

Visual Studio 2015

- ASP.NET com C#

Acesso a dados



Visual Studio 2015

- ASP.NET com C#

Acesso a dados



IMPACTA
EDITORADA

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."

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Coordenação Geral
Marcia M. Rosa

Revisão Ortográfica e Gramatical
Fernanda Monteiro Laneri

Coordenação Editorial
Henrique Thomaz Bruscagin

Diagramação
Bruno de Oliveira Santos

Atualização
José Eduardo Machado Grasso

Edição nº 1 | 1798_0
Julho/ 2016



Este material constitui uma nova obra e é uma derivação da seguinte obra original, produzida por TechnoEdition Editora Ltda., em Out/2014: ASP.NET 2013 com C# - Acesso a Dados

Autoria: José Eduardo Machado Grasso

Sumário

Informações sobre o treinamento	09
Capítulo 1 - Acesso a dados com ASP.NET	11
1.1. Introdução	12
1.2. Conceitos.....	12
1.2.1. Banco de dados.....	12
1.2.2. Base de dados.....	12
1.2.3. SGBD – Sistema Gerenciador de Banco de Dados	12
1.2.4. Repositório de dados	13
1.3. Modelos de acesso a dados	13
1.3.1. ADO.NET – ActiveX Data Objects for .NET	14
1.3.2. ORM – Object-Relational Mapping.....	16
1.3.3. Micro-ORM – Micro Object-Relacional Mapping	17
1.3.4. NoSQL Database – Not Only SQL Database	19
1.3.5. WCF Data Services.....	19
1.4. Bancos de dados de exemplo	21
1.4.1. Northwind.....	21
1.4.2. Pubs	26
1.4.3. AdventureWorks.....	30
Pontos principais	38
Teste seus conhecimentos.....	39
Mãos à obra!.....	43
Capítulo 2 - ADO.NET com Web Forms	57
2.1. Introdução	58
2.2. Classes do ADO.NET	58
2.2.1. DbConnection	60
2.2.2. DbCommand.....	63
2.2.3. DbDataReader.....	73
2.3. Transação	78
2.3.1. DbTransaction	79
2.4. Dados desconectados	81
2.4.1. DataSet	82
2.4.1.1. DataTable	83
2.5. Melhores práticas.....	90
2.6. Considerações sobre o uso das classes ADO.NET	94
Pontos principais	95
Teste seus conhecimentos.....	97
Mãos à obra!.....	101

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Capítulo 3 - ADO.NET com MVC.....	119
3.1. Introdução	120
3.2. Separação das responsabilidades	120
3.3. Retorno de métodos	121
3.3.1. DbDataReader.....	121
3.3.2. DataTable ou DataSet.....	122
3.3.3. Colecões fortemente tipadas (arrays ou coleções genéricas).....	123
3.4. Controllers e Views para exibir/alterar dados	124
3.4.1. Listagem	128
3.4.2. Inclusão	132
3.4.3. Inclusão – POST.....	137
3.4.4. Exibição de um registro (Details).....	138
3.4.5. Alteração de um registro (Update).....	141
3.4.6. Exclusão de um registro (Delete).....	143
Pontos principais	145
 Teste seus conhecimentos.....	147
 Mãos à obra!.....	151
Capítulo 4 - Entity Framework (Code First).....	161
4.1. Introdução	162
4.2. Preparando o ambiente	164
4.3. Visão geral.....	167
4.3.1. Modelo de domínio	167
4.3.2. Contexto da conexão	172
4.3.3. Mapeamento	180
4.3.4. Mapeamento por código	186
4.3.4.1. Método OnModelCreating	186
4.3.5. Database Initializer	191
4.3.6. Migrations	192
Pontos principais	196
 Teste seus conhecimentos.....	197
 Mãos à obra!.....	201

Sumário

Capítulo 5 - Entity Framework (Model/Database First).....	215
5.1. Introdução	216
5.2. Model First.....	217
5.2.1. Adicionando uma Entity (entidade).....	219
5.2.2. Adicionando propriedades	221
5.2.3. Adicionando associações	224
5.2.4. Criando o banco de dados	228
5.2.5. Entendendo os arquivos gerados.....	231
5.2.6. Usando o modelo criado	238
5.3. Database First.....	239
5.3.1. Usando o modelo criado	249
Pontos principais	250
Teste seus conhecimentos.....	251
Mãos à obra!	255
Capítulo 6 - Manipulando imagens.....	281
6.1. Introdução	282
6.2. Upload de imagens (Web Forms)	282
6.3. Upload de imagens (MVC)	286
6.4. Tratamento de imagens	288
6.4.1. Criando miniaturas (thumbnails)	292
6.5. Gravando imagens no banco de dados	297
6.6. Gravando imagens com ADO.NET.....	299
6.7. Exibindo imagens com um handler	303
6.8. Gravando e exibindo imagens com Entity Framework	309
6.9. Northwind e as imagens gravadas no banco de dados	312
6.10. Vinculando imagens no GridView	314
Pontos principais	321
Teste seus conhecimentos.....	323
Mãos à obra!	327
Capítulo 7 - Data Services: WCF.....	365
7.1. Introdução	366
7.2. Por que usar serviços	366
7.3. Tecnologias envolvidas	367
7.4. Conceitos iniciais sobre WCF.....	370
7.4.1. Criando um serviço WCF	372
7.4.1.1. Web.Config	375
7.4.2. Criando um cliente.....	378
7.4.3. Data Contract.....	385
7.4.4. Criando um host	393
7.4.4.1. Criando a aplicação cliente para este host.....	396
Pontos principais	399
Teste seus conhecimentos.....	401
Mãos à obra!	405

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Capítulo 8 - Serviços: Web API	425
8.1. Introdução à Web API	426
8.2. REST na prática	427
8.3. Usando a Web API	430
8.3.1. Estrutura Web API	431
8.3.2. Criando um serviço REST.....	435
8.3.2.1. Testando o método GET com parâmetros.....	438
8.3.2.2. Testando o método POST.....	439
8.3.3. Criando um cliente REST	444
Pontos principais	453
Teste seus conhecimentos.....	455
Mãos à obra!.....	459
 Capítulo 9 - NoSQL.....	 477
9.1. Modelo relacional.....	478
9.2. NoSQL.....	479
9.3. Bancos de dados NoSQL.....	481
9.3.1. MongoDB	482
9.3.2. Instalando o servidor	483
9.3.3. Instalando o cliente.....	486
9.3.4. Visão geral.....	487
9.3.5. Conectando o banco	489
9.3.6. Mapeando o Model Domain.....	490
9.3.7. CRUD: inserindo, alterando, excluindo e pesquisando	492
9.3.7.1. Inserindo dados	493
9.3.7.2. Alterando dados	493
9.3.7.3. Excluindo dados	494
9.3.8. Usando a LINQ para obter dados em uma coleção	494
9.3.9. Armazenando objetos complexos	495
Pontos principais	502
Teste seus conhecimentos.....	503
Mãos à obra!.....	507

Informações sobre o treinamento

Para o melhor aproveitamento do curso **Visual Studio 2015 – ASP.NET com C# Acesso a Dados**, é imprescindível ter participado do curso Visual Studio 2015 – ASP.NET com C# Fundamentos, ou possuir conhecimentos equivalentes.

1

Acesso a dados com ASP.NET

- ✓ Conceitos;
- ✓ Modelos de acesso a dados;
- ✓ Bancos de dados de exemplo.



IMPACTA
EDITORA

1.1. Introdução

Neste capítulo, teremos uma visão geral das opções disponíveis nas aplicações ASP.NET para ler e manipular informações contidas em repositórios de dados. Esses repositórios podem ser informações vindas de programas gerenciadores de dados, como SQL Server ou Oracle, arquivos de texto em formatos diversos, como XML, JSON ou CSV, serviços disponibilizados na Internet ou objetos armazenados na memória de uma aplicação.

1.2. Conceitos

O primeiro passo para podermos trabalhar com essas aplicações é entender os termos e conceitos utilizados para descrever gerenciamento e armazenamento de informações. Os principais conceitos são: banco de dados, base de dados, sistemas gerenciadores de banco de dados e repositório de dados. Embora os nomes sejam parecidos, esses termos se referem a coisas diferentes, conforme veremos a seguir.

1.2.1. Banco de dados

Um banco de dados é um conjunto de informações sobre um determinado assunto. Essas informações são relacionadas, agrupadas e organizadas para facilitar a consulta e a manipulação dos dados.

1.2.2. Base de dados

Base de dados é um conjunto de informações disponíveis que pode ser utilizado como fonte para um banco de dados. As bases de dados não precisam necessariamente estar relacionadas ou organizadas.

1.2.3. SGBD – Sistema Gerenciador de Banco de Dados

Um **Sistema Gerenciador de Banco de Dados**, ou **SGBD** (Data Base Management System – DBMS), é um software que permite a criação e o gerenciamento de bancos de dados. Alguns SGBD conhecidos são: Microsoft SQL Server, Oracle, MySQL e PostgreSQL.

Esses softwares podem organizar internamente dados de diversas maneiras. As mais comuns são as seguintes:

- **Modelo relacional:** Os dados são estruturados em tabelas, linhas e colunas, e as informações são relacionadas entre as tabelas por meio de colunas em comum. **Microsoft SQL Server** e **Oracle** são gerenciadores de dados que utilizam esse modelo;
- **Orientado a documentos:** Os dados são armazenados sem uma estrutura fixa, usando chaves para localizar cada documento. **Cassandra** e **RavenDB** são gerenciadores de banco de dados que utilizam essa forma de armazenamento;
- **Orientado a objetos:** Os dados neste tipo de sistema gerenciador são armazenados por meio de instâncias de classes e podem ser recuperados realizando-se consultas que utilizem uma instância como modelo de filtro. Um famoso gerenciador de banco de dados desta categoria é o **db4o (Database for Objects)**.

É muito frequente o uso do termo “banco de dados” para se referir a um Sistema Gerenciador de Banco de Dados. Frases como “o banco de dados Oracle vai receber uma atualização este semestre” são comuns em livros e em artigos de revistas ou da Internet.

1.2.4. Repertório de dados

Um repertório de dados é a parte real (arquivo, objeto) utilizada por um programa gerenciador de dados para armazenar um banco de dados ou parte dele.

1.3. Modelos de acesso a dados

Durante o desenvolvimento de um aplicativo, existem muitas decisões que devem ser tomadas em relação ao modo de se obter, manipular, exportar e exibir dados. A plataforma .NET oferece muitas opções de implementação para cada situação. Existem Web Controls que conectam diretamente um banco de dados, frameworks que transferem dados de um repertório para objetos em memória, recursos do Visual Studio que criam telas de cadastro automaticamente e classes desenvolvidas para conectar, manipular e armazenar dados. A seguir, um pequeno resumo dos principais recursos disponibilizados pela plataforma .NET para acesso a dados.

1.3.1. ADO.NET – ActiveX Data Objects for .NET

ADO.NET é a tecnologia central de acesso a dados da plataforma .NET. São as classes exclusivas para acesso a dados. Essa é a maneira mais direta de manipulação de informações e a que apresenta melhor performance. Todas as outras formas utilizam as classes do ADO.NET, encapsulando parte do código que o programador deve escrever para ler ou alterar dados de um repositório.

A estrutura do ADO.NET é fundamentada no conceito de providers, que são classes que manipulam um formato de banco de dados específico. Todos os providers utilizam as classes abstratas do namespace **System.Data.Common**. Isso significa que, independentemente de qual repositório de dados for usado, as propriedades e os métodos utilizados serão os mesmos.

As principais classes bases para acesso a dados são as seguintes:

- **DbConnection**;
- **DbCommand**;
- **DbDataReader**.

O provider para o SQL Server, por exemplo, utiliza o namespace **System.Data.SqlClient** e implementa as seguintes classes concretas:

- **SqlConnection**;
- **SqlCommand**;
- **SqlDataReader**.

O provider para Oracle, no namespace **System.Data.OracleClient**, implementa as classes **OracleConnection**, **OracleCommand** e **OracleDataReader**. Todos os outros providers seguem o mesmo critério.

Vejamos um exemplo de um código usando ADO.NET:

```
//String de conexão
string conexao=@"Data Source=localhost\sqlexpress;
                Initial Catalog=Northwind;
                Integrated Security=True";

//Expressão SQL a ser executada
string sql="SELECT CompanyName FROM Customers";

//objeto para conectar o banco
var cn=new SqlConnection(conexao);

//objeto para executar comando
var cmd=new SqlCommand(sql, cn);

//objeto para ler dados
SqlDataReader dr;

//Abre a conexão
cn.Open();

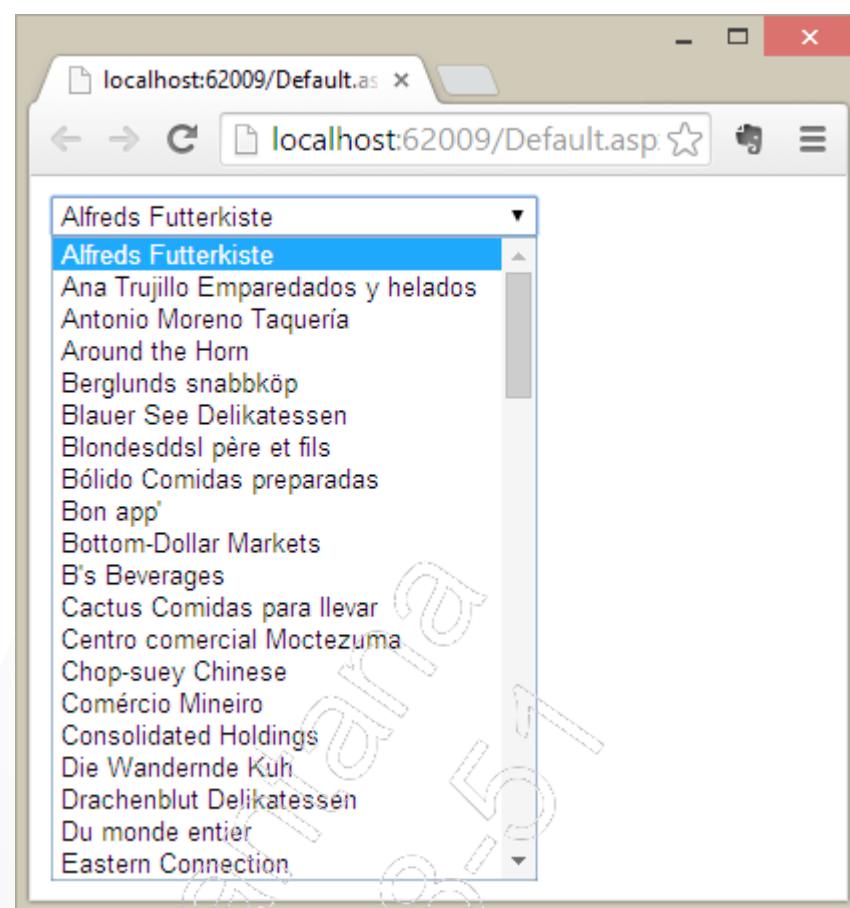
//Obtém a instância do leitor de dados
dr = cmd.ExecuteReader();

//Lê todos os dados, inserindo cada registro
//nos itens de um DropDownList
while (dr.Read())
{
    //Obtém o conteúdo de um campo do registro atual
    string nome = dr[“CompanyName”].ToString();

    //Adiciona nos itens do DropDown
    DropDownList1.Items.Add(nome);
}
//Fecha o leitor de dados
dr.Close();

//Fecha a conexão
cn.Close();
```

A seguir, veja o resultado do processamento em uma página ASP.NET usando Web Forms:



O exemplo anterior conecta o gerenciador de banco de dados **SQL Server**, instalado no computador local (localhost) com um nome de instância chamado **SQL Express**, conectando um banco de dados chamado **Northwind**, e lendo dados de uma tabela chamada **Customers**. Para o exemplo funcionar, é necessário ter o SQL Server instalado no computador e ter o banco de dados de exemplo criado. Este capítulo tem como objetivo preparar este ambiente. Os detalhes serão vistos mais adiante. No momento, é importante ter uma visão geral dos principais modos de utilizar dados em uma aplicação.

1.3.2. ORM – Object-Relational Mapping

Mapeamentos objeto-relacionais são programas ou componentes que facilitam a transposição de informações de banco de dados para objetos em memória.

Usando ADO.NET, o programador deve escrever o código manualmente para obter informações do banco, criar instâncias de classes que representam os dados e preencher essas instâncias com as informações coletadas. Esse processo por vezes é tedioso e repetitivo.

Os programas ORM encapsulam toda a parte do código relacionada a leitura e gravação, deixando o programador com a parte de escrever a lógica do sistema, sem se preocupar com o tratamento de conversão de tipos, expressões em SQL ou gerenciamento de memória.

O principal programa desta categoria é o **Entity Framework**, que é um componente open source mantido pela Microsoft e que está em constante evolução pela contribuição de programadores do mundo todo.

Vejamos o mesmo exemplo do item anterior usando o Entity Framework:

```
//Cria uma instância da classe de acesso ao banco
var db = new NorthwindEntities();

//Percorre a lista de objetos do tipo "Customers"
foreach (var cliente in db.Customers)
{
    //Adiciona da lista
    DropDownList1.Items.Add(cliente.CompanyName);
}
```

Repare que todo o processo de definir a string de conexão, definir a expressão SQL, abrir a conexão, obter e converter os dados, fechar o leitor de dados e fechar a conexão são encapsulados pelos mecanismos de acesso a dados do Entity Framework. Nos bastidores, ainda é usado o ADO.NET (ou seja, as classes **SqlConnection**, **SqlCommand** e **SqlDataReader**), mas isso fica escondido do programador, cuja única preocupação é manipular os dados por meio de classes que os representam. Para cada tabela do banco existe uma classe com propriedades equivalente aos campos desta classe.

Outros programas ORM bastante conhecidos são **Linq to SQL** e **NHibernate**.

1.3.3. Micro-ORM – Micro Object-Relacional Mapping

Assim como os ORMs, são programas que mapeiam as informações de um banco de dados para objetos em memória, mas são mais simples e mais rápidos. Eles têm, porém, menos recursos do que os grandes programas ORM como Entity Framework ou NHibernate.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Geralmente, esses programas são compostos apenas de um arquivo (DLL) que é facilmente inserido em uma aplicação. A performance desses componentes tende a ser superior por realizar apenas a tarefa de transpor informações do banco para a memória. Os ORMs mais robustos, como o Entity Framework, possuem recursos de sincronização de atualizações, carregamento dinâmico e cache de dados, tarefas estas que acabam deixando o sistema mais lento e pesado.

Alguns micro-ORMs famosos são: **Dapper**, **Massive**, **PetaPoco** e **LightSpeed**.

Vejamos, a seguir, o mesmo exemplo dos itens anteriores usando Dapper:

```
//String de conexão
string conexao=@"Data Source=localhost\sqlexpress;
                Initial Catalog=Northwind;
                Integrated Security=True";

//Expressão SQL a ser executada
string sql="SELECT CustomerId, CompanyName FROM Customers";

//Cria a conexão
var cn = new SqlConnection(conexao);

//Obtém uma lista de clientes
var clientes = cn.Query<Customer>(sql);

//Percorre a lista de clientes
foreach (Customer cliente in clientes)
{
    //Adiciona o nome do cliente na lista
    DropDownList1.Items.Add(cliente.CompanyName);
}

//Fecha a conexão
cn.Close();
```

Observando o exemplo anterior, vê-se que o Dapper cria métodos de extensão para a classe **DbConnection**. O método query permite passar uma classe (no caso, **Customers**) que é preenchida automaticamente com o resultado da expressão SQL.

```
var clientes = cn.Query<Customer>(sql);
```

A classe **Customers** tem a mesma estrutura da expressão SQL:

```
public class Customer
{
    public string CustomerId{ get; set; }
    public string CompanyName { get; set; }
}

//Expressão SQL a ser executada
string sql="SELECT CustomerId, CompanyName FROM Customers";
```

1.3.4. NoSQL Database – Not Only SQL Database

Os bancos de dados relacionais, como SQL Server, Oracle ou MySQL, nem sempre representam a melhor solução para uma aplicação. A principal característica do modelo relacional é que as estruturas de dados são fixas. Uma tabela de produtos, por exemplo, poderia ter os campos **Produtoid**, **Nome** e **Preco**. Mas o que aconteceria se, em um determinado produto, e somente neste, fosse necessário incluir o campo **DataDeValidade**? A única opção seria alterar a estrutura para todos os produtos, mesmo que a maioria não tivesse a necessidade. Outra opção seria criar outra tabela e relacionar um produto por meio de combinação de chave primária e chave estrangeira. Em ambos os casos, o trabalho seria considerável, porque não há como ter uma estrutura variável em bancos de dados relacionais.

Os bancos de dados baseados em documentos ou objetos são uma alternativa interessante para esses casos.

Os bancos de dados **MongoDB**, **Cassandra** e **db4o** são baseados em documentos e objetos.

A seguir, um exemplo de inserção de dados usando **MongoDb**:

```
Db.Uuarios.Insert( { nome:"Maria", idade:32 } )
Db.Uuarios.Insert( { nome:"Jose", email: "jose@teste.com", idade:32 } )
```

1.3.5. WCF Data Services

É uma tecnologia que permite expor dados por meio da Internet, usando um protocolo chamado **OData (Open Data)**. Esse protocolo é uma tentativa de padronizar a forma de compartilhar dados usando padrões conhecidos da Web. De forma resumida, é um modo de disponibilizar dados usando URLs e comandos HTTP para exibir, criar, excluir e alterar dados.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

O exemplo a seguir mostra como solicitar uma informação de um servidor por meio do protocolo OData, usando um serviço chamado **servico.svc** para retornar informações de um conjunto de dados chamado **clientes** e cuja cidade seja igual a **Rio de Janeiro**:

```
http://servidor.com.br/servico.svc/clientes?\$filter=cidade eq 'Rio de Janeiro'
```

A resposta pode vir em formato JSON:

```
{ "Nome": "José",
  "Email": "jose@teste.com",
  "cidade": "Rio de Janeiro"
},
{
  "Nome": "Maria",
  "Email": "maria@teste.com",
  "cidade": "Rio de Janeiro"
}
```

Resumindo, os principais modos de acesso a dados ASP.NET (e na plataforma .NET em geral) são os seguintes:

- **ADO.NET**: Classes para acesso direto;
- **ORM**: Programas que transferem e sincronizam dados do banco para objetos, sendo o mais conhecido o **Entity Framework**;
- **Micro-ORM**: Programas simples que transferem dados do banco para objetos, com recursos básicos. **Dapper** é um conhecido micro-ORM;
- **NoSQL (Not Only SQL)**: Bancos de dados orientado a objetos, sem estruturas de dados formalmente definidas. **Cassandra** é um conhecido gerenciador de dados NoSQL;
- **WCF Data Services**: Serviços que retornam e transmitem dados usando o protocolo OData, que é um padrão que usa as tecnologias Web como HTTP e JSON.

O modelo utilizado para acesso a dados em uma aplicação deve ser escolhido considerando uma série de fatores, entre eles: necessidades inerentes ao sistema, conhecimento e familiaridade da equipe de desenvolvimento, recursos de software e hardware disponíveis e expectativa de ampliação do sistema em curto e longo prazo. Por exemplo, um sistema baseado em diversos aplicativos que trocam informações é um bom candidato a usar Serviços de Dados (WCF Data Services), enquanto um sistema com dados centralizados em que a performance é o fator mais importante se beneficiaria em utilizar o acesso direto usando ADO.NET.

Nada impede que a estratégia de dados seja escolhida em um momento futuro do desenvolvimento e não no escopo inicial. Usando um desenvolvimento baseado em camadas (layer), é possível isolar a parte do código que trata do acesso a dados. Dois modelos de arquitetura bastante utilizados são o de três camadas (3-tier) e o baseado em domínio (DDD - Domain-Driven Design). Esses modelos serão vistos futuramente. Neste módulo, o foco será o acesso a dados e a interação do usuário com essas informações.

1.4. Bancos de dados de exemplo

A Microsoft utiliza diversos bancos de dados para exemplos de programação, ferramentas ou conceitos. É importante conhecer a estrutura desses bancos porque inúmeros livros, artigos e sites os utilizam quando é necessário exibir exemplos de acesso e manipulação de dados. Os mais utilizados são **Northwind**, **Pubs** e **AdventureWorks**.

1.4.1. Northwind

O banco de dados **Northwind** é um dos mais antigos. Era distribuído originalmente com o banco de dados Access. Posteriormente, passou a ser distribuído com o SQL Server. Este banco pode ser obtido no site da Microsoft:

Visual Studio 2015 - ASP.NET com C# Acesso a dados

<http://www.microsoft.com/en-us/download/details.aspx?id=23654>

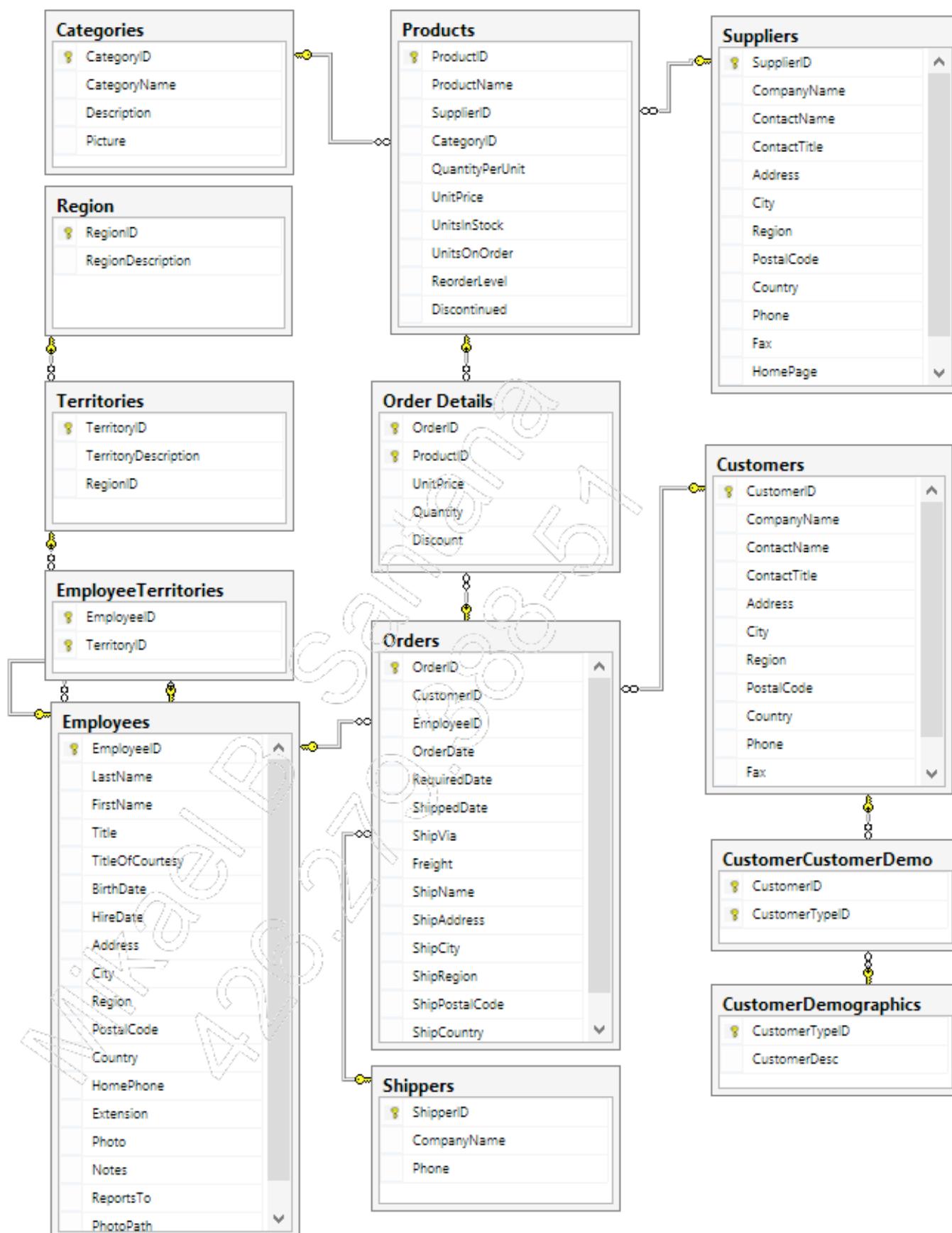


Neste banco de dados, está representada a **Northwind Traders**, uma empresa fictícia que importa e exporta alimentos do mundo todo.

As tabelas de dados representam os dados dessa empresa, como **Clientes, Produtos, Funcionários, Pedidos** etc. As seguintes tabelas estão definidas:

Tabela	Descrição
Categories	Categorias de produtos.
Suppliers	Fornecedores de produtos.
Products	Produtos.
Customers	Clientes.
Employees	Funcionários.
Shippers	Transportadoras.
Orders	Pedidos.
Order Details	Detalhes do pedido.
Region	Região geográfica.
Territories	Territórios (cidades).
EmployeeTerritories	Funcionários e áreas onde atuam.
CustomerDemographics	Área geográfica de clientes.
CustomerCustomerDemo	Relação de cliente com área geográfica.

A seguir, um diagrama com os relacionamentos. O ponto central é a tabela de pedidos (**Orders**) e detalhes do pedido (**Order Details**).



Visual Studio 2015 - ASP.NET com C# Acesso a dados

A seguir, veja alguns exemplos de como extrair informações do banco de dados **Northwind**:

- Lista de clientes:

```
SELECT * FROM CUSTOMERS
```

- Lista de produtos:

```
SELECT * FROM PRODUCTS
```

- Lista de fornecedores:

```
SELECT * FROM SUPPLIERS
```

- Lista de categorias:

```
SELECT * FROM CATEGORIES
```

- Lista de produtos com ID, nome, categoria e fornecedor:

```
SELECT P.ProductID, P.ProductName, C.CategoryName, S.CompanyName  
FROM PRODUCTS P  
INNER JOIN Categories C ON P.CategoryID=C.CategoryID  
INNER JOIN Suppliers S ON P.SupplierID=S.SupplierID
```

- Lista de clientes do Brasil com nome e cidade:

```
SELECT CustomerID, CompanyName, City  
FROM Customers  
WHERE Country='Brazil'
```

- Lista de pedidos:

```
SELECT * FROM ORDERS
```

- Lista de pedidos com o nome do cliente, do vendedor e da transportadora:

```
SELECT O.OrderID NumeroPedido,  
       O.OrderDate Data,  
       C.CompanyName Cliente,  
       e.FirstName + ' ' + e.LastName Vendedor,  
       s.CompanyName Transportadora  
  FROM ORDERS O  
 INNER JOIN Customers C ON O.CustomerID=C.CustomerID  
 INNER JOIN Employees E ON O.EmployeeID=E.EmployeeID  
 INNER JOIN Shippers S ON O.ShipVia=S.ShipperID
```

- Lista de pedidos com detalhes, data e nome dos clientes:

```
SELECT D.OrderID, O.OrderDate, C.CompanyName,  
       P.ProductName, D.Quantity, D.UnitPrice  
  FROM [Order Details] D  
 INNER JOIN Orders O ON o.OrderID=d.OrderID  
 INNER JOIN Customers C on c.CustomerID=o.CustomerID  
 INNER JOIN Products P ON P.ProductID=d.ProductID
```

- Total de vendas por produto:

```
SELECT P.ProductName, SUM(d.Quantity * d.UnitPrice) Total  
  FROM [Order Details] D  
 INNER JOIN Products P ON P.ProductID=d.ProductID  
 GROUP BY P.ProductID, P.ProductName  
 ORDER BY 2 desc
```

- Lista de produtos de uma categoria:

```
SELECT P.ProductName, P.UnitPrice  
  FROM Products P  
 WHERE P.CategoryID=1
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

- Total de clientes por país:

```
SELECT Country, COUNT(*)  
FROM Customers  
GROUP BY Country  
ORDER BY 2 DESC
```

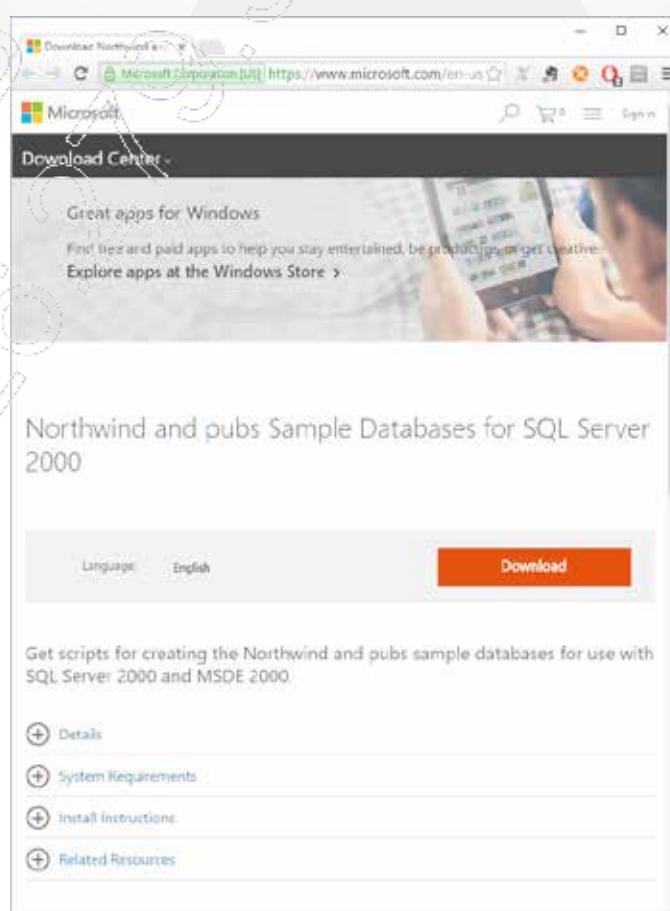
- Total de produtos por categoria:

```
SELECT Country, COUNT(*)  
FROM Customers  
GROUP BY Country  
ORDER BY 2 DESC
```

1.4.2. Pubs

O banco de dados **Pubs** é distribuído desde as primeiras versões do SQL Server e do Visual Studio. É um banco de dados com informações de livros, editoras e autores e pode ser baixado no mesmo pacote do **Northwind**:

<http://www.microsoft.com/en-us/download/details.aspx?id=23654>

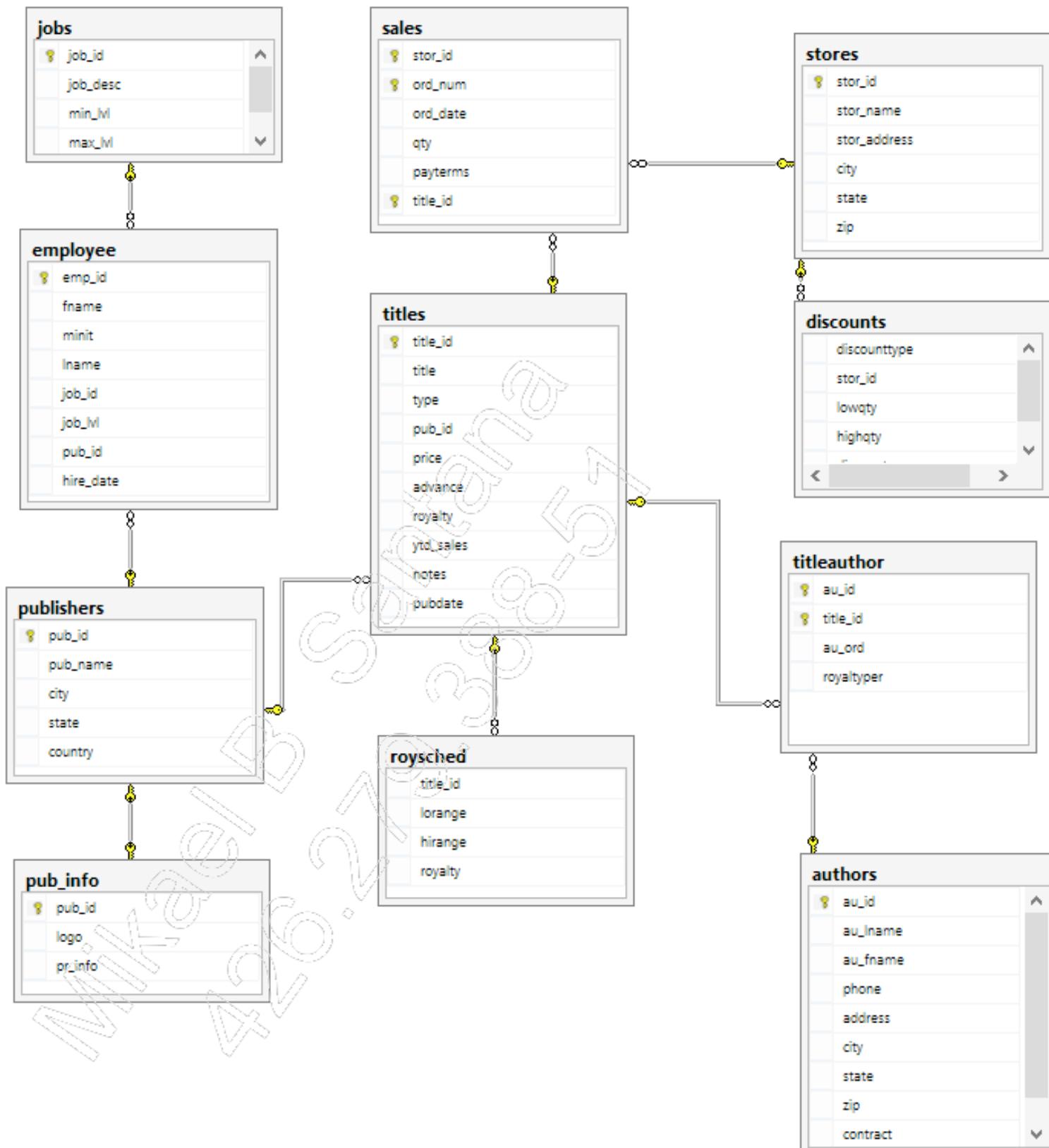


Neste banco de dados, estão definidas as seguintes tabelas:

Tabela	Descrição
Authors	Autores dos livros
Titles	Livros
TitleAuthor	Autores de um livro
Rroysched	Royalties
Publishers	Editoras de livros
Employee	Funcionários das editoras
Jobs	Cargos dos funcionários
Sales	Vendas
Stores	Lojas onde são vendidos os livros
Discounts	Descontos concedidos por quantidade
Pub_Info	Logotipo e informações das editoras

Visual Studio 2015 - ASP.NET com C# Acesso a dados

O relacionamento entre as tabelas está representado no diagrama a seguir. O ponto central é a tabela de livros (**Titles**).



A seguir, veja alguns exemplos de como extrair informações do banco de dados **Pubs**:

- Lista de livros:

```
SELECT * FROM Titles
```

- Lista de autores:

```
SELECT * FROM authors
```

- Lista de editoras:

```
SELECT * FROM publishers
```

- Lista de livrarias:

```
SELECT * FROM stores
```

- Lista de livros, autores e porcentagem de royalties:

```
select t.title Livro,
       a.au_fname + a.au_lname Autor,
       ta.royaltyper porcentagemRoyalties
  from titleauthor ta
    inner join authors a on a.au_id=ta.au_id
    inner join titles t on t.title_id=ta.title_id
  order by t.title, ta.au_ord
```

- Lista de vendas:

```
select * from sales
```

- Livros mais vendidos:

```
select t.title, sum(s.qty) Total
  from sales s
    inner join titles t on s.title_id=t.title_id
  group by t.title
  order by 2 desc
```

1.4.3. AdventureWorks

Este é o novo banco de dados da Microsoft. Contém uma estrutura mais complexa que as do **Northwind** e do **Pubs** e é amplamente utilizado para demonstrar diversos tipos de arquitetura de sistemas. Pode ser baixado no site **CodePlex**:

<http://msftdbprodsamples.codeplex.com/releases/view/55330>



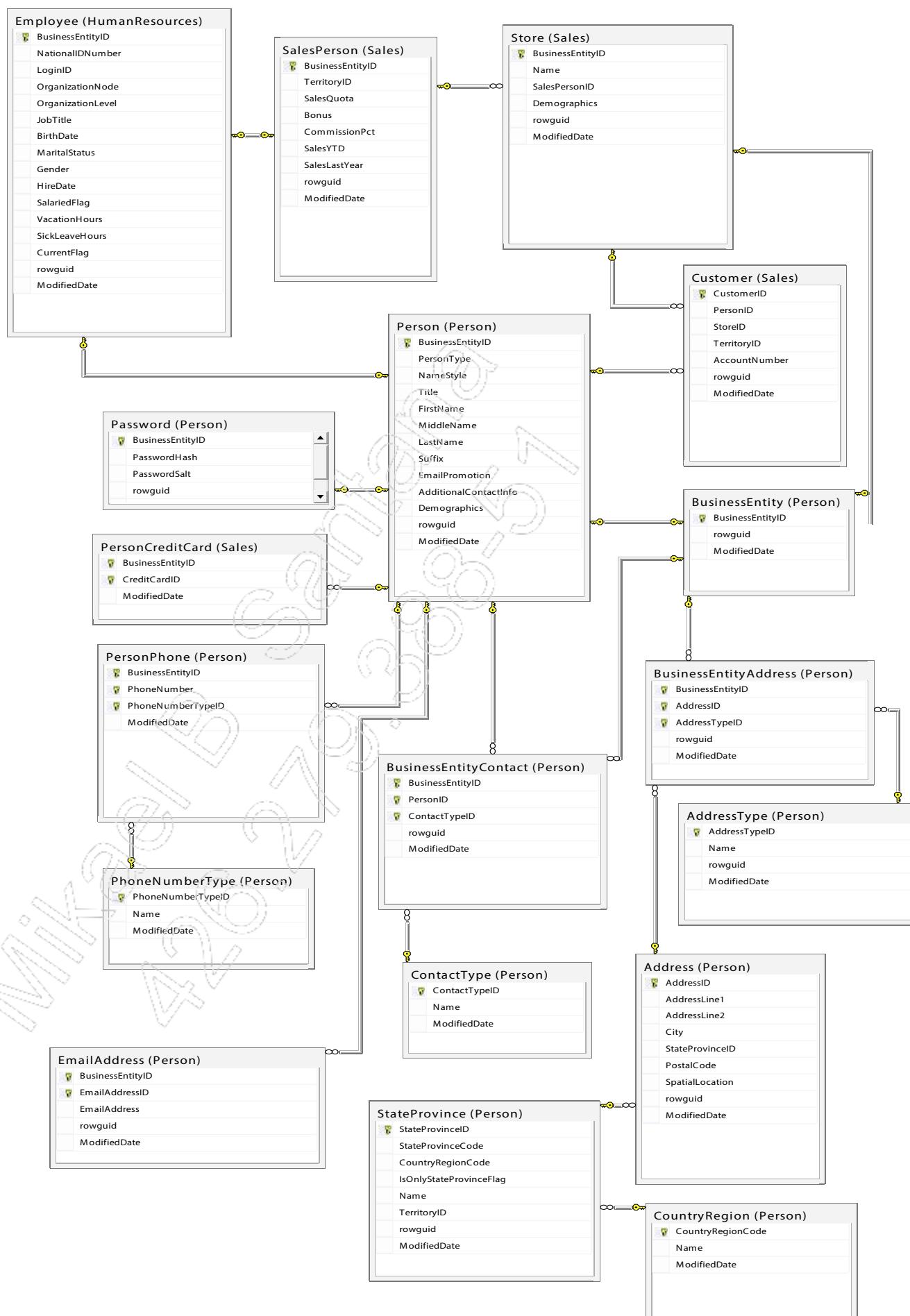
Neste banco de dados, consta a **AdventureWorks**, uma empresa fictícia que vende bicicletas no mundo todo. O banco de dados inclui dados dos clientes, produtos, fornecedores, funcionários e vendas. Um detalhe interessante é que todos os cadastros são centralizados na tabela **Person** (pessoas). Qualquer entidade (funcionário, empresa) pode ser um cliente. Ao todo são 72 tabelas.

A listagem a seguir exibe algumas das principais tabelas:

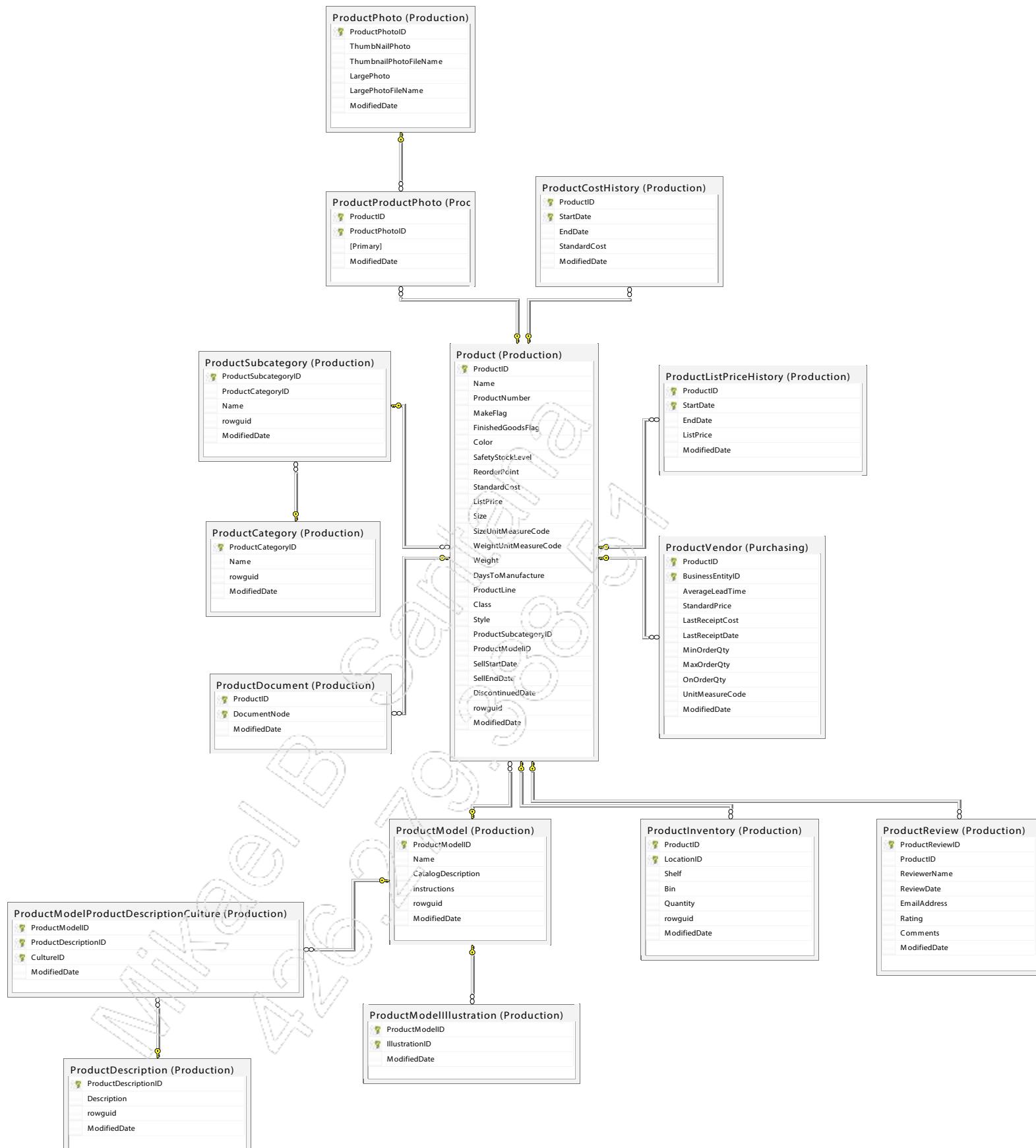
Tabela	Descrição
Person	Cadastro de pessoas
Address	Endereços
AddressType	Tipos de endereço
PersonPhone	Telefones de pessoas
EmailAddress	E-mails de pessoas
PersonPhoneNumberType	Tipos de telefones
BusinessEntity	Empresas
BusinessEntityAddress	Endereços de empresas
Employee	Funcionários
Department	Departamentos
Product	Produtos
Category	Categorias de produtos
ProductCategory	Categorias de um produto
ProductPhoto	Fotos de um produto
Customers	Clientes
Sales	Vendas
SalesOrderDetail	Detalhes da venda
SalesOrderHeader	Cabeçalho do pedido de venda
Store	Lojas

Visual Studio 2015 - ASP.NET com C# Acesso a dados

O esquema a seguir representa a parte de cadastro de pessoas (**Person**), que é vinculada a clientes (**Customer**), funcionários (**Employee**) e empresas (**BusinessEntity**):

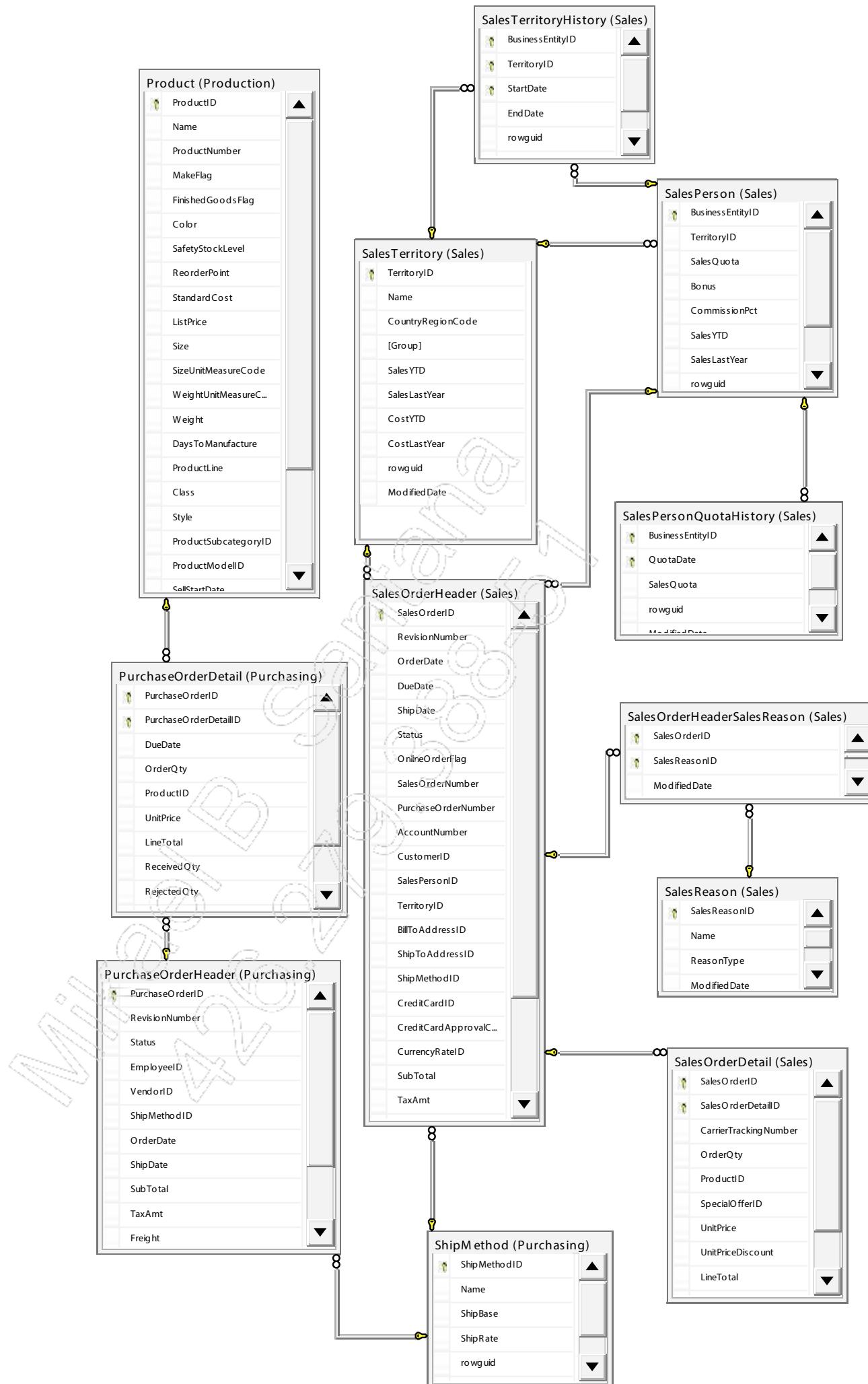


Já o esquema a seguir representa a parte de cadastro de produtos (**Product**):



Visual Studio 2015 - ASP.NET com C# Acesso a dados

O esquema adiante representa a parte de vendas (Sales):



A seguir, alguns exemplos de como extrair informações do banco de dados **AdventureWorks**:

- Lista de clientes:

```
SELECT P.FirstName, P.LastName, F.PhoneNumber, N.Name, P.PersonType  
FROM Sales.Customer C  
INNER JOIN Person.Person P on P.BusinessEntityID=C.PersonID  
INNER JOIN Person.PersonPhone F ON F.BusinessEntityID=P.BusinessEntityID  
INNER JOIN Person.PhoneNumberType N ON N.PhoneNumberTypeID=F.  
PhoneNumberTypeID  
Order by p.FirstName  
-- SC = Store Contact, IN = Individual (retail) customer, SP = Sales person,  
EM = Employee (non-sales), VC = Vendor contact, GC = General contact
```

- Lista de funcionários:

```
SELECT P.FirstName, P.LastName, F.PhoneNumber, N.Name, E.JobTitle,  
P.PersonType  
FROM HumanResources.Employee E  
INNER JOIN Person.Person P on P.BusinessEntityID=E.BusinessEntityID  
INNER JOIN Person.PersonPhone F ON F.BusinessEntityID=P.BusinessEntityID  
INNER JOIN Person.PhoneNumberType N ON N.PhoneNumberTypeID=F.  
PhoneNumberTypeID  
Order by p.FirstName
```

- Categorias de produtos:

```
Select ProductCategoryID, Name from Production.ProductCategory
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

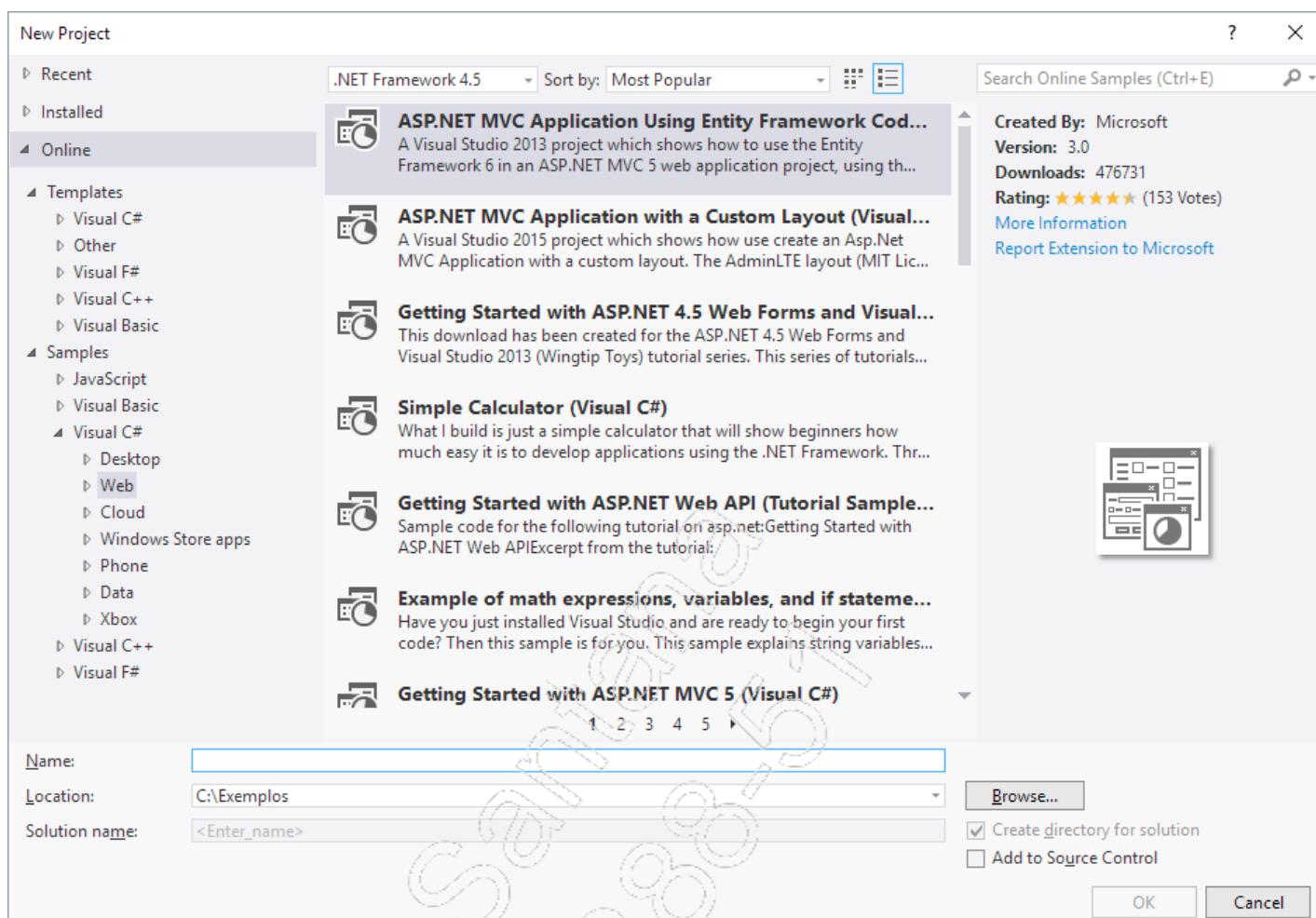
- Produtos, subcategoria, categoria e modelos:

```
Select C.Name Categoria,  
      S.Name SubCategoria,  
      M.Name Modelo,  
      P.ProductID ProdutoId,  
      P.Name Produto,  
      P.ListPrice Preco  
  
from Production.Product P  
  
inner join Production.ProductSubcategory S on  
      S.ProductSubcategoryID=P.ProductSubcategoryID  
  
inner join Production.ProductModel M on  
      P.ProductModelID=M.ProductModelID  
  
inner join Production.ProductCategory C on  
      S.ProductCategoryID=C.ProductCategoryID  
  
Order By Categoria, SubCategoria, Produto
```

É importante conhecer a estrutura desses bancos. Muitos livros e artigos os utilizam como exemplo. Há, também, o fato de terem sido criados pelas mesmas equipes que criaram as ferramentas de acesso a dados e o .NET Framework. Os bancos de dados mais antigos (**Northwind**), apesar de não representarem as necessidades atuais de um sistema de médio/grande porte, são simples o suficiente para servirem de ponto de partida e facilitam o entendimento. Os bancos de dados mais modernos (**AdventureWorks**) representam de forma mais consistente as necessidades de um aplicativo moderno com o mínimo de complexidade exigida para essa finalidade.

Acesso a dados com ASP.NET

Esses são apenas alguns bancos de dados utilizados. No Visual Studio, no menu **Help / Samples**, é possível obter os mais diversos exemplos para estudar.



Pontos principais

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

- A plataforma .NET oferece diversas maneiras de acesso a dados, das quais podemos destacar: **ADO.NET**, **ORM** (Entity Framework, NHibernate), **Micro-ORM** (Dapper, Massive), **NoSQL** (db4o, RavenDb) e **Data Services**;
- **Banco de dados** é um termo usado para descrever um conjunto de informações agrupadas de maneira coerente e útil;
- **Sistema Gerenciador de Banco de Dados (SGBD)** é um programa que permite criar e manipular banco de dados;
- **Base de dados** é um conjunto de informações disponíveis e que pode ser usada para criar bancos de dados;
- A Microsoft oferece diversos bancos de dados de exemplo que são usados para demonstrar tecnologias. Os mais conhecidos são: **Northwind**, **Pubs** e **AdventureWorks**;
- **Northwind** é um banco de dados que representa um empresa fictícia chamada **Northwind Traders** que vende alimentos pelo mundo todo;
- **Pubs** é um banco de dados que exibe informações sobre livros, autores, editoras e vendas;
- **AdventureWorks** é um banco de dados que representa uma empresa fictícia que vende bicicletas e apresenta uma estrutura de dados bem detalhada.

1

Acesso a dados com ASP.NET

Teste seus conhecimentos

Mikael Brantana
426.277-0007



IMPACTA
EDITORA

1. Qual Sistema Gerenciador de Banco de Dados listado a seguir não é um banco de dados relacional?

- a) SQL Server
- b) RavenDb
- c) Oracle
- d) MySQL
- e) PostgreSQL

2. Qual modo de acesso a dados é mais eficiente, do ponto de vista da performance?

- a) Entity Framework
- b) Data Services
- c) ADO.NET
- d) JavaScript
- e) ORMs

3. Como são chamados os Sistemas Gerenciadores de Banco de Dados que, ao contrário dos Bancos de Dados Relacionais, não utilizam o conceito de tabelas e relacionamentos?

- a) TableLess
- b) NoSQL
- c) NoTable
- d) Server
- e) ADO.NET

4. Qual banco de dados é fornecido pela Microsoft para ser usado como exemplo e apresenta uma estrutura de tabelas bastante detalhada?

- a) Access
- b) Northwind
- c) Pubs
- d) AdventureWorks
- e) Nenhuma das alternativas anteriores está correta.

5. Qual é a melhor definição de banco de dados?

- a) É um aplicativo que permite criar e manipular informações.
- b) É um conjunto de informações armazenadas em arquivos.
- c) É um modelo de programação que pode ser usado em ASP.NET.
- d) É um conjunto de informações organizadas.
- e) É um conjunto de informações divididas em tabelas.

1

Acesso a dados com ASP.NET

Mãos à obra!

Mikael
A26.27



IMPACTA
EDITORA

Laboratório 1

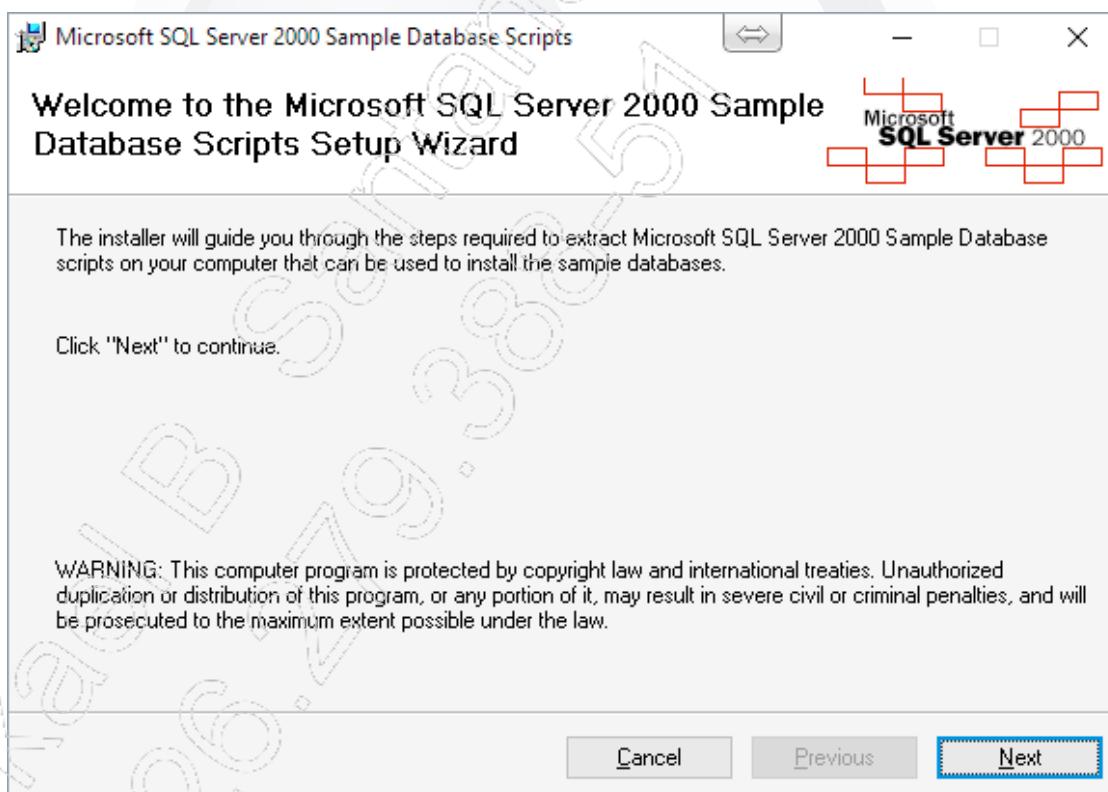
A - Instalando o Northwind

Neste laboratório, realizaremos os seguintes passos: instalar os bancos de dados exemplo da Microsoft; criar conexões no Visual Studio para esses bancos; analisar as estruturas das tabelas; e extrair informações básicas desses bancos de dados.

1. Baixe o **Northwind** pelo seguinte endereço:

<http://www.microsoft.com/en-us/download/details.aspx?id=23654>

2. Execute o programa **SQL2000SampleDb.msi**:



3. Verifique se foi criada a pasta **C:\SQL Server 2000 Sample Databases**:

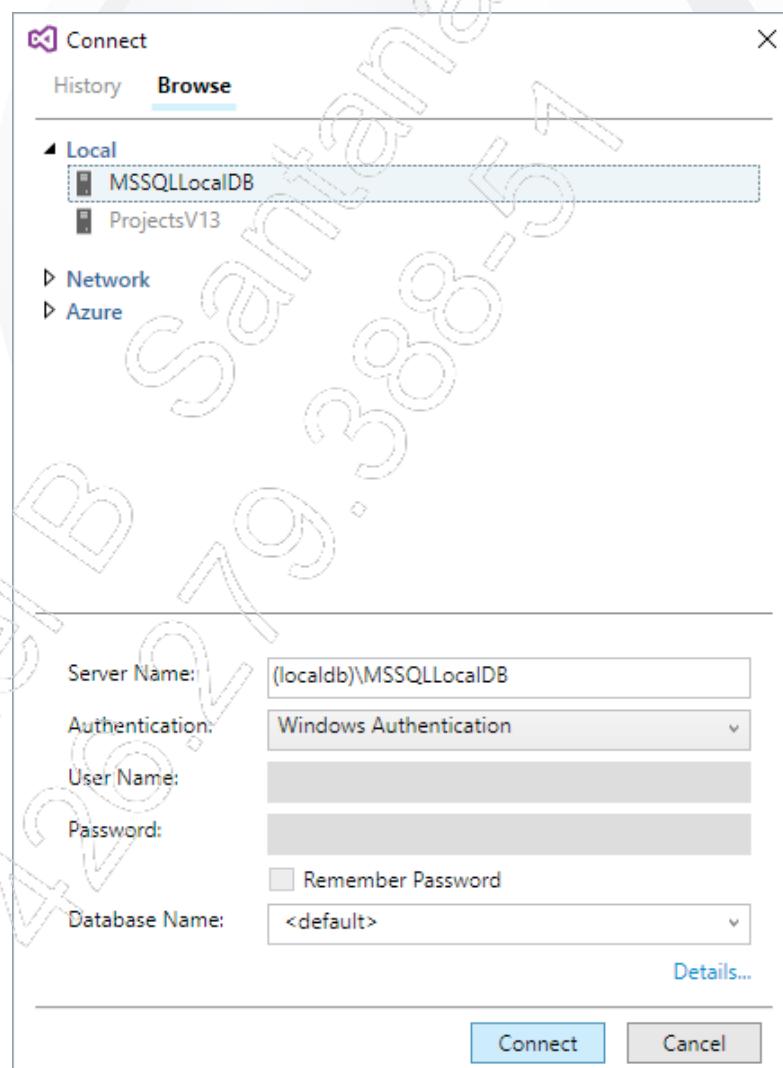
Este Computador > Windows8_OS (C:) > SQL Server 2000 Sample Databases			
Nome	Data de modificaç...	Tipo	Tamanho
instnwnd.sql	04/02/04 09:45	SQL Text File	2.066 KB
instpubs.sql	23/03/04 10:50	SQL Text File	126 KB
NORTHWND.LDF	13/12/04 17:14	Arquivo LDF	1.024 KB
NORTHWND.MDF	13/12/04 17:14	Arquivo MDF	2.688 KB
PUBS.MDF	13/12/04 17:14	Arquivo MDF	1.280 KB
PUBS_LOG.LDF	13/12/04 17:14	Arquivo LDF	768 KB
ReadMe_SQL2000SampleDbScripts.htm	23/03/04 10:50	Chrome HTML Do...	61 KB

4. Usando o menu **File / Open File**, abra o arquivo **instnwnd.sql**, que está na pasta **C:\SQL Server 2000 Sample Databases**;

5. Comente (--) as seguintes linhas (dependendo da versão do SQL Server, pode ocorrer um problema de compatibilidade):

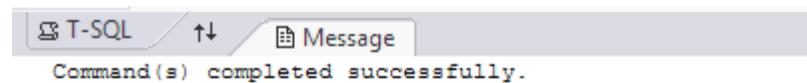
```
--exec sp_dboption 'Northwind','trunc. log on chkpt.','true'  
--exec sp_dboption 'Northwind','select into/bulkcopy','true'
```

6. Clique no botão **Execute** (▶). Uma tela de conexão deve aparecer (na primeira vez). Escolha **Local** e selecione a instância **MSSQLLocalDB**. Dependendo da instalação do Visual Studio e do SQL Server, o nome pode ser outro, como **localhost\sqlexpress**. Clique em **Connect** para conectar e executar o script;

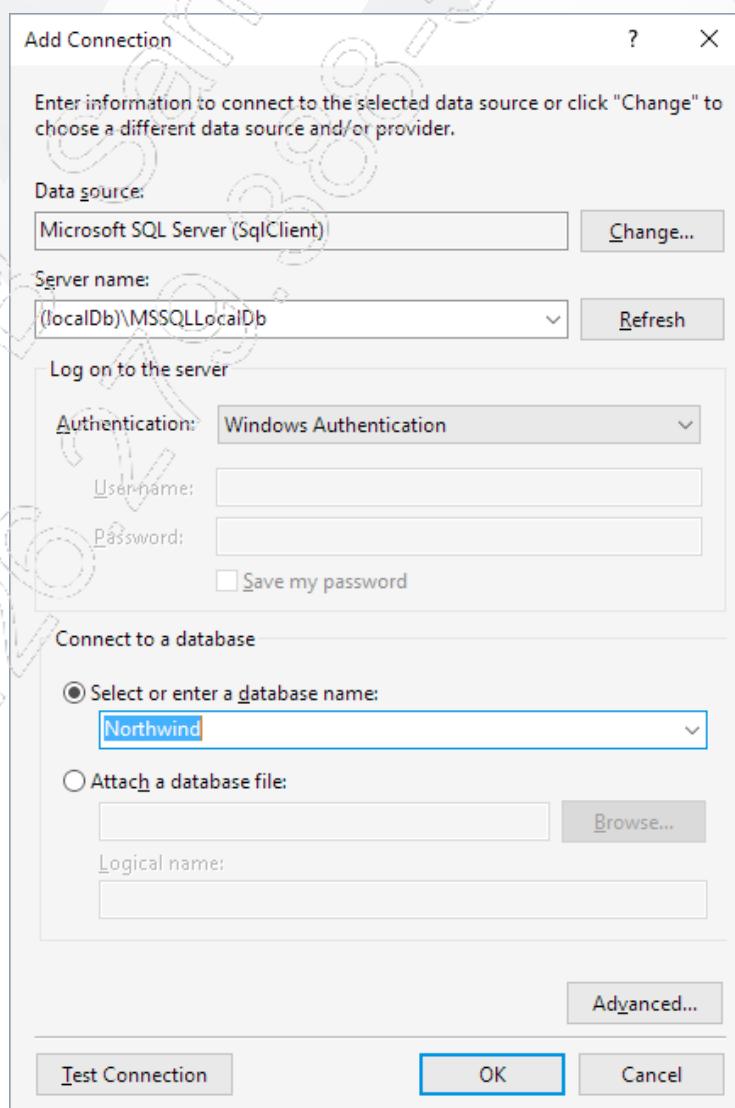


Visual Studio 2015 - ASP.NET com C# Acesso a dados

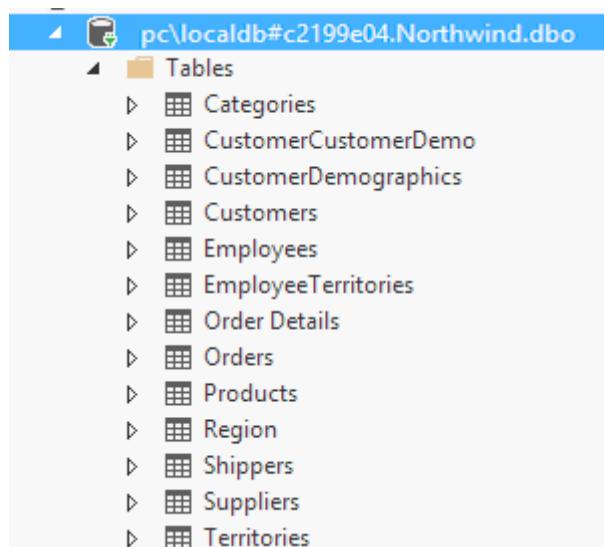
7. Espere o resultado da execução do script. Deve ser retornada a mensagem **Comando executado com sucesso:**



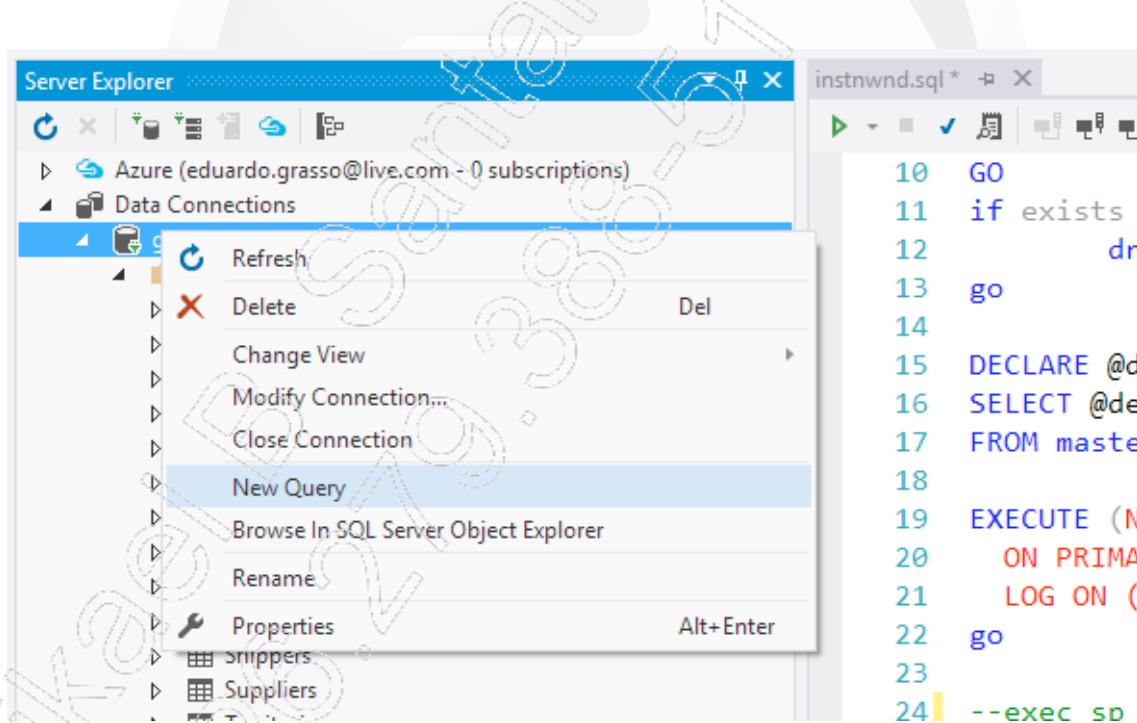
8. Abra a janela **Server Explorer**, escolha o menu de contexto **Add New Connection**, depois **SqlServer** (se aparecer a janela de escolha de provider) e preencha o nome do servidor (**(localDb)\MSSQLLocalDB**) e o escolha o nome do banco de dados (**Northwind**);



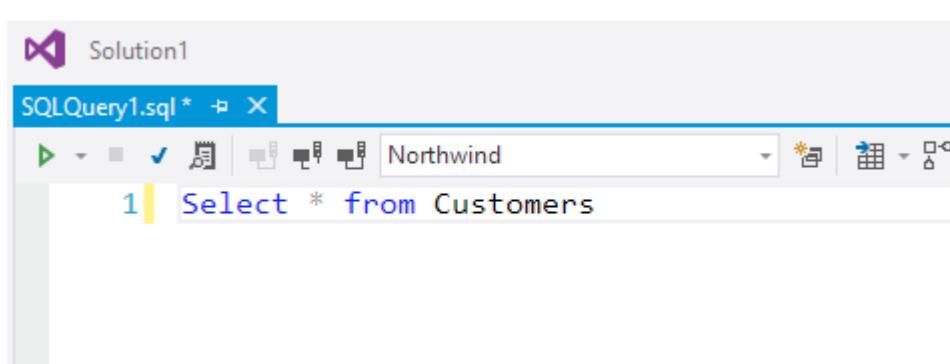
9. Expanda a árvore de objetos até ver as tabelas:



10. Teste a conexão com o banco executando uma query. Escolha, no menu de contexto do banco de dados **Northwind**, a opção **New Query**:



11. Digite uma expressão SQL e clique no botão **Execute**:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

12. Veja o resultado:

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region
1 ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin	NULL
2 ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.	NULL
3 ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	NULL
4 AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London	NULL
5 BERGS	Berglunds snabbköp	Christina Bergl...	Order Administrator	Berguvsvägen 8	Luleå	NULL
6 BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim	NULL
7 BLONP	Blondesddsl père et fils	Frédérique Cit...	Marketing Manager	24, place Kléber	Strasbourg	NULL
8 BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	C/ Araquil, 67	Madrid	NULL
9 BONAP	Bon app'	Laurence Leb...	Owner	12, rue des Bouchers	Marseille	NULL
10 BOTTM	Bottom-Dollar Markets	Elizabeth Linc...	Accounting Manager	23 Tswassen Blvd.	Tswassen	BC
11 BSBEV	B's Beverages	Victoria Ashw...	Sales Representative	Fauntleroy Circus	London	NULL
12 CACTU	Cactus Comidas para llevar	Patricia Simps...	Sales Agent	Cerito 333	Buenos Ai...	NULL
13 CENTC	Centro comercial Moctezuma	Francisco Ch...	Marketing Manager	Sieras de Granada 9993	México D.F.	NULL
14 CHOPS	Chop-suey Chinese	Yang Wang	Owner	Hauptstr. 29	Bern	NULL
15 COMMI	Com'r Meierle	Rainer Richter	Sales Associate	Alt-Jedlestrasse 20	Stuttgart	SP

B – Instalando o Pubs

1. Usando o menu **File / Open File**, abra o arquivo **instpubs.sql**, que está na pasta **C:\SQL Server 2000 Sample Databases**;

2. Comente (--) a seguinte linha (dependendo da versão do SQL Server, pode ocorrer um problema de compatibilidade):

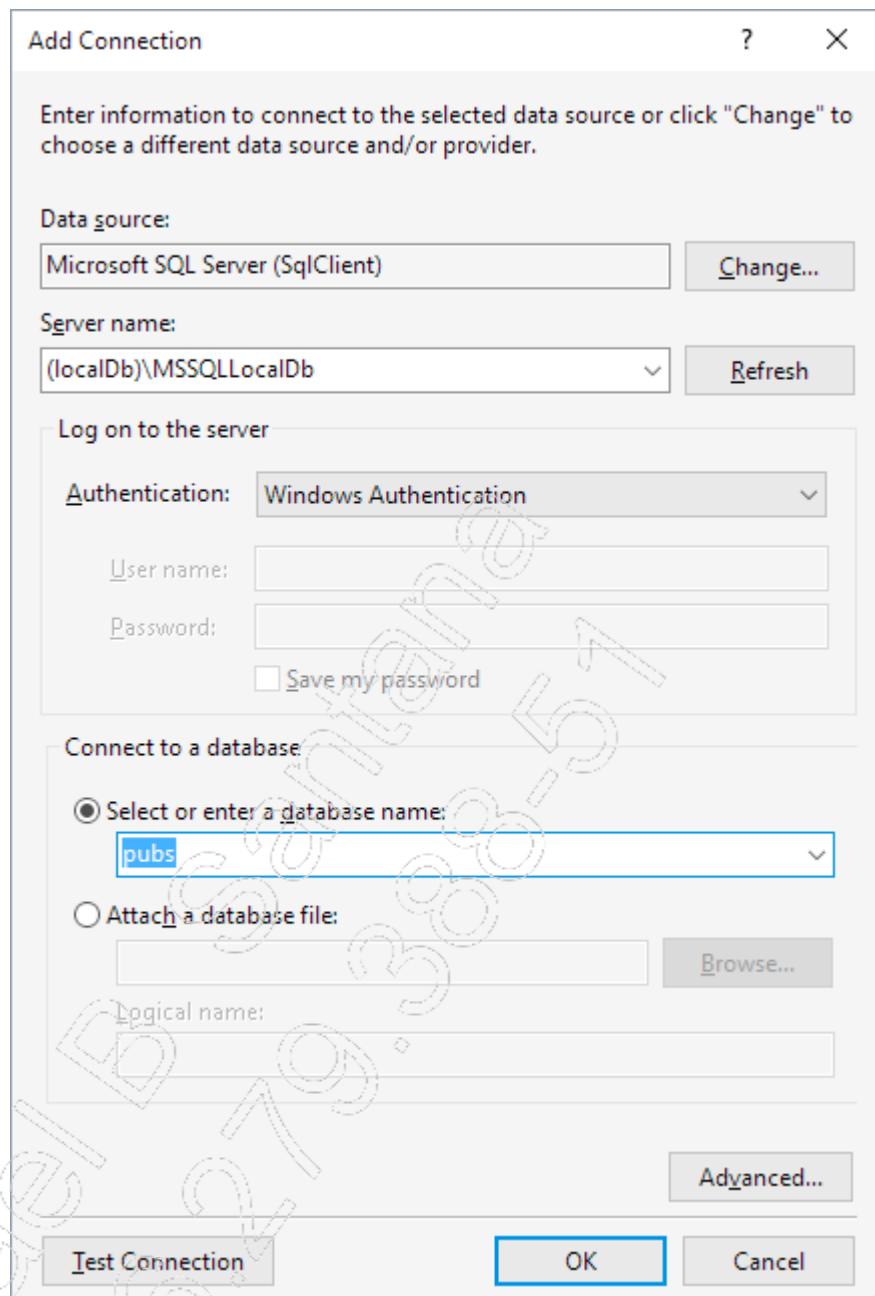
```
-- -- execute sp_dboption 'pubs' , 'trunc. log on chkpt.' , 'true'
```

3. Execute o script e espere o resultado:

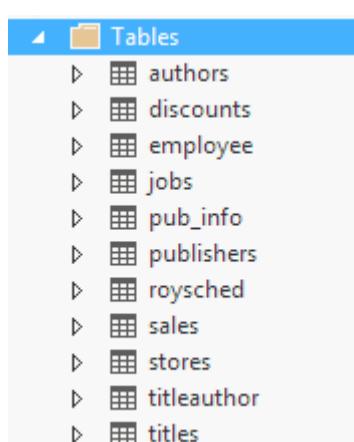
Command(s) completed successfully.

Query executed successfully at 11... | (localdb)\MSSQLLocalDB (13.... |

4. Abra a janela **Server Explorer**, escolha o menu de contexto **Add New Connection**, depois **SqlServer** (se aparecer a janela de escolha de provider) e preencha o nome do servidor (**(localDb)\MSSQLLocalDB**) e o escolha o nome do banco de dados (**pubs**);

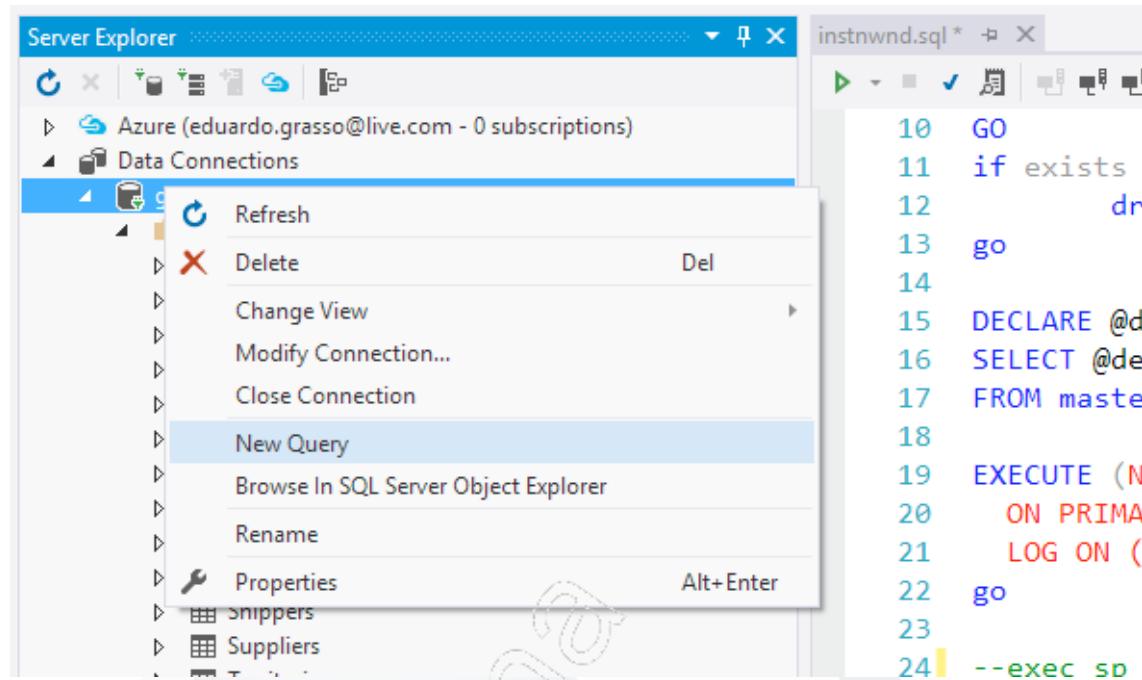


5. Expanda a árvore de objetos até ver as tabelas:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

6. Teste a conexão com o banco executando uma query. Escolha, no menu de contexto do banco de dados **Pubs**, a opção **New Query**:



7. Digite uma expressão SQL e clique no botão **Execute**. Veja o resultado:

The screenshot shows the SSMS interface with a query window titled 'SQLQuery2.sql'. The query 'Select * from authors' is entered. Below the results pane, the status bar indicates 'Connection Ready' and '(LocalDB)\MSSQLLocalDB'.

au_id	au_lname	au_fname	phone	address	city	state	zip	contract
1	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	94025	1
2	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	94618	1
3	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	94705	1
4	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	95128	1
5	Straight	Dean	415 834-2919	5420 College Av.	Oakland	CA	94609	1
6	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	66044	0

C – Instalando o AdventureWorks

1. Baixe o AdventureWorks pelo seguinte endereço:

<http://msftdbprodsamples.codeplex.com/releases/view/55330>

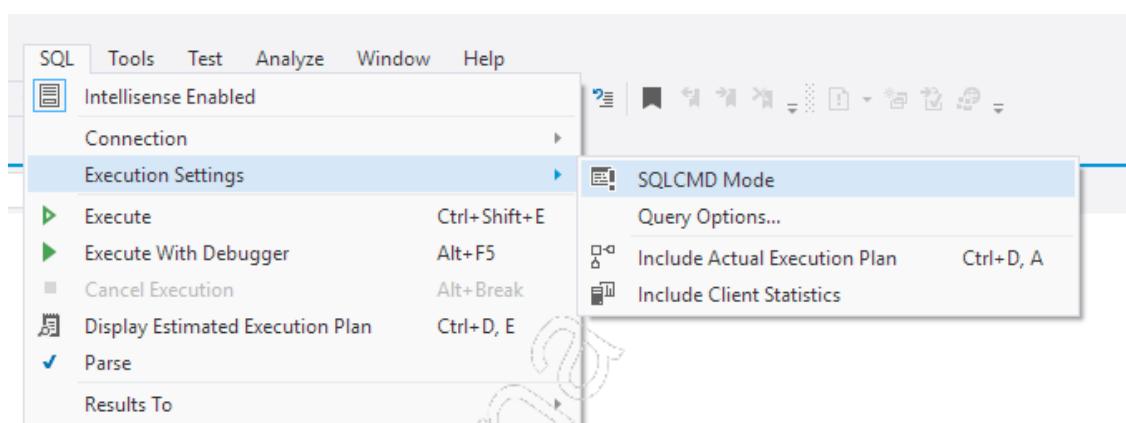
2. Escolha a opção AdventureWorks 2012 OLTP Script;

3. Descompacte o arquivo na pasta C:\AdventureWorksSource. Esse arquivo contém uma pasta chamada AdventureWorks 2012 OLTP Script com arquivos CSV e um arquivo de script chamado instawdb.sql. Os arquivos CSV contêm os dados que serão importados. O arquivo de script cria o banco de dados, tabelas, store procedures e importa os dados contidos nos arquivos CSV;

Nome	Data de modificação	Tipo	Tamanho
Department.csv	13/10/09 22:02	Arquivo de Valore...	2 KB
Document.csv	13/10/09 22:02	Arquivo de Valore...	1.131 KB
EmailAddress.csv	13/10/09 22:02	Arquivo de Valore...	4.195 KB
Employee.csv	13/10/09 22:02	Arquivo de Valore...	96 KB
EmployeeDepartmentHistory.csv	13/10/09 22:02	Arquivo de Valore...	14 KB
EmployeePayHistory.csv	13/10/09 22:02	Arquivo de Valore...	20 KB
Illustration.csv	13/10/09 22:02	Arquivo de Valore...	236 KB
instawdb.sql	14/01/13 21:03	Microsoft SQL Ser...	939 KB
JobCandidate.csv	13/10/09 22:02	Arquivo de Valore...	106 KB
Location.csv	13/10/09 22:02	Arquivo de Valore...	1 KB
Password.csv	13/10/09 22:02	Arquivo de Valore...	4.974 KB
Person.csv	13/10/09 22:02	Arquivo de Valore...	26.498 KB
PersonCreditCard.csv	13/10/09 22:02	Arquivo de Valore...	672 KB
PersonPhone.csv	13/10/09 22:02	Arquivo de Valore...	2.096 KB
PhoneNumberType.csv	13/10/09 22:02	Arquivo de Valore...	1 KB
Product.csv	13/10/09 22:02	Arquivo de Valore...	87 KB
ProductCategory.csv	13/10/09 22:02	Arquivo de Valore...	1 KB
ProductCostHistory.csv	13/10/09 22:02	Arquivo de Valore...	29 KB
ProductDescription.csv	13/10/09 22:02	Arquivo de Valore...	204 KB
ProductDocument.csv	13/10/09 22:02	Arquivo de Valore...	2 KB

4. Abra o arquivo instawdb.sql pelo Visual Studio;

5. Este arquivo deve ser executado no modo **SQLCMD**. Para ativar esse modo, escolha, no menu **SQL**, a opção **Execution Settings / SQLCMD Mode**. Esse modo de execução permite definir variáveis do ambiente e usar outros arquivos;



6. É necessário informar a pasta onde os arquivos de dados são gravados. No caso do **LocalDb**, a pasta fica dentro da pasta do usuário onde são armazenados dados de aplicativos. A linha seguinte deve ser alterada para apontar para a pasta correta (pasta padrão do SQL Server);

- Veja a linha original:

```
:setvar SqlSamplesDatabasePath    "C:\Program Files\Microsoft  
SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\"
```

- Essa linha deve ser alterada como segue:

```
:setvar SqlSamplesDatabasePath    " C:\Users\[nomeUsuario]\  
AppData\Local\Microsoft\Microsoft SQL Server Local DB\  
Instances\mssqllocaldb\"
```

7. Esta outra linha deverá ser alterada para indicar onde o arquivo foi descompactado:

```
:setvar SqlSamplesSourceDataPath "C:\Program Files\Microsoft  
SQL Server\MSSQL11.MSSQLSERVER\Tools\Samples\AdventureWorks  
2012 OLTP Script\"
```

O caminho deve apontar para a pasta correta:

```
:setvar SqlSamplesSourceDataPath "C:\AdventureWorksSource\Ad-  
ventureWorks 2012 OLTP Script\"
```

8. Se estiver usando o SQL Express, é bem provável que a opção **FullText Search** não esteja habilitada. Neste curso, não será necessário usar essa opção, por isso vamos adaptar o script para não usar **FullText** (por enquanto). Comente as linhas onde aparece **CREATE FULL TEXT**:

```
-- CREATE FULLTEXT INDEX ON Production.ProductReview(Comments)  
KEY INDEX PK_ProductReview_ProductReviewID;  
  
--CREATE FULLTEXT INDEX ON HumanResources.JobCandidate(Resume) KEY INDEX  
PK_JobCandidate_JobCandidateID;  
  
-- CREATE FULLTEXT INDEX ON Production.Document(Document TYPE COLUMN  
FileExtension, DocumentSummary) KEY INDEX PK_Document_DocumentNode;
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

9. Comente a parte do código que cria a stored procedure **uspSearchCandidateResumes**:

```
-- Começa aqui o comentário
/*
CREATE PROCEDURE [dbo].[uspSearchCandidateResumes]
    @searchString [nvarchar](1000),
    @useInflectional [bit]=0,
    @useThesaurus [bit]=0,
    @language[int]=0

WITH EXECUTE AS CALLER
AS
BEGIN
    SET NOCOUNT ON;

    ... (corpo da stored procedure omitido)
    ...
    ...

    ON  FT_TBL.[JobCandidateID] =KEY_TBL.[KEY]
END

END;
*/
-- termina aqui
GO
-- *****
-- Add Extended Properties
-- *****

PRINT '';
PRINT '** Creating Extended Properties';
GO

SET NOCOUNT ON;
GO
```

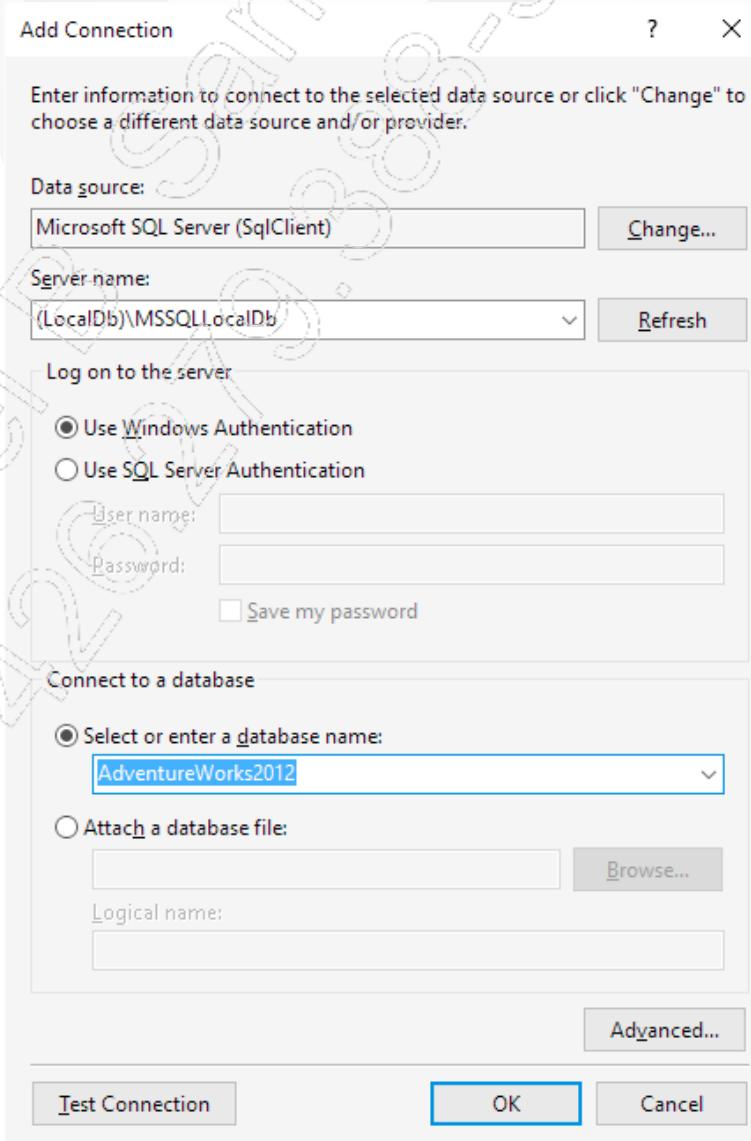
10. Clique no botão **Execute** e espere o script ser executado. Verifique se não houve nenhuma mensagem de erro. No final, é exibido um quadro demonstrativo do espaço ocupado pelo banco de dados;

	DbId	FileId	CurrentSize	MinimumSize	UsedPages	EstimatedPages
1	11	1	23336	21760	23328	23328
2	11	2	256	256	256	256

Results Message T-SQL

Query executed successfully. (localDB)\MSSQLLocalDb (12...) | PDPTI03\Grasso (52) | master | 00:00:37 | 2 rows

11. Conecte o banco de dados pela janela **Server Explorer**:

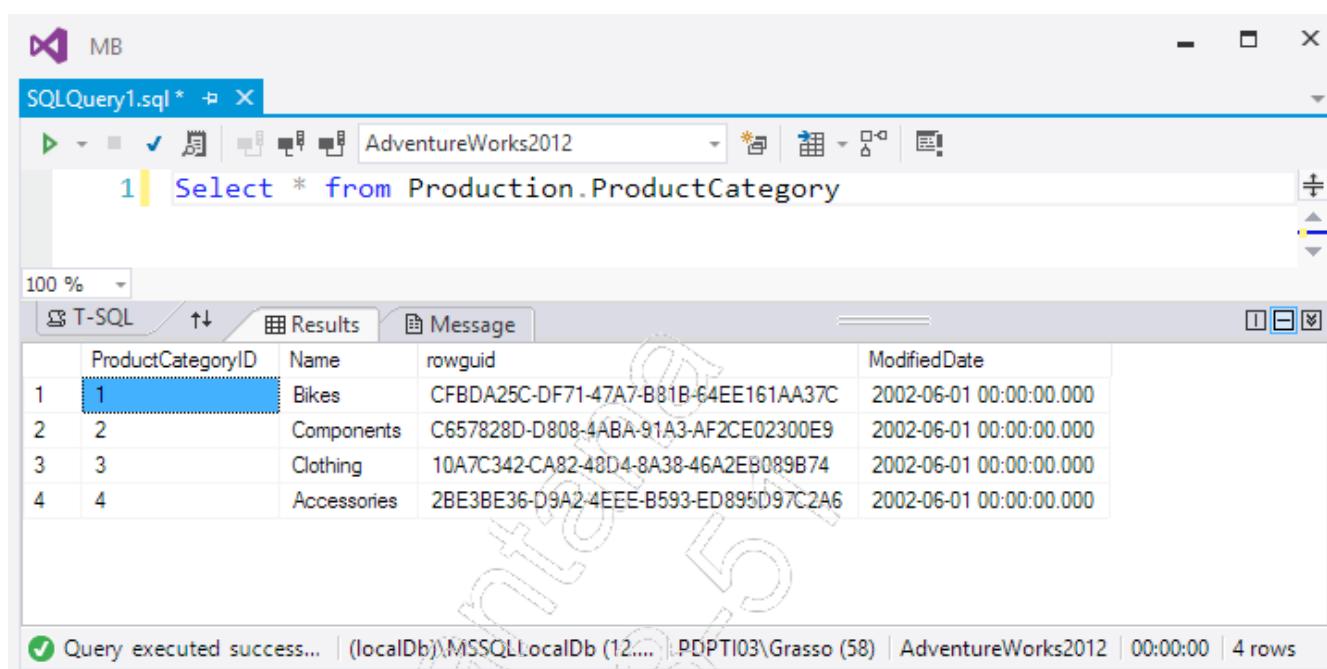


Visual Studio 2015 - ASP.NET com C# Acesso a dados

12. Teste a conexão com o banco executando uma query. Clique no botão **New Query**, selecione o banco de dados **AdventureWorks** e digite o comando SQL:

```
Select * from Production.ProductCategory
```

Em seguida, clique em **Executar** e veja o resultado:



The screenshot shows the SSMS interface with a query window titled "SQLQuery1.sql". The query "Select * from Production.ProductCategory" is entered. The results pane displays a table with four rows of data:

	ProductCategoryID	Name	rowguid	ModifiedDate
1	1	Bikes	CFBDA25C-DF71-47A7-B81B-64EE161AA37C	2002-06-01 00:00:00.000
2	2	Components	C657828D-D808-4ABA-91A3-AF2CE02300E9	2002-06-01 00:00:00.000
3	3	Clothing	10A7C342-CA82-48D4-8A38-46A2EB089B74	2002-06-01 00:00:00.000
4	4	Accessories	2BE3BE36-D9A2-4EEE-B593-ED895D97C2A6	2002-06-01 00:00:00.000

The status bar at the bottom indicates "Query executed success..." and "4 rows".

Pontos importantes:

- Os dados informados aqui se referem à versão 2012 do AdventureWorks. Pode ser usada a versão mais recente, se desejado;
- O nome do servidor pode variar. Nesses exemplos, foi considerada a versão Express local do SQL Server (**LocalDb\MySQLLocal**);
- Na página de download do banco de dados, existe sempre um documento com informações detalhadas da instalação para cada versão;
- É importante conseguir conectar e exibir informações desses bancos porque serão usados no decorrer do curso.

2

ADO.NET com Web Forms

- ✓ Classes do ADO.NET;
- ✓ Transação;
- ✓ Dados desconectados;
- ✓ Melhores práticas;
- ✓ Considerações sobre o uso das classes ADO.NET.



IMPACTA
EDITORA

2.1. Introdução

ADO.NET é o nome que se dá ao conjunto de classes de acesso a dados fornecido pela plataforma .NET. Criar um aplicativo usando ADO.NET é a forma mais eficiente do ponto de vista de performance e de controle das operações que são executadas no banco de dados.

Todos os outros modelos de acesso a dados usam internamente essas classes. Os frameworks, Web Controls e componentes da plataforma ou de terceiros são apenas programas compilados que utilizam o ADO.NET, usando boas práticas de programação e poupando grande parte do trabalho do programador de escrever cada detalhe de conexão, leitura, validação e conversão de tipos – tarefas sempre necessárias para o funcionamento de um software que manipula informações.

As formas de vincular informações vindas de um repositório de dados a elementos de tela variam de acordo como o modelo escolhido. Os Web Forms utilizam Web Controls que criam automaticamente o código HTML quando vinculados a uma fonte de dados. O MVC usa o Razor para inserir os dados no documento HTML. O programador, nesse caso, tem total controle do código gerado. As Web Pages utilizam um componente herdado da biblioteca WebMatrix chamado DataBase para criar conexões e disponibilizar dados na página.

2.2. Classes do ADO.NET

As classes para acesso a dados, chamadas de ADO.NET (ActiveX Data Objects), são independentes do modelo usado para renderizar as páginas de uma aplicação ASP.NET (Web Forms, MVC ou Web Pages).

Essas mesmas classes são utilizadas em uma aplicação Windows Forms ou aplicações baseadas em serviços, ou seja, o ADO.NET é um conjunto de classes que não estão diretamente vinculados a nenhuma arquitetura específica.

Esse aspecto é muito poderoso, pois permite criar módulos independentes que podem ser utilizados, atualizados e reaproveitados em diversos projetos.

O centro do ADO.NET é o namespace **System.Data.Common**, e suas três principais classes são as seguintes:

- **DbConnection**: Estabelece uma conexão com o banco;
- **DbCommand**: Executa comandos no banco de dados;
- **DbDataReader**: Retorna informações do banco.

Essas classes são abstratas, ou seja, não é possível criar instâncias diretamente, sendo sempre necessário usar as classes derivadas. O conjunto de classes derivadas que acessam um determinado tipo de banco de dados é chamado de **Provider**. O .NET Framework fornece alguns providers, mas praticamente todo banco de dados disponibiliza uma biblioteca de classes para conexão com os aplicativos .NET.

O .NET Framework é distribuído com os seguintes providers:

- **.NET Framework Data Provider para SQL Server**

Para acesso nativo ao SQL Server. Utiliza o namespace **System.Data.SqlClient**.

- **.NET Framework Data Provider para OLE DB**

Fornece acesso indireto (por meio do componente OLE DB) para qualquer banco de dados que tenha um conector compatível. É importante lembrar que OLE DB não é .NET, ou seja, não é um ambiente gerenciado, obrigando o sistema a fazer uma chamada externa cada vez que é usado. É preferível sempre usar um conector nativo, se possível. Hoje em dia, praticamente todo banco de dados tem um conector nativo .NET. Utiliza o namespace **System.Data.OleDb**.

- **.NET Framework Data Provider para ODBC**

Assim como o OLE DB, o provider ODBC utiliza um componente antigo de acesso a dados do Windows. A mesma recomendação é feita neste caso: é melhor usar um conector .NET nativo. Usa o namespace **System.Data.ODBC**.

- **.NET Framework Data Provider para Oracle**

Conector nativo para Oracle 8.1.7 e superior. Encontrado no namespace **System.Data.OracleClient**.

- **.NET Framework Data Provider para SQL Server Compact 4.0**

Conector nativo para SQL Server Compact 4.0, que é uma versão do SQL Server usado em dispositivos com Windows Phone e em computadores desktop para aplicações stand-alone. Usa o namespace **System.Data.SqlServerCe**.

- **.NET Framework Data Provider para EDM**

Conector nativo para Entity Data Model (EDM), que é uma série de ferramentas para descrever dados independentemente do banco de dados. É utilizado pelo Entity Framework.

Para qualquer outro banco de dados, é necessário obter o conector no site do fornecedor do banco. Por exemplo, para obter o Data Provider do MySQL, é necessário entrar no site MySQL.com e fazer o download do provider. O MySQL utiliza o namespace **Mysql.Data.MySqlClient**.

2.2.1. **DbConnection**

A classe derivada de **DbConnection** é o primeiro item a ser definido no processo de acesso a dados. O objetivo é estabelecer ou finalizar uma conexão com o banco de dados.

Vejamos, a seguir, suas principais propriedades:

- **ConnectionString**

Esta é a propriedade mais importante, pois é a que define os parâmetros de conexão com o banco. Existem muitos parâmetros possíveis, sendo os mais importantes: **Data Source** (o nome do servidor), **Initial Catalog** (o nome do banco de dados) e a autenticação que pode ser **Integrated Security** ou **User ID** (usuário) e **Password** (senha).

- Exemplo de string de conexão (**ConnectionString**) usando usuário e senha:

```
Data Source=localhost\sqlexpress;  
Initial Catalog=northwind;  
User ID=sa;  
Password=123;
```

- Exemplo de string de conexão (**ConnectionString**) usando a segurança integrada Windows:

```
Data Source=localhost\sqlexpress;  
Initial Catalog=northwind;  
Integrated Security=true;
```

- Exemplo de código usando **SqlConnection**:

```
string conexao=@"Data Source=localhost\sqlexpress;  
Initial Catalog=northwind;  
User Id=sa; Password=123";
```

```
var cn = new SqlConnection();  
cn.ConnectionString = conexao;
```

- Exemplo de código usando **SqlConnection** no construtor:

```
string conexao=@"Data Source=localhost\sqlexpress;  
Initial Catalog=northwind;  
User Id=sa; Password=123";  
  
var cn = new SqlConnection(conexao);
```

- **ConnectionTimeOut**

Retorna o tempo total, em segundos, em que uma tentativa de conexão será feita pelo método **Open()**. Esta propriedade é somente leitura. Seu valor deve ser definido na string de conexão.

- **State**

Retorna um enumerador com o status da conexão. Pode ser **Closed** (fechada), **Open** (aberta) ou **Connecting** (conectando).

- **Database**

O nome do banco de dados. Esta propriedade só fica disponível depois de a conexão estar aberta.

- **ServerVersion**

Retorna a versão do servidor onde está conectado.

Adiante, podemos ver os principais métodos da classe **DbConnection**:

- **Open()**: Abre a conexão com o banco. A string de conexão (**connectionString**) deve estar definida;
- **Close()**: Fecha a conexão com o banco. Pode ser chamado várias vezes, mesmo que a conexão esteja fechada;
- **CreateCommand()**: Retorna um objeto do tipo **DbCommand** associado à conexão atual;
- **BeginTransaction()**: Retorna um objeto do tipo **DbTransaction**, usado para criar transações no banco. Uma transação é um processo que assegura que diversas operações no banco de dados sejam executadas ou que todas as operações no processo sejam canceladas, deixando o banco sempre em um estado consistente.

Vejamos, a seguir, o principal evento da classe **DbConnection**:

- **StateChange**: Este é o único evento da classe **DbConnection**. Ocorre quando o estado da conexão muda (aberto para fechado, por exemplo).

Considere o seguinte exemplo de código usando **DbConnection**:

```
string conexao=@"Data Source=localhost\sqlexpress;  
                    Initial Catalog=northwind;  
                    User Id=sa; Password=123";  
  
using (var cn = new SqlConnection(conexao))  
{  
    cn.Open();  
  
    //comandos....  
  
    //opcional: o Dispose() fecha a conexão  
    cn.Close();  
}
```

//A partir deste ponto a variável cn não está mais disponível e
//a conexão foi fechada

```
using (var cn = new SqlConnection(conexao))  
{  
    //esta é outra conexão....  
}
```

O comando **using** garante que o objeto **cn** seja fechado e retirado da memória. Isso ocorre devido à implementação do método **Dispose()**, que faz parte da interface **IDisposable**.

O comando **using** só pode ser usado em classes que implementam essa interface. O que esse comando faz é chamar o método **Dispose()** antes de sair do bloco. Esse método fecha a conexão e libera quaisquer recursos do sistema utilizados pelo objeto.

2.2.2. **DbCommand**

As classes derivadas de **DbCommand** executam comandos nos bancos de dados. Essa classe necessita de um objeto **DbConnection** para estabelecer a conexão. É a principal classe do ADO.NET, pois é a que define o que vai ser executado no banco.

Vejamos, a seguir, suas principais propriedades:

- **CommandText**

Uma string que representa o que vai ser executado no banco. Pode ser uma expressão SQL, o nome de uma stored procedure ou o nome de uma tabela. Quando não é uma expressão SQL, é necessário definir a propriedade **CommandType**, que é um enumerador com as propriedades **Text** (expressão SQL), **StoredProcedure** e **TableDirect**:

- Exemplo de **DbCommand** usando **CommandText** com expressão SQL:

```
using (var cmd = new SqlCommand())
{
    cmd.Connection = cn; //cn é um objeto do tipo DbConnection
    cmd.CommandText = @"Select CompanyName, Phone
                      From Customers";
}
```

- Exemplo de **DbCommand** usando **CommandText** com o nome de uma stored procedure:

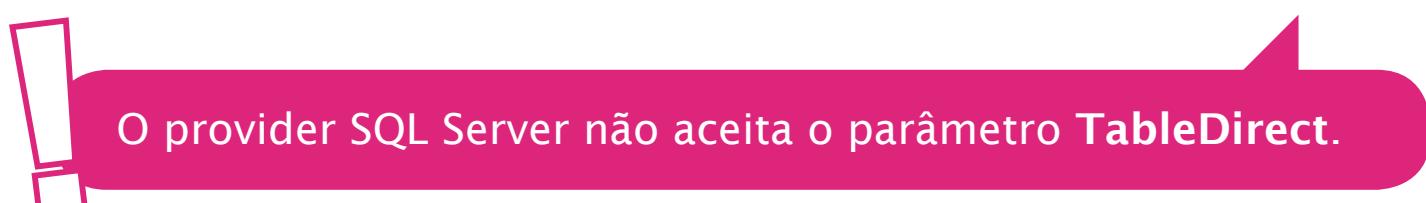
```
using (var cmd = new SqlCommand())
{
    cmd.Connection = cn; //cn é um objeto do tipo DbConnection
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.CommandText = "proc_clientesListagem";
}
```

- Exemplo de **DbCommand** usando **CommandText** com o nome de uma tabela:

```
using (var cmd = new OleDbCommand())
{
    cmd.Connection = cn; //cn é um objeto do tipo DbConnection
    cmd.CommandType = CommandType.TableDirect;

    cmd.CommandText = "Customers";
}
```



- **CommandType**

Enumerador que define o tipo de comando a ser executado. São definidas três possibilidades: **Text** (expressão SQL), **StoredProcedure** e **TableDirect**.

- **Connection**

Um objeto do tipo **DbConnection** que define a conexão a ser usada.

- **CommandTimeout**

O tempo de espera, em segundos, para um comando ser executado antes que seja gerado um erro. O padrão é 30 segundos.

- **Parameters**

Retorna uma coleção de objetos do tipo **DbParameter**. Essa coleção deve ser preenchida toda vez que o comando a ser executado tiver parâmetros.

- **Classe DbParameter**

Item da coleção **Parameters**. Contém propriedades para armazenar detalhes dos parâmetros. Suas principais propriedades são as seguintes:

- **ParameterName**: O nome do parâmetro;
- **DbType**: O tipo do parâmetro. É um enumerador, sendo que os principais tipos são: **Varchar**, **Int32**, **DateTime**, **Decimal**, **Double**, **Boolean**, **String** e **Binary**;
- **Size**: O tamanho, em bytes, do valor do parâmetro;
- **Value**: O valor do parâmetro;
- **Direction**: Um parâmetro pode ser de entrada, de saída, entrada e saída ou retorno do comando. Quando um parâmetro é de saída, o valor é preenchido na execução do comando e pode ser obtido depois da execução.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

A seguir, um exemplo de um comando com parâmetros:

```
string conexao=@"Data Source=localhost\sqlexpress;
                Initial Catalog=northwind;
                Integrated Security=true";

using (var cn = new SqlConnection(conexao))
{
    cn.ConnectionString = conexao;

    using (var cmd = new SqlCommand())
    {
        cmd.Connection = cn;
        cmd.CommandText = @"Select * from Customers
                           Where Country=@country";
        cmd.CommandType = CommandType.Text;

        var p1 = new SqlParameter();
        p1.ParameterName = "@country";
        p1.DbType = DbType.String;
        p1.Value = "Brazil";

        cmd.Parameters.Add(p1);

        cn.Open();
        using (var dr= cmd.ExecuteReader())
        {
            gv.DataSource = dr;
            gv.DataBind();
        }
    }
}
```

O provider para SQL Server apresenta uma maneira mais prática de adicionar parâmetros: o método **AddWithValue** permite passar o nome do parâmetro e o valor. O tipo é inferido com base no valor passado, ou seja: se for passado como valor um objeto do tipo **DateTime**, o tipo do parâmetro será **DateTime**; se for do tipo **int**, o tipo será do tipo **Int32**, e assim por diante.

```
string conexao=@"Data Source=localhost\sqlexpress;
                Initial Catalog=northwind;
                Integrated Security=true";

using (var cn = new SqlConnection(conexao))
{
    cn.ConnectionString = conexao;

    using (var cmd = new SqlCommand())
    {
        cmd.Connection = cn;
        cmd.CommandText = @"Select * from Customers
                           Where Country=@country";
        cmd.CommandType = CommandType.Text;

        cmd.Parameters.AddWithValue("@country", "brazil");

        cn.Open();
        using (var dr = cmd.ExecuteReader())
        {
            gv.DataSource = dr;
            gv.DataBind();
        }
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Os parâmetros de saída são variáveis manipuladas pelo comando SQL ou stored procedure. A seguir, um exemplo de como obter parâmetros de saída:

```
string conexao=@"Data Source=localhost\sqlexpress;
                Initial Catalog=northwind;
                Integrated Security=true";
using (var cn = new SqlConnection(conexao))
{
    cn.ConnectionString = conexao;

    using (var cmd = new SqlCommand())
    {
        cmd.Connection = cn;
        cmd.CommandText = @"Select @totalClientes=count(*)
                           from Customers;
                           Select @totalProdutos=count(*)
                           from Products";

        cmd.CommandType = CommandType.Text;

        var p1=new SqlParameter("@totalClientes", DbType.Int32);
        p1.Direction=ParameterDirection.Output;

        var p2=new SqlParameter("@totalProdutos", DbType.Int32);
        p2.Direction=ParameterDirection.Output;

        cmd.Parameters.Add(p1);
        cmd.Parameters.Add(p2);

        cn.Open();
        cmd.ExecuteNonQuery();
        cn.Close();

        label1.Text =
            cmd.Parameters["@totalClientes"].Value.ToString();
        label2.Text =
            cmd.Parameters["@totalProdutos"].Value.ToString();
    }
}
```

- **Transaction**

Propriedade que identifica a transação em que um **command** está inscrito. As transações são objetos do tipo **DbTransaction** criadas pelo método **BeginTransaction** da classe **DbConnection**. Um comando que faz parte de uma transação tem o seu comando processado apenas quando o método **Commit** do objeto **DbTransaction** é chamado, e tem o seu comando desfeito, voltando o banco para o estado anterior, quando é chamado o método **RollBack()** do objeto **transaction**. Um exemplo completo de transação será visto mais adiante, ao ser estudada a classe **DbTransaction**.

Adiante, podemos ver os principais métodos da classe **DbCommand**:

- **ExecuteNonQuery()**: Executa um comando e retorna um número inteiro representando o total de registros afetados pelo comando. Geralmente utilizado para comandos do tipo **Insert**, **Update**, **Delete** ou **Create**. A seguir, um exemplo do método **ExecuteNonQuery**, que aumenta em 10% o preço de produtos:

```
string conexao=@"Data Source=localhost\sqlexpress;
                Initial Catalog=northwind;
                Integrated Security=true";
using (var cn = new SqlConnection(conexao))
{
    cn.ConnectionString = conexao;

    using (var cmd = new SqlCommand())
    {
        cmd.Connection = cn;
        cmd.CommandText = @"Update Products
                           set UnitPrice=UnitPrice*1.1";

        cn.Open();
        int total = cmd.ExecuteNonQuery();
        cn.Close();

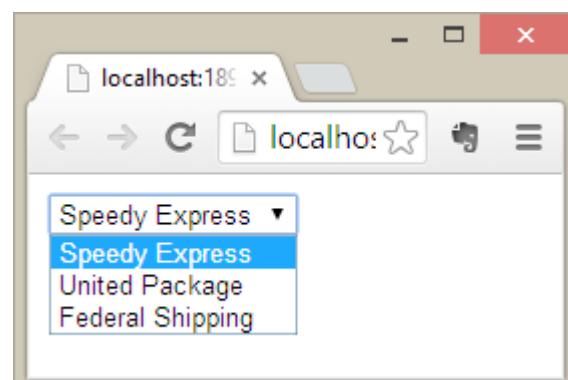
        label1.Text = "Total:" + total;
    }
}
```

- **ExecuteReader()**: Executa um comando e retorna um objeto do tipo **DbDataReader**, que permite ler o conteúdo do banco. Geralmente usado para comandos do tipo **Select**;

```
using (var cn = new SqlConnection(conexao))
{
    cn.ConnectionString = conexao;

    using (var cmd = new SqlCommand())
    {
        cmd.Connection = cn;
        cmd.CommandText = @"Select * from Shippers";

        cn.Open();
        using(var dr=cmd.ExecuteReader())
        {
            while (dr.Read())
            {
                string empresa = dr[ "CompanyName" ].ToString();
                dropDownList1.Items.Add(empresa);
            }
        }
    }
}
```



- **ExecuteReader (CommandBehavior)**: Esta sobrecarga de método permite passar um parâmetro do tipo **CommandBehavior**, um enumerador que define as seguintes possibilidades:
 - **CloseConnection**: A conexão é fechada quando o **DbDataReader** é fechado;
 - **Default**: Nenhuma ação extra é executada. Equivale a chamar o método **executeReader** sem parâmetros;
 - **KeyInfo**: Colunas extras são adicionadas com informação da chave primária;
 - **SchemaOnly**: Retorna apenas informações sobre a estrutura dos dados;
 - **SequencialAccess**: Retorna um stream em vez de retornar um registro completo. É usado para obter dados binários;
 - **SingleResult**: Retorna um único grupo de resultados, mesmo que a expressão SQL tenha dois ou mais **Selects**;
 - **SingleRow**: Retorna uma única linha.

O item **CloseConnection** é muito útil quando existe um método que retorna um **DbDataReader**, porque quem chama o método não tem acesso direto à conexão que o **DbDataReader** está usando.

Retornando um **DbDataReader** com o **CommandBehavior CloseConnection**, a conexão é fechada automaticamente quando o **DataReader** é fechado.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Vejamos um exemplo:

```
DbDataReader ListaDeClientes()
{
    string conexao = @"Data Source=localhost\sqlexpress;
                        Initial Catalog=northwind;
                        Integrated Security=true";
    var cn = new SqlConnection(conexao);
    var cmd=new SqlCommand("Select * from Shippers", cn);
    cn.Open();
    var dr= cmd.ExecuteReader(CommandBehavior.CloseConnection);
    return dr;
}

void PreencherClientes()
{
    using (var dr = ListaDeClientes())
    {
        while (dr.Read())
        {
            string nome = dr["CompanyName"].ToString();
            dropDownList1.Items.Add(nome);
        }
        //Neste ponto a conexão é fechada
        dr.Close();
    }
}
```

- **ExecuteScalar()**: Executa um comando e retorna um objeto como resultado. O comando pode ser de qualquer tipo, mas se for do tipo **Select** e retornar mais de uma linha ou coluna, apenas a informação da primeira linha e coluna será retornada.

No exemplo adiante, o total de fornecedores do banco de dados **Northwind** é retornado:

```
int total = 0;
string conexao = @"Data Source=localhost\sqlexpress;
                    Initial Catalog=northwind;
                    Integrated Security=true";

string sql = "Select count(*) from Shippers";

using(var cn = new SqlConnection(conexao))
{
    using(var cmd = new SqlCommand(sql, cn))
    {
        cn.Open();
        total = Convert.ToInt32(cmd.ExecuteScalar());
    }
}

label1.Text = total.ToString();
```

2.2.3. DbDataReader

A classe **DbDataReader** permite ler informações de um banco de dados de maneira sequencial. Isso significa que não é possível navegar pelos registros, apenas ler um de cada vez de maneira sequencial. Também não é possível alterar os dados, pois o valor fornecido pelo **DbDataReader** é do tipo **ReadOnly** (somente leitura).

A classe **DbDataReader** não pode ser instanciada diretamente. Ela só pode ser criada por meio do método **ExecuteReader()** da classe **DbCommand**.

Vejamos, a seguir, suas principais propriedades:

- **HasRows**: Retorna um valor booleano indicando se existe ou não registros a serem lidos;
- **FieldCount**: Retorna o número de campos;
- **IsClosed**: Retorna um valor booleano indicando se está fechado;
- **Item[string]**: Obtém um objeto representando o valor de um campo indicado pelo parâmetro. A propriedade **item** é padrão (**this**), portanto não precisa ser indicada. Exemplo:

```
string nome = dr["CompanyName"].ToString();
```

- **Item[int]**: Obtém um objeto representando o valor de um campo indicado pela posição do campo. A propriedade **item** é padrão (**this**), portanto não precisa ser indicada. Exemplo:

```
string nome = dr[0].ToString();
```

Adiante podemos ver os principais métodos da classe **DbDataReader**:

- **Read()**: Lê o registro atual e avança para o próximo. Retorna um valor booleano indicando se ainda existem registros. Veja o exemplo:

```
var dr = cmd.ExecuteReader();

if (dr.Read())
{
    string nome = dr["CompanyName"].ToString();
}
```

Se for esperada mais de uma linha de resultado, é comum usarmos um loop para percorrer todos os registros:

```
var dr = cmd.ExecuteReader();

while(dr.Read())
{
    string nome = dr["CompanyName"].ToString();
    dropDownList1.Items.Add(nome);
}
```

- **Close()**: Fecha o **DbDataReader**, liberando recursos do sistema;
- **GetInt32(int)**: Obtém o valor de um campo do tipo inteiro. Vejamos o exemplo:

```
var dr = cmd.ExecuteReader();  
  
if (dr.Read())  
{  
    int total = dr.GetInt32(0);  
}
```

- **GetDecimal(int)**: Obtém o valor de um campo do tipo decimal. Vejamos o exemplo:

```
var dr = cmd.ExecuteReader();  
  
if (dr.Read())  
{  
    decimal salario = dr.GetDecimal(0);  
}
```

- **GetDateTime(int)**: Obtém o valor de um campo do tipo decimal. Vejamos o exemplo:

```
var dr = cmd.ExecuteReader();  
  
if (dr.Read())  
{  
    DateTime data= dr.GetDateTime(0);  
}
```

- **GetString(int)**: Obtém o valor de um campo do tipo **Varchar** ou **Text**. Veja o exemplo:

```
var dr = cmd.ExecuteReader();  
  
if (dr.Read())  
{  
    String nome= dr.GetString(0);  
}
```

- **IsDBNull(int)**: Retorna um valor booleano indicando se uma coluna tem um valor nulo. Veja um exemplo de uso:

```
var dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
if (dr.Read())
{
    string nome;
    if (dr.IsDBNull(0))
    {
        nome = string.Empty;
    }
    else
    {
        nome = dr[0].ToString();
    }
}
```

Ou, também, usando o operador ternário:

```
var dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
if (dr.Read())
{
    string nome=dr.IsDBNull(0) ? string.Empty : dr[0].ToString();
}
```

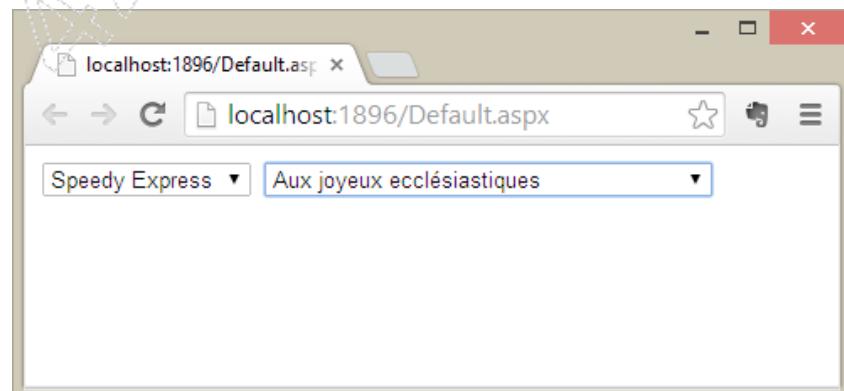
- **NextResult()**: Avança para o próximo bloco de leitura, quando a expressão SQL tem mais de um **Select**. Vejamos o exemplo:

```
string conexao = @"Data Source=localhost\sqlexpress;
                    Initial Catalog=northwind;
                    Integrated Security=true";

string sql = @"Select CompanyName from Shippers;
                Select CompanyName from Suppliers";

using (var cn = new SqlConnection(conexao))
{
    using (var cmd = new SqlCommand(sql, cn))
    {
        cn.Open();
        using (var dr = cmd.ExecuteReader())
        {
            while (dr.Read())
            {
                dropDownList1.Items.Add(dr.GetString(0));
            }

            if (dr.NextResult())
            {
                while (dr.Read())
                {
                    dropDownList2.Items.Add(dr.GetString(0));
                }
            }
        }
    }
}
```



2.3. Transação

Uma transação é um processo em que diversos comandos devem ser executados, sendo que, se algum deles falhar, todos os comandos já executados devem ser desfeitos.

Consideremos o exemplo de uma transação bancária de transferência de valores de uma conta para outra. Os seguintes passos são executados:

1. O dinheiro é retirado de uma conta;
2. O dinheiro é depositado em outra conta.

Vamos supor que o primeiro item é executado e, ao executar o segundo item, ocorra um erro. O dinheiro terá saído de uma conta, mas não estará na segunda, criando uma situação inconsistente.

Uma transação é uma garantia do banco de dados de que diversos comandos podem ser executados ou desfeitos como se fosse uma única operação.

No ADO.NET, as transações são feitas usando a classe **DbTransaction**, por meio da seguinte sequência:

1. Uma conexão é criada;
2. Essa conexão cria um objeto do tipo **DbTransaction**;
3. Diversos comandos são criados usando a mesma conexão;
4. Os comandos se "inscrevem" na transação definindo a propriedade **transaction** para o objeto criado pela conexão;
5. Os comandos são executados;
6. A transação é confirmada pelo método **Commit** da classe **DbTransaction**;
7. Ou a transação é cancelada pelo método **RollBack** da classe **DbTransaction**.

Existem diversos níveis de detalhamento dessas operações. Por exemplo, os dados devem ser bloqueados enquanto uma transação estiver ocorrendo, ou podem ser liberados apenas para leitura, ou, ainda, liberados para leitura apenas se foram modificados. Existem, ainda, as decisões que devem ser tomadas se duas ou mais transações bloquearem os registros.

É comum um erro chamado **DeadLock** (bloqueio mortal), que ocorre quando uma transação fica esperando outra terminar e esta última fica esperando a primeira terminar, deixando a aplicação em um loop infinito.

Outro erro comum é chamado **Dirty Read** (leitura suja), que ocorre quando uma transação está alterando uma informação e outra transação lê dados inconsistentes, porque a primeira pode ter desfeito as alterações.

2.3.1. DbTransaction

Esta é a classe que permite executar transações no banco e controlar o nível de isolamento das informações para evitar os erros comuns deste tipo de operação.

Vejamos, a seguir, suas principais propriedades:

- **Connection:**

A conexão que criou a instância da classe.

- **IsolationLevel:**

O nível de isolamento dos registros. É um enumerador que pode ter os seguintes atributos:

- **Chaos:** As mudanças de outras transações não podem ser sobreescritas;
- **ReadCommitted:** Os dados são bloqueados enquanto estão sendo lidos, mas podem mudar depois do final da transação;
- **ReadUncommitted:** Os dados não são bloqueados enquanto estão sendo lidos. Isso deixa a transação mais rápida, mas pode gerar leituras erradas de outras transações;
- **RepeatableRead:** Os dados são bloqueados para alteração por outras transações;

- **Serializable**: Um bloqueio é colocado em uma grupo de registros, evitando que outras transações insiram ou alterem dados. Este é o padrão;
- **Snapshop**: Armazena uma versão do estado atual do banco e executa a transação, não bloqueando outras transações e não enxergando as alterações de outras transações. É um modo rápido, mas pode ocorrer problemas se duas ou mais transações alterarem os dados que estão sendo manipulados;
- **Unspecified**: O modo de bloqueio é definido pelo banco de dados ou componente.

Adiante, podemos ver os principais métodos da classe **DbTransaction**:

- **Commit()**: Os dados são gravados;
- **RollBack()**: As alterações são desfeitas.

O exemplo a seguir retira um item de um produto do estoque e o adiciona ao estoque de outro produto. Os dois comandos devem ser executados. Vejamos como a transação resolve este caso:

```
string conexao = @"Data Source=localhost\sqlexpress ...";  
  
string sql1 = @"Update Products Set UnitsInStock=UnitsInStock-1  
Where ProductId=1";  
  
string sql2 = @"Update Products Set UnitsInStock=UnitsInStock+1  
Where ProductId=2";  
using (var cn = new SqlConnection(conexao))  
{  
    SqlTransaction transacao=null;  
    try  
    {  
        cn.Open();  
  
        //Cria a transação  
        transacao = cn.BeginTransaction();  
    }
```

```
//Cria os comandos
var cmd1 = new SqlCommand(sql1, cn);
var cmd2 = new SqlCommand(sql2, cn);

//Inscreve os comandos na transação
cmd1.Transaction = transacao;
cmd2.Transaction = transacao;

//Executa
cmd1.ExecuteNonQuery();
cmd2.ExecuteNonQuery();

//Confirma
transacao.Commit();
}

catch
{
    //Ocorreu um erro, cancela a operação
    //Antes, verifica se a transação existe e
    //se a conexão está aberta
    if (transacao != null &&
        cn.State == ConnectionState.Open)
    {
        transacao.Rollback();
    }
}
}
```

2.4. Dados desconectados

As classes do ADO.NET **DbConnection**, **DbCommand** e **DbDataReader** trabalham sempre conectadas a uma fonte de dados. O ADO.NET também oferece outras classes para trabalhar de maneira desconectada, ou seja, com informações na memória que representam os dados da aplicação.

As classes desconectadas do ADO.NET estão no namespace **System.Data**, e a principal delas é a classe **DataSet**. Esta classe contém propriedades para definir tabelas, campos, registros, relacionamentos, chaves primárias, restrições, visualizações, ou seja, tudo que existe em um banco de dados. Outras classes que representam os dados são **DataTable**, **DataColumn**, **DataRow** e **DataView**.

2.4.1. DataSet

A classe **DataSet** contém propriedades e métodos para armazenar dados em formato de tabelas, bem como métodos para exportar dados para XML. Ela pode ser facilmente manipulada para ler ou alterar informações de um banco de dados.

Vejamos, a seguir, suas principais propriedades:

- **Tables**: Uma coleção de objetos do tipo **DataTable**. A classe **DataTable** contém informações em formato de linhas e colunas;
- **Relations**: Uma coleção de relacionamentos (objetos do tipo **DataRelation**) entre as tabelas. Esses relacionamentos podem ser usados para filtrar dados e manter sua integridade;
- **DataSetName**: O nome do **DataSet**. Este nome é utilizado quando exportamos os dados para XML.

Adiante, podemos ver os principais métodos da classe **DataSet**:

- **AcceptChanges()**: Grava as informações definitivamente na memória, indicando que não houve alterações. É usado para sincronizar dados com um banco de dados;
- **Clear()**: Limpa todos os dados de todas as tabelas e relacionamentos;
- **Clone()**: Cria uma cópia da estrutura das tabelas;
- **Copy()**: Cria uma cópia da estrutura e dos dados das tabelas;
- **HasChanges()**: Retorna um valor booleano indicando se houve alteração nos dados;
- **Load()**: Preenche o **DataSet** com informações de um **DbDataReader**;
- **Merge()**: Combina o **DataSet** com outro **DataSet**, mesclando campos e dados;
- **ReadXml()**: Carrega o **DataSet** com informações de um arquivo XML;
- **WriteXml()**: Cria um arquivo XML com as informações contidas nas tabelas.

2.4.1.1. DataTable

Contém informações em formato de tabela, com linhas e colunas. Pode existir isolado ou dentro de um **DataSet**, na coleção **Tables**.

Vejamos, a seguir, suas principais propriedades:

- **Columns**

É uma coleção de objetos do tipo **DataColumn**. Representam os campos de uma tabela. É possível criar colunas em um **DataTable** usando o método **Add** e passando uma string, representando o nome, e opcionalmente o tipo. Quando o tipo não é informado, é criada uma coluna do tipo **String**. Vejamos o exemplo:

```
var tb = new DataTable();
tb.Columns.Add("Nome");
tb.Columns.Add("Telefone");
tb.Columns.Add("Idade", typeof(int));
```

- **Rows**

Uma coleção de objetos do tipo **DataRow**. Representam os registros de uma tabela. É possível adicionar linhas em um **DataTable** usando o método **Add** e passando um array de objetos, sendo um para cada campo da tabela. Veja o exemplo:

```
var tb = new DataTable();
tb.Columns.Add("Nome");
tb.Columns.Add("Telefone");
tb.Columns.Add("Idade", typeof(int));
```

```
tb.Rows.Add("José", "5555-2323", 25);
tb.Rows.Add("Maria", "5555-2323", 32);
```

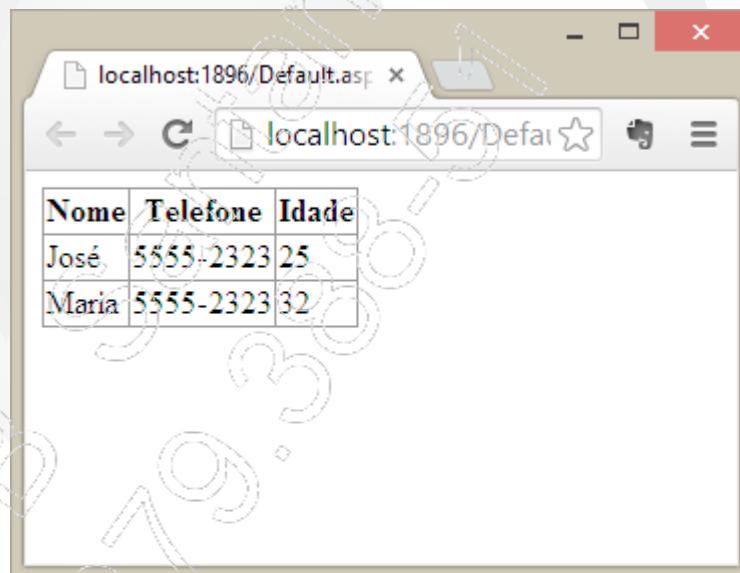
Visual Studio 2015 - ASP.NET com C# Acesso a dados

Um **DataTable** pode ser vinculado a um objeto ASP.NET que exiba campos, como **DataGrid** ou **DataList**, por meio da propriedade **DataSource** e do método **.DataBind()** desses objetos.

```
var tb = new DataTable();
tb.Columns.Add("Nome");
tb.Columns.Add("Telefone");
tb.Columns.Add("Idade", typeof(int));

tb.Rows.Add("José", "5555-2323", 25);
tb.Rows.Add("Maria", "5555-2323", 32);
```

```
gv.DataSource = tb;
gv.DataBind();
```



Usando o **DataTable** e as classes **DbConnection**, **DbCommand** e **DbDataReader**, é possível ler informações do banco de dados e transferir esses dados para a memória. Basta criar um **DataTable** com a mesma estrutura da tabela do banco, ler os dados e incluir esses dados no **DataTable** criado. Isso cria novas possibilidades, como armazenar dados na memória do servidor e usar recursos dos Web Controls como paginação automática.

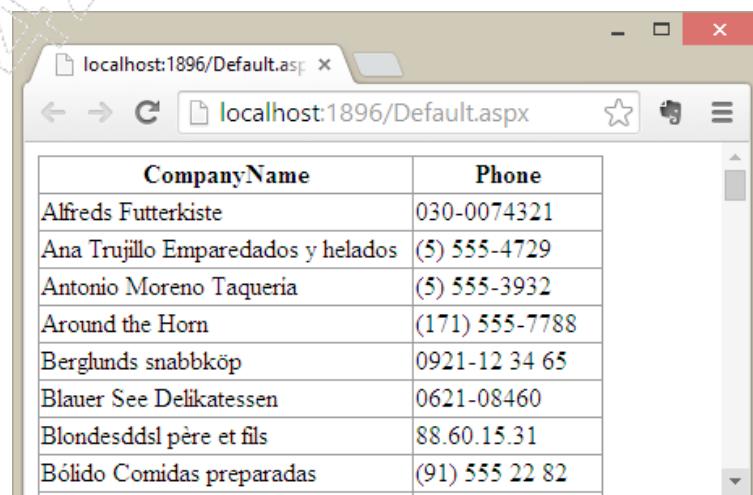
O exemplo a seguir mostra como obter dados de uma tabela do SQL Server, transferir para um **DataTable** e exibir em um **DataGrid**:

```
string conexao = @"Data Source=localhost\sqlexpress;
                    Initial Catalog=northwind;
                    Integrated Security=true";

string sql = @"Select CompanyName, Phone from Customers";

var tb = new DataTable();
tb.Columns.Add("CompanyName");
tb.Columns.Add("Phone");

using(var cn=new SqlConnection(conexao))
{
    using (var cmd = new SqlCommand(sql, cn))
    {
        cn.Open();
        using (var dr = cmd.ExecuteReader())
        {
            while (dr.Read())
            {
                string nome = dr["CompanyName"].ToString();
                string telefone = dr["Phone"].ToString();
                tb.Rows.Add(nome,telefone);
            }
        }
    }
}
gv.DataSource = tb;
gv.DataBind();
```



A screenshot of a web browser window titled "localhost:1896/Default.aspx". The page contains a DataGrid control that displays two columns: "CompanyName" and "Phone". The data is populated with records from the Northwind database's "Customers" table. The browser interface includes standard navigation buttons (back, forward, search) and a toolbar.

CompanyName	Phone
Alfreds Futterkiste	030-0074321
Ana Trujillo Emparedados y helados	(5) 555-4729
Antonio Moreno Taqueria	(5) 555-3932
Around the Horn	(171) 555-7788
Berglunds snabbköp	0921-12 34 65
Blauer See Delikatessen	0621-08460
Blondesddsl père et fils	88.60.15.31
Bólido Comidas preparadas	(91) 555 22 82

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Note que o código é muito extenso para realizar uma operação tão comum. Apenas dois campos foram usados nesse exemplo, mas, se fosse uma tabela com muitos campos, com dados de diversos tipos e, ainda, com valores nulos em alguns campos, o código seria muito maior.

O ADO.NET tem uma classe que serve exatamente para este propósito: transferir informações do banco de dados para a memória e vice-versa. Essa classe se chama **DbDataAdapter**.

A seguir, um exemplo da mesma operação usando um **DbDataAdapter**:

```
string conexao = @"Data Source=localhost\sqlexpress;  
                    Initial Catalog=northwind;  
                    Integrated Security=true";  
string sql = @"Select CompanyName, Phone from Customers";  
  
var tb = new DataTable();  
  
var da = new SqlDataAdapter(sql, conexao);  
da.Fill(tb);  
  
gv.DataSource = tb;  
gv.DataBind();
```

O construtor do **DbDataAdapter** recebe dois parâmetros: uma expressão SQL e uma string de conexão.

O método **Fill** cria no **DataTable** a estrutura necessária do comando definido pela expressão SQL, abre a conexão com o banco, percorre os registros, adiciona as linhas no **DataTable** e fecha a conexão. Tudo em apenas uma linha de código:

```
da.Fill(tb);
```

O resultado é exatamente o mesmo.

- **DataSet**

Retorna o dataset vinculado ao **DataTable**, se houver. O **DataTable** não precisa estar vinculado a um **DataSet**.

- **DefaultView**

Retorna um objeto do tipo **DataView**. A classe **DataView** é responsável pelos registros exibidos e pela(s) coluna(s) de ordenação. Depois que um **DataTable** é preenchido com dados, vindos de um banco de dados ou não, o **DataView** permite filtrar ou ordenar os dados. Isso evita solicitações ao banco de dados, o que torna a aplicação mais eficiente na manipulação de dados. A classe **DataView** contém as seguintes propriedades:

- **RowFilter**: Uma expressão SQL para definir o filtro dos dados. O exemplo adiante filtra um **DataTable**, exibindo apenas os produtos com preço unitário maior que 100:

```
minhaTabela.DefaultView.RowFilter = "UnitPrice>100";
```

- **Sort**: Uma string contendo o nome do(s) campo(s) de ordenação. O exemplo a seguir ordena os dados de um **DataTable** pelo campo **Nome**:

```
tb.DefaultView.Sort = "Nome";
```

É possível criar várias views para uma mesma tabela. E a classe **DataView** permite ser usada como parâmetro para a propriedade **DataSource**.

```
var dv = new DataView(minhaTabela);
dv.RowFilter = "UnitPrice>100";

gv.DataSource = dv;
gv.DataBind();
```

- **HasErrors**: Retorna se um **DataTable** tem erros quando os dados são sincronizados;
- **PrimaryKey**: Um array de objetos do tipo **DataColumn** contendo a chave primária da tabela;

- **TableName**: O nome da tabela. É usado quando os dados são exportados para XML.

Adiante, podemos ver os principais métodos da classe **DataTable**:

- **AcceptChanges()**: Grava as informações definitivamente na memória, indicando que não houve alterações. É usado para sincronizar dados com um banco de dados;
- **Clear()**: Limpa todas as linhas da tabela;
- **Clone()**: Cria uma cópia da estrutura das tabelas;
- **Copy()**: Cria uma cópia da estrutura e dos dados das tabelas;
- **HasChanges()**: Retorna um valor booleano indicando se houve alteração nos dados;
- **Load()**: Preenche o **DataTable** com informações de um **DbDataReader**;
- **NewRow()**: Cria uma linha com a mesma estrutura da tabela. Essa linha não está vinculada à tabela; é necessário adicioná-la à coleção **Rows** depois de preenchida. O exemplo a seguir mostra como adicionar uma linha à tabela usando este método:

```
DataRow linha = tb.NewRow();
linha[ "CompanyName" ] = "Empresa X";
linha[ "Phone" ] = "5555-3434";

tb.Rows.Add(linha);
```

- **Select()**: Retorna um array de objetos do tipo **DataRow**, contendo todos os registros;

- **Select(filtro)**: Retorna um array de objetos do tipo **DataRow**, contendo os registros filtrados de acordo com um critério. Vejamos o exemplo a seguir:

```
DataRow[] linhas = tb.Select("UnitPrice>100");
if (linhas.Length == 0)
{
    label1.Text = "Nenhum registro encontrado";
}
else
{
    label1.Text= linhas.Length + " registro(s) encontrado(s).";
}
```

- **Merge()**: Combina o **DataTable** com outro **DataTable**, mesclando campos e dados;
- **WriteXml()**: Cria um arquivo XML com as informações contidas na tabela. A tabela precisa ter um nome (propriedade **TableName**). O exemplo a seguir grava um arquivo XML na mesma pasta onde o aplicativo ASP.NET está instalado:

```
tb.TableName = "teste";
string arquivo = Server.MapPath("dados.xml");
tb.WriteXml(arquivo);
```

Vejamos o arquivo gerado:

```
<?xml version="1.0" standalone="yes"?>
<DocumentElement>
    <teste>
        <CompanyName>Alfreds Futterkiste</CompanyName>
        <Phone>030-0074321</Phone>
    </teste>
    <teste>
        <CompanyName>Ana Trujillo </CompanyName>
        <Phone>(5) 555-4729</Phone>
    </teste>
    <teste>
        <CompanyName>Antonio Moreno Taquería</CompanyName>
        <Phone>(5) 555-3932</Phone>
    </teste>
</DocumentElement>
```

- **ReadXml()**: Carrega o **DataTable** com informações de um arquivo XML. O exemplo a seguir carrega o **DataTable**:

```
string arquivo = Server.MapPath("dados.xml");
tb.ReadXml(arquivo);
```

2.5. Melhores práticas

A melhor maneira de usar o ADO.NET em uma aplicação comercial tradicional que tenha entidades como clientes, pagamentos, fornecedores, notas fiscais etc. é criando classes que representem esses dados e usando o ADO.NET para preencher instâncias dessas classes. Esse processo se chama **Domain Model**, ou **Modelo de Domínio**.

Por exemplo, em uma aplicação em que é necessário registrar os clientes de uma empresa e as solicitações que esses clientes fizeram, podemos criar duas classes: **Cliente** e **ClienteSolicitacao**. Vejamos o código a seguir:

```
public class Cliente
{
    public int ClienteId { get; set; }
    public string Nome { get; set; }
    public string Telefone { get; set; }
    public string Email { get; set; }

}

public class ClienteSolicitacao
{
    public int ClienteId { get; set; }
    public DateTime DataContato { get; set; }
    public string Solicitacao { get; set; }
    public string Observacao { get; set; }
    public bool Finalizado { get; set; }
}
```

Este modelo está propositalmente simplificado, tentando reproduzir fielmente uma ou mais tabelas de um banco de dados. Em uma aplicação real, o Modelo de Domínio não faz uso de chave primária e chave estrangeira (Clienteid) e não tem necessariamente a mesma estrutura de uma tabela.

Depois de definido o modelo, é importante separar as operações **CRUD** (Create, Read, Update e Delete – Criar, Ler, Atualizar e Excluir) em classes separadas da interface do usuário.

Uma classe isolada do resto do sistema para armazenar a conexão é importante para centralizar a string de conexão em um único local:

```
public class Db
{
    public static string Conexao =
        @"Data Source=localhost\sqlexpress;
        Initial Catalog=EmpresaApp;
        Integrated Security=true";
}
```

Uma classe para realizar as operações no banco de dados, em que os métodos de inclusão, alteração e exclusão recebem uma instância da classe de modelo de dados:

```
public class ClienteDb
{
    public void Incluir(Cliente cli)
    {
        string sql = @"Insert Into Cliente (Nome, Telefone, Email)
                       Values(@Nome, @Telefone, @Email)";

        using (var cn = new SqlConnection(Db.Conexao))
        {
            using (var cmd = new SqlCommand(sql, cn))
            {
                cmd.Parameters.AddWithValue("@Nome", cli.Nome);
                cmd.Parameters.AddWithValue("@Telefone", cli.Telefone);
                cmd.Parameters.AddWithValue("@Email", cli.Email);

                cn.Open();
                cmd.ExecuteNonQuery();
            }
        }
    }
}
```

O método **Incluir** recebe uma instância da classe **Cliente**

Os dados do cliente são passados para o comando de inclusão

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Os métodos de consulta retornam uma instância de um **DbDataReader**, um **DataTable** ou **DataSet** ou uma coleção de objetos do **Modelo de Domínio**. O exemplo a seguir mostra um método que retorna uma coleção de objetos do tipo cliente:

```
public class ClienteDb
{
    public void Incluir(Cliente cli)...

    public List<Cliente> ListaDeClientes()
    {
        var lista = new List<Cliente>();
        string sql = @"Select ClienteId, Nome, Email
                      From Clientes";
        using (var cn = new SqlConnection(Db.Conexao))
        {
            using (var cmd = new SqlCommand(sql, cn))
            {
                using (var dr = cmd.ExecuteReader())
                {
                    while (dr.Read())
                    {
                        var cli = new Cliente();
                        cli.ClienteId = Convert.ToInt32(dr["ClienteId"]);
                        cli.Nome = Convert.ToString(dr["Nome"]);
                        cli.Telefone = Convert.ToString(dr["Telefone"]);
                        cli.Email = Convert.ToString(dr["Email"]);

                        lista.Add(cli);
                    }
                }
            }
        }
        return lista;
    }
}
```

Cada objeto criado
é adicionado na lista
que será retornada.

Cada registro retornado
pelo DbDataReader e
convertido em um objeto
Cliente

A página ASP.NET fica apenas com a função de exibir os dados e passar as informações para a classe de dados:

- Exemplo do código na página ASP.NET para exibir dados do cliente:

```
var db = new ClienteDb();
gv.DataSource = db.ListaDeClientes();
gv.DataBind();
```

- Exemplo do código na página ASP.NET para executar a inclusão de um cliente:

```
var cli = new Cliente();
cli.Nome = nomeTextBox.Text;
cli.Telefone = telefoneTextBox.Text;
cli.Email = emailTextBox.Text;

var db = new ClienteDb();
db.Incluir(cli);
```

Repare que não existe na página nenhuma indicação do banco de dados que está sendo utilizado. O processo todo está encapsulado nas classes de acesso a dados.

Se for necessário trocar o banco de dados de SQL Server para MySQL, por exemplo, é necessário apenas alterar a classe **ClienteDb**. Nada precisa mudar na tela ou na maneira como a interface de usuário manipula os dados. Essa é a vantagem de criar um **Modelo de Dados**.

2.6. Considerações sobre o uso das classes ADO.NET

O ADO.NET é a base de toda a estrutura da plataforma .NET em relação ao acesso a dados. É importante conhecer as classes, suas propriedades e métodos. O uso de frameworks ou de ferramentas de arrastar e soltar facilita e diminui a quantidade de código que o programador precisa criar, porém limita os recursos disponíveis. Usando as classes do ADO.NET, não existe limite para o que é possível criar para atender as necessidades de um projeto.

É importante saber, também, que não existe um único padrão ou uma maneira correta de criar um aplicativo eficiente. Existem conceitos, padrões e modelos que podem ser seguidos porque foram testados e é sabido que funcionam, desde que inseridos no mesmo contexto em que foram elaborados ou catalogados.

Mas programação é, antes de tudo, uma atividade criativa, e novos padrões aparecem a todo momento. Os modelos apresentados aqui são sugestões de desenvolvimento usando modelos conhecidos para aproveitar ao máximo os recursos que a plataforma nos disponibiliza.

Mas lembre-se: você não está limitado a usar exatamente como foi proposto. Tudo está aberto para variações, adaptações e modificações.

Pontos principais

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

- ADO.NET é um conjunto de classes da plataforma .NET para acesso a dados;
- O ADO.NET utiliza o conceito de Providers para acessar diversos tipos de banco de dados;
- As principais classes que servem de base para os providers são: **DbConnection**, **DbCommand**, **DbDataReader** e **DbDataAdapter**;
- As classes do ADO.NET para armazenar dados na memória são **DataSet**, **DataTable**, **DataColumn**, **DataRow** e **DataView**;
- Uma transação é um processo que garante a execução de vários comandos ou o cancelamento de todos se algo falhar. A classe base **DbTransaction** serve para criar transações;
- Um **modelo de domínio** é o melhor uso do ADO.NET para aplicações comerciais.

2

ADO.NET com Web Forms

Teste seus conhecimentos

Mikael
B
426.27
107



IMPACTA
EDITORA

1. Qual classe do ADO.NET é usada para executar comandos no banco?

- a) DbConnection
- b) DbCommand
- c) DbDataReader
- d) SqlServerCommand
- e) DbAdapter

2. Em qual modelo ASP.NET pode ser usado o ADO.NET?

- a) Web Forms
- b) MVC
- c) Web Pages
- d) Todos os modelos.
- e) Nenhum modelo.

3. Qual classe permite armazenar dados de uma tabela na memória?

- a) DataReader
- b) DataTable
- c) DataColumn
- d) DataRow
- e) Table

4. Qual método é utilizado para executar um comando de alteração de banco de dados?

- a) ExecuteReader
- b) ExecuteScalar
- c) ExecuteXml
- d) ExecuteUpdate
- e) ExecuteNonQuery

5. Em ADO.NET, o que significa Provider?

- a) São classes que servem de base para outras classes de acesso a dados.
- b) São classes que acessam dados de um determinado banco ou usando determinado componente.
- c) São modelos de projetos em camadas.
- d) São classes que servem para conectar servidores de dados como SQL Server e Oracle usando código nativo .NET.
- e) É um conjunto de padrões de desenvolvimento para acesso a dados.

2

ADO.NET com Web Forms

Mãos à obra!

Mikael
A26.27
Santana



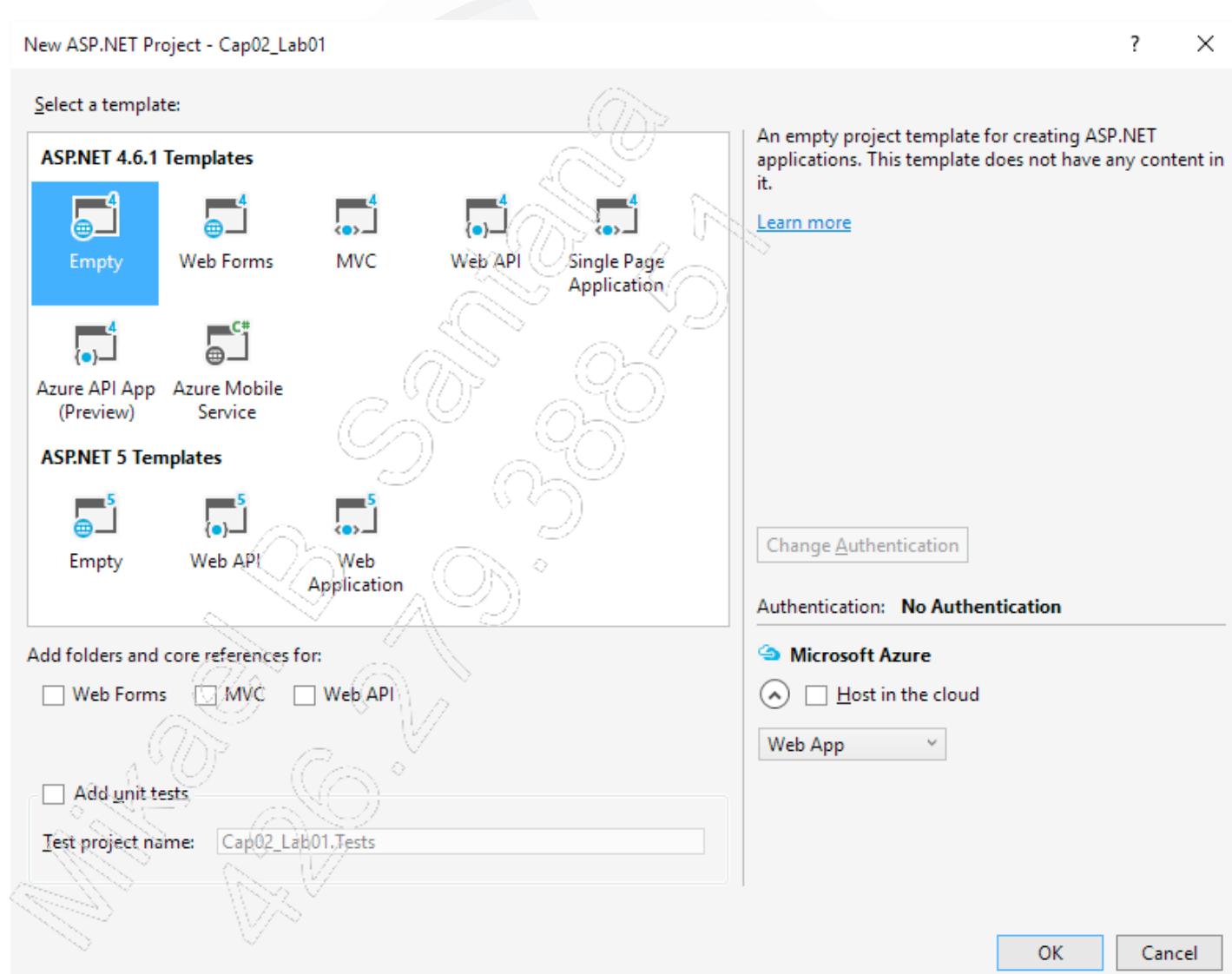
IMPACTA
EDITORA

Laboratório 1

A – Criando uma página que permita consultar os produtos da empresa Northwind Traders por categoria ou por fornecedor

Neste laboratório, usaremos o ADO.NET para extrair informações, bem como usaremos classes de acesso a dados separadas da interface e criaremos telas de consulta.

1. Crie um novo projeto Web vazio chamado **Cap02_Lab01**:



2. Adicione uma classe chamada **NorthwindDb**:

```
public class NorthwindDb
{
}
```

3. Adicione uma propriedade estática, somente leitura, chamada **Conexao**, que retorne a string de conexão com o **Northwind**:

```
public class NorthwindDb
{
    public static string Conexao
    {
        get
        {
            return @"Data Source=localhost\sqlexpress;
                    Initial Catalog=northwind;
                    Integrated Security=true";
        }
    }
}
```

4. Adicione uma classe chamada **ProdutosDb**:

```
public class ProdutosDb
{
}
```

5. Importe os namespaces **System.Data**, **System.Data.Common** e **System.Data.SqlClient**:

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

using System.Data;
using System.Data.Common;
using System.Data.SqlClient;
```

6. O objetivo é criar uma página que retorne dados dos produtos por categoria e por fornecedor. Crie um método que retorne um **DataTable** com a lista de categorias, contendo o código (**CategoryId**) e o nome (**CategoryName**). Isso vai servir para criar um **DropDownList** com a lista de categorias:

```
public class ProdutosDb
{
    //
    // Categorias Lista
    // Retorna a lista de categorias de Produtos
    //

    public DataTable CategoriasLista()
    {
        string sql = @"SELECT CategoryId, CategoryName
                      FROM Categories";

        var da = new SqlDataAdapter(sql, NorthwindDb.Conexao);

        var tb = new DataTable();

        da.Fill(tb);

        return tb;
    }
}
```

7. Adicione um método que retorne a lista de produtos de uma categoria. Esse método retorna um **DataTable** contendo o nome (**ProductName**), o preço (**UnitPrice**) e o estoque (**UnitsInStock**) dos produtos que pertencem à categoria fornecida. O campo **CategoryId** da tabela **Products** contém o código da categoria daquele produto;

```
public class ProdutosDb
{
    //
    //ProdutosPorCategoria
    //  Obtém uma lista de Produtos
    //  de uma determinada categoria
    //

    public DataTable ProdutosPorCategoria(int categoriaId)
    {
        string sql = @"SELECT ProductName,
                           UnitPrice,
                           UnitsInStock
                      FROM Products
                     WHERE CategoryId=@CategoryId";

        var da = new SqlDataAdapter(sql, NorthwindDb.Conexao);

        da.SelectCommand
            .Parameters
            .AddWithValue("@CategoryId", categoriaId);

        var tb = new DataTable();
        da.Fill(tb);
        return tb;
    }
}
```

! Esta é toda a parte necessária para criar a primeira parte da tela de pesquisa, onde é apresentada uma lista de categorias e, quando selecionada, uma lista de produtos daquela categoria.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

8. Adicione no projeto uma página chamada **Produtos.ASPX**;
9. Apague a div inserida automaticamente e adicione na página o título:

```
<body>
  <form id="form1" runat="server">

    <h1>Northwind - Produtos </h1>

  </form>
</body>
```

10. Adicione uma div e, dentro, coloque um span e dois Buttons. Isso vai servir para o usuário escolher como quer pesquisar:

```
<body>
  <form id="form1" runat="server">

    <h1>Northwind - Produtos </h1>

    <div>
      <span>Pesquisar por: </span>
      <asp:Button ID="categoriaButton"
        runat="server"
        Text="Categoria" />
      <asp:Button ID="fornecedorButton"
        runat="server"
        Text="Fornecedor" />
    </div>

  </form>
</body>
```

11. Adicione um MultiView com duas Views. Uma será para visualizar a consulta por categoria e a outra, por fornecedor;

```
<body>
    <form id="form1" runat="server">

        <h1>Northwind - Produtos </h1>

        <div>

            <span>Pesquisar por: </span>

            <asp:Button ID="categoriaButton"
                runat="server"
                Text="Categoria" />

            <asp:Button ID="fornecedorButton"
                runat="server"
                Text="Fornecedor" />

        </div>

        <asp:MultiView ID="MultiView1" runat="server">
            <asp:View ID="View1" runat="server"></asp:View>
            <asp:View ID="View2" runat="server"></asp:View>
        </asp:MultiView>

    </form>
</body>
```

12. Dentro da primeira View, coloque um Label e um DropDownList:

```
<asp:MultiView ID="MultiView1" runat="server">
    <asp:View ID="View1" runat="server">

        <asp:Label ID="categoriaLabel"
            runat="server"
            Text="Escolha uma categoria:">
        </asp:Label>

        <asp:DropDownList ID="categoriaDropDownList"
            AutoPostBack="true"
            runat="server">
        </asp:DropDownList>

    </asp:View>

    <asp:View ID="View2" runat="server"></asp:View>
</asp:MultiView>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

13. Depois da MultiView, insira uma GridView:

```
<form id="form1" runat="server">
    <h1>Northwind - Produtos </h1>

    <div>
        <span>Pesquisar por: </span>
    </div>

    <asp:MultiView ID="MultiView1" runat="server">
        <asp:View ...>
            <asp:Label ....>
            <asp:DropDownList ....>
        </asp:View>
        <asp:View ID="View2" runat="server"></asp:View>
    </asp:MultiView>

    <asp:GridView ID="produtosGridView"
        runat="server">
    </asp:GridView>

</form>
```

14. Executando neste ponto, aparecem apenas os botões, sem nenhuma formatação:



15. Crie o método do evento **click** do botão **Categoria**:

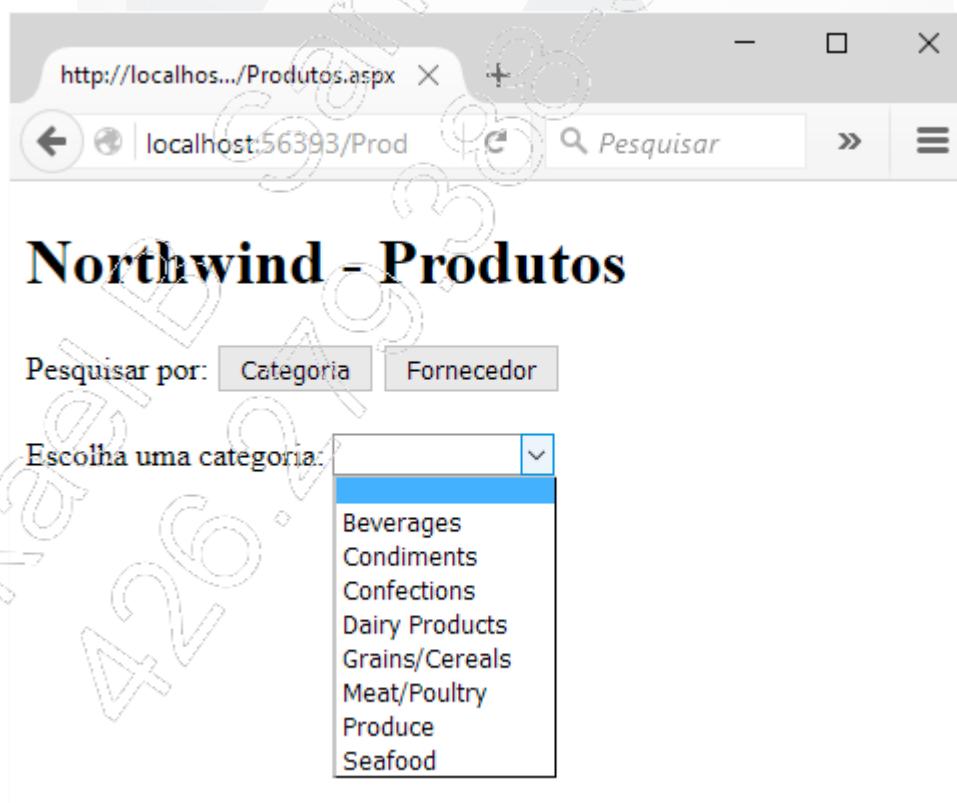
```
protected void categoriaButton_Click(...)
{
    var db = new ProdutosDb();

    categoriaDropDownList.DataValueField = "CategoryId";
    categoriaDropDownList.DataTextField = "CategoryName";
    categoriaDropDownList.DataSource = db.CategoriasLista();
    categoriaDropDownList.DataBind();

    categoriaDropDownList.Items.Insert(0, string.Empty);

    MultiView1.ActiveViewIndex = 0;
}
```

16. Teste o programa. Deve aparecer a lista de categorias com o primeiro item em branco, ao clicar no botão **Categoria**:



17. Crie o método do evento **SelectedIndexChanged** do DropDownList **Categoria**:

```
protected void CategoriaDropDownList_SelectedIndexChanged...
{
    if (categoriaDropDownList.SelectedIndex == 0)
    {
        produtosGridView.DataSource = null;
    }
    Else
    {
        int categoriaId=
Convert.ToInt32(categoriaDropDownList.SelectedValue);

        var db = new ProdutosDb();

        var tb = db.ProdutosPorCategoria(categoriaId);

        produtosGridView.DataSource = tb;
    }
    produtosGridView.DataBind();
}
```

18. Teste o programa. Agora deve ser possível ver a lista de produtos de uma categoria:

ProductName	UnitPrice	UnitsInStock
Chai	18,0000	39
Chang	19,0000	17
Guaraná Fantástica	4,5000	20
Sasquatch Ale	14,0000	111
Steeleye Stout	18,0000	20
Côte de Blaye	263,5000	17
Chartreuse verte	18,0000	69
Ipoh Coffee	46,0000	17
Laughing Lumberjack Lager	14,0000	52
Outback Lager	15,0000	15
Rhönbräu Klosterbier	7,7500	125
Lakkalikööri	18,0000	57

19. Vamos adicionar alguma formatação. Insira uma folha de estilo no projeto chamada **estilos.css**:

```
body
{
    margin:0px;
    padding:0px;
    font-family:tahoma;
}
```

```
h1
{
    padding:25px;
    background-color:#a3b0e9;
    margin-top:0px;
    color:#656f9d;
    letter-spacing:-2px;
}

section {
    padding:10px;
}

td {
    padding:6px;
}

th {
    padding:6px;
    background-color:#e5eaff;
    font-weight:normal;
}
```

20. Associe a folha de estilos à página arrastando e soltando do **Solution Explorer** para a página. Isso criar a referência no **head**:

```
<head runat="server">
    <title></title>

    <link href="Estilos.css" rel="stylesheet" type="text/css" />

</head>
```

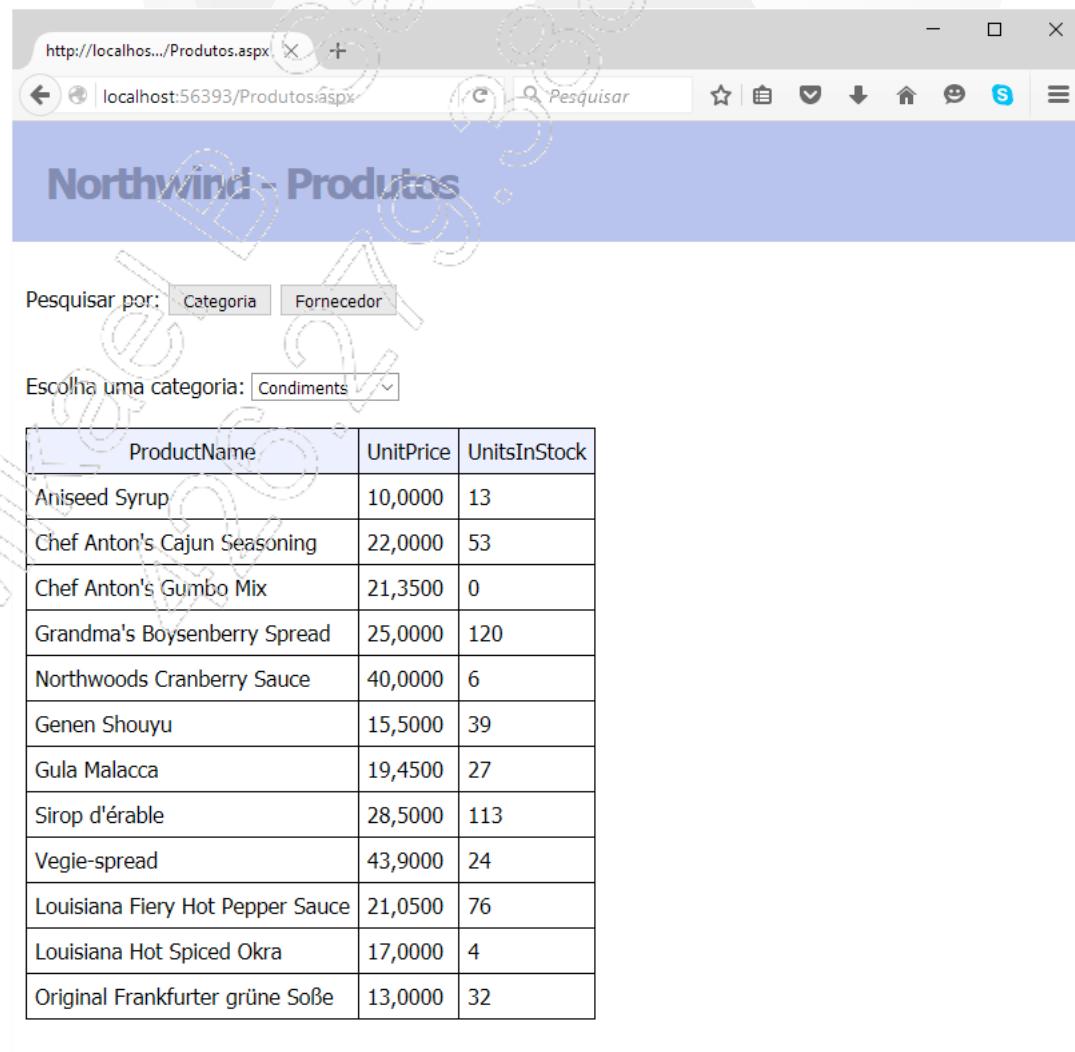
21. Na folha de estilos, foram definidos formatos para a tag **section**. A página tem três seções ou áreas: **Pesquisa**, **Escolha do Item** e o **GridView**. Coloque uma tag **Section** em cada uma delas, desta forma:

```
<section>
  <span>Pesquisar por: </span>
    <asp:Button ID="categoriaButton" ...>
    <asp:Button ID="fornecedorButton" ...>
</section>
```

```
<section>
  <asp:MultiView ID="MultiView1" ...>
    <asp:View ...>
    <asp:View ...>
  </asp:MultiView>
</section>
```

```
<section>
  <asp:GridView ...>
</section>
```

22. Teste o resultado:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

23. Note que houve uma pequena melhora, mas a Grid precisa de um formato melhor nas colunas **UnitPrice** e **UnitsInStock**. É preciso, também, traduzir esses títulos. Insira essas modificações na Grid para cancelar a geração automática de colunas e formatar as colunas corretamente:

```
<asp:GridView ID="produtosGridView"
    runat="server"
    AutoGenerateColumns="False">

    <Columns>

        <asp:BoundField HeaderText="Produto"
            DataField="ProductName" />

        <asp:BoundField HeaderText="Preço de Venda"
            DataField="UnitPrice"
            DataFormatString="{0:C}" >
            <ItemStyle HorizontalAlign="Right" />
            <HeaderStyle HorizontalAlign="Right" />
        </asp:BoundField>

        <asp:BoundField HeaderText="Unidades em Estoque"
            DataField="UnitsInStock"
            DataFormatString="{0:N0}" >
            <ItemStyle HorizontalAlign="Right" />
            <HeaderStyle HorizontalAlign="Right" />
        </asp:BoundField>

    </Columns>

</asp:GridView>
```

24. Execute e veja o resultado:

The screenshot shows a web browser window with the URL <http://localhost:56393/Produtos.aspx>. The title bar reads "Northwind - Produtos". Below the title, there are search and filter options: "Pesquisar por:" with dropdowns for "Categoria" and "Fornecedor", and a dropdown for "Escolha uma categoria" set to "Beverages". The main content is a table listing products from the Beverages category:

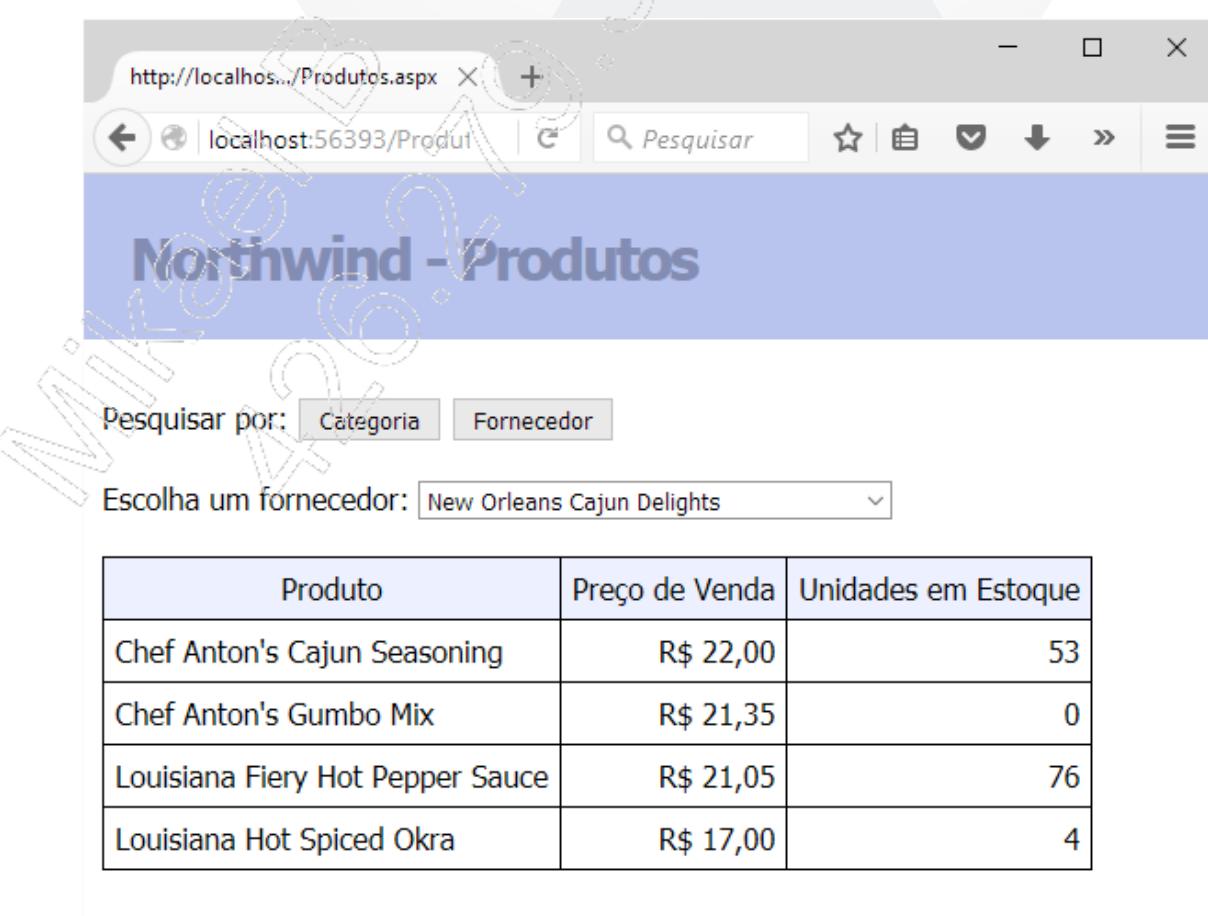
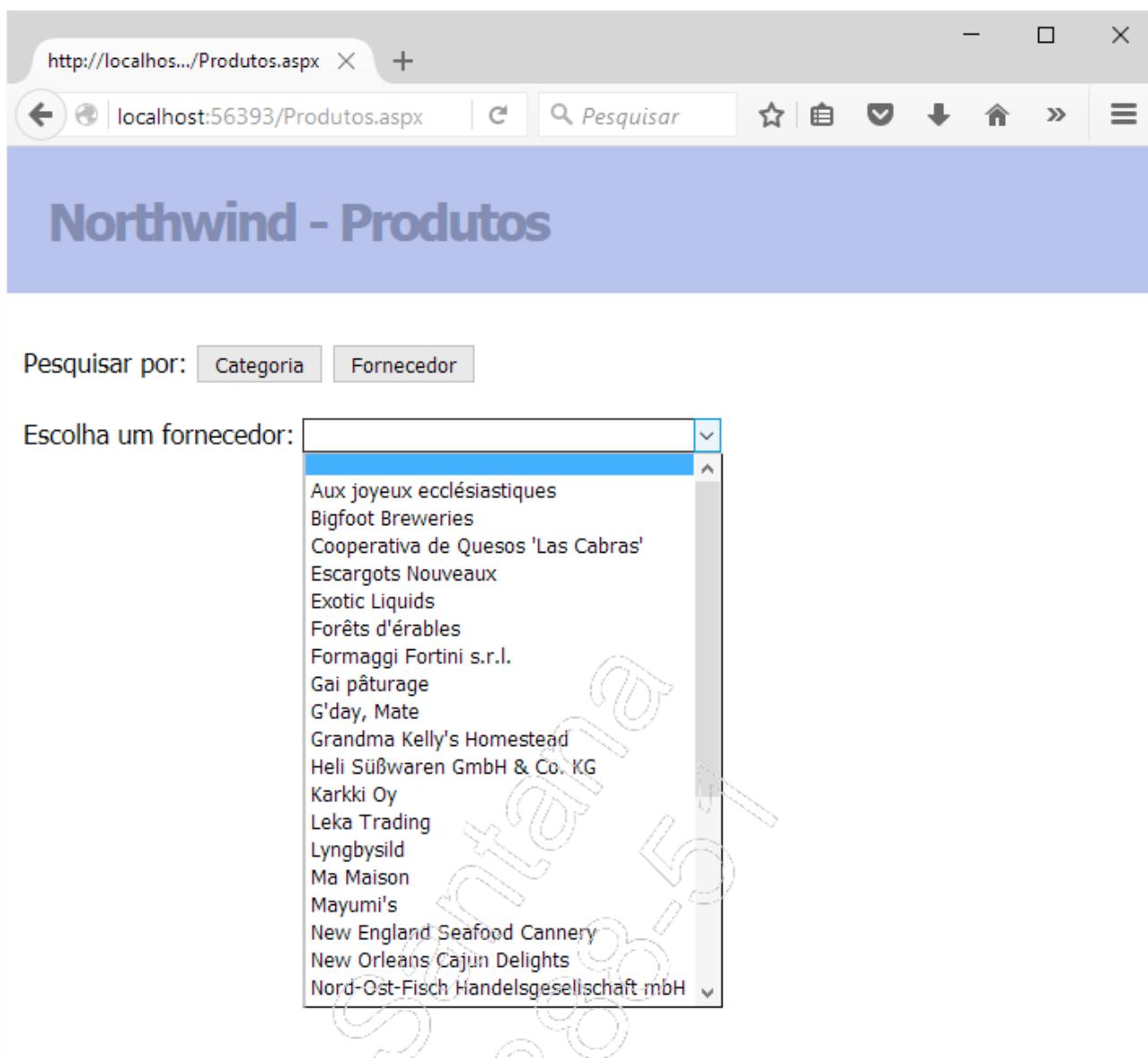
Produto	Preço de Venda	Unidades em Estoque
Chai	R\$ 18,00	39
Chang	R\$ 19,00	17
Guaraná Fantástica	R\$ 4,50	20
Sasquatch Ale	R\$ 14,00	111
Steeleye Stout	R\$ 18,00	20
Côte de Blaye	R\$ 263,50	17
Chartreuse verte	R\$ 18,00	69
Ipo Coffee	R\$ 46,00	17
Laughing Lumberjack Lager	R\$ 14,00	52
Outback Lager	R\$ 15,00	15
Rhönbräu Klosterbier	R\$ 7,75	125
Lakkalikööri	R\$ 18,00	57

25. Agora devemos terminar o programa fazendo a consulta de fornecedores. A tabela de fornecedores é a **Suppliers**. O código do fornecedor é **SupplierId** e o nome da empresa é **CompanyName**. Na tabela **Products**, o campo que define o fornecedor é **SupplierId**. Os seguintes passos deverão ser seguidos:

- Criar um método para obter a lista de fornecedores na classe **ProdutoDb**;
- Criar um método para obter a lista de produtos de um fornecedor;
- Criar na tela um **DropDownList** para exibir a lista de fornecedores;
- Criar o código para exibir na Grid os produtos de um fornecedor.

O resultado deve ser este, se o usuário escolher o botão **Fornecedor**:





3

ADO.NET com MVC

- ✓ Separação das responsabilidades;
- ✓ Retorno de métodos;
- ✓ Controllers e Views para exibir/alterar dados.

3.1. Introdução

As classes de acesso a dados do .NET Framework permitem conectar um banco de dados, obter informações sobre os dados gravados, além de criar, alterar e excluir esses dados. No capítulo anterior, foi visto como conectar um banco de dados SQL Server e como exibir dados na tela usando Web Forms e Web Controls. Neste capítulo, será visto como exibir dados na tela usando MVC.

3.2. Separação das responsabilidades

Um dos princípios mais importantes no desenvolvimento de aplicações usando linguagem orientada a objetos é a separação das responsabilidades de cada classe. De forma resumida, esse princípio diz que uma classe deve ter uma única razão para ser alterada, ou seja, uma classe deve ter uma única responsabilidade.

Por exemplo, se uma classe tem a responsabilidade de exibir dados na tela, obter informações do usuário, validar essas informações, gravar e ler dados de um banco de dados e enviar e-mails, a probabilidade de algo dar errado é muito maior do que se as tarefas fossem isoladas e divididas entre várias pequenas classes. Além disso, as funcionalidades isoladas (enviar e-mail, por exemplo) podem ser facilmente reaproveitadas por outras classes.

No caso da interface de usuário (Web Forms, MVC, Web Pages), o ideal é que a parte que exibe os dados para o usuário seja separada da parte do sistema em que os dados são definidos, criados ou validados. Por isso é sempre importante criar classes, isoladas da tela, que manipulam os dados que serão exibidos.

No modelo Web Forms, essa separação não é explícita. Cabe ao programador criar as classes corretamente para garantir um isolamento. No modelo MVC, porém, essa separação é parte estrutural.

O próprio nome MVC já diz como as funcionalidades são separadas: **Model**, **View** e **Controller**. **Model** são os dados que são apresentados; **View** é a parte visual; e **Controller** é quem recebe os eventos e manipula as chamadas.

3.3. Retorno de métodos

Os métodos criados para retornar dados de um repositório podem retornar três tipos de dados: **DbDataReader**, **DataSet** e **Coleções**. A seguir, um resumo de como decidir o melhor retorno.

3.3.1. **DbDataReader**

Acontece quando um método (ou function) retorna um fluxo de dados diretamente do servidor. Enquanto os dados são consumidos, a conexão deve ficar aberta. Apenas um registro é lido por vez, tornando o uso do **DbDataReader** excelente em termos de performance. Por outro lado, essa abordagem mantém a conexão com o servidor presa, limitando o número de usuários simultâneos. Veja um exemplo de um método que retorna um **DbDataReader**:

```
public class ProdutosDb
{
    public DbDataReader ProdutosListagem()
    {
        string sql = "Select ProductId, ProductName From Products";

        SqlDataReader reader;
        SqlConnection cn = new SqlConnection(NorthwindDb.Conexao);
        SqlCommand cmd = new SqlCommand(sql, cn);

        cn.Open();
        reader=cmd.ExecuteReader(CommandBehavior.CloseConnection);

        return reader;
    }
}
```

3.3.2. DataTable ou DataSet

Acontece quando um método (ou function) retorna um objeto do tipo **DataTable** ou **DataSet** preenchido. Os dados ficam todos na memória enquanto o objeto **DataTable** não for descartado pelo **Garbage Collector**. Os dados podem ser compartilhados por usuários em uma aplicação Web e podem ser mantidos em cache, aumentando muito a performance. Fica restrito à memória disponível do servidor. Outra característica é o fato de não ter propriedades fixas, facilitando a geração de queries dinâmicas, porém aumentando o risco em tempo de execução, já que as colunas não são conhecidas em tempo de compilação. Existem os DataSets tipados, mas foram completamente substituídos pelas classes genéricas que fornecem uma programação mais direta e limpa.

Veja um exemplo de um retorno de DataTable:

```
public DataTable ProdutosListagem()
{
    string sql = "Select ProductId, ProductName From Products";

    string conexao = NorthwindDb.Conexao;

    SqlDataAdapter da = new SqlDataAdapter(sql,conexao);

    DataTable tb = new DataTable();

    da.Fill(tb);

    return tb;
}
```

3.3.3. Coleções fortemente tipadas (arrays ou coleções genéricas)

Acontece quando um método (ou function) retorna uma coleção de objetos de dados definidos na modelagem da aplicação. Por exemplo, uma coleção de instâncias de uma classe chamada **Produto** ou **Cliente**. As coleções tipadas são facilmente manipuladas, principalmente pelo framework MVC, na construção de telas. Adiante, um exemplo de um método que retorna uma coleção de instâncias de uma classe chamada **Produto**:

```
public class Produto
{
    public int Id { get; set; }
    public string Nome { get; set; }
}

public List<Produto> ProdutosListagem()
{
    List<Produto> lista = new List<Produto>();

    string sql = "Select ProductId, ProductName From Products";

    using (var cn = new SqlConnection(NorthwindDb.Conexao))
    {
        var cmd = new SqlCommand(sql, cn);

        cn.Open();

        var reader= cmd.ExecuteReader();

        while (reader.Read())
        {
            var p = new Produto();
            p.Id = Convert.ToInt32(reader["ProductId"]);
            p.Nome = Convert.ToString(reader["ProductName"]);

            lista.Add(p);
        }
    }
    return lista;
}
```

3.4. Controllers e Views para exibir/alterar dados

No modelo MVC, as Views contêm o código HTML, que é renderizado usando o mecanismo Razor, e podem, opcionalmente, receber uma instância de uma classe com os dados que serão utilizados. As classes que armazenam as informações que serão exibidas na tela são chamadas de **Models** ou **View-Models**, visto que Models também são usados para definir as classes de negócios de uma aplicação.

Não é obrigatório existir uma separação entre **Models** e **View-Models**, mas é uma boa prática, porque isola as responsabilidades de maneira eficiente e clara.

Veja um exemplo de um Controller que retorna uma View que recebe como parâmetro uma instância da classe, sendo que o método **ProdutosListagem** retorna uma instância da classe genérica **List<T>**, neste caso preenchida com **List<Produto>**:

```
public class ProdutoController : Controller
{
    // GET: Produto
    public ActionResult Index()
    {
        var produtosDb = new ProdutosDb();
        var lista = produtosDb.ProdutosListagem();

        return View(lista);
    }
}
```

A View deve declarar que recebe essa instância da classe, usando a diretiva de compilação **@model**, conforme o exemplo a seguir:

```
@model List<Cap03_Lab01.Models.Produto>
```

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

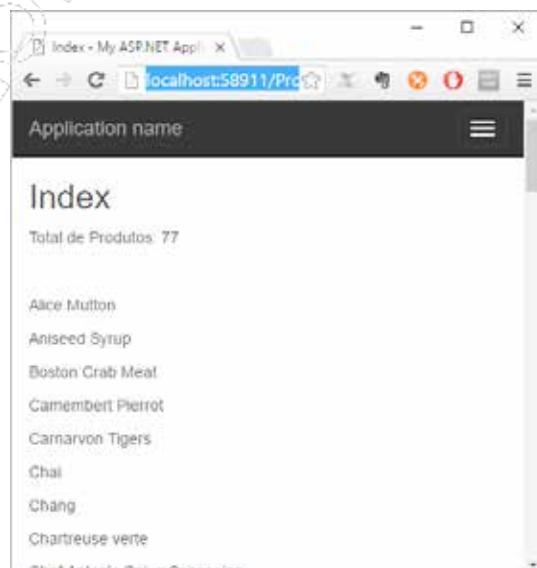
Uma vez declarado o tipo da instância que é recebido pela View, é possível interagir com ela a partir da propriedade Model e fazendo uso do intellisense:

```
@model List<Cap03_Lab01.Models.Produto>
```

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

```
Total de Produtos: @Model.Count
```

```
@foreach (var produto in Model)  
{  
    <p>@produto.Nome</p>  
}
```



Visual Studio 2015 - ASP.NET com C# Acesso a dados

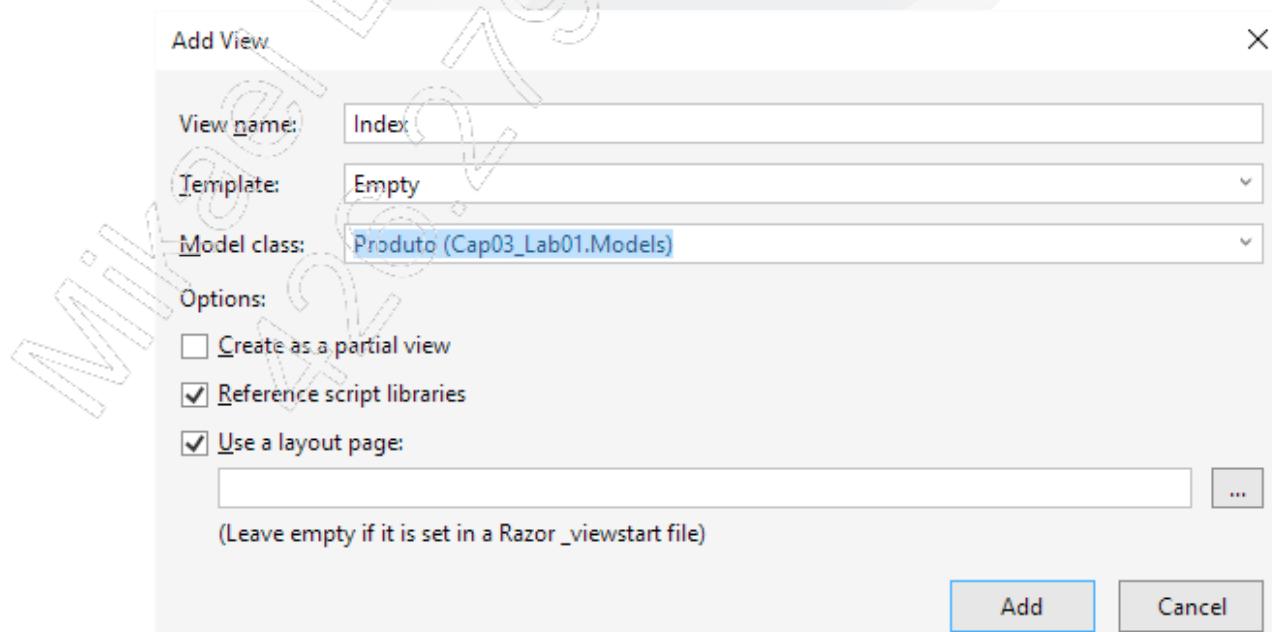
Um recurso muito interessante do framework MVC é a coleção de modelos prontos para tratamentos de dados. Esses modelos podem ser Controllers, Views e até aplicativos completos usando dados dinâmicos.

No caso das Views, operações com banco de dados como **Incluir**, **Alterar**, **Excluir**, **Exibir Registro** ou **Lista** podem ser geradas automaticamente, a partir de qualquer método que retorne um **ActionResult** e que receba uma instância de uma classe **Model**. Para isso, dentro de um método de controle, usando o menu de contexto, existe o comando **Add View**.

```
public class ProdutoController : Controller
{
    // GET: Produto
    public ActionResult Index()
    {
        var pro
        var lis
        return
    }
}
```



Esse comando permite usar um assistente para criar uma View que recebe uma instância de uma classe de modelo e que prepara a tela para executar tarefas comuns de tratamento de dados:



- **View name:** O nome da View. Geralmente o mesmo nome do método;
- **Template:** Pode ser vazio (**Empty**), novo (**Create**), alteração (**Update**), exclusão (**Delete**), exibir registro (**Details**) e exibir listagem (**List**). Cada opção gera diferentes telas para operações com dados;
- **Model class:** A classe que é o modelo de dados para exibição (**View-Model**). Esta classe deve conter propriedades para tudo o que for exibido na tela;
- **Options:**
 - **Create as a partial view:** Se marcada, esta opção gera uma View sem nenhum vínculo com outros arquivos, pois será usada para ser inserida em outra View;
 - **Reference script libraries:** Se marcada e o modelo escolhido for de alteração ou inclusão, algumas referências a bibliotecas JavaScript de validação (jQuery) são adicionadas. Este é o padrão e normalmente não é alterado. Deve ser desmarcada apenas se foram implementadas soluções alternativas e personalizadas de tratamento de formulários;
 - **Use a layout page:** Se marcada, a View usará uma página de modelo (Layout) a ser informada na caixa de texto, ou a página de modelo padrão, se a caixa de texto for deixada em branco. Se a View não vai ser baseada em nenhum modelo, esta opção deve ser desmarcada.

3.4.1. Listagem

A View gerada pelo assistente para listagem de dados é a seguinte:

```
@model IEnumerable<Cap03_Lab01.Models.Produto>

@{
    ViewBag.Title = "Listagem";
}



## Listagem



@Html.ActionLink("Create New", "Create")



| @Html.DisplayNameFor(model => model.Nome)                                                                             |  |
|-----------------------------------------------------------------------------------------------------------------------|--|
| @Html.DisplayFor(modelItem => item.Nome)<br><a href="#">Edit</a><br><a href="#">Details</a><br><a href="#">Delete</a> |  |


```

Vamos analisar os itens gerados pelo assistente:

1. Diretiva de compilação:

```
@model IEnumerable<Cap03_Lab01.Models.Produto>
```

Esta linha declara que a View receberá um coleção de instâncias da classe **Produto**. A interface **IEnumerable<T>** define que qualquer tipo de classe que implemente um enumerator, ou seja, qualquer tipo que forneça suporte a listas, poderá ser usado. Isso inclui **Array**, **List<T>**, **ArrayList**, entre outros.

2. Link para novo registro:

```
@Html.ActionLink("Create New", "Create")
```

Esta linha usa a classe Helper **Html** e o método de extensão **ActionLink** para criar uma âncora HTML apontando para o controlador atual e o método **Create**. Existem dez sobrecargas para este método; a utilizada aqui foi a seguinte:

```
@Html.ActionLink("Texto do Link", "Nome do Método")
```

O resultado HTML é o seguinte:

```
<a href="/Produto/Create">Create New</a>
```

3. Exibir a legenda da propriedade:

```
@Html.DisplayNameFor(model => model.Nome)
```

O método **DisplayNameFor** obtém a legenda da propriedade. Se não houver metadados associados à propriedade, este método retorna o nome da propriedade. Metadados são acrescentados às propriedades por meio de código ou adicionando atributos especiais (annotations).

A sintaxe utilizada neste caso é a seguinte:

```
DisplayNameFor (Expression<Func<TModel, TValue>> expression);
```

A expressão que deve ser passada é uma função que recebe como parâmetro uma instância da classe do modelo (se for uma lista, é do tipo da lista) e retorna o conteúdo de um campo qualquer. A expressão lambda utilizada é a seguinte:

```
model=>model.Nome
```

Essa expressão é o equivalente a um método igual a este exemplo:

```
string MeuMetodo( Produto p)
{
    return p.Nome;
}
```

O retorno (**string**) é consequência do tipo retornado. Como é um método anônimo, não há necessidade de indicar o nome (**MeuMetodo**). O parâmetro é representado pela palavra **model** (poderia ser qualquer outra palavra, é o nome de uma variável) e a propriedade é definida por **model.Nome**. Com essas informações (a propriedade retornada e o nome da propriedade), o framework MVC extrai a string que deve ser exibida, considerando, inclusive, metadados associados.

4. Percorrer os elementos da coleção:

```
@foreach (var item in Model) {
```

Esta linha define o início do loop para percorrer os dados do modelo (é um **IEnumerable**, portanto pode ser utilizado na lista). O nome da variável **item** é livre; pode ser usado qualquer nome.

5. Exibir dados de um item da coleção:

```
@Html.DisplayFor(modelItem => item.Nome)
```

O método **DisplayFor** é equivalente a **DisplayNameFor**, porém, o que é retornado é o conteúdo do campo identificado. Outra diferença é que o retorno que deve ser uma propriedade da variável local (**item**). O nome **modelItem** pode ser qualquer nome; não é utilizado diretamente.

6. Links para visualização, edição e exclusão:

Para cada linha da listagem, três hiperlinks são criados, apontando, cada um deles para uma página de visualização (**details**), de alteração (**update**) e de exclusão (**delete**).

```
@Html.ActionLink("Edit", "Edit", new { id=item.Id })  
@Html.ActionLink("Details", "Details", new { id=item.Id }) |  
@Html.ActionLink("Delete", "Delete", new { id=item.Id })
```

A novidade aqui é o terceiro parâmetro. Esses links utilizam a seguinte sintaxe:

```
ActionLink(string texto, string metodo, object valores);
```

O terceiro parâmetro é uma classe anônima contendo campos que serão inseridos na URL correspondente ao Controller e ao método para o qual o link aponta. Por exemplo, considere o seguinte link:

```
@Html.ActionLink("Delete", "Delete", new { id=item.Id })
```

Esse link aponta para a seguinte URL, supondo que o **Id** do produto nesta linha seja **2**:

<http://servidor/Produto/Delete/2>

Se o link fosse este...

```
@Html.ActionLink("Delete",  
"Delete", new { id=item.Id, Nome=item.Nome })
```

...a URL gerada seria esta (supondo que o nome do produto **2** fosse **caneta**):

<http://servidor/Produto/Delete/2/Caneta>

3.4.2. Inclusão

Seguindo os mesmos passos da listagem, ao gerar uma View com o modelo **Create**, a seguinte tela será gerada:

```
@model Cap03_Lab01.Models.Produto

<h2>Novo</h2>

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

    <div class="form-horizontal">
        <h4>Produto</h4>
        <hr />

        @Html.ValidationSummary(true, "", new { @class = "text-danger" })

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

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

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

<div>@Html.ActionLink("Back to List", "Index")</div>

<script src "~/Scripts/jquery-1.10.2.min.js"></script>
<script src "~/Scripts/jquery.validate.min.js"></script>
<script src "~/Scripts/jquery.validate.unobtrusive.min.js"></script>
```

Vamos analisar os itens gerados pelo assistente:

1. Diretiva de compilação:

```
@model Cap03_Lab01.Models.Produto
```

Esta linha declara um único elemento, e não uma coleção. Esta instância da classe **Produto** será gerada quando o formulário for enviado.

2. Início do formulário:

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

Esta linha declara o início de um form HTML que realizará um POST para o método de mesmo nome. O código HTML gerado é o seguinte:

```
<form action="/Produto/Novo" method="post">
```

3. Token de validação:

```
@Html.AntiForgeryToken()
```

Esta linha utiliza o método **AntiForgeryToken** para gerar um campo oculto com informações que serão verificadas quando for feito um POST. Isso garante que o POST foi feito desta página, evitando, assim, ataques e tentativas de executar códigos de outros sites.

4. Resumo da validação:

```
@Html.ValidationSummary(true, "", new { @class = "text-danger" })
```

O método **ValidationSummary** cria uma lista que é exibida quando existem erros de validação. A sintaxe utilizada é a seguinte:

```
MvcHtmlString ValidationSummary(  
    bool excludePropertyErrors,  
    string message,  
    object htmlAttributes);
```

- O parâmetro **excludePropertyErrors** é um booleano que define se os erros específicos de propriedades das classes serão exibidos. Geralmente está definido para **true** se os erros de cada campo estão sendo exibidos em outro lugar;
- O parâmetro **Message** é a mensagem padrão; normalmente é vazio;
- O parâmetro **htmlAttributes** é uma classe anônima com campos e valores que representam atributos a serem incluídos na tag HTML gerada. No exemplo anterior, o atributo **class="text-danger"** será aplicado na tag gerada. É necessário usar **@** no atributo **class** porque é uma palavra reservada; em qualquer outro atributo isso não é necessário.

Veja o HTML gerado:

```
<div class="text-danger">
  <ul>
    <li>Digite o Nome </li>
  </ul>
</div>
```

5. Legenda para o campo Nome:

```
@Html.LabelFor(model => model.Nome,
    htmlAttributes: new { @class = "control-label col-md-2" })
```

O método **LabelFor** cria uma legenda. A sintaxe utilizada aqui é esta:

```
MvcHtmlString LabelFor(
    Expression<Func<TModel, TValue>> expression,
    object htmlAttributes);
```

O parâmetro **expression** (expressão lambda representando um método que retorna o valor de uma campo) define a propriedade, e a classe anônima para **htmlAttributes** define os argumentos inseridos na tag HTML.

O código HTML gerado é este:

```
<label class="control-label col-md-2" for="Nome">  
    Nome  
</label>
```

6. Input para o nome:

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

O método **EditorFor** gera diversos tipos de código HTML, dependendo do tipo de campo. Por exemplo, se o campo for do tipo booleano, é gerado um **checkbox**, e se for texto, um campo **input**. A sintaxe utilizada neste exemplo é a seguinte:

```
EditorFor<TModel, TValue>(  
    Expression<Func<TModel, TValue>> expression,  
    object additionalViewData);
```

- **Expression** é uma expressão lambda representando um método para identificar o campo;
- **additionalViewData** é um objeto que contém elementos a serem adicionados ao objeto gerado. Esse campo é diferente do usado no **LabelFor** porque o **EditorFor** não cria apenas um único tipo de controle HTML; o **EditorFor** fornece informações para o template dos controles que ele gera. Pode ser que algum controle tenha outros atributos além de **htmlAttributes** e por isso esse parâmetro é uma coleção de objetos, e não apenas **htmlAttributes**. Essa coleção de dados que os templates recebem ficam em uma coleção chamada **ViewData**. Internamente, os modelos usados pelo **EditorFor** utilizam essas informações para gerar o HTML. Por exemplo, em um controle personalizado que utilize uma informação chamada **versão**, a sintaxe seria esta:

```
@Html.EditorFor(model => model.Nome,  
    new {  
        htmlAttributes = new { @class = "form-control" },  
        versao = "1.0" })
```

7. Mensagem de validação de campo:

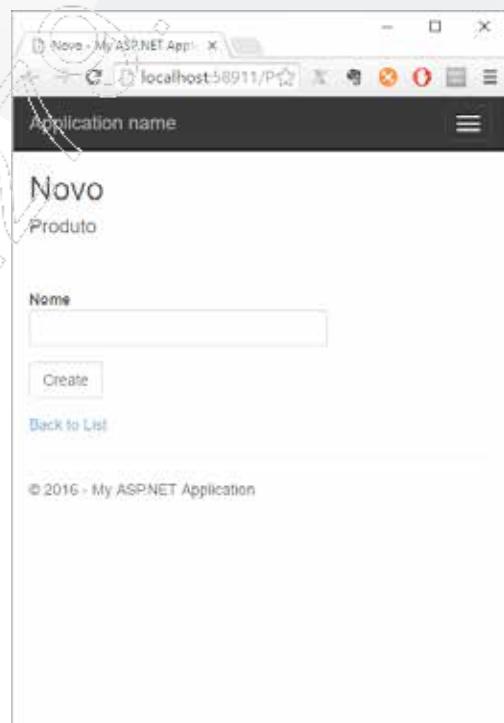
```
@Html.ValidationMessageFor(model => model.Nome, "",  
    new { @class = "text-danger" })
```

O método **ValidationMessageFor** define um lugar para exibir mensagens de validação de campo. Algumas mensagens são automáticas (por exemplo, um campo numérico com valor inválido). Outras são partes das regras de negócio e são definidas por meio de atributos ou programação.

A sintaxe utilizada no exemplo é esta:

```
ValidationMessageFor(Expression<Func<TModel, TProperty>> expression,  
    string validationMessage, object htmlAttributes);
```

- **Expression**, como nos exemplos anteriores, define a propriedade;
- O parâmetro **validationMessage** define a mensagem, porém, é comum deixar como uma string vazia e definir via código;
- O parâmetro **htmlAttributes** é uma classe anônima para definir os atributos HTML que serão adicionados ao código gerado.



3.4.3. Inclusão – POST

Ao clicar no botão de envio, o formulário é enviado para o servidor usando o verbo POST. No Controller, para receber este valor, é necessário usar o atributo **HttpPost** no método e receber como parâmetro um elemento definido no Model:

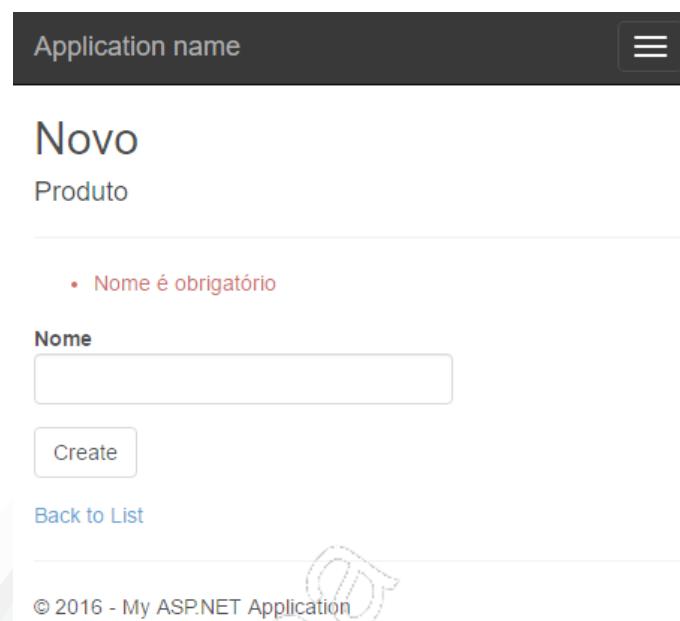
```
[HttpPost]
public ActionResult Novo(Produto p)
{
    if (string.IsNullOrEmpty(p.Nome))
    {
        ModelState.AddModelError("", "Nome é obrigatório");
    }
    else
    {
        var db = new ProdutosDb();
        db.ProdutoIncluir(p);
    }
    return View(p);
}
```

Na classe de acesso a dados, um método deve incluir um produto no banco de dados:

```
public void ProdutoIncluir(Produto p)
{
    string sql="INSERT INTO Products(ProductName) Values(@Nome)";

    using (var cn = new SqlConnection(NorthwindDb.Conexao))
    {
        var cmd = new SqlCommand(sql, cn);
        cmd.Parameters.AddWithValue("@Nome", p.Nome);
        cn.Open();
        cmd.ExecuteNonQuery();
    }
}
```

A propriedade **ModelState** permite incluir erros de validação que podem ser exibidos pelo **ValidationSummary** criado na View:



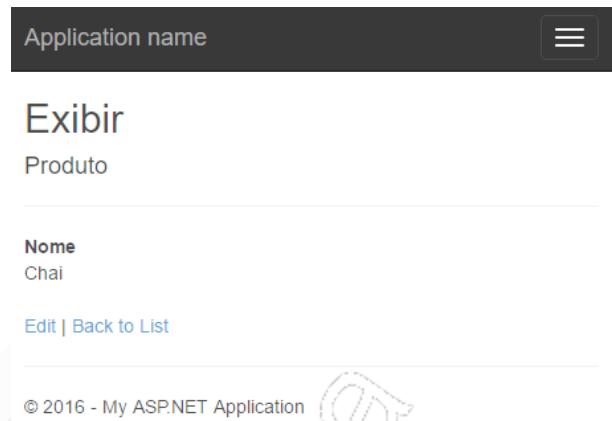
3.4.4. Exibição de um registro (Details)

A View gerada pelo assistente para exibir registros utiliza apenas os métodos **DisplayNameFor** (que exibe a legenda) e **DisplayFor** (que exibe o conteúdo).

```
@model Cap03_Lab01.Models.Produto  
  
{@  
    ViewBag.Title = "Exibir";  
}  
  
<h2>Exibir</h2>  
  
<div>  
    <h4>Produto</h4>  
    <hr />  
    <dl class="dl-horizontal">  
        <dt>  
            @Html.DisplayNameFor(model => model.Nome)  
        </dt>
```

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

</dl>
</div>
```



O método do Controller que chama esta View deve obter o produto da seguinte forma:

- A URL para acesso deve ser **produto/Exibir/1**, em que **1** é o **Id** do produto:

```
public ActionResult Exibir(int id)
{
    var db = new ProdutosDb();
    var p = db.ProdutoObter(id);

    return View(p);
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

- O método da classe de acesso a dados deve retornar um produto:

```
public Produto ProdutoObter(int id)
{
    string sql = @"Select ProductId, ProductName
                    From Products Where ProductId=@Id";

    Produto p = null;

    using (var cn = new SqlConnection(NorthwindDb.Conexao))
    {
        using (var cmd = new SqlCommand(sql, cn))
        {
            cmd.Parameters.AddWithValue("@Id", id);
            cn.Open();

            using (var dr = cmd.ExecuteReader())
            {
                if (dr.Read())
                {
                    p = new Produto();
                    p.Id = Convert.ToInt32(dr["ProductId"]);
                    p.Nome = Convert.ToString(dr["ProductName"]);
                }
            }
        }
    }

    return p;
}
```

3.4.5. Alteração de um registro (Update)

A View gerada pelo assistente para alterar registros é idêntica à View de inclusão. A única diferença é que o campo como Id do produto é inserido como um campo oculto. A listagem adiante foi simplificada. A única parte diferente está marcada:

```
@model Cap03_Lab01.Models.Produto

<h2>ProdutoAlterar</h2>

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

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

        @Html.HiddenFor(model => model.Id)

        <div class="form-group">
            @Html.LabelFor(model => model.Nome ...)
            <div class="col-md-10">
                @Html.EditorFor(model => model.Nome ...)
                @Html.ValidationMessageFor(model => model.Nome, ...)
            </div>
        </div>

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

<div>    @Html.ActionLink("Back to List", "Index")</div>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

O método do Controller deve obter o resultado do POST e atualizar os dados:

```
[HttpPost]
public ActionResult ProdutoAlterar(Produto p)
{
    var db = new ProdutosDb();
    db.ProdutoAlterar(p);
    return View(p);
}
```

O método da classe de acesso a dados atualiza a informação no servidor, localizando o registro pelo ID:

```
public void ProdutoAlterar(Produto p)
{
    string sql = @"UPDATE Products
        SET ProductName=@Nome
        WHERE ProductId=@Id";
    using (var cn = new SqlConnection(NorthwindDb.Conexao))
    {
        var cmd = new SqlCommand(sql, cn);
        cmd.Parameters.AddWithValue("@Nome", p.Nome);
        cmd.Parameters.AddWithValue("@Id", p.Id);
        cn.Open();
        cmd.ExecuteNonQuery();
    }
}
```

3.4.6. Exclusão de um registro (Delete)

A View gerada pelo assistente para excluir registro exibe os dados em primeiro lugar e depois envia um POST para confirmar:

```
@model Cap03_Lab01.Models.Produto

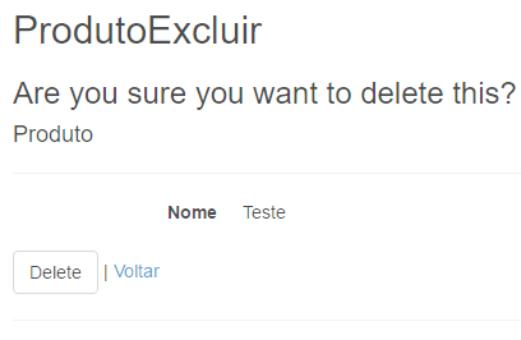
@{
    ViewBag.Title = "ProdutoExcluir";
}

<h2>ProdutoExcluir</h2>

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

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

        <div class="form-actions no-color">
            <input type="submit" value="Delete"
                class="btn btn-default" /> |
            @Html.ActionLink("Back to List", "Index")
        </div>
    }
</div>
```



Visual Studio 2015 - ASP.NET com C# Acesso a dados

O método do Controller deve obter o resultado do POST e excluir o registro:

```
[HttpPost]
public ActionResult Excluir(int id, FormCollection form)
{
    var db = new ProdutosDb();
    db.ProdutoExcluir(id);
    return View("Listagem", db.ProdutosListagem());
}
```

O método da classe de acesso a dados exclui o registro no servidor, localizando-o pelo ID:

```
public void ProdutoExcluir(int id)
{
    string sql = @"DELETE Products
                    WHERE ProductId=@Id";
    using (var cn = new SqlConnection(NorthwindDb.Conexao))
    {
        var cmd = new SqlCommand(sql, cn);

        cmd.Parameters.AddWithValue("@Id", id);
        cn.Open();
        cmd.ExecuteNonQuery();
    }
}
```

Pontos principais

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

- O retorno de informações de um banco de dados pode ser feito por meio de **DBDataReader**, **DataSet** ou conjunto de objetos;
- O framework **MVC** possui funcionalidades construídas para serem usadas com coleção de objetos, sendo, portanto, esse tipo de retorno o preferido de um método de acesso a dados;
- As Views tipadas disponibilizam uma instância da classe definida da diretriz de compilação em uma propriedade chamada **Model**;
- A classe **Html** disponibiliza métodos para exibir e coletar dados na tela. Os principais são **DisplayFor**, **DisplayNameFor** e **EditorFor**;
- A propriedade **ModelState** da classe **Controller** permite definir erros em tempo de execução;
- Os erros de modelo de dados são exibidos usando os métodos **ValidationSummary** e **ValidationMessageFor** da classe **Html**;
- O método **ActionLink** é usado para criar hiperlinks dentro do framework MCV;
- Por meio do menu de contexto de um método em uma classe Controller é possível usar um assistente para criar Views com as operações básicas de banco de dados.

3

ADO.NET com MVC

Teste seus conhecimentos

Mikael Santana
426.279.57



IMPACTA
EDITORA

1. Qual tipo de retorno de método de acesso a dados é o melhor para criar Views no MVC?

- a) DbDataReader
- b) DataSet
- c) Coleção de objetos
- d) DbConnection
- e) String

2. Qual método da classe Helper Html pode ser utilizado para exibir a legenda de um campo?

- a) Editor
- b) EditorFor
- c) DisplayNameFor
- d) DisplayFor
- e) LegendFor

3. Qual método adiante é utilizado para obter informações digitadas pelo usuário por meio de um formulário?

- a) EditorFor
- b) DisplayFor
- c) DisplayNameFor
- d) ValidationFor
- e) UserFor

4. Qual propriedade da classe Controller permite criar erros de validação?

- a) ErrorController
- b) ModelState
- c) ValidationSummary
- d) ModelValidation
- e) StateModel

5. Qual mecanismo é utilizado para renderizar as Views no MVC?

- a) html
- b) CSS
- c) Razor
- d) view-model
- e) ado.NET

3

ADO.NET com MVC

Mãos à obra!

Mikael B. Santana
426.279.3557



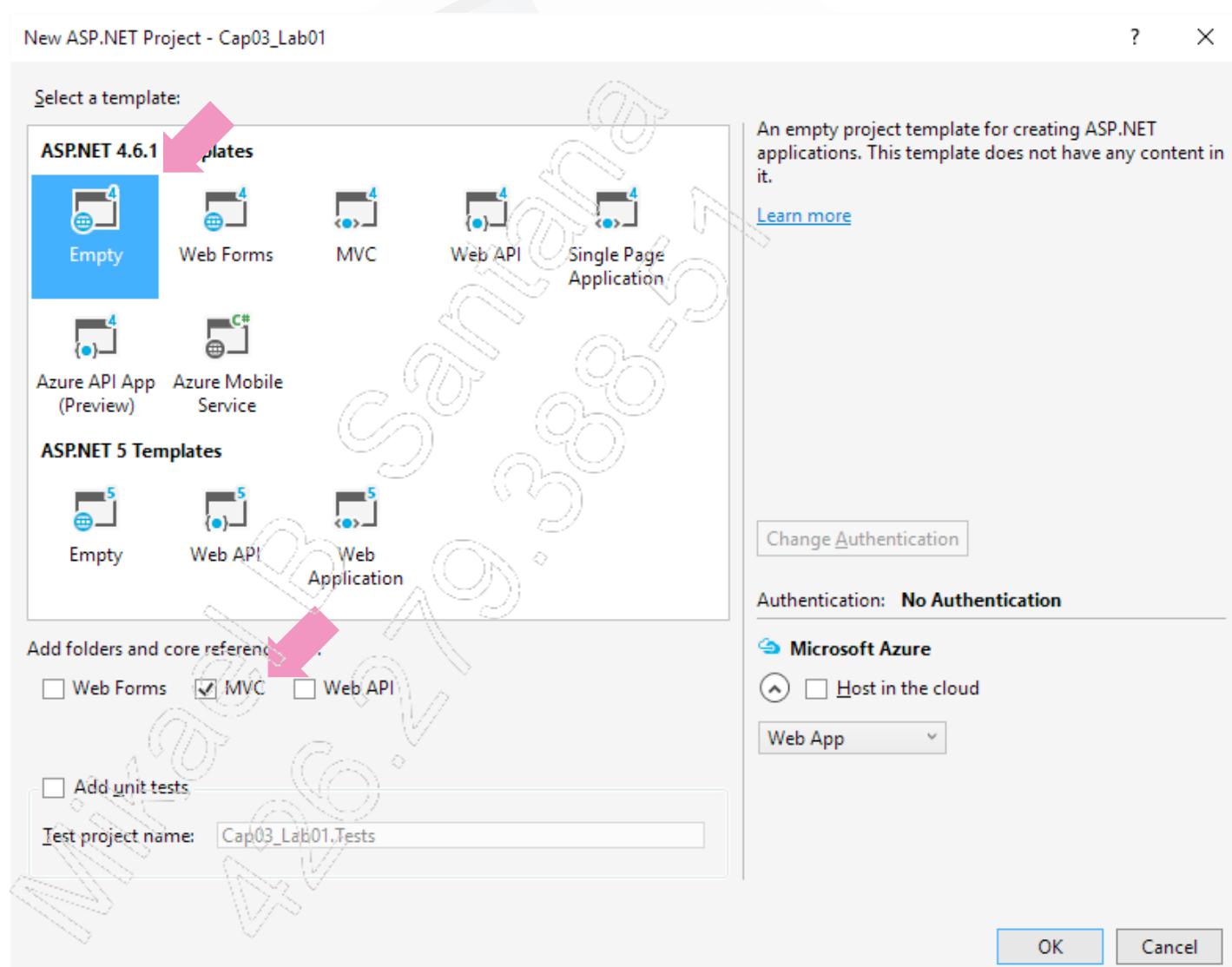
IMPACTA
EDITORA

Laboratório 1

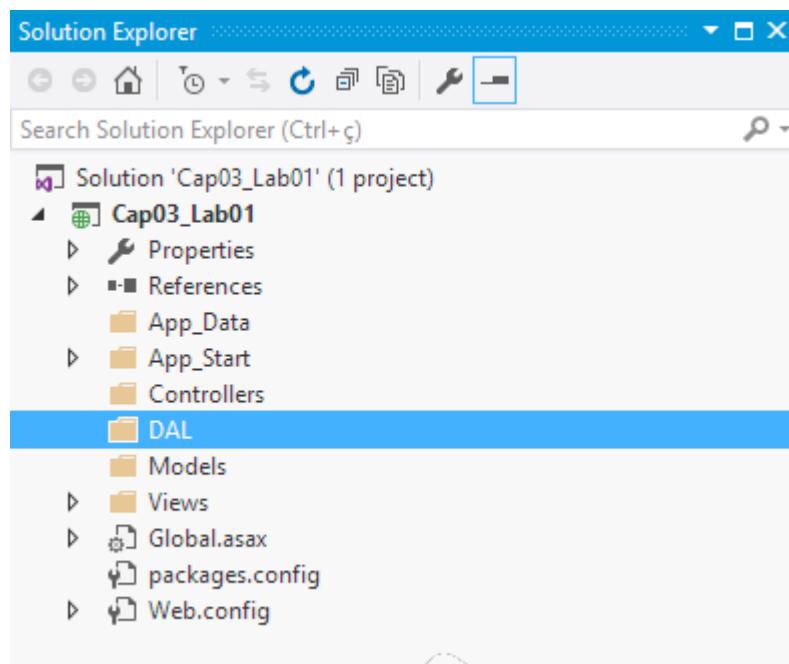
A – Criando uma página que permita consultar os clientes da empresa Northwind Traders por país

Neste laboratório, usaremos o ADO.NET para extrair informações, bem como usaremos classes de acesso a dados separadas da interface.

1. Crie um novo projeto Web Empty, com referências à biblioteca MVC, chamado **Cap03_Lab01**;



2. Adicione uma pasta chamada **DAL**:



3. Adicione, dentro da pasta **DAL**, uma classe chamada **NorthwindDb**:

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

namespace Cap03_Lab01.DAL
{
    public class NorthwindDb
    {
    }
}
```

4. Na classe **NorthwindDb**, crie uma propriedade privada para armazenar a string de conexão;

```
namespace Cap03_Lab01.DAL
{
    public class NorthwindDb
    {

        private string conexao =
            @"Data Source=(LocalDb)\MSSqlLocalDb;
Initial Catalog=Northwind;
Integrated Security=True";
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

5. Crie o método para retornar uma lista de países:

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

using System.Web;using System.Data.SqlClient;

namespace Cap03_Lab01.DAL
{
    public class NorthwindDb
    {
        private string conexao = @"Data...";

        public List<string> ClientePaisLista()
        {
            string sql=@"Select Country From Customers
                        GROUP BY Country";
            var lista = new List<string>();
            using (var cn = new SqlConnection(conexao))
            {
                using (var cmd = new SqlCommand(sql, cn))
                {
                    cn.Open();
                    using (var dr = cmd.ExecuteReader())
                    {
                        while (dr.Read())
                        {
                            lista.Add(dr[0].ToString());
                        }
                    }
                }
                return lista;
            }
        }
    }
}
```

6. Dentro da pasta **Models**, crie a classe **Cliente**:

```
using System.Linq;
using System.Web;

namespace Cap03_Lab01.Models
{
    public class Cliente
    {
        public string Id { get; set; }
        public string Nome { get; set; }
        public string Telefone { get; set; }
        public string Cidade { get; set; }
    }
}
```

7. Na classe **NorthwindDb**, escreva o método que retorna a lista de clientes de um país:

```
...
using Cap03_Lab01.Models;

..

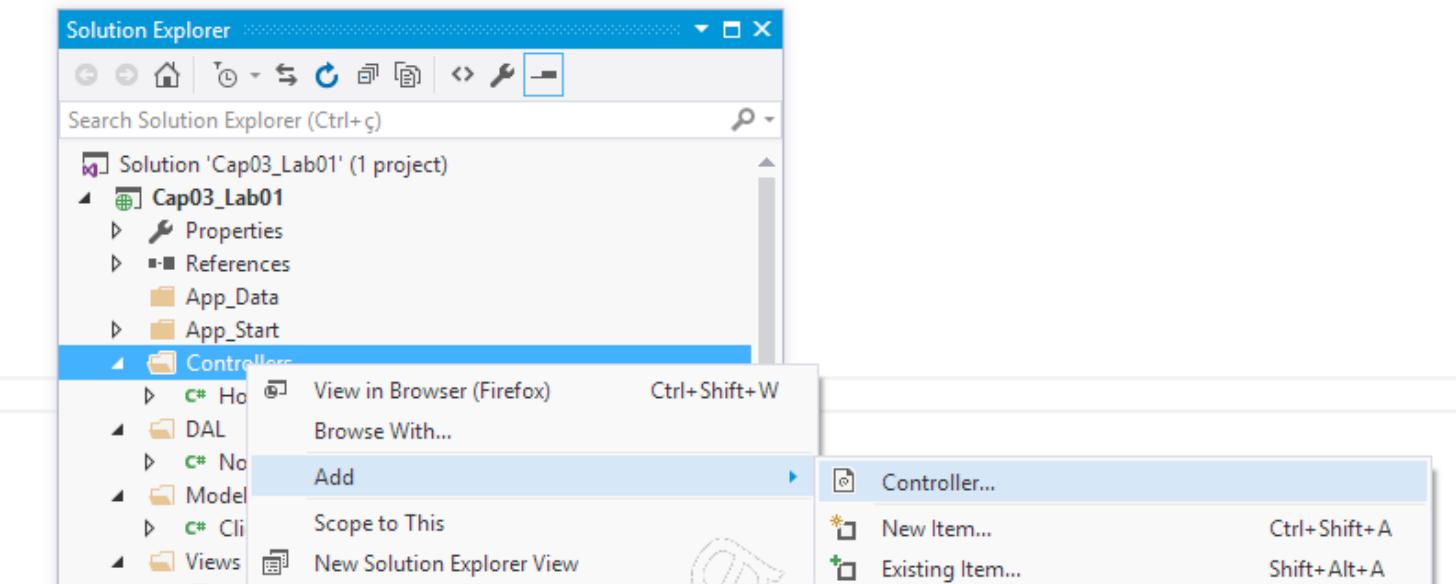
public List<Cliente> ClientePorPaisLista(string pais)
{
    string sql =
        @"SELECT CustomerId, CompanyName, Phone, City
        FROM Customers
        WHERE Country=@Pais
        ORDER By City, CompanyName";

    var lista = new List<Cliente>();
    using (var cn = new SqlConnection(conexao))
    {
        using (var cmd = new SqlCommand(sql, cn))
        {
            cn.Open();
            using (var dr = cmd.ExecuteReader())
            {
                while (dr.Read())
                {
                    var cliente = new Cliente();
                    cliente.Id = dr["CustomerId"].ToString();
                    cliente.Nome = dr["CompanyName"].ToString();
                    cliente.Telefone= dr["Phone"].ToString();
                    cliente.Cidade= dr["City"].ToString();

                    lista.Add(cliente);
                }
            }
        }
    }
    return lista;
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

8. Na pasta **Controllers**, crie o controller **HomeController** usando o menu de contexto:



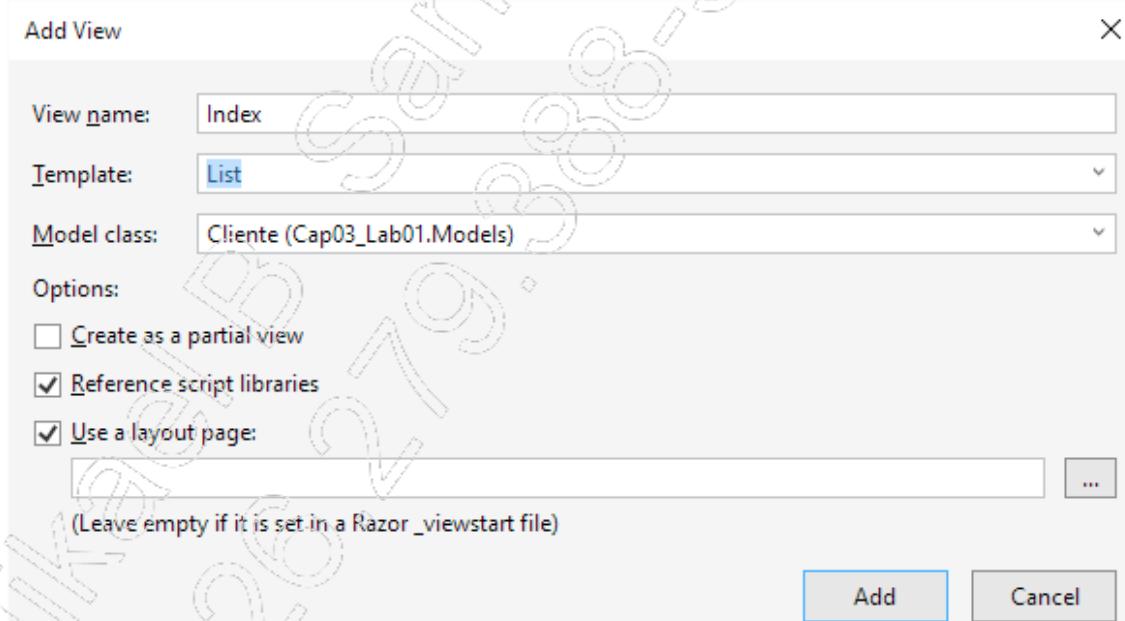
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Cap03_Lab01.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

9. Adicione a lista de países no objeto **ViewBag**. Isso servirá para exibir a lista de países para o usuário escolher:

```
public class HomeController : Controller
{
    // GET: Home
    public ActionResult Index()
    {
        var db = new DAL.NorthwindDb();
        var paisesLista = db.ClientePaisLista();
        var selectPaisesLista = new SelectList(paisesLista);
        ViewBag.PaisesLista = selectPaisesLista;
        return View();
    }
}
```

10. Adicione uma View a este método usando o menu de contexto;



11. Altere a View para exibir a lista de países e exibir os clientes apenas se houver dados:

```
@model IEnumerable<Cap03_Lab01.Models.Cliente>
{@{    ViewBag.Title = "Index";  }

<h2>Clientes por País</h2>
<hr/>

@using (Html.BeginForm())
{
    <div class="form-group">
        @Html.Label("País", "Escolha um país:", new { @class = "form-label" });
        @Html.DropDownList("País", (SelectList)ViewBag.PaisesLista,
                           new { @class = "form-control" })
    </div>

    <div class="form-group">
        <input type="submit" value="Confirmar" class="btn btn-default" />
    </div>
}

@if (Model != null)
{
    <table class="table">
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Nome)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Telefone)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Cidade)
            </th>
        </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Nome)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Telefone)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Cidade)
            </td>
        </tr>
    }
}
</table>
}
```

12. Inclua o método **Index** com POST, que recebe os dados do formulário e retorna a lista de países:

```
public class HomeController : Controller
{
    // GET: Home
    public ActionResult Index()
    {
        var db = new DAL.NorthwindDb();
        var paisesLista = db.ClientePaisLista();
        var selectPaisesLista =
            new SelectList(paisesLista);
        ViewBag.PaisesLista = selectPaisesLista;
        return View();
    }

    [HttpPost]
    public ActionResult Index(FormCollection form)
    {
        string pais = form["pais"];

        var db = new DAL.NorthwindDb();
        var clienteLista = db.ClientePorPaisLista(pais);

        var paisesLista = db.ClientePaisLista();
        var selectPaisesLista =
            new SelectList(paisesLista, pais);
        ViewBag.PaisesLista = selectPaisesLista;

        return View(clienteLista);
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

13. Teste o programa:

The image displays two screenshots of an ASP.NET application running in a browser. Both screenshots show the same page: "Clientes por País".

Screenshot 1 (Top): This screenshot shows a dropdown menu with the option "Argentina" selected. Below the dropdown is a "Confirmar" button. The page title is "Index - My ASP.NET Application" and the URL is "localhost:50163".

Screenshot 2 (Bottom): This screenshot shows a table of customer data. The table has columns: Nome, Telefone, and Cidade. The data is as follows:

Nome	Telefone	Cidade
Gourmet Lanchonetes	(11) 555-9482	Campinas
Wellington Importadora	(14) 555-8122	Resende
Hanari Carnes	(21) 555-0091	Rio de Janeiro
Que Delícia	(21) 555-4252	Rio de Janeiro
Ricardo Adocicados	(21) 555-3412	Rio de Janeiro
Comércio Mineiro	(11) 555-7647	Sao Paulo
Família Arquibaldo	(11) 555-9857	Sao Paulo
Queen Cozinha	(11) 555-1189	Sao Paulo
Tradição Hipermercados	(11) 555-2167	Sao Paulo

Both screenshots also include a watermark with the text "Milk 2017" and "Santana 57".

4

Entity Framework (Code First)

- ✓ Preparando o ambiente;
- ✓ Visão geral.

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.

O trabalho de gerar classes que são usadas para gravar e ler dados em um banco de dados é uma tarefa mecânica. Por exemplo, considere uma classe chamada **Cliente** com as propriedades **ClientId**, **Nome** e **Email**. É bem provável que no banco de dados exista uma tabela chamada **Cliente** com os campos **ClientId**, **Nome** e **Email** para armazenar essas informações. A partir disso, as operações de **Insert**, **Update**, **Delete** e **Select** podem ser facilmente geradas de forma dinâmica.

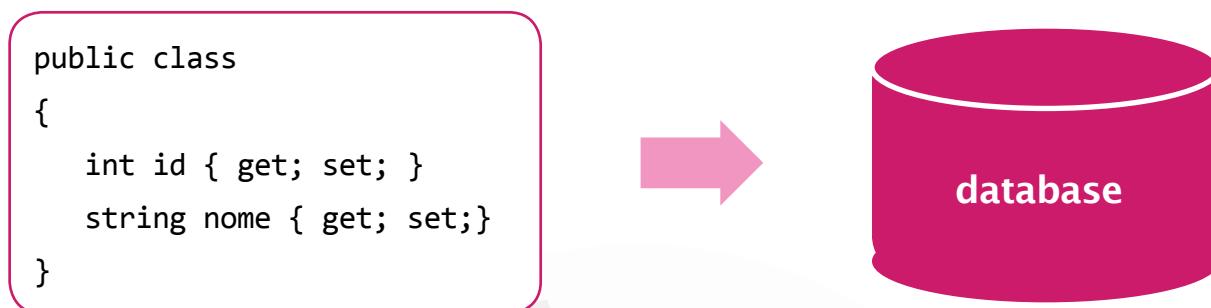
É bom considerar que nem sempre os objetos da aplicação e o banco de dados obedecem a uma relação de um-para-um (uma tabela para cada classe e vice-versa). Existem relacionamentos, agrupamentos de informações, validações, regras de negócio etc. que podem exigir configurações diferentes dos objetos usados 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 permite trabalhar de três maneiras:

Entity Framework (Code First)

- **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.



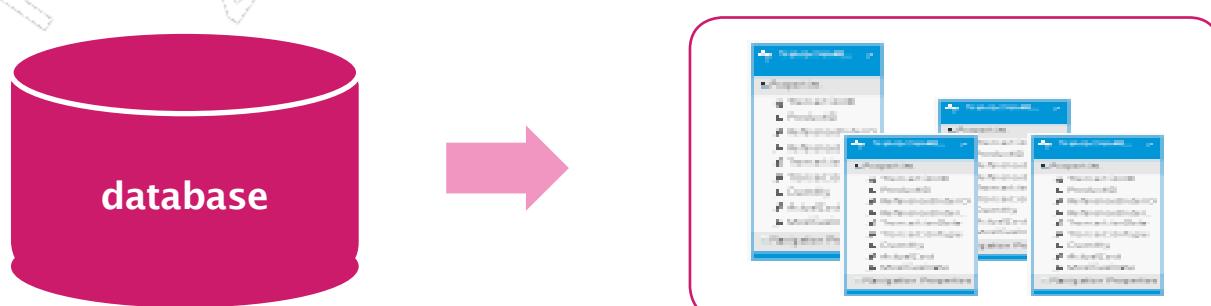
- **Model First**

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**

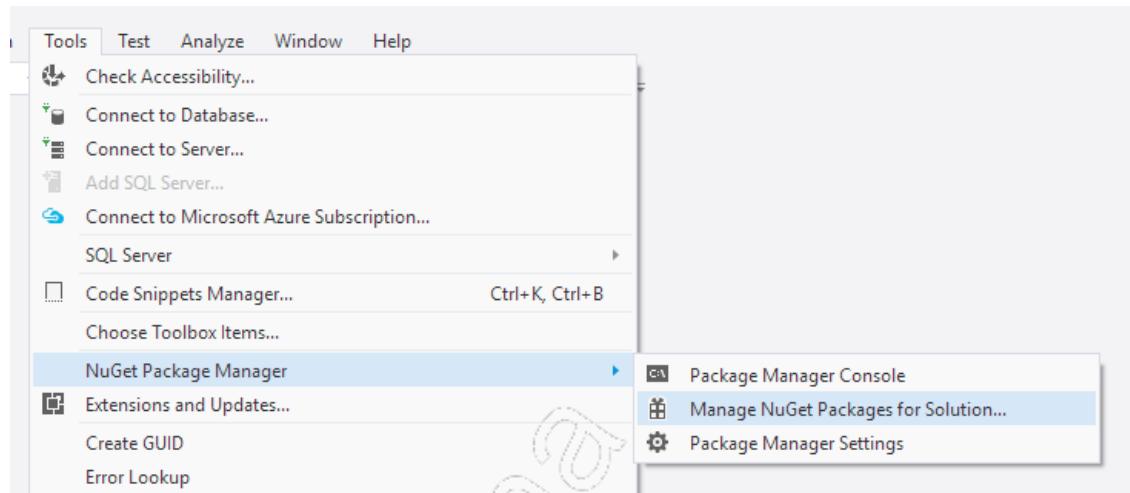
O programador usa um banco de dados existente, e o Entity Framework cria o modelo de dados baseado em suas tabelas e relacionamentos.



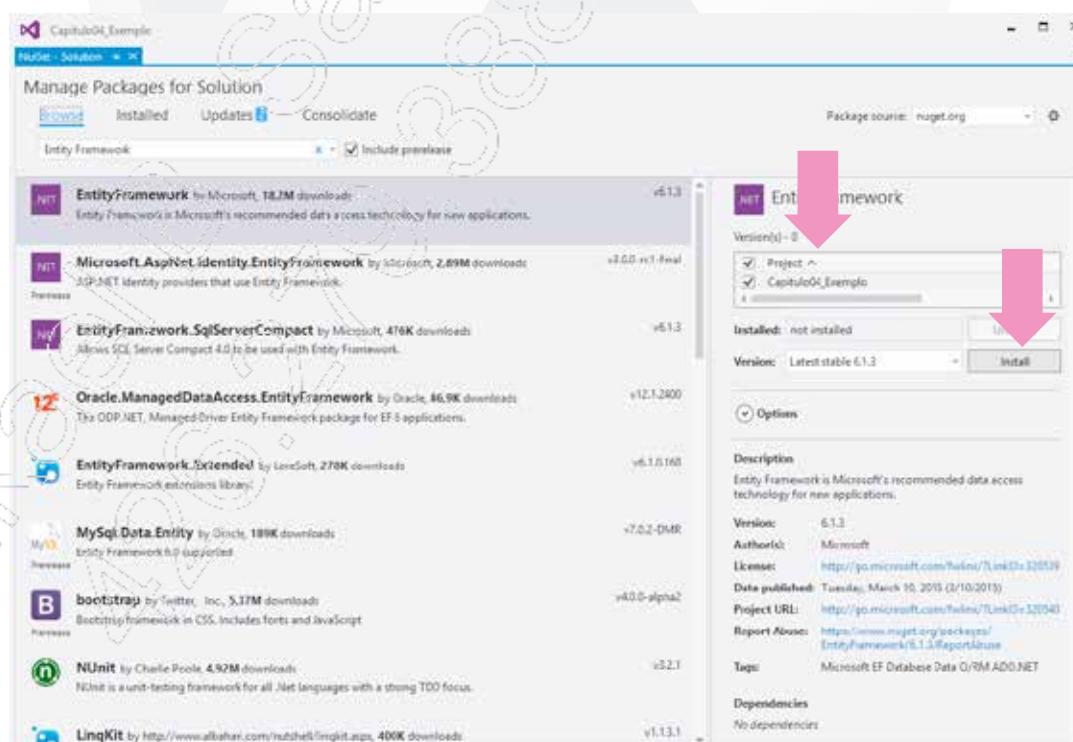
Neste capítulo, será abordado o modelo **Code First**.

4.2. Preparando o ambiente

A melhor maneira de preparar um projeto para usar o Entity Framework é usar o gerenciador de pacotes **NuGet**. No menu **Tools**, escolha a opção **NuGet Package Manager** e, depois, **Manage NuGet Packages for Solution**.

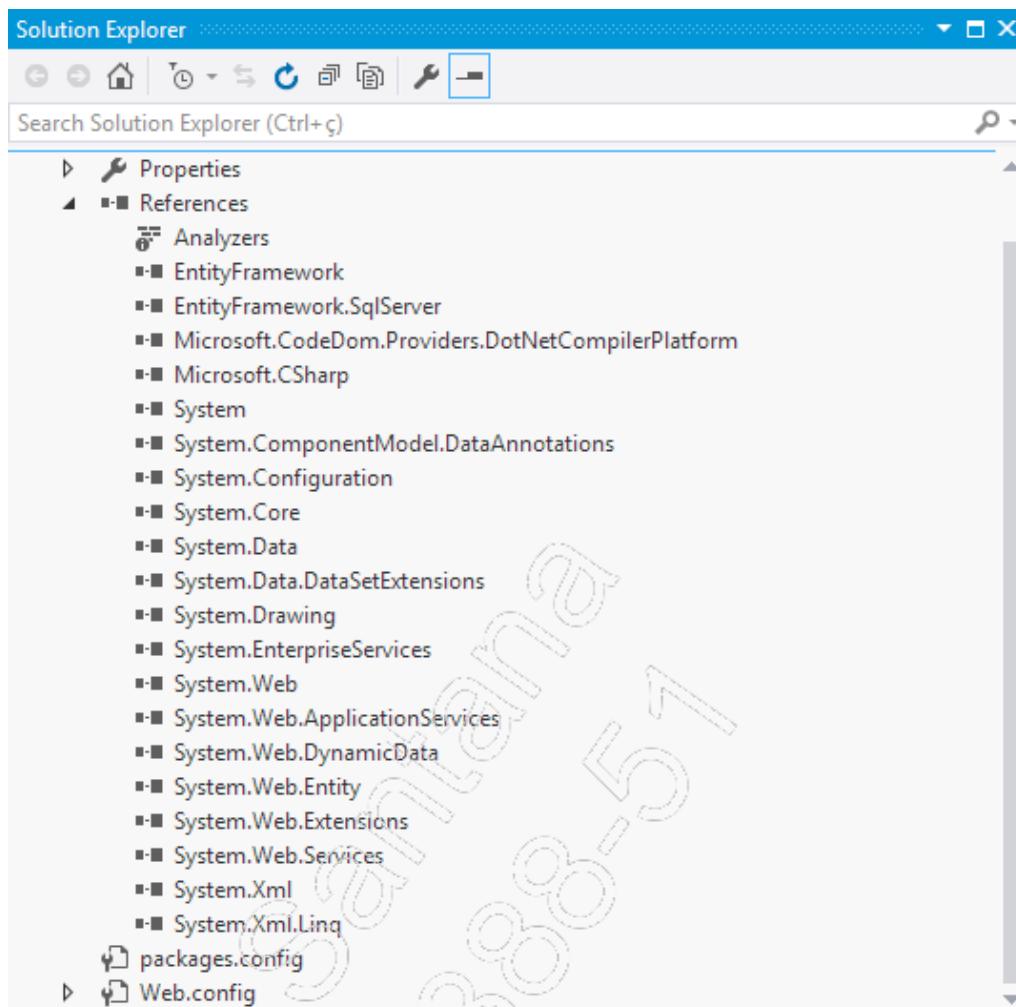


Na janela do NuGet, na caixa de pesquisa, digite **Entity Framework** e, depois de encontrada a opção, marque os projetos em que deseja instalá-lo e escolha **Install**.

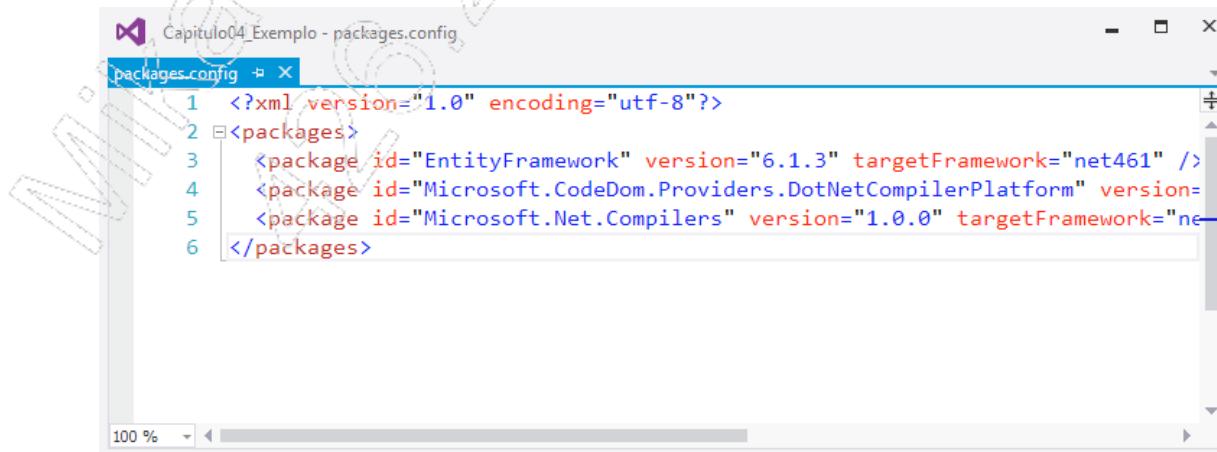


Na descrição, aparece a seguinte informação: **Entity Framework é a tecnologia de acesso a dados recomendada pela Microsoft para novas aplicações.**

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



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



Visual Studio 2015 - ASP.NET com C# Acesso a dados

Uma seção é criada no Web.Config, definindo o banco de dados padrão (**LocalDb\MSSqlLocalDb**) e o provider padrão (**SqlClient**).

```
<configuration>
  <configSections>

    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
      EntityFramework, Version=6.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089" requirePermission="false" />

  </configSections>
  <system.web>
    <compilation debug="true" targetFramework="4.6.1" />
    <httpRuntime targetFramework="4.6.1" />
  </system.web>
  <system.codedom>
    <compilers>...</compilers>
  </system.codedom>

    <entityFramework>
      <defaultConnectionFactory
        type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
        EntityFramework">
        <parameters>
          <parameter value="mssqllocaldb" />
        </parameters>
      </defaultConnectionFactory>

      <providers>
        <provider invariantName="System.Data.SqlClient"
          type="System.Data.Entity.SqlServer.SqlProviderServices,
          EntityFramework.SqlServer" />
      </providers>
    </entityFramework>

  </configuration>
```

4.3. Visão geral

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.

4.3.1. 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, Repcionista, 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**:

```
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.

Além das propriedades, uma entidade tem relacionamentos com outras entidades e apresenta comportamento e validações. Por exemplo, um produto não pode ter um preço negativo. Na tentativa de definirmos um preço negativo, o comportamento do objeto produto pode disparar um erro. Nesse caso, a definição da propriedade **Preco** ficaria da seguinte forma:

```
private decimal campoPreco;
public decimal Preco
{
    get { return campoPreco; }
    set{
        if (value < 0)
        {
            throw new Exception(
                "O preço deve ser igual ou maior que zero");
        }
        else
        {
            campoPreco = value;
        }
    }
}
```

Uma classe de modelo de domínio pode incluir funcionalidades:

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

    public string Nome { get; set; }

    private decimal campoPreco;

    public decimal Preco
    {
        get { return campoPreco; }
        set
        {
            if (value < 0)
            {
                throw new Exception(
                    "O preço deve ser igual ou maior que zero");
            }
            else
            {
                campoPreco = value;
            }
        }
    }

    public int Estoque { get; set; }

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

Visual Studio 2015 - ASP.NET com C# Acesso a dados

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:

```
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. Esse relacionamento é chamado **um-para-muitos**. 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:

```
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; }
}
```

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

```
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:

```
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:

```
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; }
}

}
```

Ou:

```
public class Categoria
{
    public Categoria()
    {
        this.Produtos = new HashSet<Produto>();
    }

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

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

4.3.2. 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.

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

public class LojaContext : DbContext
{}
```

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

```
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**?

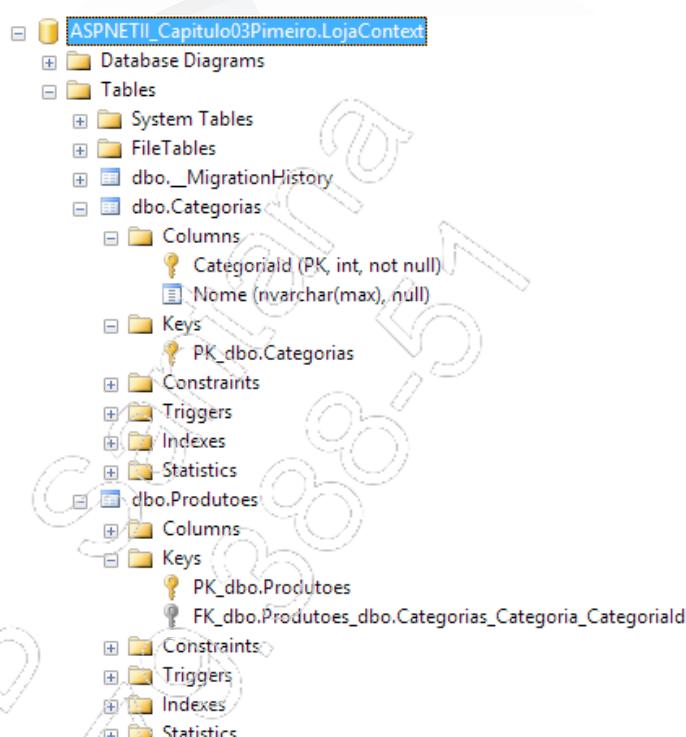
A resposta a todas essas perguntas é a mesma: O Entity Framework usou uma série de convenções. Vejamos:

- 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**;

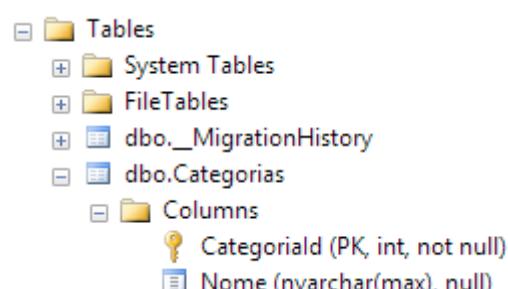
Visual Studio 2015 - ASP.NET com C# Acesso a dados

- 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.

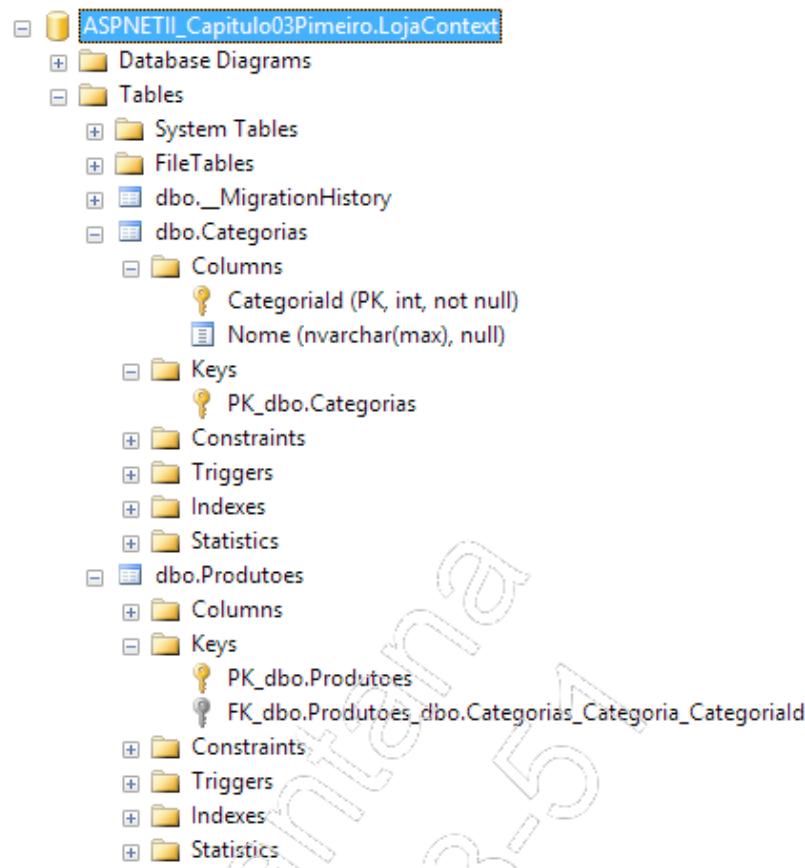
É claro que o EF dispõe de recursos para personalizar cada decisão tomada pelas convenções internas de nomenclatura e estrutura criada no banco. Mas, antes de personalizar o comportamento padrão do EF, convém analisar o que foi criado.



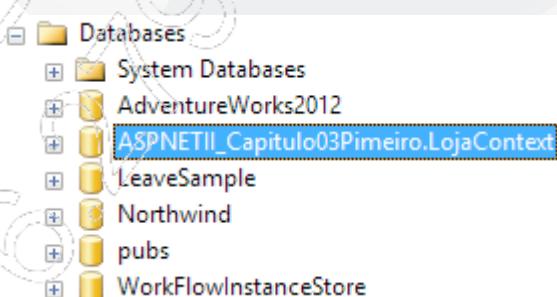
Além das tabelas **Categorias** e **Produtos**, foi criada a tabela **__MigrationHistory**. Essa tabela mantém um histórico das alterações do banco. Quando uma entidade é alterada em sua estrutura, essa tabela passa a não mais corresponder à estrutura atual, forçando o EF a recriar ou alterar o banco de dados. Esse processo pode também ser personalizado.



Um relacionamento de **um-para-muitos** foi criado com suas respectivas chaves primárias e estrangeiras.



O nome do banco de dados é o nome do projeto com o namespace:



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**.

```
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)**

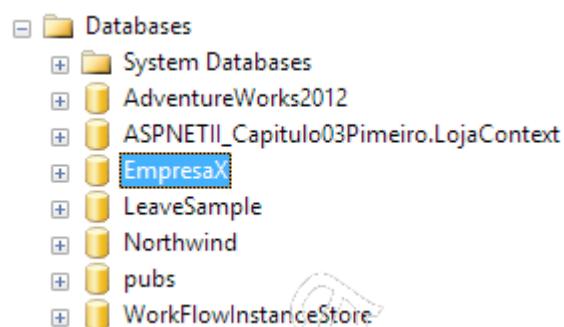
Se um parâmetro do tipo string é passado, pode significar uma **string de conexão** ou o nome de uma string de conexão definida no **Web.Config**, na seção **ConnectionStrings**. O exemplo a seguir mostra uma string de conexão sendo passada:

```
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 **DbConfig**, deve-se passar a string de conexão:

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



No caso apresentado anteriormente, a classe derivada recebe um parâmetro do tipo string, mas isso é uma opção. É perfeitamente possível criar uma classe que não receba parâmetros, mas passe o parâmetro da string de conexão para a classe **DbContext**. Veja o exemplo:

```
public class LojaContext :DbContext  
{  
    private string conexao =  
        @"Data Source=localhost\sqlexpress;  
        Initial Catalog=EmpresaZ;  
        Integrated Security=true";  
  
    public LojaContext():base(conexao)  
    {  
    }  
  
    public DbSet<Categoria> Categorias { get; set; }  
    public DbSet<Produto> Produtos { get; set; }  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Neste caso, é possível chamar a classe sem passar a string de conexão:

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

- 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**:

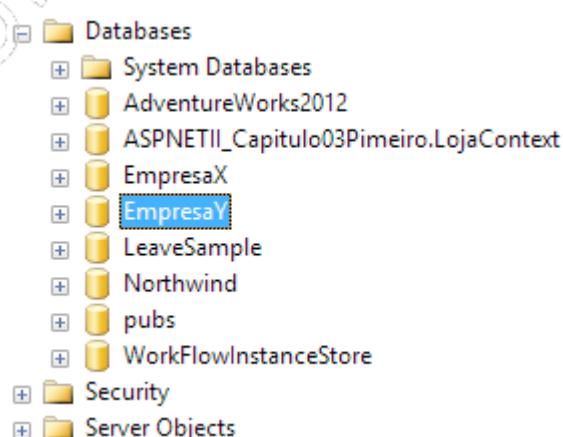
- Arquivo **Web.Config**:

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

- Durante a execução do programa:

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

- Resultado (banco de dados criado):



- 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:

```
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.

4.3.3. 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 forneçam 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;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Web;

[Table("Produtos")]
public class Produto
{
    public int ProdutoId { get; set; }
    public string Nome { get; set; }

}
```

- **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.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Web

[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; }

}
```

- **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, se chame **Id** ou o nome da classe seguido por **Id**, o Entity Framework automaticamente o define como chave primária.

```
public class Cliente
{
    [Key]
    public int ClienteId { get; set; }

    public string Nome { get; set; }

    public Endereco EnderecoResidencial { get; set; }

    public Endereco EnderecoCobranca { 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).

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

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

Os campos que não são requeridos devem 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.

4.3.4. 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.

4.3.4.1. 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>() 1
            .Property(d => d.Nome) 2
            .IsRequired(); 3
    }
}
```

- 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.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

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**:

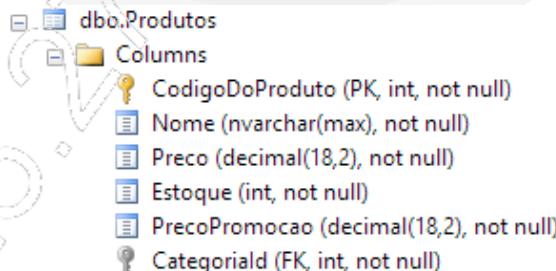
```
protected override
void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    //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);

}
```



Visual Studio 2015 - ASP.NET com C# Acesso a dados

Neste outro exemplo, um relacionamento é definido entre a tabela **Produtos** e **Categorias**:

```
protected override
void OnModelCreating(DbModelBuilder modelBuilder)

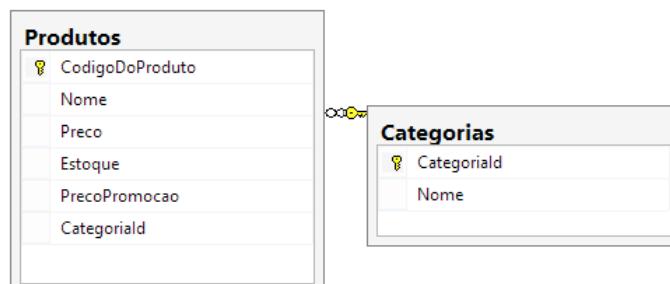
{
    base.OnModelCreating(modelBuilder);

    //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);
}
```



4.3.5. 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 reverter 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**:

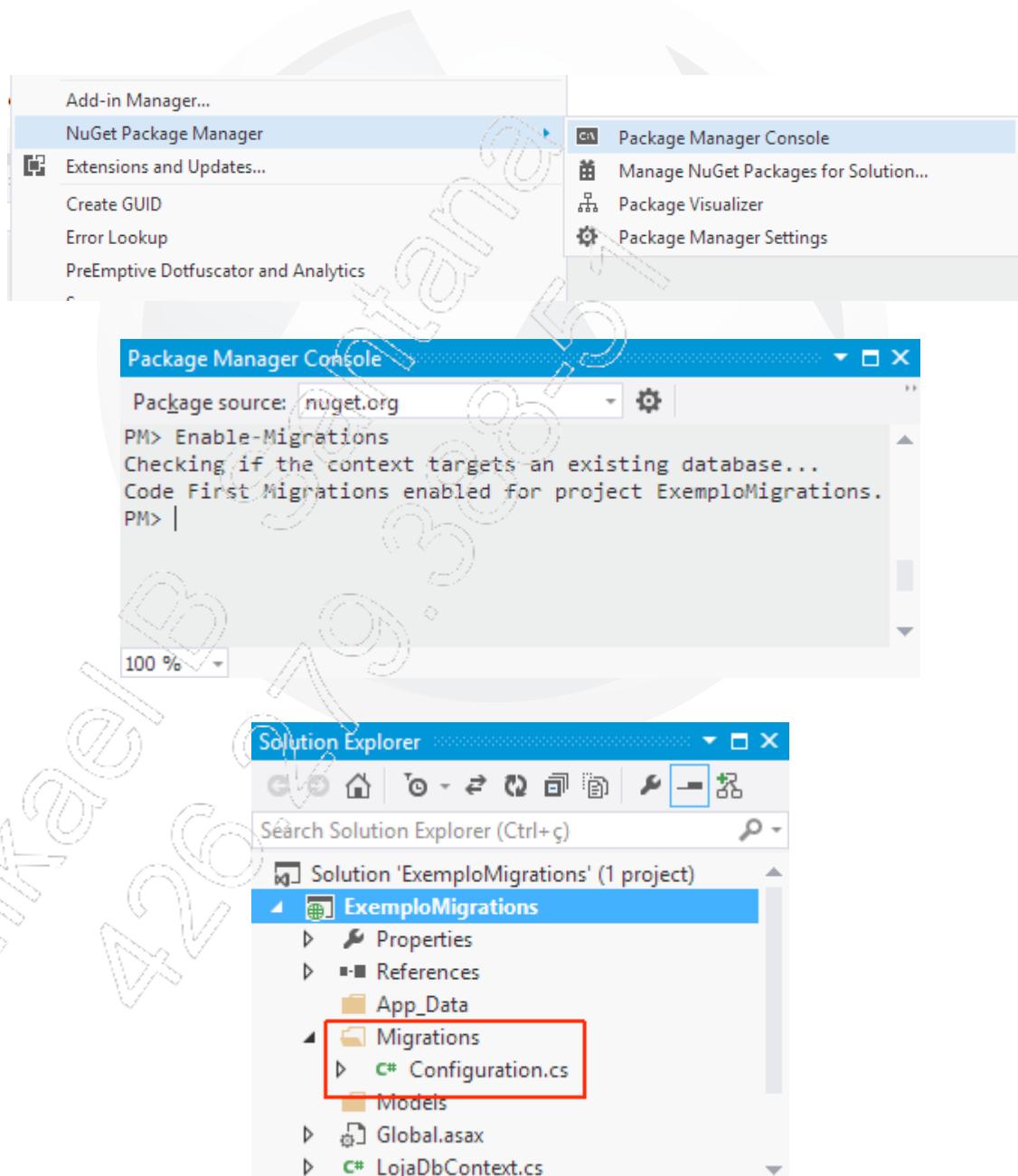
```
static LojaContext()
{
    Database.SetInitializer(
        new DropCreateDatabaseAlways<LojaContext>());
}
```

4.3.6. Migrations

O Entity Framework conta com um recurso para migrar os dados de uma versão a outra quando houver alteração de estrutura.

- **Enable-Migrations**

Para funcionar, é necessário habilitar o recurso **Migrations** para a classe de contexto. Isso é feito executando o comando **Enable-Migrations** no console do **Package Manager**. Será criada uma pasta chamada **Migrations** e uma classe chamada **Configuration**:

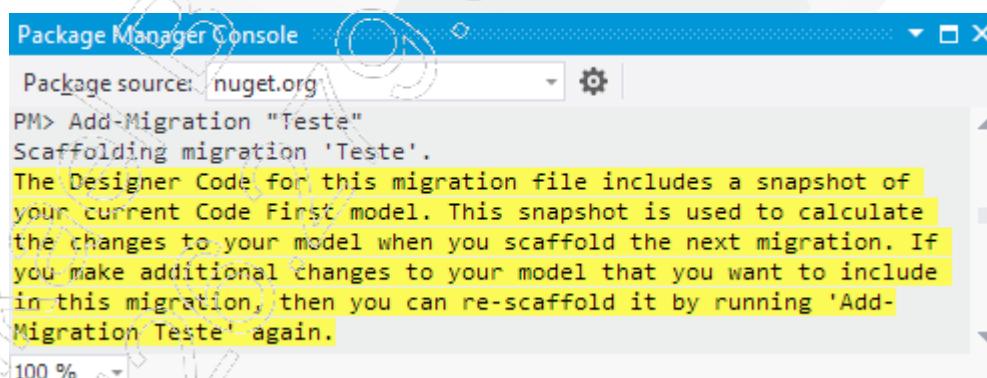


A classe **Configuration** permite configurar todos os detalhes de comportamento do Entity Framework, e é derivada da classe **DbMigrationsConfiguration<T>**, em que **<T>** é uma classe derivada de **DbContext**. No construtor dessa classe, é possível, entre outras coisas, habilitar ou não a migração automática:

```
internal sealed class Configuration :  
    DbMigrationsConfiguration<ExemploMigrations.LojaDbContext>  
{  
    public Configuration()  
    {  
        AutomaticMigrationsEnabled = false;  
    }  
}
```

- **Add-Migration**

Ao criar uma alteração em uma tabela, é possível criar uma imagem do estado atual do banco, para que no futuro este possa ser restaurado ao estado original. A alteração pode ser a inclusão, exclusão ou alteração de um campo ou tabela. Isso é feito com o comando **add-Migration "nome"**, em que "**nome**" é um identificador para esta situação.



Este procedimento cria apenas um "ponto de restauração", mas não atualiza o banco. Uma classe derivada de **DbMigration** é criada com informações das alterações dos modelos. Dois métodos são definidos: **Up()** e **Down()**. Esses métodos permitem aplicar e voltar uma versão.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

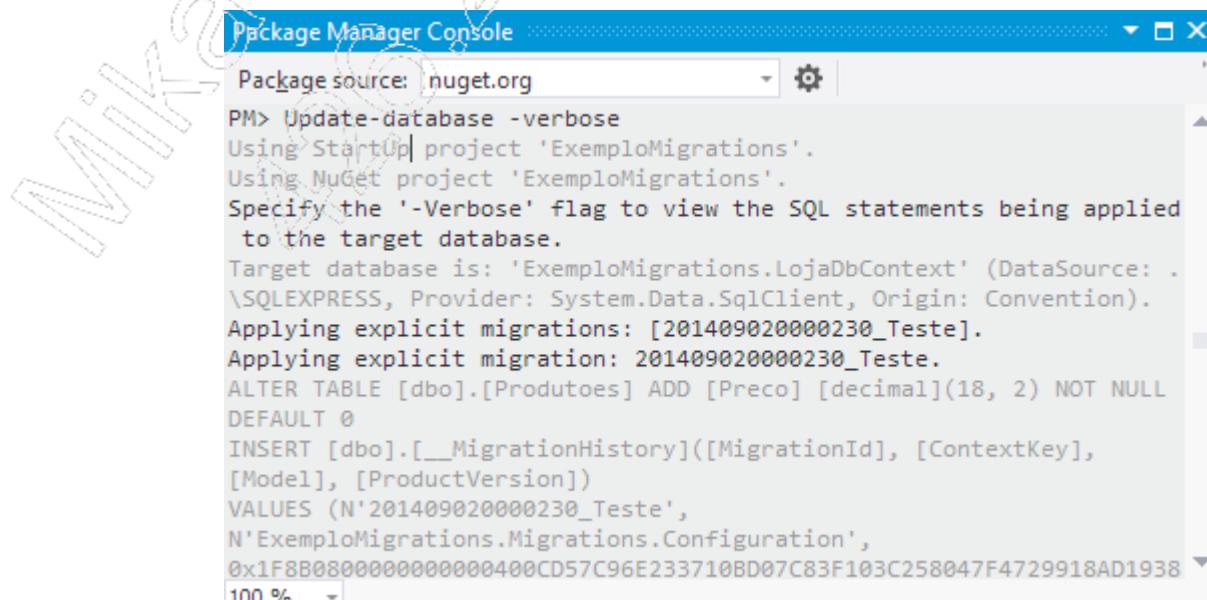
A versão a seguir demonstra a classe criada pelo comando **Add-Migration** do **NuGet** e os métodos **Up** e **Down**, que permitem executar e desfazer as alterações de uma versão (neste caso, a inclusão da coluna **Preco** na tabela **Produtos**):

```
public partial class Teste : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Produtos", "Preco",
            c =>
            c.Decimal(nullable: false,
                precision: 18,
                scale: 2));
    }

    public override void Down()
    {
        DropColumn("dbo.Produtos", "Preco");
    }
}
```

- **Update-database**

Para que as alterações nas classes sejam refletidas no banco, é necessário executar no console do NuGet o comando **Update-database**. O parâmetro **-verbose** mostra as expressões SQL que estão sendo executadas no banco.



```
PM> Update-database -verbose
Using StartUp project 'ExemploMigrations'.
Using NuGet project 'ExemploMigrations'.
Specify the '-Verbose' flag to view the SQL statements being applied
to the target database.
Target database is: 'ExemploMigrations.LoaDbContext' (DataSource: .\SQLEXPRESS, Provider: System.Data.SqlClient, Origin: Convention).
Applying explicit migrations: [201409020000230_Teste].
Applying explicit migration: 201409020000230_Teste.
ALTER TABLE [dbo].[Produtos] ADD [Preco] [decimal](18, 2) NOT NULL
DEFAULT 0
INSERT [dbo].[__MigrationHistory]([MigrationId], [ContextKey],
[Model], [ProductVersion])
VALUES (N'201409020000230_Teste',
N'ExemploMigrations.Migrations.Configuration',
0x1F8B080000000000400CD57C96E233710BD07C83F103C258047F4729918AD1938
100 %
```

- O método **Seed**

Além do construtor, a classe **Configuration** apresenta o método **Seed** (semente). Este método permite incluir registros nas tabelas quando o banco é criado. Por exemplo, uma lista de estados (SP, RJ, BH) poderia ser previamente preenchida antes de qualquer cadastro por parte do usuário.

```
public Configuration()
{
    AutomaticMigrationsEnabled = false;
}

protected override void Seed(
    ExemploMigrations.LoaDbContext context)
{

    context.Estados.Add(
        new Estado() { Sigla= "SP", Nome="São Paulo1" });
    context.Estados.Add(
        new Estado() { Sigla= "Rj", Nome="Rio de Janeiro2" });
}
```

O mecanismo **Migration** é útil quando existe um banco de dados complexo, em que alguns poucos campos mudam e esta mudança precisa ficar documentada. No início do desenvolvimento, em que o banco é pequeno e a estrutura está em definição, o mecanismo **Database Initializer** é mais adequado.

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** 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;
- **DbMigration** é a classe base para o modelo de controle que utiliza o .NET Framework;
- **Configuration** é a classe utilizada para personalizar uma base de dados;
- **Complex Types** é um agrupamento de campos que pode ser utilizado em conjunto para criar um campo.

4

Entity Framework (Code First)

Teste seus conhecimentos

Mikael
Baptana
426.2
57



IMPACTA
EDITORA

1. Qual o propósito do componente Entity Framework?

- a) Conectar um banco de dados.
- b) Automatizar o processo de persistência de dados.
- c) Gerenciar o banco de dados SQL Server.
- d) Gerenciar qualquer banco de dados.
- e) Testar expressões SQL.

2. Para conectar o banco de dados, deve ser criada uma classe derivada de qual classe do Entity Framework?

- a) DbSet
- b) Table
- c) DataBase
- d) DbContext
- e) DbConnection

3. Qual classe deve ser usada para criar um conjunto de dados que será mapeado para uma tabela?

- a) DataSet
- b) DataTable
- c) DbContext
- d) DbSet<T>
- e) TableContext

4. Quais das propriedades adiante serão automaticamente mapeadas para a chave primária de uma tabela pelo Entity Framework? (Considere uma classe chamada Cliente.)

- a) CódigoCliente
- b) Id
- c) ClientId
- d) As alternativas A e C estão corretas.
- e) As alternativas B e C estão corretas.

5. Como se chama o processo de criar e alterar automaticamente o banco de dados quando o modelo sofre uma alteração?

- a) Deploy
- b) Migration
- c) ApplicationDomain
- d) DatabaseUpdate
- e) ComplexType

4

Entity Framework (Code First)

Mãos à obra!

Mikael
Sarana
426.27



IMPACTA
EDITORA

Laboratório 1

A - Usando Entity Framework para pesquisar dados

Este laboratório criará uma página que permite consultar os produtos da empresa **Northwind Traders** por categoria ou por fornecedor. Ele já foi feito no capítulo anterior, porém usando ADO.NET. Agora, será usado o **Entity Framework** no modo **Code First**.

1. Crie um novo projeto Web vazio, chamado **Cap04_Lab01**;
2. Usando NuGet, adicione o pacote Entity Framework (menu **Tools / NuGet Package Manager / Manage NuGet Packages for Solution**). Procure por **Entity Framework**, marque o projeto atual e clique em **Install**;
3. Crie uma pasta chamada **Models** e, dentro dela, uma classe chamada **Produto**:

```
public class Produto
{
}
```

4. Adicione uma classe chamada **Categoria**, também dentro da pasta **Models**:

```
public class Categoria
{
}
```

5. Adicione as propriedades **Categoriald** e **Nome**:

```
public class Categoria
{
    public int Categoriald { get; set; }
    public string Nome { get; set; }
}
```

6. No banco de dados **Northwind**, a tabela se chama **Categories** e os campos **CategoryId** e **CategoryName**;

Faça o mapeamento necessário, na tabela **Produto** e **Categoria**. Lembre-se de colocar a instrução **using** para definir o namespace das classes **Table** e **Column**.

```
using System.ComponentModel.DataAnnotations.Schema;  
...  
  
[Table("Categories")]  
public class Categoria  
{  
    [Column("CategoryId")]  
    public int CategoriaId { get; set; }  
  
    [Column("CategoryName")]  
    public string Nome { get; set; }  
}  
  
[Table("Products")]  
public class Produto  
{  
    [Column("ProductId")]  
    public int ProdutoId { get; set; }  
  
    [Column("ProductName")]  
    public string Nome { get; set; }  
  
    [Column("UnitPrice")]  
    public decimal Preco { get; set; }  
  
    [Column("UnitsInStock")]  
    public short Estoque { get; set; }  
  
    [ForeignKey("Categoria")]  
    [Column("CategoryId")]  
    public int CategoriaId { get; set; }  
  
    public Categoria Categoria { get; set; }  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

7. Crie uma pasta chamada **DAL**. Dentro dessa pasta, crie a classe **NorthwindContext**, derivada de **DbContext**. Lembre-se de que a classe **DbContext** está no namespace **System.Data.Entity**:

```
public class NorthwindContext:DbContext
{
}
```

8. Na classe **NorthwindContext**, crie um campo **private** para armazenar a string de conexão:

```
public class NorthwindContext:DbContext
{
    private const string Conexao =
        @"Data Source=(LocalDb)\MSSQLLocalDb;
        Initial Catalog=Northwind;
        Integrated Security=True";
}
```

9. Defina o construtor da classe, chamando o construtor da classe base (**DbContext**) e passando a string de conexão como parâmetro. Sempre compile o seu código para ter certeza de que não há erros;

```
public class NorthwindContext:DbContext
{
    private const string Conexao =
        @"Data Source=localhost\sqlexpress;
        Initial Catalog=Northwind;
        Integrated Security=True";

    public NorthwindContext() : base(Conexao) { }
}
```

10. Defina os conjuntos de dados: **Produtos** e **Categorias**. Lembre-se de que as classes **Produto** e **Categoria** estão no namespace **Cap04_Lab01.Models**:

```
public class NorthwindContext:DbContext
{
    private const string Conexao =
        @"Data Source=localhost\sqlexpress;
        Initial Catalog=Northwind;
        Integrated Security=True";

    public NorthwindContext() : base(Conexao) { }

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

    public DbSet<Categoria> Categorias { get; set; }

}
```

11. Crie uma nova classe chamada **CategoriaDb**, dentro da pasta **DAL**:

```
public class CategoriaDb
{
```

12. Crie um método para retornar a lista de categorias chamado **CategoriasLista**:

```
public class CategoriaDb
{
    //
    // CategoriasLista
    // Retorna a lista de Categorias
    //

    public List<Categoria> CategoriasLista()
    {
        using (var db = new NorthwindContext())
        {
            return db.Categorias.ToList();
        }
    }

}// Final da Classe CategoriaDb
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

13. Crie um método para retornar os produtos de uma categoria chamado **ProdutosPorCategoria**:

```
//  
//Produtos por Categoria  
// retorna uma lista de produtos de uma categoria  
//  
public List<Produto> ProdutosPorCategoria(int categoriaId)  
{  
    using (var db = new NorthwindContext())  
    {  
  
        var query = from c in db.Produtos  
                    where c.CategoriaId == categoriaId  
                    select c;  
        var lista = query.ToList();  
  
        return lista;  
    }  
}
```

14. Crie um método para retornar dados de uma categoria de produtos, a partir de seu Id:

```
//  
// CategoriaObter  
// retorna uma categoria a partir do Id  
//  
public Categoria CategoriaObter(int id)  
{  
    using (var db = new NorthwindContext())  
    {  
        return db.Categorias  
            .Where(m => m.CategoriaId == id).FirstOrDefault();  
    }  
}
```

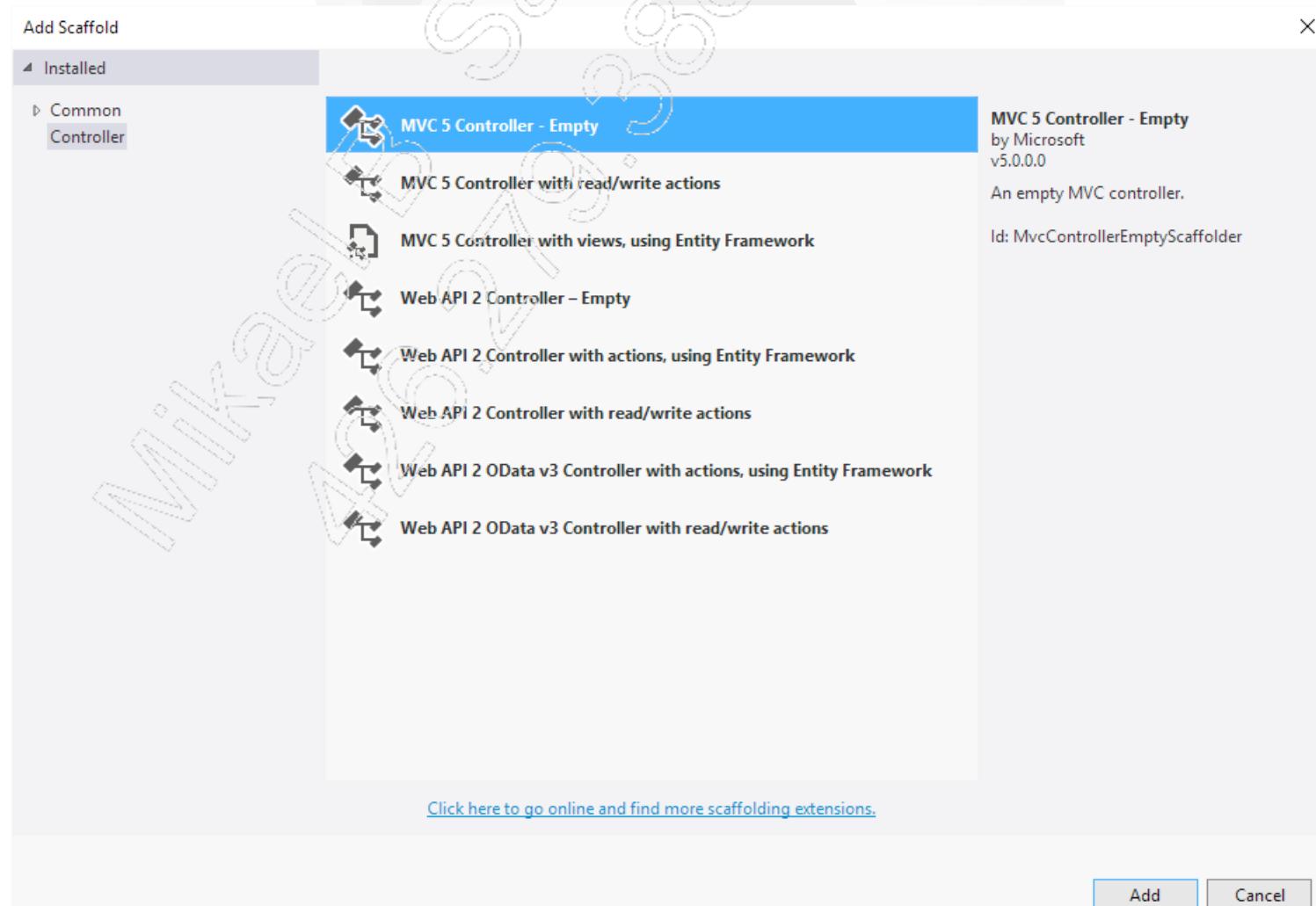
15. A classe **categoriaDb** está completa com seus três métodos:



A parte de acesso a dados está completa. Agora é possível criar facilmente telas para manipular os dados. Será criada uma tela para visualizar os dados usando MVC.

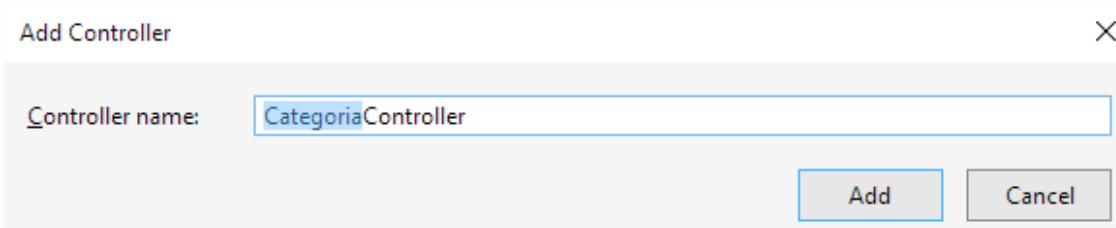
B – Criando telas de consulta com MVC

1. Crie uma pasta chamada **Controllers**. Dentro dessa pasta, escolha, no menu de contexto, **Add / Controller / MVC 5 Controller - Empty**:

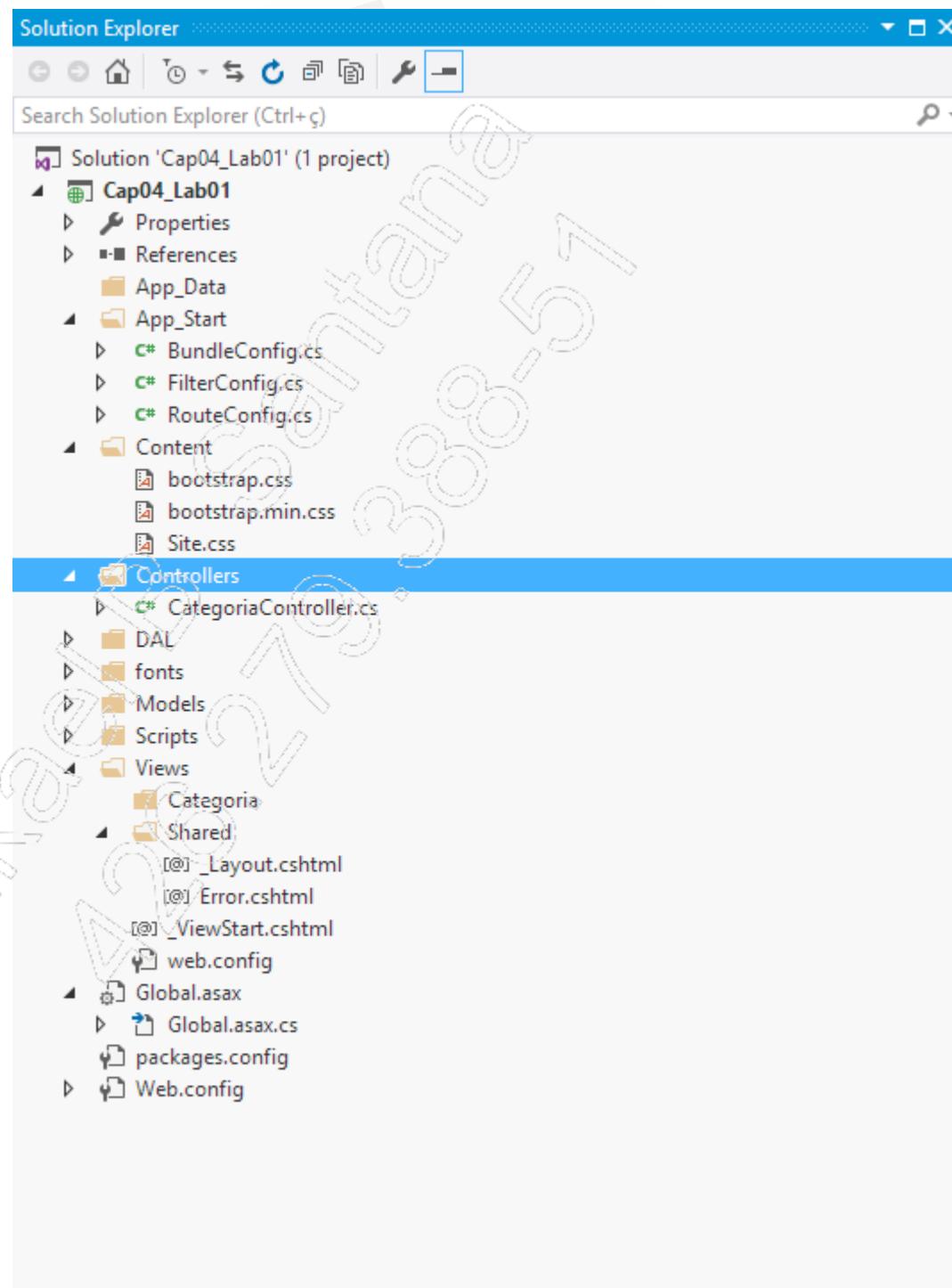


Visual Studio 2015 - ASP.NET com C# Acesso a dados

2. Escolha **CategoriaController** como nome;



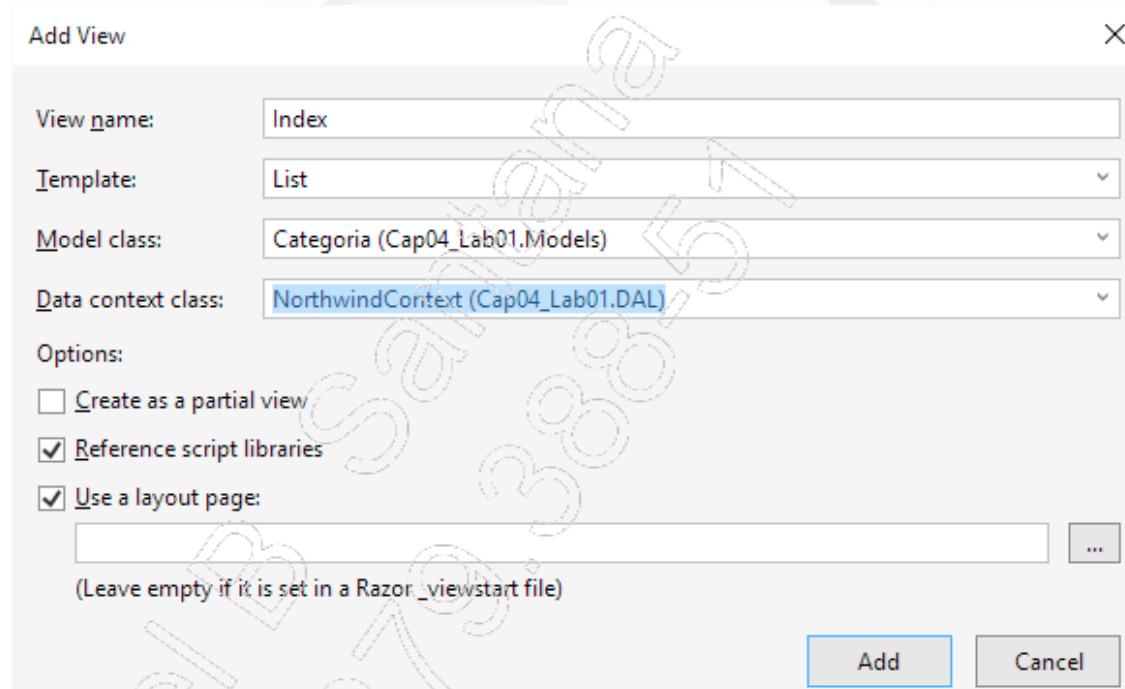
3. Repare que diversos componentes são adicionados ao projeto:



4. Dentro do método **Index** criado pelo assistente, escolha, no menu de contexto, **Add / View**;

Preencha o assistente da View usando o template **List**, a classe de modelo **Castegoria** e a classe de contexto **NorthwindContext**.

```
public class CategoriaController : Controller
{
    // GET: Categoria
    public ActionResult Index()
    {
        return View();
    }
}
```



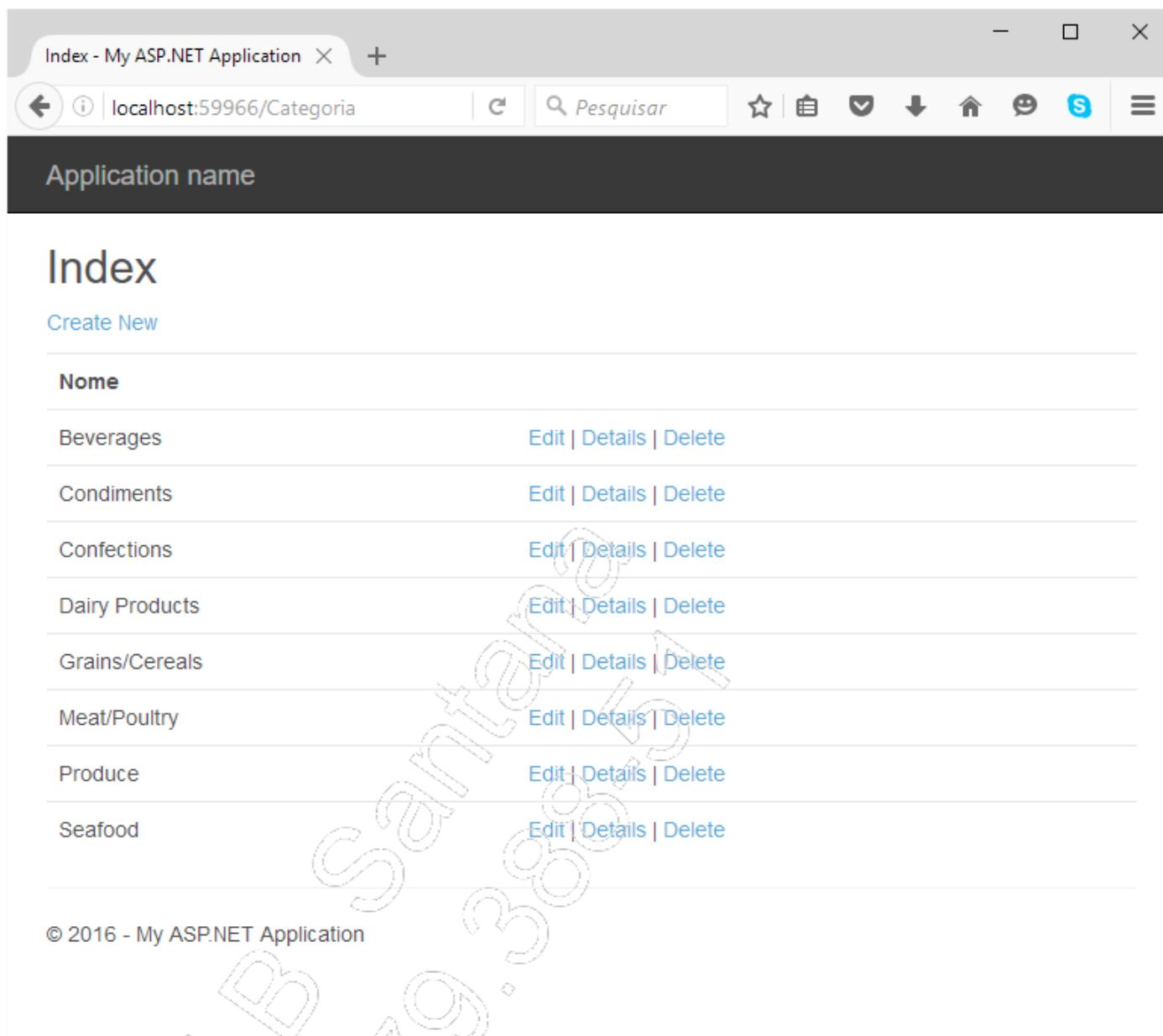
5. Altere o método **Index** para receber uma lista de categorias:

```
public class CategoriaController : Controller
{
    // GET: Categoria
    public ActionResult Index()
    {
        var db = new CategoriaDb();
        var lista = db.CategoriasLista();

        return View(lista);
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

6. É possível visualizar a View criada pelo assistente:



7. A View gerada é uma tela para incluir, alterar e excluir dados das categorias. No nosso exemplo, o objetivo é bem mais simples: exibir a lista de categorias e, ao clicar em uma categoria, exibir a lista de produtos. Altere a View **Index** para o seguinte código:

```
@model IEnumerable<Cap04_Lab01.Models.Categoria>

@{
    ViewBag.Title = "Index";
}



## Categorias

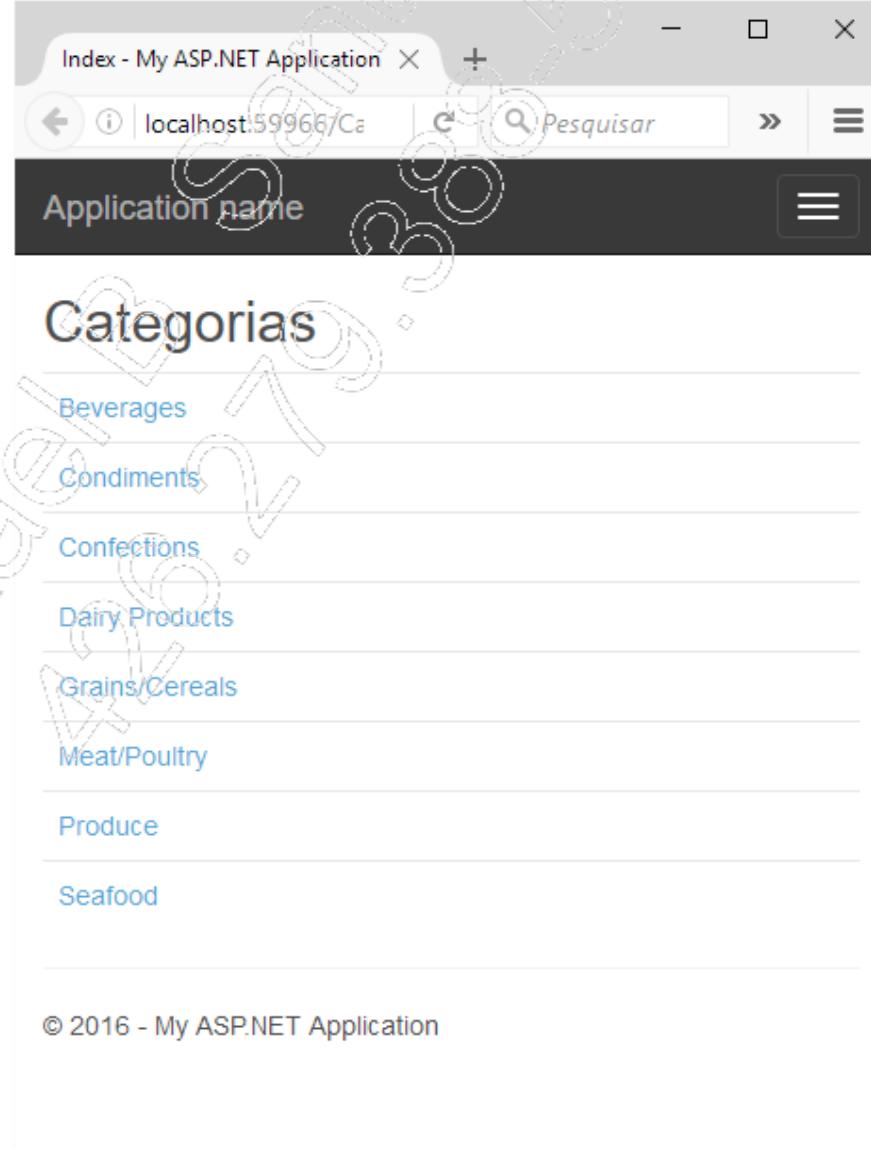


| Nome |  |
|------|--|
|------|--|


```

```
@foreach (var item in Model)
{
    <tr>
        <td>
            @Html.ActionLink(item.Nome,
                "ProdutosPorCategoria",
                new { id=item.CategoriaId})
        </td>
    </tr>
}
</table>
```

8. Teste a visualização das categorias de produtos:



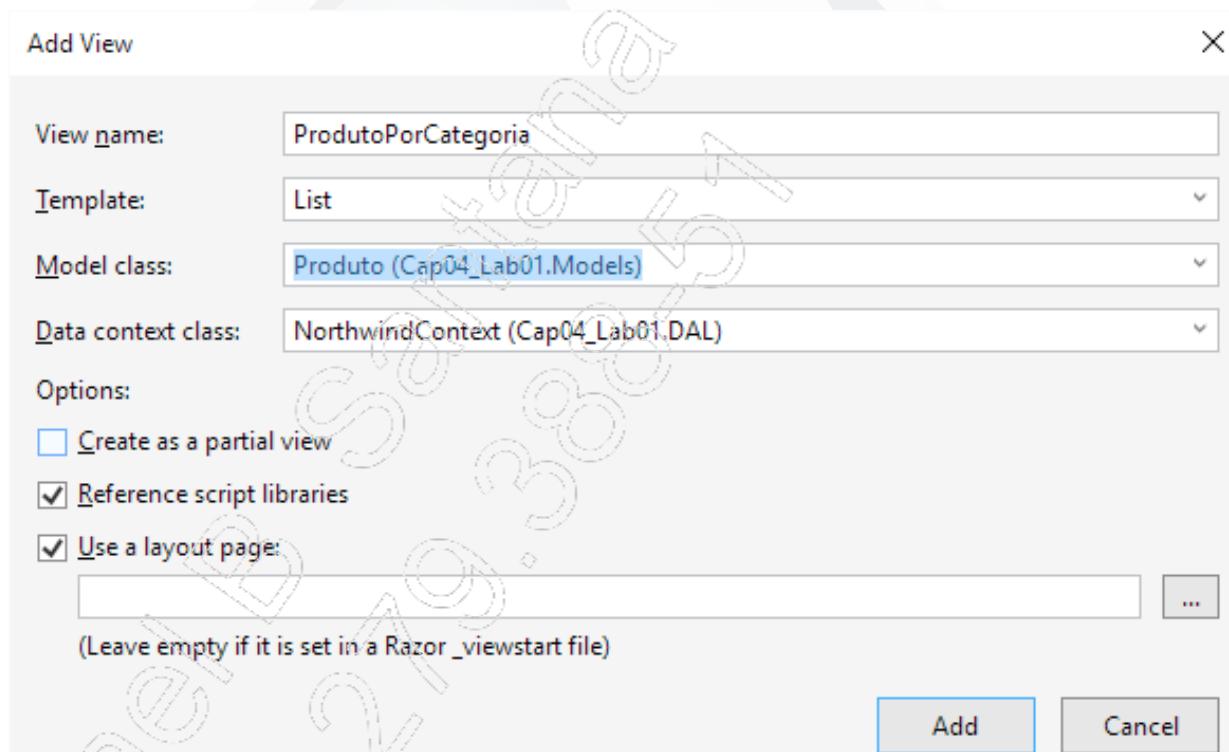
9. No Controller Categoria, crie o método ProdutoPorCategoria:

```
public ActionResult ProdutoPorCategoria(int id)
{
    var db = new CategoriaDb();
    var lista = db.ProdutosPorCategoria(id);

    var categoria = db.CategoriaObter(id);
    ViewBag.CategoriaNome = categoria.Nome;

    return View(lista);
}
```

10. Adicione uma View usando o menu de contexto:



11. Altere o conteúdo da View **ProdutosPorCategoria** para exibir a lista de produtos:

```
@model IEnumerable<Cap04_Lab01.Models.Produto>

{@
    ViewBag.Title = "Produtos Por Categoria";
}

<h2>@ViewBag.CategoriaNome</h2>

<table class="table">
    <tr>

        <th>
            @Html.DisplayNameFor(model => model.Nome)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Preco)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Estoque)
        </th>

    </tr>

    @foreach (var item in Model) {
        <tr>

            <td>
                @Html.DisplayFor(modelItem => item.Nome)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Preco)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Estoque)
            </td>

        </tr>
    }
}

</table>

@Html.ActionLink("Voltar", "Index")
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

12. Teste o programa completo:

The image displays two screenshots of an ASP.NET application running in a browser. Both screenshots have a watermark in the center reading "Mikael Santana 3000-57".

Screenshot 1: Categories

The title bar says "Index - My ASP.NET Application". The URL in the address bar is "localhost:59966/Categoria". The page header "Application name" is visible. The main content shows a list of product categories:

- Beverages
- Condiments
- Confections
- Dairy Products
- Grains/Cereals
- Meat/Poultry
- Produce
- Seafood

At the bottom, it says "© 2016 - My ASP.NET Application".

Screenshot 2: Products by Category

The title bar says "Produtos Por Categoria - My ...". The URL in the address bar is "localhost:59966/Categoria/ProdutosPorCategoria/5". The page header "Application name" is visible. The main content shows a table for the "Grains/Cereals" category:

Nome	Preço	Estoque
Gustaf's Knäckebrot	21,00	104
Tunnbröd	9,00	61
Singaporean Hokkien Fried Mee	14,00	26
Filo Mix	7,00	38
Gnocchi di nonna Alice	38,00	21
Ravioli Angelo	19,50	36
Wimmers gute Semmelknödel	33,25	22

A "Voltar" (Back) link is at the bottom left. At the bottom, it says "© 2016 - My ASP.NET Application".

5

Entity Framework (Model/Database First)

- ✓ Model First;
- ✓ Database First.



IMPACTA
EDITORA

5.1. Introdução

O modo **Code First** do Entity Framework permite um controle total do código incluído nas classes de modelo de domínio. Esse controle, porém, tem um preço: é necessário escrever manualmente cada um dos campos. Em um modelo no qual já existe um banco de dados ou no qual o arquiteto tem mais familiaridade com ferramentas de análise do que com programação, é mais produtivo deixar o Entity Framework criar as classes de modelo usando as ferramentas de design do Visual Studio.

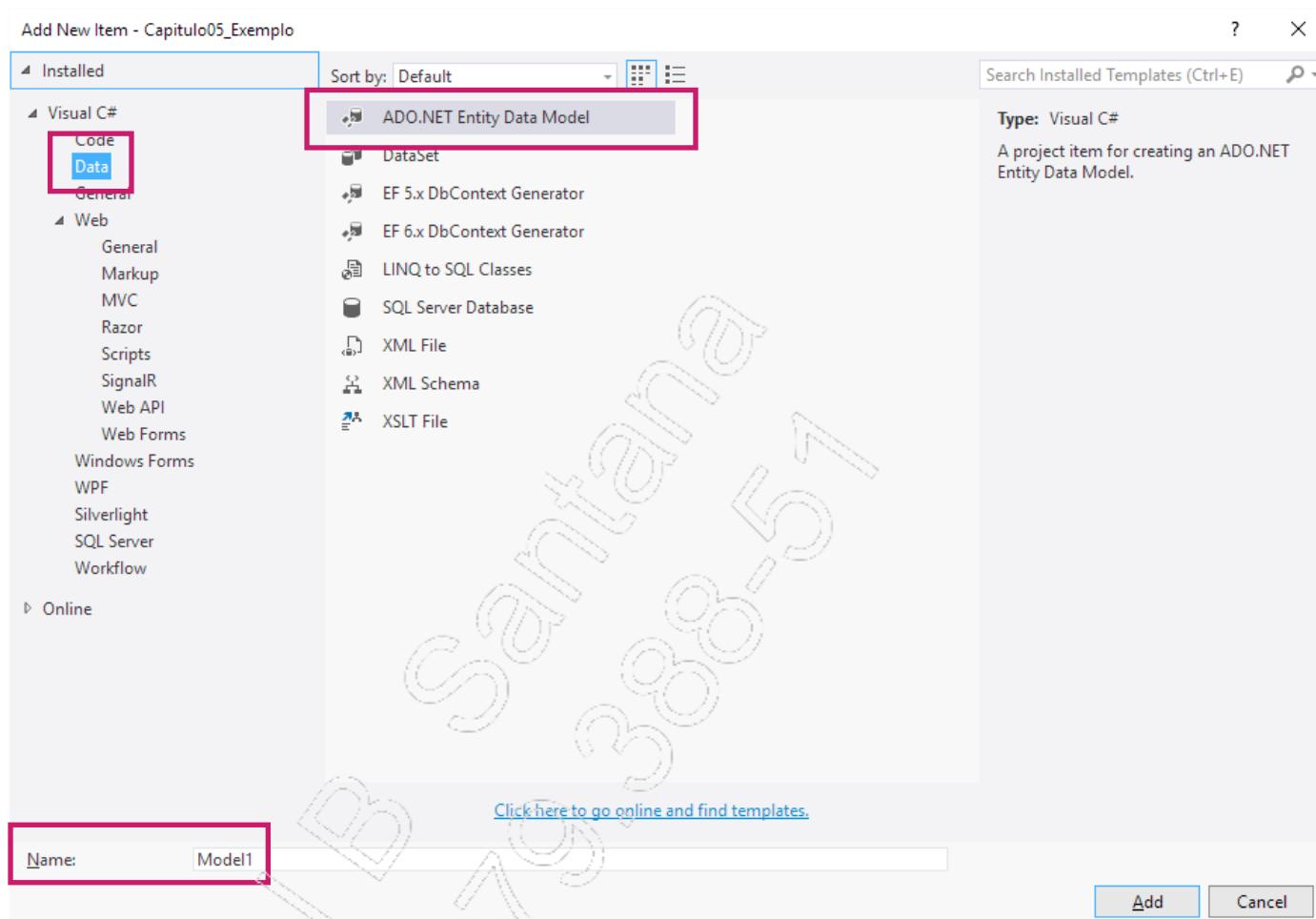
Além do Code First, existem dois modos de criar as classes de modelos de domínio:

- **Model First:** O modelo é criado usando a ferramenta de design do Visual Studio e, então, é gerado o banco de dados com as tabelas e relacionamentos, tendo sua estrutura definida com base no modelo conceitual;
- **Database First:** Um banco de dados existente é utilizado para gerar os modelos de domínio, usando as tabelas e relacionamentos como base para as classes.

De um modo geral, cada tabela gera uma classe, e cada campo dessa tabela gera uma propriedade. Os relacionamentos geram os campos de navegação e as coleções. As classes geradas podem ser modificadas e ajustadas para representar mais claramente as informações do sistema, se necessário. Esse processo se chama **mapeamento**.

5.2. 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:

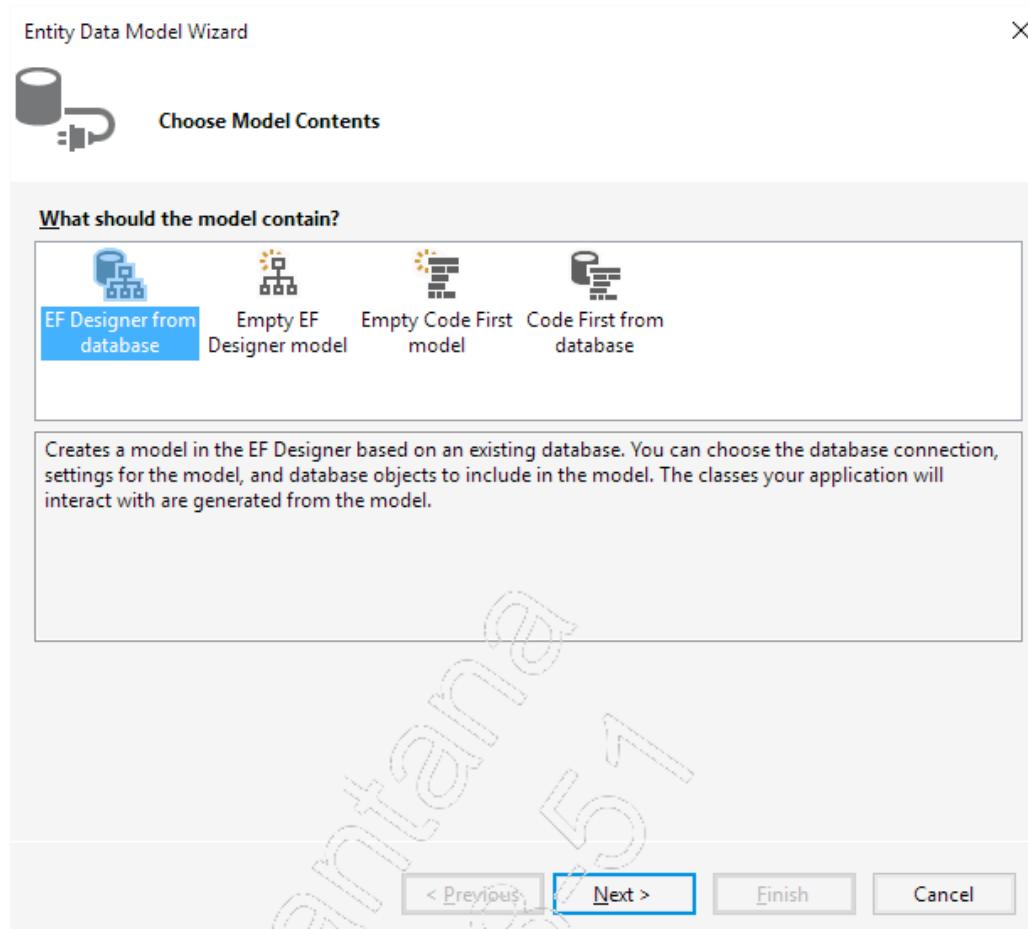


A caixa de diálogo que surgirá exibe as opções de criação de modelos. Elas podem ser as seguintes:

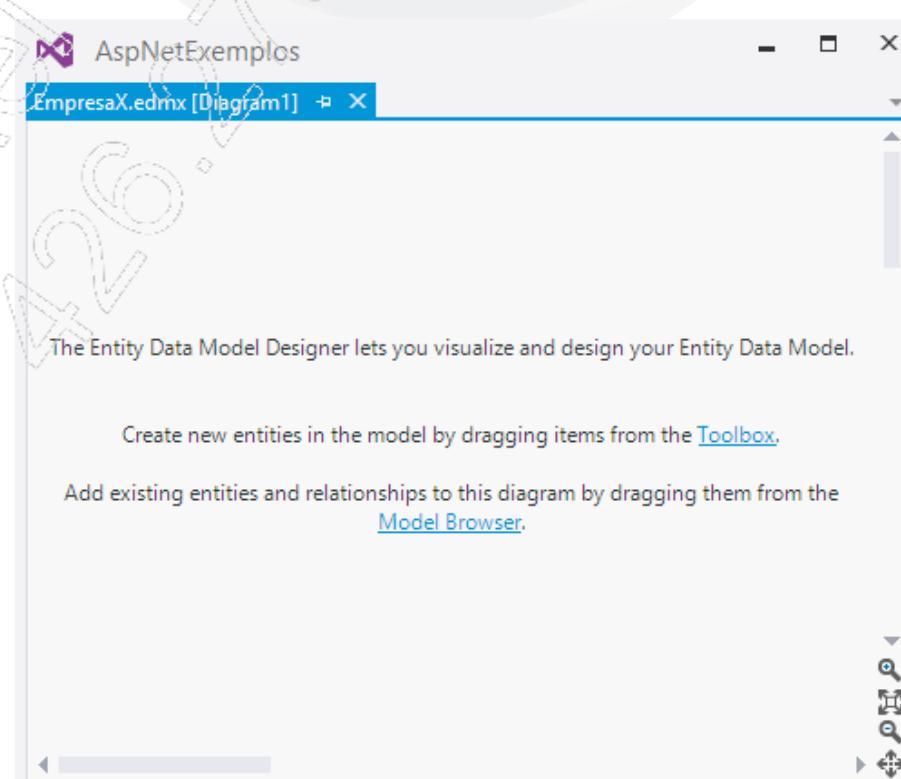
- **EF Designer from database:** Cria um modelo a partir de um banco de dados, usando a ferramenta de design do Visual Studio;
- **Empty EF Designer model:** Um modelo vazio, para criar o modelo de dados usando a ferramenta designer do Visual Studio;
- **Empty Code First model:** Um modelo vazio, sem designer, para criar manualmente classes;

Visual Studio 2015 - ASP.NET com C# Acesso a dados

- **Code First from database:** Cria classes com o modelo **Code First**, usando um banco de dados como base.

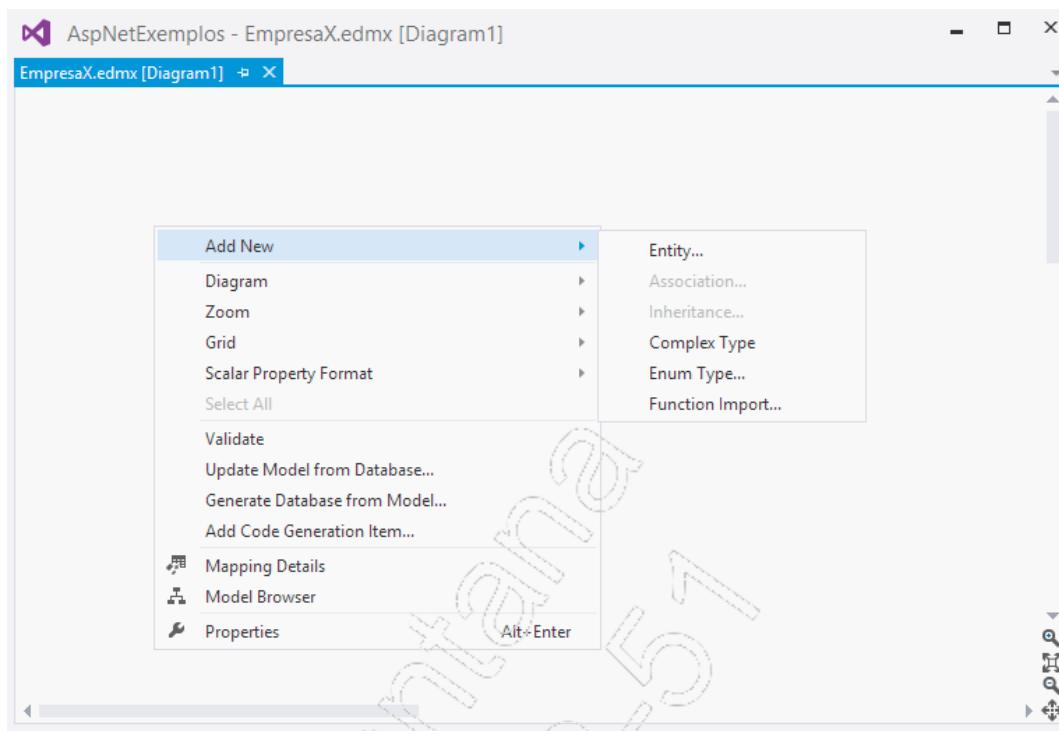


Escolhendo **Empty EF Designer model**, a ferramenta de designer do Entity Framework é exibida. Neste modo, é possível criar entidades e relacionamento para depois criar ou utilizar um banco de dados.

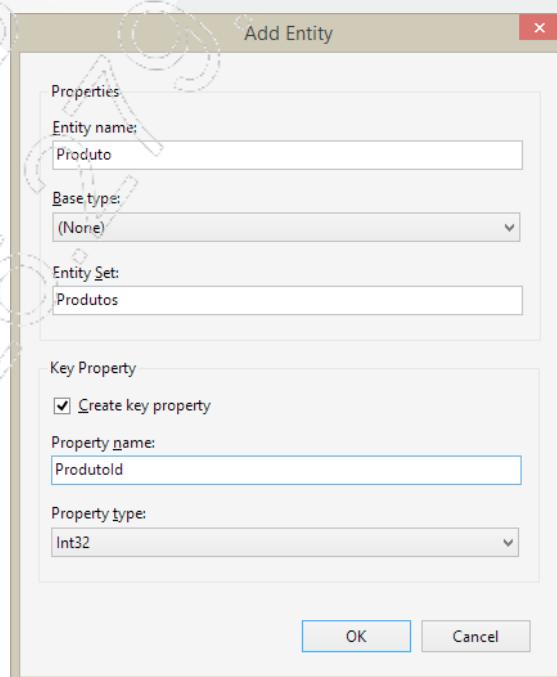


5.2.1. Adicionando uma Entity (entidade)

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:

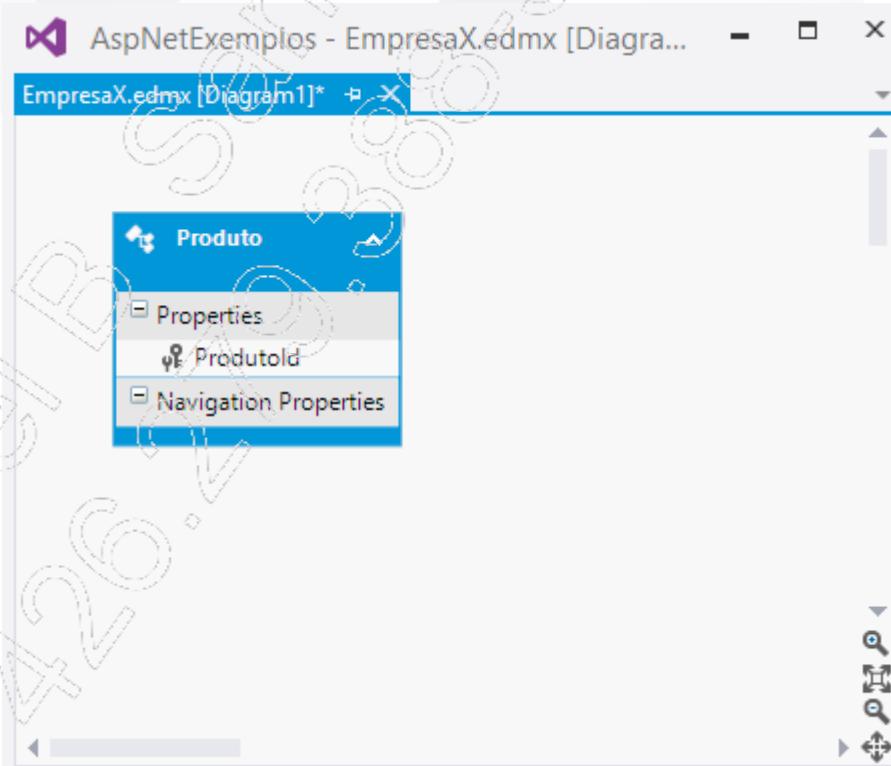


Visual Studio 2015 - ASP.NET com C# Acesso a dados

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.

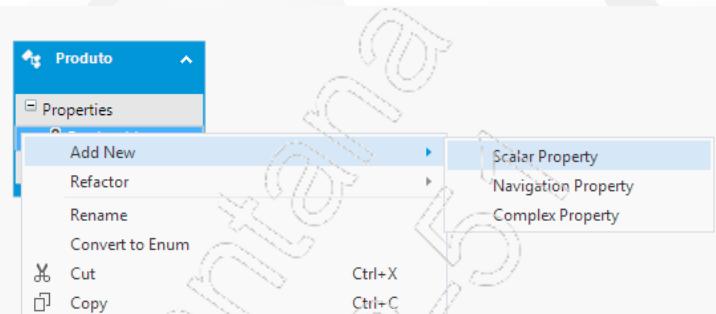
Ao término do preenchimento, o modelo aparecerá no EF Designer:



5.2.2. Adicionando propriedades

Clicando com o botão direito sobre o diagrama da classe **Produto** e, 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.



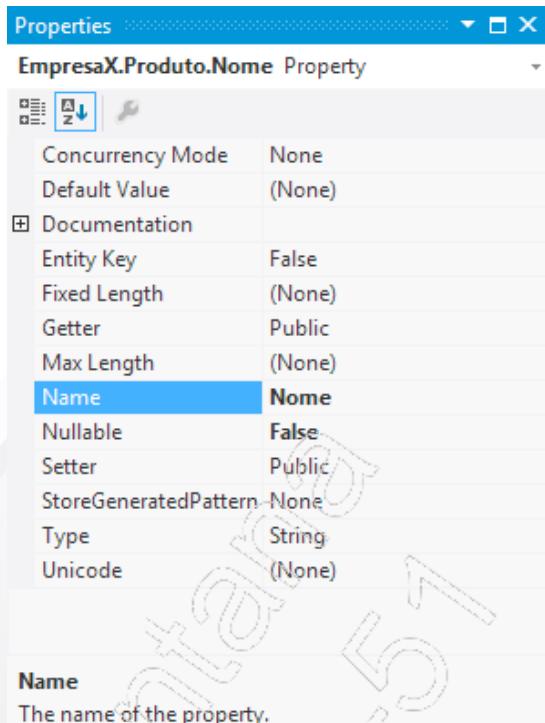
Apontando para uma propriedade e abrindo o menu de contexto, existe a opção **Properties**, em que é possível definir detalhes sobre este elemento. Os itens que podem ser definidos são os seguintes:

- **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**;

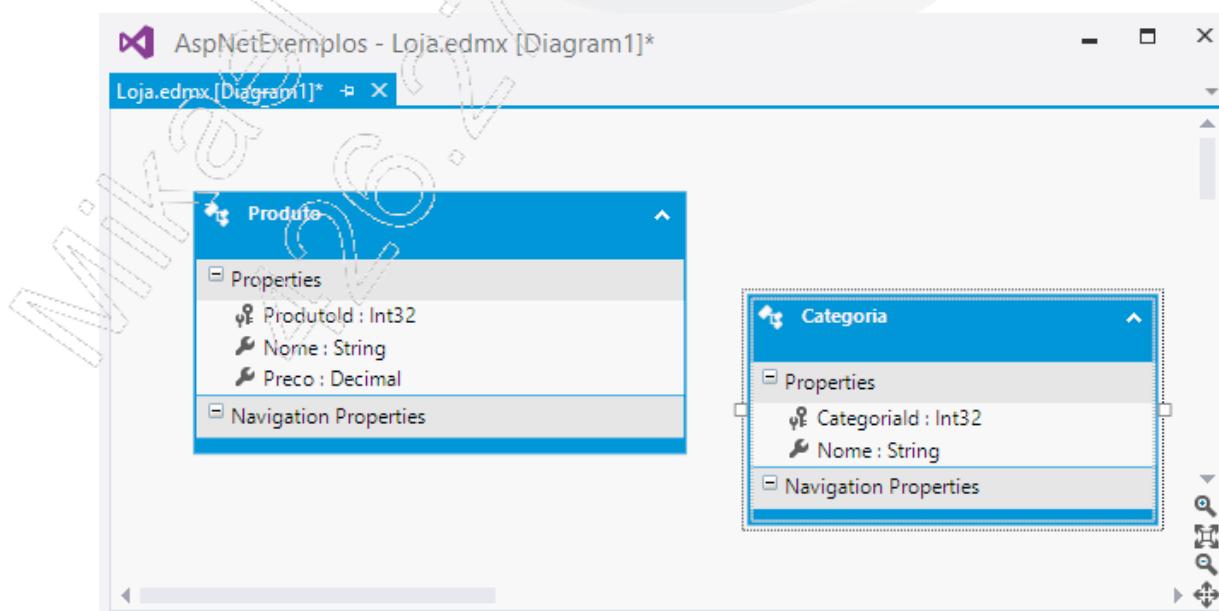
- **Default Value:** O valor padrão do campo. Este valor será inserido automaticamente nos novos registros, se nenhum valor for fornecido para este campo;
- **Documentation:** Estrutura com os campos **Long Description** para armazenar uma descrição detalhada do campo e **Summary** para uma descrição 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;

Entity Framework (Model/Database First)

- **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**.



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

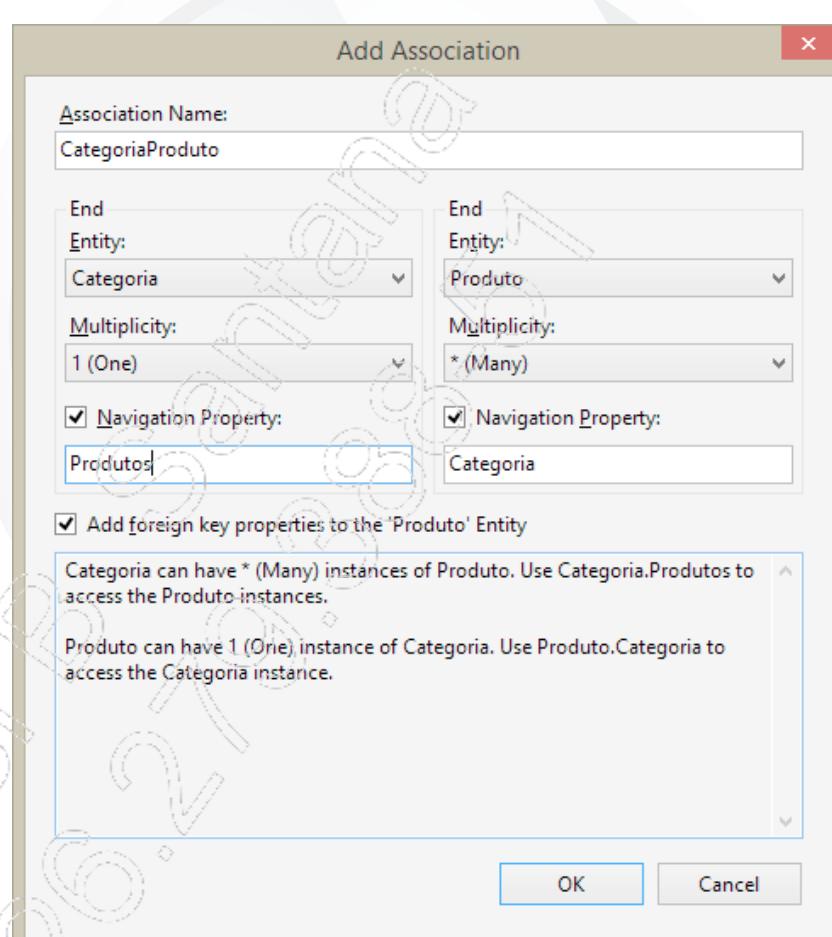


5.2.3. Adicionando associações

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**.

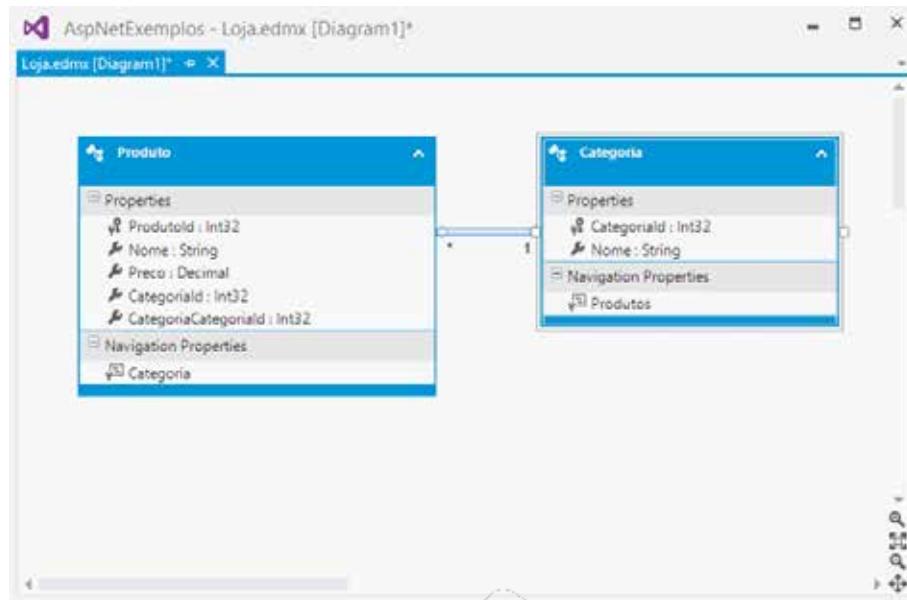
- **Associação um-para-muitos**

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



Entity Framework (Model/Database First)

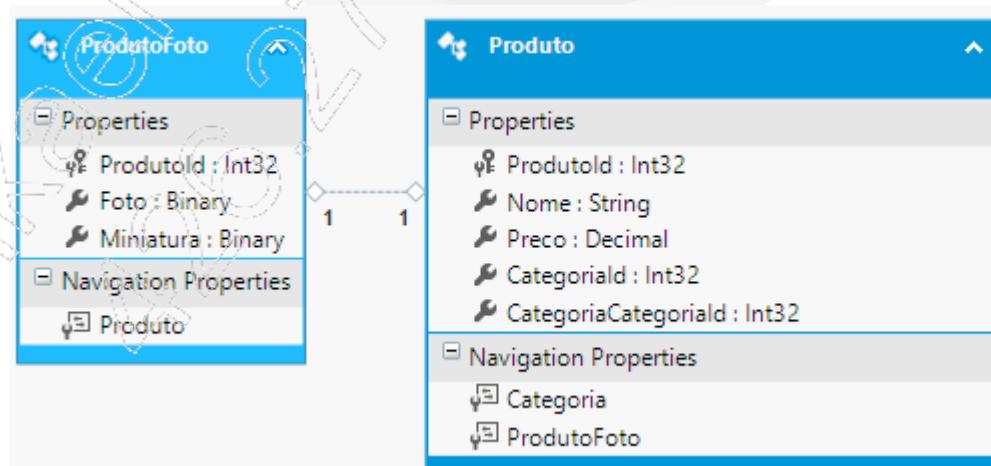
A propriedade **Categoria.Produtos** retorna a lista de produtos de uma categoria, e a propriedade **Produto.Categoria** retorna a categoria de um produto.



- **Associação um-para-um**

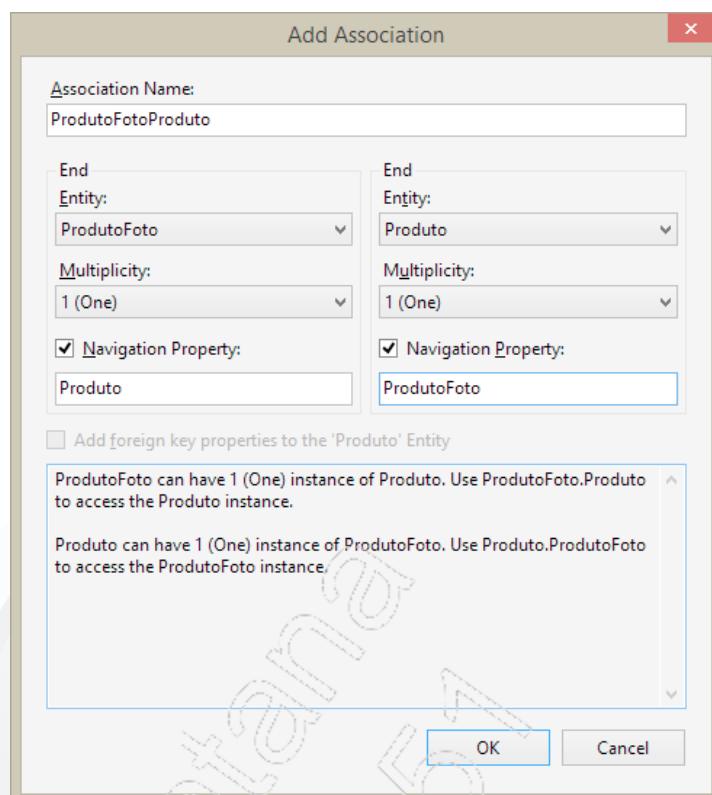
O tipo de associação na qual uma instância de uma classe se associa a outra instância de outra classe é chamado um-para-um.

Por exemplo, a classe **Produtos** exibe os dados de um produto. Uma outra classe, chamada **ProdutoFoto**, pode ser usada para armazenar as imagens de um produto.



Visual Studio 2015 - ASP.NET com C# Acesso a dados

Clicando com o botão direito no **model design** e escolhendo **Add Association**, a janela de associação pode ser preenchida com as informações das duas classes:



Note que ProdutoFoto se refere a uma instância de produto e Produto se refere a uma instância de ProdutoFoto.

A foto de um produto pode ser obtida com o seguinte código:

```
Byte[] foto = produto.ProdutoFoto.Foto;
```

O produto ao qual uma foto pertence pode ser obtido com o seguinte código:

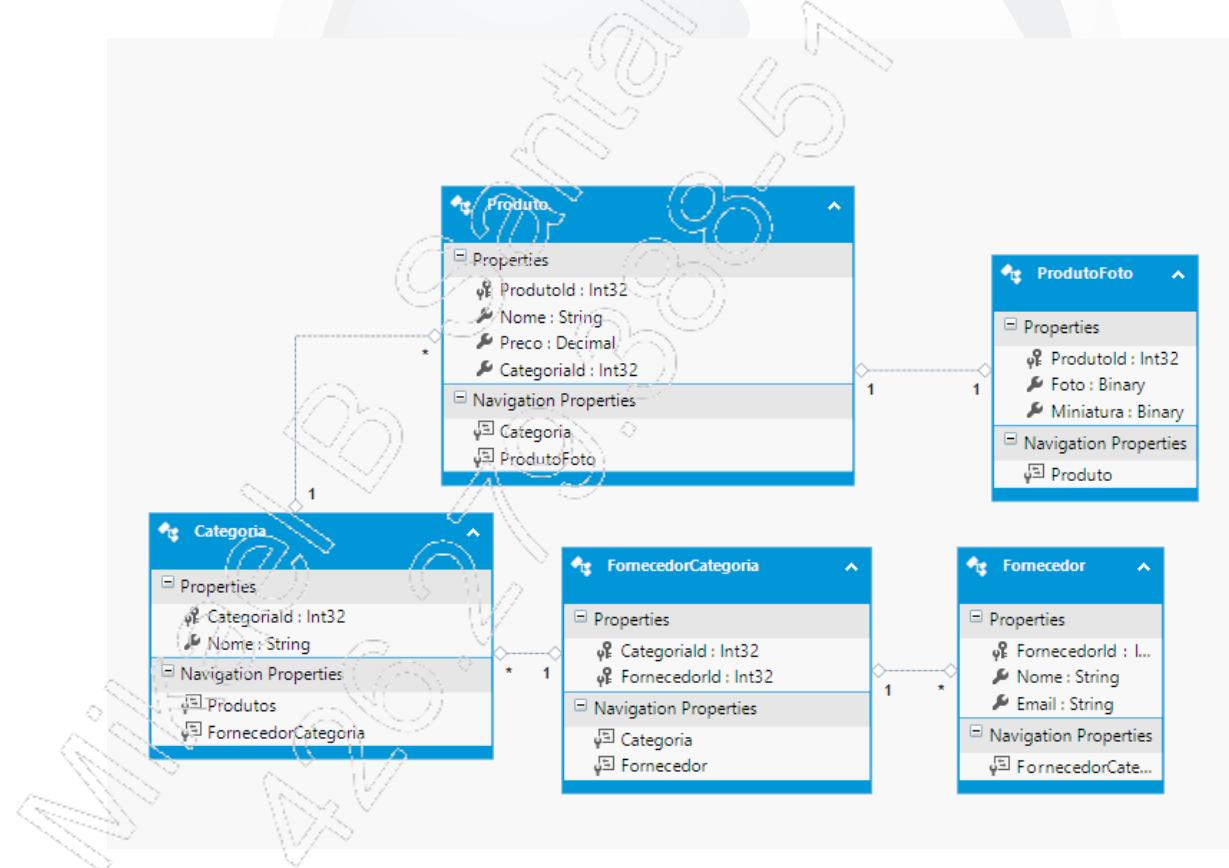
```
Produto p=produtoFoto.Produto.Foto;
```

- **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**.

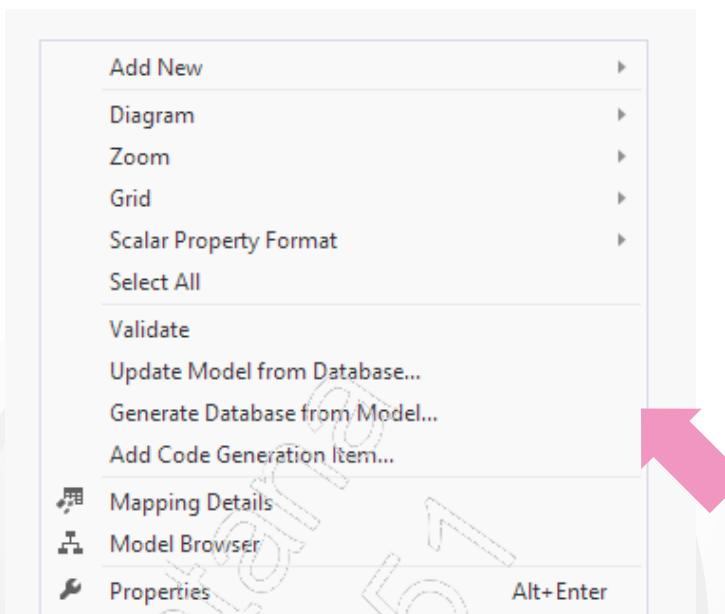


Para criar uma relação de **muitos-para-muitos** é necessária uma tabela (ou entidade) intermediária (existem outras formas, mas por enquanto o objetivo é ficar o mais próximo possível da estrutura de um banco de dados).

No exemplo anterior de uma relação de **muitos-para-muitos**, a entidade **FornecedorCategoria** relaciona cada fornecedor a diversas categorias e consequentemente cada categoria a diversos fornecedores.

5.2.4. Criando o banco de dados

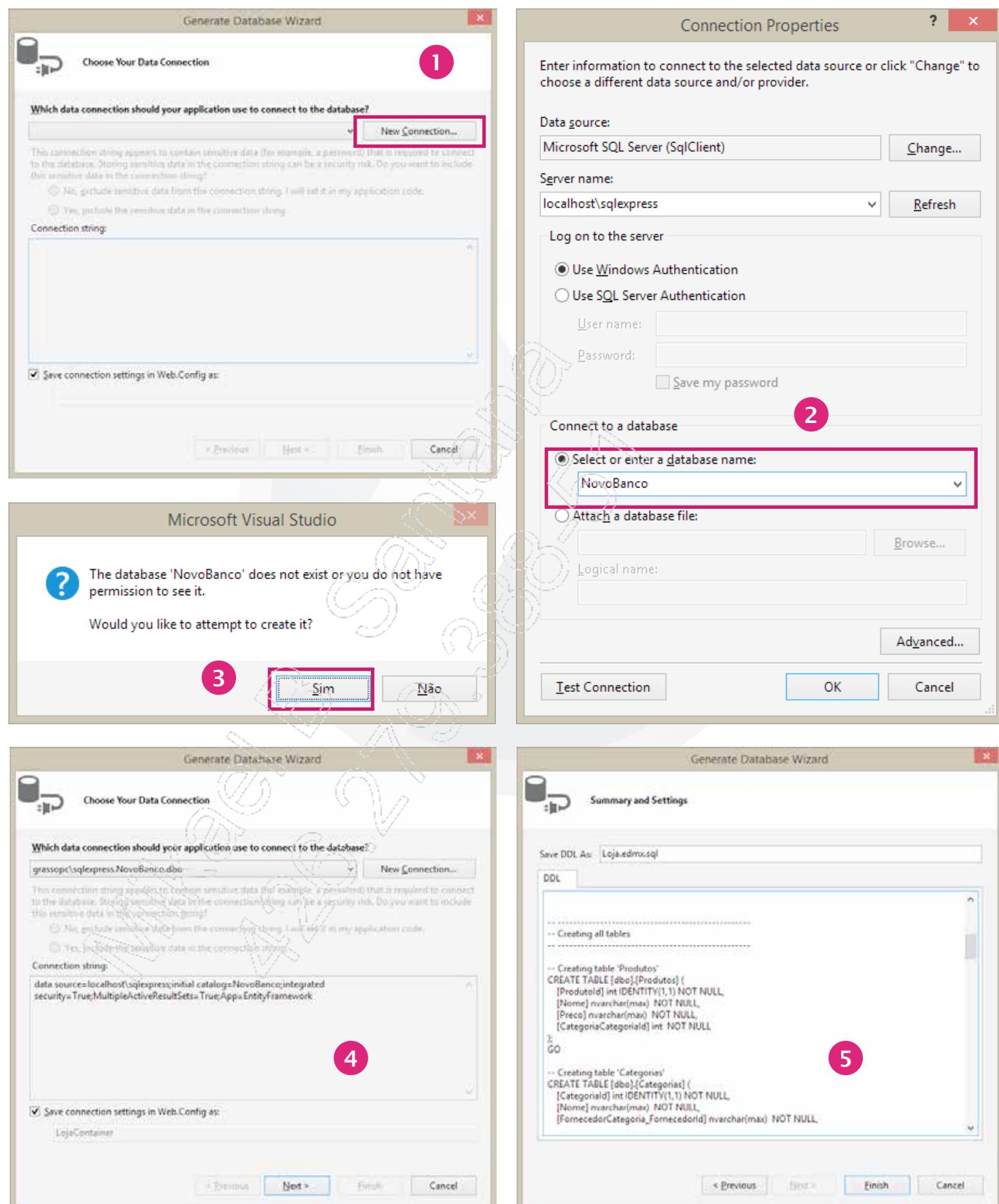
Uma vez criado o modelo, é hora de associá-lo a um banco de dados. Para criar o banco de dados, é necessário acionar o menu de contexto do EF Designer e escolher **Generate Database from Model**.



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.

Entity Framework (Model/Database First)

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.



Visual Studio 2015 - ASP.NET com C# Acesso a dados

O script de criação do banco é gerado. Neste ponto, pode ser executado o script, dentro do Visual Studio, usando o menu de contexto:

```
-- Creating table 'Produtos'
CREATE TABLE [dbo].[Produtos] (
    [ProdutoId] int IDENTITY(1,1) NOT NULL,
    [Nome] nvarchar(max) NOT NULL,
    [Preco] decimal(18,0) NOT NULL,
    [CategoriaCategoriaId] int NOT NULL
);
GO

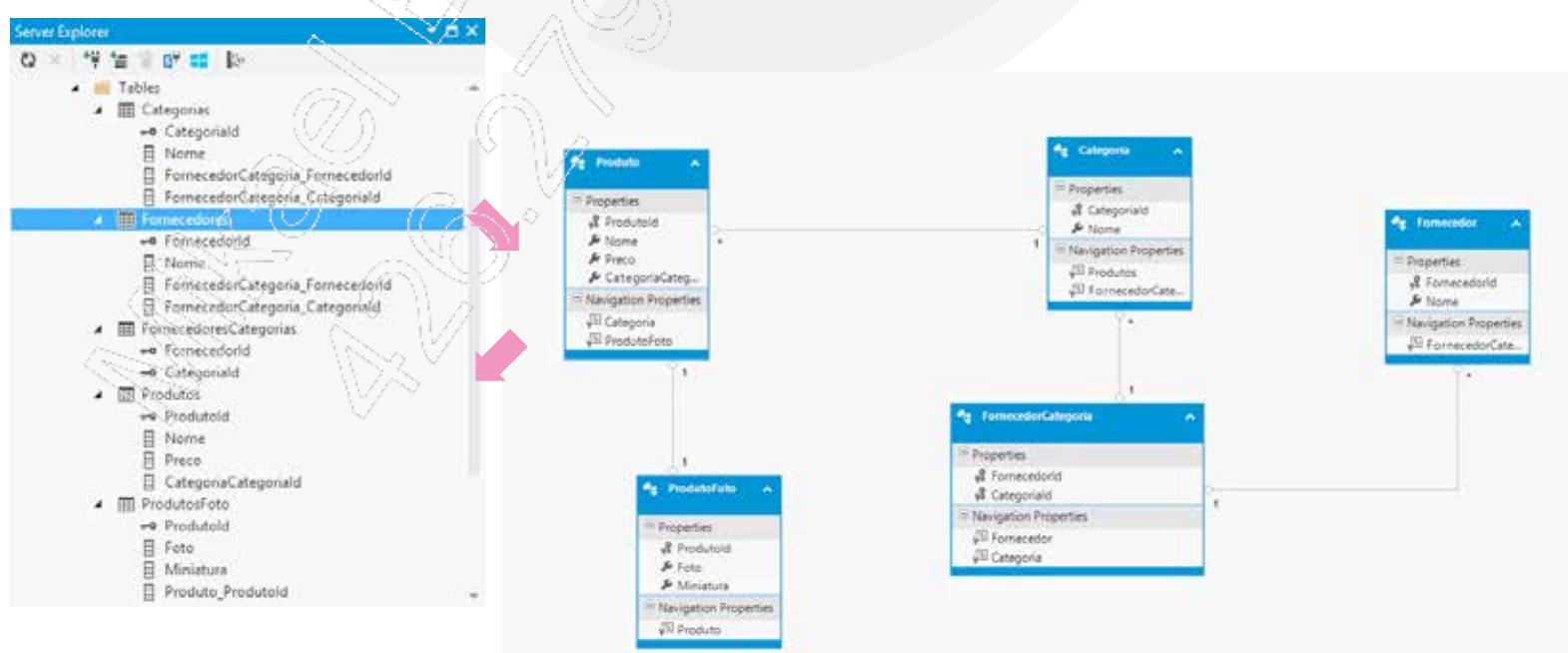
-- Creating table 'Categoria'
CREATE TABLE [dbo].[Categoria] (
    [CategoriaId] int IDENTITY(1,1) NOT NULL,
    [Nome] nvarchar(max) NOT NULL,
    [FornecedorCategoria_FornecedorId] int NOT NULL,
    [FornecedorCategoria_CategoriaId] int NOT NULL
);
```

00 % T-SQL ↑ Message
Command(s) completed successfully.

Context menu options shown (Execute is highlighted):

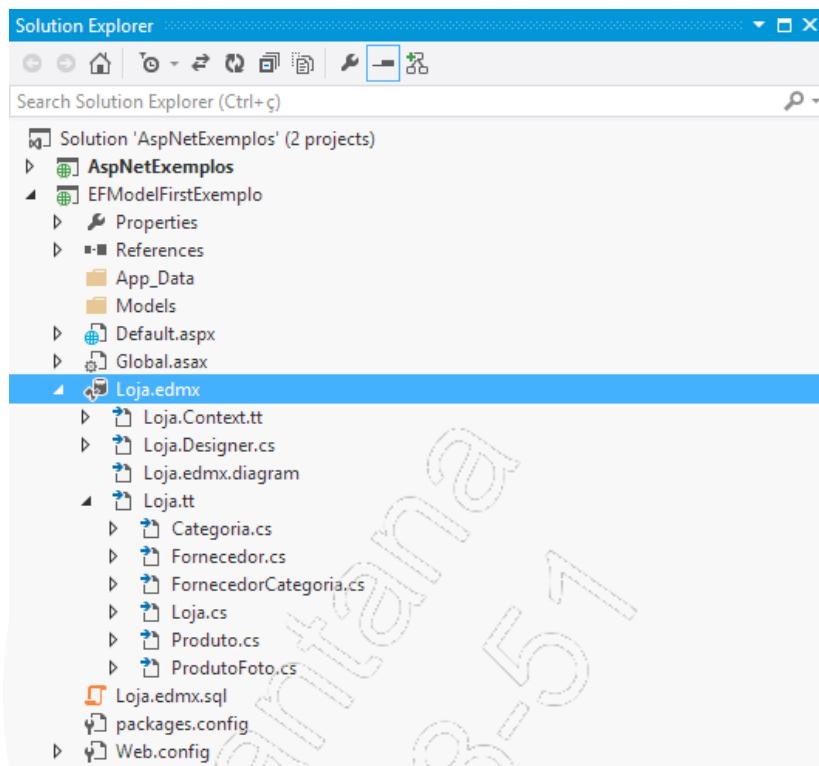
- Insert Snippet... Ctrl+K, Ctrl+X
- Surround With... Ctrl+K, Ctrl+S
- Peek Definition Alt+F12
- Run Flagged Threads To Cursor
- Cut Ctrl+X
- Copy Ctrl+C
- Paste Ctrl+V
- Outlining
- Intellisense Enabled
- Connection
- Execution Settings
- Execute Ctrl+Shift+E
- Execute With Debugger Alt+F5
- Cancel Execution Alt+Break
- Display Estimated Execution Plan Ctrl+D, E
- Parse
- Results To

Abrindo o banco de dados no Server Explorer, é possível comparar as tabelas criadas com o modelo de domínio:



5.2.5. Entendendo os arquivos gerados

Observando pelo Solution Explorer, encontramos vários arquivos que fazem parte da estrutura do modelo gerado pelo .NET Framework:



- **Loja.edmx**: É o arquivo do diagrama dos modelos de dados;
- **Loja.Context.tt**: É um template (modelo) para criar a classe derivada de `DbContext`;
- **Loja.Designer.cs**: Contém código relativo à tela de designer. No padrão do EF 6.0, esse arquivo fica vazio;
- **Loja.edmx.diagram**: Contém um XML para a ferramenta Designer posicionar os elementos no mesmo lugar em que estavam da última vez que foram gravados;
- **Loja.tt**: É um modelo para criar as classes de entidades. Relacionados ao arquivo **Loja.tt** estão os arquivos de entidades criados pelo Entity Framework como **Categoria.cs**, **Produto.cs**, **Fornecedor.cs** e os outros tipos do modelo;

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Vejamos a classe **Categoria** criada pelo Entity Framework:

```
//-----
// <auto-generated>
//     This code was generated from a template.
//
//     Manual changes to this file may cause unexpected
//         behavior in your application.
//     Manual changes to this file will be
//         overwritten if the code is regenerated.
// </auto-generated>
//-----
```

```
namespace EFModelFirstExemplo
{
    using System;
    using System.Collections.Generic;

    public partial class Categoria
    {

        public Categoria()
        {
            this.Produtos = new HashSet<Produto>();
        }

        public int CategoriaId { get; set; }

        public string Nome { get; set; }

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

        public virtual FornecedorCategoria FornecedorCategoria
        { get; set; }
    }
}
```

A seguir, alguns detalhes sobre o código gerado:

- A classe é marcada como **partial** para facilitar a inclusão de código personalizado. Basta criar uma classe com o mesmo nome e incluir os métodos e propriedades extras;
- As coleções e propriedades de navegação estão marcadas como **Virtual**. O Entity Framework usa isso para criar o que é chamado de **Lazy Load**, em que os dados de navegação são carregados apenas quando solicitados;
- As coleções utilizam a interface **Icollection<T>** e são inicializadas no construtor da classe. O EF aceita apenas coleções que implementam essa interface ou interfaces derivadas dela.
- **Loja.edmx.sql**: Arquivo com o script para a criação das tabelas no banco de dados.

```
-- Entity Designer DDL Script for SQL Server 2005, 2008, 2012 and Azure
```

```
-- Date Created: 09/05/2014 00:16:55
```

```
-- Generated from EDMX file: .... Loja.edmx
```

```
SET QUOTED_IDENTIFIER OFF;
GO
USE [NovoBanco];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO
```

Abre a conexão com
o banco

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
-- -----  
-- Dropping existing FOREIGN KEY constraints  
-- -----  
  
IF OBJECT_ID(N'[dbo].[FK_CategoriaProduto]', 'F') IS NOT NULL  
    ALTER TABLE [dbo].[Produtos] DROP CONSTRAINT [FK_CategoriaProduto];  
GO  
IF OBJECT_ID(N'[dbo].[FK_FornecedorCategoriaFornecedor]', 'F') IS NOT NULL  
    ALTER TABLE [dbo].[Fornecedores] DROP CONSTRAINT [FK_  
FornecedorCategoriaFornecedor];  
GO  
IF OBJECT_ID(N'[dbo].[FK_FornecedorCategoriaCategoria]', 'F') IS NOT NULL  
    ALTER TABLE [dbo].[Categorias] DROP CONSTRAINT [FK_  
FornecedorCategoriaCategoria];  
GO  
IF OBJECT_ID(N'[dbo].[FK_ProdutoFotoProduto]', 'F') IS NOT NULL  
    ALTER TABLE [dbo].[ProdutosFoto] DROP CONSTRAINT [FK_ProdutoFotoProduto];  
GO  
  
-- -----  
-- Dropping existing tables  
-- -----  
  
IF OBJECT_ID(N'[dbo].[Produtos]', 'U') IS NOT NULL  
    DROP TABLE [dbo].[Produtos];  
GO  
IF OBJECT_ID(N'[dbo].[Categorias]', 'U') IS NOT NULL  
    DROP TABLE [dbo].[Categorias];  
GO  
IF OBJECT_ID(N'[dbo].[Fornecedores]', 'U') IS NOT NULL  
    DROP TABLE [dbo].[Fornecedores];  
GO  
IF OBJECT_ID(N'[dbo].[ProdutosFoto]', 'U') IS NOT NULL  
    DROP TABLE [dbo].[ProdutosFoto];  
GO  
IF OBJECT_ID(N'[dbo].[FornecedoresCategorias]', 'U') IS NOT NULL  
    DROP TABLE [dbo].[FornecedoresCategorias];  
GO
```

Exclui os índices

Exclui as tabelas

```
-- Creating all tables
-- Creating table 'Produtos'
CREATE TABLE [dbo].[Produtos] (
    [ProdutoId] int IDENTITY(1,1) NOT NULL,
    [Nome] nvarchar(max) NOT NULL,
    [Preco] decimal(18,0) NOT NULL,
    [CategoriaCategoriaId] int NOT NULL
);
GO

-- Creating table 'Categorias'
CREATE TABLE [dbo].[Categorias] (
    [CategoriaId] int IDENTITY(1,1) NOT NULL,
    [Nome] nvarchar(max) NOT NULL,
    [FornecedorCategoria_FornecedorId] int NOT NULL,
    [FornecedorCategoria_CategoriaId] int NOT NULL
);
GO

-- Creating table 'Fornecedores'
CREATE TABLE [dbo].[Fornecedores] (
    [FornecedorId] int IDENTITY(1,1) NOT NULL,
    [Nome] nvarchar(max) NOT NULL,
    [FornecedorCategoria_FornecedorId] int NOT NULL,
    [FornecedorCategoria_CategoriaId] int NOT NULL
);
GO

-- Creating table 'ProdutosFoto'
CREATE TABLE [dbo].[ProdutosFoto] (
    [ProdutoId] int IDENTITY(1,1) NOT NULL,
    [Foto] varbinary(max) NOT NULL,
    [Miniatura] varbinary(max) NOT NULL,
    [Produto_ProdutoId] int NOT NULL
);
GO

-- Creating table 'FornecedoresCategorias'
CREATE TABLE [dbo].[FornecedoresCategorias] (
    [FornecedorId] int NOT NULL,
    [CategoriaId] int NOT NULL
);
GO
```

Cria as tabelas

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
-- Creating all PRIMARY KEY constraints
-----
-- Creating primary key on [ProdutoId] in table 'Produtos'
ALTER TABLE [dbo].[Produtos]
ADD CONSTRAINT [PK_Produtos]
    PRIMARY KEY CLUSTERED ([ProdutoId] ASC);
GO

-- Creating primary key on [CategoriaId] in table 'Categorias'
ALTER TABLE [dbo].[Categorias]
ADD CONSTRAINT [PK_Categorias]
    PRIMARY KEY CLUSTERED ([CategoriaId] ASC);
GO

-- Creating primary key on [FornecedorId] in table 'Fornecedores'
ALTER TABLE [dbo].[Fornecedores]
ADD CONSTRAINT [PK_Fornecedores]
    PRIMARY KEY CLUSTERED ([FornecedorId] ASC);
GO

-- Creating primary key on [ProdutoId] in table 'ProdutosFoto'
ALTER TABLE [dbo].[ProdutosFoto]
ADD CONSTRAINT [PK_ProdutosFoto]
    PRIMARY KEY CLUSTERED ([ProdutoId] ASC);
GO

-- Creating primary key on [FornecedorId], [CategoriaId] in table
-- 'FornecedoresCategorias'
ALTER TABLE [dbo].[FornecedoresCategorias]
ADD CONSTRAINT [PK_FornecedoresCategorias]
    PRIMARY KEY CLUSTERED ([FornecedorId], [CategoriaId] ASC);
GO
```

Cria as chaves primárias

Entity Framework (Model/Database First)

```
-- Creating all FOREIGN KEY constraints
-----
-- Creating foreign key on [CategoriaCategoriaId] in table 'Produtos'
ALTER TABLE [dbo].[Produtos]
ADD CONSTRAINT [FK_CategoriaProduto]
FOREIGN KEY ([CategoriaCategoriaId])
REFERENCES [dbo].[Categorias]
([CategoriaId])
ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY 'FK_CategoriaProduto'
CREATE INDEX [IX_FK_CategoriaProduto]
ON [dbo].[Produtos]
([CategoriaCategoriaId]);
GO

-- Creating foreign key on [FornecedorCategoria_FornecedorId],
[FornecedorCategoria_CategoriaId] in table 'Fornecedores'
ALTER TABLE [dbo].[Fornecedores]
ADD CONSTRAINT [FK_FornecedorCategoriaFornecedor]
FOREIGN KEY ([FornecedorCategoria_FornecedorId], [FornecedorCategoria_CategoriaId])
REFERENCES [dbo].[FornecedoresCategorias]
([FornecedorId], [CategoriaId])
ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY 'FK_FornecedorCategoriaFornecedor'
CREATE INDEX [IX_FK_FornecedorCategoriaFornecedor]
ON [dbo].[Fornecedores]
([FornecedorCategoria_FornecedorId], [FornecedorCategoria_CategoriaId]);
GO

-- Creating foreign key on [FornecedorCategoria_FornecedorId],
[FornecedorCategoria_CategoriaId] in table 'Categorias'
ALTER TABLE [dbo].[Categorias]
ADD CONSTRAINT [FK_FornecedorCategoriaCategoria]
FOREIGN KEY ([FornecedorCategoria_FornecedorId], [FornecedorCategoria_CategoriaId])
REFERENCES [dbo].[FornecedoresCategorias]
([FornecedorId], [CategoriaId])
ON DELETE NO ACTION ON UPDATE NO ACTION;
GO
```

Cria as chaves estrangeiras (relacionamentos)

```
-- Creating non-clustered index for FOREIGN KEY 'FK_FornecedorCategoriaCategoria'
CREATE INDEX [IX_FK_FornecedorCategoriaCategoria]
ON [dbo].[Categorias]
([FornecedorCategoria_FornecedorId], [FornecedorCategoria_CategoriaId]);
GO

-- Creating foreign key on [Produto_ProdutoId] in table 'ProdutosFoto'
ALTER TABLE [dbo].[ProdutosFoto]
ADD CONSTRAINT [FK_ProdutoFotoProduto]
FOREIGN KEY ([Produto_ProdutoId])
REFERENCES [dbo].[Produtos]
([ProdutoId])
ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating non-clustered index for FOREIGN KEY 'FK_ProdutoFotoProduto'
CREATE INDEX [IX_FK_ProdutoFotoProduto]
ON [dbo].[ProdutosFoto]
([Produto_ProdutoId]);
GO

--- -----
-- Script has ended
---
```

Cria os índices

5.2.6. 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. Nesse caso, não há diferença entre o modo **Code First** e o modo **Model First**. O exemplo a seguir mostra dados de um produto em uma página .aspx usando **Web Forms**.

- Página .aspx

```
<h1>Exemplo de uso do Entity Framework</h1>
<asp:Label ID="mensagemLabel" runat="server"></asp:Label>
<br />
<asp:GridView ID="gv" runat="server"></asp:GridView>
```

- Código-fonte da página (no evento Page_Load)

```
var db = new LojaContainer();
List<Categoria> lista = db.Categorias.ToList();
gv.DataSource = lista;
gv.DataBind();
```

Vejamos o resultado:



5.3. Database First

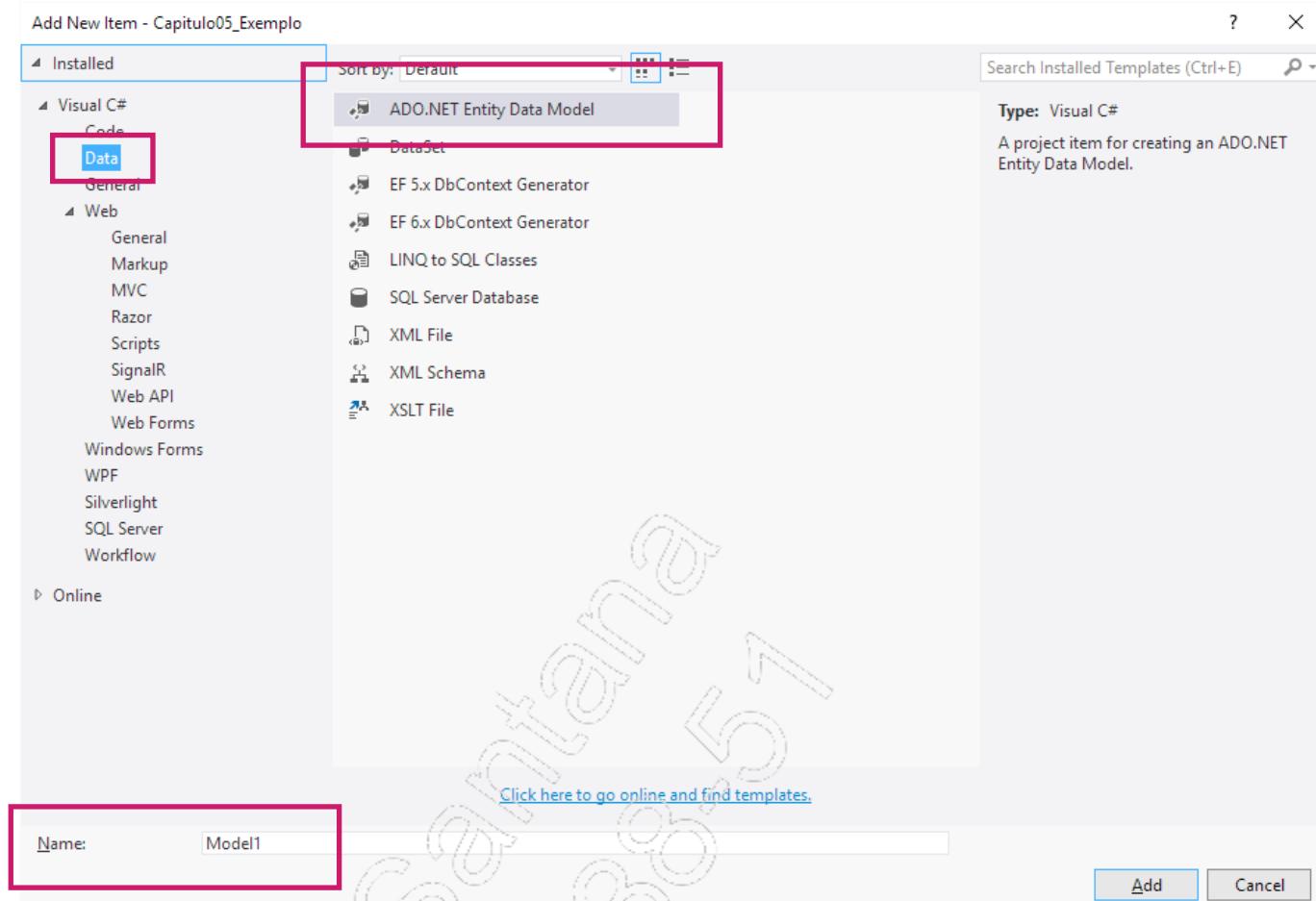
Outro modo de criar o modelo de domínio é usar um banco de dados existente e deixar o Entity Framework gerar as classes. Automaticamente, tabelas, campos, relacionamentos e restrições são mapeados e se tornam propriedades das classes geradas.

Siga os passos adiante:

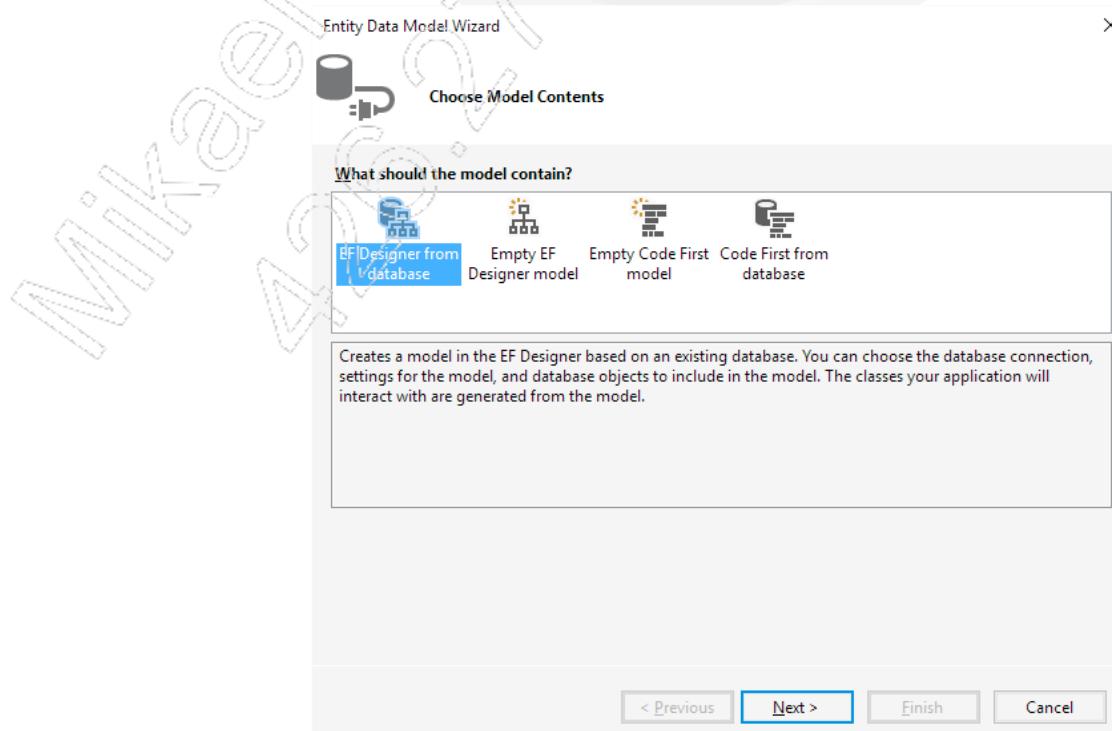
1. Em um projeto do Visual Studio, escolha, no menu, a opção **Project / Add New Item**;

Visual Studio 2015 - ASP.NET com C# Acesso a dados

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:

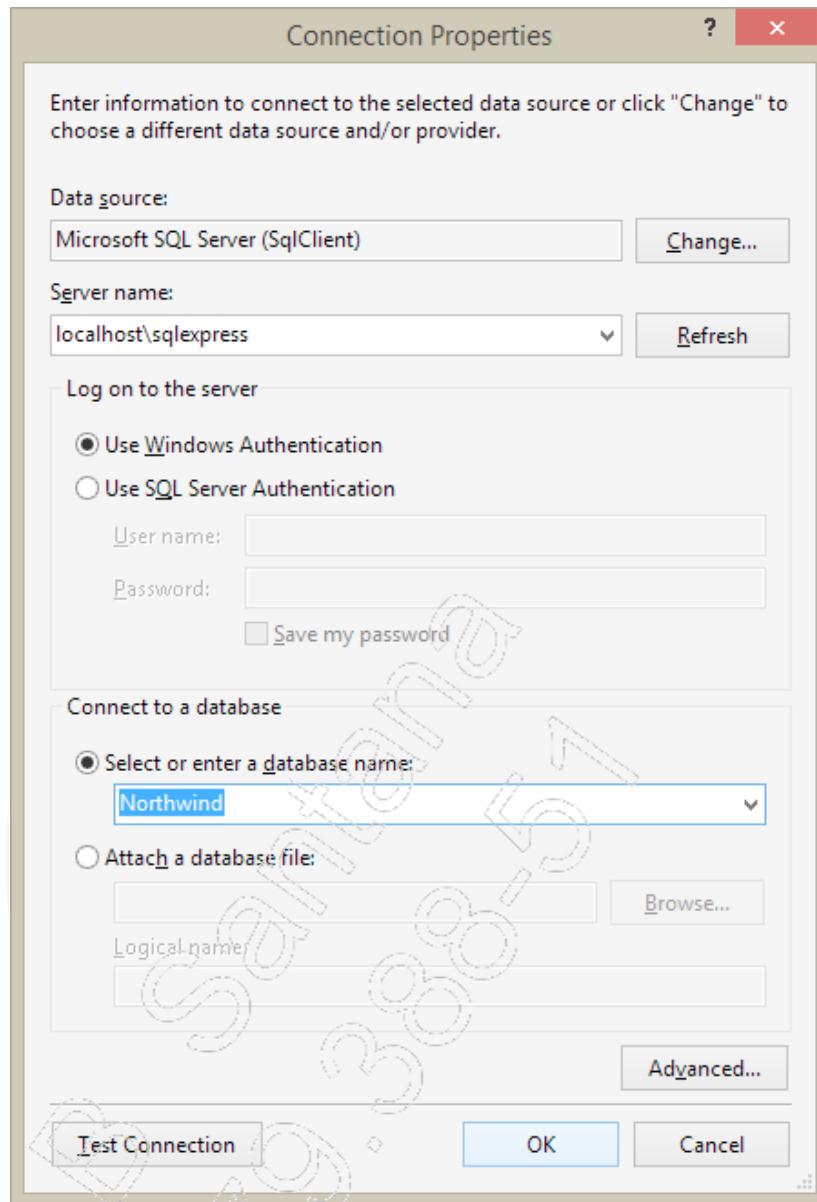


3. Na janela seguinte, escolha **EF Designer from database**:



Entity Framework (Model/Database First)

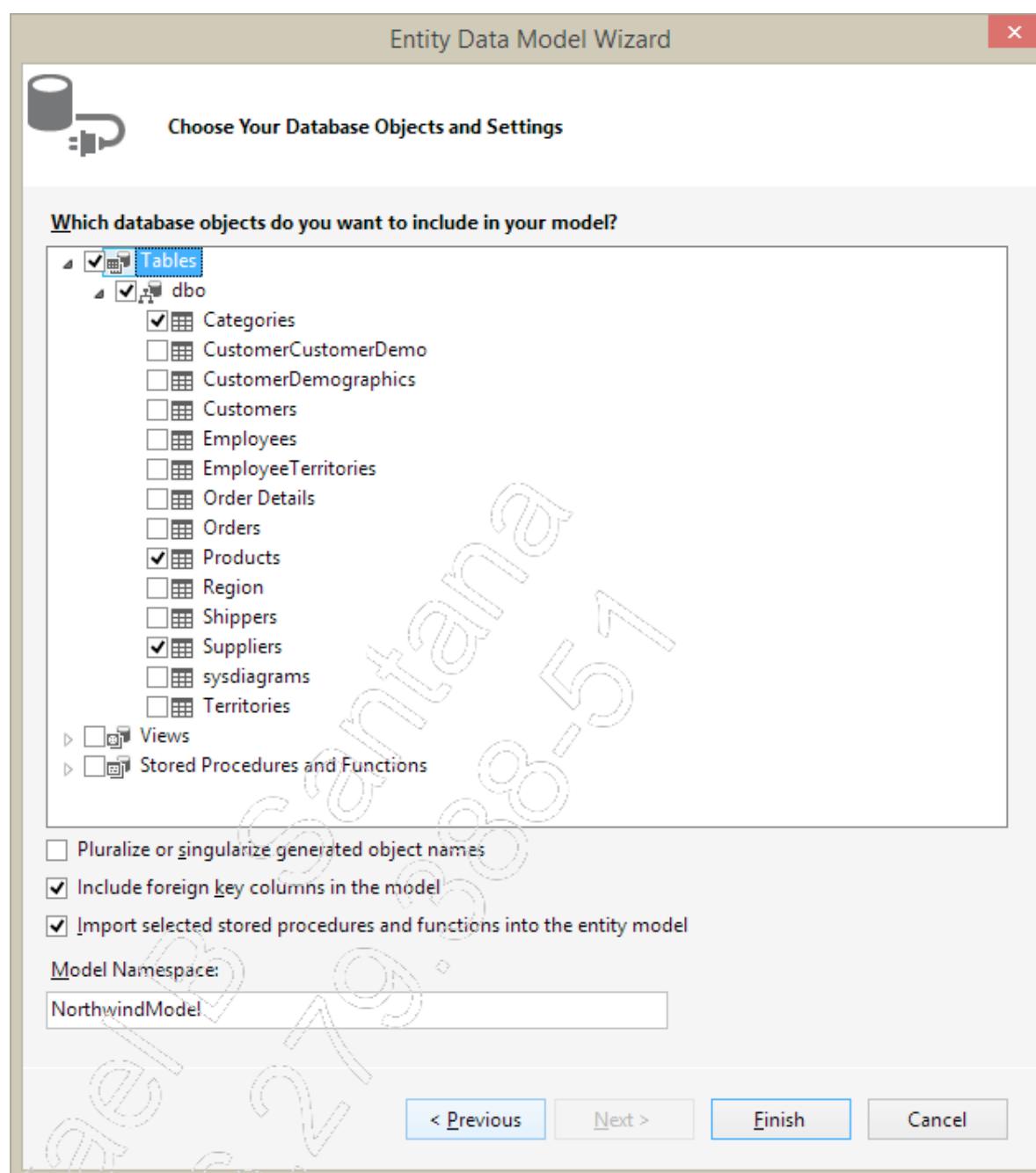
4. Na janela de conexão, escolha o banco de dados;



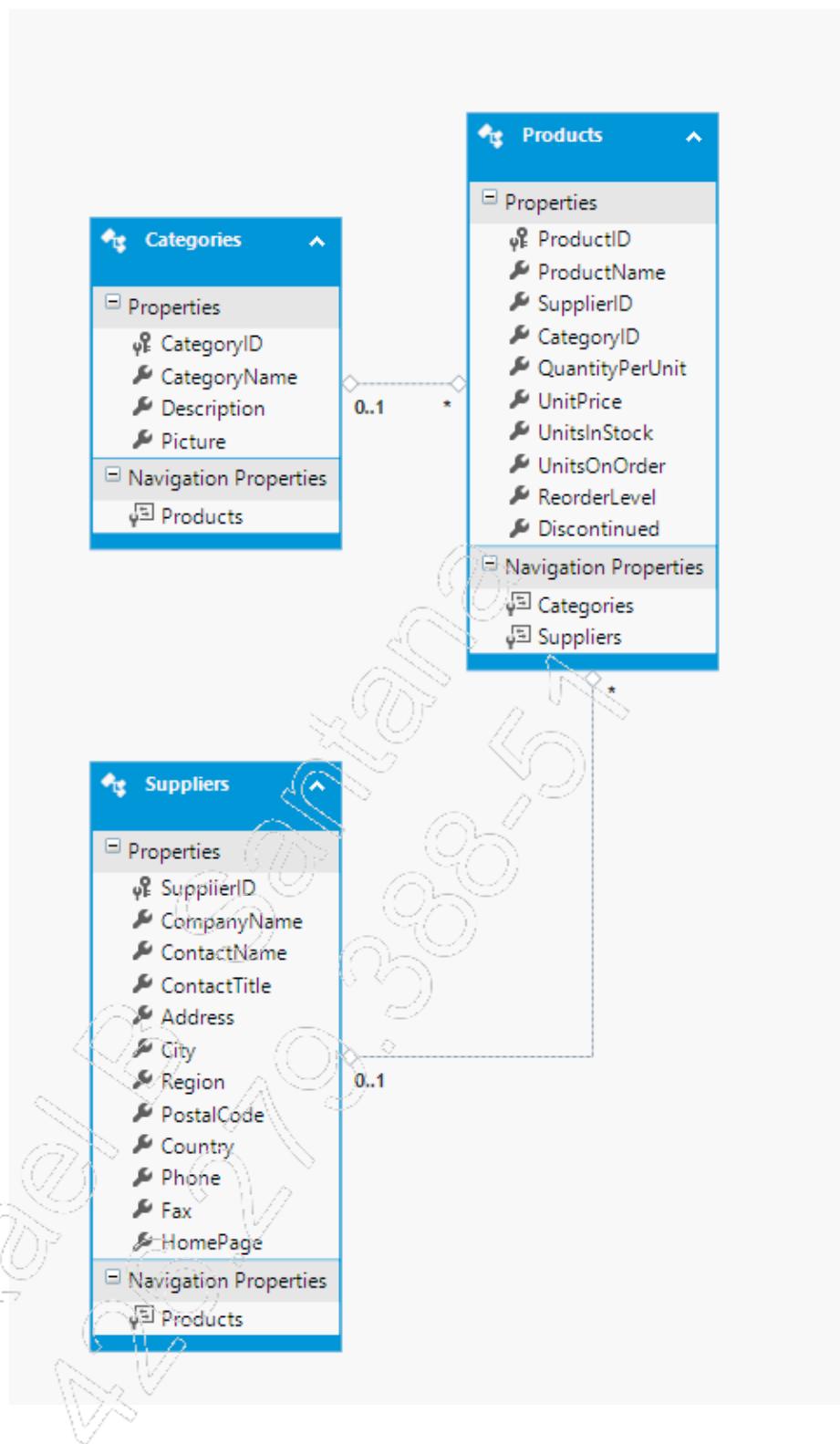
5. A próxima janela permite selecionar quais objetos serão inseridos no modelo. Um modelo de domínio não precisa ter todo o banco de dados, mas apenas as tabelas que fazem sentido juntas. É possível criar diversos modelos para um banco de dados. No banco de dados **Northwind**, por exemplo, faz sentido incluir as tabelas **Products**, **Suppliers** e **Categories** se o objetivo é manipular a tabela de produtos, pois essa tabela precisa dessas referências. Outra combinação seria as tabelas **Orders**, **Order Details**, **Customers**, **Employees**, **Products** e **Shippers**, porque essas tabelas fazem parte do pedido.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

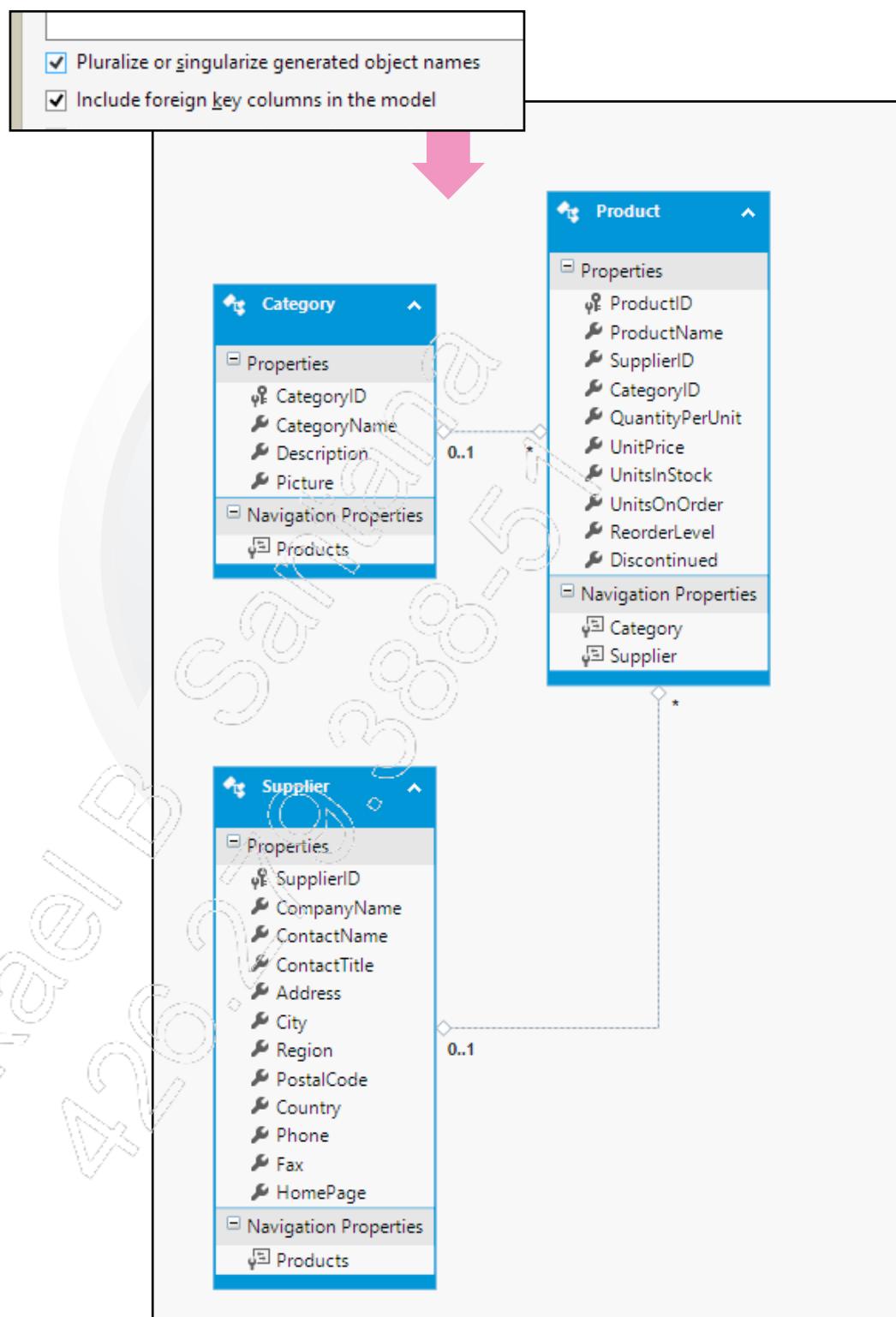
Neste exemplo, serão selecionadas as tabelas **Products**, **Categories** e **Suppliers** (produtos, categorias, e fornecedores):



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



7. É possível alterar o modo como são criadas as classes que representam as tabelas do banco. Na janela de escolha dos objetos que farão parte do modelo, existe a opção **Pluralize or singularize generated object names**. Se estiver marcada, os nomes de tabelas sempre ficarão no plural, e o nome das entidades, no singular:



Entity Framework (Model/Database First)

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

```
//  
//Produtos  
//  
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 **Products**, 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**.

```
//  
//Categorias de produtos  
  
public partial class Category  
{  
    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; }  
  
    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.

```
//  
//Fornecedores  
//  
public partial class Supplier  
{  
    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; }  
  
    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 EFModelFirstExemplo
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

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

        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 NorthwindEntities1():base("name=NorthwindEntities")
{}
```

- No **Web.Config**

```
<connectionStrings>
  <add name="NorthwindEntities" connectionString="..." />
</connectionStrings>
```

5.3.1. Usando o modelo criado

Uma vez que o modelo esteja criado, o modo de usar é exatamente igual ao modo **Code First** ou **Model First**. A classe derivada de **DbContext** fornece acesso aos dados, lendo e gravando informações no banco de dados usando as classes de mapeamento criadas.

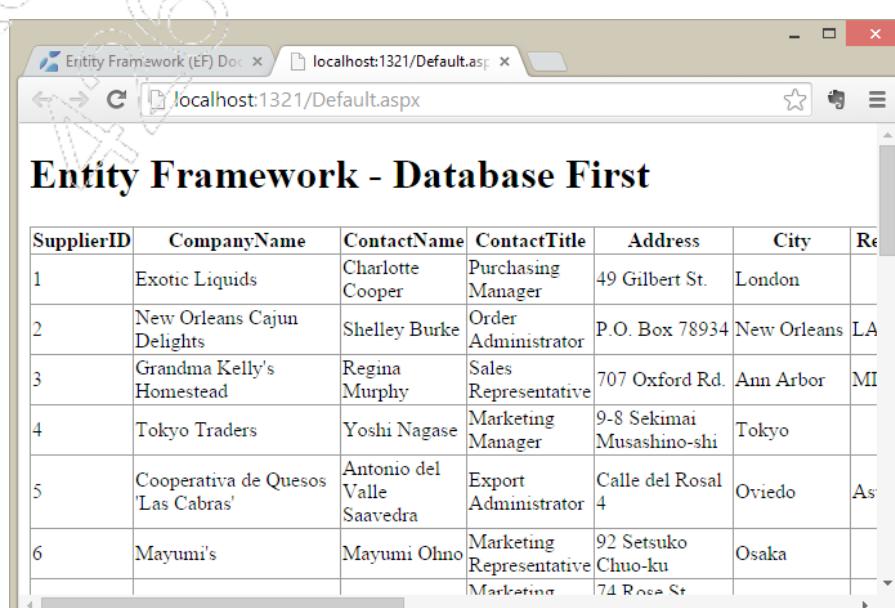
- Página **.aspx**

```
<h1>Entity Framework - Database First</h1>
<asp:GridView runat="server" ID="gv"></asp:GridView>
```

- Código-fonte da página (no evento **Page_Load**)

```
var db = new NorthwindEntities();
gv.DataSource = db.Suppliers.ToList();
gv.DataBind();
```

Vejamos o resultado:



The screenshot shows a Microsoft Edge browser window with the title "Entity Framework - Database First". The address bar shows "localhost:1321/Default.aspx". The main content area displays a grid of supplier information. The columns are: SupplierID, CompanyName, ContactName, ContactTitle, Address, City, and Region. The data is as follows:

SupplierID	CompanyName	ContactName	ContactTitle	Address	City	Region
1	Exotic Liquids	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	UK
2	New Orleans Cajun Delights	Shelley Burke	Order Administrator	P.O. Box 78934	New Orleans	LA
3	Grandma Kelly's Homestead	Regina Murphy	Sales Representative	707 Oxford Rd.	Ann Arbor	MI
4	Tokyo Traders	Yoshi Nagase	Marketing Manager	9-8 Sekimai Musashino-shi	Tokyo	JP
5	Cooperativa de Quesos 'Las Cabras'	Antonio del Valle Saavedra	Export Administrator	Calle del Rosal 4	Oviedo	AS
6	Mayumi's	Mayumi Ohno	Marketing Representative	92 Setsuko Chuo-ku	Osaka	JP

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 extraiendo 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;
- Os principais elementos a serem adicionados no EF Designer são: **Entity**, **Properties** e **Association**;
- Marcar a opção **Pluralize or singularize generated object names** faz com que o EF Designer automaticamente crie as classes com o nome no singular e as coleções no plural, independentemente de como estão gravadas no banco.

5

Entity Framework (Model/Database First)

Teste seus conhecimentos



IMPACTA
EDITORA

1. Qual é o modo de trabalho em que a criação das classes de modelo é realizada automaticamente a partir da estrutura de um banco de dados?

- a) Code First
- b) Database First
- c) Model First
- d) Migration
- e) Entity Framework Designer

2. Como são chamadas as propriedades que fazem referência a outras entidades?

- a) Scalar Properties
- b) Complex Properties
- c) Foreign Key Properties
- d) Reference Properties
- e) Navigation Properties

3. Considere uma empresa que vende imóveis, em que cada vendedor é responsável por um grupo de imóveis. Um vendedor não pode vender um imóvel que não esteja no grupo atribuído a ele. Esse tipo de relação, do vendedor para os imóveis que ele vende, é de qual tipo?

- a) Um-para-um
- b) Um-para-muitos
- c) Muitos-para-muitos
- d) Com as informações apresentadas não é possível classificar o tipo de associação.
- e) Pode ser um-para-muitos ou muitos-para-muitos.

4. Considere uma tabela de um banco de dados SQL Server com um campo não obrigatório chamado Cidade, do tipo string, e outro campo, também não obrigatório, chamado Idade, do tipo inteiro. Como o Entity Framework cria esses dois campos nas classes de modelo de domínio?

- a) String e int.
- b) Nullable<string> e int.
- c) Nullable<string> e Nullable<int>.
- d) String e Nullable<int>.
- e) String e Nullable<long>.

5. Quais são os principais elementos a serem adicionados em um modelo de dados?

- a) Connection, Relations e Properties.
- b) Entity, Properties e Association.
- c) Entity e Tables.
- d) Tables e Relations.
- e) Tables, Relations e Fields.

5

Entity Framework (Model/Database First)

Mãos à obra!



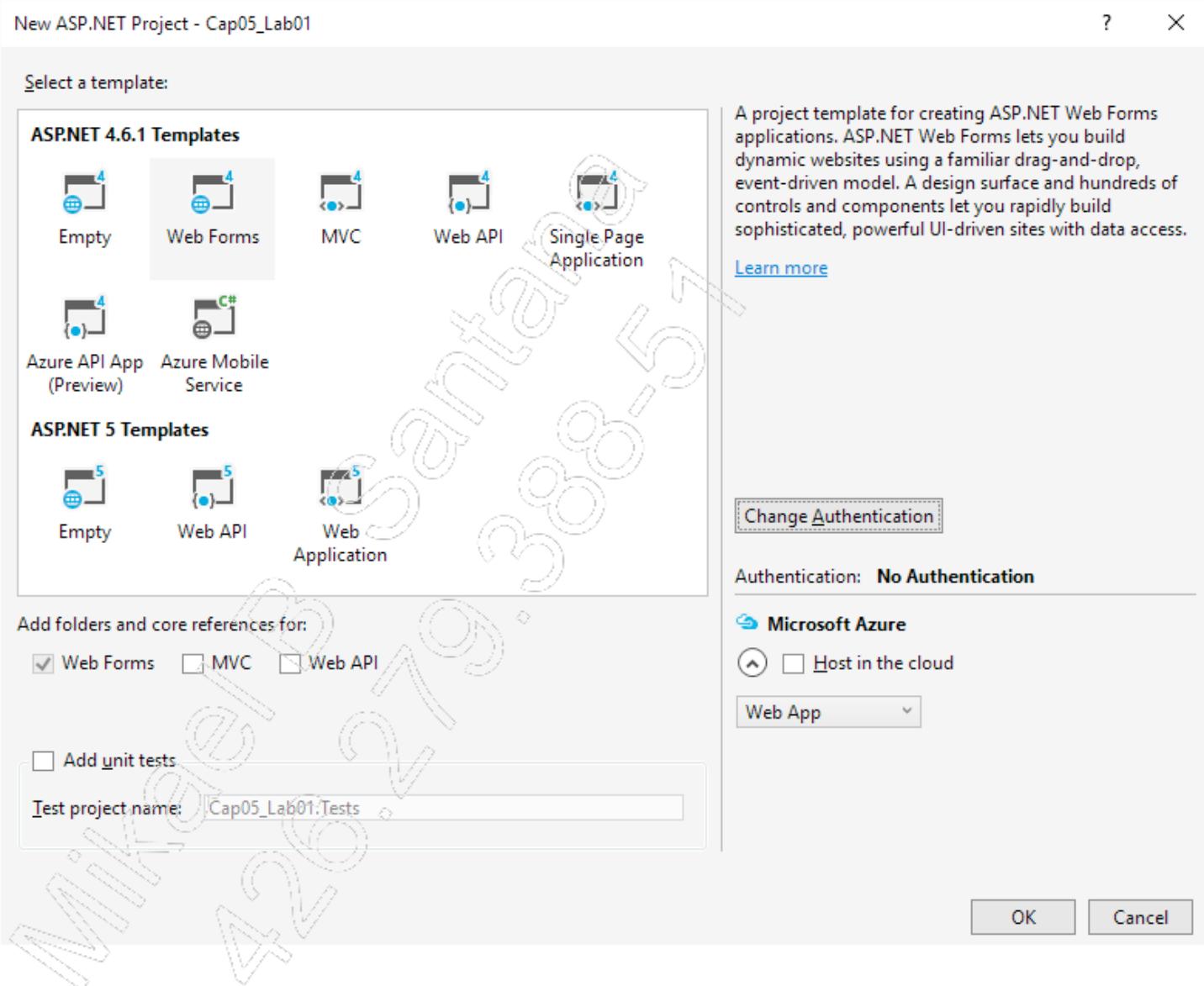
IMPACTA
EDITORA

Laboratório 1

Neste laboratório, usaremos o Entity Framework no modo **Database First** para exibir os dados do banco de dados **Pubs**, utilizando o modelo **Web Forms**.

A – Criando a interface de usuário (User Interface – UI)

1. Crie um novo projeto **Web Forms**, sem autenticação, chamado **Cap05_Lab01**:



2. Abra o arquivo **Site.Master** e altere o nome do aplicativo e o menu:

- Título da página

```
<title><%: Page.Title %> - Biblioteca OnLine</title>
```

- Navegador

```
<div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" ...>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>

            <a class="navbar-brand"
                runat="server"
                href="~/">Biblioteca OnLine</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">

                <li> <a runat="server" href "~/">Início</a></li>
                <li><a runat="server" href "~/">Livros</a></li>
                <li><a runat="server" href "~/">Editoras</a></li>
                <li> <a runat="server" href "~/">Autores</a></li>

            </ul>
        </div>
    </div>
</div>
```

- Rodapé

```
<p>&copy; <%: DateTime.Now.Year %> - Biblioteca OnLine</p>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

3. Altere a página **Default.aspx** para o seguinte conteúdo:

```
<%@ Page Title="Início" Language="C#" ... %>

<asp:Content ID="BodyContent" ...>

    <div class="jumbotron">
        <h1>Biblioteca</h1>
        <p class="lead">
            Biblioteca OnLine é um serviço que fornece acesso aos últimos
            lançamentos de livros do mundo todo
        </p>
    </div>

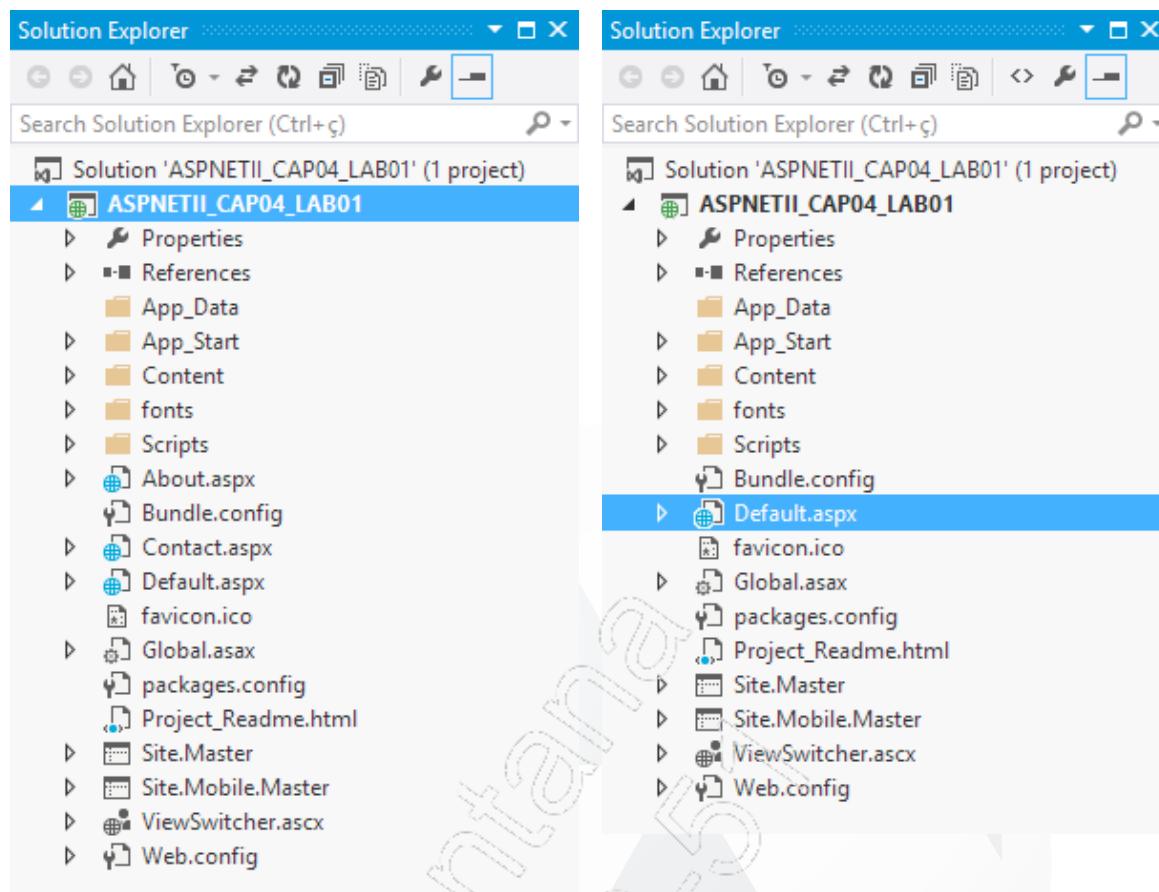
</asp:Content>
```

4. Visualize a página:

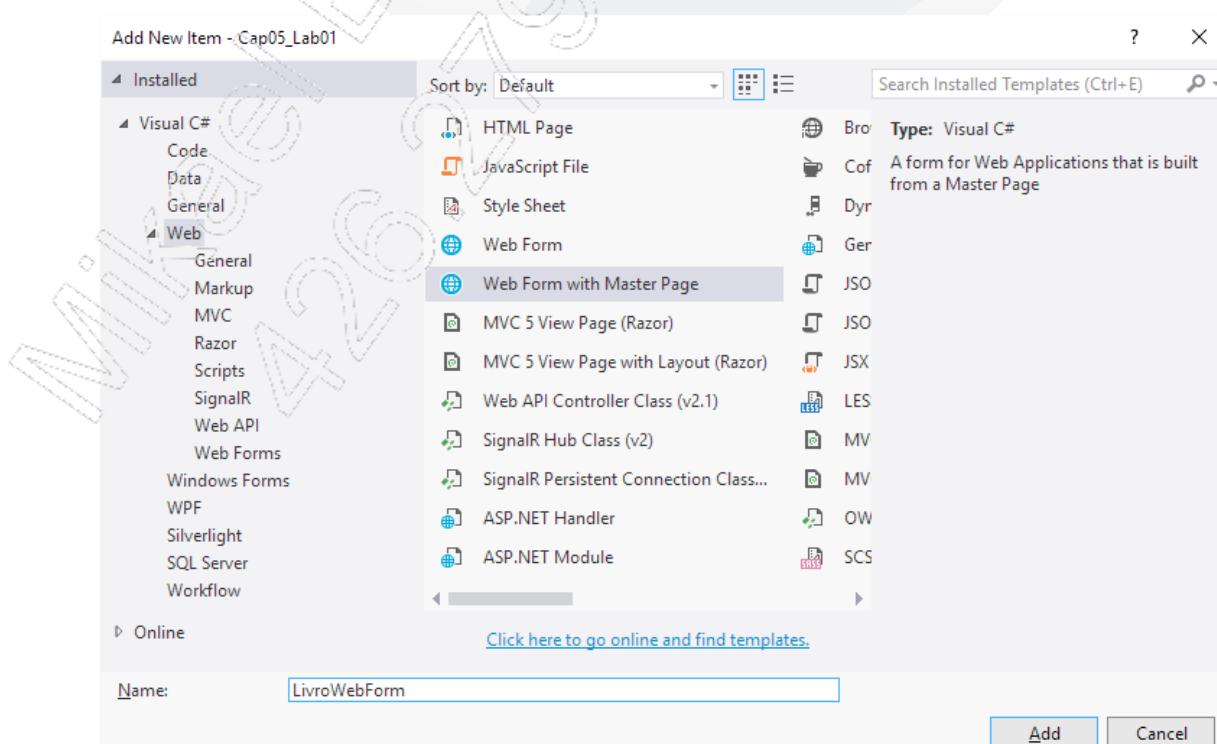


Entity Framework (Model/Database First)

5. Exclua as páginas **Contact.aspx** e **About.aspx**:



6. Adicione as seguintes páginas baseadas na Master Page: **LivroWebForm.aspx**, **AutorWebForm.aspx** e **EditoraWebForm.aspx**:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

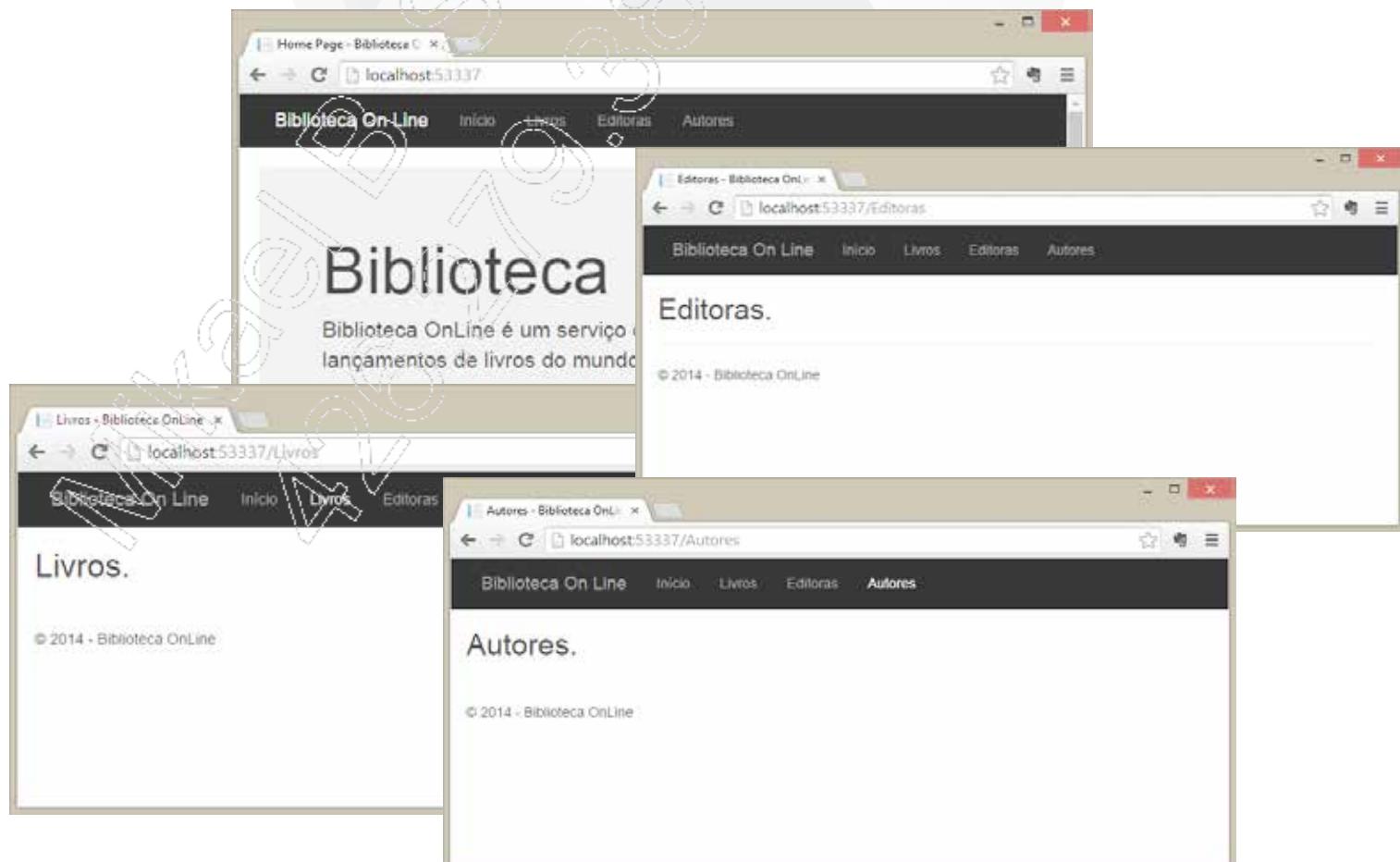
7. Altere o menu na Master Page para chamar as páginas criadas:

```
<ul class="nav navbar-nav">  
  
<li><a runat="server" href("~/")>Início</a></li>  
<li><a runat="server" href "~/LivroWebForm.aspx">Livros</a></li>  
<li><a runat="server" href "~/EditoraWebForm.aspx">Editoras</a></li>  
<li><a runat="server" href "~/AutorWebForm.aspx">Autores</a></li>  
</ul>
```

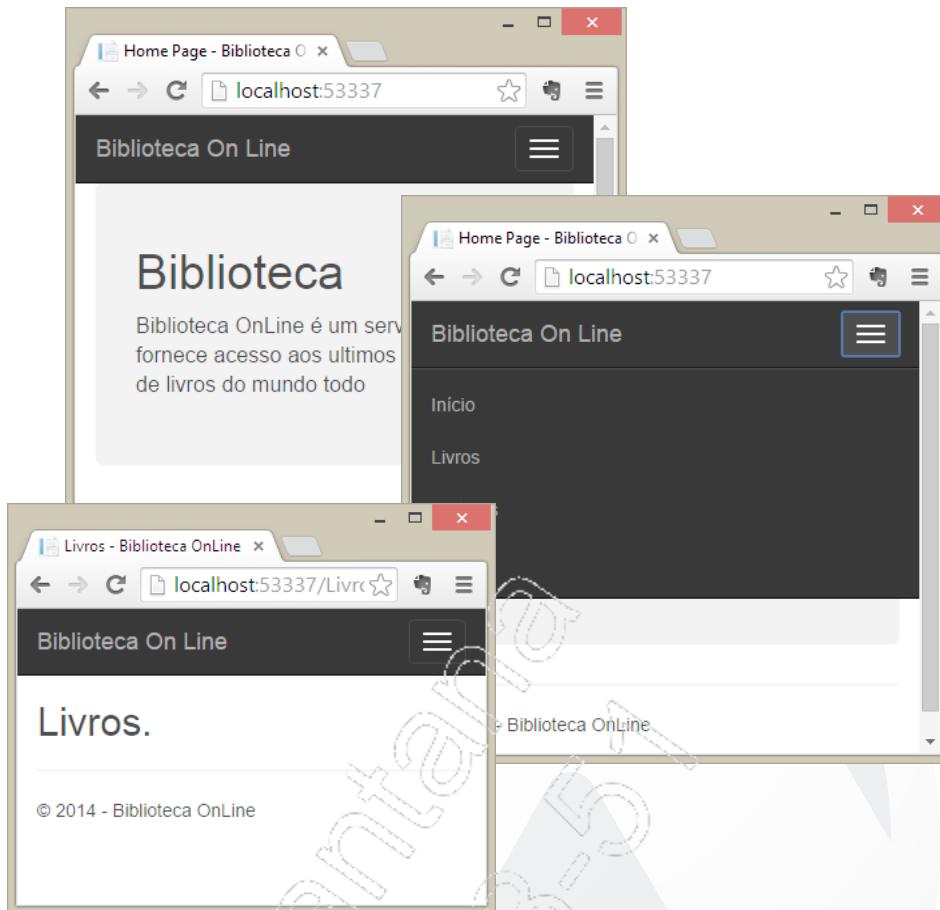
8. Altere cada página derivada da Master Page (**LivroWebForm.aspx**, **EditoraWebForm.aspx** e **AutorWebForm.aspx**) para conter o título:

```
<%@ Page Title="Livros" Language="C#" ... %>  
  
<asp:Content ID="Content1" ....>  
  
    <h2><%: Title %>.</h2>  
  
</asp:Content>
```

9. Teste a navegação:



10. Teste, também, a navegação para tablet/mobile, diminuindo o tamanho do navegador:

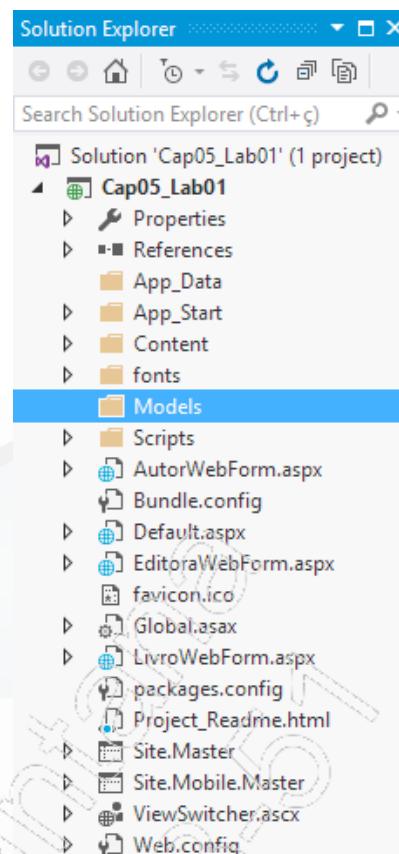


A estrutura HTML está pronta. O próximo passo é criar a parte que vai exibir os livros, editoras e autores. Será usado o banco de dados **Pubs**, da Microsoft. Como o banco de dados já existe, a melhor estratégia é usar o modo **Database First** e, depois, adaptar o modelo.

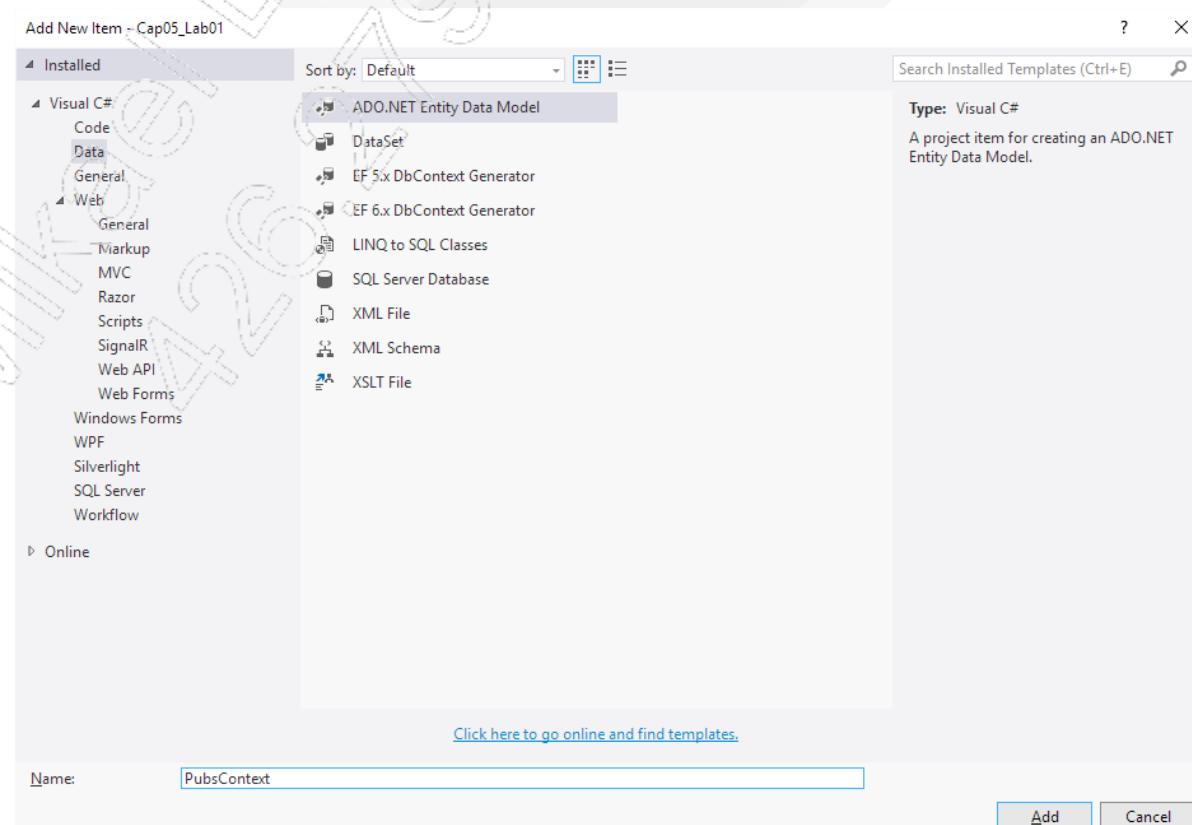
Visual Studio 2015 - ASP.NET com C# Acesso a dados

B – Trabalhando com acesso a dados

1. Adicione uma pasta ao projeto chamada **Models**:

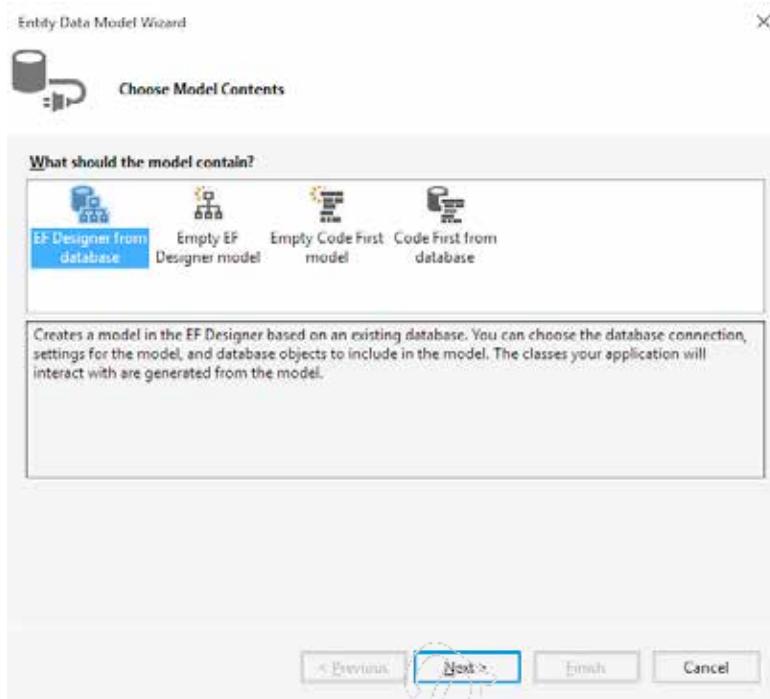


2. Na pasta **Models**, adicione um **ADO.NET Entity Data Model**, chamado **PubsContext**:

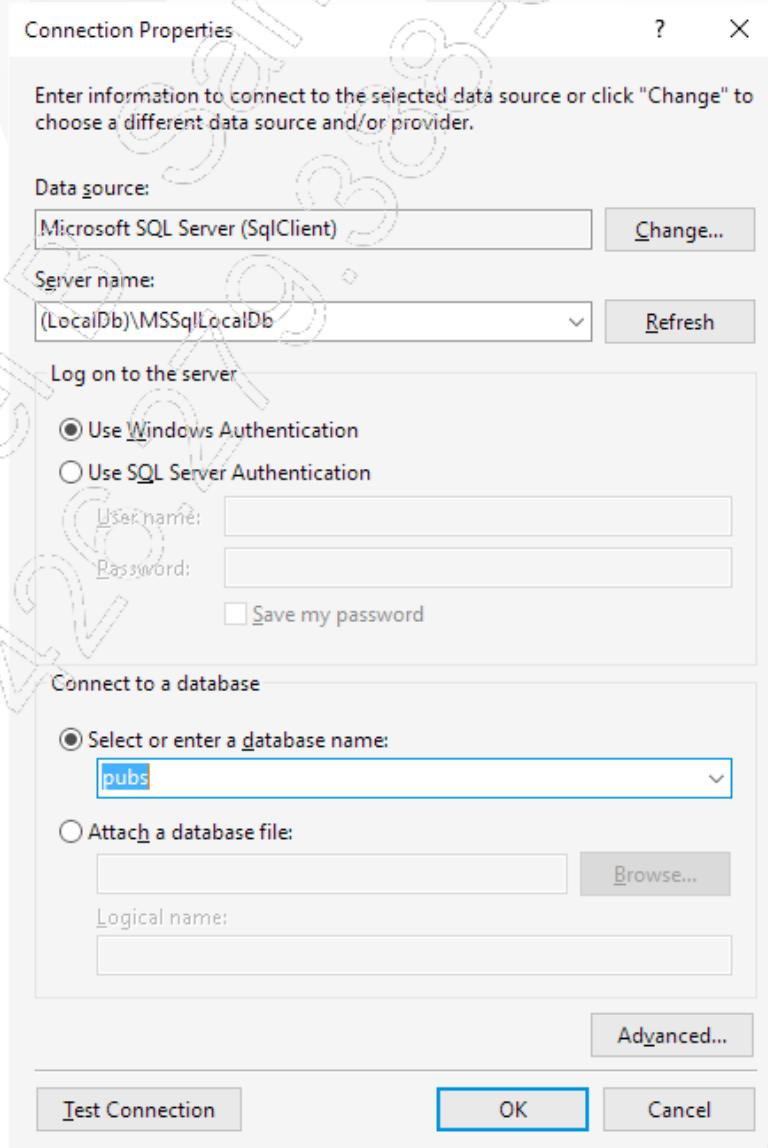


Entity Framework (Model/Database First)

3. Escolha a opção EF Designer from database:

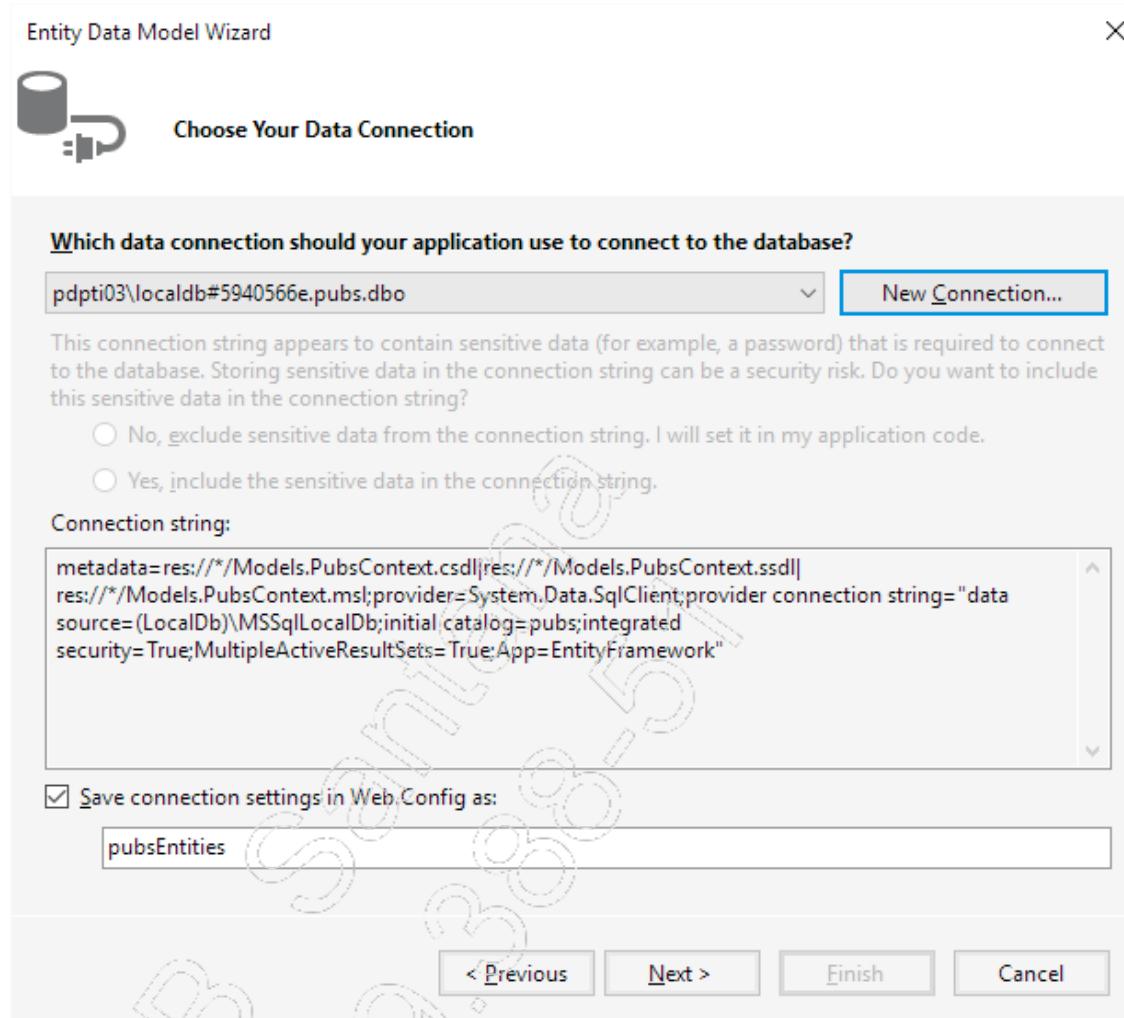


4. Clique em New Connection, escolha como servidor (**LocalDb)\MSSqlLocalDb**, autenticação **Windows Authentication**, banco de dados **Pubs** e clique em OK;



Visual Studio 2015 - ASP.NET com C# Acesso a dados

5. De volta à tela anterior, clique em **Next**:

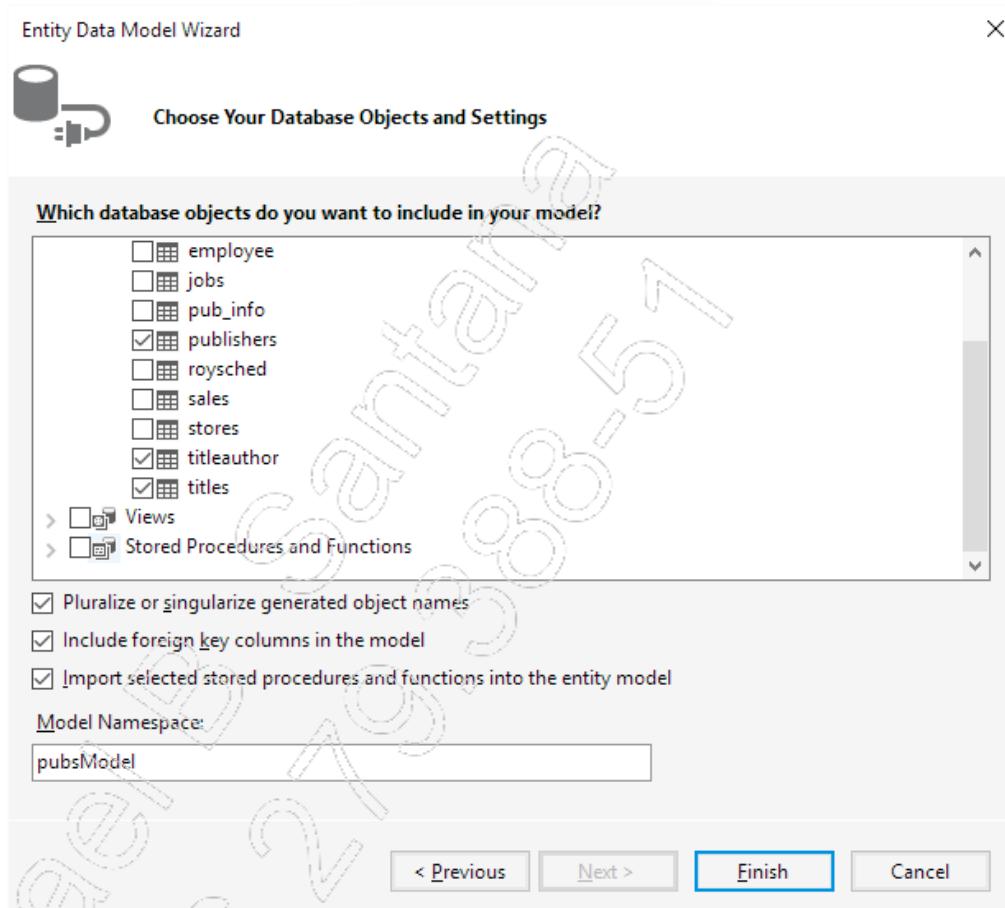


6. Se aparecer uma tela perguntando a versão do EF, deixe na opção padrão e clique em **Next**:

Entity Framework (Model/Database First)

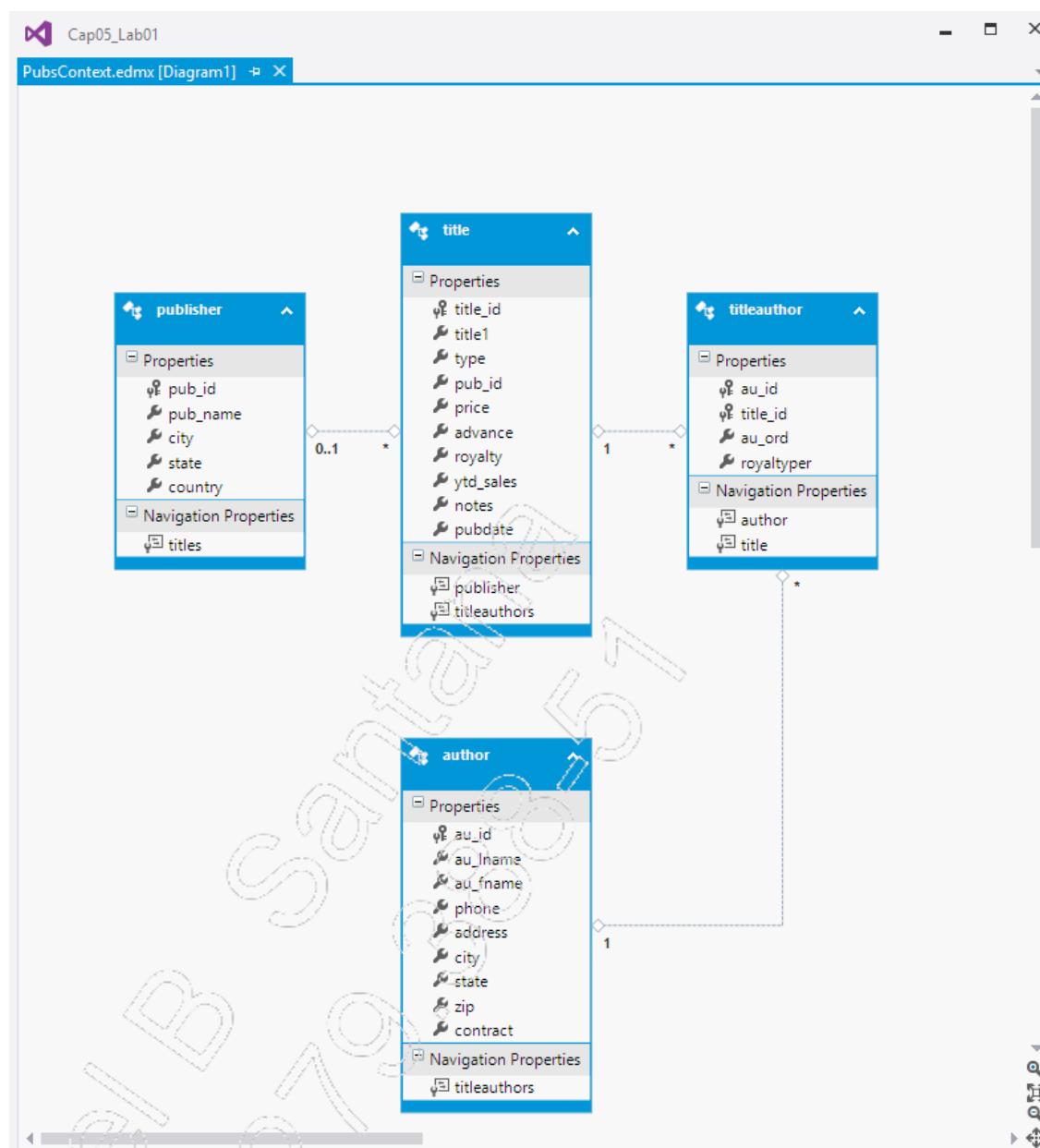
Na tela para escolher as tabelas que farão parte deste contexto, escolha as tabelas **authors** (autores), **publishers** (editoras), **titles** (livros) e **titleauthor** (livros e autores).

Marque a opção **Pluralize or singularize generated object names** para que o nome das tabelas seja gerado no plural e o nome dos objetos no singular.

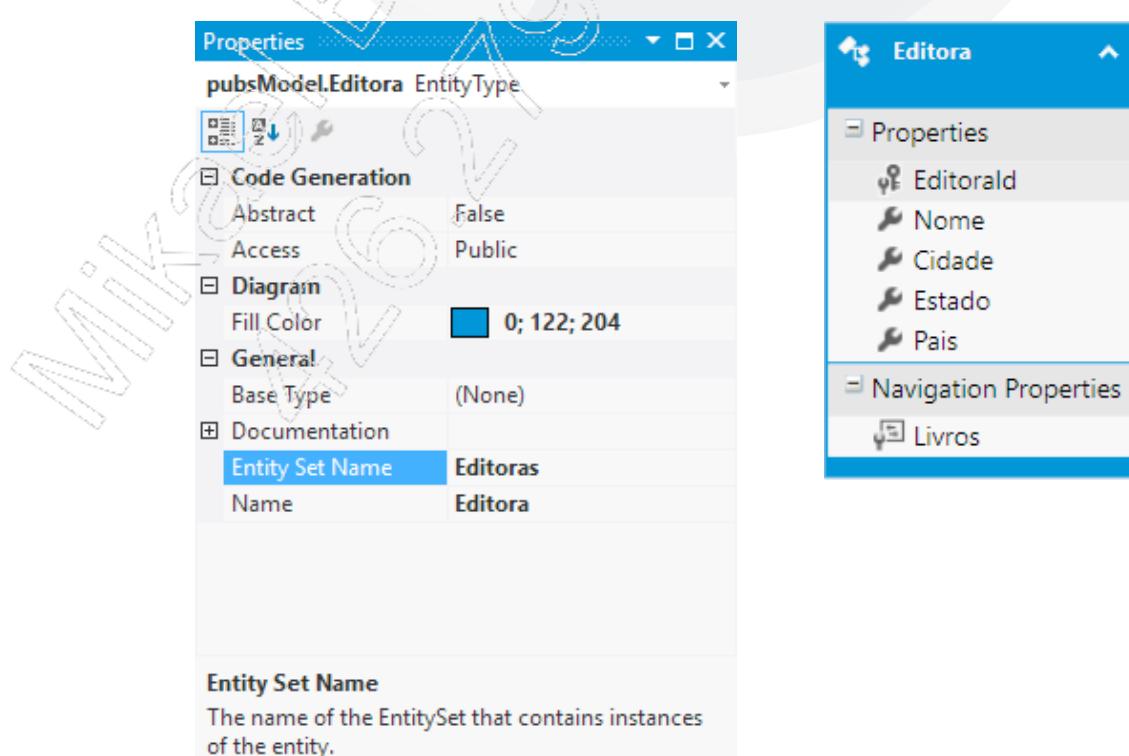
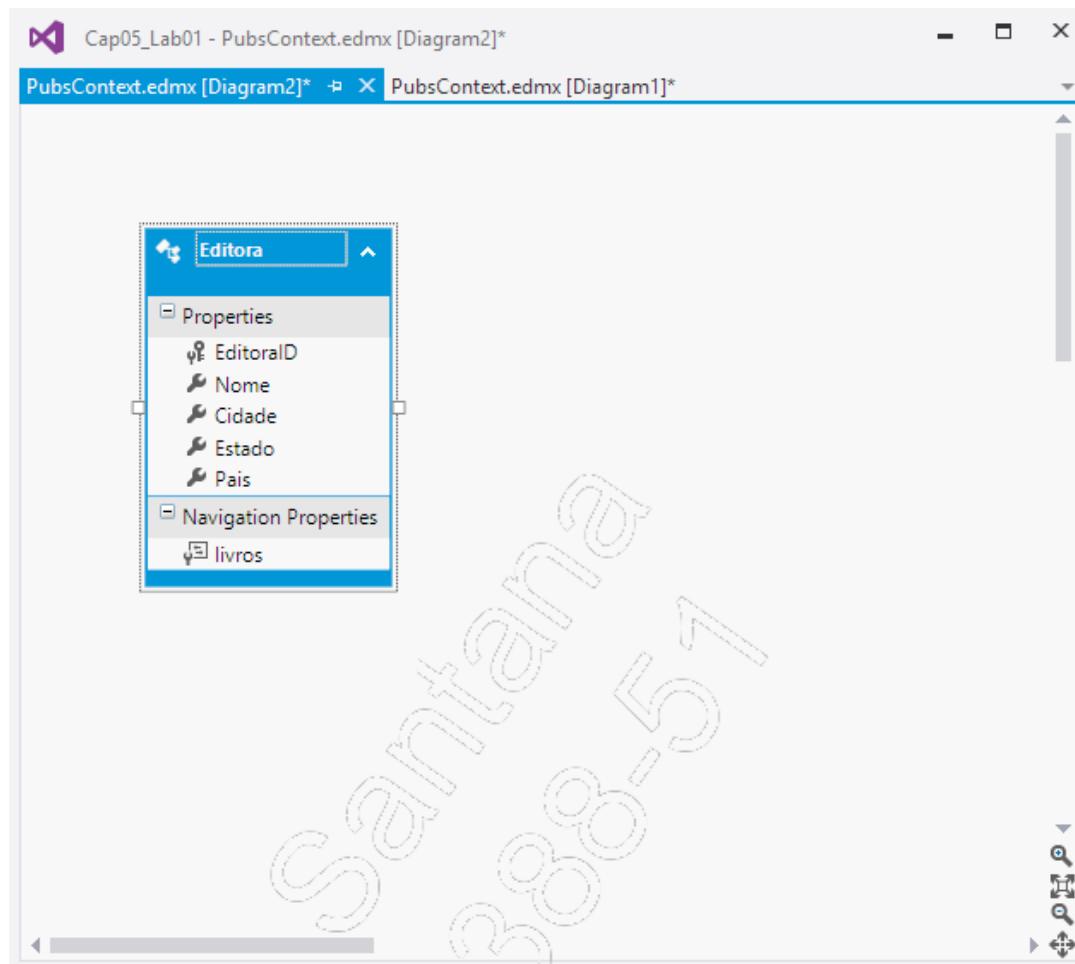


Visual Studio 2015 - ASP.NET com C# Acesso a dados

O diagrama foi criado e os relacionamentos entre as tabelas criaram as propriedades de navegação:

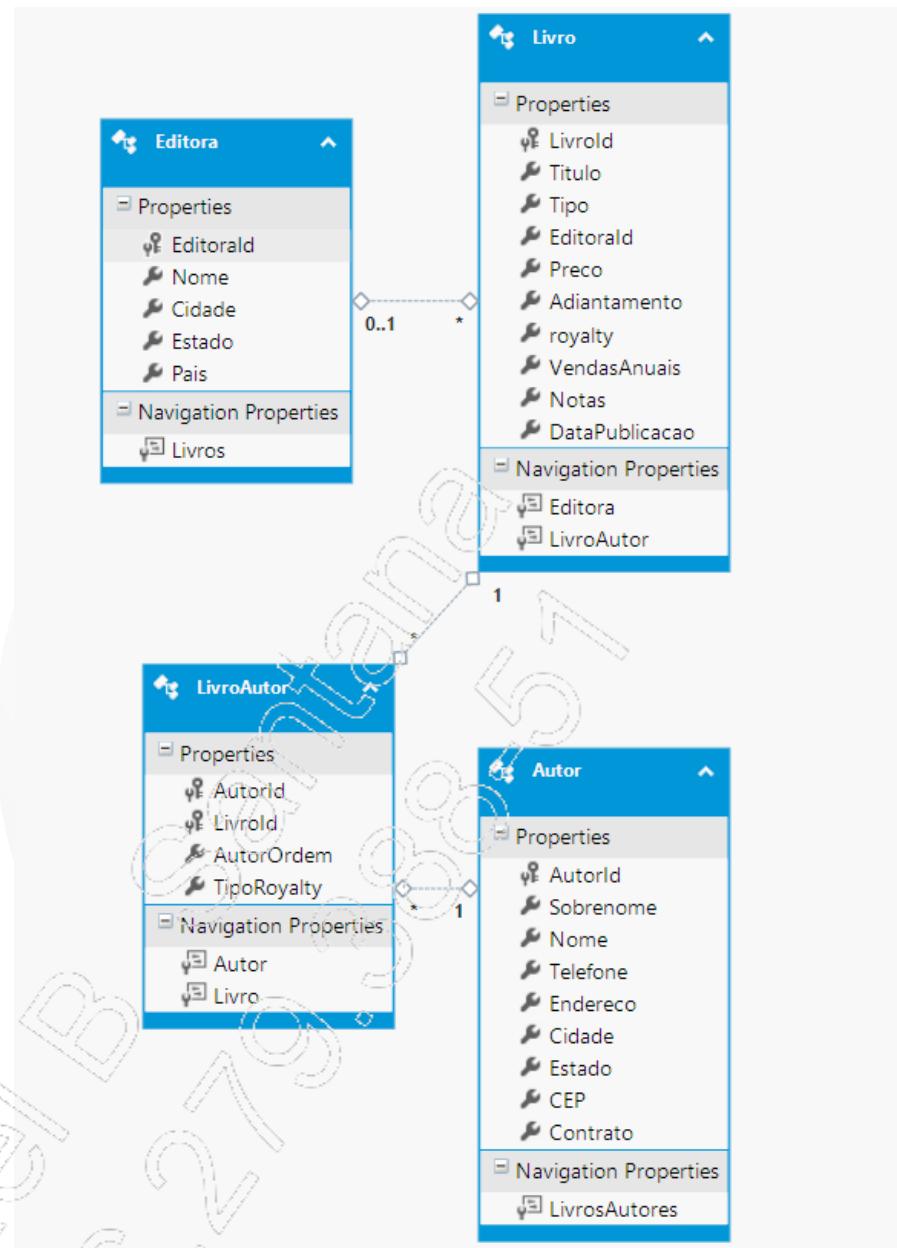


7. As propriedades criadas podem ser alteradas. A entidade **Publishers**, por exemplo, pode ser alterada para **Editoras (Entity Set Name)** e **Editora (Name)**, e as propriedades, alteradas para **EditorId**, **Nome**, **Cidade**, **Estado** e **Pais**. O campo de navegação **Titles** pode ser alterado para **Livros**;



Visual Studio 2015 - ASP.NET com C# Acesso a dados

8. Altere as outras propriedades, conforme o esquema adiante. Lembre-se de alterar o nome da coleção para cada entidade por meio da janela de propriedades. Nem sempre a pluralização funciona para a língua portuguesa. **Livro**, por exemplo, fica **Livroes**, e não **Livros**:



9. Ainda na pasta **Models**, adicione uma classe chamada **BibliotecaDb**. Como é uma aplicação pequena e são poucos dados, a classe será declarada como **static**:

```
public static class BibliotecaDb
{
}
```

10. Na classe **BibliotecaDb**, adicione o método que retorna a lista de editoras:

```
//  
// EditorasLista  
// Retorna a lista de todas as editoras  
//  
public static List<Editora> EditorasLista()  
{  
    using (var db = new pubsEntities())  
    {  
        return db.Editoras.ToList();  
    }  
}
```

11. Na página de editoras, crie a GridView para exibir as editoras. A coluna com o nome da editora é um hiperlink, que chama a página **LivroWebForm.aspx** e passa como parâmetro o **id** da **Editora** para exibir os livros desta editora;

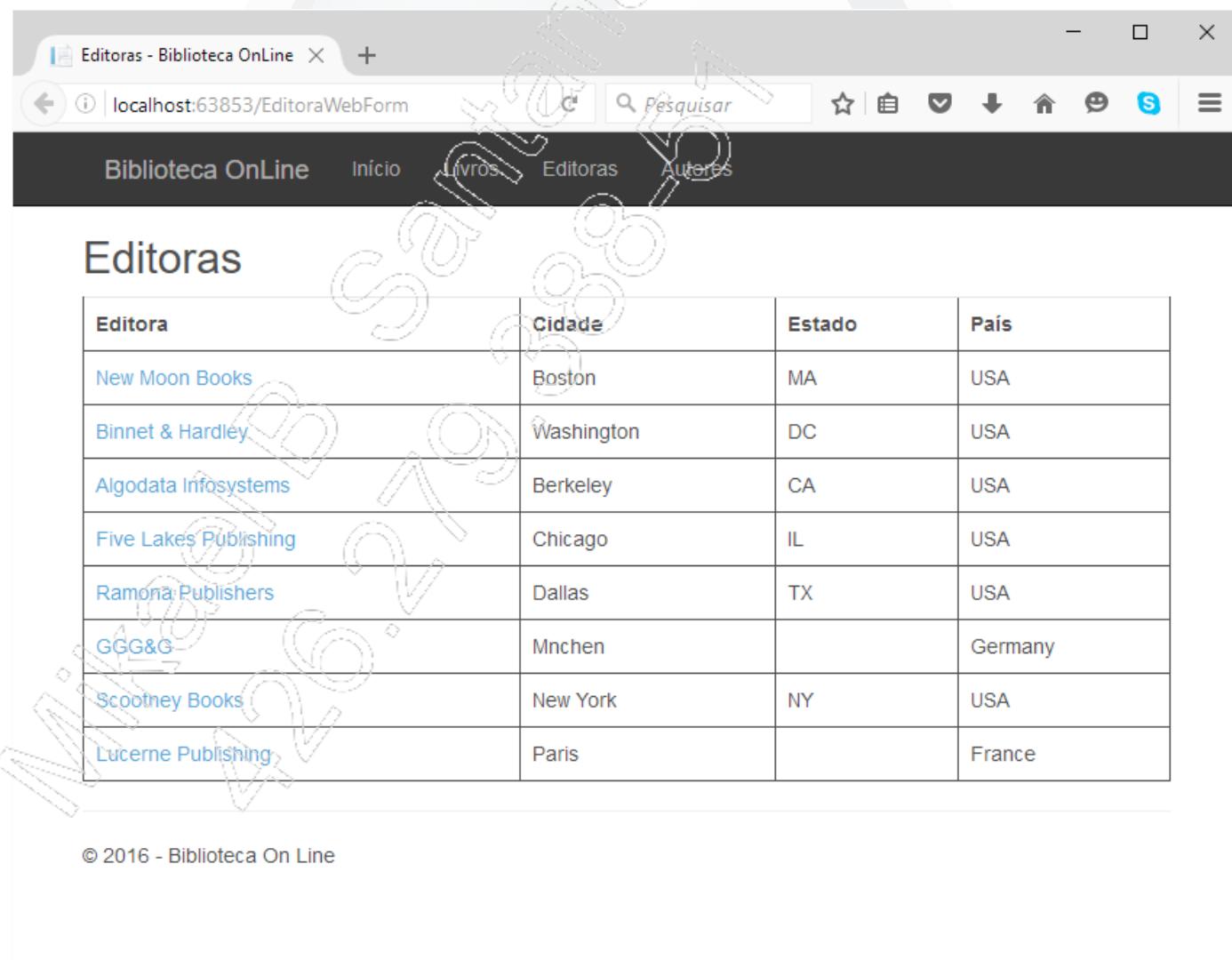
```
<asp:GridView SelectedRowStyle-BackColor="#f7f7f7"  
    CssClass="table"  
    runat="server"  
    ID="editorasGridView"  
    AutoGenerateColumns="false">  
    <Columns>  
        <asp:HyperLinkField  
            DataNavigateUrlFields="EditoraId"  
            DataTextField="Nome"  
            DataNavigateUrlFormatString=  
                "LivroWebForm.aspx?editoraId={0}"  
            HeaderText="Editora"/>  
        <asp:BoundField DataField="Cidade"  
            HeaderText="Cidade"/>  
        <asp:BoundField DataField="Estado"  
            HeaderText="Estado"/>  
        <asp:BoundField DataField="Pais"  
            HeaderText="País"/>  
    </Columns>  
</asp:GridView>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

12. Na página de editoras, crie code-behind para exibir a lista de editoras:

```
public partial class EditorasWebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        editorasGridView.DataSource=BibliotecaDb.EditorasLista();
        editorasGridView.DataBind();
    }
}
```

13. Compile e observe a página:



Entity Framework (Model/Database First)

14. A página de livros exibe todos os livros, os livros de uma editora ou os livros de um autor. São necessários três métodos. O primeiro retorna a lista completa de livros. Crie esse método na classe **BibliotecaDb**:

```
public static List<Livro> LivrosLista()
{
    using (var db = new pubsEntities())
    {
        return db.Livros.ToList();
    }
}
```

15. Ainda na classe **BibliotecaDb**, esse outro método retorna os livros de uma determinada editora:

```
public static List<Livro> LivrosPorEditora(string editoraId)
{
    using (var db = new pubsEntities())
    {
        return (from l in db.Livros
                where l.EditoraId==editoraId
                select l).ToList();
    }
}
```

16. Será necessário mostrar o nome da editora e, talvez, outros dados, como o endereço e o telefone. É necessário um método para retornar a editora, a partir do **Id**. Este método retorna a editora:

```
public static Editora EditoraPorId(string editoraId)
{
    using(var db=new pubsEntities())
    {
        var editora=( from c in db.Editoras
                      where c.EditoraId==editoraId
                      select c).FirstOrDefault();
        return editora;
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

17. Na página de livros (**LivroWebForm.aspx**), adicione um Label do lado do título da página para exibir o nome da editora;

```
<h2>
    <%:Title %>
    <asp:Label ID="mensagemLabel" runat="server"></asp:Label>
</h2>
```

18. Ainda na página de livros (**LivroWebForm.aspx**), insira a GridView que exibe os livros:

```
<asp:GridView runat="server"
    id="livrosGridView"
    CssClass="table"
    AutoGenerateColumns="false">
    <Columns>
        <asp:BoundField DataField="Titulo"
            HeaderText="Livro" />

        <asp:BoundField DataField="Tipo"
            HeaderText="Tipo" />

        <asp:BoundField DataField="Preco"
            DataFormatString = "{0:c}"
            HeaderText="Preço"
            ItemStyle-HorizontalAlign="Right"
            ItemStyle-Wrap="false" />

        <asp:BoundField DataField = "DataPublicacao"
            HeaderText="Publicado Em"
            DataFormatString="{0:d}" />

        <asp:BoundField DataField="Notas"
            HeaderText="Observações" />
    </Columns>
    <EmptyDataTemplate>
        <p>Nenhum resultado encontrado</p>
    </EmptyDataTemplate>
</asp:GridView>
```

19. No code-behind da página **LivroWebForm.aspx**, a lista completa será exibida ou a lista dos livros de uma editora. Depende da informação passada via query string:

```
protected void Page_Load(object sender, EventArgs e)
{
    List<Livro> listaDeLivros = null;

    string editoraId = Request.QueryString["editoraId"];

    if(!string.IsNullOrEmpty(editoraId))
    {
        listaDeLivros = BibliotecaDb.LivrosPorEditora(editoraId);
        var editora = BibliotecaDb.EditoraPorId(editoraId);
        mensagemLabel.Text = " da Editora " + editora.EditoraNome;
    }
    else
    {

        listaDeLivros = BibliotecaDb.LivrosLista();
    }

    livrosGridView.DataSource = listaDeLivros;
    livrosGridView.DataBind();
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

20. Teste a página **LivroWebForm.aspx** chamando-a diretamente e por meio da página **EditoraWebForm.aspx**:

The screenshot shows two separate browser windows side-by-side.

Top Window (Editoras):

Address bar: localhost:63853/EditoraWebForm

Page title: Editoras - Biblioteca OnLine

Header: Biblioteca OnLine, Início, Livros, Editoras, Autores

Content: A table titled "Editoras" listing eight editor companies with their city, state, and country:

Editora	Cidade	Estado	País
New Moon Books	Boston	MA	USA
Binnet & Hardley	Washington	DC	USA
Algodata Infosystems	Berkeley	CA	USA
Five Lakes Publishing	Chicago	IL	USA
Ramona Publishers	Dallas	TX	USA
GGG&G	Mnchen		Germany
Scootney Books	New York	NY	USA
Lucerne Publishing	Paris		France

Bottom right corner of the page: © 2016 - Biblioteca On Line

Bottom Window (Livros):

Address bar: localhost:63853/LivroWebForm?editoraid=0

Page title: Livros - Biblioteca OnLine

Header: Biblioteca OnLine, Início, Livros, Editoras, Autores

Content: A table titled "Livros da Editora New Moon Books" listing five books from that publisher:

Livro	Tipo	Preço	Publicado Em	Observações
You Can Combat Computer Stress!	business	R\$ 2,99	30/06/91	The latest medical and psychological techniques for living with the electronic office. Easy-to-understand explanations.
Is Anger the Enemy?	psychology	R\$ 10,95	15/06/91	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.
Life Without Fear	psychology	R\$ 7,00	05/10/91	New exercise, meditation, and nutritional techniques that can reduce the shock of daily interactions. Popular audience. Sample menus included, exercise video available separately.
Prolonged Data Deprivation: Four Case Studies	psychology	R\$ 19,99	12/06/91	What happens when the data runs dry? Searching evaluations of information-shortage effects.
Emotional Security: A New Algorithm	psychology	R\$ 7,99	12/06/91	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids emphasized.

21. Finalmente, a classe **Autores**. Na classe **BibliotecaDb**, crie os métodos que retornam a lista de autores, um autor e os livros de um autor:

- **Lista de autores**

```
public static List<Autor> AutoresLista()
{
    using (var db = new pubsEntities())
    {
        return db.Autores.ToList();
    }
}
```

- **Autor por Id**

```
public static Autor AutorPorId(string autorId)
{
    using (var db = new pubsEntities())
    {
        var autor=(from a in db.Autores
                   where a.AutorId==autorId
                   select a).FirstOrDefault();

        return autor;
    }
}
```

- **Livros por autor**

```
public static List<Livro> LivrosPorAutor(string autorId)
{
    using (var db = new pubsEntities())
    {
        var livrosAutor= ( from a in db.LivrosAutores
                           where a.AutorId.Equals(autorId)
                           select a.LivroId).ToArray();

        var livros=   from l in db.Livros
                      where livrosAutor
                            .Contains(l.LivroId)
                            select l;

        return livros.ToList();
    }
}
```

22. A classe **Autor** gerada automaticamente pelo Visual Studio não pode ser alterada diretamente. Se for alterada, uma modificação na classe pelo Designer pode ser perdida;

A solução correta é criar uma classe **partial**, que não é afetada pelo Designer. No caso dos autores, os nome são divididos em **FirstName** (Nome) e **LastName** (Sobrenome).

Na classe parcial, haverá um método que junta o nome e o sobrenome. Insira uma classe (na pasta **Models**) em um arquivo chamado **AutorCustom.cs** e altere para o seguinte código:

```
public partial class Autor
{
    public string NomeCompleto
    {
        get { return Nome + " " + Sobrenome; }
    }
}
```

Na hora da compilação, todas as classes parciais de mesmo nome são unidas e se tornam uma classe só. Esse comportamento é perfeito para esse caso, em que uma ferramenta cria e recria o código. Lembre-se que a classe personalizada **Autor** e a classe gerada pelo assistente **Autor** devem estar no mesmo namespace.

23. Crie a Grid e o Label do autor na página HTML na página **AutorWebForm.aspx**:

```
<h2><%: Page.Title %>
  <asp:Label runat="server"
    ID="mensagemLabel">
  </asp:Label>
</h2>

<asp:GridView runat="server"
  ID="autoresGridView"
  CssClass="table"
  AutoGenerateColumns="false">

  <Columns>
    <asp:HyperLinkField runat="server"
      DataNavigateUrlFields="AutorId"
      DataTextField="NomeCompleto"
      HeaderText="Autor"
      DataNavigateUrlFormatString=
        "~/livroWebForm.aspx?autorId={0}" />

    <asp:BoundField runat="server"
      DataField="Endereco"
      HeaderText="Endereço" />

    <asp:BoundField runat="server"
      DataField="Cidade"
      HeaderText="Cidade" />

    <asp:BoundField runat="server"
      DataField="Estado"
      HeaderText="Estado" />

    <asp:BoundField runat="server"
      DataField="CEP"
      HeaderText="CEP" />
  </Columns>
</asp:GridView>
```

24. O code-behind da página **AutorWebForm.aspx** exibe os dados:

```
protected void Page_Load(object sender, EventArgs e)
{
    autoresGridView.DataSource = BibliotecaDb.AutoresLista();
    autoresGridView.DataBind();
}
```

25. O code-behind da página **Livro (LivroWebForm.aspx)** deve ser adaptado para exibir, também, o autor:

```
protected void Page_Load(object sender, EventArgs e)
{

    List<Livro> listaDeLivros = null;

    string editoraId = Request.QueryString["editoraId"];
    if(!string.IsNullOrEmpty(editoraId))
    {
        listaDeLivros =
            BibliotecaDb.LivrosPorEditora(editoraId);
        var editora = BibliotecaDb.EditoraPorId(editoraId);
        mensagemLabel.Text = " da Editora " + editora.Nome;

    }
    else
    {
        string autorId = Request.QueryString["autorId"];
        if (!string.IsNullOrEmpty(autorId))
        {

            listaDeLivros = BibliotecaDb.LivrosPorAutor(autorId);
            var autor = BibliotecaDb.AutorPorId(autorId);
            mensagemLabel.Text =
                " do autor " + autor.NomeCompleto;
        }
        else
        {
            listaDeLivros = BibliotecaDb.LivrosLista();
        }
    }
    livrosGridView.DataSource = listaDeLivros;
    livrosGridView.DataBind();
}
```

Entity Framework (Model/Database First)

26. Teste todo o programa:

Biblioteca OnLine Início Livros Editoras Autores

Biblioteca

Biblioteca OnLine é um serviço que fornece acesso aos lançamentos de livros do mundo todo

© 2016 - Biblioteca On Line

Biblioteca OnLine Início Livros Editoras Autores

Livros

Livro	Tipo	Preço	Publicado Em	Observações
The Busy Executive's Database Guide	business	R\$ 19,99	12/06/91	An overview of available database systems with emphasis on common business applications- illustrated
Cooking with Computers: Surreptitious Balance Sheets	business	R\$ 11,95	09/06/91	Helpful hints on how to use your electronic resources to the best advantage.
You Can Combat Computer Stress!	business	R\$ 2,99	30/06/91	The latest medical and psychological techniques for living with the electronic office. Easy-to-understand explanations.

Biblioteca OnLine Início Livros Editoras Autores

Editoras

Editora	Cidade	Estado	País
New Moon Books	Boston	MA	USA
Binet & Hardley	Washington	DC	USA
Algadata Infosystems	Berkeley	CA	USA
Five Lakes Publishing	Chicago	IL	USA
Ramona Publishers	Dallas	TX	USA
GGG&G	München		Germany
Scootney Books	New York	NY	USA
Lucerne Publishing	Paris		France

© 2016 - Biblioteca On Line

Biblioteca OnLine Início Livros Editoras Autores

Livros da Editora New Moon Books

Livro	Tipo	Preço	Publicado Em	Observações
You Can Combat Computer Stress!	business	R\$ 2,99	30/06/91	The latest medical and psychological techniques for living with the electronic office. Easy-to-understand explanations
Is Anger the Enemy?	psychology	R\$ 10,95	15/06/91	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.
Life Without Fear	psychology	R\$ 7,00	05/10/91	New exercise, meditation, and nutritional techniques that can reduce the shock of daily interactions. Popular audience. Simple menus included, exercise video available separately
Prolonged Data Deprivation: Four Studies	psychology	R\$ 19,99	12/06/91	What happens when the data runs dry? Searching evaluations of information-shortage effects.
Personal Security: A Algorithm	psychology	R\$ 7,99	12/06/91	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids emphasized

Biblioteca OnLine Início Livros Editoras Autores

Autores

Autor	Endereço	Cidade	Estado	CEP
Johnson White	10932 Bigge Rd	Menlo Park	CA	94025
Marjorie Green	309 63rd St #4411	Oakland	CA	94618
Cheryl Carson	589 Dahlia Ln	Berkeley	CA	94705
Michael O'Leary	22 Cleveland Av #14	San Jose		
Dean Straight	5420 College Av	Oakland		
Meander Smith	10 Mississippi Dr	Lauren		
Abraham Bennet	6223 Bateman St	Berkole		
Ann Dull	3416 Blonde St	Palo Alt		
Burt Gringleby	PO Box 792	Covelo		
Charlene Lockley	18 Broadway Av	San Fri		

© 2016 - Biblioteca On Line

Biblioteca OnLine Início Livros Editoras Autores

Livros do autor Dean Straight

Livro	Tipo	Preço	Publicado Em	Observações
Straight Talk About Computers	business	R\$ 19,99	22/06/91	Annotated analysis of what computers can do for you: a no-hype guide for the critical user

© 2016 - Biblioteca On Line

6

Manipulando imagens

- ✓ Upload de imagens (Web Forms);
- ✓ Upload de imagens (MVC);
- ✓ Tratamento de imagens;
- ✓ Gravando imagens no banco de dados;
- ✓ Gravando imagens com ADO.NET;
- ✓ Exibindo imagens com um handler;
- ✓ Gravando e exibindo imagens com Entity Framework;
- ✓ Northwind e as imagens gravadas no banco de dados;
- ✓ Vinculando imagens no GridView.

6.1. Introdução

O ASP.NET oferece muitos recursos para enviar, manipular, armazenar, recuperar e exibir imagens em aplicativos Web. O namespace **System.Drawing** contém classes para gerar imagens, alterar a resolução e o tamanho, criar miniaturas e manipular qualquer característica que for necessária.

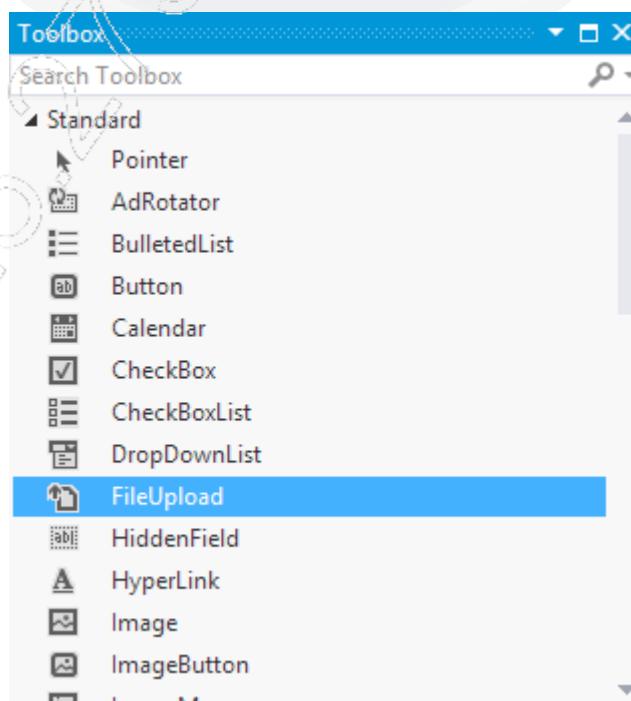
O namespace **System.IO**, por sua vez, contém classes para criar streams na memória ou em arquivo, permitindo gravar e ler imagens, sejam elas gravadas no sistema de arquivos ou armazenadas em campos BLOB (Binary Large Object) de um banco de dados.

O ASP.NET oferece o recurso de handlers personalizados para enviar imagens para o navegador e Web Controls para fazer upload de imagens para o servidor.

Todas as ferramentas necessárias para trabalhar com imagens, desde a criação, passando por armazenamento e manipulação e finalizando com a apresentação, estão disponíveis nas classes no .NET Framework.

6.2. Upload de imagens (Web Forms)

Para enviar imagens ao servidor no modelo Web Forms, é utilizado o Web Control **FileUpload**, que se encontra na guia **Standard** da janela **Toolbox**.



As principais propriedades desse controle são as seguintes:

- **HasFiles**: Um valor booleano indicando se algum arquivo foi enviado;
- **HasFile**: Um valor booleano indicando se um único arquivo foi enviado;
- **FileName**: O nome do arquivo (no cliente);
- **FileBytes**: Retorna um array de bytes com o conteúdo enviado para o servidor;
- **FileContent**: Retorna um stream apontando para o conteúdo enviado;
- **PostedFile**: Uma instância da classe **HttpPostedFile** com informações do arquivo enviado. As principais propriedades da classe **HttpPostedFile** são as seguintes:
 - **ContentLength**: O tamanho, em bytes, do arquivo;
 - **ContentType**: O tipo do arquivo;
 - **FileName**: O nome do arquivo enviado;
 - **InputStream**: Um stream contendo o arquivo.

Vejamos o principal método da classe **HttpPostedFile**:

- **Save()**: Grava o conteúdo enviado ao servidor em uma pasta.

O namespace **System.IO** contém classes para manipular arquivos e streams. A classe **Path** desse namespace é útil para obter o nome do arquivo (sem o nome da pasta) e a extensão para validar o arquivo.

O método **MapPath** da página permite mapear um endereço virtual (como **~imagens/produtos**, por exemplo) para um endereço físico (por exemplo, **C:\inetpub\wwwroot\website\imagens\produtos**).

Usando essas classes (**FileUpload**, **Path**, **Page**), o processo de enviar um arquivo para o servidor e gravá-lo em uma pasta pode ser realizado de maneira simples.

O próximo exemplo define uma página que envia uma imagem para o servidor, que, por sua vez, irá gravá-la no disco rígido local e exibi-la na tela:

- **Código HTML**

Neste exemplo, apenas serão usados um Web Control do tipo **FileUpload**, um **Button** para confirmar o envio, um **Label** para exibir dados adicionais e um **Image** para exibir a imagem:

```
<h1>Upload de Imagens</h1>

<asp:FileUpload runat="server" ID="imagemFileUpload" />

<br />
<br />

<asp:Button ID="enviarButton" runat="server" Text="Enviar" />

<br />
<br />

<asp:Label runat="server" ID="mensagemLabel"></asp:Label>

<br />
<br />

<asp:Image runat="server" ID="fotoImage" />
```

- **Código da página (botão Enviar)**

```
if (imagemFileUpload.HasFile)
{
    string pastaGravarEmVirtual = "~/imagens";
    string pastaGravarEmFisico = MapPath(pastaGravarEmVirtual);
    string arquivo = Path.GetFileName(imagemFileUpload.FileName);
    string caminhoCompletoFisco =
        Path.Combine(pastaGravarEmFisico, arquivo);
    string caminhoCompletoVirtual =
        string.Format("{0}/{1}", pastaGravarEmVirtual, arquivo);
    imagemFileUpload.SaveAs(caminhoCompletoFisco );
    fotoImage.ImageUrl = caminhoCompletoVirtual;
    mensagemLabel.Text = "Arquivo gravado com sucesso";
}
else
{
    mensagemLabel.Text = "Nenhum arquivo enviado.";
}
```

Vejamos o resultado:



O exemplo anterior recebe o arquivo e simplesmente grava-o na pasta **Imagens** com o mesmo nome que foi enviado. Propositalmente, o exemplo foi elaborado o mais simples possível para exemplificar o processo. Em um programa real, algumas melhorias serão necessárias, como as seguintes:

- Validar a extensão para aceitar apenas arquivos de imagens (**.jpg**, **.gif**, **.png**);
- Validar se o arquivo enviado é realmente uma imagem e não um vírus renomeado para parecer um arquivo de imagem;
- Validar o tamanho do arquivo. Normalmente existe um mínimo e um máximo de resolução para que o arquivo não seja nem muito pequeno e nem muito grande;
- Criar thumbnails (miniaturas) para exibição em listagem.

6.3. Upload de imagens (MVC)

No MVC não existe o conceito de code-behind e Web Controls. O processo HTML deve ser construído manualmente, configurando o formulário corretamente e usando o controle HTML **input type=file**. O recebimento do arquivo pode ser obtido por meio do contexto da requisição.

1. O formulário deve ter o atributo **enctype = "multipart/form-data"**;
2. Uma tag HTML **input type=file** deve estar no formulário:

```
@{
    ViewBag.Title = "Index";
}

<h2>Transferindo imagens para o servidor</h2>

<!--
    Parâmetros do Formulário:
    Action (Enviar Imagem)
    Controller (Home)
    Route Values (null)
    Método de Postagem (Post/Get)
    Atributos Adicionados (enctype)
-->

@using (Html.BeginForm("Index", "Home", null, FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <hr/>
    <p>Escolha um arquivo</p>

    <input type="file" name="arquivo" />
    <br/>
    <br />

    <input type="submit" class="btn btn-default" value="Enviar" />
    <br/>
    <br />

    @ViewBag.Mensagem

    if (!string.IsNullOrEmpty(ViewBag.ImagemUrl))
    {
        <br/>
        
    }
}
```

No Controller:

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

namespace Capitulo06_Exemplo.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            ViewBag.Mensagem = string.Empty;
            return View();
        }

        [HttpPost]
        public ActionResult Index(HttpPostedFileBase arquivo)
        {
            if (arquivo != null)
            {
                string pic =
                    Path.GetFileName(arquivo.FileName);
                string path =
                    Path.Combine(Server.MapPath("~/imagens"), pic);

                arquivo.SaveAs(path);
                ViewBag.Mensagem = "Arquivo enviado com sucesso";
                ViewBag.ImagemUrl = "~/imagens/" + pic;
            }
            else
            {
                ViewBag.Mensagem = "Nenhuma imagem enviada";
                ViewBag.ImagemUrl = string.Empty;
            }

            return View();
        }
    }
}
```

Veja o resultado do processamento:



6.4. Tratamento de imagens

Para validar a extensão de um arquivo enviado ao servidor, é necessário extrair a extensão do nome do arquivo. Isso pode ser feito usando o método **GetExtension** da classe **System.IO.Path**. O nome do arquivo enviado é obtido por meio da propriedade **FileName** da Web Control **FileUpload** (Web Forms), ou propriedade **FileName** no parâmetro **HttpPostedFileBase** no modelo MVC.

O método a seguir verifica a extensão do arquivo e retorna uma string com uma mensagem de erro; se não houver erros, retorna **null**:

```
//  
// Validar Extensão  
  
private string ValidarExtensao()  
{  
    string extensao =  
        Path.GetExtension(imagemFileUpload.FileName)  
            .ToLower();  
  
    string[] extensoesValidadas =  
        { ".jpg", ".gif", ".jpeg", ".png", ".bmp" };  
  
    if (!extensoesValidadas.Contains(extensao))  
    {  
        return @"Extensão de arquivo inválida.  
        Deve ser um arquivo de imagem.";  
    }  
    else  
    {  
        return null;  
    }  
}
```

A maneira mais simples de validar um arquivo de imagem é tentar criar um objeto do tipo bitmap. Se o .NET Framework não conseguir criar é porque é um formato que não pode ser manipulado pelo aplicativo.

A classe bitmap aceita diversos construtores. A lista a seguir exibe alguns deles. O construtor que aceita um stream é perfeito para o caso de um upload de um arquivo. A propriedade **PostedFile.InputStream** retorna um stream do arquivo a ser enviado.

```
var imagem=new Bitmap(stream);
var imagem=new Bitmap(array de bytes);
var imagem=new Bitmap(outra Imagem);
var imagem=new Bitmap(string: nome do arquivo);
```

Para criar um bitmap usando a entrada vinda da página, pode ser usado o seguinte código:

```
var bitmap=new Bitmap(fileUpload.PostedFile.InputStream);
```

Colocando esse comando dentro de uma estrutura de tratamento de erro, é possível validar se o arquivo é uma imagem. Se for possível criar o objeto bitmap, as propriedades como largura (**Width**) e altura (**Height**) podem ser verificadas, conforme o exemplo a seguir:

```
if ( bitmap.Width > larguraMaxima ||
    bitmap.Height > alturaMaxima)
{
    //Tratamento de Erro...
}
```

O método a seguir implementa essas validações:

```
//  
// Validar Propriedades do Arquivo  
  
private string ValidarPropriedadesArquivo()  
{  
    long larguraMaxima = 2024;  
    long alturaMaxima = 2024;  
    int larguraMinima = 20;  
    int alturaMinima = 20;  
    try  
    {  
        using (var bitmap =  
            new Bitmap(imagemFileUpload.PostedFile.InputStream))  
        {  
            //Arquivo muito grande  
            if (bitmap.Width > larguraMaxima ||  
                bitmap.Height > alturaMaxima)  
            {  
                return string.Format(  
                    "Arquivo muito grande.  
                    Tamanho máximo: {0}px de largura por  
                    {1}px de altura",  
                    larguraMaxima, alturaMaxima);  
            }  
  
            //Arquivo muito pequeno  
            if (bitmap.Width < larguraMinima ||  
                bitmap.Height < alturaMinima)  
            {  
                return string.Format(  
                    "Arquivo muito pequeno.  
                    Tamanho mínimo: {0}px de largura  
                    por {1}px de altura",  
                    larguraMinima, alturaMinima);  
            }  
        }  
        return null;  
    }  
    catch (Exception)  
    {  
        return "Arquivo de imagem inválido.";  
    }  
}
```

6.4.1.Criando miniaturas (thumbnails)

A classe **Image**, que é a classe abstrata da qual a classe bitmap é derivada, contém um método chamado **GetThumbnailImage**, que permite criar uma imagem a partir de outra imagem, passando como parâmetros as dimensões desejadas. O comando para executá-lo tem a seguinte sintaxe:

```
Image bmp.GetThumbnailImage(largura, altura, callback,dados);
```

Já os parâmetros a serem passados são os seguintes:

- **Largura**: A largura, em pixel, da nova imagem;
- **Altura**: A altura, em pixel, da nova imagem;
- **Callback**: Um delegate do tipo **GetThumbnailImageAbort**. Esse método não é usado diretamente, mas é necessário para as chamadas de criação de imagens do .NET Framework;
- **Dados**: Deve ser o valor fixo **IntPtr.Zero** (um ponteiro para um valor zero). Não é usado diretamente.

Vejamos um exemplo de uso:

```
Image bmp.GetThumbnailImage(largura, altura, null ,IntPtr.Zero);
```

O método a seguir cria miniaturas da imagem enviada para o servidor:

```
//  
// Criar Thumbnail  
//  
private void CriarThumbnail(  
    int largura,  
    int altura,  
    string pastaGravarThumbnailsVirtual,  
    string pastaGravarThumbnailsFisico,  
    string arquivo)  
{  
  
    // O local virtual da imagem  
    // Exemplo: imagens/miniaturas/foto.gif  
    string caminhoCompletoVirtual =  
        string.Format("{0}/{1}",  
            pastaGravarThumbnailsVirtual, arquivo);  
  
    // O local virtual da imagem  
    // Exemplo: c:\meusite\imagens\miniaturas\foto.gif  
    string caminhoCompletoFisico =  
        Path.Combine(pastaGravarThumbnailsFisico, arquivo);  
  
    // O objeto BitMap Original  
    using (var bitmap =  
        new Bitmap(imagemFileUpload.PostedFile.InputStream))  
{  
  
        // A miniatura criada a partir do original  
        var bmpMini = bitmap.GetThumbnailImage(  
            largura, altura, null, IntPtr.Zero);  
  
        // Gravando a miniatura no disco  
        bmpMini.Save(caminhoCompletoFisico);  
    }  
  
    // Exibindo a miniatura  
    miniaturaImage.ImageUrl = caminhoCompletoVirtual;  
    miniaturaImage.Visible = true;  
}
```

O método **validar** reúne as validações apresentadas anteriormente. Esse método chama os métodos de validação que devem retornar **Null**, se não houver erros, ou uma string com a descrição do erro, se houver.

```
//  
// Validar Imagem  
//  
private string ValidarImagen()  
{  
  
    string erro = null;  
  
    //Nenhum arquivo enviado  
    if (!imagemFileUpload.HasFile)  
    {  
        return "Nenhum imagem enviada";  
    }  
  
    //Extensão inválida  
    erro = ValidarExtensao();  
    if (erro != null) return erro;  
  
    //Arquivo inválido  
    erro = ValidarPropriedadesArquivo();  
    if (erro != null) return erro;  
  
    //Ok. Passou por todas as validações  
    return null;  
}
```

A listagem a seguir mostra o método **EnviarButton_Click** adaptado para tratar as validações das imagens:

```
fotoImage.Visible = false;
miniaturaImage.Visible = false;

string erro = ValidarImagem();
if (erro == null)
{
    string pastaGravarEmVirtual = "~/imagens";

    string pastaGravarThumbnailsVirtual =
        "~/imagens/miniaturas";

    string pastaGravarEmFisico = MapPath(pastaGravarEmVirtual);
    string pastaGravarThumbnailsFisico =
        MapPath(pastaGravarThumbnailsVirtual);

    string arquivo =
        Path.GetFileName(imagemFileUpload.FileName);

    string caminhoCompletoFisco =
        Path.Combine(pastaGravarEmFisico, arquivo);

    string caminhoCompletoVirtual =
        string.Format("{0}/{1}",
                      pastaGravarEmVirtual, arquivo);

    int larguraMiniatura = 50, alturaMiniatura = 50;

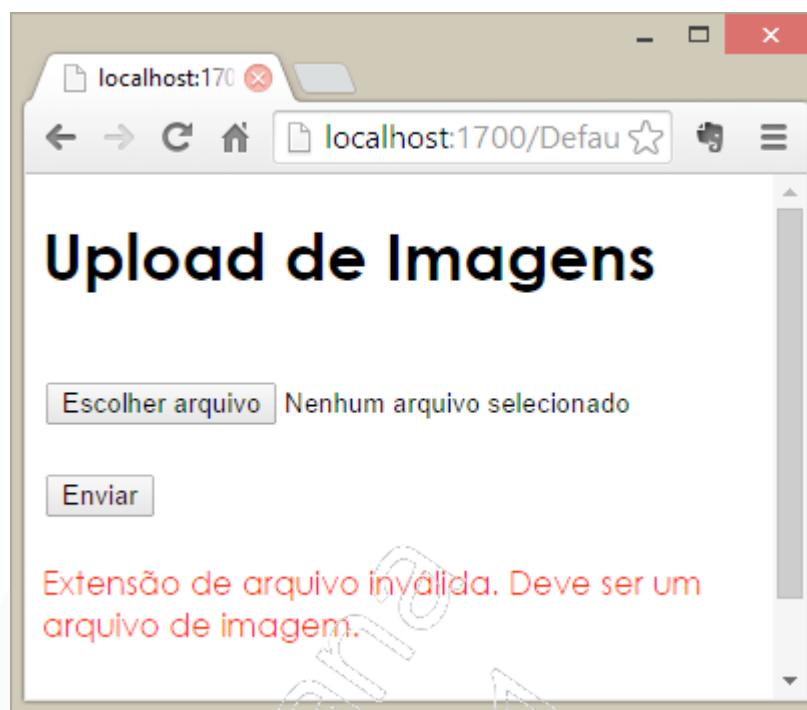
    imagemFileUpload.SaveAs(caminhoCompletoFisco);
    CriarThumbnail(larguraMiniatura, alturaMiniatura,
                  pastaGravarThumbnailsVirtual,
                  pastaGravarThumbnailsFisico, arquivo);

    fotoImage.ImageUrl = caminhoCompletoVirtual;
    fotoImage.Visible = true;

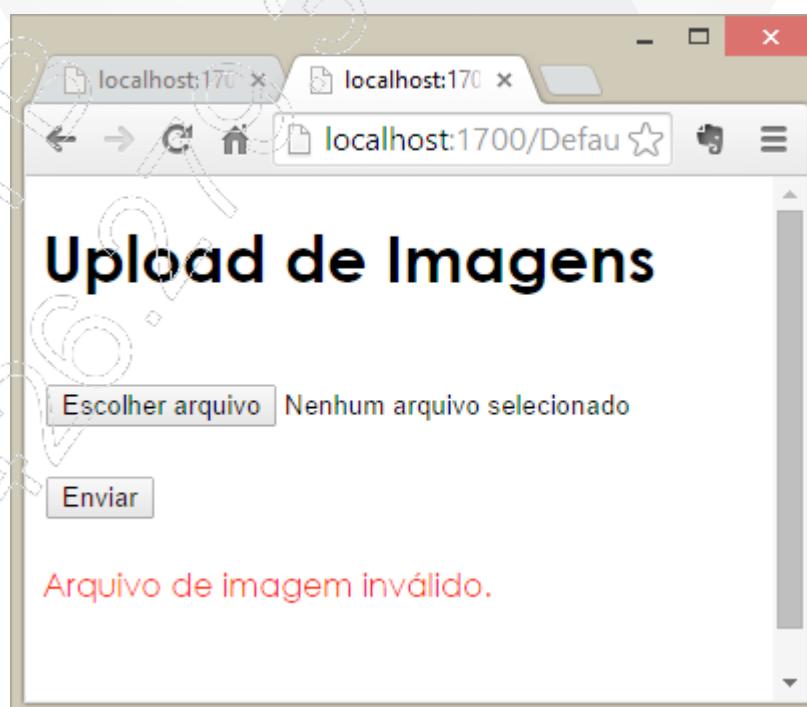
    mensagemLabel.Text = "Arquivo gravado com sucesso";
    mensagemLabel.ForeColor = Color.Black;
}
else
{
    mensagemLabel.Text = erro;
    mensagemLabel.ForeColor = Color.Red;
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Vejamos um exemplo de erro ocorrido quando uma imagem com extensão diferente das definidas é enviada:



A seguir, vejamos o retorno quando um arquivo que não é de imagem com a extensão renomeada é enviado:



Por fim, vejamos um exemplo de um envio com uma imagem válida:

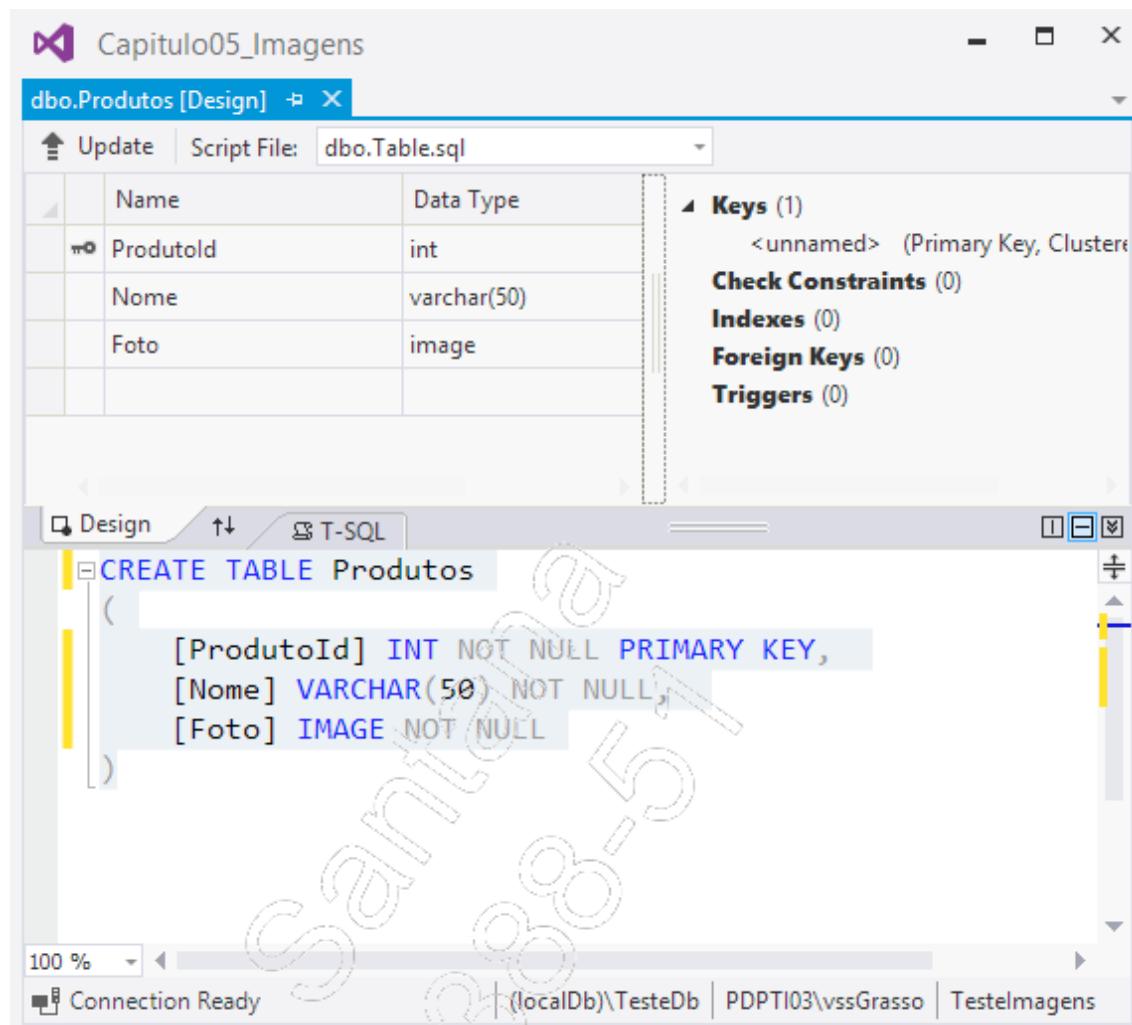


6.5. Gravando imagens no banco de dados

O SQL Server dispõe de um campo especialmente criado para armazenar imagens, chamado **Image**. No .NET Framework, usando classes de mapeamento, é possível utilizar um array de bytes ou uma propriedade do tipo **bitmap**. Usar o tipo array de bytes diminui a possibilidade de erros e não vincula diretamente a imagem armazenada a um tipo específico de renderização. Por outro lado, exige uma certa programação para a imagem ser exibida.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

A imagem adiante serve de exemplo para inserir e recuperar imagens em um banco de dados:



A classe a seguir será usada para obter informações de um produto e inserir na tabela descrita anteriormente:

```
public class Produto
{
    public int ProdutoId { get; set; }
    public string Nome { get; set; }
    public byte[] Foto { get; set; }
}
```

6.6. Gravando imagens com ADO.NET

A classe de acesso a dados fornece o método para incluir um produto com imagem:

```
public class ProdutoDb
{
    public static string Conexao =
        @"Data Source=localhost\sqlexpress;
          Initial Catalog=TesteImagens;
          Integrated Security=True;";

    public void Incluir(Produto p)
    {
        string sql = "INSERT INTO PRODUTOS(Nome, Foto)
                      VALUES(@Nome, @Foto)";

        using (var cn = new SqlConnection(Conexao))
        {
            using (var cmd = new SqlCommand(sql, cn))
            {
                cmd.Parameters.AddWithValue("@nome", p.Nome);
                cmd.Parameters.AddWithValue("@foto", p.Foto);
                cn.Open();
                cmd.ExecuteNonQuery();
            }
        }
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Para incluir um produto com foto na tabela **Produtos**, uma página ASPX será usada:

```
<h1>Inclusão de Produtos</h1>

<fieldset>
<p>
    <asp:Label runat="server" ID="nomeLabel"
        Text="Nome:" CssClass="legenda"></asp:Label>

    <asp:TextBox runat="server" ID="nomeTextBox"
        CssClass="campo"></asp:TextBox>
</p>

<p>
    <asp:Label runat="server" ID="fotoLabel"
        Text="Foto:" CssClass="legenda"></asp:Label>

    <asp:FileUpload runat="server" ID="imagemFileUpload" />

    <asp:Image runat="server" ID="fotoImage" CssClass="foto" />
</p>

<p>
    <asp:Button ID="enviarButton" runat="server"
        Text="Enviar" OnClick="enviarButton_Click" />
</p>

<asp:Label runat="server" ID="mensagemLabel"
    CssClass="mensagem"></asp:Label>

</fieldset>
```

Os estilos definidos para esta página definem o básico dos elementos:

```
body {  
    font-family :'Century Gothic';  
}  
  
h1 { padding-bottom:20px; }  
  
.legenda {  
    display:block;  
}  
  
.campo {  
    display:block;  
}  
  
fieldset {  
    border:none;  
}  
fieldset p {  
    margin-bottom:20px;  
}  
  
.mensagem {  
    display:block;  
    margin: 20px 0px 20px 0px;  
}  
  
.foto {  
    width:70px;  
    margin-top:20px;  
    margin-bottom:20px;  
    display:block;  
}
```

O código da página, no evento click, envia os dados para o banco de dados. Propositalmente, não foi feito nenhum tratamento e nenhuma captura de erros para deixar o código para enviar dados o mais claro possível.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

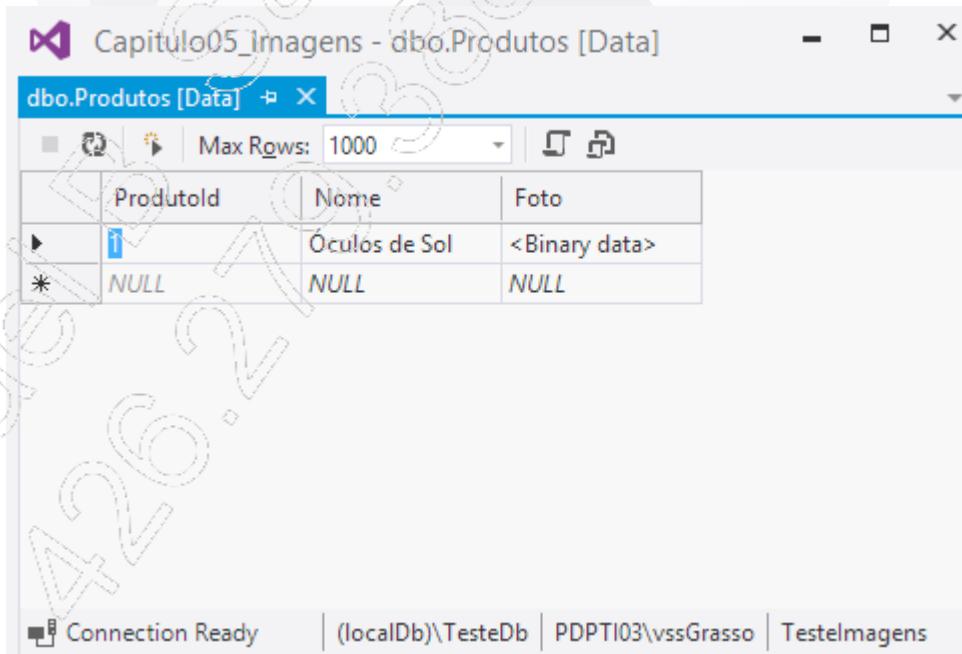
Uma instância da classe **Produto** é preenchida com as informações da tela. Essa instância é enviada para o método **Incluir** da classe **ProdutoDb**.

```
protected void enviarButton_Click(object sender, EventArgs e)
{
    var p = new Produto();
    p.Nome = nomeTextBox.Text;
    p.Foto = imagemFileUpload.FileBytes;

    var db = new ProdutoDb();
    db.Incluir(p);

    mensagemLabel.Text = "Produto Incluído com sucesso";
}
```

Depois de executar a página e verificar o conteúdo inserido na tabela **Produtos**, percebe-se que um conteúdo binário foi inserido:



Produtoid	Nome	Foto
1	Óculos de Sol	<Binary data>
NULL	NULL	NULL

O ideal seria exibir a imagem após fazer o upload, para que o usuário confirme o que acabou de cadastrar. Mas como exibir esse conteúdo, se não existe um arquivo?

Existem duas possibilidades. Uma delas seria gravar os bytes que estão no campo **Foto** do produto em um arquivo e usar como **ImageUrl** do controle **Image** da página. Essa abordagem pode gerar um grande número de imagens que pode lotar o servidor. Além disso, teria que ser tratada a concorrência.

Se duas pessoas usam o mesmo nome para a foto e gravam ao mesmo tempo, algo teria que ser feito para diferenciar uma foto de outra, talvez criando um **Guid** ou incluindo o código dentro do nome da imagem. De qualquer forma, seria necessário criar um meio de manipular arquivos de imagens.

Outra abordagem é utilizar um handler. Um handler é um arquivo que captura as chamadas do IIS e devolve qualquer tipo de arquivo, não só HTML. Pode retornar um arquivo nos formatos XML, PDF, CSV, HTML, entre outros, e também, é claro, imagens.

6.7. Exibindo imagens com um handler

O ASP.NET dispõe de arquivos com a extensão ASHX, chamados de **GenericHandler**. São classes compiladas que implementam a interface **IHTTPHandler**. Essa interface declara um método chamado **ProcessRequest**, responsável por processar a requisição, e uma propriedade chamada **IsReusable**, que determina se a instância criada no momento da chamada permanece em memória. Vejamos, a seguir, o código de um **GenericHandler** adicionado no projeto. Por padrão, é retornado um texto com o conteúdo **Hello World**:

```
public class ExibirImagem : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello World");
    }

    public bool IsReusable
    {
        get{return false; }
    }
}
```

O método **ProcessRequest** recebe um parâmetro do IIS do tipo **HttpContext** que contém dados sobre a requisição. Os principais métodos e propriedades dessa classe **HttpContext** são os seguintes:

- **Request**

Objeto do tipo **HttpRequest** com informações sobre a requisição. Neste exemplo, a propriedade **QueryString** é importante, pois será utilizada para saber qual foto de um produto deverá ser exibida. A chamada do handler para exibir uma foto será esta:

ExibirImagen.ashx?produtoid=1

A variável **Produtoid** passada na URL é obtida usando a propriedade **QueryString**, desta forma:

```
string produtoid=context.Request.QueryString["produtoid"];
```

- **Response**

Objeto do tipo **HttpResponse** que contém informações sobre o que vai ser enviado para o browser. A propriedade importante, nesse caso, é a **ContentType**, que é uma string que define o tipo de conteúdo que será enviado. Veja alguns exemplos de **ContentType**:

Exemplo de ContentType	Descrição
text/plain	Texto puro
text/html	HTML (linguagem de HyperText)
text/css	CSS (folhas de estilo)
image/jpeg	Imagem do tipo JPEG
image/gif	Imagem do tipo GIF
application/vnd.ms-excel	Planilha do Excel
application/pdf	Arquivos PDF (Adobe)
application/xml	Arquivos XML
audio/basic	Arquivo de som (MP3, WAV)
audio/x-wave	Arquivo de som WAV

No caso de exibir uma imagem, o **ContentType** correto é **image/jpeg** ou **image/png** ou, ainda, **image/gif**. No caso da exibição da imagem do produto, o comando será este:

```
Context.Response.ContentType= "image/jpeg";
```

- **BinaryWrite**

Para escrever os dados binários de uma foto no objeto **Response**, é necessário usar o método **BinaryWrite**. Uma vez obtidos os bytes da foto no banco de dados, esses bytes serão enviados nesta resposta:

```
Context.Response.BinaryWrite( bytesDaFoto);
```

Para criar o handler totalmente funcional, é necessário criar um método na classe **ProdutoDb** que retorne um array de bytes de um determinado produto:

```
public class ProdutoDb
{
    public Byte[] ObterFoto(int produtoId)
    {
        string sql = @"Select Foto
                      From Produtos
                      Where ProdutoId=@ProdutoId";

        byte[] foto = null;

        using (var cn = new SqlConnection(Conexao))
        {

            using (var cmd = new SqlCommand(sql, cn))
            {
                cmd.Parameters.AddWithValue("@produtoId", produtoId);
                cn.Open();
                foto = (byte[])cmd.ExecuteScalar();
            }
        }

        return foto;
    }

    public void Incluir(Produto p)...
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

O handler, então, pode obter o id do produto, procurar no banco de dados pela foto e retornar a foto que pode ser exibida por qualquer controle de imagem, por meio da propriedade **ImageUrl**.

```
public void ProcessRequest(HttpContext context)
{
    // Obtém a query string
    string stringProdutoId =
        context.Request.QueryString["produtoId"];

    // Valida se é um número inteiro
    int produtoId=0;
    if (!int.TryParse(stringProdutoId, out produtoId))
    {
        return;
    }
    if (produtoId == 0)
    {
        return;
    }

    // Cria uma instância da Classe ProdutoDb
    var produtoDb = new ProdutoDb();

    // Obtém os bytes da foto
    byte[] foto = produtoDb.ObterFoto(produtoId);

    // Define o tipo de arquivo que será retornado
    context.Response.ContentType = "image/jpeg";

    // Escreve os bytes na resposta
    context.Response.BinaryWrite(foto);
}
```

Para testar, basta chamar o arquivo ASHX, passando como parâmetro um id válido de um produto (<http://servidor/exibirImagem.ashx?produtoid=1>).

Vejamos o resultado:



Para finalizar, no cadastro do produto, é necessário obter o ID que foi inserido. Isso pode ser conseguido efetuando uma pequena mudança no método incluir, para que retorne o ID. Em destaque estão os itens alterados:

```
public int Incluir(Produto p)
{
    string sql = "INSERT INTO PRODUTOS(Nome, Foto)
                  Values(@Nome, @Foto);
                  SELECT @@IDENTITY";
    int produtoId = 0;
    using (var cn = new SqlConnection(Conexao))
    {
        using (var cmd = new SqlCommand(sql, cn))
        {
            cmd.Parameters.AddWithValue("@nome", p.Nome);
            cmd.Parameters.AddWithValue("@foto", p.Foto);
            cn.Open();
            produtoId = Convert.ToInt32(cmd.ExecuteScalar());
        }
    }
    return produtoId;
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Finalmente, na tela de cadastro, os seguintes passos serão executados:

1. Um novo produto é cadastrado, com a foto enviada;
2. O código desse novo produto é obtido;
3. Usando esse código, o handler é chamado e solicitado para exibir a imagem do novo produto;
4. A URL que chama o handler é atribuída à propriedade **ImageUrl** do controle **Image**.

```
protected void enviarButton_Click(object sender, EventArgs e)
{
    // 1
    var p = new Produto();
    p.Nome = nomeTextBox.Text;
    p.Foto = imagemFileUpload.FileBytes;

    // 2
    var db = new ProdutoDb();
    int produtoId = db.Incluir(p);

    // 3
    string url = "ExibirImagen.ashx?produtoId=" + produtoId;

    // 4
    fotoImage.ImageUrl = url;
    imagemFileUpload.Visible = false;
    fotoImage.Visible = true;
    enviarButton.Visible = false;

    mensagemLabel.Text = "Produto Incluído com sucesso";
}
```

Observe o resultado:



6.8. Gravando e exibindo imagens com Entity Framework

Não existe nenhuma configuração especial a ser feita com o Entity Framework em relação ao ADO.NET para gravar e ler fotos em campos do tipo **Image**. Porém, para o projeto apresentado funcionar com o Entity Framework, uma alteração necessária é definir o nome da tabela.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Isso é necessário porque o EF, por padrão, altera o nome das classes para o plural com as regras da língua inglesa. "Produto" vira "Produtos" e não "Produtos". É necessário, então, definir o nome correto da tabela na classe de dados:

```
[Table("Produtos")]
public class Produto
{
    public int ProdutoId { get; set; }
    public string Nome { get; set; }
    public byte[] Foto { get; set; }
}
```

A classe **ProdutoContext** contém a conexão com o banco e a definição dos conjuntos de dados:

```
public class ProdutoContext : DbContext
{
    public ProdutoContext()
        : base(Db.Conexao)
    { }

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

}
```

A classe de manipulação de dados de produtos fica bem mais simples usando os recursos de mapeamento do Entity Framework:

```
public class ProdutoDb
{
    public Byte[] ObterFoto(int produtoId)
    {
        var db = new ProdutoContext();
        Produto prod =
            (from p in db.Produtos
             select p).FirstOrDefault();
        return prod.Foto;
    }
}
```

```
public int Incluir(Produto p)
{
    var db = new ProdutoContext();
    db.Produtos.Add(p);
    db.SaveChanges();
    return p.ProdutoId;

}
```

Nada precisa ser alterado no código das páginas. Esta é a vantagem do encapsulamento. A página ASPX não sabe se o método **Incluir** ou **ObterFoto** usa o Entity Framework ou ADO.NET.

```
var p = new Produto();
p.Nome = nomeTextBox.Text;
p.Foto = imagemFileUpload.FileBytes;

var db = new ProdutoDb();
int produtoId=db.Incluir(p);

string url = "ExibirImagen.ashx?produtoId=" + produtoId;
fotoImage.ImageUrl = url;
imagemFileUpload.Visible = false;
fotoImage.Visible = true;
enviarButton.Visible = false;

mensagemLabel.Text = "Produto Incluído com sucesso";
```

Executando o programa, o resultado é exatamente o mesmo. Os bytes da imagem são incluídos no banco de dados e podem ser recuperados por meio do mesmo handler.



6.9. Northwind e as imagens gravadas no banco de dados

Na tabela **Employee** do banco de dados **Northwind**, existe o campo **Photo** que contém a foto de cada funcionário. Os bytes gravados, porém, não correspondem a um arquivo de imagem, e os procedimentos exibidos anteriormente não funcionam para visualizar as imagens já gravadas nesse banco de dados. Isso acontece apenas nas primeiras versões do Northwind.

O problema é que o formato em que estão gravadas as imagens é o **OLE DB Object**, que tem 78 bytes antes da imagem. Uma solução simples para esse caso é pular os 78 primeiros bytes. A imagem começa no byte 79 até o fim. Isso é facilmente executado colocando os bytes na memória usando **MemoryStream**:

```
public void ProcessRequest(HttpContext context)
{
    int id = 0;
    int.TryParse(context.Request.QueryString["id"], out id);
    if (id <= 0) return;

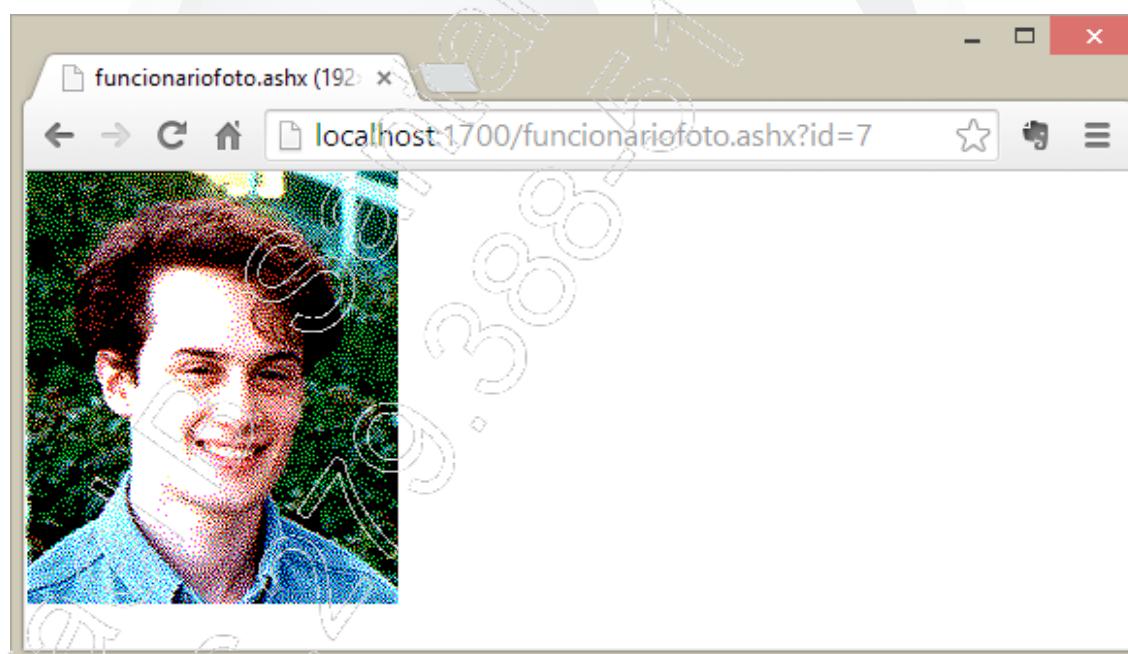
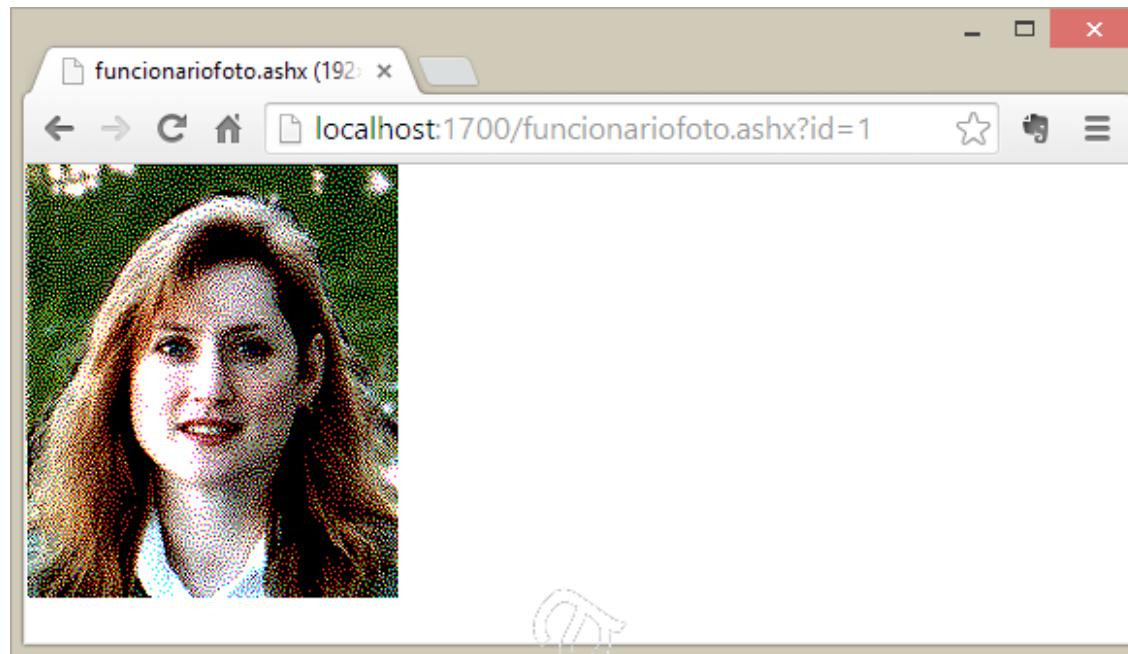
    var db = new NorthwindEntities();
    var f = db.Employees.Find(id);
    if (f == null) return;
    byte[] foto = f.Photo;

    int offset = 78;

    //Cria um stream na memória começando no byte 79 até o fim
    var m=new MemoryStream(foto, offset, foto.Length-offset);

    //Escreve no response um array com a imagem
    context.Response.ContentType = "image/jpeg";
    context.Response.BinaryWrite(m.ToArray());
}
```

Vejamos o resultado ao obtermos as imagens da tabela **Employee** no formato **OLE DB Object**:



6.10. Vinculando imagens no GridView

O User Control **GridView** aceita diversos tipos de colunas para visualização de dados. Para exibir imagens, é possível usar colunas do tipo **Image**, **Template**, **Command** ou **HyperLink**. É possível, também, exibir imagens em colunas comuns (**Binding Columns**) inserindo código HTML diretamente no texto, sem usar encoding. Observe, a seguir, uma lista dos tipos de colunas do **GridView**:

- **BoundField:** Mostra o valor de um campo de uma fonte de dados. Este é o tipo de coluna criado automaticamente;
- **ButtonField:** Exibe um Button ou um LinkButton para cada linha. Executa um comando que pode ser capturado no evento **RowCommand**;
- **CheckBoxField:** Exibe um CheckBox para campos do tipo booleano;
- **CommandField:** Exibe um botão do tipo Button ou HyperLink com comandos pré-definidos como **Incluir**, **Alterar**, **Excluir**, **Gravar** etc.;
- **HyperLinkField:** Exibe um HyperLink permitindo criar vínculos para outros campos ou páginas;
- **ImageField:** Exibe uma imagem;
- **TemplateField:** Campo personalizável. É possível colocar qualquer tipo de objeto.

Um controle **GridView** sem nenhuma personalização tem a seguinte sintaxe e resultado:

- **HTML**

```
<asp:GridView runat="server" ID="gv"></asp:GridView>
```

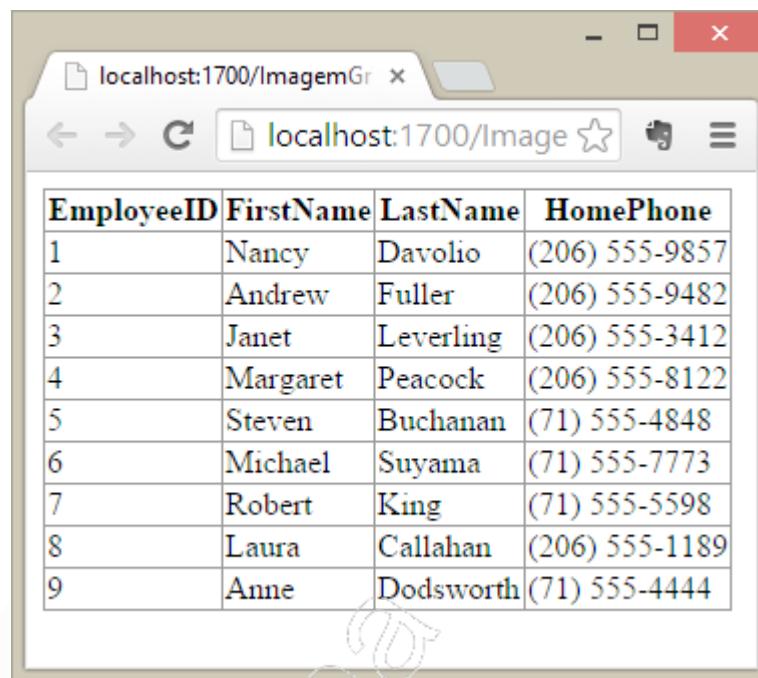
- **Code-behind (evento Page_Load)**

```
var db = new NorthwindEntities();
var lista = from c in db.Employees
           select new
           {
               c.EmployeeID,
               c.FirstName,
               c.LastName,
               c.HomePhone };

gv.DataSource = lista.ToList();
gv.DataBind();
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Vejamos o resultado:



A screenshot of a web browser window titled "localhost:1700/ImagenGr" showing a table of employee data. The table has four columns: EmployeeID, FirstName, LastName, and HomePhone. The data is as follows:

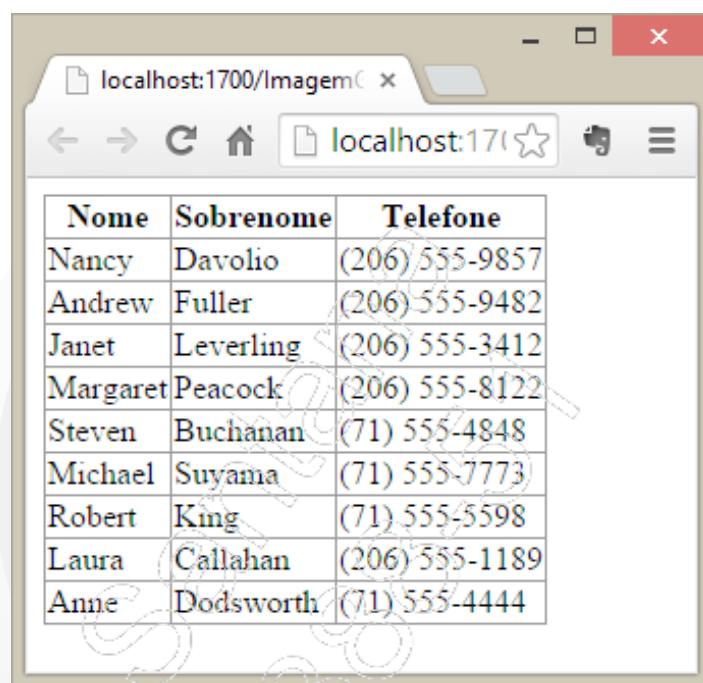
EmployeeID	FirstName	LastName	HomePhone
1	Nancy	Davolio	(206) 555-9857
2	Andrew	Fuller	(206) 555-9482
3	Janet	Leverling	(206) 555-3412
4	Margaret	Peacock	(206) 555-8122
5	Steven	Buchanan	(71) 555-4848
6	Michael	Suyama	(71) 555-7773
7	Robert	King	(71) 555-5598
8	Laura	Callahan	(206) 555-1189
9	Anne	Dodsworth	(71) 555-4444

A seguir, uma pequena mudança para definir manualmente as colunas:

```
<asp:GridView runat="server"
    ID="gv"
    AutoGenerateColumns="False">
    <Columns>
        <asp:BoundField DataField="FirstName"
            HeaderText="Nome" />
        <asp:BoundField DataField="LastName"
            HeaderText="Sobrenome" />
        <asp:BoundField DataField="Phone"
            HeaderText="Telefone" />
    </Columns>
</asp:GridView>
```

A coleção **Columns** contém um conjunto de objetos derivados de **DataControlField**, nesse caso, a classe **BoundField** (**System.Web.Controls**). As principais propriedades da classe **BoundField** são as seguintes:

- **DataField**: O nome da propriedade do objeto vinculado à grid que está associada a esta coluna;
- **HeaderText**: O texto que será exibido no cabeçalho da coluna.



The screenshot shows a web browser window with the URL "localhost:1700/Imagen". Inside the browser, a GridView control is displayed, showing a table with three columns: Nome, Sobrenome, and Telefone. The data rows are as follows:

Nome	Sobrenome	Telefone
Nancy	Davolio	(206) 555-9857
Andrew	Fuller	(206) 555-9482
Janet	Leverling	(206) 555-3412
Margaret	Peacock	(206) 555-8122
Steven	Buchanan	(71) 555-4848
Michael	Suyama	(71) 555-7773
Robert	King	(71) 555-5598
Laura	Callahan	(206) 555-1189
Anne	Dodsworth	(71) 555-4444

A GridView contém a propriedade **AutoGeneratedColumns** para definir se as colunas serão geradas dinamicamente. Nesse caso, é necessário definir em **False** para que não seja gerada nenhuma coluna, pois elas serão definidas manualmente.

```
<asp:GridView runat="server"
    ID="gv"
    AutoGenerateColumns="False">
    ...
</asp:GridView>
```

Para exibir a foto, pode ser usada uma coluna do tipo **ImageField**, classe também derivada de **DataControlfield**.

```
<asp:GridView runat="server"
    ID="gv"
    AutoGenerateColumns="False">

    <Columns>
        <asp:BoundField DataField="FirstName"
            HeaderText="Nome" />

        <asp:BoundField DataField="LastName"
            HeaderText="Sobrenome" />

        <asp:BoundField DataField="Phone"
            HeaderText="Telefone" />

        <asp:ImageField
            DataImageUrlField="EmployeeId"
            DataImageUrlFormatString="FuncionarioFoto.ashx?id={0}"
            HeaderText="Foto">

    </Columns>
</asp:GridView>
```

As seguintes propriedades estão sendo usadas na definição deste campo:

- **DataImageUrlField**: O nome da propriedade da fonte de dados relacionada a esta coluna. Nesse caso, é o ID do funcionário, porque a exibição da imagem será feita pelo handler que recebe o id como parâmetro. Exemplo:

FuncionarioFoto.ashx?id=1

- **DataImageUrlFormatString**: Esta é a propriedade que faz tudo funcionar. Uma string de formatação é aplicada no(s) campo(s) definido(s) da propriedade **DataImageUrlField**. Nesse caso, é a URL do handler com o parâmetro necessário, que é o ID do funcionário:

Funcionario.ashx?id={0}

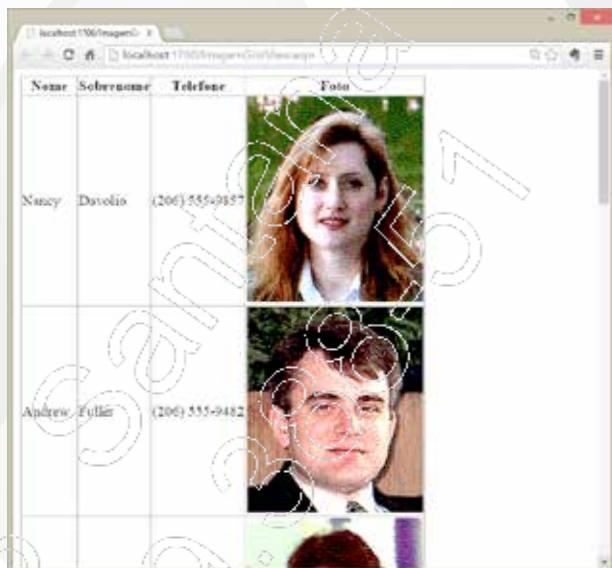
Quando a GridView é vinculada, o registro que contém o ID número 1 vai ser renderizado desta forma:

Funcionario.ashx?id=1

Quando o ID do funcionário for o número 37, a expressão será renderizada desta forma:

Funcionario.ashx?id=37

Ou seja, o parâmetro {0} é substituído dinamicamente pelo ID do funcionário. O resultado é a lista com as fotos:



Para definir um estilo para a foto, a melhor opção é criar um estilo CSS. Para a imagem, utilizamos a propriedade **ControlStyle-CssClass** da coluna:

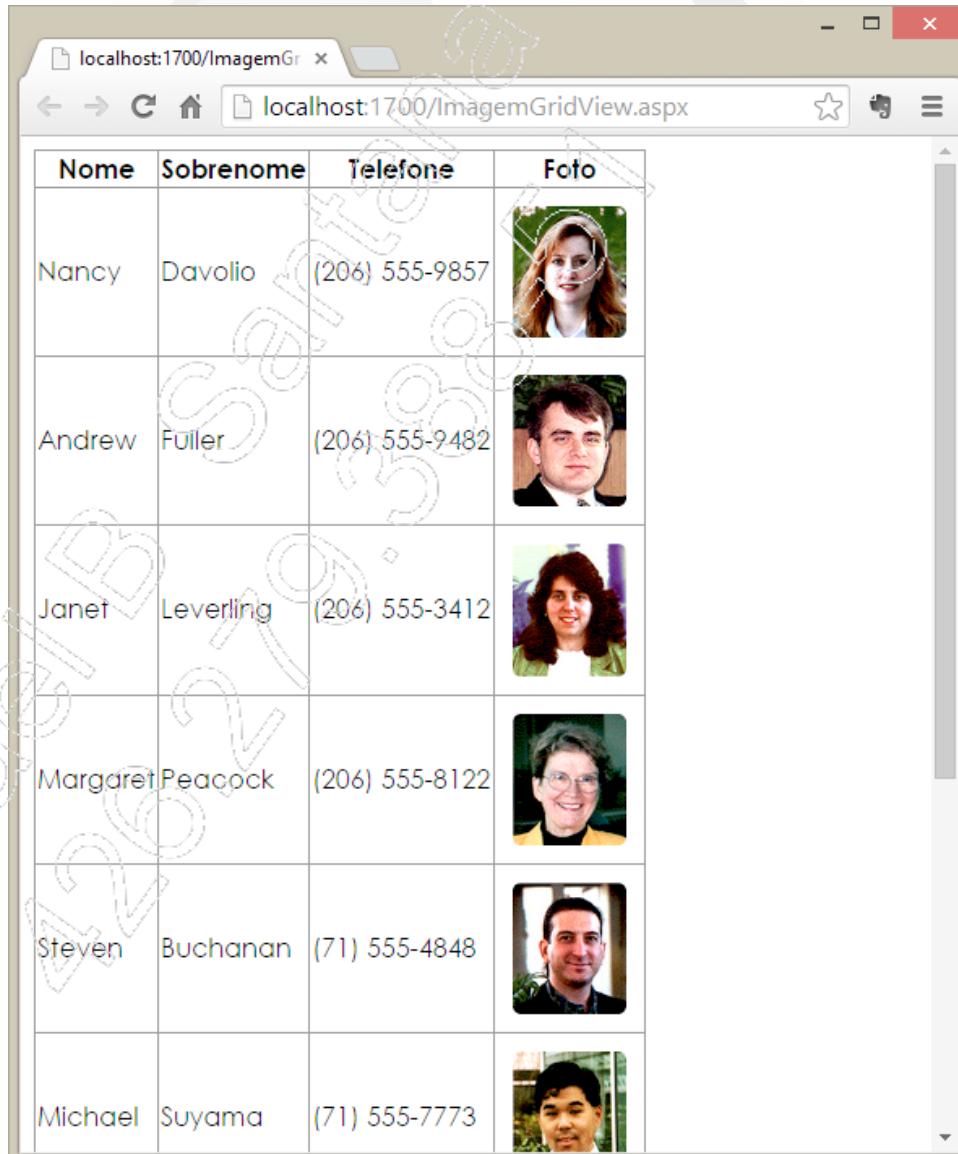
```
<asp:ImageField  
    DataImageUrlField="EmployeeId"  
    DataImageUrlFormatString="FuncionarioFoto.ashx?id={0}"  
    HeaderText="Foto"  
    ControlStyle-CssClass="FotoGridView">
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Na folha de estilo, a classe CSS deve ser definida. No exemplo a seguir, foram definidos uma largura fixa de 70px, uma margem ao redor da imagem de 10px e os cantos arredondados:

```
.FotoGridView {  
    width:70px;  
    display:block;  
    margin:10px;  
    border-radius:5px;  
}
```

Vejamos o resultado:



A screenshot of a web browser window titled "localhost:1700/ImagenGrid.aspx". The page displays a GridView control with four columns: Nome, Sobrenome, Telefone, and Foto. The data rows represent employees from the Northwind database. The "Foto" column contains small profile pictures for each employee.

Nome	Sobrenome	Telefone	Foto
Nancy	Davolio	(206) 555-9857	
Andrew	Fuller	(206) 555-9482	
Janet	Leverling	(206) 555-3412	
Margaret	Peacock	(206) 555-8122	
Steven	Buchanan	(71) 555-4848	
Michael	Suyama	(71) 555-7773	

Pontos principais

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

- Para enviar imagens para o servidor, usa-se o Web Control **FileUpload** (Web Forms) ou **HttpPostedFileBase** (MVC);
- O namespace **System.Drawing** contém classes para tratamento de imagens, sendo as mais importantes a classe abstrata **Image** e a classe **Bitmap**;
- O ADO.NET pode gravar imagens em campos do tipo **Image** no SQL Server e, no .NET Framework, a classe de mapeamento pode usar um array de bytes. Nenhuma conversão é necessária, nem usando ADO.NET, nem usando o Entity Framework;
- O método **GetThumbnailImage** da classe **Bitmap** pode ser usado para criar uma miniatura de um imagem;
- O namespace **System.IO** contém classes para manipulação de streams e de arquivos, sendo útil na hora de gravar e tratar imagens;
- O método **BinaryWrite** deve ser usado para gravar informações binárias no arquivo de saída de um handler.

6

Manipulando imagens

Teste seus conhecimentos

Mikael Braga
426.273.8657



IMPACTA
EDITORA

1. Qual Web Control é usado para enviar arquivos ao servidor quando o modelo de interface é Web Forms?

- a) FileSend
- b) StreamWriter
- c) Submit Button
- d) SendFile
- e) FileUpload

2. Em qual namespace estão as classes para manipulação de arquivos?

- a) System.Files
- b) System.FileUpload
- c) System.IO
- d) System.Common.Files
- e) System.FileUpload

3. Em qual namespace estão as classes para manipulação de imagens?

- a) System.IO
- b) System.FileUpload
- c) System.Images
- d) System.Bitmap
- e) System.Drawing

**4. Qual é a principal classe para manipulação de imagens?
Essa classe é derivada de qual classe?**

- a) Image, que é derivada de Bitmap.
- b) Bitmap, que é derivada de PostedFile.
- c) Image, que é derivada de PostedFile.
- d) PostedFile, que é derivada de Bitmap.
- e) Bitmap, que é derivada de Image.

5. Para gravar uma imagem no SQL Server, qual tipo de campo deve ser utilizado?

- a) VARCHAR(MAX)
- b) BITMAP
- c) IMAGE
- d) BYTE(MAX)
- e) FILE

6

Manipulando imagens

Mãos à obra!

Mikael Alana
426.270.3888



IMPACTA
EDITORA

Laboratório 1

A - Criando uma tela de cadastro de funcionários com foto

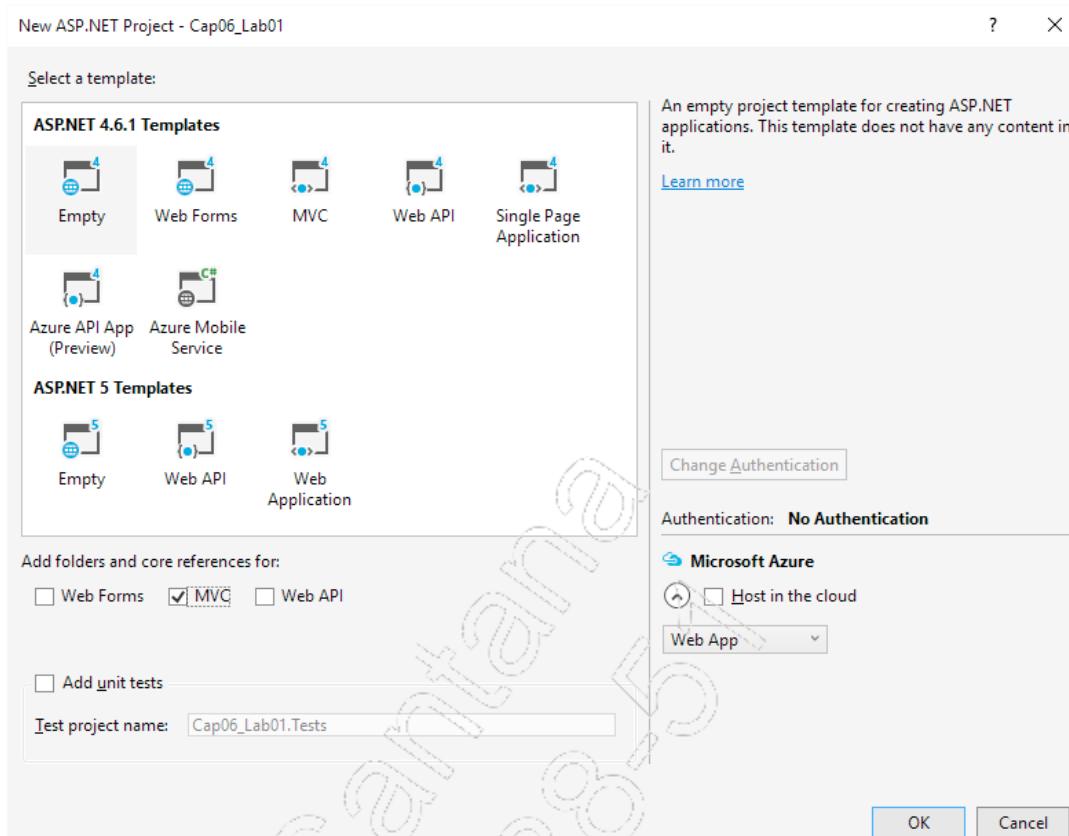
Neste laboratório, iremos criar uma tela de cadastro de funcionários da empresa Northwind com foto. As imagens a seguir mostram exemplos das telas da lista de funcionários e do formulário:

The image contains three screenshots of the Northwind application interface:

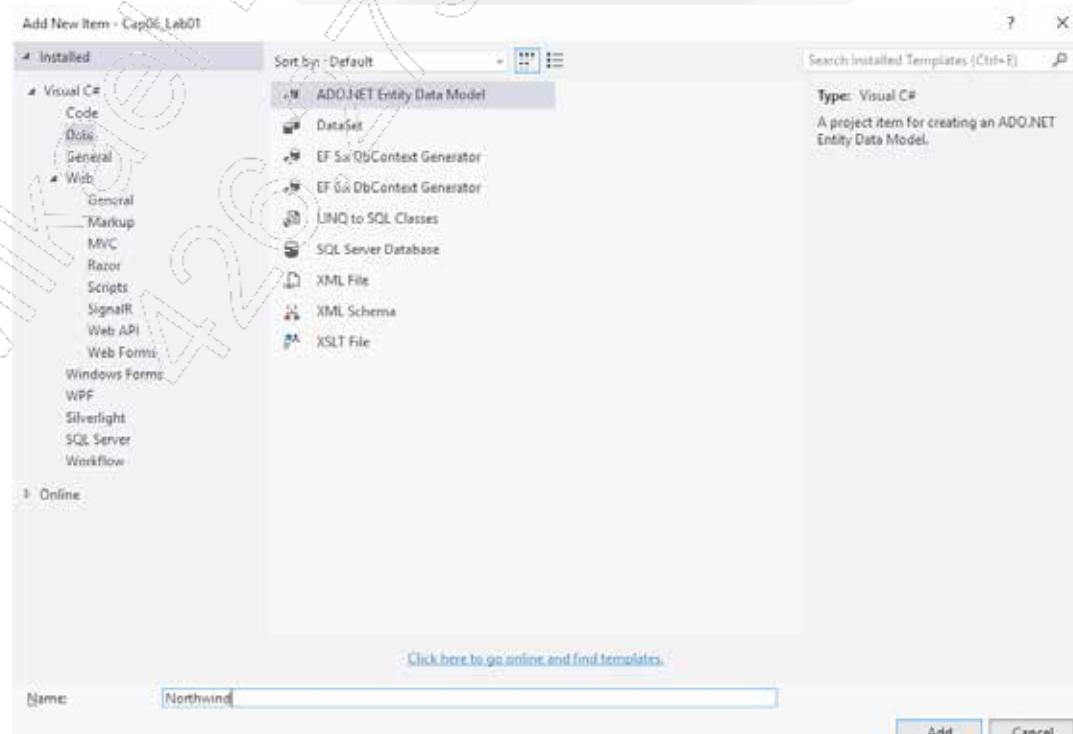
- Screenshot 1 (Left): Funcionários List**
A grid view showing a list of employees. Each row includes a small photo, the employee's name, city, country, and phone number. At the bottom right of the grid, there are buttons for 'Novo', 'Editar', and 'Excluir'.
- Screenshot 2 (Top Right): Alterar Funcionário Form**
A detailed form for editing an employee. It includes fields for Name (Nome), Birthdate (Nascimento), Address (Endereço), City (Cidade), State (Estado), Zip (CEP), Phone (Telefone), and Email (Email). There is also a 'Notas' (Notes) text area containing a short biography about Anne Buchanan.
- Screenshot 3 (Bottom Right): Another Alterar Funcionário Form**
This form is identical to the one above but shows a different employee profile picture of Andrew Sorenson.

Parte 1 – Criando o projeto e adicionando o modelo de dados

1. Inicie criando a estrutura do projeto. Para isso, crie um novo projeto Web vazio chamado **Cap06_Lab01** e marque referências para MVC;

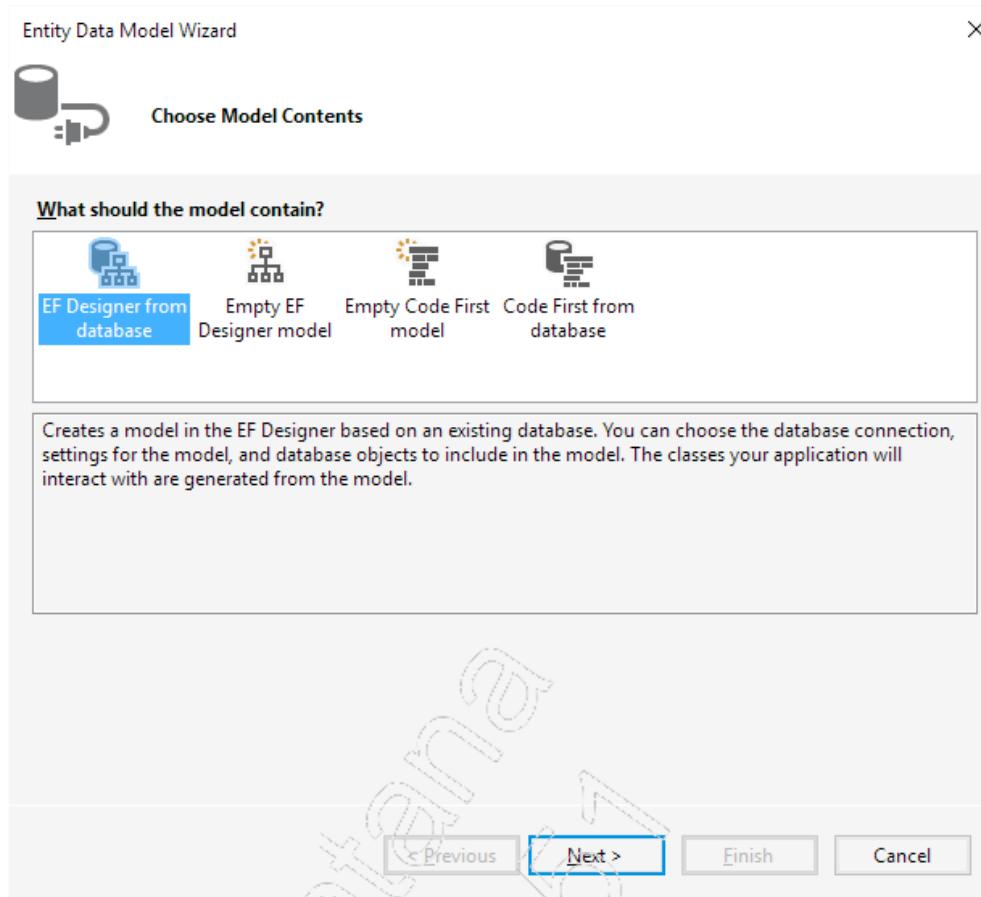


2. Adicione no projeto um **ADO.NET Entity Data Model** chamado **Northwind**;

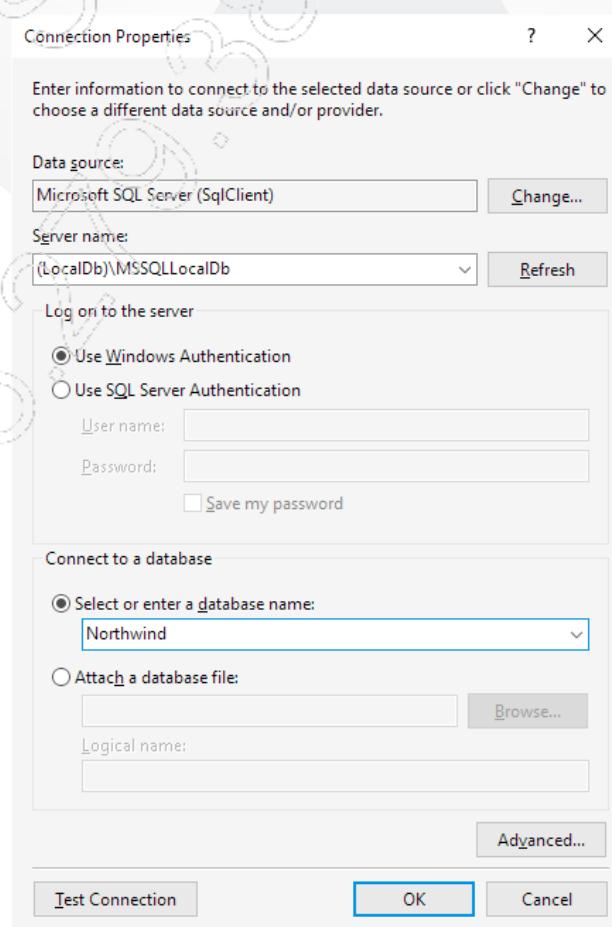


Visual Studio 2015 - ASP.NET com C# Acesso a dados

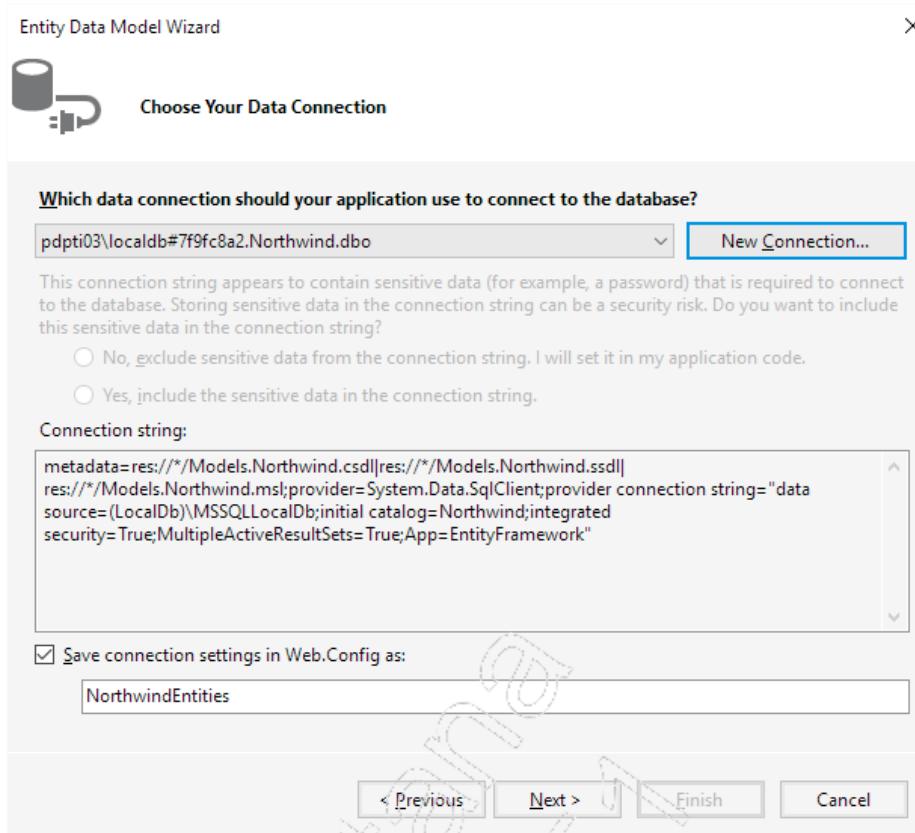
3. No assistente, escolha EF Designer from database:



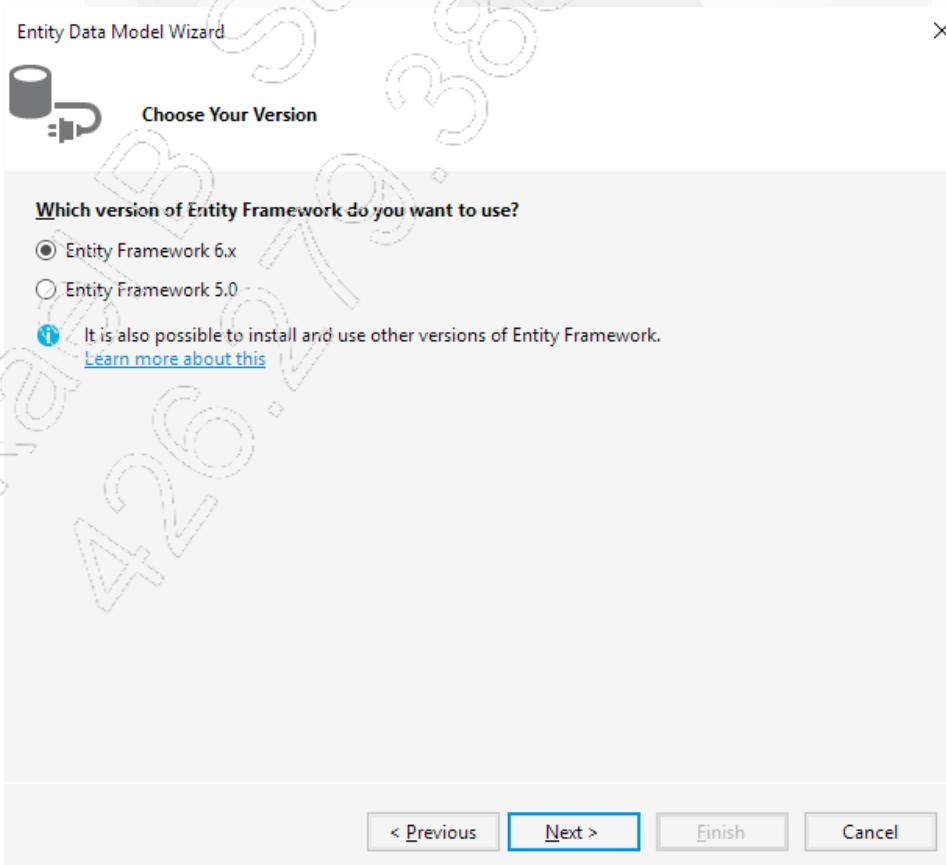
4. Conecte o Northwind usando o server (LocalDb)\MSSQLLocalDb;



5. Clique em **Next**:

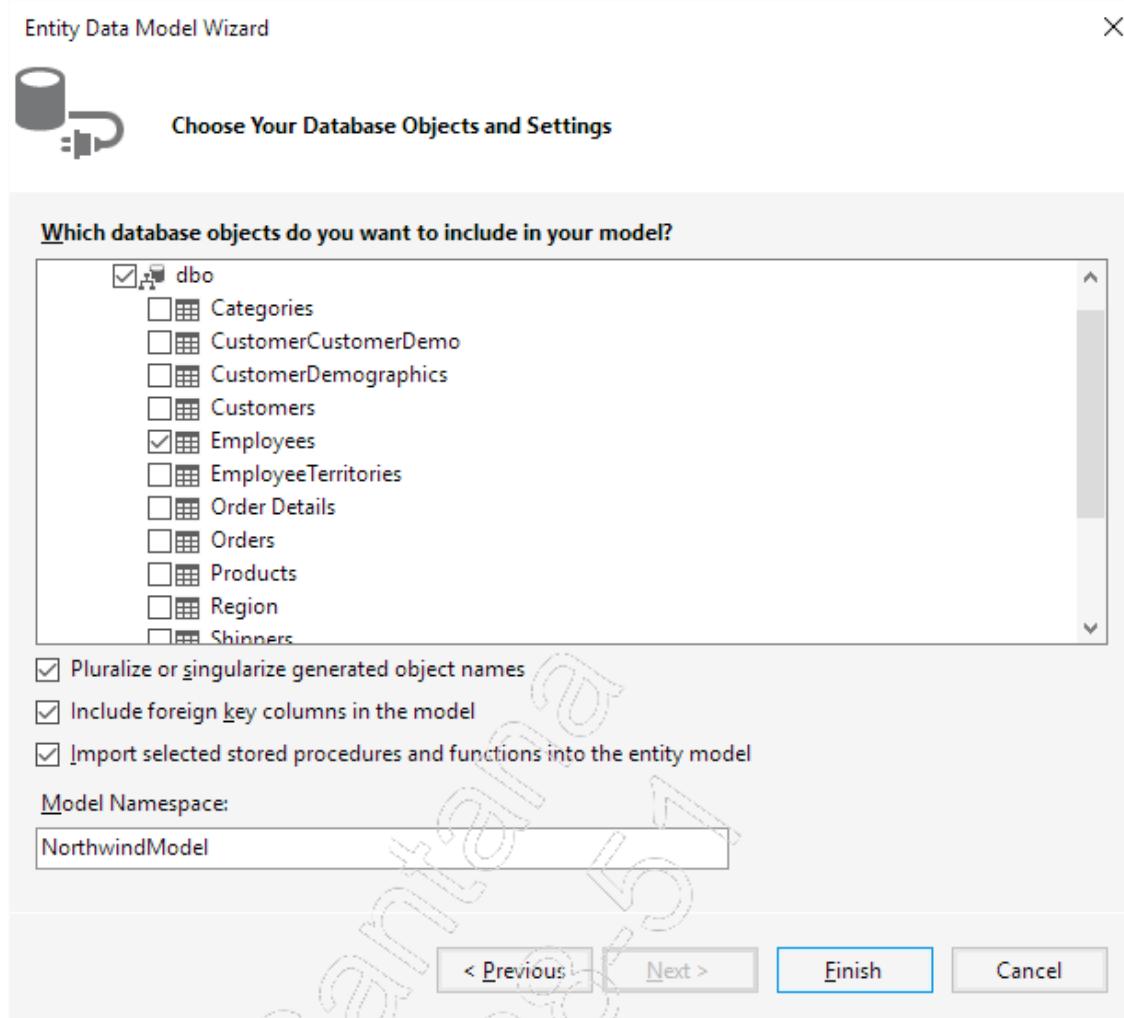


6. Se for questionada a versão, escolha a mais recente:

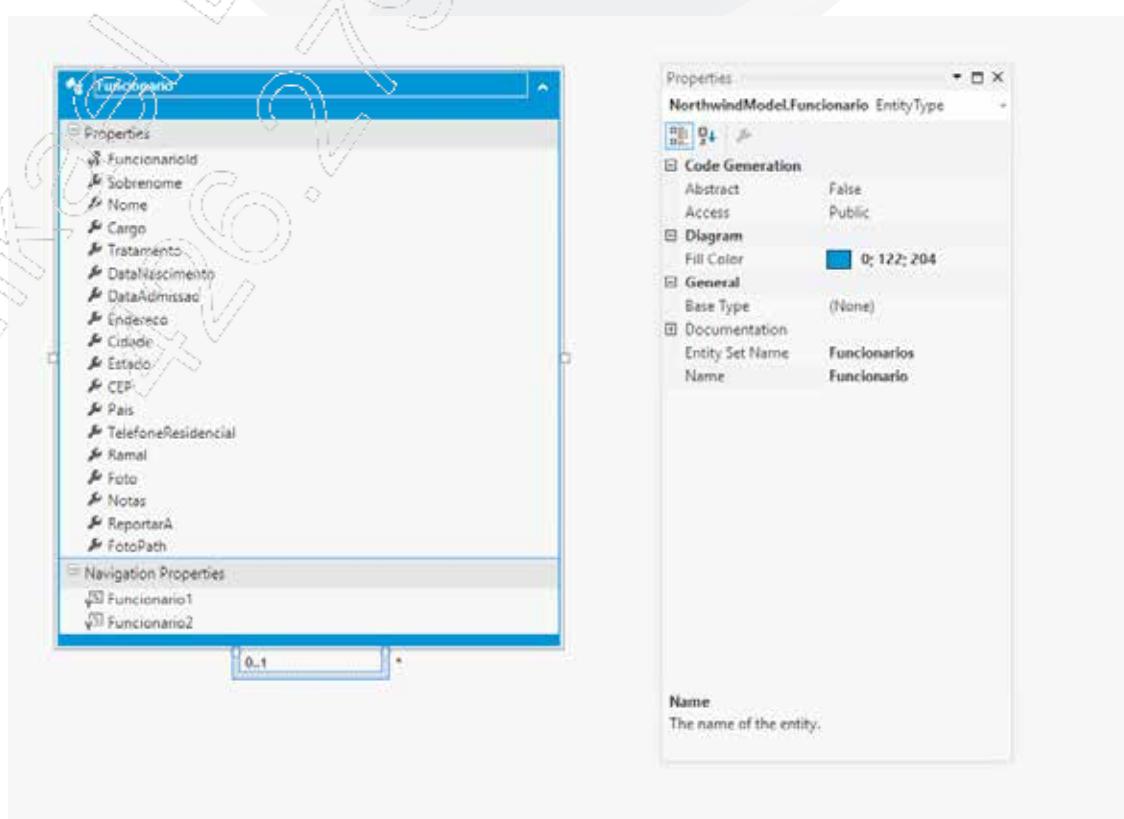


Visual Studio 2015 - ASP.NET com C# Acesso a dados

7. Selecione a tabela **Employees** (funcionários);



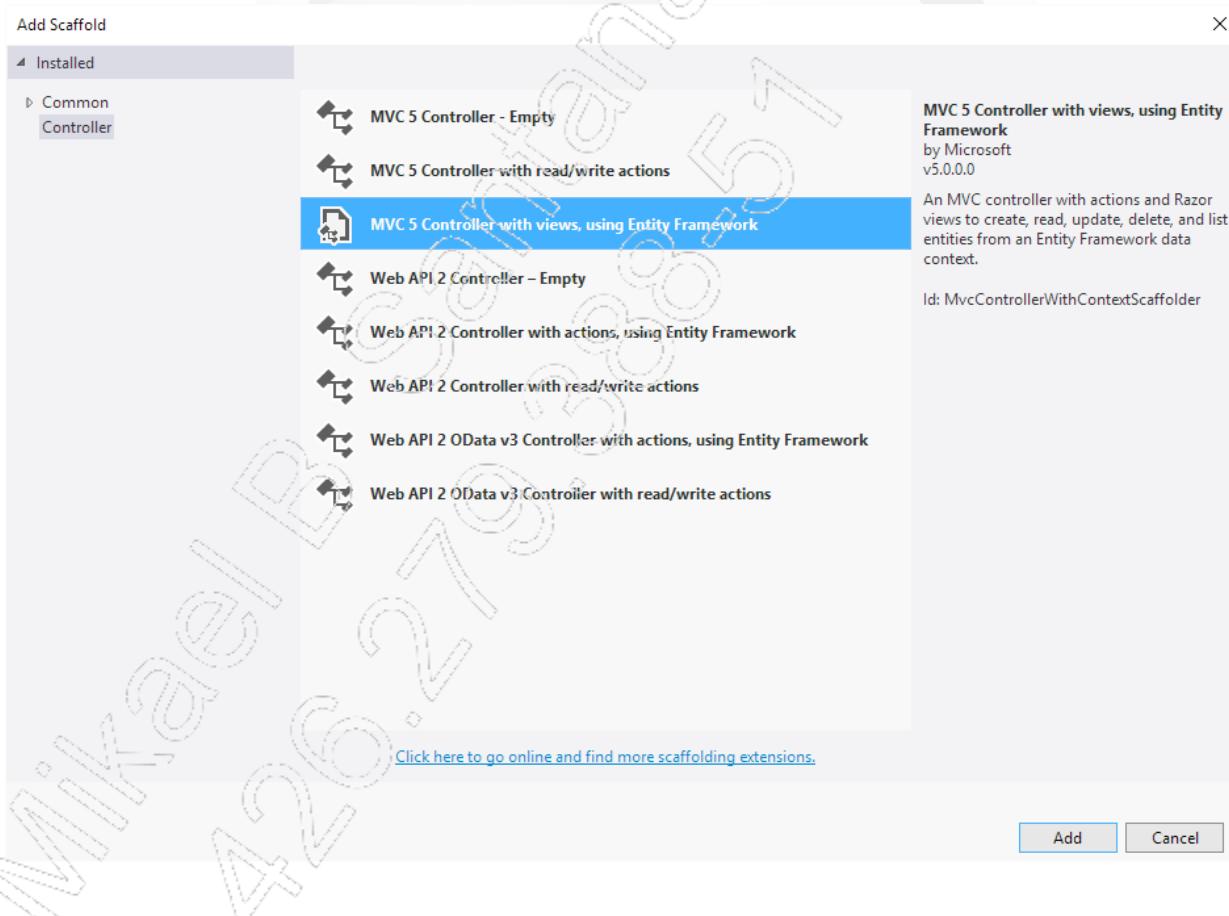
8. No diagrama, altere o nome dos campos. Altere, também, o nome da entidade (**Funcionario**) e o nome do conjunto (**Funcionarios**);



9. Salve todos os arquivos e compile o projeto. Tenha certeza de que não existem erros.

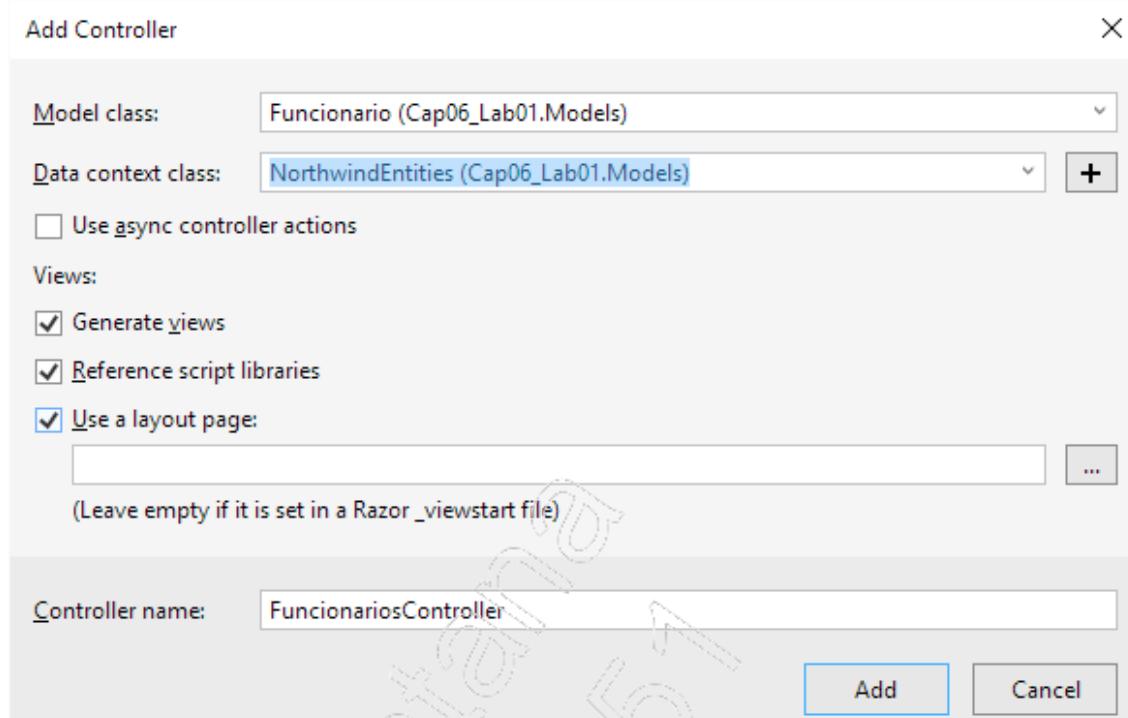
Parte 2 – Adicionando o Controller e as Views usando o assistente do MVC

1. Na pasta **Controllers**, escolha, no menu de contexto, **Add / Controller**. Escolha **MVC 5 Controller with views, using Entity Framework**. Esse assistente vai criar um Controller com um esqueleto das operações básicas (**Listar, Incluir, Alterar e Excluir**) e, também, as telas de visualização e edição;



Visual Studio 2015 - ASP.NET com C# Acesso a dados

2. Na janela de definição, é necessário definir a classe de modelo de dados (**Funcionario**) e a classe de acesso a dados do Entity Framework (**NorthwindEntities**). Marque a opção **Generate views** para criar as telas;



O assistente cria diversas telas, usando todos os campos. Neste ponto, é possível executar o programa. Não é qualquer tipo de campo que é tratado corretamente. O campo **Foto**, por exemplo, não tem um editor.

O layout das telas apenas lista os campos, sem agrupamento ou tratamento de formatos. Vamos usar o que foi gerado como ponto de partida, mas todas as telas serão alteradas.

3. Na pasta **Views \ Funcionarios**, abra o arquivo **Index.cshtml**. Altere as informações. Deixe apenas os campos **Nome, Cidade, País e Telefone Residencial**. As linhas alteradas estão destacadas:

```
@model IEnumerable<Cap06_Lab01.Models.Funcionario>

{@
    ViewBag.Title = "Index";
}

<h2>Funcionários</h2>
<hr/>

<div class="container">

    <p>
        @Html.ActionLink("Novo", "Create")
    </p>

<table class="table">
    <tr>
        <th>
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Nome)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Cidade)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Pais)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.TelefoneResidencial)
        </th>
    <th></th>
    </tr>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
@foreach (var item in Model)
{
    <tr>

        <td>
            @if (!string.IsNullOrEmpty(item.FotoPath))
            {
                
            }
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Nome)
            @Html.DisplayFor(modelItem => item.Sobrenome)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Cidade)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Pais)
        </td>

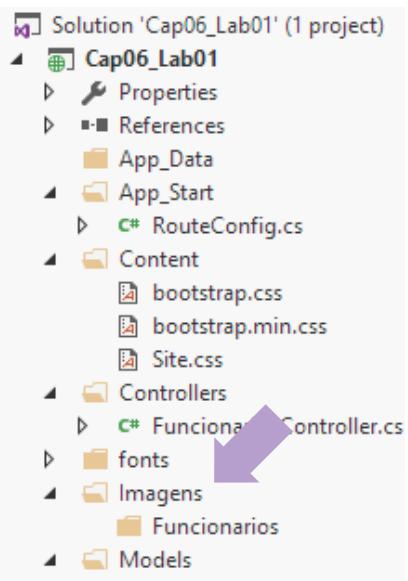
        <td>
            @Html.DisplayFor(modelItem => item.TelefoneResidencial)
        </td>

        <td>
            @Html.ActionLink("Alterar", "Edit", new { id = item.FuncionarioId }) |
            @Html.ActionLink("Exibir", "Details", new { id = item.FuncionarioId }) |
            @Html.ActionLink("Excluir", "Delete", new { id = item.FuncionarioId })
        </td>
    </tr>
}
```

</table>

</div>

4. Crie uma pasta chamada **Imagens / Funcionarios** na solução;



- **Listagem**

5. Teste a listagem. Neste ponto, ainda não serão exibidas as imagens. Quando forem cadastradas as imagens, a listagem ficará assim:

Funcionários					
Novo		Nome	Cidade	País	Telefone
	Nancy Davolio	Seattle	USA	(206) 555-9657	Alterar Excluir Exportar
	Andrew Fuller	Tacoma	USA	(206) 555-9482	Alterar Excluir Exportar
	Janet Leverling	Kirkland	USA	(206) 555-3412	Alterar Excluir Exportar
	Margaret Peacock	Redmond	USA	(206) 555-8122	Alterar Excluir Exportar
	Steven Buchanan	London	UK	(71) 555-4840	Alterar Excluir Exportar
	Michael Suyama	London	UK	(71) 555-7773	Alterar Excluir Exportar
	Robert King	London	UK	(71) 555-5690	Alterar Excluir Exportar
	Laura Callahan	Seattle	USA	(206) 555-1189	Alterar Excluir Exportar
	Anne Dodsworth	London	UK	(71) 555-4444	Alterar Excluir Exportar

6. Na classe **FuncionarioController**, não é necessária nenhuma alteração no método **Index**. O método **Include** carrega os dados do campo **ReportarA**, que é uma referência a um funcionário da mesma tabela, nesse caso, a quem esse funcionário se reporta, ou seja, quem é o chefe dele. Esse chefe também é um funcionário. Esse relacionamento do banco de dados Northwind de uma tabela com a mesma tabela é muito usado em sistemas que usam listas encadeadas;

```
public ActionResult Index()
{
    var funcionarios = db.Funcionarios.Include(f => f.Fucionario2);
    return View(funcionarios.ToList());
}
```

- **Inclusão de dados**

7. A inclusão precisa tratar as imagens. Na classe **FuncionarioController**, crie os seguintes métodos:

```
private static void ObterFotoUpload(Funcionario funcionario,
                                     HttpPostedFileBase fotoUpload)
{
    if (fotoUpload != null && fotoUpload.ContentLength > 0)
    {
        MemoryStream ms = new MemoryStream();
        fotoUpload.InputStream.CopyTo(ms);
        var byts = ms.ToArray();
        ms.Dispose();

        byte[] foto = byts;
        funcionario.Foto = foto;
    }
}
```

```
private void AtualizarFoto(Funcionario funcionario)
{
    string fotoPathVirtual =
        "~/imagens/" + funcionario.FuncionarioId + ".jpg";
    string fotoPathFisico = Server.MapPath(fotoPathVirtual);
    funcionario.FotoPath = fotoPathVirtual;
    if (funcionario.Foto != null && funcionario.Foto.Length > 0)
    {
        System.IO.File.WriteAllBytes(fotoPathFisico, funcionario.Foto);
        db.Entry(funcionario).State = EntityState.Modified;
    }

    db.SaveChanges();
}
```

8. Crie os métodos para gerar o formulário e inserir um funcionário:

```
public ActionResult Create()
{
    ViewBag.ReportarA =
        new SelectList(db.Funcionarios, "FuncionarioId", "Sobrenome");

    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "FuncionarioId,Sobrenome,Nome,Cargo,Tratamento,DataNascimento,DataAdmissao,Endereco,Cidade,Estado,CEP,Pais,-TelefoneResidencial,Ramal,Notas,ReportarA")] Funcionario funcionario, HttpPostedFileBase fotoUpload)
{
    if (ModelState.IsValid)
    {
        ObterFotoUpload(funcionario, fotoUpload);
        try
        {
            db.Funcionarios.Add(funcionario);
            db.SaveChanges();
            AtualizarFoto(funcionario);
            return RedirectToAction("Index");
        }
        catch (System.Data.Entity.Validation.DbEntityValidationException ex)
        {
            foreach (var erro in ex.EntityValidationErrors)
            {
                foreach (var item in erro.ValidationErrors)
                {
                    ModelState.AddModelError("", item.ErrorMessage);
                }
            }

            ViewBag.ReportarA =
                new SelectList(db.Funcionarios, "FuncionarioId", "Sobrenome");
        }
    }
}

ViewBag.ReportarA =
    new SelectList(db.Funcionarios, "FuncionarioId", "Sobrenome",
        funcionario.Reportara);

return View(funcionario);
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

9. Altere a tela da View **Create**, para implementar o novo layout. A parte principal é a definição das propriedades, que utiliza os métodos **LabelFor**, **EditorFor** e **ValidationMessageFor**:

```
@model Cap06_Lab01.Models.Funcionario

{@
    ViewBag.Title = "Create";
}

<h2>Incluir Funcionário</h2>

@using (Html.BeginForm("Create", "Funcionarios", null, FormMethod.Post, new
{ enctype = "multipart/form-data" }))
{
    @Html.AntiForgeryToken()

    <h4>Funcionario</h4>
    <hr />

    @Html.ValidationSummary(true, "", new { @class = "text-danger" })

    <div class="row">
        <div class="col-md-5">
            <div class="row">

                <div class="form-group col-md-6">
                    @Html.LabelFor(model => model.Nome)
                    @Html.EditorFor(model => model.Nome, new { htmlAttributes =
= new { @class = "form-control" } })
                    @Html.ValidationMessageFor(model => model.Nome, "", new { @class =
= "text-danger" })
                </div>

                <div class="form-group col-md-6">
                    @Html.LabelFor(model => model.Sobrenome)
                    @Html.EditorFor(model => model.Sobrenome, new {
htmlAttributes = new { @class = "form-control" } })
                    @Html.ValidationMessageFor(model => model.Sobrenome,
"", new { @class = "text-danger" })
                </div>

            </div>
        </div>
    </div>
}
```

```
<div class="row">
    <div class="form-group col-md-8">
        @Html.LabelFor(model => model.Cargo)
        @Html.EditorFor(model => model.Cargo, new { htmlAttributes
= new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Cargo, "", new { @class = "text-danger" })
    </div>
```

```
<div class="form-group col-md-4">
    @Html.LabelFor(model => model.Tratamento)
    @Html.EditorFor(model => model.Tratamento, new {
htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.Tratamento,
"", new { @class = "text-danger" })
</div>
```

```
<div class="row">
    <div class="form-group col-md-6">
        @Html.LabelFor(model => model.DataNascimento)
        @Html.EditorFor(model => model.DataNascimento, new {
htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.DataNascimento,
"", new { @class = "text-danger" })
    </div>
```

```
<div class="form-group col-md-6">
    @Html.LabelFor(model => model.DataAdmissao)
    @Html.EditorFor(model => model.DataAdmissao, new {
htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.DataAdmissao,
"", new { @class = "text-danger" })
</div>
```

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

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
<div class="form-group">
    <input type="file" name="fotoUpload" id="fotoUpload" />
</div>

</div>

<div class="col-md-5">
    <div class="form-group">
        @Html.LabelFor(model => model.Endereco)
        @Html.EditorFor(model => model.Endereco, new { htmlAttributes =
            new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Endereco, "", new { @class =
            "text-danger" })
    </div>

```



```
<div class="form-group">
    <div class="row">
        <div class="col-md-5">
            @Html.LabelFor(model => model.Cidade)
            @Html.EditorFor(model => model.Cidade, new {
                htmlAttributes = new { @class = "form-control" } })
            @Html.ValidationMessageFor(model => model.Cidade, "", new { @class =
                "text-danger" })
        </div>

```



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

```



```
<div class="col-md-4">
    @Html.LabelFor(model => model.CEP)
    @Html.EditorFor(model => model.CEP, new {
        htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.CEP, "", new { @class =
        "text-danger" })
</div>

```



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

```
<div class="form-group">  
    </div>  
  
<div class="form-group">  
    @Html.LabelFor(model => model.Pais)  
    @Html.EditorFor(model => model.Pais, new { htmlAttributes  
= new { @class = "form-control" } })  
    @Html.ValidationMessageFor(model => model.Pais, "", new  
{ @class = "text-danger" })  
    </div>
```

```
<div class="row">  
    <div class="form-group col-md-8">  
        @Html.LabelFor(model => model.TelefoneResidencial)  
        @Html.EditorFor(model => model.TelefoneResidencial,  
new { htmlAttributes = new { @class = "form-control" } })  
        @Html.ValidationMessageFor(model => model.  
TelefoneResidencial, "", new { @class = "text-danger" })  
    </div>
```

```
    <div class="form-group col-md-4">  
        @Html.LabelFor(model => model.Ramal)  
        @Html.EditorFor(model => model.Ramal, new { htmlAttributes  
= new { @class = "form-control" } })  
        @Html.ValidationMessageFor(model => model.Ramal, "",  
new { @class = "text-danger" })  
    </div>  
    </div>
```

```
<div class="form-group">  
    @Html.LabelFor(model => model.Notas)  
    @Html.EditorFor(model => model.Notas, new { htmlAttributes  
= new { @class = "form-control", @Rows = 10 } })  
    @Html.ValidationMessageFor(model => model.Notas, "", new  
{ @class = "text-danger" })  
    </div>
```

```
</div>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
<div class="col-md-2">  
    </div>  
</div>  
  
<div class="form-group">  
    <input type="submit" value="Gravar" class="btn btn-default" />  
    @Html.ActionLink("Voltar", "Index", null, new  
    { @class = "btn btn-default" })  
</div>  
}  
  
<script src="~/Scripts/jquery-1.10.2.min.js"></script>  
<script src="~/Scripts/jquery.validate.min.js"></script>  
<script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>  
<script src="~/Scripts/ValidacaoPtBr.js"></script>
```

10. Crie o arquivo **ValidacaoPtBr.js** dentro da pasta **Scripts**. Esse arquivo corrige alguns problemas de formato de dados brasileiro:

```
/Scripts/ValidacaoPtBr.js  
jQuery.extend(jQuery.validator.methods, {  
    date: function (value, element) {  
        return this.optional(element) || /^(\d\d?|/\d\d?|/\d\d\d?\d?)$/.  
test(value);  
    },  
    number: function (value, element) {  
        return this.optional(element) || /^-?(?:\d+|\d{1,3}(?:\.\.\d{3})+)  
(?:,\d+)?$/ .test(value);  
    }  
});
```

11. Algumas anotações são necessárias para definir a legenda de alguns campos. Inclua uma classe na pasta **Models** chamada **FuncionarioAtributos**:

```
using System.ComponentModel.DataAnnotations;

namespace Cap06_Lab01.Models
{
    [MetadataType(typeof(FuncionarioAtributos))]
    public partial class Funcionario { }

    public class FuncionarioAtributos
    {
        [Required]
        [Display(Name = "Telefone")]
        public string TelefoneResidencial { get; set; }

        [Display(Name = "País")]
        public string Pais { get; set; }

        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}",
                      ApplyFormatInEditMode = true)]
        [Display(Name = "Data de Nascimento")]
        public Nullable<System.DateTime> DataNascimento { get; set; }

        [Display(Name = "Data de Admissão")]
        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}",
                      ApplyFormatInEditMode = true)]
        public Nullable<System.DateTime> DataAdmissao { get; set; }

        [DataType(DataType.MultilineText)]
        public string Notas { get; set; }
    }
}
```

12. No Web.Config, defina formato de dados e idioma como **Portugues – Brasil**:

```
<system.web>
    <compilation debug="true" targetFramework="4.6.1" />
    <httpRuntime targetFramework="4.6.1" />

    <globalization culture="pt-br" uiCulture="pt-br"/>

</system.web>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

13. Inclua alguns estilos a mais no arquivo **Site.css** que está na pasta **Content**:

```
.tamanhoMaximo-300 {    max-width:600px; }

.espaco5 * {    padding-bottom:5px; }

.margemInferior20 {    margin-bottom:20px; }
```

14. Teste a inclusão. Repare nos campos **Telefone**, **Data de Nascimento** e **Data de Admissão** que foram alterados por meio de atributos colocados na classe:

The screenshot shows the Northwind application's 'Incluir Funcionário' (Add Employee) page. The form contains fields for Nome (Name), Sobrenome (Last Name), Endereço (Address), Cidade (City), Estado (State), CEP (ZIP Code), Cargo (Position), Tratamento (Title), Data de Nascimento (Birth Date), Data de Admissão (Hire Date), País (Country), Telefone (Phone), and Notas (Notes). Below the form is a list of employees with their names, cities, countries, and phone numbers:

Laura Callahan	Seattle	USA	(206) 555-1189
Annie Dodsworth	London	UK	(71) 555-4444
José Silva	São Paulo	Brasil	2123-2323

At the bottom left, there are 'Gravar' (Save) and 'Voltar' (Back) buttons, and a copyright notice: © 2016 - Northwind.

- Alteração de dados

15. Adicione, na classe **FuncionariosController**, o método de edição:

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Funcionario funcionario = db.Funcionarios.Find(id);

    if (funcionario == null)
    {
        return HttpNotFound();
    }

    ViewBag.ReportarA =
        new SelectList(db.Funcionarios,
                      "FuncionarioId", "Sobrenome", funcionario.ReportarA);

    return View(funcionario);
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "FuncionarioId,Sobrenome,Nome,
Cargo,Tratamento,DataNascimento,DataAdmissao,Endereco,Cidade,Estado,CEP,
Pais,TelefoneResidencial,Ramal,Notas,ReportarA")] Funcionario funcionario,
HttpPostedFileBase fotoUpload)
{
    if (ModelState.IsValid)
    {
        ObterFotoUpload(funcionario, fotoUpload);
        db.Entry(funcionario).State = EntityState.Modified;
        db.SaveChanges();
        AtualizarFoto(funcionario);
        return RedirectToAction("Index");
    }

    ViewBag.ReportarA =
        new SelectList(db.Funcionarios,
                      "FuncionarioId", "Sobrenome", funcionario.ReportarA);
    return View(funcionario);
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

16. Altere a tela de edição. Ela é praticamente igual à tela de cadastro:

```
@model Cap06_Lab01.Models.Funcioario

{@
    ViewBag.Title = "Edit";
}

<h2>Alterar</h2>

@using (Html.BeginForm("Edit", "Funcionarios", null, FormMethod.Post, new
{ enctype = "multipart/form-data" }))
{
    @Html.AntiForgeryToken()

    <h4>Funcionario</h4>
    <hr />
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })
    @Html.HiddenFor(model => model.FuncionarioId)

    <div class="row">
        <div class="col-md-5">
            <div class="row">
                <div class="form-group col-md-6">
                    @Html.LabelFor(model => model.Nome)
                    @Html.EditorFor(model => model.Nome, new {
                        htmlAttributes = new { @class = "form-control" } })
                    @Html.ValidationMessageFor(model => model.Nome, "", new { @class = "text-danger" })
                </div>
            </div>
        </div>
    </div>

    <div class="form-group col-md-6">
        @Html.LabelFor(model => model.Sobrenome)
        @Html.EditorFor(model => model.Sobrenome, new {
            htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Sobrenome,
        "", new { @class = "text-danger" })
    </div>
</div>
```

```
<div class="row">
    <div class="form-group col-md-8">
        @Html.LabelFor(model => model.Cargo)
        @Html.EditorFor(model => model.Cargo, new {
            htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Cargo,
        "", new { @class = "text-danger" })
    </div>
```

```
<div class="form-group col-md-4">
    @Html.LabelFor(model => model.Tratamento)
    @Html.EditorFor(model => model.Tratamento, new {
        htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Tratamento,
        "", new { @class = "text-danger" })
    </div>
    <div class="row">
```

```
<div class="form-group col-md-6">
    @Html.LabelFor(model => model.DataNascimento)
    @Html.EditorFor(model => model.DataNascimento, new {
        { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.
        DataNascimento, "", new { @class = "text-danger" })
    </div>
```

```
<div class="form-group col-md-6">
    @Html.LabelFor(model => model.DataAdmissao)
    @Html.EditorFor(model => model.DataAdmissao, new {
        htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.DataAdmissao,
        "", new { @class = "text-danger" })
    </div>
```

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

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
<div class="form-group">

    @if (!string.IsNullOrEmpty(Model.FotoPath))
    {
        
    }

    <input type="file" name="fotoUpload" id="fotoUpload" />
    <hr />
</div>

</div>

<div class="col-md-5">

    <div class="form-group">
        @Html.LabelFor(model => model.Endereco)
        @Html.EditorFor(model => model.Endereco, new {
            htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Endereco, "", new { @class = "text-danger" })
    </div>

    <div class="form-group">
        <div class="row">

            <div class="col-md-5">
                @Html.LabelFor(model => model.Cidade)
                @Html.EditorFor(model => model.Cidade, new {
                    htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Cidade, "", new { @class = "text-danger" })
            </div>
        </div>
    </div>

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

```
<div class="col-md-4">
    @Html.LabelFor(model => model.CEP)
    @Html.EditorFor(model => model.CEP, new {
        htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.CEP,
    "", new { @class = "text-danger" })
</div>
```

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

<div class="form-group">
</div>
```

```
<div class="form-group">
    @Html.LabelFor(model => model.Pais)
    @Html.EditorFor(model => model.Pais, new { htmlAttributes =
    = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.Pais, "", new
    { @class = "text-danger" })
</div>
```

```
<div class="row">
    <div class="form-group col-md-8">
        @Html.LabelFor(model => model.TelefoneResidencial)
        @Html.EditorFor(model => model.TelefoneResidencial,
        new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.
        TelefoneResidencial, "", new { @class = "text-danger" })
    </div>
```

```
<div class="form-group col-md-4">
    @Html.LabelFor(model => model.Ramal)
    @Html.EditorFor(model => model.Ramal, new {
        htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.Ramal,
    "", new { @class = "text-danger" })
</div>
```

```
</div>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
<div class="form-group">
    @Html.LabelFor(model => model.Notas)
    @Html.EditorFor(model => model.Notas, new { htmlAttributes
= new { @class = "form-control", @Rows = 10 } })
    @Html.ValidationMessageFor(model => model.Notas, "", new { @class = "text-danger" })
</div>

</div>

<div class="col-md-2">
</div>
</div>

<div class="form-group">
    <input type="submit" value="Alterar" class="btn btn-default" />
    @Html.ActionLink("Voltar", "Index", null, new { @class = "btn
btn-default" })
</div>

}

<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<script src="~/Scripts/jquery.validate.min.js"></script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
<script src="~/Scripts/ValidacaoPtBr.js"></script>
```

17. Teste a alteração de dados:

Northwind

Alterar

Funcionario:

Nome:	Sobrenome:	Endereço		
Janet	Leverling	722 Moss Bay Blvd.		
Cargo:	Tratamento:	Cidade:	Estado:	CEP:
Sales Representative	Ms.	Kirkland	WA	98033
Data de Nascimento:	Data de Admissão:	País:		
01/01/1960	01/01/2010	USA		
ReportarA:	Telefone:		Ranial:	
Fuller	(206) 555-3412		3366	
 <input type="button" value="Escolher arquivo"/> Nenhum arquivo selecionado				
Notas: Janet has a BS degree in chemistry from Boston College (1984). She has also completed a certificate program in food retailing management. Janet was hired as a sales associate in 1991 and promoted to sales representative in February 1992.				

Northwind

Funcionários

[Novo](#)

Nome	Cidade	País	Telefone	Ações
Nancy Davolio	Seattle	USA	(206) 555-9875	Alterar Exibir Excluir
Andrew Fuller	Tacoma	USA	(206) 555-9402	Alterar Exibir Excluir
Janet Leverling	Kirkland	USA	(206) 555-3412	Alterar Exibir Excluir
Margaret Peacock	Redmond	USA	(206) 555-8122	Alterar Exibir Excluir
Steven Buchanan	London	UK	(71) 555-4648	Alterar Exibir Excluir
Michael Suyama	London	UK	(71) 555-7773	Alterar Exibir Excluir
Robert King	London	UK	(71) 555-6598	Alterar Exibir Excluir
Laura Callahan	Seattle	USA	(206) 555-1189	Alterar Exibir Excluir
Anne Dodsworth	London	UK	(71) 555-4444	Alterar Exibir Excluir
José Silva	São Paulo	Brasil	2123-2323	Alterar Exibir Excluir

© 2016 - Northwind

Visual Studio 2015 - ASP.NET com C# Acesso a dados

- **Exibir detalhes**

18. No arquivo **FuncionariosController**, insira o método para exibir o perfil dos funcionários:

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Funcionario funcionario = db.Funcionarios.Find(id);
    if (funcionario == null)
    {
        return HttpNotFound();
    }

    return View(funcionario);
}
```

19. Pouca alteração é necessária no código gerado. Apenas insira a imagem:

```
@model Cap06_Lab01.Models.Funcionario

@{
    ViewBag.Title = "Details";
}



## Perfil



#### Funcionario



---



@Html.DisplayNameFor(model => model.Sobrenome)
:   @Html.DisplayFor(model => model.Sobrenome)


```

```
<dt>
    @Html.DisplayNameFor(model => model.Nome)
</dt>
```

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

```
<dt>
    @Html.DisplayNameFor(model => model.Cargo)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Cargo)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Tratamento)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Tratamento)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.DataNascimento)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.DataNascimento)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.DataAdmissao)
```

```
<dd>
    @Html.DisplayFor(model => model.DataAdmissao)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Endereco)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Endereco)
</dd>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
<dt>
    @Html.DisplayNameFor(model => model.Cidade)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Cidade)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Estado)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Estado)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.CEP)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.CEP)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Pais)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Pais)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.TelefoneResidencial)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.TelefoneResidencial)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Ramal)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Ramal)
</dd>
```

```
<dt>
```

```
</dt>

<dd>
    <div class="tamanhoMaximo-300">
        model.FotoPath)" />
    </div>
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Notas)
</dt>

<dd>
    @Html.DisplayFor(model => model.Notas)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.ReportarA)
</dt>

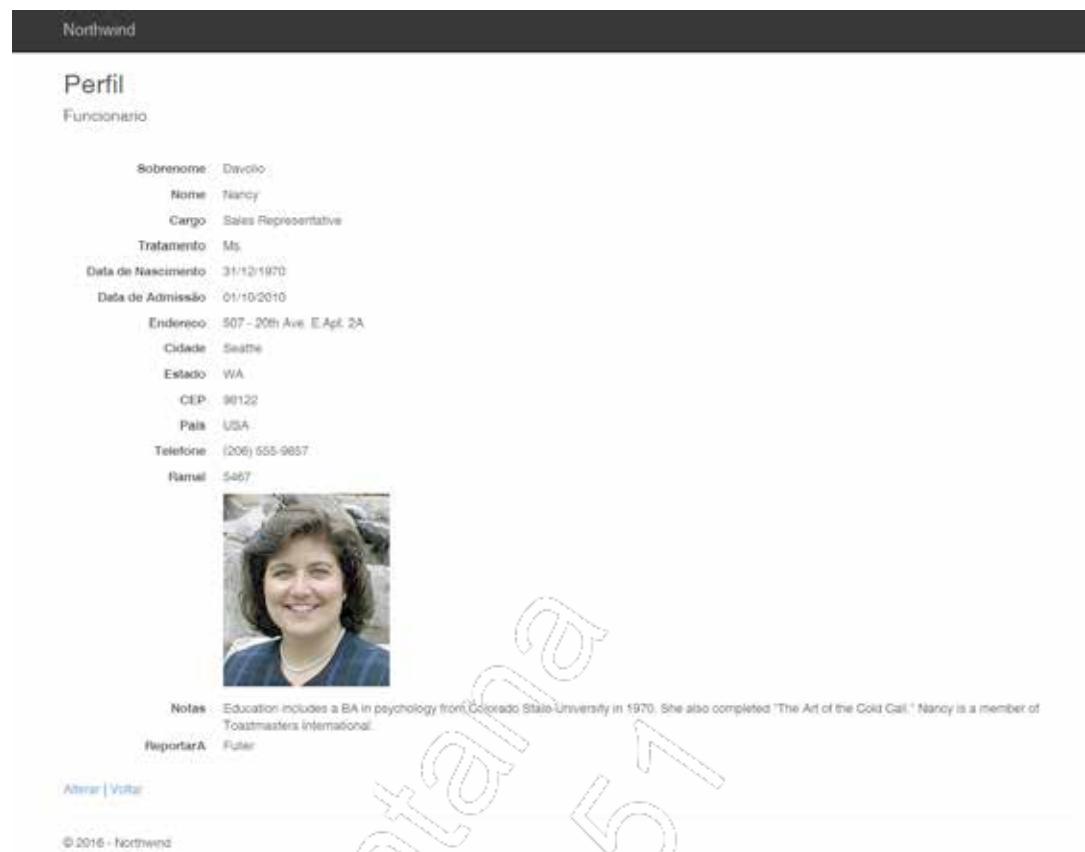
<dd>
    @Html.DisplayFor(model => model.Funcionario2.Sobrenome)
</dd>
```

```
</dl>
</div>
```

```
<p>
    @Html.ActionLink("Alterar", "Edit", new { id = Model.FuncionarioId }) |
    @Html.ActionLink("Voltar", "Index")
</p>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

20. Teste a visualização do perfil do funcionário (detalhes):



- **Exclusão de dados**

21. No arquivo **FuncionariosController**, insira os métodos para excluir funcionários:

```
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Funcionario funcionario = db.Funcionarios.Find(id);

    if (funcionario == null)
    {
        return HttpNotFound();
    }

    return View(funcionario);
}
```

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Funcionario funcionario = db.Funcionarios.Find(id);

    db.Funcionarios.Remove(funcionario);

    db.SaveChanges();

    return RedirectToAction("Index");
}
```

22. Altere a View de exclusão (delete):

```
@model Cap06_Lab01.Models.Funcionario

@{
    ViewBag.Title = "Delete";
}



## Excluir



### Confirma a exclusão deste registro?



@Html.DisplayNameFor(model => model.Sobrenome)


@Html.DisplayFor(model => model.Sobrenome)
  


@Html.DisplayNameFor(model => model.Nome)


@Html.DisplayFor(model => model.Nome)


```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
<dt>
    @Html.DisplayNameFor(model => model.Cargo)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Cargo)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Tratamento)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Tratamento)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.DataNascimento)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.DataNascimento)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.DataAdmissao)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.DataAdmissao)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Endereco)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Endereco)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Cidade)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Cidade)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Estado)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Estado)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.CEP)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.CEP)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Pais)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Pais)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.TelefoneResidencial)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.TelefoneResidencial)
</dd>
```

```
<dt>
    @Html.DisplayNameFor(model => model.Ramal)
</dt>
```

```
<dd>
    @Html.DisplayFor(model => model.Ramal)
</dd>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
<dt>
</dt>

<dd>
    <div class="tamanhoMaximo-300">
        model.FotoPath)" />
    </div>
</dd>

<dt>
    @Html.DisplayNameFor(model => model.Notas)
</dt>

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

<dt>
    @Html.DisplayNameFor(model => model.ReportarA)
</dt>

<dd>
    @Html.DisplayFor(model => model.Funcionario2.Sobrenome)
</dd>

</dl>

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

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

23. Teste a exclusão:

Northwind

Excluir

Confirma a exclusão deste registro?

Funcionario

Sobrenome	Silva
Nome	José
Cargo	Diretor
Tratamento	Mr
Data de Nascimento	01/10/1970
Data de Admissão	01/01/2001
Endereço	Rua ABC, 100
Cidade	São Paulo
Estado	SP
CEP	012323
País	Brasil
Telefone	2123-2323
Ramal	10

Notas: Teste
Reportar A: Davolio

[Excluir](#) | [Voltar](#)

Northwind

Funcionários

Name	City	Country	Phone	Action
Nancy Davolio	Seattle	USA	(206) 555-4887	Alterar Excluir Excluir
Andrew Fuller	Tacoma	USA	(206) 555-4412	Alterar Excluir Excluir
Jane English	Kirkland	USA	(206) 555-4412	Alterar Excluir Excluir
Margaret Peacock	Rainmond	USA	(206) 555-8122	Alterar Excluir Excluir
Steven Buchanan	London	UK	(71) 555-4548	Alterar Excluir Excluir
Michael Suyama	London	UK	(71) 888-7772	Alterar Excluir Excluir
Robert King	London	UK	(71) 555-5595	Alterar Excluir Excluir
Laura Callahan	Seattle	USA	(206) 555-1139	Alterar Excluir Excluir
Anna Adams	London	UK	(71) 888-6444	Alterar Excluir Excluir

© 2010 - Northwind

7

Data Services: WCF

- ✓ Por que usar serviços;
- ✓ Tecnologias envolvidas;
- ✓ Conceitos iniciais sobre WCF.

7.1. Introdução

Os dados de uma aplicação podem ser expostos na forma de serviços para serem utilizados por qualquer aplicativo que conheça o protocolo e o formato de dados usado, independente da plataforma, linguagem ou ambiente. A arquitetura que define serviços como principal meio de comunicação entre os componentes de um aplicativo é chamada de **SOA (Service-Oriented Application Architecture)**.

7.2. Por que usar serviços

São muitas as vantagens de criar serviços relativamente independentes que retornem informação. Podemos destacar algumas:

- **Independência de plataforma:** Um componente de uma aplicação pode trocar informações com o componente de outra aplicação, independente do sistema operacional;
- **Redução do acoplamento:** Uma aplicação baseada em serviços produz componentes relativamente independentes. Isso torna o programa mais fácil de manter e de assimilar modificações. A dependência excessiva entre métodos ou classes exige que qualquer alteração, antes que seja feita, passe por um processo de verificação para detectar possíveis efeitos colaterais em outros pontos do sistema;
- **Uso de protocolos e formatos conhecidos:** A possibilidade de integração entre sistemas está diretamente ligada ao fato de os sistemas envolvidos compartilharem os mesmos conhecimentos à respeito da troca de informações. Usando formatos conhecidos como XML, JSON, CSV e protocolos como HTTP, a chance de sucesso na integração de sistemas é muito maior do que quando envolvidos usam formatos diversos;

- **Tendência em terceirizar processos:** Quanto mais complexos os sistemas são, mais complicado é para uma empresa manter o controle de todos os recursos envolvidos na informatização de seus processos. Cada vez mais se mostra vantajoso delegar para setores da empresa ou para terceiros parte dos processos, porque pessoas ou empresas especializadas em um assunto podem realizar tarefas mais rapidamente e com mais segurança do que se estas fossem realizadas dentro da empresa. Um exemplo muito comum é a emissão de nota fiscal, que, até pouco anos atrás, era feita inteiramente por processo manual. Com a implantação da nota fiscal eletrônica, esse processo se tornou mais complexo do que simplesmente preencher um formulário. É necessário criar um arquivo XML, enviar para um serviço, tratar os erros, obter uma chave de validação para imprimir uma cópia válida, armazenar eletronicamente os arquivos recebidos, entre outras tarefas que nem sempre um funcionário não especializado em tecnologia consegue realizar.

7.3. Tecnologias envolvidas

Vejamos, a seguir, as tecnologias envolvidas:

- **Web Services**

A primeira tecnologia formalmente definida para expor informações por meio de serviços que não fossem limitados à comunicação pela rede interna de uma empresa se chamou **Web Services**. São páginas hospedadas em um servidor Web (como o IIS), que recebe solicitações usando o protocolo HTTP e retorna dados no formato XML. Esses dados retornados podem ser utilizados pela aplicação que os solicitou, bastando apenas que esta saiba interpretar o formato dos dados retornados.

Os Web Services foram (e ainda são) uma parte muito importante de um grande número de aplicações distribuídas (os serviços da Nota Fiscal Eletrônica, no Brasil, usam Web Services).

Com o tempo, porém, algumas limitações dos Web Services ficaram mais aparentes, dentre as quais podemos destacar as seguintes:

- **Fortemente ligados à Internet:** Como o próprio nome diz, os Web Services necessitam de um servidor Web e de um protocolo HTTP para existir. Nem sempre o uso da Internet como meio de transporte de informação é o mais eficiente. Em uma rede interna, por exemplo, a comunicação binária e não serializada (transformada em caracteres) é mais rápida e eficiente;
- **XML:** Apesar de ser possível retornar informações em outros formatos diferentes de XML, os Web Services, em sua natureza, são formatados para trabalhar com XML. Nem sempre o XML é a melhor solução. Por exemplo, para o uso de AJAX, o melhor formato é JSON, que é interpretado naturalmente pela linguagem JavaScript. É possível retornar dados em formato JSON, mas exige o uso de atributos e a alteração do modo padrão;
- **Protocolos:** Como os Web Services são intrinsecamente ligados à Internet, o único protocolo disponível é HTTP ou HTTPS. O protocolo SOAP, descrito na documentação, é, na verdade, um formato de dados usando POST/GET, (ou seja, HTTP) e algumas regras de XML. Não é possível usar um protocolo específico, personalizado, para uma determinada situação;
- **HOST:** Para hospedar um Web Service, é necessário um Servidor Web, com o IIS. Uma aplicação ou um serviço pode até hospedar um Web Service, mas não é uma tarefa simples e envolve a mudança do modo padrão de implementar.

Pensando em resolver essas limitações, a Microsoft desenvolveu outra tecnologia utilizada para criar serviços que retornem dados para aplicações distribuídas. Essa tecnologia se chama **WCF (Windows Communication Foundations)**.

- **WCF**

Segundo a própria definição da Microsoft, "**WCF (Windows Communication Foundation)** é o modelo unificado da Microsoft para construir aplicações orientadas a serviços e habilita os desenvolvedores a criar aplicações seguras, confiáveis e transacionais que se integram em diferentes plataformas e interagem com os investimentos existentes".

Resumindo e simplificando, WCF é um conjunto de tecnologias para criar serviços e interligar sistemas.

O WCF é um robusto substituto para os Web Services e foi implementado em diversos produtos da Microsoft. Porém, assim como os Web Services, o WCF apresenta algumas limitações. A principal é a curva de aprendizado. Algumas tarefas simples como disponibilizar dados em um site para ser consumido por páginas HTML exigem um certo trabalho de configuração e implementação.

Além disso, o WCF é um produto totalmente voltado para as tecnologias Microsoft e existe uma grande demanda por serviços open source e de simples implementação. Pensando nisso, foi desenvolvida outra tecnologia para simplificar a criação de aplicações baseada em serviços: **Web API**.

- **Web API**

Web API é um framework que torna fácil a criação de serviços que utilizam a Web e seus protocolos, como o HTTP. As aplicações que utilizam exclusivamente os protocolos Web, sem nenhum componente ou formato de dados de terceiros, são chamadas de aplicações **RESTful**.

REST (Representational State Transfer) é um estilo de arquitetura que utiliza os comandos da Web (GET/POST/PUT/DELETE) para enviar e receber comandos e dados entre aplicativos. Isso garante que uma aplicação não dependa de uma arquitetura específica ou de algum componente proprietário. O framework Web API da Microsoft implementa as funcionalidades necessárias para criar aplicações RESTful.

Resumidamente, existem três tecnologias desenvolvidas pela Microsoft para criar aplicações distribuídas:

- **Web Services**: Serviços que são executados em um servidor Web e que retornam dados XML;
- **WCF**: Windows Communication Foundation são serviços que rodam em um aplicativo host e que retornam dados de qualquer natureza de maneira nativa;
- **Web API**: Framework para criar aplicações RESTful, que utilizam exclusivamente a Web e os protocolos HTTP como meio de comunicação, sem componentes, formatos ou protocolos de terceiros.

7.4. Conceitos iniciais sobre WCF

O WCF trabalha enviando e recebendo mensagens usando um meio de transporte ou protocolo. A mensagem sempre é solicitada por algum programa chamado **cliente** e respondida por outro programa chamado **service host**. O host cria pontos de entrada (entry points) para que clientes entrem em contato. Veja, a seguir, alguns termos importantes no universo de aplicações distribuídas:

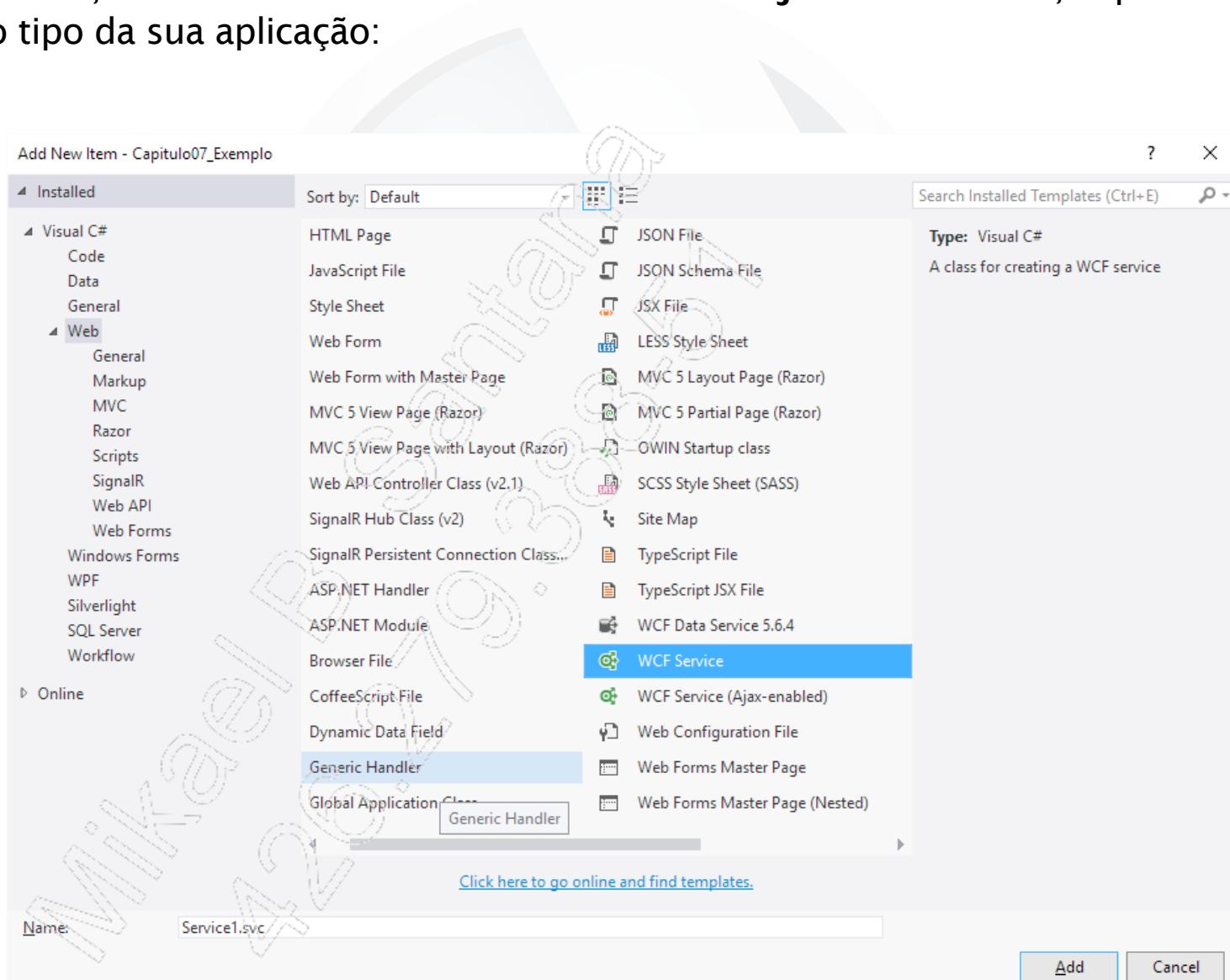
- **Message**: Mensagem é o principal componente do WCF. É a informação que é enviada de um cliente (quem solicita) para um serviço (quem responde);
- **Protocol**: Protocolo é o meio e a forma que uma mensagem é enviada;
- **Contract**: Contrato, no mundo das aplicações distribuídas, é o conjunto de regras que os envolvidos no processo de comunicação devem obedecer. Quem envia uma mensagem deve saber como chamar o serviço, qual protocolo usar e qual o formato da mensagem;
- **Client**: Cliente é o programa que solicita uma informação de um Service (serviço);
- **Service**: Serviço é a aplicação que responde a uma solicitação de um cliente;
- **Data contract**: Contrato de dados é o modelo de dados que será enviado e recebido pelos envolvidos no processo de comunicação;

- **Host:** Hospedeiro é o sistema no qual está instalado o serviço que responderá às solicitações de um cliente;
- **SOAP:** Simple Object Application Protocol é um formato de mensagem baseado em XML usado para transmitir informações com recursos para incluir dados adicionais (metadados) e tratamento de erro;
- **Channel:** Canal é o meio pelo qual as mensagens são enviadas e recebidas;
- **Endpoint:** Ponto final é o endereço pelo qual os serviço é chamado. Um serviço pode ter diferentes endpoints, cada qual com propriedades e retornos diferentes;
- **Binding:** Vínculo é a definição de como um endpoint se comunica;
- **Behaviors:** Comportamentos são a definição de características de um serviço: o tipo de protocolo que aceita, a autenticação necessária para acesso ao serviço, como são tratados dados processados simultaneamente (concorrência), entre outros;
- **HTTP:** HyperText Transfer Protocol é um protocolo de comunicação para transmissão de arquivos de hipertexto (arquivo com vínculos para outros arquivos);
- **XML:** Extensible Markup Language é o formato de dados em que as informações são marcadas (markup) para definição das informações de uma mensagem;
- **Serialization:** Serialização é o processo de transformar um objeto na memória em uma sequência de valores (caracteres ou bytes) para que possam ser transmitidos e posteriormente reconstruídos. A serialização é a base de todo sistema baseado em mensagens.

7.4.1. Criando um serviço WCF

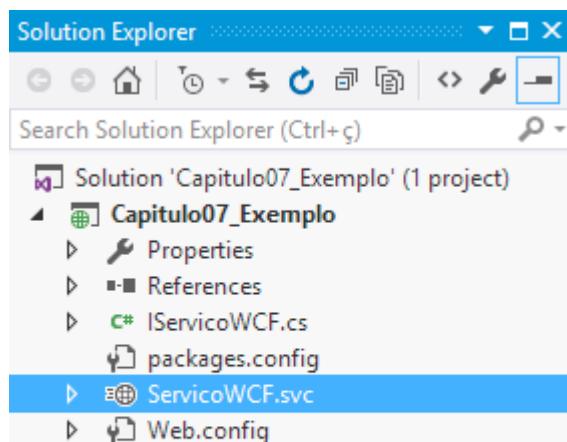
Um serviço WCF deve ser hospedado por um aplicativo (host) que seja capaz de receber solicitações. O IIS, por sua natureza, é um host para páginas Web, aplicações ASP.NET e pode ser usado para hospedar um serviço WCF. Mas é importante saber que não é obrigatório usar o IIS para criar um serviço. Uma aplicação Console, Windows, Windows Service ou Class Library pode ser usada como host. Neste exemplo, será usada como host uma aplicação ASP.NET.

Em uma aplicação Web ou Web Project, pode ser adicionado um item do tipo **WCF Service**, escolhendo **Add New Item** do menu **Project** ou **Web Site**, dependendo do tipo da sua aplicação:



Os seguintes arquivos são adicionados ao projeto (assumindo que o nome do serviço seja **ServicoWCF**):

- ServicoWCF.svc



É um arquivo de texto contendo uma diretiva de compilação `@ServiceHost`. Esse arquivo associa o arquivo com a extensão `.svc` a uma classe que implementa o serviço:

```
<%@ ServiceHost Language="C#" Debug="true"
   Service="Capitulo07_Exemplo.ServicoWCF"
   CodeBehind="ServicoWCF.svc.cs"
%>
```

- ServicoWCF.svc.cs

Código-fonte da classe que implementa o serviço. Repare que essa classe implementa uma interface: `IServicoWCF`. Um método simples sem parâmetros é definido apenas para servir de ponto de partida. Esse método está definido na interface. Os serviços WCF devem, geralmente, implementar uma interface. É na interface que estão definidos alguns requerimentos e particularidades de um serviço:

```
public class ServicoWCF : IServicoWCF
{
    public void DoWork()
    {
    }
}
```

- **IServicoWCF.cs**

É a interface onde estão definidas as operações que o serviço vai realizar. Neste caso, apenas um método **DoWork** (faça algo). É importante lembrar que a classe **ServicoWCF** implementa este método:

```
using System.ServiceModel;

[ServiceContract]
public interface IServicoWCF
{
    [OperationContract]
    void DoWork();
}
```

Dois atributos foram aplicados nessa interface: **ServiceContract** e **OperationContract**. A seguir, confira alguns detalhes destes atributos:

- **ServiceContract**: Este atributo é uma instância da classe **ServiceContractAttribute** presente no namespace **System.ServiceModel** e serve para indicar que a interface é um contrato de serviço que será disponibilizado em uma aplicação WCF;
- **OperationContract**: É uma instância da classe **OperationContractAttribute**, também no namespace **System.ServiceModel**, e indica uma operação que faz parte de um contrato.

De modo geral, essas são as únicas configurações necessárias para um serviço WCF estar disponível, quando hospedado em um servidor Web.

7.4.1.1.Web.Config

Alguns itens foram adicionados no Web.Config, conforme se pode observar:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
    For more information on how to configure your ASP.NET application, please
visit
http://go.microsoft.com/fwlink/?LinkId=169433
-->
<configuration>
    <system.web>
        <compilation debug="true" targetFramework="4.5" />
        <httpRuntime targetFramework="4.5" />
    </system.web>

    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior name="">
                    <serviceMetadata
                        httpGetEnabled="true"
                        httpsGetEnabled="true" />
                    <serviceDebug
                        includeExceptionDetailInFaults="false" />
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <serviceHostingEnvironment
            aspNetCompatibilityEnabled="true"
            multipleSiteBindingsEnabled="true" />
    </system.serviceModel>
</configuration>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

A seção **System.ServiceModel** contém a tag **Behaviors**, que é uma coleção de definições do tipo **ServiceBehavior**. **Behaviors** definem como os pontos de entrada se comportam. O código criado pelo Visual Studio define um comportamento em que são aceitas requisições GET usando ou não uma conexão segura:

```
<serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
```

Neste exemplo, será usada uma interface diferente para que retorne um valor. A implementação, com o método **DoWork**, não produz nenhum resultado visível:

```
[ServiceContract]
public interface IServicowCF
{
    [OperationContract]
    string Mensagem();

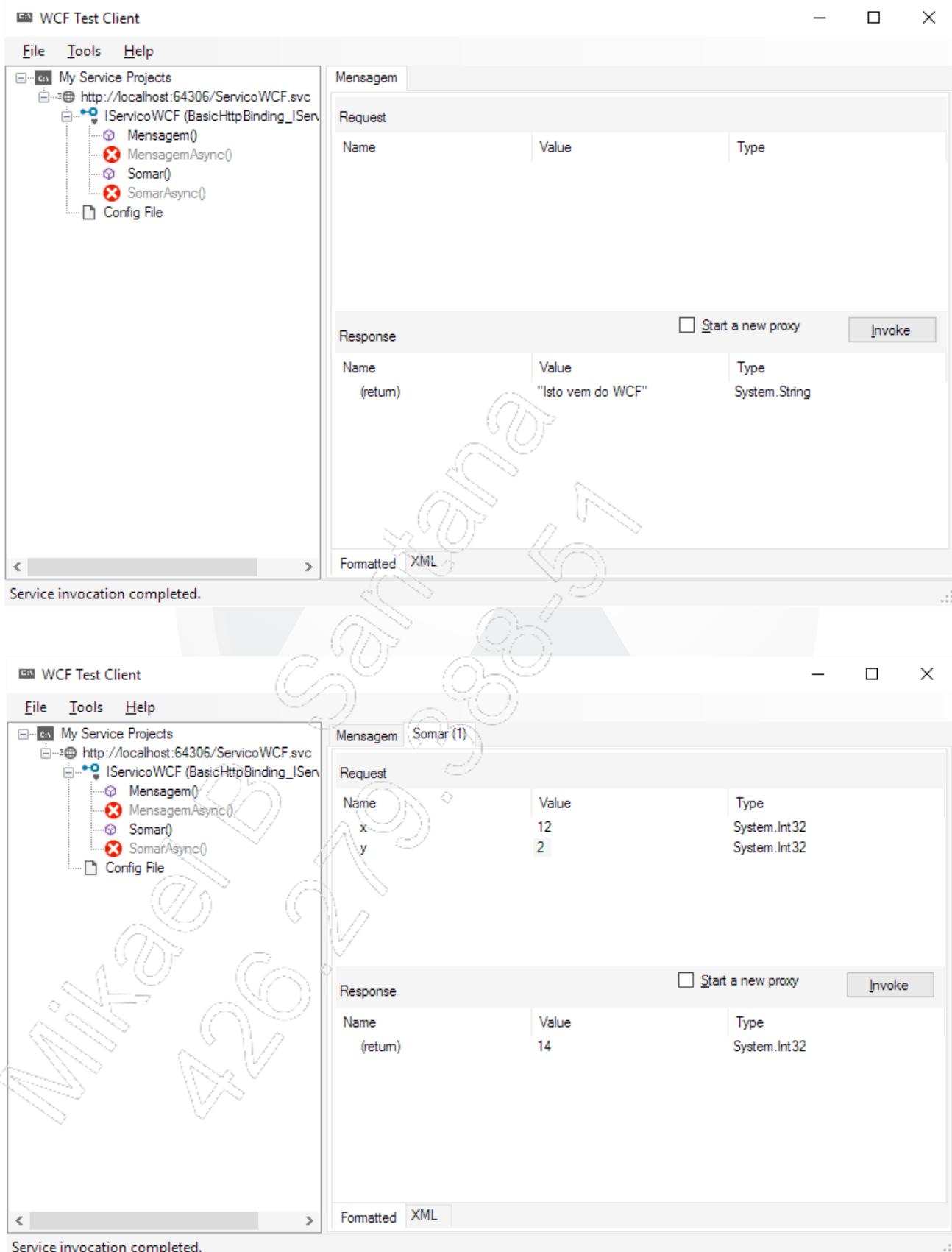
    [OperationContract]
    int Somar(int x, int y);
}
```

No arquivo da classe, é necessário implementar os métodos da nova interface:

```
public class ServicowCF : IServicowCF
{
    public string Mensagem()
    {
        return "Isto vem do WCF";
    }

    public int Somar(int x, int y)
    {
        return x + y;
    }
}
```

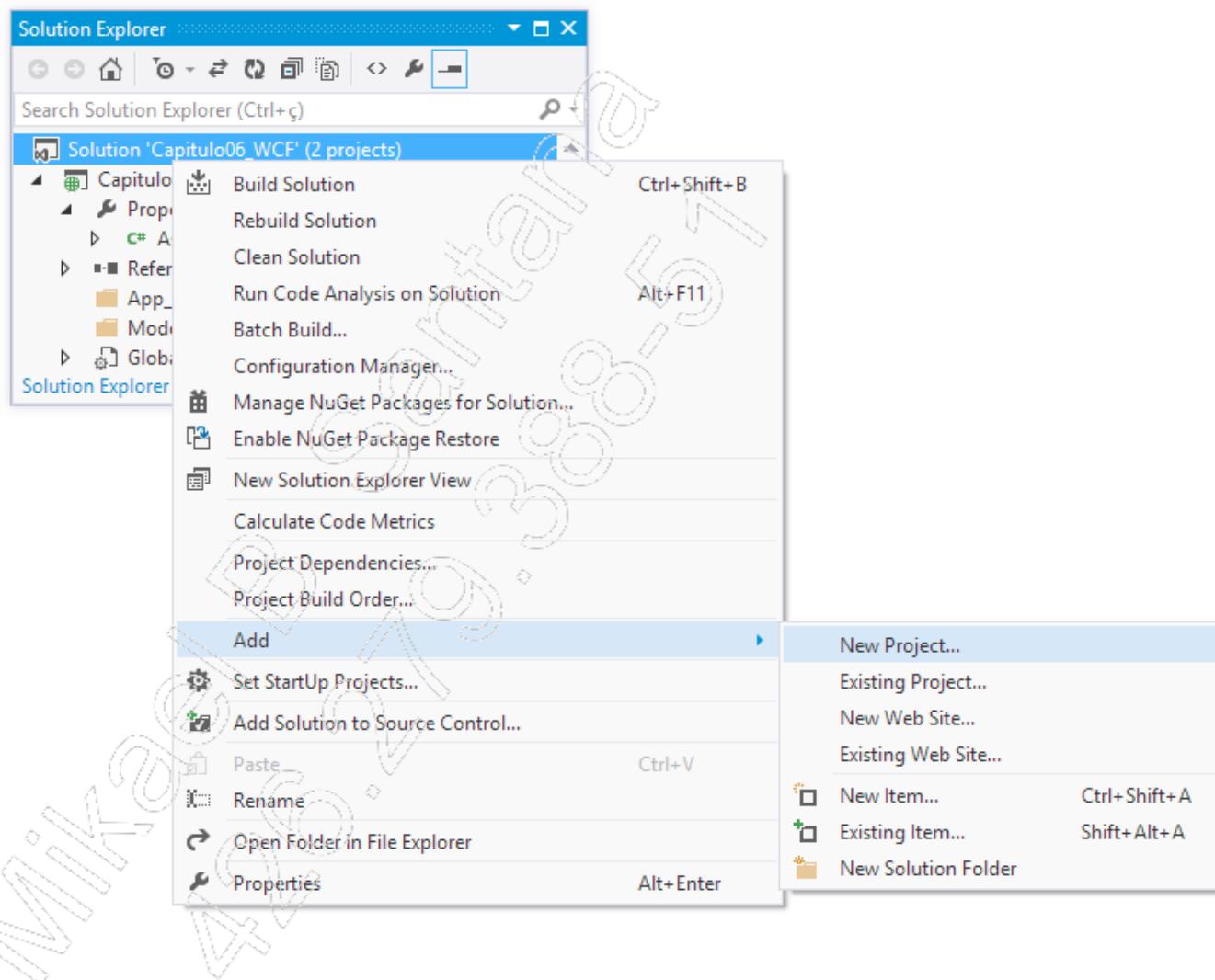
Depois de compilado, é possível testar o serviço ao executar dentro do Visual Studio:



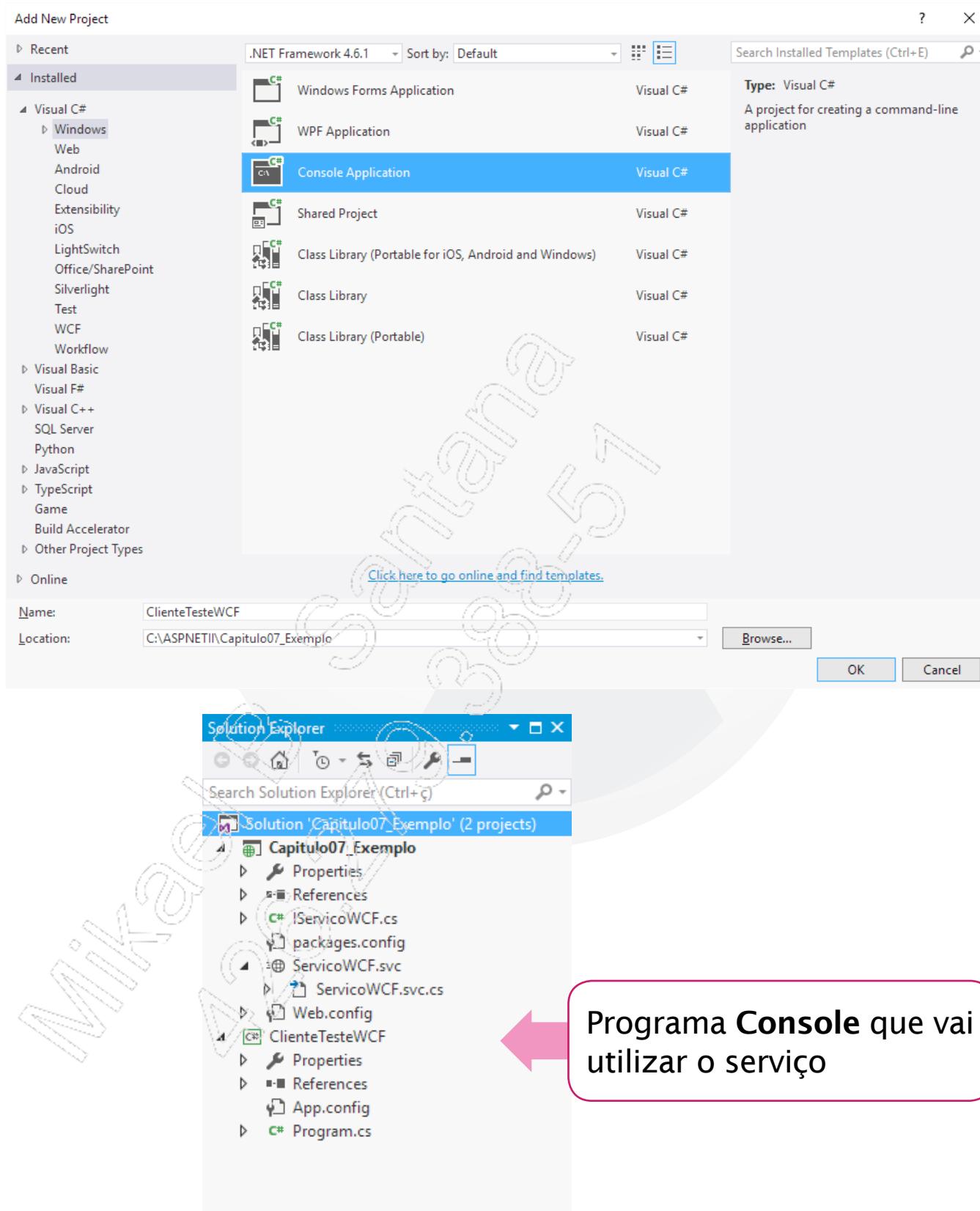
7.4.2. Criando um cliente

O próximo passo é criar um aplicativo cliente que consuma esse serviço. Isso pode ser feito de três maneiras: manualmente, por meio de uma ferramenta que gera uma classe chamada **svutil.exe**, ou adicionando uma referência ao serviço em um aplicativo .NET.

Usando o Visual Studio, a maneira mais fácil de criar um aplicativo para testar o serviço WCF criado é adicionar um projeto à solução existente. Para isso, basta abrir o menu de contexto e escolher **Add / New Project**:

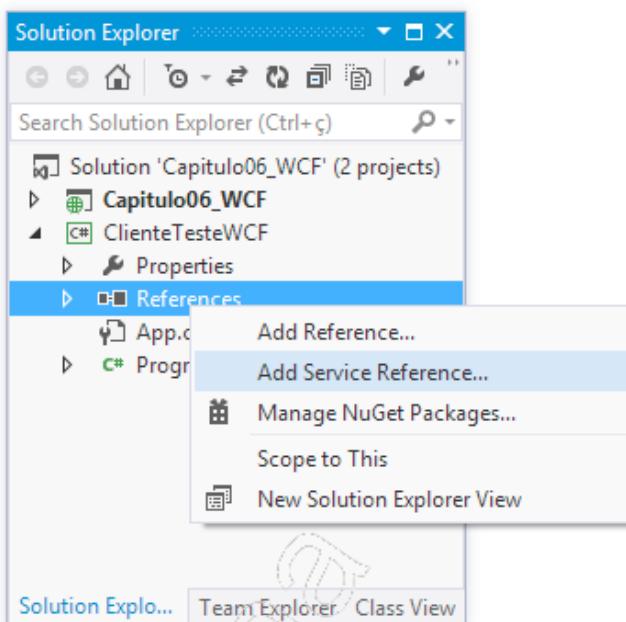


Qualquer projeto pode servir de cliente para um serviço WCF. Neste exemplo, será usado um projeto do tipo **Console**. Esse tipo de projeto é ótimo para testar tecnologias porque é o mais simples de todos e não tem nenhuma referência além das classes básicas do .NET Framework:

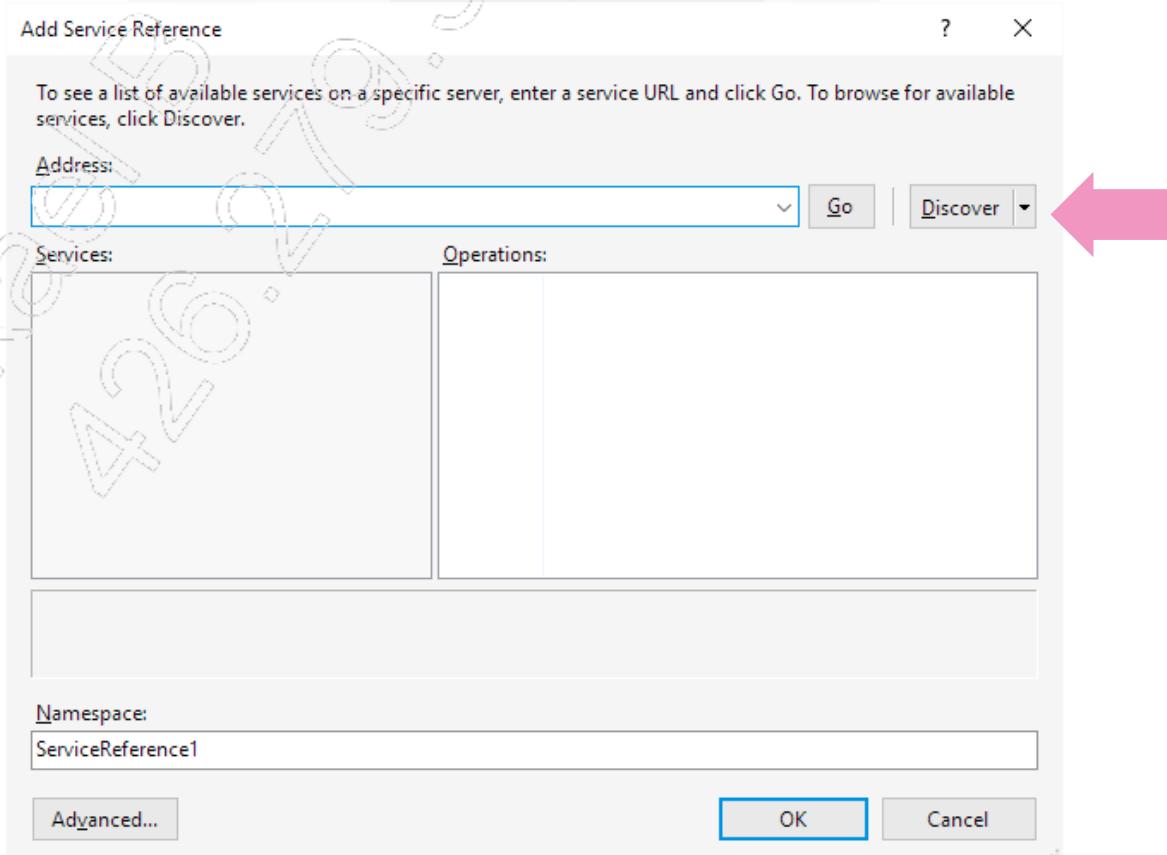


Visual Studio 2015 - ASP.NET com C# Acesso a dados

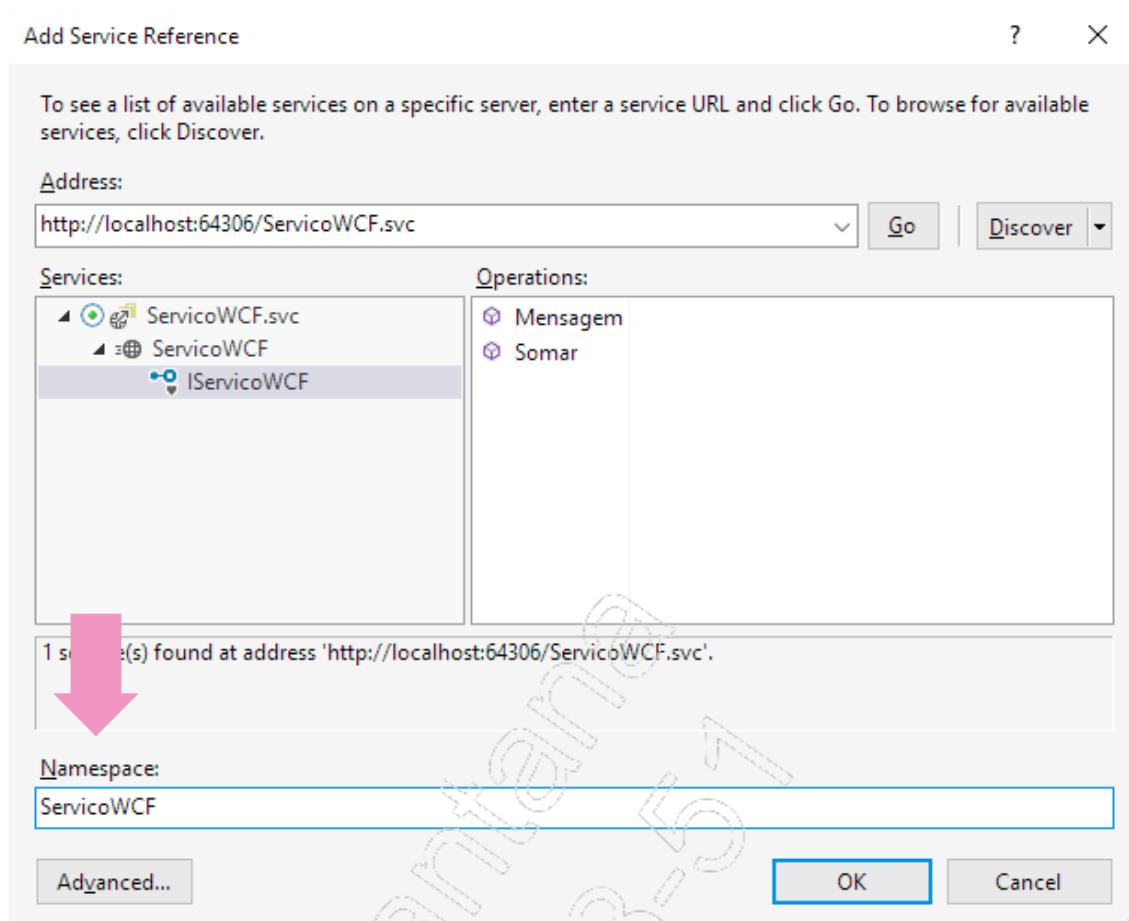
Abrindo o menu de contexto do programa **Console** adicionado na solução, no item **References**, a opção **Add Service Reference** possibilita criar a classe que vai se comunicar com o serviço. Esse tipo de classe é chamada de **Proxy**:



A janela a seguir serve para informar o endpoint do serviço, que é a URL da página. Como o projeto de serviço está na mesma solução, basta clicar no botão **Discover** (descobrir) para o Visual Studio encontrar o endereço correto:



Nesta janela, deve-se informar o namespace onde o **Proxy** será criado:



Algumas referências são adicionadas ao projeto:

- **System.Data.DataSetExtensions;**
- **System.Runtime.Serialization;**
- **System.ServiceModel.**

Esses namespaces contêm classes para serializar e desserializar objetos e definir contratos, dados e operações.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

No arquivo **App.Config**, algumas configurações são definidas:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime
            version="v4.0"
            sku=".NETFramework,Version=v4.5" />
    </startup>

    <system.serviceModel>
        <bindings>
            <basicHttpBinding>
                <binding name="BasicHttpBinding_IServicoWCF" />
            </basicHttpBinding>
        </bindings>
        <client>
            <endpoint
                address="http://localhost:59642/ServicoWCF.svc"
                binding="basicHttpBinding"
                bindingConfiguration="BasicHttpBinding_IServicoWCF"
                contract="servicoWcfRef.IServicoWCF"
                name="BasicHttpBinding_IServicoWCF" />
        </client>
    </system.serviceModel>
</configuration>
```

A opção **endpoint** define a URL onde será chamado o serviço (**address**), o tipo de vínculo (**binding** e **bindingConfiguration**), a classe de contrato (**contract**) e o nome do endpoint (**name**). Como este é um arquivo de texto, pode ser modificado, alterando essas informações (principalmente o **address**) para refletir o servidor em produção.

A classe **Proxy** que vai se comunicar com o serviço está oculta na janela **Solution**. Isso porque ela é gerada dinamicamente e não deve ser alterada pelo programador. Implementações personalizadas podem (e devem) ser feitas criando outra classe com o mesmo nome, porque ela é marcada como **partial**, exatamente para esse propósito. Para exibir a classe, é necessário exibir todos os arquivos da pasta **servicoWcfRef** (esse é o nome da referência informada na caixa de diálogo anterior).

A classe **Proxy** contém, para cada operação disponibilizada pelo serviço, dois métodos: um com o mesmo nome e um com o final **Async** adicionado. Esse método extra é para realizar chamadas assíncronas, ou seja, chamadas não sincronizadas, em que não é esperada a resposta do servidor, deixando a captura da resposta para uma função chamada **callback**.

A listagem adiante apresenta a classe gerada. Atributos, construtores, sobrecargas e implementações foram removidos para deixar a leitura mais fácil e focar nos métodos criados:

```
public partial class ServicoWCFClient
{
    public ServicoWCFClient() {}

    public string Mensagem() {}

    public Task<string> MensagemAsync() {}

    public int Somar(int x, int y) {}

    public Task<int> SomarAsync(int x, int y) {}

}
```

A classe no servidor se chama **ServicoWCF** e, neste aplicativo, é chamada, convenientemente, de **ServicoWCFClient**. Internamente, os métodos da classe **ServicoWCFClient** estabelecem a conexão com o servidor, chamam as operações necessárias, recebem o resultado serializado, o desserializam e retornam este valor. Do ponto de vista do programador, é como se estivesse chamando um método de uma classe local.

Essa abstração do serviço é um ponto muito importante em aplicações SOA (Service-Oriented Application Architecture), porque os serviços podem mudar não só a implementação interna, mas as restrições de segurança, os protocolos usados e o formato das mensagens. Programando dessa forma, não é necessário mudar o código do aplicativo solicitante quando o serviço sofre alguma alteração.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Agora que uma referência foi criada, basta chamar a classe **Proxy** de dentro do aplicativo **Console**:

```
class Program
{
    static void Main(string[] args)
    {
        //Cria uma instância do Proxy
        var wcf = new servicoWcfRef.ServicowCFClient();

        //Chama o método mensagem e guarda o resultado
        string mensagem = wcf.Mensagem();

        //Chama o método somar e guarda o resultado
        int total = wcf.Somar(3, 6);

        //fecha a comunicação com o serviço
        wcf.Close();

        //Exibe os dados
        Console.WriteLine("Mensagem: " + mensagem);
        Console.WriteLine("Soma de 3 e 6: " + total);

        Console.ReadLine();
    }
}
```

Veja o resultado:



7.4.3. Data Contract

No exemplo anterior, apenas tipos primitivos (**int**, **string**, **bool**, **DateTime**, etc.) foram manipulados e transmitidos do servidor para o cliente e vice-versa. Quando os dados a serem retornados são tipos primitivos, nenhuma configuração é necessária, além das que foram apresentadas. Porém, quando os dados a serem retornados são instâncias de classes, são necessárias algumas configurações.

Consideremos a seguinte classe a ser retornada por um método no serviço WCF:

```
public class Livro
{
    public int LivroId { get; set; }

    public string Titulo { get; set; }

    public string Autor { get; set; }

    public string Editora { get; set; }

    public int NumeroPaginas { get; set; }
}
```

Para que essa classe seja manipulada pelo serviço, é necessário adicionar alguns atributos. Esse processo é conhecido como definir um contrato de dados (**Data Contact**). Dois atributos são necessários: **DataContract**, que deve ser aplicado à classe, e **DataMember**, que deve ser aplicado nas propriedades que queremos disponibilizar:

```
[DataContract]
public class Livro
{
    [DataMember]
    public int LivroId { get; set; }

    [DataMember]
    public string Titulo { get; set; }

    [DataMember]
    public string Autor { get; set; }

    [DataMember]
    public string Editora { get; set; }

    public int NumeroPaginas { get; set; }
}
```

Repare que a propriedade **NumeroPaginas** não foi marcada com o atributo **DataMember**, portanto, essa propriedade não vai ser considerada pelo mecanismo do WCF.

Agora, é necessário alterar a interface para retornar um **Livro** por meio de uma operação. Nesse caso, uma simples consulta pela propriedade **LivroId**:

- Arquivo da interface, **IServicoWCF.cs**, no projeto do serviço

```
[ServiceContract]
public interface IServicoWCF
{
    [OperationContract]
    string Mensagem();

    [OperationContract]
    int Somar(int x, int y);

    [OperationContract]
    Livro ObterLivro(int livroId);
}
```

Qualquer mudança na interface deve ser acompanhada pelas classes que a implementam. Essa é uma grande vantagem de usar interfaces para definir as operações de um sistema.

Se duas ou mais classes implementam uma interface e ela muda, o compilador do Visual Studio não consegue compilar, gerando um erro.

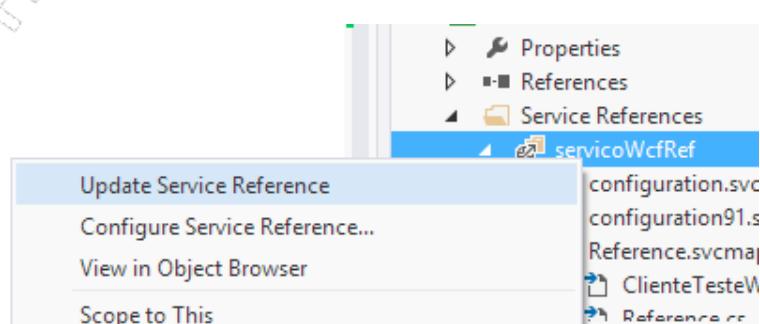
A classe **ServicoWCF** deve implementar o método **ObterLivro**. Depois de compilado, o **Proxy** deve ser atualizado, pois existe um método a mais e um tipo (**Livro**) que não existiam antes. Como o projeto de **serviço** e o projeto **Client** estão na mesma solução, a atualização do **Proxy** acontece automaticamente.

A listagem a seguir mostra uma implementação simples do método **ObterLivro**, apenas para teste, retornando uma instância da classe **Livro**, se for passado o código 1, e retornando **null** para todos os outros.

```
public class ServicoWCF : IServicoWCF
{
    public string Mensagem()...
    public int Somar(int x, int y)....
```

```
public Livro ObterLivro(int livroId)
{
    //Método apenas de teste
    //Retorna apenas o livro 1
    if (livroId == 1)
    {
        var livro = new Livro()
        {
            LivroId = 1,
            Titulo = "Uma breve história do Tempo",
            Autor = "Stephen Hawking",
            Editora = "Rocco",
            NumeroPaginas = 300
        };
        return livro;
    }
    else
    {
        return null;
    }
}
```

Depois de compilado, o novo serviço está no ar. Se não estivesse na mesma solução, o **Proxy** deveria ser atualizado por meio do menu de contexto da referência do serviço no Solution Explorer, na opção **Update Service Reference**:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

O código de teste do programa **Console** pode agora chamar o serviço e testar o retorno de um livro existente e o retorno **null** de um livro inexistente:

```
class Program
{
    static void Main(string[] args)
    {
        //Cria uma instância do Proxy
        var wcf = new servicoWcfRef.ServicoWCFClient();

        //Chama o método mensagem e guarda o resultado
        string mensagem = wcf.Mensagem();

        //Chama o método somar e guarda o resultado
        int total = wcf.Somar(3, 6);

        //Obtém um livro
        var livro = wcf.ObterLivro(1);

        //Falha ao obter um livro
        var livroInexistente = wcf.ObterLivro(2);

        //Fecha a comunicação com o serviço
        wcf.Close();

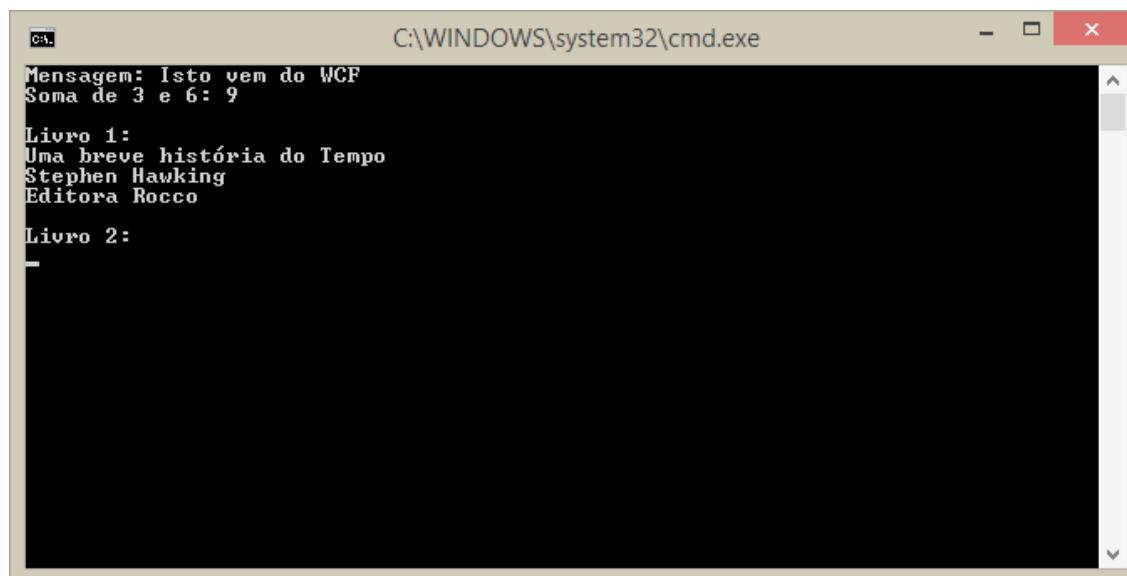
        //Exibe os dados
        Console.WriteLine("Mensagem: " + mensagem);
        Console.WriteLine("Soma de 3 e 6: " + total);

        Console.WriteLine(
            "\nLivro 1: \n{0}\n{1}\nEditora {2}\n",
            livro.Titulo, livro.Autor, livro.Editora);

        Console.WriteLine("Livro 2:", livroInexistente);

        Console.ReadLine();
    }
}
```

Veja o resultado:



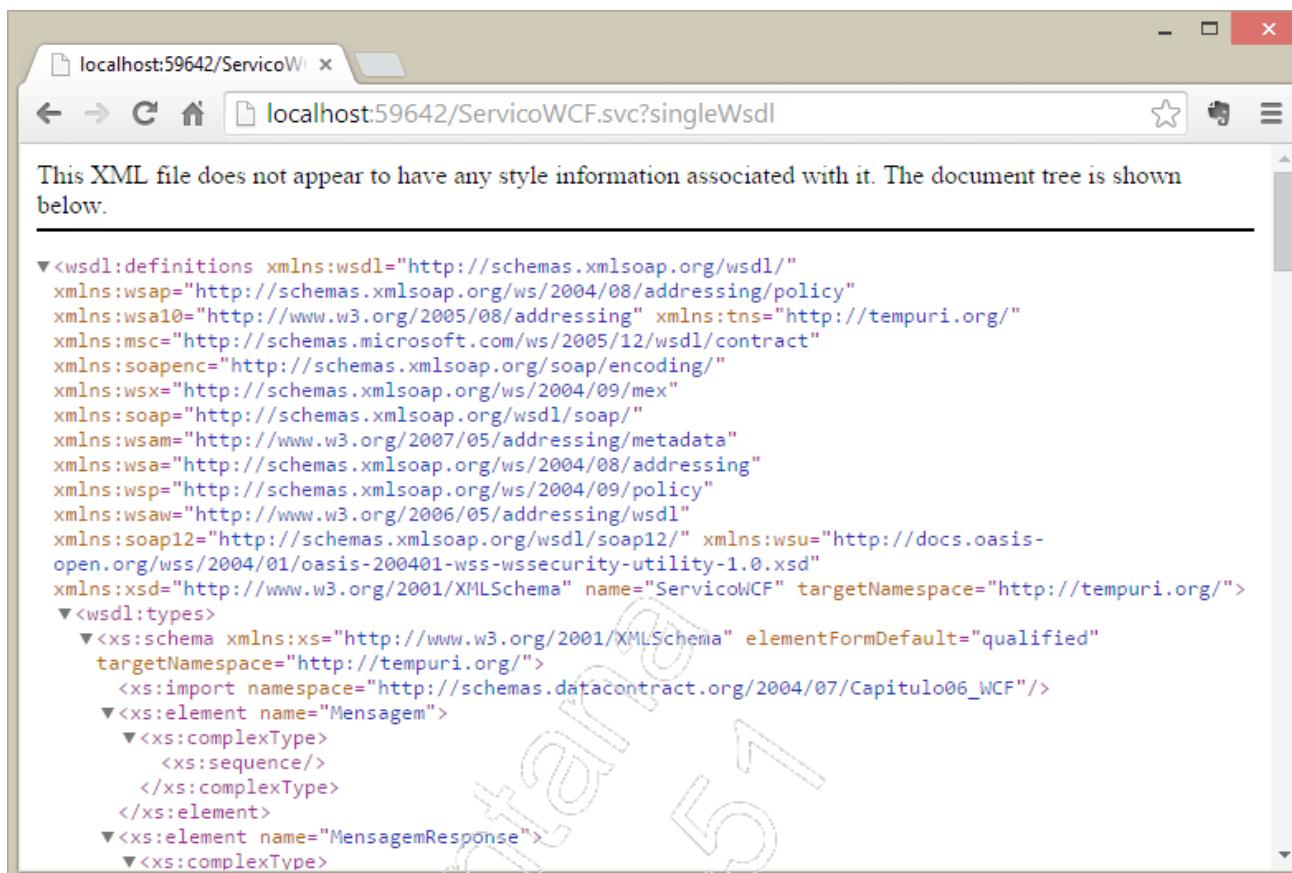
Na seguinte linha, os dados do livro são exibidos:

```
Console.WriteLine(
    "\nLivro 1: \n{0}\n{1}\nEditora {2}\n",
    livro.Titulo, livro.Autor, livro.Editora);
```

A classe **Livro**, assim como a classe **ServicoWCFClient**, foi gerada no momento da criação ou atualização da referência ao serviço. Mas como esse processo de referência consegue obter dados sobre o serviço e os tipos de dados envolvidos no processo? A resposta a isso é a especificação **WSDL (Web Services Description Language)**. O WSDL é um arquivo XML com todas as informações sobre o serviço: os métodos, os tipos envolvidos, o protocolo e as restrições. Quando uma referência a um serviço é efetuada no Visual Studio, é chamado o serviço acrescentando o texto **?SingleWsdl** no final, desta forma: **http://localhost:59642/ServicoWCF.svc?SingleWsdl**.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Isso retorna o arquivo XML com as especificações do serviço:



The screenshot shows a web browser window with the URL `localhost:59642/ServicoWCF.svc?singleWsdl`. The page displays the WSDL (Web Services Description Language) XML document. The XML defines a service named "ServicoWCF" with a target namespace "http://tempuri.org/". It includes definitions for types, such as "Mensagem" and "MensagemResponse", which are complex types containing sequences. The XML uses namespaces from various W3C and Microsoft schemas.

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy" xmlns:wsa10="http://www.w3.org/2005/08/addressing" xmlns:tns="http://tempuri.org/" xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing" xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="ServicoWCF" targetNamespace="http://tempuri.org/">
```

O Visual Studio cuida para deixar esses detalhes ocultos ao programador e raramente é necessário interpretar manualmente essas informações. Mas vale a pena olhar o arquivo e identificar onde estão definidos os elementos do serviço (partes da listagem foram omitidas para melhor leitura):

```
<wsdl:definitions ...>
  <wsdl:types>
    <xss:schema ...>
      <xss:import >
```

```
  <xss:element name="Mensagem">
    <xss:complexType>
      <xss:sequence/>
    </xss:complexType>
  </xss:element>
```

```
  <xss:element name="MensagemResponse">
    <xss:complexType>
      <xss:sequence>
        <xss:element ...>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
```

```
  <xss:element name="Somar">
    <xss:complexType>
      <xss:sequence>
        <xss:element minOccurs="0" name="x" type="xss:int"/>
        <xss:element minOccurs="0" name="y" type="xss:int"/>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
```

```
  <xss:element name="SomarResponse">
    <xss:complexType>
      <xss:sequence>
        <xss:element ...>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
```

```
  <xss:element name="ObterLivro">
    <xss:complexType>
      <xss:sequence>
        <xss:element ....>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

...

```
<xs:complexType name="Livro">
  <xs:sequence>
    <xs:element name="Autor" type="xs:string"/>
    <xs:element name="Editora" type="xs:string"/>
    <xs:element name="LivroId" type="xs:int"/>
    <xs:element name="NumeroPaginas" type="xs:int"/>
    <xs:element name="Titulo" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:element name="Livro" ...>
</xs:schema>
</wsdl:types>

<wsdl:message name="IServicoWCF_Mensagem_InputMessage">
  <wsdl:part name="parameters" element="tns:Mensagem"/>
</wsdl:message>

<wsdl:portType name="IServicoWCF">
  <wsdl:operation name="Mensagem"> ...
</wsdl:portType>
<wsdl:binding name="BasicHttpBinding_IServicoWCF" >
  <wsdl:operation name="Somar">
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      ...
    <soap:address
      location=
        "http://localhost:59642/ServicoWCF.svc"/>
  </wsdl:operation>
</wsdl:binding>
</wsdl:portType>
</wsdl:service>
</wsdl:definitions>
```

Em resumo, estes são os itens que fazem parte da descrição do serviço:

```
<wsdl:definitions>
    <wsdl:types> ...
    <wsdl:message name="IServicoWCF_Mensagem_InputMessage" ...>
    <wsdl:portType name="IServicoWCF" ...>
    <wsdl:binding name="BasicHttpBinding_IServicoWCF" ....>
    <wsdl:service name="ServicoWCF" ...>
</wsdl:definitions>
```

Em que:

- **types**: Define os métodos e tipos de classes que fazem parte do serviço;
- **message**: Define os dados de entrada e saída das mensagens;
- **portType**: Relaciona as mensagens aos tipos definidos em **Types**;
- **binding**: Define os protocolos de comunicação;
- **service**: Define a URL do serviço.

7.4.4. Criando um host

Para hospedar um serviço WCF, é necessário usar um programa chamado **host**. O IIS, por natureza, é um serviço para páginas HTML e aplicativos. Em muitos casos, porém, é melhor criar um host personalizado. Por exemplo, dentro da mesma rede, uma aplicação pode ser usada como host, seja ela **Windows Service**, **Console**, **Windows Forms**, ou qualquer outra. Esse tipo de aplicativo é chamado **Self-Host**.

Para criar um exemplo, será usado um projeto **Console**. Esse projeto precisa ter uma referência ao assembly **System.ServiceModel**. Usaremos a seguinte interface e implementação:

Visual Studio 2015 - ASP.NET com C# Acesso a dados

- Arquivo **ISimples.cs**

```
namespace ExemploSelfHost
{
    [ServiceContract]
    public interface ISimples
    {
        [OperationContract]
        string Mensagem();
    }
}
```

- Arquivo **Simples.cs**

```
namespace ExemploSelfHost
{
    public class Simples:ISimples
    {

        public string Mensagem()
        {
            return "Este é um WCF. Agora são " +
                DateTime.Now.ToShortTimeString();
        }
    }
}
```

Na classe **Program**, será criado um host:

```
class Program
{
    static void Main(string[] args)
    {
        //Define o endereço
        var url = new Uri("http://localhost:8989/simples");

        //Cria um Host
        var host = new ServiceHost(typeof(Simples), url);

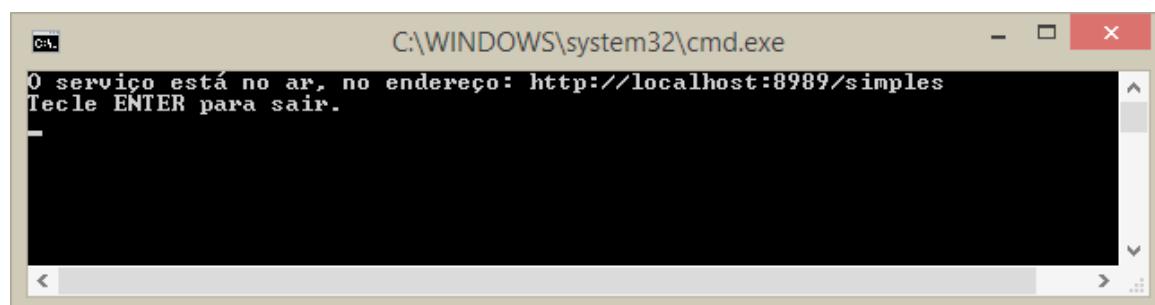
        //Adiciona um Behavior permitindoHttpGet
        var smb = new ServiceMetadataBehavior();
        smb.HttpGetEnabled = true;
        host.Description.Behaviors.Add(smb);

        //Inicia o serviço
        host.Open();

        Console.WriteLine("O serviço está no ar, no endereço:{0}",url);
        Console.WriteLine("Tecle ENTER para sair.");
        Console.ReadLine();

        // Fecha o serviço
        host.Close();
    }
}
```

Observe a execução do programa:



7.4.4.1.Criando a aplicação cliente para este host

Com o programa em execução, o serviço está disponível no endereço **http://localhost:8989/Simples**. Ao abrir o navegador e digitar o endereço na URL, deve aparecer a página de entrada:

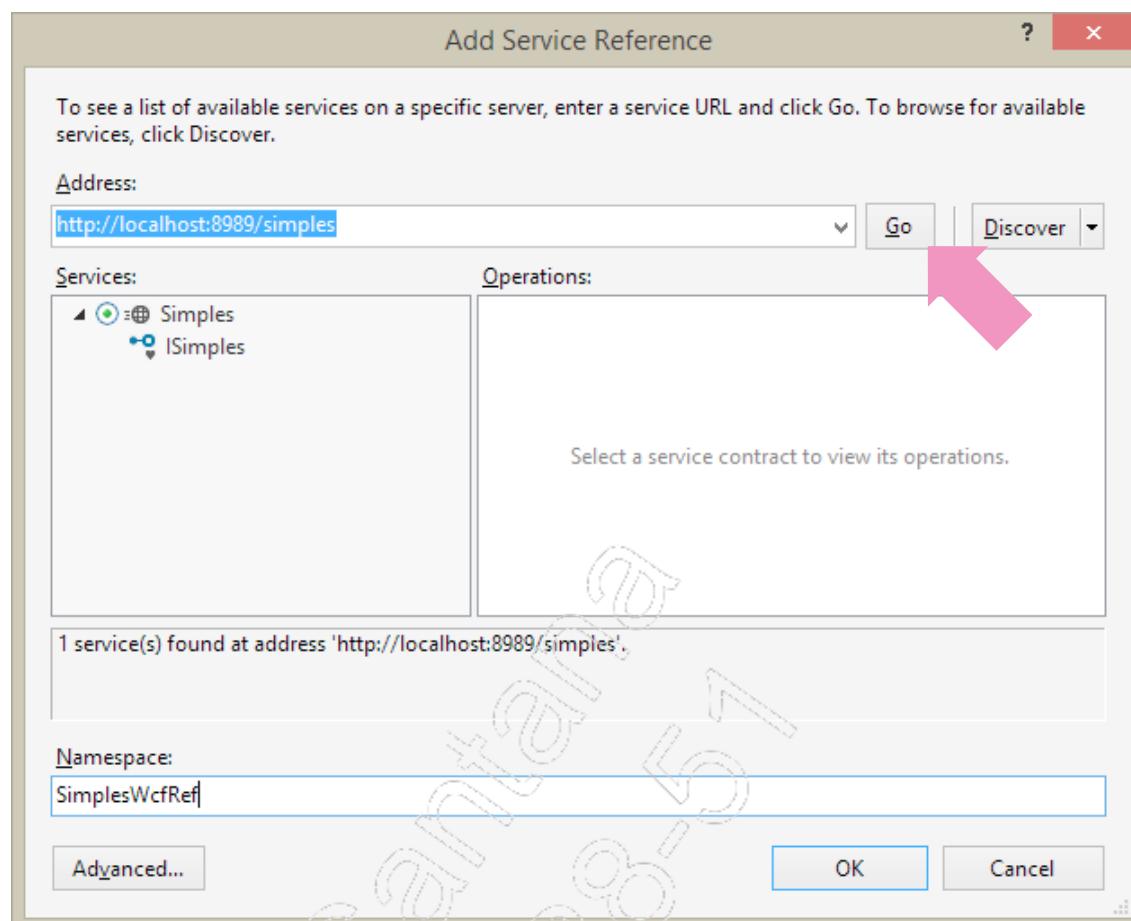


Não há nenhum procedimento especial para criar um client para esse serviço. Em qualquer tipo de projeto, basta adicionar uma referência ao serviço e digitar o endereço. Não é possível usar o botão **Discover** porque o endereço é criado dinamicamente, em tempo de execução.

No arquivo **Program.cs** do projeto que contém o serviço, esta é a linha que cria o endereço:

```
//Define o endereço  
var url = new Uri("http://localhost:8989/simples");
```

Em um novo projeto, escolha para adicionar uma referência a um serviço e digitar o endereço. Clique em **Go**, e não em **Discover**:



Após a referência estar no ar, basta chamar a classe **Proxy** criada e chamar os métodos disponíveis. Neste exemplo, foi usado um projeto **Console**:

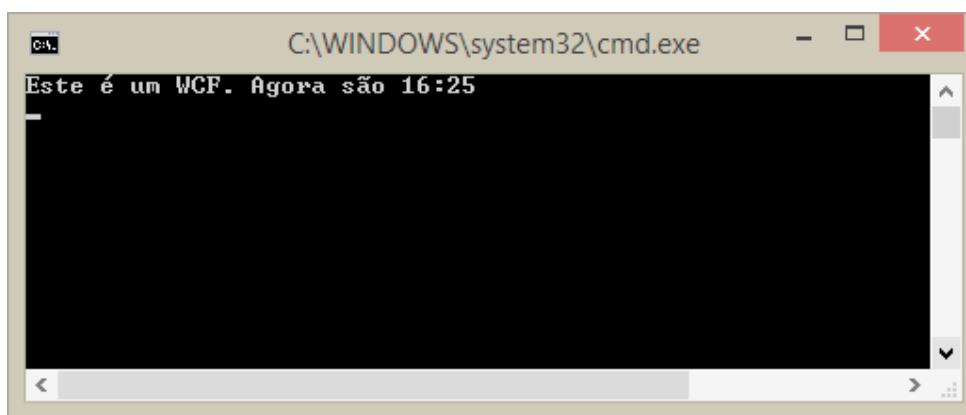
```
class Program
{
    static void Main(string[] args)
    {
        var wcf = new SimplesWcfRef.SimplesClient();
        string msg = wcf.Mensagem();

        Console.WriteLine(msg);
        Console.ReadLine();

    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Ao executar o programa, o serviço hospedado pelo aplicativo é chamado e o retorno é obtido:



O WCF é uma feliz combinação de tecnologias integradas e que podem ser moldadas para realizar qualquer tipo de tarefa. Mas é importante lembrar que a arquitetura baseada em serviços deve ser usada apenas quando é necessária. Quando as informações estão presentes no mesmo domínio, é sempre mais simples e eficiente criar **Class Library** e definir referências diretas aos arquivos.

O WCF e toda a arquitetura de serviços são úteis apenas quando a necessidade do projeto é de integração com sistemas ou dispositivos que não podem estar no mesmo domínio e estrutura física. O processo de serialização e transporte via HTTP (ainda é o mais usado) consome muitos recursos da rede e do ambiente de execução como um todo.

Pontos principais

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

- WCF é um conjunto de tecnologias que servem como base para criar aplicações orientadas a serviços;
- O atributo **ServiceContract** deve ser definido em uma interface para criar um serviço WCF;
- O atributo **OperationContract** deve ser definido em um método de uma interface para definir uma operação deste contrato;
- O atributo **DataContract** deve ser definido em uma classe que será recebida ou enviada por um serviço WCF;
- O atributo **DataMember** deve ser definido em uma propriedade de uma classe que será recebida ou transmitida por um serviço WCF;
- As três principais opções para criar aplicações distribuídas são: **WCF**, **Web API** e **Web Services**;
- Para ter acesso a um serviço, é necessário adicionar, no Visual Studio, uma referência a esse serviço. Isso cria uma classe chamada **Proxy** que se comunica de maneira transparente com o host onde o serviço está hospedado.

7

Data Services: WCF

Teste seus conhecimentos

Mikael B
Santana
57
426.279.
0000



IMPACTA
EDITORA

1. Qual dos cenários a seguir não é apropriado para uma aplicação que usa serviços?

- a) Um blog pessoal com um administrador.
- b) Um sistema de pedidos com integração com fornecedores.
- c) Um programa mobile que coleta informações de estoque e atualiza um servidor em tempo real.
- d) Um sistema de cadastro, geração e envio de nota fiscal eletrônica.
- e) Nenhuma das alternativas anteriores está correta.

2. Em ordem cronológica, quais foram as tecnologias para criação de serviços disponibilizadas aos programadores .NET pela Microsoft?

- a) Web API, Web Services e WCF.
- b) WCF, Web Services e Web API.
- c) Web Services, WCF e Web API.
- d) Web Services, Web API e WCF.
- e) Nenhuma das alternativas anteriores está correta.

3. Como se chama o processo de transformar um objeto na memória em um segmento de dados que pode ser armazenado fisicamente ou transferido pela rede?

- a) Compilação
- b) AJAX
- c) SOA
- d) Serialização
- e) Dessorialização

4. Quais atributos devem ser usados na interface e na declaração de um método para o uso em um Serviço WCF?

- a) ServiceContract na interface e OperationContract no método.
- b) ServiceContract no método e OperationContract na interface.
- c) Service na interface e Operation no método.
- d) Serializable na interface e Operation no método.
- e) Serializable no método e Operation na interface.

5. Onde devem ser usados os atributos **DataContract** e **DataMember**?

- a) Na interface que está marcada com o atributo **ServiceContract**.
- b) Na classe que implementa a interface marcada com o atributo **ServiceContract**.
- c) Na classe que será enviada ou recebida por um serviço WCF.
- d) Na classe que representa o serviço WCF no cliente (Proxy).
- e) Na classe que representa o serviço WCF no servidor.

7

Data Services: WCF

Mãos à obra!

Mikael B
Santana
426.279.38857



IMPACTA
EDITORA

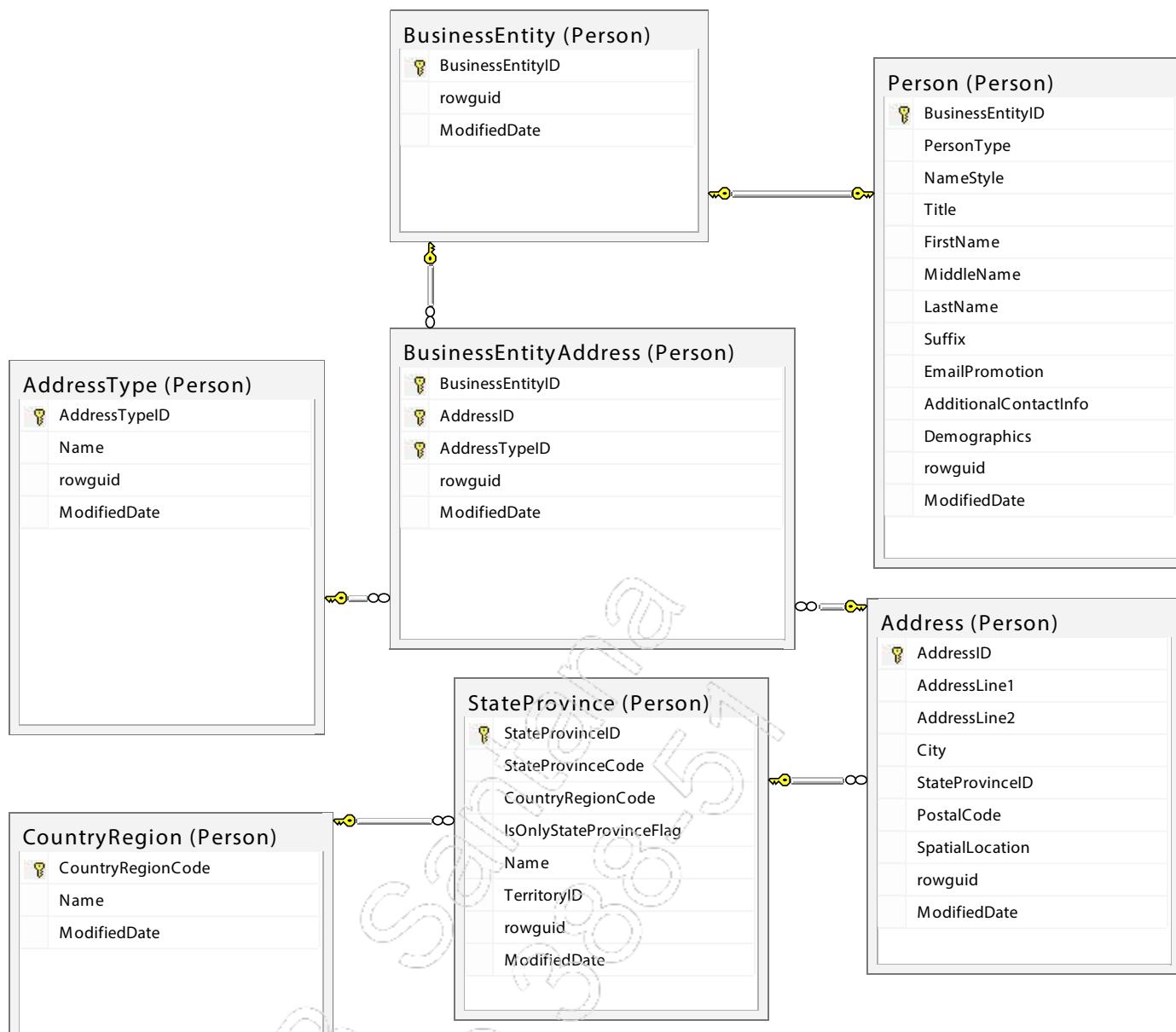
Laboratório 1

A - Criando um serviço WCF e um cliente Web Form

Neste laboratório, vamos criar um serviço WCF para exibir dados dos clientes da empresa **AdventureWorks**.

Neste banco de dados exemplo da Microsoft, as tabelas são construídas usando o modelo **esquema.Tabela**. Existem os esquemas **Person**, **Sales**, **HumanResources**, **Purchase** e **Production**. As seguintes tabelas fazem parte dos dados dos clientes:

- **Person.BusinessEntity**: Esta tabela contém apenas a chave primária **BusinessEntityId**, que é utilizada na tabela **Person.Person** e nas outras tabelas que definem **Funcionarios**, **Clientes**, **Vendedores** e **Fornecedores**. Esta tabela centraliza o código de várias entidades;
- **Person.Person**: Define os dados de uma pessoa, como **Nome** e **Sobrenome**;
- **Person.BusinessEntityAddress**: Lista de endereços de uma pessoa;
- **Person.Address**: Os endereços de uma pessoa listados em **BusinessEntityAddress**. Contém os dados como **Rua**, **Número**, **CEP**, **Cidade** e **Estado**;
- **Person.StateProvince**: Dados sobre um estado ou província de um país;
- **Person.CountryRegion**: Tabela contendo o nome dos países.



A expressão SQL a seguir retorna os países:

```

SELECT Pais.Name AS País
FROM Person.Person AS pessoa
INNER JOIN Person.BusinessEntityAddress AS enderecos
ON enderecos.BusinessEntityID = pessoa.BusinessEntityID
INNER JOIN Person.Address AS endereco ON
endereco.AddressID = enderecos.AddressID
INNER JOIN Person.StateProvince AS estado ON
estado.StateProvinceID = endereco.StateProvinceID
INNER JOIN Person.CountryRegion AS País ON
País.CountryRegionCode = estado.CountryRegionCode
GROUP BY País.Name
    
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Esta expressão retorna os estados ou regiões de um país (sempre usando a tabela **Person** para obter apenas resultados em que existem pessoas cadastradas):

```
SELECT estado.name  
  
FROM Person.Person AS pessoa  
    INNER JOIN Person.BusinessEntityAddress AS enderecos  
        ON enderecos.BusinessEntityID = pessoa.BusinessEntityID  
  
    INNER JOIN Person.Address AS endereco  
        ON endereco.AddressID = enderecos.AddressID  
  
    INNER JOIN Person.StateProvince AS estado ON  
        estado.StateProvinceID = endereco.StateProvinceID  
  
    INNER JOIN Person.CountryRegion AS Pais ON  
        Pais.CountryRegionCode = estado.CountryRegionCode  
  
WHERE Pais.Name=@pais  
GROUP BY estado.Name
```

Para obter os dados dos clientes, é necessário incluir a tabela **Sales.Customers** (vendas).

Outras informações relacionadas às pessoas estão nas tabelas **Person.EmailAddress**, **Person.PersonPhone** e **Person.PhoneNumberType**.

As tabelas adicionadas são as seguintes:

- **Sales.Customers**: Vendas dos clientes, pessoas ou empresas;
- **Person.EmailAddress**: E-mail dos registros da tabela **Person**. Uma pessoa ou empresa pode ter vários e-mails;
- **Person.PersonPhone**: A lista de telefones de uma pessoa;
- **Person.PhoneNumerType**: Os tipos possíveis de telefones, como fax, residência, celular etc.

A expressão a seguir retorna a lista de clientes de um estado:

```

SELECT
    pessoa.BusinessEntityID as Id,
    pessoa.FirstName + ' ' + pessoa.LastName as Nome,
    telefone.PhoneNumber + ' - ' + tipoTelefone.Name
        AS Telefone,
    email.EmailAddress as Email,
    endereco.City Cidade,
    Pais.Name AS Pais

FROM      Person.Person AS pessoa

    INNER JOIN Person.BusinessEntityAddress AS enderecos ON
        enderecos.BusinessEntityID = pessoa.BusinessEntityID

    INNER JOIN Person.Address AS endereco ON
        endereco.AddressID = enderecos.AddressID

    INNER JOIN Person.StateProvince AS estado ON
        estado.StateProvinceID = endereco.StateProvinceID

    INNER JOIN Person.CountryRegion AS Pais ON
        Pais.CountryRegionCode = estado.CountryRegionCode

    INNER JOIN Sales.Customer AS cliente
        ON cliente.PersonID = pessoa.BusinessEntityID

    LEFT JOIN Person.EmailAddress AS email ON
        email.BusinessEntityID = pessoa.BusinessEntityID

    LEFT JOIN Person.PersonPhone AS telefone ON
        telefone.BusinessEntityID = pessoa.BusinessEntityID

    LEFT JOIN Person.PhoneNumberType AS tipoTelefone ON
        tipoTelefone.PhoneNumberTypeID = telefone.PhoneNumberTypeID

WHERE      (cliente.StoreID IS NULL)
        AND Estado.Name=@estado
        AND Pais.Name=@pais

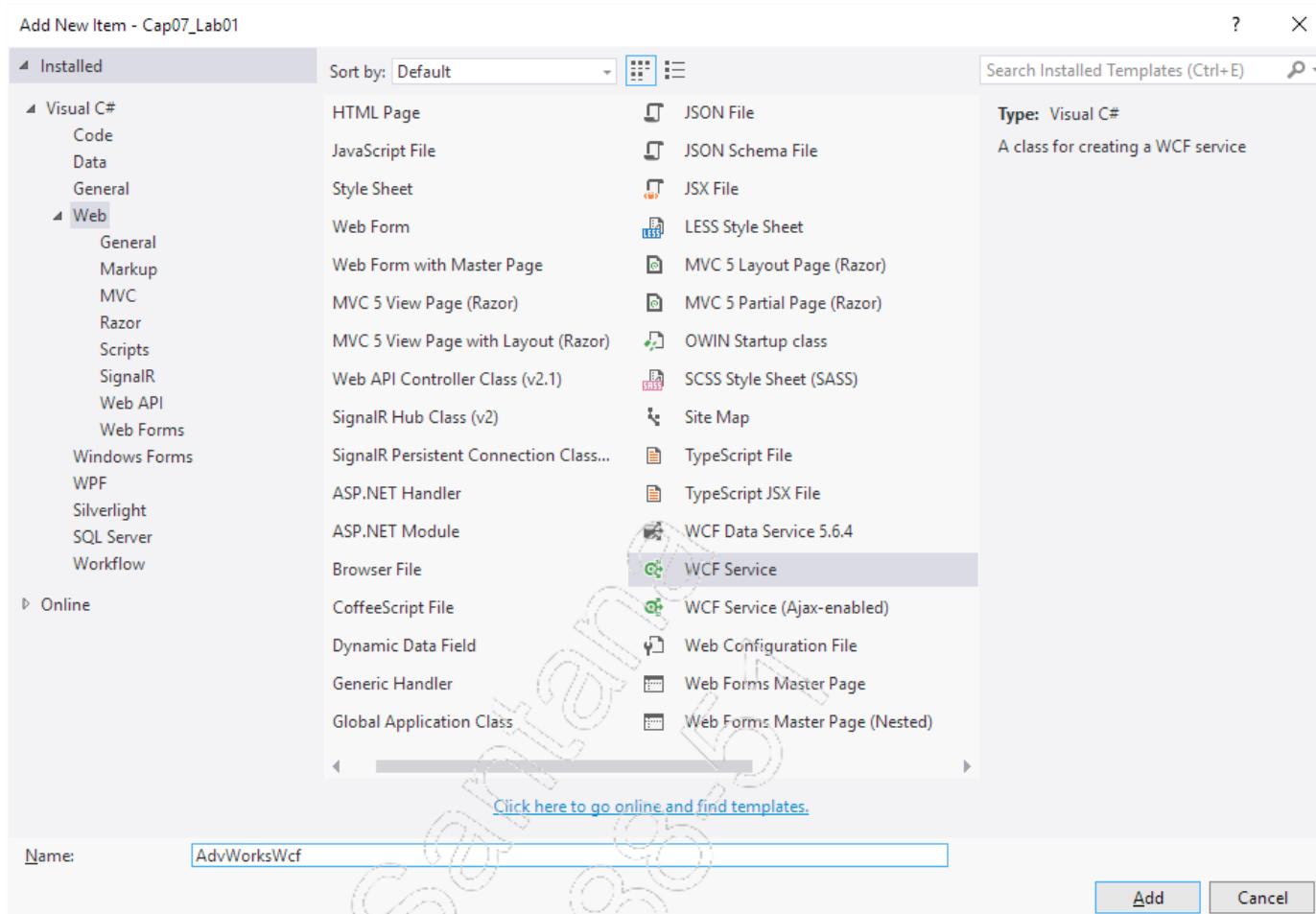
ORDER BY 2

```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

1. Crie um novo projeto Web vazio chamado **Cap07_Lab01**;

2. Adicione um serviço WCF chamado **AdvWorksWcf**;



3. Crie uma pasta chamada **Models**. Dentro dessa pasta, inclua uma classe chamada **ClienteAvulso** e marque-a com os atributos **DataContract** e **DataMember**:

```
[DataContract]
public class ClienteAvulso
{
    [DataMember(Order=0)]
    public string Nome { get; set; }

    [DataMember(Order=1)]
    public string Telefone { get; set; }

    [DataMember(Order=2)]
    public string Email { get; set; }

    [DataMember(Order=3)]
    public string Cidade { get; set; }
}
```

Essa classe será usada para retornar a lista de clientes de um estado e país. Para este exemplo, serão criados três métodos:

- **ClientePaisLista()**: Retorna a lista de países;
- **ClienteEstadoLista(pais)**: Retorna os estados de um país;
- **ClientesAvulsosPorPaisEstado(pais, estado)**: Retorna os clientes avulsos de um estado e país.

4. Na interface **IAdvWorksWcf**, insira os atributos **ServiceContract** e **OperationContract** e altere as declarações conforme o seguinte código:

```
[ServiceContract]
public interface IAdvWorksWcf
{
    [OperationContract]
    List<string> ClientePaisLista();

    [OperationContract]
    List<string> ClienteEstadoLista(string pais);

    [OperationContract]
    List<ClienteAvulso> ClientesAvulsosPorPaisEstado(
        string pais, string estado);
}
```

5. Antes de implementar os métodos na classe de serviço, vamos incluir uma classe para acesso aos dados chamada **AwDb** (AdventureWorks DataBase). Neste exemplo, vamos usar o ADO.NET tradicional, preenchendo a classe **ClienteAvulso**. Adicione a classe chamada **AwDb**:

```
public class AwDb
{
}
```

6. Crie uma região para armazenar a string de conexão e alguns métodos que facilitam executar comandos:

```
public class AwDb
{
    #region Database-helper

    #endregion

}
```

7. Dentro dessa região, inclua a string de conexão:

```
public class AwDb
{
    #region Database-helper

    //
    //Retorna a string de conexão
    //
    private static string conexao =
        @"Data Source=localhost\sqlexpress;
        Initial Catalog=AdventureWorks2012;
        Integrated Security=True; ";

    #endregion

}
```

O próximo passo é incluir um método que retorne um objeto do tipo **DbCommand** uma vez recebidos uma expressão SQL e um conjunto de parâmetros. Esse método é de grande ajuda para criar aplicações que utilizam ADO.NET. A sintaxe é a seguinte:

```
DbCommand ObterCommand(string sql, params object[] parametros)
```

Repare que ele recebe como parâmetros uma expressão SQL e um array de objetos. Veja possíveis usos:

- **Sem parâmetros**

```
var cmd=ObterCommand("Select * from Produtos");
```

- Com um parâmetro

```
string sql="Select * from Produto Where Código=@codigo";
var cmd=ObterComand(sql, "@codigo", 1);
```

- Com dois parâmetros

```
string sql="SELECT * from Clientes Where Cidade=@cidade and
Idade>@Idade";
var cmd=ObterComand(sql, "@cidade", "São Paulo", "@idade", 18)
```

A única regra dos parâmetros é que sempre deve ser um ou mais pares na sequência: **nomeDoParametro(string)**, **valorDoParametros(objeto)**. No exemplo anterior, o primeiro par era "@cidade", "São Paulo" (duas strings) e o segundo era "@idade", 18 (uma string e um inteiro).

8. Inclua o método que retorna o objeto cmdCommand:

```
//
//Retorna um objeto DbCommand com a conexão,
//expressão SQL e Parâmetros preenchidos
//
private static DbCommand ObterCommand(
    string sql, params object[] parametros)
{
    var cn = new SqlConnection(conexao);
    var cmd = new SqlCommand();
    cmd.CommandText = sql;
    cmd.Connection = cn;
    if (parametros.Length > 0)
    {
        for (int i = 0; i < parametros.Length ; i+=2)
        {
            cmd.Parameters.AddWithValue(
                parametros[i].ToString(),
                parametros[i + 1]);
        }
    }
    return cmd;
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

O próximo método retorna uma DbDataReader. Isso permite que o aplicativo que chamou o método decida se quer utilizar uma lista genérica, um DataSet ou qualquer outro método de interagir com os dados.

9. Inclua o método que retorna o objeto **DbDataReader**:

```
//  
//Obter Data Reader  
//Retorna uma instância de um DataReader  
private static DbDataReader ObterDataReader(DbCommand cmd)  
{  
    cmd.Connection.Open();  
    var reader=cmd.ExecuteReader(  
        CommandBehavior.CloseConnection);  
    return reader;  
}
```

10. Inclua uma região chamada **Expressões SQL** (que vai conter as expressões SQL) e defina três variáveis:

```
public class AwDb  
{  
    #region Database-helper  
    private static string conexao...  
    private static DbCommand ObterCommand...  
    private static DbDataReader ObterDataReader....  
    #endregion  
  
    #region Expressões SQL  
    private const string sqlPaisesLista=@"";  
    private const string sqlEstadosLista=@"";  
    private const string sqlClientesAvulsosPorEstado=  
        @"";  
    #endregion  
}
```

11. Altere a constante **SqlPaisesLista**. O conteúdo dessa expressão é o seguinte:

```
//  
//Lista de países  
//  
private const string sqlPaisesLista=  
    @"SELECT Pais.Name AS País  
  
        FROM Person.Person AS pessoa  
  
        INNER JOIN Person.BusinessEntityAddress AS enderecos  
            ON enderecos.BusinessEntityID =  
                pessoa.BusinessEntityID  
  
        INNER JOIN Person.Address AS endereco ON  
            endereco.AddressID = enderecos.AddressID  
  
        INNER JOIN Person.StateProvince AS estado ON  
            estado.StateProvinceID =  
                endereco.StateProvinceID  
  
        INNER JOIN Person.CountryRegion AS País ON  
            País.CountryRegionCode =  
                estado.CountryRegionCode  
  
    GROUP BY País.Name";
```

12. Altere a expressão **SqlEstadosLista**, conforme o seguinte código:

```
//  
//Lista de cidade  
//  
private const string sqlEstadosLista =  
    @"SELECT estado.name  
  
        FROM Person.Person AS pessoa  
  
        INNER JOIN Person.BusinessEntityAddress AS enderecos ON enderecos.BusinessEntityID =  
            pessoa.BusinessEntityID  
        INNER JOIN Person.Address AS endereco ON  
            endereco.AddressID = enderecos.AddressID  
  
        INNER JOIN Person.StateProvince AS estado ON  
            estado.StateProvinceID =  
                endereco.StateProvinceID  
  
        INNER JOIN Person.CountryRegion AS País ON  
            País.CountryRegionCode =  
                estado.CountryRegionCode  
  
    WHERE País.Name=@pais  
  
    GROUP BY estado.Name";
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

13. Altere a expressão **SqlClientesAvulsosPorEstado**, de acordo com o código a seguir:

```
//  
//Lista de Clientes Avulsos  
//  
private const string sqlClientesAvulsosPorEstado=  
@"SELECT pessoa.BusinessEntityID as Id,  
  
    pessoa.FirstName + ' ' + pessoa.LastName as Nome,  
    telefone.PhoneNumber + ' - ' + tipoTelefone.Name AS ]  
    Telefone,  
    email.EmailAddress as Email,  
    endereco.City Cidade,  
    Pais.Name AS Pais  
  
    FROM Person.Person AS pessoa  
    INNER JOIN Person.BusinessEntityAddress AS enderecos  
        ON enderecos.BusinessEntityID =  
            pessoa.BusinessEntityID  
  
    INNER JOIN Person.Address AS endereco ON  
        endereco.AddressID = enderecos.AddressID  
  
    INNER JOIN Person.StateProvince AS estado ON  
        estado.StateProvinceID =  
            endereco.StateProvinceID  
  
    INNER JOIN Person.CountryRegion AS Pais ON  
        Pais.CountryRegionCode =  
            estado.CountryRegionCode  
  
    INNER JOIN Sales.Customer AS cliente ON  
        cliente.PersonID = pessoa.BusinessEntityID  
  
    LEFT JOIN Person.EmailAddress AS email ON  
        email.BusinessEntityID = pessoa.BusinessEntityID  
  
    LEFT JOIN Person.PersonPhone AS telefone ON  
        telefone.BusinessEntityID =  
            pessoa.BusinessEntityID  
    LEFT JOIN Person.PhoneNumberType AS tipoTelefone ON  
        tipoTelefone.PhoneNumberTypeID =  
            telefone.PhoneNumberTypeID  
    WHERE (cliente.StoreID IS NULL) AND  
        Estado.Name=@estado AND  
        Pais.Name=@pais  
  
    ORDER BY 2";
```

14. Defina a última região, que é a que engloba os métodos:

```
#region Métodos De Acesso a Dados

//  
//Cliente País Lista  
//  
public static DbDataReader ClientePaisLista()  
{  
    var cmd = ObterCommand(sqlPaisesLista);  
    return ObterDataReader(cmd);  
}  
  
//  
//Cliente Estado Lista  
//  
public static DbDataReader ClienteEstadoLista(  
    string pais)  
{  
    var cmd = ObterCommand(sqlEstadosLista,  
        "@pais", pais);  
    return ObterDataReader(cmd);  
}  
  
//  
//Clientes Avulsos Por País  
//  
public static DbDataReader  
    ClientesAvulsosPorPaísEstado(string pais,  
        string estado)  
{  
    var cmd = ObterCommand(sqlCientesAvulsosPorEstado,  
        "@Pais", pais, "@estado", estado);  
    return ObterDataReader(cmd);  
}  
#endregion
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

15. No arquivo **IAdvWorksWcf**, ou seja, na classe que implementa a interface, crie os métodos para implementação:

```
public class AdvWorksWcf : IAdvWorksWcf
{
    public List<string> ClientePaisLista()
    {
        var lista = new List<string>();
        using (var dr = AwDb.ClientePaisLista())
        {
            while (dr.Read())
            {
                lista.Add(dr[0].ToString());
            }
        }

        return lista;
    }

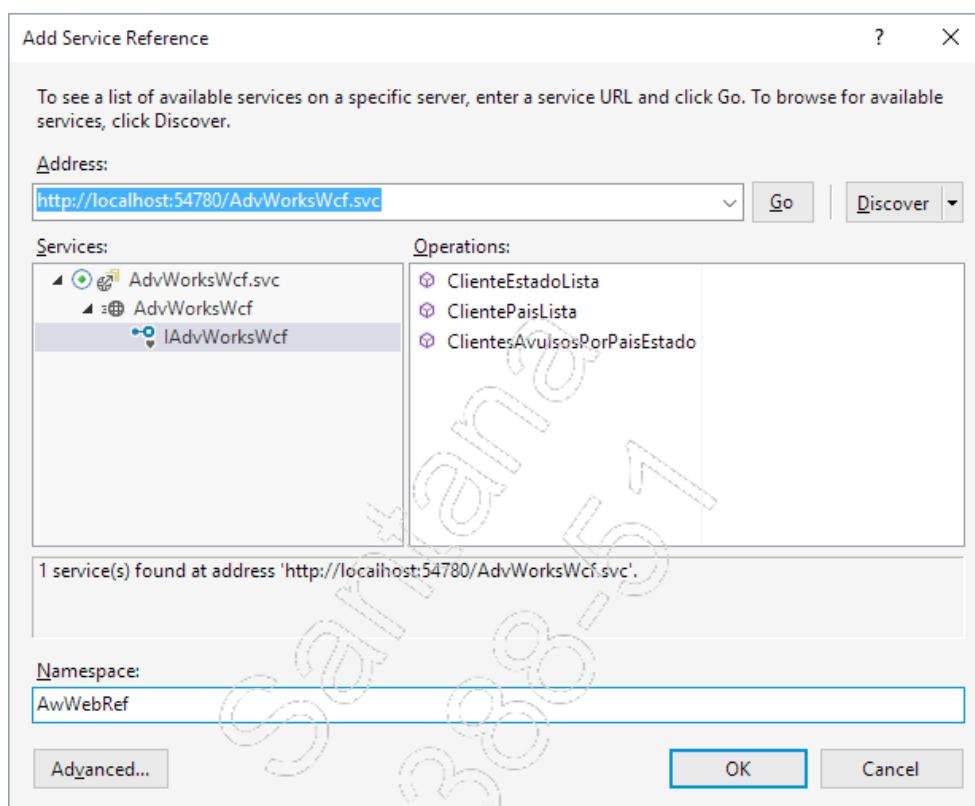
    public List<string> ClienteEstadoLista(string pais)
    {
        var lista = new List<string>();
        using (var dr = AwDb.ClienteEstadoLista(pais))
        {
            while (dr.Read())
            {
                lista.Add(dr[0].ToString());
            }
        }

        return lista;
    }

    public List<ClienteAvulso> ClientesAvulsosPorPaisEstado(
        string pais, string estado)
    {
        var lista = new List<ClienteAvulso>();
        using (var dr = AwDb.ClientesAvulsosPorPaisEstado(
            pais, estado))
        {
            while (dr.Read())
            {
                var cli = new ClienteAvulso()
                {
                    Nome = dr["Nome"].ToString(),
                    Email = dr["Email"].ToString(),
                    Cidade = dr["Cidade"].ToString(),
                    Telefone = dr["Telefone"].ToString()
                };
                lista.Add(cli);
            }
        }
        return lista;
    }
}
```

16. Para testar o serviço, adicione um projeto ASP.NET vazio na solução chamado **Cap07_Lab01_Teste**. Pode visualizar o serviço no ar. Esse projeto vai utilizar Web Forms para criar uma interface de consulta;

17. Adicione, nesse novo projeto, uma referência ao serviço criado. Use o botão **Discovery** para localizar o endereço. Use o namespace **AwWebRef** (AdventureWorks Web Reference):



18. Adicione, nesse novo projeto, um Web Form chamado **Default.aspx**;

A página HTML tem três partes: uma lista de países (**dropdownList**), uma lista de estados (**DropDownList**) e uma lista de clientes (**GridView**). O usuário escolhe o país, então, a lista de estados daquele país é exibida. Ao escolher um estado, a lista de clientes daquele estado é exibida.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

19. Insira o código HTML dentro do form, na página **Default.aspx**, dentro do Form:

```
<h1>Adventure Works</h1>
```

```
<h2>Clientes</h2>
```

```
<section>
    Escolha um país:<br />
    <asp:DropDownList ID="paisesDropDownList" runat="server"
                      AutoPostBack="true">
    </asp:DropDownList>
</section>
```

```
<asp:Panel runat="server" ID="estadoPanel" Visible="false">
    <section>
        Escolha um Estado:<br />
        <asp:DropDownList ID="estadoDropDownList" runat="server"
                          AutoPostBack="true">
        </asp:DropDownList>
    </section>
</asp:Panel>
```

```
<asp:Panel runat="server" ID="gridPanel" Visible="false">
    <section>
        <asp:GridView ID="clientesGridView" runat="server"
                      AllowPaging="True" CssClass="tabela">
        </asp:GridView>
    </section>
</asp:Panel>
```

20. Adicione nesse projeto uma folha de estilos (**estilos.css**) e faça a referência na página **Default.aspx**:

```
<head runat="server">
    <title></title>
```

```
    <link href="estilos.css" rel="stylesheet"
          type="text/css" />
```

```
</head>
```

21. Insira o conteúdo da folha de estilo:

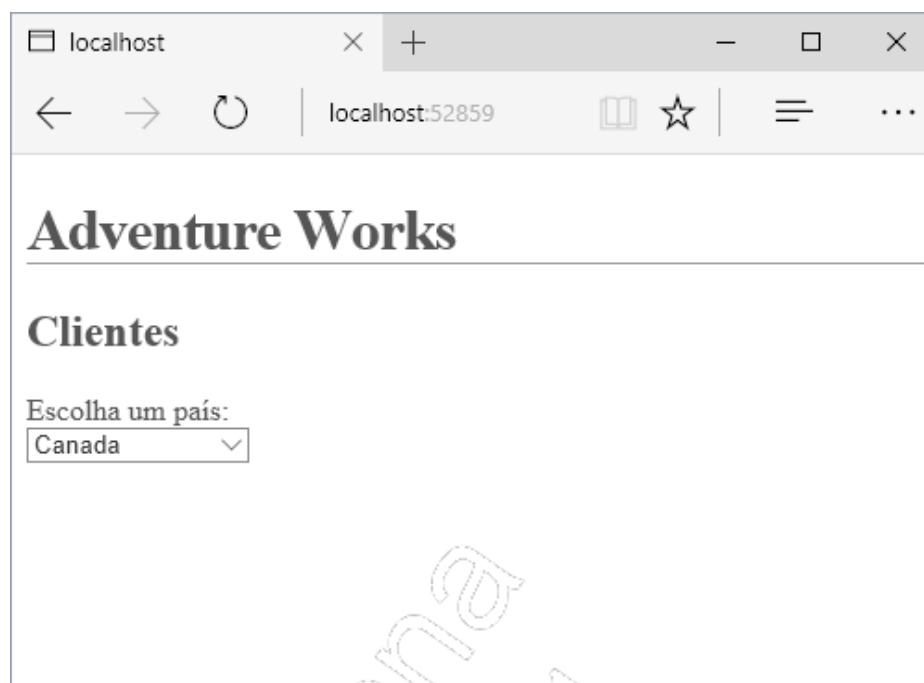
```
body {  
    color:#393939;  
}  
  
h1 {  
    border-bottom:1px solid #808080;  
    letter-spacing:-1px;  
}  
  
section{  
    margin-bottom:30px;  
}  
  
.tabela th{  
    padding:6px;  
}  
  
.tabela td {  
    padding:6px;  
}
```

22. No code-behind, no evento **Load**, a lista de países será carregada por meio de uma chamada do método. Insira o seguinte código:

```
protected void Page_Load(object sender, EventArgs e)  
{  
    if (!Page.IsPostBack)  
    {  
        CarregarPaises();  
    }  
}  
  
private void CarregarPaises()  
{  
    var wcf = new AwWebRef.AdvWorksWcfClient();  
    paisesDropDownList.DataSource =  
        wcf.ClientePaisLista();  
    paisesDropDownList.DataBind();  
    paisesDropDownList.Items.Insert(0, string.Empty);  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

23. Defina que esse projeto é o projeto inicial e execute para ver a lista de países:



24. Ao escolher um país, os estados devem aparecer. Insira o método do evento **SelectedIndexChanged** do dropdown País:

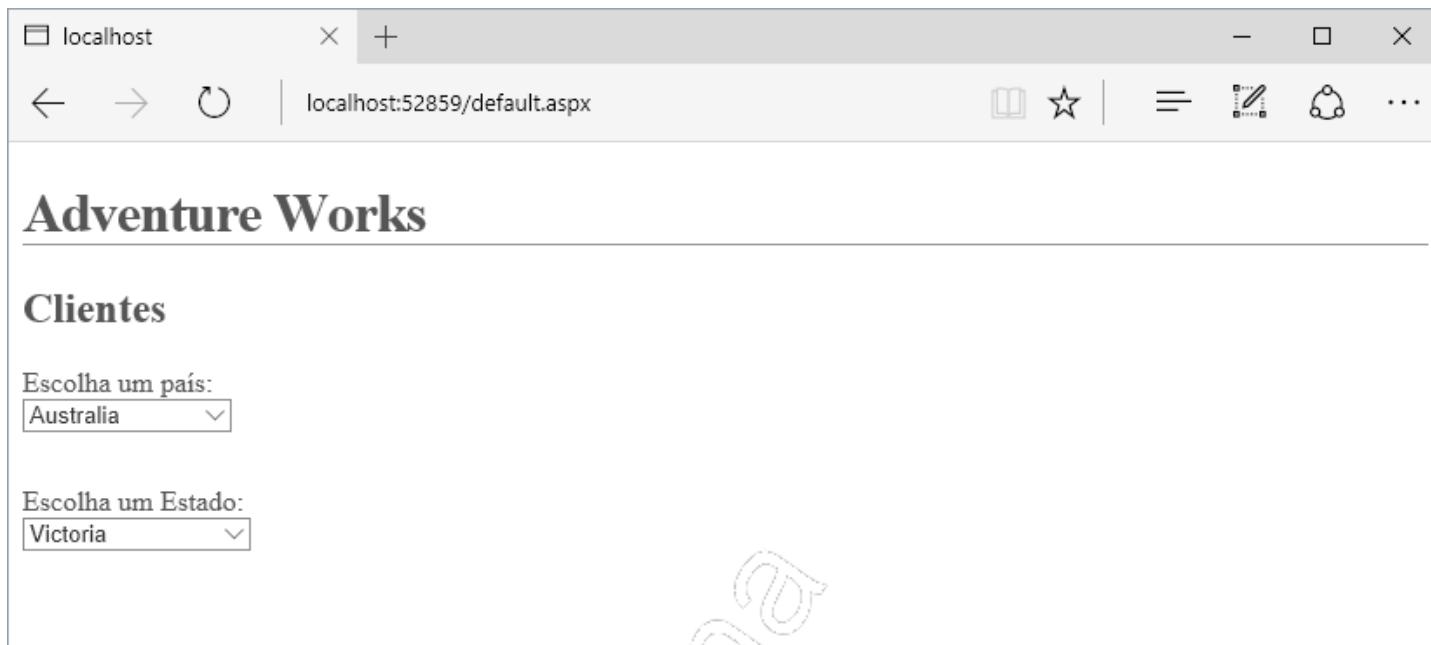
```
protected void paisesDropDownList_SelectedIndexChanged(
    object sender, EventArgs e)
{
    gridPanel.Visible = false;
    estadoPanel.Visible = false;
    if (paisesDropDownList.SelectedIndex > 0)
    {
        string pais = paisesDropDownList.SelectedValue;
        CarregarEstadosDeUmPaís(pais);
    }
}

private void CarregarEstadosDeUmPaís(string pais)
{
    var wcf = new AwWebRef.AdvWorksWcfClient();
    estadoPanel.Visible = true;

    estadoDropDownList.DataSource =
        wcf.ClienteEstadoLista(pais);
    estadoDropDownList.DataBind();
    estadoDropDownList.Items.Insert(0, string.Empty);

}
```

25. Teste e veja a lista de estados de um país aparecer:



26. Quando um estado é escolhido, a lista de clientes daquele estado e país deve aparecer. O evento **SelectedIndexChanged** do controle **estadoDropDownList** é o responsável por fazer aparecer a lista. Acrescente o código adiante:

```
protected void estadoDropDownList_SelectedIndexChanged(
    object sender, EventArgs e)
{
    if (estadoDropDownList.SelectedIndex > 0)
    {
        string pais = paisesDropDownList.SelectedValue;
        string estado = estadoDropDownList.SelectedValue;
        CarregarClientesDeUmEstado(pais, estado);
    }
}

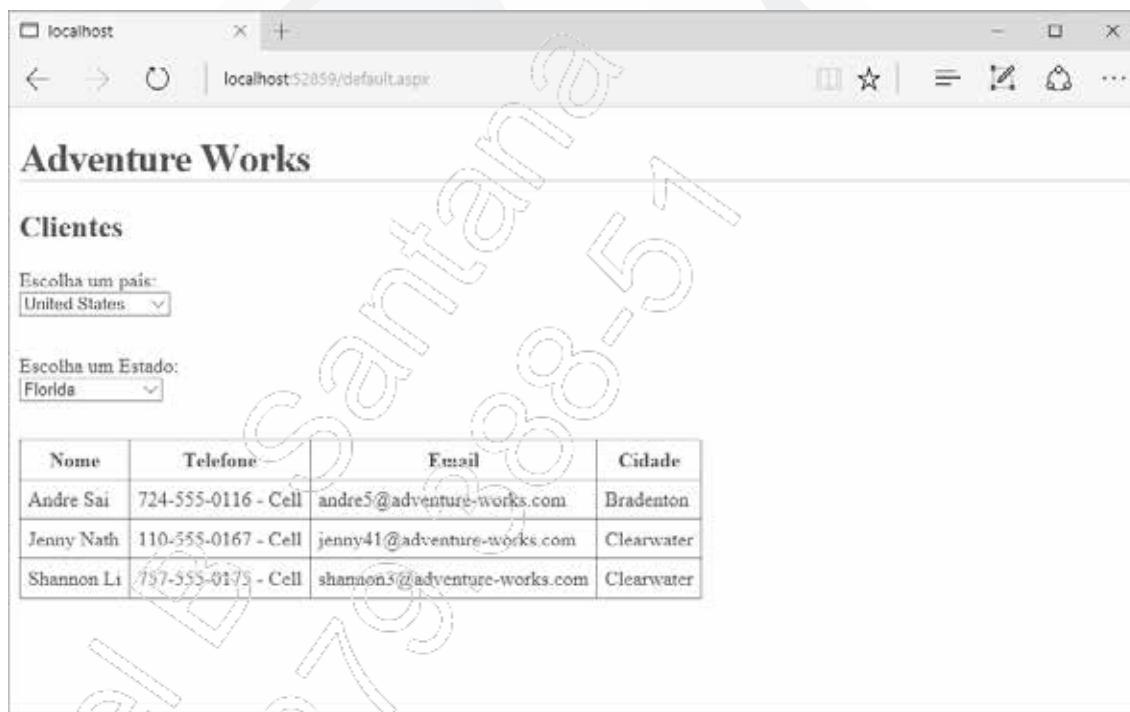
private void CarregarClientesDeUmEstado(
    string pais, string estado)
{
    var wcf = new AwWebRef.AdvWorksWcfClient();
    var lista = wcf.ClientesAvulsosPorPaisEstado(
        pais, estado);
    clientesGridView.DataSource = lista;
    clientesGridView.DataBind();
    gridPanel.Visible = true;
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

27. Finalmente, quando uma página é clicada, o evento **SelectedPageChanging** da **GridView** deve ser definido. Utilize o seguinte código:

```
void clientesGridView_PageIndexChanging(  
    object sender, GridViewEventArgs e)  
{  
    clientesGridView.PageIndex = e.NewPageIndex;  
  
    string pais = paisesDropDownList.SelectedValue;  
    string estado = estadoDropDownList.SelectedValue;  
    CarregarClientesDeUmEstado(pais, estado);  
}
```

28. Teste o programa inteiro:



Sugestões de estudo:

- Criar um client do mesmo serviço usando o modelo MVC;
- Retornar páginas específicas do serviço. Um serviço deve sempre retornar o mínimo de dados possível;
- Teste com todos os países. Algumas listagens podem dar erro devido ao volume grande de dados. Seria interessante mais outro filtro. Estude a estrutura do banco de dados e tente definir um filtro que selecione um conjunto menor de registros de cada vez. Identificar esse tipo de situação e pensar em soluções é algo que deve ser sempre treinado. Um sistema sempre pode ser melhor. Tente identificar e imaginar soluções de otimização.

8

Serviços: Web API

- ✓ Introdução à Web API;
- ✓ REST na prática;
- ✓ Usando a Web API.

8.1. 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.

8.2. 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 à 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;
- **Code on demand:** Todo framework ASP.NET utiliza o recurso de enviar JavaScript para ser executado no cliente. Um bom exemplo disso são os validadores (**RequiredFieldValidator**, **Custom Validator**, **CompareValidator**, **RangeValidator**).

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
```

 Existem vários programas que permitem ver as requisições e respostas vindas da rede. O Fiddler é um dos mais conhecidos e pode ser obtido pela seguinte URL: <http://www.telerik.com/fiddler>.

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:

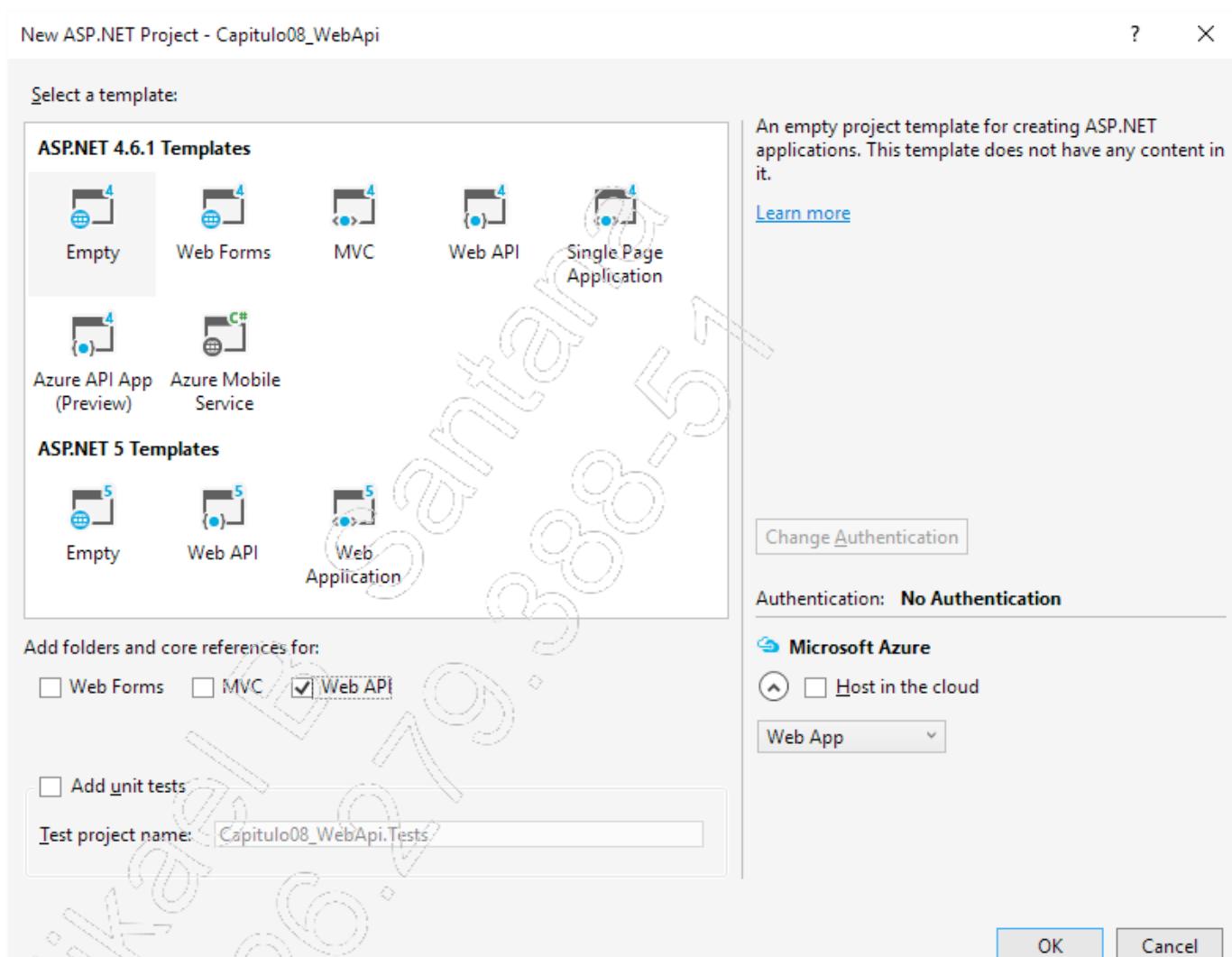
```
GET http://localhost:1513/api/Produtos HTTP/1.1
Accept: application/json
```

Resumindo, o ambiente Web usando o protocolo HTTP atende a todos os requisitos para criar aplicativos RESTful.

8.3. Usando a Web API

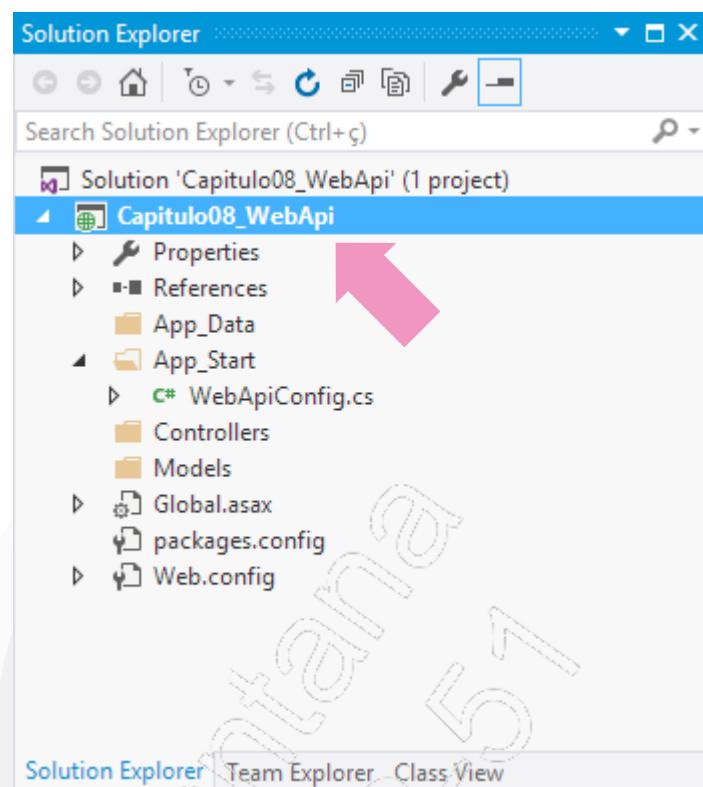
O framework Web API contém toda a funcionalidade necessária para criar aplicações RESTful e pode ser usado em qualquer tipo de projeto, como Web Forms, MVC ou simples páginas HTML.

Para usar o framework Web API, basta selecionar, na caixa de diálogo **Web API**, a criação de qualquer novo projeto Web:



8.3.1. Estrutura Web API

No caso de um projeto Web vazio, a seguinte estrutura é criada:



- **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**:

```
public class Global : HttpApplication
{
    void Application_Start(object sender, EventArgs e)
    {

        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.

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 seguir, está a definição da classe gerada pelo modelo do Visual Studio:

```
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 }  
        );  
    }  
}
```

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:

```
config.Routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
);
```

- **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:

```
config.Routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
);
```

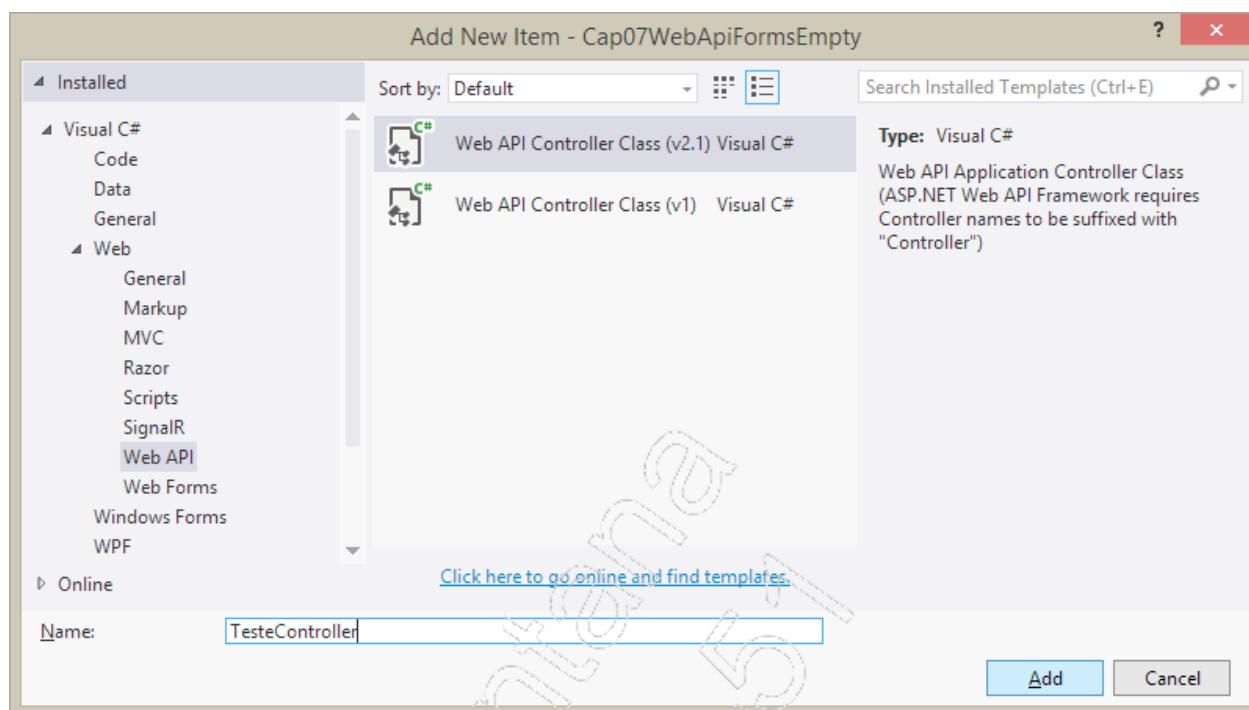
O parâmetro **defaults** é do tipo **object**, ou seja, pode ser passado a qualquer classe. O aplicativo analisa as propriedades do objeto e verifica se coincide com o template. Nessa configuração gerada pelo Visual Studio, é usado um tipo anônimo com a propriedade **Id** tendo o valor da propriedade **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.

8.3.2. Criando um serviço REST

Uma vez que o ambiente está preparado, para criar um serviço usando o framework Web API, basta adicionar um item no projeto do tipo Web.Api service:



É importante saber que o nome da classe deve acabar em **Controller**. Nesse exemplo, utilizamos o nome **TesteController**. A classe pode ser gravada em qualquer lugar no projeto, porém, existe uma pasta especial já criada chamada **Controllers**. A menos que haja uma razão especial, é recomendado colocar nessa pasta para manter o mais padronizado possível.

A classe gerada automaticamente contém os seguintes métodos: GET, POST, PUT e DELETE, que são exatamente os nomes dos verbos mais comuns do protocolo HTTP. Isso não é uma coincidência! O nome do método, no caso da Web API, faz toda a diferença. É por meio no verbo usado na solicitação HTTP que o framework Web API decide qual método chamar.

Assim, se o verbo usado para chamar a URL **Servidor / Teste** for GET, o método chamado é o método GET. Se o verbo usado para chamar a mesma URL for POST, o método POST é chamado, e assim por diante. Muito simples e direto.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

A listagem a seguir mostra apenas o nome dos métodos, para maior clareza:

```
public class TesteController : ApiController
{
    public IEnumerable<string> Get()

    public string Get(int id)

    public void Post([FromBody]string value)

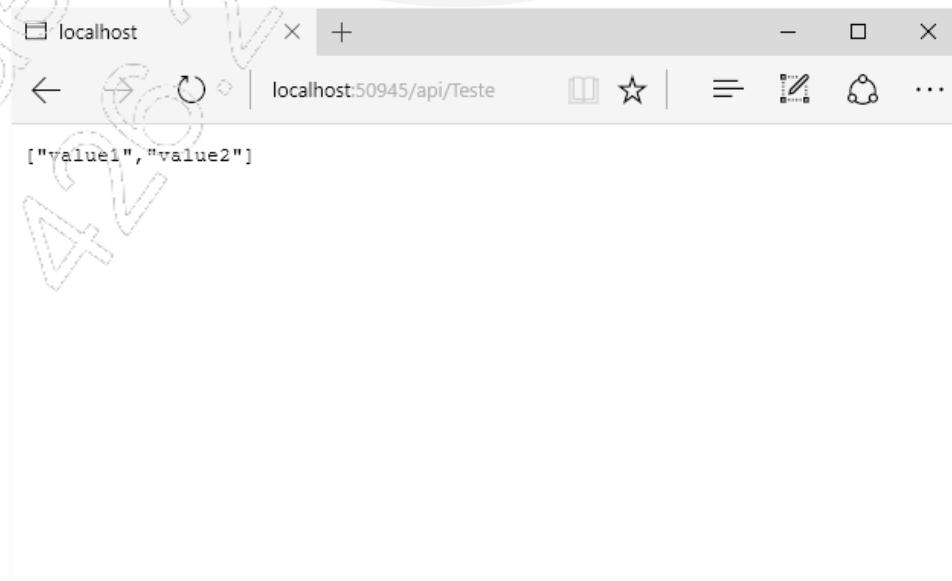
    public void Put(int id, [FromBody]string value)

    public void Delete(int id)
}
```

É possível testar os métodos GET apenas com o navegador, porque não precisa enviar dados. Para testar os outros métodos, é necessário construir um aplicativo cliente ou usar programas como Fiddler ou PostMan.

Para testar o GET, basta abrir o navegador e digitar a URL no formato correto.

```
// GET api/<controller>
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}
```



Este método retorna uma coleção de strings. Pode ser retornada qualquer coleção que implemente a interface **IEnumerable<T>**. Mas isso é apenas um exemplo, o método pode retornar qualquer objeto serializável. Veja alguns exemplos:

- **Retornando uma string**

```
public string Get()  
{  
    return "Olá WEB API";  
}
```

- **Retornando uma data**

```
public DateTime Get()  
{  
    return DateTime.Now;  
}
```

- **Retornando um valor booleano**

```
public Boolean Get()  
{  
    return false;  
}
```

- **Retornando um número inteiro**

```
public int Get()  
{  
    return 10;  
}
```

- **Retornando um DataTable**

```
public DataTable Get()  
{  
    var tb = new DataTable("agenda");  
    tb.Columns.Add("Nome");  
    tb.Columns.Add("Email");  
    tb.Rows.Add("Maria", "maria@teste.com.br");  
    return tb;  
}
```

8.3.2.1. Testando o método GET com parâmetros

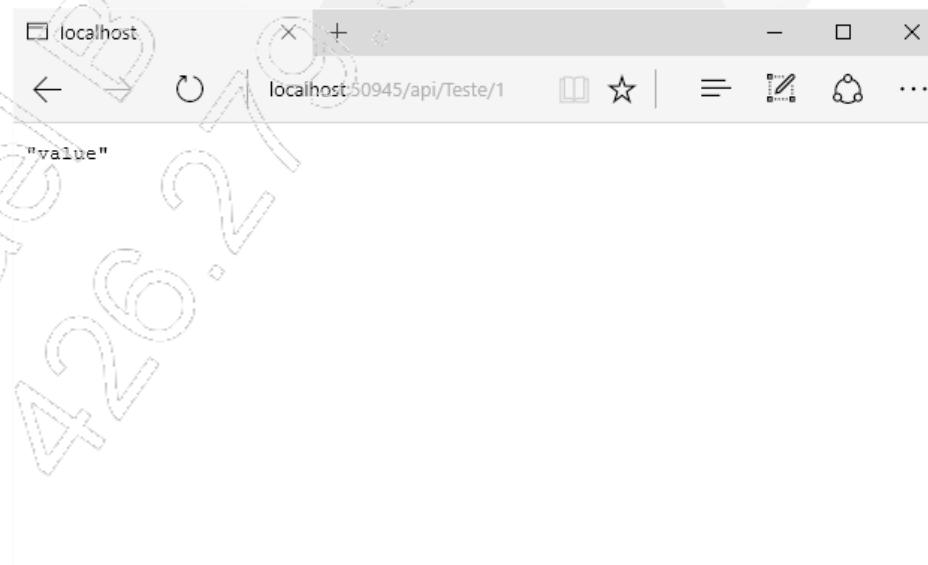
O método GET, na classe **TesteController**, apresenta uma sobrecarga que recebe um parâmetro:

```
public class TesteController : ApiController
{
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    public string Get(int id)
    {
        return "value";
    }

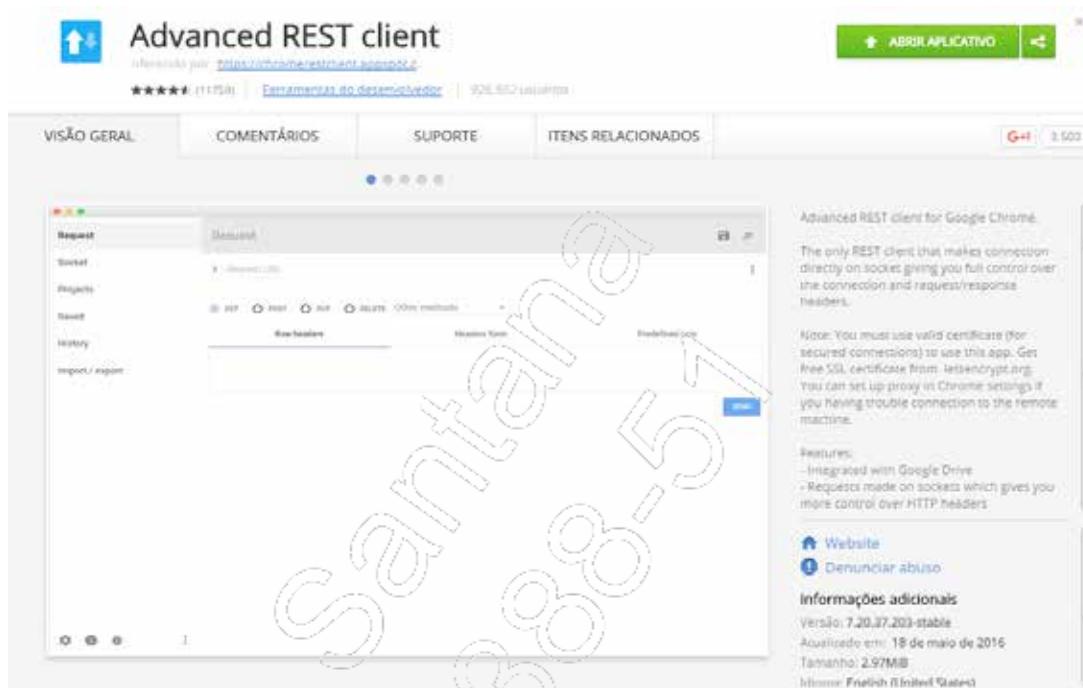
    public void Post([FromBody]string value) ...
    public void Put(int id, [FromBody]string value) ...
    public void Delete(int id) ...
}
```

Para chamar o segundo método, a URL deve fornecer um parâmetro: <http://servidor/api/teste/1>.

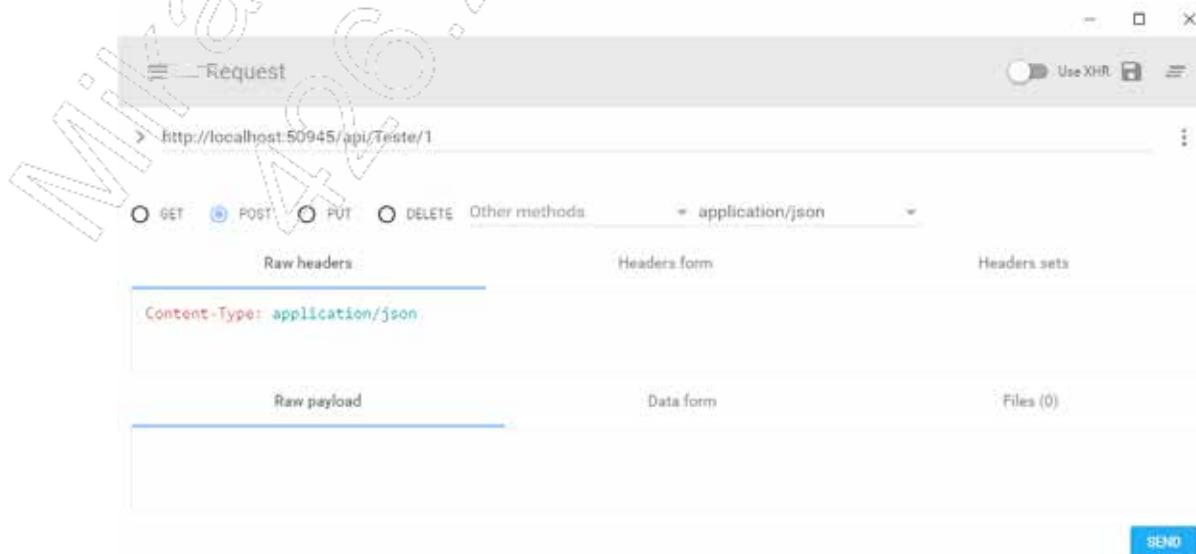


8.3.2.2. Testando o método POST

Para testar o método POST, 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**. No Chrome, existe um aplicativo chamado **Advanced REST client**. Esse aplicativo está citado aqui apenas como exemplo, mas qualquer programa do tipo **REST Client** terá o funcionamento similar.



A requisição (request) pode ser definida nos detalhes como URL, método, cabeçalhos, formulários, cookies ou metadados. O botão **SEND** envia a requisição:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

A resposta (response) pode ser analisada em todos os detalhes:

Status: 204: No Content ? Loading time: 16 ms			
Response headers (8)	Request headers (2)	Redirects (0)	Timings
Cache-Control: no-cache Pragma: no-cache Expires: -1 Server: Microsoft-IIS/10.0 X-Aspnet-Version: 4.0.30319 X-Sourcefiles: =?UTF-8?B?QzpcQVNQTkVUSUlcQ2FwaXR1bG8wOF9XZWJBcGlcQ2FwaXR1bG8wOF9XZWJBcGlcYXBpXFRIc3RlXDE=?= X-Powered-By: ASP.NET Date: Fri, 03 Jun 2016 15:23:08 GMT			

Uma das automações existentes na Web API é a conversão automática de dados vindos de um formulário diretamente em objetos. Alterando o método POST para receber um objeto, tornamos mais claro o poder de processamento da Web API. Considere a seguinte classe:

```
public class Usuario
{
    public string Nome { get; set; }
    public string Email { get; set; }
    public int Idade { get; set; }
}
```

Para usar o método POST e retornar uma string, utilizamos o seguinte código:

```
public string Post([FromBody] Usuario usuario)
{
    //Código para incluir um usuário....
    //ou outro processamento qualquer...

    return "OK";
}
```

O código a seguir cria um formulário em uma página HTML comum:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
</head>
<body>

    <h1>Cadastro de Usuário</h1>

    <form action="api/Teste" method="post">

        Nome: <br/>
        <input type="text" name="Nome" /> <br/><br />

        Email: <br />
        <input type="text" name="Email" /> <br /><br />

        Idade: <br />
        <input type="text" name="Idade" /> <br /><br />

        <input type="submit" value="Enviar" /> <br /><br />
    </form>

</body>
</html>
```

Repare a seguinte linha que define o destino do formulário e o verbo usado:

```
<form action="api/Teste" method="post">
```

As caixas de texto são nomeadas de acordo com as propriedades da classe usuário:

```
<input type="text" name="Nome" />
<input type="text" name="Email" />
<input type="text" name="Idade" />
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

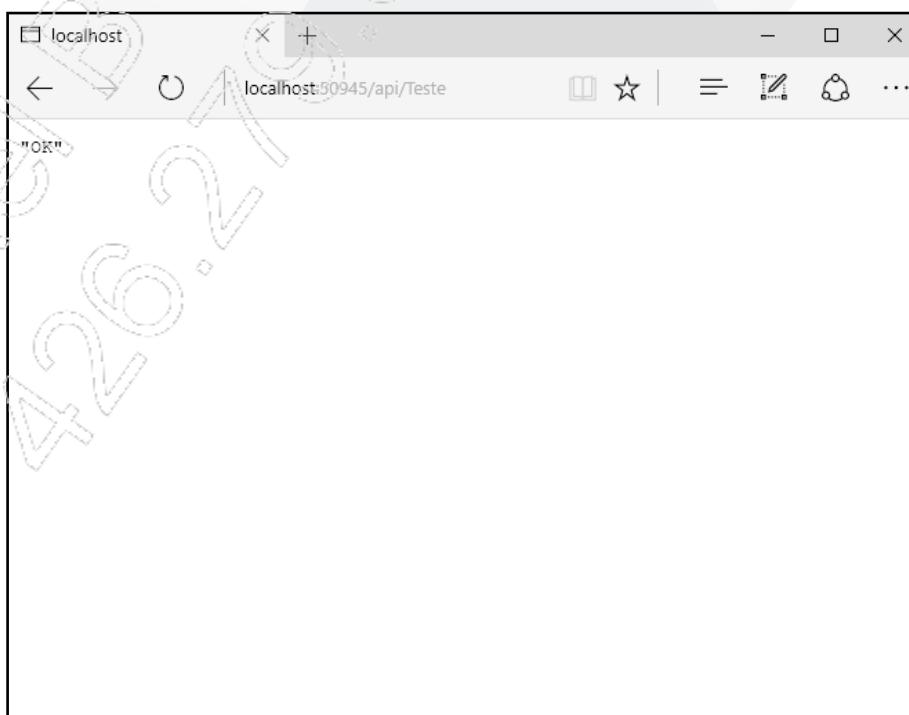
O botão que envia o formulário deve ser do tipo **submit**:

```
<input type="submit" value="Enviar" />
```

Esta é a tela de cadastro gerada pelo código HTML:

A screenshot of a web browser window titled "localhost". The address bar shows "localhost:50945/TesteHtml.html". The page content is titled "Cadastro de Usuário". It contains three input fields labeled "Nome:", "Email:", and "Idade:", each with a corresponding text input box. Below these fields is a button labeled "Enviar".

Confira o resultado, ao clicar no botão Enviar:



Os outros métodos da classe **Teste** seguem o mesmo padrão: o verbo usado determina o método a ser utilizado e a URL define o controller.

O método PUT deve ser usado para alterar os dados. A URL fica desta forma:

```
http://Servidor/api/Teste/1  
public void Put(int id, [FromBody]string value)  
{  
  
}
```

O método DELETE precisa apenas do ID para processar:

```
http://Servidor/api/Teste/1  
public void Delete(int id)  
{  
  
}
```

Repare que a URL é a mesma, o que muda é o verbo usado para fazer a chamada. O formato de saída de dados é escolhido pela aplicação cliente usando o cabeçalho **HTTP Accept**.

Neste exemplo, modificando o método PUT para retornar uma instância de usuário, é possível ver a atuação dos cabeçalhos HTTP:

```
public Usuario Put(int id, [FromBody]Usuario usuario)  
{  
    return usuario;  
}
```

8.3.3. Criando um cliente REST

A seguir, veremos os procedimentos para a criação de um cliente REST.

- **Usando HTML e jQuery**

Qualquer tipo de aplicativo que consiga realizar uma requisição HTTP pode utilizar um serviço REST. O exemplo a seguir usa o serviço **Teste** criado no exemplo anterior, por meio de um Web Form usando AJAX com a biblioteca jQuery.

Veja o conteúdo do Form:

```
<form id="form1" runat="server">

    <button type="button">Testar Servico</button>
    <br />
    <div id="mensagemDiv"></div>

</form>
```

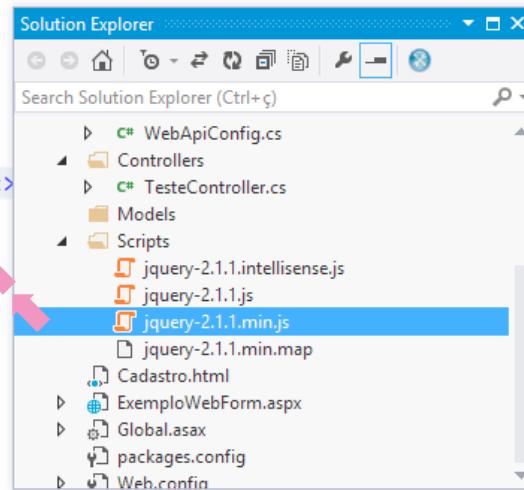
Então, adicionamos a biblioteca jQuery no formulário. Se não estiver com o arquivo jQuery disponível, é possível acrescentar usando NuGet:



Basta arrastar o jQuery do Solution Explorer para o formulário:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <script src="Scripts/jquery-2.1.1.min.js"></script>
</head>
<body>
    <form id="form1" runat="server">
        <button type="button">Testar Servico</button>
        <br />
        <div id="mensagemDiv"></div>
    </form>
</body>
</html>
```



Uma referência à biblioteca é adicionada na seção **Head** da página:

```
<head runat="server">
    <script src="Scripts/jquery-2.1.1.min.js"></script>
</head>
```

O código a seguir adiciona a chamada ao serviço **Teste** e exibe o retorno na div **mensagemDiv**:

```
<head runat="server">
    <script src="Scripts/jquery-2.1.1.min.js"></script>
    <script type="text/javascript">
        $(document).ready(iniciar);

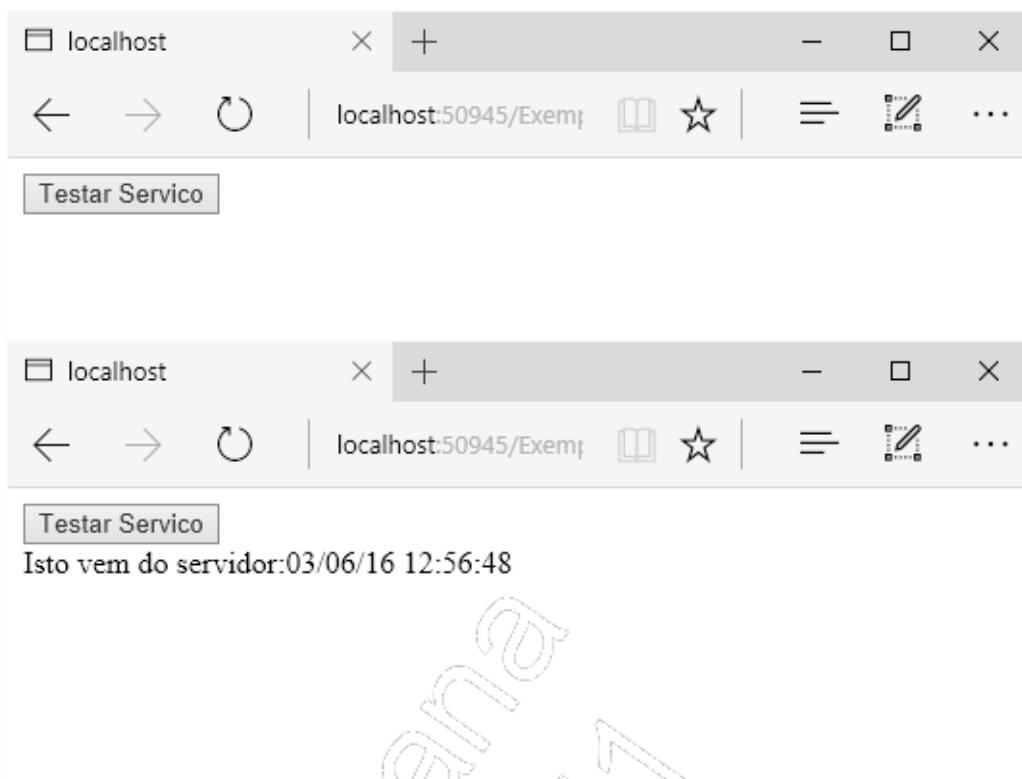
        function iniciar()
        {
            $('#testeButton').click(testeButtonClick);
        }

        function testeButtonClick() {
            $.getJSON("api/teste", respostaCallBack);
        }

        function respostaCallBack(dados) {
            $('#mensagemDiv').text(dados);
        }
    </script>
</head>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Confira o resultado:



A seguir, veja a explicação de cada linha do script:

```
$(document).ready(iniciar);
```

Esta linha chama a função **iniciar** quando o documento estiver carregado. O símbolo **\$(...)** é como o jQuery seleciona elementos. Neste caso, o elemento é o documento que o browser vai exibir. A variável **document** é parte dos objetos que o navegador disponibiliza para serem usados com scripts. A função **ready** executa uma função quando o **document** estiver carregado na memória. Neste caso, é a função **iniciar**:

```
function iniciar()
{
    $('#testeButton').click(testeButtonClick);
}
```

A função **iniciar** define uma função a ser executada no botão quando este for pressionado. A sintaxe '**#testeButton**' é como o jQuery localiza um elemento com base no ID. Neste caso, a declaração **\$('#testeButton')** vai retornar uma referência ao objeto **botão**, que é o único cujo ID é "**testeButton**".

Neste objeto retornado, o jQuery associa a função **testeButtonClick** ao evento **Click**:

```
function testeButtonClick() {  
    $.getJSON("api/teste", respostaCallBack);  
}
```

Esta função é disparada quando é clicado o botão cujo ID é "**testeButton**". A única tarefa desta função é realizar uma chamada AJAX, usando GET e obtendo o resultado no formato JSON, usando a URL "**api/teste**". A chamada é assíncrona (como toda chamada AJAX) e uma função chamada **respostaCallBack** é definida para receber o resultado do processamento. O método **getJSON** é autoexplicativo: faz uma chamada usando o verbo GET e obtém o resultado no formato JSON.

```
function respostaCallBack(dados) {  
    $('#mensagemDiv').text(dados);  
}
```

A função **respostaCallBack** recebe a resposta do servidor (apenas se não houve erro) e os dados da resposta são gravados no parâmetro que ela recebe (neste caso, chamado de **dados**). A resposta do servidor é uma string. A declaração **\$('#mensagemDiv')** seleciona o elemento cujo ID é "**mensagemDiv**" e insere o resultado da resposta do servidor dentro deste elemento, usando a função **text** do jQuery.

Existem muitas formas de escrever o código em jQuery. Neste exemplo, foram colocadas várias funções para melhorar a legibilidade. Na verdade, todo o código poderia ser escrito em uma única linha:

```
<script type="text/javascript">  
  
$(function(){ $('#testeButton').click(function () { $.getJSON("api/teste",  
    function (dados) { $('#mensagemDiv').text(dados) } );});  
  
</script>
```

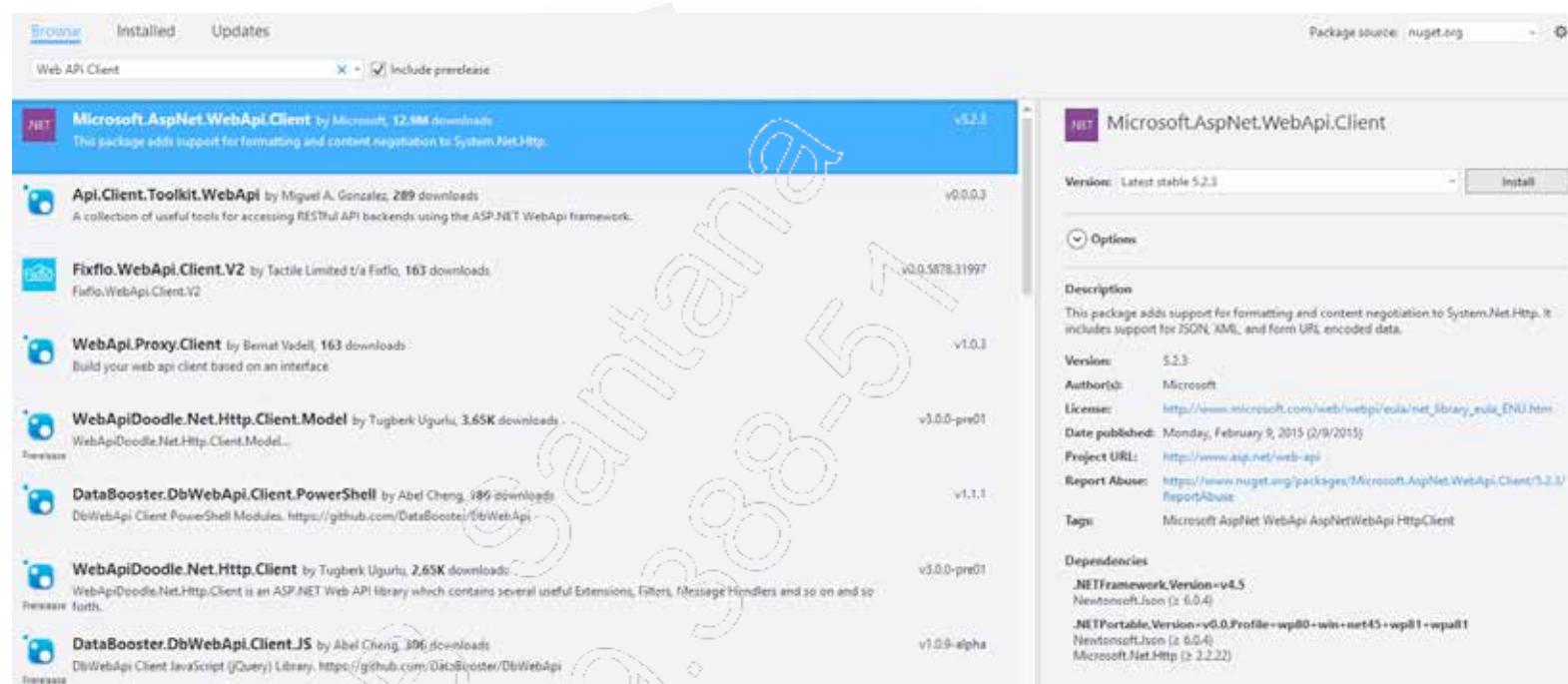
Apesar de realizar a mesma tarefa e encurtar o que deve ser escrito, abusar de funções anônimas pode prejudicar o entendimento do código.

Visual Studio 2015 - ASP.NET com C# Acesso a dados

- Criando um cliente REST via código

Para chamar um serviço REST diretamente do código de um aplicativo, é necessário realizar uma chamada HTTP. Isso pode ser feito manualmente, usando as classes que manipulam protocolos de rede, mas é muito mais fácil usando as bibliotecas **Web.API Client**.

O exemplo a seguir usa uma aplicação **Console** para realizar uma chamada ao serviço **Teste**. O primeiro passo é adicionar a biblioteca **Web.API Client** por meio do NuGet:



Na aplicação **Console**, é necessário realizar as chamadas de forma assíncrona, pois muitas das chamadas HTTP são realizadas dessa forma. Alguns namespaces são importantes para realizar chamadas Web por meio de programação. É necessário que o programa **Console** tenha as seguintes declarações no início do programa:

```
using System.Threading.Tasks;
using System.Net.Http;
using System.Net.Http.Headers;
```

- **System.Threading.Tasks** contém classes para realização de chamadas assíncronas;
- **System.net.Http** contém as classes para criar clientes HTTP;
- O namespace **System.Net.Http.Header** contém classes comuns para definir os cabeçalhos enviados nas chamadas.

O código a seguir obtém um valor string vindo do servidor:

```
class Program
{
    static void Main()
    {
        RunAsync().Wait();
    }

    static async Task RunAsync()
    {
        var formato =
            new MediaTypeWithQualityHeaderValue("application/json");

        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri("http://localhost:54361/");
            client.DefaultRequestHeaders.Accept.Clear();
            client.DefaultRequestHeaders.Accept.Add(formato);

            var resposta= await client.GetAsync("api/Teste");
            string conteudo = await
                resposta.Content.ReadAsAsync<string>();

            Console.WriteLine(conteudo);
        }
        Console.ReadLine();
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

Vamos analisar o código. O primeiro item a ser observado é o método `RunAsync()` criado para realizar todo o processo:

```
static async Task RunAsync()
{
}
```

Este método é chamado na entrada do programa, que fica esperando o seu término:

```
static void Main()
{
    RunAsync().Wait();
}
```

Para ficar mais claro o que está acontecendo, o método `Main` poderia ser escrito desta forma:

```
static void Main()
{
    Task t = RunAsync();
    t.Wait();
}
```

O método retorna uma instância da classe `Task`. O método `wait()` cria um bloqueio neste (e apenas neste) processo até que o método termine.

A primeira coisa que o método `RunAsync` faz é criar uma instância da classe `HttpClient`, tomando o cuidado de deixá-la dentro de um `using`:

```
using (var client = new HttpClient())
{
}
```

Em seguida, uma instância da classe **MediaTypeWithQualityHeaderValue** é criada. Essa instância é usada para adicionar informações no cabeçalho de uma solicitação HTTP (na prática, ela vai gerar um **Accept:application/json**):

```
using (var client = new HttpClient())
{
    var formato = new MediaTypeWithQualityHeaderValue("application/json");

}
```

Em seguida, é definido o endereço básico do serviço (sem as routes):

```
client.BaseAddress = new Uri("http://localhost:54361/");
```

Qualquer cabeçalho **Accept** presente é excluído e apenas o formato **application/json** é inserido. Isso elimina qualquer valor padrão assumido:

```
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(formato);
```

Aqui é onde tudo acontece! O método **GetAsync** é chamado e a resposta, em formato de stream, é retornada pelo servidor e fica sendo armazenada em um objeto do tipo **HttpResponseMessage**. A cláusula **await** espera o processo terminar:

```
var resposta= await client.GetAsync("api/Teste");
```

Quando acabar essa parte, a variável **resposta** terá uma instância da classe **HttpResponseMessage** e pode finalmente ser consultada.

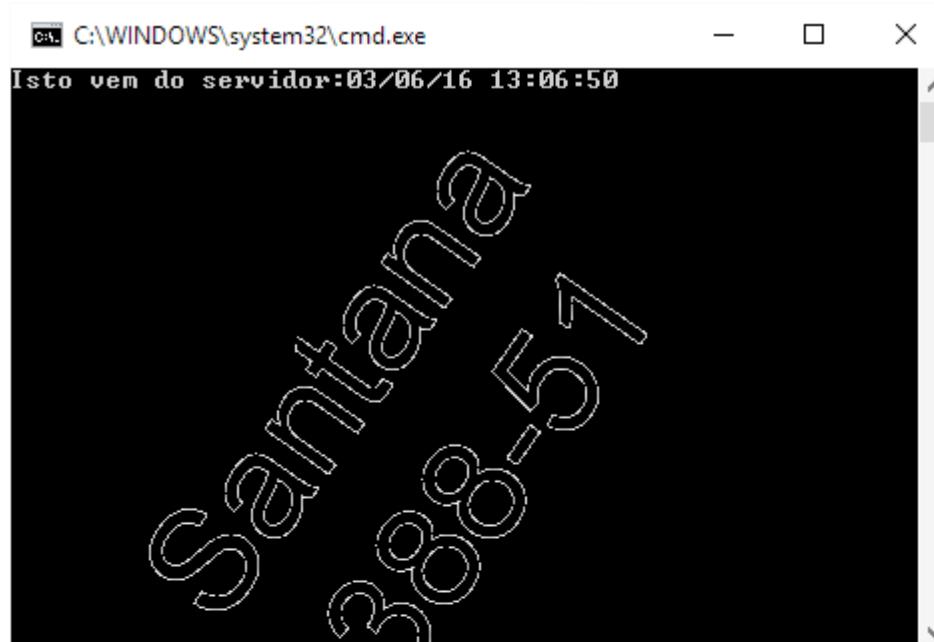
Como sabemos que o conteúdo retornado é uma string, basta chamar o método de extensão **ReadAsStringAsync<string>** da classe **Content**:

```
string conteudo = await resposta.Content.ReadAsStringAsync<string>();
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

O resultado é a string retornada pelo serviço. O próximo comando exibe a string e espera o usuário teclar algo para sair do programa **Console**:

```
Console.WriteLine(conteudo);
...
Console.ReadLine();
```



Qualquer outro tipo de retorno pode ser obtido usando a mesma técnica. O que varia de um sistema para outro é o tipo de dado que é retornado, o verbo, URL e conteúdo usado para realizar as chamadas. A biblioteca Web API é uma poderosa ferramenta para criar rapidamente serviços compatíveis com qualquer plataforma ou dispositivo. Segundo a própria definição da Microsoft: "ASP.NET Web API é um framework que torna fácil construir serviços HTTP que alcancem muitos tipos de clientes, incluindo browsers e dispositivos móveis. ASP.NET Web API é uma plataforma ideal para construir aplicações RESTful no .NET Framework."

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.

8

Serviços: Web API

Teste seus conhecimentos

Mikael B
426.279.3557



IMPACTA
EDITORA

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

8

Serviços: Web API

Mãos à obra!

Mikael B
426.279.57



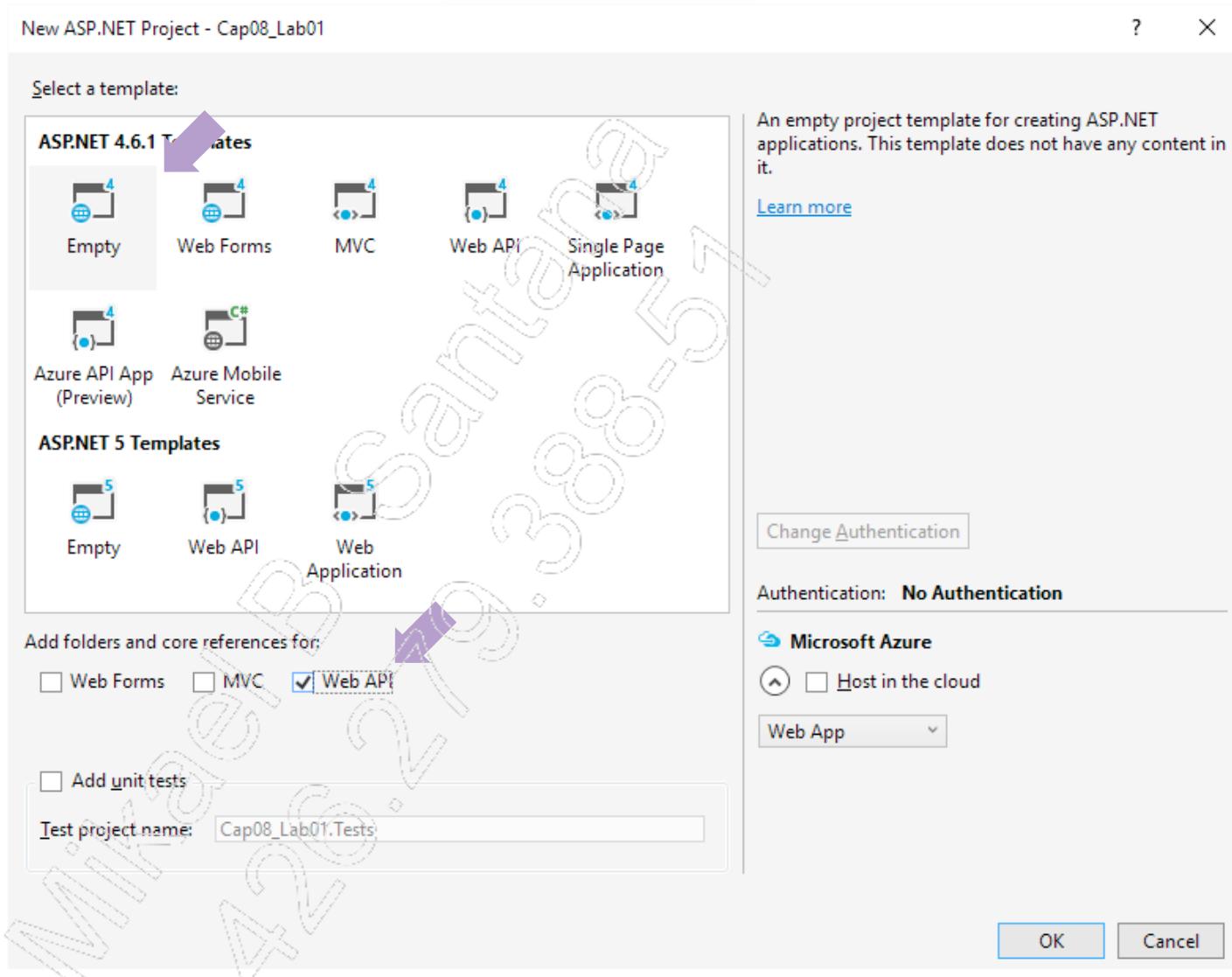
IMPACTA
EDITORA

Laboratório 1

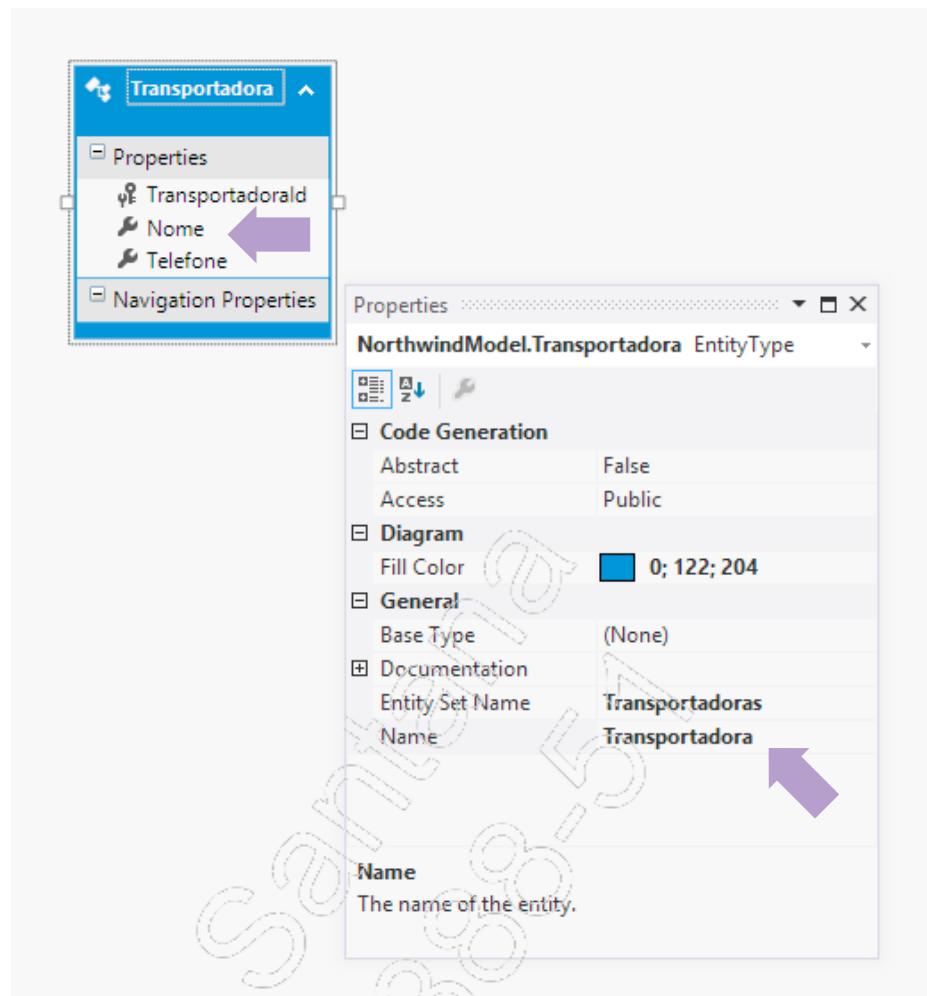
A - Criando um serviço REST

Neste laboratório, vamos criar um serviço REST para incluir, alterar, excluir e listar as transportadoras da empresa Northwind.

1. Crie um novo projeto Web vazio chamado **Cap08_Lab01**. Marque **Web API** nas opções de novo projeto:



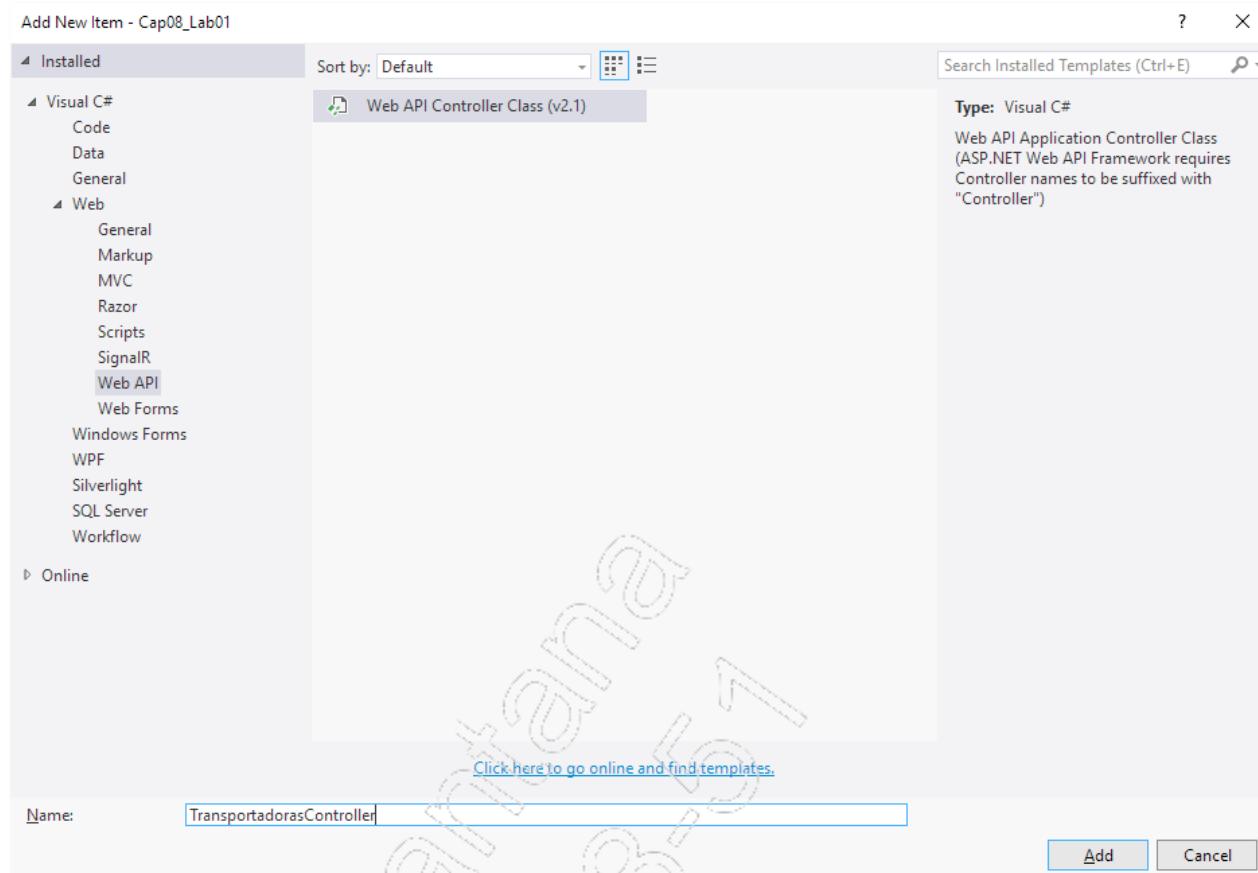
2. Na pasta **Models**, insira um novo **ADO.NET Entity Model** chamado **Northwind**. Conecte o **Northwind** e escolha apenas a tabela **Shippers**. Altere o nome dos campos, da entidade e do conjunto. Salve e compile:



Mikael B
426.270.
3300

Visual Studio 2015 - ASP.NET com C# Acesso a dados

3. Na pasta **Controllers**, insira um novo **Web API Controller Class** chamado **TransportadorasController**:



4. Na classe **TransportadorasController**, defina o método GET, que deverá retornar a lista das transportadoras:

```
// GET api/<controller>
public IEnumerable<Transportadora> Get()
{
    var db = new NorthwindEntities();
    return db.Transportadoras.ToList();
}
```

5. Teste o método chamando a URL <http://servidor/api/Transportadoras>:



```
localhost:52736/api/trans... localhost:52736/api/transportadoras
This XML file does not appear to have any style information associated with it. The document tree is as follows:
<ArrayOfTransportadora xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/Cap08_Lab01.Models">
  <Transportadora>
    <Nome>Speedy Express</Nome>
    <Telefone>(503) 555-9831</Telefone>
    <TransportadoraId>1</TransportadoraId>
  </Transportadora>
  <Transportadora>
    <Nome>United Package</Nome>
    <Telefone>(503) 555-3199</Telefone>
    <TransportadoraId>2</TransportadoraId>
  </Transportadora>
  <Transportadora>
    <Nome>Federal Shipping</Nome>
    <Telefone>(503) 555-9931</Telefone>
    <TransportadoraId>3</TransportadoraId>
  </Transportadora>
</ArrayOfTransportadora>
```

6. Defina o método GET com parâmetro, que deve retornar uma transportadora e faça um teste:

```
// GET api/<controller>/5
public Transportadora Get(int id)
{
    var db = new NorthwindEntities();
    return db.Transportadoras.Find(id);
}
```

7. Teste o método chamando a URL <http://servidor/api/Transportadoras/id>:



```
localhost:52736/api/trans... IIS 10.0 Detailed Error - 404 ...
localhost:52736/api/transportadoras/3
This XML file does not appear to have any style information associated with it. The document tree is as follows:
<Transportadora xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/Cap08_Lab01.Models">
  <Nome>Federal Shipping</Nome>
  <Telefone>(503) 555-9931</Telefone>
  <TransportadoraId>3</TransportadoraId>
</Transportadora>
```

8. Defina o método POST, que adiciona uma nova transportadora. Só será possível testar esse método criando uma aplicação cliente ou usando um programa como PostMan, Fiddle ou TestUnit. Deixaremos para testar quando criar a aplicação cliente:

```
// POST api/<controller>
public void Post([FromBody]Transportadora t)
{
    var db = new NorthwindEntities();
    db.Transportadoras.Add(t);
    db.SaveChanges();
}
```

9. Defina o método PUT, que altera uma transportadora:

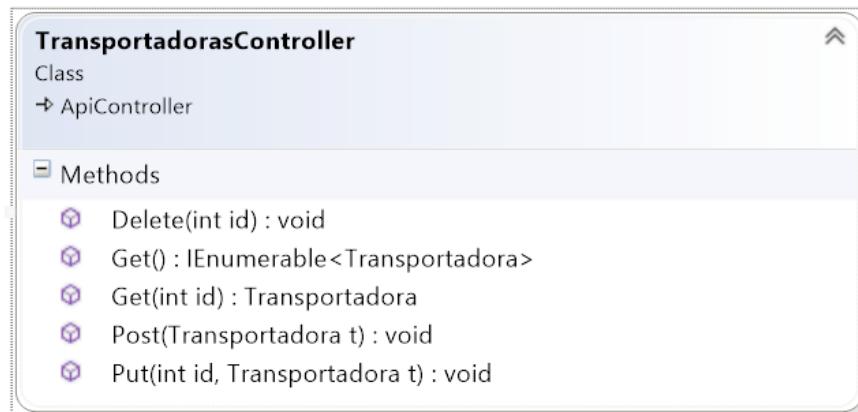
```
// PUT api/<controller>/5
public void Put(int id, [FromBody]Transportadora t)
{
    var db = new NorthwindEntities();
    var original = db.Transportadoras.Find(id);
    original.Nome = t.Nome;
    original.Telefone = t.Telefone;
    db.SaveChanges();

}
```

10. Defina o método DEL, que exclui transportadora:

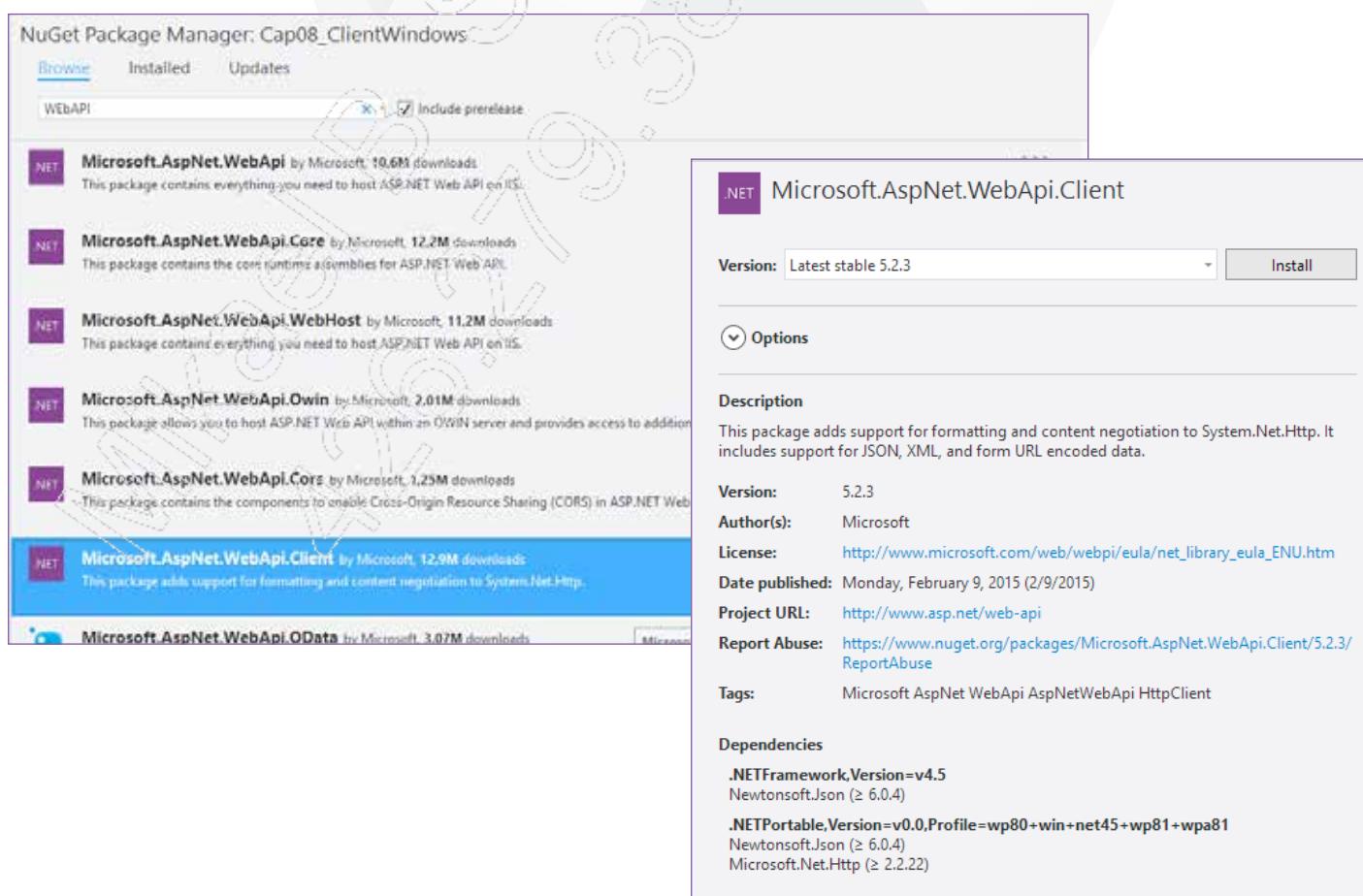
```
// DELETE api/<controller>/5
public void Delete(int id)
{
    var db = new NorthwindEntities();
    var original = db.Transportadoras.Find(id);
    db.Transportadoras.Remove(original);
    db.SaveChanges();
}
```

A API está completa! Os seguintes métodos foram criados:



B – Criando a aplicação cliente

1. Adicione, na solução, um projeto do tipo Windows Forms (qualquer tipo de aplicativo pode ser **client** desse serviço);
2. No projeto Windows, usando NuGet, adicione as bibliotecas **Web API Client**:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

3. No form principal, crie a seguinte tela usando os controles **DataGridView** e **Buttons**:



4. Adicione uma classe chamada **Transportadora** com a mesma estrutura da classe do serviço. Existem outras maneiras de usar os modelos, por exemplo, separando em biblioteca de classes distinta do serviço, mas, para manter a simplicidade e focar no principal do exemplo, que é a chamada ao serviço, vamos criar uma cópia da classe:

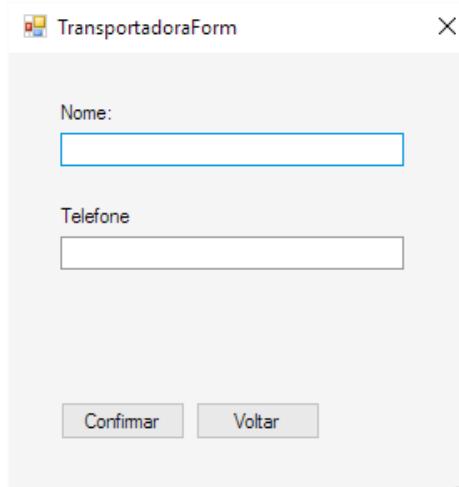
```
public class Transportadora
{
    public int TransportadoraId { get; set; }
    public string Nome { get; set; }
    public string Telefone { get; set; }
}
```

5. Adicione um **Form** chamado **TransportadoraForm**, que será responsável por exibir e retornar uma instância da classe **Transportadora**;

Use **Labels**, **TextBoxes** e **Buttons** para criar a tela.

A propriedade **DialogResult** do botão **Confirmar** deve ser **OK** e este deve ser o **AcceptButton** do formulário.

O botão voltar é o **CancelButton** do formulário:



6. O formulário **TransportadoraForm** contém dois métodos: um para exibir uma transportadora e outro para retornar os dados de uma transportadora que está na tela. O campo **private transportadoraId** armazena o ID da transportadora, pois este não aparece na tela:

```
public partial class TransportadoraForm : Form
{
    public TransportadoraForm()
    {
        InitializeComponent();
    }

    private int transportadoraId = 0;

    public void ExibirTransportadora(Transportadora t)
    {
        nomeTextBox.Text = t.Nome;
        telefoneTextBox.Text = t.Telefone;
        transportadoraId = t.TransportadoraId;
    }

    public Transportadora ObterTransportadora()
    {
        var t = new Transportadora();
        t.TransportadoraId = transportadoraId;
        t.Nome = nomeTextBox.Text;
        t.Telefone = telefoneTextBox.Text;
        return t;
    }
}
```

}

7. De volta ao **form** principal (**Form1**), crie um método para retornar uma instância de **HttpClient**, que é a classe usada para realizar a comunicação com o serviço REST:

```
//  
// Obtém uma instância de HttpClient  
// já configurado com o formato e a url do servidor  
//  
private HttpClient ObterHttpClient()  
{  
    var formato =  
        new MediaTypeWithQualityHeaderValue(  
            "application/json");  
  
    var client = new HttpClient();  
  
    client.BaseAddress =  
        new Uri("http://localhost:50090/");  
  
    client.DefaultRequestHeaders.Accept.Clear();  
  
    client.DefaultRequestHeaders  
        .Accept  
        .Add(formato);  
  
    return client;  
}
```

8. Crie outro método para validar a resposta do serviço:

```
//  
// Analisa se o comando foi bem sucedido  
//  
private void verificarResposta(  
    HttpResponseMessage resposta)  
{  
    if (!resposta.IsSuccessStatusCode)  
    {  
        MessageBox.Show("Erro no servidor:" +  
            resposta.StatusCode);  
    }  
}
```

9. Crie um método para carregar a grid com os dados do serviço:

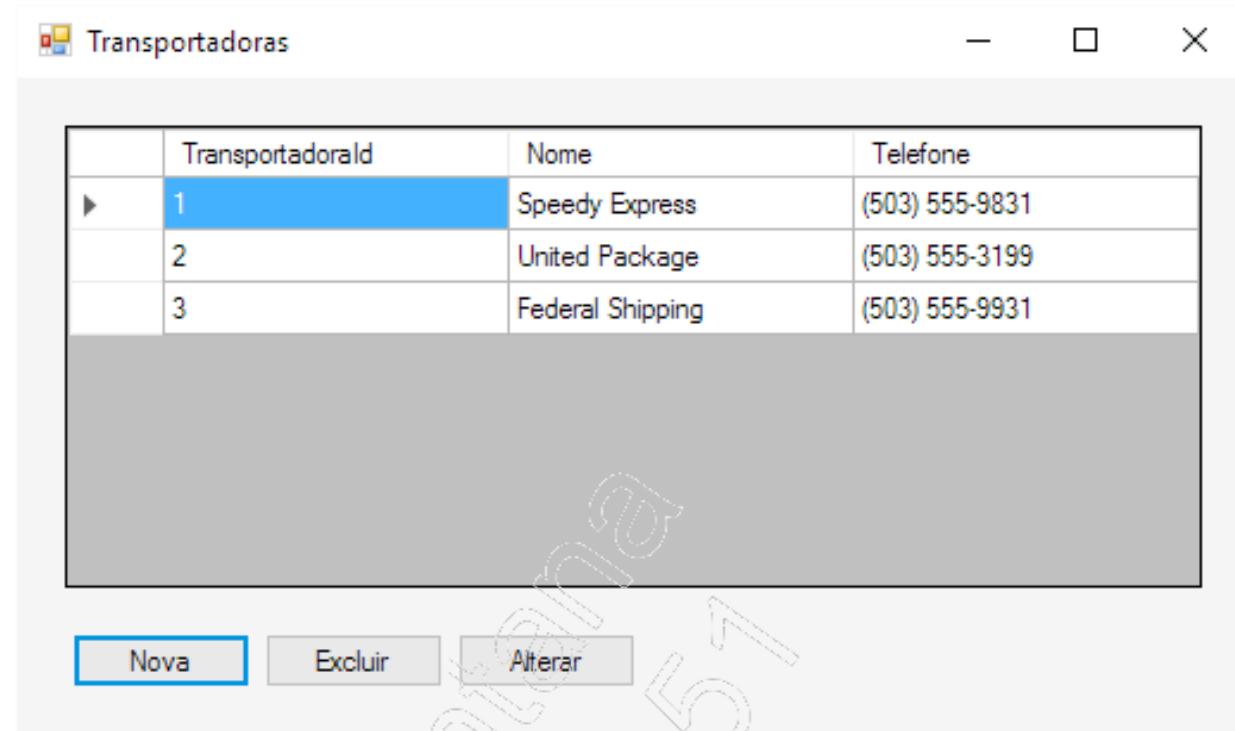
```
//  
// GET: Obtém todas as transportadoras  
//  
private async void CarregarGrid()  
{  
  
    using (var client = ObterHttpClient())  
    {  
        var resposta = await client  
            .GetAsync("api/Transportadoras");  
  
        var conteudo = await resposta  
            .Content  
            .ReadAsAsync<Transportadora[]>();  
  
        dataGridView1.DataSource = conteudo;  
        dataGridView1.ReadOnly = true;  
        dataGridView1.AutoSizeColumnsMode =  
            DataGridViewAutoSizeColumnsMode.Fill;  
    }  
}
```

10. No evento **Load** do formulário, chame o seguinte método:

```
//  
// Load  
//  
private void Form1_Load(object sender, EventArgs e)  
{  
    CarregarGrid();  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

11. Teste o programa. Deve aparecer a listagem. Lembre-se de definir o projeto **Windows** como sendo o **StartUp Project** e de que serviço deve estar no ar (podemos incluir uma página HTML no projeto do servidor e usar "**View in Browser**"):



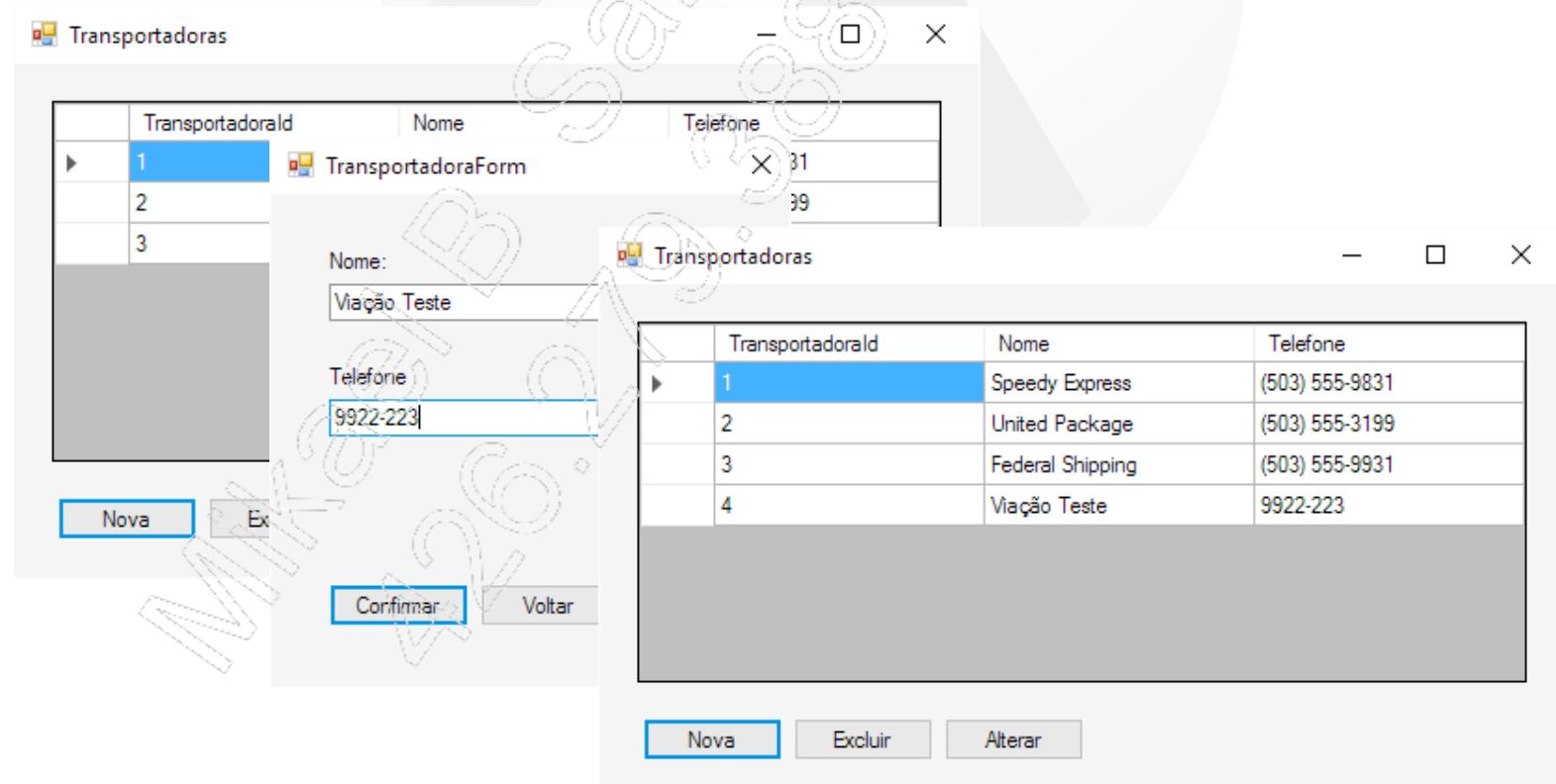
12. Escreva o método para incluir uma transportadora. Esse método deve chamar o serviço e passar os dados da transportadora por meio de POST:

```
//  
// POST: Adiciona uma nota Transportadora  
//  
private async void IncluirTransportadora(Transportadora t)  
{  
    using (var client = ObterHttpClient())  
    {  
        var resposta =  
            await client.PostAsJsonAsync<Transportadora>  
                ("api/Transportadoras", t);  
  
        verificarResposta(resposta);  
    }  
  
    CarregarGrid();  
}
```

13. Inclua o método do evento **novoButton_Click**. Esse método deve exibir o formulário para preenchimento e retornar uma instância de **Transportadora**:

```
//  
// Incluir Transportadora  
  
private void novoButton_Click(object sender, EventArgs e)  
{  
    var f = new TransportadoraForm();  
  
    if (f.ShowDialog() == DialogResult.OK)  
    {  
        var t = f.ObterTransportadora();  
        IncluirTransportadora(t);  
    }  
}
```

14. Teste a inclusão:



Visual Studio 2015 - ASP.NET com C# Acesso a dados

15. Inclua o método do evento **ObterTransportadoraSelecionada** que retorna a transportadora selecionada na **DataGridview** ou **Null** se não houver nenhuma selecionada:

```
//  
// Obter a transportadora selecionada da Grid  
//  
private Transportadora ObterTransportadoraSelecionada()  
{  
    if ( dataGridView1.DataSource != null &&  
        dataGridView1.CurrentRow != null &&  
        dataGridView1.CurrentRow.DataBoundItem is  
            Transportadora)  
    {  
        return (Transportadora)dataGridView1  
            .CurrentRow.DataBoundItem;  
    }  
    else  
    {  
        return null;  
    }  
}
```

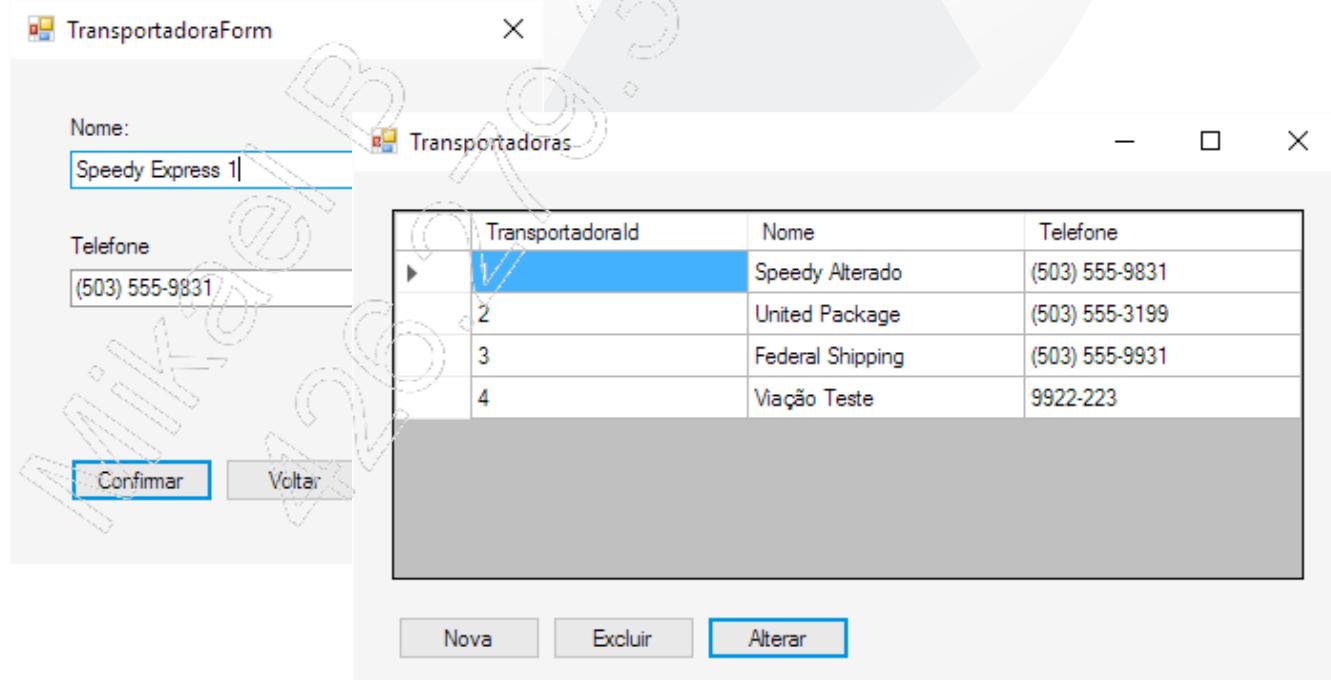
16. Inclua o método **Alterar** que chama o **Servico** para alterar dados:

```
//  
// PUT: Alterar  
//  
private async void Alterar(Transportadora t)  
{  
    using (var client = ObterHttpClient())  
    {  
        var resposta =  
            await client.PutAsJsonAsync<Transportadora>(  
                "api/Transportadoras/" +  
                t.TransportadoraId, t);  
    }  
    CarregarGrid();  
}
```

17. Inclua o método do evento **AlterarButton_Click** que chama o **Servico** para alterar dados:

```
//  
// Alterar os dados da transportadora  
  
private void alterarButton_Click(object sender, EventArgs e)  
{  
    var t = ObterTransportadoraSelecionada();  
    if (t == null) return;  
    var f = new TransportadoraForm();  
    f.ExibirTransportadora(t);  
    if (f.ShowDialog() == DialogResult.OK)  
    {  
        t = f.ObterTransportadora();  
        Alterar(t);  
    }  
}
```

18. Teste a alteração:



19. Crie o método **Pergunta** que retorna o resultado de uma MessageBox:

```
//  
// Retorna true ou false  
//  
private bool Pergunta(string msg)  
{  
    var result = MessageBox.Show(  
        msg,  
        "Confirmação",  
        MessageBoxButtons.OKCancel,  
        MessageBoxIcon.Question);  
  
    return result == DialogResult.OK;  
}
```

20. Crie o método **Excluir** que chama o serviço para excluir uma transportadora:

```
//  
// DEL: Excluir uma transportadora  
//  
private async void Excluir(int id)  
{  
    using (var client = ObterHttpClient())  
    {  
        var resposta =  
            await client.DeleteAsync(  
                "api/Transportadora/" + id);  
  
        verificarResposta(resposta);  
    }  
  
    CarregarGrid();  
  
}
```

21. Finalmente, crie o método do evento **ExcluirButton_Click**, que confirma a operação e faz chamada o método **Excluir**:

```
//  
// DEL: Exclui uma transportadora  
  
private void excluirButton_Click(object sender, EventArgs e)  
{  
  
    var t = ObterTransportadoraSelecionada();  
  
    if (t != null)  
    {  
        if (Pergunta("Confirma a exclusão"))  
        {  
            Excluir(t.TransportadoraId);  
        }  
    }  
}
```

22. Teste o programa inteiro: inclusão, alteração e exclusão.

Sugestões para estudo:

- Fazer o mesmo exercício com a classe **Suppliers**;
- Fazer um cliente Web usando Web Forms com as mesmas funcionalidades desse exemplo em Windows. É interessante ver as adaptações de interface de usuário que são necessárias;
- Se conseguir fazer com a classe **Suppliers**, tentar também com a classe **Employee**. Existe o tratamento da imagem que deve ser adaptado, usando o que foi aprendido nos capítulos anteriores.

9

NoSQL

- ✓ Modelo relacional;
- ✓ NoSQL;
- ✓ Bancos de dados NoSQL.



IMPACTA
EDITORA

9.1. Modelo relacional

Bancos de dados relacionais são aqueles em que a informação é organizada em estruturas fixas e que podem ser interligadas. O modelo relacional se mostra muito eficiente ao agrupar e consolidar informações, eliminando redundâncias e mantendo a integridade dos dados. A linguagem SQL, que é adotada por praticamente todos os bancos de dados relacionais, é muito eficiente ao agrupar e consolidar informações que estão fragmentadas.

Este modelo, no entanto, apresenta algumas limitações e não é adequado para algumas situações específicas. A principal limitação é a exigência de uma estrutura predefinida.

No modelo relacional, uma vez criada uma tabela e definida sua estrutura, não é possível adicionar novas informações sem redefinir a estrutura. Se quisermos armazenar informações de clientes, como **Nome** e **Email**, não há como inserir um registro com uma informação a mais, como um **Telefone**, depois que a estrutura foi definida. A única maneira de fazer isso é alterando a estrutura da tabela e acrescentando um campo para o armazenamento do telefone. Se apenas um cliente precisa dessa informação e centenas de outros não, todos os que não precisam são obrigados a reservar um espaço para o telefone – espaço este que sempre estará vazio ou nulo.

A solução poderia ser criar uma outra tabela contendo um campo que identifique um cliente e um campo para armazenar o telefone. Apenas os clientes que tenham telefone estarão relacionados nessa outra tabela. Essa solução funciona, mas cria uma complexidade muitas vezes desnecessária. Por exemplo, a elaboração de uma listagem simples como **Nome**, **Email** e **Telefone** exigiria uma expressão SQL que utilize JOINS contendo a declaração de qual campo é usado para estabelecer a relação entre essas tabelas.

Um modelo simples como um pedido de venda contendo o número da nota, a data, os dados do cliente, a lista de produtos, os dados do pagamento e o local de entrega pode exigir cerca de 10 tabelas diferentes: **Pedido**, **Cliente**, **Produto**, **PedidoProduto**, **Estado**, **Fatura**, **FaturaDetalhes**, **ClienteEnderecos**, **FaturaFormaPagamento**, **ProdutoImposto**, etc.

9.2. NoSQL

Bancos de dados chamados **NoSQL** são aqueles que não precisam de uma estrutura fixa para armazenar dados. Não existem tabelas, campos, relacionamentos... Apenas a informação final é armazenada, na forma de um documento.

O autor do termo NoSQL foi o italiano Carlo Strozzi, que, em 1998, estava desenvolvendo um gerenciador de banco de dados. Na época, quase todos os gerenciadores de banco de dados tinham a palavra SQL no nome: MSSQL, MySQL, PostgreSQL... O termo NoSQL se referia ao fato de que esse software não era mais um desses do mercado. Uma ironia da história é que esse gerenciador de banco de dados era uma banco de dados relacional. O termo passou a definir bancos de dados sem estrutura somente mais tarde (2009), quando outros desenvolvedores, como Eric Evans, passaram a usar o termo para designar bancos de dados que não utilizam o conceito de tabelas e relacionamentos.

Um pedido de venda, por exemplo, teria um campo para o **Número do Pedido**, um para a **Data**, um para os **Dados do Cliente**, outro para a **Lista de Produtos**, e assim por diante. Todas as informações necessárias para aquele pedido estariam gravadas no **Documento**. Outro registro de **Pedido** poderia conter as mesmas informações básicas e mais um campo para armazenar outra informação, como os dados do vendedor.

À primeira vista, pode parecer que um banco sem estrutura definida pode virar um conjunto de dados embaralhados, redundantes, sem consistência. E isso pode acontecer, sem dúvida. Por essa razão, o conceito NoSQL não serve para todo o tipo de aplicação, principalmente as que precisam agrupar e totalizar dados, assim como o modelo relacional não funciona para aplicações em que a informação é dinâmica e a estrutura não pode ser prevista.

São tipos de aplicações em que o **modelo NoSQL** é muito utilizado: blogs, sites de notícias, sites para advogados e informações de processos, sistemas educacionais, escolas, sites de busca, redes sociais...

São tipos de aplicações em que o **modelo relacional** é muito utilizado: sistemas de gestão, comércio, sistemas administrativos, folhas de pagamento, gerenciamento de projetos...

Visual Studio 2015 - ASP.NET com C# Acesso a dados

A seguir, vemos uma comparação de um banco de dados relacional com um banco de dados NoSQL:

- **Banco de dados relacional**

Produtoid	Nome	FornecedorId	Categoriald
1	Notebook	23	2
2	Livro	2	4

FornecedorId	Nome
2	Editora ABC
23	Empresa XYZ

Categoriald	Nome
2	Informática
4	Impressos

- **Banco de dados NoSQL**

```
{  
    ProdutoId: 2,  
    Nome: Livro,  
    Fornecedor: Editora ABC,  
    Categoria: Impressos  
},  
{  
    ProdutoId:1,  
    Nome: Notebook,  
    Fornecedor: Empresa XYZ,  
    Categoria: Informática  
}
```

9.3. Bancos de dados NoSQL

Existe uma variedade enorme de bancos de dados não relacionais disponíveis. Grande parte deles é open source. A lista a seguir enumera os mais utilizados:

- **MongoDB;**
- **RavenDB;**
- **Cassandra;**
- **Db4o;**
- **Caché;**
- **Amazon SimpleDB;**
- **Google Cloud Data;**
- **DocumentDB – Azure;**
- **Google LevelDB;**
- **VelocityDB.**

A forma como cada banco de dados armazena as informações pode variar. Com poucas exceções, qualquer banco NoSQL organiza os dados usando um dos seguintes modelos:

- **Chave/Valor:** Neste modelo, os dados são armazenados contendo duas informações: uma chave para acesso aos dados e o valor, que pode ser qualquer informação. É parecido com a estrutura de um dicionário. São exemplos de bancos deste tipo: SimpleDB, Voldemort e Oracle NoSQL;
- **Documento:** Este é o modelo mais famoso. Cada informação é um documento que possui campos e valores, sem uma estrutura fixa. Os documentos podem ser agrupados em coleções. Exemplos: RavenDB, MongoDB, IBM Lotus Domino;

- **Colunas/Tabular:** Este modelo é parecido com um banco de dados relacional. Os dados são organizados em colunas e agrupados em famílias que, por sua vez, são agrupadas em keyspaces. A maneira como os dados são organizados facilita a escalabilidade. A popularidade deste modelo se deve ao projeto BigTable do Google, um dos primeiros a usar NoSQL com uma documentação técnica detalhada. Exemplos: Cassandra, Hypertable, Google BigTable;
- **Grafo:** Este modelo cria redes de relacionamentos de informação, vinculando as informações por meio de características comuns. Exemplos: OrientDB, VelocityGraph, GraphDB;
- **Orientado a objetos:** Este utiliza o mesmo conceito de classes, herança e tipos encontrado na programação orientada a objetos. Exemplos: db4o, Versant, Caché.

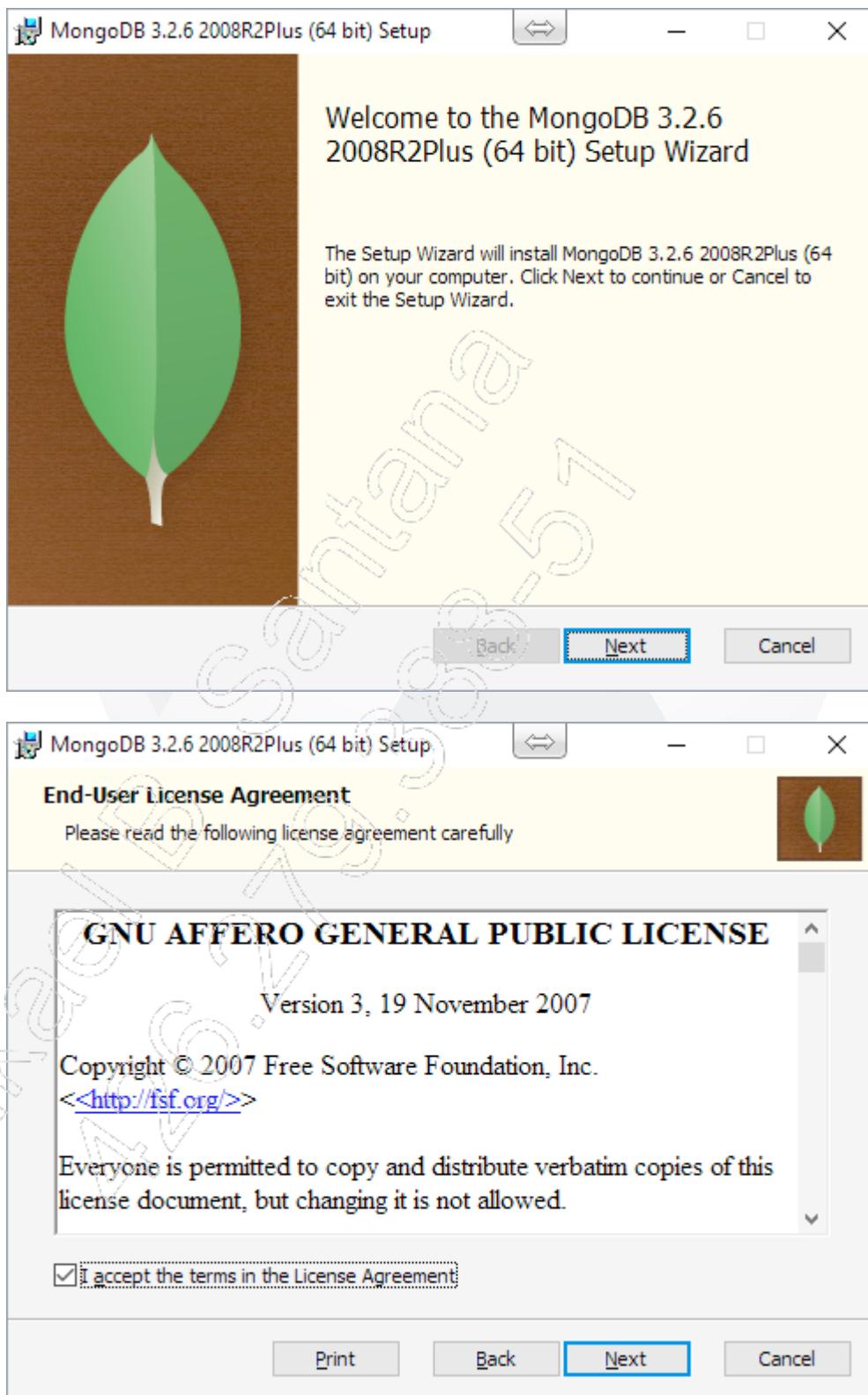
9.3.1. MongoDB

Neste exemplo, será usado o banco de dados MongoDB. Várias características deste banco de dados são interessantes para quem trabalha na plataforma .NET:

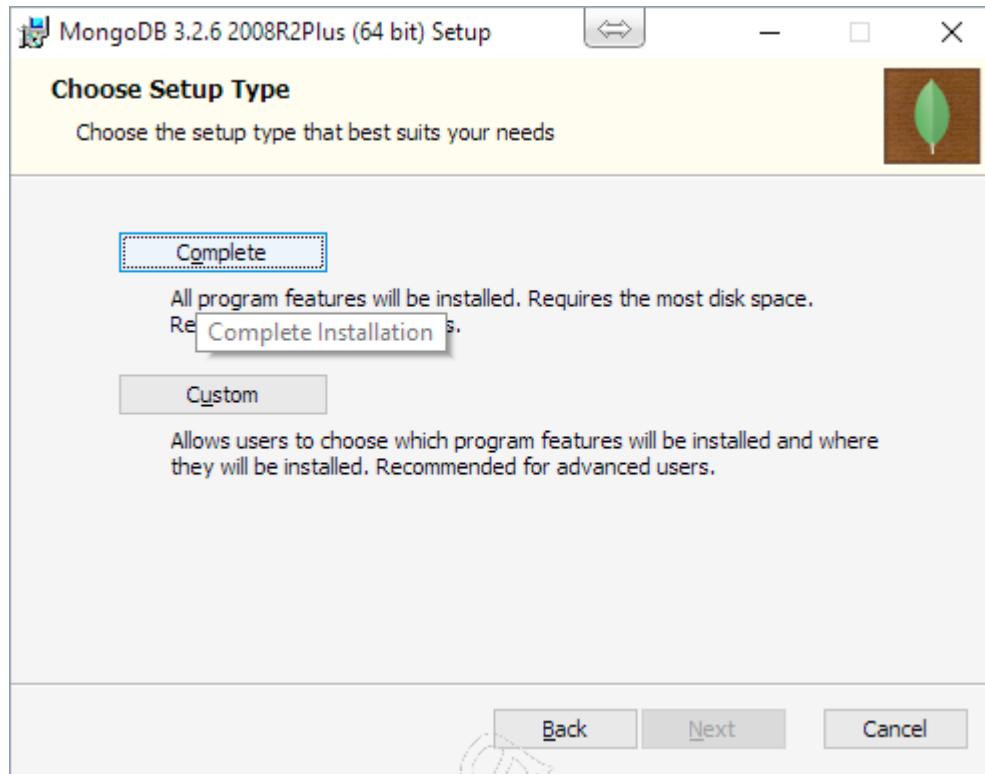
- Open source;
- Escrito em C++ (ótima performance);
- É o mais conhecido NoSQL open source;
- Amplamente utilizado;
- Instalável via NuGet.

9.3.2. Instalando o servidor

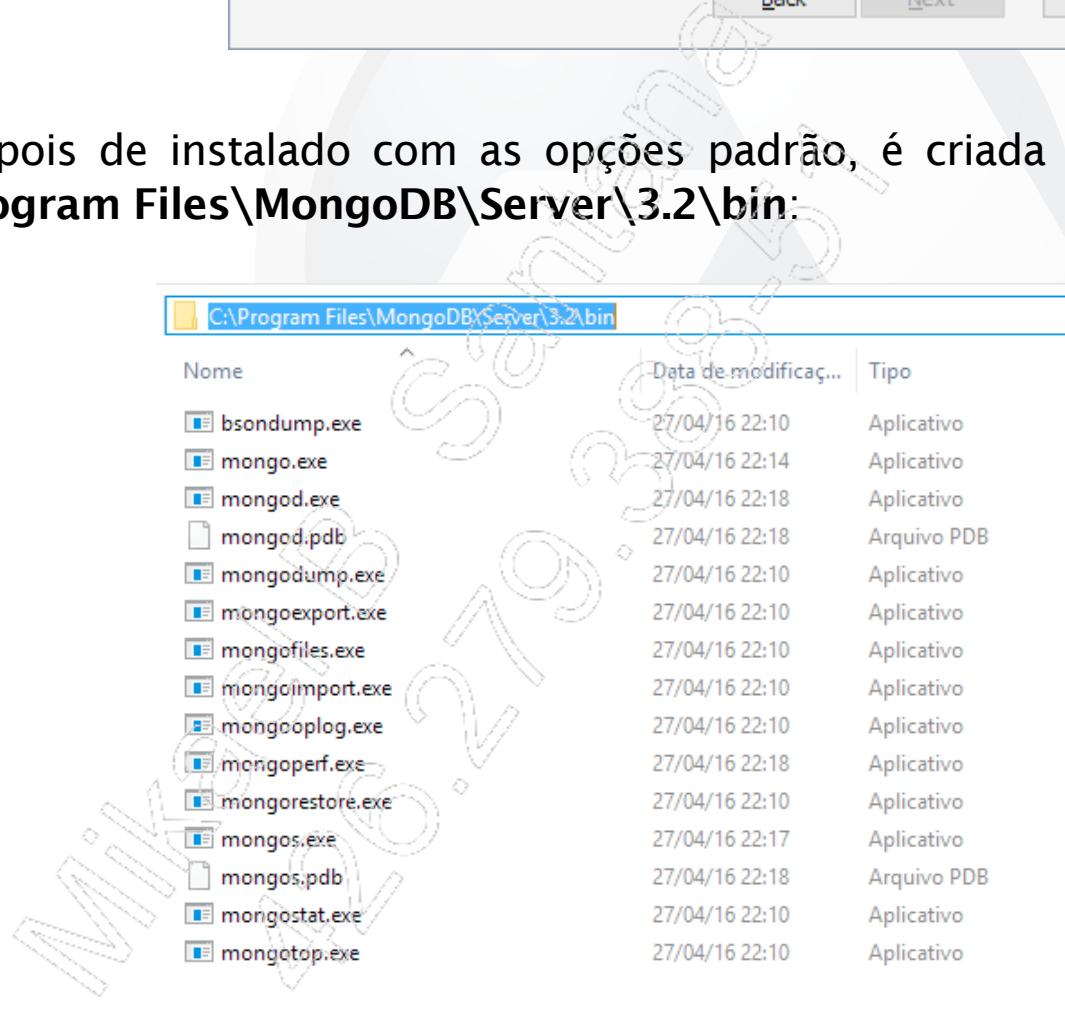
Para utilizar o MongoDB, é necessário instalar o banco de dados. O instalador Windows pode ser obtido no site oficial: <http://www.MongoDB.org>. A instalação é simples, por meio de um assistente:



Visual Studio 2015 - ASP.NET com C# Acesso a dados



Depois de instalado com as opções padrão, é criada a seguinte pasta: **C:\Program Files\MongoDB\Server\3.2\bin**:

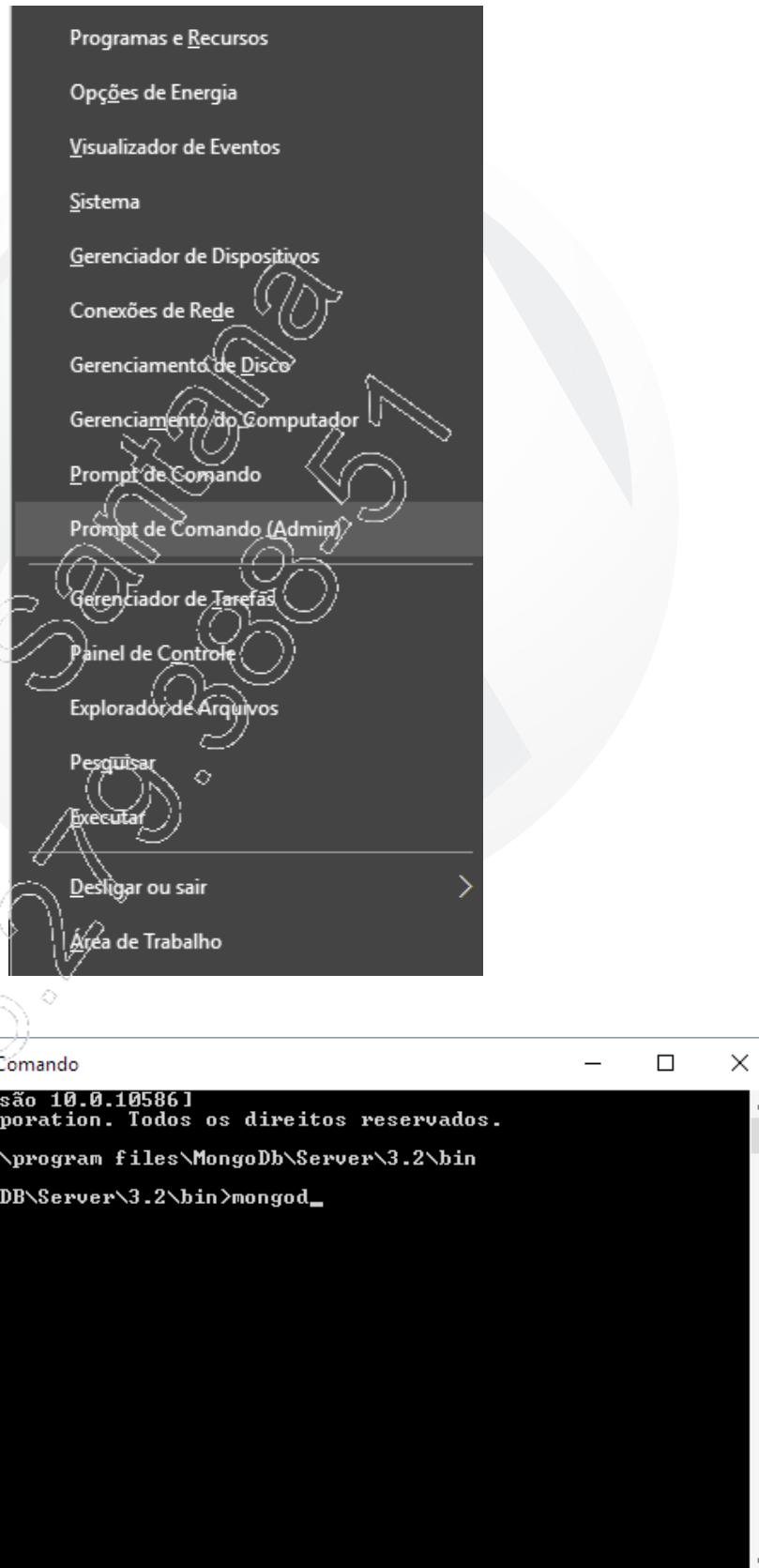


Nome	Data de modificação...	Tipo	Tamanho
bsondump.exe	27/04/16 22:10	Aplicativo	4.665 KB
mongo.exe	27/04/16 22:14	Aplicativo	9.787 KB
mongod.exe	27/04/16 22:18	Aplicativo	19.334 KB
mongod.pdb	27/04/16 22:18	Arquivo PDB	158.356 KB
mongodump.exe	27/04/16 22:10	Aplicativo	26.350 KB
mongoexport.exe	27/04/16 22:10	Aplicativo	6.387 KB
mongofiles.exe	27/04/16 22:10	Aplicativo	6.234 KB
mongoimport.exe	27/04/16 22:10	Aplicativo	6.507 KB
mongoclog.exe	27/04/16 22:10	Aplicativo	5.960 KB
mongoperf.exe	27/04/16 22:18	Aplicativo	16.678 KB
mongorestore.exe	27/04/16 22:10	Aplicativo	44.918 KB
mongos.exe	27/04/16 22:17	Aplicativo	8.154 KB
mongos.pdb	27/04/16 22:18	Arquivo PDB	85.724 KB
mongostat.exe	27/04/16 22:10	Aplicativo	6.184 KB
mongotop.exe	27/04/16 22:10	Aplicativo	6.047 KB

Depois da instalação, o próximo passo é criar uma pasta onde ficarão os arquivos do banco de dados. Por padrão, essa pasta tem o seguinte endereço: **C:\data\db**.

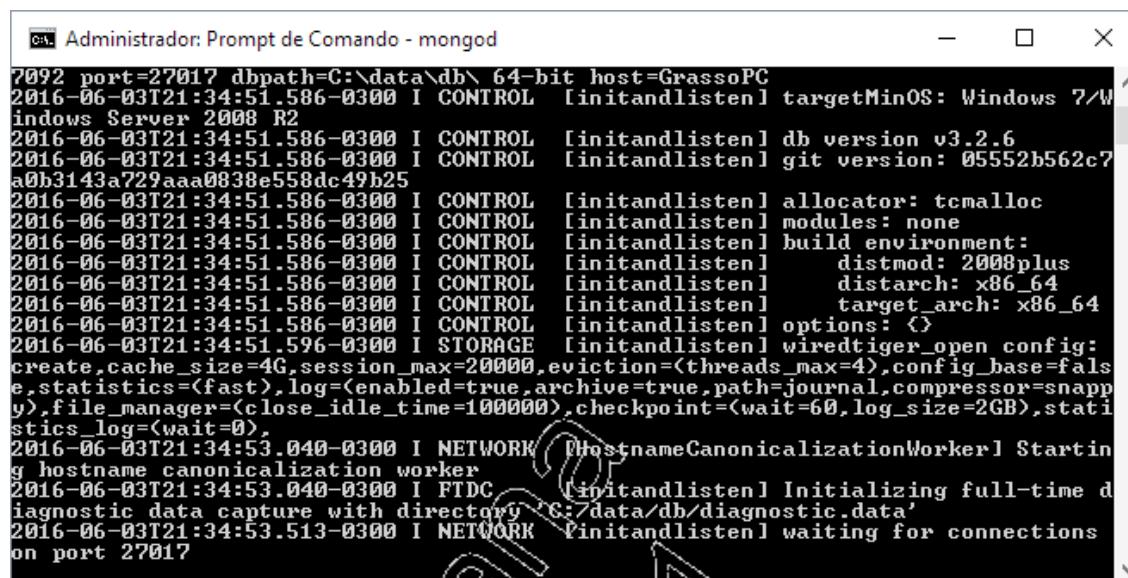
E, por último, por meio do Command Prompt, iniciado como administrador, é necessário chamar o servidor para iniciar o serviço. O caminho padrão de instalação do programa é este: **C:\Program Files\MongoDB\Server\3.2\bin\mongod.exe**.

Uma dica para abrir o Command Prompt como administrador é pressionar o botão Windows + X e escolher **Command Prompt (Admin)** no menu.



O console vai gerar uma série de mensagens, e a última, depois de inicializados todos os módulos do programa, é **waiting for connections on port XXXX: C:\Program Files\MongoDB 2.6 Standard\bin>mongod**.

A resposta será: ... [initandlisten] waiting for connections on port 27017.



```
7092 port=27017 dbpath=C:\data\db\ 64-bit host=GrassoPC
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] targetMinOS: Windows 7/W
indows Server 2008 R2
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] db version v3.2.6
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] git version: 05552b562c7
a0b3143a729aaa0838e558dc49b25
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] allocator: tcmalloc
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] modules: none
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] build environment:
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] distmod: 2008plus
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] distarch: x86_64
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] target_arch: x86_64
2016-06-03T21:34:51.586-0300 I CONTROL [initandlisten] options: <>
2016-06-03T21:34:51.596-0300 I STORAGE [initandlisten] wiredtiger_open config:
create,cache_size=4G,session_max=20000,eviction=<threads_max=4>,config_base=fals
e,statistics=<fast>,log=<enabled=true,archive=true,path=journal,compressor=snap
p>,file_manager=<close_idle_time=100000>,checkpoint=<wait=60,log_size=2GB>,stati
stics_log=<wait=0>,
2016-06-03T21:34:53.040-0300 I NETWORK [hostnameCanonicalizationWorker] Startin
g hostname canonicalization worker
2016-06-03T21:34:53.040-0300 I FTDC [initandlisten] Initializing full-time d
iagnostic data capture with directory 'C:\data\db\diagnostic.data'
2016-06-03T21:34:53.513-0300 I NETWORK [initandlisten] waiting for connections
on port 27017
```

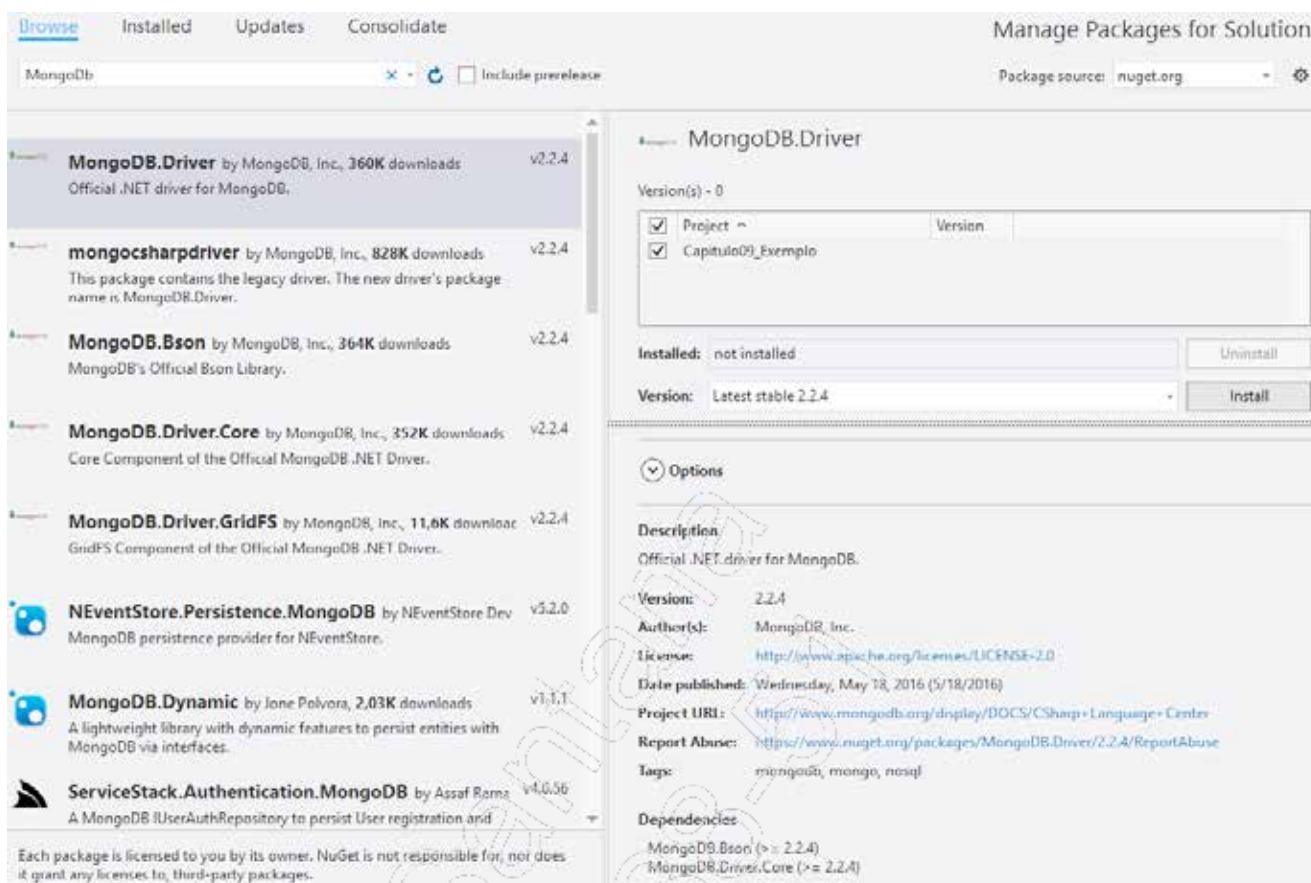
O serviço está no ar e já pode ser consumido pelas aplicações cliente. Existem várias maneiras de instalar e iniciar o serviço do servidor de dados. Em ambiente de produção, o ideal é instalar como um serviço Windows.

Para finalidades de teste, rodar o banco de dados diretamente como foi exposto é o suficiente.

9.3.3. Instalando o cliente

Em qualquer aplicação .NET, usando NuGet, é possível instalar os pacotes de conexão com MongoDB. Neste exemplo, será usada uma simples aplicação console.

No menu **Tools**, escolha **NuGet Package Manager** e **Manage NuGet Packages for Solution** e, então, procure on-line por **MongoDB**. Deve ser encontrado o **MongoDB.Driver**.



A instalação adiciona as bibliotecas necessárias para utilizar o servidor MongoDB.

9.3.4. Visão geral

O banco de dados MongoDB é do tipo **Document**. Isso significa que todo objeto armazenado no banco é um documento (neste caso, **BsonDocument**) e que pode estar dentro de uma coleção (**MongoCollection**). A estrutura de cada documento pode variar e os subitens são armazenados dentro do próprio documento.

Existe uma tendência, principalmente de quem usou apenas bancos de dados relacionais, de comparar um documento com um registro ou linha de uma tabela de um banco de dados. Essa comparação não está de todo errada, mas existem algumas diferenças importantes:

- Cada documento pode ter estrutura interna diversa;
- Os documentos são agrupados em coleções, mas podem conter outros documentos, e a estrutura de ambos é dinâmica. Isso é muito diferente de uma tabela em um banco de dados relacional, porque o princípio desta é definir uma estrutura fixa, que não é o caso em bancos de dados NoSQL.

As seguintes classes são utilizadas para interagir com o servidor de banco de dados MongoDB:

- **Namespace MongoDB.Server**
 - **MongoServer**: Classe que representa o servidor MongoDB. Permite criar, alterar, excluir e obter instâncias de bancos de dados;
 - **MongoClient**: Classe que representa um cliente do banco de dados;
 - **MongoDatabase**: Classe que representa um banco de dados;
 - **MongoCollection**: Representa uma coleção de documentos. É o conceito mais próximo de tabela de um banco relacional. Uma coleção NoSQL, no entanto, não tem campos fixos. Cada documento na coleção pode ter uma estrutura interna diferente das demais;
 - **QueryDocument**: Representa uma pesquisa a ser efetuada no banco de dados;
 - **CommandResult**: O resultado de um comando enviado ao servidor.
- **Namespace MongoDB.Bson**
 - **BsonDocument**: Representa um documento no formato BSON. Este é o item principal de armazenamento. Comparando com bancos de dados relacionais, é o equivalente a um registro;
 - **BsonValue**: Classe abstrata que representa um valor de um determinado tipo (**inteiro**, **string**, **data** etc.);

- **BsonElementAttribute**: Classe usada para mapear uma classe que será gravada como um documento BSON.
- **Namespace MongoDB.DriverBuilders**
 - **Query**: Classe utilizada para criar pesquisas.

9.3.5. Conectando o banco

Para conectar ou criar um banco de dados, a classe **MongoClient** é utilizada:

```
var client=new MongoClient();
```

Esta classe fornece acesso os bancos de dados no servidor, por meio do método **GetDatabase**, que retorna uma instância de uma classe que implemente a interface **IMongoDatabase**. Nesta versão do driver (2.2), a classe retornada se chama **MongoDatabaseImpl**. O banco de dados não precisa existir. Quando for adicionado **documentos**, ele será criado automaticamente.

```
var bancoDeDados = client.GetDatabase("loja");
```

Por meio de uma instância da classe **MongoDatabaseImpl**, é possível criar coleções de documentos. O resultado de uma operação de criação é uma instância da classe **CommandResult**, que fornece a propriedade **OK**, usada para saber se o comando foi executado com êxito:

```
var resultado = bancoDeDados.CreateCollection("produtos");
if (!resultado.Ok)
{
    //Tratamento de erro....
```

Para obter acesso a uma coleção, é necessário usar o método **GetCollection** da classe **MongoDatabase**:

```
var produtos=
    bancoDeDados.GetCollection<BsonDocument>("produtos");
```

Para inserir um documento em uma coleção, usa-se o método **InsertOne** da classe **MongoCollection**. Mas, antes, é necessário criar um documento (uma instância da classe **BsonDocument**):

```
var produto = new BsonDocument {
    { "Nome", "Notebook" },
    { "Preco", 5400.00 },
    { "Categoria", "Informática" }
};

produtos.InsertOne(produto);
```

Para obter uma lista completa de todos os documentos de uma coleção, o método **Find()** da classe **MongoCollection** é utilizado:

```
var docs = colecao.Find(new BsonDocument()).ToList();

foreach (BsonDocument docGravado in docs)
{
    Console.WriteLine(docGravado.ToString());
}
```

O servidor insere automaticamente uma propriedade chamada **_id** do tipo **ObjectId**, contendo uma string que identifica o documento. É obrigatório haver uma coluna identificadora, como uma chave primária de uma tabela relacional.

9.3.6. Mapeando o Model Domain

O exemplo anterior usa a classe **BsonDocument** para armazenar um documento em uma coleção. Em um projeto real, o ideal é mapear as entidades do domínio para que sejam gravadas.

A única exigência do MongoDB é que cada registro tenha um identificador, como uma chave primária, do tipo **ObjectId**. A classe a seguir será usada como modelo para armazenar documentos:

```
public class Usuario
{
    public ObjectId Id { get; set; }
    public string Nome { get; set; }
    public string Email { get; set; }
    public bool Administrador { get; set; }
}
```

Se houver uma propriedade na classe chamada **ID** ou **_id** do tipo **ObjectId**, o MongoDB automaticamente associa a chave do documento a essa propriedade. É possível, também, usar classes de atributos para mapear os campos:

```
public class Usuario
{
    [BsonElementAttribute("_id")]
    public ObjectId Id { get; set; }

    [BsonElementAttribute("Nome")]
    public string Nome { get; set; }

    [BsonElementAttribute("Email")]
    public string Email { get; set; }

    [BsonElementAttribute("Administrador")]
    public bool Administrador { get; set; }
}
```

Veja a listagem completa do método **Main**:

```
static void Main(string[] args)
{
    //Servidor e Banco
    var client = new MongoClient();
    var bancoDeDados = client.GetDatabase("loja");

    //Criando um usuário
    var usuario = new Usuario()
    {
        Nome = "Maria",
        Email = "maria@teste.com.br",
        Administrador = false
    };

    //Obtendo a coleção produtos
    var usuarios =
        bancoDeDados.GetCollection<Usuario>("usuarios");

    //Insere na coleção
    usuarios.InsertOne(usuario);
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
//Lendo todos os itens da coleção
var lista = usuarios.Find(new BsonDocument()).ToList();
foreach (Usuario user in lista)
{
    Console.WriteLine("Id:" + user.Id);
    Console.WriteLine("Nome:" + user.Nome);
    Console.WriteLine("Email:" + user.Email);
    Console.WriteLine("Administrador:" +
                      (user.Administrador ? "Sim" : "Não"));
    Console.WriteLine("-----");
}

Console.ReadLine();
```



9.3.7. CRUD: inserindo, alterando, excluindo e pesquisando

Os exemplos a seguir utilizam a coleção **usuarios** demonstrada no exemplo anterior:

```
//Servidor, Banco e Coleção
var client=new MongoClient();
var bancoDeDados = client.GetDatabase("loja");
var usuarios=bancoDeDados.GetCollection<Usuario>("usuarios");
```

9.3.7.1. Inserindo dados

A forma de inserir é usar o método **Insert** com uma nova instância da classe da coleção:

```
var usuario = new Usuario()
{
    Nome = "Maria",
    Email = "maria@teste.com.br",
    Administrador = false
};
usuarios.InsertOne(usuario);
```

9.3.7.2. Alterando dados

Para alterar, é necessário criar uma instância da classe **Builder** e os métodos **Update** e **UpdateOne** para definir os dados a serem alterados.

```
var update = Builders<Usuario>
    .Update
    .Set(user => user.Nome, "Maria Teste");

var result = usuários
    .UpdateOne<Usuario>(m=>m.Nome=="Maria", update);
```

Veja um exemplo de alteração completo:

```
static void Main(string[] args)
{
    //Servidor e Banco
    var client = new MongoClient();
    var bancoDeDados = client.GetDatabase("loja");

    //Obtendo a coleção produtos
    var usuarios =
        bancoDeDados.GetCollection<Usuario>("usuarios");

    var update = Builders<Usuario>
        .Update
        .Set(user => user.Nome, "Maria Teste");

    var result = usuários
        .UpdateOne<Usuario>(m=>m.Nome=="Maria", update);

}
```

9.3.7.3. Excluindo dados

O processo de exclusão é muito simples, usando apenas o método **DeleteOne** da coleção:

```
var result = usuarios.DeleteOne<Usuario>(m=>m.Nome=="Maria");
```

9.3.8. Usando a LINQ para obter dados em uma coleção

A linguagem LINQ (Language-Integrated Query) pode ser usada sobre uma coleção de dados vinda do servidor MongoDB. É necessário apenas importar o namespace **MongoDb.Query.Link** para obter os métodos de extensão necessários. Para obter uma coleção a partir de uma expressão LINQ, é usado o método de extensão **AsQueryable()** que retorna um objeto que implementa a interface **IQueryable<T>**.

```
using MongoDB.Driver.Linq;
...

//Obtém uma referência ao Servidor
var client = new MongoClient();
var bancoDeDados = client.GetDatabase("loja");
var usuarios = bancoDeDados.GetCollection<Usuario>("usuarios");

//Query Linq
var lista = from u in usuarios.AsQueryable()
            where u.Nome.StartsWith("M")
            select u;

//Lendo todos os itens da coleção
foreach (Usuario user in lista)
{
    Console.WriteLine("Id:" + user.Id);
    Console.WriteLine("Nome:" + user.Nome);
    Console.WriteLine("Email:" + user.Email);
    Console.WriteLine("Administrador:" +
                      (user.Administrador?"Sim":"Não"));
    Console.WriteLine("-----");
}
```

9.3.9. Armazenando objetos complexos

O verdadeiro poder do banco de dados NoSQL se dá quando uma estrutura complexa, com subitens, precisa ser armazenada e consultada. A classe listada adiante representa um pedido de venda:

```
public class Pedido
{
    public Pedido()
    {
        Produtos=new List<Produto>();
    }
    public ObjectId Id { get; set; }
    public DateTime Data { get; set; }
    public Destinatario Cliente { get; set; }
    public List<Produto> Produtos { get; protected set; }
    public decimal Total
    {
        get{ return Produtos.Sum(m => m.Total); }
    }

    public class Produto
    {
        public string Nome { get; set; }
        public decimal Preco { get; set; }
        public int Quantidade { get; set; }
        public decimal Total
        {
            get { return Quantidade * Preco; }
        }
    }

    public class Destinatario
    {
        public string Nome { get; set; }
        public string Endereco { get; set; }
        public string Cidade { get; set; }
        public string Estado { get; set; }
        public string Cep { get; set; }
    }
}
```

O primeiro campo da classe **Pedido** é o **ID**, do tipo **ObjectId**, para que o banco de dados gere um ID automático.

```
public class Pedido
{
    public ObjectId Id { get; set; }
}
```

O segundo campo é **Data do Pedido**, do tipo **DateTime**:

```
public class Pedido
{
    public ObjectId Id { get; set; }
    public DateTime Data { get; set; }
}
```

O terceiro campo é do tipo **Destinatario**, uma classe interna que define os dados de um cliente:

```
public class Pedido
{
    public ObjectId Id { get; set; }
    public DateTime Data { get; set; }
    public Destinatario Cliente { get; set; }

}
```

A classe **Destinatario** está definida dentro da classe **Pedido**:

```
public class Destinatario
{
    public string Nome { get; set; }
    public string Endereco { get; set; }
    public string Cidade { get; set; }
    public string Estado { get; set; }
    public string Cep { get; set; }
}
```

O próximo campo é uma lista de produtos que pertencem ao pedido:

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

A instância dessa lista é criada dentro da classe **Pedido** ou derivadas, por isso, o **Set** está marcado como **Protected**, para evitar criar essa lista de outra maneira. A lista é criada no construtor:

```
public Pedido()
{
    Produtos=new List<Produto>();
}
```

A classe **Produto** está definida dentro de pedido:

```
public class Produto
{
    public string Nome { get; set; }
    public decimal Preco { get; set; }
    public int Quantidade { get; set; }
    public decimal Total
    {
        get { return Quantidade * Preco; }
    }
}
```

E o último campo é um campo que calcula o total, usando a sintaxe LINQ:

```
public decimal Total
{
    get{ return Produtos.Sum(m => m.Total); }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

A seguir, veja novamente a listagem da classe, mas, desta vez, dividida em partes para melhor visualização:

```
//  
//Classe pedido  
//  
public class Pedido  
{  
  
    //Construtor  
    public Pedido()  
{  
        Produtos=new List<Produto>();  
    }  
  
    //Propriedades  
    public ObjectId Id { get; set; }  
    public DateTime Data { get; set; }  
    public Destinatario Cliente { get; set; }  
    public List<Produto> Produtos { get; protected set;}  
    public decimal Total{get{return Produtos.Sum(m=>m.Total);}}  
  
    //Classe Produto  
    public class Produto  
{  
        public string Nome { get; set; }  
        public decimal Preco { get; set; }  
        public int Quantidade { get; set; }  
        public decimal Total { get { return Quantidade * Preco; } }  
    }  
  
    //Classe Destinatário  
    public class Destinatario  
{  
        public string Nome { get; set; }  
        public string Endereco { get; set; }  
        public string Cidade { get; set; }  
        public string Estado { get; set; }  
        public string Cep { get; set; }  
    }  
}
```

O seguinte código insere um pedido na coleção **Pedidos**:

```
//Obtém uma referência ao Servidor  
var client = new MongoClient();  
var bancoDeDados = client.GetDatabase("loja");  
var pedidos = bancoDeDados.GetCollection<Pedido>("pedidos");
```

```
//Cria um pedido  
var ped = new Pedido()  
{  
    Data = DateTime.Now,  
    Cliente = new Pedido.Destinatario()  
    {  
        Nome = "José da Silva",  
        Endereco = "Avenida Paulista, 900",  
        Cidade = "São Paulo",  
        Estado = "SP",  
        Cep = "04034-234"  
    }  
};
```

```
//Adiciona Produtos...  
ped.Produtos.Add(new Pedido.Produto()  
{ Nome = "Mouse", Quantidade = 1, Preco = 100 });  
  
ped.Produtos.Add(new Pedido.Produto()  
{ Nome = "Monitor", Quantidade = 2, Preco = 500 });
```

```
//Pedido completo. Adiciona na coleção pedidos  
pedidos.InsertOne(ped);
```

```
//Obtém o número do pedido criado  
ObjectId numeroPedido = ped.Id;
```

```
//Localiza o pedido na coleção  
var filter = Builders<Pedido>.Filter.Eq(m => m.Id, numeroPedido);  
var pedido = pedidos.Find(filter).FirstOrDefault();
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
//Se encontra, exibe...
if (pedido != null)
{
    Console.WriteLine("");
    Console.WriteLine("Número do Pedido:" + pedido.Id);
    Console.WriteLine("Data:" + pedido.Data);
    Console.WriteLine("");

    Console.WriteLine("Cliente:");
    Console.WriteLine(" " + pedido.Cliente.Nome);
    Console.WriteLine(" " + pedido.Cliente.Endereco);
    Console.WriteLine(" " + pedido.Cliente.Cidade + ", "
        + pedido.Cliente.Estado);
    Console.WriteLine(" CEP:" + pedido.Cliente.Cep);
    Console.WriteLine("");

    Console.WriteLine("Produtos:");

    foreach (var p in pedido.Produtos)
    {
        Console.WriteLine(" Produto:" + p.Nome);
        Console.WriteLine(" Preço:" + p.Preco.ToString("C"));
        Console.WriteLine(" Quantidade:" + p.Quantidade);
        Console.WriteLine(" Total:" + p.Total.ToString("C"));
        Console.WriteLine("");
    }

    Console.WriteLine("Total do Pedido:" +
        pedido.Total.ToString("c"));
}

else
{
    Console.WriteLine("Pedido não encontrado.");
}
```

Confira o resultado:

```
C:\WINDOWS\system32\cmd.exe - □ X

Número do Pedido:5752b0ff6d800234f87efb81
Data:04/06/16 10:44:15

Cliente:
José da Silva
Avenida Paulista, 900
São Paulo, SP
CEP:04034-234

Produtos:
Produto:Mouse
Preço:R$ 100,00
Quantidade:1
Total:R$ 100,00

Produto:Monitor
Preço:R$ 500,00
Quantidade:2
Total:R$ 1.000,00

Total do Pedido:R$ 1.100,00
Pressione qualquer tecla para continuar...  
Mikael Bento 5752b0ff6d800234f87efb81
```

Pontos principais

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

- Um banco de dados NoSQL do tipo **Document** usa um conceito de documentos e coleções para manipular dados;
- **MongoDB** é um banco de dados NoSQL do tipo **Document**;
- As principais classes são: **MongoClient**, **MongoDatabase** e **MongoCollection**;
- Os documentos do banco de dados MongoDB são armazenados no formato **BSON**, que é uma versão binária do formato **JSON**;
- Os tipos de bancos de dados NoSQL são: **Document**, **Chave/Valor**, **Grafo**, **Orientado a objetos** e **Colunas/Tabular**;
- A classe que representa um documento é a **BsonDocument**;
- A classe **Filter** é utilizada para realizar pesquisa nos dados armazenados pelo MongoDB.

9

NoSQL

Teste seus conhecimentos

Mikael B
426.279.3857
Santana



IMPACTA
EDITORA

1. Qual alternativa contém um tipo que não é um banco de dados NoSQL?

- a) Documento
- b) Chave/Valor
- c) Relacional
- d) Colunas/Tabular
- e) Orientado a objetos

2. Qual das alternativas a seguir é uma biblioteca das que precisamos referenciar para criar uma aplicação que utilize um servidor MongoDB?

- a) System.Data.OleDb
- b) System.Data.Xml
- c) MongoDB.Driver
- d) MongoDB.Provider
- e) Nenhuma das alternativas anteriores está correta.

3. Qual classe é utilizada para estabelecer a conexão com o servidor MongoDB?

- a) DbConnection
- b) MongoClient
- c) MongoDatabase
- d) DbClient
- e) DbServer

4. Quais são as principais classes de acesso ao servidor Mongo e aos dados armazenados?

- a) MongoClient, MongoDB e MongoProvider.
- b) MongoServer, MongoDB e MongoTable.
- c) MongoClient, MongoDatabase e MongoTable.
- d) MongoClient, MongoDatabase e MongoCollection.
- e) MongoServer, MongoProvider e MongoTables.

5. Qual classe é utilizada para selecionar registros?

- a) MongoSelect
- b) Filter
- c) Array
- d) Select
- e) GetQuery

9

NoSQL

Mãos à obra!

Mikael B. Santana
426.279.3857



IMPACTA
EDITORA

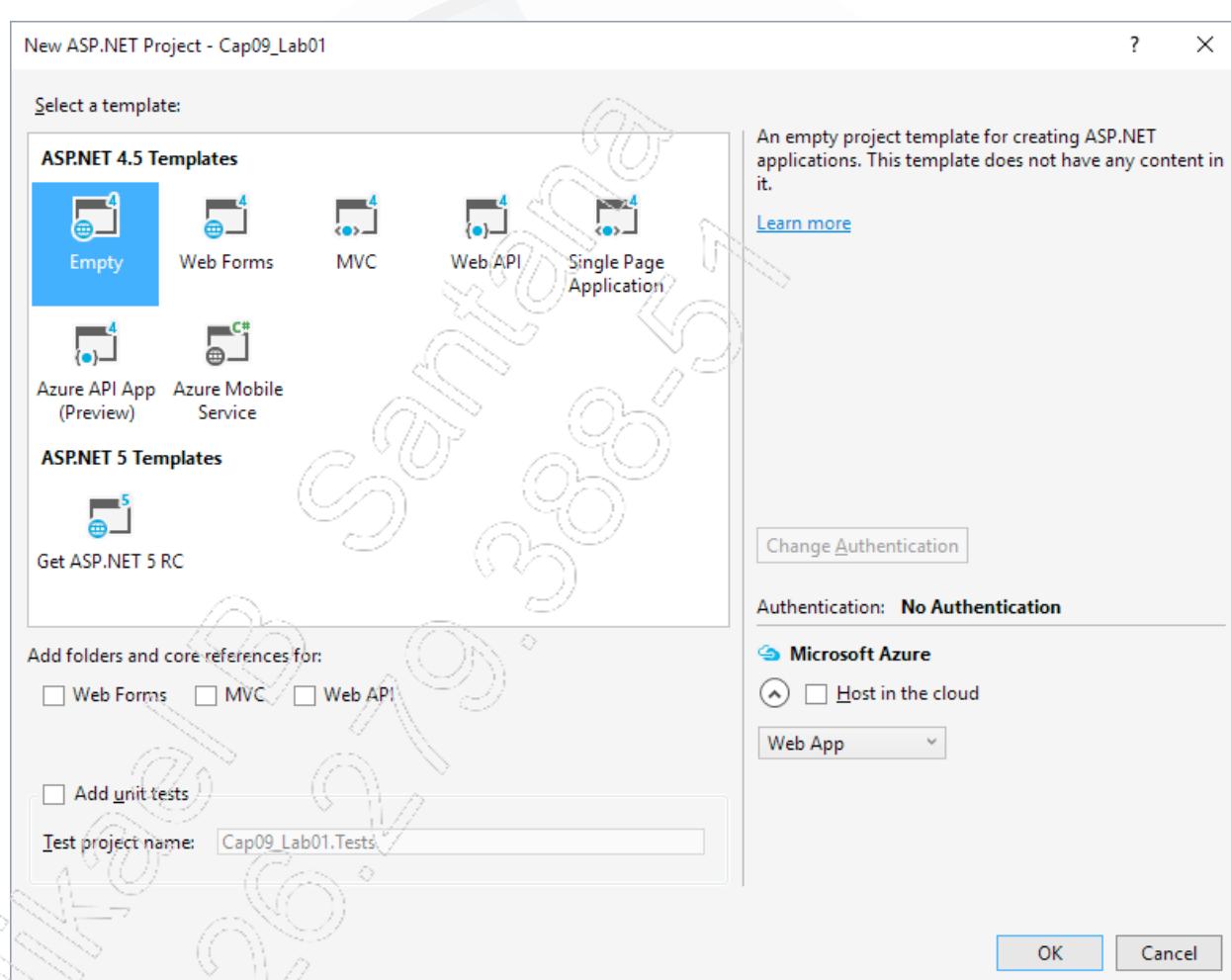
Laboratório 1

A - Criando um blog coletivo

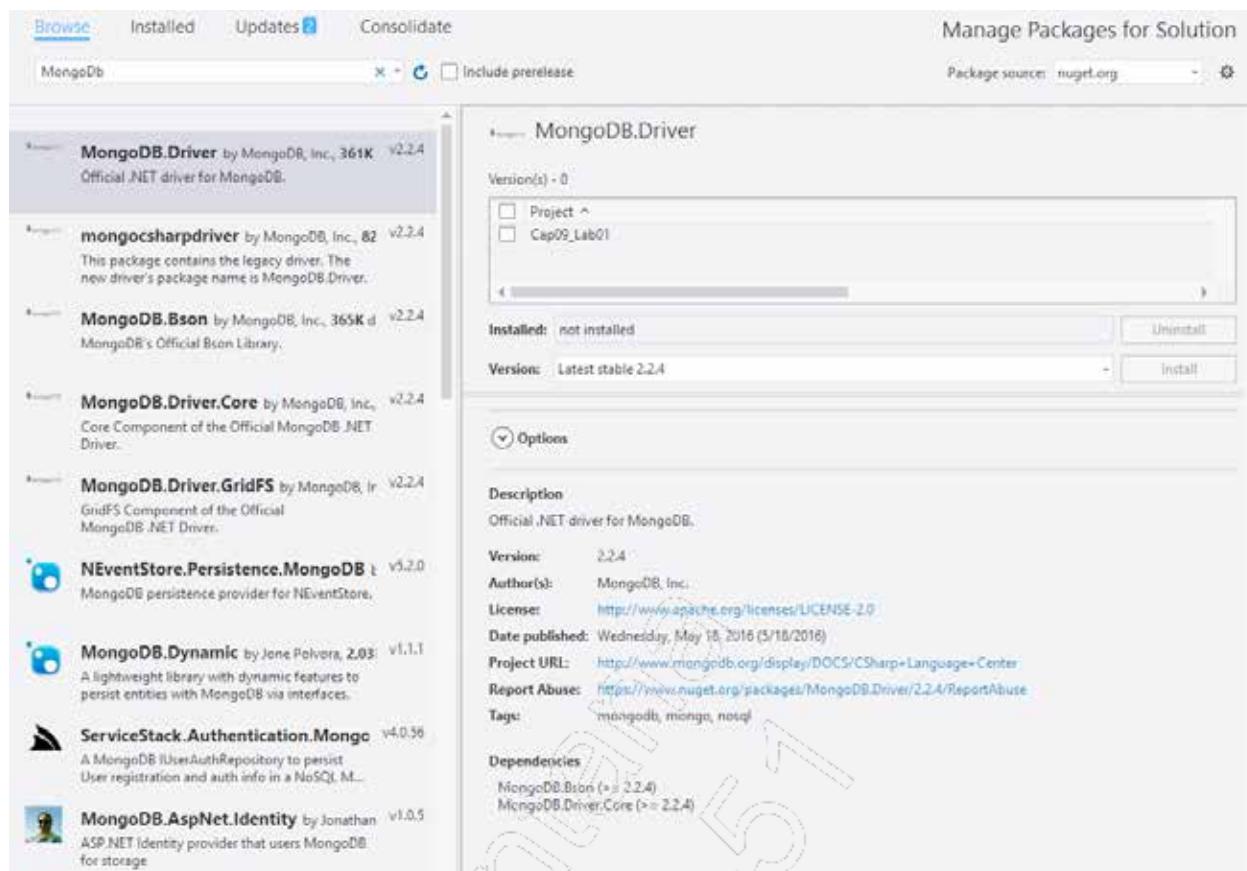
Neste laboratório, vamos criar um blog coletivo e armazenar as publicações e comentários no banco de dados MongoDB.

- **Arquivos da aplicação**

1. Crie um novo projeto Web vazio chamado **Cap09_Lab01**;



2. Usando NuGet, adicione uma referência à biblioteca MongoDB.Driver:

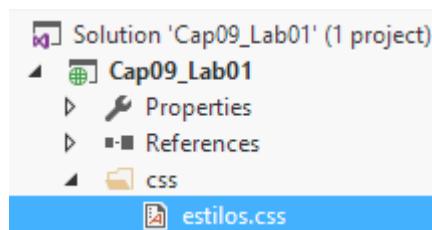


3. Se não foi feito ainda, instale e ative o servidor MongoDB (caso necessário, verifique as instruções neste capítulo):

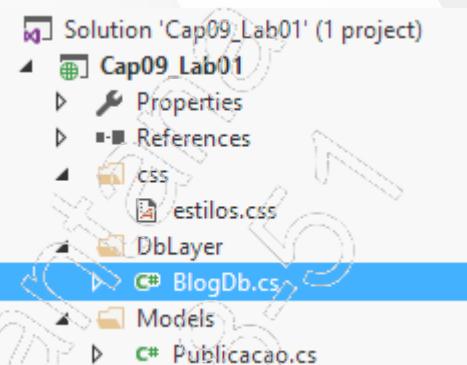
```
2016-06-04T07:36:28.299-0300 I NETWORK [initandlisten] connection accepted from 127.0.0.1:57968 #60 <1 connection now open>
2016-06-04T07:36:28.304-0300 I NETWORK [initandlisten] connection accepted from 127.0.0.1:57969 #61 <2 connections now open>
2016-06-04T07:36:28.471-0300 I NETWORK [conn61] end connection 127.0.0.1:57969
<1 connection now open>
2016-06-04T07:36:28.471-0300 I NETWORK [conn60] end connection 127.0.0.1:57968
<1 connection now open>
2016-06-04T07:42:18.839-0300 I NETWORK [initandlisten] connection accepted from 127.0.0.1:57989 #62 <1 connection now open>
2016-06-04T07:42:19.919-0300 I NETWORK [initandlisten] connection accepted from 127.0.0.1:57990 #63 <2 connections now open>
2016-06-04T07:42:19.008-0300 I NETWORK [conn63] end connection 127.0.0.1:57990
<1 connection now open>
2016-06-04T07:42:19.008-0300 I NETWORK [conn62] end connection 127.0.0.1:57989
<1 connection now open>
2016-06-04T07:44:15.032-0300 I NETWORK [initandlisten] connection accepted from 127.0.0.1:58002 #64 <1 connection now open>
2016-06-04T07:44:15.112-0300 I NETWORK [initandlisten] connection accepted from 127.0.0.1:58003 #65 <2 connections now open>
2016-06-04T07:44:15.200-0300 I NETWORK [conn64] end connection 127.0.0.1:58002
<1 connection now open>
2016-06-04T07:44:15.200-0300 I NETWORK [conn65] end connection 127.0.0.1:58003
<1 connection now open>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

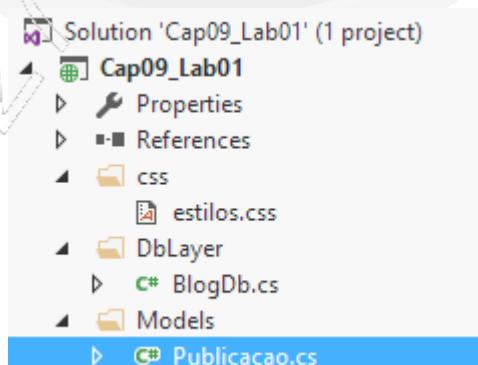
4. Crie uma pasta chamada **css** e insira uma folha de estilos CSS chamada **estilos.css**:



5. Adicione uma pasta chamada **DbLayer** e adicione uma classe chamada **BlogDb**:

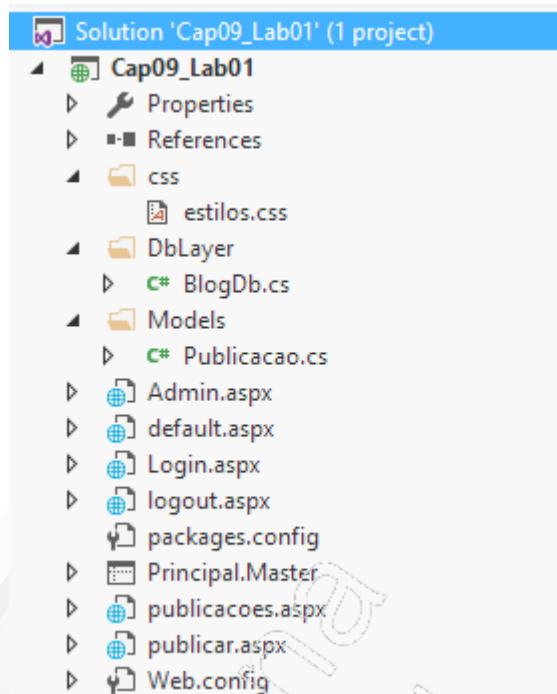


6. Crie uma pasta **Models** e, nesta pasta, adicione uma classe chamada **Publicacao**:



7. Adicione ao projeto uma Master Page chamada **Principal.Master**:

8. Adicione ao projeto as seguintes páginas baseadas na Master Page: **Admin.aspx**, **default.aspx**, **Login.aspx**, **logout.aspx**, **publicacoes.aspx** e **publicar.aspx**. Isso termina a estrutura do site;



- **Infraestrutura de acesso a dados**

9. Defina a classe **Publicacao** conforme descrito. O identificador é do tipo **Guid**. Isso evita duplicidade quando houver acessos simultâneos. Os bancos de dados NoSQL geralmente, não têm o tipo **Identity** como o SQL Server. Outra particularidade é a lista de comentários definida como somente leitura e gravação apenas dentro da classe ou classes derivadas (**protected set**). A lista é criada no construtor:

```
public class Publicacao
{
    public Publicacao()
    {
        this.Comentarios = new List<Comentario>();
    }

    public Guid Id { get; set; }

    public string Titulo { get; set; }

    public DateTime DataPublicacao { get; set; }

    public string Autor { get; set; }

    public string Texto { get; set; }

    public List<Comentario> Comentarios{get; protected set; }
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
public class Comentario
{
    public Guid Id { get; set; }

    public string Autor { get; set; }

    public string Texto { get; set; }

    public DateTime DataPublicacao{ get; set; }

}
```

10. Na classe **BlogDb**, na qual o acesso aos dados é executado, defina o método **Validar**. O título do texto é obrigatório. O autor é obtido automaticamente por meio do login e a data de publicação é obtida do sistema operacional, portanto, esses dois campos não precisam de validação:

```
private static void Validar(Publicacao p)
{
    if (string.IsNullOrEmpty(p.Titulo))
    {
        throw new Exception(
            "É necessário informar o título");
    }

    if (string.IsNullOrEmpty(p.Texto))
    {
        throw new Exception(
            "É necessário informar o texto");
    }
}
```

11. Praticamente todos os métodos dessa classe vão chamar a coleção de publicações. Crie o método que retorna a coleção:

```
private static IMongoCollection<Publicacao> ObterPublicacoes()
{
    var client = new MongoClient();
    var bancoDeDados = client.GetDatabase("blog");

    var publicacoes =bancoDeDados
                    .GetCollection<Publicacao>("publicacoes");
    return publicacoes;
}
```

12. Na classe **BlogDb**, crie o método **Incluir**. O processo é bem simples: conseguindo uma referência à coleção de publicações, basta usar o método **InsertOne** e passar a instância recebida como parâmetro pelo formulário;

```
// Incluir
public static bool Incluir(Publicacao p)
{
    Validar(p);
    var publicacoes = ObterPublicacoes();

    publicacoes.InsertOne(p);

    return true;
}
```

13. Na classe **BlogDb**, crie o método **Excluir**:

```
// Excluir
public static bool Excluir(Guid publicacaoId)
{
    var publicacoes = ObterPublicacoes();
    publicacoes.DeleteOne(m=>m.Id==publicacaoId);
    return true;
}
```

14. Na classe **BlogDb**, crie o método **Alterar**:

```
// Alterar
public static bool Alterar(Publicacao p)
{
    Validar(p);
    var publicacoes = ObterPublicacoes();
    publicacoes.ReplaceOne(m => m.Id == p.Id, p);
    return true;
}
```

15. Até agora, foram definidos os métodos **Incluir**, **Alterar** e **Excluir**. Os próximos métodos retornam informações. Defina o método **ObterPublicacao**, usado para exibir o conteúdo completo de uma publicação, inclusive com os comentários:

```
//  
// Obter Publicação  
//  
public static Publicacao ObterPublicacao(Guid id)  
{  
  
    var publicacoes = ObterPublicacoes();  
    var publicacao = publicações  
        .Find(m => m.Id == id).FirstOrDefault();  
    return publicacao;  
}
```

16. Crie o método **PublicacaoLista**. Esse método retorna todas as publicações em ordem decrescente de data;

```
//  
// Lista de Publicações  
//  
public static List<Publicacao> PublicacaoLista()  
{  
    var publicacoes = ObterPublicacoes();  
    var lista = from u in publicacoes.AsQueryable()  
               orderby u.DataPublicacao descending  
               select u;  
  
    return lista.ToList();  
}
```

17. Crie o método **MaisRecentePublicacao**. Esse método retorna a última publicação gravada. É praticamente igual ao método anterior, retornando apenas o primeiro registro:

```
//  
// Publicação Mais Recente  
//  
public static Publicacao MaisRecentePublicacao()  
{  
    var publicacoes = ObterPublicacoes();  
    var publicacao =  
        (from u in publicacoes.AsQueryable()  
         orderby u.DataPublicacao descending  
         select u).FirstOrDefault();  
    return publicacao;  
}
```

18. Crie o método seguinte. Ele não faz parte do banco de dados, mas faz parte da estrutura do programa. O método **Login** obtém a senha gravada no **Web.Config** e compara com uma senha enviada. Se for igual, o usuário é aceito e um cookie é gravado para autenticá-lo. O método **UrlEncode** da classe **HttpUtility** é utilizado para evitar problemas com acentuação:

```
//  
// Login  
//  
internal static bool Login(string nome, string senha)  
{  
  
    string senhaBlog =  
        ConfigurationManager.AppSettings["senha"];  
  
    if (string.IsNullOrEmpty(nome))  
    {  
        return false;  
    }  
  
    if (senha != senhaBlog)  
    {  
        return false;  
    }  
  
    HttpContext.Current.Response.Cookies["usuario"].Value =  
        HttpUtility.UrlEncode(nome);  
  
    return true;  
}
```

19. Crie o método **ObterUsuario**, que retorna o nome do usuário atual, caso este tenha passado pelo processo de login e o cookie de autenticação esteja gravado no navegador:

```
//  
// Obter Usuário  
//  
public static string ObterUsuario()  
{  
  
    if (HttpContext.Current.Request.Cookies["usuario"] == null)  
    {  
        return null;  
    }  
  
    else  
    {  
        string usuario = HttpUtility.UrlDecode(  
            HttpContext  
                .Current  
                .Request  
                .Cookies["usuario"].Value  
        );  
  
        return usuario;  
    }  
}
```

20. Crie o método **IncluirComentário**, que adiciona um comentário na coleção **Comentarios** da classe **Publicacao**:

```
//  
// Incluir Comentário  
//  
public static bool IncluirComentario(  
    Guid publicacaoId, string nome, string texto)  
{  
  
    if (string.IsNullOrEmpty(nome))  
    {  
        throw new Exception("É necessário informar o nome");  
    }  
  
    if (string.IsNullOrEmpty(texto))  
    {  
        throw new Exception("É necessário informar o texto");  
    }  
  
    var client = new MongoClient("mongodb://localhost");  
    var servidor = client.GetServer();  
    var bancoDeDados = servidor.GetDatabase("blog");  
  
    var publicacoes = bancoDeDados  
        .GetCollection<Publicacao>("publicacoes");  
  
    var docQuery = new QueryDocument("_id", publicacaoId);  
  
    Publicacao original = publicacoes.FindOne(docQuery);  
  
    if (original != null)  
    {  
        var comentario = new Publicacao.Comentario();  
        comentario.Id = Guid.NewGuid();  
        comentario.Autor = nome;  
        comentario.Texto = texto;  
        comentario.DataPublicacao = DateTime.Now;  
        original.Comentarios.Add(comentario);  
  
        publicacoes.Save(original);  
    }  
  
    return true;  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

21. Crie o método **ExcluirComentário**, que remove um comentário da coleção **comentarios** da classe **Publicacao**:

```
public static bool ExcluirComentario(Guid publicacaoId,
                                         Guid comentarioId)
{
    var publicacoes = ObterPublicacoes();

    Publicacao original = publicações
        .Find(m => m.Id == publicacaoId)
        .FirstOrDefault();
    var comentario = original.Comentarios
        .Find(m => m.Id == comentarioId);

    if (comentario != null)
    {
        original.Comentarios.Remove(comentario);
    }
    Publicações
        .FindOneAndReplace(m => m.Id == publicacaoId, original);

    return true;
}
```

A infraestrutura está pronta! A classe **BlogDb** tem todas as operações necessárias para incluir, alterar, excluir e pesquisar as publicações e comentários, assim como os recursos de validação. O restante da implementação é criar a interface. A imagem a seguir mostra todos os métodos da classe:



- Construindo a interface

22. No arquivo **Principal.Master**, crie o HTML necessário: referência à folha de estilos e os elementos **header**, **nav**, **section** e **footer**:

```
<%@ Master ....>

<!DOCTYPE html>

<html>
<head runat="server">
    <title></title>
    <link href="css/estilos.css" rel="stylesheet" />
</head>
<body>
    <form id="form1" runat="server">

        <header>
            <h1>Wiki Blog </h1>
        </header>

        <nav>
        </nav>

        <section class="conteudo">
            <asp:ContentPlaceHolder
                ID="ContentPlaceHolder1" runat="server">
            </asp:ContentPlaceHolder>
        </section>

        <footer>
            <p>&copy; 2014 - Desenvolvido para o curso de ASP.NET</p>
        </footer>

    </form>
</body>
</html>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

23. Na folha de estilos, crie os estilos necessários para esses elementos:

```
*  
{  
    margin:0px;  
}  
  
body {  
    margin:0px;  
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;  
}  
  
header {  
    background-color:#8badc6;  
    padding:30px;  
    color:#f3f3f3;  
}  
  
.conteudo {  
    padding:20px;  
    min-height:100px;  
}  
  
.conteudo h2 {  
    margin-bottom:20px;  
    color:#5d7180;  
}  
  
footer{  
    padding:20px;  
}  
  
nav {  
    padding:10px 0px 10px 20px;  
    border-bottom:1px solid #ccc;  
}
```

24. No arquivo **Principal.Master**, dentro do elemento **nav**, crie os itens de navegação. Repare que alguns itens são âncoras HTML e outros hiperlinks. Os que são hiperlinks vão aparecer ou não se tiver um usuário logado ou não. Por exemplo, o link **Publicar** só aparece se o usuário estiver logado:

```
<nav>

    <asp:Label ID="usuarioLabel" runat="server"
        CssClass="usuario">
    </asp:Label>

    <ul>
        <li><a href="default.aspx">Início</a></li>

        <li><a href="publicacoes.aspx">Publicações</a></li>

        <li>
            <asp:HyperLink
                NavigateUrl="~/publicar.aspx"
                ID="publicarHyperLink"
                Visible="false"
                runat="server">Publicar</asp:HyperLink>
        </li>

        <li>
            <asp:HyperLink
                NavigateUrl="~/login.aspx"
                ID="loginHyperLink"
                runat="server">Login</asp:HyperLink>

            <asp:HyperLink
                Visible="false"
                NavigateUrl="~/logout.aspx"
                ID="logoutHyperLink"
                runat="server">Encerrar Sessão</asp:HyperLink>
        </li>

    </ul>
</nav>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

25. Na folha de estilos, crie os estilos no navegador:

```
nav {  
    padding:10px 0px 10px 20px;  
    border-bottom:1px solid #ccc;  
  
}  
  
nav ul {  
    padding:0px;  
  
}  
nav li {  
    display:inline-block;  
}  
  
nav a {  
    text-decoration:none;  
    color:#303b44;  
}
```

26. Na página **Default.aspx**, dentro do segundo **Content**, crie os elementos iniciais:

```
<asp:Label ID="mensagemLabel"  
    runat="server"  
    CssClass="mensagem">  
</asp:Label>
```

```
<asp:Panel runat="server" ID="publicacaoPanel">  
    <section class="publicacao">  
        </section>  
  
    <section class="comentarios">  
        </section>  
  
    <section class="comentario-add">  
        </section>  
  
    <asp:Panel ID="editarPanel" visible="false" runat="server">  
        <section class="admin-area">  
            </section>  
    </asp:Panel>  
  
</asp:Panel>
```

27. Na folha de estilos, crie os elementos relacionados:

```
.publicacao {  
    padding:20px;  
}  
  
.comentarios {  
    padding:20px;  
    font-size:90%;  
}  
  
.comentario-add {  
    padding:20px;  
    font-size:90%;  
}  
  
.admin-area {  
    margin-top:40px;  
    margin-bottom:40px;  
}  
    .admin-area h3 {  
        color:red;  
        margin-bottom:20px;  
    }  
.usuario {  
    color:red;  
    position:absolute;  
    right:10px;  
    max-width:200px;  
    overflow:hidden;  
    white-space:nowrap;  
}  
.legenda {  
    display:block;  
}  
.campo {  
    display:block;  
    padding:4px;  
}  
  
.mensagem {  
    display:block;  
    margin-bottom:20px;  
    margin-top:20px;  
    font-weight:bold;  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

28. Dentro da seção **publicacao** (**default.aspx**), insira os itens da **publicacao**:

```
<section class="publicacao">

    <asp:Label runat="server"
        ID="tituloLabel"
        CssClass="publicacao-titulo">
    </asp:Label>

    <asp:Label runat="server"
        ID="autorLabel"
        CssClass="publicacao-autor">
    </asp:Label>

    <asp:Label runat="server"
        ID="dataPublicacaoLabel"
        CssClass="publicacao-data">
    </asp:Label>

    <asp:Label runat="server"
        ID="textoLabel"
        CssClass="publicacao-texto" >
    </asp:Label>

</section>
```

29. Na folha de estilo, crie os itens relacionados:

```
.publicacao-titulo{ font-size:160%;      display:block;
}

.publicacao-autor { display:block;  font-size:90%;
                    font-style:italic; margin-right:10px;
}

.publicacao-data{ display:inline-block;  font-size:80%;
                    font-style:italic;
}

.publicacao-texto{ display:block;  font-size:100%;
                    margin-top:30px; min-height:200px;
}
```

30. Na seção **Comentarios** da página **default.aspx**, insira os itens. Os vínculos, devido ao atributo **ItemType**, permitem usar a classe:

```
<h3>Comentários:</h3>
<asp:Repeater
    runat="server"
    ID="comentariosRepeater"      ItemType="ASPII_CAP08_LAB01.Models.
Publicacao+Comentario">

    <ItemTemplate>
        <div class="comentario-item">

            <asp:Label runat="server"
                ID="autorComentario"
                Text="<%# Item.Autor %>"
                CssClass="comentario-autor">
            </asp:Label>

            <asp:Label runat="server" ID="dataLabel"
Text='<%# Item.DataPublicacao.ToString("dd/mm/yyyy - HH:mm") %>' 
                CssClass="comentario-data">
            </asp:Label>

            <asp:Label runat="server" ID="textocomentario"
                Text="<%# Item.Texto %>"
                CssClass="comentario-texto">
            </asp:Label>

            <asp:Panel runat="server"
                ID="panelExcluirComentario"
                Visible='<%# Request.Cookies["usuario"]!=null %>'>

                <div class="form-botoes">
                    <asp:Button runat="server"
                        ID="excluirComentarioButton"
                        Text="Excluir comentário"
                        CommandArgument="<%# Item.Id.ToString() %>">
                    <asp:Button runat="server"
                        ID="botaoExcluir"
                        Text="Excluir"
                        OnClick="botaoExcluir_Click" />
                </div>
            </asp:Panel>
        </div>
    </ItemTemplate>
</asp:Repeater>
```

31. Na folha de estilo, crie os itens relacionados:

```
.comentarios {  
    padding:20px;  
    font-size:90%;  
}  
.comentario-item {  
    border-bottom:1px solid #ccc;  
    margin-bottom:5px;  
}  
.comentario-autor {  
    display:block;  
    font-size:90%;  
    margin-bottom:1px;  
    font-weight:bold;  
    font-style:italic;  
}  
.comentario-data {  
    font-size:80%;  
}  
.comentario-texto{  
    display:block;  
    padding-top:5px;  
    padding-bottom:30px;  
}  
.comentario-add {  
    padding:20px;  
    font-size:90%;  
}  
.comentario-add h3{  
    color:#5d7180;  
    margin-bottom:20px;  
}  
.comentarios h3{  
    color:#5d7180;  
    margin-bottom:20px;  
}
```

```
.comentario-add-legenda {  
    display: block;  
}  
.comentario-add-campo {  
    display: block;  
    width: 100%;  
    max-width: 600px;  
    padding: 5px;  
}  
  
.comentario-add-texto {  
    display: block;  
    min-height: 60px;  
    width: 100%;  
    max-width: 600px;  
    padding: 5px;  
}  
  
.form-botoes {  
    display: block;  
    padding-top: 10px;  
  
    margin-bottom: 10px;  
}  
.botao {  
    padding: 6px 6px 6px 6px;  
    border-radius: 3px;  
    border: 1px solid #808080;  
}
```

32. Na seção **comentario-add**, que é o formulário que insere os comentários, insira os itens seguintes:

```
<section class="comentario-add">

    <h3>Insira seu comentário:</h3>

    <div class="form-linha">

        <asp:Label runat="server" ID="usuarioLabel"
            Text="Nome:"
            CssClass="comentario-add-legenda"></asp:Label>

        <asp:TextBox runat="server" ID="nomeTextBox"
            CssClass="comentario-add-campo"></asp:TextBox>

    </div>

    <div class="form-linha">
        <asp:Label runat="server" ID="comentarioLabel"
            Text="Comentário"
            CssClass="comentario-add-legenda"></asp:Label>

        <asp:TextBox runat="server" ID="comentarioTextBox"
            TextMode="MultiLine"
            CssClass="comentario-add-texto"></asp:TextBox>
    </div>

    <div class="form-botoes">
        <asp:Button runat="server"
            ID="incluirComentarioButton"
            Text="Incluir Comentário"
            CssClass="botao"
            OnClick="incluirComentarioButton_Click" />
    </div>
</section>
```

33. Finalmente, no item **EditarPanel**, insira o HTML:

```
<asp:Panel ID="editarPanel" visible="false" runat="server">
    <section class="admin-area">

        <h3>Área do Administrador</h3>

        <asp:Button CssClass="botao"
            runat="server"
            ID="editarButton"
            Text="Editar esta publicação"
            OnClick="editarButton_Click" />
    </section>
</asp:Panel>
```

A página inicial está pronta! Quando estiver funcionando, esta será a aparência:



34. A página de login tem o código HTML seguinte. Insira o código em destaque:

```
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
runat="server">

    <h2>Login</h2>

    <asp:Label runat="server"
        Visible="false" ID="mensagemLabel"
        CssClass="mensagem"></asp:Label>

    <section>
        <div class="form-linha">
            <asp:Label runat="server" ID="nomeLabel"
                CssClass="legenda" Text="Nome:></asp:Label>

            <asp:TextBox CssClass="campo" runat="server"
                ID="nomeTextBox"></asp:TextBox>
        </div>

        <div class="form-linha">
            <asp:Label runat="server" ID="senhaLabel"
                CssClass="legenda" Text="Senha:></asp:Label>

            <asp:TextBox CssClass="campo" runat="server"
                ID="senhaTextBox"
                TextMode="Password"></asp:TextBox>
        </div>

        <div class="form-botoes">
            <asp:Button CssClass="botao"
                runat="server" ID="enviarButton"
                Text="Enviar" OnClick="enviarButton_Click" />
        </div>
    </section>

</asp:Content>
```

35. O código da página de login valida o usuário e redireciona para a página de publicação. Importe o namespace **[projeto].DbLayer** para acesso à classe **BlogDb**:

```
public partial class Login : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }

    protected void enviarButton_Click(
        object sender, EventArgs e)
    {

        string nome = nomeTextBox.Text;

        string senha = senhaTextBox.Text;

        if (BlogDb.Login(nome, senha))
        {
            Response.Redirect("~/publicar.aspx");
        }
        else
        {
            mensagemLabel.Text =
                "Usuário ou senha inválida";
            mensagemLabel.Visible = true;
        }
    }
}
```

36. Insira o HTML da página **publicar**:

```
<asp:Content
    ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
    runat="server">

    <h2>Publicar</h2>

    <asp:Label runat="server" ID="usuarioLabel"
        CssClass="usuario"></asp:Label>

    <asp:Label runat="server" ID="mensagemLabel"
        Visible="false" CssClass="mensagem"></asp:Label>

    <fieldset>

        <div class="form-linha">
            <asp:Label runat="server" ID="tituloLabel"
                CssClass="legenda"
                Text="Titulo:"></asp:Label>

            <asp:TextBox
                runat="server" ID="tituloTextBox"
                CssClass="campo_publicar-titulo"></asp:TextBox>
        </div>

        <div class="form-linha">
            <asp:Label runat="server" ID="Label1"
                CssClass="legenda" Text="Texto:"></asp:Label>

            <asp:TextBox runat="server" ID="textoTextBox"
                TextMode="MultiLine"
                CssClass="textoBlog"></asp:TextBox>
        </div>

        <div class="form-botoes">
            <asp:Button CssClass="botao" runat="server"
                ID="enviarButton"
                Text="Publicar"
                OnClick="enviarButton_Click" />

            <asp:Button CssClass="botao" runat="server"
                ID="excluirButton" Text="Excluir"
                OnClick="excluirButton_Click" />
        </div>
    </fieldset>
</asp:Content>
```

37. Insira o código da página **publicar.aspx**, que obtém os dados da tela e insere, altera ou exclui uma publicação:

```
public partial class Publicar : System.Web.UI.Page
{
    // Início da classe
    //

    //
    // Load
    //
    protected void Page_Load(object sender, EventArgs e)
    {
        string usuario = BlogDb.ObterUsuario();
        if (usuario == null)
        {
            Response.Redirect("~/Login.aspx");
        }

        if (!Page.IsPostBack)
        {
            excluirButton.Visible = false;
            string id = Request.QueryString["id"];
            if (!string.IsNullOrEmpty(id))
            {
                var publicacao =
                    BlogDb.ObterPublicacao(Guid.Parse(id));
                if (publicacao != null)
                {
                    Exibir(publicacao);
                    excluirButton.Visible = true;
                }
            }
        }
    }
}
```

```
//
// Mensagem
//
private void Mensagem(string msg)
{
    mensagemLabel.Text = msg;
    mensagemLabel.Visible = !string.IsNullOrEmpty(msg);
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
//  
// Exibir  
//  
private void Exibir(Publicacao publicacao)  
{  
    ViewState["id"] = publicacao.Id;  
    tituloTextBox.Text = publicacao.Titulo;  
    textoTextBox.Text = publicacao.Texto;  
    Mensagem(null);  
}
```

```
//  
// Publicar  
//  
protected void enviarButton_Click(object sender, EventArgs e)  
{  
    try  
    {  
        GravarPublicacao();  
        Mensagem("Artigo publicado com sucesso");  
    }  
    catch (Exception ex)  
    {  
        Mensagem(ex.Message);  
    }  
}
```

```
private void GravarPublicacao()  
{  
    var p = new Publicacao();  
    p.Texto = textoTextBox.Text;  
    p.Autor = BlogDb.ObterUsuario();  
    p.DataPublicacao = DateTime.Now;  
    p.Titulo = tituloTextBox.Text;  
    if (ViewState["id"] != null)  
    {  
        p.Id = Guid.Parse(ViewState["id"].ToString());  
        BlogDb.Alterar(p);  
    }  
    else  
    {  
        p.Id = Guid.NewGuid();  
        ViewState["id"] = p.Id.ToString();  
        BlogDb.Incluir(p);  
    }  
    enviarButton.Enabled = false;  
}
```

```
protected void excluirButton_Click(object sender, EventArgs e)
{
    if (ViewState["id"] != null)
    {
        Guid Id = Guid.Parse(ViewState["id"].ToString());
        BlogDb.Excluir(Id);
    }
    Mensagem("Artigo Excluido com sucesso");
    excluirButton.Enabled = false;
    enviarButton.Enabled = false;
}
```

```
//
//Final da Classe
//
}
```

38. Insira o código da Master Page, que exibe ou esconde os links que um usuário logado pode ver, dependendo do cookie estar ou não disponível na requisição:

```
public partial class Principal : System.Web.UI.MasterPage
{
    protected void Page_Load(object sender, EventArgs e)

        if (Request.Cookies["usuario"] != null)
        {
            publicarHyperLink.Visible = true;
            loginHyperLink.Visible = false;
            logoutHyperLink.Visible = true;
            usuarioLabel.Text =
                HttpUtility.UrlDecode(
                    Request.Cookies["usuario"].Value);
        }
        else
        {
            publicarHyperLink.Visible = false;
            loginHyperLink.Visible = true;
            logoutHyperLink.Visible = false;
            usuarioLabel.Text = string.Empty;
        }
    }
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

39. Insira o código da página **default**, que exibe uma publicação e comentários:

```
public partial class _default : System.Web.UI.Page
{
    //
    // Load
    //
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            if (!string.IsNullOrEmpty(Request.QueryString["id"]))
            {
                string id = Request.QueryString["id"];
                CarregarPublicacao(Guid.Parse(id));
            }
            else
            {
                CarregarPublicacaoMaisRecente();
            }
        }
    }

    //Admin Visível?
    editarPanel.Visible = Request.Cookies["usuario"] != null;
}

//
// Carregar Publicação
//
private void CarregarPublicacao(Guid id)
{
    var p = BlogDb.ObterPublicacao(id);
    if (p == null)
    {
        mensagem("Não foi possível carregar publicação");
        publicacaoPanel.Visible = false;
    }
    else
    {
        ExibirPublicacao(p);
        publicacaoPanel.Visible = true;
    }
}
```

```
//  
// Carregar a última publicação  
//  
private void CarregarPublicacaoMaisRecente()  
{  
    var p = BlogDb.MaisRecentePublicacao();  
    if (p == null)  
    {  
        mensagem("Não existem publicações");  
        publicacaoPanel.Visible = false;  
    }  
    else  
    {  
        ExibirPublicacao(p);  
        publicacaoPanel.Visible = true;  
    }  
}  
  
//  
// Exibir Publicação  
//  
private void ExibirPublicacao(Publicacao p)  
{  
    tituloLabel.Text = p.Titulo;  
    autorLabel.Text = p.Autor;  
    dataPublicacaoLabel.Text = p.DataPublicacao.ToString("dd/MM/yyyy - HH:mm");  
    textoLabel.Text = p.Texto;  
    ViewState["id"] = p.Id;  
  
    comentariosRepeater.DataSource = p.Comentarios;  
    comentariosRepeater.DataBind();  
  
    LimparFormComentario();  
}  
  
//  
// Limpar o Form de Comentário  
//  
private void LimparFormComentario()  
{  
    comentarioTextBox.Text = string.Empty;  
    nomeTextBox.Text = string.Empty;  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

```
//  
// Exibe/Econde mensagem  
//  
private void mensagem(string msg)  
{  
    mensagemLabel.Text=msg;  
    mensagemLabel.Visible = !string.IsNullOrEmpty(msg);  
  
}  
  
//  
// Alterar este post?  
//  
protected void editarButton_Click(  
    object sender, EventArgs e)  
{  
    Response.Redirect("~/publicar.aspx?id=" + ViewState["id"].ToString());  
}  
  
//  
// Incluir Comentário  
//  
protected void incluirComentarioButton_Click(  
    object sender, EventArgs e)  
{  
    try  
    {  
        var id = Guid.Parse(ViewState["id"].ToString());  
        BlogDb.IncluirComentario(id, nomeTextBox.Text, comentarioTextBox.  
Text);  
        Response.Redirect(  
            "~/Default.aspx?id=" + id.ToString());  
    }  
    catch (Exception ex)  
    {  
        mensagem(ex.Message);  
    }  
}
```

```
//  
// Excluir Comentário  
//  
protected void excluirComentarioButton_Click(  
    object sender, EventArgs e)  
{  
try  
{  
    Button btn = (Button)sender;  
    Guid publicacaoId = Guid.Parse(ViewState["id"].ToString());  
    Guid comentarioId = Guid.Parse(btn.CommandArgument);  
    BlogDb.ExcluirComentario(publicacaoId, comentarioId);  
    Response.Redirect("~/Default.aspx?id=" + publicacaoId.ToString());  
}  
catch (Exception ex)  
{  
    mensagem(ex.Message);  
}  
}  
}
```

40. Insira o HTML da página logout, que apenas exibe uma mensagem:

```
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"  
runat="server">  
  
    <h2>Encerrar Sessão</h2>  
  
    <p>  
        Sua Sessão foi encerrada com sucesso.  
    </p>  
</asp:Content>
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

41. Utilize o código a seguir, para que o código da página logout invalide o cookie de autenticação:

```
public partial class Logout : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (Request.Cookies["usuario"] != null)
        {
            Response.Cookies["usuario"].Expires =
                DateTime.Now.AddDays(-10);

            Response.Redirect("logout.aspx");
        }
    }
}
```

42. Insira o código HTML da página **publicacoes**, que exibe uma lista de publicações. O usuário, ao clicar, é direcionado para a página **Default.aspx** para exibir determinada publicação. O ID da publicação é passado via query string:

```
<asp:Content ID="Content2" ContentPlaceholderID="ContentPlaceHolder1"
runat="server">

    <h2>Publicações</h2>

    <asp:Label runat="server" CssClass="mensagem"
        ID="mensagemLabel"></asp:Label>

    <section>
        <asp:GridView CssClass="publicacoes-tabela"
            AutoGenerateColumns="false" runat="server"
            ID="publicacoesGridView">

            <Columns>
                <asp:HyperLinkField
                    DataTextField="Titulo"
                    DataNavigateUrlFields="Id"
                    DataNavigateUrlFormatString="default.aspx?id={0}"
                    HeaderText="Título" />

                <asp:BoundField DataField="Autor"
                    HeaderText="Autor" />
            </Columns>
        </asp:GridView>
    </section>
</asp:Content>
```

```
<asp:BoundField  
    DataField="DataPublicacao"  
    HeaderText="Publicado Em"  
    DataFormatString="{0:dd/MM/yyyy - HH:mm}" />  
</Columns>  
  
</asp:GridView>  
  
</section>  
</asp:Content>
```

43. Insira o código da página publicações, para que carregue a lista de publicações e exiba no grid:

```
public partial class Publicacoes : System.Web.UI.Page  
{  
    protected void Page_Load(object sender, EventArgs e)  
    {  
        var lista = BlogDb.PublicacaoLista();  
  
        publicacoesGridView.DataSource = lista;  
        publicacoesGridView.DataBind();  
  
        if (lista.Count == 0)  
        {  
            mensagemLabel.Text = "Não existem publicações";  
            mensagemLabel.Visible = true;  
        }  
    }  
}
```

44. Insira, na folha de estilos, os seguintes códigos para a tabela exibida:

```
.publicacoes-tabela {  
    width:100%;  
}  
.publicacoes-tabela td {  
    padding:6px;  
}  
  
.publicacoes-tabela hd {  
    padding:6px;  
}
```

Visual Studio 2015 - ASP.NET com C# Acesso a dados

45. No arquivo **Web.Config**, defina a senha e o formato de data e hora:

```
<configuration>

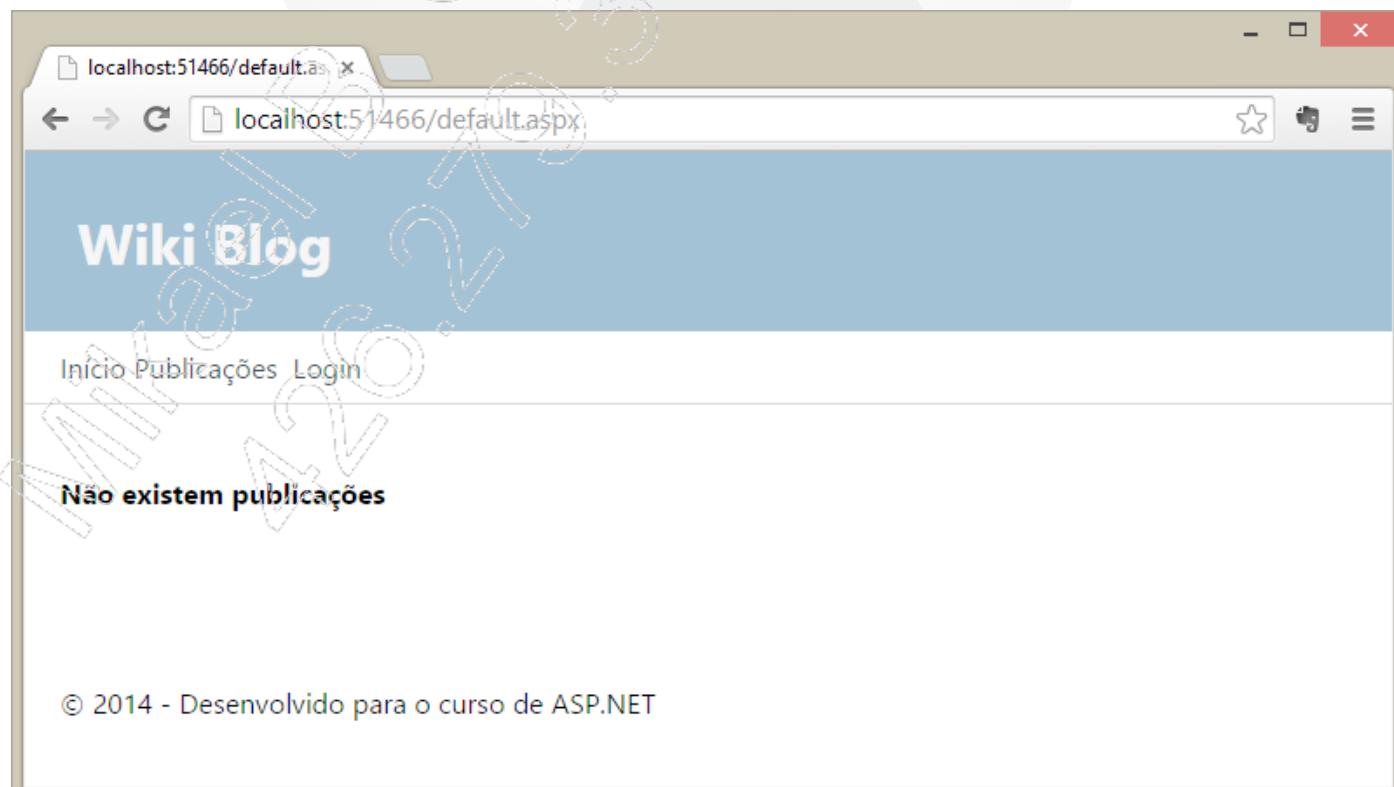
    <appSettings>
        <add key="senha" value="123"/>
    </appSettings>

    <system.web>
        <globalization culture="pt-BR"/>
        <compilation debug="true" targetFramework="4.5.1" />
        <httpRuntime targetFramework="4.5.1" />
    </system.web>
</configuration>
```

46. Teste o aplicativo.

Como resultado, obteremos o seguinte:

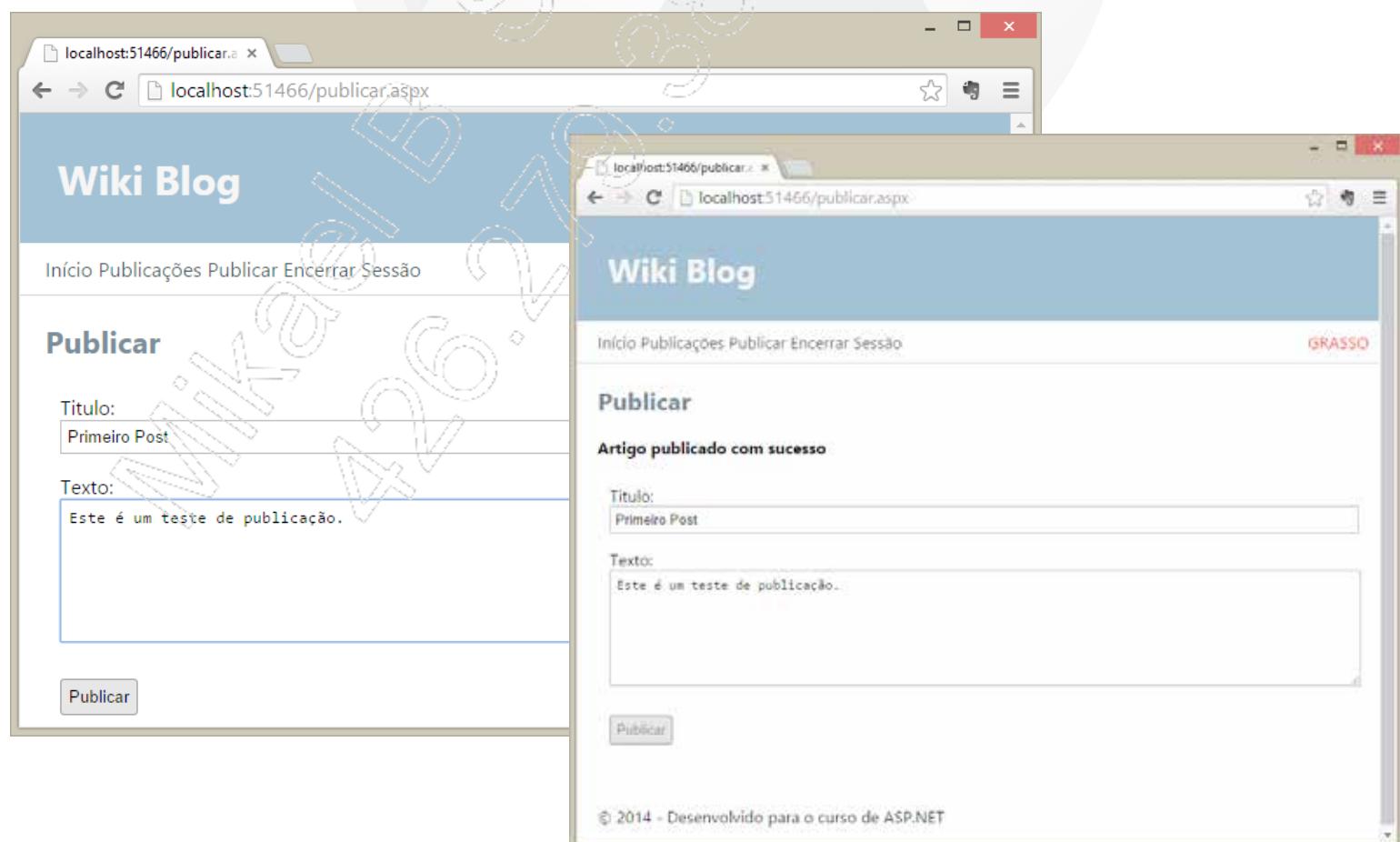
- **Tela inicial**



- Login com a senha definida no Web.Config



- Publicando



Visual Studio 2015 - ASP.NET com C# Acesso a dados

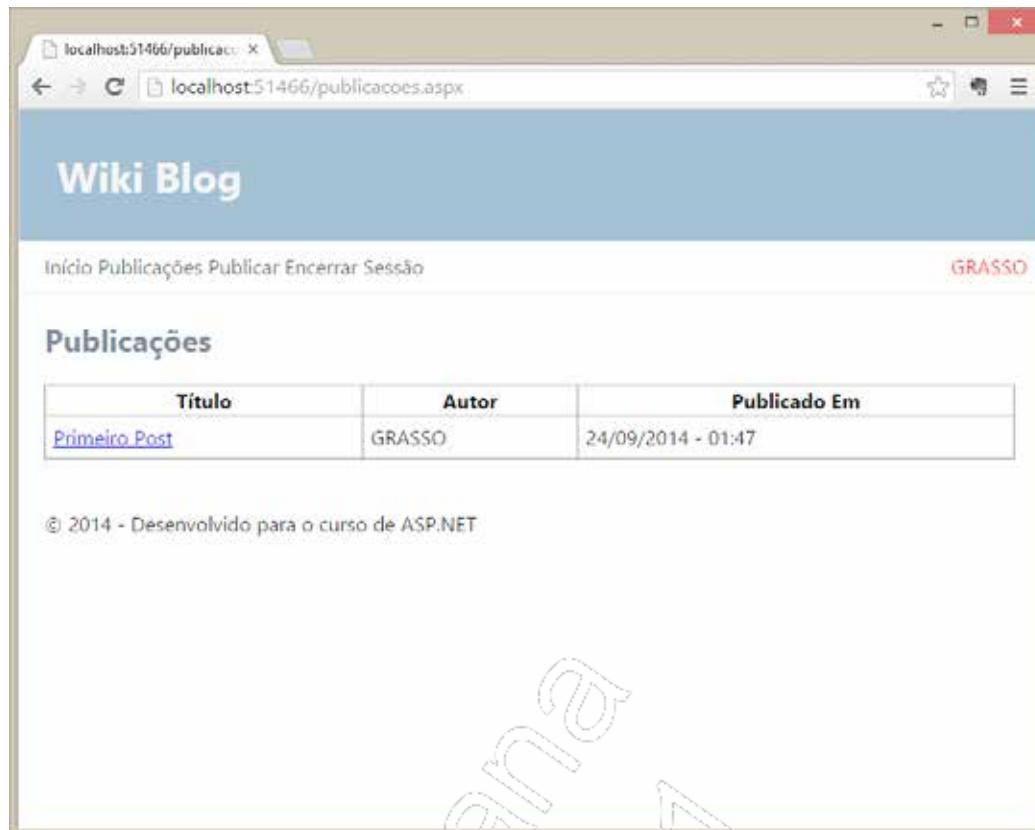
- Vendo a publicação



- Inserindo um comentário

The left window displays a blog post titled "Primeiro Post" by "GRASSO" on "24/07/2014 - 01:47". The post content is "Este é um teste de publicação.". Below the post is a comment section with a placeholder "Insira seu comentário:" and a name input field labeled "Nome:". The right window shows the same blog post after a comment has been added. The comment is listed under "Comentários:" with the name "Maria" and the date "24/07/2014 - 01:48". The comment text is "Muito bom!".

- Vendo a lista de publicações



- Excluindo um comentário

The image displays two side-by-side screenshots of a web browser showing a blog post. The left screenshot shows the full blog post with a comment. The right screenshot shows the same post after the comment has been deleted.

Left Screenshot (Initial State):

- The blog post title is "Primeiro Post".
- The author is "GRASSO".
- The publication date is "24/09/2014 - 01:47".
- The post content is "Este é um teste de publicação."
- The comment section shows a comment by "Maria" from "24/09/2014 - 01:48" with the text "Muito bom!".
- A button labeled "Excluir comentário" is visible.
- An input field for "Insira seu comentário:" is present.

Right Screenshot (After Deletion):

- The blog post title is "Primeiro Post".
- The author is "GRASSO".
- The publication date is "24/09/2014 - 01:47".
- The post content is "Este é um teste de publicação."
- The comment section is empty, showing only the heading "Comentários:".

- **Encerrando a sessão**



Sugestões para estudo:

- Incluir uma aprovação de comentários (não publicar de imediato);
- Fazer upload de foto para o artigo;
- Fazer upload de foto (avatar) para os usuários que comentem;
- Configurar para que o usuário consiga apagar apenas os seus comentários;
- Configurar para que só o usuário que criou o texto possa editá-lo;
- Configurar para que o usuário registre nome e senha na primeira vez e, depois, use essa informação para fazer login;
- Mudar a camada de acesso a dados para armazenar em um banco de dados relacional.