oculto (hidden) dentro do formulário;

35. Implemente a versão POST dos actions Alterar e Remover:

```

using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ClientesController : Controller
    {
        // GET: Clientes
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Incluir(Cliente cliente)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            try
            {
                ClientesDao.IncluirCliente(cliente);
            }
            catch
            {
                ModelState.AddModelError("Inclusão", "Ocorreu um erro ao tentar incluir o cliente.");
            }
        }
    }
}

```

```
        return RedirectToAction("Listar");
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

public ActionResult Listar()
{
    try
    {
        return View(ClientesDao.ListarClientes());
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

private ActionResult Buscar(string id, string viewName)
{
    try
    {
        if(id == null)
        {
            throw new Exception("O documento não foi informado corretamente");
        }
        var cliente = ClientesDao.BuscarCliente(id);
        if(cliente == null)
        {
            throw new Exception("Nenhum cliente encontrado");
        }
        return View(viewName, cliente);
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

public ActionResult Alterar(string id)
{
    return Buscar(id, "Alterar");
}

public ActionResult Detalhes(string id)
{
    return Buscar(id, "Detalhes");
}

public ActionResult Remover(string id)
{
    return Buscar(id, "Remover");
}
```

```
[HttpPost]
public ActionResult Alterar(Cliente cliente)
{
    try
    {
        ClientesDao.AlterarCliente(cliente);
        return RedirectToAction("Listar");
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

[HttpPost]
public ActionResult Remover(Cliente cliente)
{
    try
    {
        ClientesDao.RemoverCliente(cliente);
        return RedirectToAction("Listar");
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
```

36. Teste estas novas funcionalidades.

C - Cadastro de Produtos

37. Iniciaremos agora o cadastro de produtos. Para este cadastro, implementaremos apenas os actions **Incluir**, **Alterar** e **Listar**. Os demais seguem a mesma estrutura que fizemos para o cadastro de clientes. A novidade neste cadastro é a inclusão de imagens. Outro ponto é a seleção da categoria do produto. Por questões de simplicidade, não implementaremos um cadastro de categoria. Em vez disso, incluiremos algumas categorias manualmente no banco de dados e, na aplicação, escreveremos um método para listá-las. Vamos iniciar definindo a classe **ProdutosDao**, na pasta **Data**. Nesta classe, escreveremos também a listagem das categorias:

```
using Lab.MVC.Models;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace Lab.MVC.Data
{
    public class ProdutosDao
    {
        //método para incluir um novo produto
        public static void IncluirProduto(Produto produto)
        {
            using (var ctx = new DbVendasEntities())
            {
                ctx.Produtos.Add(produto);
                ctx.SaveChanges();
            }
        }

        //método para buscar um produto pelo id
        public static Produto BuscarProduto(int id)
        {
            using (var ctx = new DbVendasEntities())
            {
                return ctx.Produtos.FirstOrDefault(p =>
                    p.Id == id);
            }
        }

        //método para listar todos os produtos
        public static IEnumerable<Produto> ListarProdutos()
        {
            using (var ctx = new DbVendasEntities())
            {
                return ctx.Produtos.ToList();
            }
        }
}
```

```
//método para alterar um produto
public static void AlterarProduto(Produto produto)
{
    using (var ctx = new DbVendasEntities())
    {
        ctx.Entry<Produto>(produto).State = EntityState.Modified;
        ctx.SaveChanges();
    }
}

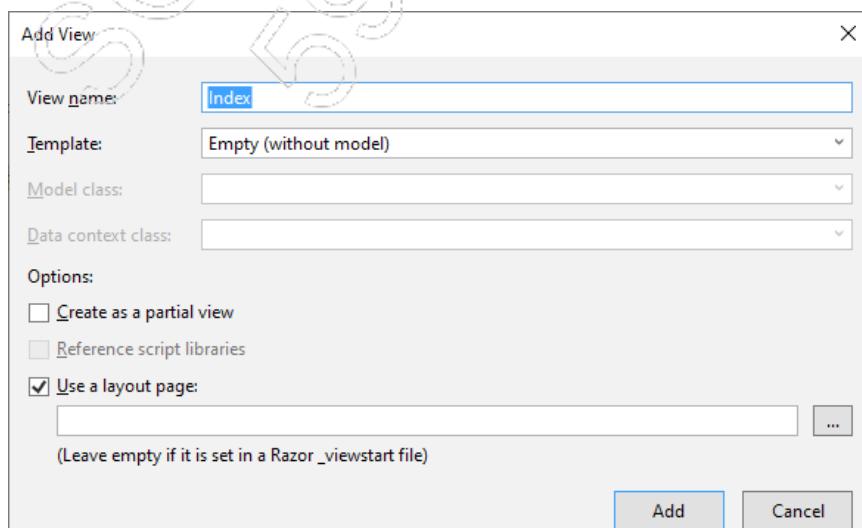
//método para listar todas as categorias
public static IEnumerable<Categoria> ListarCategorias()
{
    using (var ctx = new DbVendasEntities())
    {
        return ctx.Categorias.ToList();
    }
}
}
```

38. Defina um novo controller, chamado **ProdutosController**:

```
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ProdutosController : Controller
    {
        // GET: Produtos
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

39. Adicione a view para o action **Index**:



```
@{  
    ViewBag.Title = "Produtos";  
}  
  
<h2>Gestão de clientes - Escolha uma opção</h2>  
  
<ul>  
    <li>@Html.ActionLink("Incluir novo produto", "Incluir")</li>  
    <li>@Html.ActionLink("Listar Produtos", "Listar")</li>  
</ul>
```

40. Inclua a chamada a este action no Layout:

```
<!DOCTYPE html>  
<html>  
    <head>  
        <meta charset="utf-8" />  
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
        <title>@ViewBag.Title - Vendas</title>  
        <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />  
        <link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />  
        <script src="~/Scripts/modernizr-2.8.3.js"></script>  
    </head>  
    <body>  
        <div class="navbar navbar-inverse navbar-fixed-top">  
            <div class="container">  
                <div class="navbar-header">  
                    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">  
                        <span class="icon-bar"></span>  
                        <span class="icon-bar"></span>  
                        <span class="icon-bar"></span>  
                    </button>  
                    @Html.ActionLink("Vendas Impacta", "Index", "Home",  
                        new { area = "" }, new { @class = "navbar-brand" })  
                </div>  
                <div class="navbar-collapse collapse">  
                    <ul class="nav navbar-nav">  
                        <li>@Html.ActionLink("Clientes", "Index", "Clientes")</li>  
                        <li>@Html.ActionLink("Produtos", "Index", "Produtos")</li>  
                    </ul>  
                </div>  
            </div>  
        </div>
```

```

<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Gestão de Vendas -
            Turma ASP.NET - Impacta</p>
    </footer>
</div>

<script src="~/Scripts/jquery-3.3.1.min.js"></script>
<script src="~/Scripts/bootstrap.min.js"></script>
</body>
</html>

```

41. Vamos agora escrever o action **Incluir** e, em seguida, a sua view. Neste caso, a view permitirá a seleção de uma imagem e, por conta disso, o formulário deve ser definido como **multipart**, já que a imagem trafega em um local diferente do corpo da requisição. Observe que no action nós buscamos as categorias e enviamos seu conteúdo para a view através de um ViewBag. Esta tarefa é bastante comum, uma vez que nossa view está tipada para **Produto**, e a categoria será uma propriedade do produto:

```

using Lab.MVC.Data;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ProdutosController : Controller
    {
        // GET: Produtos
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir()
        {
            ViewBag.ListaDeCategorias = new SelectList(
                ProdutosDao.ListarCategorias(), "Id", "Descricao");
            return View();
        }
    }
}

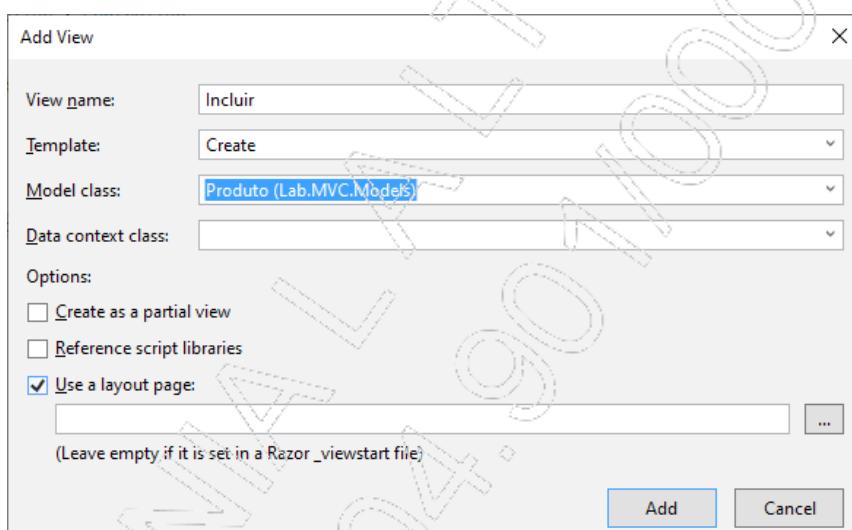
```

Ao objeto ViewBag atribuímos uma instância de **SelectList**. Esta classe é requerida no componente **DropDownListFor** que usaremos na view. Como parâmetros, informamos a coleção, a propriedade que representará o item selecionado (**Id**) e a propriedade que representará o item visualizado (**Descricao**).

No banco de dados, devemos incluir manualmente algumas categorias. Como sugestão, algumas estão mostradas a seguir:

	Id	Descricao
	1	ALIMENTAÇÃO
	2	INFORMÁTICA
	3	VESTUÁRIO
▶	4	CONSTRUÇÃO
	5	MÓVEIS
*	NULL	NULL

Com esses dados prontos, vamos gerar a view para inclusão de produtos:



42. Neste exercício, apresentaremos o conteúdo da View porque temos diversas alterações. O que foi alterado está destacado:

```
@model Lab.MVC.Models.Produto  
  
{@  
    ViewBag.Title = "Incluir Produto";  
}  
  
<h2>Incluir Produto</h2>  
  
@using (Html.BeginForm("Incluir", "Produtos", FormMethod.Post,  
    new { enctype="multipart/form-data" }))  
{  
    @Html.AntiForgeryToken()  
  
    <div class="form-horizontal">  
  
        <hr />
```

```
@Html.ValidationSummary(true, "", new { @class = "text-danger" })
<div class="form-group">
    @Html.LabelFor(model => model.IdCategoria, htmlAttributes:
        new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownListFor(model => model.IdCategoria,
            (SelectList)ViewBag.ListaDeCategorias,
            new { @class = "form-control" })

        @Html.ValidationMessageFor(model => model.IdCategoria, "",
            new { @class = "text-danger" })
    </div>
</div>

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

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

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

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

<div class="form-group">
    @Html.LabelFor(model => model.Foto, htmlAttributes:
    new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        <input type="file" name="Image" class="form-control" />
    </div>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Incluir Produto"
        class="btn btn-info" />
    </div>
</div>
</div>
}

<div>
    @Html.ActionLink("Voltar para o início", "Index")
</div>
```

43. Quando os dados deste formulário forem submetidos, além do conteúdo do formulário, enviamos também a imagem. Para que esta tarefa seja realizada, escrevemos a versão POST do action **Incluir** com dois parâmetros:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ProdutosController : Controller
    {
        // GET: Produtos
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

```
[HttpGet]
public ActionResult Incluir()
{
    ViewBag.ListaDeCategorias = new SelectList(
        ProdutosDao.ListarCategorias(), "Id", "Descricao");
    return View();
}

[HttpPost]
public ActionResult Incluir(Produto produto, HttpPostedFileBase
image)
{
    if (!ModelState.IsValid)
    {
        return Incluir();
    }

    try
    {
        if (image != null)
        {
            produto.MimeType = image.ContentType;
            byte[] bytes = new byte[image.ContentLength];
            image.InputStream.Read(bytes, 0, image.ContentLength);
            produto.Foto = bytes;
        }
        ProdutosDao.IncluirProduto(produto);
        return RedirectToAction("Listar");
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
}
```

O segundo parâmetro é do tipo **HttpPostedFileBase**, e é usado para receber a imagem do formulário. Seu nome deve ser o mesmo que usamos para definir, na view, o elemento `<input type="file" name="Image" ... />`. Se nenhuma foto for escolhida, seu valor é nulo;

44. Para completar a tarefa, vamos adicionar algumas formatações e validações no model **Produto**:

```
namespace Lab.MVC.Models
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;
    using System.Web.Mvc;

    public partial class Produto
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
        "CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Produto()
        {
            this.Itens = new HashSet<Item>();
        }

        public int Id { get; set; }

        [Required]
        [Display(Name = "Categoria")]
        public int IdCategoria { get; set; }

        [Required]
        [Display(Name = "Descrição")]
        public string Descricao { get; set; }

        [Required]
        public string Unidade { get; set; }

        [MaxLength]
        public byte[] Foto { get; set; }

        [HiddenInput(DisplayValue = false)]
        public string MimeType { get; set; }

        [DataType(DataType.Currency)]
        public Nullable<double> Preco { get; set; }

        public virtual Categoria CategoriaInfo { get; set; }
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
        "CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Item> Itens { get; set; }
    }
}
```

O atributo **HiddenInput** faz com que o elemento fique oculto nas listagens ou em outras formas de apresentação;

45. A inclusão do produto redirecionará a view para a lista de produtos. Então vamos deixá-la pronta. No controller **ProdutosController**, vamos incluir o action **Listar** e, além deste action, devemos adicionar outro action que retorna a referência à imagem que será exibida na listagem. Este action se chamará **BuscarFoto**. Vamos então escrever os dois:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ProdutosController : Controller
    {
        // GET: Produtos
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir()
        {
            ViewBag.ListaDeCategorias = new SelectList(
                ProdutosDao.ListarCategorias(), "Id", "Descricao");
            return View();
        }

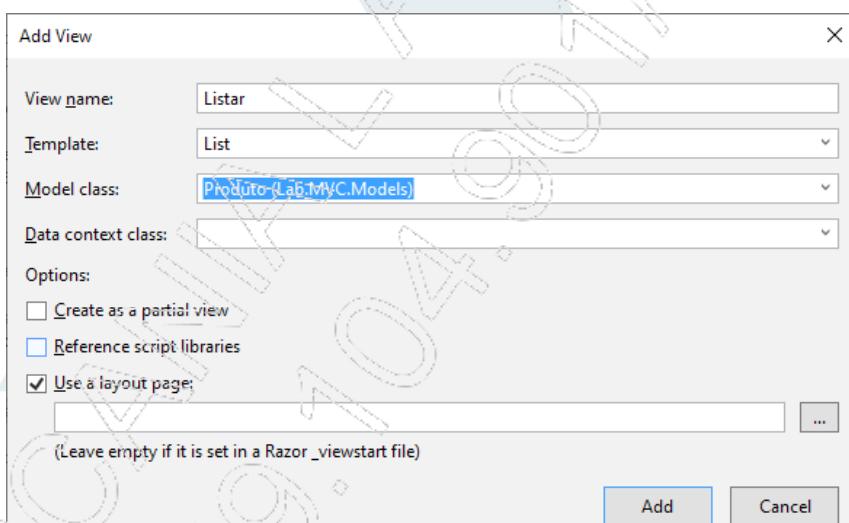
        [HttpPost]
        public ActionResult Incluir(Produto produto, HttpPostedFileBase
image)
        {
            if (!ModelState.IsValid)
            {
                return Incluir();
            }
            try
            {
                if (image != null)
                {
                    produto.MimeType = image.ContentType;
                    byte[] bytes = new byte[image.ContentLength];
                    image.InputStream.Read(bytes, 0, image.ContentLength);
                    produto.Foto = bytes;
                }
                ProdutosDao.IncluirProduto(produto);
                return RedirectToAction("Listar");
            }
            catch (Exception ex)
            {
```

```
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }

    public FileResult BuscarFoto(int id)
    {
        var foto = ProdutosDao.BuscarProduto(id);
        return File(foto.Foto, foto.MimeType);
    }

    public ActionResult Listar()
    {
        return View(ProdutosDao.ListarProdutos());
    }
}
```

46. Agora vamos gerar a listagem de produtos. Incluiremos apenas o link para alterar o produto. Além disso, na listagem, apresentaremos a foto do produto em tamanho reduzido:



```
@model IEnumerable<Lab.MVC.Models.Produto>

 @{
    ViewBag.Title = "Listar";
}

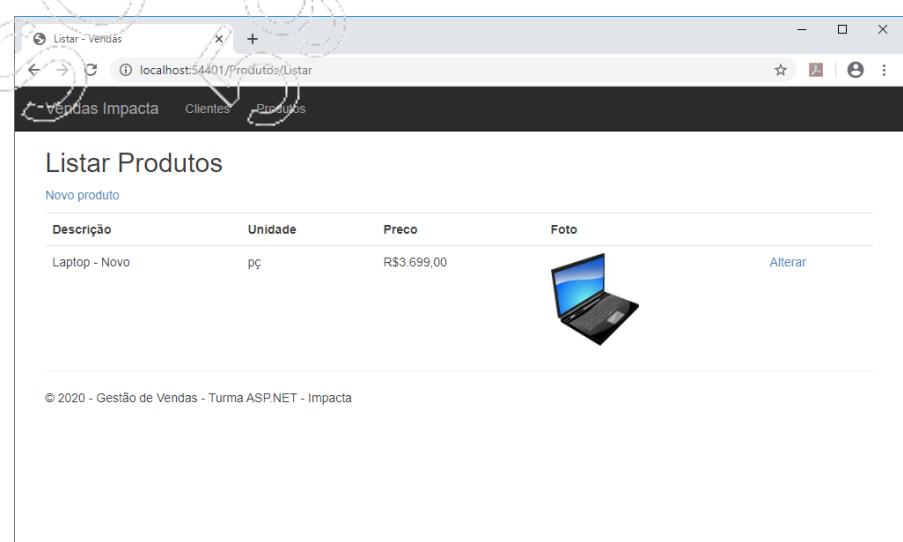
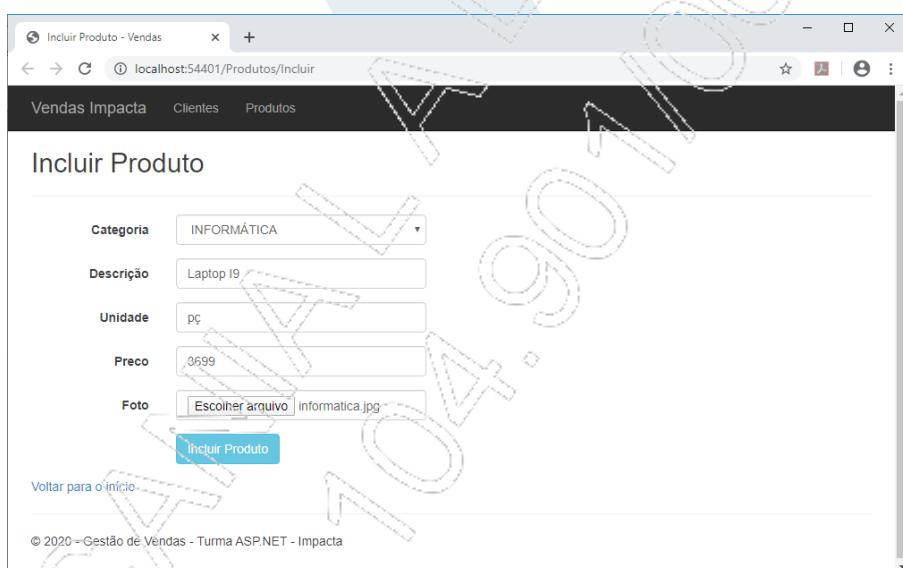
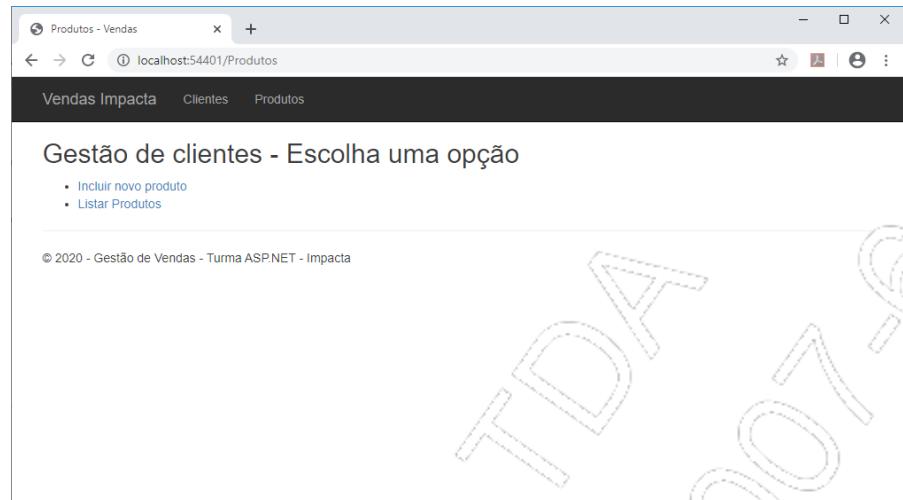
<h2>Listar Produtos</h2>

<p>
    @Html.ActionLink("Novo produto", "Incluir")
</p>


| @Html.DisplayNameFor(model => model.Descricao) | @Html.DisplayNameFor(model => model.Unidade) | @Html.DisplayNameFor(model => model.Preco) | @Html.DisplayNameFor(model => model.Foto)                                                                     |                                                            |
|------------------------------------------------|----------------------------------------------|--------------------------------------------|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
| @Html.DisplayFor(modelItem => item.Descricao)  | @Html.DisplayFor(modelItem => item.Unidade)  | @Html.DisplayFor(modelItem => item.Preco)  |  | @Html.ActionLink("Alterar", "Alterar", new { id=item.Id }) |


```

Com estes dados devidamente definidos, podemos proceder com a inclusão de produtos. Execute a aplicação e selecione a opção para **incluir novo produto**:



47. O procedimento de alteração é semelhante ao de inclusão no que diz respeito ao formulário e à listagem. Porém, um detalhe que deve ser observado é que o usuário pode alterar os dados do produto, mas não alterar a imagem. Neste caso, devemos manter a mesma imagem. Para isso, devemos guardar o valor das propriedades **Foto** e **MimeType** em um campo oculto na view. Se a foto for alterada, estas propriedades serão substituídas pela nova. Vamos então incluir, no controller, o action **Alterar**, nas versões GET e POST:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ProdutosController : Controller
    {
        // GET: Produtos
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir()
        {
            ViewBag.ListaDeCategorias = new SelectList(
                ProdutosDao.ListarCategorias(), "Id", "Descricao");
            return View();
        }

        [HttpPost]
        public ActionResult Incluir(Produto produto, HttpPostedFileBase
image)
        {
            if (!ModelState.IsValid)
            {
                return Incluir();
            }

            try
            {
                if (image != null)
                {
                    produto.MimeType = image.ContentType;
                    byte[] bytes = new byte[image.ContentLength];
                    image.InputStream.Read(bytes, 0, image.ContentLength);
                    produto.Foto = bytes;
                }
            }
        }
    }
}
```

```
        ProdutosDao.IncluirProduto(produto);
        return RedirectToAction("Listar");
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

public FileResult BuscarFoto(int id)
{
    var foto = ProdutosDao.BuscarProduto(id);
    return File(foto.Foto, foto.MimeType);
}

public ActionResult Listar()
{
    return View(ProdutosDao.ListarProdutos());
}

[HttpGet]
public ActionResult Alterar(int? id)
{
    try
    {
        if(id == null)
        {
            throw new Exception("Nenhum código fornecido");
        }

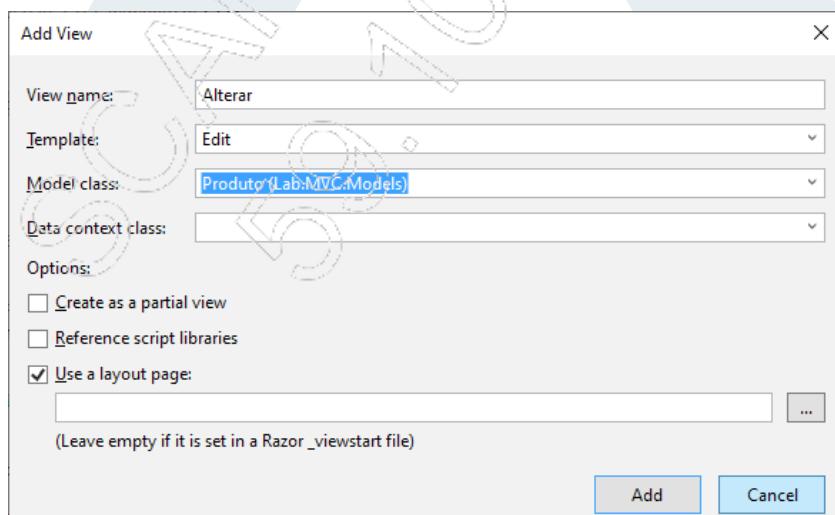
        var produto = ProdutosDao.BuscarProduto((int)id);
        if(produto == null)
        {
            throw new
                Exception("Nenhum produto encontrado com este código");
        }
        ViewBag.ListaDeCategorias = new SelectList(
            ProdutosDao.ListarCategorias(), "Id", "Descricao");
        return View(produto);
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
```

```
[HttpPost]
public ActionResult Alterar(Produto produto, HttpPostedFileBase
image)
{
    if (!ModelState.IsValid)
    {
        return Alterar(produto.Id);
    }

    try
    {
        if (image != null)
        {
            produto.MimeType = image.ContentType;
            byte[] bytes = new byte[image.ContentLength];
            image.InputStream.Read(bytes, 0, image.ContentLength);
            produto.Foto = bytes;
        }
        ProdutosDao.AlterarProduto(produto);
        return RedirectToAction("Listar");
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

}
```

48. Gere a view para a alteração:



```
@model Lab.MVC.Models.Produto

 @{
    ViewBag.Title = "Alterar";
}

<h2>Alterar Produtos</h2>

@using (Html.BeginForm("Alterar", "Produtos", FormMethod.Post,
    new { enctype = "multipart/form-data" }))
{
    @Html.AntiForgeryToken()

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

        <div class="form-group">
            @Html.LabelFor(model => model.IdCategoria, htmlAttributes:
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.DropDownListFor(model => model.IdCategoria,
                    (SelectList)ViewBag.ListaDeCategorias,
                    new { @class = "form-control" })

                @Html.ValidationMessageFor(model => model.IdCategoria, "",
                    new { @class = "text-danger" })
            </div>
        </div>

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

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

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

<div class="form-group">
    @Html.LabelFor(model => model.Foto, htmlAttributes:
        new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        <input type="file" name="Image" class="form-control" />
    </div>
</div>

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

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Alterar Produto"
            class="btn btn-Info" />
    </div>
</div>
</div>
}

<div>
    @Html.ActionLink("Voltar para o início", "Index")
</div>
```

49. Para testar a alteração, selecione o item **Alterar** no menu contido na listagem de produtos. Altere algumas informações sem alterar a foto. Em seguida, altere a foto. O que você pode observar?

D - Cadastro de Pedidos

50. Na sequência, incluiremos os recursos para adicionar novos pedidos a clientes. Criaremos um novo controller, uma nova classe para acesso aos dados, os actions e as views necessárias para esta tarefa. Iniciaremos por adicionar os atributos validadores na entidade **Pedido**:

```
namespace Lab.MVC.Models
{
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;

    public partial class Pedido
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
        "CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Pedido()
        {
            this.Itens = new HashSet<Item>();
        }

        public int Id { get; set; }

        [Required]
        [Display(Name = "Cliente")]
        public string DocCliente { get; set; }

        [Required]
        [Display(Name = "Data do Pedido")]
        [DataType(DataType.Date)]
        public System.DateTime Data { get; set; }

        [Required]
        [Display(Name = "Nº do Pedido")]
        public string NumeroPedido { get; set; }

        public virtual Cliente ClienteInfo { get; set; }
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
        "CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Item> Itens { get; set; }
    }
}
```

51. Crie a classe **PedidosDao** na pasta **Data**, com os métodos para incluir, remover, buscar e listar os pedidos. A listagem pode ser por cliente, ou todos os pedidos (quaisquer métodos necessários ao longo do desenvolvimento, nós incluiremos oportunamente):

```
using Lab.MVC.Models;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace Lab.MVC.Data
{
    public class PedidosDao
    {
        //método para incluir um novo pedido
        public static void IncluirPedido(Pedido pedido)
        {
            using (var ctx = new DbVendasEntities())
            {
                ctx.Pedidos.Add(pedido);
                ctx.SaveChanges();
            }
        }

        //método para listar os pedidos
        public static IEnumerable<Pedido> ListarPedidos(string doc)
        {
            using (var ctx = new DbVendasEntities())
            {
                var lista = ctx.Pedidos.ToList();
                if (!string.IsNullOrEmpty(doc))
                {
                    lista = lista.Where(p => p.DocCliente
                        .Equals(doc)).ToList();
                }
                return lista;
            }
        }

        //método para buscar um pedido
        public static Pedido BuscarPedido(int id)
        {
            using (var ctx = new DbVendasEntities())
            {
                return ctx.Pedidos.FirstOrDefault(p => p.Id == id);
            }
        }

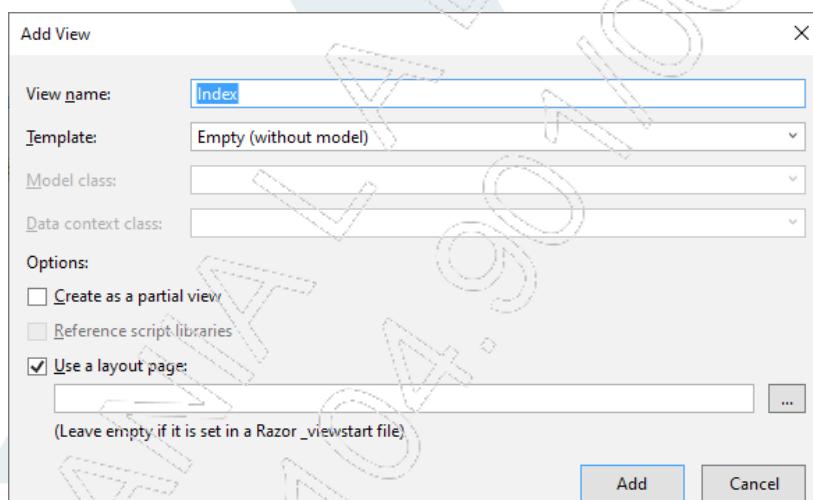
        //método para remover um pedido
        public static void RemoverPedido(Pedido pedido)
        {
            using (var ctx = new DbVendasEntities())
            {
                ctx.Entry<Pedido>(pedido).State = EntityState.Deleted;
                ctx.SaveChanges();
            }
        }
    }
}
```

52. Crie o controller PedidosController:

```
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class PedidosController : Controller
    {
        // GET: Pedidos
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

53. Crie a view para o action Index. Esta view representará a página de gestão de pedidos, assim como aconteceu com clientes e produtos:



```
@{
    ViewBag.Title = "Pedidos";
}



## Gestão de Pedidos - Selecione uma opção



- @Html.ActionLink("Incluir novo pedido", "Incluir")
- @Html.ActionLink("Listar pedidos por cliente", "Listar")

```

54. Adicione o link para este item no layout, logo após o link para Produtos;

55. Em **PedidosController**, adicione o action **Incluir**, versões GET e POST. Na versão GET, devemos levar a lista de clientes para que o usuário o selecione, de forma a criar o pedido para ele. A lista de clientes será enviada através de um objeto **ViewBag**. Escreva também o action **Listar**, que deverá receber opcionalmente o documento do cliente, de forma a apresentar a lista dos pedidos de um determinado cliente:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class PedidosController : Controller
    {
        // GET: Pedidos
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir()
        {
            ViewBag.ListaDeClientes = new SelectList(
                ClientesDao.ListarClientes(), "Documento", "Nome");
            return View();
        }

        [HttpPost]
        public ActionResult Incluir(Pedido pedido)
        {
            pedido.Data = DateTime.Now;

            if (!ModelState.IsValid)
            {
                return Incluir();
            }

            try
            {
                PedidosDao.IncluirPedido(pedido);

                ViewBag.ListaDeClientes = new SelectList(
                    ClientesDao.ListarClientes(), "Documento", "Nome");

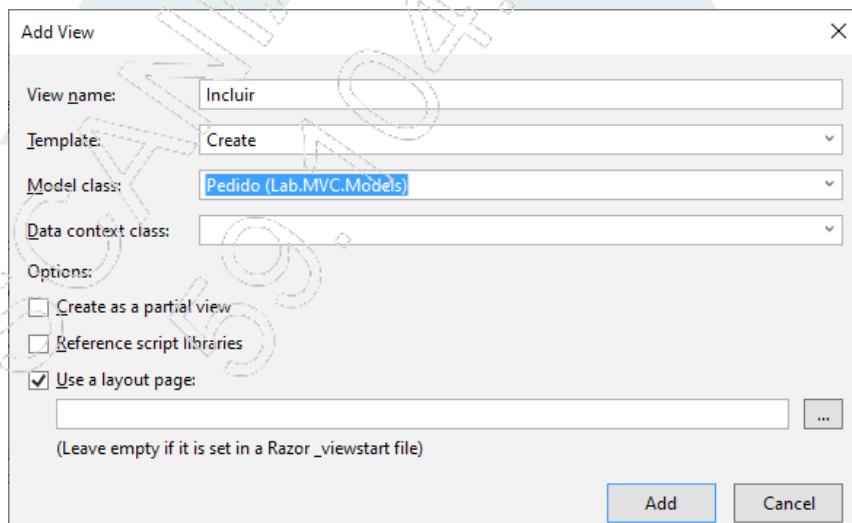
                return Redirect("/Pedidos/Listar?id=" + pedido.DocCliente);
            }
            catch (Exception ex)
            {
```

```
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }

    public ActionResult Listar(string id)
    {
        try
        {
            ViewBag.ListaDeClientes = new SelectList(
                ClientesDao.ListarClientes(), "Documento", "Nome");
            return View(PedidosDao.ListarPedidos(id));
        }
        catch (Exception ex)
        {
            ViewBag.MensagemErro = ex.Message;
            return View("_Erro");
        }
    }
}
```

Observe que o action **Listar** também recebe a lista de clientes, pois ela será usada em um formulário para que o usuário possa selecionar o cliente e apresentar seus pedidos. Após a inclusão do pedido, o usuário é direcionado para a listagem, já com o numero do documento passado como parâmetro;

56. Adicione a view **Incluir**:



```
@model Lab.MVC.Models.Pedido

 @{
    ViewBag.Title = "Incluir";
}

<h2>Incluir Pedido</h2>

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

    <div class="form-horizontal">
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.DocCliente, htmlAttributes:
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.DropDownListFor(
                    model => model.DocCliente,
                    (SelectList)ViewBag.ListaDeClientes,
                    new { @class = "form-control" })

                @Html.ValidationMessageFor(model => model.DocCliente, "",
                    new { @class = "text-danger" })
            </div>
        </div>

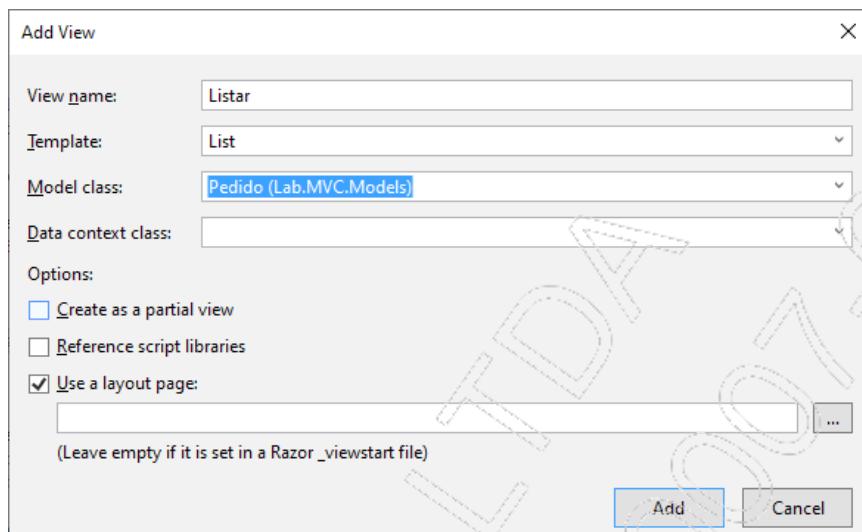
        <div class="form-group">
            @Html.LabelFor(model => model.NumeroPedido, htmlAttributes:
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.NumeroPedido,
                    new { htmlAttributes = new {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

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Incluir Pedido"
                    class="btn btn-info" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Voltar para o menu", "Index")
</div>
```

Omitimos a data do pedido, porque no controller assumimos a data atual;

57. Adicione a view Listar:



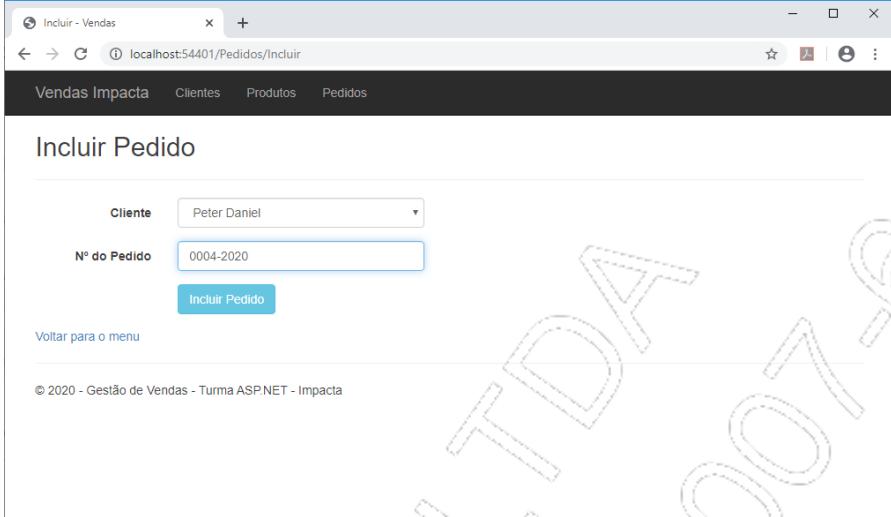
```
@model IEnumerable<Lab.MVC.Models.Pedido>
@{
    ViewBag.Title = "Listar";
}
<p> @Html.ActionLink("Novo pedido", "Incluir")</p>
<h2>Listar Pedidos por Cliente</h2>
@using (Html.BeginForm("Listar", "Pedidos", FormMethod.Get))
{
    <div class="form-horizontal">
        <div class="form-group">
            @Html.Label("Cliente", htmlAttributes:
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.DropDownList("id",
                    (SelectList)ViewBag.ListaDeClientes,
                    "TODOS",
                    new { @class = "form-control" })
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Buscar Pedidos"
                    class="btn btn-info" />
            </div>
        </div>
        <hr/>
    </div>
}
```

```
<table class="table">
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.DocCliente)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Data)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.NumeroPedido)
    </th>
    <th></th>
  </tr>

  @foreach (var item in Model)
  {
    <tr>
      <td>
        @Html.DisplayFor(modelItem => item.DocCliente)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Data)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.NumeroPedido)
      </td>
      <td>
        @Html.ActionLink("Incluir Itens", "Incluir",
          "Itens", new { idPedido = item.Id }, null) |
        @Html.ActionLink("Remover", "Remover",
          new { id = item.Id })
      </td>
    </tr>
  }
</table>
```

Nesta view, nós personalizamos o formulário com os HTML Helpers **Label** e **DropDownList**. Neste caso, os dados dos clientes não são originados no model, mas passados para a view através de um ViewBag. Como nosso objetivo é selecionar o cliente, submeter o formulário e retornar os dados para ele mesmo, usamos o método HTTP GET, que permite montar uma URL com o documento do cliente obtido pela leitura do componente **DropDownList**. O valor obtido é passado para o action indicado através do seu nome, que, no nosso caso, coincide com o dado declarado na rota;

58. Execute a aplicação, ação o cadastro de pedidos, selecione um cliente na lista, informe o número do pedido e adicione;



Incluir Pedido

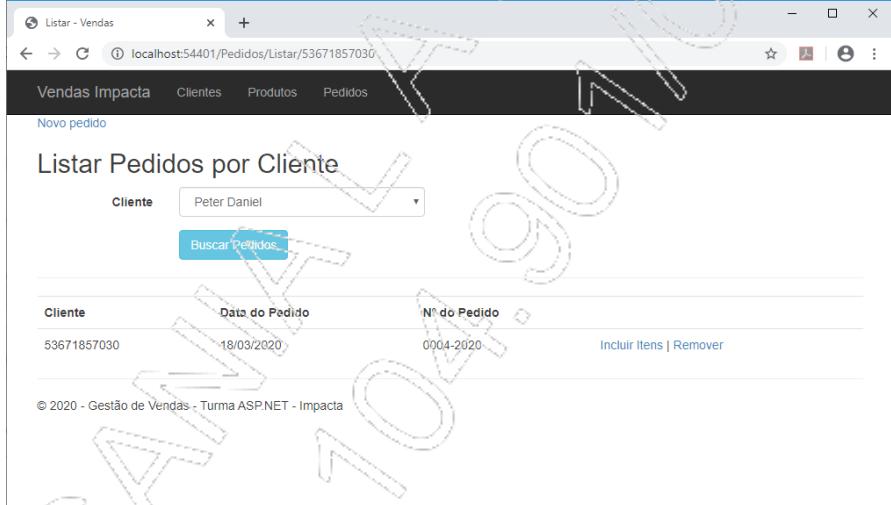
Cliente Peter Daniel

Nº do Pedido 0004-2020

Incluir Pedido

Voltar para o menu

© 2020 - Gestão de Vendas - Turma ASP.NET - Impacta



Listar Pedidos por Cliente

Cliente Peter Daniel

Buscar Pedidos

Cliente	Data do Pedido	Nº do Pedido
53671857030	18/03/2020	0004-2020

Incluir Itens | Remover

© 2020 - Gestão de Vendas - Turma ASP.NET - Impacta

59. Na listagem de pedidos, temos um link para remover o pedido em questão. Vamos implementar o action **Remover** e proceder com esta tarefa:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class PedidosController : Controller
    {
```

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

[HttpGet]
public ActionResult Incluir()
{
    ViewBag.ListaDeClientes = new SelectList(
        ClientesDao.ListarClientes(), "Documento", "Nome");
    return View();
}

[HttpPost]
public ActionResult Incluir(Pedido pedido)
{
    pedido.Data = DateTime.Now;

    if (!ModelState.IsValid)
    {
        return Incluir();
    }

    try
    {
        PedidosDao.IncluirPedido(pedido);

        ViewBag.ListaDeClientes = new SelectList(
            ClientesDao.ListarClientes(), "Documento", "Nome");

        return Redirect("/Pedidos>Listar?id=" + pedido.DocCliente);
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

public ActionResult Listar(string id)
{
    try
    {
        ViewBag.ListaDeClientes = new SelectList(
            ClientesDao.ListarClientes(), "Documento", "Nome");
        return View(PedidosDao.ListarPedidos(id));
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
```

```
public ActionResult Remover(int? id)
{
    try
    {
        if (id == null)
        {
            throw new Exception("Nenhum código fornecido");
        }

        var pedido = PedidosDao.BuscarPedido((int)id);
        if (pedido == null)
        {
            throw new
                Exception("Nenhum pedido encontrado");
        }
        var doc = pedido.DocCliente;

        PedidosDao.RemoverPedido(pedido);
        return Redirect("/Pedidos>Listar?id=" + doc);
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
```

E - Cadastro de Itens

60. Já estamos prontos para elaborar a inclusão de itens a um pedido. Nesta tarefa, o usuário selecionará um pedido, porém o pedido deve vir acompanhado do nome do cliente, proprietário do pedido em questão. Só que não temos nenhuma classe que contenha tanto o nome do cliente quanto o número do pedido. Em casos como este, criamos classes com as propriedades desejadas. Estas classes são chamadas de **ViewModels** e, quando instanciadas, receberão informações provenientes de operações de junções (joins) entre as entidades. Isso nos trará algumas novidades também no uso do Entity Framework, além do LINQ TO SQL!

Para elucidar estas novas inclusões, iniciaremos com a inclusão de validadores e formatadores na entidade **Item**:

```
namespace Lab.MVC.Models
{
    using System.ComponentModel.DataAnnotations;

    public partial class Item
    {
        public int Id { get; set; }
        [Required]
        [Display(Name = "Pedido")]
        public int IdPedido { get; set; }

        [Required]
        [Display(Name = "Produto")]
        public int IdProduto { get; set; }

        [Required]
        [Display(Name = "Quantidade")]
        public double Quantidade { get; set; }

        public virtual Pedido PedidoInfo { get; set; }
        public virtual Produto ProdutoInfo { get; set; }
    }
}
```

61. Para adicionar um novo item, nós deveremos selecionar o pedido (incluindo o nome do cliente), o produto e a quantidade. Com estas informações, adicionaremos o item ao pedido.

Continuando na implementação, crie uma pasta chamada **ViewModels** ao projeto. Nesta pasta, inclua a classe **ClientePedidoViewModel**:

```
namespace Lab.MVC.ViewModels
{
    public class ClientePedidoViewModel
    {
        public string Documento { get; set; }
        public string NomeCliente { get; set; }
        public string NumeroPedido { get; set; }
        public int IdPedido { get; set; }
    }
}
```

62. Para listar os itens de um dado pedido, precisaremos agregar mais informações, como o produto adicionado, em qual pedido, quantos itens e o valor total (só pra citar algumas). Vamos então criar mais uma ViewModel. Esta se chamará **ItensPedidoViewModel**:

```
namespace Lab.MVC.ViewModels
{
    public class ItensPedidoViewModel
    {
        public int IdItem { get; set; }
        public double QuantItens { get; set; }
        public double TotalItem { get; set; }
        public string DescProduto { get; set; }
        public int IdPedido { get; set; }
        public string NumeroPedido { get; set; }
    }
}
```

63. Agora vamos criar a classe **ItensDao** na pasta **Data**. Nesta classe, incluiremos os métodos para listar os pedidos com os nomes dos clientes, e para listar os itens por pedido, conforme descrito. Observe a sintaxe usada nos métodos:

```
using Lab.MVC.Models;
using Lab.MVC.ViewModels;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Web;

namespace Lab.MVC.Data
{
    public class ItensDao
    {
        public static void IncluirItem(Item item)
        {
            using(var ctx = new DbVendasEntities())
            {
                ctx.Entry<Item>(item).State = EntityState.Added;
                ctx.SaveChanges();
            }
        }

        public static void RemoverItem(Item item)
        {
            using (var ctx = new DbVendasEntities())
            {
                ctx.Entry<Item>(item).State = EntityState.Deleted;
                ctx.SaveChanges();
            }
        }
}
```

```
public static Item BuscarItem(int id)
{
    using (var ctx = new DbVendasEntities())
    {
        return ctx.Itens.FirstOrDefault(p => p.Id == id);
    }
}

public static IEnumerable<ClientePedidoViewModel> ListarPedidos()
{
    using(var ctx = new DbVendasEntities())
    {
        var lista = ctx.Clientes.Join(
            ctx.Pedidos,
            c => c.Documento,
            p => p.DocCliente,
            (c, p) => new ClientePedidoViewModel
            {
                Documento = c.Documento,
                NomeCliente = c.Nome + " - " + p.NumeroPedido,
                IdPedido = p.Id,
                NumeroPedido = p.NumeroPedido
            });
        return lista.ToList();
    }
}

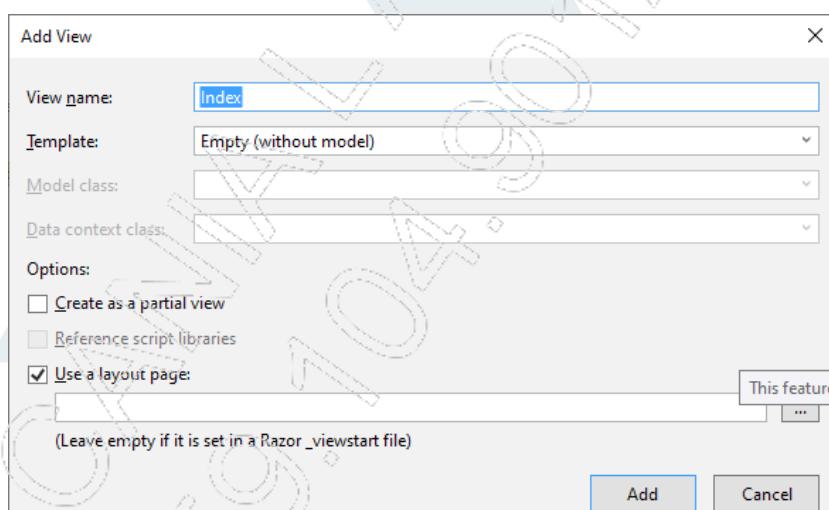
public static IEnumerable<ItensPedidoViewModel> ListarItensPorPedido(
    int? idPedido)
{
    List<ItensPedidoViewModel> lista = new List<ItensPedidoViewModel>();
    if (idPedido != null)
    {
        using (var ctx = new DbVendasEntities())
        {
            lista = (from pedido in ctx.Pedidos
                     join item in ctx.Itens
                     on pedido.Id equals item.IdPedido
                     join produto in ctx.Produtos
                     on item.IdProduto equals produto.Id
                     where pedido.Id == (int)idPedido
                     select new ItensPedidoViewModel
                     {
                         IdItem = item.Id,
                         QuantItens = item.Quantidade,
                         IdPedido = pedido.Id,
                         NumeroPedido = pedido.NumeroPedido,
                         DescProduto = produto.Descricao,
                         TotalItem = item.Quantidade *
                                     (double)produto.Preco
                     }).ToList();
        }
    }
    return lista;
}
```

64. O acesso à inclusão de itens poderá ocorrer através do menu de opções no layout ou através do link na listagem de pedidos. Em ambos os casos, os actions serão os mesmos. Vamos iniciar esta implementação criando o controller **ItensController**:

```
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ItensController : Controller
    {
        // GET: Itens
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

65. Assim como fizemos nos demais controllers neste projeto, vamos adicionar uma view como página inicial do núcleo de inclusão de itens. Para tanto, gere uma view a partir do action **Index**:



```
@{
    ViewBag.Title = "Produtos";
}

<h2>Inclusão de itens aos pedidos - Escolha uma opção</h2>

<ul>
    <li>@Html.ActionLink("Incluir novo item", "Incluir")</li>
</ul>
```

66. Inclua o acesso a este action no menu de opções no Layout, logo após o link par aos pedidos;

67. Adicione o action **Incluir** ao controller **ItensController**. Neste action, enviaremos a lista de produtos, a lista de pedidos obtida pelo método **ListarPedidos** da classe **ItensDao** e a lista de itens por pedido. Incluiremos também a versão POST do action:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ItensController : Controller
    {
        // GET: Itens
        public ActionResult Index()
        {
            return View();
        }

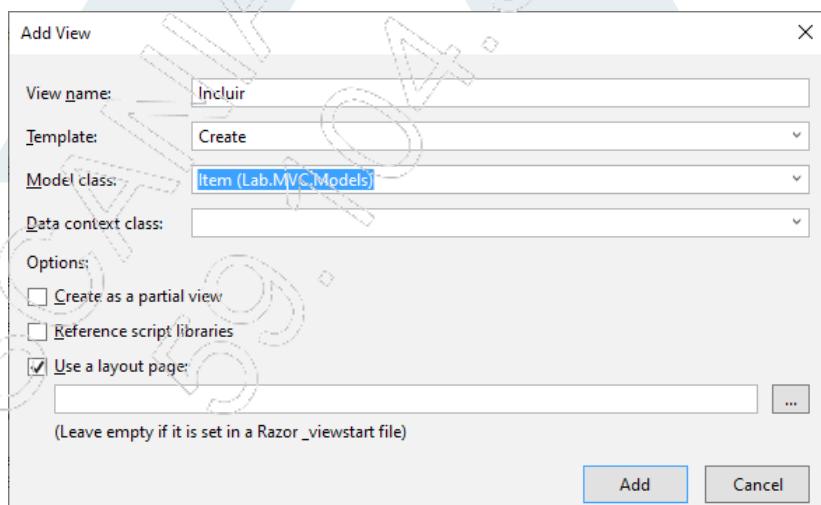
        [HttpGet]
        public ActionResult Incluir(int? idPedido)
        {
            try
            {
                ViewBag.ListaDeProdutos = new SelectList(
                    ProdutosDao.ListarProdutos(), "Id", "Descricao");
                ViewBag.ListaDePedidos = new SelectList(
                    ItensDao.ListarPedidos(), "IdPedido", "NomeCliente");
                ViewBag.ListaDeItens = ItensDao.ListarItensPorPedido(idPedido);

                return View();
            }
            catch (Exception ex)
            {
                ViewBag.MensagemErro = ex.Message;
                return View("_Erro");
            }
        }
    }
}
```

```
[HttpPost]
public ActionResult Incluir(Item item, int? idPedido)
{
    if (!ModelState.IsValid)
    {
        return Incluir(idPedido);
    }
    try
    {
        item.IdPedido = (int)idPedido;
        ItensDao.IncluirItem(item);
        return RedirectToAction("Incluir", new {
            idPedido = (int)idPedido });
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
}
```

A lista de itens (**ViewBag.ListaDeItens**) será apresentada ao lado do formulário de inclusão de itens. Ela representará a lista de itens relativa ao pedido selecionado. Vamos ver como ficará nossa view;

68. Adicione a view para o action **Incluir**:



```
@model Lab.MVC.Models.Item
@using Lab.MVC.ViewModels;

 @{
    ViewBag.Title = "Itens";
}

<h2>Inclusão de Itens</h2>

<div class="row">
    <div class="col-md-6">
        <h3>Cadastro</h3>
        @using (Html.BeginForm())
        {
            @Html.AntiForgeryToken()

            <div class="form-horizontal">
                <hr />
                @Html.ValidationSummary(true, "", new { @class = "text-danger" })
                <div class="form-group">
                    @Html.LabelFor(model => model.IdPedido, htmlAttributes:
                        new { @class = "control-label col-md-2" })
                    <div class="col-md-10">
                        @Html.DropDownList("idPedido",
                            (SelectList)ViewBag.ListaDePedidos,
                            "SELEÇÃO",
                            new { @class = "form-control" })
                    </div>
                </div>

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

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

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Adicionar Item"
               class="btn btn-info" />
    </div>
</div>
<div class="col-md-6">
    <h3>Lista de Itens do Pedido</h3>
    @{
        List<ItensPedidoViewModel> lista = ViewBag.ListaDeItens;
        if (lista == null || lista.Count == 0)
        {
            <div class="alert alert-danger">
                Nenhum item ou nenhum pedido selecionado
            </div>
        }
        else
        {
            foreach (var item in lista)
            {
                <div class="alert alert-info">
                    <div style="float:right">
                        @Html.ActionLink("x", "Remover", "Itens", new
                        {
                            id = item.IdItem
                        },
                        new { role = "button",
                              style = "text-decoration:none;" })
                    </div>
                    <strong>Pedido:</strong>
                    @item.NumeroPedido
                    &nbsp;&nbsp; <strong>Produto:</strong>
                    @item.DescProduto
                    <br />
                    <strong>Quant. Itens:</strong>
                    @item.QuantItens
                    &nbsp;&nbsp; <strong>Valor:</strong>
                    @item.TotalItem.ToString("c")
                </div>
            }
        }
    </div>
    </div>
    @section Scripts{
        <script type="text/javascript">
            $(document).ready(function () {
                $("#idPedido").change(function () {
                    var selecao = $(this).val();
                    $(location).attr("href", "/Itens/Incluir?idPedido=" + selecao);
                });
            });
        </script>
    }
}
```

Vamos entender como esta view foi elaborada:

- Além do **@model**, usamos a diretiva **@import** para referenciar o namespace onde a classe **ItensPedidoViewModel** foi definida;
- Alteramos a aparência da view, dividindo seu conteúdo em duas colunas. Para isso, usamos a classe **row** do **bootstrap** em uma **div**, com dois outros elementos **div** dentro dela. O objetivo foi definir o espaço em duas colunas, cada uma ocupando seis espaços (**classe col-md-6**);
- Para a lista de pedidos, colocamos o helper **Html.DropDownList** em vez do **Html.DropDownListFor**. Isso foi feito para viabilizar a passagem do id do pedido como parâmetro do action na URL, já que o action **Incluir** está sendo chamado em diferentes locais. Quando o id do pedido é informado como parâmetro, o elemento **Html.DropDownList** apresenta o pedido selecionado, facilitando a usabilidade;
- A lista de itens, representada pelo método **ListarItensPorPedido** da classe **ItensDao** apresenta os itens do pedido selecionado. Tanto no cadastro como na consulta, apenas os itens daquele pedido são mostrados para o usuário;
- Finalmente, escrevemos uma função **Javascript** com a sintaxe **JQuery**, que permite executar a URL com o pedido selecionado cada vez que o usuário seleciona um pedido na lista. Como o formulário é enviado para o controller via POST para inclusão do item, optamos por acrescentar este recurso. Além de mais eficiente, enriquecemos a experiência do usuário. Observe que definimos um **section** para o javascript.

69. Já que foi definido um bloco **section** com o nome **Scripts**, devemos adicioná-lo no layout da aplicação. Escreva a seguinte instrução no final do layout (colocamos aqui apenas o trecho necessário):

```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Gestão de Vendas -
            Turma ASP.NET - Impacta</p>
    </footer>
</div>

<script src="~/Scripts/jquery-3.3.1.min.js"></script>
<script src="~/Scripts/bootstrap.min.js"></script>

@RenderSection("Scripts", required: false)
```

70. Para finalizar, vamos adicionar o action que, com base no id do item, permita sua remoção. O action deve se chamar **Remover**, de acordo com a instrução colocada na view **Incluir**:

```
foreach (var item in lista)
{
    <div class="alert alert-info">
        <div style="float:right">
            @Html.ActionLink("x", "Remover", "Itens", new
            {
                id = item.IdItem
            },
            new { role = "button",
                  style = "text-decoration:none;" })
        </div>
        <strong>Pedido: </strong>
            @item.NumeroPedido
        &nbsp;&nbsp; <strong>Produto: </strong>
            @item.DescProduto
        <br />
        <strong>Quant. Itens:</strong>
            @item.QuantItens
        &nbsp;&nbsp; <strong>Valor: </strong>
            @item.TotalItem.ToString("c")
    </div>
}
```

Conteúdo do action **Remover** no controller **ItensController**:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class ItensController : Controller
    {
        // GET: Itens
        public ActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public ActionResult Incluir(int? idPedido)
        {
            try
            {
```

```
ViewBag.ListaDeProdutos = new SelectList(  
    ProdutosDao.ListarProdutos(), "Id", "Descricao");  
ViewBag.ListaDePedidos = new SelectList(  
    ItensDao.ListarPedidos(), "IdPedido", "NomeCliente");  
ViewBag.ListaDeItens = ItensDao.ListarItensPorPedido(idPedido);  
  
        return View();  
    }  
    catch (Exception ex)  
    {  
        ViewBag.MensagemErro = ex.Message;  
        return View("_Erro");  
    }  
}  
  
[HttpPost]  
public ActionResult Incluir(Item item, int? idPedido)  
{  
    if (!ModelState.IsValid)  
    {  
        return Incluir(idPedido);  
    }  
    try  
    {  
        item.IdPedido = (int)idPedido;  
        ItensDao.IncluirItem(item);  
        return RedirectToAction("Incluir", new {  
            idPedido = (int)idPedido });  
    }  
    catch (Exception ex)  
    {  
        ViewBag.MensagemErro = ex.Message;  
        return View("_Erro");  
    }  
}
```

```
public ActionResult Remover(int? id)
{
    try
    {
        if(id == null)
        {
            throw new Exception("Código não informado");
        }
        var item = ItensDao.BuscarItem((int)id);
        if (item == null)
        {
            throw new Exception("Item não encontrado");
        }
        int idPedido = item.IdPedido;
        ItensDao.RemoverItem(item);

        return RedirectToAction("Incluir", new
        {
            idPedido
        });
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
```

71. Podemos chamar esta view de diversas formas:

- A partir da lista de pedidos;
- A partir do menu no layout;
- Após incluir um novo item;
- Após a remoção do item.

Execute a aplicação, aplicando todos estes processos.

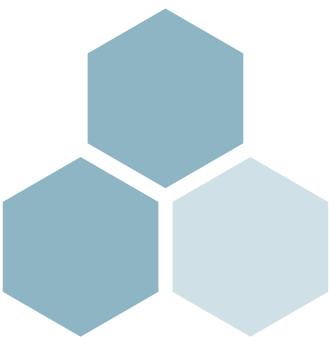
Neste capítulo, pudemos aprender e aplicar uma variedade de opções no desenvolvimento do projeto. Ainda assim, ele poderia ter muitos outros recursos. Sinta-se à vontade para alterar as views ou mesmo o processo usado, como forma de treinar as técnicas apresentadas.



5

Segurança – ASP.NET Identity

- ASP.NET Identity;
- OWIN;
- Implementações do Visual Studio.



5.1. Introdução

O ASP.NET Identity é uma parte do framework cuja finalidade é fornecer recursos de segurança para a aplicação. Esses recursos consistem em: criar novos usuários, criar regras de utilização para os usuários, realizar login, entre muitas outras coisas.

Diferente de outros modelos que existiam antes do advento do ASP.NET Identity, como o Membership Provider, por exemplo, ele oferece uma camada de abstração que permite sua utilização não só com diferentes bancos de dados mas também com mecanismos de autenticação externos à aplicação, como o Google e o Facebook, por exemplo. Os componentes responsáveis por essa conversão são **OWIN** e **OAuth**.

5.2. ASP.NET Identity

O ASP.NET Identity permite o uso de qualquer modelo de dados ou provedor de autenticação (como Google, Facebook, Twitter ou Windows Live). O esquema de dados é totalmente controlado pela aplicação e todo o processo é modular, permitindo a substituição ou modificação de qualquer componente do processo. Esse componente se tornou válido na versão 4.6 no .NET Framework.

A modelagem do ASP.NET Identity utiliza interfaces para usuário, gerenciamento de usuário, grupos de usuários, senhas e todo tipo de informação que seja necessária à implementação de um sistema de segurança.

A principal interface é a que define um usuário, chamada **IUser**:

```
public interface IUser<out TKey>
{
    TKey Id { get; }
    string UserName { get; set; }
}

public interface IUser : IUser<string>
{ }
```

A interface **IUser** define apenas um **Id** e **Nome de Usuário**. O **Id** pode ser de qualquer tipo e uma sobrecarga é implementada, por padrão, definindo o tipo da chave como **string**. Repare que não há nenhuma outra implementação como **Senha**, **Nome Completo**, ou **Data de Cadastro**. O objetivo é exatamente este: manter o mais simples possível. Outras informações podem ser incluídas por meio de diferentes interfaces disponíveis. Uma implementação concreta dessa interface deve ser, no mínimo, conforme a listagem a seguir:

```
public class Usuario:IUser
{
    public string Id { get; set; }
    public string UserName { get; set; }
}
```

Nada impede, porém, que acrescentemos outras informações na classe básica:

```
public class Usuario:IUser
{
    public string Id { get; set; }
    public string UserName { get; set; }
    public string Cargo { get; set; }
    public DateTime DataNascimento { get; set; }
}
```

Para gravar as informações de um usuário, é necessário criar uma classe que implemente a interface **IUserStore**:

```
public interface IUserStore<TUser, in TKey> :
    IDisposable where TUser : class,
    IUser<TKey>
{
    Task CreateAsync(TUser user);
    Task DeleteAsync(TUser user);
    Task<TUser> FindByIdAsync(TKey userId);
    Task<TUser> FindByNameAsync(string userName);
    Task UpdateAsync(TUser user);
}
```

A última classe necessária para gerenciar os usuários se chama **UserManager** e está no namespace **Microsoft.AspNet.Identity**, o mesmo das interfaces **IUser** e **IUserStore**. Para instanciar, é necessário passar a classe de usuário, a chave e a classe que implementa a interface **IUserStore**. A listagem a seguir mostra apenas a declaração da classe, o método construtor e o método **Create**, que será usado neste exemplo:

```
public class UserManager<TUser, TKey> : IDisposable
    where TUser : class, IUser<TKey>
    where TKey : IEquatable<TKey>
{
    public UserManager(IUserStore<TUser, TKey> store);
    public virtual Task<IdentityResult> CreateAsync(TUser user);
    ... outros métodos omitidos
}
```

A classe **UserManager** contém vários métodos de extensão fornecidos pela biblioteca básica do ASP.NET Identity. Um dos métodos se chama **Create** e é utilizado para criar um novo usuário:

```
public static class UserManagerExtensions
{
    public static IdentityResult Create<TUser, TKey>(
        this UserManager<TUser, TKey> manager, TUser user)
        where TUser : class, IUser<TKey>
        where TKey : IEquatable<TKey>

    public static IdentityResult Create<TUser, TKey>
        (this UserManager<TUser, TKey> manager, TUser user)
        where TUser : class, IUser<TKey>
        where TKey : IEquatable<TKey>

    ...
    ... outros métodos omitidos
}
```

O exemplo a seguir inclui um usuário:

```
1. var usuarioStore = new UsuarioStore();
2. var gerenciador = new UserManager<Usuario>(usuarioStore);
3. var usuario = new Usuario() { UserName = "Maria", Id="1" };
4. var resposta=gerenciador.Create(usuario);
5. if (resposta.Succeeded)
{
    //OK
}
```

Vejamos a explicação do código:

1. Uma instância da classe **UsuarioStore** é criada. Essa classe implementa a interface **IUserStore** e é necessária porque deve ser informada quando a classe **UserManager** for criada. Apenas para recapitular, a classe **UsuarioStore** executa as ações no banco de dados ou meio de armazenamento definido;
2. A classe **UserManager** é instanciada. Nessa classe, deve ser passado o parâmetro informando o tipo do usuário (aquele que implementa a interface **IUser**) e a classe **xxxStore**, que implementa a interface **IUserStore**. A classe **UserManager** chama os métodos para gerenciar usuários. Neste exemplo, apenas a parte de criação de dados foi implementada. Outras interfaces podem ser implementadas (**Senhas**, **Validações**, **Autenticadores**) e tudo fica reunido nessa classe;

3. Uma instância de **Usuário** com os dados do usuário a ser cadastrado é criada. Neste exemplo, para manter a simplicidade, os dados foram colocados manualmente;

4. Esta é a parte principal. O método **Create** é chamado, disparando os métodos apropriados na classe **xxxStore** e gravando o usuário. A classe **UserManager** chama o método **FindByName** e, se o nome do usuário for encontrado com um Id diferente do informado, a resposta retorna negativa. Caso contrário, o método **CreateAsync** é chamado para a criação do usuário;

5. A resposta do método **Create** da classe **UserManager** é uma instância da classe **IdentityResult**. Essa classe tem uma propriedade chamada **Succeeded**, do tipo booleana, para informar se a criação foi bem sucedida ou não. Caso tenha dado algum erro, a coleção de strings **Erros** retorna uma lista de erros.

A seguir, vejamos a lista de classes e interfaces utilizadas até o momento, que estão no namespace **Microsoft.AspNet.Identity**:

Nome	Tipo	Função
IUser	Interface	Define Id e nome de usuário. Opcionalmente, podem ser definidos outros dados.
IUserStore	Interface	Define os métodos para incluir, alterar, excluir e pesquisar nos dados do usuário.
UserManager	Classe	Chama os métodos das classes que implementam as funcionalidades do sistema.
IdentityResult	Classe	Contém a resposta às chamadas.

Adiante está a lista de classes criadas pelo aplicativo:

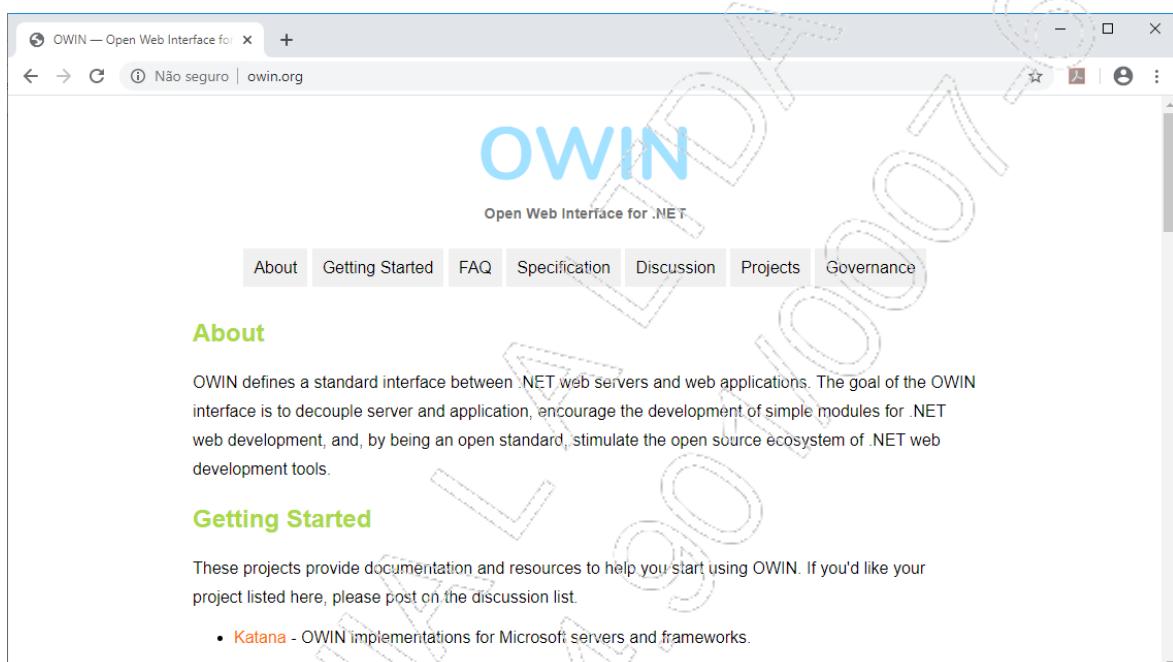
Nome	Tipo	Função
Usuario	Classe	Implementação da interface IUser .
UsuarioStore	Classe	Implementação da interface IUserStore .

Todo o processo apresentado aqui implementa o cadastro de usuário manualmente. O ASP.NET contém uma implementação completa usando o Entity Framework e gravando o usuário em uma tabela do SQL Server, porém, essa implementação do Entity Framework é mais complexa e utiliza outros pacotes de funcionalidades. Para manter o foco nas classes principais do ASP.NET Identity, esta primeira parte será construída com o mínimo de componentes externos, mantendo, dentro do possível, apenas a biblioteca principal do ASP.NET Identity.

5.3. OWIN

Para realizar o processo de login, o ASP.NET utiliza componentes compatíveis com uma especificação chamada **OWIN**. Essa especificação define regras para que qualquer programa, em qualquer sistema operacional, possa hospedar aplicações do .NET Framework. Isso permite criar servidores mais leves do que o IIS e, também, utilizar autenticação de terceiros, como Facebook ou Google.

A sigla **OWIN** significa **Open Web Interface for .NET**.



A ideia é ampliar e padronizar os recursos de integração de componentes criados com a plataforma .NET.

A especificação OWIN em si foge ao propósito deste capítulo. No entanto, é importante conhecer os componentes OWIN disponíveis para autenticação e como o ASP.NET Identity utiliza os recursos de autenticação definidos nessa especificação.

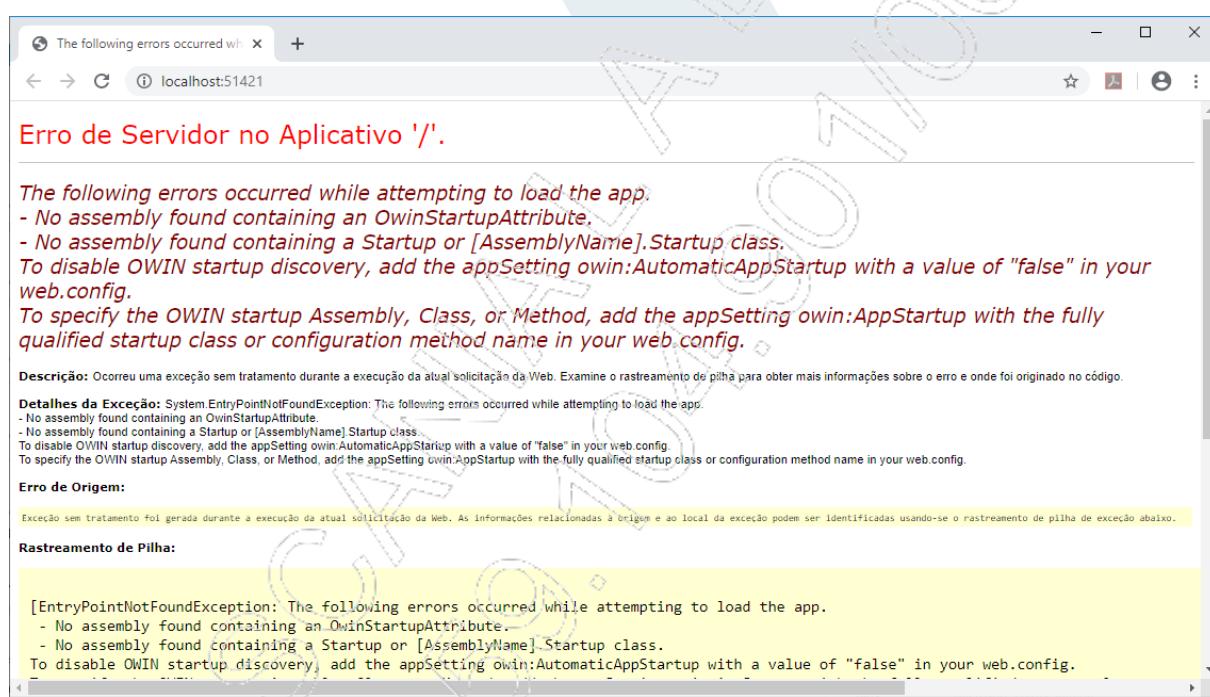
Consideremos um novo projeto MVC vazio chamado **Capítulo05.AspNetIdentity**. Por meio do NuGet, dois componentes devem ser instalados na aplicação:

- **Microsoft.AspNet.Identity.Owin**: Contém diversos métodos de extensão para que os componentes padrão do ASP.NET Identity utilizem o padrão OWIN;
- **Microsoft.Owin.Host.System.Web**: Permite que o IIS possa ser usado para rodar aplicações OWIN.

Esses pacotes dependem de outros pacotes de componentes, portanto, o número de itens adicionados à solução é maior:

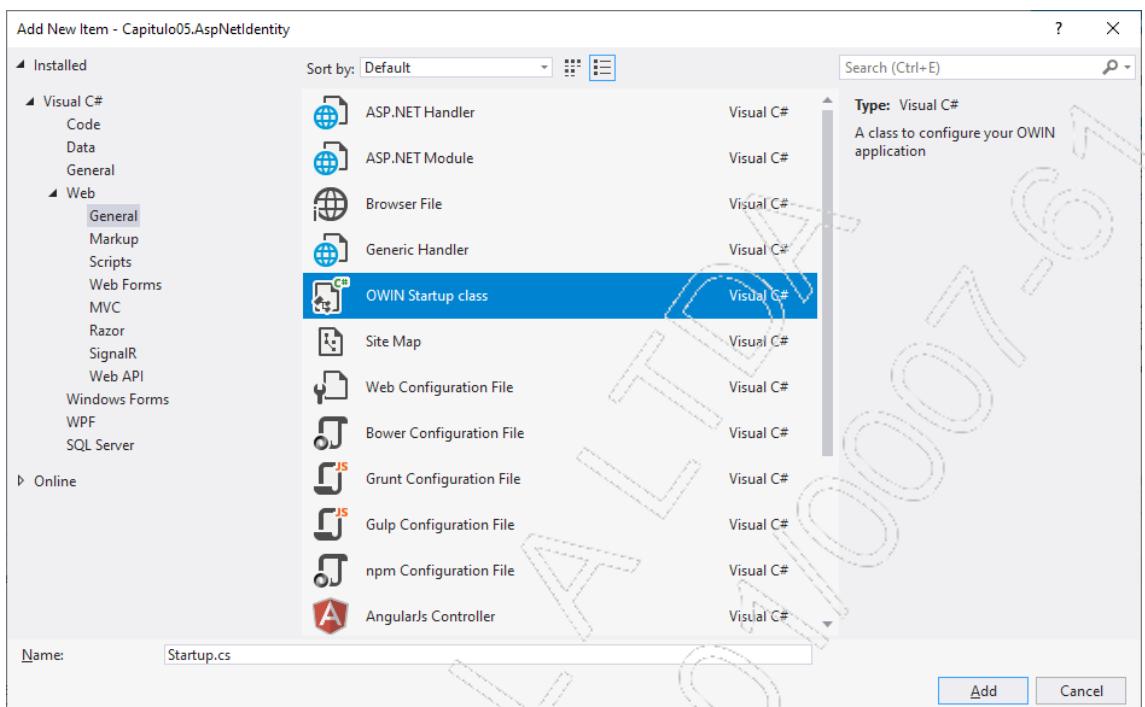
- **Microsoft.AspNet.Identity.Owin;**
- **Microsoft.Owin;**
- **Microsoft.Owin.Host.SystemWeb;**
- **Microsoft.Owin.Security;**
- **Microsoft.Owin.Security.Cookies;**
- **Microsoft.Owin.Security.Oauth;**
- **Newtonsoft.Json;**
- **Owin.**

Mesmo antes de adicionar controllers ao projeto, vamos executá-lo. Podemos notar que ocorre o seguinte erro:



O OWIN necessita de um módulo para seu início. Devemos lembrar que o OWIN é uma especificação, e que possui sua própria configuração.

Portanto, para definir a autenticação OWIN é necessário incluir uma classe declarada com o atributo **OwinStartup**. O nome mais comum para essa classe é **Startup.cs**. No projeto, clicando com o botão direito do mouse, podemos selecionar o item **OWIN Startup Class**, no template **General**:



A classe gerada possui o seguinte código inicial:

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(Capitulo05.AspNetIdentity.Startup))]

namespace Capitulo05.AspNetIdentity
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
        }
    }
}
```

O OWIN utiliza o método **Configuration**, através do parâmetro **IAppBuilder**, para fornecer o objeto por intermédio de injeção de dependência. Nosso trabalho é codificar este método.

Sendo assim, consideremos o seguinte código:

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;

[assembly: OwinStartup(typeof(Capitulo05.AspNetIdentity.Startup))] 1

namespace Capitulo05.AspNetIdentity
{
    public class Startup 2
    {
        public void Configuration(IAppBuilder app) 3
        {
            var opcoes = new CookieAuthenticationOptions();

            opcoes.AuthenticationType =
                DefaultAuthenticationTypes.ApplicationCookie;
            opcoes.LoginPath = new PathString("/login"); 4

            app.UseCookieAuthentication(opcoes); 5
        }
    }
}
```

- 1 – O atributo **OwinStartup** recebe um parâmetro com o nome da classe (**namespace.nome**), contendo o método **Configuration**, que recebe um parâmetro do tipo **IAppBuilder**. Isso faz o mecanismo OWIN instanciar a classe e chamar o método no início da aplicação. Esse método define as configurações iniciais do aplicativo. Isso é uma **convenção**. Se não houver esse método, ocorre o erro apresentado anteriormente;
- 2 – A classe **Startup**, chamada no item 1;
- 3 – O método exigido pelo OWIN. Ele deve se chamar **Configuration** e deve receber como parâmetro uma instância de uma classe que implemente a interface **IappBuilder**;
- 4 – O método principal desse processo se chama **UseCookieAuthentication**. Esse método espera receber uma instância da classe **CookieAuthenticationOptions**. Essa parte do código cria esta instância e define duas opções: o tipo de autenticação (**Cookies**) e a página de URL de Login;
- 5 – Finalmente, a instância passada como parâmetro (**meuApp**) é usada para definir as opções de autenticação, por meio do método **UseCookieAuthentication**.

Está completo o processo. O código que cria o usuário pode ser usado para autenticar esse usuário no sistema:

```
var usuarioStore = new UsuarioStore();
    var gerenciador = new UserManager<Usuario>(usuarioStore);
var usuario = new Usuario() { UserName = "Maria", Id = Guid.NewGuid().
ToString() };
var resposta = gerenciador.Create(usuario);

if (resposta.Succeeded)
{
    var authenticationManager = HttpContext.Current.GetOwinContext().
Authentication;
    var tipoAutenticacao=DefaultAuthenticationTypes.ApplicationCookie;
    var userIdentity = gerenciador.CreateIdentity(usuario,
    tipoAutenticacao);
    authenticationManager.SignIn(userIdentity);

    //OK
}
```

O método de extensão **GetOwinContext** retorna informações sobre a execução atual. Uma das informações é a propriedade **Authentication**, um gerenciador de autenticação, que é uma instância de uma classe que implementa a interface **IAuthenticationManager**. Essa interface define o método **SignIn**, usado para autenticar o usuário.

As seguintes classes e interfaces foram usadas nesse processo de autenticação por cookies, usando componentes compatíveis com a especificação OWIN:

Namespace Owin		
Nome	Tipo	Função
IAppBuilder	Interface	Interface que deve ser passada (por meio de uma instância que a implemente) para o programa de inicialização de um site.

Namespace Microsoft.Owin		
Nome	Tipo	Função
OwinStartup	Classe	Atributo para definir uma classe de inicialização.
PathString	Struct	Manipula informações de uma URL.
IOwinContext	Interface	Permite extrair dados de ambiente OWIN usando tipos definidos (strong-typed accessors).

Namespace Microsoft.Owin.Security		
Nome	Tipo	Função
IAuthenticationManager	Interface	Usado para interagir como componentes OWIN.

Namespace Microsoft.Owin.Security.Cookies		
Nome	Tipo	Função
CookieAuthenticationOptions	Classe	Opções de autenticação por cookies.

Namespace Microsoft.AspNet.Identity		
Nome	Tipo	Função
DefaultAuthenticationTypes	Classe	Tipos de autenticação.

5.4. Implementações do Visual Studio

O Visual Studio oferece uma série de implementações de segurança usando toda a funcionalidade das classes do ASP.NET Identity. Para armazenar os dados de usuário em um banco de dados, é possível usar o Entity Framework no modelo Code First, o que facilita bastante a personalização. Pode-se usar um provedor de autenticação externo, como Facebook ou Google.

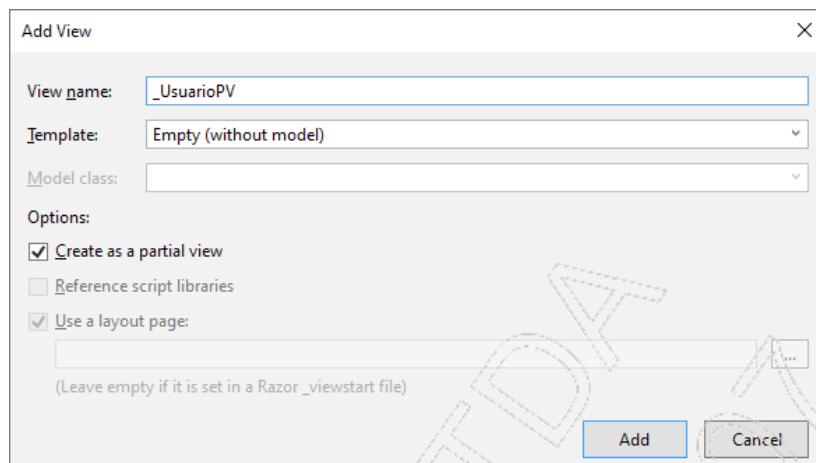
Vamos usar o projeto criado neste capítulo, **Capítulo05.AspNetIdentity**, para explanar o processo de criação de contas e autenticação.

5.4.1. Incluindo um menu de acesso

Vamos iniciar o processo criando um menu a ser adicionado no layout do projeto. Este menu será elaborado através de uma View Parcial, pois sua visualização será dinâmica, ou seja, quando o usuário não estiver logado, ele apresentará as opções **Login** e **Cadastre-se**. Estando logado, o mesmo componente apresentará as opções **Bem vindo**, **usuário** ou **Logout**.

Para iniciar, vamos incluir um controller chamado **Home** (classe **HomeController**). Mantenha o action **Index**, e gere uma view sem modelo, mas com layout. Não precisamos nos preocupar com o conteúdo desta view no momento. Em seguida, execute a aplicação para que o layout e todos os componentes da camada de visualização sejam incluídos pelo Visual Studio no projeto.

Vamos agora criar, na pasta **Views/Shared**, uma view chamada **_UsuarioPV.cshtml**. Ela deve ser criada como uma view parcial.



Seu conteúdo deverá ser análogo ao mostrado adiante:

```
@if (Request.IsAuthenticated)
{
    <ul class="nav navbar-nav navbar-right">
        <li>@Html.ActionLink(User.Identity.Name, "Index", "Home")</li>
        <li>@Html.ActionLink("Logout", "Logout", "Home")</li>
    </ul>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li>@Html.ActionLink("Cadastrar-se", "Criar", "Home")</li>
        <li>@Html.ActionLink("Login", "Login", "Home")</li>
    </ul>
}
```

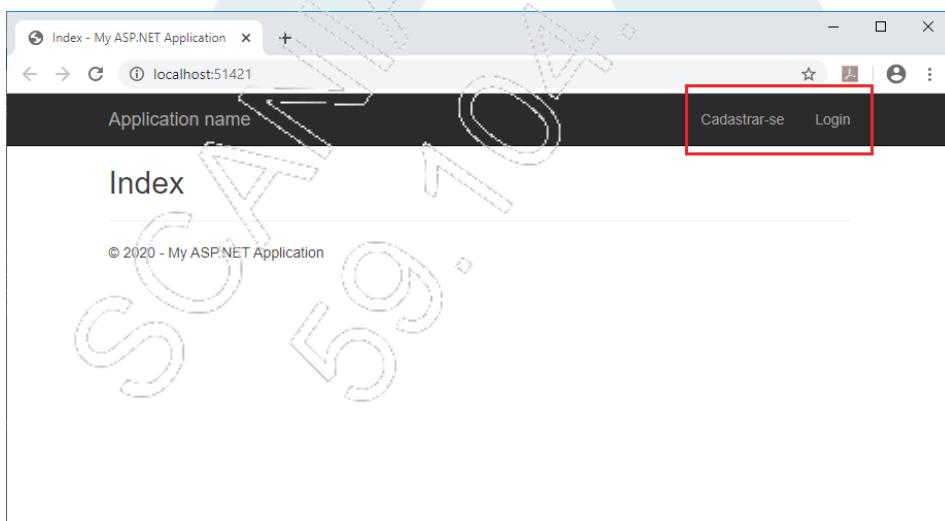
Agora vamos adicionar esta view parcial ao layout, na parte do menu. Sua posição será do lado direito, em relação ao restante do menu. Localize a parte referente ao menu e inclua a instrução:

```
<div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle"
                data-toggle="collapse" data-target=".navbar-collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            @Html.ActionLink("Application name", "Index",
                "Home", new { area = "" }, new { @class = "navbar-brand" })
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
            </ul>

            @Html.RenderPartial("_UsuarioPV");

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

Execute a aplicação e visualize o menu.



5.4.2. Criando os models e os actions

Vamos definir os models para login e para criação de um novo usuário. Para criar um novo usuário, usaremos as propriedades: **Nome**, **Senha** e **Confirma**. No caso do login, usaremos as mesmas propriedades, menos **Confirma**. Adicionaremos também os validadores necessários. As classes se chamarão **LoginView** e **UsuarioView**, e ficarão na pasta **Models**:

- Classe **LoginView**

```
namespace Capitulo05.AspNetIdentity.Models
{
    public class LoginView
    {
        [Required]
        public string Nome { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }
    }
}
```

- Classe **UsuarioView**

```
namespace Capitulo05.AspNetIdentity.Models
{
    public class UsuarioView
    {
        [Required]
        public string Nome { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }

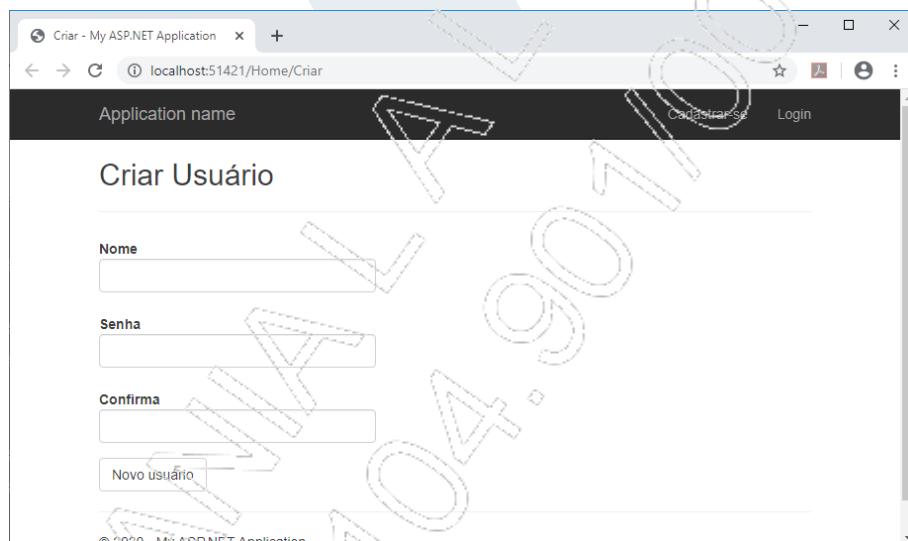
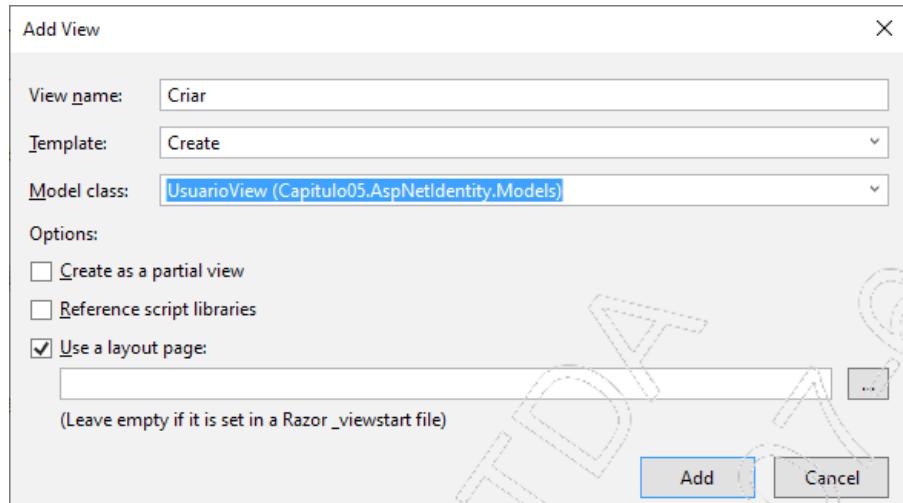
        [Compare("Senha")]
        [DataType(DataType.Password)]
        public string Confirma { get; set; }
    }
}
```

No controller **Home**, definiremos os seguintes actions: **Login**, **Logout** e **Criar**. A view **Login.cshtml** usará como modelo a classe **LoginView**, e a view **Criar.cshtml** usará a classe **UsuarioView**. O action **Logout** não terá view.

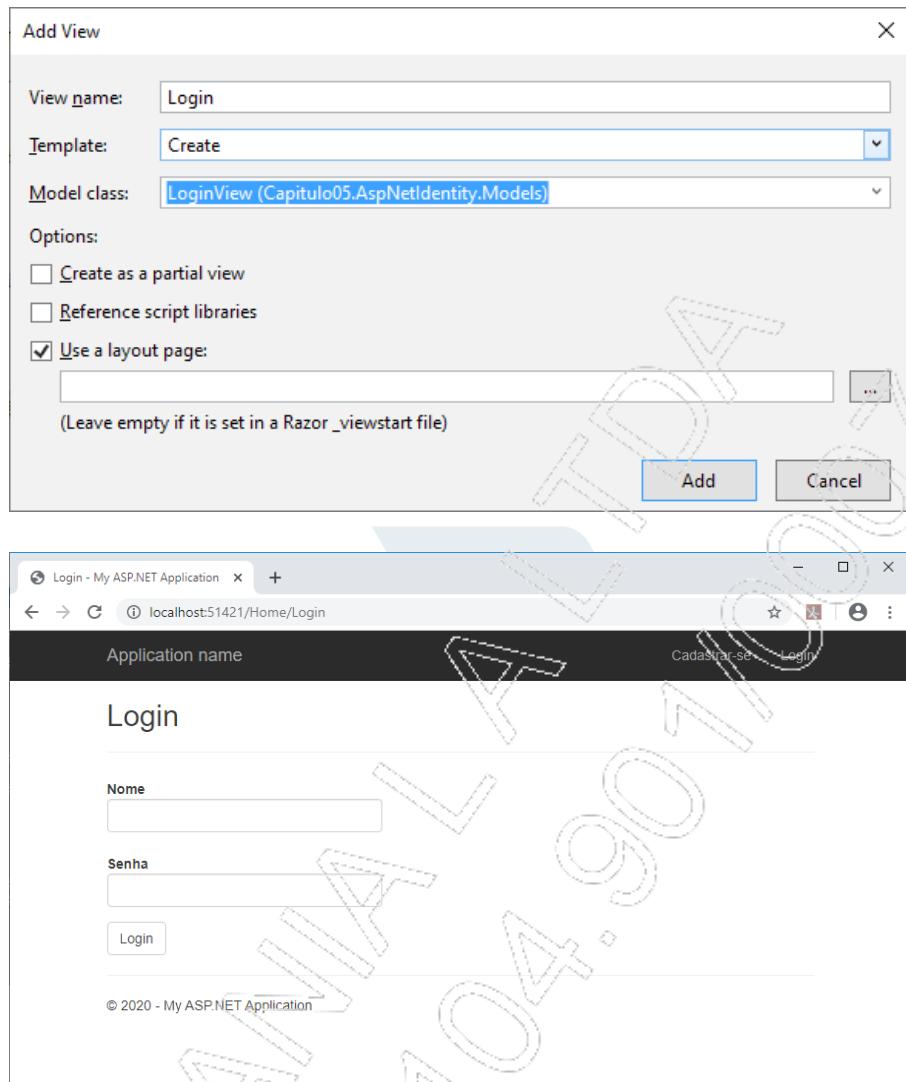
- Definição dos actions no controller:

```
{  
    public class HomeController : Controller  
    {  
        // GET: Home  
        public ActionResult Index()  
        {  
            return View();  
        }  
  
        public ActionResult Criar()  
        {  
            return View();  
        }  
  
        public ActionResult Login()  
        {  
            return View();  
        }  
  
        public ActionResult Logout()  
        {  
            return View();  
        }  
    }  
}
```

- Criação da view **Criar.cshtml**:



- Criação da view **Login.cshtml**:



Faça as adaptações nas views geradas, conforme sua necessidade.

5.4.3. Escrevendo o código para criar um novo usuário

As referências a seguir devem ser incluídas via NuGet:

- **Asp.Net Identity Entity**
- **Asp.Net Identity Owin**
- **Owin Host**

Se já foram incluídas nas etapas anteriores, pode-se mantê-las.

No arquivo **Startup.cs**, configuraremos o modo de autenticação e o caminho do Login, que será processado pelo ASP.NET Identity, caso o usuário tente acessar algum recurso protegido (que requer um usuário autenticado). Mantenha o seguinte conteúdo no método **Configuration**:

```
[assembly: OwinStartup(typeof(Capitulo05.AspNetIdentity.Startup))]

namespace Capitulo05.AspNetIdentity
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes
                    .ApplicationCookie,
                LoginPath = new PathString("/Home/Login")
            });
        }
    }
}
```

5.4.3.1. Definindo a string de conexão

Vamos configurar a string de conexão no arquivo **Web.config**. Por default, o ASP.NET Identity utiliza uma conexão chamada **DefaultConnection**. Nosso propósito é gerar o banco de dados com as tabelas do Identity no SQL Server (verifique as configurações do seu servidor na hora da implementação).

```
<connectionStrings>
    <add name="DefaultCornection"
        connectionString="Integrated Security=SSPI;Persist Security
        Info=False;Initial Catalog=DBUSUARIOS;Data Source=.\SQLEXPRESS"
        providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Observe que usamos como nome do banco de dados **DBUSUARIOS**.

5.4.3.2. Implementação do cadastro de usuários

Já temos um action chamado **Criar**. Vamos agora implementar a versão **POST** do método, ou seja, o action que receberá os dados do formulário para cadastro:

```
[HttpPost]
public ActionResult Criar(UsuarioView usuario)
{
    if (!ModelState.IsValid)
    {
        return View();
    }

    //variável para referenciar os dados do usuário,
    //equivalente à tabela
    var usuarioStore = new UserStore<IdentityUser>();

    //variável usada para expor os métodos a serem executados
    //sobre o usuário
    var usuarioManager = new UserManager<IdentityUser>(usuarioStore);

    //Cria uma identidade
    var usuarioInfo = new IdentityUser()
    {
        UserName = usuario.Nome
    };

    //Cria o usuário
    IdentityResult resultado = usuarioManager.Create(
        usuarioInfo, usuario.Senha);

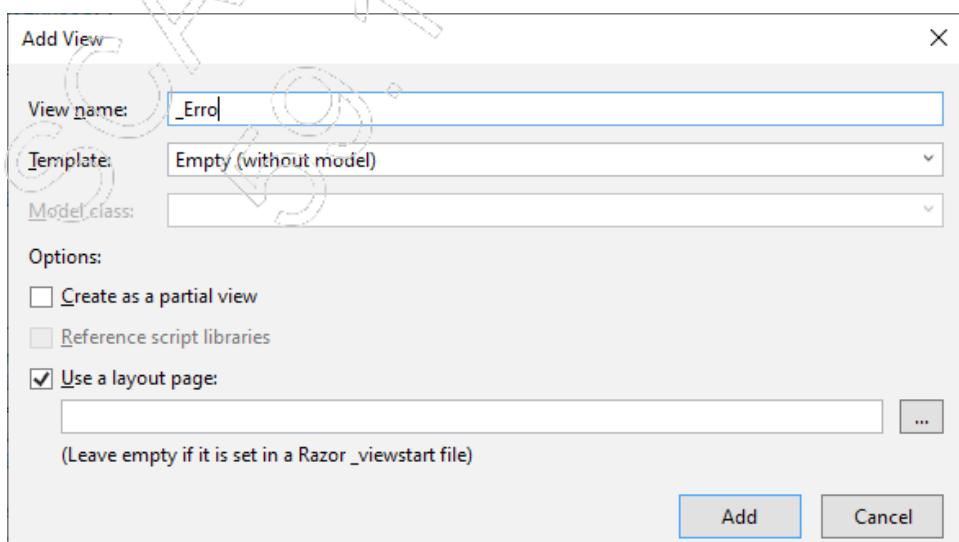
    //se o usuário foi criado, o autentica
    if (resultado.Succeeded)
    {
        //Autentica e volta para a página inicial
        var autManager = System.Web.HttpContext
            .Current.GetOwinContext().Authentication;
        var identidadeUsuario = usuarioManager
            .CreateIdentity(usuarioInfo,
            DefaultAuthenticationTypes.ApplicationCookie);

        autManager.SignIn(new AuthenticationProperties() { },
            identidadeUsuario);
        return RedirectToAction("Index");
    }
    else
    {
        ViewBag.Erro = resultado.Errors.FirstOrDefault();
        return View("_Erro");
    }
}
```

A classe base **IdentityUser** e **IdentityUser<Tkey, Tlogin, Trole, Tclaim>** fornecem as propriedades que representam o usuário. Veja a definição:

```
namespace Microsoft.AspNet.Identity.EntityFramework
{
    public class IdentityUser< TKey, TLogin, TRole, TClaim > : IUser< TKey >
        where TLogin : IdentityUserLogin< TKey >
        where TRole : IdentityUserRole< TKey >
        where TClaim : IdentityUserClaim< TKey >
    {
        public IdentityUser();
        public virtual string Email { get; set; }
        public virtual bool EmailConfirmed { get; set; }
        public virtual string PasswordHash { get; set; }
        public virtual string SecurityStamp { get; set; }
        public virtual string PhoneNumber { get; set; }
        public virtual bool PhoneNumberConfirmed { get; set; }
        public virtual bool TwoFactorEnabled { get; set; }
        public virtual DateTime? LockoutEndUtc { get; set; }
        public virtual bool LockoutEnabled { get; set; }
        public virtual int AccessFailedCount { get; set; }
        public virtual ICollection< TRole > Roles { get; }
        public virtual ICollection< TClaim > Claims { get; }
        public virtual ICollection< TLogin > Logins { get; }
        public virtual TKey Id { get; set; }
        public virtual string UserName { get; set; }
    }
}
```

Estamos também considerando uma view chamada **Erro.cshtml**, que receberá a requisição caso ocorra algum erro no processamento. Esta view deverá ser criada na pasta **Views/Shared**:



Seu conteúdo será o mostrado a seguir:

```
@{  
    ViewBag.Title = "_Erro";  
}  
  
<h2 class="text-danger">Ocorreu o seguinte erro:</h2>  
  
<div class="alert alert-danger">  
    @ViewBag.Erro  
</div>
```

5.4.3.3. Implementação do Login

A implementação do login utilizará boa parte do código que usamos para a inclusão do usuário. A diferença está na busca pelo usuário, em vez da criação. O processo de autenticação é semelhante.

```
[HttpPost]  
public ActionResult Login(LoginView usuario, string returnUrl)  
{  
    if (!ModelState.IsValid)  
    {  
        return View();  
    }  
    else  
    {  
        var usuarioStore = new UserStore<IdentityUser>();  
        var usuarioManager = new UserManager<IdentityUser>(usuarioSto  
re);  
  
        var usuarioInfo = usuarioManager  
            .Find(usuario.Nome, usuario.Senha);  
  
        if (usuarioInfo != null)  
        {  
            var autManager = System.Web.HttpContext  
                .Current.GetOwinContext().Authentication;  
  
            var identidadeUsuario = usuarioManager  
                .CreateIdentity(usuarioInfo,  
                    DefaultAuthenticationTypes.ApplicationCookie);  
  
            autManager.SignIn(new AuthenticationProperties()  
            { IsPersistent = false }, identidadeUsuario);  
  
            return returnUrl == null ? Redirect("Home/Index") :  
                Redirect(returnUrl);  
        }  
        else  
        {  
            ViewBag.Erro = "Usuário ou senha inválidos";  
            return View("_Erro");  
        }  
    }  
}
```

Neste action, o parâmetro **ReturnUrl** é configurado pelo Identity quando o usuário tentar acessar algum recurso protegido e não estiver autenticado. Será incluído na URL um parâmetro **ReturnUrl** com o caminho para o recurso desejado e, quando o usuário realizar o login, ele será encaminhado para onde tentou originalmente.

5.4.3.4. Implementação do Logout

Como já mencionamos anteriormente, o action **Logout** não utilizará nenhuma view. O propósito é desconectar o usuário e encaminhá-lo para a página inicial. Sua implementação está apresentada a seguir:

```
public ActionResult Logout()
{
    var autManager = System.Web.HttpContext.Current
        .GetOwinContext().Authentication;
    autManager.SignOut();

    return RedirectToAction("Index");
}
```

5.4.4. Executando a aplicação

Para executar a aplicação e tentar o acesso a um recurso protegido, criaremos um action chamado **AreaRestrita**. O propósito é mostrar o redirecionamento para Login, se o usuário tentar acessar este action sem estar autenticado. O conteúdo da view **AreaRestrita** será apenas um texto informando que se trata de uma área restrita.

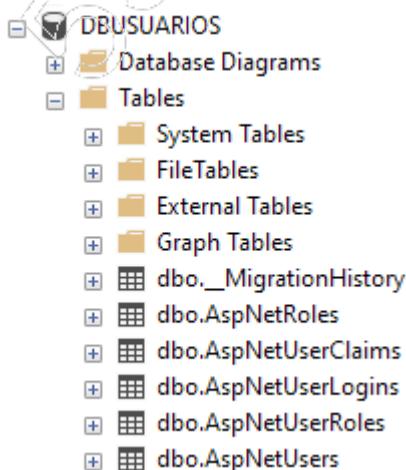
```
[Authorize]
public ActionResult AreaRestrita()
{
    return View();
}
```

O atributo **Authorize** fará com que este action seja executado apenas por usuários devidamente autenticados.

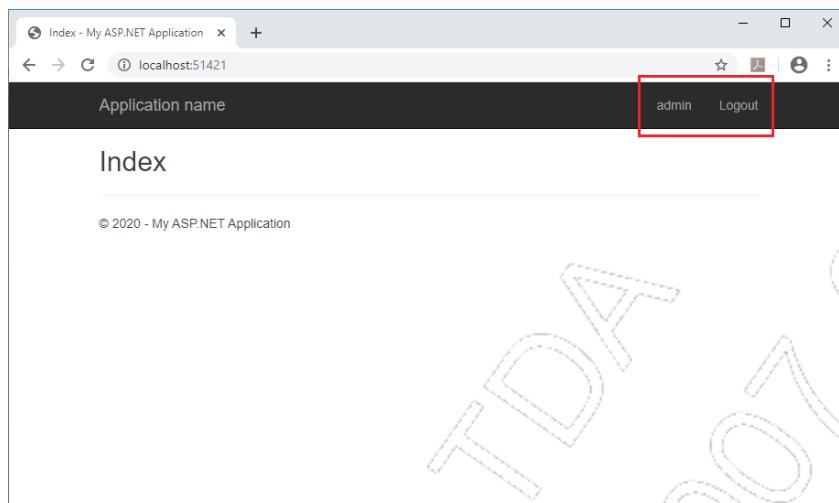
Primeiro, executaremos a aplicação para adicionar um novo usuário.

The first screenshot shows the application's home page ('Index') with a navigation bar containing 'Application name', 'Cadastrar-se' (highlighted with a red box), and 'Login'. The second screenshot shows a user creation form ('Criar Usuário') with fields for 'Nome' (containing 'admin'), 'Senha' (containing dots), 'Confirmar' (containing dots), and a 'Novo usuário' button (highlighted with a red box).

Por ser a primeira vez que estamos executando no sentido de incluir um novo usuário, o ASP.NET Identity cria o banco de dados que definimos na string de conexão.

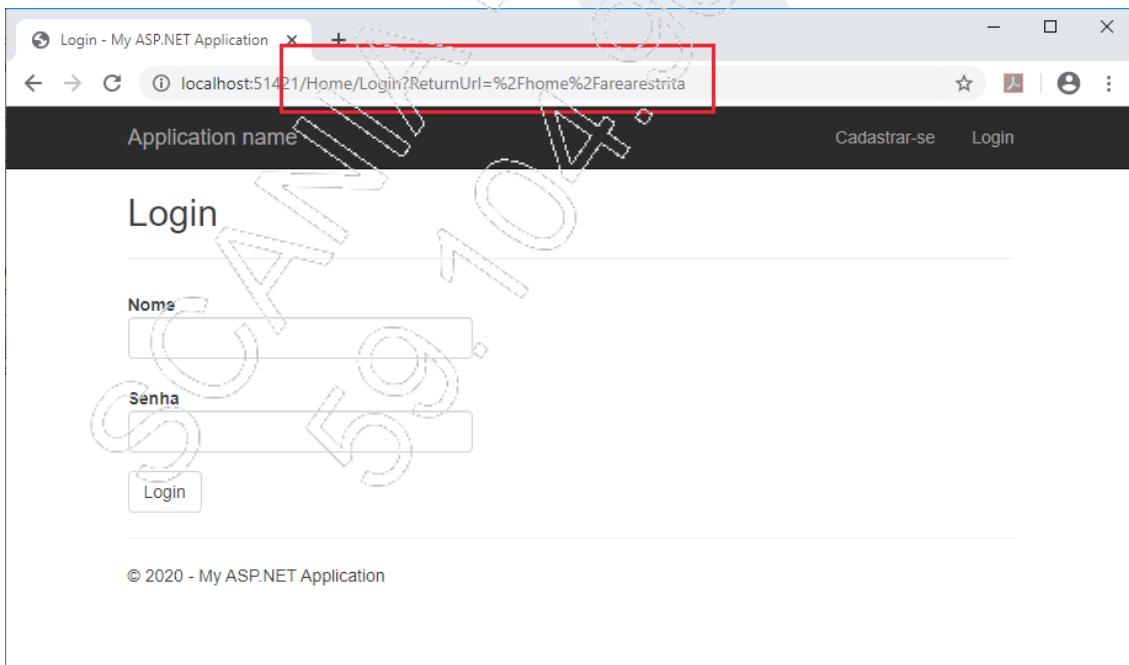


No processo de criação, o usuário é autenticado, e o menu da View Parcial que adicionamos no início muda a visualização:

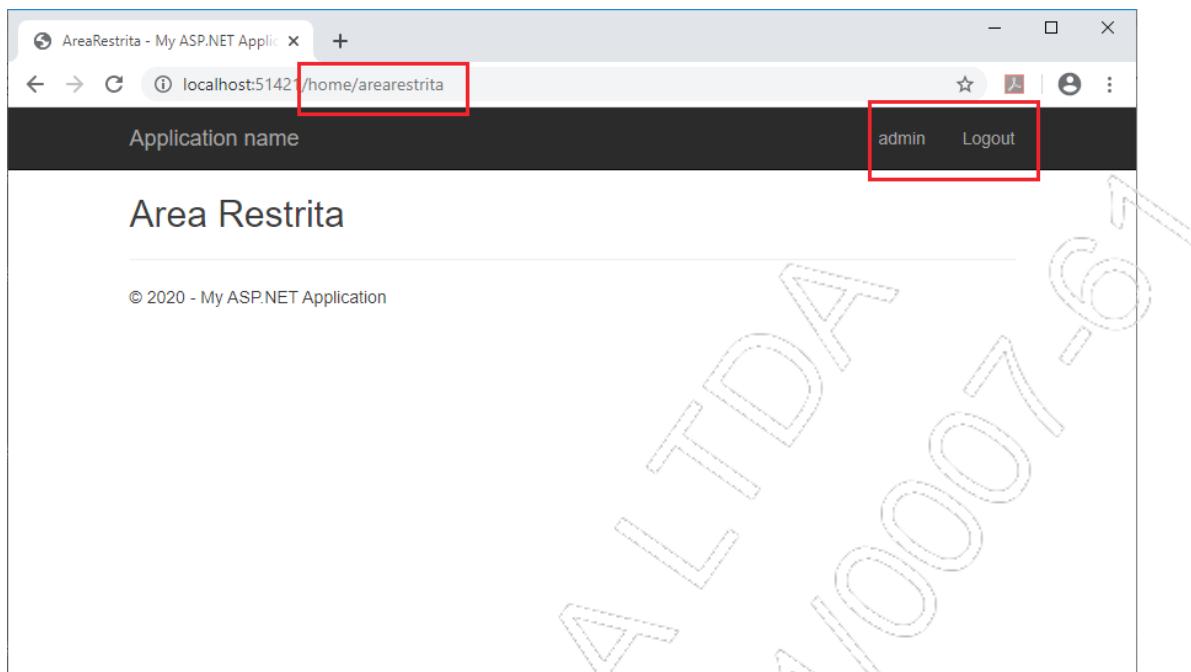


Acionando o menu **Logout**, desconectamos o usuário e retornamos para a página inicial.

O objetivo agora é tentar acessar o action **AreaRestrita**, sem autenticação. Veremos que seremos direcionados para **Login**, com o parâmetro **ReturnUrl** adicionado na URL, apontando para o recurso desejado.

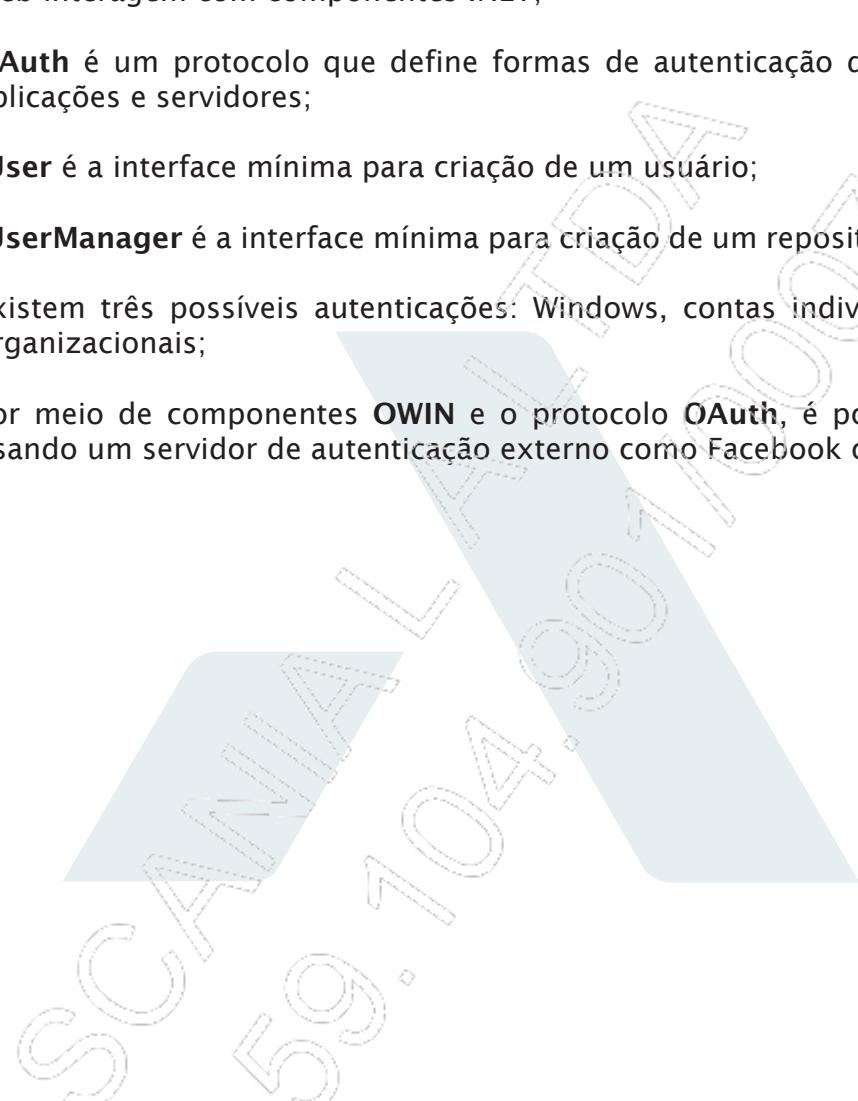


Ao fornecer as credenciais, o usuário é autenticado e direcionado para a view desejada **Área Restrita**.



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

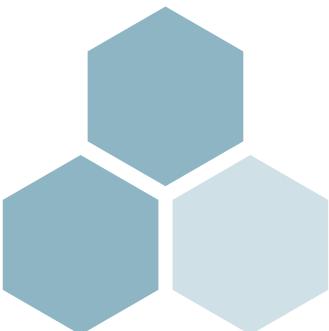
- **OWIN** é uma especificação que define como os componentes de uma aplicação Web interagem com componentes .NET;
 - **OAuth** é um protocolo que define formas de autenticação de usuários entre aplicações e servidores;
 - **IUser** é a interface mínima para criação de um usuário;
 - **IUserManager** é a interface mínima para criação de um repositório de usuários;
 - Existem três possíveis autenticações: Windows, contas individuais ou contas organizacionais;
 - Por meio de componentes **OWIN** e o protocolo **OAuth**, é possível autenticar usando um servidor de autenticação externo como Facebook ou Google.
- 



5

Segurança – ASP.NET Identity

Teste seus conhecimentos



1. Qual é a interface utilizada para definir um usuário no ASP.NET Identity?

- a) IUserManager
- b) OWIN
- c) ApplicationUser
- d) IUser
- e) User.Identity

2. Qual interface deve ser implantada na classe que armazena dados de um usuário no ASP.NET Identity?

- a) IUser
- b) IStore
- c) IUserDb
- d) IUserManager
- e) IUserStore

3. A classe que gerencia as informações de um usuário deve implementar qual interface?

- a) IUser
- b) IUserStore
- c) IUserManager
- d) IUserIdentity
- e) OWIN

4. Que tipo é retornado pelo método Create da classe que gerencia os dados de um usuário?

- a) IdentityResult
- b) User
- c) Principal
- d) Role
- e) ApplicationUser

5. Qual atributo devemos usar para indicar que uma classe é a classe Owin Startup?

- a) OAuth
- b) Owin
- c) OData
- d) OwinStartup
- e) Startup



5

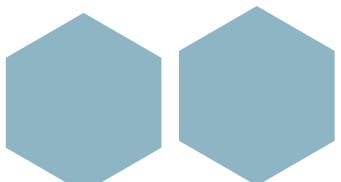
Segurança - ASP.NET Identity



Mãos à obra!



Editora
IMPACTA

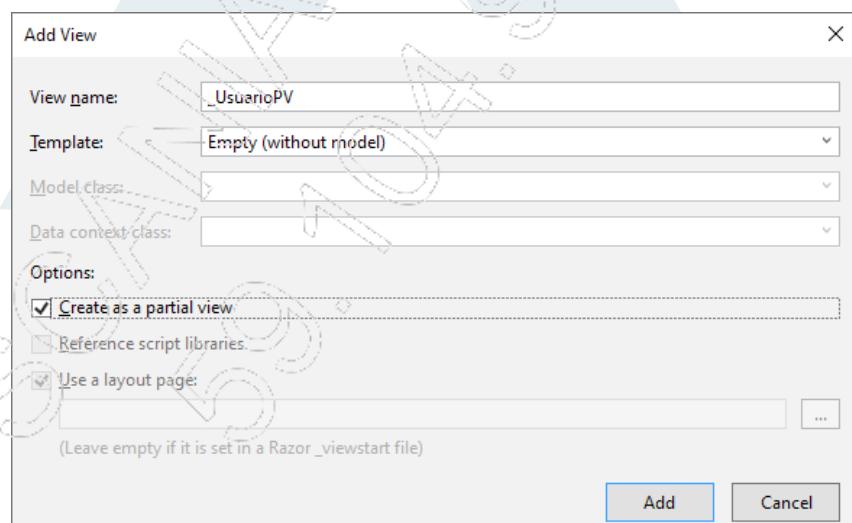


Objetivos:

Neste laboratório, incluiremos os recursos para criação de usuários, login e autorização de acesso. Para isso, continuaremos o projeto desenvolvido até o Capítulo 4, utilizando o mecanismo ASP.NET Identity, OWIN para autenticação e User Entity Framework para persistência dos dados.

Laboratório 1

1. Na pasta **Labs**, crie uma solution vazia (**Blank solution**) chamada **Capitulo05.Labs**;
2. Copie o projeto **Lab.MVC** do solution **Capitulo04.Labs** para a pasta do seu novo solution, **Capitulo05.Labs**;
3. Adicione este projeto ao novo solution. Execute a aplicação para testar se está tudo ok;
4. Vamos iniciar preparando um trecho do menu que apresentará as opções para **Cadastro** e **Login**, se o usuário não estiver autenticado, e **nome do usuário** e **Logout**, se o usuário estiver autenticado. Como se trata de um menu dinâmico, o incluiremos como uma view parcial. Na pasta **View/Shared**, inclua a view **_UsuarioPV**. Marque a opção como **Partial View**:



5. Escreva o seguinte conteúdo:

```
@if (Request.IsAuthenticated)
{
    <ul class="nav navbar-nav navbar-right">
        <li>@Html.ActionLink(User.Identity.Name, "Index",
            "Home")</li>
        <li>@Html.ActionLink("Logout", "Logout", "Usuarios")</li>
    </ul>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li>@Html.ActionLink("Cadastrar-se", "CriarUsuario",
            "Usuarios")</li>
        <li>@Html.ActionLink("Login", "Login", "Usuarios")</li>
    </ul>
}
```

Observe que os actions **Login**, **Logout** e **CriarUsuario** estarão no controller **Usuarios** (classe **UsuariosController**), que definiremos em breve;

6. Insira esta view parcial no layout, ao lado do menu de opções (observe o ponto de inserção e o método utilizado para inseri-lo):

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li>
            @Html.ActionLink("Clientes", "Index", "Clientes")
        </li>
        <li>
            @Html.ActionLink("Produtos", "Index", "Produtos")
        </li>
        <li>
            @Html.ActionLink("Pedidos", "Index", "Pedidos")
        </li>
        <li>
            @Html.ActionLink("Itens", "Index", "Itens")
        </li>
    </ul>
    @Html.RenderPartial("_UsuarioPV");
</div>
```

7. Execute a aplicação para visualizar o novo menu:



8. Na pasta **Model**, inclua as classes **UsuarioViewModel** e **LoginViewModel**. Elas serão usadas no controller para gerar os actions e as views para inclusão de usuários e para login, respectivamente;

```
using System.ComponentModel.DataAnnotations;

namespace Lab.MVC.Models
{
    public class UsuarioViewModel
    {
        [Required]
        public string Nome { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }

        [Compare("Senha")]
        [DataType(DataType.Password)]
        public string Confirma { get; set; }
    }

    using System.ComponentModel.DataAnnotations;

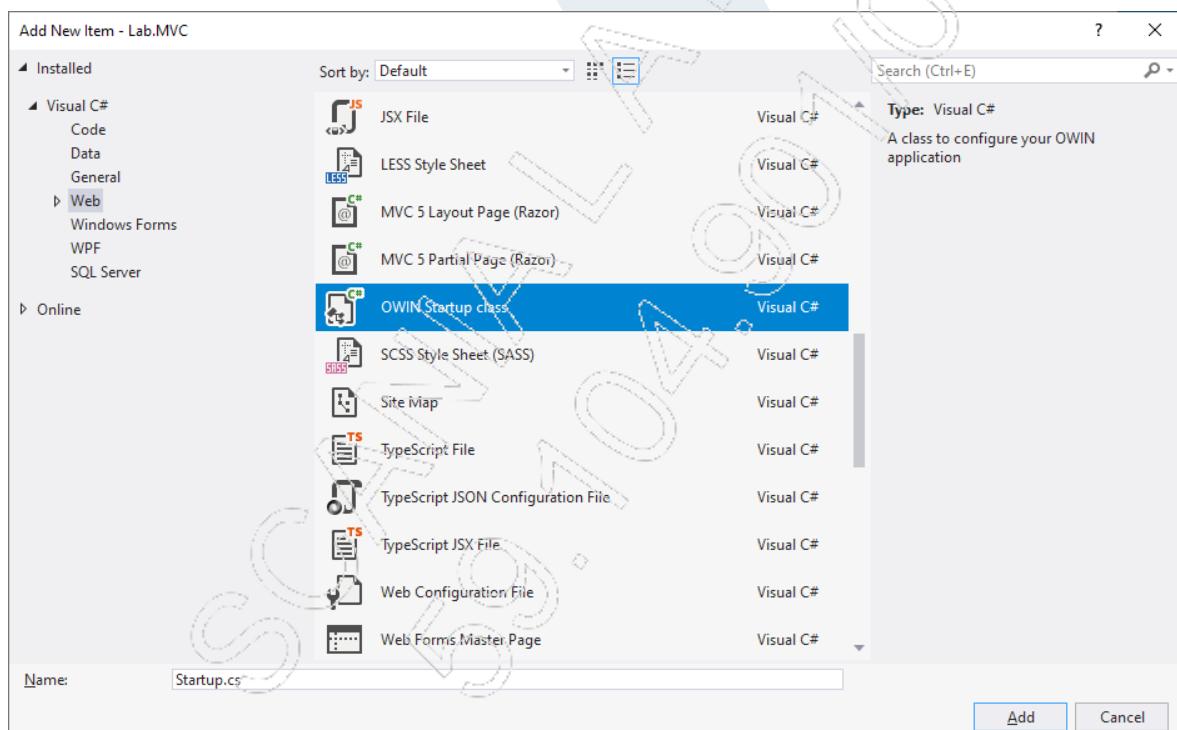
    namespace Lab.MVC.Models
    {
        public class LoginViewModel
        {
            [Required]
            public string Nome { get; set; }

            [Required]
            [DataType(DataType.Password)]
            public string Senha { get; set; }
        }
    }
}
```

9. Agora vamos implementar o registro de usuário usando ASP.NET Identity e autenticação com componentes OWIN. Através do Nuget, instale os componentes:

Asp.Net Identity Entity
Asp.Net Identity Owin
Owin Host

10. Na raiz da aplicação, insira o arquivo de configuração OWIN, denominado **Startup.cs**. Com o botão direito do mouse, selecione a opção **Add / New Item**. Selecione a opção **Owin Startup Class**. Chame o arquivo de **Startup.cs**:



Esse arquivo é usado pelo framework OWIN. Vamos adicionar a sintaxe adiante:

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;

[assembly: OwinStartup(typeof(Lab.MVC.Startup))]

namespace Lab.MVC
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.UseCookieAuthentication(new
                CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes
                    .ApplicationCookie,
                LoginPath = new PathString("/Usuarios/Login"),
                LogoutPath = new PathString("/Usuarios/Logout")
            });
        }
    }
}
```

Este método é chamado pelo próprio OWIN;

11. Vamos definir a string de conexão para que o ASP.NET Identity possa usá-la para criar o banco de dados. Por padrão, o Identity utiliza uma conexão chamada **DefaultConnection**. Para defini-la, abra o arquivo **Web.config** e localize a string de conexão definida para o banco de dados **DB_VENDAS**. Adicione a seguinte instrução ao elemento **<connectionStrings>**:

```
<add  
    name="DefaultConnection"  
    connectionString="Integrated Security=SSPI;Persist Security  
Info=False;Initial Catalog=DB_USUARIOS;Data Source=.\SQLEXPRESS"  
    providerName="System.Data.SqlClient" />
```

Lembre-se de adequar a string de conexão ao banco de dados que você estiver usando;

12. Agora vamos implementar os actions para incluir usuários, realizar login e logout. Crie o controller **UsuariosController** com os actions **Login**, **Logout** e **CriarUsuario**. Apenas o action **Logout** não terá a versão POST:

```
using Lab.MVC.Models;  
using Microsoft.AspNet.Identity;  
using Microsoft.AspNet.Identity.EntityFramework;  
using Microsoft.Owin.Security;  
using System;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;  
  
namespace Lab.MVC.Controllers  
{  
    public class UsuariosController : Controller  
    {  
        [HttpGet]  
        public ActionResult CriarUsuario()  
        {  
            return View();  
        }  
  
        [HttpPost]  
        public ActionResult CriarUsuario(UsuarioViewModel usuario)  
        {  
            if (!ModelState.IsValid)  
            {  
                return View();  
            }  
  
            try  
            {
```

```
//Tabela de usuários gerenciada pelo Identity
var usuarioStore = new UserStore<IdentityUser>();

//Objeto para manipular os usuários
var usuarioManager = new
    UserManager<IdentityUser>(usuarioStore);

//Cria uma identidade
var usuarioInfo = new IdentityUser()
{
    UserName = usuario.Nome
};

//Cria o usuário
IdentityResult resultado = usuarioManager.Create(
    usuarioInfo, usuario.Senha);

if (resultado.Succeeded)
{
    //Autentica e volta para a página inicial
    var autManager = System.Web.HttpContext
        .Current.GetOwinContext().Authentication;
    var identidadeUsuario = usuarioManager
        .CreateIdentity(usuarioInfo,
        DefaultAuthenticationTypes.ApplicationCookie);

    autManager.SignIn(new AuthenticationProperties() { },
        identidadeUsuario);
    return RedirectToAction("Index", "Home");
}
else
{
    throw new Exception(resultado
        .Errors.FirstOrDefault());
}
catch (Exception ex)
{
    ViewBag.MensagemErro = ex.Message;
    return View("_Erro");
}

[HttpGet]
public ActionResult Login()
{
    return View();
}
```

```
[HttpPost]
public ActionResult Login(LoginViewModel usuario,
    string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View();
    }

    try
    {
        var usuarioStore = new UserStore<IdentityUser>();
        var usuarioManager = new
            UserManager<IdentityUser>(usuarioStore);

        var usuarioInfo = usuarioManager
            .Find(usuario.Nome, usuario.Senha);

        if (usuarioInfo != null)
        {
            var autManager = System.Web.HttpContext
                .Current.GetOwinContext().Authentication;

            var identidadeUsuario = usuarioManager
                .CreateIdentity(usuarioInfo,
                    DefaultAuthenticationTypes.ApplicationCookie);

            autManager.SignIn(new AuthenticationProperties()
            { IsPersistent = false }, identidadeUsuario);

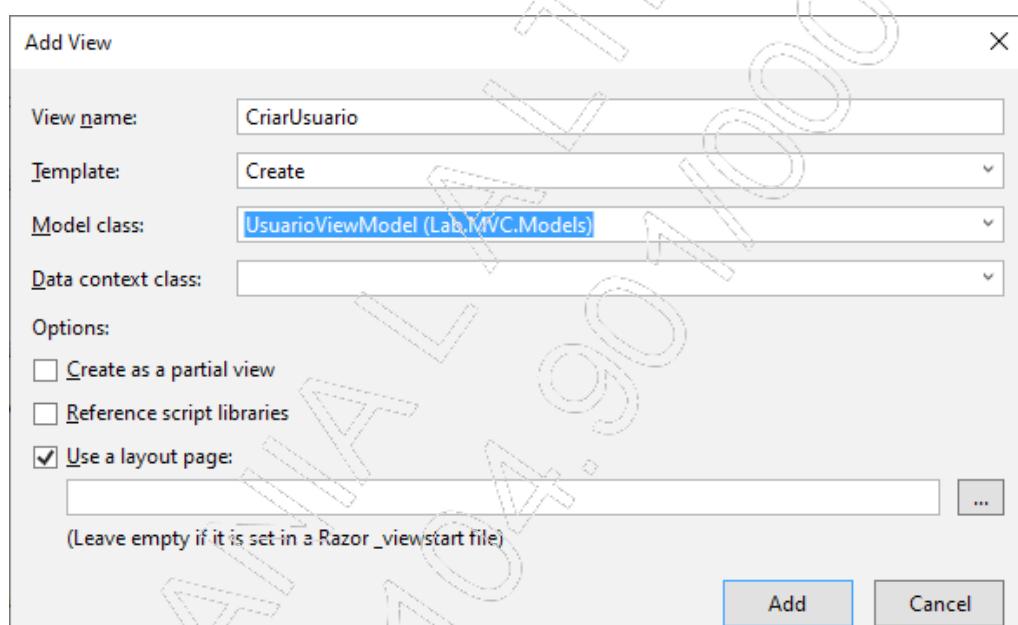
            return returnUrl == null ? Redirect("/Home/Index") :
                Redirect(returnUrl);
        }
        else
        {
            throw new Exception("Usuário ou senha inválidos");
        }
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

[HttpGet]
public ActionResult Logout()
{
    var autManager = System.Web.HttpContext
        .Current.GetOwinContext().Authentication;
    autManager.SignOut();

    return RedirectToAction("Index", "Home");
}
}
```

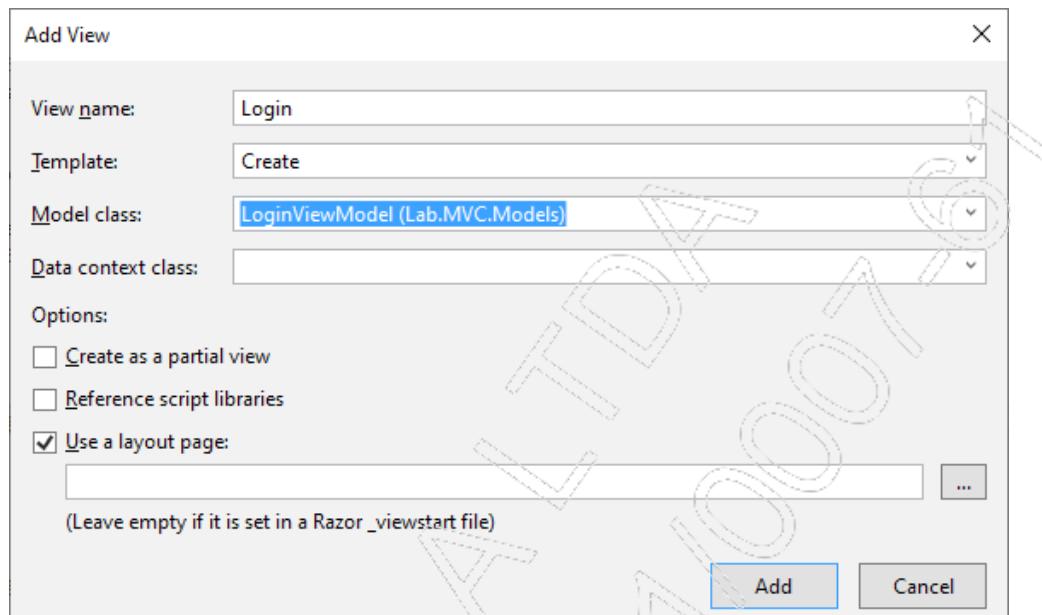
No action **Login**, versão POST, incluímos um parâmetro adicional chamado **returnUrl**. Este valor é atribuído pelo ASP.NET à URL quando tentamos acessar um recurso protegido. Se não houver nenhum usuário autenticado, a aplicação redireciona o usuário da aplicação para a tela de login, conforme especificado no arquivo **Startup.cs**. Neste caso, após efetuar o login, o usuário é encaminhado para o recurso que ele estava tentando acessar;

13. Crie a view para o action **CriarUsuario**.



```
@model Lab.MVC.Models.UsuarioViewModel  
{@  
    ViewBag.Title = "Criar Usuário";  
}  
  
<h2>Criar Usuário</h2>  
  
@using (Html.BeginForm())  
{  
    @Html.AntiForgeryToken()  
  
    <div class="form-horizontal">  
  
        <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">  
            @Html.LabelFor(model => model.Senha, htmlAttributes:  
                new { @class = "control-label col-md-2" })  
            <div class="col-md-10">  
                @Html.EditorFor(model => model.Senha, new { htmlAttributes =  
                    new { @class = "form-control" } })  
                @Html.ValidationMessageFor(model => model.Senha, "",  
                    new { @class = "text-danger" })  
            </div>  
        </div>  
  
        <div class="form-group">  
            @Html.LabelFor(model => model.Confirma, htmlAttributes:  
                new { @class = "control-label col-md-2" })  
            <div class="col-md-10">  
                @Html.EditorFor(model => model.Confirma, new { htmlAttributes =  
                    new { @class = "form-control" } })  
                @Html.ValidationMessageFor(model => model.Confirma, "",  
                    new { @class = "text-danger" })  
            </div>  
        </div>  
  
        <div class="form-group">  
            <div class="col-md-offset-2 col-md-10">  
                <input type="submit" value="Criar Usuário"  
                    class="btn btn-info" />  
            </div>  
        </div>  
    </div>  
}
```

14. Crie a view Login:



```
@model Lab.MVC.Models.LoginViewModel  
{@  
    ViewBag.Title = "Login";  
}  


## Login

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


---

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

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Login"
            class="btn btn-info" />
    </div>
</div>
</div>
}
```

15. No action **Incluir** dos controllers **ClientsController** e **ProdutosController**, versão GET, adicione o atributo **[Authorize]**:

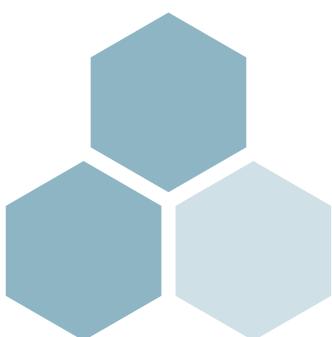
```
[HttpGet]
[Authorize]
public ActionResult Incluir()
{
    return View();
}
```

16. Execute a aplicação. Selecione o menu **Cadastrar-se**. Realize um cadastro de usuário. Observe que, na primeira vez, ocorrerá certa demora porque o banco de dados será criado. Em seguida, efetue **Logout**. Acesse o cadastro de clientes. O que você pode observar na URL? Após fazer login, o que ocorre?

6

Serviços - Web API

- ◆ Introdução à Web API;
- ◆ REST na prática;
- ◆ Usando Web API;
- ◆ Criando um projeto para consumir o serviço.



6.1. Introdução à Web API

Web API é um framework da Microsoft para criar serviços que usam a Internet e o protocolo HTTP como o meio de transmitir informações. Aplicações que usam esse tipo de princípio são conhecidas como aplicações **RESTful**.

REST significa **Representational State Transfer** (transferência de estado representativo) e é um conjunto de princípios e regras para utilização de serviços usando a Internet. Os conceitos mais importantes do REST são recurso e identificador. Um recurso é uma informação disponível, e o identificador é um texto único (no caso da Web, uma URL) que fornece acesso a esse recurso.

Os princípios fundamentais de uma aplicação REST são os seguintes:

- **Uniform interface:** O princípio da interface **Uniform** determina que os recursos (informações disponíveis) são sempre obtidos a partir de um identificador único. No caso da Web, o identificador é uma URL. Por exemplo, o identificador do produto cujo código é 32, da empresa X, pode ser obtido pelo endereço <http://www.empresax.com.br/restApp/Produto/32>;
- **Stateless:** O servidor não deve manter nenhuma informação do cliente (aplicação que solicita um recurso). Toda a informação necessária para realizar uma operação deve vir na própria solicitação, usando a URL, os dados postados internamente, o verbo de comando (GET, POST, PUT, DELETE) e os cabeçalhos HTTP;
- **Cacheable:** Os recursos disponibilizados por um serviço REST devem ter a capacidade de ser gravados em cache pelo cliente. Na prática, isso significa que as informações retornadas não podem depender de recursos externos, por exemplo, uma conexão aberta com um banco de dados;
- **Client-server:** A aplicação deve respeitar os princípios das aplicações cliente-servidor, em que um não sabe os detalhes de implementação do outro. Não importa para o cliente como a informação foi gerada e nem importa para o servidor a tecnologia usada para fazer uma solicitação. Assim, ambas as partes podem evoluir apenas respeitando as regras de protocolo de comunicação;
- **Layered system:** Os sistemas devem ser construídos em camadas, podendo o cliente chamar um servidor que, por sua vez, chama outro, e este chama ainda outro. Como cada componente não sabe os detalhes de implementação, todos podem se interligar, sem afetar outras partes da aplicação;
- **Code on demand:** O servidor pode disponibilizar código (geralmente em JavaScript) para que o cliente execute. Este princípio é muito utilizado pela Microsoft, principalmente no framework AJAX.

6.2. REST na prática

Os princípios REST utilizam todo o potencial do protocolo HTTP. Apesar de não ter sido concebido para uso exclusivo desse protocolo, todas as restrições impostas são parte integrante dos recursos disponíveis no ambiente Web e na comunicação entre o browser e um servidor.

Veja algumas dessas características:

- **Todo recurso deve ter um identificador:** Na Web, isso já é a norma. Uma URL como www.microsoft.com é um identificador único para um recurso, no caso, a página de entrada da Microsoft;
- **Stateless:** A comunicação via HTTP é stateless (sem estado), por natureza. O que o IIS e outros servidores fazem é usar o recurso de cookies para inserir, no cliente, um código que fica gravado no servidor e que é recuperado toda vez que há uma solicitação em que este código é enviado novamente. Na verdade, é uma maneira de burlar a característica stateless do ambiente Web;
- **O cliente deve solicitar o formato em que deseja receber a resposta:** Uma requisição HTTP (request) contém uma série de informações do tipo chave/valor que são chamadas cabeçalhos (headers). Toda chamada a uma página pode passar a chave **Accept** com uma string descrevendo o tipo de dado que está esperando. Por exemplo: **Accept: text/xml**;
- **Toda informação necessária à tarefa solicitada deve estar na solicitação:** É neste ponto que entra a parte mais importante do processo: o verbo. Na solicitação HTTP, o verbo é uma palavra que descreve o que está sendo enviado. Os verbos mais conhecidos são GET e POST. Os verbos mais utilizados na tecnologia REST são os seguintes:
 - **GET:** Solicitação para obter uma informação do servidor;
 - **POST:** Comando para incluir uma informação no servidor;
 - **PUT:** Solicitação para que o servidor processe uma informação sendo enviada pelo cliente. No ambiente REST, este é um comando para alterar dados;
 - **DELETE:** Comando para excluir uma informação do servidor.

Existem outras palavras, como PATCH, OPTIONS e HEAD, mas, do ponto de vista de acesso a dados, os verbos exibidos na lista são suficientes para a maioria das aplicações.

- **Cacheable:** Colocar as informações em cache é algo que os browsers já fazem desde o início da Internet. Como tudo que é transmitido está no formato de texto, é muito fácil criar um sistema de armazenamento interno;
- **Layered-System/Client-Server:** O ambiente Web é inherentemente construído em camadas (Layered System). Quando alguém faz uma compra em um site e paga com cartão de crédito, nenhuma conexão é estabelecida entre o browser e o serviço da operadora de cartão de crédito. Tudo acontece no servidor, que deve chamar um componente, que deve chamar outro, e assim por diante, até chegar ao serviço disponibilizado pela administradora do cartão.

O exemplo adiante é a declaração HTTP exatamente como é enviada ao servidor. Neste caso, a URL é usada para ter acesso a um recurso, o verbo POST e o conteúdo no body são usados para incluir informações e os cabeçalhos HTTP (headers) são usados para definir o formato da resposta que deve ser enviada:

```
POST http://localhost:1513/api/Funcionario HTTP/1.1
Host: localhost:1513
Content-Length: 19
Accept: application/xml
Content-Type: application/x-www-form-urlencoded

nome=Maria+da+Silva
```

Este outro exemplo usa URL para ter acesso à lista **Produtos**. O verbo GET juntamente com a URL **api/Produtos** define o que deve ser executado. Neste caso, o retorno deve ser no formato JSON:

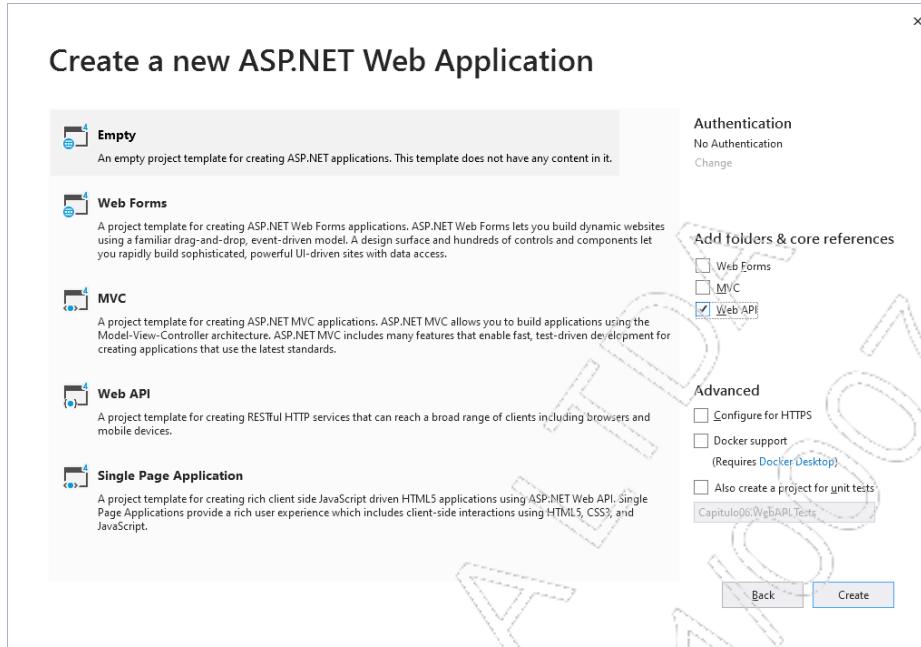
```
GET http://localhost:1513/api/Produtos HTTP/1.1
Accept: application/json
```

Resumindo, o ambiente Web usando o protocolo HTTP atende a todos os requisitos para criar aplicativos RESTful.

6.3. Usando Web API

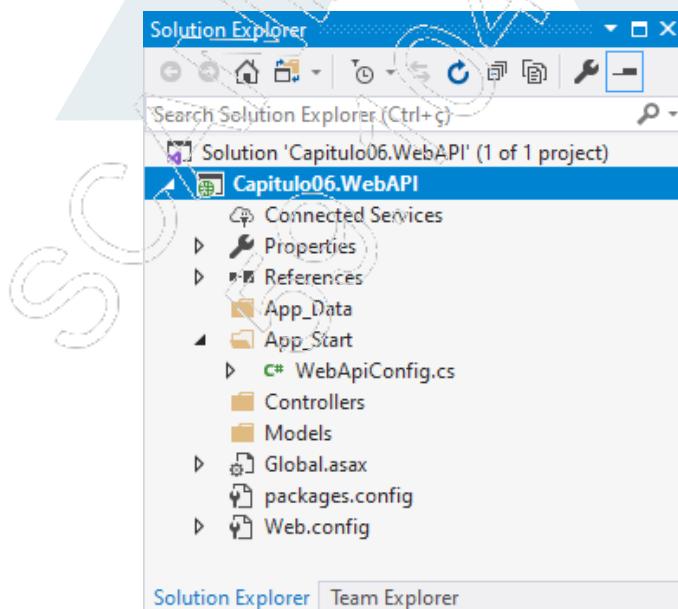
O framework **Web API** contém toda a funcionalidade necessária para criar aplicações RESTful e pode ser usado em qualquer tipo de projeto, como Web Forms, MVC ou simples páginas HTML.

Para apresentar o uso do framework Web API, vamos definir um novo projeto chamado **Capítulo06.WebAPI**. Será um projeto vazio modelo **ASP.NET Web Application (.NET Framework)**, com template **Web Api**:



6.3.1. Estrutura Web API

O projeto criado nos padrões especificados produz o seguinte conteúdo no Solution Explorer:



Vamos destacar os arquivos mais proeminentes do projeto:

- **Global.asax**

Neste arquivo, é criada, dentro do evento **Application_Start**, uma chamada ao método **Configure** da classe **GlobalConfiguration**, passando como parâmetro o método **Register** da classe **WebApiConfig**, que está definida dentro da pasta **App_Start**. O método **Configure** espera um delegate, ou seja, um método, cuja assinatura é um parâmetro do tipo **HttpApplication**:

```
using System.Web.Http;

namespace Capitulo06.WebAPI
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

- **App_Start / WebApiConfig.cs**

Dentro da pasta **App_Start** está definida a classe **WebApiConfig**. Essa classe registra, no mecanismo do ASP.NET, uma route (rota), recurso também usado pelo MVC. No caso da Web API, o modelo de roteamento inclui a pasta **api** como prefixo.

```
using System.Web.Http;

namespace Capitulo06.WebAPI
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services
            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

O formato da URL para utilizar a Web API é o seguinte:

```
Servidor/api/yyyy/xxxx
```

Em que **xxxx** é o Controller e **yyyy** é o método.

```
Servidor/api/Produto  
Servidor/api/Produto/1  
Servidor/api/Clientes  
Servidor/api/NotaFiscal/435930
```

A parte mais importante está neste trecho:

```
config.Routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
)
```

Em que:

- **config** é o parâmetro recebido do tipo **HttpConfiguration** e é a classe que é usada para iniciar a aplicação;
- **Routes** é uma coleção de objetos do tipo **IhttpRoute** e que serve para identificar um caminho;
- **MapHttpRoute** é um método de extensão que adiciona, na coleção, um objeto do tipo **IhttpRoute** com as configurações da rota desejada, neste caso, **Servidor/api/yyyy/xxxx**. Este método pode aceitar, dentre outras opções, três parâmetros:
 - **name**: O nome da rota desejada;
 - **routeTemplate**: O modelo da rota. No modelo, podem ser inseridas strings delimitadas por chaves, chamadas de tokens. O token "**{controller}**" faz o mecanismo da Web API procurar uma classe do tipo **Controller**, cujo nome seja igual ao definido na URL:

```
config.Routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
)
```

- **Defaults:** São valores preenchidos automaticamente quando não é passado um dos parâmetros. Por exemplo: `api/{controller}/{id}` como template e `api/produtos` como URL chamada. O `{id}` não foi chamado na URL anterior. Mas este parâmetro tem um valor padrão que, no caso, é opcional, isto é, não precisa ser passado.

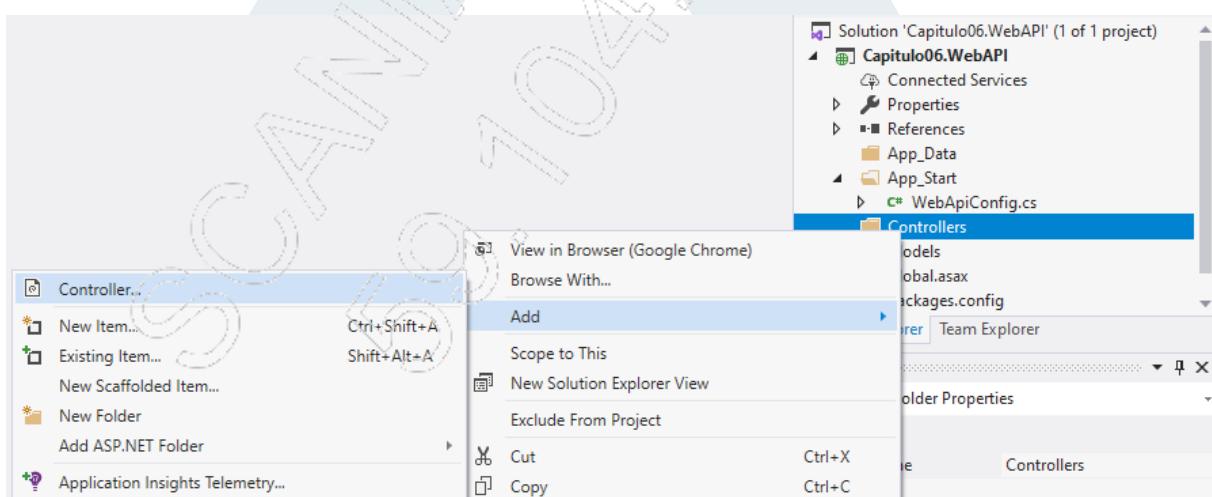
O parâmetro **defaults** é do tipo **object**, ou seja, pode ser passado a qualquer classe. O aplicativo analisa as propriedades do objeto e verifica se coincide com o template. Nessa configuração gerada pelo Visual Studio, é usado um tipo anônimo com a propriedade **Id** tendo o valor da propriedade **Opcional** da classe **RouteParameters**.

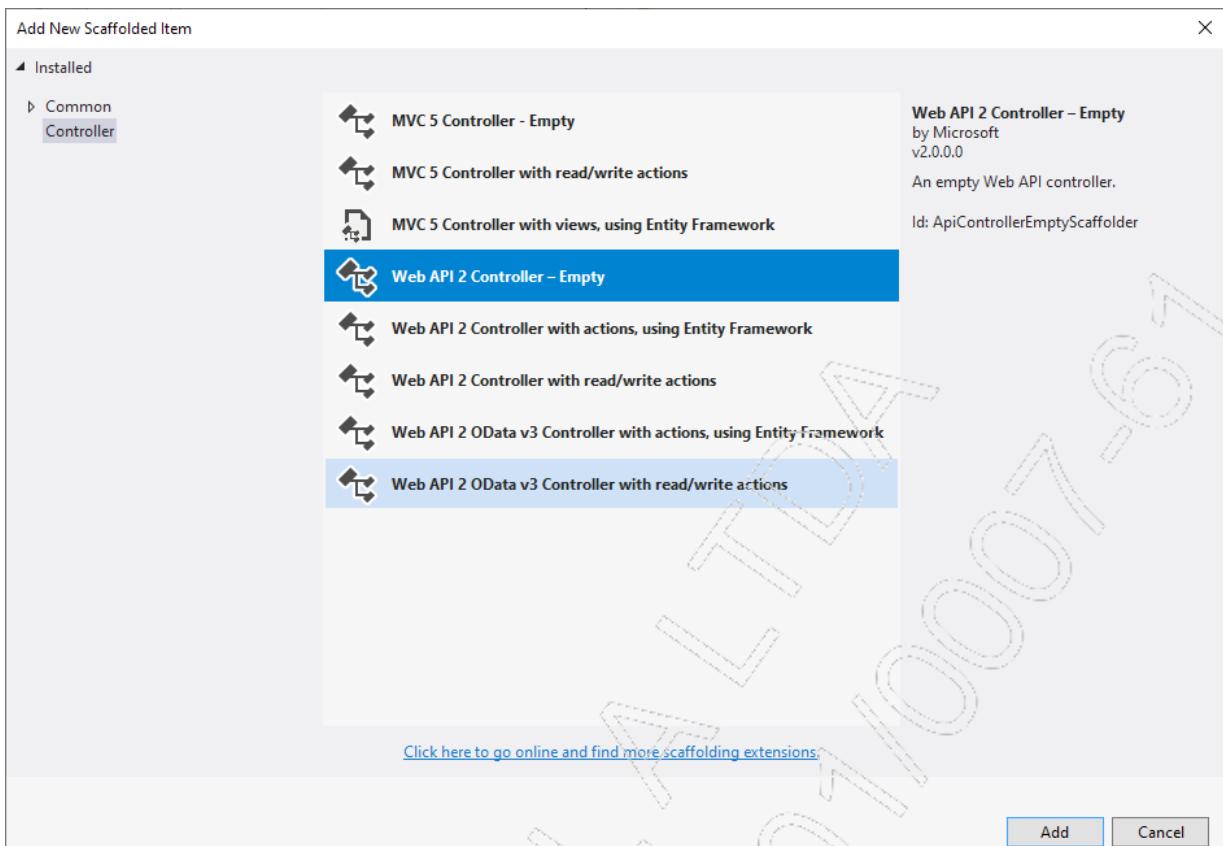
Como será visto no próximo tópico, usar a Web API é muito mais fácil do que entender todo o mecanismo envolvido para que tudo funcione. O propósito de existir templates que criem o ambiente necessário é facilitar a escrita do código e fazer com que o programador se concentre no que a aplicação deve fazer, e não nos detalhes técnicos envolvidos.

Apesar disso, é importante conhecer o ambiente para não ficar dependente de implementações de terceiros. Muitas vezes, é necessário adaptar ou estender o comportamento padrão dos componentes, e isso só pode ser feito se houver um bom entendimento dos conceitos envolvidos no processo.

6.3.2. Criando um serviço REST

Uma vez que o projeto tenha sido criado, vamos adicionar um controller a ele. Este controller deverá ser do tipo **Web API**. Para tanto, clique com o botão direito do mouse sobre a pasta **Controllers**, e no menu de contexto, selecione **Add / Controller...**:





Chamaremos a classe de **HomeController**.

Vale lembrar que as regras e convenções para o controller em uma aplicação MVC também se aplicam ao controller do projeto Web API. Observe que a classe gerada é subclasse de **ApiController**.

```
using System.Web.Http;

namespace Capitulo06.WebAPI.Controllers
{
    public class HomeController : ApiController
    {
    }
}
```

Além da convenção aplicada aos controllers, o Web API também mantém uma convenção aplicada aos nomes dos actions que, neste caso, representam verbos HTTP. A convenção considera o seguinte:

- Métodos iniciados com GET: executam HTTP GET;
- Métodos iniciados com POST: executam HTTP POST;
- Métodos iniciados com PUT: executam HTTP PUT.

Não é necessário listar todos os métodos aqui, mas a convenção é aplicada aos demais verbos HTTP.

6.3.3. Definindo actions para o verbo HTTP GET

Para exemplificar, vamos criar alguns objetos de teste e verificar sua aplicabilidade no projeto:

1. Na pasta **Models**, crie uma classe chamada **Produto** com as propriedades apresentadas a seguir:

```
namespace Capitulo06.WebAPI.Models
{
    public class Produto
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
    }
}
```

2. Crie uma pasta chamada **Dados**;

3. Nessa pasta, crie uma classe chamada **DbDados**, contendo os seguintes métodos e atributos:

```
using Capitulo06.WebAPI.Models;
using System.Collections.Generic;

namespace Capitulo06.WebAPI.Dados
{
    public class DbDados
    {
        public static List<Produto> produtos = new List<Produto>()
        {
            new Produto(){ Id = 10, Descricao = "Mouse", Preco = 40.00 },
            new Produto(){ Id = 20, Descricao = "Camisa", Preco = 59.90 },
            new Produto(){ Id = 30, Descricao = "Bike", Preco = 450.00 }
        };

        public static IEnumerable<string> ListarCursos()
        {
            return new List<string>()
            {
                "C#", "PHP", "Javascript", "SQL Server"
            };
        }

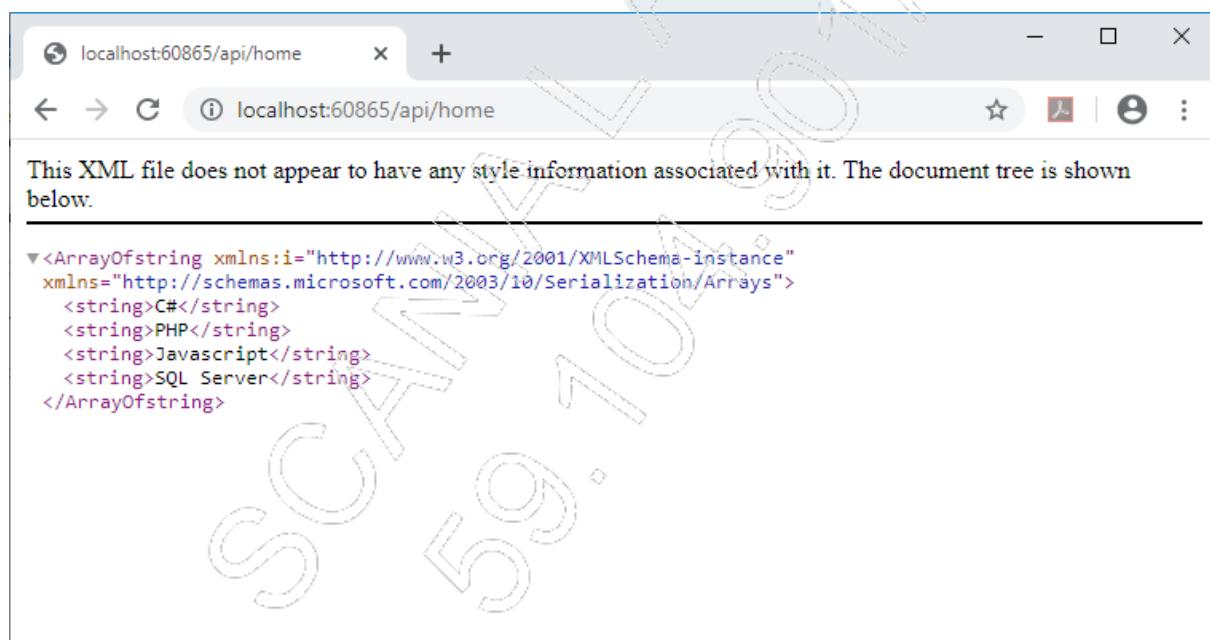
        public static IEnumerable<Produto> ListarProdutos()
        {
            return produtos;
        }
    }
}
```

4. No controller **Home**, escreva os seguintes métodos (actions):

```
using Capitulo06.WebAPI.Dados;
using System.Collections.Generic;
using System.Web.Http;

namespace Capitulo06.WebAPI.Controllers
{
    public class HomeController : ApiController
    {
        public IEnumerable<string> GetCursos()
        {
            return DbDados.ListarCursos();
        }
    }
}
```

Observe que o action **GetCursos** inicia com o prefixo **Get**, significando que sua chamada será realizada quando realizarmos uma requisição GET, simplesmente chamando o controller na URL de acordo com a rota estabelecida no arquivo **WebApiConfig.cs**. Para verificar, execute a aplicação e adicione **/api/home** na URL:



O serviço foi executado e a lista de cursos foi obtida no formato XML. Desejamos que o formato padrão seja JSON. Para isso, vamos remover a formatação XML e adicionar a formatação JSON, na classe `WebApiConfig` (arquivo `WebApiConfig.cs`), usando o parâmetro `config` do método `Register`:

```
using System.Web.Http;

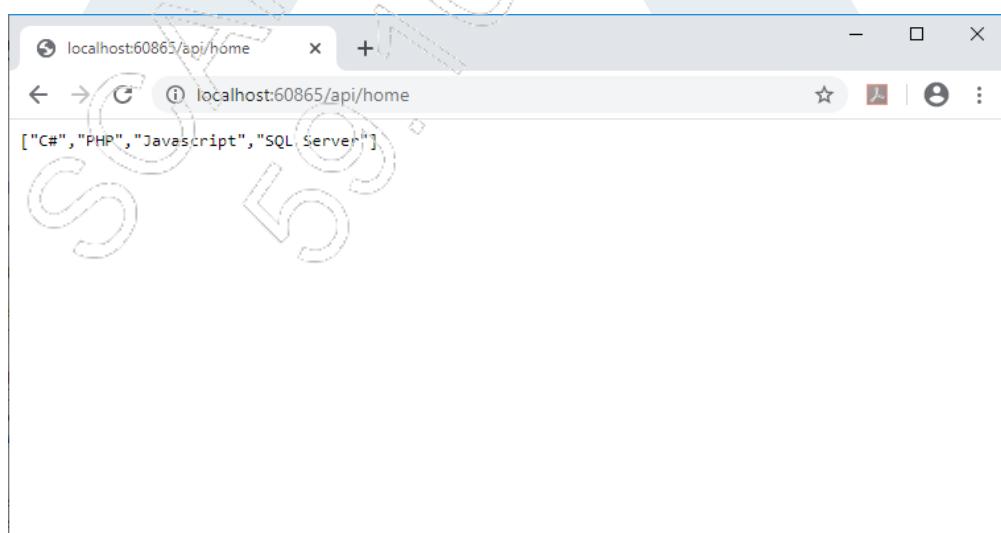
namespace Capitulo06.WebAPI
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            config.Formatters.Remove(config.Formatters.XmlFormatter);
            config.Formatters.Add(config.Formatters.JsonFormatter);
        }
    }
}
```

Executando novamente, temos o resultado mostrado a seguir:



A lista de cursos, definida como uma lista de string, foi consumida e seu resultado foi retornado como um objeto JSON.

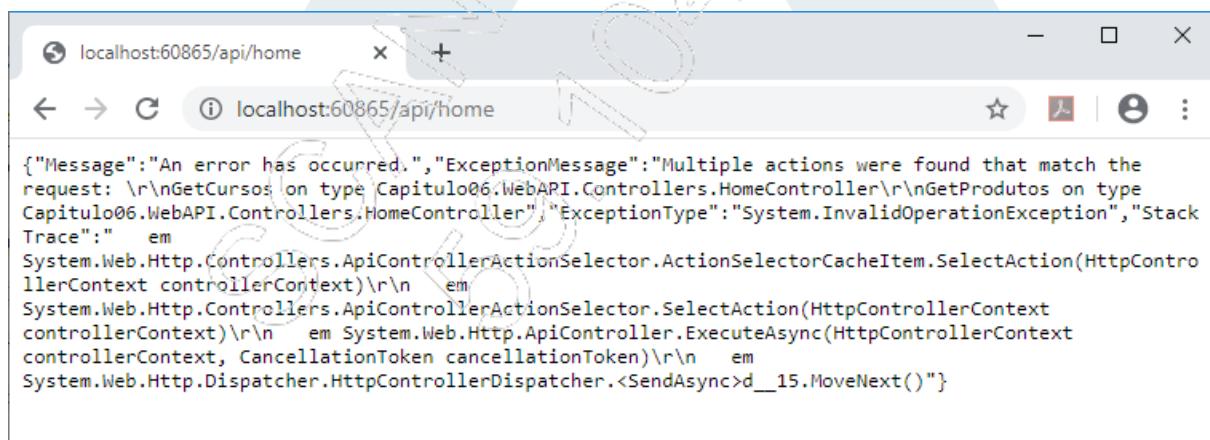
Além desta lista de cursos, definimos também um método com uma lista de objetos da classe **Produto**. Para que o serviço disponibilize a consulta a esta lista, devemos adicionar um action ao controller:

```
using Capitulo06.WebAPI.Dados;
using Capitulo06.WebAPI.Models;
using System.Collections.Generic;
using System.Web.Http;

namespace Capitulo06.WebAPI.Controllers
{
    public class HomeController : ApiController
    {
        public IEnumerable<string> GetCursos()
        {
            return DbDados.ListarCursos();
        }

        public IEnumerable<Produto> GetProdutos()
        {
            return DbDados.ListarProdutos();
        }
    }
}
```

Ao executar a aplicação e realizar a chamada à URL `/api/home`, teremos o resultado:



Analisando o erro produzido, podemos ver que a convenção de nomes aplicada pelo Web API causou um conflito, pois temos dois actions iniciando com Get, sem parâmetros: **GetCursos** e **GetProdutos**. O framework não tem como saber qual dos dois chamar, pois a URL fornecida atende a ambos.

Em casos como este, devemos personalizar a rota para um dos dois actions, de forma a eliminar a ambiguidade. Vamos alterar o último action adicionado ao controller: **GetProdutos**:

```
using Capitulo06.WebAPI.Dados;
using Capitulo06.WebAPI.Models;
using System.Collections.Generic;
using System.Web.Http;

namespace Capitulo06.WebAPI.Controllers
{
    public class HomeController : ApiController
    {
        public IEnumerable<string> GetCursos()
        {
            return DbDados.ListarCursos();
        }

        [Route("listaProdutos")]
        public IEnumerable<Produto> GetProdutos()
        {
            return DbDados.ListarProdutos();
        }
    }
}
```

O atributo **Route** permite especificar uma rota para o action que o utilizar. No nosso caso, somente podemos consumir a lista de produtos através desta nova rota: **listaProdutos**. Vamos ver o novo resultado:



Apesar de termos retornado uma coleção em ambos os métodos, é possível que ele retorne quaisquer tipos válidos no C#. Seu resultado será avaliado pela aplicação que consome-lo.

6.3.4. Testando o método GET com parâmetros

Nos métodos que definimos até agora no controller, nenhum parâmetro foi informado. Vamos agora definir novos actions capazes de buscar um curso por partes do nome e um produto pelo seu Id. Aqui também vale a regra de eliminação de ambiguidade, ou seja, em um dos actions deveremos configurar sua rota:

```
using Capitulo06.WebAPI.Dados;
using Capitulo06.WebAPI.Models;
using System.Collections.Generic;
using System.Web.Http;

namespace Capitulo06.WebAPI.Controllers
{
    public class HomeController : ApiController
    {
        public IEnumerable<string> GetCursos()
        {
            return DbDados.ListarCursos();
        }

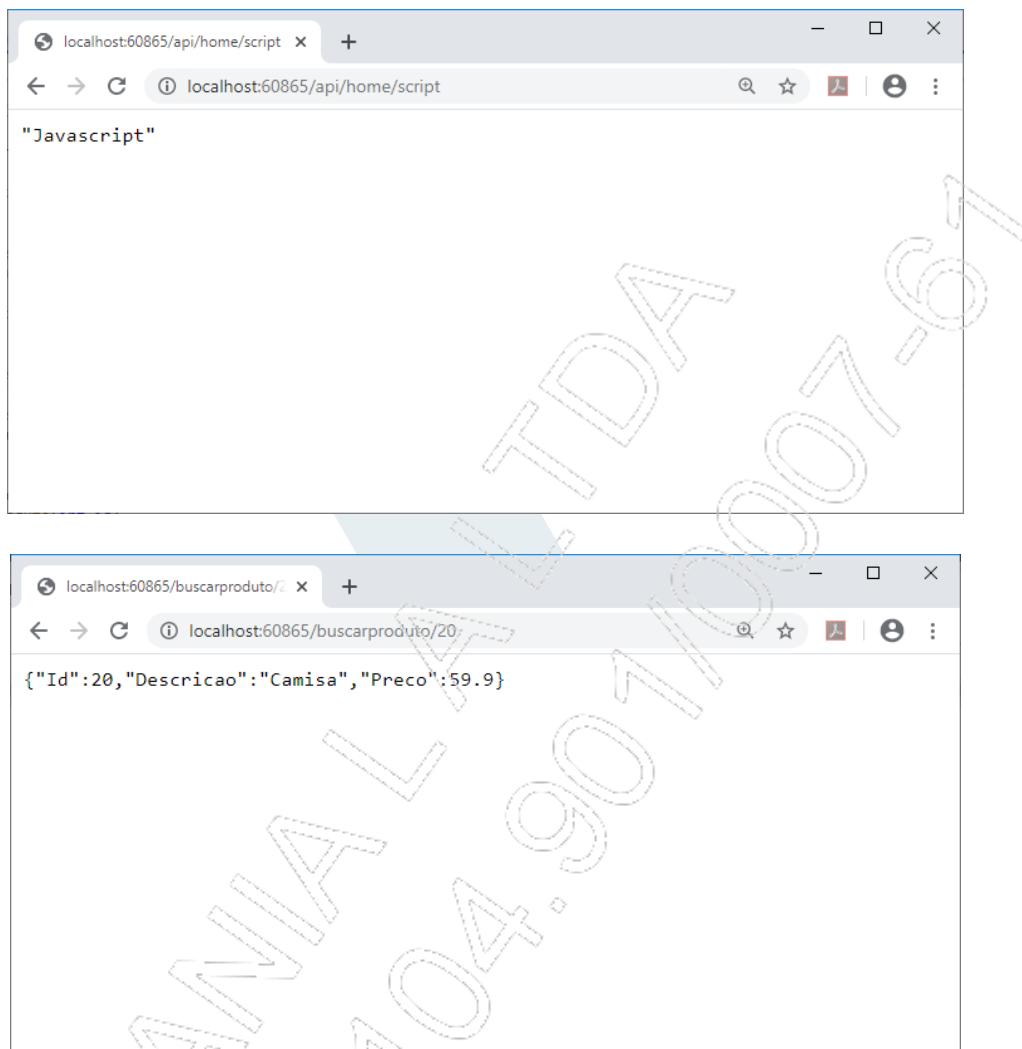
        [Route("listaProdutos")]
        public IEnumerable<Produto> GetProdutos()
        {
            return DbDados.ListarProdutos();
        }

        public string GetCurso(string id)
        {
            List<string> cursos = (List<string>)DbDados.ListarCursos();
            return cursos.Find(p => p.Contains(id));
        }

        [Route("buscarProduto/{id}")]
        public Produto GetProduto(int id)
        {
            List<Produto> produtos = (List<Produto>)DbDados.ListarProdutos();
            return produtos.Find(p => p.Id == id);
        }
    }
}
```

Os parâmetros em ambos os casos devem ter o mesmo nome especificado na rota, ou seja, id. No caso da rota **buscarProduto/{id}**, o valor entre chaves deve coincidir com o parâmetro do action e da rota. Se não for utilizado o nome especificado na rota, a chamada deverá usar queryString, que não é convencional em serviços REST.

Para testar, vamos buscar um curso que contenha a palavra "script" e o produto com Id = 20, respectivamente:



6.3.5. Definindo actions para os verbos HTTP POST, PUT e DELETE

Para testar os métodos POST, PUT e DELETE, é preciso criar uma requisição HTTP com informações no corpo, mas não é possível fazer isso digitando uma URL no browser. Existem muitos programas para esse tipo de tarefa. Esses programas são chamados de **REST Client**. Um bastante familiar é o **Postman**, mas outros disponíveis no mercado fazem um trabalho semelhante.

No nosso projeto, vamos definir os actions necessários para incluir, alterar e remover um produto. Manteremos as convenções de nomes para estes três métodos. Em seguida, criaremos uma aplicação para consumir nosso serviço com as quatro operações.

- Adicionando métodos na classe **DbDados**:

Vamos simular a inclusão, alteração e remoção de produtos em memória, usando para isso a coleção de produtos disponível nesta classe:

```
using Capitulo06.WebAPI.Models;
using System.Collections.Generic;

namespace Capitulo06.WebAPI.Dados
{
    public class DbDados
    {
        public static List<Produto> produtos = new List<Produto>()
        {
            new Produto() { Id = 10, Descricao = "Mouse", Preco = 40.00 },
            new Produto() { Id = 20, Descricao = "Camisa", Preco = 59.90 },
            new Produto() { Id = 30, Descricao = "Bike", Preco = 450.00 }
        };

        public static IEnumerable<string> ListarCursos()
        {
            return new List<string>()
            {
                "C#", "PHP", "Javascript", "SQL Server"
            };
        }

        public static IEnumerable<Produto> ListarProdutos()
        {
            return produtos;
        }

        public static Produto IncluirProduto(Produto produto)
        {
            produtos.Add(produto);
            return produto;
        }

        public static Produto AlterarProduto(Produto produto)
        {
            var prod = produtos.Find(p => p.Id == produto.Id);
            if(prod != null)
            {
                prod.Descricao = produto.Descricao;
                prod.Preco = produto.Preco;
            }
            return prod;
        }

        public static Produto RemoverProduto(int id)
        {
            var prod = produtos.Find(p => p.Id == id);
            if (prod != null)
            {
                produtos.Remove(prod);
            }
            return prod;
        }
    }
}
```

- Adicionando actions no controller **Home**:

```
using Capítulo06.WebAPI.Dados;
using Capítulo06.WebAPI.Models;
using System.Collections.Generic;
using System.Web.Http;

namespace Capítulo06.WebAPI.Controllers
{
    public class HomeController : ApiController
    {
        public IEnumerable<string> GetCursos()
        {
            return DbDados.ListarCursos();
        }

        [Route("listaProdutos")]
        public IEnumerable<Produto> GetProdutos()
        {
            return DbDados.ListarProdutos();
        }

        public string GetCurso(string id)
        {
            List<string> cursos = (List<string>)DbDados.ListarCursos();
            return cursos.Find(p => p.Contains(id));
        }

        [Route("buscarProduto/{id}")]
        public Produto GetProduto(int id)
        {
            List<Produto> produtos = (List<Produto>)DbDados.ListarProdutos();
            return produtos.Find(p => p.Id == id);
        }

        //HTTP POST
        public Produto PostProduto(Produto produto)
        {
            return DbDados.IncluirProduto(produto);
        }

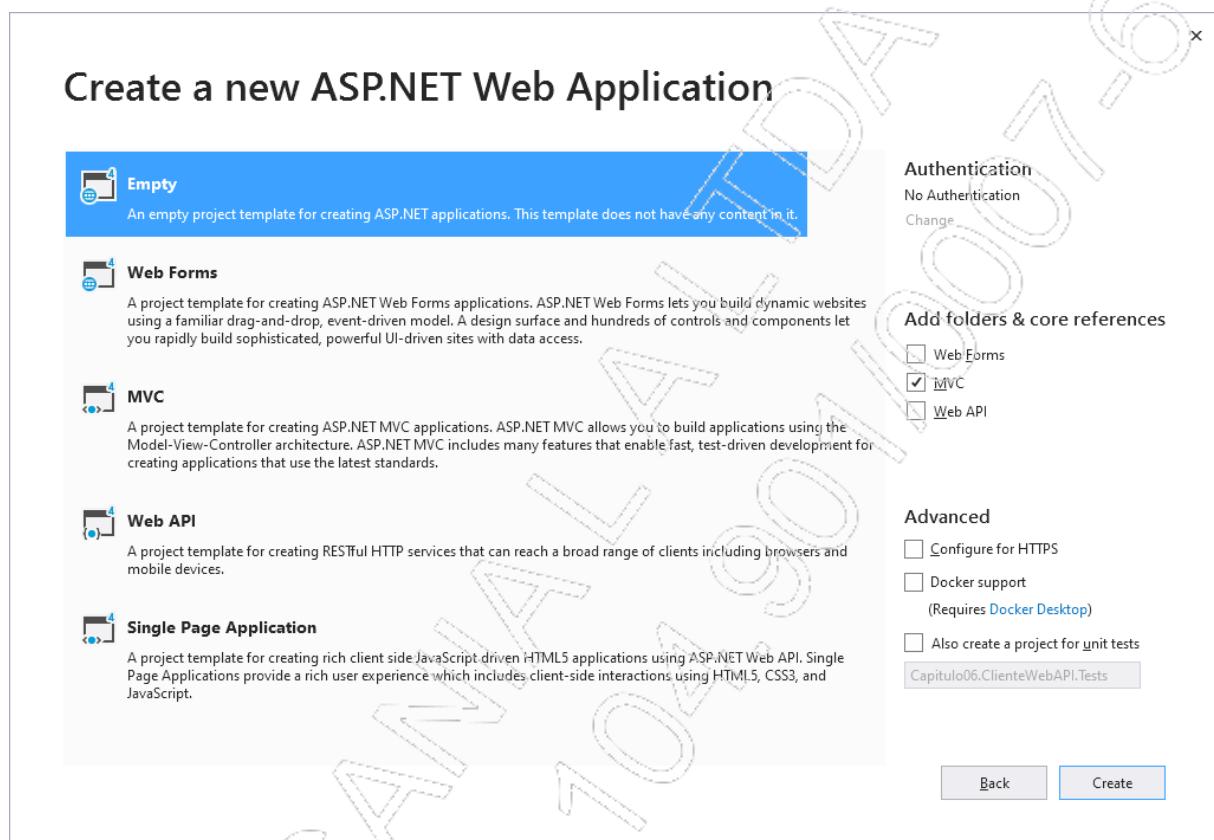
        //HTTP PUT
        public Produto PutProduto(Produto produto)
        {
            return DbDados.AlterarProduto(produto);
        }

        //HTTP DELETE
        public Produto DeleteProduto(int id)
        {
            return DbDados.RemoverProduto(id);
        }
    }
}
```

6.4. Criando um projeto para consumir o serviço

Já estamos prontos para definir uma aplicação capaz de consumir nosso Webservice. Vamos definir um novo projeto MVC para esta finalidade.

Usando o modelo **ASP.NET Web Application (.NET Framework)**, crie um projeto MVC vazio chamado **Capítulo06.ClienteWebAPI**.



Neste projeto, adicione uma classe chamada **Produto** na pasta **Models**, com as mesmas propriedades definidas na classe **Produto** do projeto Web API.

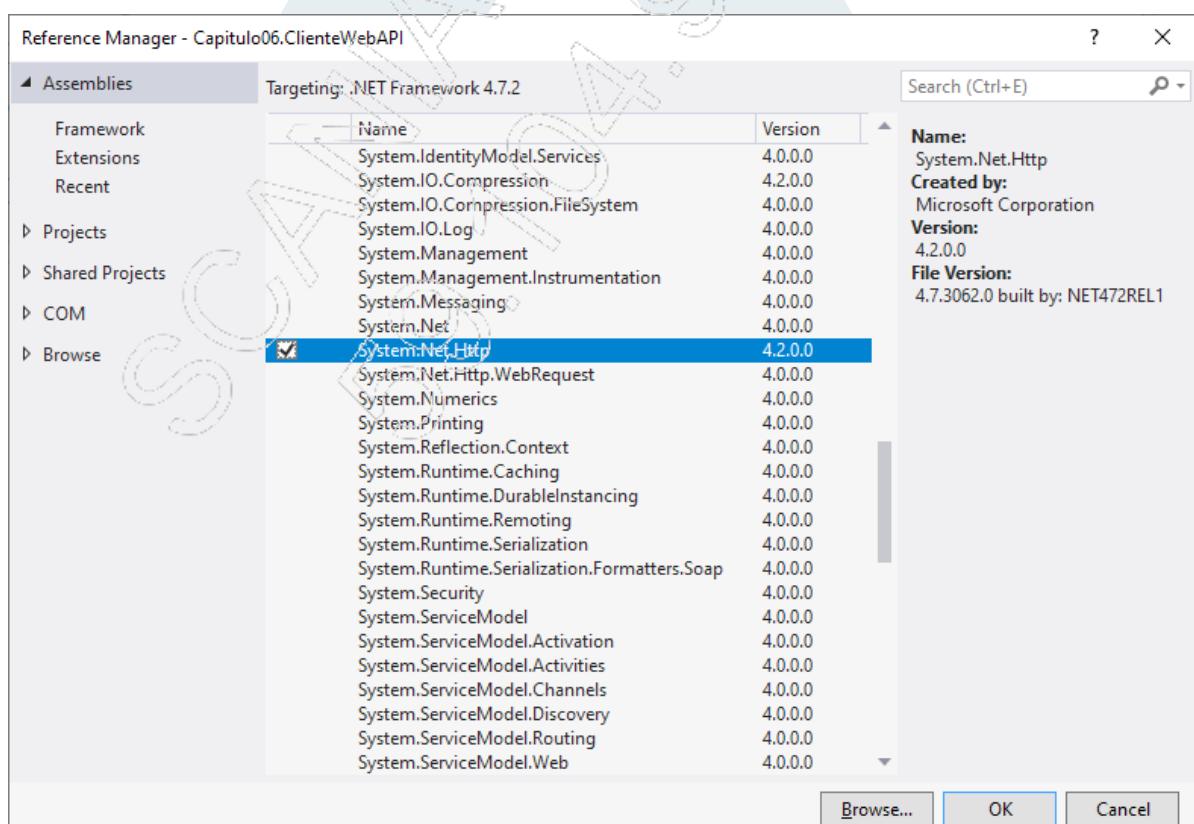
É importante que o cliente e o serviço se comuniquem através de propriedades conhecidas por ambos os lados. As classes não precisam ser as mesmas, mas o conteúdo JSON retornado e/ou enviado para o serviço deve possuir informações conhecidas pela aplicação.

```
namespace Capitulo06.ClienteWebAPI.Models
{
    public class Produto
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
    }
}
```

Inclua também, no projeto cliente, um controller chamado **Cliente** (classe **ClienteController**). Este controller terá os seguintes actions:

- **Index**: Terá itens de menu para as demais operações;
- **Incluir**: Será escrito nas versões GET e POST. Ele permitirá incluir um novo produto, cujas informações serão fornecidas em um formulário;
- **Listar**: Usado para apresentar a lista dos produtos;
- **Editar**: Action que terá os dados do produto selecionado na lista para alteração. Também terá as versões GET e POST;
- **Remover**: Será usado para remover um produto selecionado na lista. Também terá as versões GET e POST.

Além dos actions, deveremos ter no controller um objeto do tipo `HttpClient`, disponível no namespace **System.Net.Http**, cuja referência deverá ser adicionada ao projeto. Para adicioná-la, clique com o botão direito sobre o item **References**, opção **Add Reference**:



Com esta nova referência, inclua o seguinte código no controller:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Web.Mvc;

namespace Capitulo06.ClienteWebAPI.Controllers
{
    public class ClienteController : Controller
    {
        HttpClient client;

        public ClienteController()
        {
            if(client == null)
            {
                client = new HttpClient();
                client.BaseAddress = new Uri("http://localhost:60865/");
                client.DefaultRequestHeaders.Accept.Add(new
                    MediaTypeWithQualityHeaderValue("application/json"));
            }
        }

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

No construtor, definimos uma variável chamada **client**, do tipo **HttpClient**. Ainda no construtor, nós a instanciamos, definimos o endereço base do serviço a ser consumido e a informação do cabeçalho da requisição, especificando o MIME TYPE da comunicação como sendo **application/json**.

É importante observar que o valor definido para a propriedade **BaseAddress será diferente em projetos diferentes, uma vez que o número da porta atribuído pelo IIS Express é diferente para cada um.**

Com esta configuração concluída, vamos iniciar pelo action **Incluir**.

- Incluindo um novo produto

No controller **Home**, adicione o action **Incluir**:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Web.Mvc;

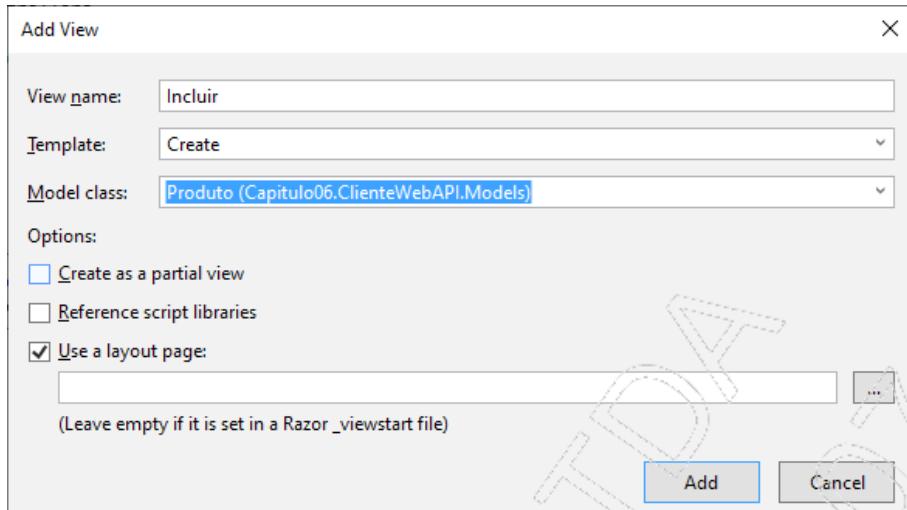
namespace Capitulo06.ClienteWebAPI.Controllers
{
    public class ClienteController : Controller
    {
        HttpClient client;

        public ClienteController()
        {
            if(client == null)
            {
                client = new HttpClient();
                client.BaseAddress = new Uri("http://localhost:60865/");
                client.DefaultRequestHeaders.Accept.Add(new
                    MediaTypeWithQualityHeaderValue("application/json"));
            }
        }

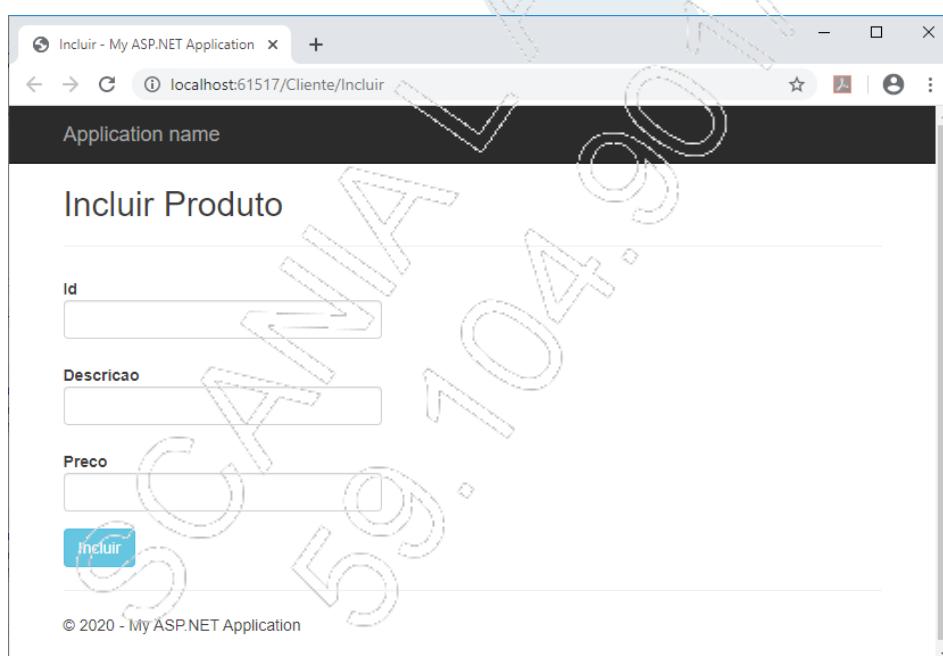
        public ActionResult Index()
        {
            return View();
        }

        //incluindo um novo produto
        [HttpGet]
        public ActionResult Incluir()
        {
            return View();
        }
    }
}
```

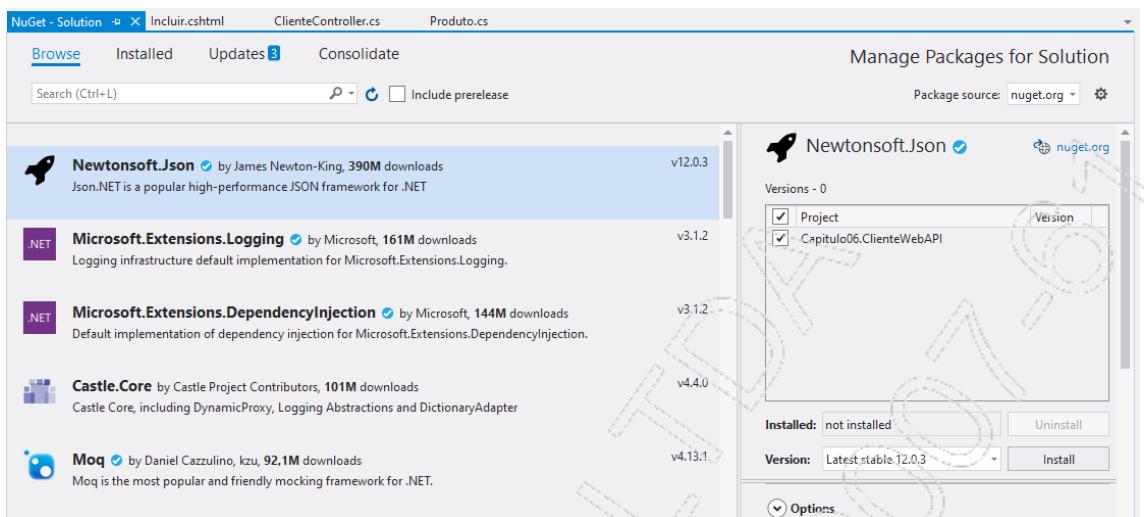
Em seguida, adicione uma nova view, com o template **Create** e model **Produto**:



Por conta do nome da propriedade **Id** no modelo, talvez tenhamos que adicionar o campo para fornecermos seu valor, já que não estamos usando nenhuma base de dados.



Vamos escrever a versão POST do action **Incluir**. Usaremos uma classe chamada **JsonConvert**, presente no namespace **Newtonsoft.Json**. Este namespace deverá ser adicionado através do NuGet:



O código que permite inserir um novo produto é dado a seguir:

```
[HttpPost]
public async Task<ActionResult> Incluir(Produto produto)
{
    try
    {
        string json = JsonConvert.SerializeObject(produto);

        HttpContent content = new StringContent(json,
Encoding.Unicode, "application/json");

        var response = await client.PostAsync("api/home", content);

        if (response.IsSuccessStatusCode)
        {
            return RedirectToAction("Listar");
        }
        else
        {
            var mensagem = response.ReasonPhrase;
            throw new Exception(mensagem);
        }
    }
    catch (Exception ex)
    {
        ViewBag.Erro = ex.Message;
        return View("_Erro");
    }
}
```

Neste método, temos as seguintes instruções principais:

```
string json = JsonConvert.SerializeObject(produto);
```

O objeto passado como parâmetro para o action está sendo convertido para o formato JSON.

```
HttpContent content = new StringContent(json,  
Encoding.Unicode, "application/json");
```

Um objeto contendo o fluxo de bytes a ser enviado para o serviço esta sendo criado. Este objeto é referenciado pela variável **content**.

```
var response = await client.PostAsync("api/home", content);
```

É nesta instrução que o fluxo de bytes é enviado para o serviço. Observe que o método é assíncrono, e que o comando **await** está sendo aplicado. É por isso que nosso action foi definido como **async**, com **ActionResult** tratado como uma tarefa (**Task**).

Na sequência, se o processo for bem sucedido, o usuário é encaminhado para a lista de produtos (a ser desenvolvida). Caso algum erro ocorra, a requisição é transferida para **_Erro.cshtml**.

Para testar a inclusão, incluiremos os actions **Listar** e **_Erro**. **Listar** ainda não terá nenhum conteúdo. **_Erro.cshtml** será colocado na pasta **Views/Shared**.

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

- Conteúdo da view **Listar.cshtml**:

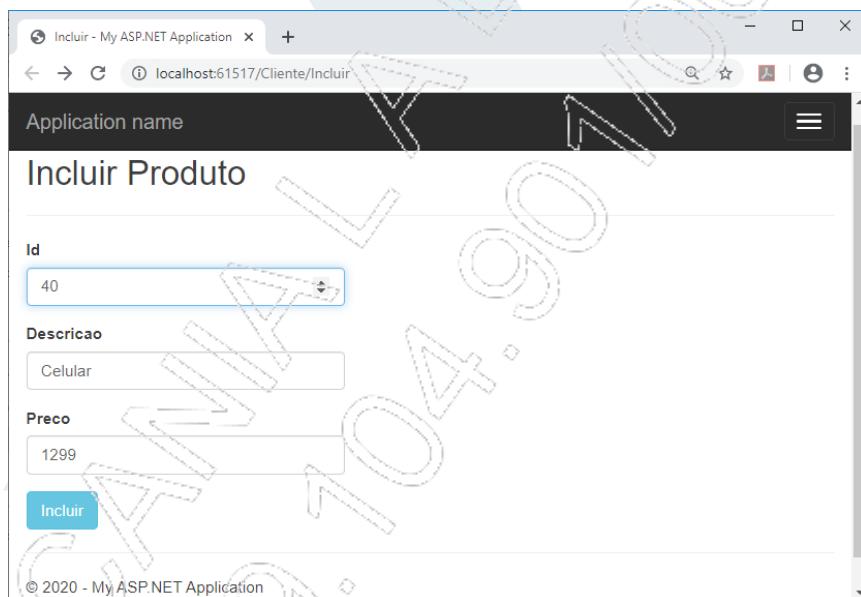
```
@{  
    ViewBag.Title = "Listar";  
}  
  
<h2>Listar - teste</h2>
```

- Conteúdo da view `_Erro.cshtml`:

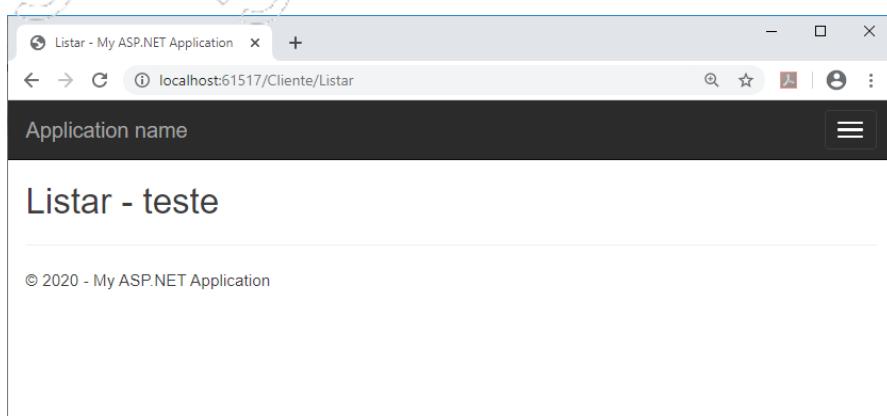
```
@{  
    ViewBag.Title = "Erro";  
}  
  
<h2 class="text-danger">Ocorreu o seguinte erro:</h2>  
  
<div class="alert alert-danger">  
    @ViewBag.Erro  
</div>
```

Para testar a aplicação, deveremos ter o projeto referente ao serviço em execução. Após o teste da inclusão, veremos o resultado no browser (enquanto a listagem não está concluída).

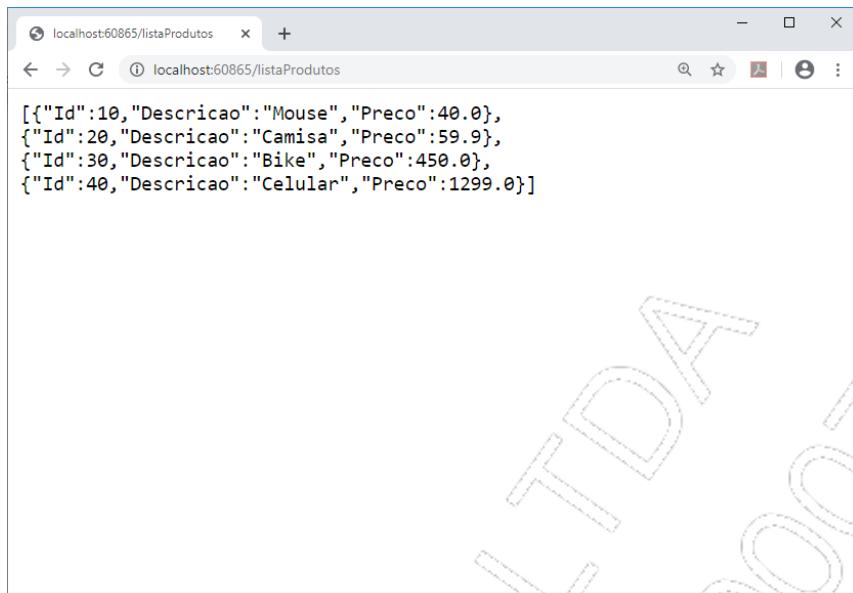
Para testar, execute a aplicação cliente e forneça os dados adiante:



Uma vez adicionados:



O resultado da consulta ao Webservice será algo semelhante ao apresentado a seguir:

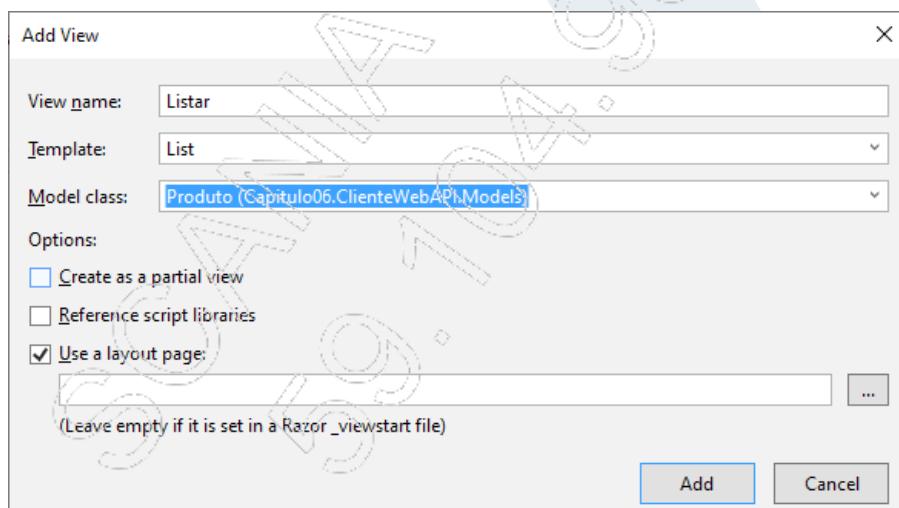


A screenshot of a web browser window titled "localhost:60865/listaProdutos". The URL in the address bar is "localhost:60865/listaProdutos". The page content displays a JSON array of product objects:

```
[{"Id":10,"Descricao":"Mouse","Preco":40.0}, {"Id":20,"Descricao":"Camisa","Preco":59.9}, {"Id":30,"Descricao":"Bike","Preco":450.0}, {"Id":40,"Descricao":"Celular","Preco":1299.0}]
```

- **Listando os produtos**

O processo de listagem equivale a buscarmos os dados através da URL, só que trazendo-os para a aplicação. Para ilustrar, vamos remover a view **Listar.cshtml** e adicioná-la com o template **List** e model **Produto**:



Vamos manter os links para **Editar** e **Remover** o produto.

```
@model IEnumerable<Capitulo06.ClienteWebAPI.Models.Produto>

{@
    ViewBag.Title = "Listar";
}

<h2>Listar Produtos</h2>

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Descricao)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Preco)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Descricao)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Preco)
        </td>
        <td>
            @Html.ActionLink("Editar", "Editar", new { id=item.Id }) |
            @Html.ActionLink("Remover", "Remover", new { id=item.Id })
        </td>
    </tr>
}
</table>
```

Na sequência, adicionaremos o código no action **Listar** para buscar a lista de produtos no Webservice:

```
public async Task<ActionResult> Listar()
{
    try
    {
        var response = await client.GetAsync("listaProdutos");
        if (response.IsSuccessStatusCode)
        {
            var resultado = await response.Content.ReadAsStringAsync();
            var lista = JsonConvert
                .DeserializeObject<Produto[]>(resultado).ToList();

            return View(lista);
        }
        else
        {
            throw new Exception(response.ReasonPhrase);
        }
    }
    catch (Exception ex)
    {
        ViewBag.Erro = ex.Message;
        return View("_Erro");
    }
}
```

Na listagem, o processo é inverso ao usado na inclusão. Observe que na instrução...

```
var response = await client.GetAsync("listaProdutos");
```

...usamos a rota personalizada no action correspondente no Webservice. Em seguida, o conteúdo foi lido do fluxo de bytes e convertido em uma string com os dados em JSON:

```
var resultado = await response.Content.ReadAsStringAsync();
```

E, logo em seguida, este resultado foi repassado para um array de objetos **Produto**, já que o array é o formato geral de coleções nos Webservices REST. O resultado obtido foi convertido para **List<Produto>** através do método **ToList**.

```
var lista = JsonConvert
    .DeserializeObject<Produto[]>(resultado).ToList();
return View(lista);
```

O resultado desta execução é apresentado a seguir:

Descrição	Preço	
Mouse	40	Editar Remover
Camisa	59,9	Editar Remover
Bike	450	Editar Remover

Quando um novo produto for incluído, a listagem já o apresentará também.

- **Alterando um produto**

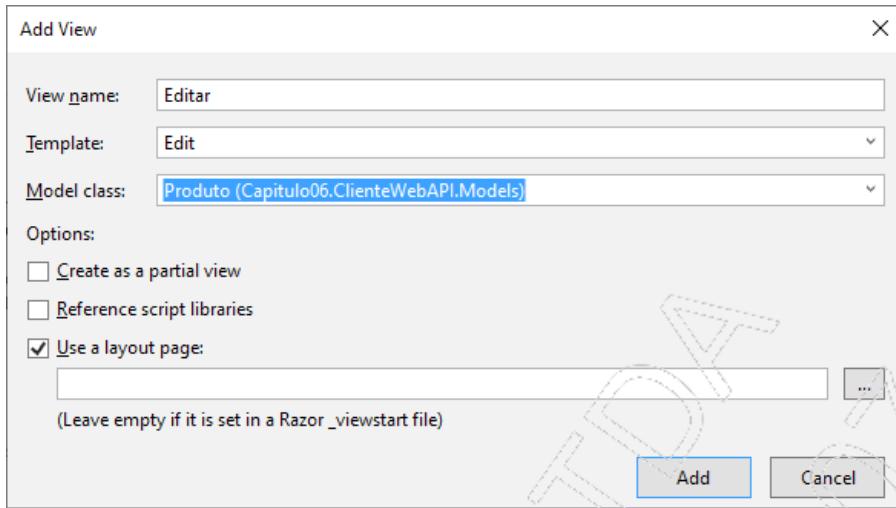
A alteração ocorre de forma análoga à inclusão. A principal diferença está no método de acesso ao Webservice: em vez de **PostAsync**, chamamos **PutAsync**.

Para ilustrar a inclusão, adicionaremos o action **Editar**, versão GET e POST. Na versão GET, buscaremos o produto com o **id** informado como parâmetro, e no método POST executaremos a alteração conforme especificado no Webservice. Vamos começar por implementar o método GET:

```
public async Task<ActionResult> Editar(int id)
{
    try
    {
        var response = await client.GetAsync("buscarProduto/" + id);
        if (response.IsSuccessStatusCode)
        {
            var resultado = await response.Content.ReadAsStringAsync();
            var produto = JsonConvert.DeserializeObject<Produto>(resultado);

            return View(produto);
        }
        else
        {
            throw new Exception(response.ReasonPhrase);
        }
    }
    catch (Exception ex)
    {
        ViewBag.Erro = ex.Message;
        return View("_Erro");
    }
}
```

A view a ser criada usará o template **Edit** com o model **Produto**:



Na listagem, selecionamos um produto através do link **Editar** e visualizamos seus dados para proceder com a alteração.

Para testar, vamos implementar a versão POST deste action:

```
[HttpPost]
public async Task<ActionResult> Editar(Produto produto)
{
    try
    {
        string json = JsonConvert.SerializeObject(produto);

        HttpContent content = new StringContent(json,
            Encoding.Unicode, "application/json");

        var response = await client.PutAsync("api/home", content);

        if (response.IsSuccessStatusCode)
        {
            return RedirectToAction("Listar");
        }
        else
        {
            var mensagem = response.ReasonPhrase;
            throw new Exception(mensagem);
        }
    }
    catch (Exception ex)
    {
        ViewBag.Erro = ex.Message;
        return View("_Erro");
    }
}
```

Agora podemos executar a aplicação a partir da listagem, escolher um produto e alterar seus dados. Veremos que sua alteração será refletida na listagem resultante:

Descrição	Preço	
Mouse	40	Editar Remover
Camisa	59,9	Editar Remover
Bike	450	Editar Remover

Descrição
Mouse Optico

Preço
40
Save

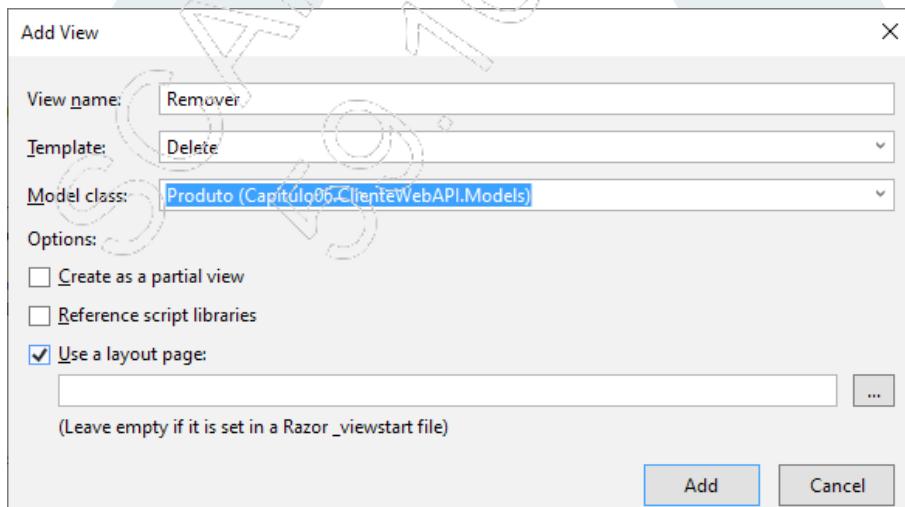
Descrição	Preço	
Mouse Optico	40	Editar Remover
Camisa	59,9	Editar Remover
Bike	450	Editar Remover

- Removendo um Produto

Para fechar o ciclo, removeremos um produto. Para tanto, deveremos implementar o action **Remover** com as versões GET e POST. A versão GET servirá para solicitarmos uma confirmação de remoção, antes de procedê-la em definitivo. O conteúdo do action **Remover** através do GET é idêntico ao implementado no **Editar**:

```
public async Task<ActionResult> Remover(int id)
{
    try
    {
        var response = await client.GetAsync("buscarProduto/" + id);
        if (response.IsSuccessStatusCode)
        {
            var resultado = await response.Content.ReadAsStringAsync();
            var produto = JsonConvert.DeserializeObject<Produto>(resultado);

            return View(produto);
        }
        else
        {
            throw new Exception(response.ReasonPhrase);
        }
    }
    catch (Exception ex)
    {
        ViewBag.Erro = ex.Message;
        return View("_Erro");
    }
}
```



O conteúdo da view deverá ter disponível o Id do produto, pois é o parâmetro usado para a remoção:

```
@model Capitulo06.ClienteWebAPI.Models.Produto

@{
    ViewBag.Title = "Remover";
}



## Remover um produto



### Tem certeza que deseja remover este produto?



---



<dt>
            @Html.DisplayNameFor(model => model.Descricao)
        </dt>

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

        <dt>
            @Html.DisplayNameFor(model => model.Preco)
        </dt>

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



    @using (Html.BeginForm()) {
        @Html.AntiForgeryToken()
        @Html.HiddenFor(model => model.Id)
        <div class="form-actions no-color">
            <input type="submit" value="Sim, remover" class="btn btn-danger" /> |
            @Html.ActionLink("Não, retornar para a lista", "Listar")
        </div>
    }


```

A versão POST do action **Remover** pode ser definida como:

```
[HttpPost]
public async Task<ActionResult> Remover(Produto produto)
{
    try
    {
        var response = await client.DeleteAsync("api/home/" + produto.
Id);
        if (response.IsSuccessStatusCode)
        {
            return RedirectToAction("Listar");
        }
        else
        {
            throw new Exception(response.ReasonPhrase);
        }
    }
    catch (Exception ex)
    {
        ViewBag.Erro = ex.Message;
        return View("_Erro");
    }
}
```

Para testar, vamos executar o fluxo completo, desde a listagem, a seleção do produto a ser removido, a confirmação e o retorno à listagem:

The figure consists of three vertically stacked screenshots of a web application. The top screenshot shows a list of products with a 'Remover' link next to each entry. The middle screenshot shows a confirmation dialog for removing a product. The bottom screenshot shows the updated product list after one item has been removed.

Screenshot 1: Listar Produtos

Descrição	Preço	Ações
Mouse Óptico	40	Editar Remover
Camisa	59,9	Editar Remover
Bike	450	Editar Remover

Screenshot 2: Remover um produto

Tem certeza que deseja remover este produto?

Descrição	Preço
Mouse Óptico	40

[Sim, remover](#) [Não, retornar para a lista](#)

Screenshot 3: Listar Produtos

Descrição	Preço	Ações
Camisa	59,9	Editar Remover
Bike	450	Editar Remover

Em um cenário real, o Webservice deve implementar as alterações em um banco de dados. Quando elaborarmos o projeto referente a este capítulo, teremos a oportunidade de aplicar algumas validações, além de acesso a uma base de dados consistente.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

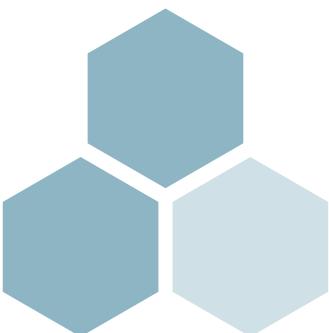
- REST é um conjunto de princípios de desenvolvimento. O protocolo HTTP e o ambiente Web atendem totalmente aos princípios REST;
- Web API é um framework da Microsoft para criar aplicações RESTful;
- Os principais verbos de uma requisição são **GET**, **POST**, **PUT** e **DELETE**;
- A classe que é chamada quando uma requisição é recebida pelo servidor é chamada de **Controller**;
- O framework Web API usa o recurso **routes** para definir qual método vai ser executado quando chega uma requisição;
- A combinação de verbo (GET, POST, PUT, DELETE), URL e cabeçalhos define como uma aplicação cliente consegue acesso a um recurso disponibilizado por um servidor que siga os padrões REST.



6

Serviços - Web API

Teste seus conhecimentos



1. O que é REST?

- a) É um protocolo de comunicação para enviar dados pela Internet.
- b) É uma linguagem de programação para criar serviços.
- c) É um conjunto de princípios para criação de serviços.
- d) É um framework da Microsoft para desenvolver serviços.
- e) É um padrão de arquitetura para criação de Web sites.

2. Segundo os princípios REST, quais características deve ter um servidor?

- a) O servidor deve usar o .NET Framework.
- b) O servidor deve armazenar informações das solicitações para identificar quando uma chamada vem do mesmo cliente.
- c) O servidor não deve armazenar dados das solicitações. O cliente deve enviar, em cada solicitação, toda informação necessária para realizar a tarefa que está sendo solicitada.
- d) O servidor nunca deve enviar para o cliente qualquer informação além de dados em formato XML ou JSON.
- e) Nenhuma das alternativas anteriores está correta.

3. Sobre a Web API, qual a alternativa correta?

- a) É um framework da Microsoft para criar e consumir serviços REST.
- b) É uma linguagem de programação para criar aplicativos.
- c) É criada em formato de dados parecido com JSON.
- d) É uma extensão do Visual Studio para trabalhar com JavaScript.
- e) Nenhuma das alternativas anteriores está correta.

4. Que tipo de aplicativo pode consumir um serviço REST criado com a Web API?

- a) Windows Form
- b) Web Form
- c) MVC
- d) Console
- e) Todas as alternativas anteriores estão corretas.

5. Qual classe do .NET Framework é utilizada em uma aplicação para estabelecer conexão e usar um serviço REST?

- a) HttpClient
- b) Task
- c) ServiceContract
- d) MediaTypeWithQualityHeaderValue
- e) RestClient



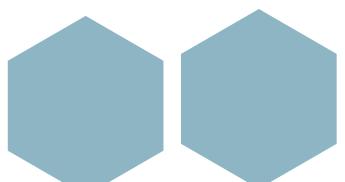
6

Serviços - Web API



Mãos à obra!

SCALING
50.
SCALING
40.
SCALING
30.
SCALING
20.
SCALING
10.
SCALING
0.



Objetivos:

Neste laboratório, desenvolveremos um serviço REST usando o Web API. Trata-se de um novo projeto onde simularemos uma administradora de cartões de crédito. O objetivo é receber os pagamentos dos produtos adquiridos no nosso projeto MVC, cujo desenvolvimento se estendeu até o Capítulo 5.

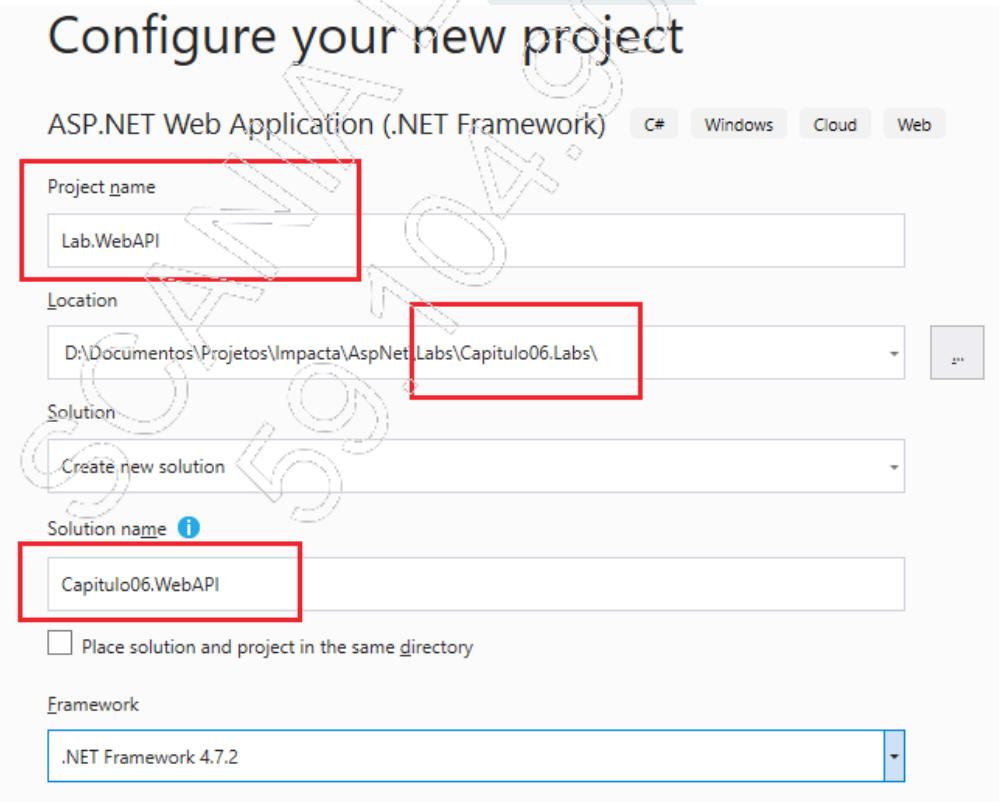
Na primeira parte, criaremos o serviço. Na segunda parte, desenvolveremos o cliente do nosso serviço, implementando o recurso de pagamento na aplicação.

Aproveitaremos o projeto do serviço para apresentar o procedimento **Code First** para a criação do banco de dados.

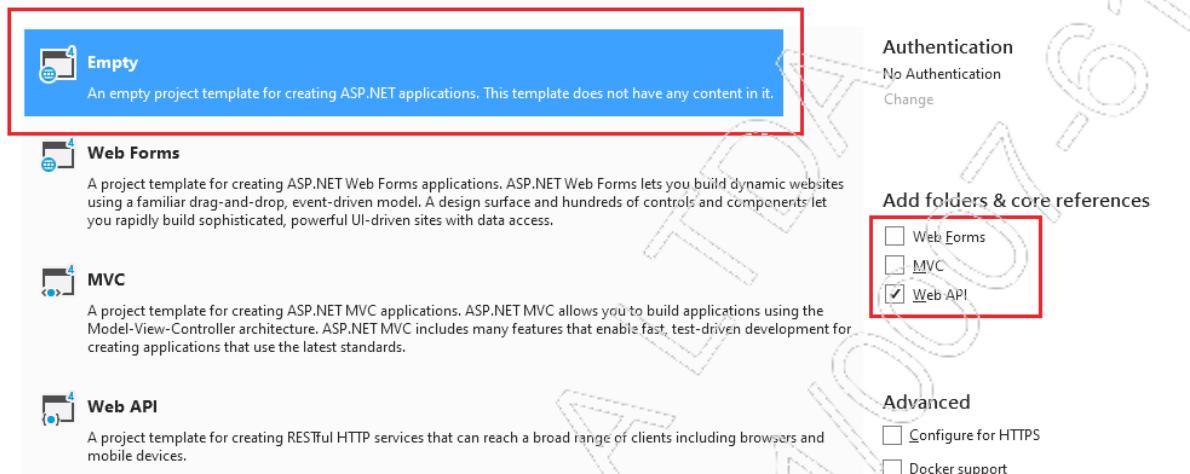
Laboratório 1

A - Desenvolvimento do serviço Web API

1. Na pasta **Labs**, crie uma pasta chamada **Capítulo06.Labs**;
2. Nesta nova pasta, crie um projeto chamado **Lab.WebAPI** em uma solution chamada **Capítulo06.WebAPI**:



Create a new ASP.NET Web Application



3. Com o projeto criado, vamos iniciar implementando a camada de acesso a dados com o **Entity Framework, Code First**. Por meio do NuGet, adicione a referência ao **Entity Framework**:

Package	Version	Description
Microsoft.EntityFrameworkCore	v3.1.2	Entity Framework Core is a lightweight and extensible version of the popular Entity Framework data access technology.
Microsoft.EntityFrameworkCore.Relational	v3.1.2	Shared Entity Framework Core components for relational database providers.
EntityFramework	v6.4.0	Entity Framework 6 (EF6) is a tried and tested object-relational mapper for .NET with many years of feature development and stabilization.
Microsoft.EntityFrameworkCore.Abstractions	v3.1.2	Provides abstractions and attributes that are used to configure Entity Framework Core
Microsoft.EntityFrameworkCore.Design	v3.1.2	Shared design-time components for Entity Framework Core tools.

4. Na pasta **Models**, adicione as seguintes classes (entidades):

```
namespace Lab.WebAPI.Models
{
    public class Cartao
    {
        public int Id { get; set; }
        public string NumeroCartao { get; set; }
        public double Limite { get; set; }
    }
}

namespace Lab.WebAPI.Models
{
    public class Fatura
    {
        public int Id { get; set; }
        public string NumeroCartao { get; set; }
        public string NumeroPedido { get; set; }
        public double Valor { get; set; }
        public int Status { get; set; }
    }
}
```

5. No arquivo **Web.config**, dentro do elemento <**configuration**>, adicione o elemento referente à string de conexão:

```
<connectionStrings>
    <add name="PagamentosConnection"
        connectionString="Integrated Security=SSPI;Persist Security
Info=False;Initial Catalog=DB_pagamentos;Data Source=.\SQLEXPRESS"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

6. Na pasta **Models**, adicione a classe **PagamentosContext**. Esta classe terá as definições do banco de dados a ser criado e, posteriormente, a ser acessado no serviço. Sobrescrevemos o método **OnModelCreating** para especificar os detalhes de criação do banco:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Web;

namespace Lab.WebAPI.Models
{
    public class PagamentosContext : DbContext
    {
        public PagamentosContext():
            base("PagamentosConnection")
        {
        }

        public DbSet<Cartao> Cartoes { get; set; }
        public DbSet<Fatura> Faturas { get; set; }

        protected override void OnModelCreating(
            DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<Cartao>().ToTable("TBCartoes");

            modelBuilder.Entity<Cartao>()
                .Property(p => p.NumeroCartao)
                .IsRequired()
                .HasMaxLength(16);
            modelBuilder.Entity<Cartao>()
                .Property(p => p.Limite)
                .IsRequired();

            modelBuilder.Entity<Fatura>().ToTable("TBFaturas");
            modelBuilder.Entity<Fatura>()
                .Property(p => p.NumeroCartao)
                .IsRequired()
                .HasMaxLength(16);
            modelBuilder.Entity<Fatura>()
                .Property(p => p.NumeroPedido)
                .IsRequired()
                .HasMaxLength(20);
            modelBuilder.Entity<Fatura>()
                .Property(p => p.Valor)
                .IsRequired();
        }
    }
}
```

O banco de dados será criado quando fizermos seu primeiro acesso;

7. Crie uma enumeração representando os estados da fatura. Estes estados descreverão possíveis erros na inclusão de um pagamento, como:

- Saldo indisponível no cartão;
- Pedido já pago;
- Cartão inexistente;
- Pagamento OK.

Chame essa enumeração de **StatusPagamento**. Crie-a em uma nova pasta chamada **Enumerations**:

```
namespace Lab.WebAPI.Enumerations
{
    public enum StatusPagamento
    {
        SALDO_INDISPONIVEL,
        PEDIDO_JA_PAGO,
        CARTAO_INEXISTENTE,
        PAGAMENTO_OK
    }
}
```

8. Defina uma classe chamada **PagamentosDao** em uma nova pasta chamada **Data**. Nesta classe, definiremos os métodos responsáveis pela inclusão do pagamento, pela listagem das faturas, entre outros que puderem ser consumidos pelo serviço através do controller;

```
using Lab.WebAPI.Enumerations;
using Lab.WebAPI.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace Lab.WebAPI.Data
{
    public class PagamentosDao
    {
        //método para listar todas as faturas
        public static IEnumerable<Fatura> ListarFaturas()
        {
            using(var ctx = new PagamentosContext())
            {
                return ctx.Faturas.ToList();
            }
        }
    }
}
```

```
//método para incluir um novo pagamento
//Observe o retorno deste método
public static StatusPagamento IncluirFatura(Fatura fatura)
{
    using (var ctx = new PagamentosContext())
    {
        //verificando se o cartão existe
        //Se existir, definimos uma referência a ele
        var cartao = ctx.Cartoes.FirstOrDefault(p =>
            p.NumeroCartao.Equals(fatura.NumeroCartao));

        if(cartao == null)
        {
            return StatusPagamento.CARTAO_INEXISTENTE;
        }

        //verificando se o pedido já foi pago
        var fat = ctx.Faturas.FirstOrDefault(p =>
            p.NumeroPedido.Equals(fatura.NumeroPedido));

        if(fat != null)
        {
            return StatusPagamento.PEDIDO_JA_PAGO;
        }

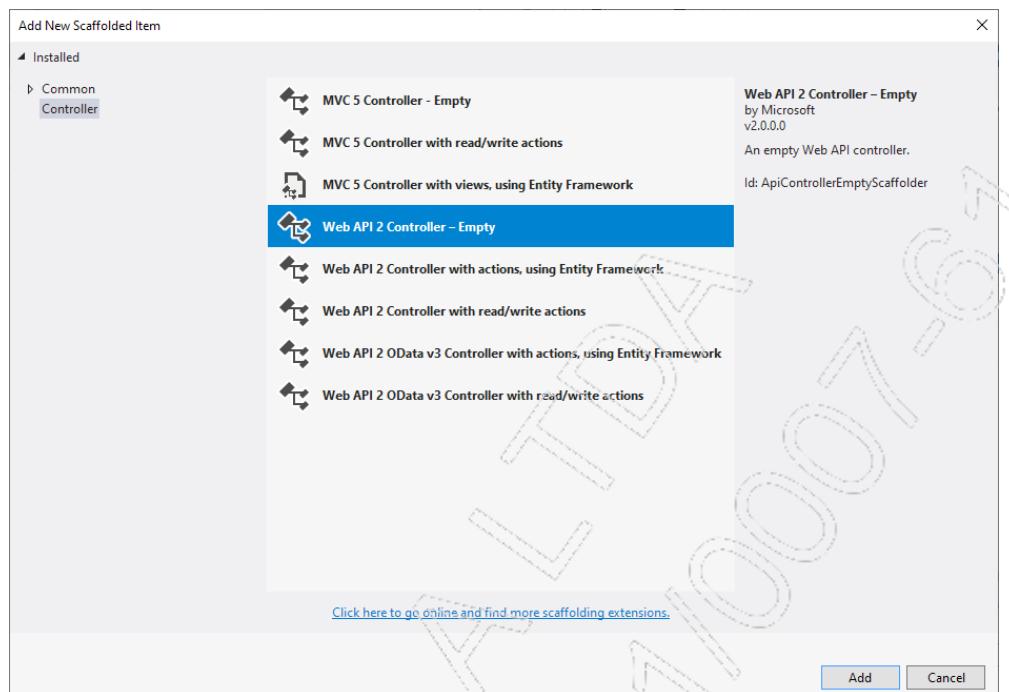
        //verificando se existe saldo no cartão, assumindo que:
        //Status = 1 -> Fatura ainda não paga (em aberto)
        //Status = 2 -> Fatura já paga (fechada)
        double total = fatura.Valor;

        var pagamentos = ctx.Faturas.Where(p =>
            p.NumeroCartao.Equals(fatura.NumeroCartao));
        if(pagamentos.Count() > 0)
        {
            total += pagamentos.Sum(s => s.Valor);
        }

        if(total > cartao.Limite)
        {
            return StatusPagamento.SALDO_INDISPONIVEL;
        }

        //Se passar pelas validações, efetuamos o pagamento
        ctx.Faturas.Add(fatura);
        ctx.SaveChanges();
        return StatusPagamento.PAGAMENTO_OK;
    }
}
```

9. Inclua um controller chamado **PagamentosController**. Desta vez, o controller é do tipo **Web API 2 Controller - Empty**:



```
using System.Web.Http;  
  
namespace Lab.WebAPI.Controllers  
{  
    public class PgamentosController : ApiController  
    {  
    }  
}
```

Neste controller, escreveremos os actions tomando como base as convenções de nomes do Web API;

10. No controller, escreva os actions **GetPagamentos** e **PostPagamentos**:

```
using Lab.WebAPI.Data;  
using Lab.WebAPI.Enumerations;  
using Lab.WebAPI.Models;  
using System;  
using System.Collections.Generic;  
using System.Net;  
using System.Net.Http;  
using System.Web.Http;  
  
namespace Lab.WebAPI.Controllers  
{  
    public class PagamentosController : ApiController  
    {
```

```
//Listando todos os pagamentos (faturas)
public IEnumerable<Fatura> GetFaturas()
{
    return PagamentosDao.ListarFaturas();
}

//Incluindo um novo pagamento
public HttpResponseMessage PostFatura(Fatura fatura)
{
    StatusPagamento status = PagamentosDao.IncluirFatura(fatura);

    if(status != StatusPagamento.PAGAMENTO_OK)
    {
        string mensagem = "";
        switch (status)
        {
            case StatusPagamento.CARTAO_INEXISTENTE:
                mensagem = "O Cartão informado não existe";
                break;
            case StatusPagamento.PEDIDO_JA_PAGO:
                mensagem = "Já foi realizado pagamento deste pedido";
                break;
            case StatusPagamento.SALDO_INDISPONIVEL:
                mensagem = "O Cartão não possui saldo disponível";
                break;
        }

        var erro = new HttpResponseMessage(HttpStatusCode.BadRequest)
        {
            Content = new StringContent("Erro no servidor"),
            ReasonPhrase = mensagem
        };

        throw new HttpResponseException(erro);
    }
    else
    {
        //cria o cabeçalho da resposta
        var response = Request.CreateResponse<Fatura>(
            HttpStatusCode.Created, fatura);

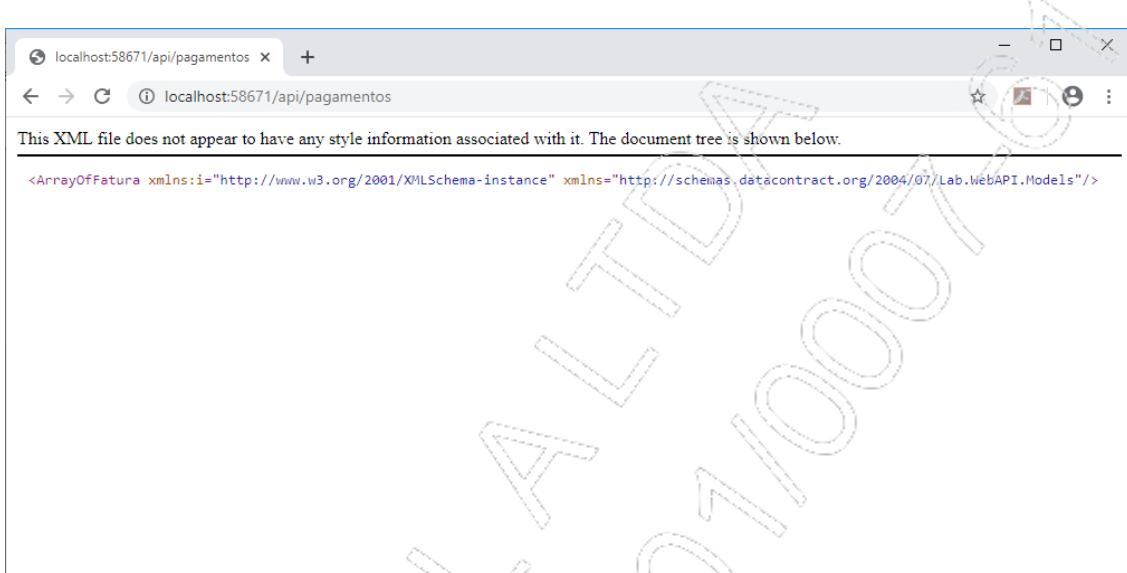
        string uri = Url.Link("DefaultApi", new { id = fatura.Id });

        response.Headers.Location = new Uri(uri);
        return response;
    }
}
```

11. Execute a aplicação e forneça a rota `api/{controller}`, conforme especificado no arquivo `WebApiConfig.cs`, pasta `App_Start`:

api/pagamentos

Pode demorar um pouco, mas é o tempo para que o banco de dados seja criado. O resultado é mostrado a seguir:



12. O resultado produzido está no formato **XML**. Nós usaremos o formato **JSON**. Para isso, devemos configurar este formato em `WebApiConfig`. Abra esta classe. Ela está na pasta `App_Start`:

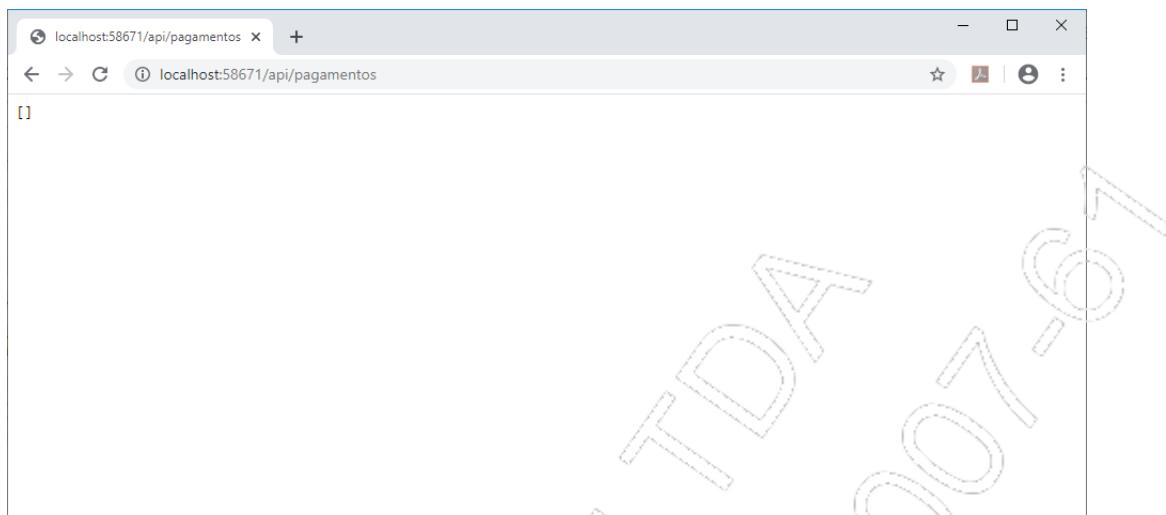
```
using System.Web.Http;

namespace Lab.WebAPI
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services
            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            config.Formatters.Remove(config.Formatters.XmlFormatter);
            config.Formatters.Add(config.Formatters.JsonFormatter);
        }
    }
}
```

13. Execute novamente. Veja que recebemos uma lista vazia, pois ainda não temos nenhuma fatura criada.



B - Desenvolvimento do cliente Web API

14. Na pasta **Capítulo06.Labs**, adicione um novo solution, vazio, chamado **Capítulo06.ClienteWebAPI**;

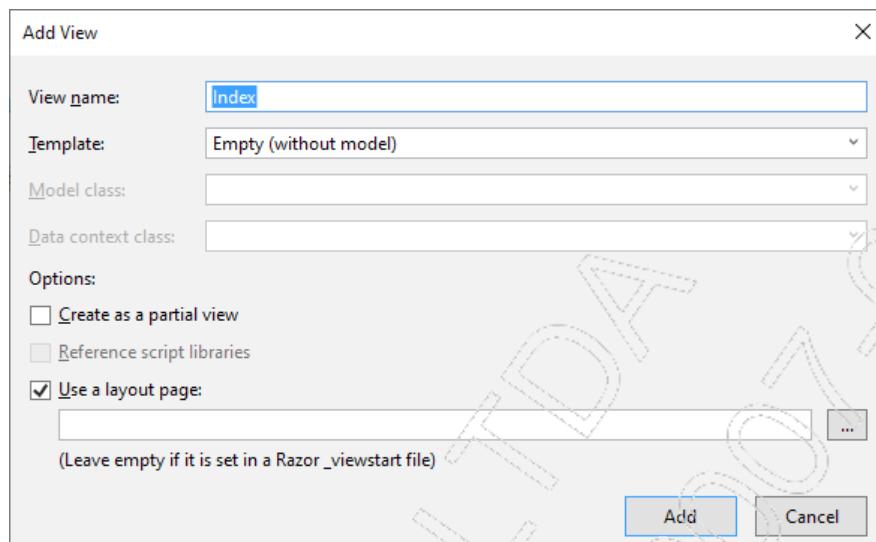
15. Copie o projeto **Lab.MVC** do solution **Capítulo05.Labs** para a pasta do seu novo solution, **Capítulo06.ClienteWebAPI**;

16. Adicione este projeto ao novo solution. Execute a aplicação para testar se está tudo OK;

17. Estando tudo OK, vamos criar um novo controller MVC chamado **FaturasController**.

```
using System.Web.Mvc;
namespace Lab.MVC.Controllers
{
    public class FaturasController : Controller
    {
        // GET: Faturas
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

18. Defina a view para o action **Index**. Esta view será a view de apresentação para os pagamentos:

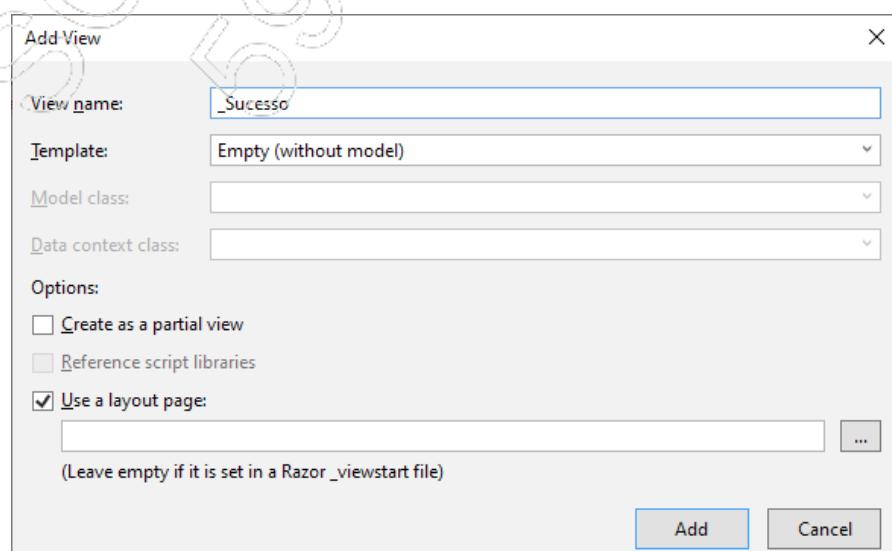


```
@{
    ViewBag.Title = "Faturas";
}

<h2>Faturas - Escolha uma opção</h2>

<ul>
    <li>@Html.ActionLink("Efetuar Pagamento", "EfetuarPagamento")</li>
    <li>@Html.ActionLink("Listar Faturas", "ListarFaturas")</li>
</ul>
```

19. Na pasta **Views/Shared**, adicione uma nova view chamada **_Sucesso**. Ela será apresentada quando o pagamento for efetuado com sucesso. A view **_Erro** continuará a ser usada em caso de erro;



```
@{  
    ViewBag.Title = "Erro";  
}  
  
<h2 class="text-success">Pagamento OK</h2>  
  
<div class="alert alert-success text-center">  
    Pagamento efetuado com sucesso!  
</div>
```

20. Adicione à chamada a view **Index** no Layout:

```
<div class="navbar-collapse collapse">  
    <ul class="nav navbar-nav">  
        <li>  
            @Html.ActionLink("Clientes", "Index", "Clientes")  
        </li>  
        <li>  
            @Html.ActionLink("Produtos", "Index", "Produtos")  
        </li>  
        <li>  
            @Html.ActionLink("Pedidos", "Index", "Pedidos")  
        </li>  
        <li>  
            @Html.ActionLink("Itens", "Index", "Itens")  
        </li>  
        <li>  
            @Html.ActionLink("Pagamentos", "Index", "Faturas")  
        </li>  
    </ul>  
  
    @{Html.RenderPartial("_UsuarioPV");}  
</div>
```

21. Na pasta **Models**, adicionar uma classe chamada **Fatura**. Esta classe deve ter as mesmas propriedades da classe Fatura definida no serviço, pois suas propriedades serão serializadas e enviadas através da rede. Atingindo o serviço, as informações serão desserializadas e, desta forma, deverá haver uma correspondência de nomes para que o Web API faça o trabalho de inclusão;

```
namespace Lab.MVC.Models
{
    public class Fatura
    {
        public class Fatura
        {
            public int Id { get; set; }

            [Display(Name="Cartão")]
            public string NumeroCartao { get; set; }

            [Display(Name = "Pedido")]
            public string NumeroPedido { get; set; }
            public double Valor { get; set; }
            public int Status { get; set; }
        }    }
}
```

22. No controller **FaturasController**, adicione o action **EfetuarPagamento** nas versões GET e POST. A versão GET terá uma lista com os pedidos e um campo para fornecer o número do cartão. O valor total da fatura será calculado com base nos itens do pedido a ser pago. A seguir, a implementação do action, versão GET:

```
using Lab.MVC.Data;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class FaturasController : Controller
    {
        // GET: Faturas
        public ActionResult Index()
        {
            return View();
        }

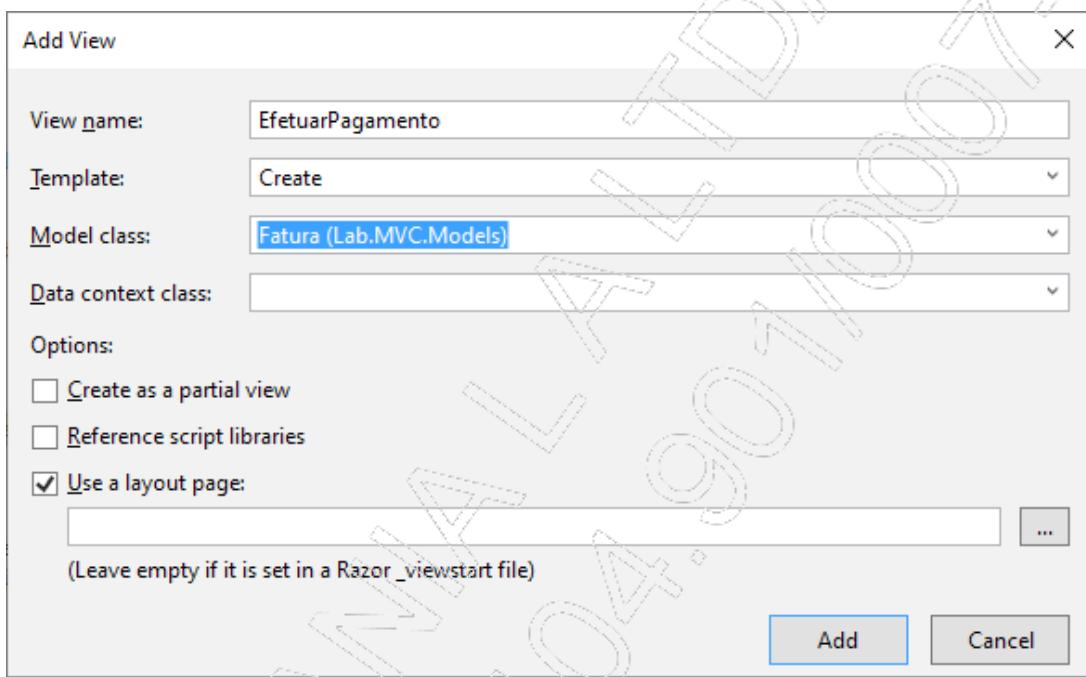
        [HttpGet]
        public ActionResult EfetuarPagamento()
        {
            try
            {
                ViewBag.ListaDePedidos = new SelectList(
                    ItensDao.ListarPedidos(), "NumeroPedido", "NomeCliente");

                return View();
            }
            catch (Exception ex)
            {
```

```
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
}
```

Observe que alteramos o "value" do nosso objeto **SelectList** para **NumeroPedido** em vez de **IdPedido**. Isso porque nossa classe **Fatura** possui o número do pedido;

23. Vamos gerar a view para este action:



```
@model Lab.MVC.Models.Fatura
@{
    ViewBag.Title = "Efetuar Pagamento";
}
<h2>Efetuar Pagamento</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
```

```
<hr />
@Html.ValidationSummary(true, "", new { @class = "text-danger" })


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



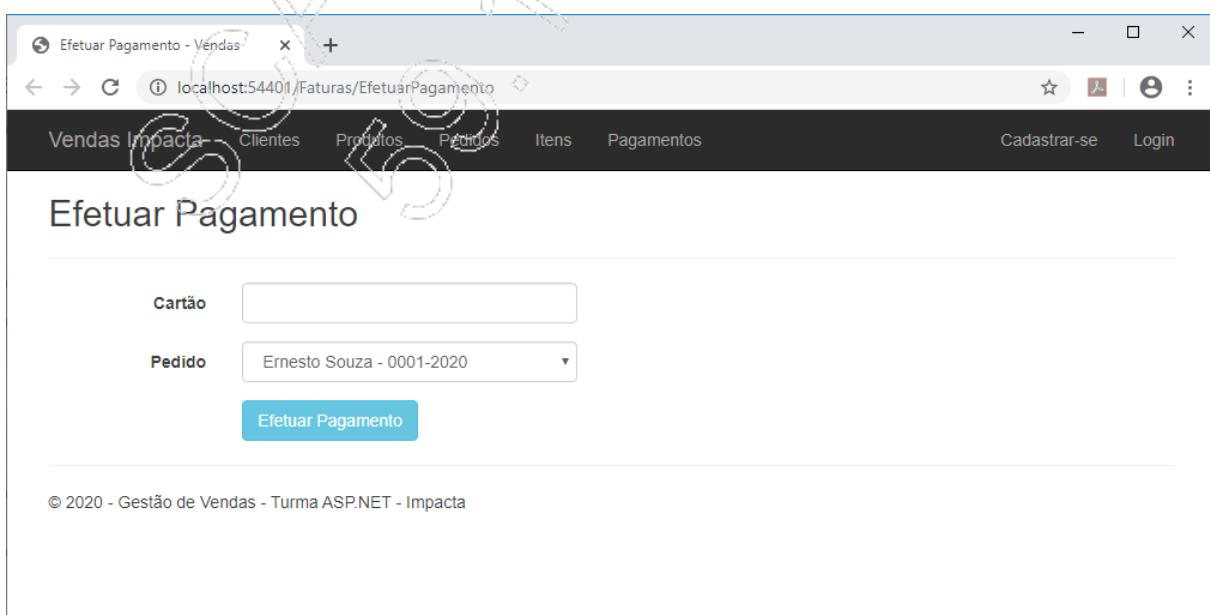
@Html.LabelFor(model => model.NumeroPedido, htmlAttributes:
    new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownListFor(model => model.NumeroPedido,
        (SelectList)ViewBag.ListaDePedidos,
        new { @class = "form-control" })
        @Html.ValidationMessageFor(model => model.NumeroPedido, "",
        new { @class = "text-danger" })
    </div>
</div>



<div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Efetuar Pagamento"
        class="btn btn-info" />
    </div>
</div>
}
}


```

24. A view terá apenas dois campos de entrada:



25. Para determinar o valor total dos itens, necessitaremos do Id do pedido, já que ele é usado no método **ListarItensPorPedido** da classe **ItensDao**. Vamos acrescentar um método na classe **PedidosDao** que receba como parâmetro o número do pedido e retorne seu Id:

```
public static int BuscarId(string numeroPedido)
{
    using (var ctx = new DbVendasEntities())
    {
        var pedido = ctx.Pedidos.FirstOrDefault(p =>
            p.NumeroPedido.Equals(numeroPedido));

        return pedido.Id;
    }
}
```

26. Para desenvolver a versão POST do action **EfetuarPagamento**, devemos levar em conta o acesso ao serviço. Definiremos um objeto do tipo **HttpClient**, responsável por se comunicar com nossa aplicação Web API. O objeto **HttpClient** está no namespace **System.Net.Http**. Se não estiver disponível, será necessário adicionar sua referência ao projeto. Além disso, usaremos chamadas assíncronas de métodos de **HttpClient**. Isso significa que nosso action deverá ser escrito como **async**, retornando um **Task<ActionResult>**:

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class FaturasController : Controller
    {
        HttpClient client;

        public FaturasController()
        {
            if (client == null)
            {
                client = new HttpClient();
                client.BaseAddress = new Uri("http://localhost:58671/");
                client.DefaultRequestHeaders.Accept.Add(new
                    MediaTypeWithQualityHeaderValue("application/json"));
            }
        }
    }
```

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

[HttpGet]
public ActionResult EfetuarPagamento()
{
    try
    {
        ViewBag.ListaDePedidos = new SelectList(
            ItensDao.ListarPedidos(), "NúmeroPedido", "NomeCliente");

        return View();
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}

[HttpPost]
public async Task<ActionResult> EfetuarPagamento(Fatura fatura)
{
    try
    {
        //obtendo o id do pedido
        int idPedido = PedidosDao.BuscarId(fatura.NumeroPedido);

        //obtendo a soma dos itens do pedido
        double totalPedido = ItensDao
            .ListarItensPorPedido(idPedido)
            .ToList()
            .Sum(p => p.TotalItem);

        //completando o objeto Fatura
        fatura.Valor = totalPedido;
        fatura.Status = 1;

        //obtendo o conteúdo JSON a partir do objeto
        string json = JsonConvert.SerializeObject(fatura);

        //serializando o objeto
        HttpContent content = new StringContent(
            json, Encoding.Unicode, "application/json");
    }
}
```

```
//enviando o conteúdo serializado para o serviço
var response = await client.PostAsync("api/pagamentos",
    content);
if (response.IsSuccessStatusCode)
{
    return View("_Sucesso");
}
else
{
    throw new Exception(response.ReasonPhrase);
}
catch (Exception ex)
{
    ViewBag.MensagemErro = ex.Message;
    return View("_Erro");
}
}
```

! É importante observar que a URL informada na propriedade **BaseAddress** deve ser igual à URL do seu serviço. Portanto, ela não deve ser copiada!

27. No controller **FaturasController**, escreva o action **ListarFaturas**. Este action retornará a lista de faturas do Webservice.

```
using Lab.MVC.Data;
using Lab.MVC.Models;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;

namespace Lab.MVC.Controllers
{
    public class FaturasController : Controller
    {
        HttpClient client;

        public FaturasController()
        {
            if (client == null)
            {
```

```
client = new HttpClient();
client.BaseAddress = new Uri("http://localhost:58671/");
client.DefaultRequestHeaders.Accept.Add(new
    MediaTypeWithQualityHeaderValue("application/json"));
}

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

[HttpGet]
public ActionResult EfetuarPagamento()
{
    try
    {
        ViewBag.ListaDePedidos = new SelectList(
            ItensDao.ListarPedidos(), "NumeroPedido", "NomeCliente");

        return View();
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("Erro");
    }
}

[HttpPost]
public async Task<ActionResult> EfetuarPagamento(Fatura fatura)
{
    try
    {
        //obtendo o id do pedido
        int idPedido = PedidosDao.BuscarId(fatura.NumeroPedido);

        //obtendo a soma dos itens do pedido
        double totalPedido = ItensDao
            .ListarItensPorPedido(idPedido)
            .ToList()
            .Sum(p => p.TotalItem);

        //completando o objeto Fatura
        fatura.Valor = totalPedido;
        fatura.Status = 1;

        //obtendo o conteúdo JSON a partir do objeto
        string json = JsonConvert.SerializeObject(fatura);

        //serializando o objeto
        HttpContent content = new StringContent(
            json, Encoding.Unicode, "application/json");
    }
}
```

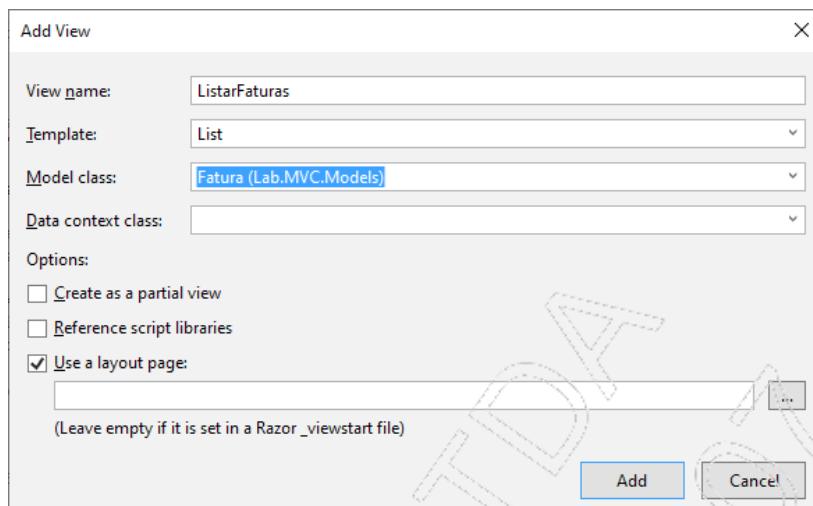
```
//enviando o conteúdo serializado para o serviço
var response = await client.PostAsync("api/pagamentos",
    content);
if (response.IsSuccessStatusCode)
{
    return View("_Sucesso");
}
else
{
    throw new Exception(response.ReasonPhrase);
}
catch (Exception ex)
{
    ViewBag.MensagemErro = ex.Message;
    return View("_Erro");
}

public async Task<ActionResult> ListarFaturas()
{
    try
    {
        HttpResponseMessage response = client
            .GetAsync("api/pagamentos").Result;

        if (response.IsSuccessStatusCode)
        {
            var resultado = await response.Content
                .ReadAsStringAsync();

            var lista = JsonConvert
                .DeserializeObject<Fatura[]>(resultado)
                .ToList();
            return View(lista);
        }
        else
        {
            throw new Exception(response.ReasonPhrase);
        }
    }
    catch (Exception ex)
    {
        ViewBag.MensagemErro = ex.Message;
        return View("_Erro");
    }
}
```

28. Gere a view para o action **ListarFaturas**:



```
@model IEnumerable<Lab.MVC.Models.Fatura>

@{
    ViewBag.Title = "Faturas";
}



## Lista de Faturas



| @Html.DisplayNameFor(model => model.NumeroCartao) | @Html.DisplayNameFor(model => model.NumeroPedido) | @Html.DisplayNameFor(model => model.Valor) | @Html.DisplayNameFor(model => model.Status) |
|---------------------------------------------------|---------------------------------------------------|--------------------------------------------|---------------------------------------------|
|---------------------------------------------------|---------------------------------------------------|--------------------------------------------|---------------------------------------------|


```

```
@foreach (var item in Model) {  
    <tr>  
        <td>  
            @Html.DisplayFor(modelItem => item.NumeroCartao)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.NumeroPedido)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Valor)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Status)  
        </td>  
    </tr>  
}  
  
</table>
```

29. Para testar a aplicação, deveremos:

- Incluir alguns cartões na tabela **TBCartoes**, no banco de dados gerado no projeto Web API (Afinal, o cartão deve existir e ter saldo suficiente para que o pagamento possa ser efetuado).
- Executar a aplicação Web API.
- Em uma nova instância do Visual Studio, executar a aplicação MVC.

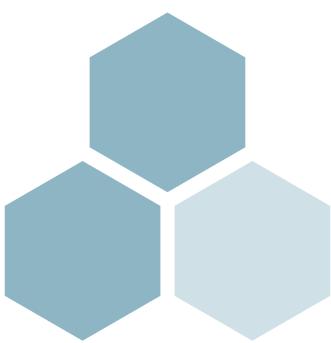
Cumpridas estas tarefas, analise os resultados.

Para que as aplicações funcionem simultaneamente, o ideal é que elas sejam publicadas no IIS, cada uma com um link diferente. Assim, outras aplicações também poderão consumi-las.



ASP.NET Core

- ◆ .NET Core e seus projetos;
- ◆ Comandos do .NET Core;
- ◆ Fluxo de execução de uma aplicação ASP.NET Core;
- ◆ Controllers e Actions;
- ◆ Microsoft.AspNet.Mvc.TagHelpers.



7.1. Introdução

Até o último instante, a versão que seria a sucessora da versão 4.6 do ASP.NET estava sendo chamada de ASP.NET 5. Poucos dias antes do lançamento oficial, a Microsoft anuncia que a nova versão terá o nome de **ASP.NET Core**. Essa mudança reflete exatamente o que houve com a nova versão da plataforma: uma reformulação geral, partindo para um modelo completamente novo.

O framework .NET Core da Microsoft é uma plataforma híbrida *cross-platform* unificada, elaborada para desenvolvimento em diversas plataformas (dentre elas a plataforma Web), capazes de serem executadas nos sistemas operacionais Windows, Linux e Mac.

Suas principais características são:

- Open source;
- Cross platform;
- Plataforma escalável de alto desempenho;
- Suporta containers *Docker* (ambiente de virtualização em nível de sistema) operacionais capazes de executar contêineres Linux em um único host de controle;
- Suporta a arquitetura de microserviços;
- Suporta integração com Github e NuGet;
- Suporta ferramentas de linhas de comando;
- Projetado para suportar *deploy* em ambientes na nuvem.

Essa nova direção abre muitas possibilidades de desenvolvimento e integração de sistemas. Até a versão 4.6, os recursos do ASP.NET e do .NET como um todo estavam limitados a servidores ou computadores desktop utilizando sistema operacional Windows. Com a inserção do .NET Core, qualquer sistema operacional, biblioteca, ferramenta de programação ou framework pode ser utilizado para desenvolver, executar ou hospedar aplicativos .NET.

Na prática, isso significa que agora é possível criar um aplicativo para dispositivos Android, iOS ou para um computador executando Linux. Além disso, é possível distribuir o ASP.NET como parte da aplicação, criando servidores virtuais e serviços de gerenciamento de informações da Internet que são distribuídos junto com o aplicativo.

7.2. .NET Core e seus projetos

No que diz respeito a tipos de projetos, o framework .NET é usado no desenvolvimento de aplicações Windows Forms e WPF, além de aplicações Web com MVC e Web Forms. O .NET Core suporta as bibliotecas UWP e ASP.NET Core. UWP é usado para o desenvolvimento na plataforma Windows 10 e o ASP.NET Core é usado em aplicações Web nas plataformas Windows, Linux e Mac.

O propósito do nosso curso é o desenvolvimento Web usando o framework .NET Core.

7.3. Comandos do .NET Core

A plataforma .NET Core fornece os componentes necessários para executar programas em diversos sistemas operacionais. A página oficial <https://dotnet.microsoft.com/download> fornece os links para a instalação do kit de desenvolvimento (SDK) para as diversas plataformas. Para os usuários Windows, é preferível utilizar o instalador do Visual Studio, que já faz o download e configura o ambiente de desenvolvimento automaticamente. Se este não foi instalado com o Visual Studio (e mesmo que estejamos em outros sistemas operacionais), a melhor forma é baixá-lo a partir do link apresentado anteriormente.

Uma vez instalado o SDK, é possível criar e executar aplicativos, seja usando o Visual Studio, seja através do prompt de comandos.

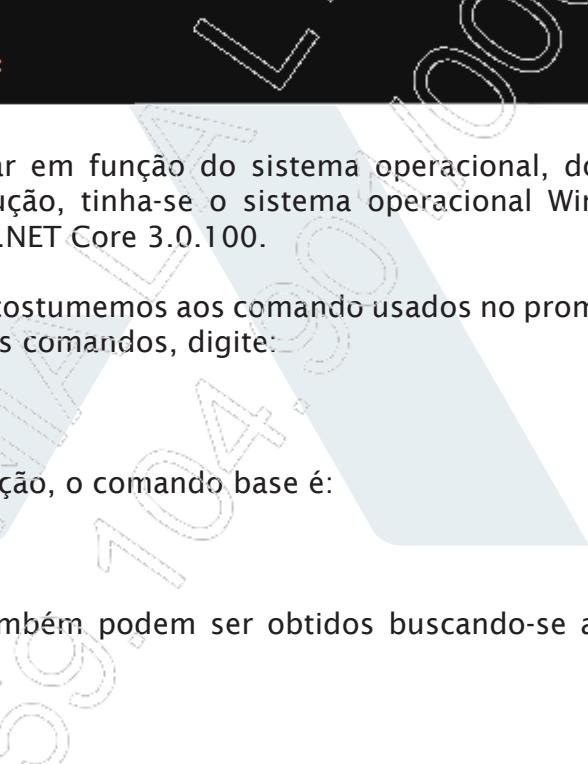
7.3.1. Criando aplicativos por linhas de comandos

Vamos agora apresentar os procedimentos para criar e executar um aplicativo .NET Core a partir de linhas de comando. O principal comando usado no console é o **dotnet**. A partir dele, podemos informar parâmetros e especificar os detalhes do projeto. Vamos conhecer alguns destes parâmetros, juntamente com o comando **dotnet**:

1. Usaremos o prompt de comando (Console). Como dito anteriormente, o principal comando usado é o **dotnet**. Um exemplo:

```
>dotnet --info
```

Com ele obtemos um resultado semelhante a este:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [versão 10.0.18363.657]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

D:\Documentos\Projetos\Impacta\AspNet>dotnet --info
SDK do .NET Core (refletindo qualquer global.json):
  Version: 3.0.100
  Commit: 04339c3a26

  Ambiente de runtime:
    OS Name: Windows
    OS Version: 10.0.18363
    OS Platform: Windows
    RID: win10-x64
    Base Path: C:\Program Files\dotnet\sdk\3.0.100\

  Host (useful for support):
    Version: 3.0.0
    Commit: 7d57652f33

  .NET Core SDKs installed:
```

Os detalhes podem mudar em função do sistema operacional, do framework etc. No momento desta execução, tinha-se o sistema operacional Windows 10 com a compilação 10.0.18363 e .NET Core 3.0.100.

É recomendável que nos acostumemos aos comandos usados no prompt de comandos. Para conhecer os principais comandos, digite:

>dotnet --help

Para criar uma nova aplicação, o comando base é:

>dotnet new

Os demais parâmetros também podem ser obtidos buscando-se ajuda, através do comando:

>dotnet new --help

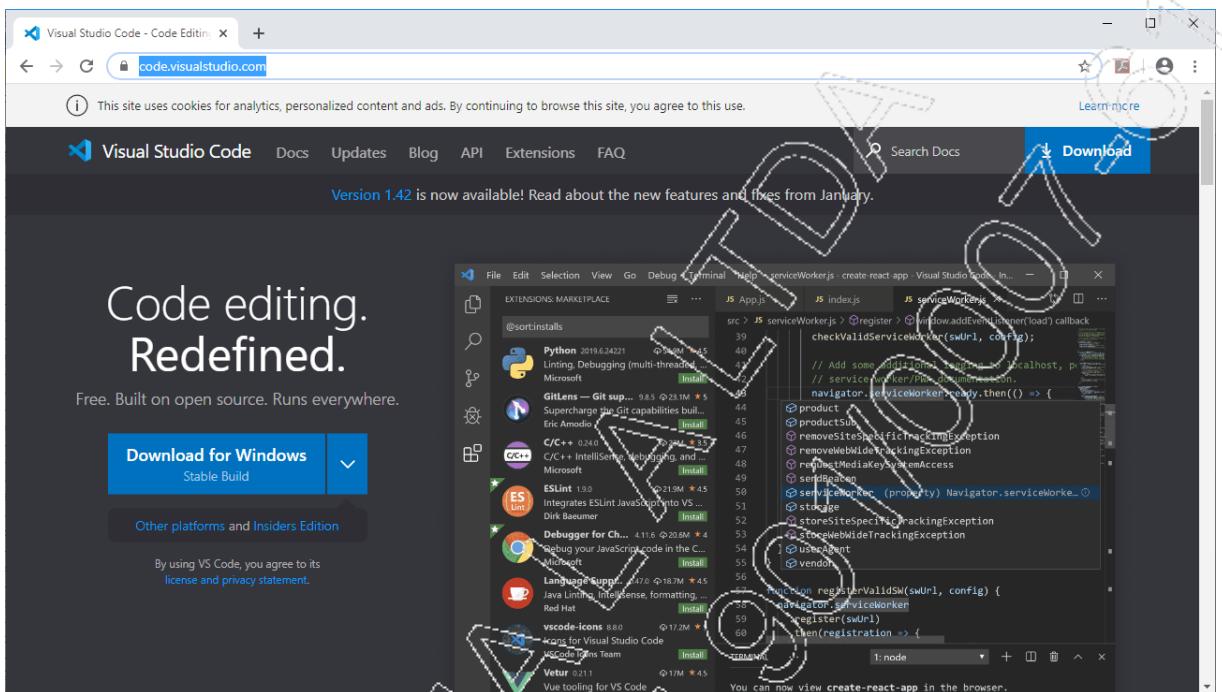
Para exemplificar, vamos criar um novo projeto chamado **Capítulo07.NetCore.Console**:

>dotnet new console --name Capítulo07.NetCore.Console

Se o atributo **--name** juntamente com o nome sugerido for omitido, será criado um projeto com o nome da pasta selecionada.

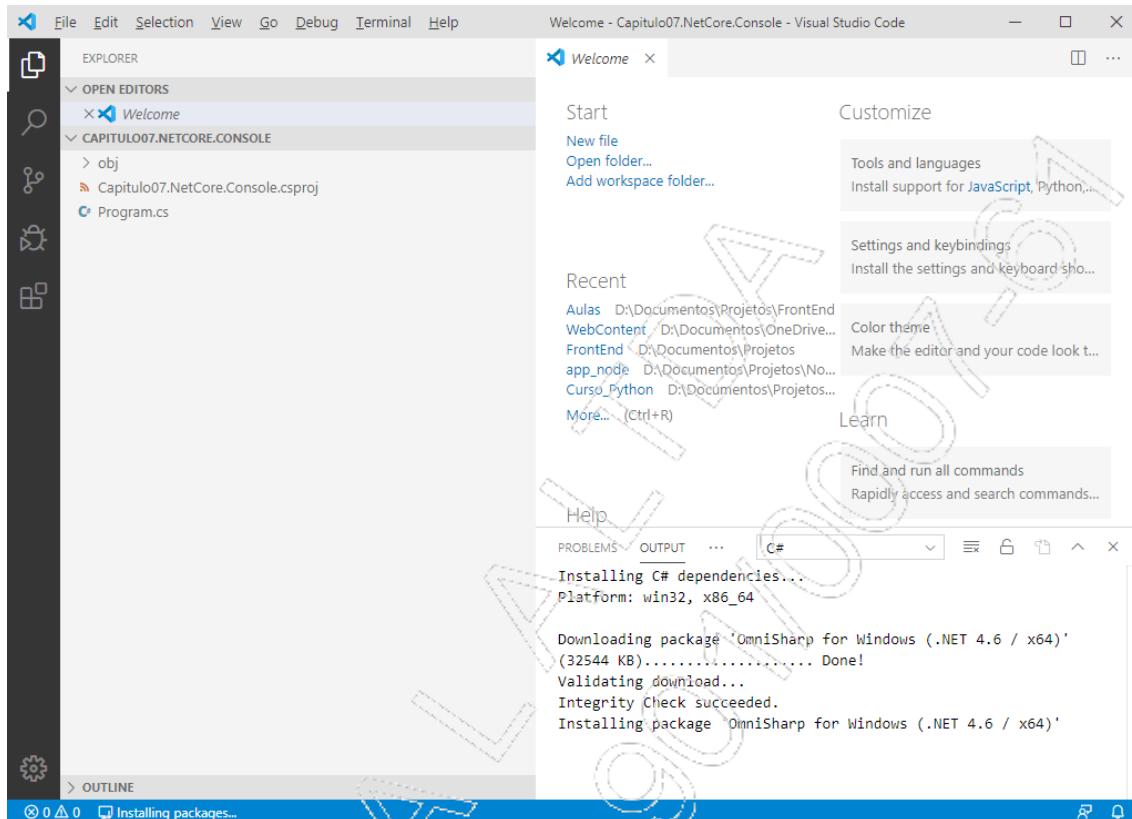
7.3.2. Usando o Visual Studio Code

Com o projeto criado, podemos analisar seu conteúdo. Como ele foi criado através do console, não precisa necessariamente ser aberto no Visual Studio. A opção recomendada é o **Visual Studio Code** (ou simplesmente **VSCode**), especialmente quando estivermos em outros sistemas operacionais diferentes do Windows. O Visual Studio Code pode ser obtido no link <https://code.visualstudio.com/>.



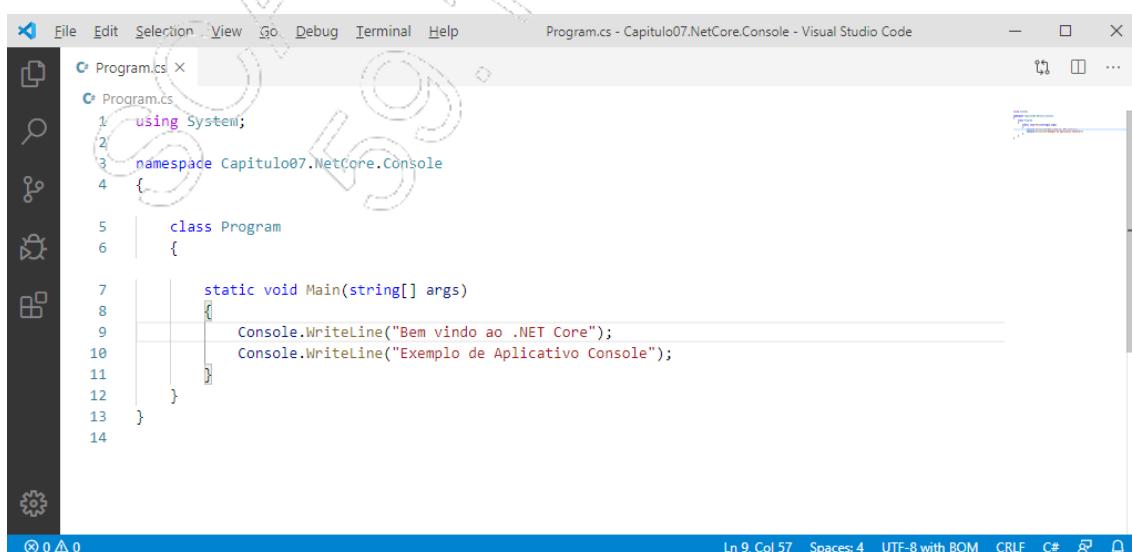
O VSCode é um editor bastante leve, que utiliza extensões para uma grande variedade de aplicações. Com ele, podemos elaborar aplicações em HTML, CSS, JavaScript, C#, Java, Python, e muitas outras, desde que tenhamos as extensões adequadas. Diferentemente dos projetos que criamos no Visual Studio, ele não trabalha com estrutura baseada em projetos, e sim com diretórios.

Para utilizarmos o VSCode, vamos abri-lo e, no menu **File**, selecionar a opção **Open Folder**. Selecione a pasta do projeto que criamos, ou seja, **Capítulo07.NetCore.Console**:



Ao abrir o projeto, é possível que o próprio editor reconheça seu conteúdo e tente baixar as extensões necessárias. Se isso ocorrer, aguarde o final da instalação.

Em seguida, abra o arquivo **Program.cs**. Vamos realizar algumas pequenas alterações.



Para executá-lo, vamos usar o prompt de comandos. Execute a sequência de comandos:

```
>cd Capítulo07.NetCore.Console
>dotnet build --output build_app
>dotnet build_app/Capítulo07.NetCore.Console.dll
```

Essa sequência de comandos permite compilar o projeto em uma DLL na pasta **build_app** e, em seguida, executá-lo.

A seguir, apresentamos algumas das opções comuns disponíveis para o comando **dotnet**:

Comando	Descrição
-v --verbose	Habilita output com bastante detalhes (verbose).
--version	Apresenta as informações de versão do .NET CLI.

Alguns comandos comuns são:

Comando	Descrição
new	Inicia um novo projeto .NET básico.
restore	Restaura as dependências especificadas no projeto.
build	Compila um projeto .NET Core.
publish	Publica um projeto .NET para <i>deployment</i> (incluindo o <i>runtime</i>).
run	Compila e imediatamente executa um projeto.
test	Executa testes unitários usando o <i>test runner</i> especificado no projeto.
pack	Cria um pacote NuGet.

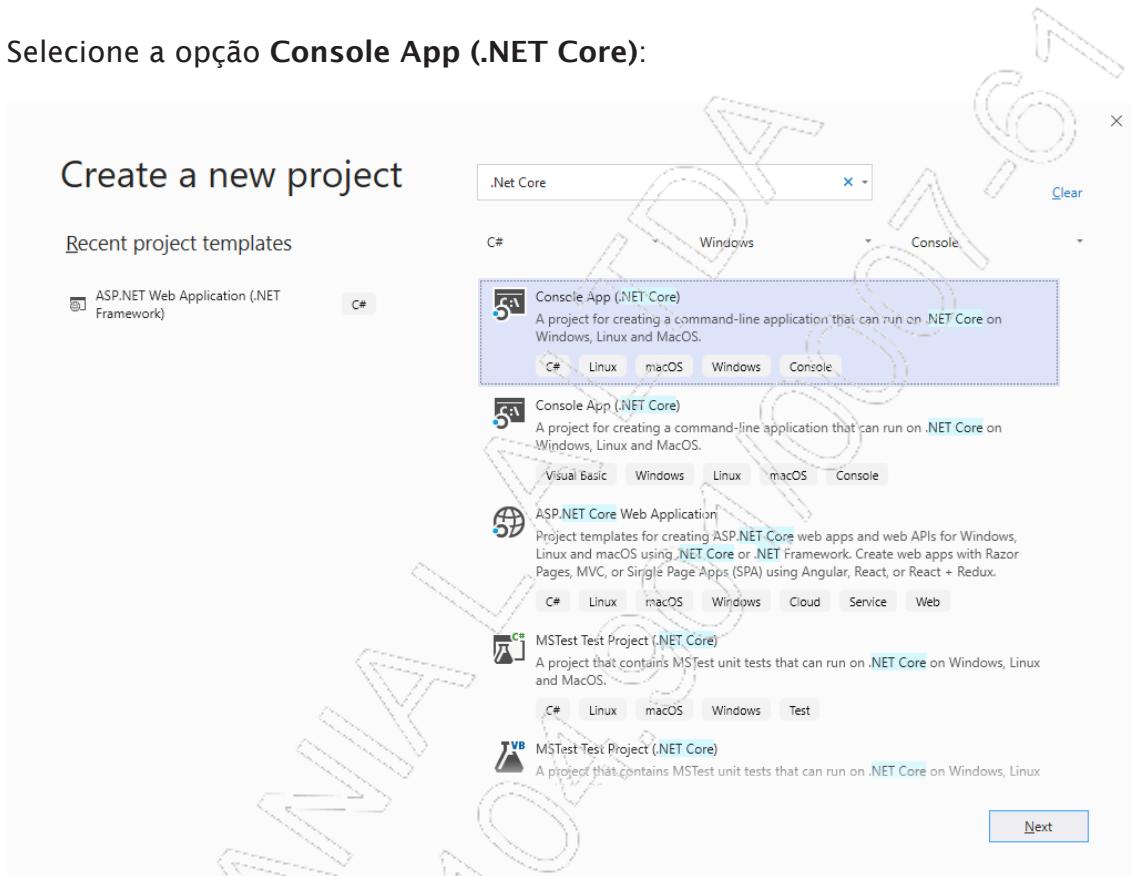
Esta abordagem foi apresentada como forma de demonstrar que não precisamos obrigatoriamente do Visual Studio para criarmos projetos baseados no .NET Core.

Ao longo das aulas, utilizaremos o **Visual Studio**.

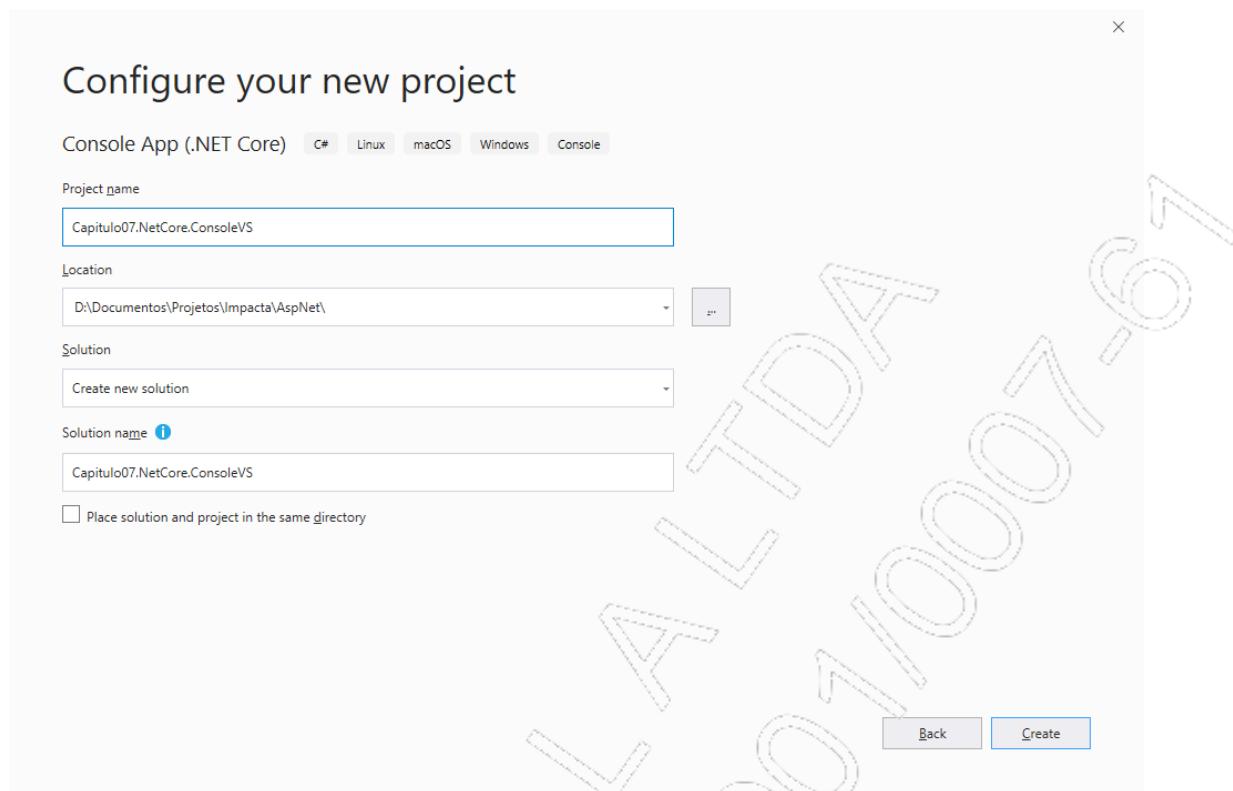
7.3.3. Usando o Visual Studio

Já estamos habituados com o Visual Studio, e veremos que criar uma aplicação ASP.NET Core é tão simples como outras aplicações que criamos até o momento. Vamos seguir as etapas para criar nosso primeiro projeto:

1. Abra o Visual Studio;
2. Selecione a opção **Console App (.NET Core)**:



3. Na próxima etapa, selecione uma pasta adequada e forneça o nome **Capítulo07.NetCore.ConsoleVS**:



4. Selecione a opção **Create**.

Temos nosso projeto **Console**, de forma semelhante ao que fizemos via linha de comandos. Analogamente ao projeto anterior, vamos realizar algumas alterações no arquivo **Program.cs**:

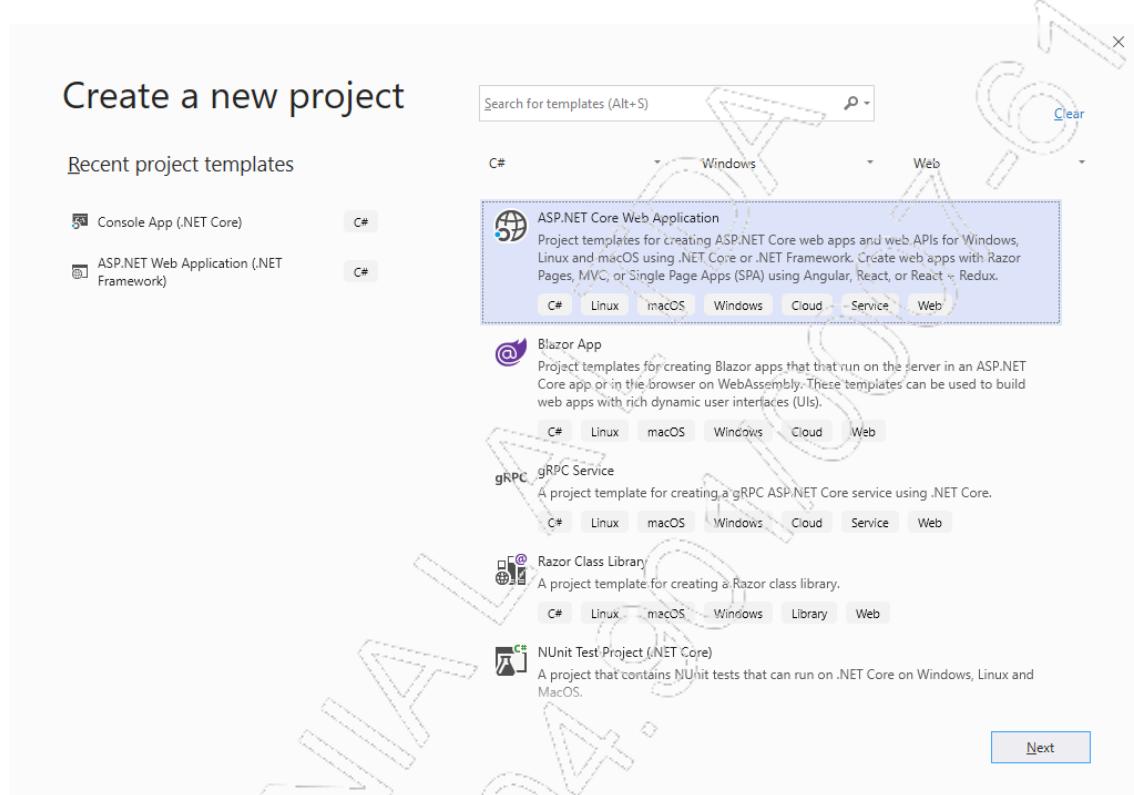
```
using System;

namespace Capítulo07.NetCore.ConsoleVS
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Aplicativo Console, .NET Core");
            Console.ReadKey();
        }
    }
}
```

7.3.4. Aplicação ASP.NET Core

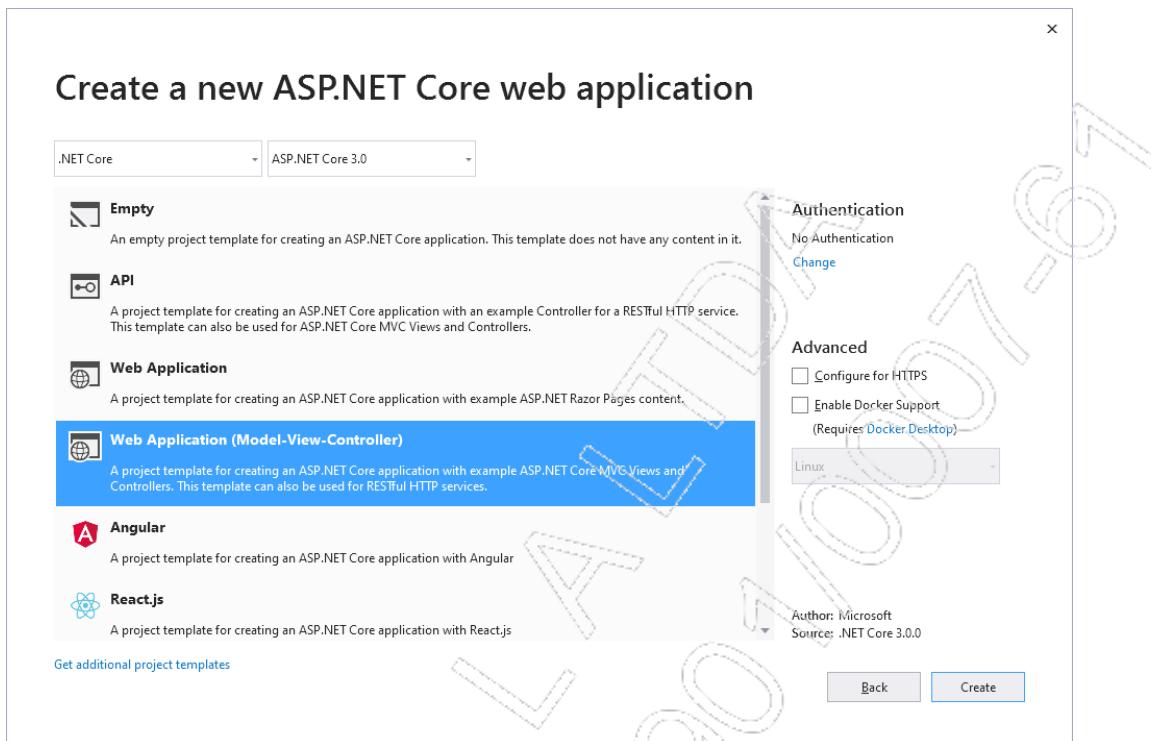
O modelo de Web App para o .NET Core é um bom exemplo de como configurar e implementar os componentes ASP.NET Core. Para incluir um Web App usando .NET Core, vamos acompanhar estes passos:

1. No menu **File / New Project**, selecione a opção **ASP.NET Core Web Application**:



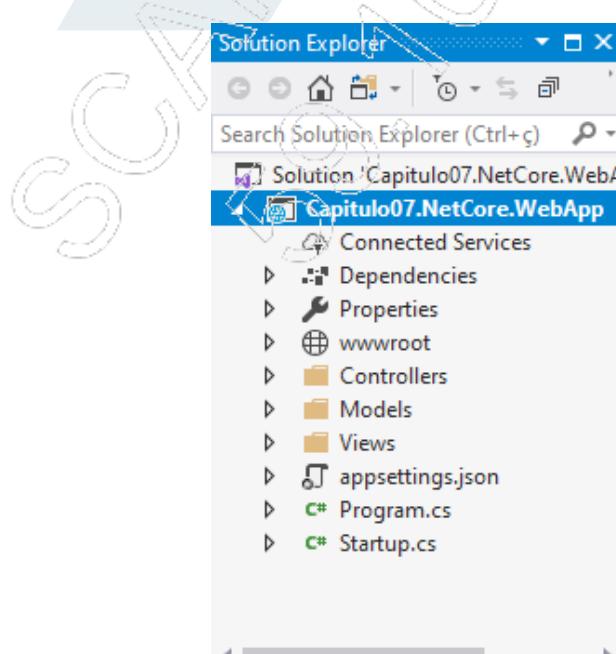
2. No próximo passo, selecione um local e um nome para o projeto. Chame o projeto de **Capítulo07.NetCore.WebApp**:

3. Existem alguns templates disponíveis. Observe que, como camada Front End, podemos escolher também o Angular e o React. Como nosso objetivo é trabalhar com a arquitetura MVC, vamos selecionar esta opção (como vantagem, o Visual Studio adiciona as referências necessárias para o projeto):



4. Crie o projeto.

Analizando a janela **Solution Explorer**, podemos ver que a estrutura de um projeto ASP.NET Core é bem diferente daquela gerada no projeto ASP.NET com .NET Framework:



Algumas importantes alterações estruturais são as seguintes:

- O formato XML não é utilizado mais para arquivos de sistema. O formato agora é JSON;
- Não existe mais **Web.Config**. O arquivo de configuração agora se chama **appsettings.json**;
- Mesmo sendo uma aplicação Web, a execução parte do método **Main** presente no arquivo **Program.cs**;
- O ciclo de vida e configuração dos componentes da aplicação são realizados no arquivo **Startup.cs**.

Nas próximas linhas, conheceremos o fluxo de execução de uma aplicação ASP.NET Core, principalmente no que diz respeito ao arquivo **Startup.cs**.

7.4. Fluxo de execução de uma aplicação ASP.NET Core

Quando uma aplicação ASP.NET Core inicia, os comandos seguintes são executados, na sequência adiante:

1. Classe Startup – Método de entrada Main

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}
```

2. Classe Startup – Método construtor

Uma instância da classe **ConfigurationBuilder** é criada e associada à propriedade **Configuration** da classe **Startup**. Esta instância é referenciada por **IConfiguration**.

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

3. Classe Startup - Método ConfigureServices

Este método é chamado em tempo de execução pelo framework, e é responsável por adicionar serviços e tarefas à aplicação, como é o caso do Entity Framework Core, Identity Core, dentre outros.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
}
```

Neste exemplo ela está adicionando, por meio do método `AddControllersWithViews`, o recurso para que o ciclo de vida de uma aplicação MVC seja garantido.

4. Classe Startup – Método Configure

A classe executa diversos métodos para iniciar os serviços.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
    app.UseStaticFiles();

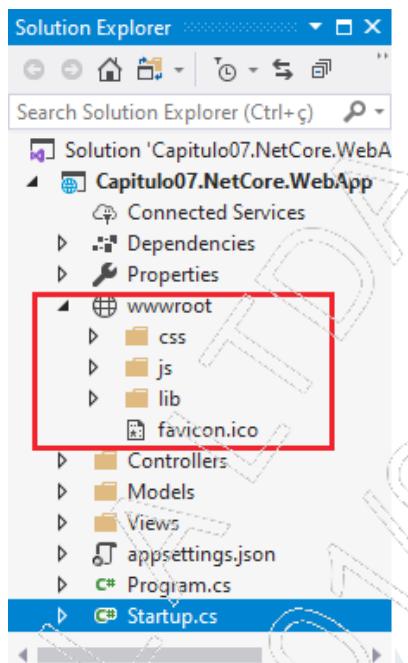
    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Como exemplos, podemos destacar os métodos:

- **UseStaticFiles**: Permite a inclusão de componentes estáticos, com imagens, arquivos CSS, conteúdo JavaScript, ou mesmo páginas puramente HTML. Neste caso, este conteúdo deve usar como raiz a pasta **wwwroot**;



- **UseRouting**: Habilita o uso do mecanismo de rotas na aplicação;
- **UseEndPoints**: Permite definir, dentre outros, o mapeamento das rotas, incluindo o controller e o action default (assim como fizemos na aplicação MVC).

Esta é uma amostra do arquivo Startup. O .NET Core utiliza extensivamente o recurso de **injeção de dependência**, já que o container permite a execução em diferentes plataformas. Nosso trabalho é especificar as atividades por meio das interfaces. As instâncias serão fornecidas pelo próprio .NET Core.

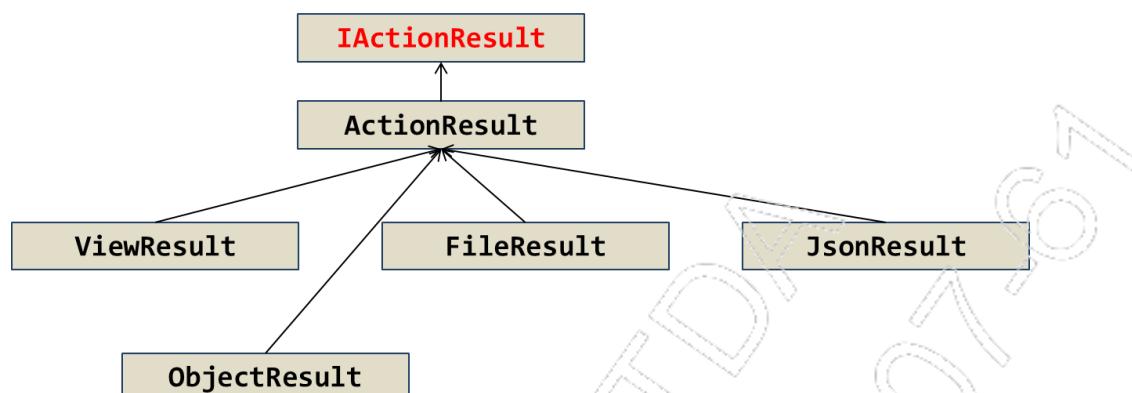
Ao longo do nosso desenvolvimento, teremos a oportunidade de estudar este recurso, especialmente quando aplicarmos o acesso a dados.

7.5. Controllers e Actions

Os **controllers** compõem a parte fundamental de uma aplicação MVC, principalmente por conta da arquitetura de mesmo nome.

Um controller é, na verdade, uma classe contendo métodos capazes de gerar a camada de interação com o usuário. Esses métodos são chamados de **actions**.

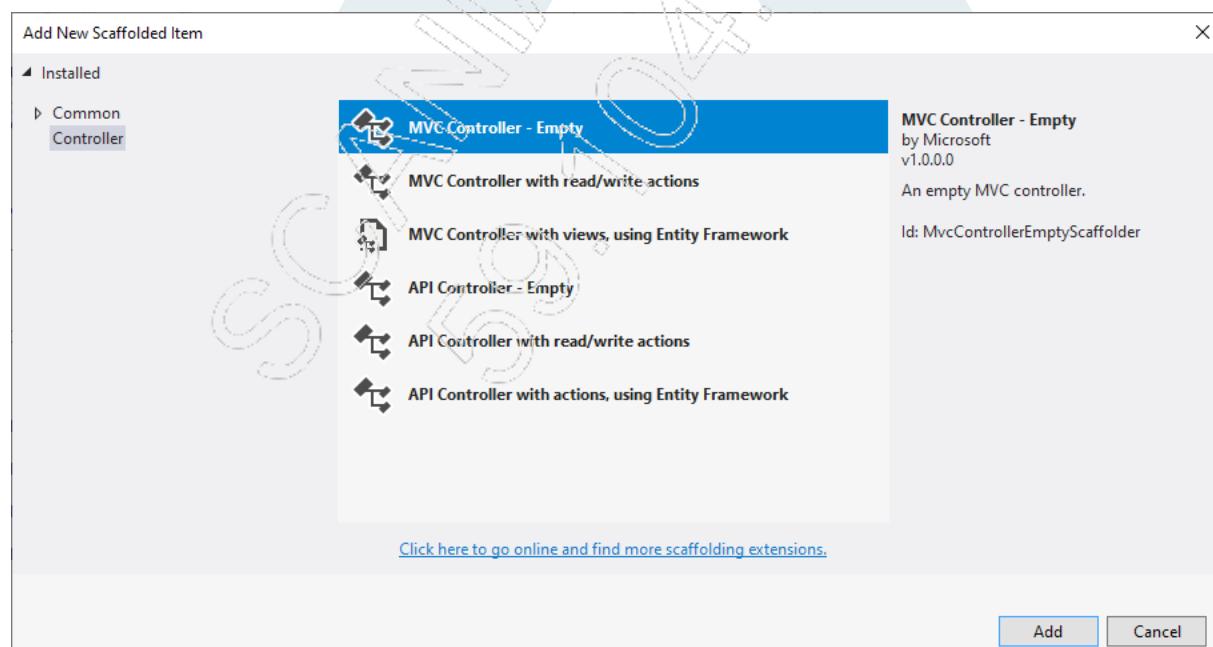
Já tivemos a oportunidade de estudar os controllers no capítulo referente ao ASP.NET MVC. No ASP.NET Core, o procedimento de criação de controllers é bastante similar, porém existe uma diferença fundamental com relação ao retorno dos actions: foi introduzida uma camada utilizando uma interface, **IActionResult**.



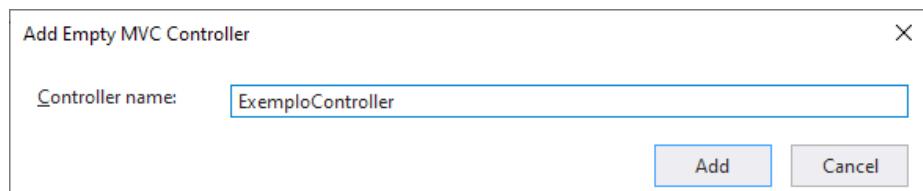
Ainda é possível trabalharmos com o retorno **ActionResult**, mas é altamente recomendado usar as interfaces sempre que possível, para viabilizar a injeção de dependência.

7.5.1. Adicionando um controller ao projeto

Vamos usar nosso projeto **Capítulo07.NetCore.WebApp**. Nele, adicionaremos um controller chamado **Exemplo** (classe **ExemploController**). Para tanto, clique com o botão direito do mouse na pasta **Controllers** e selecione a opção **Add / Controller**:



Para uma aplicação Web, selecione a opção **MVC Controller**. Em seguida, atribua o nome desejado:



Obtemos nosso controller com o action **Index**:

```
using Microsoft.AspNetCore.Mvc;  
  
namespace Capitulo07.NetCore.WebApp.Controllers  
{  
    public class ExemploController : Controller  
    {  
        public IActionResult Index()  
        {  
            return View();  
        }  
    }  
}
```

Podemos verificar que a estrutura é similar, mas a origem das classes é bem diferente. O .NET Core foi obtido, como já dito antes, através de um "redesenho" do .NET Framework.

A partir do controller, podemos gerar as views, de acordo com os actions que tivermos. As views podem ser escritas com conteúdo HTML, com elementos Razor ou com um tipo especial de elementos introduzidos no .NET Core: as **Tag Helpers**. Vamos conhecê-las a seguir.

7.6. Microsoft.AspNet.Mvc.TagHelpers

Algumas novidades foram introduzidas na geração do código HTML. A classe **TagHelpers** define alguns novos elementos HTML:

- **environment**

Este elemento renderiza ou não um código HTML baseado no valor da propriedade **Microsoft.AspNet.Hosting.IHostingEnvironment.EnvironmentName**, que é um valor inserido automaticamente quando a aplicação é iniciada.

No exemplo adiante, os elementos **link** serão renderizados apenas se o ambiente de execução estiver definido como **Development**:

```
<environment names="Development">
    <link rel="stylesheet" href="css/bootstrap.css" />
    <link rel="stylesheet" href("~/css/site.css" />
</environment>
```

- **asp-controller**

Este atributo define o nome de um controlador. É utilizado para criar uma âncora HTML (hiperlink).

```
<a asp-controller="Home" asp-action="Index">Home</a>
```

- **asp-action**

Este atributo define o nome de um método de um controlador. É utilizado para criar uma âncora HTML (hiperlink).

```
<a asp-controller="Home" asp-action="Index">Home</a>
```

Para elementos de formulários, temos os elementos referentes aos campos de entrada, como os **inputs**. Apesar de ser um elemento HTML tradicional, a inclusão de certos atributos os tornam elementos de servidor. Para exemplificar, vamos adicionar um model ao nosso projeto, um novo action no controller **Exemplos** e, a partir dele, uma nova view com elementos de formulário:

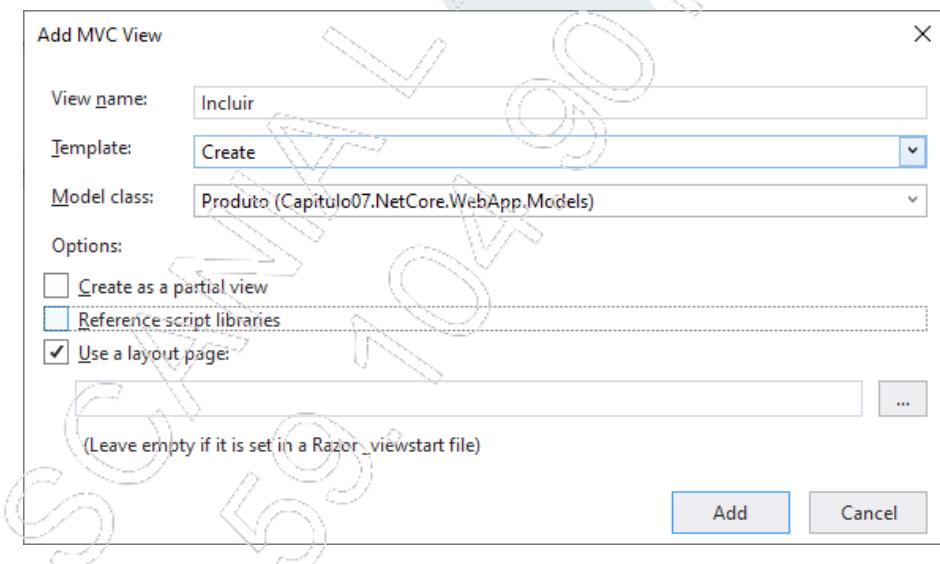
- **model Produto**

```
namespace Capitulo07.NetCore.WebApp.Models
{
    public class Produto
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
    }
}
```

- **action Incluir**

```
using Microsoft.AspNetCore.Mvc;  
  
namespace Capitulo07.NetCore.WebApp.Controllers  
{  
    public class ExemploController : Controller  
    {  
        public IActionResult Index()  
        {  
            return View();  
        }  
  
        public IActionResult Incluir()  
        {  
            return View();  
        }  
    }  
}
```

- **view Incluir.cshtml**



```

@model Capitulo07.NetCore.WebApp.Models.Produto

 @{
    ViewData["Title"] = "Incluir";
}

<h1>Incluir</h1>

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Incluir">
            <div asp-validation-summary="ModelOnly"
                class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Id" class="control-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Descricao" class="control-label"></label>
                <input asp-for="Descricao" class="form-control" />
                <span asp-validation-for="Descricao"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Preco" class="control-label"></label>
                <input asp-for="Preco" class="form-control" />
                <span asp-validation-for="Preco" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Incluir" class="btn btn-
info" />
            </div>
        </form>
    </div>
    <div>
        <a asp-action="Index">Voltar</a>
    </div>

```

Trata-se de uma view fortemente tipada, onde cada campo de entrada se relaciona com uma propriedade por meio do atributo **asp-for**:

```
<input asp-for="Descricao" class="form-control" />
```

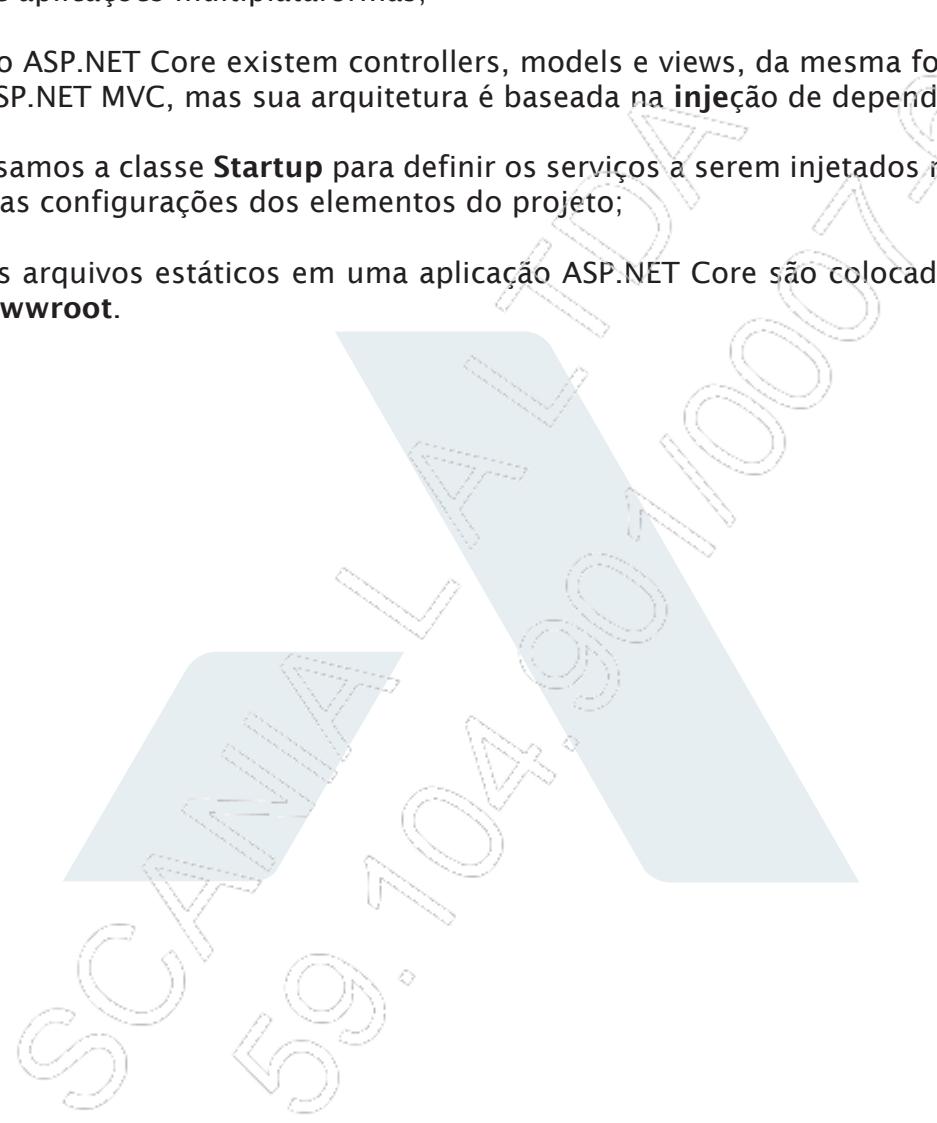
O formulário indica o action a ser executado quando submetido, através do atributo **asp-action**:

```
<form asp-action="Incluir">
```

Diferente de um formulário tradicional, este formulário envia, por default, seu conteúdo via HTTP POST. Se quisermos que seja pelo HTTP GET, teremos que indicar.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

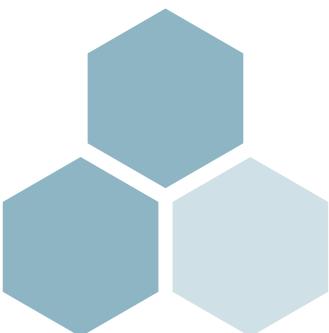
- O .NET Core é uma evolução do .NET Framework que permite o desenvolvimento de aplicações multiplataformas;
 - No ASP.NET Core existem controllers, models e views, da mesma forma que no ASP.NET MVC, mas sua arquitetura é baseada na **injeção de dependência**;
 - Usamos a classe **Startup** para definir os serviços a serem injetados na aplicação e as configurações dos elementos do projeto;
 - Os arquivos estáticos em uma aplicação ASP.NET Core são colocados na pasta **wwwroot**.
- 



7

ASP.NET Core

Teste seus conhecimentos



1. Qual a principal característica do .NET Core?

- a) Utiliza qualquer linguagem de programação.
- b) Utiliza JavaScript.
- c) Hospeda aplicações em servidores Windows.
- d) Multiplataforma.
- e) Limitado ao SQL Server.

2. Qual o comando usado para criarmos aplicações baseadas em .NET Core a partir do prompt de comandos?

- a) create
- b) dotnet
- c) console
- d) new
- e) project

3. Para executar uma aplicação desenvolvida no .NET Core, qual o primeiro método a ser executado?

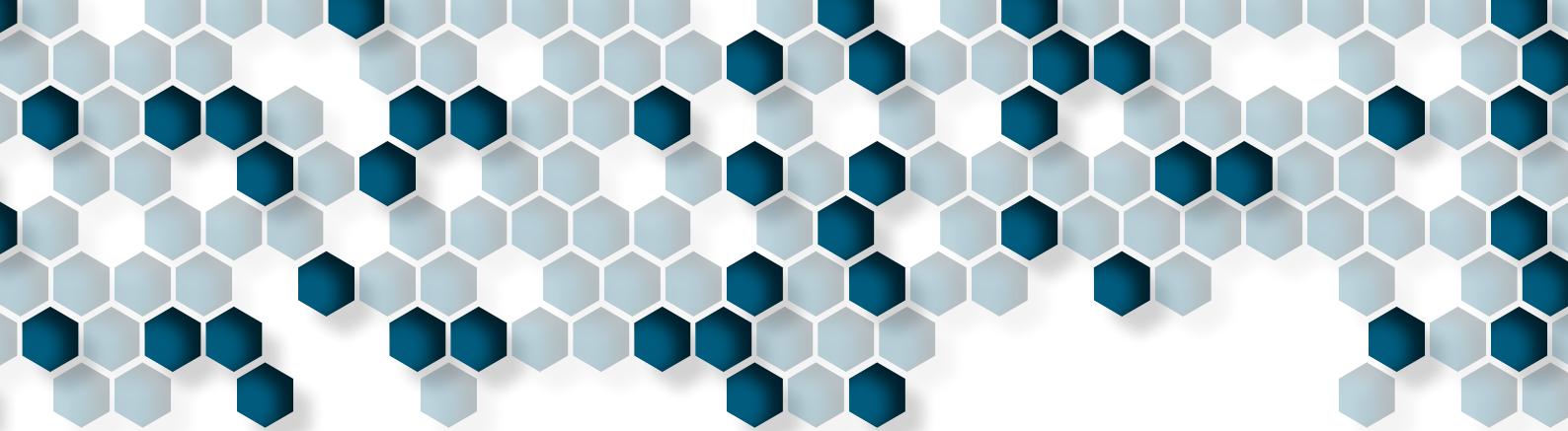
- a) Main
- b) Startup
- c) Configure
- d) Services
- e) Init

4. O arquivo de configurações de um projeto .NET Core se chama:

- a) web.config
- b) wpp.Config
- c) wpp.json
- d) settings.json
- e) appsettings.json

5. O método chamado em tempo de execução para adicionar serviços na aplicação se chama:

- a) Startup
- b) Configure
- c) Main
- d) ConfigureServices
- e) ConfigureProviders

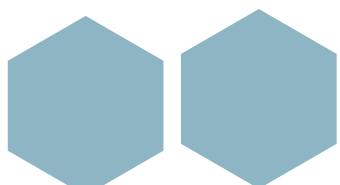


7

ASP.NET Core



Mãos à obra!



Objetivos:

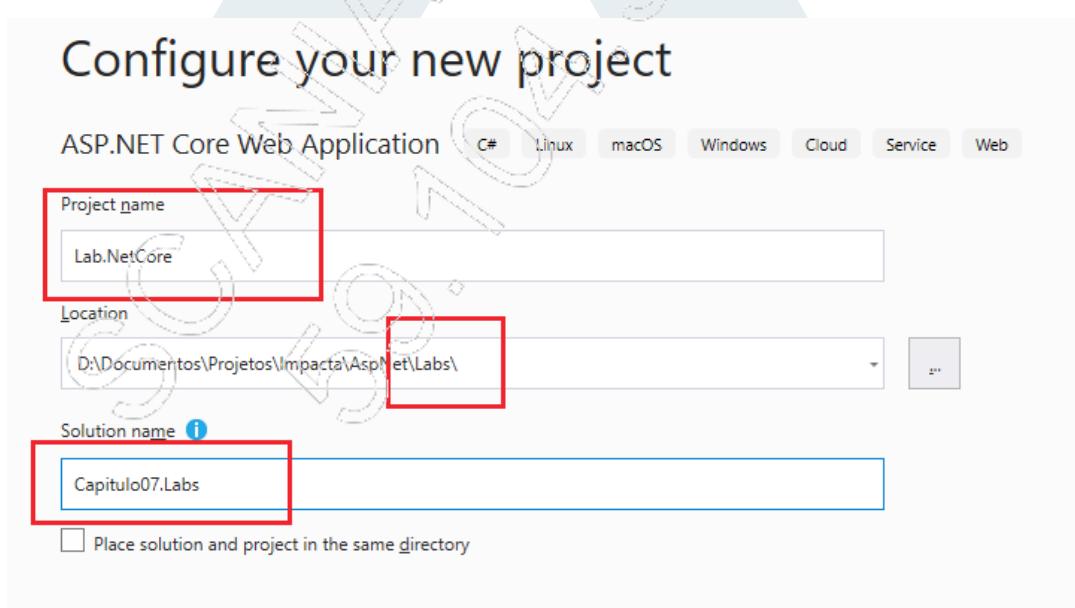
A partir deste capítulo, iniciaremos o desenvolvimento de um novo projeto visando contemplar os recursos do ASP.NET Core.

O projeto é sobre cadastro de eventos e participantes. Um administrador é responsável pelo cadastro de eventos. Podemos entender como eventos todas as atividades que permitem a presença de participantes, por exemplo: formatura, festa, um jogo de futebol etc. Alguns eventos são pagos e outros não, mas no cadastro será contemplado o preço do evento. De posse da lista de eventos, um participante se inscreve em um evento da sua escolha, fornecendo seus dados básicos. A aplicação permite que, a partir de um evento, a lista de participantes seja obtida.

Para que o participante efetue o pagamento, ele poderá usar um cartão de crédito. Uma segunda aplicação simulando uma administradora de cartões de crédito (similar àquela que fizemos no projeto finalizado no Capítulo 6) será criada, contemplando a parte que permite interação com aplicações externas, ou seja, a aplicação representará um *Webservice*, consumível pela nossa aplicação principal (o que permite a efetivação do pagamento).

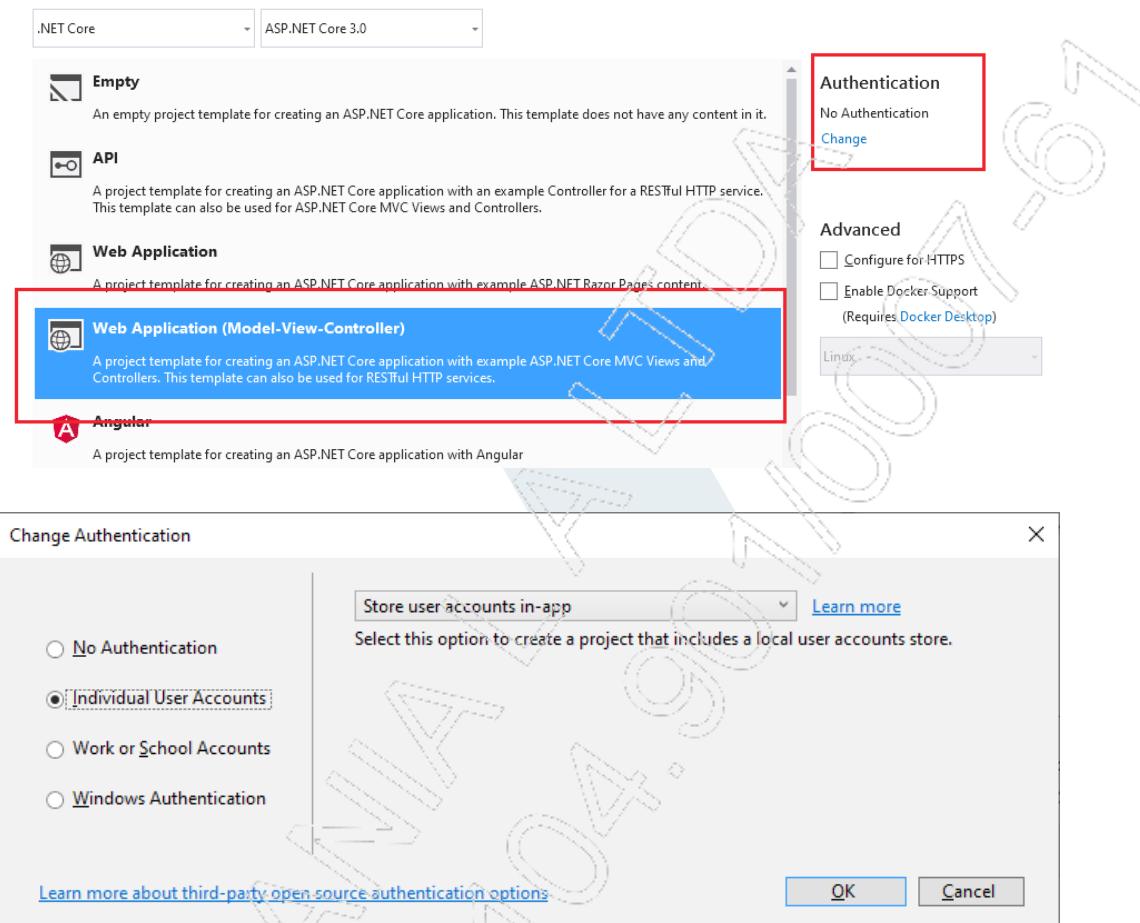
Laboratório 1

1. Na pasta **Labs**, crie um novo projeto **ASP.NET Core Web Application** chamado **Lab.NetCore**, em um solution chamado **Capitulo07.Labs**:

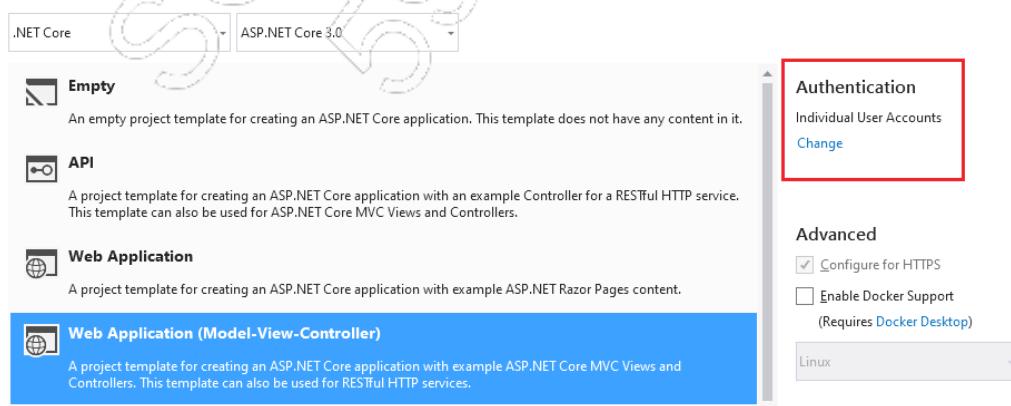


2. Na próxima etapa, selecione a opção **Web Application (Model-View-Controller)**. Adicione também o recurso para **Contas de Usuário Individuais (Individual User Accounts)**:

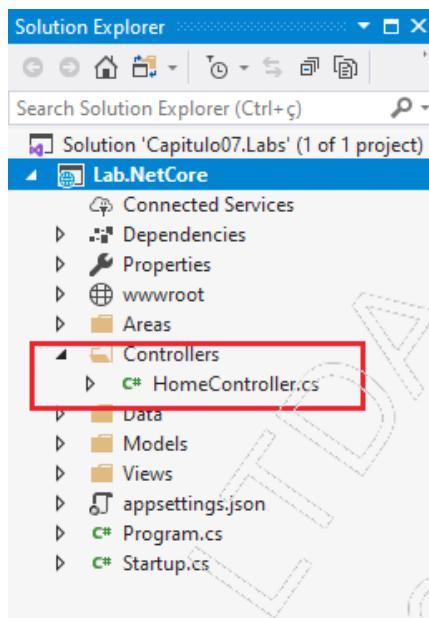
Create a new ASP.NET Core web application



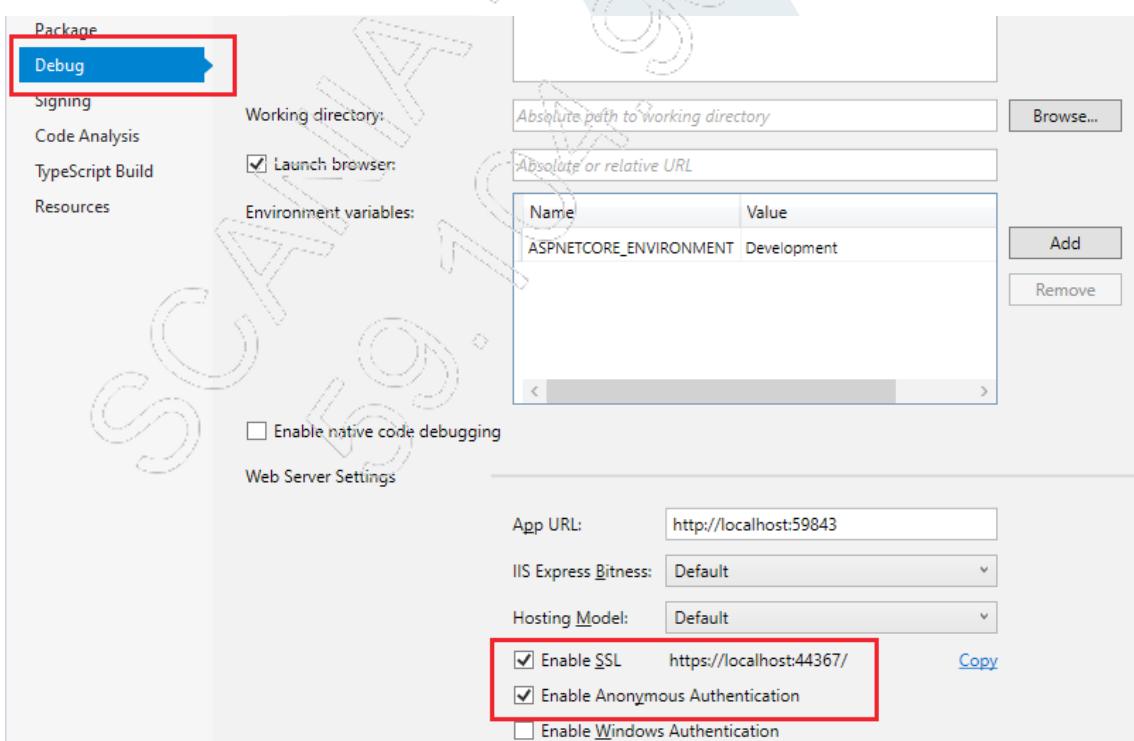
Create a new ASP.NET Core web application



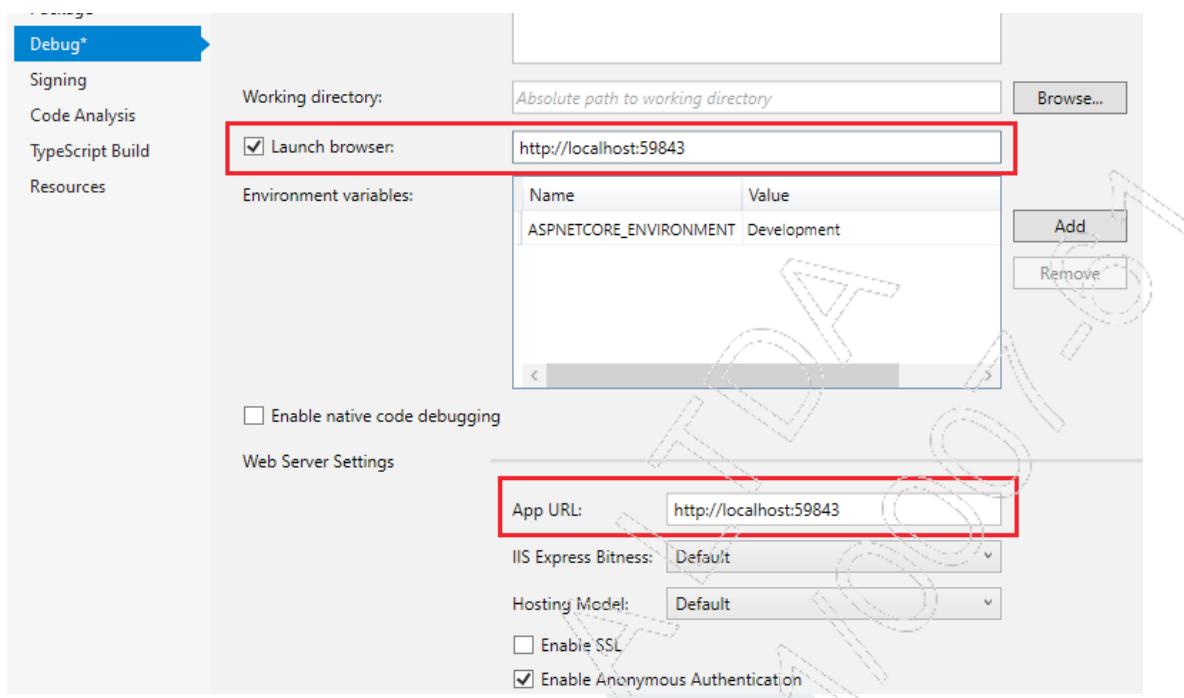
3. Observe que o projeto criado já possui o controller **HomeController.cs**:



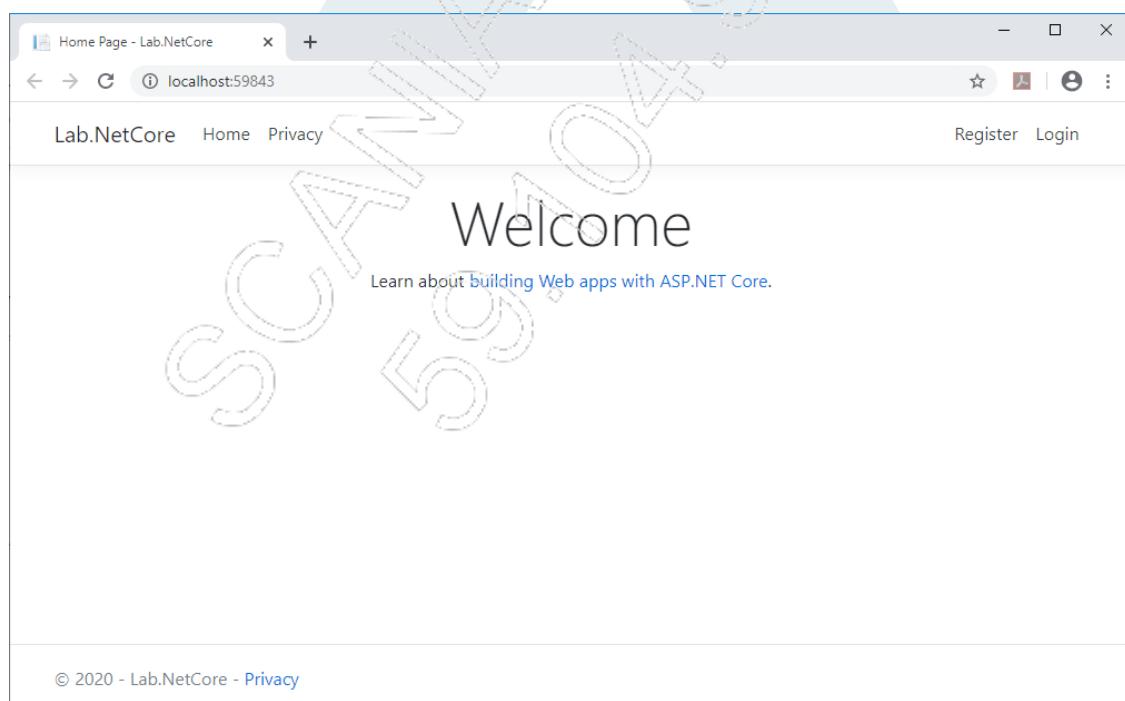
4. Na janela do Solution, dê um duplo-clique em **Properties**. Na tela que abrir, selecione a aba **Debug** (do lado esquerdo). Verifique se a opção **Enable SSL** está marcada:



5. Se estiver, desmarque a opção e copie o conteúdo do campo **App URL** para o campo **Launch browser**:



6. Salve o projeto e execute a aplicação. Provavelmente você terá um resultado semelhante ao mostrado a seguir:



7. Abra o arquivo de layout, localizado na pasta **Views/Shared**. Altere os valores destacados:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Eventos</title>
    <link rel="stylesheet"
        href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet"
        href("~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm
            navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home"
                    asp-action="Index">Eventos Impacta</a>
                <button class="navbar-toggler" type="button"
                    data-toggle="collapse"
                    data-target=".navbar-collapse"
                    aria-controls="navbarSupportedContent"
                    aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
                    flex-sm-row-reverse">
                    <partial name="_LoginPartial" />
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
                                asp-controller="Home"
                                asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
                                asp-controller="Eventos"
                                asp-action="Index">Eventos</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
                                asp-controller="Participantes"
                                asp-action="Index">Participantes</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
```

```
        @RenderBody()
    </main>
</div>

<footer class="border-top footer text-muted">
    <div class="container">
        &copy; @DateTime.Now.Year - Gestão de Eventos Impacta
    </div>
</footer>
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
@RenderSection("Scripts", required: false)
</body>
</html>
```

8. Abra o arquivo **Index.cshtml** na pasta **Views/Home**. Deixe seu conteúdo como mostrado no seguinte código:

```
@{
    ViewData["Title"] = "Home";
}

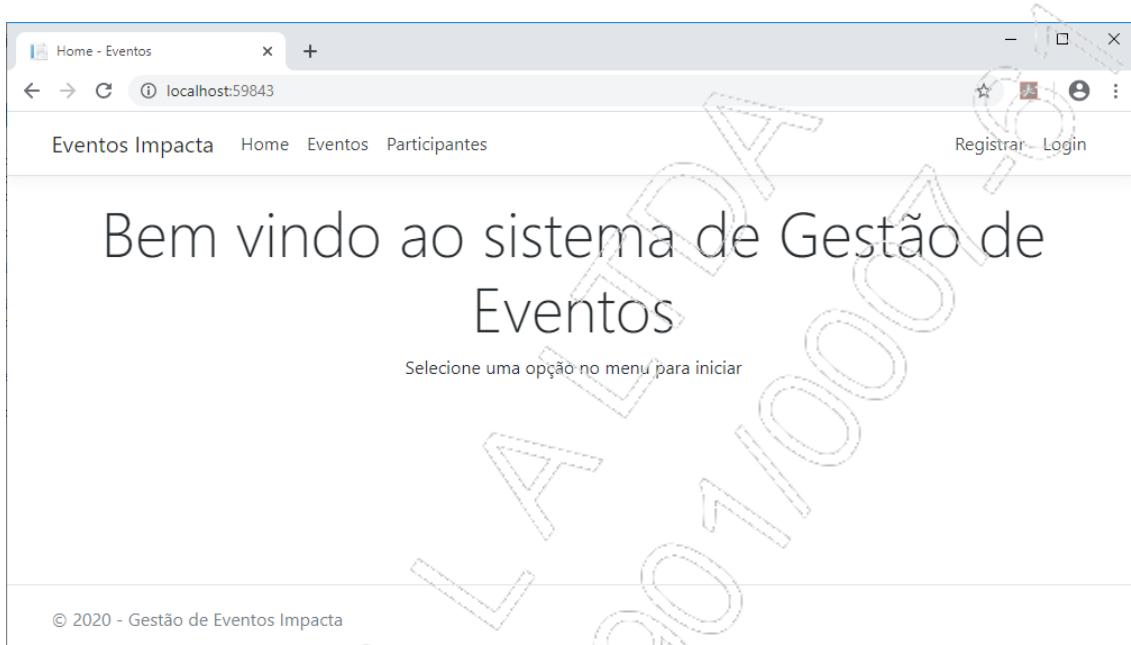
<div class="text-center">
    <h1 class="display-4">Bem vindo ao sistema de Gestão de Eventos</h1>
    <p>Selecione uma opção no menu para iniciar</p>
</div>
```

9. Abra o controller **HomeController**. Mantenha como o código a seguir:

```
using Microsoft.AspNetCore.Mvc;

namespace Lab.NetCore.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

10. Execute a aplicação novamente. O conteúdo deverá ser parecido com o mostrado a seguir:

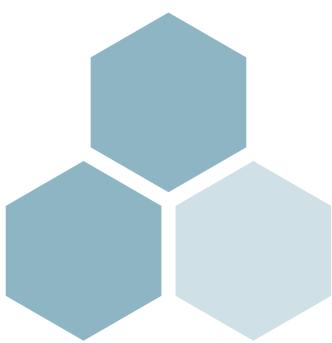


11. Remova o arquivo **Privacy.cshtml** da pasta **Views/Home**.

8

Entity Framework Core

- ◆ Entity Framework Core (EF Core);
- ◆ Database Providers;
- ◆ Usando o Entity Framework Core.



8.1. Introdução

O Entity Framework é parte integrante do framework, responsável pelo acesso a banco de dados. Não é o único, mas certamente é o mais simples e rápido de usar, além de ter uma estrutura otimizada que permite desde a criação até operações fundamentais sobre um banco de dados.

Já abordamos a estrutura do Entity Framework anteriormente, e o propósito deste capítulo é elucidar o conceito do Entity Framework Core (EF Core), comparado ao Entity Framework 6 (EF 6).

8.2. Entity Framework Core (EF Core)

Após a decisão de tornar o .NET Framework *cross-platform*, obtendo assim o .NET Core, houve também a necessidade de reimplementação do Entity Framework. Como resultado, foi criado o **EF Core**.

O Entity Framework Core foi desenvolvido para ser extensível, além de bastante leve, contendo apenas as operações necessárias. Ele utiliza apenas a abordagem **Code First**. A ideia é permitir seu uso em uma grande variedade de plataformas.

8.3. Database Providers

O EF Core é uma camada entre a aplicação e o banco de dados. É usado, portanto, para conectar o código com o banco.

Um provider é uma biblioteca cujo objetivo é estabelecer a conexão com um banco de dados específico. A instalação de um provider específico é realizada por meio da ferramenta **dotnet** ou do **NuGet**:

dotnet add package nome_provider

Ou:

install-package nome_provider

Alguns providers disponíveis estão listados a seguir:

- **Microsoft SQL Provider**

Este é o provider mais comumente usado pelo EF Core. Ele conecta a aplicação ao SQL Server.

- **SQLite Provider**

O SQLite é um banco de dados bastante popular e open-source. Podemos usar o SQLite em nossas aplicações com o EF Core.

- **InMemory Provider**

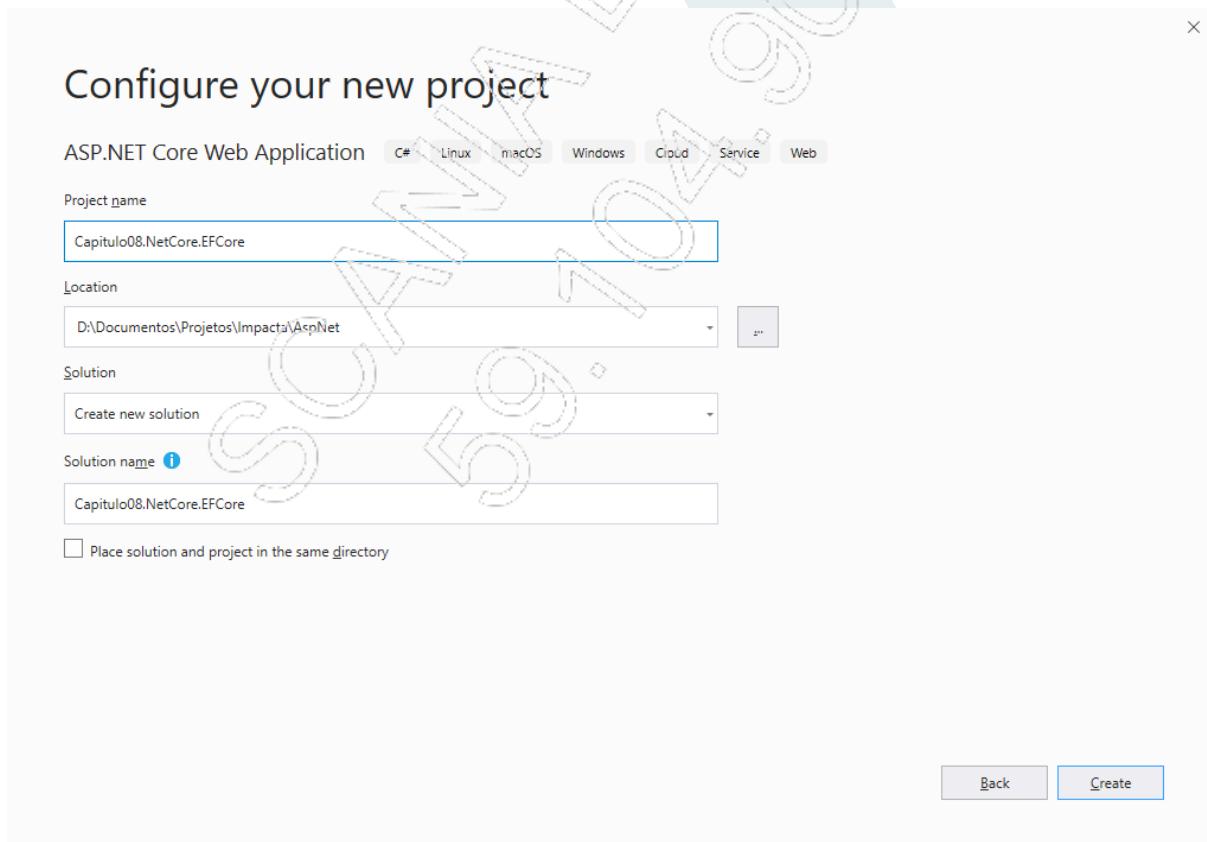
O EF Core adicionou o provider InMemory. É importante ter em mente que ele não conecta a aplicação a um banco de dados real. Ele é usado como uma ferramenta para testarmos a aplicação.

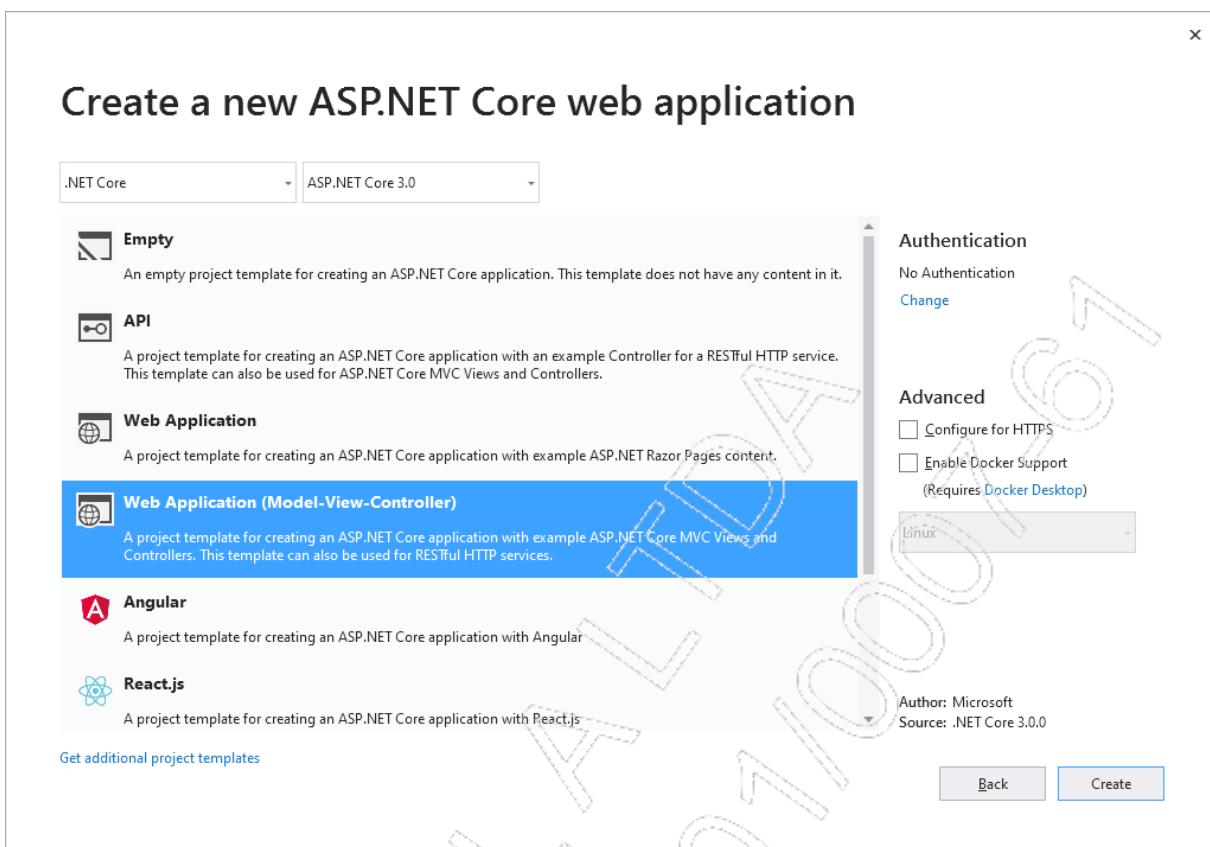
- **Outros providers**

Além dos providers mencionados, existe uma gama de outros providers disponíveis, como aqueles que permitem a conexão com MySQL, Maria DB ou DB2, por exemplo. No entanto, estes providers são gerenciados pelos próprios fabricantes, e sua obtenção deve ser realizada separadamente. Neste caso, a consulta à documentação é indispensável para sua configuração.

8.4. Usando o Entity Framework Core

Para estudarmos o uso do EF Core, vamos criar um novo projeto do tipo ASP.NET Core, template MVC. O projeto deve se chamar **Capítulo08.NetCore.EFCore**.





Já dissemos que o EF Core utiliza a abordagem Code First para a criação do banco de dados. Neste capítulo, apresentaremos os passos para a criação das classes necessárias à utilização do EF Core.

8.4.1. Definindo a entidade

Nossa entidade se chamará **Pessoa**. Vamos criá-la na pasta **Models**:

```
namespace Capitulo08.NetCore.EFCore.Models
{
    public class Pessoa
    {
        public int PessoaId { get; set; }
        public string Nome { get; set; }
        public string SobreNome { get; set; }
    }
}
```

8.4.2. Definindo o contexto

No EF Core, o contexto é usado para estabelecer o modo de acesso ao banco de dados, sem a necessidade de abstrações adicionais. Além de fornecer recursos para adicionar, remover, alterar ou buscar informações, ele realiza também as operações de abertura e fechamento da conexão.

Para adicionar um contexto à nossa aplicação, escreveremos uma classe herdada de `DbContext`. Nossa contexto se chamará `PessoasContext`:

```
using Microsoft.EntityFrameworkCore;

namespace Capitulo08.NetCore.EFCore.Models
{
    public class PessoasContext : DbContext
    {
    }
}
```

É necessário instalar, através do NuGet, o pacote **Microsoft.EntityFrameworkCore**.

8.4.3. Adicionando o DbSet ao contexto

Para representar a coleção de pessoas (objetos da classe `Pessoa`), devemos adicionar uma propriedade do tipo `DbSet` ao contexto:

```
using Microsoft.EntityFrameworkCore;

namespace Capitulo08.NetCore.EFCore.Models
{
    public class PessoasContext : DbContext
    {
        public DbSet<Pessoa> Pessoas { get; set; }
    }
}
```

8.4.4. Conectando o EF Core ao Provider SQL Server

O vínculo da aplicação ao banco de dados escolhido deve ser realizado através do seu correspondente Provider.

Para usarmos o contexto na aplicação, devemos registrar o serviço correspondente através do método `AddDbContext<>`, passando a classe referente ao contexto como elemento de parametrização. Este registro é realizado no método `ConfigureServices`, na classe `Startup`. Observe que a string de conexão será necessária:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<PessoasContext>(options =>
        options.UseSqlServer(Configuration
            .GetConnectionString("PessoasConnection")));

    services.AddControllersWithViews();
}
```

O método `UseSqlServer` está sendo usado para configurar o provider para o SQL Server. O seu parâmetro aponta para uma string de conexão cujo nome é `PessoasConnection`, que deverá ser definido no arquivo `appsettings.json`:

```
{
  "ConnectionStrings": {
    "PessoasConnection": "Integrated Security=SSPI;Persist Security
Info=False;Initial Catalog=DBPessoas;Data Source=.\\SQLEXPRESS"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

A configuração especificada no método **ConfigureServices** é recebida pelo contexto por injeção de dependência, através do seu construtor. Vamos alterá-lo para ilustrar este processo:

```
using Microsoft.EntityFrameworkCore;

namespace Capitulo08.NetCore.EFCore.Models
{
    public class PessoasContext : DbContext
    {
        public PessoasContext(DbContextOptions<PessoasContext> options)
            : base(options)
        { }

        public DbSet<Pessoa> Pessoas { get; set; }
    }
}
```

Se desejarmos que o banco de dados seja criado com uma estrutura estabelecida de forma diferente das entidades, devemos sobreescriver o método **OnModelCreating** da classe referente ao contexto. Este processo é similar ao que fizemos na utilização do EF 6.

```
using Microsoft.EntityFrameworkCore;

namespace Capitulo08.NetCore.EFCore.Models
{
    public class PessoasContext : DbContext
    {
        public PessoasContext(DbContextOptions<PessoasContext> options)
            : base(options)
        { }

        public DbSet<Pessoa> Pessoas { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Pessoa>().ToTable("TBPessoas");

            modelBuilder.Entity<Pessoa>()
                .Property(p => p.Nome)
                .IsRequired()
                .HasMaxLength(50);

            modelBuilder.Entity<Pessoa>()
                .Property(p => p.SobreNome)
                .IsRequired()
                .HasMaxLength(50);
        }
    }
}
```

8.4.5. Definindo o componente responsável por criar o banco de dados

Vamos agora definir o método a ser usado para criar o banco de dados.

Inclua no projeto uma pasta chamada **Dados**. Nesta, inclua uma classe estática chamada **DbInitializer**:

```
using Capitulo08.NetCore.EFCore.Models;

namespace Capitulo08.NetCore.EFCore.Dados
{
    public class DbInitializer
    {
        public static void Initialize(PessoasContext context)
        {
            context.Database.EnsureCreated();
        }
    }
}
```

A instrução `context.Database.EnsureCreated()`; cria o banco de dados, se este não existir.

8.4.6. Configurando a aplicação para a injeção de dependência

Sabemos que a classe **Program** contém o método **Main**, representando o ponto de partida de uma aplicação. Vamos alterar o código desta classe.

Usaremos um método de extensão disponível no namespace **Microsoft.Extensions.DependencyInjection** que deve ser importado. Nós definiremos a referência ao contexto da aplicação, obtida através do serviço definido em **Startup.cs**. Para que isso seja possível, devemos obter a referência ao provider correspondente. Para isso, deveremos realizar algumas alterações no método **Main**. Comecemos por executar o método **Run()** a partir de uma variável:

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

namespace Capitulo08.NetCore.EFCore
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            host.Run();
        }
    }
}
```

```
        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Em seguida, escreveremos o código adiante, usando a nova variável **host** como base:

```
using Capitulo08.NetCore.EFCore.Dados;
using Capitulo08.NetCore.EFCore.Models;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace Capitulo08.NetCore.EFCore
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services
                        .GetRequiredService<PessoasContext>();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services
                        .GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, ex.Message);
                    throw;
                }
            }

            host.Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Teste a aplicação até este ponto. Se tudo estiver correto, teremos nosso banco de dados criado, e a aplicação pronta para funcionar com este banco.

8.4.7. Preparando o controller para incluir e listar registros

Uma vez que o banco de dados tenha sido criado, já estamos prontos para testá-lo em nossa aplicação. Para isso, vamos adicionar um controller chamado **PessoasController**. Ele deverá ter os seguintes actions:

- **Incluir** (versões GET e POST)
- **Listar**

```
using Capitulo08.NetCore.EFCore.Models;
using Microsoft.AspNetCore.Mvc;

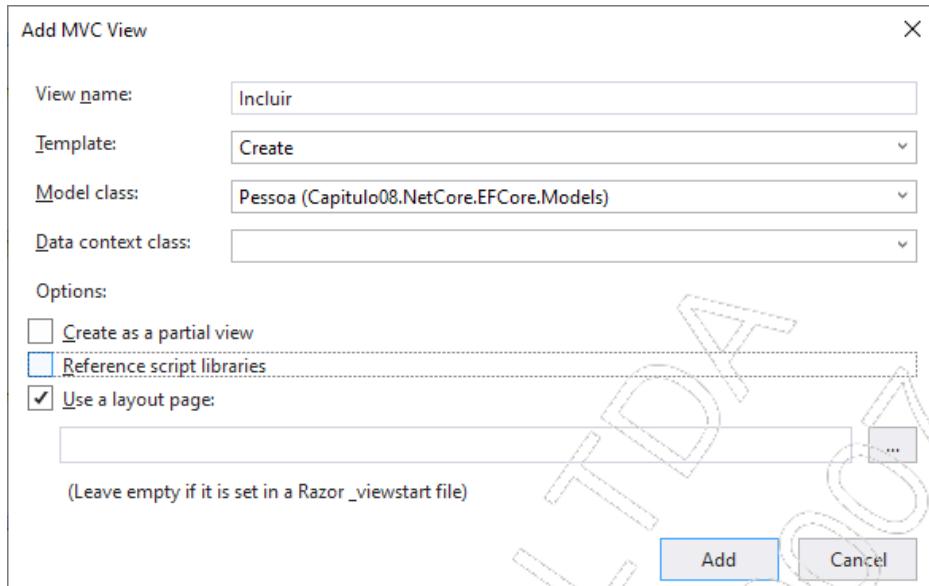
namespace Capitulo08.NetCore.EFCore.Controllers
{
    public class PessoasController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public IActionResult Incluir()
        {
            return View();
        }

        [HttpPost]
        public IActionResult Incluir(Pessoa pessoa)
        {
            return View();
        }

        public IActionResult Listar()
        {
            return View();
        }
    }
}
```

Usaremos a entidade **Pessoa** para gerar a view **Incluir**:



```
@model Capitulo08.NetCore.EFCore.Models.Pessoa
@{
    ViewData["Title"] = "Incluir";
}
<h1>Incluir</h1>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Incluir">
            <div asp-validation-summary="ModelOnly"
                class="text-danger"></div>
            @*<div class="form-group">
                <label asp-for="Pessoaid" class="control-label"></label>
                <input asp-for="Pessoaid" class="form-control" />
                <span asp-validation-for="Pessoaid"
                    class="text-danger"></span>
            </div>*@
            <div class="form-group">
                <label asp-for="Nome" class="control-label"></label>
                <input asp-for="Nome" class="form-control" />
                <span asp-validation-for="Nome" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="SobreNome" class="control-label"></label>
                <input asp-for="SobreNome" class="form-control" />
                <span asp-validation-for="SobreNome"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="INcluir" class="btn btn-info" />
            </div>
        </form>
    </div>
</div>
```

O contexto é passado para o controller por meio de injeção de dependência. Neste caso, devemos adicionar um construtor ao controller para que este seja recebido corretamente:

```
using Capitulo08.NetCore.EFCore.Models;
using Microsoft.AspNetCore.Mvc;

namespace Capitulo08.NetCore.EFCore.Controllers
{
    public class PessoasController : Controller
    {
        private PessoasContext Context { get; set; }
        public PessoasController(PessoasContext context)
        {
            this.Context = context;
        }

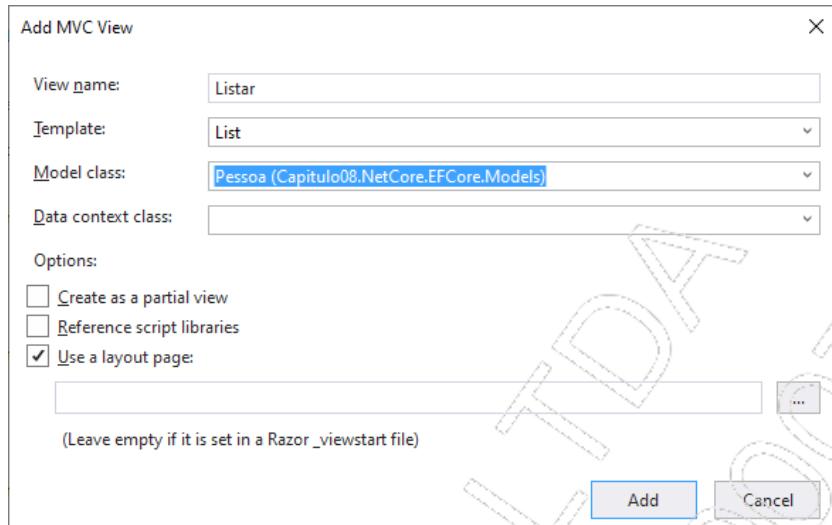
        public IActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public IActionResult Incluir()
        {
            return View();
        }

        [HttpPost]
        public IActionResult Incluir(Pessoa pessoa)
        {
            return View();
        }

        public IActionResult Listar()
        {
            return View();
        }
    }
}
```

Com isso, podemos codificar a versão POST do action **Incluir**, para adicionar o registro no banco de dados. Mas, antes, codificaremos o action **Listar**, com sua view. Após a inclusão, redirecionaremos o usuário para a lista de pessoas:



Foram realizadas algumas alterações na view resultante, de modo a possibilitar melhor visualização dos resultados para este propósito:

```
@model IEnumerable<Capitulo08.NetCore.EFCore.Models.Pessoa>

 @{
    ViewData["Title"] = "Listar";
}



# Listar



| @Html.DisplayNameFor(model => model.Pessoaid) | @Html.DisplayNameFor(model => model.Nome) | @Html.DisplayNameFor(model => model.SobreNome) |
|-----------------------------------------------|-------------------------------------------|------------------------------------------------|
|-----------------------------------------------|-------------------------------------------|------------------------------------------------|


```

```
<tbody>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.PessoaId)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Nome)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.SobreNome)
        </td>
    </tr>
}
</tbody>
</table>
```

A codificação dos actions **Incluir** e **Listar** ficarão, portanto, da seguinte forma:

```
using Capitulo08.NetCore.EFCore.Models;
using Microsoft.AspNetCore.Mvc;
using System.Linq;

namespace Capitulo08.NetCore.EFCore.Controllers
{
    public class PessoasController : Controller
    {
        private PessoasContext Context { get; set; }
        public PessoasController(PessoasContext context)
        {
            this.Context = context;
        }

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

        [HttpGet]
        public IActionResult Incluir()
        {
            return View();
        }
}
```

```
[HttpPost]
public IActionResult Incluir(Pessoa pessoa)
{
    Context.Add<Pessoa>(pessoa);
    Context.SaveChanges();

    return RedirectToAction("Listar");
}

public IActionResult Listar()
{
    return View(Context.Pessoas.ToList());
}
}
```

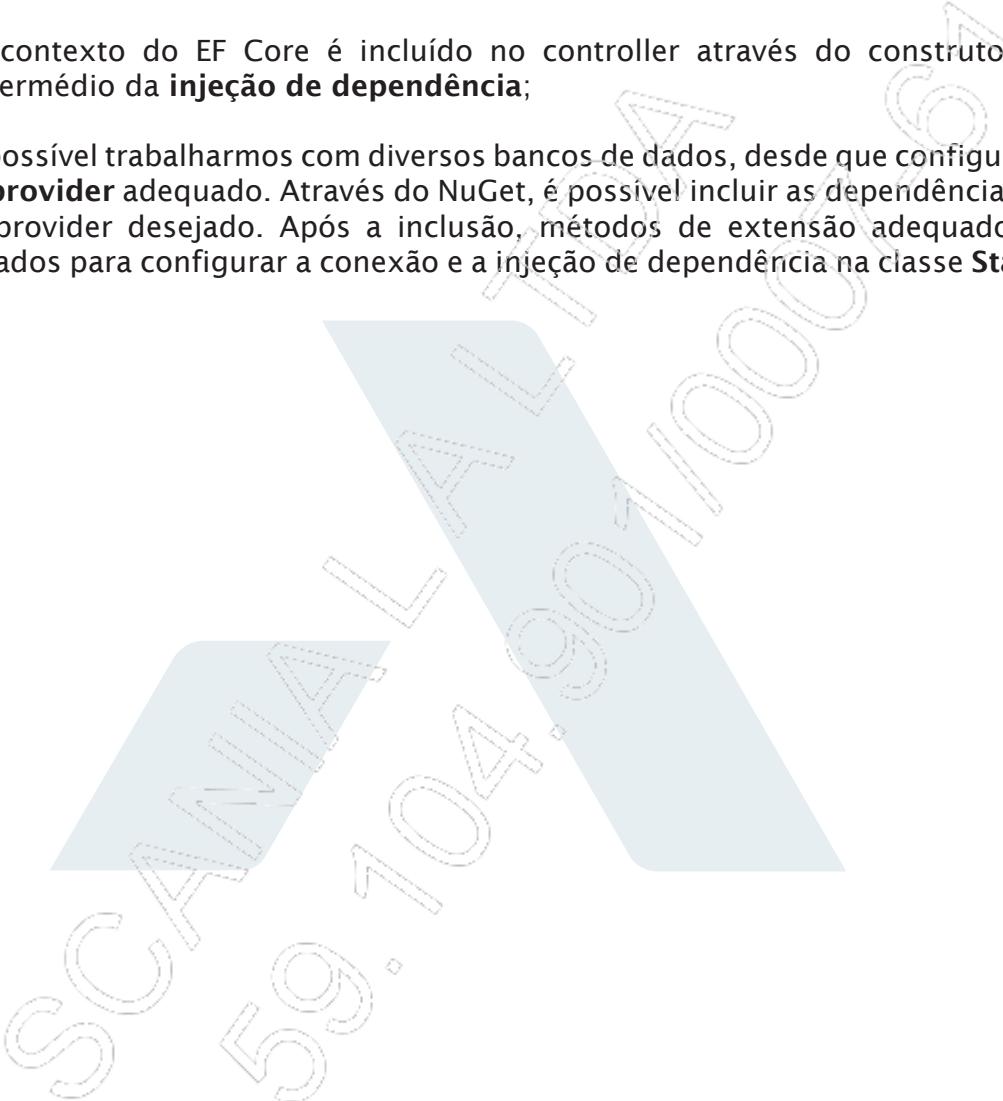
Vamos executar a aplicação e chamar o action **Incluir**. A partir dele, inclua uma nova pessoa e verifique seu resultado na listagem:

The image contains two screenshots of a web application interface. The top screenshot shows the 'Incluir' (Insert) page. It has fields for 'Nome' (Nome: Jose) and 'SobreNome' (SobreNome: Silva). A blue button labeled 'Incluir' is visible. The bottom screenshot shows the 'Listar' (List) page, which displays a table with one row: PessoaId 1, Nome Jose, and SobreNome Silva.

PessoaId	Nome	SobreNome
1	Jose	Silva

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

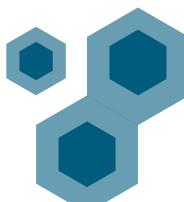
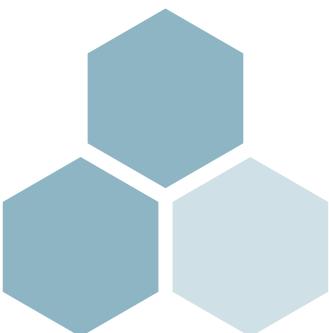
- Diferentemente do Entity Framework tradicional, o EF Core segue o modelo Code First por default;
 - O contexto do EF Core é incluído no controller através do construtor, por intermédio da **injeção de dependência**;
 - É possível trabalharmos com diversos bancos de dados, desde que configuremos o **provider** adequado. Através do NuGet, é possível incluir as dependências para o provider desejado. Após a inclusão, métodos de extensão adequados são usados para configurar a conexão e a injeção de dependência na classe **Startup**.
- 



8

Entity Framework Core

Teste seus conhecimentos



1. A abordagem adotada para a administração de um banco de dados no EF Core é:

- a) Code First
- b) Model First
- c) Service First
- d) Data First
- e) Database First

2. Para incluirmos um banco de dados diferente do SQL Server, devemos ter disponível:

- a) O arquivo de dados.
- b) O nome das tabelas em um arquivo JSON.
- c) O provider adequado.
- d) O sistema gerenciador do banco de dados.
- e) O token de acesso ao banco de dados.

3. Para que o serviço do EF Core seja adicionado no sistema, qual método devemos executar?

- a) AddSystem
- b) AddDatabase
- c) AddDbContext
- d) AddContext
- e) AddEntityFramework

4. A configuração especificada no método ConfigureServices é recebida pelo contexto no construtor por injeção de dependência através de um parâmetro do tipo:

- a) DbContextOptions
- b) DbContext
- c) Options
- d) DependencyInjection
- e) DblInjection

5. Podemos usar a seguinte instrução para garantir que um banco de dados seja criado, caso não exista:

- a) DbContext.Create();
- b) contexto.CreateAsync();
- c) DbSet.Create();
- d) context.Database.EnsureCreated();
- e) Database.CreateDatabase();

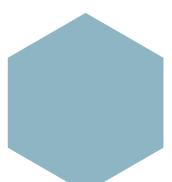


8

Entity Framework Core



Mãos à obra!



Objetivos:

Vamos dar continuidade ao laboratório desenvolvido no Capítulo 7. Aplicaremos os recursos de acesso a dados com o Entity Framework Core.

Laboratório 1

1. Na pasta **Labs**, crie um novo solution (**blank solution**) chamado **Capítulo08.Labs**;
2. Copie a pasta **Lab.NetCore** do solution **Capítulo07.Labs** para a pasta do novo solution **Capítulo08.Labs**;
3. Adicione o projeto copiado ao solution recém-criado;
4. Execute a aplicação para ver se está tudo funcionando corretamente. Criaremos um banco de dados chamado **DB_EVENTOS**, com duas tabelas: **TBEVENTOS** e **TBParticipantes**. O banco de dados será descrito na string de conexão e as tabelas serão geradas a partir das entidades;
5. Na pasta **Models** (se não existir, crie-a), inclua as duas classes mostradas adiante. Essas classes são as entidades a serem manipuladas na aplicação:

```
using System;
using System.Collections.Generic;

namespace Lab.NetCore.Models
{
    public class Evento
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
        public DateTime Data { get; set; }
        public string Local { get; set; }
        public double Preco { get; set; }

        public ICollection<Participante> Participantes { get; set; }
    }
}
```

```
using System;

namespace Lab.AspNetCore.Models
{
    public class Participante
    {
        public int Id { get; set; }
        public int EventoInfoId { get; set; }
        public string Nome { get; set; }
        public string Email { get; set; }
        public string Cpf { get; set; }
        public DateTime DataNascimento { get; set; }

        public Evento EventoInfo { get; set; }
    }
}
```

6. No projeto, crie uma pasta chamada **Dao**. Nessa pasta, defina o contexto do banco de dados, **EventosContext**:

```
using Lab.AspNetCore.Models;
using Microsoft.EntityFrameworkCore;

namespace Lab.AspNetCore.Dao
{
    public class EventosContext : DbContext
    {
        public EventosContext(DbContextOptions<EventosContext>
            opcoes) : base(opcoes)
        { }

        public DbSet<Evento> Eventos { get; set; }
        public DbSet<Participante> Participantes { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<Evento>().ToTable("TBEventos");
            modelBuilder.Entity<Evento>()
                .Property(p => p.Data)
                .IsRequired();
            modelBuilder.Entity<Evento>()
                .Property(p => p.Descricao)
                .IsRequired()
                .HasMaxLength(50);
            modelBuilder.Entity<Evento>()
                .Property(p => p.Local)
                .IsRequired()
                .HasMaxLength(50);
            modelBuilder.Entity<Evento>()
                .Property(p => p.Preco)
                .IsRequired();
        }
}
```

```
modelBuilder.Entity<Participante>().ToTable("TBPARTICIPANTES");
modelBuilder.Entity<Participante>()
    .Property(p => p.Cpf)
    .IsRequired()
    .HasMaxLength(11);
modelBuilder.Entity<Participante>()
    .Property(p => p.Nome)
    .IsRequired()
    .HasMaxLength(50);
modelBuilder.Entity<Participante>()
    .Property(p => p.Email)
    .IsRequired()
    .HasMaxLength(60);
modelBuilder.Entity<Participante>()
    .Property(p => p.DataNascimento)
    .IsRequired();
}
}
}
```

Importe os namespaces necessários. A entidade **Participantes** poderia ter sido omitida, uma vez que existe um relacionamento entre elas. As tabelas serão criadas com o mesmo nome das entidades, porém em muitos casos é conveniente alterá-las, como fizemos na classe;

7. Vamos agora registrar a injeção de dependência. Abra o arquivo **Startup.cs**. Inclua a instrução a seguir no método **ConfigureServices**:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<EventosContext>(options => options.UseSqlServer(
        Configuration.GetConnectionString("EventosConnection")));

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => options
        .SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

8. Agora, vamos abrir o arquivo **appsettings.json**. Configure a string de conexão como indicado a seguir (você deve alterá-la para refletir um banco de dados disponível na sua máquina ou na sua rede). Deixe como está mostrado a seguir:

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Integrated Security=SSPI;Persist Security  
Info=False;Initial Catalog=DB_IDENTITY;Data Source=.\SQLEXPRESS",  
    "EventosConnection": "Integrated Security=SSPI;Persist Security  
Info=False;Initial Catalog=DB_EVENTOS;Data Source=.\SQLEXPRESS"  
  },  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    }  
  },  
  "AllowedHosts": "*"  
}
```

9. Na pasta **Dao**, inclua uma classe estática chamada **DbInitializer**:

```
namespace Lab.NetCore.Dao  
{  
  public class DbInitializer  
  {  
    public static void Initialize(EventosContext context)  
    {  
      context.Database.EnsureCreated();  
    }  
  }  
}
```

A instrução `context.Database.EnsureCreated();` cria o banco de dados, se este não existir.

10. Sabemos que a classe **Program.cs** contém o método **Main()**, representando o ponto de partida de uma aplicação. Vamos alterar o código desta classe. Usaremos um método de extensão disponível no namespace **Microsoft.Extensions.DependencyInjection** que deve ser importado. Em seguida, deixe o método **Main()** como no modelo adiante:

```
using Lab.NetCore.Dao;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

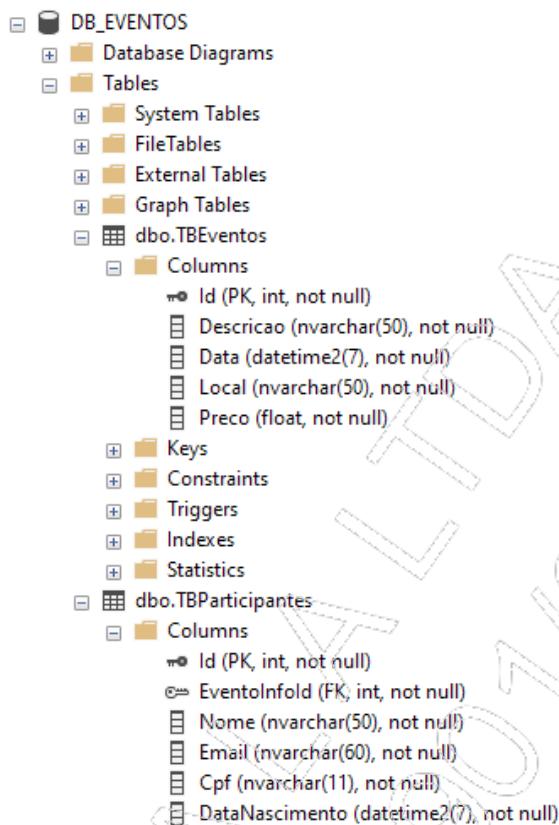
namespace Lab.NetCore
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<EventosContext>();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, ex.Message);
                    throw;
                }
            }

            host.Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Teste a aplicação até este ponto. Se tudo estiver correto, teremos nosso banco de dados criado, e a aplicação pronta para funcionar com este banco.



B - Definindo controllers, actions e views

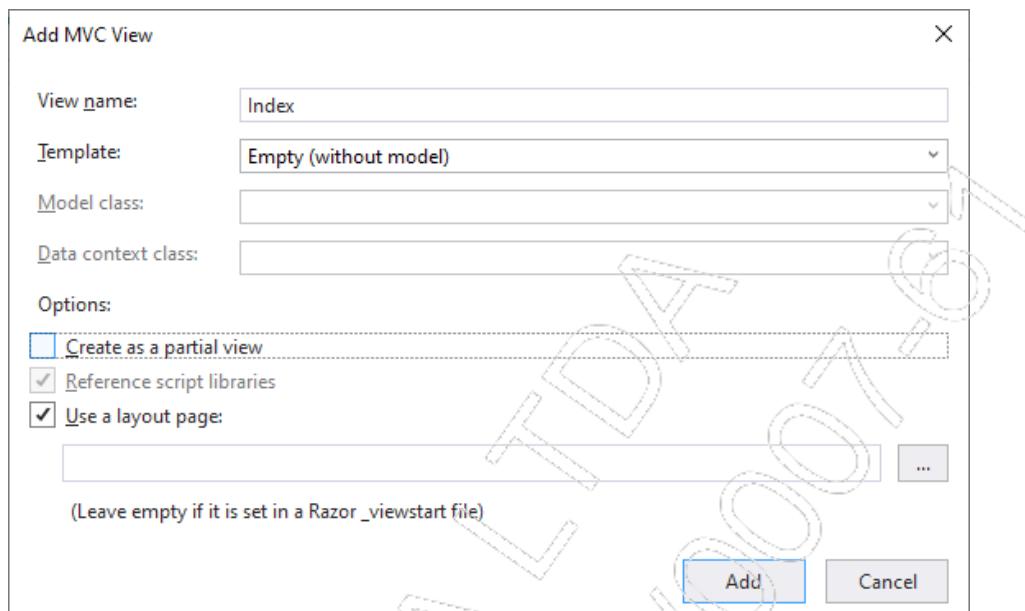
11. Neste momento, vamos iniciar a parte fundamental da nossa aplicação: a elaboração de novos controllers. Na pasta **Controllers**, insira um novo controller vazio, **modelo MVC**, chamado **Eventos** (classe **EventosController**):

```

using Microsoft.AspNetCore.Mvc;

namespace Lab.NetCore.Controllers
{
    public class EventosController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
  
```

12. A partir do action **Index**, crie uma view sem modelo, com layout. Deixe com o conteúdo a seguir:



```
@{
    ViewData["Title"] = "Gestão de Eventos";
}

<h2>Gestão de Eventos</h2>
<ul>
    <li>
        <a asp-controller ="Eventos" asp-action="IncluirEvento">
            Incluir Novo Evento</a>
    </li>
    <li>
        <a asp-controller ="Eventos" asp-action="ListarEventos">
            Listar Eventos</a>
    </li>
</ul>
```

13. No controller **EventosController**, adicione os actions **IncluirEvento** e **ListarEventos**:

```
using Microsoft.AspNetCore.Mvc;

namespace Lab.AspNetCore.Controllers
{
    public class EventosController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult IncluirEvento()
        {
            return View();
        }

        public IActionResult ListarEventos()
        {
            return View();
        }
    }
}
```

14. Para permitir o acesso ao banco de dados por meio da aplicação, criaremos uma estrutura otimizada, tentando unificar as operações comuns em poucos métodos, pois o **Entity Framework Core** nos permitirá fazer isso. Na pasta **Dao**, crie uma classe chamada **GenericDao** com o seguinte conteúdo:

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Lab.AspNetCore.Dao
{
    public enum TipoOperacaoBD
    {
        Detached = 0,
        Unchanged = 1,
        Deleted = 2,
        Modified = 3,
        Added = 4
    }
}
```

```
public abstract class GenericDao<T> where T: class
{
    private EventosContext Context { get; set; }

    public GenericDao(EventosContext context)
    {
        this.Context = context;
    }

    public void Executar(T item, TipoOperacaoBD tipo)
    {
        Context.Entry<T>(item).State = ( EntityState )tipo;
        Context.SaveChanges();
    }

    public IEnumerable<T> Listar()
    {
        return Context.Set<T>().ToList();
    }

    public T BuscarPorId(int id)
    {
        return Context.Set<T>().Find(id);
    }
}
```

15. Para lidar com as operações relacionadas a eventos, crie a classe **EventosDao** na pasta **Dao**. Esta classe deve ser subclasse da **GenericDao**:

```
using Lab.NetCore.Models;

namespace Lab.NetCore.Dao
{
    public class EventosDao : GenericDao<Evento>
    {
        public EventosDao(EventosContext context)
            : base(context)
        {
        }
    }
}
```

16. Vamos alterar o controller **EventosController** para receber o contexto por injeção de dependência. Implemente os actions **IncluirEvento** (POST) e **ListarEventos** deste controller:

```
using Lab.NetCore.Dao;
using Lab.NetCore.Models;
using Microsoft.AspNetCore.Mvc;

namespace Lab.NetCore.Controllers
{
    public class EventosController : Controller
    {
        private EventosDao eventosDao { get; set; }

        //EventosContext é passado por injeção de dependência
        public EventosController(EventosContext context)
        {
            this.eventosDao = new EventosDao(context);
        }

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

        public IActionResult IncluirEvento()
        {
            return View();
        }

        [HttpPost]
        public IActionResult IncluirEvento(Evento evento)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            eventosDao.Executar(evento, TipoOperacaoBD.Added);
            return RedirectToAction("ListarEventos");
        }

        public IActionResult ListarEventos()
        {
            var lista = eventosDao.Listar();
            return View(lista);
        }
    }
}
```

17. Para que o modelo seja validado, e que a instrução `ModelState.IsValid` retorne `true`, vamos alterar as entidades para incluir os validadores:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Lab.AspNetCore.Models
{
    public class Evento
    {
        public int Id { get; set; }

        [Required]
        [Display(Name = "Descrição")]
        public string Descricao { get; set; }

        [Required]
        [Display(Name = "Data do evento")]
        [DataType(DataType.Date)]
        public DateTime Data { get; set; }

        [Required]
        public string Local { get; set; }

        [Required]
        [DataType(DataType.Currency)]
        public double Preco { get; set; }

        public ICollection<Participante> Participantes { get; set; }
    }
}

using System;
using System.ComponentModel.DataAnnotations;

namespace Lab.AspNetCore.Models
{
    namespace Lab.AspNetCore.Models
    {
        public class Participante
        {
            public int Id { get; set; }

            [Display(Name = "Evento")]
            public int EventoInfoId { get; set; }

            [Required]
            public string Nome { get; set; }
        }
    }
}
```

```
[Required]
[EmailAddress]
public string Email { get; set; }

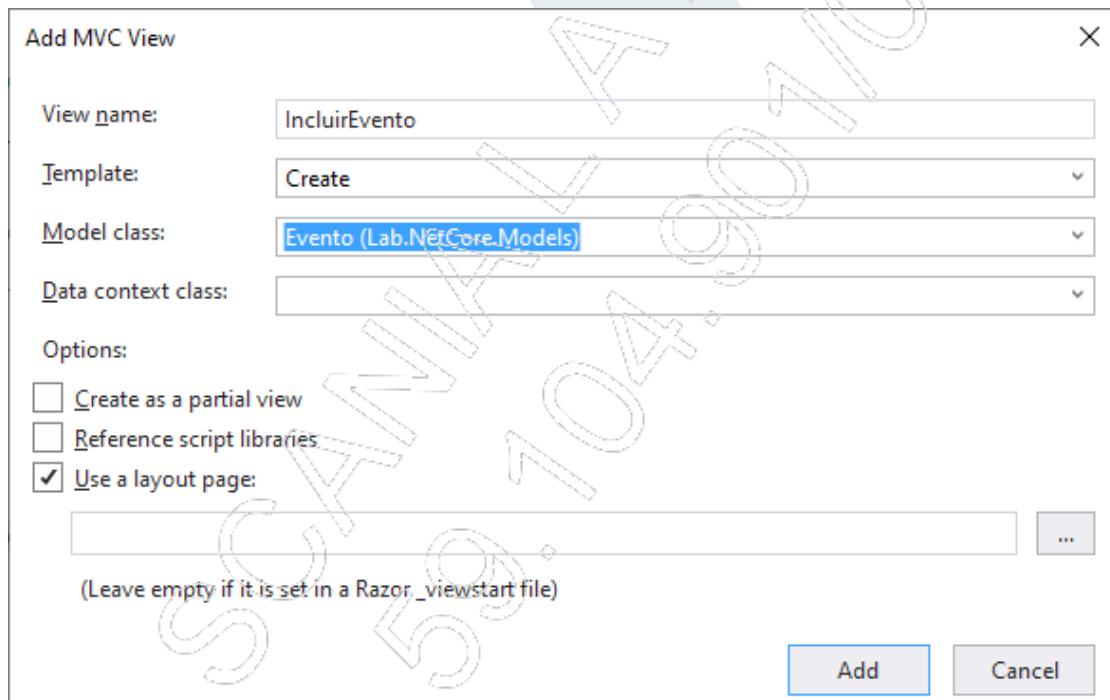
[Required]
[StringLength(11, MinimumLength = 11)]
public string Cpf { get; set; }

[DataType(DataType.Date)]
[Display(Name = "Data Nascimento")]
public DateTime DataNascimento { get; set; }

public Evento EventoInfo { get; set; }

}
}}
```

18. No action **IncluirEvento**, defina a view com o layout e com o modelo **Evento**:



Faça as alterações necessárias, deixando a view como no código a seguir:

```
@model Lab.NetCore.Models.Evento

 @{
    ViewData["Title"] = "Incluir Evento";
}

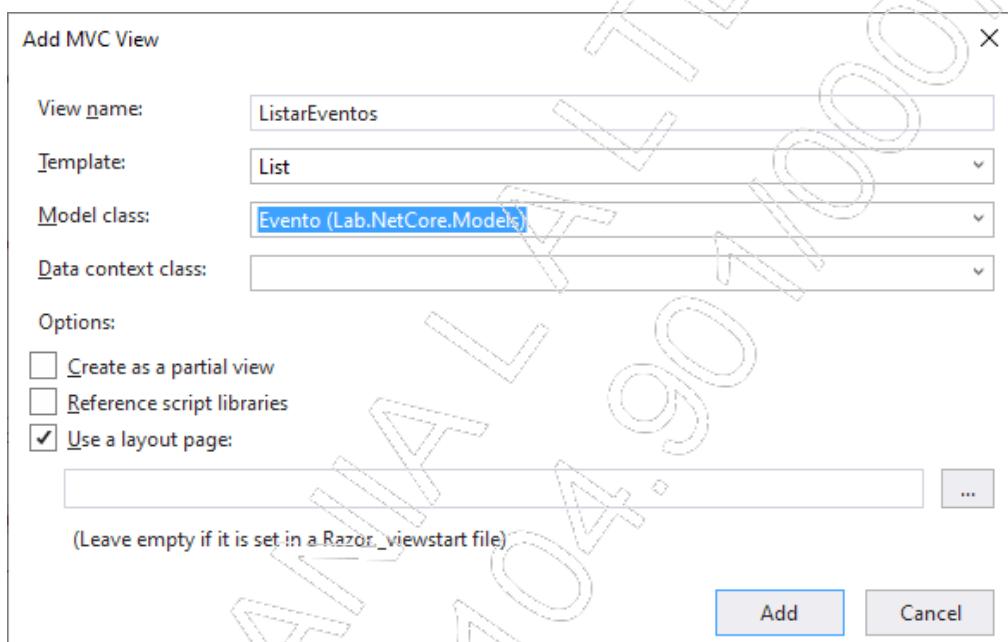
<h1>Incluir Evento</h1>

<h4>Incluir Evento</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="IncluirEvento">
            <div asp-validation-summary="ModelOnly"
                class="text-danger"></div>
            @*<div class="form-group">
                <label asp-for="Id" class="control-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id"
                    class="text-danger"></span>
            </div>*@
            <div class="form-group">
                <label asp-for="Descricao"
                    class="control-label"></label>
                <input asp-for="Descricao"
                    class="form-control" />
                <span asp-validation-for="Descricao"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Data"
                    class="control-label"></label>
                <input asp-for="Data"
                    class="form-control" />
                <span asp-validation-for="Data"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Local"
                    class="control-label"></label>
                <input asp-for="Local"
                    class="form-control" />
                <span asp-validation-for="Local"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Preco"
                    class="control-label"></label>
                <input asp-for="Preco"
                    class="form-control" />
                <span asp-validation-for="Preco"
                    class="text-danger"></span>
            </div>
        </form>
    </div>
</div>
```

```
</div>
<div class="form-group">
    <input type="submit" value="Incluir"
           class="btn btn-primary" />
</div>
</form>
</div>
</div>

<div>
    <a asp-action="Index">Voltar para o menu</a>
</div>
```

19. Adicione a view para a listagem de eventos:



```
@model IEnumerable<Lab.NetCore.Models.Evento>
@{
    ViewData["Title"] = "Listar Eventos";
}



# Listar Eventos



<a asp-action="IncluirEvento">Novo Evento</a>



|--|


```

```
        @Html.DisplayNameFor(model => model.Descricao)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Data)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Local)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Preco)
    </th>
    <th></th>
</tr>
</thead>
<tbody>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Descricao)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Data)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Local)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Preco)
        </td>
        <td>
            @Html.ActionLink("Alterar", "AlterarEvento",
                new { id=item.Id }) |
            @Html.ActionLink("Remover", "RemoverEvento",
                new { id=item.Id })
        </td>
    </tr>
}
</tbody>
</table>
```

20. Execute a aplicação, incluindo eventos:

Incluir Evento

Descrição	Palestra Microsoft
Data do evento	25/03/2020
Local	Impacta
Preco	R\$150,00

[Incluir](#)

[Voltar para o menu](#)

© 2020 - Gestão de Eventos Impacta

Listar Eventos

Novo Evento	Descrição	Data do evento	Local	Preco	Ações
	Palestra Microsoft	25/03/2020	Impacta	R\$150,00	Alterar Remover

© 2020 - Gestão de Eventos Impacta

21. Na listagem de registros, deixamos um link para **Alterar** um registro. É este action que definiremos agora. Inclua as versões GET e POST para o action **AlterarEvento**:

```
using Lab.NetCore.Dao;
using Lab.NetCore.Models;
using Microsoft.AspNetCore.Mvc;

namespace Lab.NetCore.Controllers
{
    public class EventosController : Controller
    {
        private EventosDao eventosDao { get; set; }

        //EventosContext é passado por injeção de dependência
        public EventosController(EventosContext context)
        {
            this.eventosDao = new EventosDao(context);
        }

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

        public IActionResult IncluirEvento()
        {
            return View();
        }

        [HttpPost]
        public IActionResult IncluirEvento(Evento evento)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            eventosDao.Executar(evento, TipoOperacaoBD.Added);
            return RedirectToAction("ListarEventos");
        }

        public IActionResult ListarEventos()
        {
            var lista = eventosDao.Listar();
            return View(lista);
        }
    }
}
```

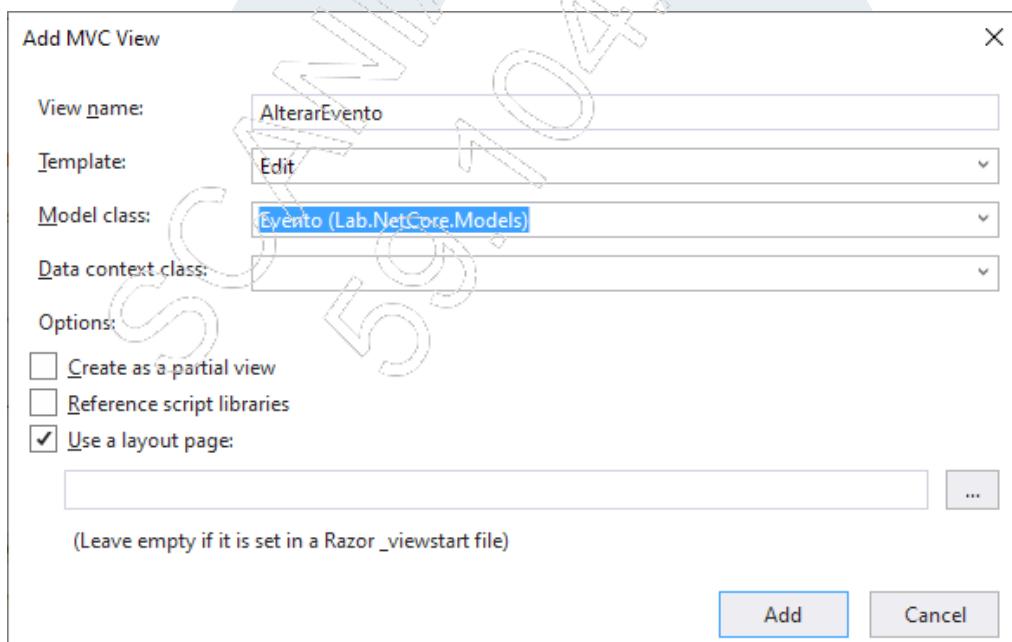
```
//action HTTP Get Comum
private IActionResult ExecutarEvento(int id, string viewName)
{
    var evento = eventosDao.BuscarPorId(id);

    if (evento == null)
    {
        ViewData["MensagemErro"] = "Nenhum evento encontrado";
        return View("_Erro");
    }
    return View(viewName, evento);
}

//Alterando o evento
[HttpGet]
public IActionResult AlterarEvento(int id)
{
    return ExecutarEvento(id, "AlterarEvento");
}

[HttpPost]
public IActionResult AlterarEvento(Evento evento)
{
    eventosDao.Executar(evento, TipoOperacaoBD.Modified);
    return RedirectToAction("ListarEventos");
}
}
```

22. Adicione a view AlterarEvento:



```
@model Lab.NetCore.Models.Evento

 @{
     ViewData["Title"] = "Alterar Evento";
 }

<h1>Alterar Evento</h1>

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="AlterarEvento">
            <div asp-validation-summary="ModelOnly"
                class="text-danger"></div>
            @*<div class="form-group">
                <label asp-for="Id" class="control-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id"
                    class="text-danger"></span>
            </div>*@
            <div class="form-group">
                <label asp-for="Descricao"
                    class="control-label"></label>
                <input asp-for="Descricao"
                    class="form-control" />
                <span asp-validation-for="Descricao"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Data"
                    class="control-label"></label>
                <input asp-for="Data"
                    class="form-control" />
                <span asp-validation-for="Data"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Local"
                    class="control-label"></label>
                <input asp-for="Local"
                    class="form-control" />
                <span asp-validation-for="Local"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Preco"
                    class="control-label"></label>
                <input asp-for="Preco"
                    class="form-control" />
                <span asp-validation-for="Preco"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Alterar"
                    class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

```
        </form>
    </div>
</div>

<div>
    <a asp-action="ListarEventos">Voltar para a lista</a>
</div>
```

23. Adicione e remova eventos, para testar;

24. Vamos remover um registro, mas lembrando que, para remover, é importante solicitarmos uma confirmação do usuário. No controller, inclua o action **RemoverEvento** (GET e POST):

```
using Lab.NetCore.Dao;
using Lab.NetCore.Models;
using Microsoft.AspNetCore.Mvc;

namespace Lab.NetCore.Controllers
{
    public class EventosController : Controller
    {
        private EventosDao eventosDao { get; set; }

        //EventosContext é passado por injeção de dependência
        public EventosController(EventosContext context)
        {
            this.eventosDao = new EventosDao(context);
        }

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

        public IActionResult IncluirEvento()
        {
            return View();
        }

        [HttpPost]
        public IActionResult IncluirEvento(Evento evento)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            eventosDao.Executar(evento, TipoOperacaoBD.Added);
            return RedirectToAction("ListarEventos");
        }
    }
}
```

```
public IActionResult ListarEventos()
{
    var lista = eventosDao.Listar();
    return View(lista);
}

//action HTTP Get Comum
private IActionResult ExecutarEvento(int id, string viewName)
{
    var evento = eventosDao.BuscarPorId(id);

    if (evento == null)
    {
        ViewData["MensagemErro"] = "Nenhum evento encontrado";
        return View("_Erro");
    }
    return View(viewName, evento);
}

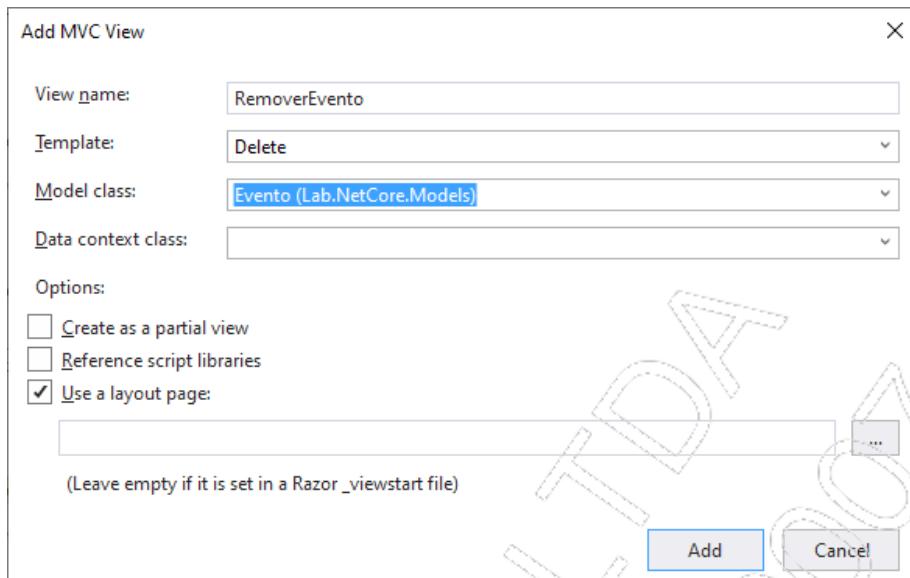
//Alterando o evento
[HttpGet]
public IActionResult AlterarEvento(int id)
{
    return ExecutarEvento(id, "AlterarEvento");
}

[HttpPost]
public IActionResult AlterarEvento(Evento evento)
{
    eventosDao.Executar(evento, TipoOperacaoBD.Modified);
    return RedirectToAction("ListarEventos");
}

//removendo um evento
[HttpGet]
public IActionResult RemoverEvento(int id)
{
    return ExecutarEvento(id, "RemoverEvento");
}

[HttpPost]
public IActionResult RemoverEvento(Evento evento)
{
    eventosDao.Executar(evento, TipoOperacaoBD.Deleted);
    return RedirectToAction("ListarEventos");
}
}
```

25. Crie a view RemoverEvento:



E deixe a view com o conteúdo a seguir:

```
@model Lab.NetCore.Models.Evento

 @{
    ViewData["Title"] = "Remover Evento";
}

<h1 class="text-danger">Confirmação</h1>
<h3 class="text-danger">
    Tem certeza que deseja remover este evento?
</h3>

<div>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Descricao)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Descricao)
        </dd>
    </dl>
    <form asp-action="RemoverEvento">
        <input type="hidden" asp-for="Id" />
        <input type="submit" value="Sim, remover"
               class="btn btn-danger" />
        <a asp-action="ListarEventos">Não, voltar para a lista</a>
    </form>
</div>
```

26. Teste a alteração e a remoção;

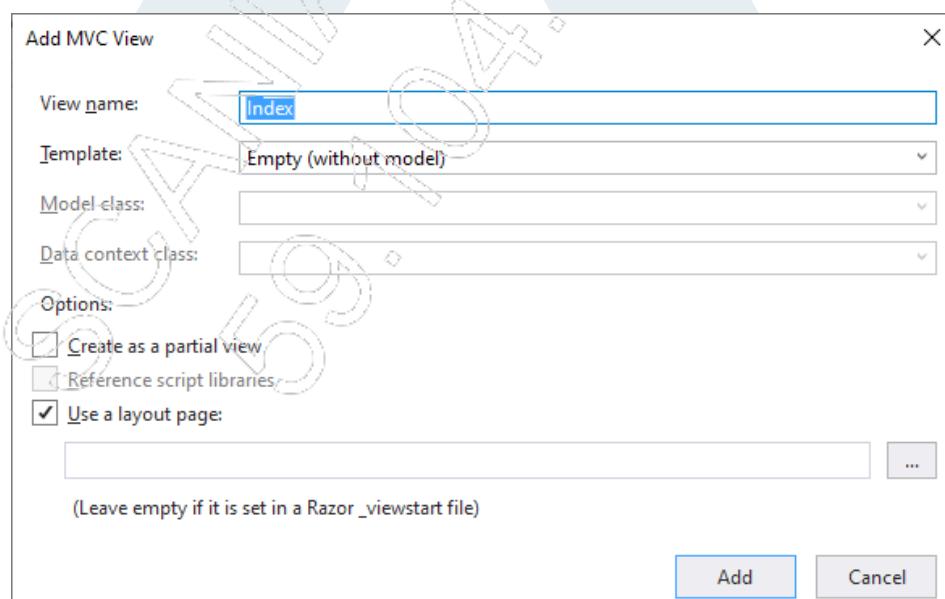
27. Da mesma forma que pudemos incluir e listar eventos, vamos fazer o mesmo com os participantes. Neste caso, temos duas características distintas:

- Para cadastrar um participante, temos que fornecer também a lista de eventos para a view;
- Para listar os participantes, temos que considerar a lista de participantes por evento, que será selecionado em um componente **DropDownList**.

Crie um novo controller chamado **ParticipantesController**:

```
using Microsoft.AspNetCore.Mvc;  
  
namespace Lab.NetCore.Controllers  
{  
    public class ParticipantesController : Controller  
    {  
        public IActionResult Index()  
        {  
            return View();  
        }  
    }  
}
```

28. Defina a view para o action **Index**:



```
@{  
    ViewData["Title"] = "Index";  
}  
  
<h2>Gestão de Participantes - Escolha uma opção</h2>  
  
<ul>  
    <li>  
        <a asp-action="IncluirParticipante"  
           asp-controller="Participantes">  
            Incluir Participante  
        </a>  
    </li>  
    <li>  
        <a asp-action="ListarParticipantes"  
           asp-controller="Participantes">  
            Listar Participantes por Evento  
        </a>  
    </li>  
</ul>
```

29. Na pasta **Dao**, inclua o arquivo **ParticipantesDao**, como subclasse de **GenericDao**:

```
using Lab.NetCore.Models;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace Lab.NetCore.Dao  
{  
    public class ParticipantesDao: GenericDao<Participante>  
    {  
        private EventosContext Context { get; set; }  
  
        public ParticipantesDao(EventosContext context)  
        : base(context)  
        {  
            this.Context = context;  
        }  
  
        //método para listar os participantes por evento  
        public IEnumerable<Participante> ListarPorEvento(  
            int idEvento)  
        {  
            return Context.Participantes  
                .Where(p => p.EventoInfoId == idEvento)  
                .ToList();  
        }  
    }  
}
```

30. Implemente o action **IncluirParticipante**, versões GET e POST, no controller **ParticipantesController**. Inclua também as referências ao contexto (injeção de dependência), **EventosDao** e **ParticipantesDao**:

```
using Lab.NetCore.Dao;
using Lab.NetCore.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Lab.NetCore.Controllers
{
    public class ParticipantesController : Controller
    {
        private EventosDao eventosDao { get; set; }
        private ParticipantesDao participantesDao { get; set; }

        public ParticipantesController(EventosContext context)
        {
            this.eventosDao = new EventosDao(context);
            this.participantesDao = new ParticipantesDao(context);
        }

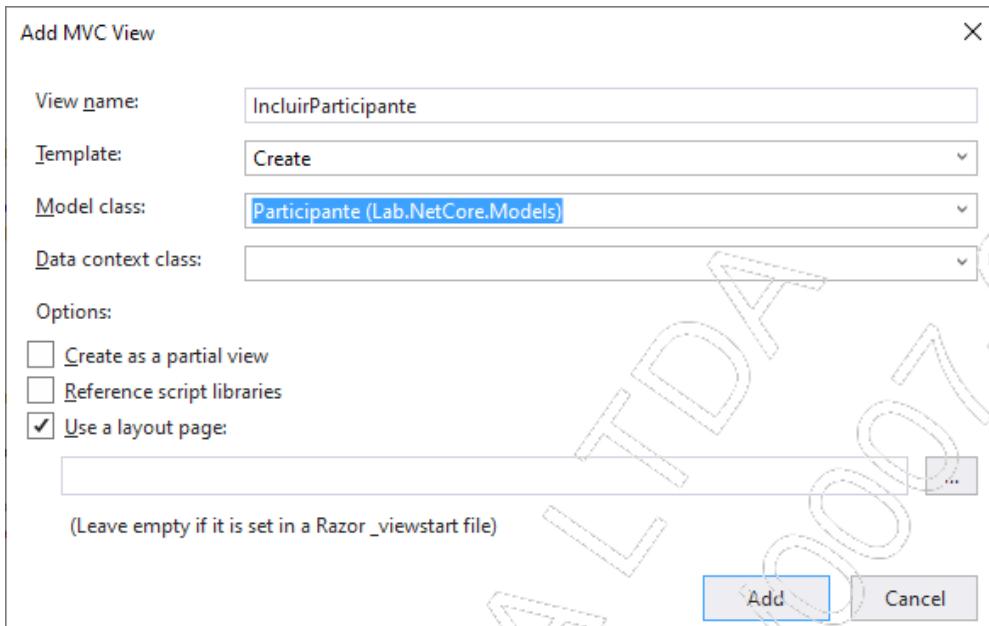
        public IActionResult Index()
        {
            return View();
        }

        [HttpGet]
        public IActionResult IncluirParticipante()
        {
            ViewBag.ListaDeEventos = new SelectList(eventosDao.Listar(), "Id", "Descricao");
            return View();
        }

        [HttpPost]
        public IActionResult IncluirParticipante(Participante participante)
        {
            if (participante.EventoInfoId == 0)
            {
                ModelState.AddModelError("IdEvento",
                    "Nenhum evento selecionado ");
            }

            if (!ModelState.IsValid)
            {
                return IncluirParticipante();
            }
            participantesDao.Executar(participante, TipoOperacaoBD.Added);
            return RedirectToAction("Index");
        }
    }
}
```

31. Gere a view para o action **IncluirParticipante**. Observe as alterações realizadas, especialmente na seleção de eventos:



```
@model Lab.NetCore.Models.Participante
 @{
    ViewData["Title"] = "Incluir Participante";
}

<h1>Incluir Participante</h1>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="IncluirParticipante">
            <div asp-validation-summary="ModelOnly"
                class="text-danger"></div>

            <div class="form-group">
                <label asp-for="EventoInfoId"
                    class="control-label"></label>
                <select asp-for="EventoInfoId"
                    asp-items="(SelectList)ViewBag.ListaDeEventos"
                    class="form-control">
                    <option>SELEÇÃO</option>
                </select>

                <span asp-validation-for="EventoInfoId"
                    class="text-danger"></span>
            </div>
        </form>
    </div>
</div>
```

```
</div>
<div class="form-group">
    <label asp-for="Nome"
        class="control-label"></label>
    <input asp-for="Nome"
        class="form-control" />
    <span asp-validation-for="Nome"
        class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="Email"
        class="control-label"></label>
    <input asp-for="Email"
        class="form-control" />
    <span asp-validation-for="Email"
        class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="Cpf"
        class="control-label"></label>
    <input asp-for="Cpf"
        class="form-control" />
    <span asp-validation-for="Cpf"
        class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="DataNascimento"
        class="control-label"></label>
    <input asp-for="DataNascimento"
        class="form-control" />
    <span asp-validation-for="DataNascimento"
        class="text-danger"></span>
</div>
<div class="form-group">
    <input type="submit" value="Incluir"
        class="btn btn-primary" />
</div>
</form>
</div>
<div>
    <a asp-action="Index">Voltar para o menu</a>
</div>
```

Verifique que a listagem de eventos passada para a view através de um componente **ViewBag** foi recuperada em um elemento **select**. Note também que, ao final do cadastro, direcionamos o usuário para o action **Index** em vez da listagem. Isso porque faremos a listagem usando **JavaScript**. Teste a inclusão de participantes.

C - Listando os participantes por evento via JavaScript - Ajax

Nesta parte do projeto, implementaremos o recurso de Partial View. O propósito é apresentar a lista de participantes por evento, sempre que o usuário selecionar um evento na lista. O conteúdo será mostrado de forma assíncrona, através de Ajax.

Teremos a view principal e a view parcial. A view principal será chama quando o usuário selecionar a lista de participantes; a partir desse ponto, basta escolher um evento para ver a lista de participantes daquele evento.

32. No controller **ParticipantesController**, vamos acrescentar o action **ListarParticipantes**. Observe o retorno de uma view parcial:

```
using Lab.NetCore.Dao;
using Lab.NetCore.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Lab.NetCore.Controllers
{
    public class ParticipantesController : Controller
    {
        private EventosDao eventosDao { get; set; }
        private ParticipantesDao participantesDao { get; set; }

        public ParticipantesController(EventosContext context)
        {
            this.eventosDao = new EventosDao(context);
            this.participantesDao = new ParticipantesDao(context);
        }

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

        [HttpGet]
        public IActionResult IncluirParticipante()
        {
            ViewBag.ListaDeEventos = new
                SelectList(eventosDao.Listar(), "Id", "Descricao");

            return View();
        }
}
```

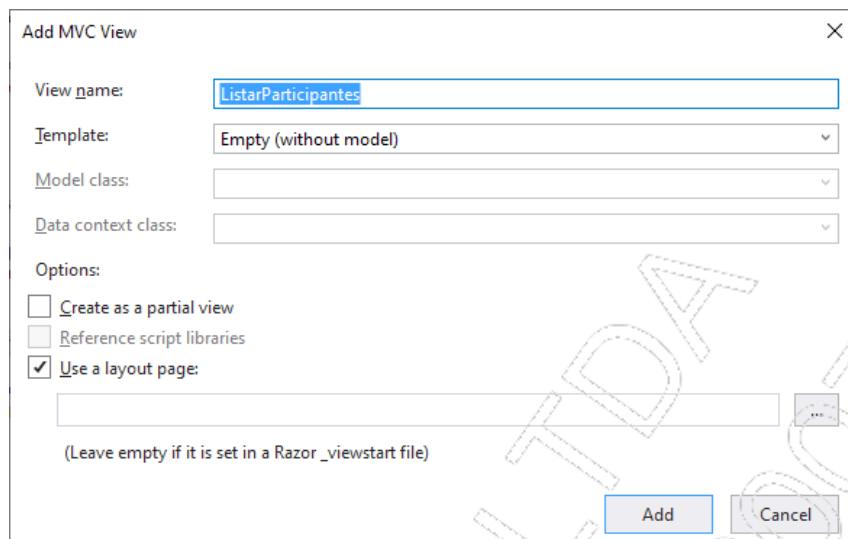
```
[HttpPost]
public IActionResult IncluirParticipante(Participante participante)
{
    if (participante.EventoInfoId == 0)
    {
        ModelState.AddModelError("IdEvento",
            "Nenhum evento selecionado ");
    }

    if (!ModelState.IsValid)
    {
        return IncluirParticipante();
    }
    participantesDao.Executar(participante, TipoOperacaoBD.Added);
    return RedirectToAction("Index");
}

public IActionResult ListarParticipantes(int idEvento)
{
    ViewBag.ListaDeEventos = new
        SelectList(eventosDao.Listar(), "Id", "Descricao");

    if (idEvento == 0)
    {
        return View();
    }
    else
    {
        var lista = participantesDao.ListarPorEvento(idEvento);
        return PartialView("_ListarParticipantes", lista);
    }
}
```

33. Crie a view para o action **ListarParticipantes**. Observe que a view não terá nenhum model:



```
@{
    ViewData["Title"] = "Listar Participantes";
}

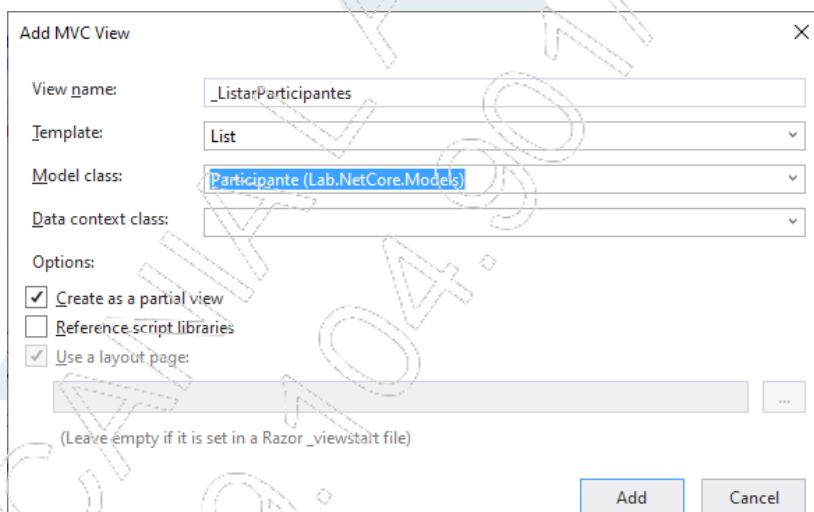
<h2>Listar Participantes por Evento - Ajax</h2>

<form asp-action="ListarParticipantesAjax"
      asp-controller="Eventos"
      method="get">
    <div class="row">
        <label class="control-label">ID Evento</label>
        <select id="idEvento" name="idEvento"
                class="form-control"
                asp-items="(SelectList)ViewBag.ListaDeEventos">
            <option value="0">SELEÇÃO</option>
        </select>
    </div>
</form>

<div class="row" id="resultado"></div>
```

```
@section Scripts{
    <script>
        $(document).ready(function () {
            $("#idEvento").change(function () {
                var selecao = $(this).val();
                if (selecao == "0") {
                    var erro = "<div class='alert alert-danger'>" +
                        "Nenhum evento selecionado</div>";
                    $("#resultado").html(erro);
                } else {
                    $("#resultado")
                        .load("/Participantes>ListarParticipantes?idEvento=" +
                            + selecao);
                }
            });
        });
    </script>
}
```

34. Crie a view parcial. Na pasta Views/Shared, inclua a view _ListarParticipantes:



Mantenha o conteúdo:

```
@model IEnumerable<Lab.NetCore.Models.Participante>



| @Html.DisplayNameFor(model => model.Nome)           |
|-----------------------------------------------------|
| @Html.DisplayNameFor(model => model.Email)          |
| @Html.DisplayNameFor(model => model.Cpf)            |
| @Html.DisplayNameFor(model => model.DataNascimento) |
| @Html.DisplayFor(modelItem => item.Nome)            |
| @Html.DisplayFor(modelItem => item.Email)           |
| @Html.DisplayFor(modelItem => item.Cpf)             |
| @Html.DisplayFor(modelItem => item.DataNascimento)  |


```

35. Observe que no layout já temos a instrução:

```
@RenderSection("Scripts", required: false)
```

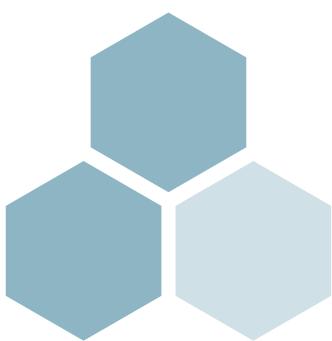
Neste ponto, será inserido o conteúdo da section com o nome **Scripts**. Podemos definir na view tantos sections quanto desejarmos e, no layout, especificar um local adequado para inseri-lo. Este processo é semelhante ao **RenderBody()**, só que nomeado, e com a opção de obrigatoriedade ou não;

36. Execute a aplicação, selecione a lista de participantes, selecione um evento e veja o resultado. É importante que haja pelo menos dois eventos e, para cada evento, uma boa quantidade de participantes.



ASP.NET Core Identity

- ◆ Configuração do ASP.NET Core Identity;
- ◆ Utilização do ASP.NET Core Identity na aplicação;
- ◆ Implementação da autorização.



9.1. Introdução

Existem dois processos básicos envolvidos na segurança dos dados disponibilizados por uma aplicação: **autenticação** e **autorização**. A **autenticação** é o processo de identificar a pessoa, software ou serviço que realizará alguma tarefa no programa. Já a **autorização** é o processo de permitir ou proibir o acesso às funcionalidades do sistema pelo usuário identificado.

É comum criarmos grupos de usuários e atribuirmos permissões a esses grupos, incluindo sempre o usuário identificado em um ou mais grupos definidos no sistema. Esses grupos teriam permissões diferentes. Por exemplo, um fornecedor poderia incluir um novo produto, um administrador poderia liberar o produto para venda, um cliente poderia visualizar o seu (e apenas o seu) carrinho de compras, um fornecedor não poderia visualizar informações de venda de produtos de outros fornecedores, e assim por diante.

 Repare que o processo de autorização é sempre uma operação de **permitir** ou **proibir** determinadas ações.

O **ASP.NET Core Identity** é um sistema integrado que permite incluir funcionalidade de logon ao aplicativo.

Os usuários podem criar uma conta contendo um nome de usuário e uma senha, que seguem determinados padrões. Assim como discutido no Capítulo 5, no qual estudamos o ASP.NET Identity do .NET Framework, o Identity Core também permite logon em diferentes providers, como Facebook, Google, Microsoft Account, Twitter ou outros.

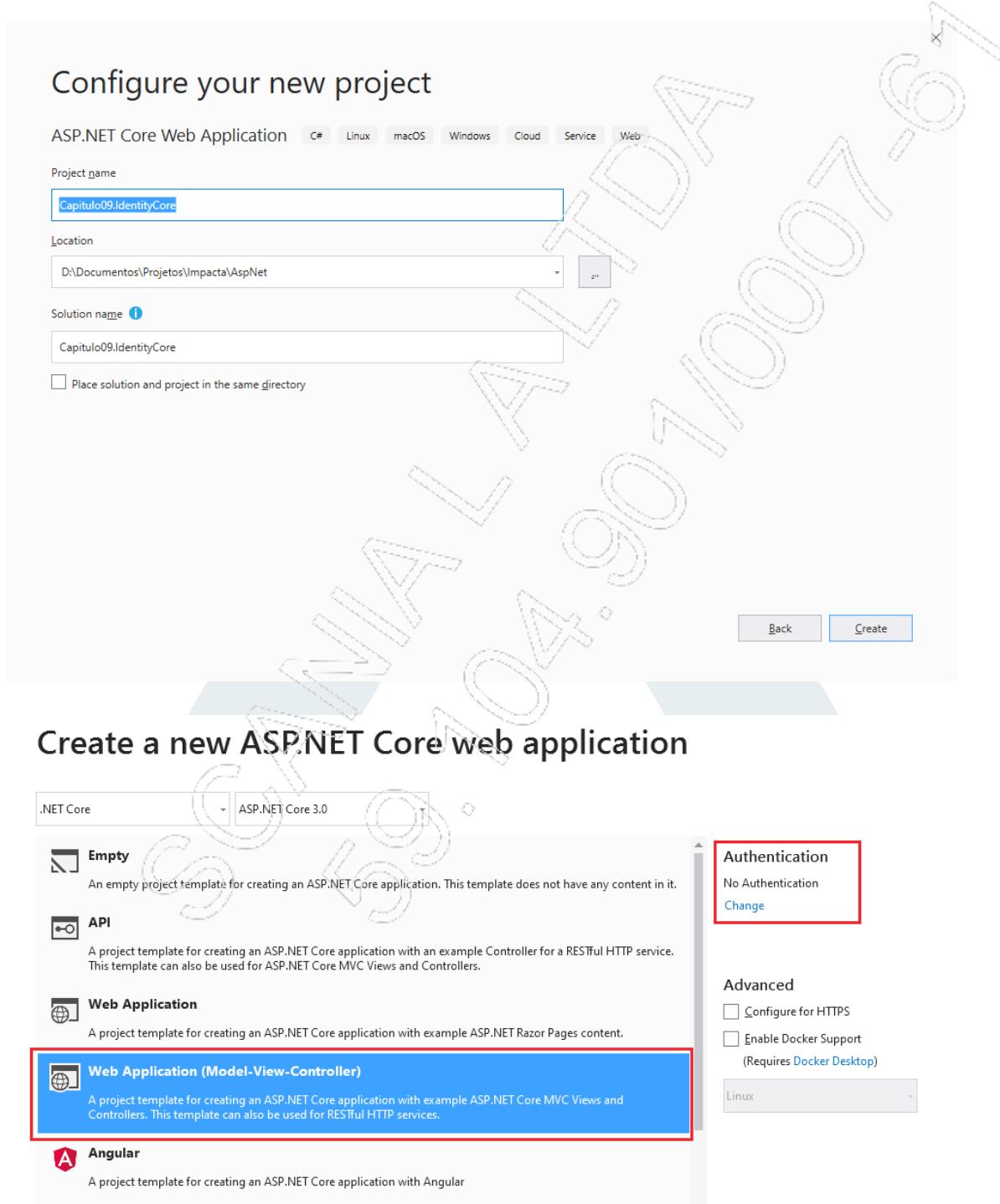
Neste capítulo, veremos como incluir recursos do Identity Core na nossa aplicação.

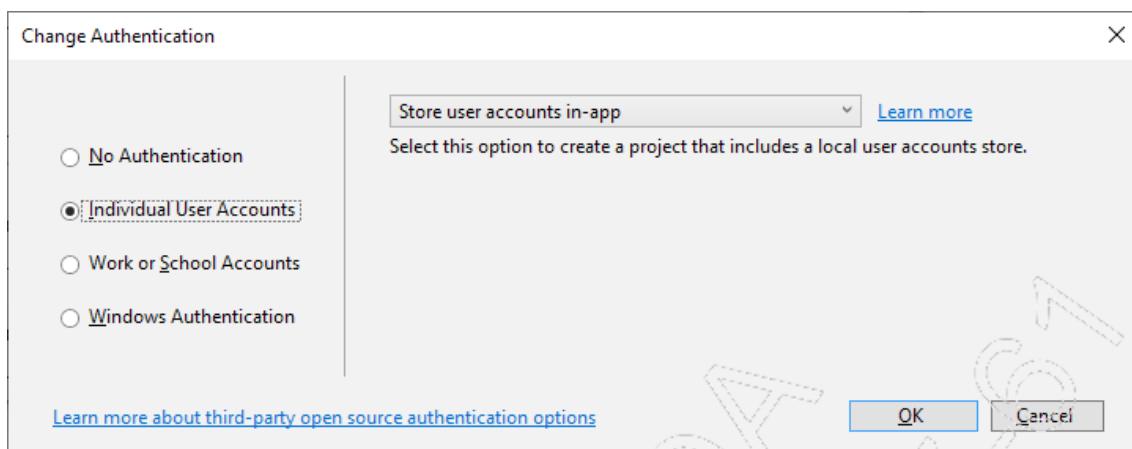
9.2. Configuração do ASP.NET Core Identity

A utilização do ASP.NET Core Identity requer a configuração de classes e serviços na aplicação, uma vez que grande parte dos seus elementos é obtida por meio da injeção de dependência. Nos próximos tópicos, veremos como adicionar o recurso de autenticação e autorização nas aplicações ASP.NET Core.

9.2.1. Definindo o projeto

A melhor maneira de adicionarmos os recursos de autenticação por meio do ASP.NET Core Identity é definindo um projeto adicionando o recurso de contas de usuário individuais (Individual User Accounts). Para demonstrar, vamos criar um novo projeto ASP.NET Core Web Application, com o template MVC com o recurso de contas de usuário individuais adicionado. O nome do projeto será **Capítulo09.IdentityCore**:





A habilitação dessa opção permitirá que o Visual Studio importe as dependências necessárias para trabalharmos com o ASP.NET Core Identity.

9.2.2. Definindo a classe de usuário

A definição da classe com os dados do usuário envolve a decisão de quais propriedades serão necessárias. A classe `IdentityUser` do namespace `Microsoft.AspNetCore.Identity` é disponibilizada pelo .NET Core e já oferece algumas propriedades comuns, como `UserName`, `PasswordHash`, `EmailAddress`, entre outras.

Se for necessário adicionar outras propriedades, como o CPF do usuário, torna-se necessário escrever uma subclasse de `IdentityUser` e aplicá-la ao projeto.

O código seguinte é um exemplo de definição de uma classe personalizada:

```
using Microsoft.AspNetCore.Identity;
using System.ComponentModel.DataAnnotations;

namespace Capitulo09.IdentityCore.Models
{
    public class UsuarioApp : IdentityUser
    {
        [MaxLength(11)]
        public string Cpf { get; set; }
    }
}
```

Nessa classe, disponibilizamos o CPF como uma alternativa para identificação do usuário.

9.2.3. Definindo o contexto do banco de dados

O ASP.NET Core Identity utiliza como padrão o Entity Framework Core como mecanismo de acesso a dados. Neste tópico, nós definiremos o contexto que suportará o ASP.NET Core Identity. Neste caso, devemos criar uma subclasse de **IdentityDbContext**. Essa classe é importante, pois encapsula a criação das propriedades do tipo **DbSet**, comuns no Entity Framework Core. A lógica empregada para manipular essas propriedades está implementada na classe **IdentityDbContext**.

O código a seguir apresenta a criação do contexto:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Capitulo09.IdentityCore.Models
{
    public class UsuariosContext : IdentityDbContext<UsuarioApp>
    {
        public UsuariosContext(DbContextOptions<UsuariosContext> options)
        :
            base(options)
        {
        }
    }
}
```

Observe que a classe **AuthenticationContext** fornece, como classe de usuário, a classe que criamos, **UsuarioApp**.

9.2.4. Configurando o Identity Core na classe Startup

Após a definição do contexto, devemos codificar as chamadas aos serviços e middlewares do framework. Como usaremos o SQL Server para armazenar os dados do usuário, necessitaremos do namespace **Microsoft.EntityFrameworkCore.SqlServer** (busque no NuGet) e da conexão definida em **appsettings.json**. O conteúdo de **appsettings.json** está mostrado a seguir:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Integrated Security=SSPI;Persist Security Info=False;Initial Catalog=DBIdentity;Data Source=.\\SQLEXPRESS"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Observe que definimos um banco de dados chamado **DBIdentity**. Sinta-se à vontade para alterá-lo.

No método **ConfigureServices** na classe **Startup**, será necessário chamar o método **AddDbContext** a partir do parâmetro **IServiceCollection**, utilizando a subclasse de **IdentityDbContext**. Sua instância ocorre da mesma forma como qualquer outra subclasse de **DbContext**, com os mesmos parâmetros:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<UsuariosContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<UsuarioApp>()
        .AddEntityFrameworkStores<UsuariosContext>();
    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

Será necessário, também, chamar o método **AddDefaultIdentity** parametrizado com o usuário definido (a classe **IdentityUser** ou uma subclasse).

Será necessário, ainda, encadear uma chamada ao método **AddEntityFrameworkStores** parametrizado com a classe referente ao contexto logo após **AddDefaultIdentity**. Ela será usada para conectar o objeto **IdentityDbContext** ao **IdentityServices** adicionado por meio da chamada a **AddDefaultIdentity**.

Nesse exemplo, podemos ver que os serviços do ASP.NET Core Identity são instanciados por meio da chamada ao método **AddDefaultIdentity**. Essa aplicação usa a classe **UsuarioApp** para conter os dados do usuário. Ela conecta, então, o service do Identity ao **IdentityDbContext** fornecido por **AuthenticationContext**.

Finalmente, no método **Configure** da classe **Startup**, será necessário executar o middleware para habilitar o processo de autenticação. Essa tarefa é realizada por meio da chamada ao método **UseAuthentication** do objeto **IApplicationBuilder**. É importante que esse método seja chamado antes de **UseEndpoints**. Isso garante que a autenticação sempre estará pronta para uso nos controllers da aplicação. Incluiremos, também, nesse método, o parâmetro **UsuariosContext** para garantir que o banco de dados seja criado quando iniciarmos a aplicação pela primeira vez e que esteja disponível na aplicação. O código a seguir é um exemplo de uso do middleware:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
                      UsuariosContext context)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseStaticFiles();

    app.UseRouting();

    context.Database.EnsureCreated();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}
```

A instrução `context.Database.EnsureCreated()` faz com que o banco de dados seja criado.

9.3. Utilização do ASP.NET Core Identity na aplicação

Os próximos subtópicos abordarão a definição do controller; dos modelos; das operações de login e de registro; e do componente com os itens de cadastro, login e logout.

9.3.1. Definindo o controller

Neste tópico, apresentaremos os procedimentos para criar usuários e efetuar login. Para sua codificação, criaremos um novo controller chamado **ContasController** com os actions necessários.

Para realizar operações de login, é necessário injetar os serviços **SignInManager<T>** e **UserManager<T>**, sendo **T** a classe **IdentityUser** ou sua subclasse:

```
using Capitulo09.IdentityCore.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace Capitulo09.IdentityCore.Controllers
{
    public class ContasController : Controller
    {
        private readonly UserManager<UsuarioApp> _userManager;
        private readonly SignInManager<UsuarioApp> _signInManager;

        public ContasController(UserManager<UsuarioApp> userManager,
                               SignInManager<UsuarioApp> signInManager)
        {
            this._userManager = userManager;
            this._signInManager = signInManager;
        }

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

9.3.2. Definindo os modelos

Vamos, agora, implementar as operações de inclusão de um novo usuário e de login. Para isso, devemos definir as classes necessárias que representarão os Models do projeto para o login e para o registro. Para o login, criaremos a classe **LogonViewModel** e, para o registro, **UsuarioViewModel**:

```
using System.ComponentModel.DataAnnotations;

namespace Capitulo09.IdentityCore.Models
{
    public class LogonViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace Capitulo09.IdentityCore.Models
{
    public class UsuarioViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirma Senha")]
        [Compare("Senha")]
        public string ConfirmaSenha { get; set; }
    }
}
```

9.3.3. Definindo as operações de login e de registro

No controller **ContasController**, incluiremos as operações para registro e login. Escreveremos os actions necessários para essas operações:

```
using Capitulo09.IdentityCore.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace Capitulo09.IdentityCore.Controllers
{
    public class ContasController : Controller
    {
        private readonly UserManager<UsuarioApp> _userManager;
        private readonly SignInManager<UsuarioApp> _signInManager;

        public ContasController(UserManager<UsuarioApp> userManager,
                               SignInManager<UsuarioApp> signInManager)
        {
            this._userManager = userManager;
            this._signInManager = signInManager;
        }

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

```
//Métodos auxiliares
private void AddErrors(IdentityResult result)
{
    foreach (var error in result.Errors)
    {
        ModelState.AddModelError(string.Empty, error.Description);
    }
}

private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction("Index", "Home");
    }
}

//Métodos de registro
[HttpGet]
[AllowAnonymous]
public IActionResult Registrar(string returnUrl = null)
{
    ViewBag.ReturnUrl = returnUrl;
    return View();
}

[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Registrar(UsuarioViewModel
model,
{
    string returnUrl = null)
    {
        ViewBag.ReturnUrl = returnUrl;

        if (ModelState.IsValid)
        {
            var user = new UsuarioApp
            {
                UserName = model.Email,
                Email = model.Email
            };

            var result = await _userManager.CreateAsync(user, model.
Senha);
            if (result.Succeeded)
            {
                await _signInManager.SignInAsync(user, isPersistent:
false);
                return RedirectToLocal(returnUrl);
            }
            AddErrors(result);
        }
        return View(model);
    }
}
```

```
//Login
[HttpGet]
[AllowAnonymous]
public IActionResult Login(string returnUrl = null)
{
    ViewBag.ReturnUrl = returnUrl;
    return View();
}

[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Login(LogonViewModel model,
    string returnUrl)
{
    ViewBag.ReturnUrl = returnUrl;

    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(
            model.Email, model.Senha, false, lockoutOnFailure:
false);

        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError(
                string.Empty, "Usuário ou senha inválidos.");
            return View(model);
        }
    }
    return View(model);
}

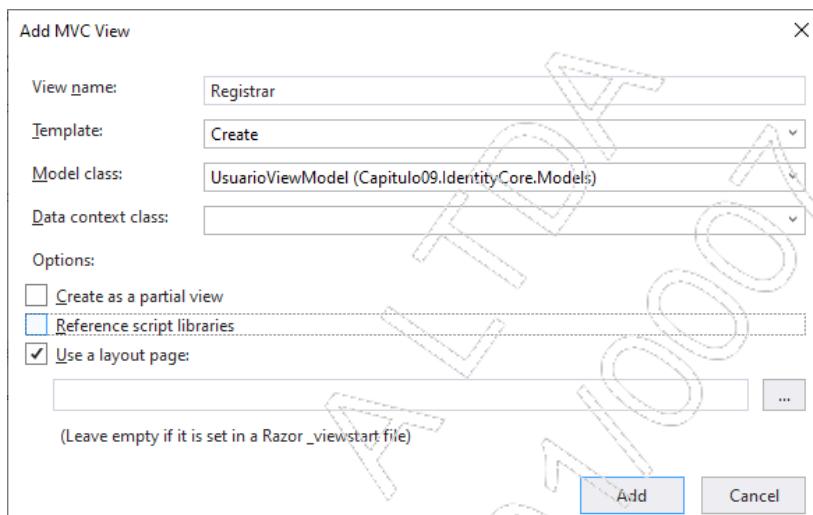
//Logout
[HttpGet]
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}
}
```

No método `PasswordSignInAsync`, o parâmetro `lockoutOnFailure` determina se o usuário será bloqueado caso o login esteja incorreto.

Uma vez que a autenticação seja realizada com sucesso, o usuário pode ser redirecionado para uma URL apropriada, que pode ser a página inicial (caso a operação de login seja a primeira operação no projeto) ou a página desejada (caso o usuário tente acessar um recurso protegido).

Na sequência, definiremos as views **Registrar.cshtml** e **Login.cshtml**:

- **Registrar.cshtml**



```
@model Capitulo09.IdentityCore.Models.UsuarioViewModel

@{
    ViewData["Title"] = "Registrar";
}



# Registrar



---

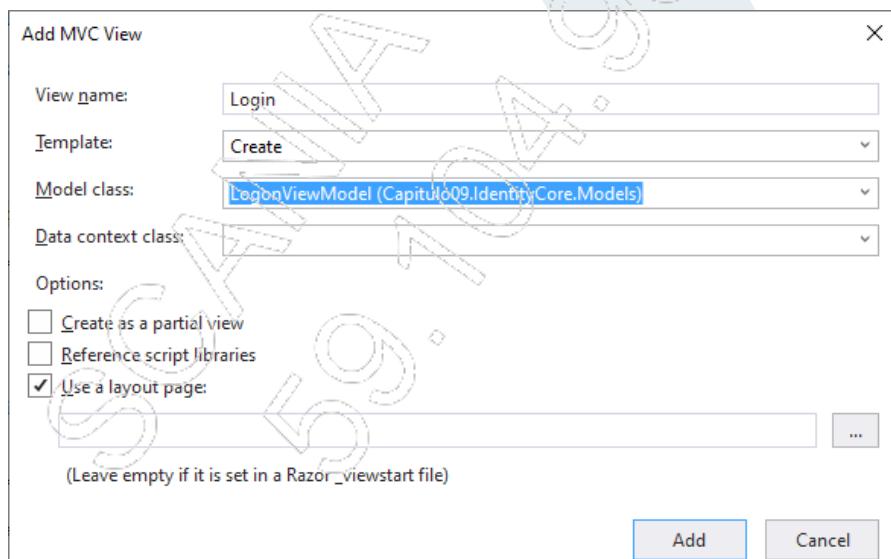


<form asp-action="Registrar">
            <div asp-validation-summary="ModelOnly"
                 class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
        </form>


```

```
span>
    </div>
    <div class="form-group">
        <label asp-for="Senha" class="control-label"></label>
        <input asp-for="Senha" class="form-control" />
        <span asp-validation-for="Senha" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmaSenha" class="control-label"></label>
        <input asp-for="ConfirmaSenha" class="form-control" />
        <span asp-validation-for="ConfirmaSenha"
            class="text-danger"></span>
    </div>
    <div class="form-group">
        <input type="submit" value="Incluir Usuário"
            class="btn btn-primary" />
    </div>
</form>
</div>
</div>
```

- **Login.cshtml**



```
@model Capitulo09.IdentityCore.Models.LogonViewModel

 @{
    ViewData["Title"] = "Login";
}

<h1>Login</h1>

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Login">
            <div asp-validation-summary="ModelOnly"
                class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Senha" class="control-label"></label>
                <input asp-for="Senha" class="form-control" />
                <span asp-validation-for="Senha"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Login"
                    class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

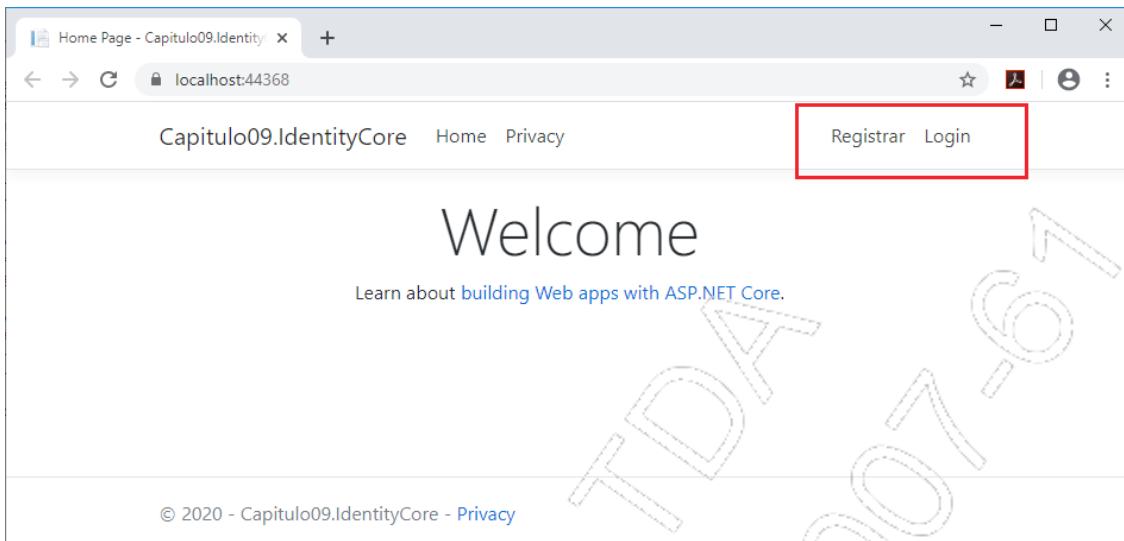
9.3.4. Definindo o componente com os itens de cadastro, login e logout

Quando criamos o projeto, foi adicionada uma Partial View chamada `_LoginPartial.cshtml` na pasta `Views/Shared`. Nossa tarefa é redefinir esse componente de forma a corresponder aos itens que definimos no projeto até o momento. Deixe esse arquivo com as configurações a seguir (os itens alterados / implementados estão destacados):

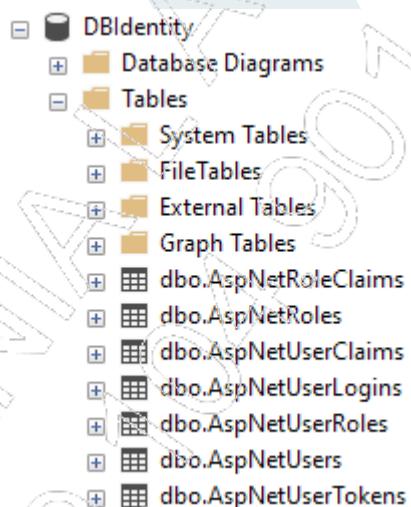
```
@using Microsoft.AspNetCore.Identity  
@inject SignInManager<UsuarioApp> SignInManager  
@inject UserManager<UsuarioApp> UserManager  
  


  
@if (SignInManager.IsSignedIn(User))  
{  
    <li class="nav-item">  
        <a class="nav-link text-dark"  
            asp-controller="Home"  
            asp-action="Index">@User.Identity.Name!</a>  
    </li>  
    <li class="nav-item">  
        <a class="nav-link text-dark"  
            asp-controller="Contas"  
            asp-action="Logout">Logout</a>  
    </li>  
}  
else  
{  
    <li class="nav-item">  
        <a class="nav-link text-dark"  
            asp-controller="Contas"  
            asp-action="Registrar">Registrar</a>  
    </li>  
    <li class="nav-item">  
        <a class="nav-link text-dark"  
            asp-controller="Contas"  
            asp-action="Login">Login</a>  
    </li>  
}  
</ul>
```

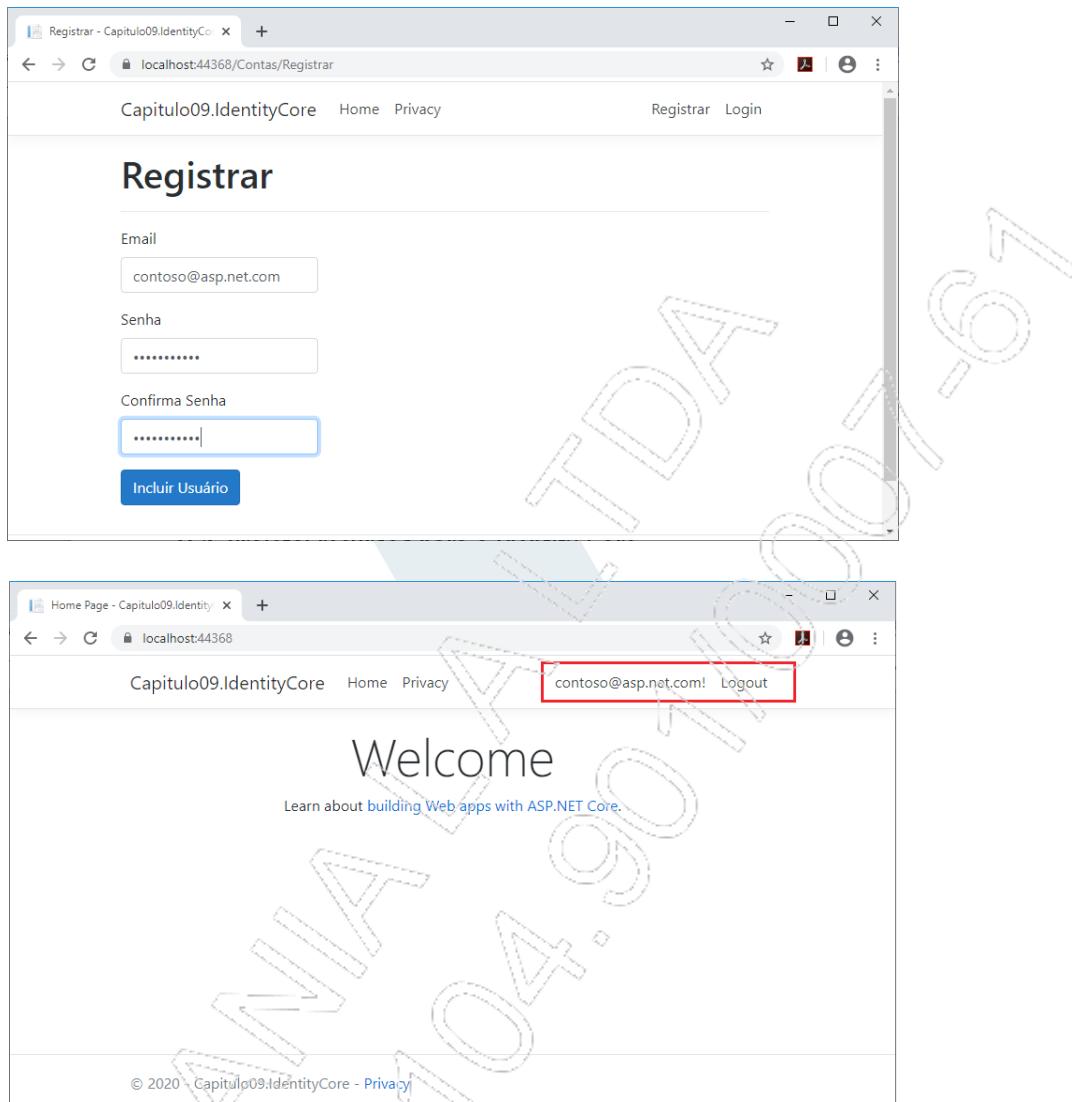
Com essas implementações, estamos prontos para executar a aplicação:



O banco de dados foi criado com as tabelas mostradas a seguir:



Vamos acessar o menu **Registrar** e adicionar um novo usuário:



9.4. Implementando autorização

A forma mais comum de autorização consiste no bloqueio de usuários não autenticados tentando executar determinados actions. Nesse caso, o usuário normalmente é redirecionado para o formulário de login para que o efetue.

A implementação da autorização é obtida por meio da inclusão do atributo **[Authorize]** na classe ou no action em específico, dependendo do nível de autorização desejado.

No caso de o atributo **[Authorize]** ser aplicado na classe, todos os actions deverão ser executados mediante autorização. Havendo um action em particular, por exemplo, que deve ser acessado de forma anônima (sem autenticação), este deve ter o atributo **[AllowAnonymous]** definido.

Para exemplificar, vamos implementar a view **Index** do controller **Contas**, de forma que sua execução exija autenticação:

```
using Capitulo09.IdentityCore.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace Capitulo09.IdentityCore.Controllers
{
    public class ContasController : Controller
    {
        private readonly UserManager<UsuarioApp> _userManager;
        private readonly SignInManager<UsuarioApp> _signInManager;

        public ContasController(UserManager<UsuarioApp> userManager,
            SignInManager<UsuarioApp> signInManager)
        {
            this._userManager = userManager;
            this._signInManager = signInManager;
        }

        [Authorize]
        public IActionResult Index()
        {
            return View();
        }

        //...
    }
}

@{
    ViewData["Title"] = "Index";
}

<h1>Esta página requer autorização</h1>
```

Vamos executar a aplicação e tentar acessar essa página, sem autenticação. O acesso a um item protegido, por default, redireciona o usuário para **Account/Login**. Devemos configurar o redirecionamento para a nossa página de login. Para tanto, devemos sobrescrever o comportamento padrão.

Na classe **Startup**, escreva o seguinte código, após o método **AddDefaultIdentity**:

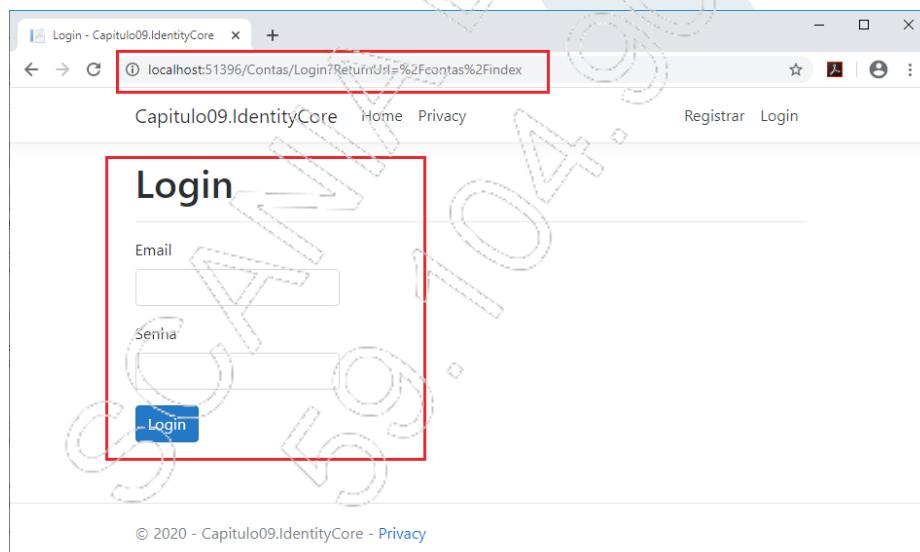
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<UsuariosContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<UsuarioApp>()
        .AddEntityFrameworkStores<UsuariosContext>();

    services.ConfigureApplicationCookie(options =>
    {
        options.LoginPath = "/Contas/Login";
        options.LogoutPath = "/Contas/Logout";
    });

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

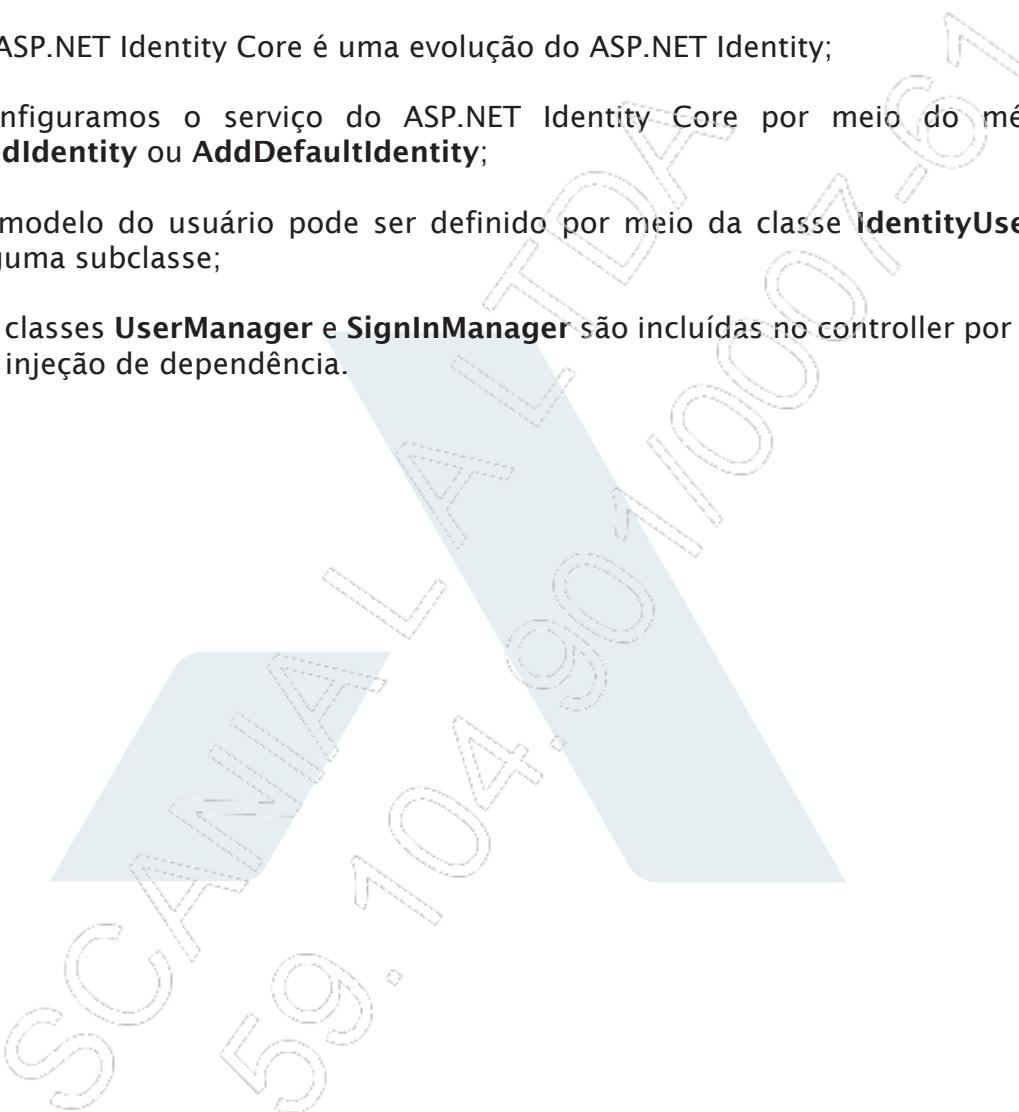
Agora podemos executar e tentar acessar o recurso protegido:



Após o login, o usuário é direcionado para a página desejada.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

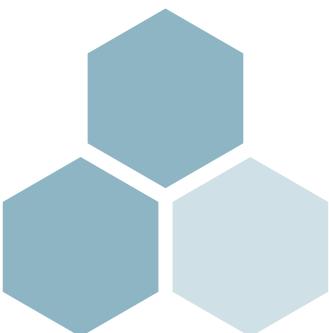
- O ASP.NET Identity Core é uma evolução do ASP.NET Identity;
 - Configuramos o serviço do ASP.NET Identity Core por meio do método **AddIdentity** ou **AddDefaultIdentity**;
 - O modelo do usuário pode ser definido por meio da classe **IdentityUser** ou alguma subclasse;
 - As classes **UserManager** e **SignInManager** são incluídas no controller por meio de injeção de dependência.
- 



9

ASP.NET Core Identity

Teste seus conhecimentos



1. Em qual namespace está a classe IdentityUser?

- a) Microsoft.AspNet.Identity
- b) Microsoft.NetCore.Identity
- c) System.AspNetCore.Identity
- d) Microsoft.AspNetCore.Identity
- e) System.NetCore.Identity

2. Para adicionar o serviço do Identity Core na aplicação, usamos qual método?

- a) AddServiceIdentity
- b) AddDefaultService
- c) AddDefaultIdentity
- d) AddUsers
- e) AddIdentityCore

3. Para habilitar o processo de autenticação, usamos qual método?

- a) EnableAuthentication
- b) ConfigureAuthentication
- c) UseAuthentication
- d) AllowAuthentication
- e) AddAuthentication

4. Para permitir que um usuário seja autenticado no sistema, devemos injetar no controller um objeto de qual classe?

- a) SignInManager
- b) LoginManager
- c) PrincipalManager
- d) RoleManager
- e) UserManager

5. Para bloquear usuários não autorizados, qual atributo devemos incluir no action a ser bloqueado?

- a) BlockUser
- b) AllowUsers
- c) AllowAnonymous
- d) Authorize
- e) NotAllowUsers



9

ASP.NET Core Identity



Mãos à obra!



Objetivos:

Vamos dar continuidade ao laboratório desenvolvido no Capítulo 8. Adicionaremos os recursos para autenticação e autorização de usuários por meio do ASP.NET Core Identity.

Laboratório 1

A - Adicionando recursos para autenticação e autorização de usuários

1. Na pasta **Labs**, crie um novo solution (**blank solution**) chamado **Capitulo09.Labs**;
2. Copie a pasta **Lab.NetCore** do solution **Capitulo08.Labs** para a pasta do novo solution, **Capitulo09.Labs**;
3. Adicione o projeto copiado ao solution recém-criado;
4. Execute a aplicação para ver se está tudo funcionando corretamente;

Nosso projeto foi criado considerando o recurso de contas de usuário individuais, mas apresentaremos, também, as opções de inclusão.

5. Vamos criar a classe **Usuario** na pasta **Models**, responsável por representar o usuário (observe a superclasse):

```
using Microsoft.AspNetCore.Identity;  
  
namespace Lab.NetCore.Models  
{  
    public class Usuario : IdentityUser  
    {  
    }  
}
```

É sempre uma boa prática criar uma subclasse de **IdentityUser**, pois esse recurso permite adicionar propriedades relevantes para a aplicação.

6. Na pasta **Dao**, defina a classe que representa o acesso a dados, o **DbContext**. Verifique a sua superclasse (o parâmetro **options** será passado via injeção de dependência):

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Lab.AspNetCore.Dao
{
    public class UsuariosDbContext : IdentityDbContext<Usuario>
    {
        public UsuariosDbContext(DbContextOptions<UsuariosDbContext>
            options) : base(options)
        { }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }
    }
}
```

7. Abra o arquivo **Startup.cs**. Inclua a configuração no método **ConfigureServices**, alterando a configuração original, considerando que o recurso de contas de usuário individuais tenha sido selecionado no início do projeto. Sobrescrevemos, também, o comportamento padrão do ASP.NET Identity Core, que direciona usuários não autenticados para **Account/Login**:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<EventosContext>(options => options.UseSqlServer(
        Configuration.GetConnectionString("EventosConnection")));

    //services.AddDbContext<ApplicationDbContext>(options =>
    //    options.UseSqlServer(
    //        Configuration.GetConnectionString("DefaultConnection")));

    services.AddDbContext<UsuariosDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<Usuario>(options =>
        options
            .SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<UsuariosDbContext>();

    //redefinição dos padrões do Identity Core
    services.ConfigureApplicationCookie(options =>
    {
        options.LoginPath = "/Usuarios/Login";
        options.LogoutPath = "/Usuarios/Logout";
        options.AccessDeniedPath = "/Usuarios/AccessDenied";
    });

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

8. Vamos, agora, trabalhar nas páginas de registro e validação de usuários. Na pasta **Models**, inclua as classes:

```
using System.ComponentModel.DataAnnotations;

namespace Lab.AspNetCore.Models
{
    public class UsuarioViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirma Senha")]
        [Compare("Senha")]
        public string ConfirmaSenha { get; set; }
    }
}

using System.ComponentModel.DataAnnotations;

namespace Lab.AspNetCore.Models
{
    public class LogonViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }
    }
}
```

9. Retornando ao arquivo **Startup.cs**, adicionaremos um recurso para que, quando a aplicação for iniciada, uma lista de perfis (Roles) seja adicionada ao banco de dados do ASP.NET Identity Core:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.EntityFrameworkCore;
using Lab.NetCore.Data;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Lab.NetCore.Dao;
using Lab.NetCore.Models;

namespace Lab.NetCore
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<EventosContext>(options =>
                options.UseSqlServer(
                    Configuration.GetConnectionString("EventosConnection")));

            //services.AddDbContext<ApplicationDbContext>(options =>
            //    options.UseSqlServer(
            //        Configuration.GetConnectionString("DefaultConnection")));

            services.AddDbContext<UsuariosDbContext>(options =>
                options.UseSqlServer(
                    Configuration.GetConnectionString("DefaultConnection")));

            //services.AddDefaultIdentity<Usuario>(options => options
            //    .SignIn.RequireConfirmedAccount = true)
            //    .AddEntityFrameworkStores<UsuariosDbContext>();

            services.AddIdentity<Usuario, IdentityRole>()
                .AddEntityFrameworkStores<UsuariosDbContext>()
                .AddDefaultTokenProviders();
        }
    }
}
```

```
//redefinição dos padrões do Identity Core
services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Usuarios/Login";
    options.LogoutPath = "/Usuarios/Logout";
    options.AccessDeniedPath = "/Usuarios/AccessDenied";
});

services.AddControllersWithViews();
services.AddRazorPages();
}

public void Configure(IApplicationBuilder app,
                      IWebHostEnvironment env,
                      IServiceProvider serviceProvider,
                      UsuariosDbContext context)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    context.Database.EnsureCreated();
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}

CreateRoles(serviceProvider).Wait();
}
```

```
//definir um método para incluir novos perfis (roles)
private async Task CreateRoles(IServiceProvider serviceProvider)
{
    var roleManager = serviceProvider
        .GetRequiredService<RoleManager<IdentityRole>>();

    var userManager = serviceProvider
        .GetRequiredService<UserManager<Usuario>>();

    string[] roleNames = { "ADMIN", "USER", "GUEST" };

    IdentityResult result;
    foreach (var role in roleNames)
    {
        var roleExist = await roleManager.RoleExistsAsync(role);
        if (!roleExist)
        {
            result = await
                roleManager.CreateAsync(new IdentityRole(role));
        }
    }
}
```

10. Escreva o controller **UsuariosController**. Observe a injeção de dependência no construtor. Além disso, no action **Registrar**, incluímos o usuário em um perfil:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Lab.NetCore.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Lab.NetCore.Controllers
{
    public class UsuariosController : Controller
    {
        private readonly UserManager<Usuario> _userManager;
        private readonly SignInManager<Usuario> _signInManager;
        private readonly RoleManager<IdentityRole> _roleManager;

        public UsuariosController(UserManager<Usuario> userManager,
            SignInManager<Usuario> signInManager,
            RoleManager<IdentityRole> roleManager)
        {
            this._userManager = userManager;
            this._signInManager = signInManager;
            this._roleManager = roleManager;
        }
    }
}
```

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

//métodos auxiliares
private void AddErrors(IdentityResult result)
{
    foreach (var error in result.Errors)
    {
        ModelState.AddModelError(string.Empty,
            error.Description);
    }
}

private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction("Index", "Home");
    }
}

//métodos de registro
[HttpGet]
public IActionResult Registrar(string returnUrl = null)
{
    ViewBag.ReturnUrl = returnUrl;

    var roles = _roleManager.Roles.ToList();
    var listaRoles = roles.Select(p => p.Name).ToList();

    ViewBag.Roles = new SelectList(listaRoles);
    return View();
}

[HttpPost]
public async Task<IActionResult> Registrar(
    UsuarioViewModel model,
    string perfil,
    string returnUrl = null)
{
    ViewBag.ReturnUrl = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new Usuario
        {
            UserName = model.Email,
            Email = model.Email
        };

        var result = await _userManager
            .CreateAsync(user, model.Senha);
    }
}
```

```
        if (result.Succeeded)
    {
        var appRole = await _roleManager
            .FindByNameAsync(perfil);
        if (appRole != null)
        {
            await _userManager.AddToRoleAsync(user, perfil);
        }

        await _signInManager.SignInAsync(user,
            isPersistent: false);
        return RedirectToLocal(returnUrl);
    }
    AddErrors(result);
}
return View(model);
}

public IActionResult Login(string returnUrl = null)
{
    ViewBag.ReturnUrl = returnUrl;
    return View();
}

[HttpPost]
public async Task<IActionResult> Login(
    LogonViewModel model, string returnUrl)
{
    ViewBag.ReturnUrl = returnUrl;
    if (ModelState.IsValid)
    {
        var result = await _signInManager
            .PasswordSignInAsync(
                model.Email, model.Senha, false,
                lockoutOnFailure: false);
        if (result.Succeeded)
        {
            return RedirectToLocal(ReturnUrl);
        }
        else
        {
            ModelState.AddModelError(
                string.Empty, "Usuário ou senha inválidos.");
            return View(model);
        }
    }
    return View(model);
}

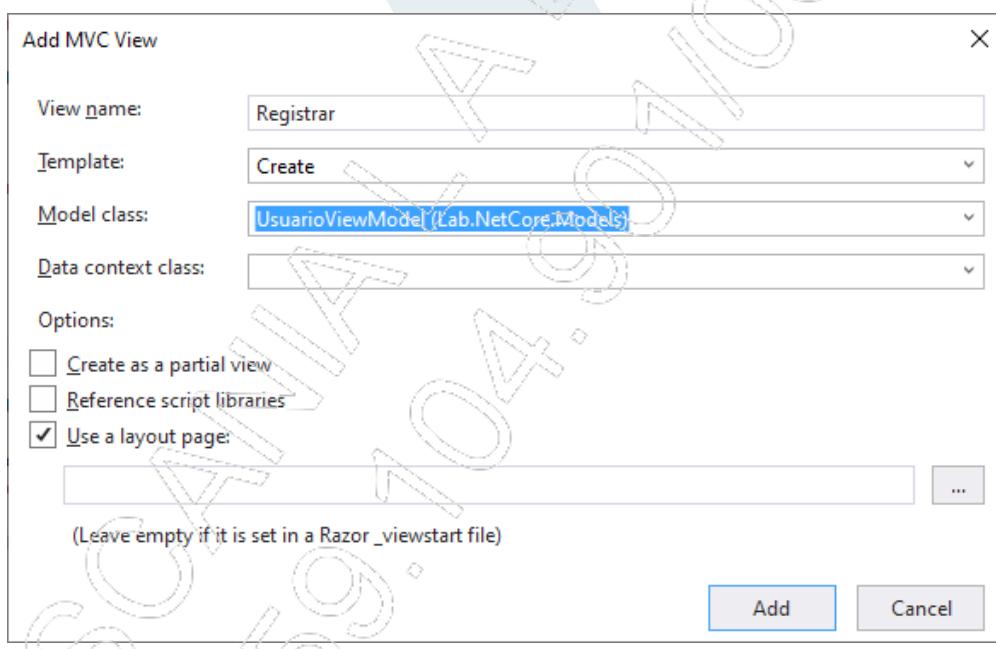
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}
```

```
public IActionResult AccessDenied()
{
    ViewData["MensagemErro"] =
        "Você não tem permissão para acessar este recurso!!";
    return View("_Erro");
}
```

11. Um arquivo que permite apresentar todos os recursos a serem importados pelas classes é o `_ViewImports.cshtml`. Verifique seu conteúdo, se possui os itens assinalados. Se não possuir, inclua-os:

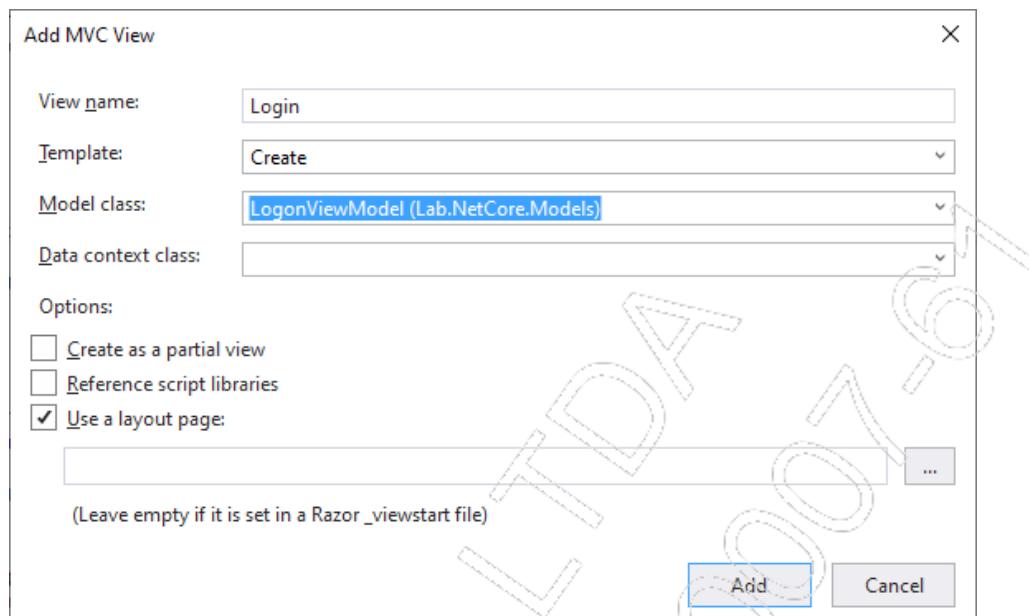
```
@using Microsoft.AspNetCore.Identity
@using Lab.NetCore
@using Lab.NetCore.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

12. Crie a view **Registrar**:



```
@model Lab.NetCore.Models.UsuarioViewModel  
{@  
    ViewData["Title"] = "Registrar";  
}  
  
<h1>Registrar Usuários</h1>  
  
<hr />  
<div class="row">  
    <div class="col-md-4">  
        <form asp-action="Registrar">  
            <div asp-validation-summary="ModelOnly"  
                class="text-danger"></div>  
            <div class="form-group">  
                <label asp-for="Email" class="control-label"></label>  
                <input asp-for="Email" class="form-control" />  
                <span asp-validation-for="Email" class="text-danger"></span>  
            </div>  
            <div class="form-group">  
                <label asp-for="Senha" class="control-label"></label>  
                <input asp-for="Senha" class="form-control" />  
                <span asp-validation-for="Senha" class="text-danger"></span>  
            </div>  
            <div class="form-group">  
                <label asp-for="ConfirmaSenha" class="control-label"></label>  
                <input asp-for="ConfirmaSenha" class="form-control" />  
                <span asp-validation-for="ConfirmaSenha"  
                    class="text-danger"></span>  
            </div>  
            <div class="form-group">  
                <label class="control-label">Perfil</label>  
                <select name="perfil" class="form-control"  
                    asp-items="(SelectList)ViewBag.Roles"></select>  
            </div>  
            <div class="form-group">  
                <input type="submit" value="Registrar"  
                    class="btn btn-primary" />  
            </div>  
        </form>  
    </div>  
</div>
```

13. Crie a view Login:



```
@model Lab.NetCore.Models.LogonViewModel
@inject SignInManager<Usuario> SignInManager
@{
    ViewData["Title"] = "Login";
}



# Login



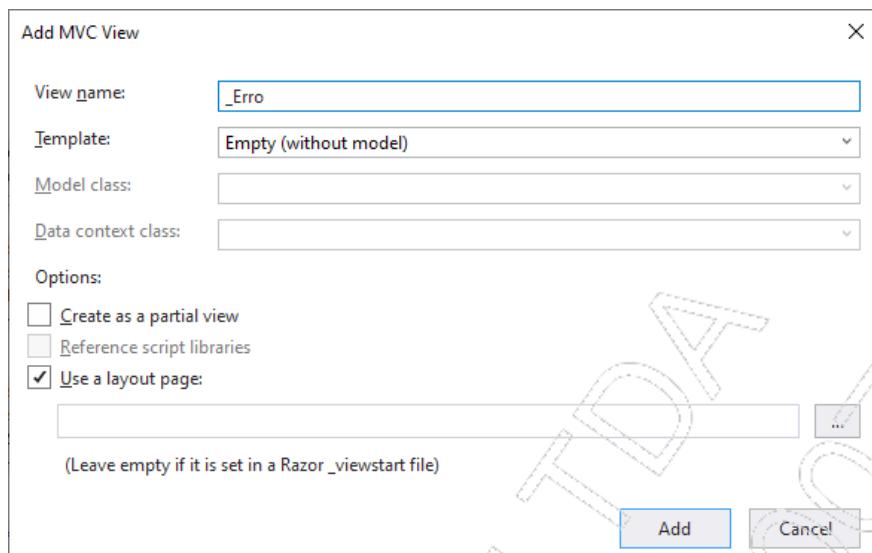
---



<div class="col-md-4">
        <form asp-route-returnurl="@ViewBag.ReturnUrl">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Senha" class="control-label"></label>
                <input asp-for="Senha" class="form-control" />
                <span asp-validation-for="Senha" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Login" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>


```

14. Na pasta **Views/Shared**, inclua a view **_Erro**:



```
@{
    ViewData["Title"] = "Erro";
}

<h2 class="text-danger">Ocorreu o seguinte erro:</h2>
<div class="alert alert-danger">
    @ViewData["MensagemErro"]
</div>
```

15. Crie a view parcial na pasta **views/shared** chamada **_LoginPartial.cshtml** (se existir, altere seu conteúdo):

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<Usuario> SignInManager
@inject UserManager<Usuario> UserManager

<ul class="navbar-nav">
    @if (SignInManager.IsSignedIn(User))
    {
        <li class="nav-item">
            <a class="nav-link text-dark"
               asp-controller="Home"
               asp-action="Index">@User.Identity.Name!</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark"
               asp-controller="Usuarios"
               asp-action="Logout">Logout</a>
        </li>
    }
    else
```

```
{  
    <li class="nav-item">  
        <a class="nav-link text-dark"  
            asp-controller="Usuarios"  
            asp-action="Registrar">Registrar</a>  
    </li>  
    <li class="nav-item">  
        <a class="nav-link text-dark"  
            asp-controller="Usuarios"  
            asp-action="Login">Login</a>  
    </li>  
}  
</ul>
```

16. Altere o arquivo `_Layout.cshtml` de forma a incluir essa view parcial (na execução, observe no menu à direita):

```
<div class="navbar-collapse collapse d-sm-inline-flex  
      flex-sm-row-reverse">  
    <partial name="_LoginPartial" />  
    <ul class="navbar-nav flex-grow-1">  
        <li class="nav-item">  
            <a class="nav-link text-dark" asp-area=""  
                asp-controller="Home"  
                asp-action="Index">Home</a>  
        </li>  
        <li class="nav-item">  
            <a class="nav-link text-dark" asp-area=""  
                asp-controller="Eventos"  
                asp-action="Index">Eventos</a>  
        </li>  
        <li class="nav-item">  
            <a class="nav-link text-dark" asp-area=""  
                asp-controller="Participantes"  
                asp-action="Index">Participantes</a>  
        </li>  
    </ul>  
</div>
```

17. Uma vez definidos os actions para registro e login, vamos, agora, controlar o que deve ser executado por um usuário autenticado. Por exemplo: Considere que, para realizar um cadastro de eventos, o usuário deva ser, por exemplo, um administrador devidamente logado. No action `IncluirEvento`, devemos incluir o atributo:

```
[HttpGet]  
[Authorize(Roles = "ADMIN")]  
public IActionResult IncluirEvento()  
{  
    return View();  
}
```

18. Confira a string de conexão no arquivo **appsettings.json**:

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Integrated Security=SSPI;Persist Security  
Info=False;Initial Catalog=DB_IDENTITY;Data Source=.\SQLEXPRESS",  
    "EventosConnection": "Integrated Security=SSPI;Persist Security  
Info=False;Initial Catalog=DB_EVENTOS;Data Source=.\SQLEXPRESS"  
  },  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    }  
  },  
  "AllowedHosts": "*"  
}
```

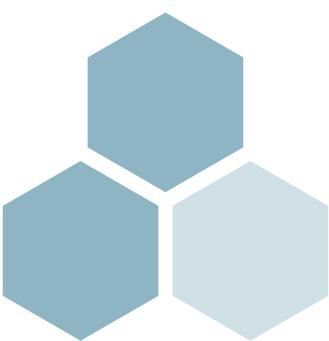
19. Ao executar a aplicação, selecione o item do menu para criar novos usuários. Crie usuários com diferentes perfis. Tente acessar o cadastro de eventos com um usuário pertencente a um perfil que não seja **admin**. O que é possível observar?



10

Web API Core

- ◆ Representação de um Web service REST;
- ◆ Media Types;
- ◆ Desenvolvimento de serviços baseados em Web API Core;
- ◆ Criação do projeto Web API.



10.1. Introdução

Web API Core é a tecnologia da Microsoft para criar e consumir serviços baseados em REST. Houve diversas melhorias em relação à Web API baseada no .NET Framework. Os conceitos de serviços REST que são sustentados pelo Web API Core serão descritos na sequência.

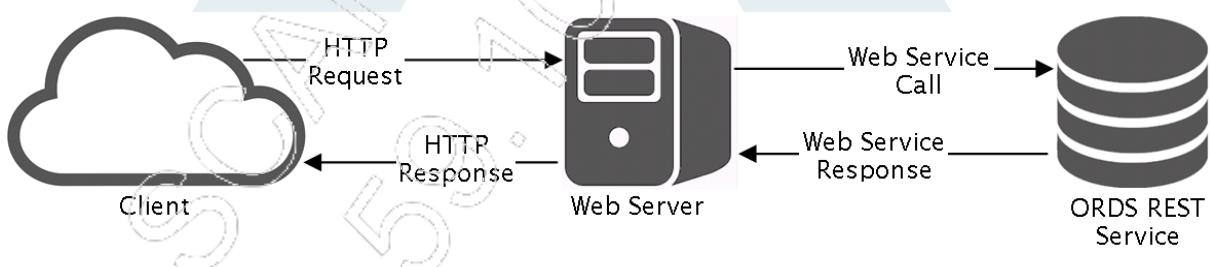
REST significa **Representational State Transfer**. Um recurso no sentido RESTful é qualquer coisa que tenha uma URI representando seus recursos.

Então, Web Services RESTful exigem não apenas recursos para representar, mas também operações chamadas pelo cliente desses recursos. No centro da abordagem RESTful está a percepção de que HTTP é uma API e não simplesmente um protocolo de transporte. HTTP tem seus verbos bem conhecidos, oficialmente chamados de **métodos**.

Uma diretriz importante do estilo RESTful é respeitar os significados originais dos verbos HTTP.

A abordagem REST não implica que os recursos ou o processamento necessários para gerar representações adequadas deles sejam simples. A abordagem RESTful mantém a complexidade referente ao mecanismo de transferência fora do nível de transporte, conforme uma representação de recurso é transferida para o cliente como o corpo de uma mensagem de resposta HTTP.

10.2. Representação de um Web service REST



Considere que uma aplicação Web necessite se comunicar com outra aplicação. O Web service é o canal de comunicações entre as aplicações.

10.3. Media Types

O protocolo HTTP foi concebido originalmente para transferir hipertexto. No entanto, alguns recursos possuem outros formatos. É o caso das imagens e dos vídeos, por exemplo, que utilizam o HTTP para transportar diferentes tipos de formatos.

Para essa tarefa, o HTTP utiliza tipos baseados em **Multipurpose Internet Mail Extensions** (MIME), também chamados de **media types**.

Media types são informados como parte do cabeçalho da requisição. Ou seja, quando o cliente realiza uma requisição, ele pode enviar uma lista de media types, assim pode recebê-los como resposta.

Assim, tanto cliente como servidor se comunicam de modo a fornecer informações compatíveis durante uma transação.

10.4. Desenvolvimento de serviços baseados em Web API Core

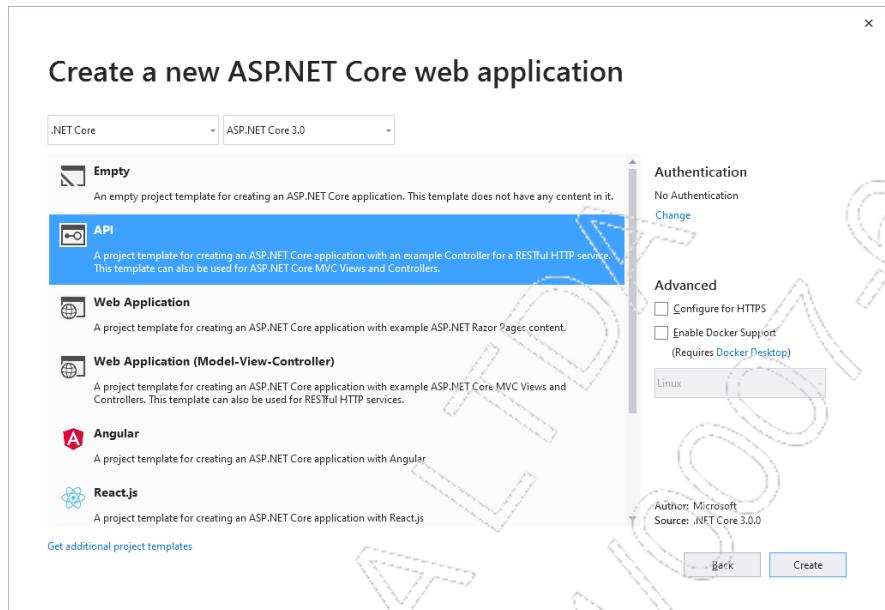
Em um projeto Web API no .NET Core, o processo de requisição e resposta ocorre de forma análoga ao que vimos quando estudamos Web API no .NET Framework. Existem similaridades na definição dos actions (que são as tarefas), mas a arquitetura é diferente:

- O action no controller pode retornar **IActionResult**;
- O objeto **HttpClient**, usado por uma aplicação cliente para consumir a Web API, pode ser obtido por injeção de dependência, desde que devidamente registrado na classe **Startup** e configurado como parâmetro do construtor do controller;
- A superclasse do controller é a superclasse usada em uma aplicação MVC Core: **ControllerBase**. Um atributo **[ApiController]** é usado para especificar que se trata de uma Web API.

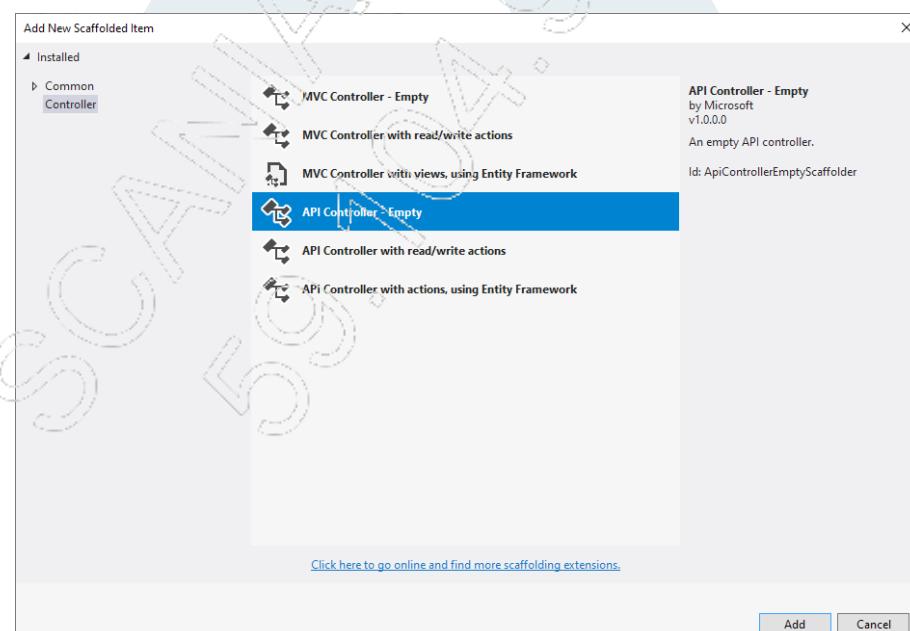
Nos tópicos seguintes, ilustraremos esses novos conceitos durante a elaboração de um projeto.

10.5. Criação do projeto Web API

Crie um novo projeto chamado **Capítulo10.WebAPI**. Escolha a opção API:



Iniciaremos pela definição do controller. Na pasta **Controllers**, adicione um novo controller chamado **HomeController**. No processo de criação, selecione a opção **API Controller - Empty**:



Uma das configurações que podemos fornecer ao controller é a rota, representada pelo atributo **Route**. Veja que a classe possui a configuração `[Route("api/[controller]")]`, indicando a forma como o acesso ao serviço é realizada. No nosso exemplo, o acesso seria realizado como `api/home`. Veja a classe gerada nesse processo:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
    }
}
```

10.5.1. Definição dos actions

O Web API Core também considera o padrão de nomenclatura dos actions para sua execução de acordo com o verbo HTTP. Por exemplo, os métodos **GetXXX** são executados para uma requisição HTTP GET; os métodos **PostXXX** são executados para uma requisição HTTP POST; e assim por diante. Esse padrão é o mesmo usado no Web API do .NET Framework.

Se desejarmos especificar o método HTTP em um método que não siga esses padrões, devemos usar um dos seguintes atributos: **HttpGet**, **HttpPut**, **HttpPost** ou **HttpDelete**, conforme o objetivo. Considere o action a seguir:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpGet]
        public string MostrarNome()
        {
            return "Asp.Net Core";
        }
    }
}
```

Ele será executado quando uma requisição HTTP GET for realizada por conta do seu atributo.

Um problema surge quando desejarmos que mais de um action seja executado usando o mesmo verbo HTTP:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpGet]
        public string MostrarNome()
        {
            return "Asp.Net Core";
        }

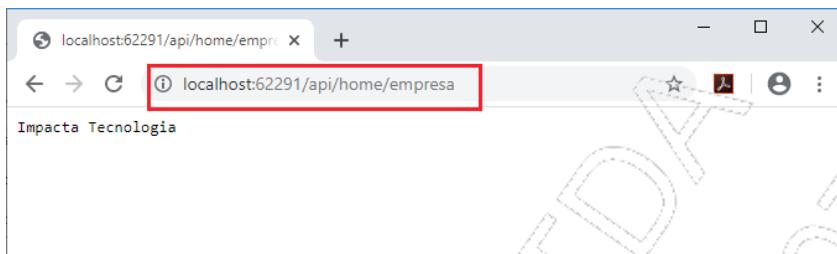
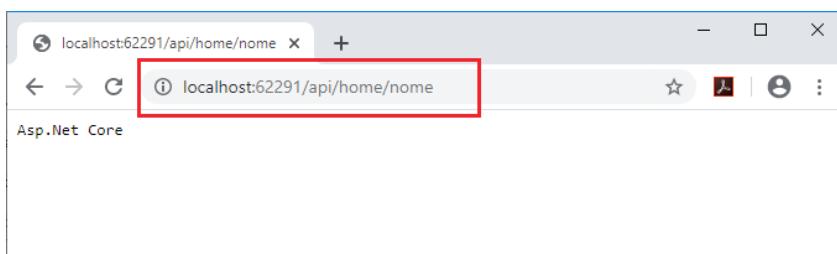
        [HttpGet]
        public string MostrarEmpresa()
        {
            return "Impacta Tecnologia";
        }
    }
}
```

Nesse caso, devemos especificar um nome a ser usado na sua chamada. O exemplo a seguir ilustra esse procedimento, e, na sequência, temos uma execução:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpGet("nome")]
        public string MostrarNome()
        {
            return "Asp.Net Core";
        }

        [HttpGet("empresa")]
        public string MostrarEmpresa()
        {
            return "Impacta Tecnologia";
        }
    }
}
```



Embora não seja comum, podemos especificar na rota o nome do action:

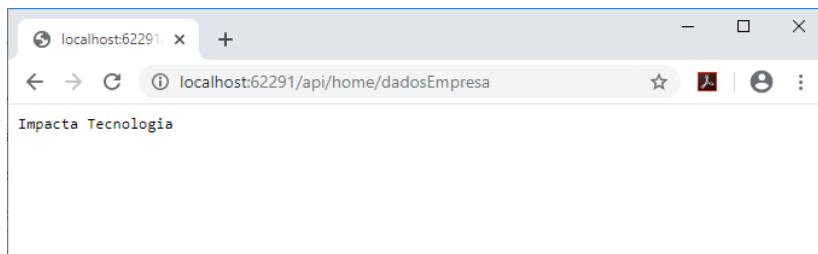
[Route("api/[controller]/[action]")]

A chamada ao serviço implica em fornecer o nome do action na rota.

É possível, ainda, especificarmos a rota no action:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [Route("dadosEmpresa")]
        public string MostrarEmpresa()
        {
            return "Impacta Tecnologia";
        }
    }
}
```



10.5.1.1. Actions com parâmetros

É comum um action receber parâmetros. Isso ocorre, por exemplo, quando desejamos remover um item pelo seu id, ou buscar um produto pelo código. Para permitir a entrada de parâmetros no método, podemos proceder de algumas formas. Vejamos alguns exemplos:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]/[id]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpGet]
        public string MostrarNome(int id)
        {
            return "Asp.Net Core - código = " + id;
        }
    }
}
```

Nesse exemplo, especificamos o id na rota configurada no controller. O problema dessa abordagem é que ela considera que todos os actions terão o parâmetro id. Uma abordagem alternativa é a seguinte:

```
using Microsoft.AspNetCore.Mvc;

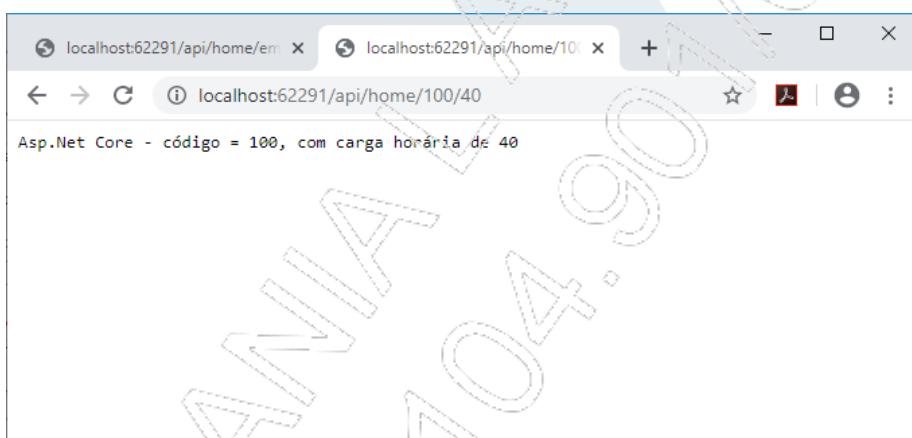
namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpGet("{id}")]
        public string MostrarNome(int id)
        {
            return "Asp.Net Core - código = " + id;
        }
    }
}
```

É possível, também, informarmos mais de um parâmetro no action:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpGet("{id}/{ch}")]
        public string MostrarNome(int id, int ch)
        {
            return "Asp.Net Core - código = " + id +
                   ", com carga horária de " + ch;
        }
    }
}
```

Um exemplo de execução produz o seguinte resultado:



10.5.1.2. Valores de retorno de actions

Nos exemplos anteriores, apresentamos actions retornando apenas String. Na verdade, é o tipo mais incomum de retorno, e os usamos por questões meramente didáticas.

Um action pode retornar listas, objetos ou, mesmo, **IActionResult**. Veja os exemplos de actions a seguir:

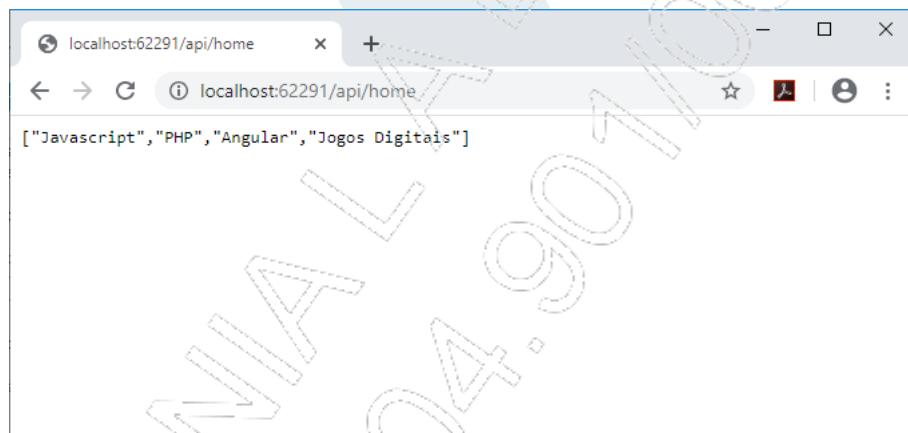
```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace Capitulo10.WebAPI.Controllers
{
```

```
[Route("api/[controller]")]
[ApiController]
public class HomeController : ControllerBase
{
    private static readonly IEnumerable<string> cursos = new
        List<string>()
    {
        "Javascript", "PHP", "Angular", "Jogos Digitais"
    };

    public IEnumerable<string> GetCursos()
    {
        return cursos;
    }
}
```

Veja, agora, o resultado da execução:



O action a seguir retorna um `ActionResult<string>`, indicando que é possível retornar uma string ou um recurso indicando erro:

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

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        private static readonly List<string> cursos = new List<string>()
        {
            "Javascript", "PHP", "Angular", "Jogos Digitais"
        };
    }
}
```

```
public IEnumerable<string> GetCursos()
{
    return cursos;
}

[HttpGet("{id}")]
public ActionResult<string> GetCurso(int id)
{
    if(id >= cursos.Count)
    {
        return NotFound();
    }
    return cursos[id];
}
```

10.5.1.3. Objetos como parâmetros

Na maioria das vezes, um action recebe um objeto como parâmetro para que seja, por exemplo, incluído em um banco de dados. Em situações como essa, especificamos que os dados do objeto são recebidos por meio do corpo da requisição, por meio da rota etc. Os atributos exibidos a seguir representam o que chamamos de **Binding Sources**:

- **FromBody**: Determina que o parâmetro do action é vinculado (bound) usando o corpo da requisição, normalmente por meio de um POST ou PUT;
- **FromQuery**: Determina que o parâmetro do action é vinculado usando uma query string;
- **FromRoute**: Determina que o parâmetro do action é vinculado usando informações da rota;
- **FromForm**: Determina que o parâmetro do action é vinculado usando o conteúdo de um form;
- **FromHeader**: Determina que o parâmetro do action é vinculado usando o cabeçalho da requisição.

Para exemplificar, vamos considerar uma classe chamada **Pessoa** com as propriedades **Nome** e **Idade**. O action receberá um objeto, cujas informações são originadas no corpo de uma requisição POST:

- **Classe Pessoa:**

```
namespace Capitulo10.WebAPI
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
    }
}
```

- **Controller:**

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpPost]
        public void PostPessoa([FromBody] Pessoa pessoa)
        {
            //processa a operação
        }
    }
}
```

10.5.2. Consumindo o Web API Core no cliente

Para consumir o Web API Core desenvolvido neste capítulo, vamos criar um novo projeto chamado **Capitulo10.ClienteWebAPI** do tipo ASP.NET Core Web Application, modelo MVC.

No lado cliente, o ASP.NET Core disponibiliza a classe **System.Net.Http.HttpClient**. Ela contém as funcionalidades básicas para processar requisições e respostas.

Além disso, a classe **HttpClient** mantém uma integração consistente com Web API Core por meio dos objetos **HttpRequestMessage** e **HttpResponseMessage**, responsáveis pela manipulação das mensagens HTTP. Ela possui diversos métodos assíncronos para o processamento dos serviços:

- **GetAsync:** Executa uma requisição GET para uma URI específica;
- **PostAsync:** Envia uma requisição POST para uma URI específica;
- **PutAsync:** Envia uma requisição PUT para uma URI específica;
- **DeleteAsync:** Envia uma requisição DELETE para uma URI específica.

No ASP.NET Core, para definir uma instância de **HttpClient**, podemos usar o serviço **IHttpClientFactory**. Para registrá-lo, devemos chamar o método de extensão **AddHttpClient** no método **ConfigureServices**, na classe **Startup**. O código a seguir mostra como registrar o serviço **IHttpClientFactory**:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddHttpClient();
}
```

Após registrado o serviço **IHttpClientFactory**, o processo de injeção de dependência pode ser aplicado para injetá-lo no controller.

Uma vez injetado, podemos usar o método **CreateClient** para obter uma instância de **HttpClient**. Para demonstrar, criaremos o controller **ClientesController** no projeto. Em seguida, adicionaremos o construtor que receberá o **IHttpClientFactory**:

```
using Microsoft.AspNetCore.Mvc;
using System.Net.Http;

namespace Capitulo10.ClienteWebAPI.Controllers
{
    public class ClientesController : Controller
    {
        HttpClient client;

        public ClientesController(IHttpClientFactory httpClient)
        {
            client = httpClient.CreateClient();
        }

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

Assim, podemos chamar um serviço Web API Core por meio dos métodos apresentados. Vamos escrever um action chamado **Buscar** e, nele, o código para consumir o Web service desenvolvido neste capítulo:

```
using Microsoft.AspNetCore.Mvc;
using System;
using System.Net.Http;
using System.Net.Http.Headers;

namespace Capitulo10.ClienteWebAPI.Controllers
{
    public class ClientesController : Controller
    {
        HttpClient client;

        public ClientesController(IHttpClientFactory httpClient)
        {
            client = httpClient.CreateClient();
        }

        public IActionResult Buscar()
        {
            client.BaseAddress = new Uri("http://localhost:62291/");
            client.DefaultRequestHeaders.Accept.Add(new
                MediaTypeWithQualityHeaderValue("application/json"));
            HttpResponseMessage response = cliente
                .GetAsync("api/home").Result;

            if (response.IsSuccessStatusCode)
            {
                //processa a resposta do Web service
                return View();
            }
            else
            {
                return Content("Ocorreu um erro");
            }
        }

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

Para finalizar o exemplo, vamos criar uma classe chamada **Pessoa**, com as mesmas propriedades da pessoa que definimos no projeto **Capítulo10.WebAPI**. Essa classe será usada para ilustrar a busca por pessoas na API e para enviar um objeto para ela. Vamos criar a classe **Pessoa** na pasta **Models**:

```
namespace Capítulo10.ClienteWebAPI.Models
{
    public class Pessoa
    {
        public string Nome { get; set; }
        public int Idade { get; set; }
    }
}
```

Agora, vamos completar o método **Buscar** de forma a trazer a lista de pessoas da API:

```
public async Task<IActionResult> Buscar()
{
    client.BaseAddress = new Uri("http://localhost:62291/");
    client.DefaultRequestHeaders.Accept.Add(new
        MediaTypeWithQualityHeaderValue("application/json"));
    HttpResponseMessage response = client.GetAsync("api/home").Result;

    if (response.IsSuccessStatusCode)
    {
        var pessoas = await response.Content.ReadAsAsync<Pessoa[]>();
        return View(pessoas.ToList());
    }
    else
    {
        return Content("Ocorreu um erro");
    }
}
```

Fizemos uma chamada assíncrona e, para isso, tornamos nosso action assíncrono.

A chamada ao método **ReadAsAsync** requer a presença de **System.Net.Http.Formatting**, local onde ele está definido como método de extensão.

Para passarmos e recebermos objetos da API, o **HttpClient** ainda disponibiliza os seguintes métodos, responsáveis por serializar os objetos a serem passados:

- **PostAsJsonAsync**: Envia uma requisição POST para uma URI específica com o objeto serializado como JSON;
- **PostAsXmlAsync**: Envia uma requisição POST para uma URI específica com o objeto serializado como XML;
- **PutAsJsonAsync**: Envia uma requisição PUT para uma URI específica com o objeto serializado como JSON;
- **PutAsXmlAsync**: Envia uma requisição PUT para uma URI específica com o objeto serializado como XML.

No código a seguir, apresentamos o código que exemplifica o uso desses métodos. No projeto, vamos escrever um action chamado **Incluir**, cujo propósito é receber de um formulário os dados de um objeto **Pessoa** e enviá-los para a API:

```
[HttpPost]
public async Task<IActionResult> Incluir(Pessoa pessoa)
{
    client.BaseAddress = new Uri("http://localhost:62291/");
    client.DefaultRequestHeaders.Accept.Add(new
        MediaTypeWithQualityHeaderValue("application/json"));

    HttpResponseMessage response = await client
        .PostAsJsonAsync<Pessoa>("api/home", pessoa);

    if (response.IsSuccessStatusCode)
    {
        return Content("Incluído com sucesso");
    }
    else
    {
        return Content("Ocorreu um erro");
    }
}
```

Nesse exemplo, estamos assumindo que haja o método **PostPessoa** implementado na nossa API. Para relembrar, o código que recebe o objeto é este:

```
using Microsoft.AspNetCore.Mvc;

namespace Capitulo10.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : ControllerBase
    {
        [HttpPost]
        public void PostPessoa([FromBody] Pessoa pessoa)
        {
            //processa a operação
        }
    }
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

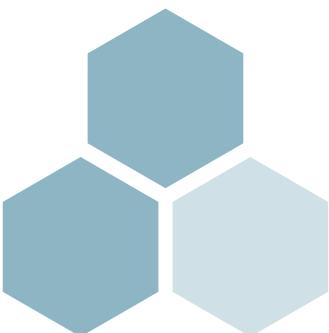
- Assim como todas as tarefas no .NET Core, o Web API Core também utiliza os serviços por meio de injeção de dependência;
- Para que um action no Web API Core receba objetos da requisição, podemos usar os seguintes Binding Sources: **FromBody**, **FromQuery**, **FromRoute**, **FromForm** e **FromHeader**;
- Para utilizar a classe **HttpClient**, devemos registrar o serviço **IHttpClientFactory** por meio do método **AddHttpClient**.



10

Web API Core

Teste seus conhecimentos



1. Para especificar que um controller representará um serviço Web API Core, qual atributo podemos adicionar?

- a) WebApi
- b) ApiService
- c) ApiCoreService
- d) ApiController
- e) ServiceController

2. Para ler informações do corpo de uma requisição por meio do verbo HTTP POST, qual atributo podemos usar no parâmetro do action?

- a) FromBody
- b) FromQuery
- c) FromRoute
- d) FromJson
- e) FromSource

3. Qual serviço podemos registrar para definir uma instância de HttpClient?

- a) IApiFactory
- b) IApiClientFactory
- c) IHttpclientFactory
- d) IHttpclientService
- e) IApiService

4. Para criar uma instância de HttpClient no controller, podemos usar qual método dessa classe?

- a) CreateClient
- b) CreateHttp
- c) CreateHttpClient
- d) CreateInstance
- e) GetInstance

5. Para lidar com retornos provenientes da execução do método GetAsync e chamar a propriedade IsSuccessStatusCode, qual o tipo de retorno esperado?

- a) HttpCreateResponse
- b) HttpRequestMessage
- c) HttpClientMessage
- d) HttpResponseMessage
- e) HttpResponse

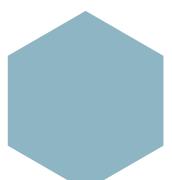


10

Web API Core



Mãos à obra!



Laboratório 1

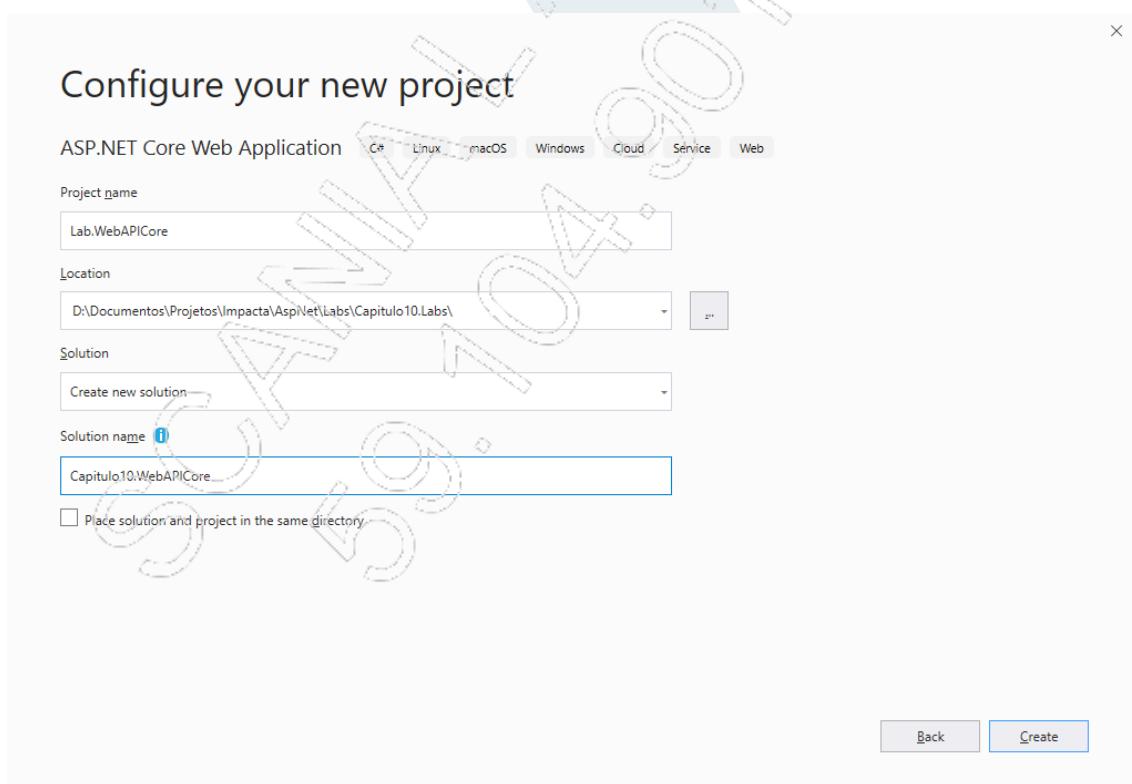
Objetivos:

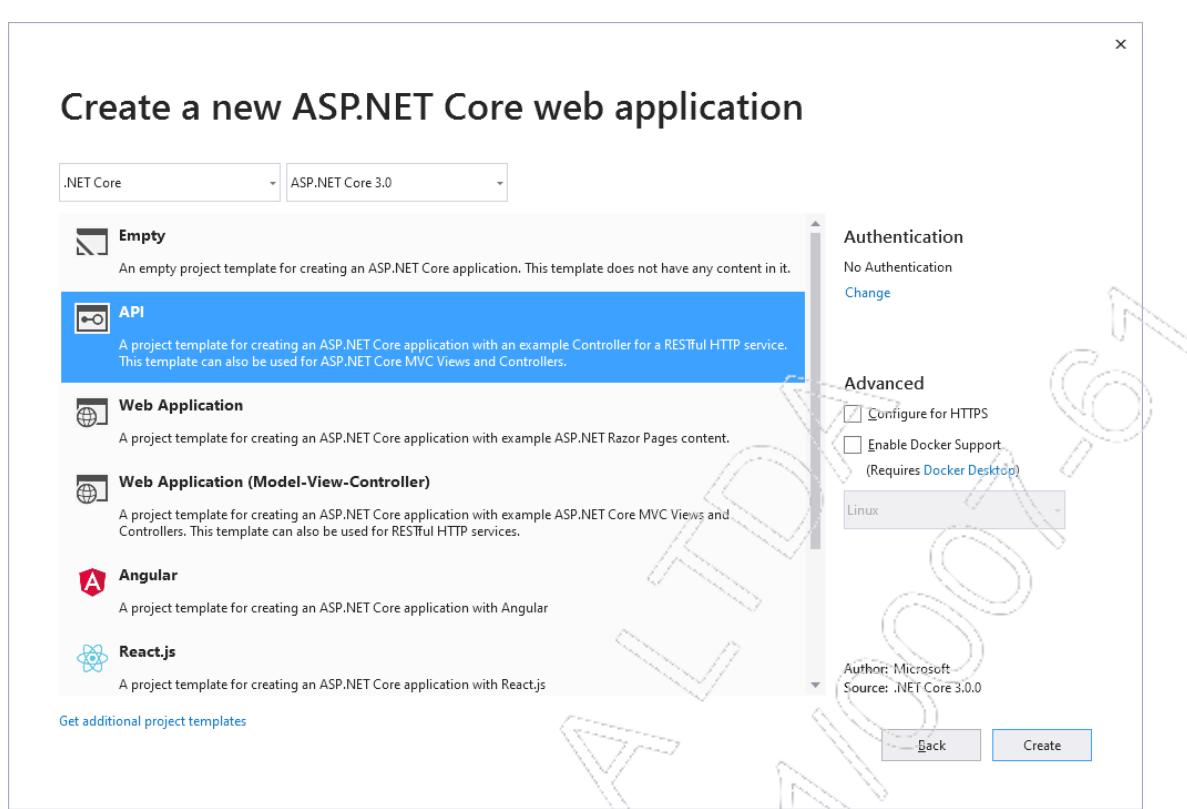
Neste laboratório, desenvolveremos um serviço REST usando o Web API Core. Seu objetivo é semelhante àquele que definimos no Capítulo 6, mas, desta vez, o pagamento será de um evento por um convidado. A proposta é ilustrar o uso do Web API Core.

Na primeira parte, criaremos o serviço. Na segunda parte, desenvolveremos o cliente do nosso serviço, implementando o recurso de pagamento na aplicação.

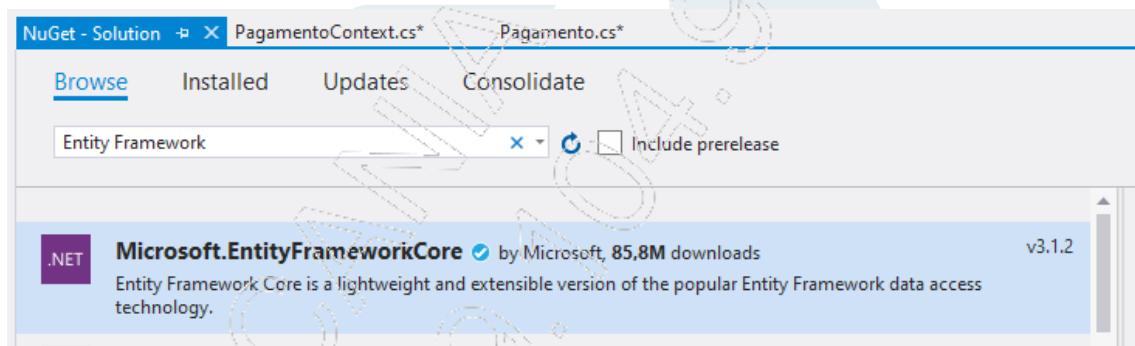
A - Desenvolvimento do serviço Web API Core

1. Na pasta **Labs**, crie uma pasta chamada **Capítulo10.Labs**;
2. Nessa nova pasta, crie um projeto chamado **Lab.WebAPICore** em uma solution chamada **Capítulo10.WebAPICore**. O projeto deve ser do tipo **ASP.NET Core Web Application**:

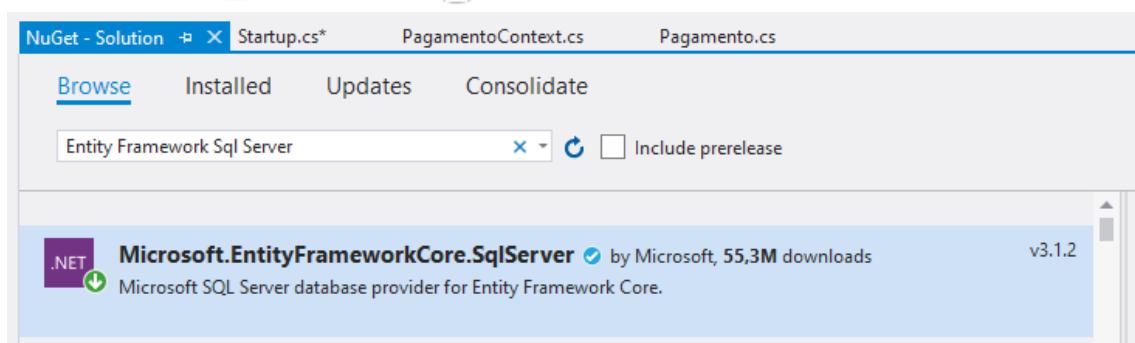




3. Adicione a referência ao Entity Framework Core, via NuGet:



Adicione, também, a referência **Microsoft.EntityFrameworkCore.SqlServer**:



4. Vamos criar a pasta **Models** e, nessa pasta, a classe **Pagamento**:

```
namespace Lab.WebAPICore.Models
{
    public class Pagamento
    {
        public int Id { get; set; }
        public int IdEvento { get; set; }
        public string Cpf { get; set; }
        public string NumeroCartao { get; set; }
        public double Valor { get; set; }
        public int Status { get; set; }
    }
}
```

5. Nessa mesma pasta, crie a classe **PagamentoContext**:

```
using Microsoft.EntityFrameworkCore;

namespace Lab.WebAPICore.Models
{
    public class PagamentoContext : DbContext
    {
        public PagamentoContext(DbContextOptions<PagamentoContext>
            options) : base(options) { }

        public DbSet<Pagamento> Pagamentos { get; set; }

        protected override void OnModelCreating(ModelBuilder
            modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<Pagamento>().ToTable("TBpagamentos");

            modelBuilder.Entity<Pagamento>()
                .Property(p => p.Cpf)
                .IsRequired()
                .HasMaxLength(11);
            modelBuilder.Entity<Pagamento>()
                .Property(p => p.NumeroCartao)
                .IsRequired()
                .HasMaxLength(16);
        }
    }
}
```

6. Vamos registrar o **DbContext** com o container de injeção de dependência. Esses serviços estarão disponíveis para os controladores. Esse trabalho será realizado na classe **Startup.cs**, método **ConfigureServices**:

```
using Lab.WebAPICore.Models;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Lab.WebAPICore
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<PagamentoContext>(options =>
                options.UseSqlServer(Configuration
                    .GetConnectionString("ApiConnection")));

            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

7. Abra o arquivo **appsettings.json**. Inclua o trecho destacado, representando a string de conexão:

```
{  
    "ConnectionStrings": {  
        "ApiConnection": "Integrated Security=SSPI;Persist Security Info=False;Initial Catalog=DB_PAGAMENTOS_CORE;Data Source=.\SQLEXPRESS"  
    },  
  
    "Logging": {  
        "LogLevel": {  
            "Default": "Information",  
            "Microsoft": "Warning",  
            "Microsoft.Hosting.Lifetime": "Information"  
        }  
    },  
    "AllowedHosts": "*"  
}
```

8. Na pasta **Models**, inclua o arquivo **DBInitializer.cs** com o conteúdo indicado (essa classe, como no outro projeto, será usada para criar o banco de dados):

```
namespace Lab.WebAPICore.Models  
{  
    public class DBInitializer  
    {  
        public static void Initialize(PagamentoContext context)  
        {  
            context.Database.EnsureCreated();  
        }  
    }  
}
```

9. Na classe **Program.cs**, importe o namespace **Microsoft.Extensions.DependencyInjection**. Em seguida, abra o método **Main()**:

```
using Lab.WebAPICore.Models;  
using Microsoft.AspNetCore.Hosting;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
using Microsoft.Extensions.Logging;  
using System;  
  
namespace Lab.WebAPICore  
{  
    public class Program  
    {  
        public static void Main(string[] args)  
        {  
            var host = CreateHostBuilder(args).Build();  
  
            using (var scope = host.Services.CreateScope())  
            {
```

```

        var services = scope.ServiceProvider;
        try
        {
            var context = services
                .GetRequiredService<PagamentoContext>();
            DBInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services
                .GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, ex.Message);
            throw;
        }
    }

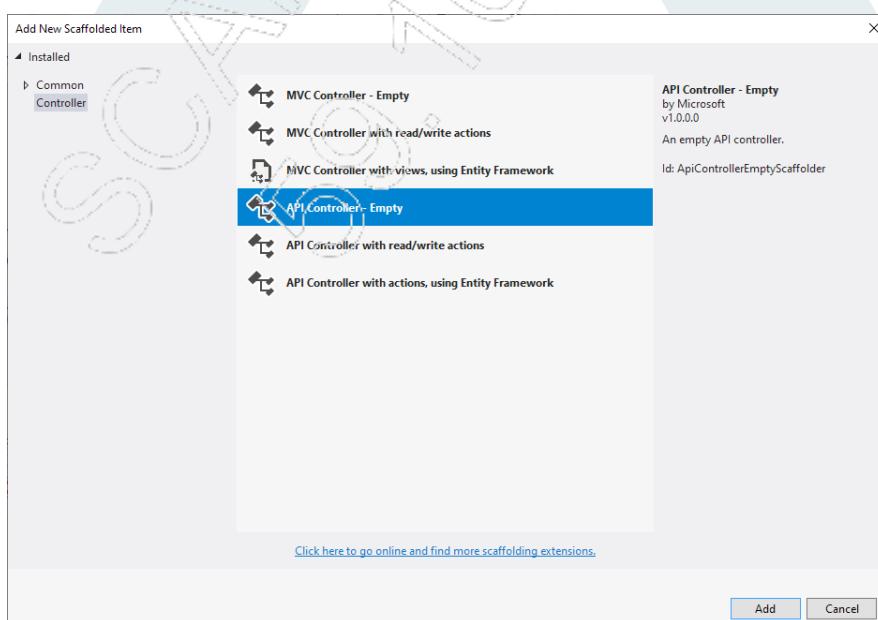
    host.Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
}

```

10. Em seguida, execute essa aplicação e verifique se o banco de dados foi criado;

11. Adicione um controller na pasta **Controllers** chamado **PagamentosController**. A opção deve ser **API Controller**:



12. Em seguida, substitua o conteúdo pelo indicado adiante:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Lab.WebAPICore.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Lab.WebAPICore.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class PagamentosController : ControllerBase
    {
        private readonly PagamentoContext _context;

        public PagamentosController(PagamentoContext context)
        {
            this._context = context;
        }

    }
}
```

13. Adicione, agora, os métodos com os serviços:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Lab.WebAPICore.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Lab.WebAPICore.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class PagamentosController : ControllerBase
    {
        private readonly PagamentoContext _context;

        public PagamentosController(PagamentoContext context)
        {
            this._context = context;
        }

        //HTTP GET - Listar todos os pagamentos
        [HttpGet]
        public IEnumerable<Pagamento> GetAll()
        {
            return _context.Pagamentos.ToList<Pagamento>();
        }
}
```

```
//HTTP POST - Inclusão de um pagamento
public IActionResult PostPagamento([FromBody] Pagamento pago)
{
    var pagamento = _context.Pagamentos.FirstOrDefault(p =>
        p.IdEvento == pago.IdEvento && p.Cpf.Equals(pago.Cpf));

    if (pagamento == null)
    {
        pago.Status = 1;
        _context.Pagamentos.Add(pago);
        _context.SaveChanges();

        return Ok();
    }
    else
    {
        return BadRequest();
    }
}
```

14. Altere o conteúdo do arquivo **launchSettings.json**, na pasta **Properties**. Abra esse arquivo e altere as propriedades assinaladas adiante:

```
{
    "$schema": "http://json.schemastore.org/launchsettings.json",
    "iisSettings": {
        "windowsAuthentication": false,
        "anonymousAuthentication": true,
        "iisExpress": {
            "applicationUrl": "http://localhost:61396",
            "sslPort": 0
        }
    },
    "profiles": {
        "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": true,
            "launchUrl": "api/Pagamentos",
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        },
        "Lab.WebAPICore": {
            "commandName": "Project",
            "launchBrowser": true,
            "launchUrl": "api/Pagamentos",
            "applicationUrl": "http://localhost:5000",
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        }
    }
}
```

15. Execute a aplicação. O banco de dados será criado e a rota **api/Pagamentos** será executada. Como ela executa implicitamente o verbo HTTP GET, teremos uma lista de pagamentos vazia:



B - Desenvolvimento do cliente Web API Core

1. Na pasta **Capítulo10.Labs**, adicione um novo solution, vazio, chamado **Capítulo10.ClienteWebAPICore**;
2. Copie o projeto **Lab.NetCore** do solution **Capítulo09.Labs** para a pasta do seu novo solution, **Capítulo10.ClienteWebAPICore**;
3. Adicione esse projeto ao novo solution. Execute a aplicação para testar se está tudo OK;
4. Vamos realizar as alterações necessárias no projeto para contemplar o recurso de pagamento. Na pasta **Models**, crie a classe **PagamentoEvento**. Essa classe deve ter as mesmas propriedades da classe **Pagamento**, descrita no Web service (esse procedimento é importante para evitar configurações desnecessárias para o nosso propósito):

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

namespace Lab.NetCore.Models
{
    public class PagamentoEvento
    {
        public int IdEvento { get; set; }
        public string Cpf { get; set; }
        public string NumeroCartao { get; set; }
        public double Valor { get; set; }
        public int Status { get; set; }
    }
}
```

5. No arquivo `_ListarParticipantes.cshtml` (pasta `Views/Shared`), defina um link direcionando o usuário para o pagamento:

```
@model IEnumerable<Lab.NetCore.Models.Participante>



| @Html.DisplayNameFor(model => model.Nome)                                                                     |
|---------------------------------------------------------------------------------------------------------------|
| @Html.DisplayNameFor(model => model.Email)                                                                    |
| @Html.DisplayNameFor(model => model.Cpf)                                                                      |
| @Html.DisplayNameFor(model => model.DataNascimento)                                                           |
| </th>                                                                                                         |
| @Html.DisplayFor(modelItem => item.Nome)                                                                      |
| @Html.DisplayFor(modelItem => item.Email)                                                                     |
| @Html.DisplayFor(modelItem => item.Cpf)                                                                       |
| @Html.DisplayFor(modelItem => item.DataNascimento)                                                            |
| @Html.ActionLink("Efetuar Pagamento", "EfetuarPagamento", new { id = item.Id, idEvento = item.EventoInfoId }) |


```

6. Vamos, agora, definir o action **EfetuarPagamento**, responsável por gerar a view para fornecer os dados do cartão de crédito, e a versão POST, que efetiva o pagamento no Web service. Escreva esse action no controller **ParticipantesController**. Além dos actions, estamos fornecendo os recursos de injeção de dependência para o serviço **IHttpClientFactory**:

```
using Lab.NetCore.Dao;
using Lab.NetCore.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Newtonsoft.Json;
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;

namespace Lab.NetCore.Controllers
{
    public class ParticipantesController : Controller
    {
        private EventosDao eventosDao { get; set; }
        private ParticipantesDao participantesDao { get; set; }

        private HttpClient client;

        public ParticipantesController(EventosContext context,
            IHttpClientFactory httpClient)
        {
            this.eventosDao = new EventosDao(context);
            this.participantesDao = new ParticipantesDao(context);

            client = httpClient.CreateClient();
        }

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

        [HttpGet]
        public IActionResult IncluirParticipante()
        {
            ViewBag.ListaDeEventos = new
                SelectList(eventosDao.Listar(), "Id", "Descricao");

            return View();
        }
    }
}
```

```
[HttpPost]
public IActionResult IncluirParticipante(Participante participante)
{
    if (participante.EventoInfoId == 0)
    {
        ModelState.AddModelError("IdEvento",
            "Nenhum evento selecionado ");
    }

    if (!ModelState.IsValid)
    {
        return IncluirParticipante();
    }
    participantesDao.Executar(participante, TipoOperacaoBD.Added);
    return RedirectToAction("Index");
}

public IActionResult ListarParticipantes(int idEvento)
{
    ViewBag.ListaDeEventos = new
        SelectList(eventosDao.Listar(), "Id", "Descricao");

    if (idEvento == 0)
    {
        return View();
    }
    else
    {
        var lista = participantesDao.ListarPorEvento(idEvento);
        return PartialView("_ListarParticipantes", lista);
    }
}

[HttpGet]
public IActionResult EfetuarPagamento(int id, int idEvento)
{
    var participante = participantesDao.BuscarPorId(id);
    var evento = eventosDao.BuscarPorId(idEvento);

    PagamentoEvento pagamento = new PagamentoEvento()
    {
        IdEvento = evento.Id,
        Cpf = participante.Cpf,
        Valor = evento.Preco
    };
    return View(pagamento);
}

[HttpPost]
public async Task EfetuarPagamento(
    PagamentoEvento pagamento)
{
    try
    {
        client.BaseAddress = new Uri("http://localhost:62291/");
        client.DefaultRequestHeaders.Accept.Add(new
            MediaTypeWithQualityHeaderValue("application/json"));
    
```

```
//gerar o json a partir do objeto
string json = JsonConvert.SerializeObject(pagamento);

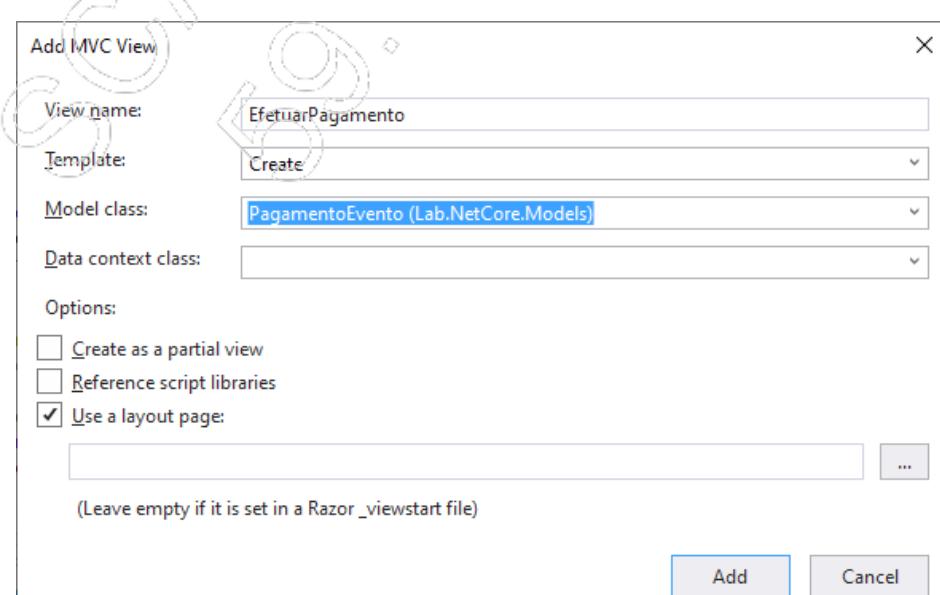
//gerar o fluxo de bytes a ser enviado para o serviço
HttpContent content = new StringContent(
    json, Encoding.Unicode, "application/json");

        var response = await client.PostAsync("api/pagamentos",
content);

//return RedirectToAction("Index");

if (response.IsSuccessStatusCode)
{
    return RedirectToAction("Index");
}
else
{
    string msg = response.StatusCode + " - " +
        response.ReasonPhrase;
    throw new Exception(msg);
}
}
catch (Exception ex)
{
    ViewData["MensagemErro"] = ex.Message;
    return View("Erro");
}
}
}
}
```

7. Vamos adicionar a view para efetuar o pagamento, usando o template **Create** com o model **PagamentoEvento**. Observe as alterações realizadas no início da view:



```
@model Lab.NetCore.Models.PagamentoEvento

{@
    ViewData["Title"] = "Efetuar Pagamento";
}

<h1>Efetuar Pagamento</h1>

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="EfetuarPagamento">
            <div asp-validation-summary="ModelOnly"
                class="text-danger"></div>
            <input type="hidden" asp-for="IdEvento" />
            @*<div class="form-group">
                <label asp-for="IdEvento" class="control-label"></label>
                <input asp-for="IdEvento" class="form-control" />
                <span asp-validation-for="IdEvento"
                    class="text-danger"></span>
            </div>*@
            <div class="form-group">
                <label asp-for="Cpf" class="control-label"></label>
                <input asp-for="Cpf" class="form-control" />
                <span asp-validation-for="Cpf"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="NumeroCartao" class="control-label"></label>
                <input asp-for="NumeroCartao" class="form-control" />
                <span asp-validation-for="NumeroCartao"
                    class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Valor" class="control-label"></label>
                <input asp-for="Valor" class="form-control"
                    readonly="readonly" />
                <span asp-validation-for="Valor"
                    class="text-danger"></span>
            </div>
            @*<div class="form-group">
                <label asp-for="Status" class="control-label"></label>
                <input asp-for="Status" class="form-control" />
                <span asp-validation-for="Status"
                    class="text-danger"></span>
            </div>*@
            <div class="form-group">
                <input type="submit" value="Efetuar Pagamento"
                    class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

8. No método **ConfigureServices** da classe **Startup**, adicione o método **AddHttpClient** para habilitar o serviço **IHttpClientFactory**:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<EventosContext>(options => options.UseSqlServer(
        Configuration.GetConnectionString("EventosConnection")));

    //services.AddDbContext<ApplicationDbContext>(options =>
    //    options.UseSqlServer(
    //        Configuration.GetConnectionString("DefaultConnection")));

    services.AddDbContext<UsuariosDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    //services.AddDefaultIdentity<Usuario>(options => options
    //    .SignIn.RequireConfirmedAccount = true)
    //    .AddEntityFrameworkStores<UsuariosDbContext>();

    services.AddIdentity<Usuario, IdentityRole>()
        .AddEntityFrameworkStores<UsuariosDbContext>()
        .AddDefaultTokenProviders();

    //redefinição dos padrões do Identity Core
    services.ConfigureApplicationCookie(options =>
    {
        options.LoginPath = "/Usuarios/Login";
        options.LogoutPath = "/Usuarios/Logout";
        options.AccessDeniedPath = "/Usuarios/AccessDenied";
    });

    services.AddHttpClient();
    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

Considere estas duas observações importantes:

- O projeto referente ao Web service deve estar em execução, para que possamos acessá-lo;
- O link mostrado nesse exemplo deve ser alterado para o link correto no momento da sua execução, pois o número da porta certamente mudará de projeto para projeto.

9. Teste a aplicação, fornecendo os dados do cartão. Após a inclusão, teste o Web service para ver se a lista de pagamentos no formato JSON aparece no browser.

Como tarefa / desafio, apresente a lista de pagamentos na aplicação. Implemente um novo action e uma nova view. Forneça, também, um link para a listagem de pagamentos do serviço.