TREINAMENTOS

Desenvolvimento Web com ASP.NET MVC 3

Desenvolvimento Web com ASP.NET MVC

22 de junho de 2011

Sumário

1 Banco de dados **1**
- 1.1 Sistemas gerenciadores de banco de dados . . . 1
- 1.2 SQL Server . . . 2
- 1.3 Bases de dados (*Databases*) . . . 2
  - 1.3.1 Criando uma base de dados no SQL Server Express . . . 2
- 1.4 Tabelas . . . 4
  - 1.4.1 Criando tabelas no SQL Server Express . . . 5
- 1.5 Operações Básicas . . . 8
- 1.6 Chaves Primária e Estrangeira . . . 10
- 1.7 Consultas Avançadas . . . 11
- 1.8 Exercícios . . . 11

2 ADO.NET **29**
- 2.1 Driver . . . 29
- 2.2 ODBC . . . 30
- 2.3 ODBC Manager . . . 30
- 2.4 Criando uma conexão . . . 31
- 2.5 Inserindo registros . . . 32
- 2.6 Exercícios . . . 32
- 2.7 SQL Injection . . . 33
- 2.8 Exercícios . . . 34
- 2.9 Listando registros . . . 35
- 2.10 Exercícios . . . 36
- 2.11 Fábrica de conexões (Factory) . . . 37
- 2.12 Exercícios . . . 38

3 Entity Framework 4.1 **41**
- 3.1 Múltiplas sintaxes da linguagem SQL . . . 41
- 3.2 Orientação a Objetos VS Modelo Entidade Relacionamento . . . 41
- 3.3 Ferramentas ORM . . . 41
- 3.4 Configuração . . . 42
- 3.5 Mapeamento . . . 42
- 3.6 Gerando o banco . . . 46
- 3.7 Exercícios . . . 47
- 3.8 Sobrescrevendo o nome do Banco de Dados . . . 50
- 3.9 Exercícios . . . 51

3.10 Manipulando entidades . . . 51
3.10.1 Persistindo . . . 51
3.10.2 Buscando . . . 51
3.10.3 Removendo . . . 52
3.10.4 Atualizando . . . 52
3.10.5 Listando . . . 52
3.11 Exercícios . . . 52
3.12 Repository . . . 53
3.13 Exercícios . . . 54

4 Visão Geral do ASP .NET MVC **57**
4.1 Necessidades de uma aplicação web . . . 57
4.2 Visão Geral do ASP .NET MVC . . . 57
4.3 Aplicação de exemplo . . . 58
4.3.1 Testando a aplicação . . . 59
4.3.2 Trocando a porta do servidor . . . 59
4.4 Página de Saudação . . . 59
4.5 Exercícios . . . 60
4.6 Alterando a página inicial . . . 60
4.7 Exercícios . . . 60
4.8 Integrando o Banco de Dados . . . 60
4.9 Exercícios . . . 61
4.10 Listando entidades . . . 63
4.11 Exercícios . . . 65
4.12 Inserindo entidades . . . 65
4.13 Exercícios . . . 68
4.14 Alterando entidades . . . 70
4.14.1 Link de alteração . . . 74
4.15 Exercícios . . . 75
4.16 Removendo entidades . . . 77
4.17 Exercícios . . . 80

5 Tratamento de Erros **81**
5.1 Try-Catch . . . 82
5.2 Exercícios . . . 83
5.3 Custom Errors . . . 83
5.4 Exercícios . . . 84
5.5 Erros do HTTP . . . 84
5.6 Exercícios . . . 86

6 Camada de Apresentação (View) **87**
6.1 Inline Code . . . 87
6.2 Utilizando Inline Code . . . 89
6.3 ViewData, ViewBag e Model . . . 91
6.4 HTML Helpers . . . 93
6.4.1 ActionLink Helper . . . 94

6.4.2 Helpers de Formulários . . . 94
6.4.3 DropDownList Helper . . . 97
6.4.4 Editor Helper . . . 98
6.5 Exercícios . . . 103
6.6 Master Pages . . . 106
6.6.1 Conteúdo comum . . . 107
6.6.2 Lacunas . . . 108
6.7 Exercícios . . . 112
6.8 Importação Automática . . . 116
6.9 Exercícios . . . 117
6.10 Dividindo o conteúdo . . . 117
6.11 Exercícios . . . 119

7 Controle (Controller) 121
7.1 Actions . . . 121
7.2 ActionResult . . . 122
7.3 Parâmetros . . . 123
7.3.1 Vários parâmetros . . . 123
7.3.2 Por objeto . . . 124
7.4 Exercícios . . . 124
7.5 TempData . . . 125
7.6 Exercícios . . . 126

8 Rotas 127
8.1 Adicionando uma rota . . . 128
8.2 Definindo parâmetros . . . 128
8.3 Exercícios . . . 128

9 Validação 131
9.1 Controller . . . 131
9.2 View . . . 132
9.3 Exercícios . . . 134
9.4 Anotações . . . 138
9.4.1 Required . . . 138
9.4.2 Alterando a mensagem . . . 139
9.4.3 Outros validadores . . . 139
9.5 Validação no lado do Cliente . . . 140
9.6 Exercícios . . . 140

10 Sessão 145
10.1 Sessão . . . 145
10.1.1 Identificando o Usuário . . . 145
10.2 Utilizando Session no ASP.NET . . . 146
10.3 Session Mode . . . 147
10.4 Exercícios . . . 148

11 Filtros **157**
11.1 Filtro de Autenticação . . . 157
11.2 Exercícios . . . 159
11.3 *Action Filters* . . . 159
11.4 Exercícios . . . 160

12 Projeto **161**
12.1 Modelo . . . 161
12.2 Exercícios . . . 161
12.3 Persistência - Mapeamento . . . 162
12.4 Exercícios . . . 162
12.5 Persistência - Configuração . . . 163
12.6 Exercícios . . . 163
12.7 Persistência - Repositórios . . . 164
12.8 Exercícios . . . 164
12.8.1 Unit of Work . . . 166
12.9 Exercícios . . . 166
12.10 Apresentação - Template . . . 168
12.11 Exercícios . . . 168
12.12 Cadastrando e Listando Seleções . . . 170
12.13 Exercícios . . . 170
12.14 Removendo Seleções . . . 173
12.15 Exercícios . . . 173
12.16 Cadastrando, Listando e Removendo Jogadores . . . 175
12.17 Exercícios . . . 175
12.18 Removendo Jogadores . . . 178
12.19 Exercícios . . . 178
12.20 Membership e Autorização . . . 181
12.21 Exercícios . . . 181
12.21.1 Adicionando um Usuário Administrador com ASP .NET Configuration 189
12.22 Exercícios . . . 189
12.22.1 Autorização Role-based . . . 191
12.23 Exercícios . . . 191
12.24 Controle de Erro . . . 193
12.25 Exercícios . . . 193
12.26 Enviando email . . . 194
12.27 Exercícios . . . 194

Capítulo 1

Banco de dados

O nosso objetivo é desenvolver aplicações em **C#**. Essas aplicações necessitam armazenar as informações relacionadas ao seu domínio em algum lugar. Por exemplo, uma aplicação de gerenciamento de uma livraria deve armazenar os dados dos livros que ela comercializa. Uma forma de suprir essa necessidade seria armazenar essas informações em arquivos. Contudo, alguns fatores importantes nos levam a descartar tal opção.

A seguir, apresentamos as principais preocupações a serem consideradas ao trabalhar com dados:

Segurança: As informações potencialmente confidenciais devem ser controladas de forma que apenas usuários e sistemas autorizados tenham acesso a elas.

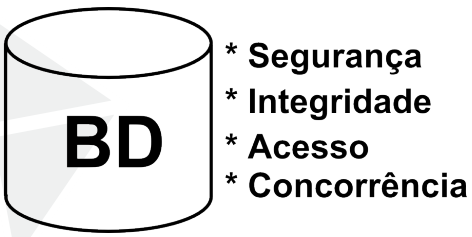
Integridade: Eventuais falhas de software ou hardware não devem corromper os dados.

Acesso: As funções de consulta e manipulação dos dados devem ser implementadas.

Concorrência: Usualmente, diversos sistemas e usuários acessarão as informações de forma concorrente. Apesar disso, os dados não podem ser perdidos ou corrompidos.

Considerando todos esses aspectos, concluímos que seria necessária a utilização de um sistema complexo para manusear as informações das nossas aplicações.

Felizmente, tal tipo de sistema já existe e é conhecido como **Sistema Gerenciador de Banco de Dados** (SGBD).



1.1 Sistemas gerenciadores de banco de dados

No mercado, há diversas opções de sistemas gerenciadores de banco de dados. A seguir, apresentamos os mais populares:

- Oracle
- SQL Server
- SQL Server Express
- PostgreSQL

1.2 SQL Server

Neste treinamento, utilizaremos o SQL Server Express, que é mantido pela Microsoft. O SQL Server Express pode ser obtido a partir do site:
http://www.microsoft.com/express/Database/.

Microsoft SQL Server Management Studio Express

Para interagir com o SQL Server Express, utilizaremos um cliente com interface gráfica chamado de Microsoft SQL Server Management Studio Express.

1.3 Bases de dados (*Databases*)

Um sistema gerenciador de banco de dados é capaz de gerenciar informações de diversos sistemas ao mesmo tempo. Por exemplo, as informações dos clientes de um banco, além dos produtos de uma loja virtual.

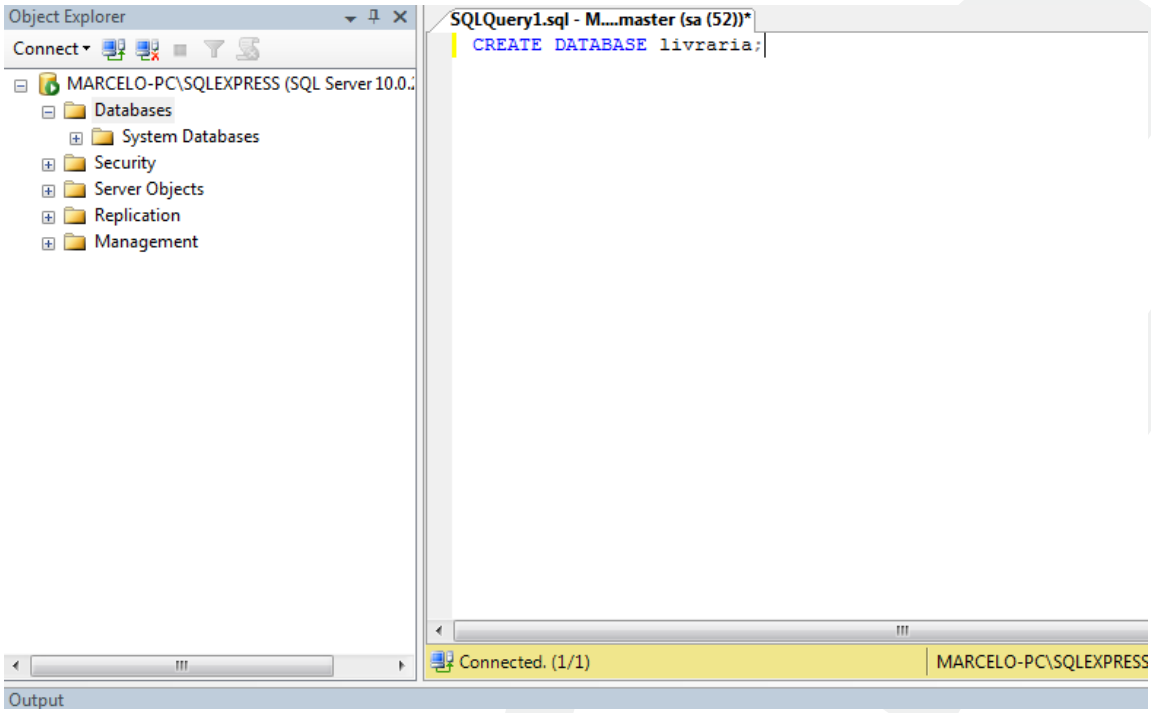
Caso os dados fossem mantidos sem nenhuma separação lógica, a organização ficaria prejudicada. Além disso, seria mais difícil implementar regras de segurança referentes ao acesso dos dados. Tais regras criam restrições quanto ao conteúdo acessível por cada usuário. Determinado usuário, por exemplo, poderia ter permissão de acesso aos dados dos clientes do banco, mas não às informações dos produtos da loja virtual, ou vice-versa.

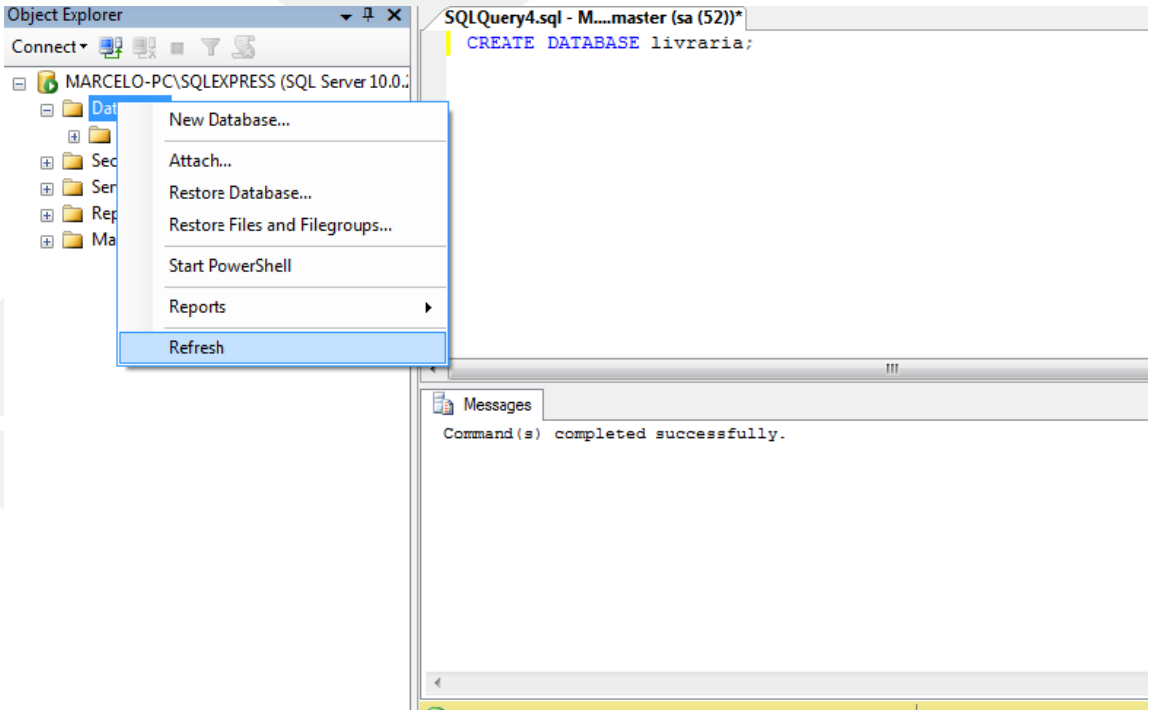
Então, por questões de organização e segurança, os dados devem ser armazenados separadamente no SGBD. Daí surge o conceito de **base de dados** (database).

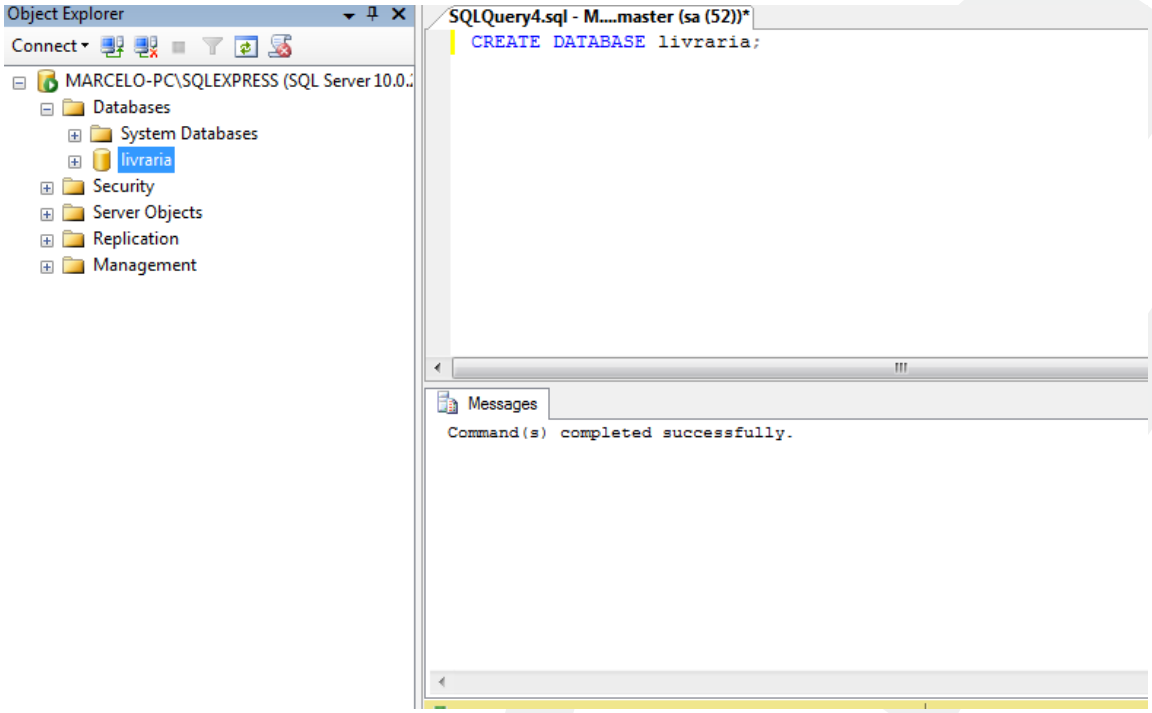
Uma base de dados é um agrupamento lógico das informações de um determinado domínio, como, por exemplo, os dados da nossa livraria.

1.3.1 Criando uma base de dados no SQL Server Express

Para criar uma base de dados no SQL Server Express, utilizamos o comando CREATE DATABASE.


Object Explorer
Connect
MARCELO-PC\SQLEXPRESS (SQL Server 10.0.
Databases
System Databases
Security
Server Objects
Replication
Management
SQLQuery1.sql - M....master (sa (52))*
CREATE DATABASE livraria;
Connected. (1/1)
MARCELO-PC\SQLEXPRESS
Output


Object Explorer
Connect
MARCELO-PC\SQLEXPRESS (SQL Server 10.0.
New Database...
Attach...
Restore Database...
Restore Files and Filegroups...
Start PowerShell
Reports
Refresh
SQLQuery4.sql - M....master (sa (52))*
CREATE DATABASE livraria;
Messages
Command(s) completed successfully.



Repare que além da base de dados **livraria** há outras bases. Essas bases foram criadas automaticamente pelo próprio SQL Server Express para teste ou para guardar algumas configurações.

Quando uma base de dados não é mais necessária, ela pode ser removida através do comando DROP DATABASE.

1.4 Tabelas

Um servidor de banco de dados é dividido em bases de dados com o intuito de separar as informações de sistemas diferentes. Nessa mesma linha de raciocínio, podemos dividir os dados de uma base a fim de agrupá-los segundo as suas correlações. Essa separação é feita através de **tabelas**. Por exemplo, no sistema de um banco, é interessante separar o saldo e o limite de uma conta, do nome e CPF de um cliente. Então, poderíamos criar uma tabela para os dados relacionados às contas e outra para os dados relacionados aos clientes.

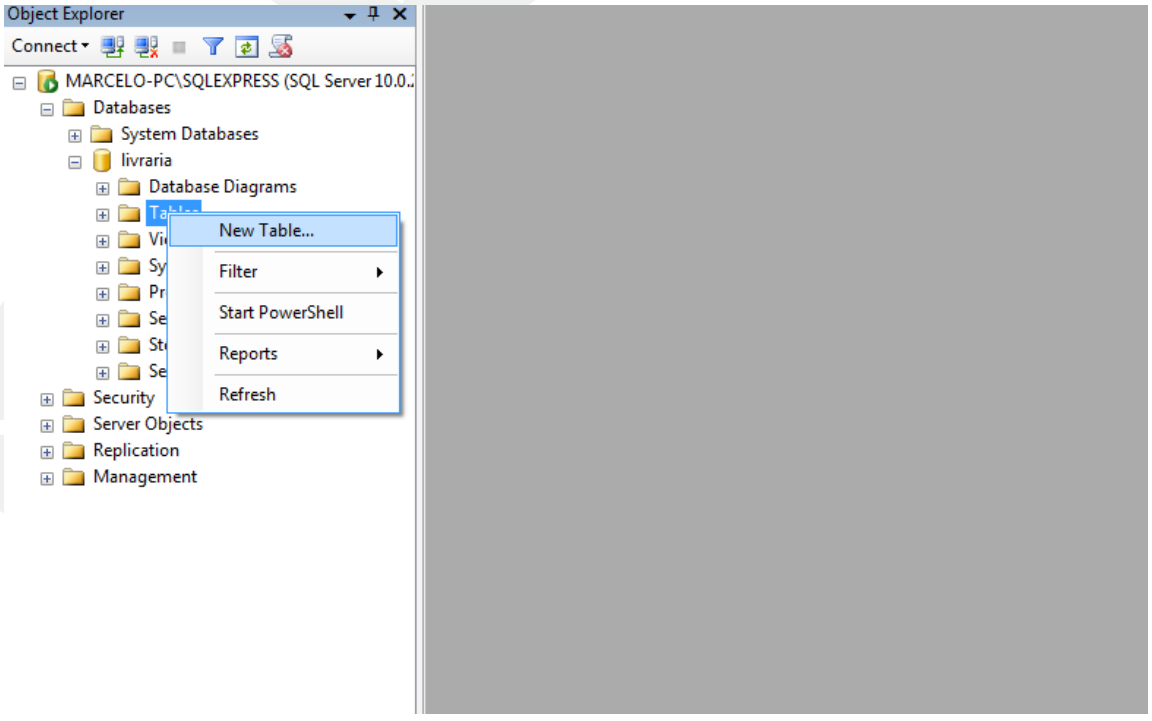
Clientes		
Nome	Idade	Cpf
José	27	31875638735
Maria	32	30045667856

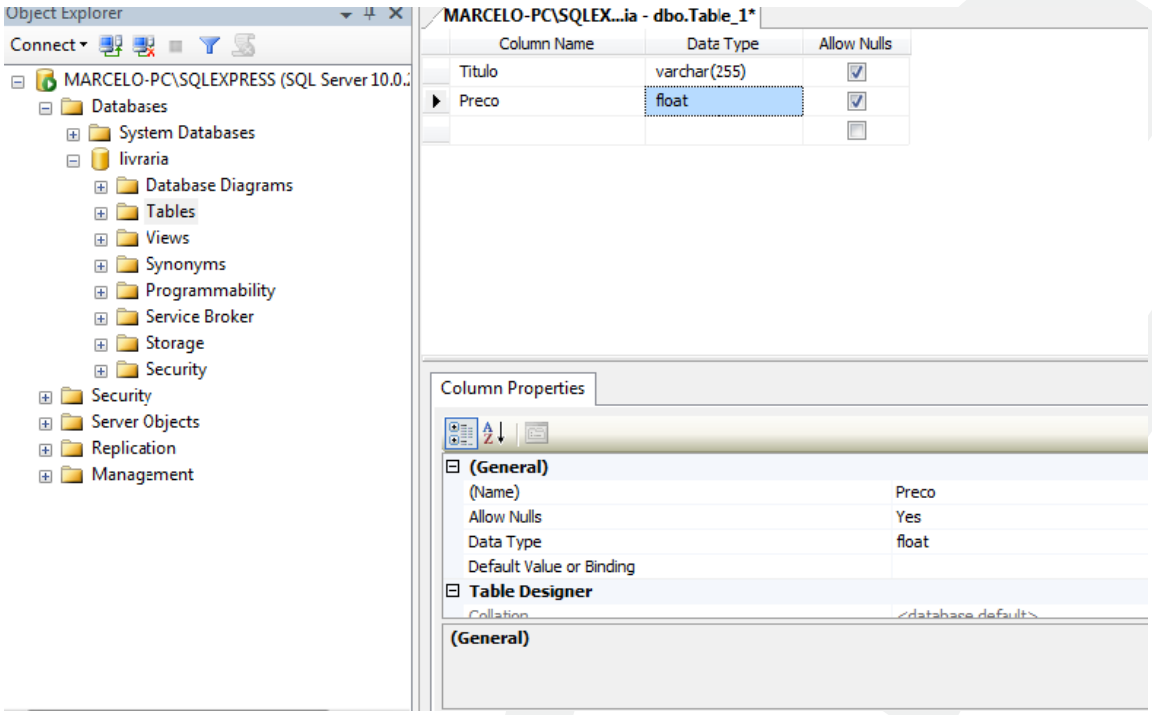
Contas		
Numero	Saldo	Limite
1	1000	500
2	2000	700

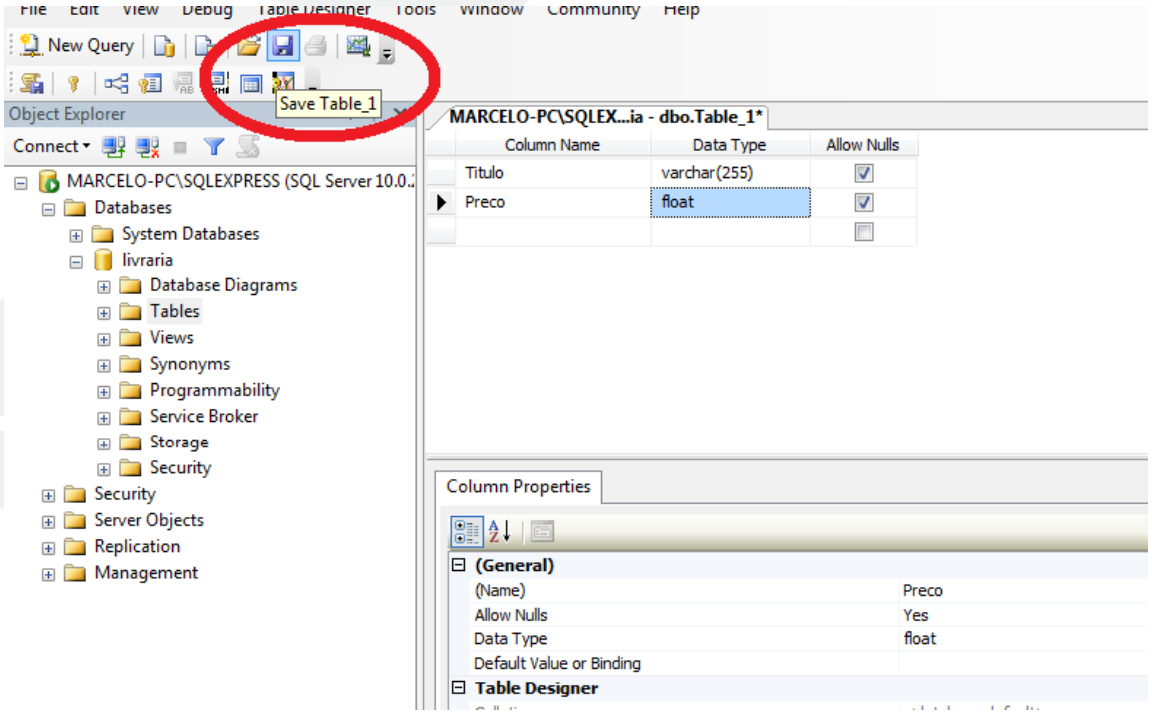
Uma tabela é formada por **registros**(linhas) e os registros são formados por **campos**(colunas). Por exemplo, suponha uma tabela para armazenar as informações dos clientes de um banco. Cada registro dessa tabela armazena em seus campos os dados de um determinado cliente.

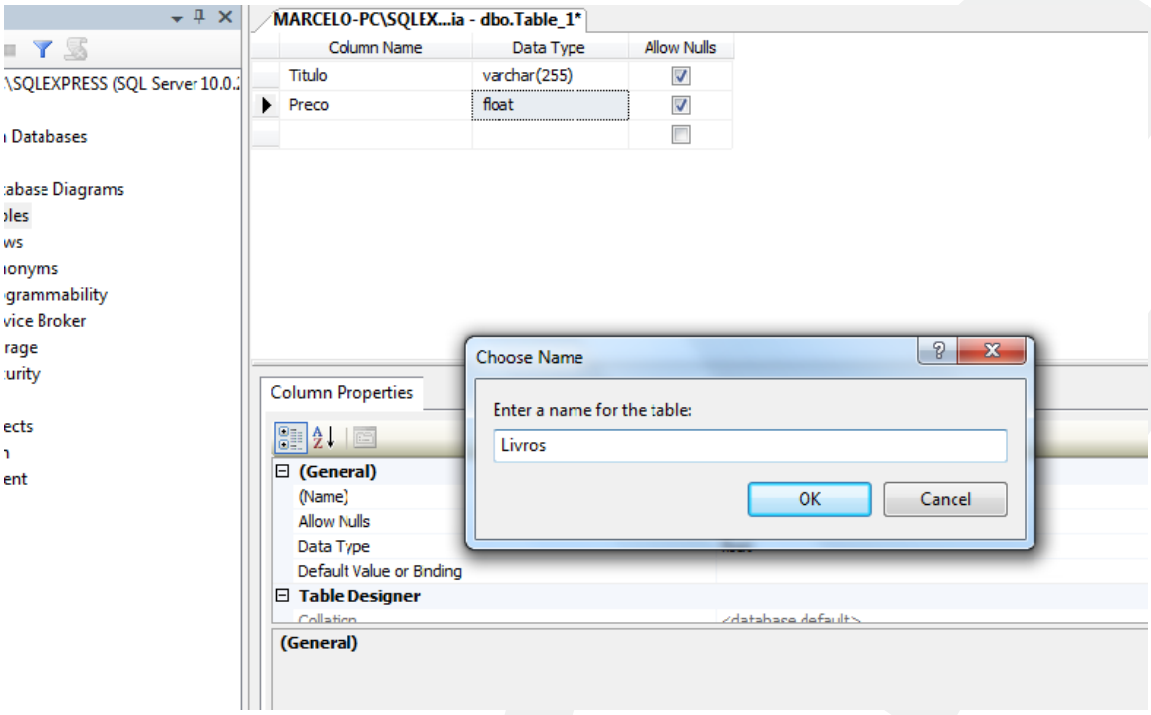
1.4.1 Criando tabelas no SQL Server Express

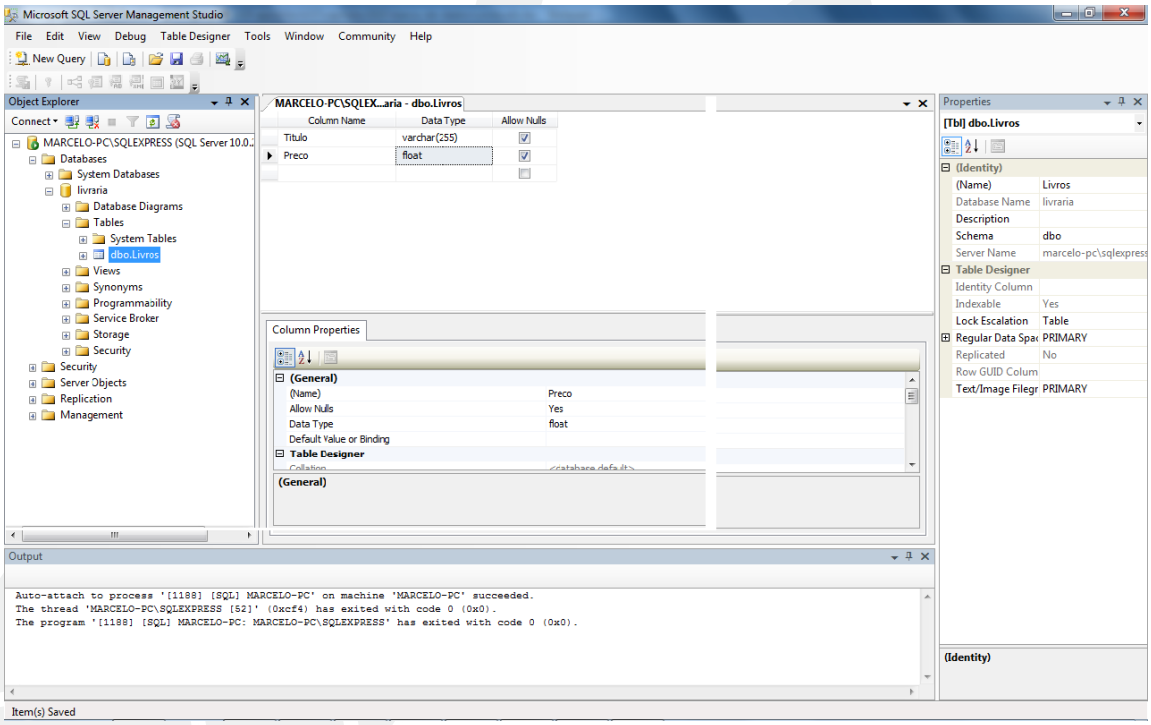
As tabelas no SQL Server Express são criadas através do comando CREATE TABLE. Na criação de uma tabela é necessário definir quais são os nomes e os tipos das colunas.




Object Explorer
Connect
MARCELO-PC\SQLEXPRESS (SQL Server 10.0.
Databases
System Databases
livraria
Database Diagrams
Tables
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
MARCELO-PC\SQLEX...ia - dbo.Table_1*
Column Name
Data Type
Allow Nulls
Titulo
varchar(255)
Preco
float
Column Properties
(General)
(Name)
Preco
Allow Nulls
Yes
Data Type
float
Default Value or Binding
Table Designer
(General)


File Edit View Debug Table Designer Tools Window Community Help
New Query
Save Table_1
Object Explorer
Connect
MARCELO-PC\SQLEXPRESS (SQL Server 10.0.
Databases
System Databases
livraria
Database Diagrams
Tables
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
MARCELO-PC\SQLEX...ia - dbo.Table_1*
Column Name
Data Type
Allow Nulls
Titulo
varchar(255)
Preco
float
Column Properties
(General)
(Name)
Preco
Allow Nulls
Yes
Data Type
float
Default Value or Binding
Table Designer





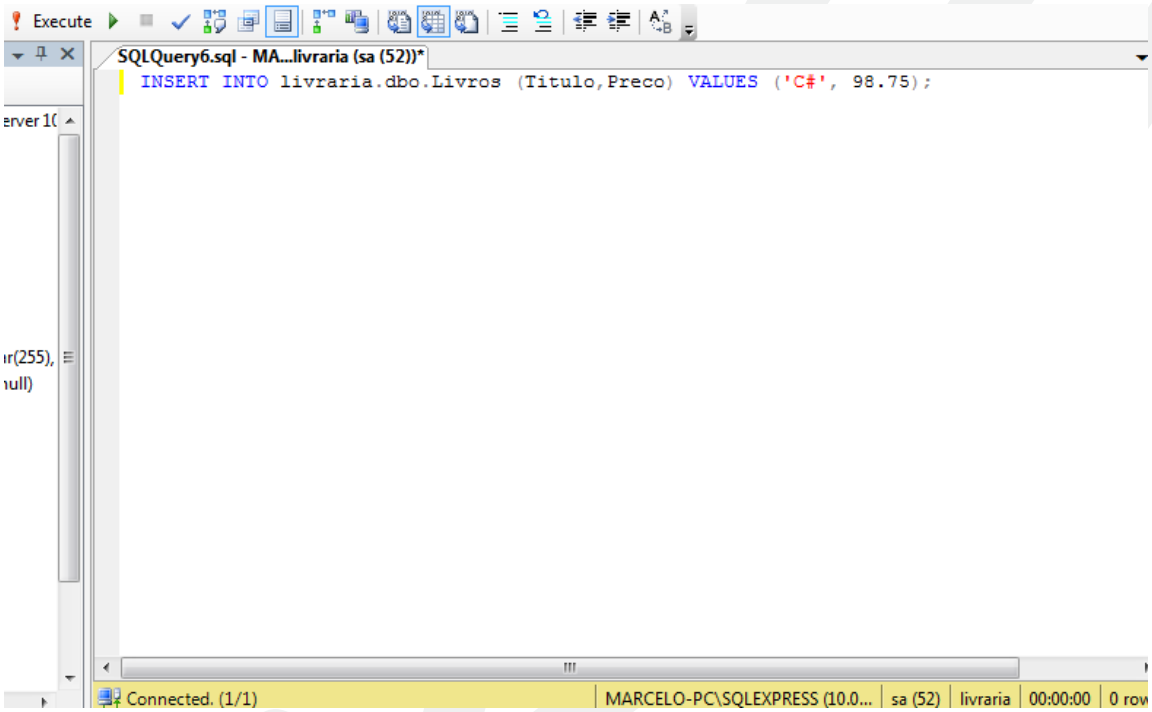
No SQL Server os nomes das tabelas são precedidas pelo ID do usuário que possui a tabela. No caso do usuário **sa**, o ID é **dbo**. Portanto o nome da tabela Livros fica *dbo.Livros*.

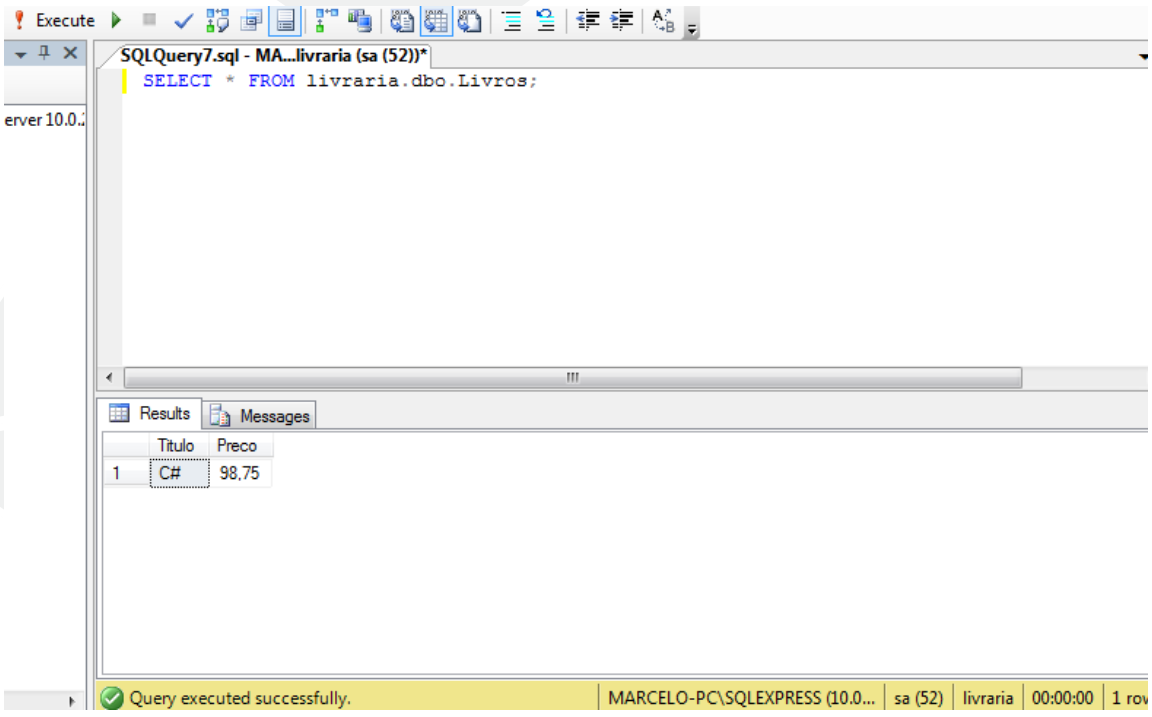
Se uma tabela não for mais desejada ela pode ser removida através do comando DROP TABLE.

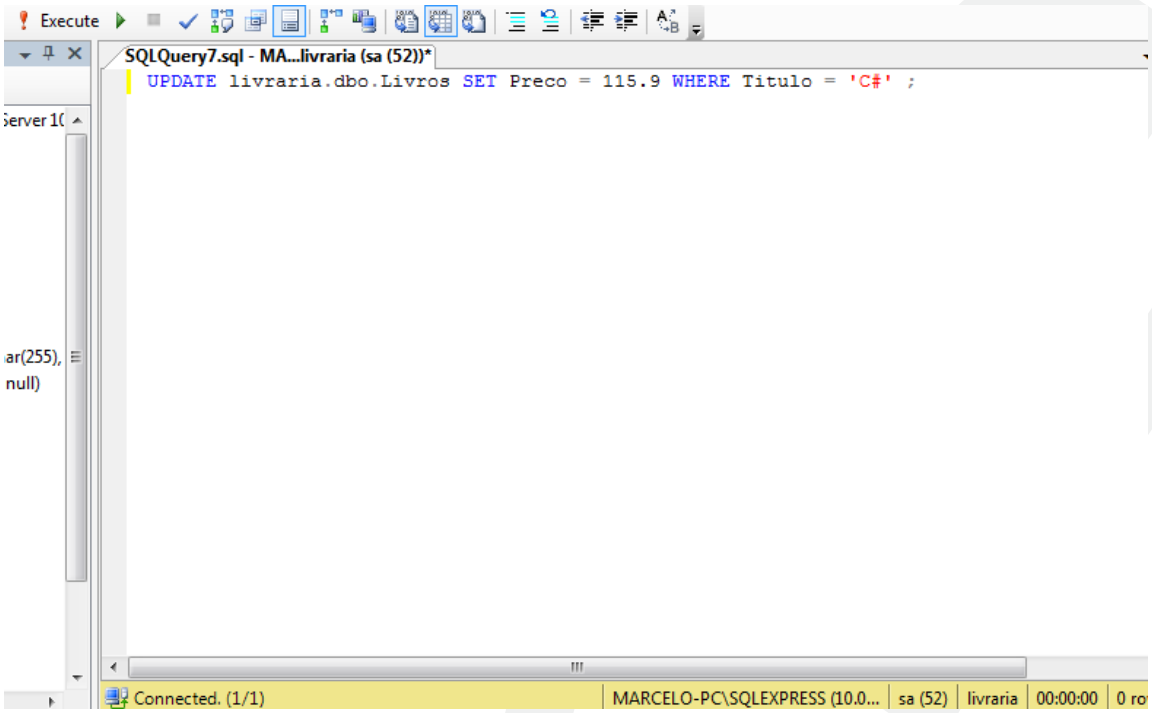
1.5 Operações Básicas

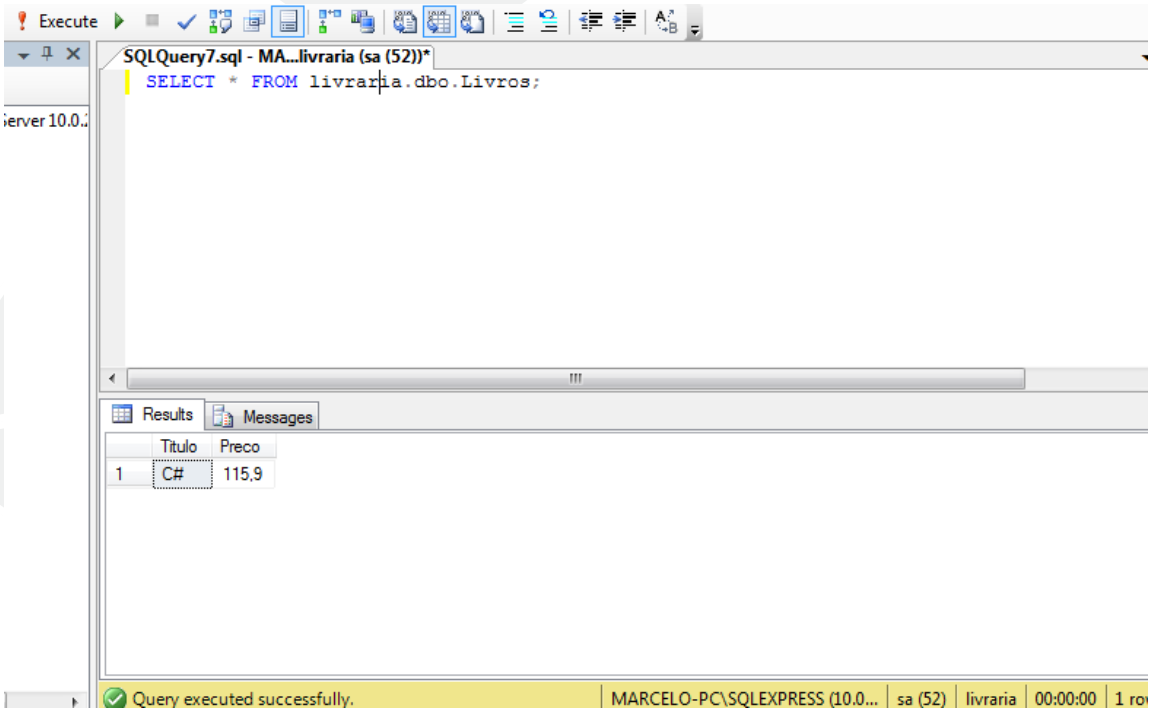
As operações básicas para manipular os dados das tabelas são: inserir, ler, alterar e remover.

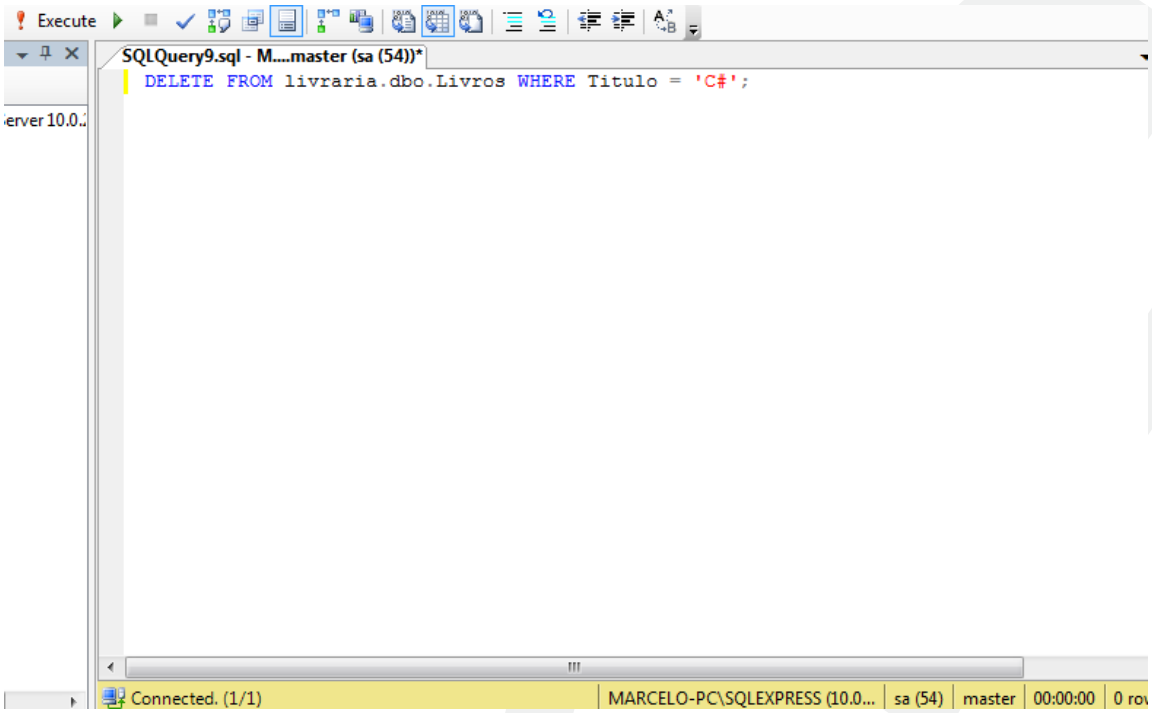
Essas operações são realizadas através da linguagem de consulta denominada **SQL**. Esta linguagem oferece quatro comandos básicos: INSERT, SELECT, UPDATE e DELETE. Estes comandos são utilizados para inserir, ler, alterar e remover registros respectivamente.

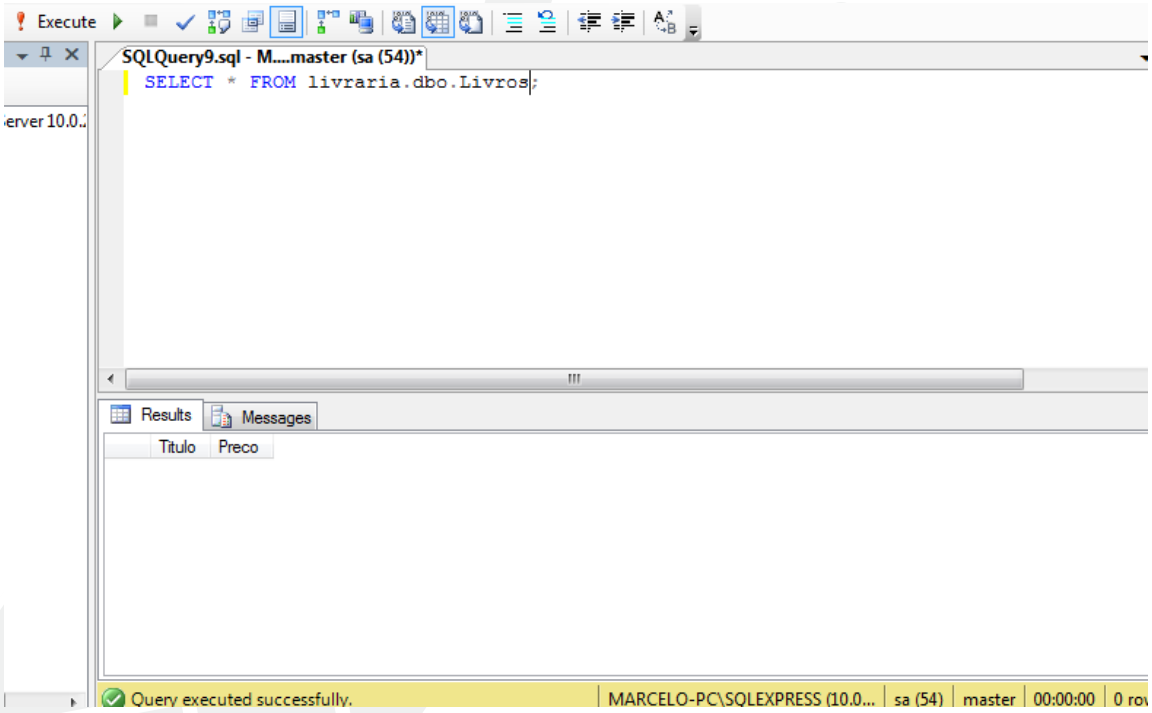





Execute
SQLQuery7.sql - MA...livraria (sa (52))*
UPDATE livraria.dbo.Livros SET Preco = 115.9 WHERE Titulo = 'C#' ;
Connected. (1/1)
MARCELO-PC\SQLEXPRESS (10.0...
sa (52)
livraria
00:00:00


Execute
SQLQuery7.sql - MA...livraria (sa (52))*
SELECT * FROM livraria.dbo.Livros;
Results
Messages
Titulo
Preco
1
C#
115,9
Query executed successfully.
MARCELO-PC\SQLEXPRESS (10.0...
sa (52)
livraria
00:00:00





1.6 Chaves Primária e Estrangeira

Suponha que os livros da nossa livraria são separados por editoras. Uma editora possui nome e telefone. Para armazenar esses dados, uma nova tabela deve ser criada.

Nesse momento, teríamos duas tabelas (Livros e Editoras). Eventualmente, será necessário

descobrir qual é a editora de um determinado livro ou quais são os livros de uma determinada editora. Para isso, os registros da tabela **Editoras** devem estar relacionados aos da tabela **Livros**.

Na tabela Livros, poderíamos adicionar uma coluna para armazenar o nome da editora a qual ele pertence. Dessa forma, se alguém quiser recuperar as informações da editora de um determinado livro, deve consultar a tabela Livros para obter o nome da editora correspondente. Depois, com esse nome, deve consultar a tabela Editoras para obter as informações da editora.

Porém, há um problema nessa abordagem, a tabela Editoras aceita duas editoras com o mesmo nome. Dessa forma, eventualmente, não conseguiríamos descobrir os dados corretos da editora de um determinado livro. Para resolver esse problema, deveríamos criar uma restrição na tabela Editoras que proíba a inserção de editoras com o mesmo nome.

Para resolver esse problema no SQL Server Express, poderíamos adicionar a propriedade **UNIQUE** no campo nome da tabela Editoras. Porém ainda teríamos mais um problema: na tabela livro poderíamos adicionar registros com editoras inexistentes, pois não há nenhum vínculo explícito entre as tabelas. Para solucionar estes problemas, devemos utilizar o conceito de **chave primária** e **chave estrangeira**.

Toda tabela pode ter uma chave primária, que é um conjunto de um ou mais campos que devem ser únicos para cada registro. Normalmente, um campo numérico é escolhido para ser a chave primária de uma tabela, pois as consultas podem ser realizadas com melhor desempenho.

Então, poderíamos adicionar um campo numérico na tabela Editoras e torná-lo chave primária. Vamos chamar esse campo de **Id**. Na tabela Livros, podemos adicionar um campo numérico chamado **EditoraId** que deve ser utilizado para guardar o valor da chave primária da editora correspondente ao livro. Além disso, o campo EditoraId deve estar explicitamente vinculado com o campo id da tabela Editoras. Para estabelecer esse vínculo o campo EditoraId deve ser uma chave estrangeira associada ao campo Id.

Uma chave estrangeira é um conjunto de uma ou mais colunas de uma tabela que possuem valores iguais aos da chave primária de outra tabela.

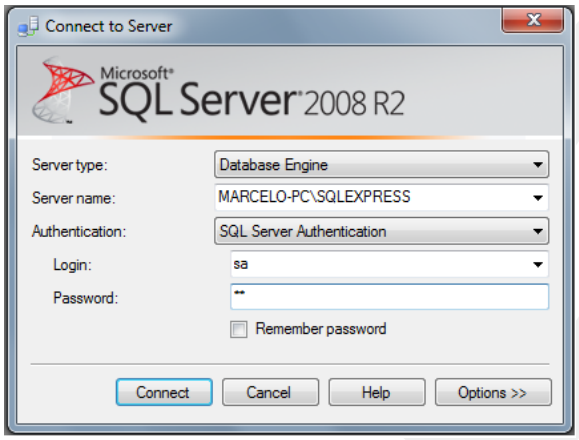
Com a definição da chave estrangeira, um livro não pode ser inserido com o valor do campo EditoraId inválido. Caso tentássemos obteríamos uma mensagem de erro.

1.7 Consultas Avançadas

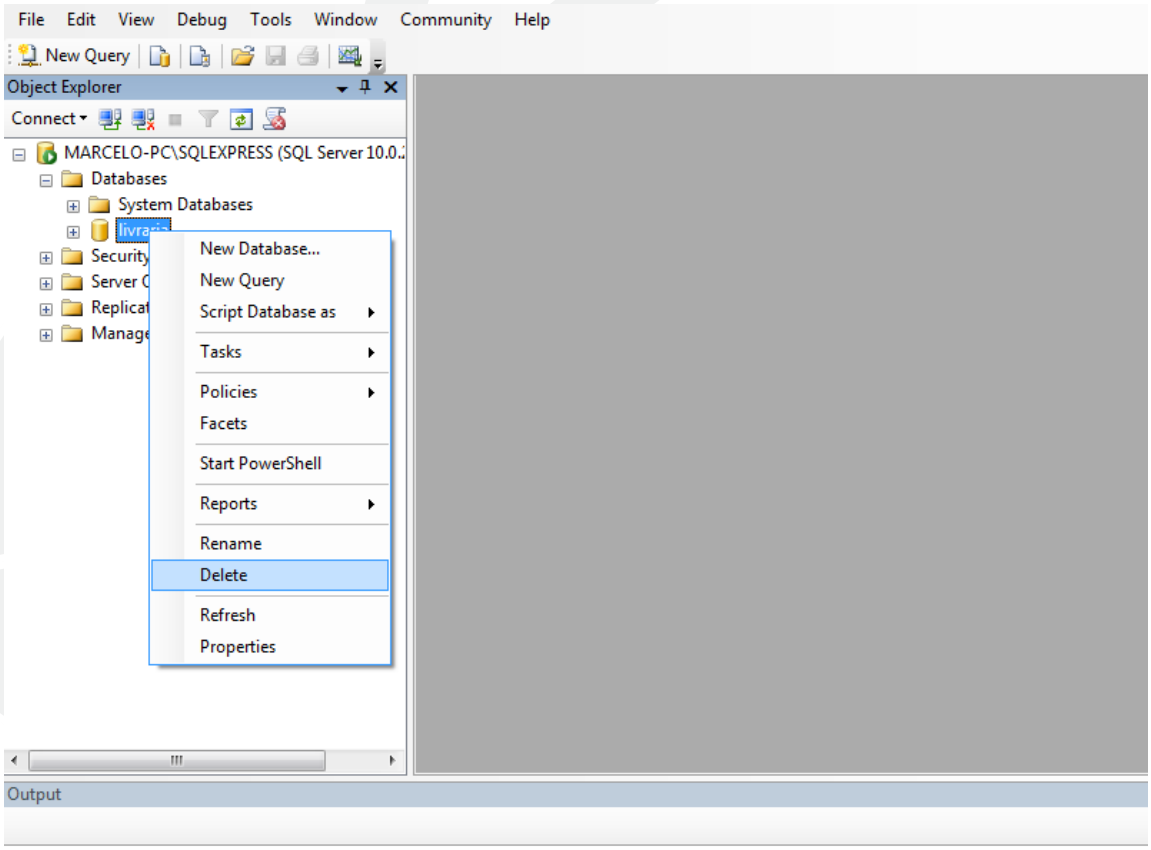
Com o conceito de chave estrangeira, podemos fazer consultas complexas envolvendo os registros de duas ou mais tabelas. Por exemplo, descobrir todos os livros de uma determinada editora.

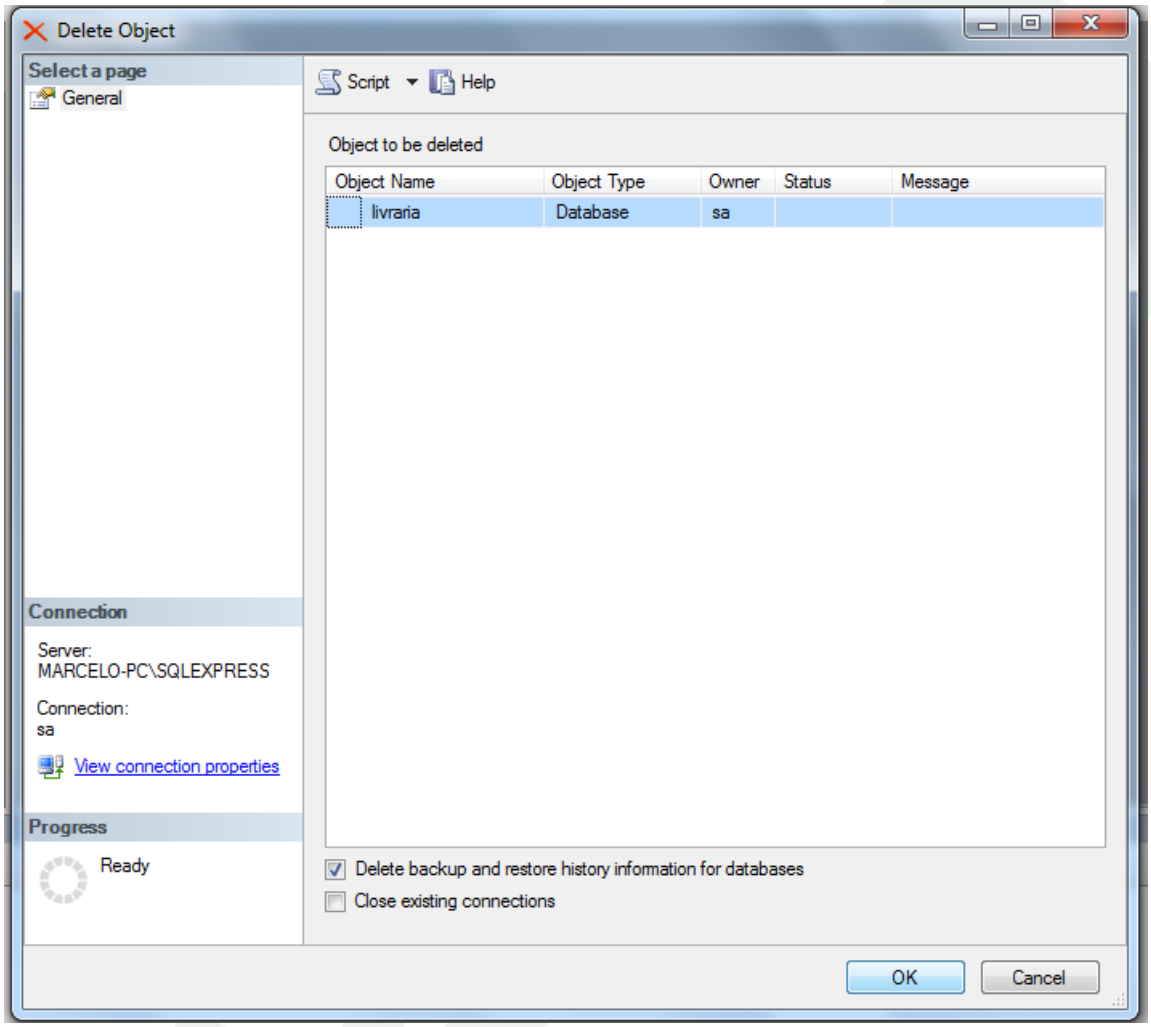
1.8 Exercícios

1. Abra o **Microsoft SQL Server Management Studio Express** utilizando **NOME_DA_MAQUINA SQLEXPRESS** como Server Name, **SQL Server Authentication** como Authentication, **sa** como Login e **sa** como Password.

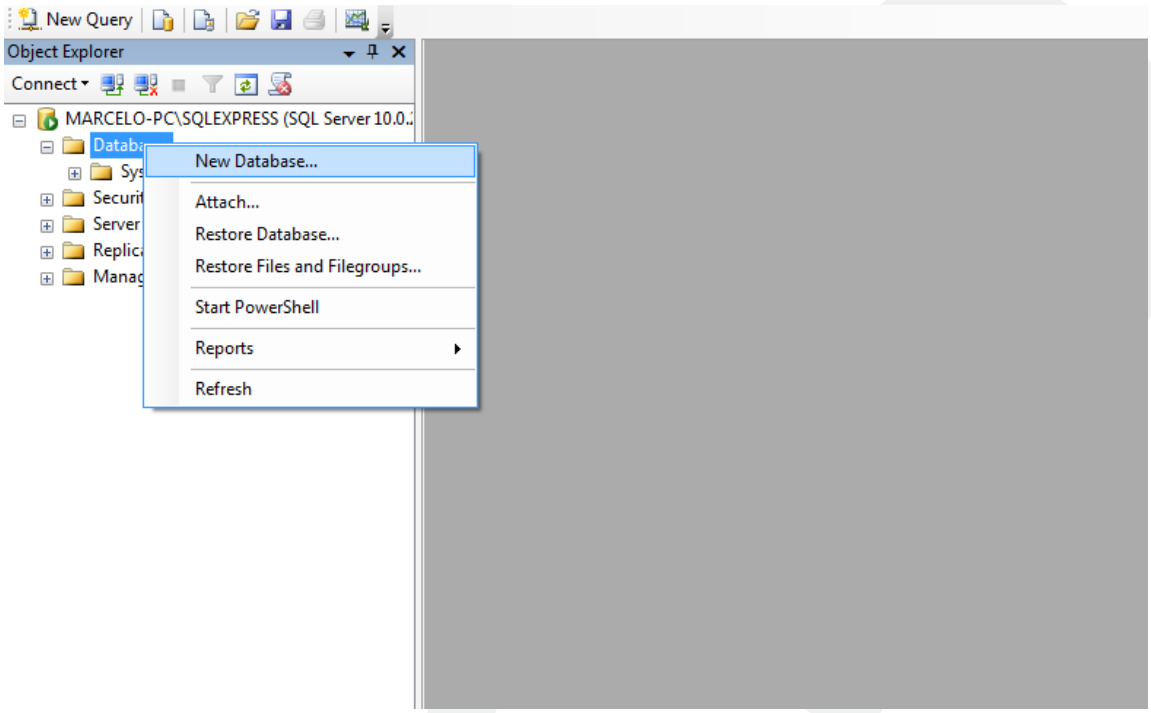


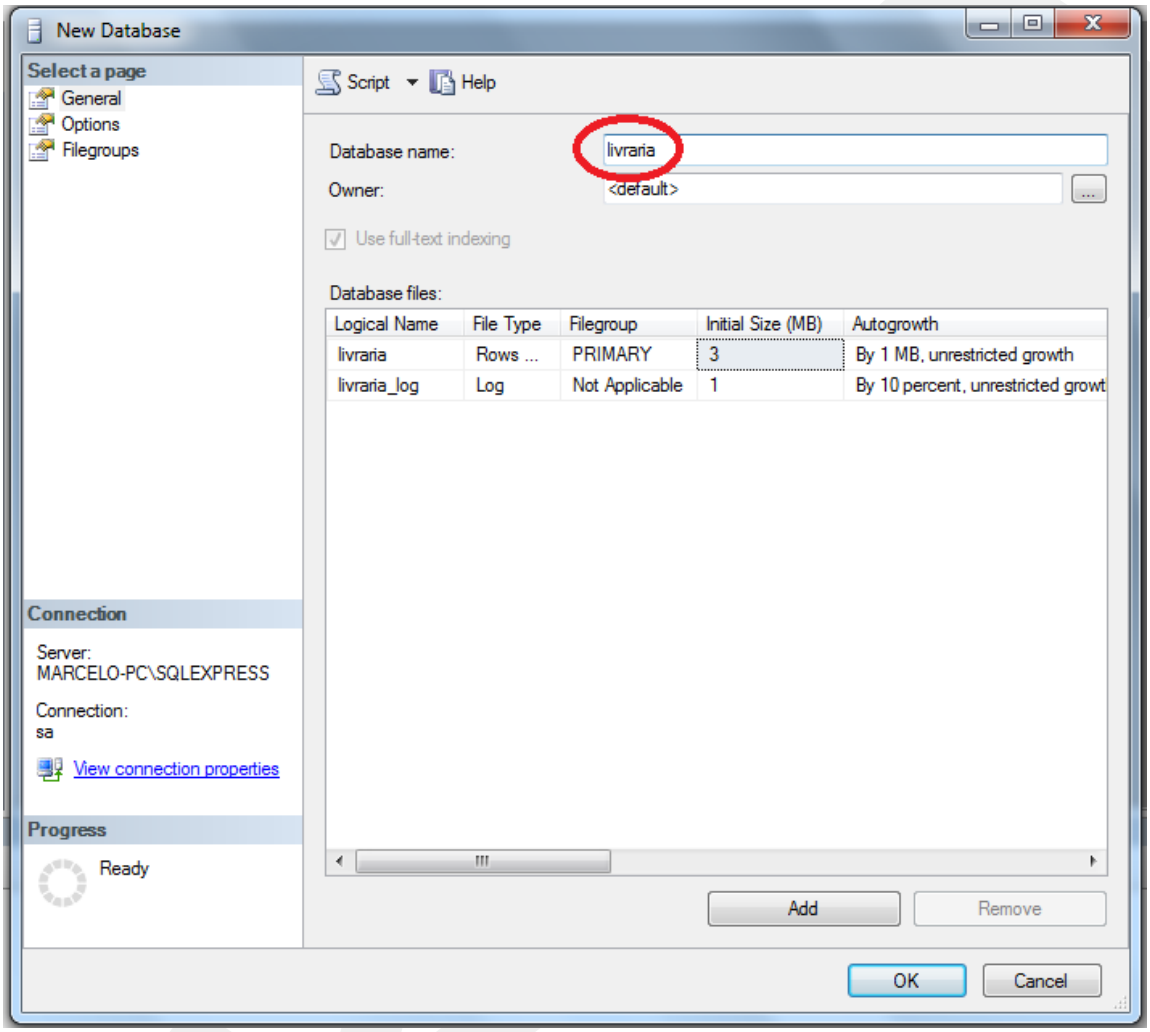
2. Caso exista uma base de dados chamada **Livraria**, remova-a conforme a figura abaixo:



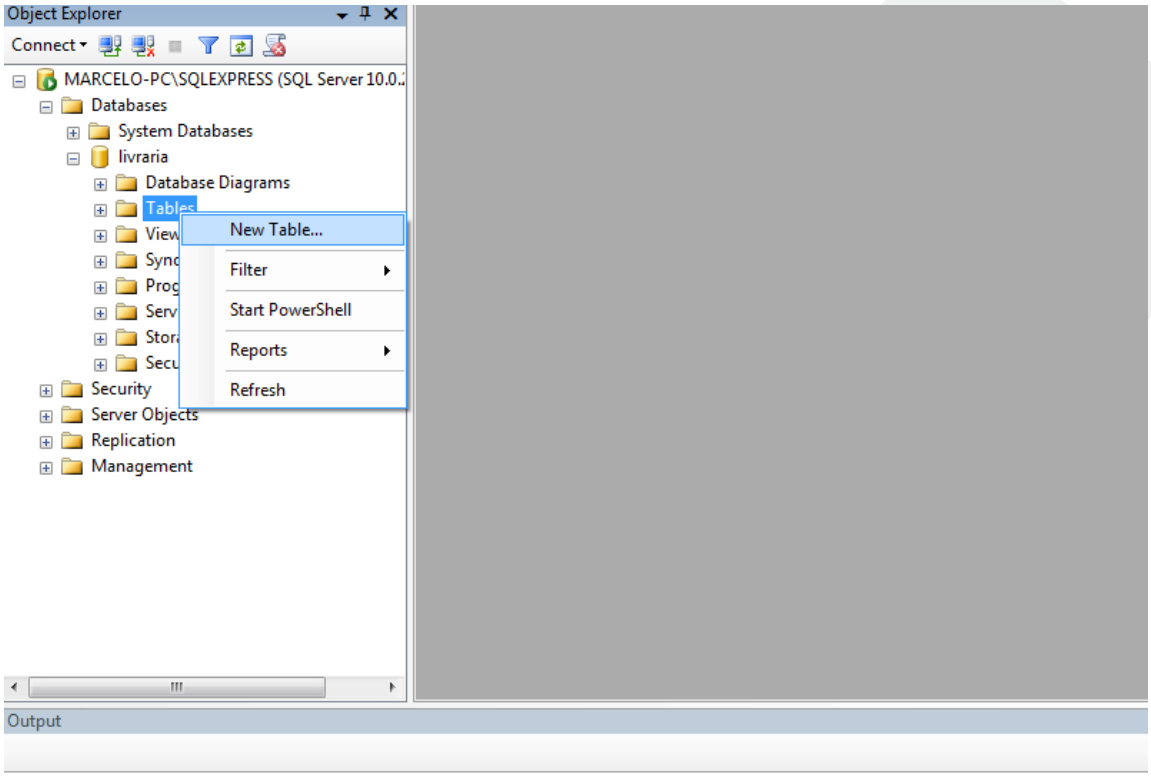


3. Crie uma nova base de dados chamada **livraria**, conforme mostrado na figura abaixo. Você vai utilizar esta base nos exercícios seguintes.

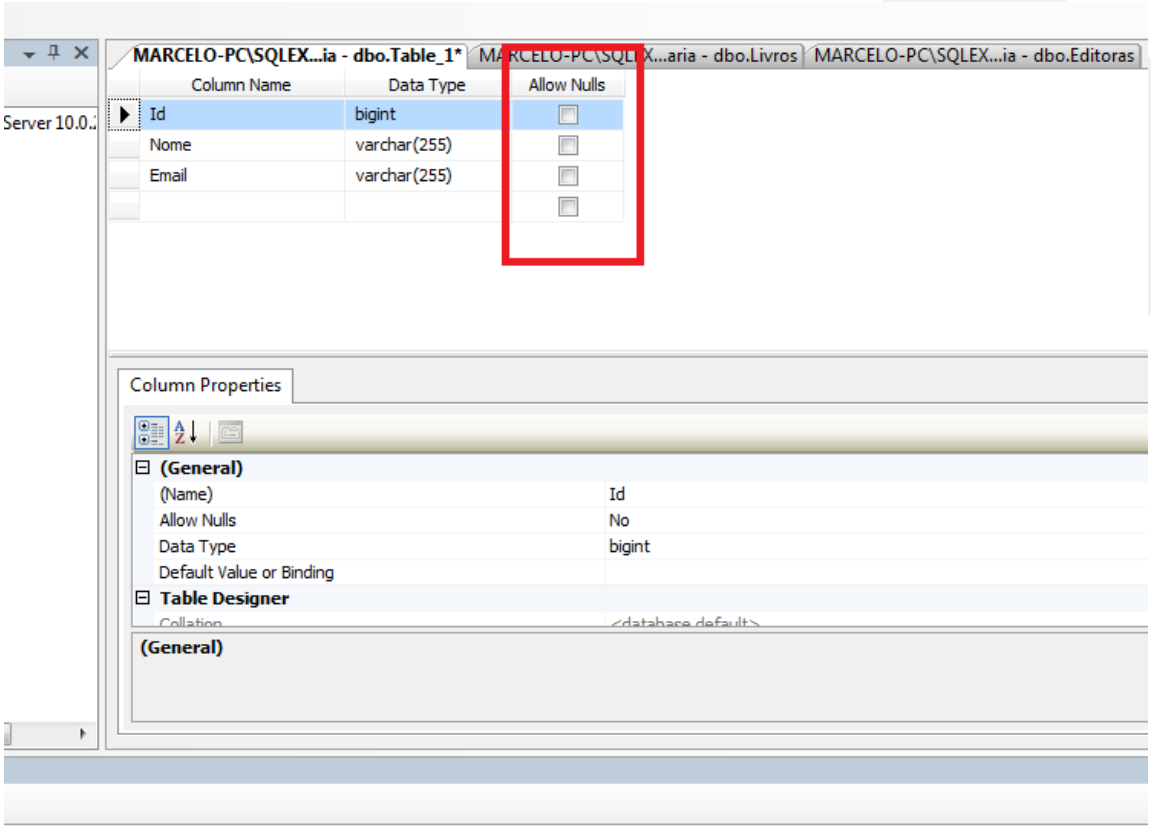

New Query
Object Explorer
Connect
MARCELO-PC\SQLEXPRESS (SQL Server 10.0.
New Database...
Attach...
Restore Database...
Restore Files and Filegroups...
Start PowerShell
Reports
Refresh



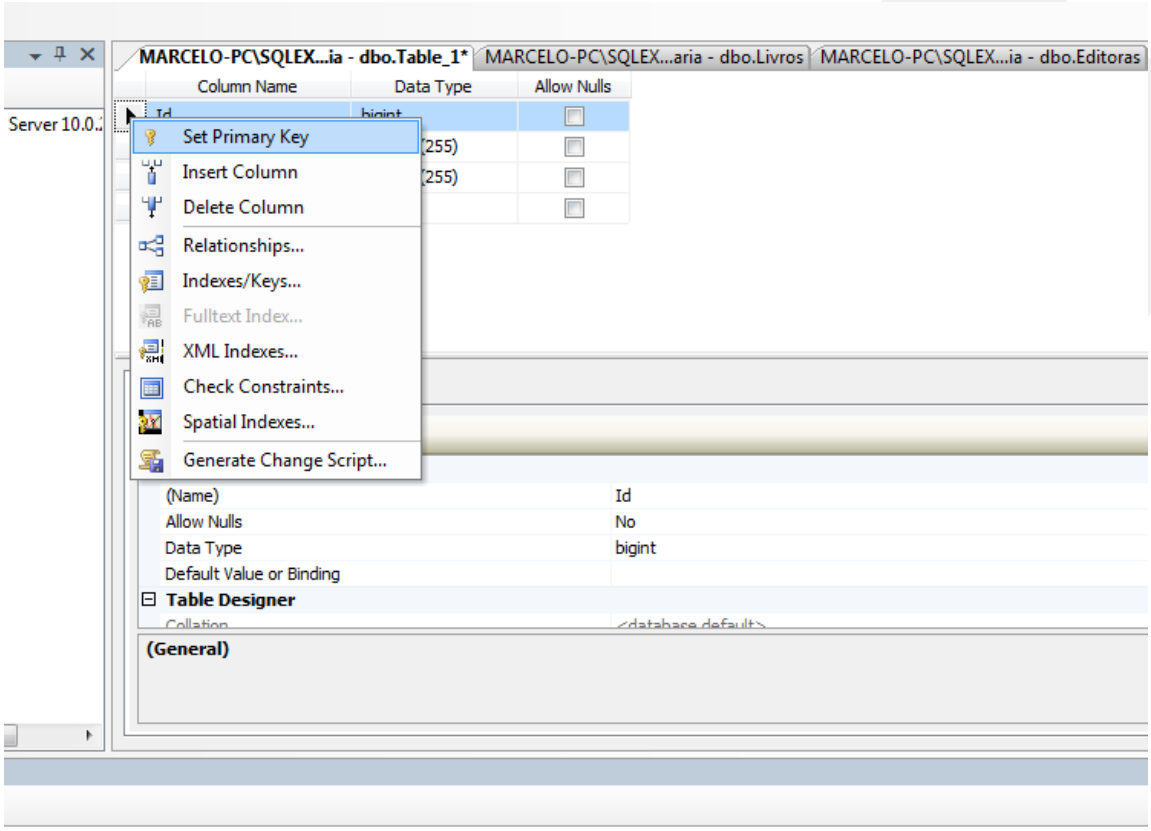
4. Crie uma tabela chamada **Editoras** conforme as figuras abaixo.



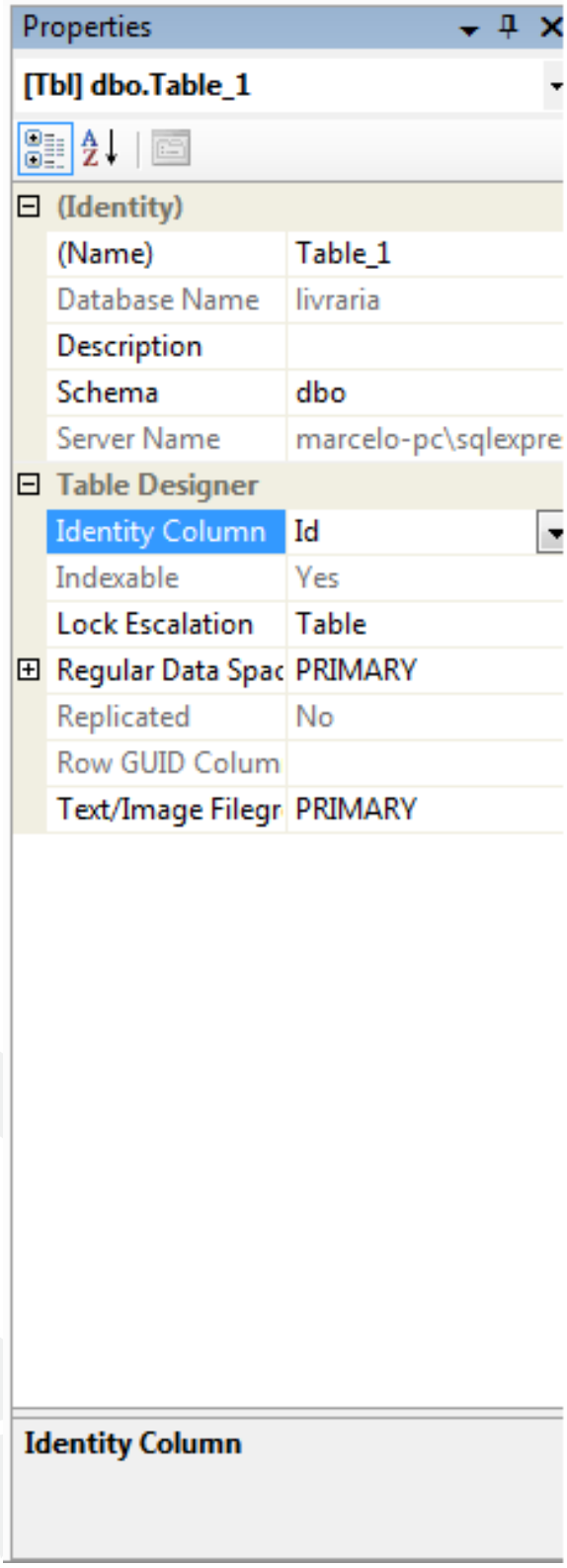
Altere os campos para torná-los obrigatórios, NÃO permitindo que eles fiquem em branco $NULL$.



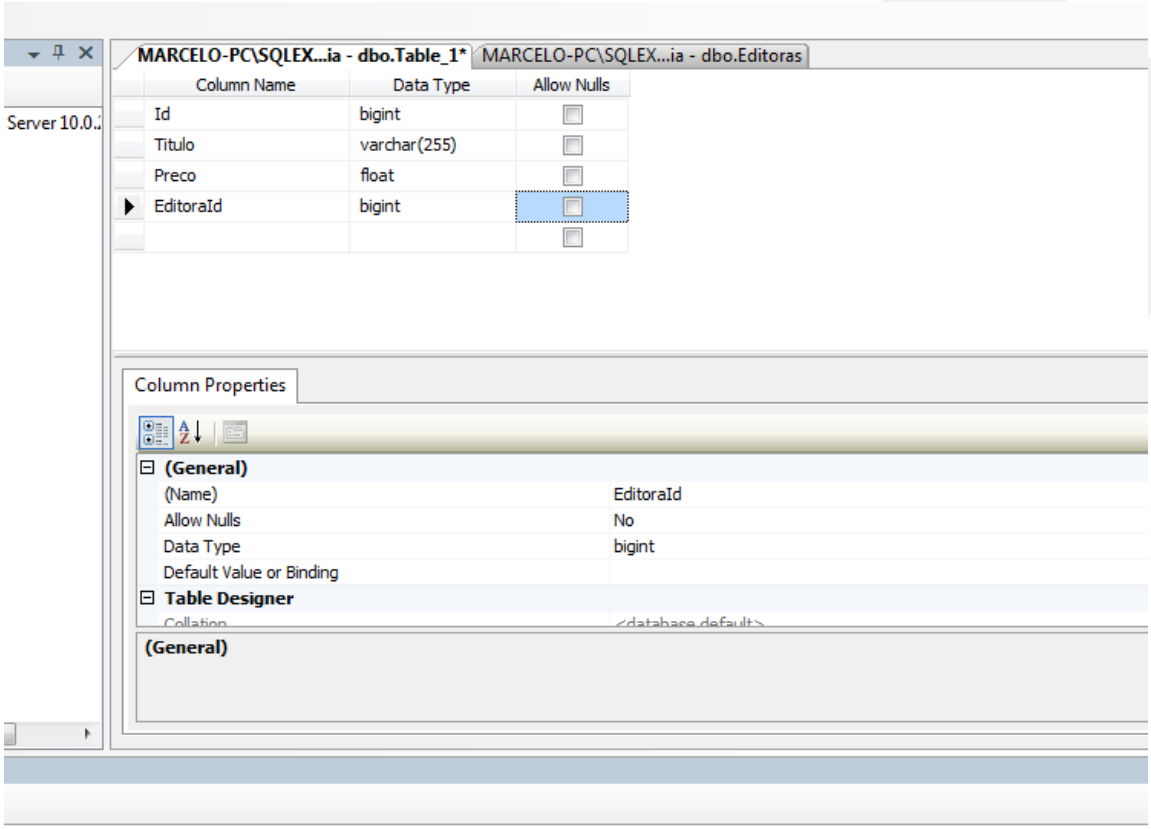
Além disso o campo **Id** deve ser uma chave primária.



O campo **Id** dever ser incrementado automaticamente. Defina ele com a propriedade **Identity** segundo a figura abaixo:

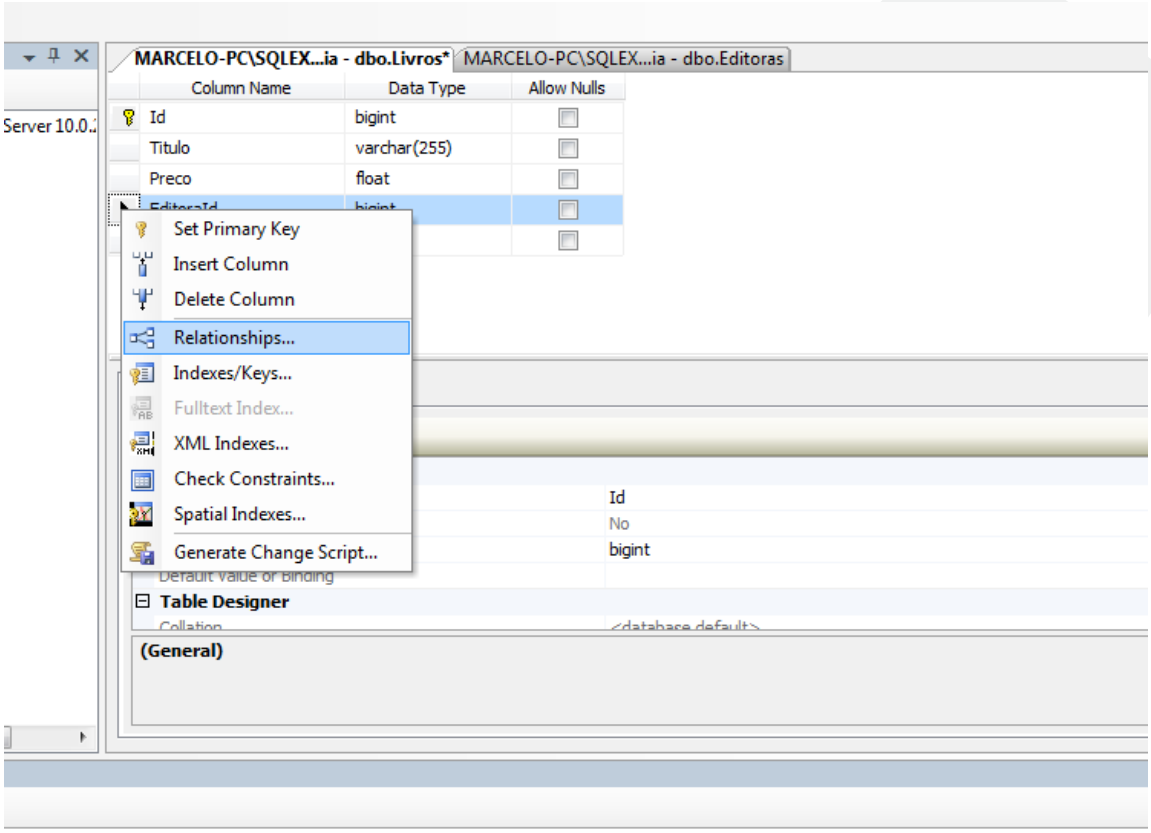


5. Crie uma tabela chamada **Livros** conforme as figuras abaixo:

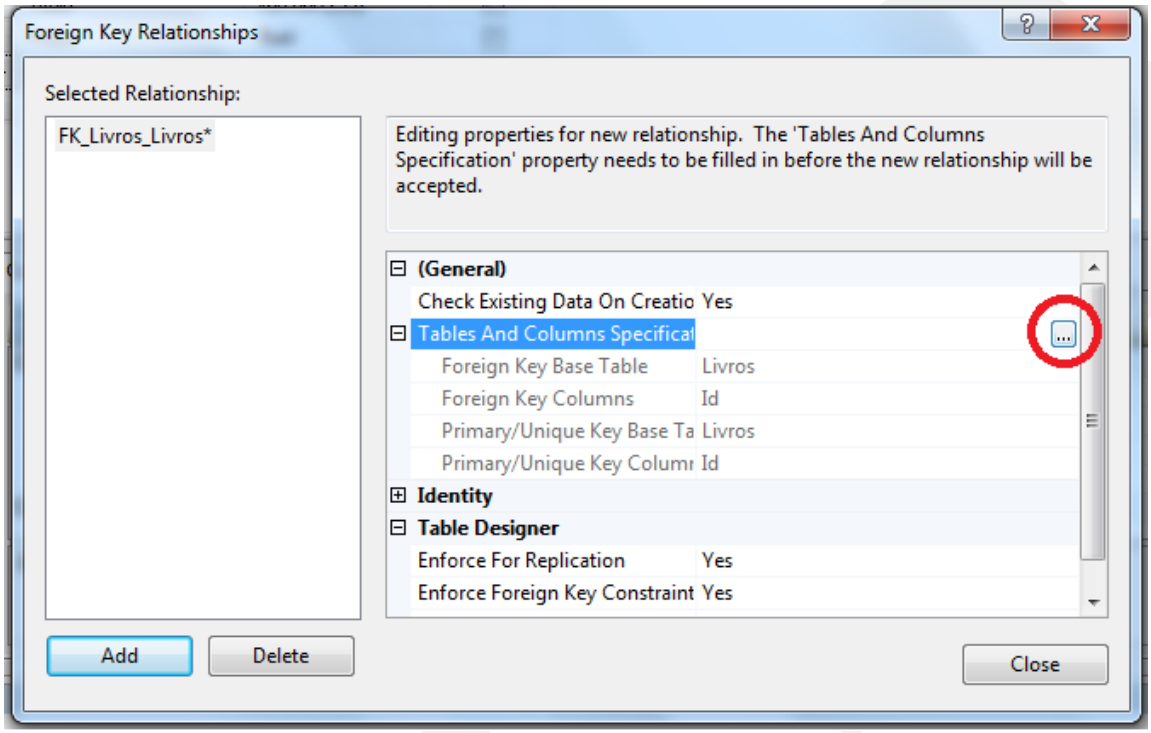


Lembrando de NÃO marcar a opção **ALLOW NULL**. Além disso o campo **Id** deve ser uma chave primária e automaticamente incrementada.

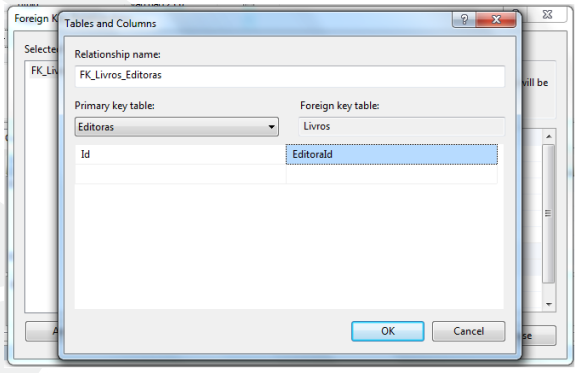
Você precisa tornar o campo **EditoraId** uma chave estrangeira. Clique com o botão direito sobre a coluna **EditoraId** e selecione a opção **Relantioships...**, conforme a figura abaixo:



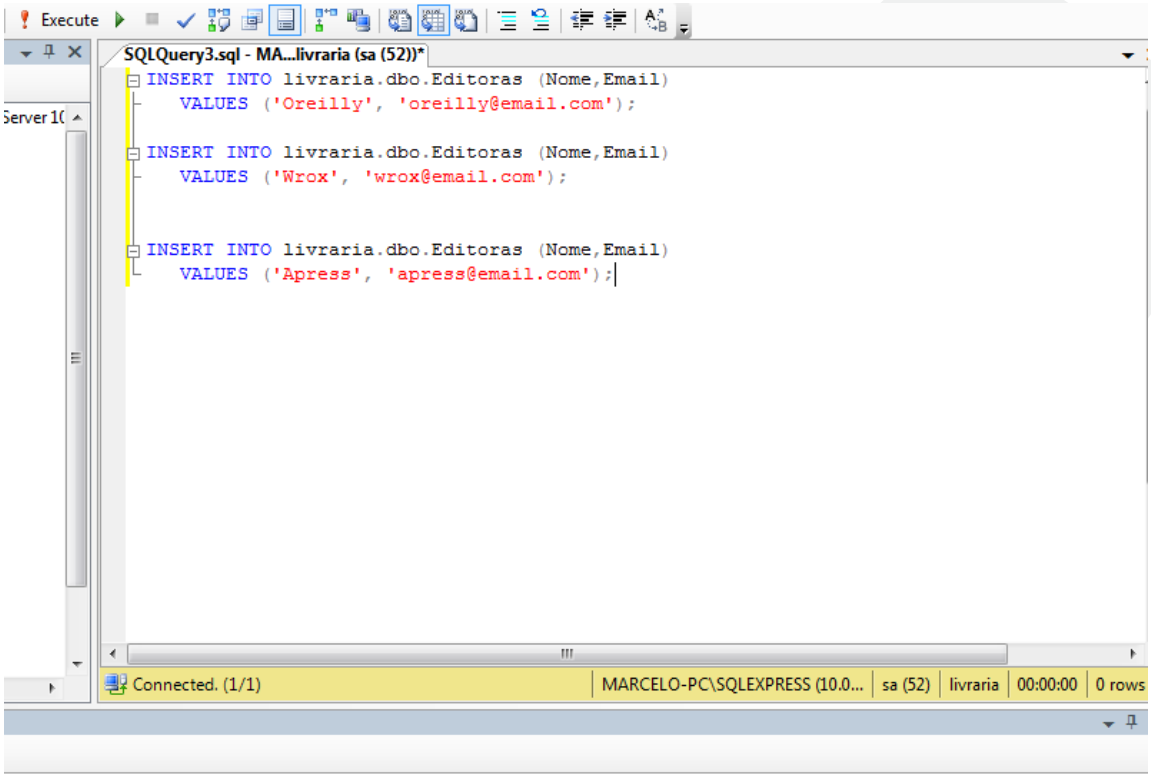
Devemos acrescentar o relacionamento entre livro e editora. Clique em **Add** e posteriormente no botão à direita na linha *Tables and Columns Specification*.



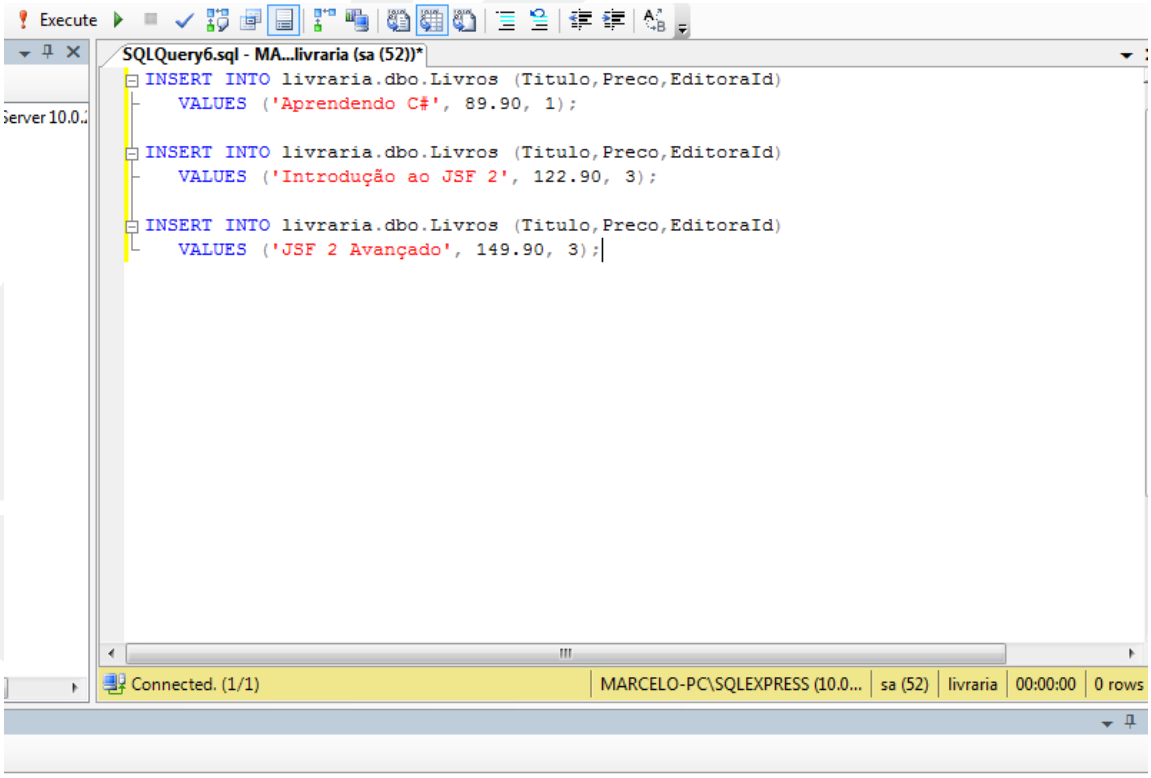
Devemos informar qual é a chave primária que a coluna *EditoraId* da tabela Livros faz referência. Para isto, informe a tabela Editoras como **Primary Key Table** e indique a coluna *Id* como a chave primária referenciada. Selecione a coluna **EditoraId** como a coluna que irá fazer referência a chave primária da tabela Editoras.



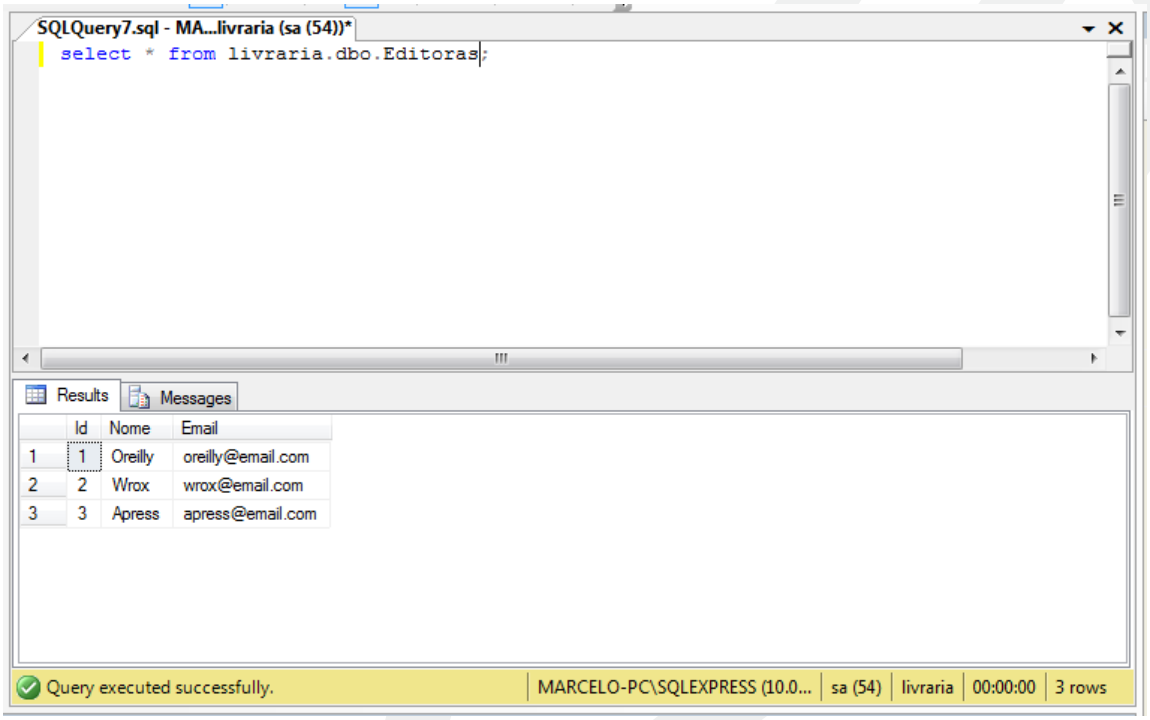
6. Adicione alguns registros na tabela Editoras. Veja exemplos na figura abaixo:

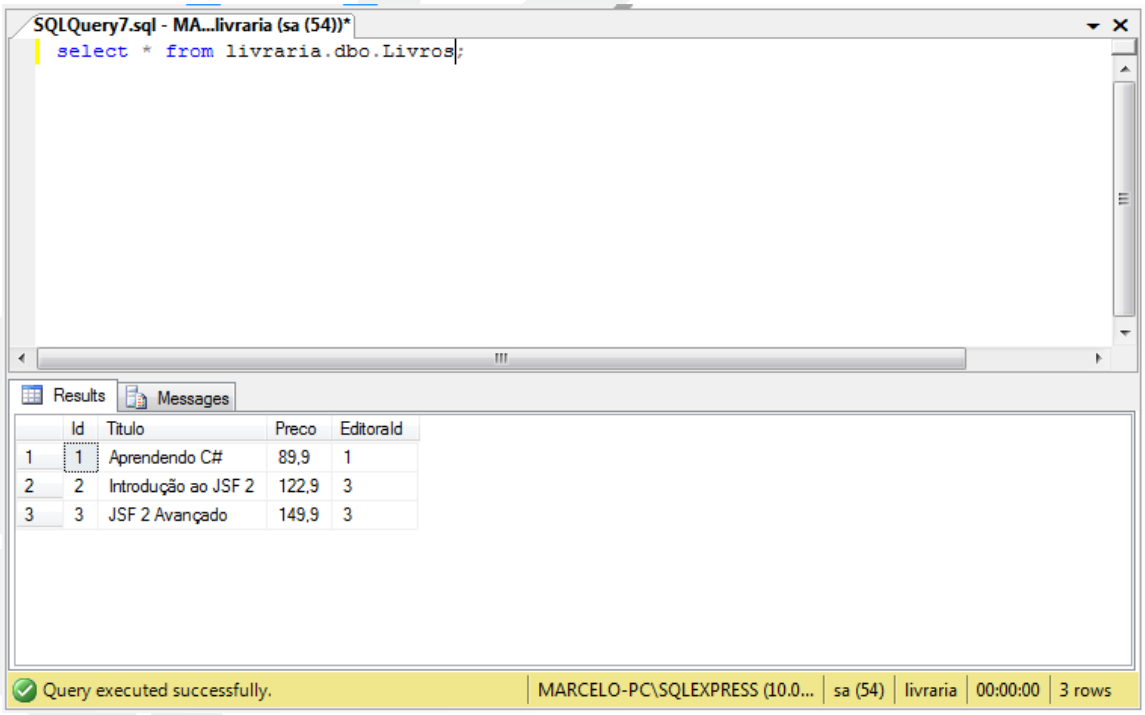


7. Adicione alguns registros na tabela Livros. Veja exemplos na figura abaixo:



8. Consulte os registros da tabela Editoras, e em seguida consulte a tabela Livros. Veja exemplos logo abaixo:



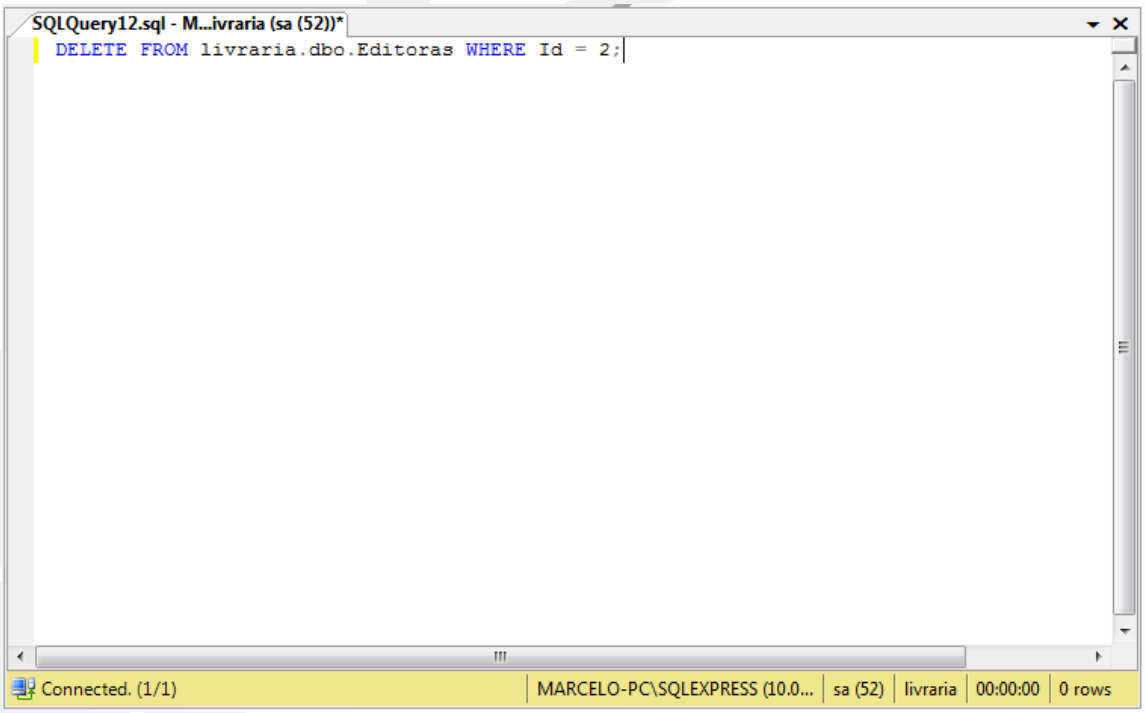


9. Altere alguns dos registros da tabela Livros. Veja o exemplo abaixo:

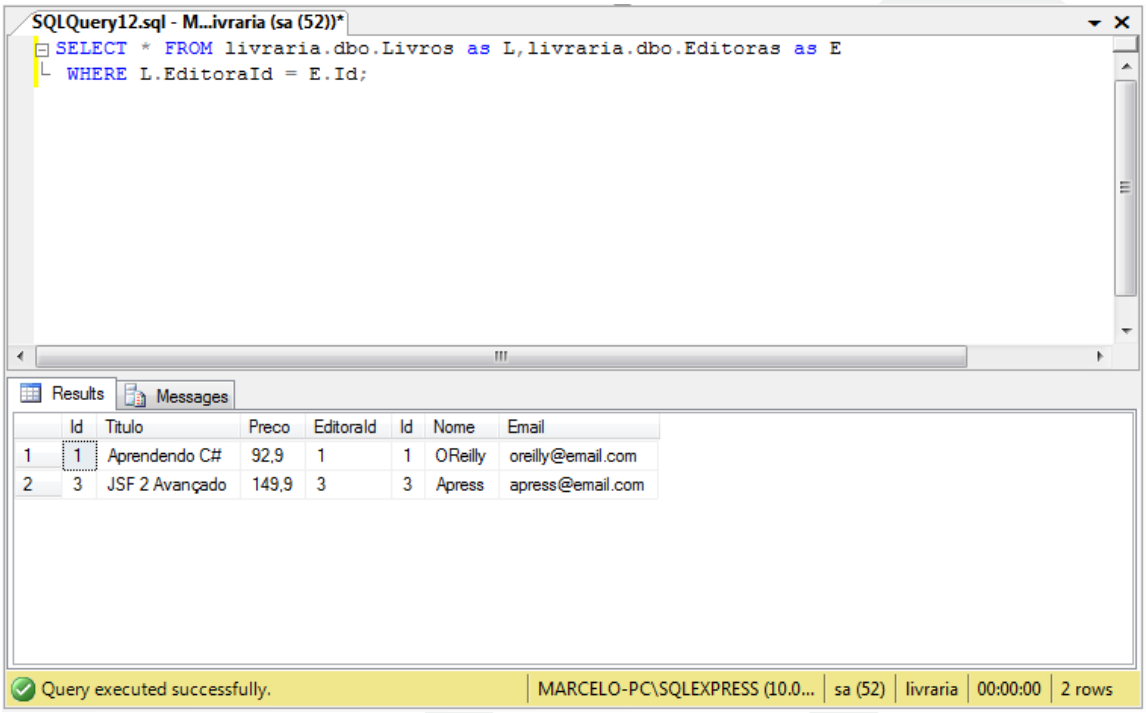
10. Altere alguns dos registros da tabela Editoras. Veja o exemplo abaixo:

11. Remova alguns registros da tabela Livros. Veja o exemplo abaixo:

12. Remova alguns registros da tabela Editoras. Preste atenção para não remover uma editora que tenha algum livro relacionado já adicionado no banco. Veja o exemplo abaixo:



13. Faça uma consulta para buscar todos os livros de uma determinada editora. Veja um exemplo na figura abaixo:


SQLQuery12.sql - M...ivraria (sa (52))*
SELECT * FROM livraria.dbo.Livros as L,livraria.dbo.Editoras as E
WHERE L.EditoraId = E.Id;
Results
Messages
Id Titulo Preco EditoraId Id Nome Email
1 1 Aprendendo C# 92,9 1 1 OReilly oreilly@email.com
2 3 JSF 2 Avançado 149,9 3 3 Apress apress@email.com
Query executed successfully.
MARCELO-PC\SQLEXPRESS (10.0...
sa (52)
livraria
00:00:00
2 rows

Capítulo 2

ADO.NET

No capítulo anterior, aprendemos que utilizar bancos de dados é uma boa solução para o armazenamento dos dados do estoque de uma loja virtual. Entretanto, você deve ter percebido que a interface de utilização do SQL Server Express (e dos outros bancos de dados em geral) não é muito amigável. Ela exige que o usuário conheça a sintaxe do SQL para escrever as consultas. Além disso, quando o volume de dados é muito grande, é mais difícil visualizar os resultados.

Na prática uma aplicação com interface simples é desenvolvida para permitir que os usuários do sistema possam manipular os dados do banco. De alguma forma, essa aplicação precisa se comunicar com o banco de dados utilizado no sistema.

Nesse capítulo vamos desenvolver uma aplicação para acessar os dados do estoque da nossa loja virtual.

2.1 Driver

Como a aplicação precisa conversar com o banco de dados, ela deve trocar mensagens com o mesmo. O formato das mensagens precisa ser definido previamente. Por questões de economia de espaço, cada bit de uma mensagem tem um significado diferente. Em outras palavras, o protocolo de comunicação é binário.

Mensagens definidas com protocolos binários são facilmente interpretadas por computadores. Por outro lado, são complexas para um ser humano compreender. Dessa forma, é mais trabalhoso e mais suscetível a erro desenvolver uma aplicação que converse com um banco de dados através de mensagens binárias.

Para resolver esse problema e facilitar o desenvolvimento de aplicações que devem se comunicar com bancos de dados, as empresas que são proprietárias desses bancos oferecem os **drivers de conexão**.

Os drivers de conexão atuam como "tradutores" de comandos escritos em uma determinada linguagem de programação para comandos no protocolo do banco de dados. Do ponto de vista do desenvolvedor da aplicação, não é necessário conhecer o complexo protocolo binário do banco.

Em alguns casos, o protocolo binário de um determinado banco de dados é fechado. Consequentemente, a única maneira de se comunicar com o banco de dados é através de um driver de conexão.

2.2 ODBC

Suponha que os proprietários dos bancos de dados desenvolvessem os drivers de maneira totalmente independente. Consequentemente, cada driver teria sua própria interface, ou seja, seu próprio conjunto de instruções. Dessa maneira, o desenvolvedor da aplicação precisa conhecer as instruções de cada um dos drivers dos respectivos bancos que ele for utilizar.

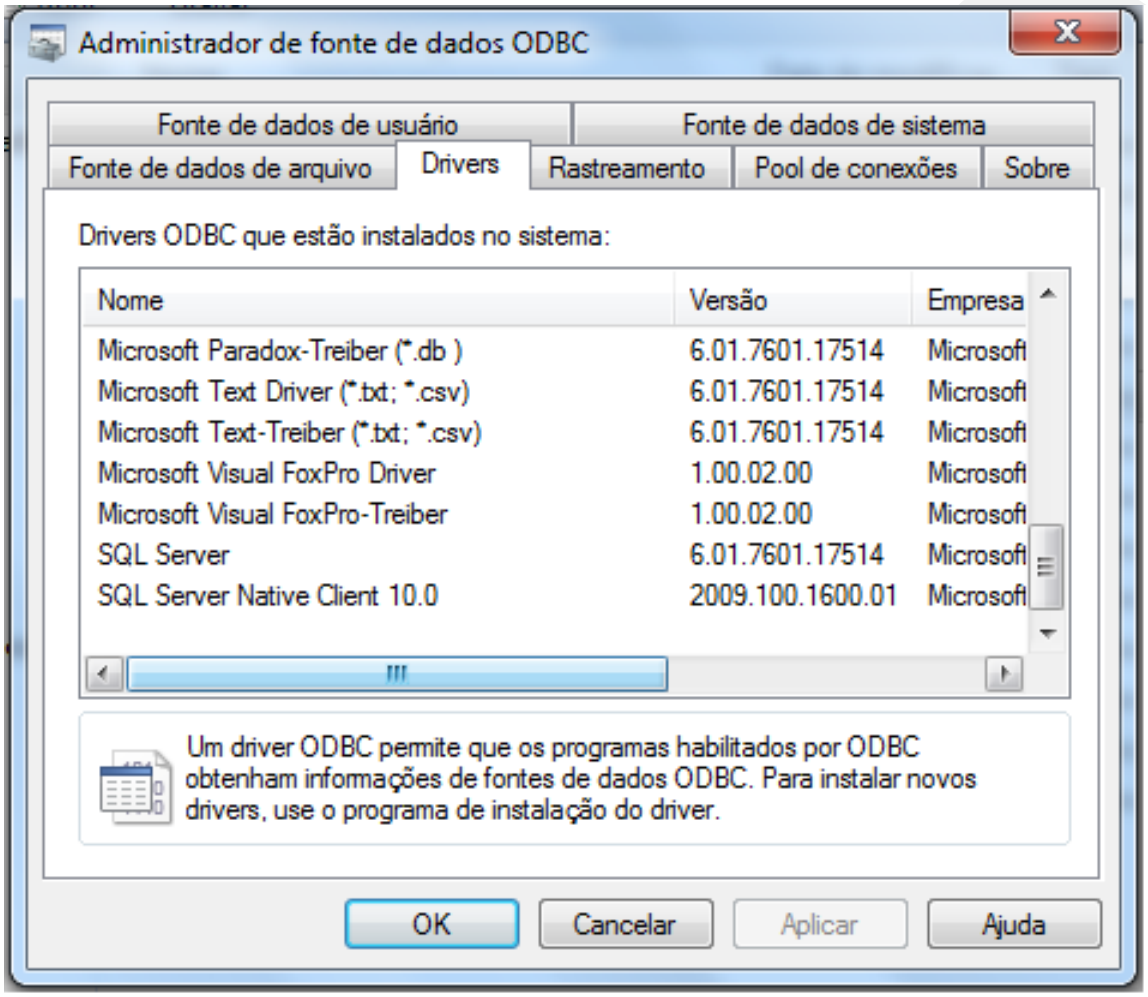
Para facilitar o trabalho do desenvolvedor da aplicação, a Microsoft® definiu uma especificação chamada ODBC (Open Database Connectivity) para padronizar a interface dos drivers de conexão. Assim, quando uma empresa proprietária de um banco de dados pretende desenvolver um driver, ela segue essa especificação com o intuito de popularizá-lo.

Os drivers de conexão que respeitam a especificação ODBC, ou seja, possuem um conjunto de comandos padronizados, são chamados de drivers de conexão ODBC.

2.3 ODBC Manager

Para que drivers ODBC possam ser instalados em uma máquina e as aplicações consigam utilizá-los é necessário ter o ODBC Manager, que já vem instalado no Windows®.

O driver de conexão ODBC já está disponível para utilização, podemos consultar o ODBC Manager do Windows®. O ODBC Manager pode ser executado através do item Ferramentas Administrativas do Painel de Controle.



2.4 Criando uma conexão

Com o driver de conexão ODBC instalado na máquina já é possível criar uma conexão com o banco de dados correspondente. O que é necessário para estabelecer uma conexão com o banco de dados?

- Escolher o driver de conexão;
- Definir a localização do banco de dados;
- Informar o nome da base de dados;
- Ter um usuário e senha cadastrados no banco de dados.

Todas essas informações são definidas na chamada **string de conexão**.

```
1 string stringDeConexao = @"driver={SQL Server};
2     server=MARCELO-PC\SQLEXPRESS;database=livraria;uid=sa;pwd=sa;";
```

Após a definição da string de conexão, podemos utilizar a classe do *.NET Framework* que é responsável por criar conexões ODBC. Esta classe que vamos utilizar é a *System.Data.Odbc.OdbcConnection*.

```
1 OdbcConnection conexao = new OdbcConnection(stringDeConexao);
```

2.5 Inserindo registros

Estabelecida a conexão com o banco de dados, já podemos executar comandos. Por exemplo, já podemos inserir registros nas tabelas. O primeiro passo para executar um comando é definí-lo em linguagem SQL.

```
1 string textoDoComando = @"INSERT INTO Editoras (Nome, Email)
2     VALUES  ('Abril', 'abril@email.com');";
```

Em seguida, devemos criar um objeto da classe *System. Data.Odbc.OdbcCommand* vinculado ao texto do comando e à conexão previamente criada. O comando não é executado quando os objetos dessa classe são instanciados.

```
1 OdbcCommand comando = new OdbcCommand(textoDoComando, conexao);
```

Por fim, o comando pode ser executado através do método *ExecuteNonQuery*. A conexão deve ser aberta antes de executar o comando.

```
1 conexao.Open();
2 comando.ExecuteNonQuery();
```

Se a aplicação mantiver as conexões abertas o banco de dados pode deixar de atender outras aplicações pois há um limite de conexões que o banco pode suportar. Portanto, é importante que as conexões sejam fechadas quando não forem mais necessárias.

```
1 conexao.Close();
```

2.6 Exercícios

1. Crie um projeto do tipo *Console Application* no *Microsoft Visual C# Express*, chamado **ODBC**.

2. Utilizando o projeto do exercício anterior, faça um programa para inserir registros na tabela *Editoras*.

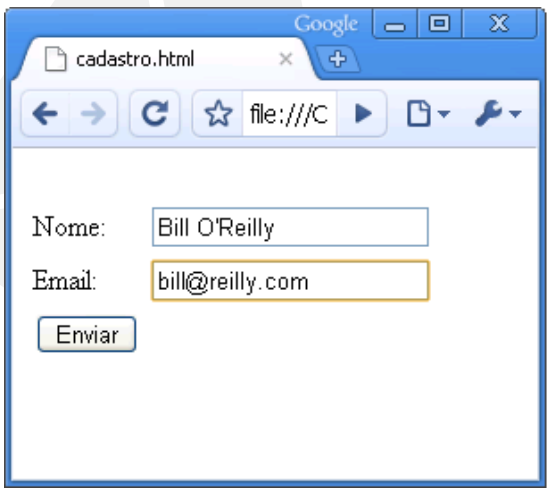
```
1  using System.Data.Odbc;
2
3  namespace Odbc
4  {
5      class InsereEditora
6      {
7          static void Main(string[] args)
8          {
9              string stringDeConexao = @"driver={SQL Server};
10     server=MARCELO-PC\SQLEXPRESS;database=livraria;uid=sa;pwd=sa;";
11
12
13
14             using (OdbcConnection conexao = new OdbcConnection(stringDeConexao))
15             {
16                 string textoInsereEditora =
17                 "INSERT INTO Editoras (Nome, Email) Values('Abril', 'abril@email.com')"↩
                       ;
18                 OdbcCommand command = new OdbcCommand(textoInsereEditora, conexao);
19                 conexao.Open();
20                 command.ExecuteNonQuery();
21
22                 // A Conexao eh automaticamente fechada
23                 // ao final do bloco Using.
24             }
25         }
26     }
27 }
```

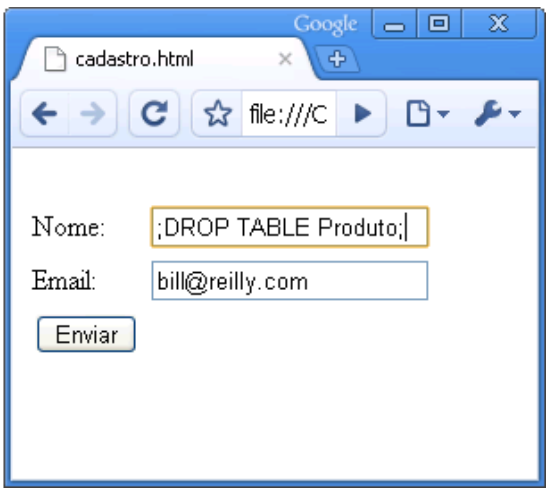
3. (Opcional) Analogamente, ao exercício anterior crie um programa para inserir livros.

2.7 SQL Injection

Da maneira que implementamos a inserção de categorias, há duas falhas: uma de segurança e outra técnica. A falha de segurança ocorre quando uma pessoa mal intencionada ao preencher um formulário, digita propositalmente uma sentença em SQL que provoca um comportamento não previsto.

Esse comportamento, pode por vezes, comprometer a segurança, às vezes mostrando à pessoa mal intencionada informações confidenciais, em outras pode apagar informações do banco. Esse tipo de ataque é conhecido como **SQL Injection**.





O problema de SQL Injection pode ser resolvido manualmente. Basta fazer "escape" dos caracteres especiais, por exemplo: ponto-e-vírgula e apóstrofo. No MySQL Server, os caracteres especiais devem ser precedidos pelo caracter "\". Então seria necessário acrescentar \ em todas as ocorrências de caracteres especiais nos valores passados pelo usuário.

Esse processo, além de trabalhoso, é diferente para cada banco de dados, pois o "\" não é padronizado e cada banco tem o seu próprio método de escape de caracteres especiais.

Tornando mais prática a comunicação com o banco de dados o próprio driver faz o tratamento das sentenças SQL. Esse processo é denominado **sanitize**.

```
1  // pegando os dados da editora pelo teclado
2  string nome = Console.ReadLine();
3  string email = Console.ReadLine();
4
5  // definindo a sentença SQL com parâmetros
6  string textoDoComando =
7      @"INSERT INTO Editoras (Nome, Email) VALUES (?, ?);";
8
9  // criando um comando odbc
10 OdbcCommand comando = new OdbcCommand(textoDoComando, conexao);
11
12 // atribuindo valores aos parâmetros
13 comando.Parameters.AddWithValue("@Nome", nome);
14 comando.Parameters.AddWithValue("@Email", email);
```

Observe que a sentença SQL foi definida com parâmetros através do caracter "**?**". Antes de executar o comando, é necessário atribuir valores aos parâmetros. Isso é feito com o método *AddWithValue*. Esse método realiza a tarefa de "sanitizar" os valores enviados pelo usuário.

2.8 Exercícios

4. Tente causar um erro de SQL Injection na classe feita no exercício de inserir editoras. (Dica: tente entradas com aspas simples)

5. Altere o código para eliminar o problema do SQL Injection. Você deve deixar a classe com o código abaixo:

```
using System;
using System.Data.Odbc;


namespace Odbc
{
    class InsereEditora
    {
        static void Main(string[] args)
        {
            string stringDeConexao = @"driver={SQL Server};
    server=MARCELO-PC\SQLEXPRESS;database=livraria;uid=sa;pwd=sa;";


            Console.WriteLine("Abrindo conexao...");
            using (OdbcConnection conexao = new OdbcConnection(stringDeConexao))
            {
                string textoInsereEditora =
                "INSERT INTO Editoras (Nome, Email) Values(?, ?)";
                Console.WriteLine("Digite o nome da Editora:");
                string nome = Console.ReadLine();
                Console.WriteLine("Digite o email da Editora:");
                string email = Console.ReadLine();
                OdbcCommand command = new OdbcCommand(textoInsereEditora, conexao);
                command.Parameters.AddWithValue("@Nome", nome);
                command.Parameters.AddWithValue("@Email", email);
                conexao.Open();
                command.ExecuteNonQuery();

                // A Conexao eh automaticamente fechada
                // ao final do bloco Using.
            }
        }
    }
}
```

6. Agora tente causar novamente o problema de SQL Injection ao inserir novas editoras.

2.9 Listando registros

Depois de inserir alguns registros, é interessante consultar os dados das tabelas para conferir se a inserção foi realizada com sucesso.

O processo para executar um comando de consulta é parecido com o de inserção. É necessário definir a sentença SQL e criar um objeto da classe *OdbcCommand*.

```
// definindo a sentença SQL
string textoDoComando = @"SELECT * FROM Editoras;";

// criando um comando odbc
OdbcCommand comando = new OdbcCommand(textoDoComando, conexao);
```

A diferença é que para executar um comando de consulta é necessário utilizar o método *ExecuteReader* ao invés do *ExecuteNonQuery*. Esse método devolve um objeto da classe *System.Data.Odbc.OdbcDataReader*

```
OdbcDataReader resultado = comando.ExecuteReader();
```

Os dados contidos no *OdbcDataReader* podem ser acessados através de indexers.

```
1 string nome = resultado["Nome"] as string;
2 string email = resultado["Email"] as string;
```

O código acima mostra como os campos do primeiro registro da consulta são recuperados. Agora, para recuperar os outros registros é necessário avançar o *OdbcDataReader* através do método *Read*.

```
1 string nome1 = resultado["nome"] as string;
2 string email1 = resultado["email"] as string;
3
4 resultado.Read();
5
6 string nome2 = resultado["nome"] as string;
7 string email2 = resultado["email"] as string;
```

O próprio método *Read* devolve um valor booleano para indicar se o reader conseguiu avançar para o próximo registro. Quando esse método devolver *false* significa que não há mais registros para serem consultados.

```
1 while(resultado.Read())
2 {
3     string nome = resultado["nome"] as string;
4     string email = resultado["email"] as string;
5 }
```

2.10 Exercícios

7. Insira algumas editoras utilizando a classe INSEREEDITORA que você criou nos exercícios acima.

8. Adicione uma nova classe ao projeto chamada **ListaEditora**. O objetivo é listar as editoras que foram salvas no banco. Adicione o seguinte código à esta classe.

```
using System;
using System.Data.Odbc;

namespace Odbc
{
    class ListaEditora
    {
        static void Main(string[] args)
        {
            string stringDeConexao = @"driver={SQL Server};
    server=MARCELO-PC\SQLEXPRESS;database=livraria;uid=sa;pwd=sa;";


            Console.WriteLine("Abrindo conexao...");
            using (OdbcConnection conexao = new OdbcConnection(stringDeConexao))
            {
                string textoListaEditora =
                "SELECT * FROM Editoras";
                OdbcCommand command = new OdbcCommand(textoListaEditora, conexao);
                conexao.Open();
                OdbcDataReader resultado = command.ExecuteReader();

                while (resultado.Read())
                {
                    long? id = resultado["Id"] as long?;
                    string nome = resultado["Nome"] as string;
                    string email = resultado["Email"] as string;
                    Console.WriteLine("{0} : {1} - {2}\n",id,nome,email);
                }
                // A Conexao eh automaticamente fechada
                // ao final do bloco Using.
            }
        }
    }
}
```

2.11 Fábrica de conexões (Factory)

Você deve ter percebido que para cada ação executada no banco de dados, nós precisamos criar uma conexão. Isso gera um problema relacionado à string de conexão ficar armazenada em diversos locais. Imagine que o driver do banco foi atualizado e mudamos a sua versão. Isso implicaria fazer diversas alterações no código em cada ocorrência da string de conexão, tornando o código mais suscetível a erros e dificultando a sua manutenção.

Para resolver esta situação, nós poderíamos criar uma classe responsável pela criação e distribuição de conexões, mantendo assim uma única referência para a string de conexão, e qualquer alteração no modo em que nos conectamos à base de dados, só implica mudanças nesta classe.

```
1  static class FabricaDeConexao
2  {
3      public static OdbcConnection CriaConexao()
4      {
5          string driver = @"SQL Server";
6          string servidor = @"MARCELO-PC\SQLEXPRESS";
7          string baseDeDados = @"livraria";
8          string usuario = @"sa";
9          string senha = @"sa";
10
11         StringBuilder connectionString = new StringBuilder();
12         connectionString.Append("driver=");
13         connectionString.Append(driver);
14         connectionString.Append(";server=");
15         connectionString.Append(servidor);
16         connectionString.Append(";database=");
17         connectionString.Append(baseDeDados);
18         connectionString.Append(";uid=");
19         connectionString.Append(usuario);
20         connectionString.Append(";pwd=");
21         connectionString.Append(senha);
22
23         return new OdbcConnection(connectionString.ToString());
24     }
25 }
```

Agora podemos obter uma nova conexão apenas chamando **FabricaDeConexao.CriaConexao()**. O resto do sistema não precisa mais conhecer os detalhes sobre a conexão com o banco de dados, diminuindo o acoplamento da aplicação.

2.12 Exercícios

9. Adicione uma nova classe chamada FABRICADECONEXAO e adicione o seguinte código:

```
using System;
using System.Data.Odbc;
using System.Text;

namespace Odbc
{
    static class FabricaDeConexao
    {
        public static OdbcConnection CriaConexao()
        {
            string driver = @"SQL Server";
            string servidor = @"MARCELO-PC\SQLEXPRESS";
            string baseDeDados = @"livraria";
            string usuario = @"sa";
            string senha = @"sa";

            StringBuilder connectionString = new StringBuilder();
            connectionString.Append("driver=");
            connectionString.Append(driver);
            connectionString.Append(";server=");
            connectionString.Append(servidor);
            connectionString.Append(";database=");
            connectionString.Append(baseDeDados);
            connectionString.Append(";uid=");
            connectionString.Append(usuario);
            connectionString.Append(";pwd=");
            connectionString.Append(senha);

            return new OdbcConnection(connectionString.ToString());
        }
    }
}
```

10. Altere as classes INSEREEDITORA e LISTAEDITORA para que elas utilizem a fábrica de conexão. Execute-as novamente.

11. (Opcional) Implemente um teste que remove uma editora pelo **id**.

12. (Opcional) Implemente um teste que altera os dados de uma editora pelo **id**.

Capítulo 3

Entity Framework 4.1

3.1 Múltiplas sintaxes da linguagem SQL

No capítulo anterior, você aprendeu a utilizar a especificação ODBC para fazer uma aplicação C# interagir com um banco de dados. Essa interação é realizada através de consultas escritas em SQL. Uma desvantagem dessa abordagem, é que a sintaxe da linguagem SQL, apesar de parecida, pode variar conforme o banco de dados que está sendo utilizado. Desse modo, os desenvolvedores teriam que aprender as diferenças entres as sintaxes do SQL correspondentes aos banco de dados que ele utilizará.

Seria bom se, ao invés de programar direcionado a um determinado banco de dados, pudéssemos programar de uma maneira mais genérica, voltado à uma interface ou especificação, assim poderíamos escrever o código independente de SQL.

3.2 Orientação a Objetos VS Modelo Entidade Relacionamento

Outro problema na comunicação entre uma aplicação C# e um banco de dados é o conflito de paradigmas. O banco de dados é organizado seguindo o modelo entidade relacionamento, enquanto as aplicações C#, geralmente, utilizam o paradigma orientado a objetos.

A transição de dados entre o modelo entidade relacionamento e o modelo orientado a objetos não é simples. Para realizar essa transição, é necessário definir um mapeamento entre os conceitos desses dois paradigmas. Por exemplo, classes podem ser mapeadas para tabelas, objetos para registros, atributos para campos e referência entre objetos para chaves estrangeiras.

3.3 Ferramentas ORM

Para facilitar a comunicação entre aplicações C# que seguem o modelo orientado a objetos e os banco de dados que seguem o modelo entidade relacionamento, podemos utilizar ferramentas que automatizam a transição de dados entre as aplicações e os diferente bancos de dados e que são conhecidas como ferramentas de **ORM** (Object Relational Mapper).

Outra consequência, ao utilizar uma ferramenta de ORM, é que não é necessário escrever consultas em SQL, pois a própria ferramenta gera as consultas de acordo com a sintaxe da

linguagem SQL correspondente ao banco que está sendo utilizado.

A principal ferramenta ORM para C# é o Entity Framework. Mas, existem outras que possuem o mesmo objetivo.

3.4 Configuração

Antes de começar a utilizar o Entity Framework, é necessário baixar no site a versão 4.1: (http://www.microsoft.com/downloads/details.aspx?FamilyID=b41c728e-9b4f

3.5 Mapeamento

Um dos principais objetivos dos frameworks ORM é estabelecer o mapeamento entre os conceitos do modelo orientado a objetos e os conceitos do modelo entidade relacionamento. Este mapeamento pode ser definido através de xml ou de maneira mais prática com **DbContext**. Quando utilizamos **DbContext**, evitamos a criação de extensos arquivos em xml.

Podemos definir as seguintes entidades:

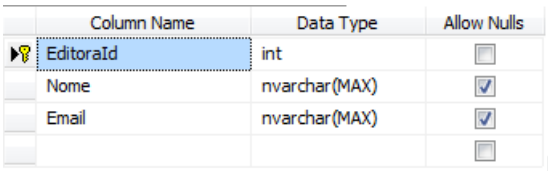
```
1
2  public class Livro
3  {
4      public int LivroId { get; set; }
5      public string Titulo { get; set; }
6      public decimal Preco { get; set; }
7      public Editora Editora { get; set; }
8  }
9
10 public class Editora
11 {
12     public int EditoraId { get; set; }
13     public string Nome { get; set; }
14     public string Email { get; set; }
15     public ICollection<Livro> Livros { get; set; }
16 }
```

Criaremos agora uma classe para ajudar a mapear as entidades para um banco de dados. A classe **EditoraContext** deriva de **DbContext** que faz parte da biblioteca **Code First**:

```
1 public class EditoraContext : DbContext
2 {
3     public DbSet<Editora> Editoras { get; set; }
4     public DbSet<Livro> Livros { get; set; }
5 }
```

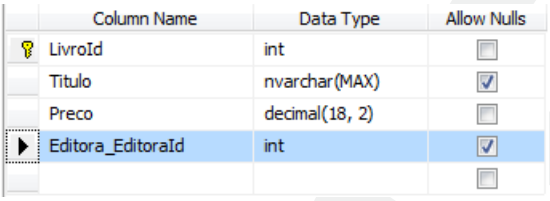
Utilizamos o padrão EF4 **Code First** para permitir a persistência no banco de dados. Isto significa que as propriedades Editoras e Livros serão mapeadas para tabelas com mesmo nome banco de dados. Cada propriedade definida na entidade **Livro** e **Editora** é mapeada para colunas das tabelas **Livros** e **Editoras**.

Abaixo segue a definição da tabela **Editoras** que foi criada em nosso banco de dados:



Column Name	Data Type	Allow Nulls
EditoraId	int	☐
Nome	nvarchar(MAX)	☑
Email	nvarchar(MAX)	☑
		☐

Abaixo segue a definição da tabela **Livros** que foi criada em nosso banco de dados:



Column Name	Data Type	Allow Nulls
LivroId	int	☐
Titulo	nvarchar(MAX)	☑
Preco	decimal(18, 2)	☐
Editora_EditoraId	int	☑
		☐

Nós não precisamos configurar nada para que a persistência e o mapeamento fossem feitos com o EF **Code First** - isto ocorreu simplesmente escrevendo as três classes acima. Não é necessário nenhuma configuração a mais.

Podemos utilizar anotações para sobrescrever a convenção padrão. Para utilizar anotação precisamos adicionar como referência *EntityFramework.dll* e *System.ComponentModel.DataAnnotations.dll* ao projeto e acrescentar **using** para o namespace *System.ComponentModel.DataAnnotations*. Segue as principais anotações:

ColumnAttribute Define o nome e o tipo da coluna no banco de dados da propriedade mapeada.

```
1  public class Livro
2  {
3      public int LivroId { get; set; }
4      [Column("NomeDoLivro",TypeName="varchar")]
5      public string Titulo { get; set; }
6      public decimal Preco { get; set; }
7      public Editora Editora { get; set; }
8  }
```

DatabaseGeneratedAttribute Utilizado para indicar que o valor do atributo é gerado pelo banco de dados. Para definir como o atributo é gerado você utiliza três constantes do enum DatabaseGenerateOption: **DatabaseGeneratedOption.Identity** que define que o valor será definido na inserção e assume que não será mais alterado. **DatabaseGeneratedOption.Computed** que é lido na inserção e a cada atualização. **DatabaseGeneratedOption.None** indica que o valor não será gerado pelo banco de dados.

ForeignKeyAttribute é adicionado a propriedade para especificar a propriedade que define a chave estrangeira do relacionamento.

```
1
2  public class Livro
3  {
4
5      public int LivroId { get; set; }
6      public string Titulo { get; set; }
7      public decimal Preco { get; set; }
8      [ForeignKey("Editora")]
9      public int EditoraId { get; set; }
10     public Editora Editora { get; set; }
11 }
12
13 public class Editora
14 {
15     public int EditoraId { get; set; }
16     public string Nome { get; set; }
17     public string Email { get; set; }
18     public ICollection<Livro> Livros { get; set; }
19 }
```

InversePropertyAttribute Indica a propriedade que define o relacionamento. Esta anotação é utilizada quando temos múltiplos relacionamentos do mesmo tipo.

Por exemplo, suponha que tenhamos uma entidade **Pessoa** pode ser autor ou revisor de um livro. Uma pessoa pode ter livros publicados e livros revisados, portanto a entidade Pessoa tem dois relacionamentos diferentes com a entidade **Livro**.

```
1
2  public class Pessoa
3  {
4      public int Id { get; set; }
5      public string Nome { get; set; }
6      public ICollection<Livro> LivrosPublicados { get; set; }
7      public ICollection<Livro> LivrosRevisados { get; set; }
8  }
9
10 public class Livro
11 {
12
13     public int LivroId { get; set; }
14     public string Titulo { get; set; }
15     public decimal Preco { get; set; }
16     public Editora Editora { get; set; }
17
18     [InverseProperty("LivrosPublicados")]
19     public Pessoa Autor { get; set; }
20
21     [InverseProperty("LivrosRevisados")]
22     public Pessoa Revisor { get; set; }
23 }
```

KeyAttribute Define a propriedades ou propriedades que identificam uma entidade.

Se a classe define propriedades com **ID** ou **Id**, ou nome da classe seguido por **ID** ou **Id**, esta propriedade é tratada como chave primária por convenção.

Caso contrário, podemos definir a nossa chave primária com o *KeyAttribute*.

```
1
2  public class Pessoa
3  {
4      [Key]
5      public int Identificador { get; set; }
6      public string Nome { get; set; }
7      public ICollection<Livro> LivrosPublicados { get; set; }
8      public ICollection<Livro> LivrosRevisados { get; set; }
9  }
```

MaxLengthAttribute Define o tamanho máximo permitido para um array ou string.

MinLengthAttribute Define o tamanho mínimo permitido para um array ou string.

```
1  public class Pessoa
2  {
3
4      public int Id { get; set; }
5      [MinLength(10,ErrorMessage="Tamanho minimo: 10")]
6      [MaxLength(255,ErrorMessage="Tamanho máximo: 255")]
7      public string Nome { get; set; }
8      public ICollection<Livro> LivrosPublicados { get; set; }
9      public ICollection<Livro> LivrosRevisados { get; set; }
10 }
```

StringLengthAttribute Define o tamanho mínimo e máximo permitido para o campo string.

```
1  public class Editora
2  {
3      public int EditoraId { get; set; }
4      [StringLength(255,MinimumLength=10)]
5      public string Nome { get; set; }
6      public string Email { get; set; }
7      public string ExtraInfo { get; set; }
8      public ICollection<Livro> Livros { get; set; }
9  }
```

NotMappedAttribute Pode ser aplicado em classes ou propriedades. Quando aplicado em classes ou propriedades indica que este não será incluído no momento de definição do banco de dados.

```
1
2  public class Editora
3  {
4      public int EditoraId { get; set; }
5      public string Nome { get; set; }
6      public string Email { get; set; }
7      [NotMapped]
8      public string ExtraInfo { get; set; }
9      public ICollection<Livro> Livros { get; set; }
10 }
```

RequiredAttribute Define que este campo é obrigatório. Este atributo é ignorado em propriedades do tipo collection. Quando definido numa referência, indica que cardinalidade é **1** e a propriedade da chave estrangeira é não-nula.

```
1
2  public class Editora
3  {
4      public int EditoraId { get; set; }
5      [Required]
6      public string Nome { get; set; }
7      public string Email { get; set; }
8      public string ExtraInfo { get; set; }
9      public ICollection<Livro> Livros { get; set; }
10 }
```

TableAttribute Define a tabela para o qual a classe é mapeada.

```
1
2  [Table("Livros")]
3  public class Livro
4  {
5      public int LivroId { get; set; }
6      public string Titulo { get; set; }
7      public decimal Preco { get; set; }
8      public Editora Editora { get; set; }
9      public Pessoa Autor { get; set; }
10     public Pessoa Revisor { get; set; }
11 }
```

3.6 Gerando o banco

Uma das vantagens de utilizar o EF4 *Code First*, é que ele é capaz de gerar as tabelas do banco para a nossa aplicação. Ele faz isso de acordo com as anotações colocadas nas classes e as informações presentes no DbContext.

As tabelas são geradas quando criamos uma instância de *DbContext()* e o contexto necessita fazer alguma requisição ao banco de dados. Por exemplo, quando adicionamos uma entidade ao contexto.

```
1
2  EditoraContext context = new EditoraContext();
3  // Neste momento será criado o banco de dados, caso não exista
4  context.Editoras.Add(new Editora{ Nome = "Abril" });
```

A política de criação das tabelas pode ser alterada através do método de classe (estático) *System.Data.Entity.Database.SetInitializer<TypeContext>*. Neste método devemos passar instâncias do tipo *System.Data.Entity.IDatabaseInitializer*. No EF 4.1 temos 3 classes que definem estratégias para criação do banco de dados: **DropCreateDatabaseAlways** criará o banco de dados a cada instância criada do contexto, ideal para ambiente de testes. **CreateDatabaseIfNotExists** somente criará o banco de dados, caso ele não exista, este é a estratégia padrão.

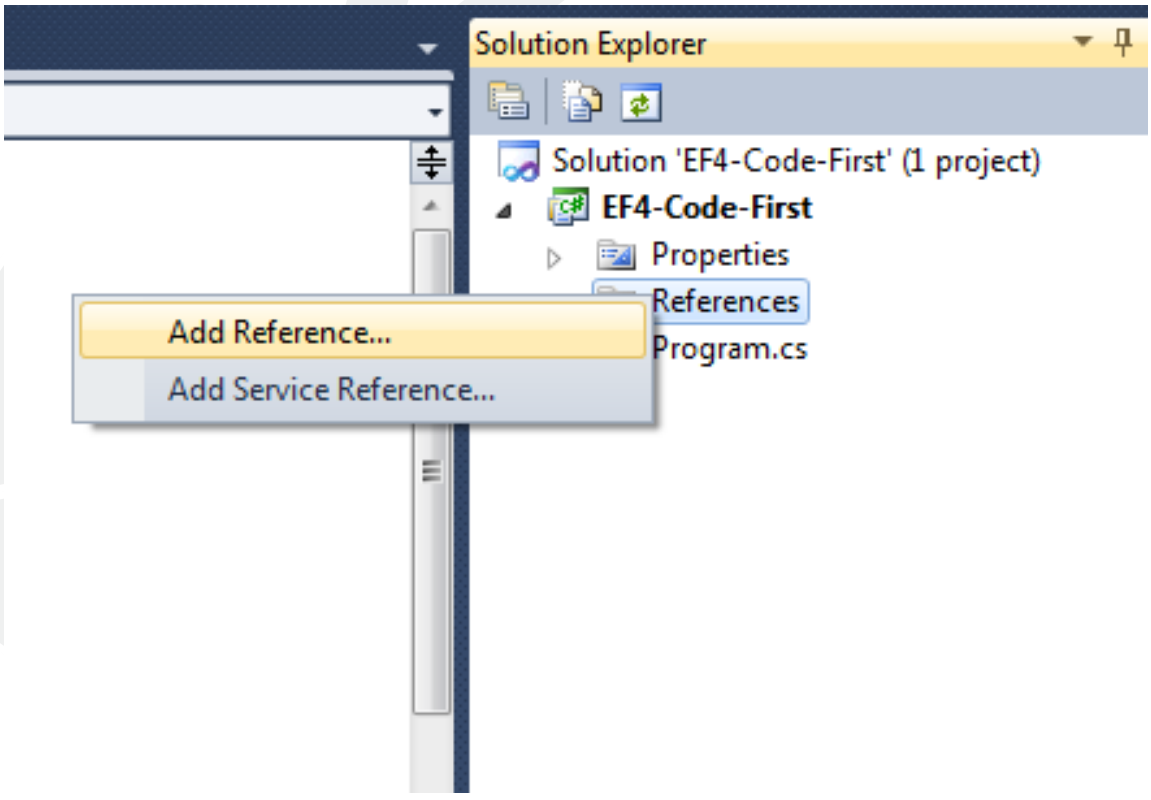
DropCreateDatabaseIfModelChanges somente criará o banco de dados caso ele não exista ou caso haja alguma alteração nas suas entidades.

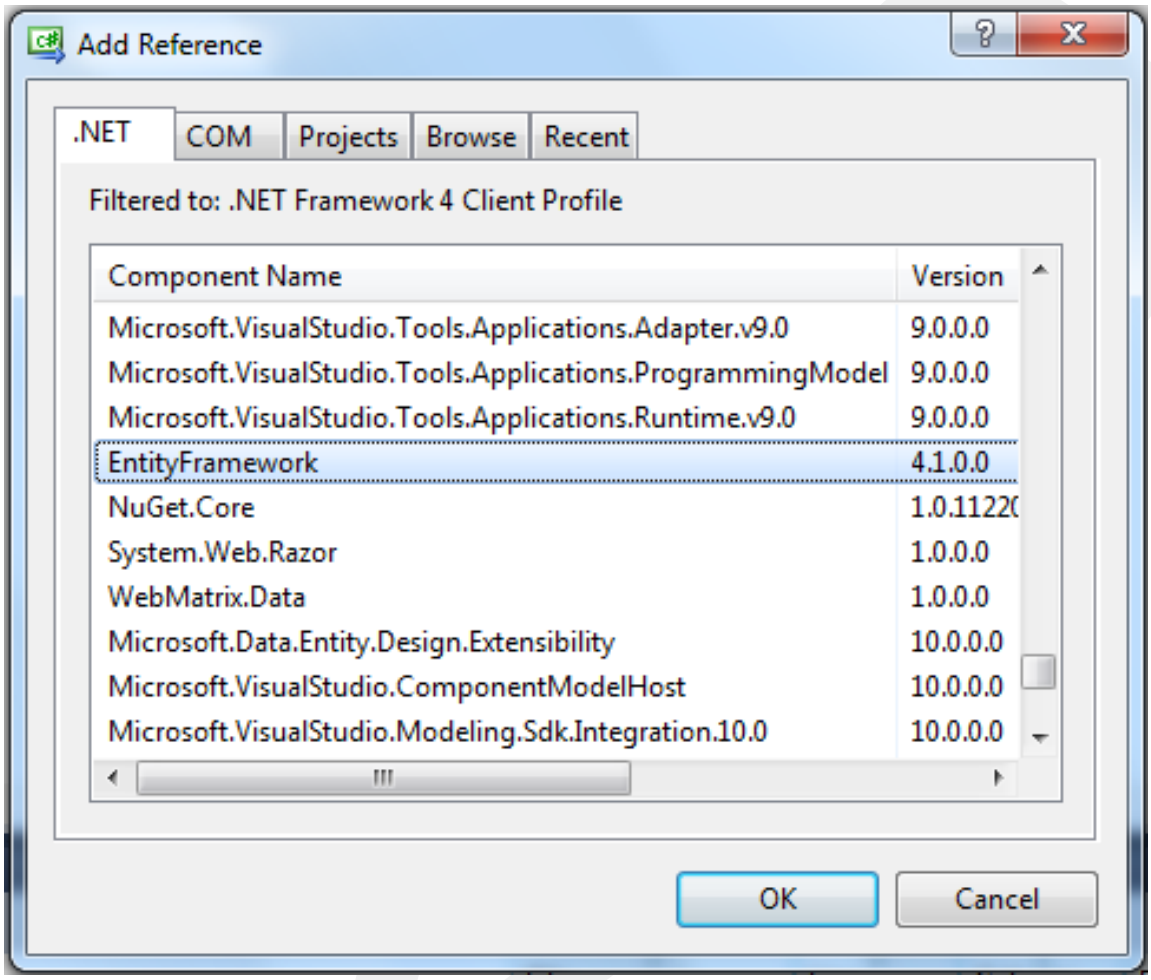
```
1 //Alterando a estratégia padrão
2 Database.SetInitializer<EditoraContext>(new DropCreateDatabaseIfModelChanges<↩
      EditoraContext>());
3
4 EditoraContext context = new EditoraContext();
5 // Neste momento será criado o banco de dados, caso não exista
6 context.Editoras.Add(new Editora{ Nome = "Abril" });
```

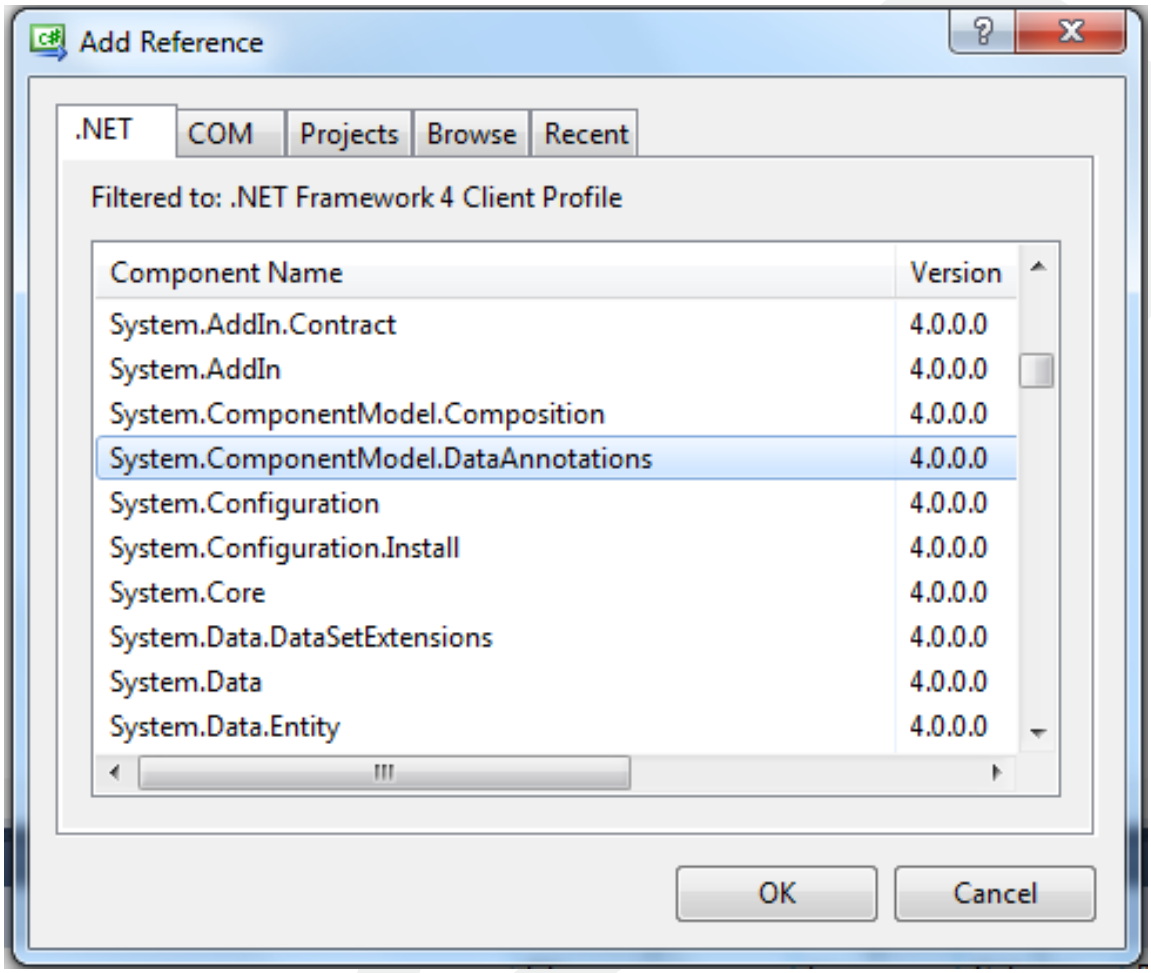
3.7 Exercícios

1. Crie um projeto do tipo *Console Application* no *Microsoft Visual C# Express*, chamado **EF4-Code-First**.

2. Adicione as seguintes dlls ao projeto: EntityFramework.dll e System.ComponentModel.DataAnnotations




Add Reference
.NET
COM
Projects
Browse
Recent
Filtered to: .NET Framework 4 Client Profile
Component Name
Version
Microsoft.VisualStudio.Tools.Applications.Adapter.v9.0
9.0.0.0
Microsoft.VisualStudio.Tools.Applications.ProgrammingModel
9.0.0.0
Microsoft.VisualStudio.Tools.Applications.Runtime.v9.0
9.0.0.0
EntityFramework
4.1.0.0
NuGet.Core
1.0.1122(
System.Web.Razor
1.0.0.0
WebMatrix.Data
1.0.0.0
Microsoft.Data.Entity.Design.Extensibility
10.0.0.0
Microsoft.VisualStudio.ComponentModelHost
10.0.0.0
Microsoft.VisualStudio.Modeling.Sdk.Integration.10.0
10.0.0.0
OK
Cancel



3. Defina uma classe *Editora* conforme código abaixo:

```
1
2  namespace EF4_Code_First
3  {
4      public class Editora
5      {
6          public int Id { get; set; }
7          public string Nome { get; set; }
8          public string Email { get; set; }
9
10     }
11 }
```

4. Defina uma classe *EditoraContext* que é derivada de *DbContext*. Nesta classe defina uma propriedade Editoras do tipo genérico DbSet<Editora>.

```
1
2  using System.Data.Entity;
3
4  namespace EF4_Code_First
5  {
6      public class EditoraContext : DbContext
7      {
8          public DbSet<Editora> Editoras { get; set; }
9      }
10 }
```

5. Defina uma classe GeraTabelas conforme código abaixo:

```
1
2  namespace EF4_Code_First
3  {
4      class GeraTabelas
5      {
6          static void Main(string[] args)
7          {
8              using (var context = new EditoraContext())
9              {
10                 // Neste momento as tabelas são geradas
11                 context.Editoras.Add(new Editora { Nome = "Abril", Email = "abril@email↩
                       .com" });
12             }
13         }
14     }
15 }
```

Através do SQL Server Management Express verifique se a tabela EDITORAS foi criada corretamente.

3.8 Sobrescrevendo o nome do Banco de Dados

Quando executamos a aplicação, o *EF4_Code_First.EditoraContext* foi criado. Uma maneira de sobrescrever a convenção do Code First para nome de banco de dados é adicionar um arquivo **App.config** ou **Web.config** que contenha a string de conexão juntamente com a classe do contexto. O arquivo **.config** pode ser acrescentado no projeto que contém o assembly também. Para adicionar o **App.config** ao projeto pelo *Microsoft Visual C# Express* basta clicar com o botão direito do mouse no projeto e clicar em **Add**, posteriormente escolher **Application Configuration File**. Acrescente o **connectionStrings** ao seu arquivo:

```
1
2  <connectionStrings>
3    <add
4      name="EditoraContext" providerName="System.Data.SqlClient"
5      connectionString="Server=.\SQLEXPRESS;Database=livraria;
6      Trusted_Connection=true;Integrated Security=True;MultipleActiveResultSets=True"/>
7  </connectionStrings>
```

Quando a aplicação for executada, o banco de dados **livraria** será criado. Devemos habilitar a opção **MultipleActiveResultSets=True** para permitir a leitura de objetos relacionados em

blocos **foreach** do C#, pois o EF tenta criar um novo leitor de dados e ocorrerá uma falha de execução a menos que a opção *MultipleActiveResultSets* seja **true**. Uma outra maneira de fazer a leitura dos objetos relacionados dentro do bloco **foreach** é através de uma **List**, que fecha o leitor de dados e permite você percorrer a coleção e acessar os objetos relacionados.

3.9 Exercícios

6. Remova o banco de dados **EF4_Code_First.EditoraContext**.

7. Sobrescreva a convenção do Code First para nome de banco de dados. Para isto acrescente ao projeto **EF4-Code-First** o arquivo App.config com a definição da string de conexão.

3.10 Manipulando entidades

Para manipular as entidades da nossa aplicação, devemos utilizar uma classe derivada de DBCONTEXT.

```
1
2 var context = new EditoraContext();
```

3.10.1 Persistindo

Para armazenar as informações de um objeto no banco de dados basta utilizar o método SAVECHANGES() do DBCONTEXT. As entidades com o estado **Added** são inseridas no banco de dados quando o método SAVECHANGES() é chamado. Utilize o método *System.Data.Entity.DbSet.Add*. O método *Add* adiciona a entidade ao contexto com o estado Added.

```
1 using (var context = new EditoraContext())
2 {
3     Editora editora = new Editora { Nome = "Abril", Email = "abril@email.com" };
4     //Adiciona o estado Added
5     context.Editoras.Add(editora);
6     context.SaveChanges();
7
8 }//fecha o contexto ao final
```

3.10.2 Buscando

Para obter um objeto que contenha informações do banco de dados basta utilizar o método *Find* do DBSET.

```
1 Editora editora = context.Editoras.Find(1);
```

3.10.3 Removendo

As entidades com o estado **Deleted** são removidas do banco de dados quando o método SAVECHANGES() é chamado. Utilize o método *System.Data.Entity.DbSet.Remove*. O método *Remove* remove a entidade do contexto e adiciona a entidade o estado Deleted.

```
1 Editora editora = context.Editoras.Find(1);
2 context.Editoras.Remove(editora);
3 context.SaveChanges();
```

3.10.4 Atualizando

Para alterar os dados de um registro correspondente a um objeto basta utilizar as propriedades. Quando as propriedades de uma entidade do contexto é alterada, o estado **Modified** é adicionado a esta entidade. Entidades com o estado Modified são atualizados no banco de dados quando o método SaveChanes é chamado.

```
1 Editora editora = context.Editoras.Find(1);
2 editora.Nome = "Abril S/A";
3 context.SaveChanges();
```

3.10.5 Listando

Para obter uma listagem com todos os objetos referentes aos registros de uma tabela, podemos utilizar o Language Integrated Query $LINQ$, que permite os desenvolvedores escreverem a consulta em C#.

```
1 var consulta = from e in context.Editoras
2                where e.Nome.Contains("A")
3                select e;
4 // Equivalente a: SELECT * FROM Editoras e where e.Nome Like 'A%'
5 foreach (var item in consulta)
6 {
7     System.Console.WriteLine("Editora: " + item.Nome);
8 }
```

3.11 Exercícios

8. Crie um teste para inserir editoras no banco de dados.

```
using System;

namespace EF4_Code_First
{
    class InsereEditoraComEF
    {
        static void Main(string[] args)
        {
            using (var context = new EditoraContext())
            {
                Editora editora = new Editora();
                Console.WriteLine("Digite o nome da Editora:");
                editora.Nome = Console.ReadLine();
                Console.WriteLine("Digite o email da Editora:");
                editora.Email = Console.ReadLine();
                //Adiciona editora ao contexto
                //Status: Added
                context.Editoras.Add(editora);

                //Persisto editora
                context.SaveChanges();
            }
        }
    }
}
```

9. Crie um teste para listar as editoras inseridas no banco de dados.

```
using System;
using System.Linq;

namespace EF4_Code_First
{
    class ListaEditoraComEF
    {
        static void Main(string[] args)
        {
            using (EditoraContext context = new EditoraContext())
            {
               var consulta = from e in context.Editoras
                              select e;
               foreach (var item in consulta)
               {
                   Console.WriteLine("{0}: {1} - {2}", item.Id,item.Nome,item.Email);
               }
            }
        }
    }
}
```

3.12 Repository

A classe DbContext e DbSet do EF oferece recursos suficientes para que os objetos do domínio sejam recuperados ou persistidos no banco de dados. Porém, em aplicações com alta complexidade e grande quantidade de código, "espalhar" as chamadas aos métodos do DbContext e DbSet pode gerar dificuldades na manutenção e no entendimento do sistema.

Para melhorar a organização das nossas aplicações, diminuindo o custo de manutenção e aumentando a legibilidade do código, podemos aplicar o padrão **Repository** do **DDD**(Domain

Driven Design).

Conceitualmente, um repositório representa o conjunto de todos os objetos de um determinado tipo. Ele deve oferecer métodos para recuperar e para adicionar elementos.

Os repositórios podem trabalhar com objetos prontos na memória ou reconstruí-los com dados obtidos de um banco de dados. O acesso ao banco de dados pode ser realizado através de ferramenta ORM como o Entity Framework.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Data.Entity;
5
6  namespace EF4_Code_First
7  {
8      public class EditoraRepository
9      {
10         DbContext context;
11
12         public EditoraRepository(DbContext context)
13         {
14             this.context = context;
15         }
16
17         public void Adiciona(Editora e)
18         {
19             context.Set<Editora>().Add(e);
20             context.SaveChanges();
21         }
22
23         public Editora Busca(int id)
24         {
25             return context.Set<Editora>().Find(id);
26         }
27
28         public List<Editora> BuscaTodas()
29         {
30             var consulta = from e in context.Set<Editora>()
31                            select e;
32             return consulta.ToList();
33         }
34
35     }
36 }
```

```
1  Database.SetInitializer<EditoraContext>(new DropCreateDatabaseIfModelChanges<↩
       EditoraContext>());
2
3  var context = new EditoraContext();
4
5  EditoraRepository repository = new EditoraRepository(context);
6
7  List<Editora> editoras = repository.BuscaTodas();
```

3.13 Exercícios

10. Implemente um repositório de editoras criando uma nova classe no projeto **EF4-Code-First**.

```
1  public class EditoraRepository
2  {
3      DbContext context;
4
5      public EditoraRepository(DbContext context)
6      {
7          this.context = context;
8      }
9
10     public void Adiciona(Editora e)
11     {
12         context.Set<Editora>().Add(e);
13         context.SaveChanges();
14     }
15
16     public Editora Busca(int id)
17     {
18         return context.Set<Editora>().Find(id);
19     }
20
21     public List<Editora> BuscaTodas()
22     {
23         var consulta = from e in context.Set<Editora>()
24                        select e;
25         return consulta.ToList();
26     }
27
28 }
```

11. Altere a classe **InsereEditoraComEF** para que ela utilize o repositório de editoras.

```
1  using System;
2
3  namespace EF4_Code_First
4  {
5      class InsereEditoraComEF
6      {
7          static void Main(string[] args)
8          {
9              var context = new EditoraContext();
10             EditoraRepository repository = new EditoraRepository(context);
11
12             Editora editora = new Editora();
13             Console.WriteLine("Digite o nome da Editora:");
14             editora.Nome = Console.ReadLine();
15             Console.WriteLine("Digite o email da Editora:");
16             editora.Email = Console.ReadLine();
17
18             repository.Adiciona(editora);
19
20         }
21     }
22 }
```

12. (Opcional) Altere a classe **ListaEditoraComEF** para que ela utilize o repositório de editoras.

Capítulo 4

Visão Geral do ASP .NET MVC

4.1 Necessidades de uma aplicação web

As aplicações web são acessadas pelos navegadores (browsers). A comunicação entre os navegadores e as aplicações web é realizada através de requisições e respostas definidas pelo protocolo **HTTP**. Portando, ao desenvolver uma aplicação web, devemos estar preparados para receber requisições HTTP e enviar respostas HTTP.

Além disso, na grande maioria dos casos, as aplicações web devem ser acessadas por diversos usuários simultaneamente. Dessa forma, o desenvolvedor web precisa saber como permitir o acesso simultâneo.

Outra necessidade das aplicações web é gerar conteúdo dinâmico. Por exemplo, quando um usuário de uma aplicação de email acessa a sua caixa de entrada, ele deseja ver a listagem atualizada dos seus emails. Portanto, é fundamental que a listagem dos emails seja gerada no momento da requisição do usuário. O desenvolvedor web precisa utilizar algum mecanismo eficiente que permita que o conteúdo que os usuários requisitam seja gerado dinamicamente.

Trabalhar diretamente com as requisições e repostas HTTP e criar um mecanismo eficiente para permitir o acesso simultâneo e para gerar conteúdo dinâmico não são tarefas simples. Na verdade, é extremamente trabalhoso implementar essas características. Por isso, a plataforma .NET oferece uma solução para diminuir o trabalho no desenvolvimento de aplicações web.

4.2 Visão Geral do ASP .NET MVC

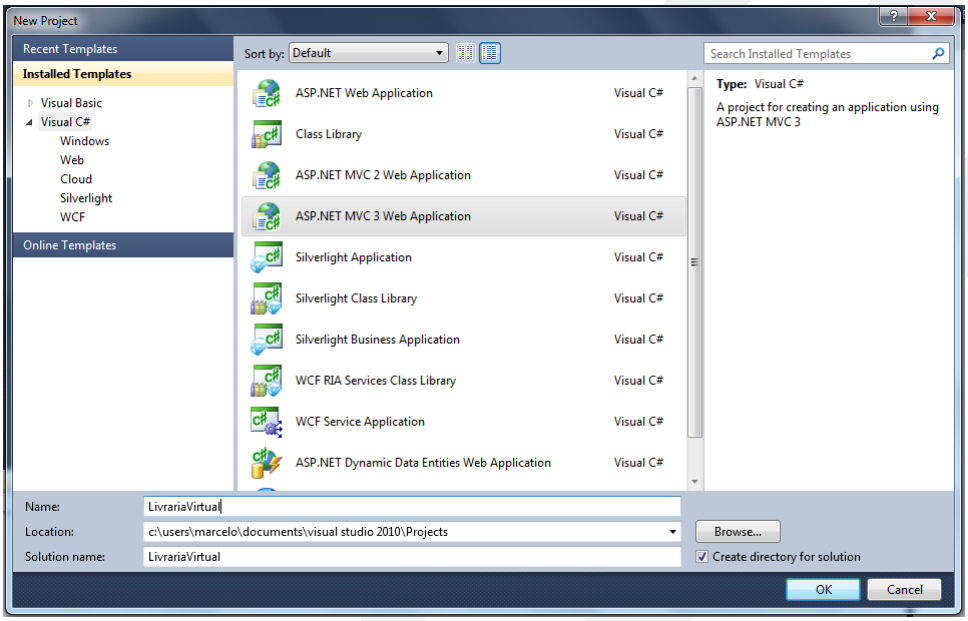
O ASP .NET MVC oferece muitos recursos para o desenvolvimento de uma aplicação web .NET. Cada um desses recursos por si só já são suficientemente grandes e podem ser abordados em separado.

Porém, no primeiro contato com ASP .NET MVC, é interessante ter uma visão geral dos recursos principais e do relacionamento entre eles sem se aprofundar em muito nos detalhes individuais de cada recurso.

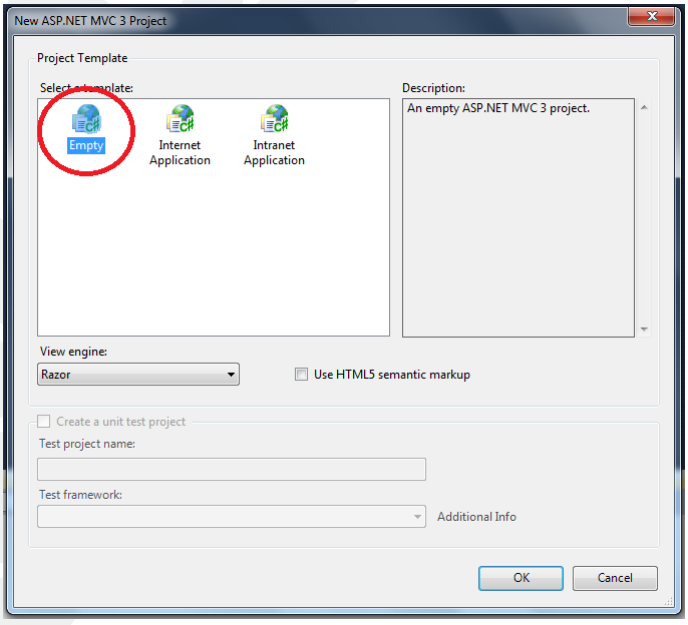
Portanto, neste capítulo, mostraremos de forma sucinta e direta o funcionamento e os conceitos principais do ASP .NET MVC. Nos próximos capítulos, discutiremos de maneira mais detalhada as diversas partes do ASP .NET MVC.

4.3 Aplicação de exemplo

O primeiro passo para construir uma aplicação web utilizando *ASP .NET MVC* é criar um projeto no *Visual Web Developer* a partir do modelo adequado. No nosso caso, o modelo de projeto que deve ser utilizado é o **ASP.NET MVC 3 Web Application**.



Devemos escolher *Empty Project* conforme figura abaixo:



O projeto criado já vem com diversas pastas e arquivos. Ao longo dos próximos capítulos, a função de cada pasta e de cada arquivo será discutida.

4.3.1 Testando a aplicação

Para verificar o funcionamento do projeto, basta executá-lo através do menu: *Debug -> Start Debugging*. Automaticamente um servidor de desenvolvimento é inicializado na máquina e a aplicação é implantada nesse servidor. Além disso, uma janela do navegador padrão do sistema é aberta na url principal da aplicação.

O servidor pode ser finalizado através do ícone *ASP.NET Development Server* que fica na barra de tarefas do Windows.

4.3.2 Trocando a porta do servidor

Para trocar a porta utilizada pelo servidor de desenvolvimento que o *Visual Web Developer* utiliza, basta alterar as propriedades do projeto clicando com o botão direito do mouse no projeto e escolhendo o item **properties** e depois a aba **web**.

4.4 Página de Saudação

Começaremos o desenvolvimento da nossa livraria virtual criando uma página de saudação. Essa página deve conter uma mensagem diferente de acordo com o horário atual.

Para implementar a página de saudação, devemos criar um **Controlador** que receberá as requisições vindas do navegador do usuário e devolverá um hipertexto XHTML gerado dinamicamente.

A base de todos os controladores é **System.Web.Mvc.ControllerBase**. A implementação padrão desta classe abstrata é **System.Web.Mvc.Controller**. Para criar nosso controlador, devemos criar uma classe que seja filha de **System.Web.Mvc.Controller** e o nome obrigatoriamente deve conter o sufixo **Controller**.

```
1  // LivrariaVirtual/Controllers/SaudacaoController.cs
2  using System;
3  using System.Web.Mvc;
4  using System.Web.Routing;
5
6  namespace LivrariaVirtual.Controllers
7  {
8      public class SaudacaoController : Controller
9      {
10         public string Index()
11         {
12             return "Welcome to ASP.NET MVC!";
13         }
14     }
15 }
```

Por padrão, a url que deve ser utilizada para enviar uma requisição a um controlador é a concatenação da url principal da aplicação seguido do nome do controlador (ex: `http://localhost/Saudacao`).

Por convenção, o arquivo *cs* contendo o código da classe do controlador deve ser colocado na pasta **Controllers**.

4.5 Exercícios

1. Crie um projeto do tipo *ASP.NET MVC 3 Web Application* chamado **LivrariaVirtual**. Crie um *Empty Project*.

2. Implemente uma página de saudação criando uma classe dentro da pasta *Controllers* chamada **SaudacaoController**.

```
1  using System.Web.Mvc;
2
3  namespace LivrariaVirtual.Controllers
4  {
5      public class SaudacaoController : Controller
6      {
7          //
8          // GET: /Saudacao/
9
10         public string Index()
11         {
12             return "Welcome to ASP.NET MVC!";
13         }
14
15     }
16 }
```

4.6 Alterando a página inicial

Por padrão, as requisições direcionadas a url principal da aplicação são enviadas ao controlador **Home**. Podemos alterar esse comportamento, modificando o arquivo de rotas da aplicação, o **Global.asax**. Este código:

```
1  new { controller = "Home", action = "Index", id = UrlParameter.Optional }
```

Deve ser substituído por este:

```
1  new { controller = "Saudacao", action = "Index", id = UrlParameter.Optional }
```

4.7 Exercícios

3. Altere a página inicial da nossa aplicação para a página de saudação criada no exercício anterior.

4.8 Integrando o Banco de Dados

A nossa aplicação web vai interagir com o banco de dados para recuperar ou armazenar informação. Vimos que a comunicação com uma base de dados pode ser encapsulada através

do padrão *Repository*. Os Repository's e as classes que representam as entidades do sistema da livraria virtual devem ser colocadas na pasta **Models** do projeto web.

4.9 Exercícios

4. Copie as classes Repository e as classes de entidades implementadas no capítulo ADO.NET para a pasta Models da nossa aplicação web. Coloque cada classe separada em um arquivo com o mesmo nome da classe.

```
1
2  using System.Collections.Generic;
3
4  namespace LivrariaVirtual.Models
5  {
6      public class Editora
7      {
8          public int Id { get; set; }
9          public string Nome { get; set; }
10         public string Email { get; set; }
11         public virtual ICollection<Livro> Livros { get; set; }
12     }
13 }
```

```
1
2  namespace LivrariaVirtual.Models
3  {
4      public class Livro
5      {
6          public int Id { get; set; }
7          public string Titulo { get; set; }
8          public decimal Preco { get; set; }
9          public int EditoraId { get; set; }
10         public virtual Editora Editora { get; set; }
11     }
12 }
```

```
1
2  using System.Data.Entity;
3
4  namespace LivrariaVirtual.Models
5  {
6      public class LivrariaVirtualContext : DbContext
7      {
8          public DbSet<Editora> Editoras { get; set; }
9          public DbSet<Livro> Livros { get; set; }
10     }
11 }
```

```
1
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace LivrariaVirtual.Models
6  {
7      public class LivroRepository
8      {
9          private LivrariaVirtualContext context = new LivrariaVirtualContext();
10
11         public void Adiciona(Livro e)
12         {
13             context.Livros.Add(e);
14             context.SaveChanges();
15         }
16
17         public Livro Busca(int id)
18         {
19             return context.Livros.Find(id);
20         }
21
22         public List<Livro> BuscaTodas()
23         {
24             var consulta = from e in context.Livros
25                            select e;
26             return consulta.ToList();
27         }
28     }
29 }
```

```
1
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace LivrariaVirtual.Models
6  {
7      public class EditoraRepository
8      {
9          private LivrariaVirtualContext context  = new LivrariaVirtualContext();
10
11         public void Adiciona(Editora e)
12         {
13             context.Editoras.Add(e);
14             context.SaveChanges();
15         }
16
17         public Editora Busca(int id)
18         {
19             return context.Editoras.Find(id);
20         }
21
22         public List<Editora> BuscaTodas()
23         {
24             var consulta = from e in context.Editoras
25                            select e;
26             return consulta.ToList();
27         }
28     }
29 }
```

4.10 Listando entidades

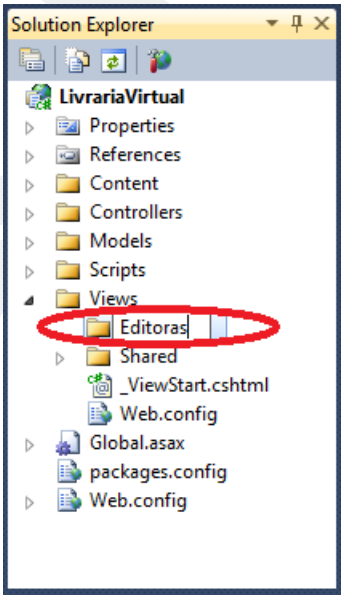
Uma funcionalidade básica que a nossa aplicação web deve oferecer para os usuários é a listagem das entidades do sistema (livro e editora). Para cada entidade, será criado um controlador que ficará encarregado de montar a lista da entidade correspondente.

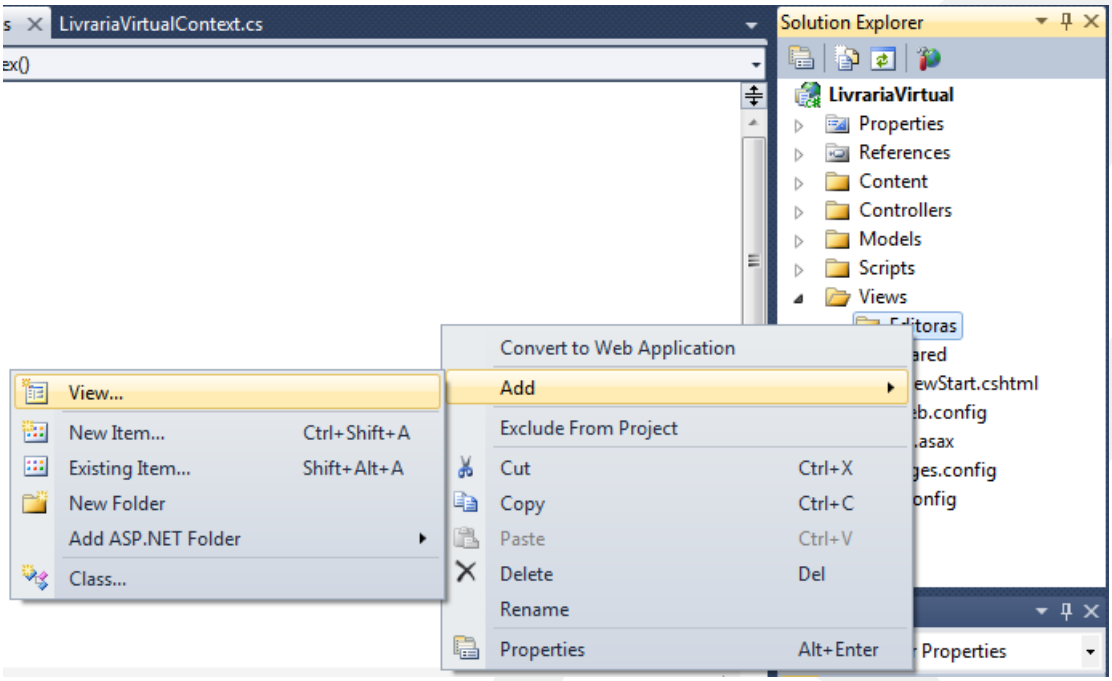
Vamos construir um exemplo que lista as editoras a partir da URL /Editoras.

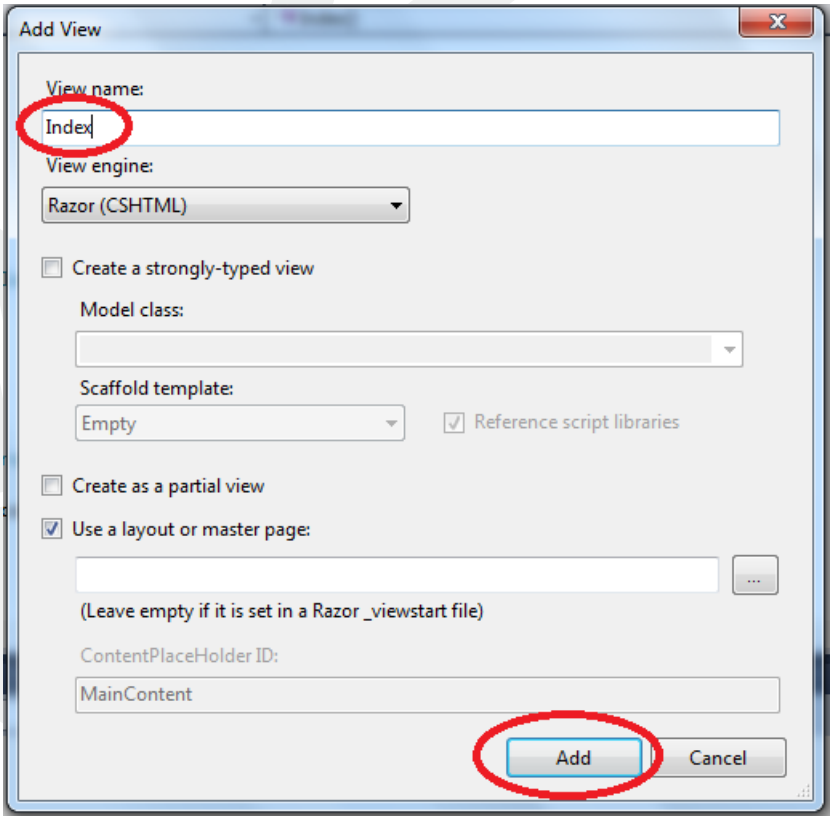
```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6  using LivrariaVirtual.Models;
7
8  namespace LivrariaVirtual.Controllers
9  {
10     public class EditorasController : Controller
11     {
12         //
13         // GET: /Editoras/
14
15         public ActionResult Index()
16         {
17             var editoraRepository = new EditoraRepository();
18
19             return View(editoraRepository.BuscaTodas());
20         }
21
22     }
23 }
```

Repare que o controlador responsável pela lista de editoras interage com o repository de Editora. Além disso, ele envia uma lista de editoras para a página através do método **View**.

Para listar as editoras que foram passados como parâmetro pelo **EditorasController** devemos criar uma página com o mesmo nome da action **Index**. Além disso, esta página deve ser criada, por convenção, dentro de uma pasta **Editoras**, pasta com o mesmo nome do nosso controller sem o sufixo Controller, dentro da pasta **Views**.




LivrariaVirtualContext.cs
Solution Explorer
LivrariaVirtual
Properties
References
Content
Controllers
Models
Scripts
Views
View...
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Add ASP.NET Folder
Class...
Convert to Web Application
Add
Exclude From Project
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Rename
Properties Alt+Enter


Add View
View name:
Index
View engine:
Razor (CSHTML)
Create a strongly-typed view
Model class:
Scaffold template:
Empty
Reference script libraries
Create as a partial view
Use a layout or master page:
(Leave empty if it is set in a Razor _viewstart file)
ContentPlaceHolder ID:
MainContent
Add
Cancel

```
1
2  @model IList<LivrariaVirtual.Models.Editora>
3
4  @{
5      ViewBag.Title = "Editoras";
6  }
7
8  <h2>Editoras</h2>
9
10 <table>
11     <tr>
12         <th>
13             Nome
14         </th>
15         <th>
16             Email
17         </th>
18         <th></th>
19     </tr>
20
21 @foreach (var item in Model) {
22     <tr>
23         <td>
24             @Html.DisplayFor(modelItem => item.Nome)
25         </td>
26         <td>
27             @Html.DisplayFor(modelItem => item.Email)
28         </td>
29         <td>
30             @Html.ActionLink("Edit", "Edit", new { id=item.Id })
31         </td>
32     </tr>
33 }
34
35 </table>
```

Para gerar o conteúdo dinâmico de nossa página, estamos utilizando **Razor** que permite acrescentar código de servidor juntamente com código HTML de forma mais clara e concisa. No ASP .NET MVC temos os **Helpers** que são classes que facilitam a criação de uma página e renderização de elementos HTML. Na nossa página **Index** estamos utilizando a propriedade **Html** que é uma instância da classe **System.Web.Mvc.HtmlHelper** que provê métodos para renderizar elementos como input, select, anchor, form. Veremos com mais detalhes os Helpers e Razor nos capítulos posteriores.

4.11 Exercícios

5. Implemente um controlador chamado *EditorasController* para que liste todas as editoras existentes na base de dados quando a url /Editoras for requisitada.

6. Implemente um controlador chamado *LivrosController* para que liste todas as editoras existentes na base de dados quando a url /Livros for requisitada.

4.12 Inserindo entidades

Outra funcionalidade fundamental que a aplicação web deve oferecer aos usuários é o cadastro de editoras e livros. O primeiro passo, para implementar essa funcionalidade, é criar um

formulário de cadastro de editora e outro de livro. Por exemplo, suponha que para criar uma editora devemos acessar a url /Editoras/Create. Primeiro devemos criar uma método para action **Create** no nosso controlador **EditorasController** que redirecionará para a página que contém o formulário, quando acessarmos a url /Editoras/Create pelo browser através de requisição *GET*. Por convenção, o nome do método é o mesmo nome da action.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6  using LivrariaVirtual.Models;
7
8  namespace LivrariaVirtual.Controllers
9  {
10     public class EditorasController : Controller
11     {
12         //
13         // GET: /Editoras/
14
15         public ActionResult Index()
16         {
17             var editoraRepository = new EditoraRepository();
18
19             return View(editoraRepository.BuscaTodas());
20         }
21
22         //
23         // GET: /Editoras/Create
24
25         public ActionResult Create()
26         {
27             return View();
28         }
29
30     }
31 }
```

Devemos agora criar a página que contém o formulário para inserir uma editora. Por padrão, esta página deverá ser criada na pasta View/Editoras com o mesmo nome da action, portanto deveremos ter um arquivo com o nome **Create.cshtml**.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>
@using (@Html.BeginForm())
{
    <fieldset>
        <legend>Editora</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
```

Quando o usuário clicar no botão "submit", uma action deverá receber a requisição com os dados preenchidos no formulário pelo usuário. Os dados serão enviados através de uma requisição *POST*, por padrão , e deveremos ter uma action **Create** que receberá os dados de uma requisição *POST*. Por convenção, deveremos ter no nosso controlador um método com o mesmo nome da action e restringiremos o método para tratar somente requisições POST com a anotação **HttpPost**. Neste método faremos a inserção da editora através da classe **EditoraRepository**.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6  using LivrariaVirtual.Models;
7
8  namespace LivrariaVirtual.Controllers
9  {
10     public class EditorasController : Controller
11     {
12         private EditoraRepository editoraRepository = new EditoraRepository();
13         //
14         // GET: /Editoras/
15
16         public ActionResult Index()
17         {
18             return View(this.editoraRepository.BuscaTodas());
19         }
20
21         //
22         // GET: /Editoras/Create
23
24         public ActionResult Create()
25         {
26             return View();
27         }
28
29         //
30         // POST: /Editoras/Create
31
32         [HttpPost]
33         public ActionResult Create(Editora editora)
34         {
35
36             editoraRepository.Adiciona(editora);
37             return RedirectToAction("Index");
38
39         }
40
41     }
42 }
```

4.13 Exercícios

7. Crie um método para action **Create** no nosso controlador **EditorasController** responsável por apresentar um formulário de cadastramento de editoras. Este formulário deverá ser acessado através de uma URL /Editoras/Create. Ao enviar os dados do formulário para o servidor através de uma requisição **POST**, defina um método para esta action que receba estes dados enviados pelo usuário e salve na base de dados utilizando a nossa classe EditoraRepository.

8. Crie um método para action **Create** no nosso controlador **LivrosController** responsável por apresentar um formulário de cadastramento de livros. Este formulário deverá ser acessado através de uma URL /Livros/Create. Ao enviar os dados do formulário para o servidor através de uma requisição **POST**, defina um método para esta action que receba estes dados enviados pelo usuário e salve na base de dados utilizando a nossa classe LivroRepository.

Devemos permitir que o usuário possa definir a editora a qual o livro pertence. Para isto, devemos ter uma caixa de seleção com todas as editoras da nossa base de dados. Antes de criar a caixa de seleção, devemos enviar uma lista para a nossa **View**, através da propriedade **ViewBag**, com todas as editoras da nossa base de dados.

```
1
2  using System;
3  using System.Collections.Generic;
4  using System.Data;
5  using System.Data.Entity;
6  using System.Linq;
7  using System.Web;
8  using System.Web.Mvc;
9  using LivrariaVirtual.Models;
10
11 namespace LivrariaVirtual.Controllers
12 {
13     public class LivrosController : Controller
14     {
15         private LivroRepository livroRepository = new LivroRepository();
16         private EditoraRepository editoraRepository = new EditoraRepository();
17
18         //
19         // GET: /Livros/
20
21         public ViewResult Index()
22         {
23
24             return View(livroRepository.BuscaTodas());
25         }
26
27         //
28         // GET: /Livros/Create
29
30         public ActionResult Create()
31         {
32             var consulta = editoraRepository.BuscaTodas().Select(e => new { e.Id, e.↩
                   Nome});
33             ViewBag.Editoras = consulta.ToList();
34             return View();
35         }
36
37         //
38         // POST: /Livros/Create
39
40         [HttpPost]
41         public ActionResult Create(Livro livro)
42         {
43             livroRepository.Adiciona(livro);
44             return RedirectToAction("Index");
45         }
46
47
48
49
50     }
51 }
```

Para construir a nossa caixa de seleção, podemos utilizar o método **DropDownListFor** da propriedade **Html**.

```
1
2  @model LivrariaVirtual.Models.Livro
3
4  @{
5      ViewBag.Title = "Insere Livro";
6  }
7
8  <h2>Insere Livro</h2>
9
10 @using (Html.BeginForm()) {
11     <fieldset>
12         <legend>Livro</legend>
13
14         <div class="editor-label">
15             @Html.LabelFor(model => model.Titulo)
16         </div>
17         <div class="editor-field">
18             @Html.EditorFor(model => model.Titulo)
19         </div>
20
21         <div class="editor-label">
22             @Html.LabelFor(model => model.Preco)
23         </div>
24         <div class="editor-field">
25             @Html.EditorFor(model => model.Preco)
26         </div>
27         <div class="editor-label">
28             @Html.LabelFor(model => model.EditoraId)
29         </div>
30         <div class="editor-field">
31             @Html.DropDownListFor(model => model.EditoraId,new SelectList(@ViewBag.↩
                   Editoras, "Id", "Nome"))
32         </div>
33
34         <p>
35             <input type="submit" value="Create" />
36         </p>
37     </fieldset>
38 }
```

4.14 Alterando entidades

Normalmente surge a necessidade de atualizar os dados de uma editora ou livro. Por exemplo, uma atualização dos preços. Portanto, a nossa aplicação deve permitir que o usuário faça alterações nos livros e nas editoras.

Suponha que para alterar as informações da editora, o usuário precise acessar a URL /Editoras/Edit/1, onde o 1 (um) define o ID da editora que o usuário deseje alterar as informações e será passado como parâmetro na nossa action. O primeiro passo é definir um método para a action **Edit** no controlador **EditorasController** que receberá o Id da editora fornecido pelo usuário e encaminhará para o formulário de edição.

```
1
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Web;
6  using System.Web.Mvc;
7  using LivrariaVirtual.Models;
8
9  namespace LivrariaVirtual.Controllers
10 {
11     public class EditorasController : Controller
12     {
13         private EditoraRepository editoraRepository = new EditoraRepository();
14         //
15         // GET: /Editoras/
16
17         public ActionResult Index()
18         {
19             return View(this.editoraRepository.BuscaTodas());
20         }
21
22         //
23         // GET: /Editoras/Create
24
25         public ActionResult Create()
26         {
27             return View();
28         }
29
30         //
31         // POST: /Editoras/Create
32
33         [HttpPost]
34         public ActionResult Create(Editora editora)
35         {
36
37             editoraRepository.Adiciona(editora);
38             return RedirectToAction("Index");
39
40         }
41
42         //
43         // GET: /Editoras/Edit/5
44
45         public ActionResult Edit(int id)
46         {
47             Editora editora = editoraRepository.Busca(id);
48             return View(editora);
49         }
50
51
52     }
53 }
```

O formulário de edição deverá vir preenchido com as informações da editora que o usuário definiu. Para isto, passamos como parâmetro **editora** no método **View** e acessamos através da propriedade **Model**.

```
1
2  @model LivrariaVirtual.Models.Editora
3
4  @{
5      ViewBag.Title = "Edit";
6  }
7
8  <h2>Edit</h2>
9
10 @using (Html.BeginForm()) {
11     <fieldset>
12         <legend>Editora</legend>
13
14         @Html.HiddenFor(model => model.Id)
15
16         <div class="editor-label">
17             @Html.LabelFor(model => model.Nome)
18         </div>
19         <div class="editor-field">
20             @Html.EditorFor(model => model.Nome)
21         </div>
22
23         <div class="editor-label">
24             @Html.LabelFor(model => model.Email)
25         </div>
26         <div class="editor-field">
27             @Html.EditorFor(model => model.Email)
28         </div>
29
30         <p>
31             <input type="submit" value="Save" />
32         </p>
33     </fieldset>
34 }
```

Ao submeter o formulário, requisição POST por padrão, devemos ter um método para esta action que receberá os dados enviados e fará a alteração em nossa base de dados.

```
1
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Web;
6  using System.Web.Mvc;
7  using LivrariaVirtual.Models;
8
9  namespace LivrariaVirtual.Controllers
10 {
11     public class EditorasController : Controller
12     {
13         private EditoraRepository editoraRepository = new EditoraRepository();
14         //
15         // GET: /Editoras/
16
17         public ActionResult Index()
18         {
19             return View(this.editoraRepository.BuscaTodas());
20         }
21
22         //
23         // GET: /Editoras/Create
24
25         public ActionResult Create()
26         {
27             return View();
28         }
29
30         //
31         // POST: /Editoras/Create
32
33         [HttpPost]
34         public ActionResult Create(Editora editora)
35         {
36
37             editoraRepository.Adiciona(editora);
38             return RedirectToAction("Index");
39
40         }
41
42         //
43         // GET: /Editoras/Edit/5
44
45         public ActionResult Edit(int id)
46         {
47             Editora editora = editoraRepository.Busca(id);
48             return View(editora);
49         }
50
51         //
52         // POST: /Editoras/Edit/5
53
54         [HttpPost]
55         public ActionResult Edit(Editora e)
56         {
57             editoraRepository.Atualiza(e);
58             return RedirectToAction("Index");
59         }
60
61     }
62 }
```

Devemos acrescentar na nossa classe **EditoraRepository** o método **Atualiza**.

```
1
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Data;
5
6  namespace LivrariaVirtual.Models
7  {
8      public class EditoraRepository
9      {
10         private LivrariaContext context  = new LivrariaContext();
11
12         public void Adiciona(Editora e)
13         {
14             context.Editoras.Add(e);
15             context.SaveChanges();
16         }
17
18         public void Atualiza(Editora e)
19         {
20             context.Entry(e).State = EntityState.Modified;
21             context.SaveChanges();
22         }
23
24         public Editora Busca(int id)
25         {
26             return context.Editoras.Find(id);
27         }
28
29         public List<Editora> BuscaTodas()
30         {
31             var consulta = from e in context.Editoras
32                            select e;
33             return consulta.ToList();
34         }
35     }
36 }
```

4.14.1 Link de alteração

Nas listagens de editoras e livros, podemos acrescentar um link alteração para cada item. Para isso, devemos alterar as páginas de listagem.

```
1
2  @model IList<LivrariaVirtual.Models.Editora>
3
4  @{
5      ViewBag.Title = "Editoras";
6  }
7
8  <h2>Editoras</h2>
9
10 <table>
11     <tr>
12         <th>
13             Nome
14         </th>
15         <th>
16             Email
17         </th>
18         <th></th>
19     </tr>
20
21 @foreach (var item in Model) {
22     <tr>
23         <td>
24             @Html.DisplayFor(modelItem => item.Nome)
25         </td>
26         <td>
27             @Html.DisplayFor(modelItem => item.Email)
28         </td>
29         <td>
30             @Html.ActionLink("Edit", "Edit", new { id=item.Id })
31         </td>
32     </tr>
33 }
34
35 </table>
```

4.15 Exercícios

9. Implemente um método para a action **Edit** no controlador **EditorasController** que será responsável por apresentar um formulário para a atualização de uma editora. Ao submeter o formulário devemos ter um método para esta action que receberá os dados enviados e fará a alteração em nossa base de dados. Não esqueça da modificar a página que lista as editoras para chamar o formulário de edição através de um link.

10. Implemente um método para a action **Edit** no controlador **LivrosController** que será responsável por apresentar um formulário para a atualização de um livro. Ao submeter o formulário devemos ter um método para esta action que receberá os dados enviados e fará a alteração em nossa base de dados. Não esqueça da modificar a página que lista os livros para chamar o formulário de edição através de um link.

    No **LivrosController** devemos ter:

```
1
2  //
3  // GET: /Livros/Edit/5
4
5  public ActionResult Edit(int id)
6  {
7      Livro livro = livroRepository.Busca(id);
8      var consulta = editoraRepository.BuscaTodas().Select(e => new { e.Id, e.Nome });
9      ViewBag.Editoras = consulta.ToList();
10     ViewBag.editora = new Editora { Nome = "iuiuiu" };
11     return View(livro);
12 }
13
14 //
15 // POST: /Editoras/Edit/5
16
17 [HttpPost]
18 public ActionResult Edit(Livro l)
19 {
20     livroRepository.Atualiza(l);
21     return RedirectToAction("Index");
22 }
```

Classe **LivroRepository**:

```
1
2  public void Atualiza(Livro l)
3  {
4      context.Entry(l).State = EntityState.Modified;
5      context.SaveChanges();
6  }
```

A página **Edit.cshtml**:

```
1
2  @model LivrariaVirtual.Models.Livro
3
4  @{
5      ViewBag.Title = "Edit";
6  }
7
8  <h2>Alterar Livro</h2>
9
10 @using (Html.BeginForm()) {
11     <fieldset>
12         <legend>Livro</legend>
13
14         @Html.HiddenFor(model => model.Id)
15
16         <div class="editor-label">
17             @Html.LabelFor(model => model.Titulo)
18         </div>
19         <div class="editor-field">
20             @Html.EditorFor(model => model.Titulo)
21         </div>
22
23         <div class="editor-label">
24             @Html.LabelFor(model => model.Preco)
25         </div>
26         <div class="editor-field">
27             @Html.EditorFor(model => model.Preco)
28         </div>
29
30         <div class="editor-label">
31             @Html.LabelFor(model => model.EditoraId)
32         </div>
33
34         <div class="editor-field">
35             @Html.DropDownListFor(model => model.EditoraId, new SelectList(ViewBag.↩
                   Editoras,"Id","Nome"))
36         </div>
37
38         <p>
39             <input type="submit" value="Save" />
40         </p>
41     </fieldset>
42 }
```

4.16 Removendo entidades

Para finalizar nosso conjunto de funcionalidades básicas, implementaremos a remoção de entidades. Para isso podemos adicionar um link de remover para cada item das listagens de editoras e livros. Assim como fizemos com os links de alteração.

Segue a listagem de Editoras acrescentando o link de remoção:

```
1  @model IList<LivrariaVirtual.Models.Editora>
2
3  @{
4      ViewBag.Title = "Editoras";
5  }
6
7  <h2>Editoras</h2>
8
9  <table>
10     <tr>
11         <th>
12             Nome
13         </th>
14         <th>
15             Email
16         </th>
17         <th></th>
18     </tr>
19
20 @foreach (var item in Model) {
21     <tr>
22         <td>
23             @Html.DisplayFor(modelItem => item.Nome)
24         </td>
25         <td>
26             @Html.DisplayFor(modelItem => item.Email)
27         </td>
28         <td>
29             @Html.ActionLink("Edit", "Edit", new { id=item.Id })
30         </td>
31         <td>
32             @Html.ActionLink("Delete", "Delete", new { id=item.Id })
33         </td>
34     </tr>
35 }
36
37 </table>
```

Depois de acrescentado o link de remoção nas listas, o próximo passo é implementar os métodos para estas actions nos controladores que farão a remoção através das classes Repository.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6  using LivrariaVirtual.Models;
7
8  namespace LivrariaVirtual.Controllers
9  {
10     public class EditorasController : Controller
11     {
12         private EditoraRepository editoraRepository = new EditoraRepository();
13
14         //
15         // GET: /Editoras/Delete/5
16         public ActionResult Delete(int id)
17         {
18             Editora editora = editoraRepository.Busca(id);
19             return View(editora);
20         }
21
22         //
23         // POST: /Editoras/Delete/5
24         [HttpPost, ActionName("Delete")]
25         public ActionResult DeleteConfirmed(int id)
26         {
27             Editora editora = editoraRepository.Busca(id);
28             editoraRepository.Remove(editora);
29             return RedirectToAction("Index");
30         }
31
32     }
33 }
```

Ao enviar uma requisição POST através da URL /Editoras/Delete/5, o método que tratará esta action será o **DeleteConfirmed**. Para isto, renomeamos a action com a anotação ActionName, pois por padrão a action contém o mesmo nome do método do controlador. Precisamos também definir uma página de confirmação da remoção da entidade:

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Apagar Editora";
}

<h2>Apagar Editora</h2>

<h3>Você tem certeza que deseja apagar esta editora?</h3>
<fieldset>
    <legend>Editora</legend>

    <div class="display-label">Nome</div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Nome)
    </div>

    <div class="display-label">Email</div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Email)
    </div>
</fieldset>
@using (Html.BeginForm()) {
    <p>
        <input type="submit" value="Delete" /> |
        @Html.ActionLink("Voltar para listagem de Editoras", "Index")
    </p>
}
```

4.17 Exercícios

11. Implemente os métodos para as actions de remoção no **EditorasController** que serão responsáveis por remover uma editora. Não se esqueça de modificar a página de listagem de editoras para incluir o link de remoção.

12. Implemente os métodos para as actions de remoção no **LivrosController** que serão responsáveis por remover um livro. Não se esqueça de modificar a página de listagem de livros para incluir o link de remoção.

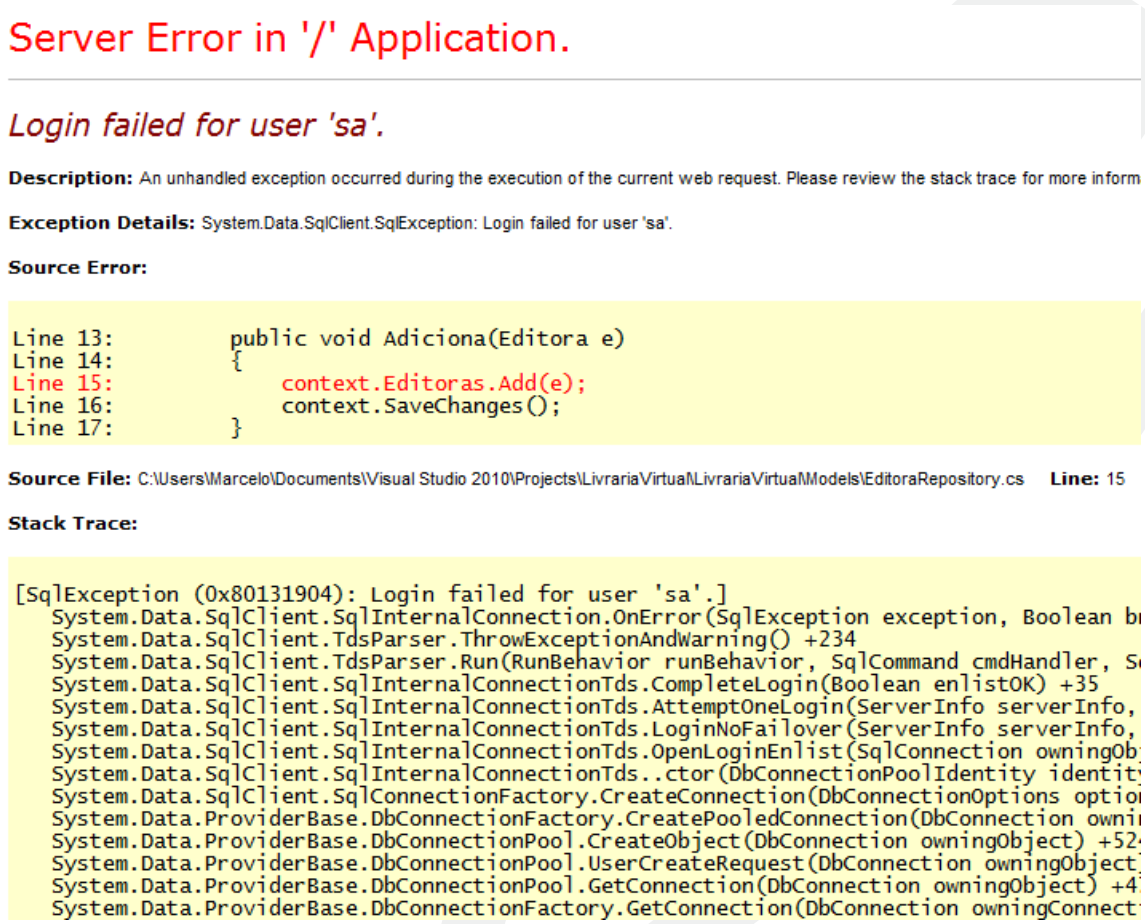
Capítulo 5

Tratamento de Erros

Inevitavelmente, as aplicações estão sujeitas a erros de várias naturezas. Por exemplo, erros gerados pelo preenchimento incorreto do campos de um formulário. Esse tipo de erro é causado por falhas dos usuários. Nesse caso, é importante mostrar mensagens informativas com o intuito de fazer o próprio usuário corrigir os valores preenchidos incorretamente.

Por outro lado, há erros que não são causados por falhas dos usuários. Por exemplo, um erro de conexão com o banco de dados. Nesses casos, é improvável que os usuários possam fazer algo que resolva o problema. E mesmo que pudessem, provavelmente, não seria conveniente esperar que eles o fizessem.

Quando um erro desse tipo ocorre, o ASP.NET cria uma página web com informações sobre o erro e a envia aos usuários. Para usuários locais, o ASP.NET envia uma página web com informações detalhadas do erro ocorrido. Para usuários remotos, a página web enviada não contém informações detalhadas.



Em geral, não é conveniente que os usuários recebam detalhes técnicos sobre os erros gerados por falhas da aplicação. A primeira justificativa é que esses detalhes podem confundir os usuários. A segunda justificativa é que esses detalhes podem expor alguma falha de segurança da aplicação deixando a mais vulnerável a ataques.

5.1 Try-Catch

Os erros de aplicação podem ser identificados através do comando *try-catch* que pode ser colocado nos métodos das actions dos controladores. Ao identificar a ocorrência de um erro, os controladores podem mostrar uma página web com alguma mensagem para o usuário.

```
[HttpPost]
public ActionResult Create(Editora editora)
{
    try
    {
        this.editoraRepository.Adiciona(editora);
    }
    catch
    {
        return View("Error");

    }

    return RedirectToAction("Index");

}
```

Devemos criar uma página **Error.cshtml**, por padrão, na pasta **Views/Shared**:

```

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <title>Erro</title>
</head>
<body>
    <h2>
        Servidor com problemas
    </h2>
    <p>
         Houve um problema no nosso servidor.<br/>
         Por favor tente novamente dentro de alguns instantes.
    </p>
</body>
</html>
```

As páginas de erro que serão mostradas pelos controladores teriam uma mensagem simples informando que houve um erro na aplicação e que não é possível atender a requisição do usuário naquele momento. Inclusive, seria conveniente padronizar a página de erro. Em outras palavras, todos os controladores teriam que mostrar a mesma página.

5.2 Exercícios

1. Altere o código do método da action **Create** do controlador *EditorasController* para capturar erros usando *try-catch*.

5.3 Custom Errors

Utilizar o comando *try-catch* nos controladores para lidar com os erros de aplicação não é uma boa alternativa pois o código do controlador fica mais complexo. Além disso, haveria replicação de código nos controladores pois provavelmente a página de erro seria padronizada.

Qualquer alteração na página de erro implicaria em alterações em todos os controladores. Por exemplo, se a mensagem tivesse que ser trocada.

Para lidar com os erros de aplicação de uma maneira mais prática e fácil de manter, podemos configurar o ASP.NET para utilizar páginas de erro padrão. O primeiro passo é alterar o arquivo de configuração **Web.config**, acrescentando a tag **customErrors** dentro da tag *system.web*.

```
1 <customErrors mode="On">
2 </customErrors>
```

O atributo **mode** da tag *customErrors* pode assumir três valores:

On: A página de erro padrão será enviada para usuários locais ou remotos.

Off: A página de erro detalhada será enviada para usuários locais ou remotos.

RemoteOnly: A página de erro detalhada será enviada para usuários locais e a padrão para os remotos.

Por convenção, o ASP .NET MVC mantém uma página de erro padrão dentro da pasta **Views/Shared** com o nome **Error.cshtml**. Vamos alterar este arquivo, o conteúdo da página de erro é basicamente XHTML.

```
1  @{
2      Layout = null;
3  }
4
5  <!DOCTYPE html>
6  <html>
7  <head>
8      <title>Erro</title>
9  </head>
10 <body>
11     <h2>
12         Servidor com problemas
13     </h2>
14     <p>
15          Houve um problema no nosso servidor.<br/>
16          Por favor tente novamente dentro de alguns instantes.
17     </p>
18 </body>
19 </html>
```

5.4 Exercícios

2. Configure nossa aplicação para utilizar páginas de erro padrão. Lembre-se que não vamos mais precisar do comando *try-catch* colocado no exercício anterior.

5.5 Erros do HTTP

Um dos erros mais conhecidos do HTTP é o 404 que ocorre quando o navegador faz uma requisição por uma url que não existe. Basicamente, esse erro é gerado por falhas dos usuários

ao tentarem digitar diretamente uma url na barra de endereço dos navegadores ou por links ou botões "quebrados" nas páginas da aplicação.

Quando o erro 404 ocorre, o ASP.NET utiliza a página padrão para erros de aplicação configurada no *Web.config* através da tag *customErrors*. Porém, esse erro não deve ser considerado um erro de aplicação pois ele pode ser gerado por falhas dos usuários. Ele também não deve ser considerado um erro de usuário pois ele pode ser gerado por falhas da aplicação. Consequentemente, é comum tratar o erro 404 de maneira particular criando uma página de erro específica para ele.

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <title>Arquivo não encontrado!</title>
</head>
<body>
    <h2>
        Esse arquivo não foi encontrado. Verifique se a url está correta.
    </h2>
</body>
</html>
```

No arquivo de configuração, podemos determinar uma página web específica para o erro 404 ou para os outros erros do HTTP.

```
<customErrors mode="On">
    <error statusCode="404" redirect="~/ErrorPages/NotFound"/>
</customErrors>
```

Devemos definir uma controlador com o nome **ErrorPages**, por padrão, além do método para a action **NotFound**.

```
using System.Web.Mvc;

namespace LivrariaVirtual.Controllers
{
    public class ErrorPagesController : Controller
    {

        public ActionResult NotFound()
        {
            return View();
        }

    }
}
```

5.6 Exercícios

3. Crie uma página de erro e um controlador específico para o erro 404 e modifique o arquivo *Web.config* para fazer redirecionamento apropriado.

Capítulo 6

Camada de Apresentação (View)

A responsabilidade da camada de apresentação é relativamente simples. Seu principal objetivo é gerar o conteúdo através do modelo. Não importa como o controlador obteve através de serviços e lógica de negócios os dados que foram necessários a construção do objeto do nosso modelo, a camada de apresentação apenas precisa saber como obter o modelo e gerar o conteúdo HTML através dele. Para desenvolver a camada de aplicação é necessário conhecimento de HTML, CSS e JavaScript. A camada de apresentação é responsável por gerar o conteúdo da nossa aplicação web e este conteúdo é dinâmico. Mostraremos neste capítulo, como podemos gerar conteúdo dinâmico através de funcionalidades do ASP .NET MVC como *Inline Code*, *HTML Helper* e *Partial Views*.

6.1 Inline Code

A maneira mais simples de gerar conteúdo dinâmico é através de **inline code** - que são blocos de código inseridos entre tags como (<% ... %>). Tags existentes também no PHP, Rails, JSP, que permite você inserir uma lógica simples para gerar o conteúdo dinâmico.

No ASP .NET MVC 3 podemos inserir código C# na View utilizando o *Razor* ao invés do tradicional **ASPX** que obriga colocar o código entre **<% ... %>**. A principal característica do Razor é ser conciso e simples, diminuindo o número de caracteres e tags scripts na View, pois diferentemente de outras sintaxes, não há necessidade de explicitar no código HTML um bloco de código de servidor.

Segue abaixo alguns exemplos código utilizando Razor e o equivalente em ASPX:

Bloco de código:

```
1 //Razor
2 @{
3   int x = 123;
4   string nome = "Marcelo";
5 }
```

```
//ASPX
<%
  int x = 123;
  string nome = "Marcelo.";
%>
```

Expressão:

```
<!-- Razor -->
<span>@model.Nome</span>
```

```
<!-- ASPX -->
<span><%= model.Nome %></span>
```

Texto e HTML

```
<!-- Razor -->
@foreach(var item in editoras) {
  <span>@item.Nome</span>
}
```

```
<!-- ASPX -->
<% foreach(var item in editoras) { %>
  <span><%= item.Nome %></span>
<% } %>
```

Código e texto

```
<!-- Razor -->
@if (foo) {
  @:Plain Text e @editora.Nome
}
```

```
<!-- ASPX -->
<% if (foo) { %>
  Plain Text e <%= editora.Nome %>
<% } %>
```

Comentários

```
<!-- Razor -->
@*
Comentário
*@
```

```
<!-- ASPX -->
<%--
Comentário
--%>
```

Alternando expressões e textos

```
<!-- Razor -->
Livro: @livro.Titulo - R$@livro.Preco.
```

```
<!-- ASPX -->
Livro: <%= livro.Titulo %> - R$<%= livro.Preco %>
```

6.2 Utilizando Inline Code

Suponha que tenhamos uma página *EditoraDetails.cshtml* e queremos mostrar as informações de um objeto da classe *Editora*:

```

public class Editora
{
    public int Id { get; set; }
    public string Nome { get; set; }
    public string Email { get; set; }
    public virtual ICollection<Livro> Livros { get; set; }
}
```

Para mostrar as informações da editora, utilizaremos a página *EditoraDetails.cshtml* e a definiremos como **strongly typed view** (veremos strongly typed views posteriormente) através do **inline code**: **@model <seu namespace>.Editora**.

```
1
2  @model LivrariaVirtual.Models.Editora
3
4  @{
5      Layout = null;
6  }
7
8  <!DOCTYPE html>
9
10 <html>
11 <head>
12         <title>@Model.Nome </title>
13     </head>
14     <body>
15         <h1>Editora @Model.Nome</h1>
16         <div>
17             Contato: @Model.Email
18         </div>
19
20         <h3>Livros:</h3>
21         <ul>
22             @foreach(var livro in @Model.Livros) {
23                 <li>
24                     <b>@livro.Titulo</b> - R$ @livro.Preco
25                 </li>
26             }
27         </ul>
28     </body>
29 </html>
```

Deveremos ter a seguinte tela:

6.3 ViewData, ViewBag e Model

No ASP .NET MVC, o controlador consegue fornecer dados a View através de:

- *Dicionário*: Através do *ViewDataDictionary* podemos fornecer dados através de key->value. **Key** é **string** e **value** é **object**. Por exemplo: ViewData["livros"] = livroRepository.BuscaTodas();

- *Propriedade Model*: Cada *ViewDataDictionary* contém um propriedade *Model* que armazena uma referência para um objeto qualquer. Podemos acessar este objeto através da palavra *Model* em nossa View, ao invés de *ViewData.Model* (ambos apontam para o mesmo objeto).

```
1  //
2  // GET: /Livros/Edit/5
3
4  public ActionResult Edit(int id)
5  {
6      Livro livro = livroRepository.Busca(id);
7      ViewData["Mensagem"] = "Hora Atual: " + DateTime.Now.ToShortTimeString();
8      ViewData["Editoras"] = editoraRepository.BuscaTodas();
9      //Equivalente a: ViewData.Model = livro;
10     return View(livro);
11 }
```

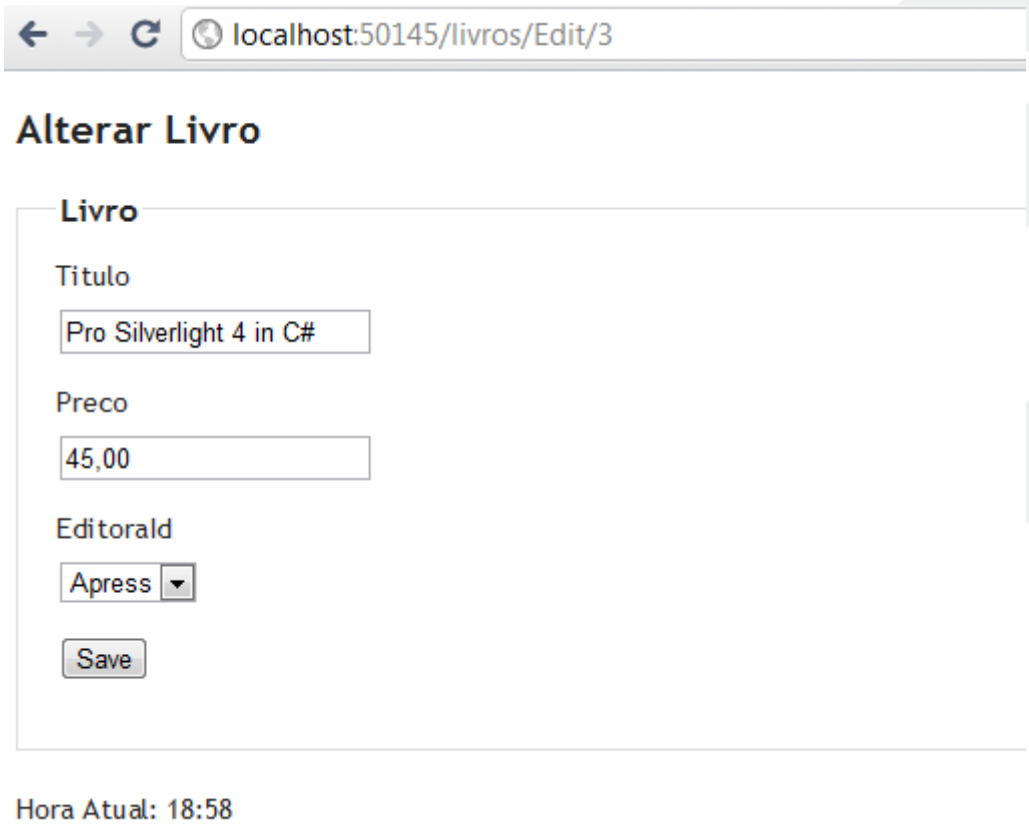
No ASP .NET MVC 3 podemos utilizar **ViewBag** ao invés de ViewData. ViewBag é uma coleção dinâmica que permite o envio de dados do controlador para a View. O suporte a coleção dinâmica é fruto do suporte a tipos dinâmicos do .NET 4. Segue um exemplo de ViewBag:

```
1  //
2  // GET: /Livros/Edit/5
3
4  public ActionResult Edit(int id)
5  {
6      Livro livro = livroRepository.Busca(id);
7      ViewBag.Mensagem = "Hora Atual: " + DateTime.Now.ToShortTimeString();
8      ViewBag.Editoras = editoraRepository.BuscaTodas();
9      //Equivalente a: ViewData.Model = livro;
10     return View(livro);
11 }
```

Segue abaixo um exemplo da nossa View strongly-typed (a classe esperada do *Model* é *Livro*):

```
1
2  @model LivrariaVirtual.Models.Livro
3
4  @{
5      ViewBag.Title = "Edit";
6  }
7
8  <h2>Alterar Livro</h2>
9
10 @using (Html.BeginForm()) {
11     <fieldset>
12         <legend>Livro</legend>
13
14         @Html.HiddenFor(model => model.Id)
15
16         <div class="editor-label">
17             @Html.LabelFor(model => model.Titulo)
18         </div>
19         <div class="editor-field">
20             @Html.TextBoxFor(model => model.Titulo)
21         </div>
22
23         <div class="editor-label">
24             @Html.LabelFor(w => w.Preco)
25         </div>
26         <div class="editor-field">
27             @Html.EditorFor(model => model.Preco)
28         </div>
29
30         <div class="editor-label">
31             @Html.LabelFor(model => model.EditoraId)
32         </div>
33
34         <div class="editor-field">
35             @Html.DropDownListFor(model => model.EditoraId, new SelectList(ViewBag.↩
                   Editoras,"Id","Nome"))
36         </div>
37
38         <p>
39             <input type="submit" value="Save" />
40         </p>
41     </fieldset>
42 }
43
44 <div>
45     @ViewBag.Mensagem
46 </div>
```

A combinação do controlador e View acima irá gerar o seguinte *response* HTML:



6.4 HTML Helpers

A função das páginas *.cshtml* é gerar hipertexto XHTML para enviar aos navegadores dos usuários. Os arquivos *.cshtml* misturam tags XHTML com scripts de servidor escritos em C# (ou outra linguagem de programação suportada pelo .NET). Essa mistura pode prejudicar a legibilidade do código em alguns casos. Veja o exemplo da listagem de editoras.

```
1
2
3  <ul>
4      @foreach(var e in Model)
5      {
6          <li>
7              @e.Nome - @e.Email
8              <a href="/Editoras/Edit/@e.Id">alterar</a>
9              <a href="/Editoras/Delete/@e.Id">remover</a>
10         </li>
11     }
12 </ul>
```

Para aumentar a legibilidade das páginas *.cshtml* e consequentemente facilitar a manutenção da aplicação, o ASP.NET oferece os chamados **HTML Helpers**. A função de um HTML Helper é encapsular um código XHTML. Por exemplo, para adicionar um link podemos usar o método **ActionLink** do objeto **Html**, ao invés, da tag *<a>*.

```
1 @Html.ActionLink("Veja os Livros", "Index", "Livros")
```

6.4.1 ActionLink Helper

O helper *ActionLink* é utilizado para gerar os links das páginas web. Esse helper aceita vários parâmetros e a maneira mais simples de utilizá-lo é passar a ele dois parâmetros: o texto do link e a ação que será chamada.

```
1 @Html.ActionLink("TEXTO PARA O USUÁRIO", "ACTION" )
```

Caso queiramos acrescentar um link para redirecionar para um outro controlador, devemos acrescentar um terceiro parâmetro:

```
1 @Html.ActionLink("TEXTO PARA O USUÁRIO", "ACTION", "CONTROLADOR" )
```

O ActionLink permite que parâmetros sejam adicionados no link gerado. Para isso, basta acrescentar um parâmetro.

```
1 @Html.ActionLink("TEXTO PARA O USUÁRIO", "ACTION", new {controller = "CONTROLADOR", ↩
      param = "valor"} )
```

6.4.2 Helpers de Formulários

Para facilitar a criação dos elementos de entrada de dados, há um conjunto de HTML Helpers.

Para criar um formulário, podemos utilizar o Helper *BeginForm*. Há duas maneiras para criar o formulário com o *BeginForm*:

Definindo o *Html.BeginForm()* e *Html.EndForm()*:

```
1 @{Html.BeginForm();}
2
3 <!-- Elementos de Formulário -->
4
5 @{Html.EndForm();}
```

Utilizando o bloco *using*:

```
1 @using(Html.BeginForm()) {
2
3 <!-- Elementos de Formulário -->
4
5 }
```

Caso não especifiquemos parâmetros na chamada do método *BeginForm*, o formulário enviará uma requisição para a mesma URL da requisição atual.

Ou podemos passar como parâmetro a **action** e o **controlador**:

```
@using(Html.BeginForm("ACTION", "CONTROLADOR")) {

<!-- Elementos de Formulário -->

}
```

Por padrão, o formulário enviará uma requisição *POST*. Devemos definir no nosso controlador o método que irá receber esta requisição.

```
//
// POST: /Editoras/Create
[HttpPost]
public ActionResult Create(Editora editora)
{
    this.editoraRepository.Adiciona(editora);
    return RedirectToAction("Index");
}
```

Para definir os campos do nosso formulário, podemos utilizar os HTML Helpers *string-based*:

Check Box:

```
<!-- Check Box Helper -->
@Html.CheckBox("meuCheckBox", false)
<!-- Sáida: -->
<input id="meuCheckBox" name="myCheckbox" type="checkbox" value="true" />
<input name="meuCheckBox" type="hidden" value="false" />
```

Text Box:

```
<!-- Text Box Helper -->
@Html.TextBox("meuTextbox", "valor")
<!-- Sáida: -->
<input id="meuTextbox" name="meuTextbox" type="text" value="valor" />
```

Text Area:

```
<!-- Text Area Helper -->
@Html.TextArea("meuTextarea", "valor")
<!-- Sáida: -->
<textarea cols="20" id="meuTextarea" name="meuTextarea" rows="2">valor</textarea>
```

Radio Button:

```
<!-- Radio Button Helper -->
@Html.RadioButton("meuRadiobutton", "valor", true)
<!-- Sáida: -->
<input checked="checked" id="meuRadiobutton" name="meuRadiobutton" type="radio" value="↩
    valor" />
```

Hidden Field:

```
<!-- Hidden Field Helper -->
@Html.Hidden("meuHidden", "valor")
<!-- Sáida: -->
<input id="meuHidden" name="meuHidden" type="hidden" value="valor" />
```

Password Field:

```
<!-- Password Field Helper -->
@Html.Password("meuPassword", "valor")
<!-- Sáida: -->
<input id="meuPassword" name="meuPassword" type="password" value="valor" />
```

Utilizando HTML Helpers *strongly typed*:
Check Box:

```
<!-- Check Box Helper -->
@Html.CheckBoxFor(x => x.IsAtivo)
<!-- Sáida: -->
<input id="IsAtivo" name="IsAtivo" type="checkbox" value="true" />
<input name="IsAtivo" type="hidden" value="false" />
```

Text Box:

```
<!-- Text Box Helper -->
@Html.TextBoxFor(x => x.Nome)
<!-- Sáida: -->
<input id="Nome" name="Nome" type="text" value="Valor do Nome" />
```

Text Area:

```
<!-- Text Area Helper -->
@Html.TextAreaFor(x => x.Descricao)
<!-- Sáida: -->
<textarea cols="20" id="Descricao" name="Descricao" rows="2">Valor da Descricao</↩
    textarea>
```

Radio Button:

```
<!-- Radio Button Helper -->
@Html.RadioButtonFor(x => x.IsAtivo, "valor")
<!-- Sáida: -->
<input checked="checked" id="IsAtivo" name="IsAtivo" type="radio" value="valor" />
```

Hidden Field:

```
<!-- Hidden Field Helper -->
@Html.HiddenFor(model => model.Id)
<!-- Sáida: -->
<input id="Id" name="Id" type="hidden" value="Valor do Id" />
```

Password Field:

```
<!-- Password Field Helper -->
@Html.PasswordFor(x => x.Password)
<!-- Sáida: -->
<input id="Password" name="Password" type="password"/>
```

Suponha que tenhamos o seguinte exemplo:

```
@Html.TextBox("Nome")
```

Isto é equivalente a:

```
@Html.TextBox("Nome", ViewData.Eval("Nome"))
```

Caso a **key** *Nome* da ViewDataDictionary da página não exista, o valor deste elemento será preenchido com *ViewData.Model.Nome*. Lembrando que para acessar o **value** associado a **key**, podemos utilizar, por exemplo, *ViewBag.Nome*, equivalente a *ViewData["Nome"]*.

No caso de Helpers HTML *strongly typed*, o valor do elemento será sempre associado a propriedade *ViewData.Model* da página.

6.4.3 DropDownList Helper

No cadastramento dos livros, os usuários podem escolher uma editora. A editora é selecionada através de um *drop down list*. Para criar um *drop down list*, podemos utilizar um HTML Helper.

O primeiro passo para utilizar o **DropDownList** é criar uma **SelectList** no controlador.

```
//
// GET: /Livros/Edit/5
public ActionResult Edit(int id)
{
    Livro livro = livroRepository.Busca(id);
    List<Editora> editoras = editoraRepository.BuscaTodas();
    ViewBag.EditoraId = new SelectList(editoras,"Id","Nome");
    return View(livro);
}
```

O segundo passo é adicionar o drop down list na página *.cshtml*.

```
<!-- String Based Helper -->
@Html.DropDownList("EditoraId")

<!-- Strongly Typed Helper -->
@Html.DropDownListFor(model => model.EditoraId, ViewBag.EditoraId as SelectList)
```

Pelo fato do Helper HTML *strongly typed* Drop Down List não aceitar tipos dinâmicos, devemos fazer o **cast** para **SelectList**.

6.4.4 Editor Helper

Suponha que tenhamos o seguinte exemplo:

```
1  public class Editora
2  {
3      public int Id { get; set; }
4      public string Nome { get; set; }
5      public string Email { get; set; }
6      public bool IsAtivo { get; set; }
7      public virtual ICollection<Livro> Livros { get; set; }
8  }
```

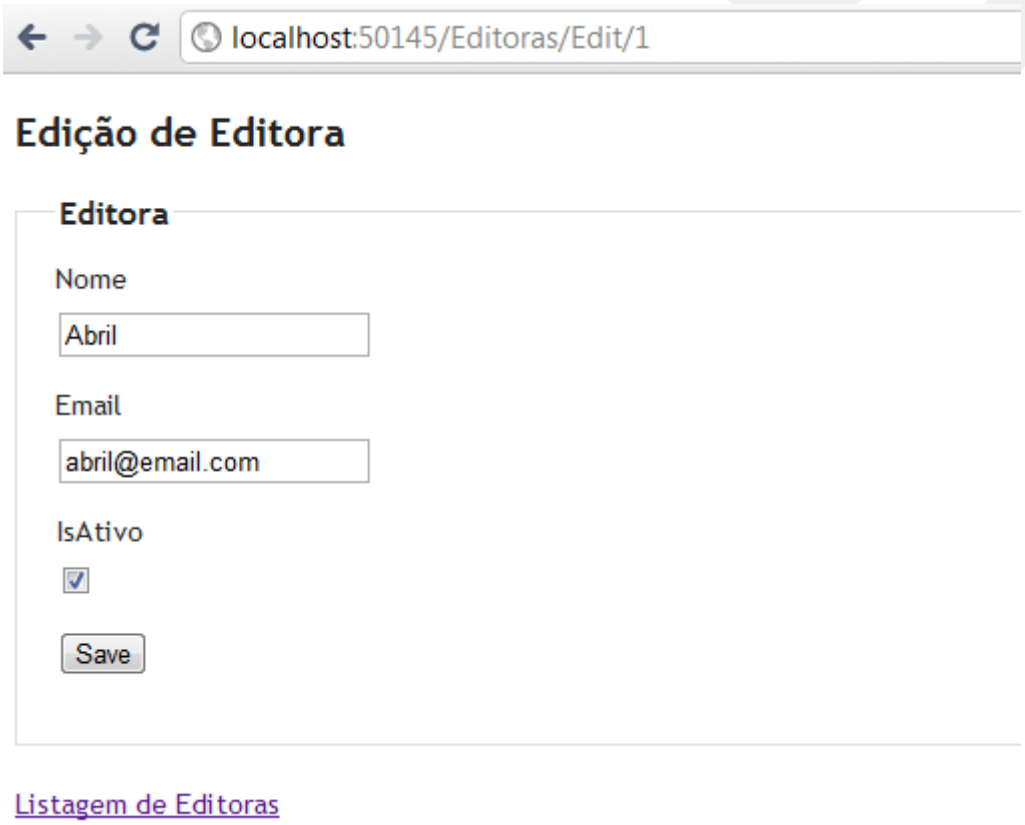
Para editar o cadastro de uma editora, teríamos uma página *Edit.cshtml* conforme exemplo abaixo:

```
1  @model LivrariaVirtual.Models.Editora
2
3  @{
4      ViewBag.Title = "Edição de Editora";
5  }
6
7  <h2>Edição de Editora</h2>
8
9  @using (Html.BeginForm()) {
10     @Html.ValidationSummary(true)
11     <fieldset>
12         <legend>Editora</legend>
13
14         @Html.HiddenFor(model => model.Id)
15
16         <div class="editor-label">
17             @Html.LabelFor(model => model.Nome)
18         </div>
19         <div class="editor-field">
20             @Html.TextBoxFor(model => model.Nome)
21         </div>
22
23         <div class="editor-label">
24             @Html.LabelFor(model => model.Email)
25         </div>
26         <div class="editor-field">
27             @Html.TextBoxFor(model => model.Email)
28         </div>
29
30         <div class="editor-label">
31             @Html.LabelFor(model => model.IsAtivo)
32         </div>
33         <div class="editor-field">
34             @Html.CheckBoxFor(model => model.IsAtivo)
35         </div>
36
37         <p>
38             <input type="submit" value="Save" />
39         </p>
40     </fieldset>
41 }
42
43 <div>
44     @Html.ActionLink("Listagem de Editoras", "Index")
45 </div>
```

Para cada propriedade do nosso modelo Editora, definimos um Helper apropriado para gerar

o elemento HTML para a entrada de dados. Por exemplo, no caso da propriedade *Nome* e *Email* utilizamos o Helper *Text Box*, já para a propriedade booleana *IsAtivo* utilizamos *CheckBox*.

Temos a seguinte tela:

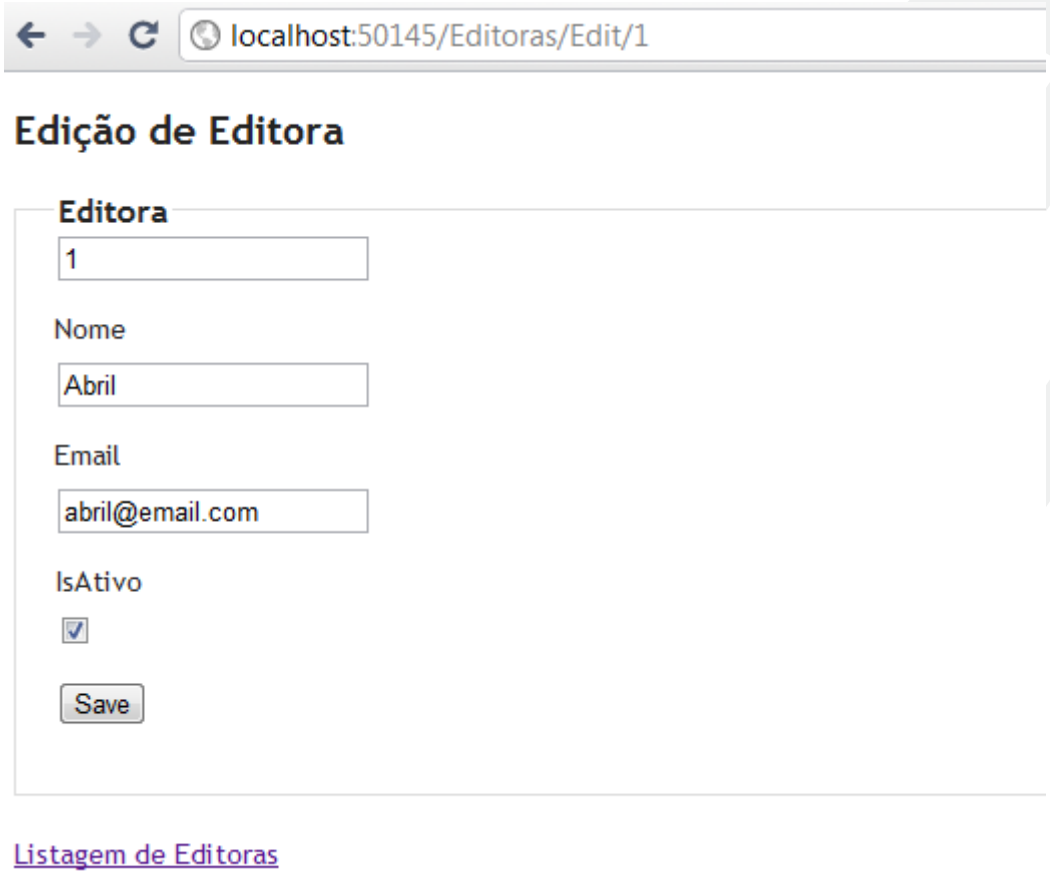


Podemos utilizar o Helper *Editor* que define o Helper HTML apropriado de acordo com o tipo da propriedade. Para propriedades e valores do tipo **booleano**, o helper utilizado será o **Check Box**, já para elementos do tipo *string*, o helper é o **Text Box**.

Podemos editar a nossa página *Edit.cshtml* para utilizar o Helper **Editor**:

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editora</legend>

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

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.IsAtivo)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.IsAtivo)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

Teremos a seguinte tela:



Podemos perceber que a tela ficou "parecida", porém para a propriedade *Id* foi definido o Helper *Text Box* e o mais apropriado é o Helper *Hidden*. O Helper *Editor* não consegue definir em todos os casos, o Helper mais apropriado. Para casos específicos, o ASP .NET MVC provê templates para definir o Helper mais apropriado para determinada propriedade do nosso modelo.

Por exemplo, para a propriedade *Id*, devemos acrescentar uma anotação indicando que para esta propriedade queremos utilizar o Helper *Hidden*.

```
1  using System.Collections.Generic;
2  using System.Web.Mvc;
3  namespace LivrariaVirtual.Models
4  {
5      public class Editora
6      {
7          [HiddenInput(DisplayValue = false)]
8          public int Id { get; set; }
9          public string Nome { get; set; }
10         public string Email { get; set; }
11         public bool IsAtivo { get; set; }
12         public virtual ICollection<Livro> Livros { get; set; }
13     }
14 }
```

Na propriedade *Id* acrescentamos a anotação *System.Web.Mvc.HiddenInputAttribute* que definirá para esta propriedade o Helper *Hidden*.

Com isto, teremos a seguinte tela:

Os templates para entrada de dados são:

Template	Descrição
HiddenInput	Utiliza o Helper *Hidden*
Text	Utiliza o Helper *Text*
String	Utiliza o Helper *TextBox*
Password	Utiliza o Helper *Password*
MultilineText	Utiliza o Helper *TextArea*
Boolean	Utiliza o Helper *CheckBox* ou *DropDownList* para nullable boolean
Decimal	Utiliza o Helper *TextBox* e formata para duas casas decimais
Object	Percorre as propriedades do objeto e define o Helper apropriado para cada uma
Collection	Percorre através do *IEnumerable* e define o Helper para cada elemento

Também temos o Helper *EditorForModel* que percorrerá todas as propriedades do *ViewData.Model* associado a página, e definirá o Helper apropriado para cada uma das propriedades.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editora</legend>

        @Html.EditorForModel()

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

6.5 Exercícios

1. Utilize o projeto *LivrariaVirtual* definido nos capítulos anteriores.

2. Altere o método da action *Edit* do controlador *Editoras* para enviar a informação da hora do servidor.

```
//
// GET: /Editoras/Edit/5

public ActionResult Edit(int id)
{
    Editora editora = editoraRepository.Busca(id);
    ViewBag.Hora = DateTime.Now.ToShortTimeString();
    return View(editora);
}
```

3. Altere a View *Editora.cshtml* para mostrar a hora do servidor.

```
<!-- Mostrando a hora atual -->
<div>
    Hora atual: @ViewBag.Hora
</div>
```

4. Altere a View *Editora.cshtml* para utilizar somente Helper's *string based*.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editora</legend>

        @Html.Hidden("Id")

        <div class="editor-label">
            @Html.Label("Nome")
        </div>
        <div class="editor-field">
            @Html.TextBox("Nome")
        </div>

        <div class="editor-label">
            @Html.Label("Email")
        </div>
        <div class="editor-field">
            @Html.TextBox("Email")
        </div>

        <div class="editor-label">
            @Html.Label("IsAtivo")
        </div>
        <div class="editor-field">
            @Html.CheckBox("IsAtivo")
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
<!-- Mostrando a hora atual -->
<div>
    Hora atual: @ViewBag.Hora
</div>
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

5. Acrescente o seguinte trecho código ao controlador *Editoras* e verifique o que acontece ao acessar a tela de edição de editoras.

```
//
// GET: /Editoras/Edit/5

public ActionResult Edit(int id)
{
    Editora editora = editoraRepository.Busca(id);
    ViewBag.Hora = DateTime.Now.ToShortTimeString();
    ViewBag.Nome = "Prioridade mais alta";
    return View(editora);
}
```

6. Altere a página *Edit.cshtml* de Editoras para utilizar o helper *Editor*.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editora</legend>

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

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.IsAtivo)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.IsAtivo)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
<!-- Mostrando a hora atual -->
<div>
    Hora atual: @ViewBag.Hora
</div>
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

7. Altere a classe Editora para que o Helper *Editor* defina o Helper *Hidden* para a propriedade *Id*.

```
using System.Collections.Generic;
using System.Web.Mvc;
namespace LivrariaVirtual.Models
{
    public class Editora
    {
        [HiddenInput(DisplayValue=false)]
        public int Id { get; set; }
        public string Nome { get; set; }
        public string Email { get; set; }
        public bool IsAtivo { get; set; }
        public virtual ICollection<Livro> Livros { get; set; }
    }
}
```

8. Altere a página *Editora.cshtml* para utilizar o Helper *EditorForModel*.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editora</legend>

        @Html.EditorForModel()

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
<!-- Mostrando a hora atual -->
<div>
    Hora atual: @ViewBag.Hora
</div>
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

6.6 Master Pages

É comum que as páginas de uma aplicação web possuam conteúdo em comum (por exemplo: um cabeçalho ou um rodapé). O conteúdo em comum pode ser replicado em todas as páginas através do CTRL+C e CTRL+V. Porém, essa não é uma boa abordagem pois quando alguma alteração precisa ser realizada, todos os arquivos devem ser modificados. Também é comum que as páginas de uma aplicação web possuam um certo padrão visual. Daí surge o conceito de **Master Page**.

6.6.1 Conteúdo comum

Tudo que é comum a todas as páginas de um determinado grupo pode ser definido em uma Master Page. Dessa forma, qualquer alteração é facilmente realizada modificando apenas um arquivo. Por exemplo, suponha que toda página da aplicação web da livraria virtual deva ter o mesmo título e a mesma formatação. Podemos criar uma Master Page com o título utilizado nas páginas e com a referência ao arquivo CSS que define a formatação padrão.

```
1  <!-- Views/Shared/LivrariaLayout.cshtml -->
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <title>@ViewBag.Title</title>
6      <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
7      <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">↩
           </script>
8  </head>
9
10 <body>
11     <div id="header">
12         @Html.ActionLink("Editoras", "Index","Editoras")
13         @Html.ActionLink("Livros", "Index","Livros")
14     </div>
15     @RenderBody()
16 </body>
17 </html>
```

Alguns detalhes:

- Estamos acrescentando *@RenderBody()* que indica onde o conteúdo das páginas será encaixado.
- No *title* acrescentamos *@ViewBag.Title*, isto permitirá que o title seja específico a cada página.

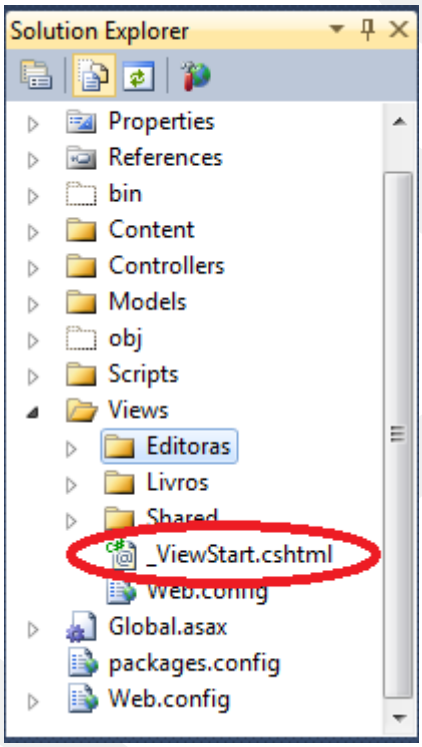
O próximo passo é indicar quais páginas utilizarão essa Master Page. Por exemplo, podemos atualizar a página *Edit.cshtml* de Editoras para utilizar a *LivrariaLayout.cshtml* como layout principal:

```
1  @model LivrariaVirtual.Models.Editora
2
3  @{
4      Layout = "~/Views/Shared/LivrariaLayout.cshtml";
5      //Define o title específico desta página
6      ViewBag.Title = "Edição de Editora";
7  }
8
9  <h2>Edição de Editora</h2>
10
11 @using(Html.BeginForm()) {
12     <fieldset>
13         <legend>Editora</legend>
14
15         @Html.EditorForModel()
16
17         <p>
18             <input type="submit" value="Save" />
19         </p>
20     </fieldset>
21 }
```

A página *Edit.cshtml* é executada antes de *LivrariaLayout.cshtml*, o que permitiu acrescentarmos valor ao *ViewBag.Title*. Isto também facilita na definição de elementos *meta* e *head*, por exemplo, para fins de SEO.

Para definir o layout de *Edit.cshtml* foi necessário acrescentar o caminho completo a propriedade *Layout*. Este procedimento não é muito prático, pois em cada página devemos definir esta propriedade.

No ASP .NET MVC 3, temos uma nova funcionalidade que permitirá definir um layout padrão a todas as páginas não havendo necessidade de definir a propriedade *Layout* em cada uma. Basta acrescentarmos a pasta *View* o arquivo *_ViewStart.cshtml*:



O *_ViewStart.cshtml* permite definirmos um código que será executado antes de cada View ser renderizada. Podemos definir, por exemplo, a propriedade *Layout*:

```
1 @{
2     Layout = "~/Views/Shared/LivrariaLayout.cshtml";
3 }
```

Como este código é executado no início de cada view, não há mais necessidade de definir a propriedade *Layout* em cada página.

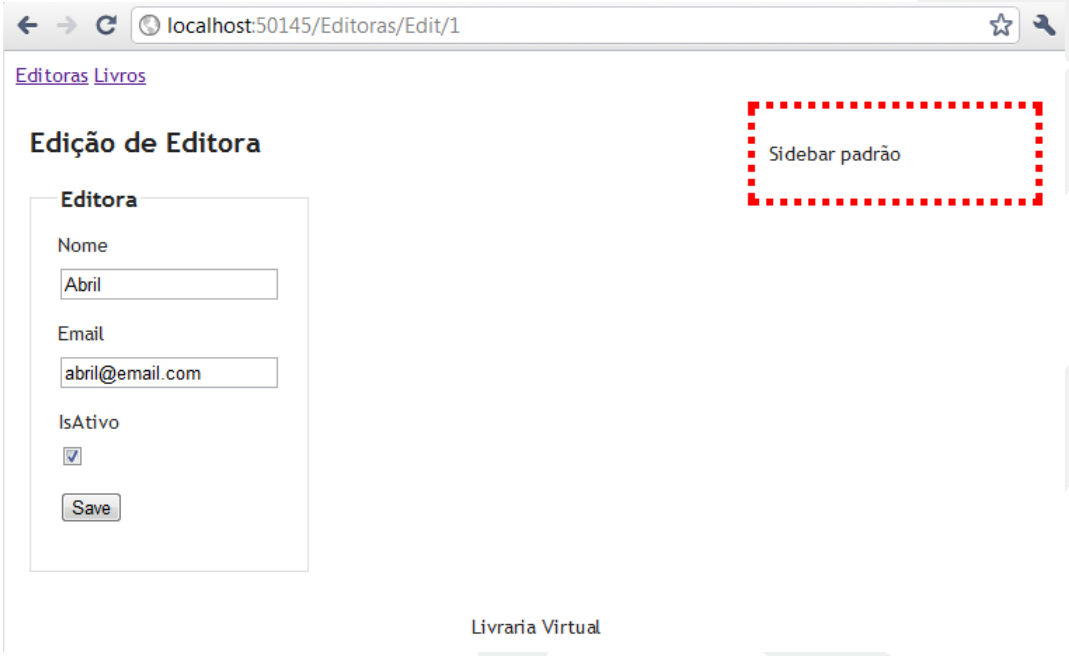
6.6.2 Lacunas

Também podemos criar "lacunas" na Master Page para serem preenchidas com conteúdos definidos nas páginas. Segue a página *LivrariaLayout.cshtml*:

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">
        </script>
</head>

<body>
    <div id="header">
        @Html.ActionLink("Editoras", "Index","Editoras")
        @Html.ActionLink("Livros", "Index","Livros")
    </div>
    <div id="sidebar">Sidebar padrão</div>
    <div id="content">@RenderBody()</div>
    <div id="footer">Livraria Virtual</div>

</body>
</html>
```

Acrescente o código abaixo ao arquivo *Content/Site.css*:

Com isto, teremos uma tela conforme a figura abaixo:



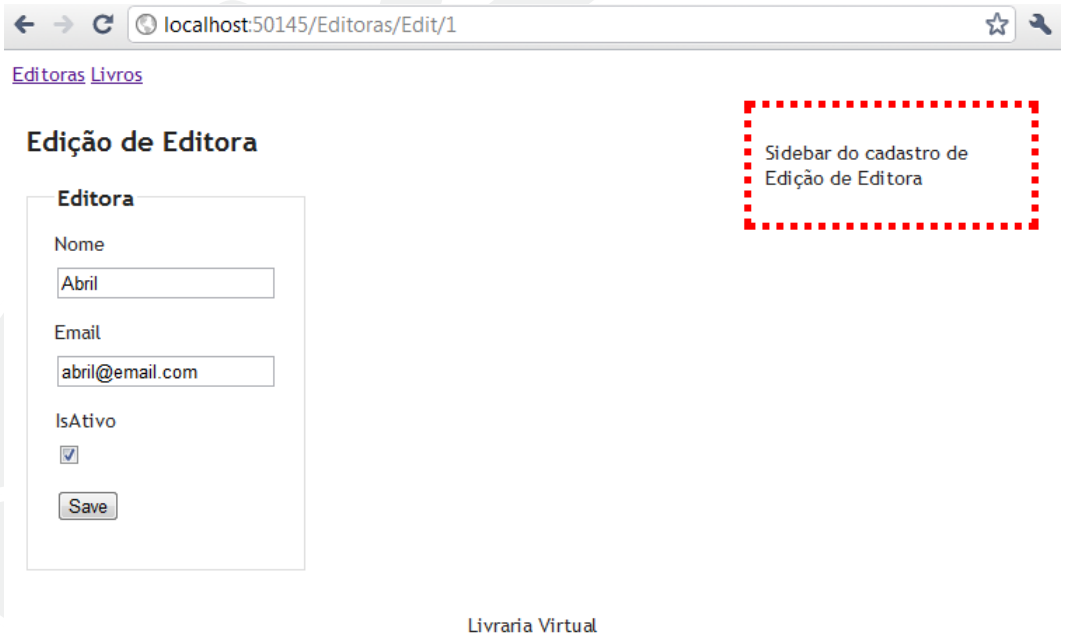
Para especificar uma "lacuna" em nosso layout, devemos utilizar o helper **@RenderSection(*string Nome Da Seção*, *bool Obrigatoriedade*)**:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>@ViewBag.Title</title>
5      <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
6      <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">↩
           </script>
7  </head>
8
9  <body>
10     <div id="header">
11         @Html.ActionLink("Editoras", "Index","Editoras")
12         @Html.ActionLink("Livros", "Index","Livros")
13     </div>
14     <!-- Definindo a lacuna "Sidebar" e indicando que é opcional (false) -->
15     <div id="sidebar">@RenderSection("sidebar",false)</div>
16     <div id="content">@RenderBody()</div>
17     <div id="footer">Livraria Virtual</div>
18
19 </body>
20 </html>
```

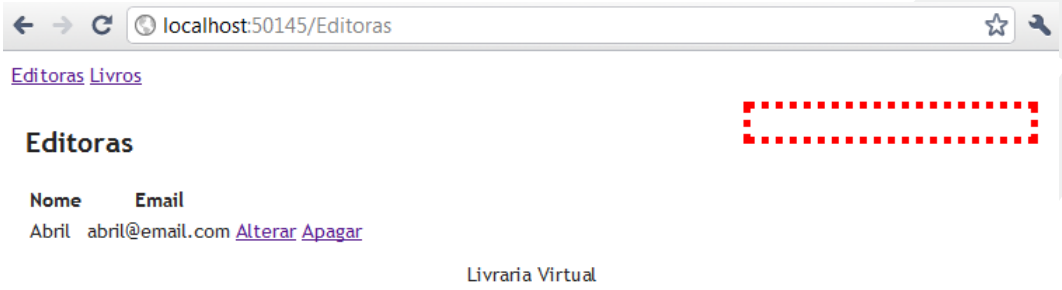
Para definir a seção "Sidebar", devemos utilizar o código *@section*. Segue a página *Edit.cshtml* de Editoras:

```
@model LivrariaVirtual.Models.Editora

@{
    Layout = "~/Views/Shared/LivrariaLayout.cshtml";
    //Define o title específico desta página
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using(Html.BeginForm()) {
    <fieldset>
        <legend>Editora</legend>

        @Html.EditorForModel()

        @section Sidebar {

            <p>Sidebar do cadastro de Edição de Editora</p>

        }

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

Com isto, ao acessar o formulário de edição de Editoras, teremos a seguinte tela:



Porém, ao acessar a tela de listagem de Editoras, temos a seguinte tela:



Para não ficar este "buraco", podemos definir um seção padrão para casos em que as páginas não definiram uma seção específica. Para isto, devemos fazer uma verificação na página *LivrariaLayout.cshtml* através do método *IsSectionDefined()*:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>@ViewBag.Title</title>
5      <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
6      <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">↩
           </script>
7  </head>
8
9  <body>
10     <div id="header">
11         @Html.ActionLink("Editoras", "Index","Editoras")
12         @Html.ActionLink("Livros", "Index","Livros")
13     </div>
14     @if (IsSectionDefined("Sidebar"))
15     {
16         <div id="sidebar">@RenderSection("Sidebar", false)</div>
17     }
18     else
19     {
20         <div id="sidebar">Sidebar padrão</div>
21     }
22
23     <div id="content">@RenderBody()</div>
24     <div id="footer">Livraria Virtual</div>
25
26 </body>
27 </html>
```

6.7 Exercícios

9. Utilize o projeto *LivrariaVirtual* para resolver os exercícios a seguir.

10. Crie uma página que servirá de layout para a nossa aplicação.

```
<!-- ~/Views/Shared/LivrariaLayout.cshtml -->
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">
        </script>
</head>

<body>
    <div id="header">
        @Html.ActionLink("Editoras", "Index","Editoras")
        @Html.ActionLink("Livros", "Index","Livros")
    </div>
   @RenderBody()

</body>
</html>
```

11. Altere a página *Edit.cshtml* de Editoras para utilizar a página de layout definido no exercício anterior.

```
@model LivrariaVirtual.Models.Editora

@{
    Layout = "~/Views/Shared/LivrariaLayout.cshtml";
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using(Html.BeginForm()) {
    <fieldset>
        <legend>Editora</legend>

        @Html.EditorForModel()

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
```

12. Defina a página *LivrariaLayout.cshtml* como layout padrão de cada view.

```
<!-- Basta criar a página _ViewStart.cshtml na pasta Views -->
@{
    Layout = "~/Views/Shared/LivrariaLayout.cshtml";
}
```

13. Defina seções na página *LivrariaLayout.cshtml*. Para isto, acrescente ao arquivo *Site.css* na pasta *Content* o trecho de código a seguir:

LivrariaLayout.cshtml Site.css

```
#sidebar
{
      float:right;
      margin:10px;
      padding:10px;
      border:dotted 5px red;
      width:180px;
}

#footer
{
    text-align:center;
    clear:both;
}

#content
{
    float:left;
    margin:10px;
}
```

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">
        </script>
</head>

<body>
    <div id="header">
        @Html.ActionLink("Editoras", "Index","Editoras")
        @Html.ActionLink("Livros", "Index","Livros")
    </div>

    <div id="sidebar">SideBar Padrão</div>
    <div id="content">@RenderBody()</div>
    <div id="footer">Livraria Virtual</div>

</body>
</html>
```

14. Acrescente uma “lacuna” a página *LivrariaLayout.cshtml*.

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">↩
        </script>
</head>

<body>
    <div id="header">
        @Html.ActionLink("Editoras", "Index","Editoras")
        @Html.ActionLink("Livros", "Index","Livros")
    </div>

    <div id="sidebar">@RenderSection("Sidebar", required:false)</div>
    <div id="content">@RenderBody()</div>
    <div id="footer">Livraria Virtual</div>

</body>
</html>
```

15. Defina na página *Edit.cshtml* de Editoras a seção “Sidebar”.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@using(Html.BeginForm()) {
    <fieldset>
        <legend>Editora</legend>

        @Html.EditorForModel()

        @section Sidebar {
            <p>Sidebar do cadastro de Edição de Editora</p>
        }

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
```

16. Acrescente uma seção padrão as páginas que não definiram a seção “Sidebar”.

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">↩
        </script>
</head>

<body>
    <div id="header">
        @Html.ActionLink("Editoras", "Index","Editoras")
        @Html.ActionLink("Livros", "Index","Livros")
    </div>
    @if (IsSectionDefined("Sidebar"))
    {
        <div id="sidebar">@RenderSection("Sidebar", required: false)</div>
    }
    else
    {
        <div id="sidebar">Sidebar padrão</div>
    }

    <div id="content">@RenderBody()</div>
    <div id="footer">Livraria Virtual</div>

</body>
</html>
```

6.8 Importação Automática

Quando é necessário utilizar uma classe ou interface nas páginas *.cshtml*, devemos acrescentar a diretiva *using* adequada. Algumas classes e interfaces são utilizadas em muitas páginas. Para não ter que adicionar a diretiva de importação em todas as páginas, podemos alterar o arquivo de configuração (Web.config) da pasta **Views** fazendo com que todas as páginas já tenham acesso a determinados *namespaces*.

```
<system.web.webPages.razor>
    <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version↩
        =3.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
    <pages pageBaseType="System.Web.Mvc.WebViewPage">
      <namespaces>
        <add namespace="System.Web.Mvc" />
        <add namespace="System.Web.Mvc.Ajax" />
        <add namespace="System.Web.Mvc.Html" />
        <add namespace="System.Web.Routing" />
        <add namespace="System.Linq"/>
        <add namespace="System.Collections.Generic"/>
        <add namespace="LivrariaVirtual.Models"/>
      </namespaces>
    </pages>
  </system.web.webPages.razor>
```

6.9 Exercícios

17. Edite o arquivo Web.config da pasta Views para fazer as importações de bibliotecas automaticamente.

6.10 Dividindo o conteúdo

Quanto mais elaborada é uma página web maior é o seu código. Quando o código é muito extenso a sua legibilidade fica prejudicada. Para organizar melhor o código, podemos dividir o conteúdo de uma página web em vários arquivos *.cshtml*.

Suponha que desejamos dividir o conteúdo de uma página em duas partes. Devemos criar um arquivo para cada parte. Normalmente, esses arquivos possuem a extesão *.ascx*.

```
1 <%-- Parte1.cshtml --%>
2 <h1> Parte 1 </h>
3 <p> Conteúdo da parte1</p>
```

```
1 <%-- Parte2.cshtml --%>
2 <h1> Parte 2 </h>
3 <p> Conteúdo da parte2</p>
```

Por fim, devemos criar um arquivo *.cshtml* principal para agrupar as partes. Utilizaremos o método *Partial* para inserir o conteúdo dos arquivos secundários no arquivo principal.

```
1  <html>
2      <head>
3          <title>Exemplo de partial</title>
4      </head>
5
6      <body>
7          Html.Partial("Parte1")
8          Html.Partial("Parte2")
9      </body>
10 </html>
```

O método *Partial* procura o arquivo *Parte1.cshtml* e *Parte2.cshtml* no mesmo diretório do arquivo principal. Caso ele não encontre, ele procura o arquivo na pasta *Views/Shared*. Isto serve também para o método *View* que utilizamos no controlador.

O *Partial View* permite criarmos conteúdo reutilizável de forma mais clara e concisa. As informações entre as Views e Partial Views podem ser compartilhadas através da *ViewBag*. Podemos, por exemplo, alterar as páginas *Create.cshtml* e *Edit.cshtml* de Editoras para acrescentar uma partial view do formulário, pois ambas as páginas compartilham o mesmo formulário.

```
<!-- ~/Views/Editoras/_Form.cshtml -->
@model LivrariaVirtual.Models.Editora

<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editora</legend>
        @if (Model != null)
        {
            @Html.EditorFor(model => model.Id)
        }

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.IsAtivo)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.IsAtivo)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
```

Segue o código das páginas de *Create.cshtml* e *Edit.cshtml*:

```
<!-- ~/Views/Editora/Create.cshtml -->
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Cadastro de Editora";
}

<h2>Cadastro de Editora</h2>

@Html.Partial("_Form")
```

```
<!-- ~/Views/Editora/Edit.cshtml -->
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@Html.Partial("_Form")
```

6.11 Exercícios

18. Crie uma partial view para o formulário de Editora.

```
<!-- ~/Views/Editoras/_Form.cshtml -->
@model LivrariaVirtual.Models.Editora

<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editora</legend>
        @if (Model != null)
        {
            @Html.EditorFor(model => model.Id)
        }

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.IsAtivo)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.IsAtivo)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
```

19. Altere as páginas *Create.cshtml* e *Edit.cshtml* de Editoras para utilizar a partial view *_Form.cshtml*.

```
<!-- ~/Views/Editora/Create.cshtml -->
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Cadastro de Editora";
}

<h2>Cadastro de Editora</h2>

@Html.Partial("_Form")
```

```
<!-- ~/Views/Editora/Edit.cshtml -->
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Edição de Editora";
}

<h2>Edição de Editora</h2>

@Html.Partial("_Form")
```

Capítulo 7

Controle (Controller)

No ASP .NET MVC as *URLs* são mapeadas para classes que são chamadas de *controllers*. Os controladores (*controllers*) processam as requisições, recebem dados enviados pelos usuários, executam comandos para recuperar dados do modelo e chamam a view apropriada para gerar o HTML para a requisição.

Os requisitos para uma classe ser considerada *controller* é:

- O nome deve terminar com o sufixo "Controller"
- A classe deve implementar a interface *IController* ou herdar da classe *System.Web.Mvc.Controller*

Raramente você definirá uma classe controller implementando a interface *IController*. Comumente definiremos uma classe **controller** herdando de *System.Web.Mvc.Controller*.

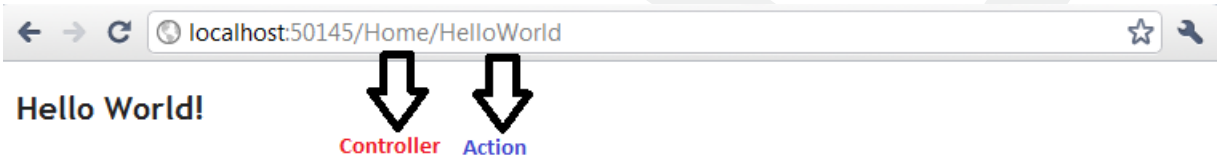
7.1 Actions

Nas aplicações ASP .NET que não utilizam MVC, as interações do usuário é em torno das páginas, em torno de eventos vindos da página e de seus controles. No ASP .NET MVC a interação do usuário é em torno dos *controllers* e *actions*. Uma classe *controller* contém métodos que são as *actions*. Uma *action* é utilizada para processar uma requisição HTTP e ela pode conter 0 (zero) ou mais argumentos. Para criar uma *action* é preciso definir o método como *public* e , na maioria das vezes, o valor de retorno será uma instância de uma classe que deriva de *ActionResult*.

Quando o usuário faz uma requisição através do Browser, o ASP .NET MVC verifica na tabela de rotas, definido no arquivo *Global.asax*, o *controller* que irá receber a requisição. O *controller* irá definir o método apropriado para processar a requisição. Por padrão, as URLs seguem a estrutura *{NomeDoControlador}/{NomeDaAction}*. Caso o usuário acesse a URL *http://www.exemplo.com/Editoras/Listagem*, por padrão, "Editoras" será considerado como o prefixo do nome do *controller* (EditorasController, o controlador termina com o sufixo *Controller*) e "Listagem" como o nome da *action*. Ao acessar a url /Editoras/Alterar/1, por padrão, "Alterar" será considerado uma action do controller "EditorasController" e 1 será enviado como parâmetro para o método "Alterar".

Exemplo de uma classe *controller* que contém uma action *HelloWorld*:

```
1  using System.Web.Mvc;
2
3  namespace LivrariaVirtual.Controllers
4  {
5      public class HomeController : Controller
6      {
7          //
8          // GET: /Home/HelloWorld
9
10         public ActionResult HelloWorld()
11         {
12             ViewBag.Mensagem = "Hello World!";
13             return View();
14         }
15
16     }
17 }
```



7.2 ActionResult

Após o *controller* receber a requisição e processá-la, ele devolve uma resposta para o usuário. O *controller* responde basicamente de três maneiras:

- Retorna uma resposta HTML ao chamar uma *View*
- Redireciona o usuário através do *HTTP Redirect*
- Retorna a resposta em outro formato. Por exemplo: XML, Json, arquivo binário

No ASP .NET MVC temos uma classe apropriada para cada tipo de retorno que é derivada de *ActionResult*.

Action Result	Descrição	Exemplo
ViewResult	Retorna uma View	*return View(); return View("NomeDaView", objetoModel);*
PartialViewResult	Retorna uma partial View, que pode ser inserida dentro de outra View	*return PartialView(); return PartialView("NomeDaPartialView", objetoModel);*
RedirectResult	Redireciona para uma URL específica	*return Redirect("http://www.k19.com.br");*
RedirectToRouteResult	Redireciona para outra action	*return RedirectToAction("OutraAction", "OutroController"); return RedirectToRoute("NomeDaRota");*
ContentResult	Retorna texto, *content-type* header opcional	*return Content("Texto","text\plain");*
JsonResult	Retorna um objeto no formato JSON	*return Json(objeto);*
JavaScriptResult	Retorna código Javascript que pode ser executado no cliente	*return JavaScript("$('#divResultText'). html('JavaScript Passed');");*
FileResult	Retorna dados binários (arquivo em disco, por exemplo)	*return File(@"c:\relatorio.pdf", "application\pdf");*
EmptyResult	Retorna um valor que é utilizado quando a action precisa retornar valor nulo	*return new EmptyResult();*

7.3 Parâmetros

Vimos que os parâmetros enviados pelos usuários podem ser recuperados nos controladores através da propriedade *Request*.

```
1 string nome = Request["nome"];
```

Mas, há outras maneiras de recuperar esses valores.

7.3.1 Vários parâmetros

Uma das maneiras de recuperar os dados enviados pelos usuários é definir um parâmetro C# para cada parâmetro HTTP enviado pelo usuário. É necessário definir os parâmetros C# com o mesmo nome dos parâmetros HTTP.

```
<html>
<head>
    <title>Cadastro de Editora</title>
</head>
<body>
    <form action="/Editoras/Salva" method="post">
        Nome: <input type="text" name="nome"/>
        Email: <input type="text" name="email"/>
        <input type="submit" />
    </form>
</body>
</html>
```

```
[HttpPost]
public ActionResult Salva(string nome, string email)
{
    Editora editora = new Editora { Nome = nome, Email = email };
    editoraRepository.Adiciona(editora);
    return View();
}
```

7.3.2 Por objeto

O ASP.NET também é capaz de montar objetos com os valores dos parâmetros HTTP enviados pelo usuário e passá-los como argumento aos controladores.

```
[HttpPost]
public ActionResult Salva(Editora e)
{
    editoraRepository.Adiciona(e);
    return View();
}
```

As propriedades dos objetos recebidos como argumentos nos métodos dos controladores precisam ter os mesmos nomes dos parâmetros HTTP.

7.4 Exercícios

1. Adicione no controlador Editoras do projeto LivrariaVirtual uma action para visualizar o formulário de cadastro.

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

2. Crie uma página *Cadastra.cshtml* de cadastro de editoras na pasta Views/Editoras conforme exemplo abaixo:

```
<h2>Cadastro de Editoras</h2>

<form action="/Editoras/Salva">
    Nome: <input name="nome" />
    Email: <input name="email" />
    <input type="submit" value="Enviar" />
</form>
```

3. Defina um método para action *Salva* no controlador Editoras que irá receber os dados enviados pelo usuário e adicionará uma editora ao banco de dados. Para receber os dados, utilize a propriedade *Request*.

```
public ActionResult Salva()
{
    Editora e = new Editora { Nome = Request["nome"], Email = Request["email"] };
    editoraRepository.Adiciona(e);
    return RedirectToAction("index");
}
```

4. Altere o método da action *Salva* para receber os dados como parâmetro.

```
public ActionResult Salva(string nome, string email)
{
    Editora e = new Editora { Nome = nome, Email = email };
    editoraRepository.Adiciona(e);
    return RedirectToAction("index");
}
```

5. Altere o método da action *Salva* para receber um objeto editora como parâmetro.

```
public ActionResult Salva(Editora e)
{
    editoraRepository.Adiciona(e);
    return RedirectToAction("index");
}
```

7.5 TempData

Ao efetuar um redirecionamento, uma nova requisição é efetuada pelo browser. Nesta nova requisição não temos mais acesso aos dados e objetos da requisição anterior ao redirecionamento. Caso haja a necessidade de preservar os dados ao longo do redirecionamento podemos utilizar o *TempData*.

Ao salvar uma Editora, por exemplo, efetuamos um redirecionamento para a tela de listagem. Podemos acrescentar uma mensagem indicando que a operação foi efetuada com sucesso.

```
[HttpPost]
public ActionResult Create(Editora editora)
{
    editoraRepository.Adiciona(editora);
    TempData["mensagem"] = "Editora criada com sucesso!";
    return RedirectToAction("Index");
}
```

Devemos acrescentar na tela de listagem o seguinte trecho de código:

```
@if (TempData["mensagem"] != null)
{
    <p>@TempData["mensagem"]</p>
}
```

7.6 Exercícios

6. Ao adicionar uma editora e redirecionar o usuário para a tela de listagem, mostre uma mensagem ao usuário indicando que a operação foi realizada com sucesso.

```
//EditorasController.cs
[HttpPost]
public ActionResult Create(Editora editora)
{
    editoraRepository.Adiciona(editora);
    TempData["mensagem"] = "Editora criada com sucesso!";
    return RedirectToAction("Index");
}
```

```
<!-- ~/Views/Editoras/Index.cshtml -->
@if (TempData["mensagem"] != null)
{
    <p>@TempData["mensagem"]</p>
}
```

7. (opcional) Mostre mensagens para o usuário nas operações de atualização e remoção de editoras.

Capítulo 8

Rotas

Para acessar uma determinada ação da nossa aplicação, os usuários devem utilizar a URL correspondente à ação. Por exemplo, para acessar a listagem de categorias, é necessário digitar na barra de endereço do navegador a seguinte url: `http://localhost/Editoras/Lista`. Perceba que o padrão é concatenar o nome do controlador com o nome do método desejado. Esse padrão é definido por uma **rota** criada no arquivo **Global.asax**.

```
1 routes.MapRoute(
2                 "Default", // Route name
3                 "{controller}/{action}/{id}", // URL with parameters
4                 new { controller = "Editoras", action = "Lista", id = UrlParameter.↩
                      Optional } // Parameter defaults
5             );
```

O primeiro argumento do método **MapRoute** é o nome da rota, o segundo é a expressão que define o formato da rota e o terceiro é o conjunto de valores padrões dos parâmetros da rota.

A expressão que determina do formato da rota define três parâmetros: o controlador que deve ser criado, o método que deve ser chamado no controlador e um argumento para esse método.

Dessa forma, se o usuário digitar a `http://localhost/Editoras/Remove/1` na barra de endereço do seu navegador o ASP.NET criará uma instância do controlador de editoras e executará o método *Remove* passando o valor *1* como argumento.

```
1 EditorasController controlador = new EditorasController(conexao);
2 controlador.Remove(1);
```

A rota define um padrão para URL e define como ela será tratada.

URL	Mapeamento da URL
/	new { controller = "Editoras", action = "Lista"}
/Livros	new { controller = "Livros", action = "Lista"}
/Livros/Adiciona	new { controller = "Livros", action = "Adiciona"}
/Livros/Remove/1	new { controller = "Livros", action = "Remove", id = 1 }

8.1 Adicionando uma rota

Para acrescentar uma rota podemos utilizar o método *MapRoute*. A ordem em que as rotas são acrescentadas é muito importante. *Acrescente rotas mais específicas antes de rotas menos específicas*. Suponha que queiramos acessar a nossa lista de livros através da URL */Biblioteca*.

```
routes.MapRoute("Nova Rota", "Biblioteca",
                new { controller = "Livros", action = "Index" });
```

8.2 Definindo parâmetros

Podemos acrescentar parâmetros a nossa rota. Podemos definir, por exemplo, a listagem de livros por editora.

```
routes.MapRoute("Nova Rota", "Biblioteca/{editora}",
                new { controller = "Livros", action = "Index" });
```

Quando definimos o parâmetro editora na rota acima, obrigatoriamente devemos passá-la na nossa URL. Para torná-la opcional, podemos utilizar *UrlParameter.Optional*.

```
routes.MapRoute("Nova Rota", "Biblioteca/{editora}",
                new { controller = "Livros", action = "Index", editora = UrlParameter.↩
                    Optional });
```

Podemos receber os parâmetros no método da nossa action.

```
//parâmetro editora
public ViewResult Index(string editora)
{
    List<Livro> livros;
    if (editora != null)
    {
        livros = livroRepository.BuscaPorEditora(editora);
    }
    else
    {
        livros = livroRepository.BuscaTodas();
    }
    return View(livros);
}
```

Ao definir parâmetros opcionais, devemos utilizar parâmetros do tipo *nullable type* nos métodos das actions. Pois quando não definimos o parâmetro na URL, é atribuído o valor **null** ao parâmetro do método. No caso de *int* e *double*, por exemplo, devemos utilizar ***int?***, ***double?***.

8.3 Exercícios

1. Acrescente uma nova rota para acessarmos a lista de livros através da URL */Biblioteca*.

```
1  //Global.asax
2  routes.MapRoute("Nova Rota", "Biblioteca",
3                  new { controller = "Livros", action = "Index" });
4
5  routes.MapRoute(
6                  "Default", // Route name
7                  "{controller}/{action}/{id}", // URL with parameters
8                  new { controller = "Editoras", action = "Index", id = UrlParameter.↩
                       Optional } // Parameter defaults
9              );
```

2. Acrescente um parâmetro *editora* a rota criada no exercício anterior.

```
1  //Global.asax
2  routes.MapRoute("Nova Rota", "Biblioteca/{editora}",
3                  new { controller = "Livros", action = "Index" });
4
5  routes.MapRoute(
6                  "Default", // Route name
7                  "{controller}/{action}/{id}", // URL with parameters
8                  new { controller = "Editoras", action = "Index", id = UrlParameter.↩
                       Optional } // Parameter defaults
9              );
```

3. Verifique o que acontece ao acessar a URL */Biblioteca* após o acréscimo do parâmetro *editora* à rota. Corrija este problema definindo o parâmetro *editora* como opcional.

```
1  //Global.asax
2  routes.MapRoute("Nova Rota", "Biblioteca/{editora}",
3                  new { controller = "Livros", action = "Index", editora = UrlParameter.↩
                       Optional });
4
5  routes.MapRoute(
6                  "Default", // Route name
7                  "{controller}/{action}/{id}", // URL with parameters
8                  new { controller = "Editoras", action = "Index", id = UrlParameter.↩
                       Optional } // Parameter defaults
9              );
```

4. Altere o método de listagem de livros para receber o parâmetro *editora* da URL.

```
1  //LivrosController.cs
2  public ViewResult Index(string editora)
3  {
4      return View(livroRepository.BuscaTodas());
5  }
```

5. (opcional) Defina a lógica para listar os livros de acordo com o parâmetro *editora* da URL.

```
1 //EditorasController.cs
2 public ViewResult Index(string editora)
3 {
4     List<Livro> livros = (editora != null) ? livroRepository.BuscaPorEditora(editora) :↩
          livroRepository.BuscaTodas();
5     return View(livros);
6 }
```

```
1 //LivroRepository.cs
2 public List<Livro> BuscaPorEditora(string nomeEditora)
3 {
4     var consulta = from e in context.Livros
5         where e.Editora.Nome.StartsWith(nomeEditora)
6         select e;
7     return consulta.ToList();
8 }
```

Capítulo 9

Validação

Os usuários podem cometer erros ao preencher um formulário. Por exemplo, esquecer de preencher um campo que é obrigatório. Os parâmetros enviados pelos usuários devem ser validados pela aplicação com o intuito de não permitir o armazenamento de informações erradas.

9.1 Controller

O primeiro passo para implementar a validação dos parâmetros enviados pelos usuários é definir a lógica de validação.

```
1 if (editora.Nome == null || editora.Nome.Trim().Length == 0)
2 {
3     // Erro de Validação
4 }
```

O segundo passo é definir mensagens informativas para enviar aos usuários. O ASP.NET possui um objeto especializado no armazenamento de mensagens de erros de validação (**ModelState**).

```
1 if (editora.Nome == null || editora.Nome.Trim().Length == 0)
2 {
3     ModelState.AddModelError("Nome", "O preenchimento do campo Nome é obrigatório");
4 }
```

As mensagens são armazenadas no *ModelState* através do método **AddModelError**. Esse método permite que as mensagens sejam agrupadas logicamente pois ele possui dois parâmetros: o primeiro é o grupo da mensagem e o segundo e a mensagem propriamente.

O código de validação pode ser colocado nos controladores, mais especificamente nas ações. Se algum erro for encontrado, o fluxo de execução pode ser redirecionado para uma página *.aspx* que mostre as mensagens informativas aos usuários. Normalmente, essa página é a mesma do formulário que foi preenchido incorretamente. O objeto *ModelState* possui uma propriedade que indica se erros foram adicionados ou não. Essa propriedade se chama **IsValid**.

```
1  [HttpPost]
2  public ActionResult Create(Editora editora)
3  {
4      if (editora.Nome == null || editora.Nome.Trim().Length == 0)
5      {
6          // Erro de Validação
7          ModelState.AddModelError("Nome", "O preenchimento do campo Nome é obrigatório."↩
               );
8      }
9      if (ModelState.IsValid)
10     {
11         editoraRepository.Adiciona(editora);
12         TempData["mensagem"] = "Editora criada com sucesso!";
13         return RedirectToAction("Index");
14     }
15     else
16     {
17         return View();
18     }
19 }
```

O ASP.NET também pode adicionar mensagens no *ModelState* antes do controlador ser chamado. Normalmente, essas mensagens estão relacionadas a erros de conversão. Por exemplo, um campo que espera um número é preenchido com letras.

9.2 View

As mensagens dos erros de validação podem ser acrescentadas na página web através do método **ValidationSummary** da propriedade **Html**. É importante salientar que esse método adiciona todas as mensagens de erro.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Cadastro de Editora";
}

<h2>Create</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary()
    <fieldset>
        <legend>Cadastro de Editora</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

Podemos utilizar o método **Html.ValidationMessage** para mostrar somente as mensagens de erro de um determinado grupo. Para não mostrar erros das propriedades devemos acrescentar **true** ao nosso método *ValidationSummary*.

```
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Cadastro de Editora";
}

<h2>Create</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Cadastro de Editora</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
            @Html.ValidationMessageFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
            @Html.ValidationMessageFor(model => model.Email)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

9.3 Exercícios

1. Insira a validação dos campos nas editoras e livros da nossa aplicação. A editora deve ter obrigatoriamente nome e email, e o livro deve ter o nome, preço e editora relacionada. No caso do livro, o preço também não pode ser menor que zero. Você deve informar ao usuário o erro ocorrido através do método **Html.ValidationMessage** ou **Html.ValidationMessageFor**.

```
1  // EditorasController.cs
2  //
3  // POST: /Editoras/Create
4
5  [HttpPost]
6  public ActionResult Create(Editora editora)
7  {
8      if (editora.Nome == null || editora.Nome.Trim().Length == 0)
9      {
10         // Erro de Validação
11         ModelState.AddModelError("Nome", "O preenchimento do campo Nome é obrigatório.");
12     }
13     if (editora.Email == null || editora.Email.Trim().Length == 0)
14     {
15         ModelState.AddModelError("Email", "O preenchimento do campo Email é obrigatório.");
16     }
17     if (ModelState.IsValid)
18     {
19         editoraRepository.Adiciona(editora);
20         return RedirectToAction("Index");
21     }
22     else
23     {
24         return View();
25     }
26 }
```

```
<!-- ~/Views/Editoras/Create.cshtml -->
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Cadastro de Editora";
}

<h2>Create</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Cadastro de Editora</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
            @Html.ValidationMessageFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
            @Html.ValidationMessageFor(model => model.Email)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

```
1  // LivrosController.cs
2  //
3  // POST: /Livros/Create
4  [HttpPost]
5  public ActionResult Create(Livro livro)
6  {
7      if (livro.Titulo == null || livro.Titulo.Trim().Length == 0)
8      {
9          ModelState.AddModelError("Titulo", "O preenchimento do campo Título é ↩
               obrigatório.");
10     }
11     if (livro.Preco <= 0)
12     {
13         ModelState.AddModelError("Preco", "Preencha o campo Preço corretamente.");
14     }
15
16     if (ModelState.IsValid)
17     {
18         livroRepository.Adiciona(livro);
19         return RedirectToAction("Index");
20     }
21     else
22     {
23         return View();
24     }
25
26 }
```

```
<!-- ~/Views/Livros/Create.cshtml -->
@model LivrariaVirtual.Models.Livro

@{
    ViewBag.Title = "Cadastro de Livros";
}

<h2>Cadastro de Livros</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Livro</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Titulo)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Titulo)
            @Html.ValidationMessageFor(model => model.Titulo)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Preco)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Preco)
            @Html.ValidationMessageFor(model => model.Preco)
        </div>

        <div class="editor-field">
            @Html.DropDownListFor(model => model.EditoraId, new SelectList(ViewBag.↩
                Editoras,"Id","Nome"))
            @Html.ValidationMessageFor(model => model.EditoraId)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Listagem de Livros", "Index")
</div>
```

9.4 Anotações

As lógicas de validação mais utilizadas também podem ser implementadas através de anotações adicionadas nas classes de modelo. Dessa forma, essas lógicas não estariam mais nos controladores, o que conceitualmente é o ideal pois nos controladores só deveria existir lógica para controlar o fluxo da execução.

Para utilizar essas anotações, é necessário adicionar uma dll na aplicação. O nome da dll é: **System.ComponentModel.DataAnnotations.dll**.

9.4.1 Required

Uma das validações mais comuns é a de campo obrigatório. Ela pode ser realizada através da anotação **Required**.

```
1  public class Editora
2  {
3      [Required]
4      public string Nome
5      {
6          get;
7          set;
8      }
9
10     // restante do código
11 }
```

Com essa anotação a lógica de validação pode ser retirada do controlador.

```
1  [HttpPost]
2  public ActionResult Create(Editora editora)
3  {
4      if (ModelState.IsValid)
5      {
6          editoraRepository.Adiciona(editora);
7          return RedirectToAction("Index");
8      }
9      else
10     {
11         return View();
12     }
13 }
```

9.4.2 Alterando a mensagem

As anotações de validações possuem mensagens padrões que podem ser alteradas através da propriedade **ErrorMessage**.

```
1  [Required(ErrorMessage="O campo Nome é obrigatório")]
2  public string Nome
3  {
4      get;
5      set;
6  }
```

9.4.3 Outros validadores

Há outras anotações para validação:

- Range
- RequlerExpression
- StringLength

9.5 Validação no lado do Cliente

Nas versões anteriores do MVC para habilitar a validação no lado do cliente era necessário chamar explicitamente o método *Html.EnableClientValidation* na View. No ASP .NET MVC 3 a validação no cliente está habilitada por padrão.

Para funcionar corretamente a validação, você deve acrescentar as referências corretas das bibliotecas javascript *jQuery* e *jQuery Validation* na View.

```
1 <script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"><↩
      /script>
2 <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/↩
      javascript"></script>
```

9.6 Exercícios

2. Altere as validações feitas anteriormente, para utilizar **DataAnnotations**. Lembre-se de alterar todas as mensagens de erro para a língua portuguesa. Acrescente também a validação no cliente.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.ComponentModel.DataAnnotations;
6
7  namespace LivrariaVirtual.Models
8  {
9      public class Editora
10     {
11         public int Id { get; set; }
12         [Required(ErrorMessage="O campo Nome é obrigatório.")]
13         public string Nome { get; set; }
14         [Required(ErrorMessage="O campo Email é obrigatório")]
15         public string Email { get; set; }
16         public virtual ICollection<Livro> Livros { get; set; }
17
18     }
19 }
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.DataAnnotations;

namespace LivrariaVirtual.Models
{
    public class Livro
    {
        public int Id { get; set; }
        [Required(ErrorMessage="O campo Título é obrigatório.")]
        public string Titulo { get; set; }
        [Required(ErrorMessage = "O campo Preço é obrigatório.")]
        public decimal Preco { get; set; }
        [Required(ErrorMessage = "O campo EditoraId é obrigatório.")]
        public int EditoraId { get; set; }
        public virtual Editora Editora { get; set; }
    }
}
```

```
//EditorasController.cs
//
// POST: /Editoras/Create
[HttpPost]
public ActionResult Create(Editora editora)
{
    if (ModelState.IsValid)
    {
        editoraRepository.Adiciona(editora);
        return RedirectToAction("Index");
    }
    else
    {
        return View();
    }
}
```

```
//LivrosController.cs
//
// POST: /Livros/Create

[HttpPost]
public ActionResult Create(Livro livro)
{
    if (ModelState.IsValid)
    {
        livroRepository.Adiciona(livro);
        return RedirectToAction("Index");
    }
    else
    {
        ViewBag.Editoras = livroRepository.BuscaTodas();
        return View();
    }
}
```

```
<!-- ~/Views/Editoras/Create.cshtml -->
@model LivrariaVirtual.Models.Editora

@{
    ViewBag.Title = "Cadastro de Editora";
}

<h2>Create</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Cadastro de Editora</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
            @Html.ValidationMessageFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Email)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Email)
            @Html.ValidationMessageFor(model => model.Email)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Listagem de Editoras", "Index")
</div>
```

```
<!-- ~/Views/Livros/Create.cshtml -->
@model LivrariaVirtual.Models.Livro

@{
    ViewBag.Title = "Cadastro de Livros";
}

<h2>Create</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Livro</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Titulo)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Titulo)
            @Html.ValidationMessageFor(model => model.Titulo)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Preco)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Preco)
            @Html.ValidationMessageFor(model => model.Preco)
        </div>

        <div class="editor-field">
            @Html.DropDownListFor(model => model.EditoraId, new SelectList(ViewBag.Editoras,"Id","Nome"))
            @Html.ValidationMessageFor(model => model.EditoraId)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Listagem de Livros", "Index")
</div>
```

Capítulo 10

Sessão

Quando um cliente for fazer compras na nossa loja virtual ele pode ter, por exemplo, um carrinho de compras, que é uma informação gerada durante a navegação. A medida que ele visita as páginas, ele pode ir adicionando ou removendo itens do seu carrinho virtual. Porém isto é um problema, pois o protocolo HTTP não armazena estado (*stateless*) das páginas visitadas anteriormente. Desse modo não podemos armazenar a informação entre uma página e outra.

10.1 Sessão

Nossa tarefa é encontrar um modo de armazenar estes dados no servidor e deixar a informação disponível para diferentes páginas da aplicação. Para solucionar este problema, é utilizado o conceito de **sessão**. Uma sessão é uma maneira de armazenar informações geradas durante a navegação no lado do servidor. Como o sistema pode ter vários usuários navegando simultaneamente, também devemos encontrar uma maneira de separar este conjunto de informações por usuário, para que não haja nehum tipo de conflito. Para isto precisamos identificar unicamente cada usuário da nossa aplicação de maneira que cada um tenha sua própria sessão.

10.1.1 Identificando o Usuário

Uma primeira abordagem seria distinguir cada usuário utilizando o endereço de IP. Porém, caso existam usuários em uma rede local, eles teríam o mesmo IP e não seria possível identificá-los individualmente. Podemos considerar ainda o caso de um usuário com o IP dinâmico, caso ele reconecte durante a navegação, toda a informação do seu histórico recente será perdida.

A solução seria deixar a cargo do servidor a geração de um identificador único e enviá-lo para cada usuário. Desta maneira, a cada requisição, o cliente envia de volta este ID de forma que a aplicação possa reconhecê-lo. O cliente pode reenviar o seu ID de diferentes formas. As mais utilizadas são:

Reescrita de URL Nesta maneira, o ID é embutido nas próprias URL's da aplicação. Sendo assim o ID pode ser reconhecido pela aplicação em todas as requisições. Uma desvantagem é que todas as páginas devem ser geradas dinamicamente para conter o ID em todos os links e actions. Outro problema é que este ID fica aparente na barra de navegação do navegador facilitando o acesso de pessoas mal intencionadas que poderiam, por sua vez,

obter informações confidenciais. Uma vez que a URL não contém mais o identificador, a aplicação considera que o usuário não tem uma sessão associada.

Cookies Cookie é um arquivo criado pelo servidor no navegador do cliente. Uma de suas funções é persistir o ID da sessão. A cada requisição a aplicação consulta o ID da sessão no cookie para recuperar as informações do usuário. Esta é a maneira mais utilizada. A sessão pode ser encerrada com a retirada do *cookie* no navegador. Podemos fazer explicitamente através de uma rotina de *logout* no servidor, ou podemos deixá-la expirar por tempo de inatividade, ou seja, caso o usuário fique um determinado tempo sem fazer novas requisições, a sessão é encerrada.

10.2 Utilizando Session no ASP.NET

No ASP.NET, a sessão é um dicionário. Para armazenar informações, você deve adicionar uma chave e um valor no objeto **Session**. Imagine um objeto do tipo **Cliente** que agrupa as informações sobre um determinado cliente. O código a seguir é um exemplo de como podemos guardar as informações relacionadas ao cliente no momento do login.

```
1  public class LoginController : Controller
2  {
3      ...
4      public ActionResult Login(Cliente cliente)
5      {
6          ...
7          // Armazenando informação na sessão
8          Session["cliente"] = cliente;
9      }
10 }
```

Você pode adicionar qualquer tipo de valor na sessão. De forma análoga, para resgatar a informação armazenada, basta acessar a chave correspondente no objeto **Session**, como no exemplo a seguir:

```
1 // Recuperando informação da sessão
2 Cliente cliente = (Cliente)Session["cliente"];
3 string saudacao = "Bem vindo " + cliente.Nome;
```

Caso um usuário deslogue do sistema é preciso eliminar a informação acumulada em sua sessão. Para isto podemos simplesmente remover todas as chaves do dicionário como no exemplo a seguir.

```
1 // Eliminando todas as informaçòes do sistema
2 Session.RemoveAll();
```

Contudo, fazendo isto não estaríamos terminando com a sessão, o que é desejável ao fazer um logout. Então, para terminar com a sessão, você deve utilizar o comando **Session.Abandon()**.

```
1 public class LoginController : Controller
2 {
3     ...
4         public ActionResult Logout()
5         {
6             ...
7             Session.Abandon();
8         }
9 }
```

10.3 Session Mode

O ASP.NET disponibiliza quatro modos de Sessão (*Session Modes*):

- InProc
- StateServer
- SQLServer
- Custom

A diferença de cada modo é a maneira com que a sessão é armazenada. Cada um tem um *State Provider* diferente.

No modo **InProc** todas as informações da sessão são armazenadas na memória do servidor. Esse é o modo mais simples e utilizado, e vem configurado como o padrão. Uma das desvantagens desse modo, é a possível sobrecarga de memória, se forem armazenadas muitas informações na sessão. Por isso não é indicado para sistemas muito grandes, com muitos usuários navegando simultaneamente. Outro problema é que o servidor não pode ser desligado, pois a informação armazenada na memória será perdida.

No modo **StateServer** as informações são serializadas e enviadas para um servidor independente do servidor da aplicação. Isto possibilita que o servidor de aplicação seja reiniciado sem que as sessões seja perdidas. Uma desvantagem é o custo de tempo de serialização e desserialização.

No modo **SQLServer** as informações são serializadas porém armazenadas em um banco de dados. Este modo também permite que o servidor de aplicação seja reiniciado, além disso possibilita um maior controle de segurança dos dados da sessão. Uma desvantagem importante é que o processo é naturalmente lento.

No modo **Custom** todo o processo de identificação de usuário e armazenamento de sessões pode ser customizado.

Para selecionar algum dos modos disponíveis basta adicionar a tag **<sessionState>** dentro da tag **<system.web>**, no arquivo Web.Config.

```
1 <sessionState mode="InProc"/>
```

Podemos personalizar nossa sessão modificando alguns atributos da tag **<sessionState>**. Por exemplo, podemos determinar o tempo de expiração do cookie em minutos através do atributo **timeout**.

```
1 <sessionState mode="InProc" timeout="30"/>
```

Outra possibilidade é desabilitar o uso de cookie com o atributo **cookieless**. Neste caso será utilizada a reescrita de URL.

```
1 <sessionState mode="InProc" cookieless="true"/>
```

Se nenhum modo for selecionado o modo InProc será utilizado por padrão.

10.4 Exercícios

1. Para introduzirmos o conceito de sessão na nossa livraria virtual precisamos primeiro adicionar uma entidade Cliente na nossa aplicação. Seguindo os moldes de Editora e Livros crie a classe Cliente, a classe ClienteRepository, um controlador ClientesController (para listar, buscar, atualizar, remover e adicionar) e todas as páginas cshtml relacionadas.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.ComponentModel.DataAnnotations;
6  using System.Web.Mvc;
7
8
9  namespace LivrariaVirtual.Models
10 {
11     public class Cliente
12     {
13         public int Id { get; set; }
14         [Required(ErrorMessage="O campo Nome é obrigatório")]
15         public string Nome { get; set; }
16         [Required(ErrorMessage="O campo Usuário é obrigatório")]
17         public string Usuario { get; set; }
18         [DataType(DataType.Password)]
19         public string Senha { get; set; }
20         [NotMapped]
21         [DataType(DataType.Password)]
22         [Compare("Senha",ErrorMessage="A confirmação da senha está incorreta.")]
23         public string ComparaSenha { get; set; }
24     }
25 }
```

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

namespace LivrariaVirtual.Models
{
    public class ClienteRepository
    {
        private LivrariaContext context = new LivrariaContext();
        public void Adiciona(Cliente c)
        {
            context.Clientes.Add(c);
            context.SaveChanges();
        }

        public void Atualiza(Cliente c)
        {
            context.Entry(c).State = EntityState.Modified;
            context.SaveChanges();
        }

        public void Remove(Cliente c)
        {
            context.Entry(c).State = EntityState.Deleted;
            context.SaveChanges();
        }

        public void Remove(int id)
        {
            Cliente c = context.Clientes.Find(id);
            context.Clientes.Remove(c);
            context.SaveChanges();
        }

        public Cliente Busca(int id)
        {
            return context.Clientes.Find(id);
        }

        public List<Cliente> BuscaTodas()
        {
            return context.Clientes.ToList();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using LivrariaVirtual.Models;

namespace LivrariaVirtual.Controllers
{
    public class ClientesController : Controller
    {
        private ClienteRepository clienteRepository = new ClienteRepository();

        //
        // GET: /Clientes/

        public ViewResult Index()
        {
```

```
21              return View(clienteRepository.BuscaTodas());
22          }
23
24          //
25          // GET: /Clientes/Details/5
26
27          public ViewResult Details(int id)
28          {
29              Cliente cliente = clienteRepository.Busca(id);
30              return View(cliente);
31          }
32
33          //
34          // GET: /Clientes/Create
35
36          public ActionResult Create()
37          {
38              return View();
39          }
40
41          //
42          // POST: /Clientes/Create
43
44          [HttpPost]
45          public ActionResult Create(Cliente cliente)
46          {
47              if (ModelState.IsValid)
48              {
49                  clienteRepository.Adiciona(cliente);
50                  return RedirectToAction("Index");
51              }
52
53              return View(cliente);
54          }
55
56          //
57          // GET: /Clientes/Edit/5
58
59          public ActionResult Edit(int id)
60          {
61              Cliente cliente = clienteRepository.Busca(id);
62              return View(cliente);
63          }
64
65          //
66          // POST: /Clientes/Edit/5
67
68          [HttpPost]
69          public ActionResult Edit(Cliente cliente)
70          {
71              if (ModelState.IsValid)
72              {
73                  clienteRepository.Atualiza(cliente);
74                  return RedirectToAction("Index");
75              }
76              return View(cliente);
77          }
78
79          //
80          // GET: /Clientes/Delete/5
81
82          public ActionResult Delete(int id)
83          {
84              Cliente cliente = clienteRepository.Busca(id);
85              return View(cliente);
86          }
87
88          //
89          // POST: /Clientes/Delete/5
90
```

```
91          [HttpPost, ActionName("Delete")]
92          public ActionResult DeleteConfirmed(int id)
93          {
94              clienteRepository.Remove(id);
95              return RedirectToAction("Index");
96          }
97      }
98  }
```

```
<!-- ~/Views/Clientes/Create.cshtml -->
@model LivrariaVirtual.Models.Cliente

@{
    ViewBag.Title = "Cadastro de Clientes";
}

<h2>Cadastro de Clientes</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Cliente</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
            @Html.ValidationMessageFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Usuario)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Usuario)
            @Html.ValidationMessageFor(model => model.Usuario)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Senha)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Senha)
            @Html.ValidationMessageFor(model => model.Senha)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.ComparaSenha)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ComparaSenha)
            @Html.ValidationMessageFor(model => model.ComparaSenha)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Listagem de Clientes", "Index")
</div>
```

2. Implemente um novo controlador chamado LoginController que será responsável pelo login e logout do usuário. Para isto o controlador precisará de uma action para login e outra para logout. Crie uma página de login chamada **index.cshtml**. Esta página deve disponibilizar um formulário com usuário e senha. Para efetuar um login a action login

deve chamar um novo método do ClienteRepository que recebe um par usuario/senha e devolve um bool inicando a existência dos dados de entrada. Se a entrada for válida, recupere as informações do usuário em um objeto tipo Cliente, armazene-o em uma chave 'cliente' da sessão e redirecione a página para a home. Se a entrada não for válida então imprima uma mensagem de falha na página de login. Na action logout faça com que a sessão seja terminada. Por último, dentro do body do nosso layout principal adicione o código para mostrar o nome do usuário logado.

```
1  //~/Models/ClienteRepository.cs
2  public Cliente Busca(string usuario, string senha)
3  {
4      var consulta = from e in context.Clientes
5          where e.Usuario.Equals(usuario) && e.Senha.Equals(senha)
6          select e;
7      string query = consulta.ToString();
8      return consulta.First();
9  }
10
11 public bool Autentica(string usuario, string senha)
12 {
13     var consulta = from e in context.Clientes
14         where e.Usuario.Equals(usuario) && e.Senha.Equals(senha)
15         select e;
16     return consulta.ToList().Count >= 1;
17 }
```

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

namespace LivrariaVirtual.Controllers
{
    public class LoginController : Controller
    {
        private ClienteRepository clienteRepository = new ClienteRepository();
        //
        // GET: /Login/

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

        [HttpPost]
        public ActionResult Index(string usuario, string senha)
        {
            if (clienteRepository.Autentica(usuario,senha))
            {
                Session["cliente"] = clienteRepository.Busca(usuario, senha);
                return RedirectToAction("Index", "Editoras");
            }
            else
            {
                TempData["mensagem"] = "Usuário e senha incorretos";
                return View();
            }

        }

        public ActionResult Logout()
        {
            Session.Abandon();
            return Redirect("/");
        }

    }
}
```

```
<!-- ~/Views/Login/Index.cshtml -->
@{
    ViewBag.Title = "Tela de Login";
}

<h2>Tela de Login</h2>
@if (TempData["mensagem"] != null)
{
    <h3>@TempData["mensagem"]</h3>
}
<h3></h3>
@using (Html.BeginForm())
{
    <p>Usuário: @Html.TextBox("Usuario")</p>
    <p>Senha: @Html.Password("Senha")</p>
    <input type="submit" value="Logar" />
}
```

```
1  <!-- ~/Views/Shared/_Layout.cshtml -->
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <title>@ViewBag.Title</title>
6      <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
7      <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript">↩
           </script>
8  </head>
9  @{LivrariaVirtual.Models.Cliente c = (LivrariaVirtual.Models.Cliente)Session["cliente"↩
       ];}
10 @if (Session["cliente"] != null)
11 {
12     <span>@c.Nome</span>
13     <span>@Html.ActionLink("Logout","Logout","Login")</span>
14 }
15 else {
16     <span>@Html.ActionLink("Login", "Index", "Login")</span>
17 }
18 <body>
19     @RenderBody()
20 </body>
21 </html>
```

Este código faz com que apareça o nome do cliente logado e um link para logout somente se o cliente estiver logado. Note que podemos navegar no sistema sem que estejamos logados. Este problema será tratado no capítulo de Filtros.

Capítulo 11

Filtros

Muitas vezes em um sistema queremos restringir o acesso à determinadas áreas, seja por segurança ou por organização.Por exemplo, na nossa aplicação poderíamos definir que para poder adicionar, alterar e remover tanto categorias quanto produtos, o usuário deve estar logado no sistema. Caso contrário, o usuário apenas pode listar as categorias e produtos.

```
1  public ActionResult FormularioCadastro()
2  {
3      if (Session["cliente"] != null)
4      {
5          return base.View();
6      }
7      else
8      {
9          return base.RedirectToAction("Index", "Login");
10     }
11 }
```

No exemplo acima, caso um usuário tente adicionar uma categoria através do formulário de cadastro, o método vai verificar se o usuário já está logado, através do uso da sessão visto no capítulo anterior. Senão estiver logado, ele será redirecionado para a página de Login . Apesar de funcionar, este código apresenta uma inconveniência. Temos que adicionar essa lógica a todas as Actions que queremos que tenha o mesmo comportamento, ou seja, que apenas permitam o acesso de usuários logados.

Em outros casos, podemos querer que algumas Actions executem alguma tarefa em comum. Por exemplo, na nossa loja virtual, poderíamos adicionar uma mensagem em um arquivo de Log sempre que uma Action fosse realizada. Desse modo, poderíamos guardar um histórico sobre o que a aplicação mais realizou, qual foi a página mais visitada, etc. Mas novamente, teríamos que adicionar a mesma tarefa em todas as Actions da nossa aplicação.

Nesses casos, em que várias Actions possuem um mesmo comportamento em comum, podemos utilizar o conceito de Filtros. Um filtro é semelhante a um método que é executado antes ou depois que uma Action é realizada.

11.1 Filtro de Autenticação

O ASP.NET já possui alguns filtros prontos para serem utilizados, como o filtro de autenticação. Podemos utilizar ele para o nosso primeiro exemplo, onde exigimos que o usuário

esteja logado (autenticado) para acessar determinadas áreas da aplicação. Para isso precisamos adicionar o seguinte código no nosso método de login:

```
1 FormsAuthentication.SetAuthCookie(cliente.Usuario, false);
```

Isto adiciona um novo cookie utilizado para a autenticação do usuário. Este novo cookie é independente do cookie utilizado para armazenar informações da sessão. O primeiro parâmetro é referente ao nome do usuário (ou algo que o identifique). O segundo parâmetro é um booleano relativo ao tipo do cookie, se é permanente ou não. Caso seja true, ele sempre irá considerar que o usuário está autenticado após a primeira autenticação.

Para eliminar o cookie de autenticação, devemos realizar o seguinte código no logout:

```
1 FormsAuthentication.SignOut();
```

Para adicionar o filtro, devemos incluir a anotação **Authorize** nas Actions em que desejamos a autenticação:

```
1 [Authorize]
2 public ActionResult FormularioCadastro()
3 {
4     return base.View();
5 }
```

Se queremos aplicar o mesmo filtro a todas as Actions de um controlador, podemos adicionar a notação **Authorize** na classe:

```
1 [Authorize]
2 public class CategoriaController : Controller
3 {
4     ...
5 }
```

Desse modo, todas as Actions presentes no controlador da categoria exigem que o usuário esteja autenticado.

Quando o filtro de autenticação "barra" um usuário de acessar uma página, podemos redirecioná-lo para a página de login. Devemos incluir o seguinte código dentro da tag **<system.web>**:

```
1 <authentication mode="Forms">
2     <forms loginUrl="~/Login" timeout="2880"/>
3 </authentication>
```

Para checar se a sessão está autenticada, podemos utilizar o atributo **IsAuthenticated**, como a seguir:

```
1 if (User.Identity.IsAuthenticated)
2 {
3     ...
4 }
```

Podemos pegar a informação de quem está autenticado através do seguinte comando:

```
1 string nome = User.Identity.Name;
```

Isto irá pegar o nome que passamos como parâmetro para o método **SetAuthCookie**.

11.2 Exercícios

1. Altere a aplicação do capítulo anterior para incluir autenticação nas Actions de adicionar, alterar e remover de editoras e livros. Na Action de Login do LoginController, adicione o cookie de autenticação como visto na seção anterior, passando o nome de usuario como parâmetro. No layout principal, altere a seção que mostra o nome do usuário, para utilizar a informação do cookie de autenticação, e não mais da sessão. Caso o usuário não esteja autenticado, e tente acessar uma das Actions acima, redirecione através do **Web.Config** para a Action de Login.

11.3 *Action Filters*

O filtro de autenticação não é o único filtro que existe no ASP.NET. Outro tipo muito usado são os chamados *Action Filters*. Geralmente são usados quando queremos executar uma ação específica para mais de uma Action. Por exemplo, quando queremos gravar as ações que estão sendo realizadas em um arquivo de log.

Para criar um Action Filter você deve utilizar o sufixo **Attribute** no nome da classe, e herdar a classe **ActionFilterAttribute**.

```
1 public class LogAttribute : ActionFilterAttribute
2 {
3     ...
4 }
```

O nome utilizado na classe é o mesmo utilizado nas anotações das Actions, excluindo o sufixo attribute. Por exemplo, para aplicar o filtro LogAttribute no método que lista as categorias:

```
1 [Log]
2 public ActionResult Lista()
3 {
4     ...
5 }
```

Para fazer o filtro funcionar, deve ser implementado um ou mais dos seguintes métodos:

- **OnActionExecuting(ActionExecutedContext filterContext)**;
- **OnActionExecuted(ActionExecutingContext filterContext)**;
- **OnResultExecuting(ResultExecutedContext filterContext)**;

- **OnResultExecuted(ResultExecutingContext filterContext)**;

Eles são executados na mesma ordem em que aparecem acima, sendo que todos são executados antes da renderização da página. Logo, o nosso exemplo com o filtro de log poderia ficar assim:

```
1  public class LogAttribute : ActionFilterAttribute
2  {
3      ...
4      public override void OnActionExecuted(ActionExecutedContext filterContext)
5      {
6          // escreve informação de log
7      }
8      ...
9  }
```

Também é possível passar "parâmetros" para o filtro da seguinte maneira:

```
1  [Log(Message="Executando lista de categorias"]
2  public ActionResult Lista()
3  {
4      ...
5  }
```

Sendo que a classe **LogAttribute** precisa ter um atributo ou propriedade com o mesmo nome do parâmetro passado.

```
1  public class LogAttribute : ActionFilterAttribute
2  {
3      public string Message { get; set; }
4
5      public override void OnActionExecuted(ActionExecutedContext filterContext)
6      {
7          // escreve a mensagem no arquivo de log
8      }
9      ...
10 }
```

Você pode passar vários parâmetros na anotação, separando-os por vírgulas.

11.4 Exercícios

2. Crie um filtro chamado LogAttribute, que grava mensagens em um arquivo de Log, chamado "log.txt" sempre que uma Action é chamada. A informação no log deve incluir a data, horário e pequena descrição da Action realizada. Aplique este filtro a todas as Actions do sistema. (Dica: utilize a classe **FileInfo** e seu método **AppendText** para criar o arquivo de log e a classe **StreamWriter** para escrever, e passe um parâmetro para o filtro para identificar a ação que está sendo realizada).

Capítulo 12

Projeto

Nos capítulos anteriores, vimos os recursos do ASP .NET MVC e do Entity Framework. Agora, vamos solidificar os conhecimentos obtidos e, além disso, mostraremos alguns padrões e conceitos relacionados ao desenvolvimento de aplicações web.

Como exemplo de aplicação desenvolveremos uma aplicação de cadastro de jogadores e seleções de futebol.

12.1 Modelo

Por onde começar o desenvolvimento de uma aplicação? Essa é uma questão recorrente. Um ótimo ponto de partida é desenvolver as entidades principais da aplicação. No nosso caso, vamos nos restringir às entidades **Selecao** e **Jogador**. Devemos estabelecer um relacionamento entre essas entidades já que um jogador atua em uma seleção.

12.2 Exercícios

1. Crie um projeto do tipo **ASP .NET MVC 3 Web Application** chamado **K19-CopaDoMundo** seguindo os passos vistos no exercício do capítulo 4.

2. Adicione na pasta **Models** as seguintes classes.

```
1 namespace K19_CopaDoMundo.Models
2 {
3     public class Selecao
4     {
5         public string Pais { get; set; }
6         public string Tecnico { get; set; }
7     }
8 }
```

```
1  namespace K19_CopaDoMundo.Models
2  {
3      public class Jogador
4      {
5          public string Nome { get; set; }
6          public string Posicao { get; set; }
7          public DateTime Nascimento { get; set; }
8          public double Altura { get; set; }
9          public Selecao Selecao { get; set; }
10
11     }
12 }
```

12.3 Persistência - Mapeamento

Depois de definir algumas entidades podemos começar o processo de implementação da persistência da nossa aplicação. Vamos aplicar os recursos do Entity Framework - Code First que aprendemos nos primeiros capítulos. Inicialmente, vamos definir o mapeamento das nossas entidades através de uma classe derivada de *DbContext* e acrescentar as propriedades referentes a chave primária e chave estrangeira. Não se esqueça de acrescentar as dll's EntityFramework e System.ComponentModel.DataAnnotations ao projeto.

12.4 Exercícios

3. Adicione as seguintes propriedades e anotações.

```
1  using System.ComponentModel.DataAnnotations;
2
3  namespace K19_CopaDoMundo.Models
4  {
5      [Table("Selecoes")]
6      public class Selecao
7      {
8          public int Id { get; set; }
9          public string Pais { get; set; }
10         public string Tecnico { get; set; }
11         public virtual ICollection<Jogador> Jogadores { get; set; }
12     }
13 }
```

```
1  using System.ComponentModel.DataAnnotations;
2
3  namespace K19_CopaDoMundo.Models
4  {
5      [Table("Jogadores")]
6      public class Jogador
7      {
8          public int Id { get; set; }
9          public string Nome { get; set; }
10         public string Posicao { get; set; }
11         public DateTime Nascimento { get; set; }
12         public double Altura { get; set; }
13         public int SelecaoId { get; set; }
14         [InverseProperty("Jogadores")]
15         public virtual Selecao Selecao { get; set; }
16
17     }
18 }
```

```
1  using System.Data.Entity;
2
3  namespace K19_CopaDoMundo.Models
4  {
5      public class CopaDoMundoContext : DbContext
6      {
7          public DbSet<Selecao> Selecoes { get; set; }
8          public DbSet<Jogador> Jogadores { get; set; }
9      }
10 }
```

12.5 Persistência - Configuração

Precisamos definir a nossa string de conexão para que a nossa aplicação utilize a base de dados *copadomundo* como padrão.

12.6 Exercícios

4. Acrescente ao arquivo **Web.config**, que fica na raiz do projeto, a string de conexão.

```
1  <connectionStrings>
2      <add
3        name="CopaDoMundoContext" providerName="System.Data.SqlClient"
4        connectionString="Server=.\SQLEXPRESS;Database=copadomundo;
5      User Id=sa; Password=sa;Trusted_Connection=False;Persist Security Info=True;↩
           MultipleActiveResultSets=True"/>
6    </connectionStrings>
```

5. Altere a estratégia de criação do banco de dados. Para isto, acrescente ao método *Application_Start* definido no arquivo *Global.asax* o seguinte trecho de código.

```
1 //Global.asax
2 protected void Application_Start()
3 {
4     AreaRegistration.RegisterAllAreas();
5     // Alterando a estratégia de criação do banco de dados
6     Database.SetInitializer(new DropCreateDatabaseIfModelChanges<K19_CopaDoMundo.Models↩
          .CopaDoMundoContext>());
7     RegisterGlobalFilters(GlobalFilters.Filters);
8     RegisterRoutes(RouteTable.Routes);
9 }
```

12.7 Persistência - Repositórios

Vamos deixar os repositórios para acessar as entidades da nossa aplicação preparados. Os repositórios precisam de DbContexts para realizar as operações de persistência. Então, cada repositório terá um construtor para receber um DbContext como parâmetro.

12.8 Exercícios

6. Crie uma classe na pasta **Models** chamada **SelecaoRepository**.

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

namespace K19_CopaDoMundo.Models
{
    public class SelecaoRepository : IDisposable
    {
        private bool disposed = false;

        private CopaDoMundoContext context;

        public SelecaoRepository(CopaDoMundoContext context)
        {
            this.context = context;
        }

        public void Adiciona(Selecao selecao)
        {
            context.Selecoes.Add(selecao);
        }

        public List<Selecao> Selecoes
        {
            get
            {
                return context.Selecoes.ToList();
            }
        }
        public void Salva()
        {
            context.SaveChanges();
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                    context.Dispose();
                }
            }
            this.disposed = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
}
```

7. Analogamente crie um repositório de jogadores.

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

namespace K19_CopaDoMundo.Models
{
    public class JogadorRepository : IDisposable
    {
        private bool disposed = false;
        private CopaDoMundoContext context;

        public JogadorRepository(CopaDoMundoContext context)
        {
            this.context = context;
        }

        public void Adiciona(Jogador jogador)
        {
            context.Jogadores.Add(jogador);
        }

        public List<Jogador> Jogadores
        {
            get { return context.Jogadores.ToList(); }
        }

        public void Salva()
        {
            context.SaveChanges();
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                    context.Dispose();
                }
            }
            this.disposed = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
}
```

12.8.1 Unit of Work

O único propósito de criar uma classe UnitOfWork é ter certeza que quando temos múltiplos repositórios eles compartilham o mesmo *DbContext*. Para isto, devemos apenas criar um método **Salva** e uma propriedade para cada repositório.

12.9 Exercícios

8. Crie uma classe **UnitOfWork** na pasta **Models**.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5
6  namespace K19_CopaDoMundo.Models
7  {
8      public class UnitOfWork : IDisposable
9      {
10         private bool disposed = false;
11         private CopaDoMundoContext context = new CopaDoMundoContext();
12         private SelecaoRepository selecaoRepository;
13         private JogadorRepository jogadorRepository;
14
15
16         public JogadorRepository JogadorRepository
17         {
18             get
19             {
20                 if (jogadorRepository == null)
21                 {
22                     jogadorRepository = new JogadorRepository(context);
23                 }
24                 return jogadorRepository;
25             }
26         }
27
28         public SelecaoRepository SelecaoRepository
29         {
30             get
31             {
32                 if (selecaoRepository == null)
33                 {
34                     selecaoRepository = new SelecaoRepository(context);
35                 }
36                 return selecaoRepository;
37             }
38         }
39
40         public void Salva()
41         {
42             context.SaveChanges();
43         }
44
45         protected virtual void Dispose(bool disposing)
46         {
47             if (!this.disposed)
48             {
49                 if (disposing)
50                 {
51                     context.Dispose();
52                 }
53             }
54             this.disposed = true;
55         }
56
57         public void Dispose()
58         {
59             Dispose(true);
60             GC.SuppressFinalize(this);
61         }
62     }
63 }
```

12.10 Apresentação - Template

Vamos definir um template para as telas da nossa aplicação. Aplicaremos algumas regras CSS para melhorar a parte visual das telas.

12.11 Exercícios

9. Na pasta **Content**, altere o arquivo **Site.css** acrescentando algumas regras CSS.

```
.logo{
    vertical-align: middle;
}

.botao {
    background-color: #064D83;
    margin: 0 0 0 20px;
    color: white;
    text-decoration: none;
    font-size: 20px;
    line-height: 20px;
    padding: 5px;
    vertical-align: middle;
}

.botao:hover{
    background-color: #cccccc;
    color: #666666;
}

.formulario fieldset{
    float: left;
    margin: 0 0 20px 0;
    border: 1px solid #333333;
}

.formulario fieldset legend{
    color: #064D83;
    font-weight: bold;
}

.botao-formulario{
    background-color: #064D83;
    color: #ffffff;
    padding: 5px;
    vertical-align: middle;
    border: none;
}

.titulo {
    color: #064D83;
    clear: both;
}

.tabela{
    border: 1px solid #064D83;
    border-collapse: collapse;
}

.tabela tr th{
    background-color: #064D83;
    color: #ffffff;
}

.tabela tr th,
.tabela tr td{
```

```
    border: 1px solid #064D83;
    padding: 2px 5px;
}


/* Styles for validation helpers
-----------------------------------------------------------*/
.field-validation-error
{
    color: #ff0000;
}

.field-validation-valid
{
    display: none;
}

.input-validation-error
{
    border: 1px solid #ff0000;
    background-color: #ffeeee;
}

.validation-summary-errors
{
    font-weight: bold;
    color: #ff0000;
}

.validation-summary-valid
{
    display: none;
}
```

10. Copie o arquivo **k19-logo.png** da pasta **K19-Arquivos** da sua Área de Trabalho para a pasta **Content**.

11. Agora altere o arquivo **_Layout.cshtml**.

```
<!-- ~/Views/Shared/_Layout.cshtml -->
<!DOCTYPE html>
<html>
<head>
    <title>Copa do Mundo</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript">
        </script>
</head>

<body>
    <div id="header">
        <img class="logo" src="@Url.Content("~/Content/k19-logo.jpg")" alt="K19 Logo" />
        @Html.ActionLink("Selecoes", "Index", "Selecoes", null, new { @class = "botao" })
        @Html.ActionLink("Jogadores", "Index", "Jogadores", null, new { @class = "botao" })
        <hr/>
    </div>
    @RenderBody()
    <div id="footer" style="text-align: center">
        <hr />
        &copy; 2010 K19. Todos os direitos reservados.
    </div>
</body>
</html>
```

12.12 Cadastrando e Listando Seleções

Na tela de seleções, vamos adicionar um formulário para cadastrar novas seleções e uma tabela para apresentar as já cadastradas. Aplicaremos regras de validação específicas para garantir que nenhum dado incorreto seja armazenado no banco de dados.

12.13 Exercícios

12. Para cadastrar a seleção, devemos definir o controlador.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Data;
4  using System.Data.Entity;
5  using System.Linq;
6  using System.Web;
7  using System.Web.Mvc;
8  using K19_CopaDoMundo.Models;
9
10 namespace K19_CopaDoMundo.Controllers
11 {
12     public class SelecoesController : Controller
13     {
14         private UnitOfWork unitOfWork = new UnitOfWork();
15
16         public ActionResult Create()
17         {
18             return View();
19         }
20
21         protected override void Dispose(bool disposing)
22         {
23             unitOfWork.Dispose();
24             base.Dispose(disposing);
25         }
26     }
27 }
```

13. Vamos criar uma tela **Create.cshtml** para cadastrar as seleções. Adicione o arquivo a pasta **Views/Selecoes** com o seguinte conteúdo.

```
<!-- ~/Views/Selecoes/Create.cshtml -->
@model K19_CopaDoMundo.Models.Selecao

<h2>Cadastro de Seleções</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Seleção</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Pais)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Pais)
            @Html.ValidationMessageFor(model => model.Pais)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Tecnico)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Tecnico)
            @Html.ValidationMessageFor(model => model.Tecnico)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Listagem de Seleções", "Index")
</div>
```

14. O próximo passo é definir a action que irá salvar a seleção no nosso banco de dados. Devemos também acrescentar as validações a nossa entidade.

```
//SelecoesController.cs
[HttpPost]
public ActionResult Create(Selecao selecao)
{
    if (ModelState.IsValid)
    {
        unitOfWork.SelecaoRepository.Adiciona(selecao);
        unitOfWork.Salva();
        return RedirectToAction("Index");
    }
    return View(selecao);
}
```

```
//Selecao.cs
using System.ComponentModel.DataAnnotations;

namespace K19_CopaDoMundo.Models
{
    [Table("Selecoes")]
    public class Selecao
    {
        public int Id { get; set; }
        [Required(ErrorMessage="O campo Pais é obrigatório.")]
        public string Pais { get; set; }
        [Required(ErrorMessage="O campo Tecnico é obrigatório.")]
        public string Tecnico { get; set; }
        public virtual ICollection<Jogador> Jogadores { get; set; }
    }
}
```

15. Defina a action e a página para listar todas as entidades de seleção.

```
//SelecoesController.cs
public ActionResult Index()
{
    return View(unitOfWork.SelecaoRepository.Selecoes);
}
```

```
<!-- ~/Views/Selecoes/Index.cshtml -->
@model IEnumerable<K19_CopaDoMundo.Models.Selecao>
<h2>Listagem de Seleção</h2>

<p>
    @Html.ActionLink("Cadastrar Nova Seleção", "Create")
</p>
<table>
    <tr>
        <th>
            Pais
        </th>
        <th>
            Tecnico
        </th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Pais)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tecnico)
        </td>
    </tr>
}

</table>
```

16. Vamos definir a tela de listagem de Seleções como a página principal do nosso site. Altere a rota padrão no arquivo **Global.asax**.

```
//Global.asax
routes.MapRoute(
        "Default", // Route name
        "{controller}/{action}/{id}", // URL with parameters
        new { controller = "Selecoes", action = "Index", id = UrlParameter.Optional } ↩
            // Parameter defaults
    );
```

12.14 Removendo Seleções

Vamos acrescentar a funcionalidade de remover seleções.

12.15 Exercícios

17. Acrescente uma coluna na tabela de listagem de seleções.

```
<!-- ~/Views/Selecoes/Index.cshtml -->
@model IEnumerable<K19_CopaDoMundo.Models.Selecao>
<h2>Listagem de Seleção</h2>

<p>
    @Html.ActionLink("Cadastrar Nova Seleção", "Create")
</p>
<table>
    <tr>
        <th>
            Pais
        </th>
        <th>
            Tecnico
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Pais)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tecnico)
        </td>
        <td>@Html.ActionLink("Remover","Delete", new {id = item.Id})</td>
    </tr>
}

</table>
```

18. Defina um método **Busca** na classe **SelecaoRepository** que retorna uma entidade seleção a partir de um parâmetro **id**.

```
1  //SelecaoRepository.cs
2  public Selecao Busca(int id)
3  {
4      return context.Selecoes.Find(id);
5  }
```

19. Defina uma action **Delete** que irá mostrar a tela de confirmação de remoção da entidade.

```
1  //SelecoesController.cs
2  public ActionResult Delete(int id)
3  {
4      Selecao selecao = unitOfWork.SelecaoRepository.Busca(id);
5      return View(selecao);
6  }
```

20. Defina a tela de confirmação de remoção da seleção.

```
1  <!-- ~/Views/Selecoes/Delete.cshtml -->
2  @model K19_CopaDoMundo.Models.Selecao
3
4  <h2>Remoção de Seleção</h2>
5
6  <h3>Você tem certeza que deseja remover esta entidade?</h3>
7  <fieldset>
8      <legend>Seleção</legend>
9
10     <div class="display-label">Pais</div>
11     <div class="display-field">
12         @Html.DisplayFor(model => model.Pais)
13     </div>
14
15     <div class="display-label">Tecnico</div>
16     <div class="display-field">
17         @Html.DisplayFor(model => model.Tecnico)
18     </div>
19 </fieldset>
20 @using (Html.BeginForm()) {
21     <p>
22         <input type="submit" value="Delete" /> |
23         @Html.ActionLink("Listagem de Seleções", "Index")
24     </p>
25 }
```

21. Defina um método na classe **SelecaoRepository** que remove uma entidade seleção a partir de um parâmetro *id*.

```
1  //SelecaoRepository.cs
2  public void Remove(int id)
3  {
4      Selecao selecao = Busca(id);
5      context.Selecoes.Remove(selecao);
6  }
```

22. Defina a action que remove a seleção do banco de dados.

```
1  //SelecoesController.cs
2  [HttpPost]
3  [ActionName("Delete")]
4  public ActionResult DeleteConfirmed(int id)
5  {
6      unitOfWork.SelecaoRepository.Remove(id);
7      unitOfWork.Salva();
8      return RedirectToAction("Index");
9  }
```

12.16 Cadastrando, Listando e Removendo Jogadores

Na tela de jogadores, vamos adicionar um formulário para cadastrar novos jogadores e uma tabela para apresentar os já cadastrados. Aplicaremos regras de validação específicas para garantir que nenhum dado incorreto seja armazenado no banco de dados.

12.17 Exercícios

23. Para cadastrar o jogador, devemos definir o controlador.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Data;
4  using System.Data.Entity;
5  using System.Linq;
6  using System.Web;
7  using System.Web.Mvc;
8  using K19_CopaDoMundo.Models;
9
10 namespace K19_CopaDoMundo.Controllers
11 {
12     public class JogadoresController : Controller
13     {
14         private UnitOfWork unitOfWork = new UnitOfWork();
15
16         public ActionResult Create()
17         {
18             ViewBag.SelecaoId = new SelectList(unitOfWork.SelecaoRepository.Selecoes, "↩
                   Id", "Pais");
19             return View();
20         }
21
22         protected override void Dispose(bool disposing)
23         {
24             unitOfWork.Dispose();
25             base.Dispose(disposing);
26         }
27     }
28 }
```

24. Vamos criar uma tela **Create.cshtml** para cadastrar os jogadores. Adicione o arquivo a pasta **Views/Jogadores** com o seguinte conteúdo.

```
<!-- ~/Views/Jogadores/Create.cshtml -->
@model K19_CopaDoMundo.Models.Jogador

<h2>Cadastro de Jogadores</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Jogador</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nome)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nome)
            @Html.ValidationMessageFor(model => model.Nome)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Posicao)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Posicao)
            @Html.ValidationMessageFor(model => model.Posicao)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Nascimento)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Nascimento)
            @Html.ValidationMessageFor(model => model.Nascimento)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Altura)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Altura)
            @Html.ValidationMessageFor(model => model.Altura)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.SelecaoId, "Selecao")
        </div>
        <div class="editor-field">
            @Html.DropDownList("SelecaoId", String.Empty)
            @Html.ValidationMessageFor(model => model.SelecaoId)
        </div>

        <p>
            <input type="submit" value="Salvar" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Listagem de Jogadores", "Index")
</div>
```

25. O próximo passo é definir a action que irá salvar o jogador no nosso banco de dados. Devemos também acrescentar as validações a nossa entidade.

```
//JogadoresController.cs
[HttpPost]
        public ActionResult Create(Jogador jogador)
        {
            if (ModelState.IsValid)
            {
                unitOfWork.JogadorRepository.Adiciona(jogador);
                unitOfWork.Salva();
            }
            ViewBag.SelecaoId = new SelectList(unitOfWork.SelecaoRepository.Selecoes, "Id", "Pais");
            return View();
        }
```

```
using System.ComponentModel.DataAnnotations;

namespace K19_CopaDoMundo.Models
{
    [Table("Jogadores")]
    public class Jogador
    {
        public int Id { get; set; }
        [Required(ErrorMessage="O campo Nome é obrigatório.")]
        public string Nome { get; set; }
        [Required(ErrorMessage = "O campo Posicao é obrigatório.")]
        public string Posicao { get; set; }
        [Required(ErrorMessage = "O campo Nascimento é obrigatório.")]
        [DataType(DataType.Date)]
        public DateTime Nascimento { get; set; }
        [Required(ErrorMessage = "O campo Altura é obrigatório.")]
        public double Altura { get; set; }
        public int SelecaoId { get; set; }
        [InverseProperty("Jogadores")]
        public virtual Selecao Selecao { get; set; }

    }
}
```

26. Defina a action e a página para listar todas as entidades de jogador.

```
//JogadoresController.cs
public ActionResult Index()
{
    return View(unitOfWork.JogadorRepository.Jogadores);
}
```

```
<!-- ~/Views/Jogadores/Index.cshtml -->
@model IEnumerable<K19_CopaDoMundo.Models.Jogador>

<h2>Listagem de Jogadores</h2>

<p>
    @Html.ActionLink("Cadastrar Jogador", "Create")
</p>
<table>
    <tr>
        <th>
            Nome
        </th>
        <th>
            Posicao
        </th>
        <th>
            Nascimento
        </th>
        <th>
            Altura
        </th>
        <th>
            Selecao
        </th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Nome)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Posicao)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Nascimento)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Altura)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Selecao.Pais)
        </td>
    </tr>
}

</table>
```

12.18 Removendo Jogadores

Vamos acrescentar a funcionalidade de remover jogadores.

12.19 Exercícios

27. Acrescente uma coluna na tabela de listagem de jogadores.

```
<!-- ~/Views/Jogadores/Index.cshtml -->
@model IEnumerable<K19_CopaDoMundo.Models.Jogador>

<h2>Listagem de Jogadores</h2>

<p>
    @Html.ActionLink("Cadastrar Jogador", "Create")
</p>
<table>
    <tr>
        <th>
            Nome
        </th>
        <th>
            Posicao
        </th>
        <th>
            Nascimento
        </th>
        <th>
            Altura
        </th>
        <th>
            Selecao
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Nome)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Posicao)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Nascimento)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Altura)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Selecao.Pais)
        </td>
        <td>@Html.ActionLink("Remover", "Delete", new{id=item.Id})</td>
    </tr>
}

</table>
```

28. Defina um método **Busca** na classe **JogadorRepository** que retorna uma entidade jogador a partir de um parâmetro **id**.

```
//JogadorRepository.cs
public Jogador Busca(int id)
{
    return context.Jogadores.Find(id);
}
```

29. Defina uma action **Delete** que irá mostrar a tela de confirmação de remoção da entidade.

```
//JogadoresController.cs
public ActionResult Delete(int id)
{
    Jogador jogador = unitOfWork.JogadorRepository.Busca(id);
    return View(jogador);
}
```

30. Defina a tela de confirmação de remoção do jogador.

```
<!-- ~/Views/Jogadores/Delete.cshtml -->
@model K19_CopaDoMundo.Models.Jogador

<h2>Remoção do Jogador</h2>

<h3>Você tem certeza que deseja remover este Jogador?</h3>
<fieldset>
    <legend>Jogador</legend>

    <div class="display-label">Nome</div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Nome)
    </div>

    <div class="display-label">Posicao</div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Posicao)
    </div>

    <div class="display-label">Nascimento</div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Nascimento)
    </div>

    <div class="display-label">Altura</div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Altura)
    </div>

    <div class="display-label">Selecao</div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Selecao.Pais)
    </div>
</fieldset>
@using (Html.BeginForm()) {
    <p>
        <input type="submit" value="Delete" /> |
        @Html.ActionLink("Listagem de Jogadores", "Index")
    </p>
}
```

31. Defina um método na classe **JogadorRepository** que remove uma entidade jogador a partir de um parâmetro *id*.

```
//JogadorRepository.cs
public void Remove(int id)
{
    Jogador jogador = context.Jogadores.Find(id);
    context.Jogadores.Remove(jogador);
}
```

32. Defina a action que remove o jogador do banco de dados.

```
1 //JogadoresController.cs
2 [HttpPost]
3 [ActionName("Delete")]
4 public ActionResult DeleteConfirmed(int id)
5 {
6     unitOfWork.JogadorRepository.Remove(id);
7     unitOfWork.Salva();
8     return RedirectToAction("Index");
9 }
```

12.20 Membership e Autorização

Na maioria dos casos, as aplicações devem controlar o acesso dos usuários. Vamos implementar um mecanismo de autenticação na nossa aplicação utilizando filtro e Membership. As requisições feitas pelos usuários passarão pelo filtro. A função do filtro é verificar se o usuário está logado ou não. Se estiver logado o filtro autoriza o acesso. Caso contrário, o filtro redirecionará o usuário para a tela de login.

12.21 Exercícios

33. Adicione a seguinte classe a pasta **Models**:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web.Mvc;
5  using System.ComponentModel.DataAnnotations;
6
7  namespace K19_CopaDoMundo.Models
8  {
9
10     public class ChangePasswordModel
11     {
12         [Required]
13         [DataType(DataType.Password)]
14         [Display(Name = "Senha")]
15         public string OldPassword { get; set; }
16
17         [Required]
18         [StringLength(100, ErrorMessage = "O {0} deve ter no mínimo {2} caracteres.", ↩
               MinimumLength = 6)]
19         [DataType(DataType.Password)]
20         [Display(Name = "Nova senha")]
21         public string NewPassword { get; set; }
22
23         [DataType(DataType.Password)]
24         [Display(Name = "Confirmação de senha")]
25         [Compare("NewPassword", ErrorMessage = "A senha e a confirmação não conferem.")↩
               ]
26         public string ConfirmPassword { get; set; }
27     }
28
29     public class LogOnModel
30     {
31         [Required]
32         [Display(Name = "Usuário")]
33         public string UserName { get; set; }
34
35         [Required]
36         [DataType(DataType.Password)]
37         [Display(Name = "Senha")]
38         public string Password { get; set; }
39
40         [Display(Name = "Lembrar?")]
41         public bool RememberMe { get; set; }
42     }
43
44     public class RegisterModel
45     {
46         [Required]
47         [Display(Name = "Usuário")]
48         public string UserName { get; set; }
49
50         [Required]
51         [DataType(DataType.EmailAddress)]
52         [Display(Name = "Email")]
53         public string Email { get; set; }
54
55         [Required]
56         [StringLength(100, ErrorMessage = "O {0} deve ter no mínimo {2} caracteres.", ↩
               MinimumLength = 6)]
57         [DataType(DataType.Password)]
58         [Display(Name = "Senha")]
59         public string Password { get; set; }
60
61         [DataType(DataType.Password)]
62         [Display(Name = "Confirmação de senha")]
63         [Compare("Password", ErrorMessage = "A senha e a confirmação não conferem.")]
64         public string ConfirmPassword { get; set; }
65     }
66
67 }
```

34. Acrescente a seguinte classe a pasta **Controllers**.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6  using System.Web.Security;
7  using K19_CopaDoMundo.Models;
8
9  namespace K19_CopaDoMundo.Controllers
10 {
11     public class UsuarioController : Controller
12     {
13         //
14         // GET: /Usuario/LogOn
15
16         public ActionResult LogOn()
17         {
18             return View();
19         }
20
21         //
22         // POST: /Usuario/LogOn
23
24         [HttpPost]
25         public ActionResult LogOn(LogOnModel model, string returnUrl)
26         {
27             if (ModelState.IsValid)
28             {
29                 if (Membership.ValidateUser(model.UserName, model.Password))
30                 {
31                     FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe↩
                           );
32                     if (Url.IsLocalUrl(returnUrl) && returnUrl.Length > 1 && returnUrl↩
                           .StartsWith("/")
33                         && !returnUrl.StartsWith("//") && !returnUrl.StartsWith("/\\")↩
                               )
34                     {
35                         return Redirect(returnUrl);
36                     }
37                     else
38                     {
39                         return RedirectToAction("Index", "Home");
40                     }
41                 }
42                 else
43                 {
44                     ModelState.AddModelError("", "O usuário e/ou a senha está ↩
                           incorreto.");
45                 }
46             }
47
48
49             return View(model);
50         }
51
52         //
53         // GET: /Usuario/LogOff
54
55         public ActionResult LogOff()
56         {
57             FormsAuthentication.SignOut();
58
59             return Redirect("/");
60         }
61
62         //
63         // GET: /Usuario/Register
64
```

```
65          public ActionResult Register()
66          {
67              return View();
68          }
69
70          //
71          // POST: /Usuario/Register
72
73          [HttpPost]
74          public ActionResult Register(RegisterModel model)
75          {
76              if (ModelState.IsValid)
77              {
78                  // Attempt to register the user
79                  MembershipCreateStatus createStatus;
80                  Membership.CreateUser(model.UserName, model.Password, model.Email, ↩
                        null, null, true, null, out createStatus);
81
82                  if (createStatus == MembershipCreateStatus.Success)
83                  {
84                      FormsAuthentication.SetAuthCookie(model.UserName, false /* ↩
                            createPersistentCookie */);
85                      return Redirect("/");
86                  }
87                  else
88                  {
89                      ModelState.AddModelError("", ErrorCodeToString(createStatus));
90                  }
91              }
92
93
94              return View(model);
95          }
96
97          //
98          // GET: /Usuario/ChangePassword
99
100         [Authorize]
101         public ActionResult ChangePassword()
102         {
103             return View();
104         }
105
106         //
107         // POST: /Usuario/ChangePassword
108
109         [Authorize]
110         [HttpPost]
111         public ActionResult ChangePassword(ChangePasswordModel model)
112         {
113             if (ModelState.IsValid)
114             {
115
116
117                 bool changePasswordSucceeded;
118                 try
119                 {
120                     MembershipUser currentUser = Membership.GetUser(User.Identity.Name↩
                            , true /* userIsOnline */);
121                     changePasswordSucceeded = currentUser.ChangePassword(model.↩
                            OldPassword, model.NewPassword);
122                 }
123                 catch (Exception)
124                 {
125                     changePasswordSucceeded = false;
126                 }
127
128                 if (changePasswordSucceeded)
129                 {
130                     return RedirectToAction("ChangePasswordSuccess");
```

```
131                 }
132                 else
133                 {
134                     ModelState.AddModelError("", "A senha atual ou a confirmação está ↩
                           incorreta.");
135                 }
136             }
137 
138 
139             return View(model);
140         }
141 
142         //
143         // GET: /Usuario/ChangePasswordSuccess
144 
145         public ActionResult ChangePasswordSuccess()
146         {
147             return View();
148         }
149 
150         #region Status Codes
151         private static string ErrorCodeToString(MembershipCreateStatus createStatus)
152         {
153             // See http://go.microsoft.com/fwlink/?LinkID=177550 for
154             // a full list of status codes.
155             switch (createStatus)
156             {
157                 case MembershipCreateStatus.DuplicateUserName:
158                     return "Este nome de usuário já existe. Defina outro usuário.";
159 
160                 case MembershipCreateStatus.DuplicateEmail:
161                     return "Este email já foi cadastrado. Defina outro email.";
162 
163                 case MembershipCreateStatus.InvalidPassword:
164                     return "Senha incorreta.";
165 
166                 case MembershipCreateStatus.InvalidEmail:
167                     return "Email inválido.";
168 
169                 case MembershipCreateStatus.InvalidAnswer:
170                     return "Resposta inválida para recuperar a senha.";
171 
172                 case MembershipCreateStatus.InvalidQuestion:
173                     return "Questão inválida para recuperar a senha.";
174 
175                 case MembershipCreateStatus.InvalidUserName:
176                     return "Usuário inválido.";
177 
178                 case MembershipCreateStatus.ProviderError:
179                     return "Ocorreu um erro durante a autenticação. Se o problema ↩
                           persistir, contate o administrador.";
180 
181                 case MembershipCreateStatus.UserRejected:
182                     return "O cadastro do usuário foi cancelado. Se o problema ↩
                           persistir, contate o administrador.";
183 
184                 default:
185                     return "Um erro inesperado ocorreu. Se o problema persistir, ↩
                           contate o administrador.";
186             }
187         }
188         #endregion
189     }
190 
191 }
```

35. Crie uma pasta **Usuario** na pasta **Views** e acrescente os quatro arquivos abaixo.

```
<!-- ~/Views/Usuario/ChangePassword.cshtml -->
@model K19_CopaDoMundo.Models.ChangePasswordModel

<h2>Alteração de Senha</h2>
<p>
    Utilize o formulário abaixo para alterar a senha
</p>
<p>
    Novas senhas devem ter no míinimo @Membership.MinRequiredPasswordLength caracteres de tamanho.
</p>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true, "Senha não foi alterada. Corrija os erros e tente novamente.")
    <div>
        <fieldset>
            <legend>Usuário</legend>

            <div class="editor-label">
                @Html.LabelFor(m => m.OldPassword)
            </div>
            <div class="editor-field">
                @Html.PasswordFor(m => m.OldPassword)
                @Html.ValidationMessageFor(m => m.OldPassword)
            </div>

            <div class="editor-label">
                @Html.LabelFor(m => m.NewPassword)
            </div>
            <div class="editor-field">
                @Html.PasswordFor(m => m.NewPassword)
                @Html.ValidationMessageFor(m => m.NewPassword)
            </div>

            <div class="editor-label">
                @Html.LabelFor(m => m.ConfirmPassword)
            </div>
            <div class="editor-field">
                @Html.PasswordFor(m => m.ConfirmPassword)
                @Html.ValidationMessageFor(m => m.ConfirmPassword)
            </div>

            <p>
                <input type="submit" value="Alterar senha" />
            </p>
        </fieldset>
    </div>
}
```

```
<!-- ~/Views/Usuario/ChangePasswordSuccess.cshtml -->
<h2>Alteração de senha</h2>
<p>
    Senha alterada com sucesso.
</p>
```

```
1  <!-- ~/Views/Usuario/LogOn.cshtml -->
2  @model K19_CopaDoMundo.Models.LogOnModel
3
4  <h2>Autenticar</h2>
5  <p>
6      Entre com usuário e senha. @Html.ActionLink("Register", "Register") se você não ↩
           tiver usuário.
7  </p>
8
9  <script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"><↩
       /script>
10 <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/↩
       javascript"></script>
11
12 @Html.ValidationSummary(true, "Login sem sucesso. Tente novamente")
13
14 @using (Html.BeginForm()) {
15     <div>
16         <fieldset>
17             <legend>Usuário</legend>
18
19             <div class="editor-label">
20                 @Html.LabelFor(m => m.UserName)
21             </div>
22             <div class="editor-field">
23                 @Html.TextBoxFor(m => m.UserName)
24                 @Html.ValidationMessageFor(m => m.UserName)
25             </div>
26
27             <div class="editor-label">
28                 @Html.LabelFor(m => m.Password)
29             </div>
30             <div class="editor-field">
31                 @Html.PasswordFor(m => m.Password)
32                 @Html.ValidationMessageFor(m => m.Password)
33             </div>
34
35             <div class="editor-label">
36                 @Html.CheckBoxFor(m => m.RememberMe)
37                 @Html.LabelFor(m => m.RememberMe)
38             </div>
39
40             <p>
41                 <input type="submit" value="Autenticar" />
42             </p>
43         </fieldset>
44     </div>
45 }
```

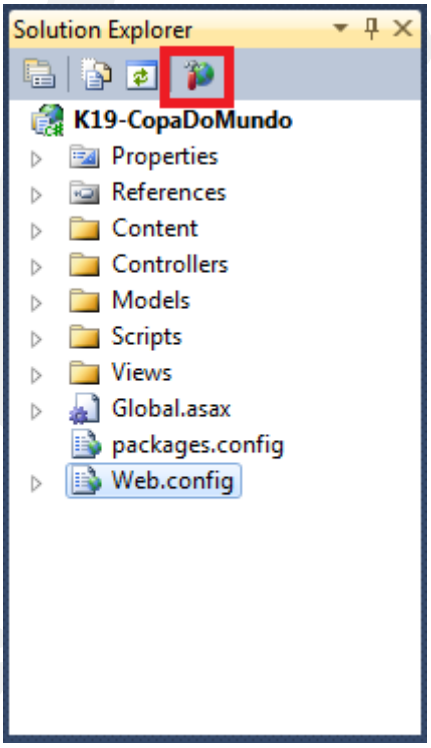
```
<!-- ~/Views/Usuario/Register.cshtml -->
@model K19_CopaDoMundo.Models.RegisterModel

<h2>Criar novo usuário</h2>
<p>
    Utilize o formulário abaixo para cadastrar um novo usuário.
</p>
<p>
    As senhas devem ter no mínimo @Membership.MinRequiredPasswordLength caracteres de tamanho.
</p>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true, "Usuário não foi criado. Corrija os erros e tente novamente")
    <div>
        <fieldset>
            <legend>Usuário</legend>

            <div class="editor-label">
                @Html.LabelFor(m => m.UserName)
            </div>
            <div class="editor-field">
                @Html.TextBoxFor(m => m.UserName)
                @Html.ValidationMessageFor(m => m.UserName)
            </div>

            <div class="editor-label">
                @Html.LabelFor(m => m.Email)
            </div>
            <div class="editor-field">
                @Html.TextBoxFor(m => m.Email)
                @Html.ValidationMessageFor(m => m.Email)
            </div>

            <div class="editor-label">
                @Html.LabelFor(m => m.Password)
            </div>
            <div class="editor-field">
                @Html.PasswordFor(m => m.Password)
                @Html.ValidationMessageFor(m => m.Password)
            </div>

            <div class="editor-label">
                @Html.LabelFor(m => m.ConfirmPassword)
            </div>
            <div class="editor-field">
                @Html.PasswordFor(m => m.ConfirmPassword)
                @Html.ValidationMessageFor(m => m.ConfirmPassword)
            </div>

            <p>
                <input type="submit" value="Cadastrar" />
            </p>
        </fieldset>
    </div>
}
```

12.21.1 Adicionando um Usuário Administrador com ASP .NET Configuration

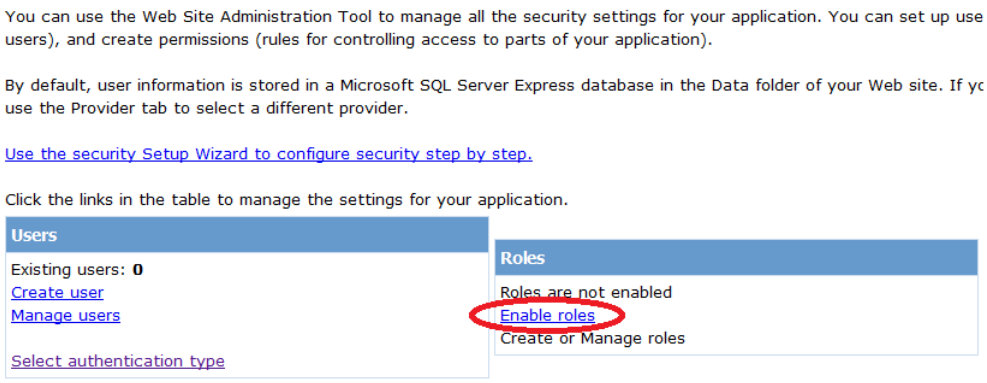
Antes de definir o filtro **Authorize** nos controladores de nosso site, vamos criar um usuário com acesso. A maneira mais fácil de criar o usuário é através do *ASP .NET Configuration*.

12.22 Exercícios

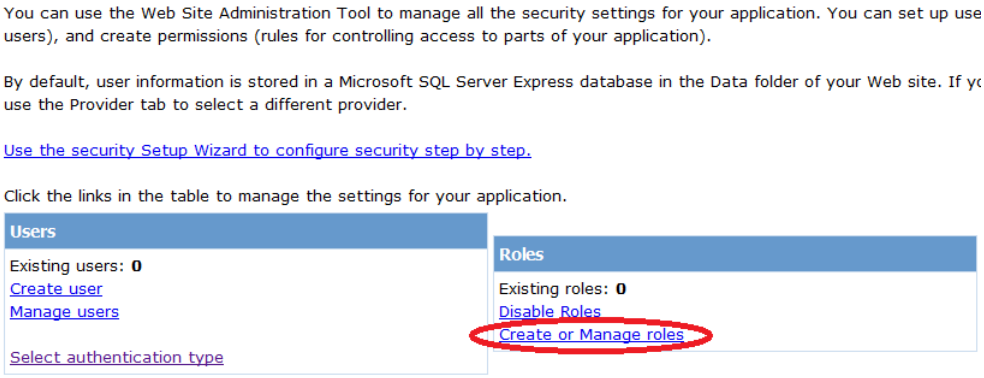
36. Execute o ASP .NET Configuration.



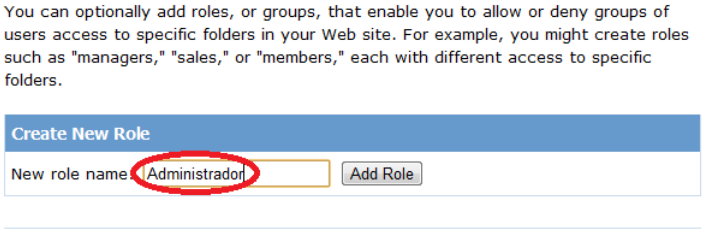
37. Isto executará um ambiente de configuração. Abra a aba "Security" e clique no link "Enable roles".



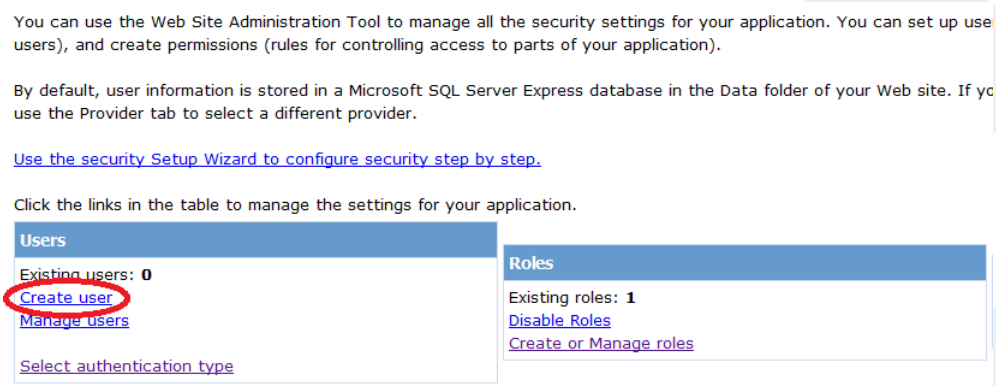
38. Posteriormente, clique em “Create or manage roles”.



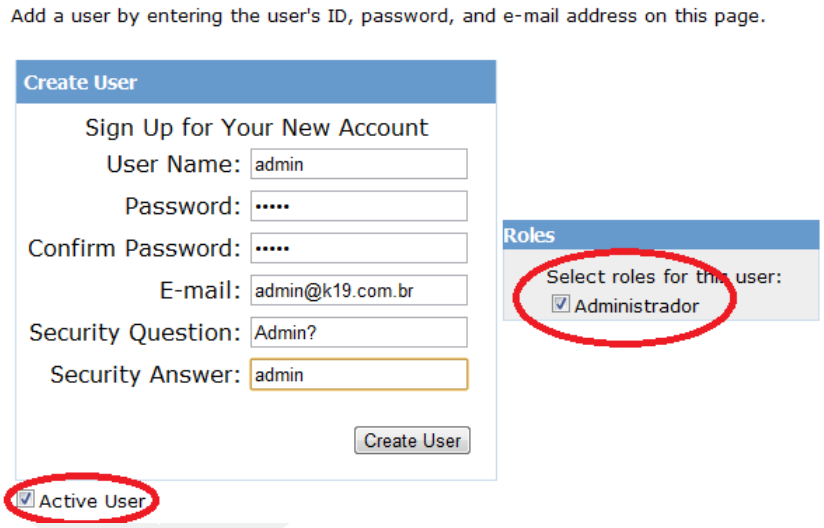
39. Defina um role “Administrador” e clique *Add Role*.



40. Clique no botão “back” e crie um usuário.



41. Defina um usuário *admin* e senha *admink19!*.



12.22.1 Autorização Role-based

Podemos restringir o acesso as páginas com o filtro **Authorize** e podemos especificar o role que o usuário precisa ter para ter acesso a página.

12.23 Exercícios

42. Altere o filtro de autenticação no **Web.config** para redirecionar o usuário para a action *LogOn* do controlador *Usuario*.

```
1 <!-- ~/Web.config -->
2 <authentication mode="Forms">
3     <forms loginUrl="~/Usuario/LogOn" timeout="2880" />
4 </authentication>
```

43. Acrescente a seguinte string de conexão no arquivo **Web.config** para definir o local que as informações dos usuários serão armazenadas (No nosso caso, teremos duas strings de conexão).

```
<!-- ~/Web.config -->
<connectionStrings>
    <add
      name="CopaDoMundoContext" providerName="System.Data.SqlClient"
      connectionString="Server=.\SQLEXPRESS;Database=copadomundo;
    User Id=sa; Password=sa;Trusted_Connection=False;Persist Security Info=True;↩
        MultipleActiveResultSets=True"/>
    <!-- Definindo o provedor para o Membership -->
    <add name="ApplicationServices"
         connectionString="data source=.\SQLEXPRESS;Integrated Security=SSPI;↩
             AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

44. Defina o provedor padrão que será utilizado. No nosso caso, utilizaremos o provedor padrão *SqlMembershipProvider*.

```
<!-- ~/Web.config -->
<system.web>
<!-- ... -->
<membership>
    <providers>
        <clear/>
        <add name="AspNetSqlMembershipProvider" type="System.Web.Security.↩
            SqlMembershipProvider" connectionStringName="ApplicationServices"
             enablePasswordRetrieval="false" enablePasswordReset="true" ↩
                 requiresQuestionAndAnswer="false" requiresUniqueEmail="false"
             maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6" ↩
                 minRequiredNonalphanumericCharacters="0" passwordAttemptWindow="10"
             applicationName="/" />
    </providers>
</membership>

<profile>
    <providers>
        <clear/>
        <add name="AspNetSqlProfileProvider" type="System.Web.Profile.↩
            SqlProfileProvider" connectionStringName="ApplicationServices" ↩
            applicationName="/" />
    </providers>
</profile>

<roleManager enabled="true">
    <providers>
        <clear/>
        <add name="AspNetSqlRoleProvider" type="System.Web.Security.SqlRoleProvider" ↩
            connectionStringName="ApplicationServices" applicationName="/" />
        <add name="AspNetWindowsTokenRoleProvider" type="System.Web.Security.↩
            WindowsTokenRoleProvider" applicationName="/" />
    </providers>
</roleManager>
<!-- ... -->
</system.web>
```

45. Acrescente o filtro de autenticação nos controladores Selecoes e Jogadores através do atributo **Authorize**.

```
1 //SelecoesController.cs
2 [Authorize(Roles="Administrador")]
3 public class SelecoesController : Controller
```

```
1 //JogadoresController.cs
2 [Authorize(Roles="Administrador")]
3 public class JogadoresController : Controller
```

12.24 Controle de Erro

Podemos configurar uma página de erro padrão para ser utilizada toda vez que um erro ocorrer.

12.25 Exercícios

46. Acrescente ao arquivo **Web.config** a tag **customErrors** para especificar a página de erro padrão. A tag **customErrors** fica dentro da tag **system.web**.

```
1 <system.web>
2 <!-- ... -->
3     <customErrors mode="On" defaultRedirect="~/Erro/Desconhecido">
4       <error statusCode="404" redirect="~/Erro/PaginaNaoEncontrada"/>
5     </customErrors>
6 <!-- ... -->
7 </system.web>
```

47. Defina o controlador **Erro** e as páginas de erros padrão. As páginas de erro padrão serão criadas dentro da pasta *Views* numa subpasta *Erro*.

```
1  namespace K19_CopaDoMundo.Controllers
2  {
3      public class ErroController : Controller
4      {
5          //
6          // GET: /Erro/Desconhecido
7
8          public ActionResult Desconhecido()
9          {
10             return View();
11         }
12
13         //
14         // GET: /Erro/PaginaNaoEncontrada
15         public ActionResult PaginaNaoEncontrada()
16         {
17             return View();
18         }
19
20     }
21 }
```

```
<!-- ~/Views/Erro/Desconhecido.cshtml -->
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>Problema no servidor</title>
</head>
<body>
    <h2>Desculpe, tivemos problema em nosso servidor. Volte dentro de alguns instantes.</h2>
</body>
</html>
```

```
<!-- ~/Views/Erro/PaginaNaoEncontrada.cshtml -->
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>Página não encontrada</title>
</head>
<body>
    <h2>Página não encontrada</h2>
</body>
</html>
```

48. Altere o método de listagem de jogadores para criar um erro em nosso site.

```
//JogadoresController.cs
// GET: /Jogadores
public ActionResult Index()
{
    //return View(unitOfWork.JogadorRepository.Jogadores);
    return View();
}
```

12.26 Enviando email

Quando um erro ocorre na nossa aplicação, podemos permitir que o usuário envie uma email para os administradores do sistema. Para enviar as mensagens, podemos utilizar o Web

12.27 Exercícios

49. Altere a tela de erro adicionando um formulário para o usuário escrever uma mensagem para os administradores da aplicação.

```
<!-- ~/Views/Erro/Desconhecido.cshtml -->
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>Problema no servidor</title>
</head>
<body>
    <h2>Desculpe, tivemos problema em nosso servidor. Volte dentro de alguns instantes.</h2>
    <p>Envie uma mensagem para os administradores do sistema.</p>
    @using (Html.BeginForm("Envia", "Email"))
    {
        <div class="editor-label">
            @Html.Label("Mensagem")
        </div>
        <div class="editor-field">
            @Html.TextArea("Mensagem")
        </div>
        <input type="submit" value="Enviar" />
    }
</body>
</html>
```

50. Crie um controlador que envie as mensagens por email utilizando o helper **WebMail**. Observação, utilize usuários, senhas e emails válidos do gmail para este exercício.

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

namespace K19_CopaDoMundo.Controllers
{
    public class EmailController : Controller
    {

        public EmailController()
        {
            WebMail.SmtpServer = "smtp.gmail.com";
            WebMail.EnableSsl = true;
            WebMail.SmtpPort = 587;
            WebMail.From = "USUARIO@gmail.com";
            WebMail.UserName = "USUARIO@gmail.com";
            WebMail.Password = "SENHA";
        }
        //
        // POST: /Email/Envia
        [HttpPost]
        public ActionResult Envia(string mensagem)
        {

            WebMail.Send("EMAIL", "Copa do Mundo - Erro", mensagem);
            return View();
        }

    }
}
```

51. Crie uma página **Envia.cshtml** para mostrar ao usuário que a mensagem foi enviada com sucesso e acrescente um link para a página inicial do site.

```
<!-- ~/Views/Email/Envia.cshtml -->
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>Envia</title>
</head>
<body>
    <div>
        Mensagem enviada com sucesso.
    </div>
    <div>
        @Html.ActionLink("Voltar para página inicial", "Index", "Selecoes")
    </div>
</body>
</html>
```