

Introdução ao ASP.NET MVC 4

C# + Entity Framework + Razor Engine

Visão Geral ASP.NET MVC.....	4
Definição	4
Models	4
Views	5
Controllers	5
Benefícios	5
Decidindo quando criar uma aplicação MVC	6
O modelo MVC oferece as seguintes vantagens:	6
Introdução ao "Razor" View Engine	7
Benefícios do "Razor" View Engine:	7
Pontos Negativos do Razor Engine	9
Entity Framework 5.0	9
Evolução do Entity Framework	10
Entity Framework Developer Workflows	11
Controle de Acesso com SimpleMembershipProvider	12
Estrutura de Dados do SimpleMembershipProvider	12
O método InitializeDatabaseConnection	13
Models, Views e Controllers relacionados	14
Regionalização das Strings e Mensagens do SimpleMembershipProvider	15
Cadastrando um Usuário	17
Cadastrando Grupos de Usuários (Roles)	17
Adicionando Usuários aos Grupos	17
Identificando o Usuário Atual	18
Encerrando uma Sessão de Usuário	18
Evitando Ataques do Tipo Cross-Site Request Forgery (CSRF)	18
Controlando o Acesso de Usuários e Grupos	19
Controle de acesso utilizando Web.config	19
Controle de acesso usando Atributos	20
Autenticação Integrada com Redes Sociais	21
Conceitos Básicos	22
Inicialização de Aplicações MVC	23
Páginas de Erros Customizadas	23
Erros gerados pelo HttpRequest	24
Erros gerados pelos Controllers, Actions e Classes	24
Arquitetura da nossa Aplicação MVC	25
O modelo Repository Pattern aplicado ao MVC	26
Repositórios, Inversion of Control (IoC) e UnitOfWork na prática	26
A estrutura do banco de dados de exemplo	28
Criando a Interface para nosso Repositório	29
Criando o mecanismo para IoC	30
O Repositório EFNatRepository	31
Otimizando LINQ Queries	33

Implementando UnitOfWork	34
Codificando nosso HomeController	34
A View Home.....	36
Finalizando a aplicação de exemplo	39
Práticas comuns (e recomendadas) para aplicações MVC.....	40
Validação com DataAnnotation Classes	40
Validando Tipos de Dados específicos	40
Validando Campos Requeridos	41
Validação com RegularExpressions	41
O atributo [Display]	41
Custom ValidationAttributes	41
Máscaras de entrada de dados	42
Exibindo notificações para o usuário.....	43
Trabalhando com Controles de Lista.....	43
Carregando DropDownLists.....	43
DropDownLists em Cascata com JQuery.....	44
Paginação de Registros no MVC	46
Implementando Paginação para o Grid	46
Conclusão Final.....	47
Referências.....	48

Visão Geral ASP.NET MVC

A arquitetura **Model-View-Controller (MVC)** proporciona a separação da aplicação em três componentes: **Model**, **View** e **Controller**. O **ASP.NET MVC** é um framework de apresentação altamente testado e integrado aos recursos do **ASP.NET**, como *master-pages* e *membership-based authentication* (da mesma forma como as aplicações desenvolvidas com **ASP.NET Web Forms**). O framework **MVC** é definido pelo Namespace **System.Web.Mvc**, parte fundamental do Namespace **System.Web**.

Definição

MVC é um modelo de design padrão, utilizado por muitos desenvolvedores. Algumas aplicações Web explorarão os novos recursos do **MVC** framework. Outras continuarão utilizando o modelo tradicional baseado em **Web Forms** e *postbacks*. Outras aplicações combinarão as duas abordagens. Nenhuma das abordagens exclui a outra.

O **MVC** framework inclui os seguintes componentes:

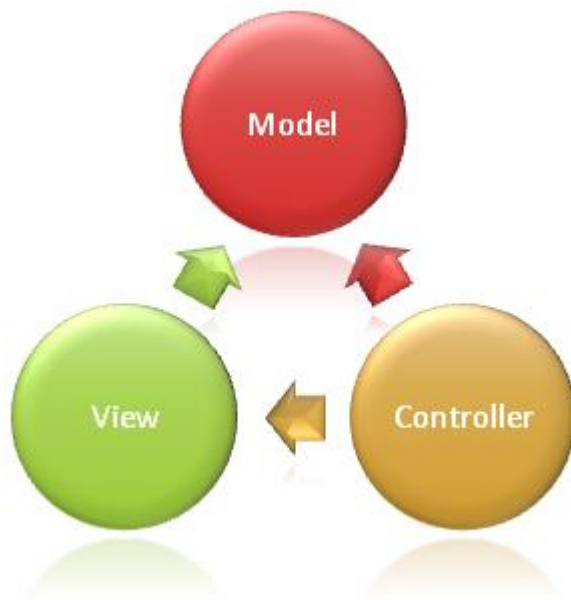


Figura 1: Componentes do ASP.NET MVC

Models

O objeto **Model** é a parte da aplicação que implementa a lógica para a camada de dados da aplicação. Frequentemente, o objeto **Model** é responsável por recuperar e persistir o estado do modelo no banco de dados. Por exemplo, um objeto Produto pode recuperar informações do banco de dados, operar com estas informações, e então persistir a informação atualizada na tabela de Produtos no **SQL Server**.

Em aplicações pequenas, o objeto **Model** é frequentemente separado conceitualmente e não fisicamente. Por exemplo, se a aplicação faz apenas a leitura de um conjunto de dados e os exibe utilizando um objeto **View**, a aplicação não terá necessariamente uma camada de dados física. Neste caso, o objeto **Model** representaria esta camada.

Views

Views são componentes que exibem a interface de usuário (**UI**). Tipicamente, esta **UI** é criada a partir do modelo de dados. Um exemplo seria um objeto **View** para editar a tabela de Produtos. Este objeto **View** exibe controles como caixas de texto (**TextBox**), caixas de listagem (**DropDownList**), e caixas de verificação (**CheckBox**) com dados que representam o estado atual do objeto Produto.

Controllers

Os objetos **Controllers** são componentes que gerenciam a interação do usuário, operam o objeto **Model**, e por último selecionam um objeto **View** para exibir os dados na camada do usuário (**UI**). Numa aplicação **MVC**, o objeto **View** apenas exibe informações, o objeto **Controller** gerencia e responde às entradas e interações do usuário. Por exemplo, o **Controller** gerencia os valores passados através de *query-string*, e passa esses valores para o objeto **Model**, que por sua vez consulta o banco de dados utilizando os valores recebidos.

Benefícios

A arquitetura **MVC** facilita a separação das diferentes camadas da aplicação (lógica de entrada de dados, lógica de negócios e lógica de apresentação), além de prover o conceito *loose coupling* entre esses componentes. Isto significa que cada componente tem ou usa, pouco ou nenhum conhecimento, sobre a implementação do outro. Esta arquitetura determina onde cada camada deverá ser implementada na aplicação. A camada de apresentação (**UI**) pertence aos objetos **View**. A lógica para gerenciar a entrada de dados e interações com o usuário pertence aos objetos **Controller**. As regras de negócio pertencem aos objetos **Model**. Esta separação auxilia a gerenciar a complexidade durante a construção da aplicação, porque ela permite que você se concentre na implementação de uma camada de cada vez. Por exemplo, você pode focar na construção de um objeto **View** sem depender ou conhecer a implementação da lógica de negócios implementada no objeto **Model**.

Além de auxiliar no gerenciamento da complexidade durante o desenvolvimento da aplicação, o modelo **MVC** facilita o trabalho de testar sua implementação, tornando-o mais fácil se comparado ao modelo **Web Forms**. Por exemplo, numa aplicação **ASP.NET** baseada em **Web Forms**, uma única classe é utilizada para exibir os dados e gerenciar as interações com o usuário. Escrever testes automatizados para uma aplicação **ASP.NET** baseada em **Web Forms** pode ser complexo, porque para testar uma página individualmente, você precisa instanciar a classe da página testada, todos os controles e classes dependentes. Como muitas classes podem ser instanciadas para executar uma única página, pode tornar-se difícil a tarefa de testar partes específicas da aplicação. Por isso testar uma aplicação **ASP.NET** baseada em **Web Forms** é uma tarefa mais difícil do que testar uma aplicação baseada no modelo **MVC**. Além disso, para testar uma aplicação **ASP.NET** baseada em **Web Forms** é necessário um **Web Server**. O framework **MVC** desconecta os componentes e faz uso de Interfaces, o que torna possível testar componentes individualmente, isolados do resto do framework.

A separação entre os principais componentes de uma aplicação **MVC** também promove o desenvolvimento paralelo (*parallel development*), onde um desenvolvedor pode trabalhar no objeto **View**, um segundo desenvolvedor pode trabalhar no objeto **Controller** enquanto um terceiro desenvolvedor se concentra nas regras de negócio implementadas no objeto **Model**.

Decidindo quando criar uma aplicação MVC

Você precisa considerar cuidadosamente quando construir uma aplicação utilizando **ASP.NET MVC** framework ou o modelo **ASP.NET baseado em Web Forms**. O framework **MVC** não substitui o modelo **Web Forms**; você pode utilizar qualquer das duas abordagens para desenvolver uma aplicação Web.

Antes de decidir pela arquitetura **MVC** ou **Web Forms** para um projeto específico, analise as vantagens e desvantagens de cada abordagem:

O modelo baseado em **Web Forms** oferece as seguintes vantagens:

- Suporta o modelo de eventos que preserva o estado da página através do protocolo **HTTP**, o que beneficia o desenvolvimento de aplicações **LOB** (*Line-of-Business*). O modelo baseado em **Web Forms** oferece dúzias de eventos que são suportados em centenas de *server-controls*.
- Utiliza *View State* e *server-based Forms*, que facilitam o gerenciamento de estado entre os *postbacks*.
- Funciona bem para pequenos times de desenvolvedores e designers que podem explorar um grande número de componentes disponíveis para o modelo de desenvolvimento **RAD** (*Rapid Application Development*).
- Em geral, apresenta uma complexidade menor, em razão dos componentes (**Page Class**, controles, modelo de eventos, etc.) bem integrados que exigem menos código quando comparados ao modelo **MVC**.

O modelo **MVC** oferece as seguintes vantagens:

- Separação das camadas da aplicação, testabilidade e desenvolvimento orientado a testes (**TDD**) são recursos nativos para o modelo **MVC**. Todos os contratos no **MVC** framework são baseados em **Interfaces** e podem ser testados com uso de objetos **mock**, que são simulações de objetos que imitam o comportamento de objetos reais da aplicação. Você pode utilizar testes unitários para validar a aplicação sem a necessidade de executar os objetos **Controllers** dentro de um processo **ASP.NET**, o que torna o processo de testes unitários mais rápido e flexível. Você pode utilizar qualquer framework de teste unitário compatível com **.NET Framework**.
- Um framework extensível e plugável. Os componentes do **MVC** framework foram desenhados de forma que possam ser facilmente substituídos ou customizados. Você pode conectar seu próprio processador de **Views**, política de roteamento de **URLs**, serialização de parâmetros para métodos, e outros componentes. O framework **ASP.NET MVC** também suporta os modelos de containers para **Injeção de Dependência** (*Dependency Injection – DI*) e **Inversão de Controle** (*Inversion of Control – IOC*). **DI** permite que você injete objetos dentro de uma classe, ao invés de utilizar a classe para criar o objeto. **IOC** especifica que se um objeto requer outro objeto, o primeiro objeto deve obter o segundo objeto de uma fonte externa como um arquivo de configuração. Isto torna o processo de testar a aplicação muito mais simples.

- Um poderoso componente de mapeamento de **URL** que permite que você construa aplicações que tenham **URLs** compreensíveis e pesquisáveis. **URLs** não precisam incluir nomes de arquivos e extensões, e são criadas para suportar modelos de nomenclatura de **URLs** que funcionam bem com os mecanismos de *search engine optimization* (**SEO**) e também com endereçamento *representational state transfer* (**REST**).
- Suporte para utilização de linguagem de marcação nas páginas **ASP.NET** (arquivos *.aspx), nos controles de usuário (arquivos *.ascx), nas *master-pages* (arquivos *.master) para criar modelos (*templates*) para objetos **View**. Você pode utilizar recursos existentes do **ASP.NET** combinados com o **MVC** framework, como *master-pages* aninhadas, expressões *in-line* (<%= %>), controles *server-based*, *data-binding*, localização, e outros.
- Suporte para recursos de segurança. A arquitetura **ASP.NET MVC** permite a utilização dos modelos **Forms Authentication**, **Windows Authentication**, **URL Authorization**, **Membership**, **Roles**, **Profiles**, **Caching**, entre outros.

Introdução ao “Razor” View Engine

O framework **ASP.NET MVC** sempre suportou o conceito de “**View Engines**”, que são módulos “plugáveis” que implementam diferentes modelos de sintaxe. O **View Engine** padrão para o **ASP.NET MVC** atualmente é o **ASPX** baseado no mesmo modelo utilizado pelos **Web Forms**. Outros **View Engines** populares para o **ASP.NET MVC** são **Spark** e **NHaml**.

O novo **View Engine** “**Razor**” foi otimizado em torno da geração de código **HTML** utilizando uma abordagem “*code-focused*”.

Benefícios do “Razor” View Engine:

Compacto, Expressivo e Fluido: O **View Engine** “**Razor**” minimiza o número de caracteres e *keystrokes* requeridos num arquivo, permitindo um fluxo de codificação rápido e conciso. Diferente da maioria dos padrões de sintaxe, você não precisa interromper sua codificação e declarar explicitamente blocos de processamento no servidor dentro do seu código **HTML**. O interpretador do “**Razor**” é inteligente o suficiente para inserir isso a partir do seu código. Isto permite uma sintaxe realmente compacta, expressiva, limpa, rápida e fácil de digitar.

Fácil de Aprender: A sintaxe do “**Razor**” é fácil de aprender e permite que você atinja um bom nível de produtividade com um mínimo de conceitos. Você pode utilizar toda sua experiência com sua linguagem **.NET** e com **HTML**.

Não é uma Linguagem: O “**Razor**” **View Engine** não é uma nova linguagem. Os desenvolvedores continuarão utilizando suas habilidades com **C#/VB.NET** em conjunto com a sintaxe “**Razor**”.

Trabalha com qualquer Editor de Texto: Não requer uma ferramenta específica e permite que você utilize qualquer editor de texto para escrever seu código. O próprio **Bloco de Notas** pode ser utilizado.

Suporte para Intellisense: Embora o “**Razor**” não exija uma ferramenta específica para codificação, o **Visual Studio 2010** e **2012** oferece suporte completo para utilizar o **Intellisense** com o “**Razor**” **View Engine**.

Teste Unitário: O “**Razor**” **View Engine** oferece suporte para testes unitários com objetos **View**, sem requerer um objeto **Controller** ou um **Servidor Web**. Os testes podem ser armazenados em qualquer projeto de teste, nenhum domínio (*app-domain*) especial é requerido.

Observe a seguir alguns exemplos das diferenças entre as sintaxes **ASPX** e **“Razor”**.

ASPX:

```
<h1>Code Nugget Example with .ASPX file</h1>

<h3>
    Hello <%=name %>, the year is <%= DateTime.Now.Year %>
</h3>

<p>
    Checkout <a href="/Products/Details/<%=productId %>">this product</a>
</p>
```

Razor:

```
<h1>Razor Example</h1>

<h3>
    Hello @name, the year is @DateTime.Now.Year
</h3>

<p>
    Checkout <a href="/Products/Details/@productId">this product</a>
</p>
```

ASPX:

```
<ul id="products">

    <% foreach(var p in products) { %>
        <li><%=p.Name%> ($<%=p.Price%>)</li>
    <% } %>

</ul>
```

Razor:

```
<ul id="products">

    @foreach(var p in products) {
        <li>@p.Name ($@p.Price)</li>
    }

</ul>
```


Pontos Negativos do Razor Engine

Pessoalmente, sou um defensor do **Razor Engine**, sua sintaxe concisa facilita a codificação, além de tornar o código mais limpo e de fácil leitura. Outros desenvolvedores podem ter preferências diferentes, é uma questão de sabor. Mas, não posso deixar de citar um ponto negativo do **Razor**. O **Visual Studio** não suporta o **Design Mode** para visualização das *Views*. Um dos motivos está relacionado aos arquivos das *Views* criadas com o **Razor**, que possuem extensão **.cshtml*, o **Visual Studio** não permite a exibição em **Design Mode** para estes arquivos.

O desenvolvedor tem duas opções para contornar este problema, a primeira é alterar as extensões dos arquivos para **.aspx*, e a segunda opção é não utilizar o **Razor Engine**. Oferecer suporte para **Design Mode** para o **Razor Engine** não está entre as prioridades da **Microsoft**.

Se você está habituado a utilizar muito o **Design Mode**, minha sugestão é que utilize o *engine* padrão do **ASP.NET** (**.aspx*), a diferença de sintaxe pode ser compensada pelo uso do **Design Mode** e o uso do engine padrão não afetará sua aplicação, a preferência pelo **Razor Engine** está relacionada à produtividade que ele proporciona durante o processo de codificação.

Entity Framework 5.0

Entity Framework é a ferramenta de **Mapeamento Objeto-Relacional (ORM)** criada pela **Microsoft** e agora liberada dentro do modelo de licença **Open Source Apache 2.0** conforme divulgado no portal **CodePlex** (<http://entityframework.codeplex.com>).

Entity Framework permite aos desenvolvedores, que trabalham com dados relacionais, utilizarem objetos (*domain-specific objects*), eliminando a necessidade do código de acesso a dados que normalmente temos que escrever.

Com **ADO.NET Providers**, o **Entity Framework** oferece suporte para **SQL Server**, **Oracle**, **MySQL**, **PostgreSQL**, **Sybase**, **Informix**, **Firebird**, **DB2**, **Caché**, e outros bancos de dados disponíveis no mercado.

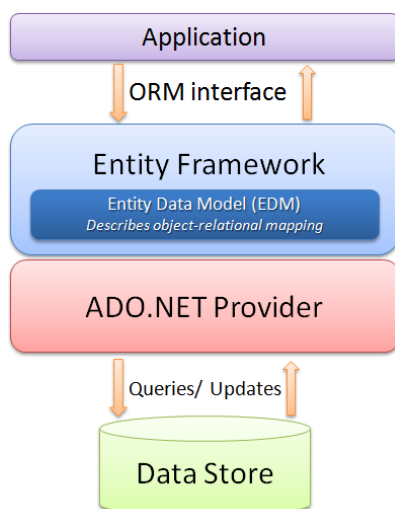


Figura 2: Arquitetura do Entity Framework

Evolução do Entity Framework

O **EF 5.0**, disponível através do **NuGet** é compatível com o **Visual Studio 2010** e **Visual Studio 2012** e pode ser utilizado em aplicações baseadas no **.NET Framework 4.0** e **4.5**.

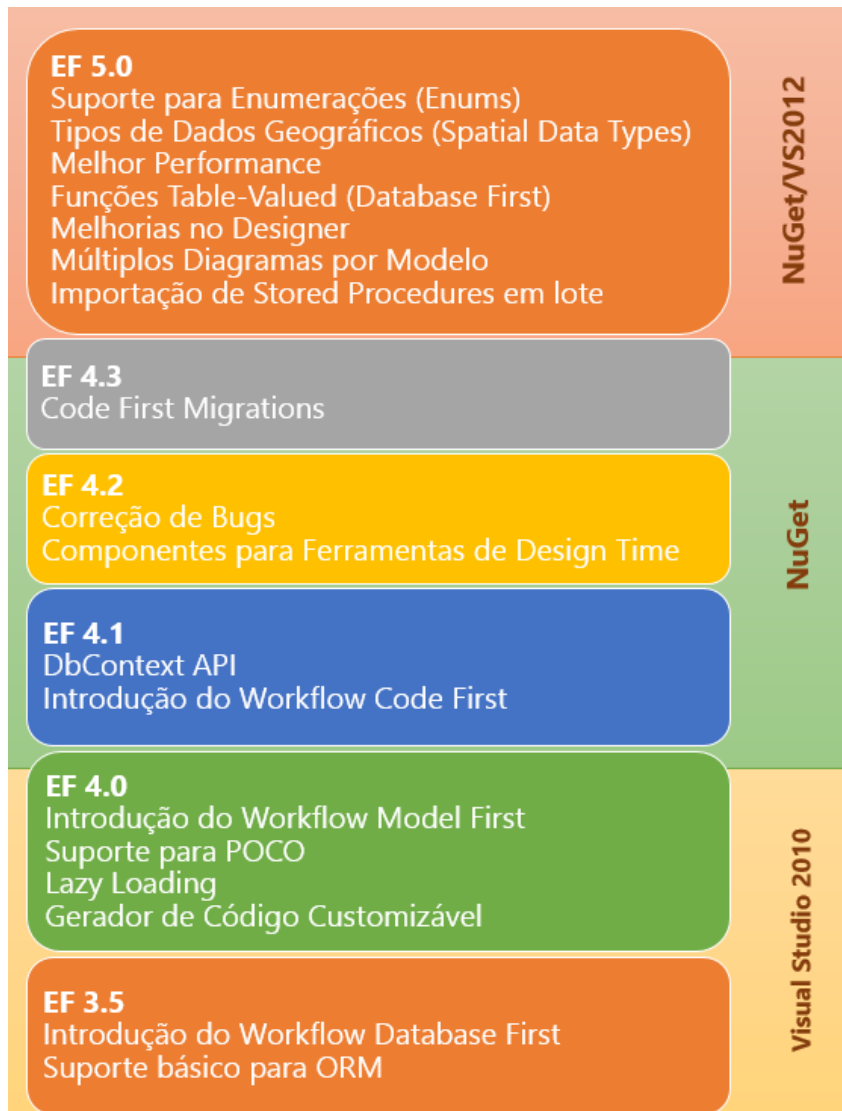


Figura 3: Evolução do Entity Framework

Entity Framework Developer Workflows

O **Entity Framework** oferece três abordagens para desenvolvimento, com objetivo de atender diferentes cenários e *skills* da equipe de desenvolvimento. As abordagens são: **Database First**, **Model First** e **Code First**. Veja abaixo qual é a abordagem mais adequada para seu projeto.

Cenários	Preferência por Código
Novo Banco de Dados	Code First <ul style="list-style-type: none"> Definição das Classes e Mapeamento no Código Cria/Atualiza o Banco de Dados em Runtime
Banco de Dados Existente	Code First <ul style="list-style-type: none"> Cria Modelo EDMX utilizando o designer; Cria/Atualiza o Banco de Dados a partir do EDMX; Classes geradas automaticamente a partir do EDMX;

Cenários	Preferência por Designer
Novo Banco de Dados	Model First <ul style="list-style-type: none"> Cria Modelo EDMX utilizando o designer Cria/Atualiza o Banco de Dados a partir do EDMX Classes geradas automaticamente a partir do EDMX
Banco de Dados Existente	Database First <ul style="list-style-type: none"> Cria Modelo EDMX utilizando Engenharia Reversa Classes geradas automaticamente a partir do EDMX

Modelos de Desenvolvimento (Developer Workflows)

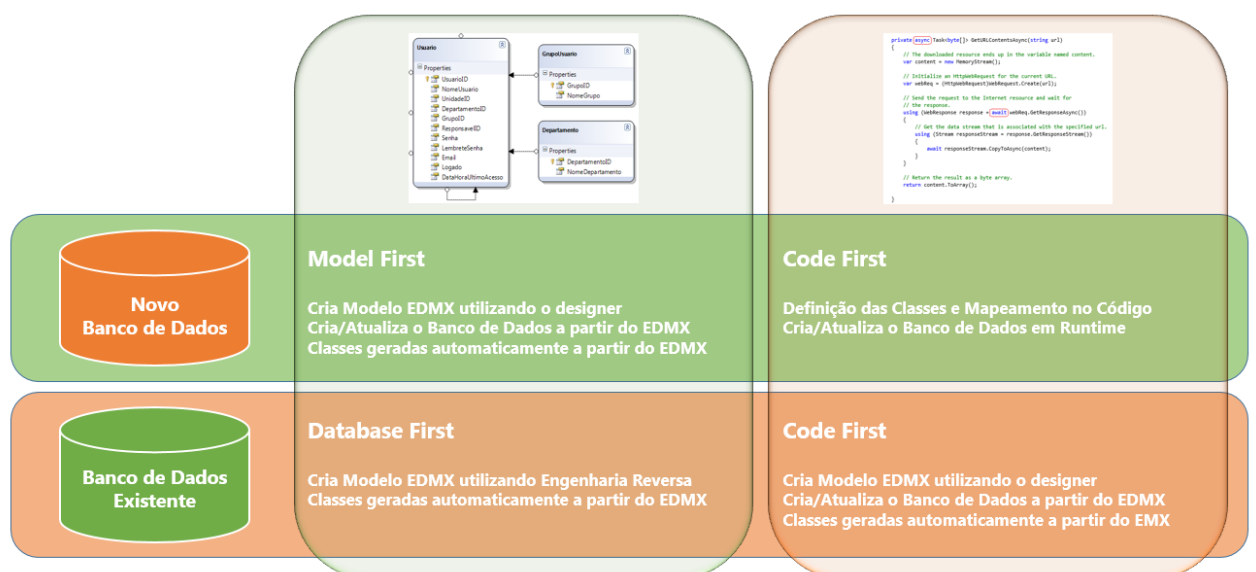


Figura 4: Developer Workflows para Entity Framework

Controle de Acesso com SimpleMembershipProvider

O modelo adotado como padrão para segurança e controle de acesso das aplicações **ASP.NET MVC** é o **SimpleMembershipProvider**. Este modelo é uma versão mais enxuta e otimizada do modelo **MembershipProvider** já conhecido pelos desenvolvedores **ASP.NET WebForms**. Sua implementação é simples.

Estrutura de Dados do SimpleMembershipProvider

Abaixo está o diagrama contendo a estrutura de dados utilizada do **SimpleMembershipProvider**. Diferente de seu “irmão mais velho”, nesta versão, as *views* e *stored procedures* foram removidas e a estrutura das tabelas foi simplificada. Num primeiro momento, o desenvolvedor poderá sentir falta da ferramenta **aspnet_regsql.exe** que utilizávamos para criar todos os objetos de dados necessários para o **MembershipProvider**. Bastava executar este utilitário, apontar para o banco de dados desejado e ele se encarregava de criar as tabelas, *views* e *stored procedures*. Nesta versão mais simplificada e otimizada esta tarefa também ficou mais simples, o método **InitializeDatabaseConnection** se encarrega desta tarefa. A estrutura de dados é criada utilizando o novo recurso do **SQL Server**, o **LocalDb**.

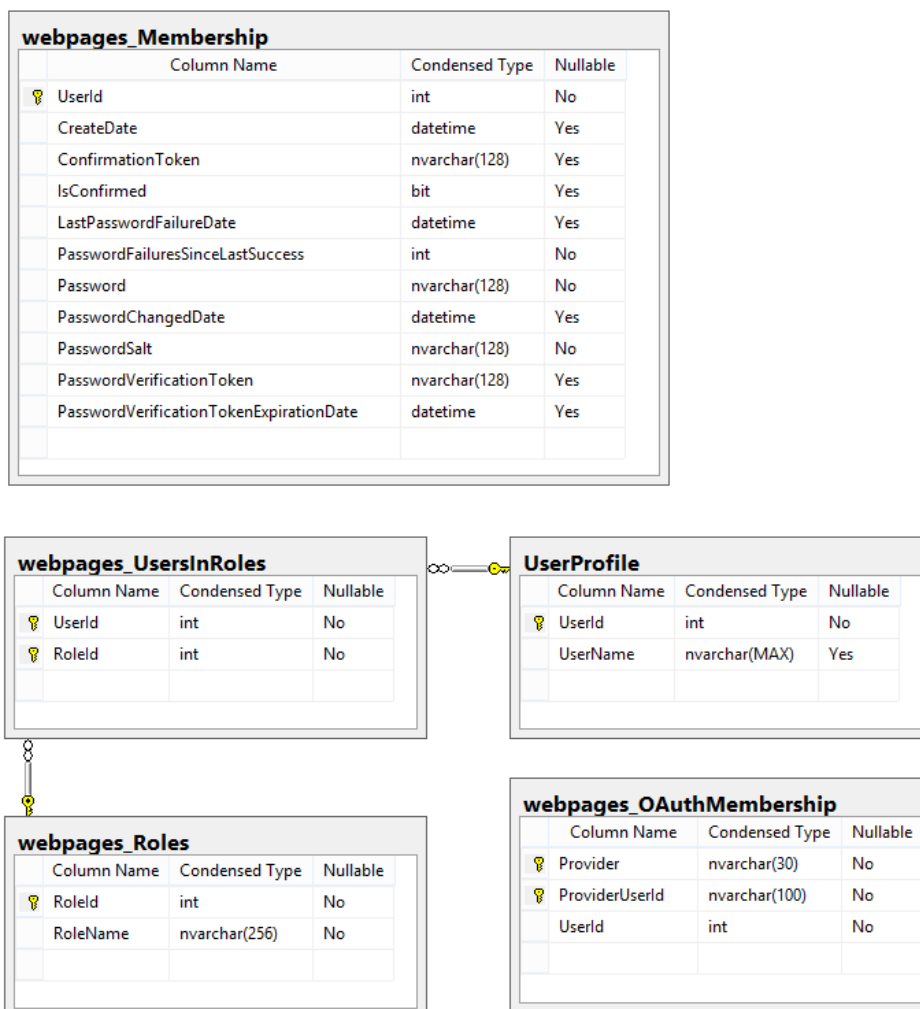


Figura 5: Diagrama de Dados SimpleMembershipProvider

A **Figura 6** mostra a *string* de conexão padrão criada para o **SimpleMembershipProvider**.

```
11 <connectionStrings>
12   <add name="DefaultConnection"
13       connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-TutorialMVC-20121118140917;
14       Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-TutorialMVC-20121118140917.mdf"
15       providerName="System.Data.SqlClient" />
16 </connectionStrings>
```

Figura 6: Connection String (default) para SimpleMembershipProvider

O método InitializeDatabaseConnection

O objeto **WebSecurity** implementa o método **InitializeDatabaseConnection**. Este método requer cinco argumentos, descritos na **Tabela 1**.

DefaultConnection	ConnectionString válida que indica o servidor e banco de dados onde serão armazenadas as informações de usuários e grupos.
UserProfile	<i>String</i> com nome da tabela que contém o perfil de usuários.
UserID	<i>String</i> com nome da coluna que contém o ID do usuário na tabela de perfil de usuários.
UserName	<i>String</i> com nome da coluna que contém o Username na tabela de perfil de usuários.
autoCreateTables	<i>Boolean</i> que determina se o mecanismo do SimpleMembershipProvider deverá criar automaticamente as tabelas, caso elas não sejam encontradas no banco de dados especificado na <i>string</i> de conexão.

Tabela 1: Argumentos do método InitializeDatabaseConnection

Este método é invocado no construtor da classe **SimpleMembershipInitializer** que é parte da implementação do atributo **InitializeSimpleMembershipAttribute**. Consulte o arquivo de mesmo nome localizado na pasta **Filter** da sua solução.

Sempre que sua aplicação consumir um método ou propriedade do objeto **WebSecurity**, certifique-se de que o atributo **InitializeSimpleMembership** foi devidamente invocado. Explicando melhor, se você tem uma classe onde vários métodos interagem com o **SimpleMembershipProvider**, defina este atributo na declaração da classe. Este atributo não pode ser invocado mais de uma vez para a mesma classe, isto provocará uma exceção.

Ao criar um novo projeto baseado no template **ASP.NET MVC Application**, o desenvolvedor não precisa preocupar-se com as referências, *assemblies* e *namespaces* necessários para implementar o mecanismo de autenticação e controle de acesso de usuários, o **Visual Studio** se encarrega desta tarefa.

Observe a **Figura 7**. Ela apresenta o atributo **InitializeSimpleMembership** na declaração da classe **AccountController** onde a maioria dos métodos interagem com o **SimpleMembershipProvider**.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Transactions;
5 using System.Web;
6 using System.Web.Mvc;
7 using System.Web.Security;
8 using DotNetOpenAuth.AspNet;
9 using Microsoft.Web.WebPages.OAuth;
10 using WebMatrix.WebData;
11 using TutorialMVC.Filters;
12 using TutorialMVC.Models;
13
14 namespace TutorialMVC.Controllers
15 {
16     [Authorize]
17     [InitializeSimpleMembership]
18     public class AccountController : Controller
19     {
20         //
21         // GET: /Account/Login
22
23         [AllowAnonymous]
24         public ActionResult Login(string returnUrl)
25         {
26             ViewBag.ReturnUrl = returnUrl;
27             return View();
28         }
29     }
30 }
```

Figura 7: O Atributo InitializeSimpleMembership

Models, Views e Controllers relacionados

Falando ainda sobre o modelo de autenticação e controle de acesso dos usuários, a solução baseada no **SimpleMembershipProvider**, diferente do modelo utilizado por **WebForms**, em lugar dos *user-controls* (*.ascx) para os implementar os controles constituintes como o formulário de Login, Cadastro de Usuários e Recuperação de senhas, no modelo utilizado no **ASP.NET MVC**, temos um **Controller**, chamado **AccountController**, uma classe **Model** chamada **AccountModels**, e as **Views** **Login**, **Manage**, **ChangePassword**, **Register** entre outras. Ao mesmo tempo que estas **Views** oferecem uma implementação rápida e prática do modelo de autenticação e controle de acesso baseado em perfis de usuários, é importante lembrar que será necessário um pouco de trabalho para a regionalização das *strings* contidas nos *labels* e mensagens deste modelo.

A **Figura 8** apresenta a janela **Solution Explorer** exibindo os arquivos gerados automaticamente para contemplar as funcionalidades do **SimpleMembershipProvider**.

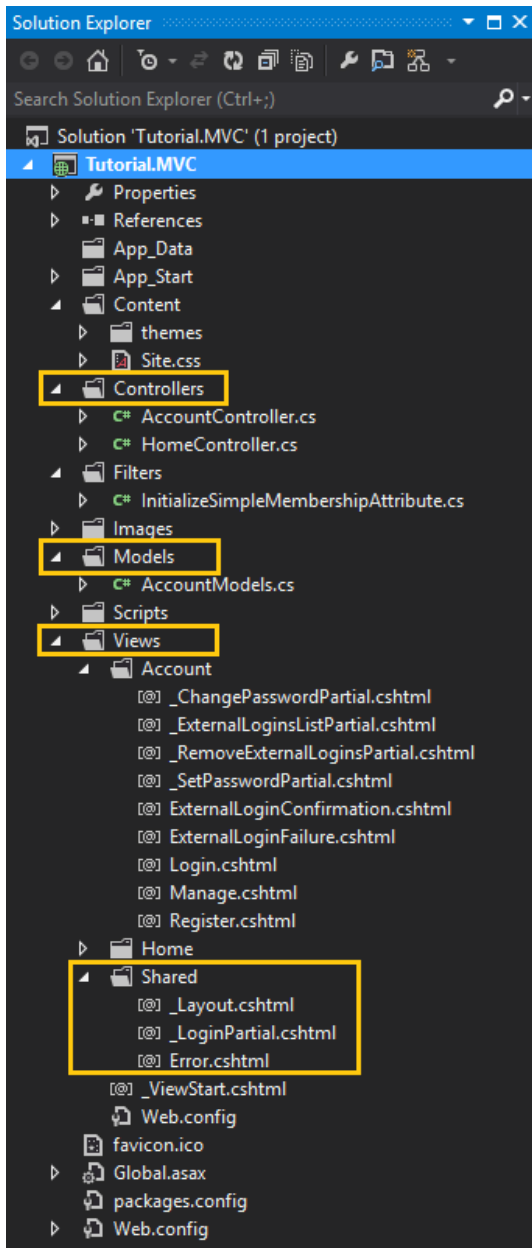


Figura 8: Arquivos do SimpleMembershiProvider

Regionalização das Strings e Mensagens do SimpleMembershipProvider

Alterar as *strings* apresentadas nos *labels* e mensagens do mecanismo de autenticação do **SimpleMembershipProvider** é uma tarefa simples. A maior parte destas *strings* estão localizadas nos atributos da classe **AccountModels.cs**. Este atributos baseiam-se no modelo **Data Annotation Classes** implementado pelo **Namespace System.ComponentModel.DataAnnotations**. Atributos desta categoria serão abordados em outro tópico deste mesmo tutorial. Para mais informações

sobre o uso dos **Data Annotation Classes Attributes**, consulte <http://msdn.microsoft.com/en-us/library/dd901590%28v=vs.95%29.aspx>. A **Figura 9** ilustra alguns destes atributos.

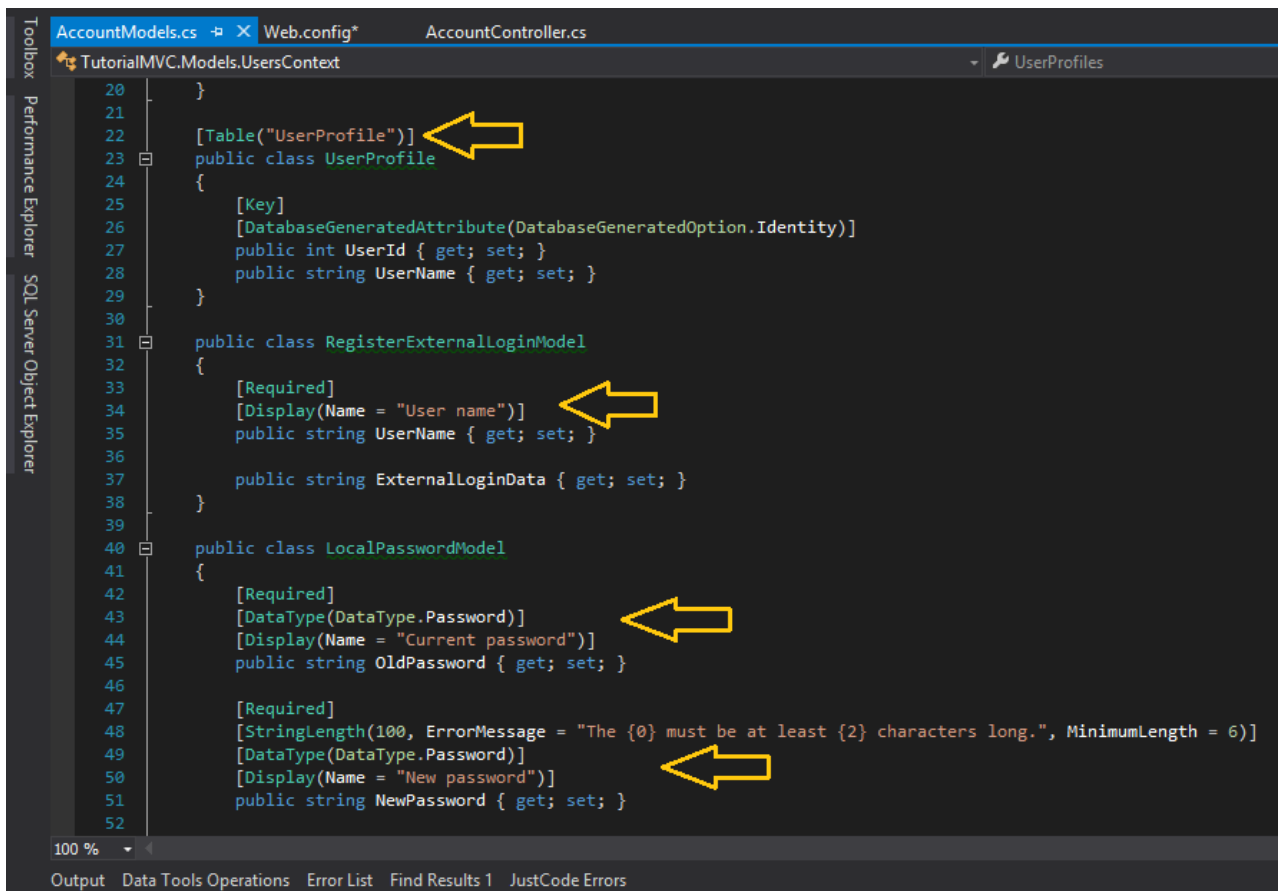


Figura 9: Data Annotation Classes Attributes

A maioria das *strings* utilizadas pelas *Views* do **SimpleMembershipProvider** baseiam-se no argumento **Name** do atributo **[Display]** para alimentar os *labels* dos controles apresentados em suas páginas e nas mensagens de erros dos atributos de validação. Por exemplo, o atributo **Display**, possui dois argumentos **Name** e **Description**. O primeiro, é o texto que será exibido ao lado do controle como se fosse um *label* do controle. A leitura deste atributo é feita através dos métodos **LabelFor** ou **DisplayFor** do objeto **Html**, usado largamente para a construção das *Views*. A sintaxe é simples, veja o exemplo a seguir:

```
(na propriedade Address da classe Customer.cs)
[Display(Name="Endereço", Description="Endereço/Logradouro do Cliente")]

(Na View, CustomerView.cshtml)
Html.DisplayFor(model => item.Address)
```

No exemplo acima, o *label* para o campo **Address** da **CustomerView** exibirá o texto "Endereço" indicado pelo argumento *Name* do Atributo **Display**. Ao posicionar o cursor sobre este campo, um *ToolTipText* será exibido contendo o texto do argumento *Description* do mesmo atributo.

Retomando o ponto inicial deste tópico, para regionalizar as *strings* utilizadas nos *labels* e mensagens do mecanismo de autenticação do **SimpleMembershipProvider**, o desenvolvedor deve substituir estas *strings* por textos do idioma desejado.

A Globalização de *strings* para suporte a múltiplos idiomas não é objeto de estudo deste tutorial. Para maiores informações sobre a globalização de aplicações **ASP.NET MVC** consulte este exemplo <http://afana.me/post/aspnet-mvc-internationalization.aspx>.

Cadastrando um Usuário

A *View* **Register.cshtml** encontrada na pasta **Views/Account** oferece uma alternativa para cadastro de novos usuários. Contudo, o desenvolvedor poderá criar outras *views* ou formulários para esta finalidade. O *Namespace* **WebMatrix.WebData** implementa o objeto **WebSecurity** que expõe o método **CreateUserAndAccount(string userName, string Password)**. Este método cria um novo registro nas tabelas **UserProfile** e **webpages_Membership**. As senhas são criptografadas e persistidas na tabela **webpages_Membership**. A ação **Register** da classe **AccountController** apresenta um exemplo de utilização do método **CreateUserAndAccount**, abaixo um fragmento de código extraído desta classe.

```
...
WebSecurity.CreateUserAndAccount(model.UserName, model.Password);
...
```

Cadastrando Grupos de Usuários (Roles)

O template **ASP.NET MVC Application** não oferece uma *View* ou formulário para gerenciar o cadastro de grupos de usuários (*Roles*). Mas o desenvolvedor pode utilizar a classe **Roles** (**System.Web.Security**) para esta tarefa. Esta classe expõe os métodos **CreateRole** e **DeleteRole**, ambos requerem um único parâmetro do tipo *string* contendo o nome da **Role** que deseja incluir ou excluir. Por tratar-se de uma classe estática, não há necessidade de instanciá-la, basta declarar o *Namespace* **System.Web.Security** para invocar os métodos desejados. Veja o exemplo abaixo:

```
...
using System.Web.Security;
...
Roles.CreateRole("Gerentes");
...
if (Roles.RoleExists("Gerentes"))
{
    Roles.DeleteRole("Gerentes");
}
...
```

Adicionando Usuários aos Grupos

Para adicionar ou remover usuários aos grupos (*Roles*) o desenvolvedor deve utilizar o mesmo objeto **Roles** (**System.Web.Security**). Este objeto oferece oito métodos (4 métodos de inclusão e 4 métodos de exclusão) que permitem o gerenciamento de usuários e grupos (*Roles*). A **Tabela 2** descreve os métodos e seus argumentos.

Método	Argumentos	Descrição
AddUsersToRole	string[] Users, string roleName	Adiciona todos usuários informados no parâmetro Users ao grupo informado no argumento roleName.
AddUsersToRoles	string[] Users, string[] Roles	Adiciona todos os usuários informados no parâmetro Users à todos os grupos informados no parâmetro Roles.
AddUserToRole	string userName, string roleName	Adiciona o usuário informado no parâmetro userName ao grupo informado no parâmetro roleName
AddUserToRoles	string userName, string[] Roles	Adiciona o usuário informado no parâmetro userName à todos os grupos informados no parâmetro Roles.

Tabela 2: Gerenciando Usuários e Grupos

Identificando o Usuário Atual

O objeto **WebSecurity** expõe duas propriedades **CurrentUserId** (*int*) e **CurrentUserName** (*string*) que são alimentadas com o código e o nome de usuário atual, ou seja, o usuário que está autenticado na sessão corrente.

Encerrando uma Sessão de Usuário

O formulário contido na View **_LoginPartial** localizada na pasta **Views/Shared/** oferece uma alternativa para encerrar a sessão do usuário corrente. Este formulário invoca a ação (*action*) **LogOff** da classe **AccountController** que se encarrega de finalizar a sessão do usuário.

O desenvolvedor pode criar outros formulários ou *views* que ofereçam um atalho para finalizar a sessão atual. Para isto, é necessário invocar o método **Logout()** do objeto **WebSecurity** (**WebMatrix.WebData**). Este método não requer nenhum parâmetro e pode ser chamado de qualquer parte da sua aplicação. Ao ser invocado, a sessão do usuário é finalizada e a aplicação direcionada para a página de **Login** ou para a **URL** redirecionada pela aplicação.

Evitando Ataques do Tipo Cross-Site Request Forgery (CSRF)

Um recurso de segurança das aplicações **ASP.NET MVC** para evitar os ataques do *tipo Cross-Site Request Forgery* é o *Token* **AntiForgeryToken**. O desenvolvedor pode utilizar este recurso para proteger sua aplicação. Sua sintaxe é simples. Veja o exemplo abaixo:

Na View, o objeto **Html** é utilizado para gerar o *Token* utilizando um *hidden field* para armazená-lo no formulário. Este *token* é enviado para o servidor sempre que o formulário é submetido.

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

Uma vez no servidor, o atributo **[ValidateAntiForgeryToken]** garante que a ação ou método será executada apenas e somente após a validação do token recebido. Observe o exemplo a seguir.

```
...
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Disassociate(string provider, string providerUserId)
...

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public ActionResult Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid && WebSecurity.Login(model.UserName, model.Password, persistCookie: model.RememberMe))
    {
        return RedirectToLocal(returnUrl);
    }

    // If we got this far, something failed, redisplay form
    ModelState.AddModelError("", "A combinação de usuário e senha informada está incorreta.");
    return View(model);
}

// ...

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult LogOff()
{
    WebSecurity.Logout();
    return RedirectToLocal("http://www.chocolatesbrasilcacao.com.br/index.php/promocao-casas-em-dobro/");
}
```

Figura 11: O Atributo **ValidateAntiForgeryToken**

Controlando o Acesso de Usuários e Grupos

Nas aplicações **ASP.NET WebForms** o desenvolvedor está habituado a utilizar os elementos **<authentication>**, **<authorization>** e **<location>** no arquivo de configuração **Web.config** para definir as permissões de acesso para os usuários da sua aplicação. As aplicações **ASP.NET MVC** oferecem os atributos **[Authorize]** e **[AllowAnonymous]** para que o desenvolvedor possa gerenciar as permissões de acesso aos *controllers* e *actions* da sua aplicação.

Controle de acesso utilizando Web.config

Tenho observado alguns sites que orientam equivocadamente os desenvolvedores a utilizarem o elemento **<location>** para definir as permissões de acesso para as rotas da sua aplicação **MVC**. Esta prática não é recomendada uma vez que sua concepção não contempla o modelo de rotas, controllers e ações, e os resultados serão prejudiciais à sua aplicação.

Contudo, a definição de permissões de acesso no arquivo de configuração é uma prática comum em aplicações web. Uma alternativa, neste caso, é criar novas sessões de configuração em seu arquivo **Web.config** para esta finalidade. Um bom exemplo desta implementação pode ser encontrado neste endereço: <http://www.ryanmwright.com/2010/04/25/dynamic-controlleraction-authorization-in-asp-net-mvc/>.

Controle de acesso usando Atributos

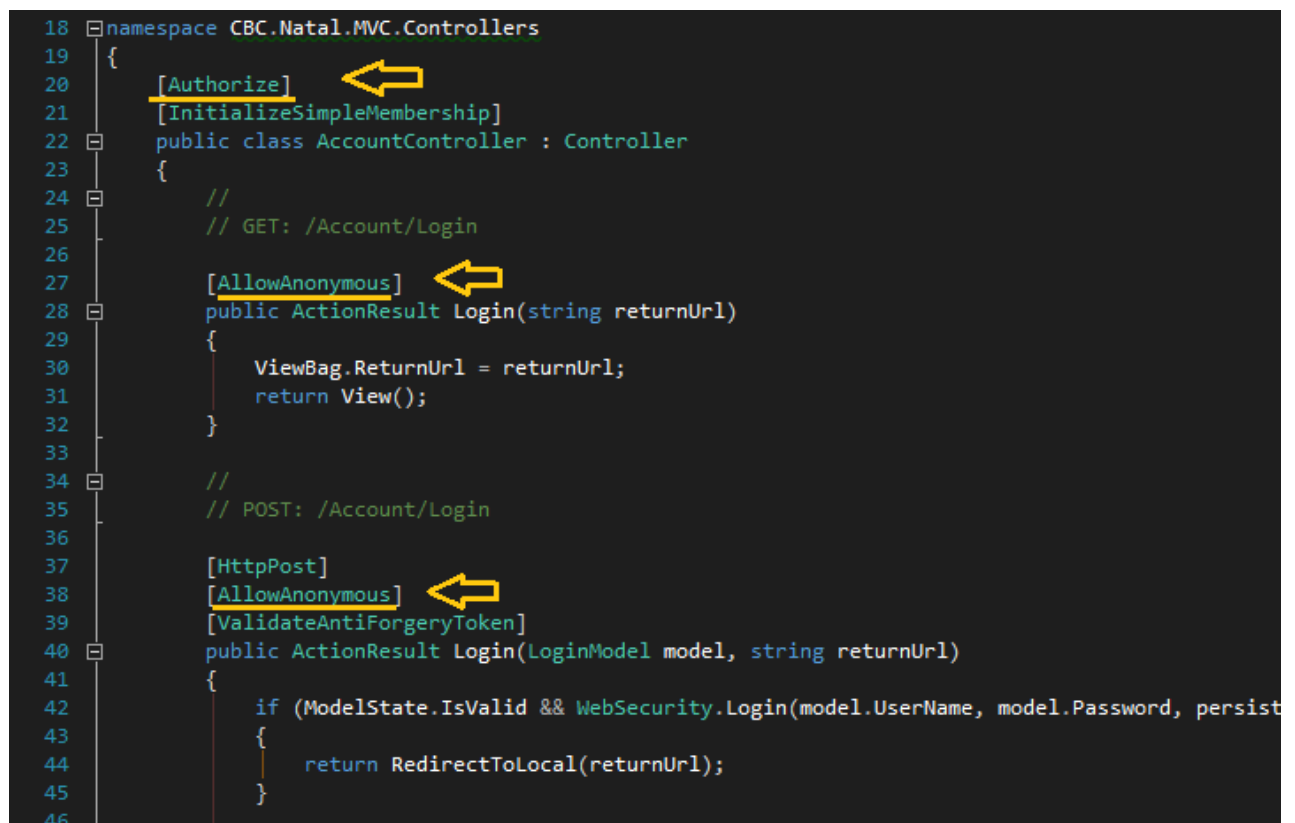
A prática recomendada baseia-se nos atributos **[Authorize]** e **[AllowAnonymous]**, ambos integrantes do *namespace* **System.Web.Mvc**.

O atributo **[Authorize]** pode ser definido no escopo de uma classe ou de um método específico. Quando definido na declaração de uma classe, o atributo **[Authorize]** exigirá um usuário autenticado para acessar qualquer recurso contido nesta classe.

O desenvolvedor deve combinar a utilização dos atributos **[Authorize]** e **[AllowAnonymous]** para atingir o objetivo desejado.

Tome como exemplo a classe **AccountController**. A maioria dos métodos/ações desta classe requerem um usuário autenticado, contudo, os métodos **Login** e **Register** devem permitir o acesso de usuários anônimos. A combinação dos atributos **[Authorize]** e **[AllowAnonymous]** contemplam este objetivo.

A **Figura 12** ilustra o exemplo da classe **AccountController.cs** com os atributos **[Authorize]** e **[AllowAnonymous]**.



```
18 namespace CBC.Natal.MVC.Controllers
19 {
20     [Authorize]
21     [InitializeSimpleMembership]
22     public class AccountController : Controller
23     {
24         //
25         // GET: /Account/Login
26
27         [AllowAnonymous]
28         public ActionResult Login(string returnUrl)
29         {
30             ViewBag.ReturnUrl = returnUrl;
31             return View();
32         }
33
34         //
35         // POST: /Account/Login
36
37         [HttpPost]
38         [AllowAnonymous]
39         [ValidateAntiForgeryToken]
40         public ActionResult Login(LoginModel model, string returnUrl)
41         {
42             if (ModelState.IsValid && WebSecurity.Login(model.UserName, model.Password, persist
43             {
44                 return RedirectToLocal(returnUrl);
45             }
46         }
```

The image shows a code editor with a dark background. The code is in C# and defines the `AccountController` class. Three yellow arrows point to specific attributes: the first arrow points to `[Authorize]` on line 20, the second arrow points to `[AllowAnonymous]` on line 27, and the third arrow points to `[AllowAnonymous]` on line 38. The code includes comments for GET and POST actions, and a `ValidateAntiForgeryToken` attribute for the POST action.

Figura 12: Os atributos **[Authorize]** e **[AllowAnonymous]**

Até agora estamos falando apenas de restringir ou permitir o acesso para usuários autenticados, não nos preocupamos com o perfil deste usuário, por exemplo, a que grupo ou grupos pertence, ou quem é este usuário especificamente. Para atingir este objetivo, utilizaremos as variações do atributo **[Authorize]**.

Este atributo possui argumentos nomeados que nos permitem informar o nome do usuário ou usuários, bem como o nome do grupo (*role*) ou grupos (*roles*) que estão autorizados a acessar o *controller* ou *action* associado ao atributo. A **Figura 13** ilustra algumas das possíveis variações do atributo **[Authorize]**.

```
15 namespace Tutorial.MVC.Controllers
16 {
17     [Authorize(Roles="Gerentes")]
18     [InitializeSimpleMembership]
19     public class ConsumidorController : Controller
20     {
21         CBCNatalContext natalContext = new CBCNatalContext();
22         int consumidorAtualID = 0;
23
24         [Authorize(Users="Alex", "Soler", "Antonio")]
25         private void GetConsumidorAtual()...
26
27         [Authorize(Roles="Gerentes, Supervisores", Users="Angelica, Ana")]
28         public ActionResult Index()...
29
30         [Authorize(Roles="Supervisores", Users="Bruno")]
31         public ActionResult Details(int id = 0)...
32
33         [Authorize(Users="Luis")]
34         public ActionResult Edit(int id = 0)...
35
36         //
37         // POST: /Consumidor/Edit/5
38     }
39 }
```

Figura 13: Variações do Atributo [Authorize]

Autenticação Integrada com Redes Sociais

Um cenário muito comum nas aplicações Web atualmente, é permitir que o usuário ou visitante do seu website possa autenticar-se utilizando as credenciais de uma rede social, onde ele já está cadastrado, evitando que ele tenha que memorizar novas senhas e proporcionando uma melhor experiência para o usuário da aplicação.

Integrar suas aplicações web com as **APIs** das redes sociais mais conhecidas, como **Facebook**, **Twitter**, **Microsoft Live** e **Google+**, apesar dos esforços destas empresas, nem sempre é uma tarefa simples. Em algumas situações, o desenvolvedor não está familiarizado com estes mecanismos e exigirá um tempo maior para analisar e entender como consumir estas **APIs**.

Para facilitar esta tarefa, a **Microsoft** integrou ao **.NET Framework 4.0** esses mecanismos de autenticação através do *Namespace* **Microsoft.Web.WebPages.OAuth**. Este recurso está disponível não apenas para aplicações **ASP.NET MVC**, mas também para todos os tipos de aplicações baseadas no **.NET Framework 4 (ou superior)** que precisam de autenticação integrada com estas redes sociais.

Como padrão, os *templates* de projetos **ASP.NET MVC** oferecem esta integração praticamente pronta. Por exemplo, se o desenvolvedor deseja oferecer aos usuários da sua aplicação uma alternativa para autenticar-se através do **Facebook**, basta acessar a área destinada aos

desenvolvedores, <https://developers.facebook.com/apps>, cadastrar sua aplicação, obter as chaves **appId** e **appSecret** e editar o arquivo **AuthConfig.cs** localizado na pasta **AppStart**. As linhas 25-27 apresentam o método **RegisterFacebookClient** que solicita estas duas chaves para integrar sua aplicação. Originalmente, estas linhas estão comentadas no arquivo **AuthConfig.cs**, é necessário remover os caracteres de comentários para concluir sua integração.

Simples assim, não é necessário nenhuma linha de código adicional, basta informar os atributos solicitados pelo método **RegisterFacebookClient** (ou o método correspondente à rede social desejada) e sua aplicação já estará pronta para permitir que seus usuários utilizem suas credenciais para acessar as áreas restritas. A **Figura 14** apresenta o arquivo **AuthConfig.cs**.

Figura 14: Autenticação Integrada com Redes Sociais

Conceitos Básicos

Até este ponto, você teve uma introdução ao modelo de desenvolvimento de uma aplicação **ASP.NET MVC**, entendeu o que são **Models**, **Views** e **Controllers**. Teve uma visão geral sobre o **Razor Engine** e o **Entity Framework**. Aprendeu como utilizar o **SimpleMembershipProvider** para autenticar e controlar o acesso dos usuários da sua aplicação, inclusive como integrar sua aplicação com as redes sociais mais conhecidas do mercado. Antes de prosseguirmos com aspectos mais avançados da implementação de uma aplicação **MVC** gostaria de abordar mais dois pontos básicos, porém não menos importantes, a inicialização das aplicações e as páginas de erros personalizadas.

Inicialização de Aplicações MVC

Existem algumas formas de inicializarmos as aplicações **MVC**. A mais simples e prática é utilizarmos o método **RegisterRoute** da classe **RouteConfig** encontrada na pasta **App_Start**. Este método utiliza a coleção **RouteCollection** para definir o padrão de rotas da aplicação.

O método **MapRoute** expõe o argumento nomeado **defaults** utilizado para informar qual será a ação (*action*) e o *controller* que serão invocados na inicialização da sua aplicação. A **Figura 15** ilustra o método **MapRoute**.

```
8 namespace TutorialMVC
9 {
10     public class RouteConfig
11     {
12         public static void RegisterRoutes(RouteCollection routes)
13         {
14             routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
15
16             routes.MapRoute(
17                 name: "Default",
18                 url: "{controller}/{action}/{id}",
19                 defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
20             );
21         }
22     }
23 }
24
```

Figura 15: Método MapRoute (RouteCollection)

No exemplo ilustrado na **Figura 15**, se o desenvolvedor desejasse que a aplicação exibisse o formulário de **Login**, implementado através da *View Login.cshtml* e gerenciado pela classe **AccountController** e o método (*action*) **Login**, os parâmetros sublinhados em amarelo deveriam ser substituídos como demonstrado a seguir:

```
...
defaults: new { controller = "Account", action = "Login", id =
UrlParameter.Optional }
...
```

Páginas de Erros Customizadas

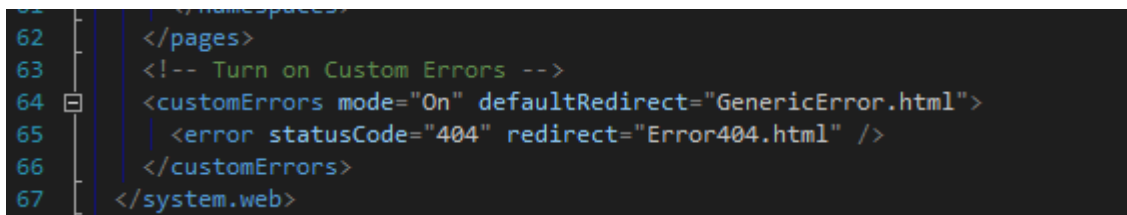
As aplicações web, em sua maioria, são utilizadas por um grande número de usuários. Cada usuário terá uma experiência diferente com sua aplicação. Um erro ou exceção gerado durante o processamento de um pedido do usuário não é algo que desejamos, mas estamos sempre sujeitos aos diversos fatores internos e externos que podem afetar o funcionamento da nossa aplicação.

Exibir páginas de erro personalizadas e com mensagens amigáveis para os usuários da sua aplicação é uma prática positiva que minimiza o impacto o erro gerado, proporcionando um melhor experiência para o usuário. Veja a seguir como adotar medidas simples que podem ajudar seu usuário e até você mesmo a identificar a causa do erro.

Erros gerados pelo HttpRequest

Os erros mais comuns são aqueles gerados pelo **HttpRequest**. Para cada erro gerado existe um **statusCode** correspondente. Por exemplo, o erro **404** indica que a página ou recurso solicitado não foi localizada no servidor, o erro **403** indica que o usuário não tem permissão para acessar o recurso que está tentando acessar, o erro **400** indica que um pedido (*request*) inválido foi enviado para o servidor. Para uma lista completa de erros gerados pelo **HttpRequest** acesse este endereço http://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

Para estes tipos de erros, o desenvolvedor continuará utilizando o elemento **customErrors** no arquivo de configuração da aplicação para definir o comportamento da aplicação para cada **statusCode** retornado pelo servidor. A seção **customErrors** possui o atributo **defaultRedirect** que deve ser utilizado para indicar a página para a qual se deseja redirecionar o *browser* quando um erro não listado nas propriedades **statusCode** da mesma seção ocorrer. A **Figura 16** apresenta o elemento **customErrors** no arquivo **Web.config**.



```
62 | </pages>
63 | <!-- Turn on Custom Errors -->
64 | <customErrors mode="On" defaultRedirect="GenericError.html">
65 |   <error statusCode="404" redirect="Error404.html" />
66 | </customErrors>
67 | </system.web>
```

Figura 16: O elemento **customErrors**

Erros gerados pelos Controllers, Actions e Classes

Além dos erros gerados pelo **HttpRequest**, temos que nos preocupar com os erros de processamento gerados pela nossa aplicação. Por exemplo, erros gerados durante a execução de um procedimento ou método de um dos *controllers* ou classes da nossa aplicação. Quando isto ocorre, o **HttpRequest** retornará para o *browser* o nosso velho conhecido erro **500 Internal Server Error**, que apresenta pouca ou nenhuma informação que possa ajudar o usuário e até mesmo você a identificar a causa do problema.

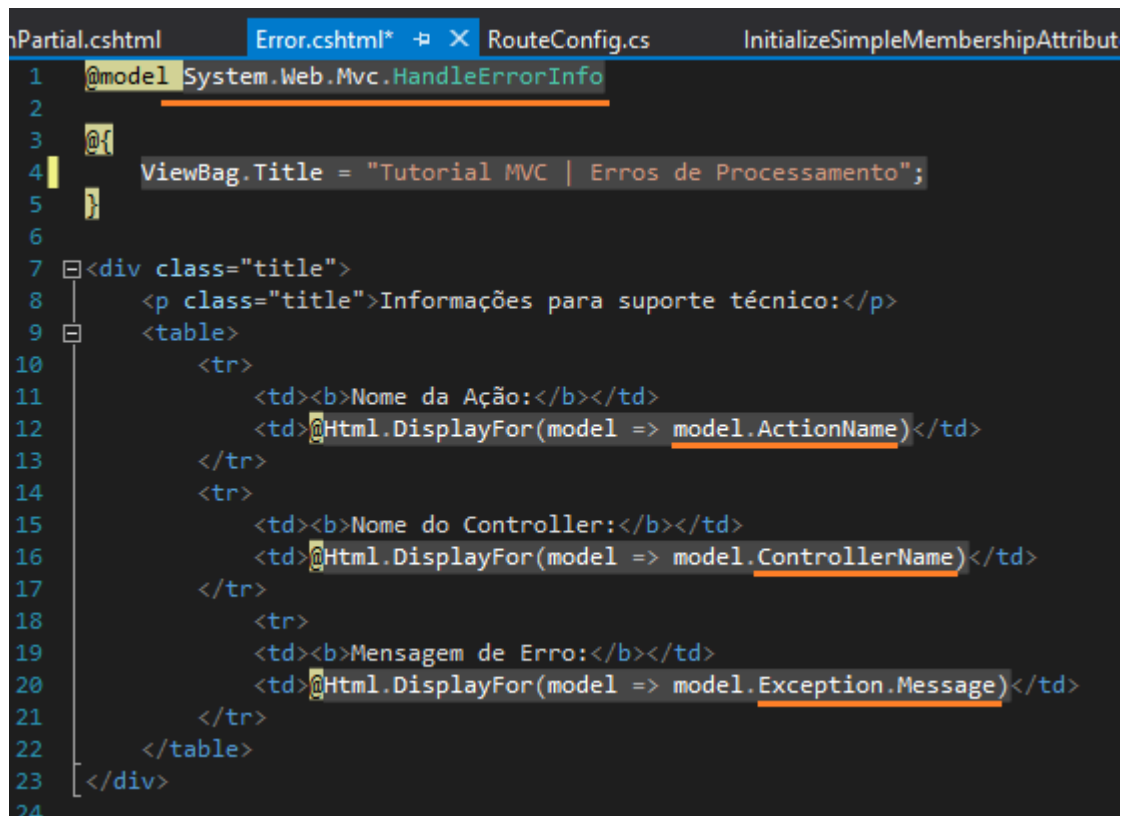
Para estes casos, o template de projeto **ASP.NET MVC Application** cria uma *View* específica, originalmente denominada **Error.cshtml**, encontrada na pasta **Views/Shared/**.

O elemento principal desta *View* é a classe **System.Web.Mvc.HandleErrorInfo** declarada no topo do arquivo. Esta classe expõe propriedades fundamentais para auxiliar o desenvolvedor a identificar a origem do erro ou exceção gerado.

O desenvolvedor deve personalizar esta *View* para torná-la adequada ao *layout* da aplicação, assim como para exibir mensagens mais amigáveis para seus usuários.

As informações expostas pelas propriedades da classe **HandleErrorInfo** podem ser exibidas nesta página de erro e/ou persistidas no *log* de erros da aplicação para análise futura.

A **Figura 17** apresenta a **View Error.cshtml** modificada para exibir as informações obtidas nas propriedades da classe **HandleErrorInfo** utilizando uma tabela **HTML**.



```
1 @model System.Web.Mvc.HandleErrorInfo
2
3
4 ViewBag.Title = "Tutorial MVC | Erros de Processamento";
5
6
7 <div class="title">
8     <p class="title">Informações para suporte técnico:</p>
9     <table>
10         <tr>
11             <td><b>Nome da Ação:</b></td>
12             <td>@Html.DisplayFor(model => model.ActionName)</td>
13         </tr>
14         <tr>
15             <td><b>Nome do Controller:</b></td>
16             <td>@Html.DisplayFor(model => model.ControllerName)</td>
17         </tr>
18         <tr>
19             <td><b>Mensagem de Erro:</b></td>
20             <td>@Html.DisplayFor(model => model.Exception.Message)</td>
21         </tr>
22     </table>
23 </div>
24
```

Figura 17: A View Error.cshtml

Arquitetura da nossa Aplicação MVC

Agora que você já tem uma boa compreensão dos conceitos e ferramentas básicas de uma aplicação **MVC**, vamos nos concentrar nos aspectos mais avançados da arquitetura desta aplicação.

Existem diversos modelos de arquitetura para implementarmos uma aplicação **MVC**. Basta uma rápida pesquisa na Internet para que o desenvolvedor encontre um grande número de diferentes “sabores” de arquitetura para aplicações **ASP.NET MVC**.

Este grande número de opções, frequentemente gera dúvidas para os desenvolvedores que estão buscando uma referência para escolher a arquitetura correta para seu projeto. A escolha correta é aquela que se encaixa ao cenário do seu projeto. Todos os modelos de arquitetura oferecem seus prós e contras. O desenvolvedor deve analisar seu projeto e optar pelo modelo que mais se aproxima das necessidades da sua empresa, do cliente, do time de desenvolvimento.

Para o nosso tutorial, optei pelo modelo que combina o uso de repositórios, **UnitOfWork** e **IoC**, também conhecido como **Model 2**, uma variação do padrão **MVC**. Este modelo oferece uma implementação relativamente simples e com resultados eficientes. Nos parágrafos seguintes explicarei como adotar este modelo para suas aplicações **MVC**.

O modelo Repository Pattern aplicado ao MVC

O uso de repositórios não é uma prática recente. A princípio o desenvolvedor pode ter a impressão de que terá que escrever uma quantidade maior de código, mas os benefícios desta prática ficarão claros ao longo do ciclo de vida do projeto.

O propósito do uso de repositórios é remover a dependência entre a camada de negócios da sua aplicação e a camada de acesso a dados. Isto permitirá que o você possa substituir a camada de acesso a dados sem afetar o funcionamento da sua aplicação. Por exemplo, o desenvolvedor pode criar falsos repositórios (*mocks*), utilizando coleções ao invés de persistir as informações diretamente no banco de dados, e utilizar estes falsos repositórios para testar sua aplicação sem se preocupar com o banco de dados.

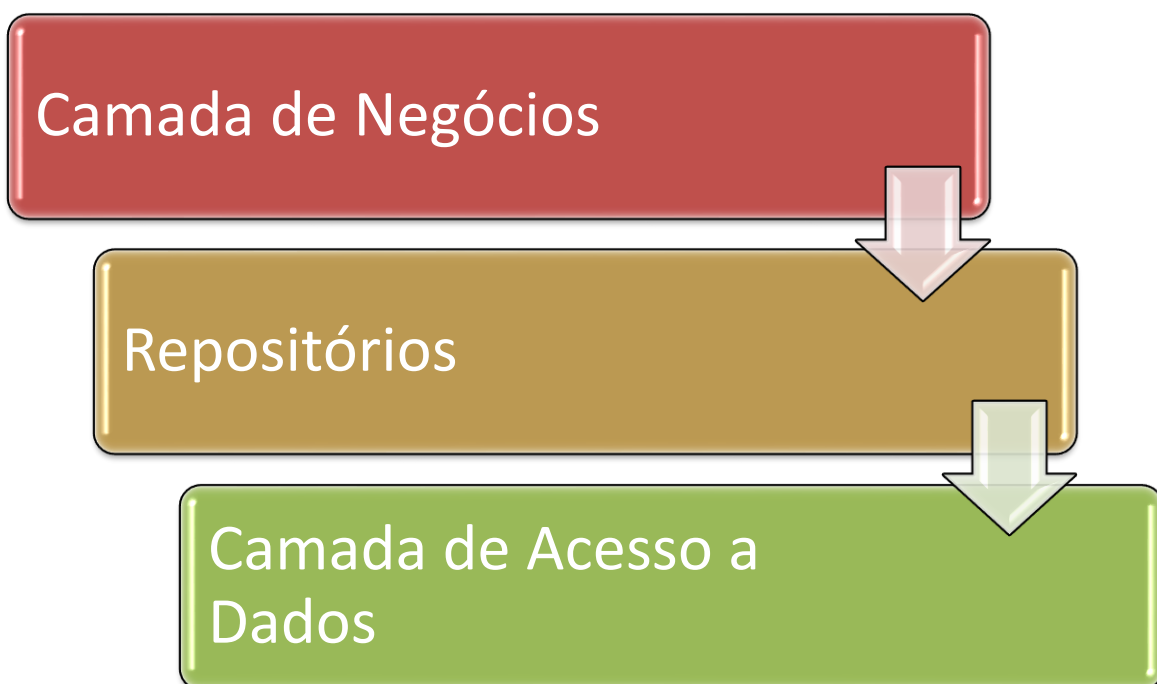


Figura 18: Repositórios para isolar as camadas de Negócio e Dados

Repositórios, Inversion of Control (IoC) e UnitOfWork na prática

Para ilustrar os conceitos acima, utilizaremos uma aplicação simples, chamada **Natter** (gíria do idioma Inglês sinônimo de **chatter**). Algo similar ao **Twitter**. Uma aplicação que nos permitirá demonstrar operações CRUD implementadas com repositórios, que tornarão nossos *Controllers* mais “limpos” e “enxutos” (prática que o desenvolvedor deve adotar para suas aplicações **MVC**).

Esta aplicação de exemplo, terá uma única página (*view*) que implementará as operações de **SELECT**, **INSERT** e **DELETE**. Não nos preocuparemos com design ou layout de interfaces, para nos concentrar na codificação desta aplicação. A **Figura 19** apresenta a interface da aplicação em tempo de execução.

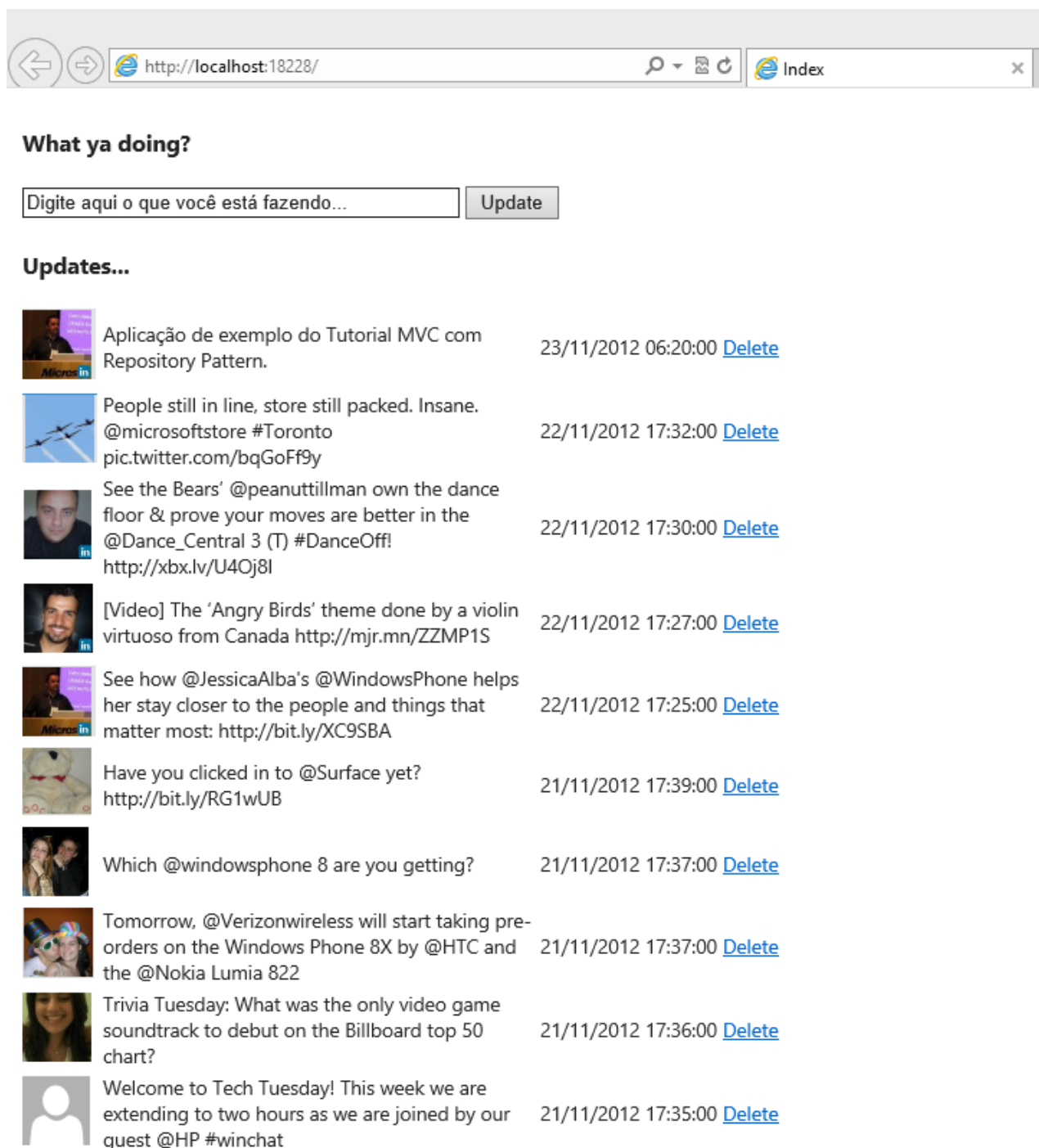


Figura 20: A aplicação de exemplo em tempo de execução.

O primeiro passo para reproduzir esta aplicação é criarmos um novo projeto baseado no template **ASP.NET MVC 4 Application**. Este *template* oferece algumas opções para o desenvolvedor, exibidas na janela **"New ASP.NET MVC 4 Project"**, utilize o modelo **"Internet Application"** e o *engine* **"Razor"** como mostra a **Figura 21**. Denomine o projeto como **NatterMvc**.

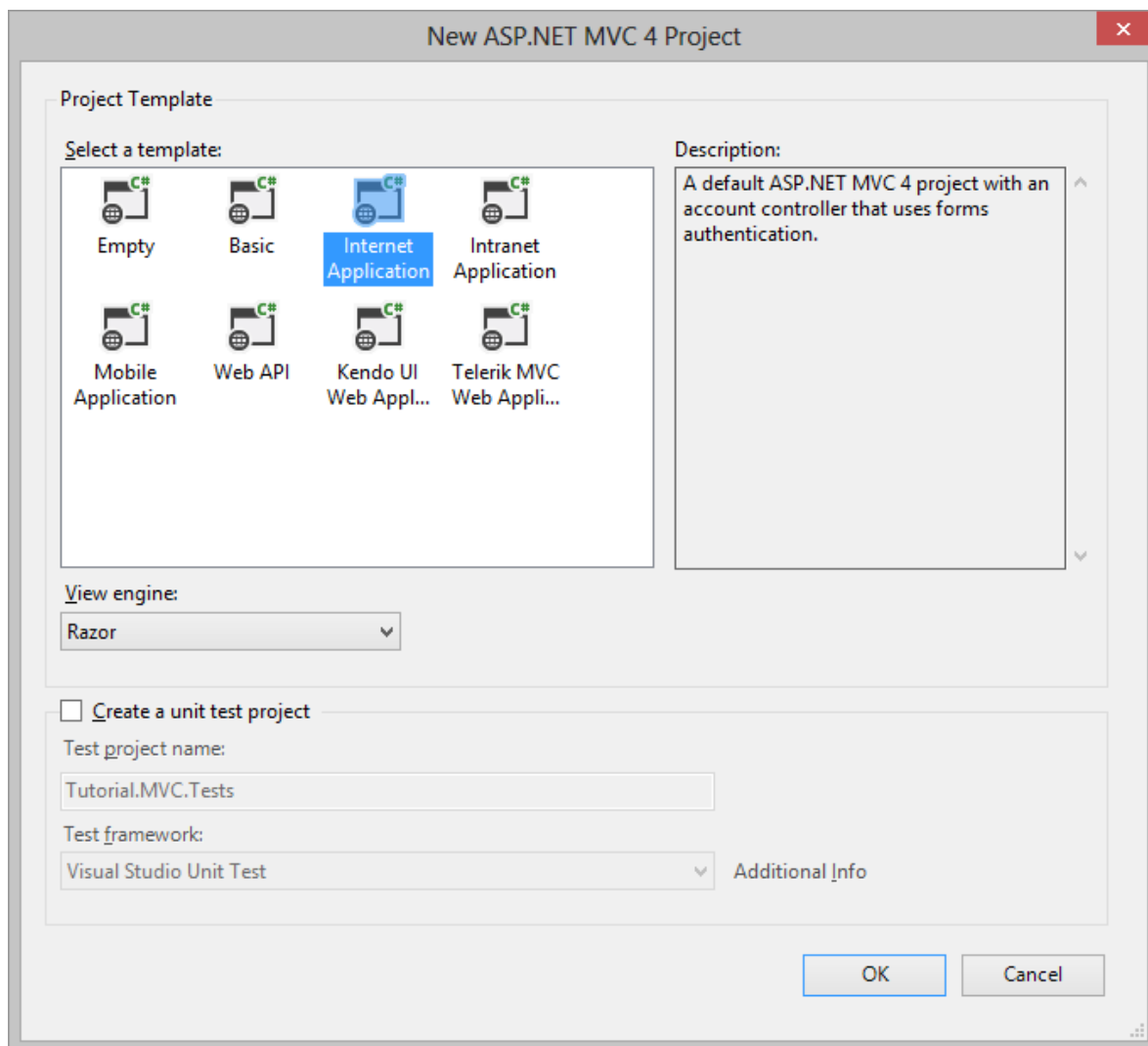


Figura 21: A janela New ASP.NET MVC 4 Project

A estrutura do banco de dados de exemplo

Para este exemplo, o desenvolvedor pode remover os *controllers* e *views* criadas pelo *template*, utilizaremos apenas a estrutura de pastas para codificar nossa aplicação. O primeiro passo é criarmos o banco de dados. A estrutura do nosso banco de dados será simples, apenas as tabelas de usuários (**User**), mensagens (**Nats**) e seguidores (**Followers**). Esta última utilizada para estabelecer um relacionamento de muitos-para-muitos, onde armazenaremos os códigos de usuários “seguidores” e “seguidos” (*followers* e *following*).

Uma vez definida a estrutura do banco de dados, crie um novo projeto do tipo **Class Library** que utilizaremos como nossa camada de acesso a dados. Denomine este projeto como **NatterModel**. Adicione uma referência para o **NatterModel** no projeto **NatterMvc**.

Adicione um novo item no projeto **NatterModel**, e selecione o item **Data -> ADO.NET Entity Data Model** para criar seu arquivo *.edmx. O **Entity Framework** se encarregará de criar as classes do nosso modelo. Veja a **Figura 22**.

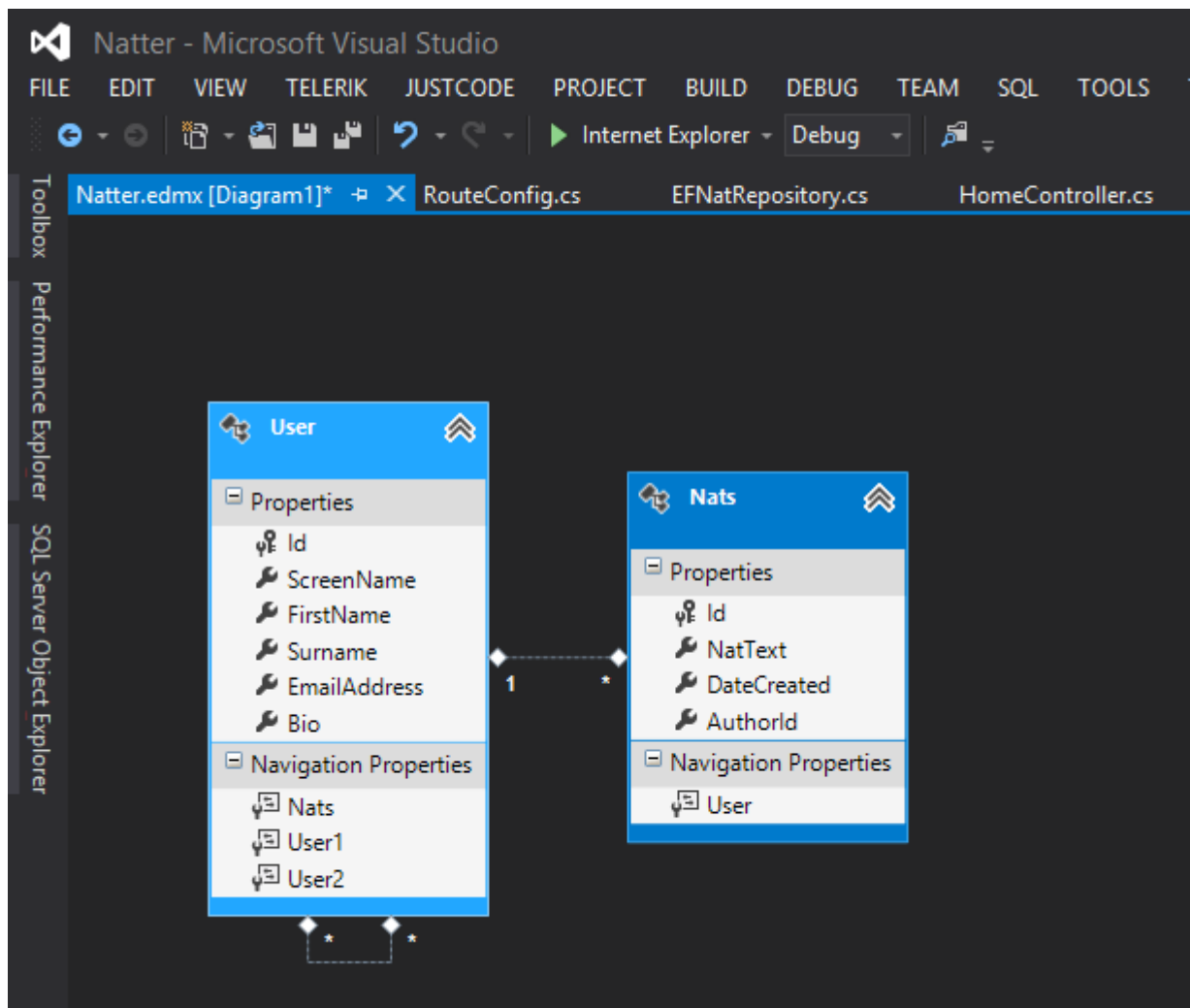
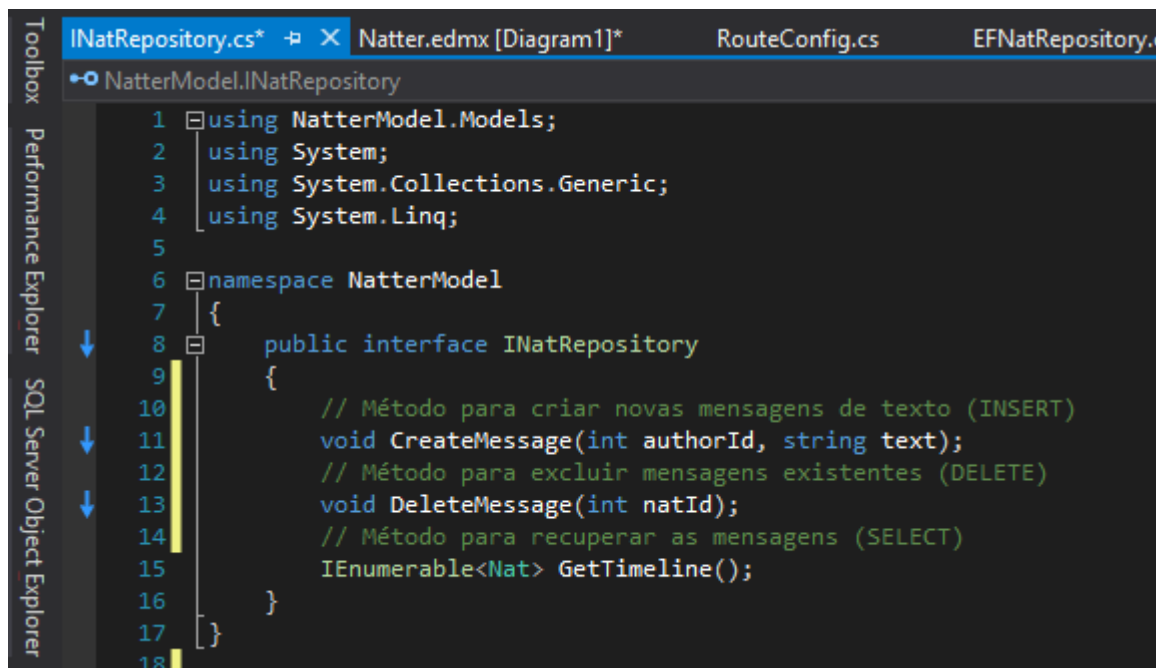


Figura 22: O diagrama do nosso Entity Data Model

Criando a Interface para nosso Repositório

O próximo passo é criarmos a Interface **INatRepository**. Esta Interface será simples, deverá conter três métodos: **CreateMessage(int authorId, string text)**, **DeleleMessage(int natId)** e **GetTimeLine()** usados para criar, excluir e recuperar respectivamente as mensagens de texto da nossa aplicação. Isto facilitará a codificação do nosso repositório. A **Figura 23** apresenta o código desta Interface.



```
1 using NatterModel.Models;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5
6 namespace NatterModel
7 {
8     public interface INatRepository
9     {
10         // Método para criar novas mensagens de texto (INSERT)
11         void CreateMessage(int authorId, string text);
12         // Método para excluir mensagens existentes (DELETE)
13         void DeleteMessage(int natId);
14         // Método para recuperar as mensagens (SELECT)
15         IEnumerable<Nat> GetTimeline();
16     }
17 }
18
```

Figura 23: A Interface INatRepository

Após codificar esta Interface, nosso projeto **NatterModel**, estará completo. Certifique-se referenciá-lo no projeto **NatterMvc** antes de seguirmos para o próximo passo.

Criando o mecanismo para IoC

Uma forma simples de ilustrarmos o conceito de **IoC** será demonstrada neste passo. No projeto **NatterMvc**, na pasta **Models**, crie uma classe chamada **Repositories**. Adicione nesta classe o código apresentado na **Figura 24**.

Esta classe permitirá ao desenvolvedor, substituir o repositório sempre que desejar, sem alterar o funcionamento da aplicação. Por exemplo, você pode utilizar repositórios falsos (*mocks*) para testar sua aplicação, ou ainda criar repositórios associados à outras fontes de dados. Em qualquer caso, basta instanciá-los na classe **Repositories** utilizando o método **Get** da propriedade **Nats**.

Esta é uma forma prática de implementarmos o conceito de **Inversion of Control (IoC)**. Esta prática facilita a manutenibilidade do código e a criação de testes unitários para validar suas regras de negócio.

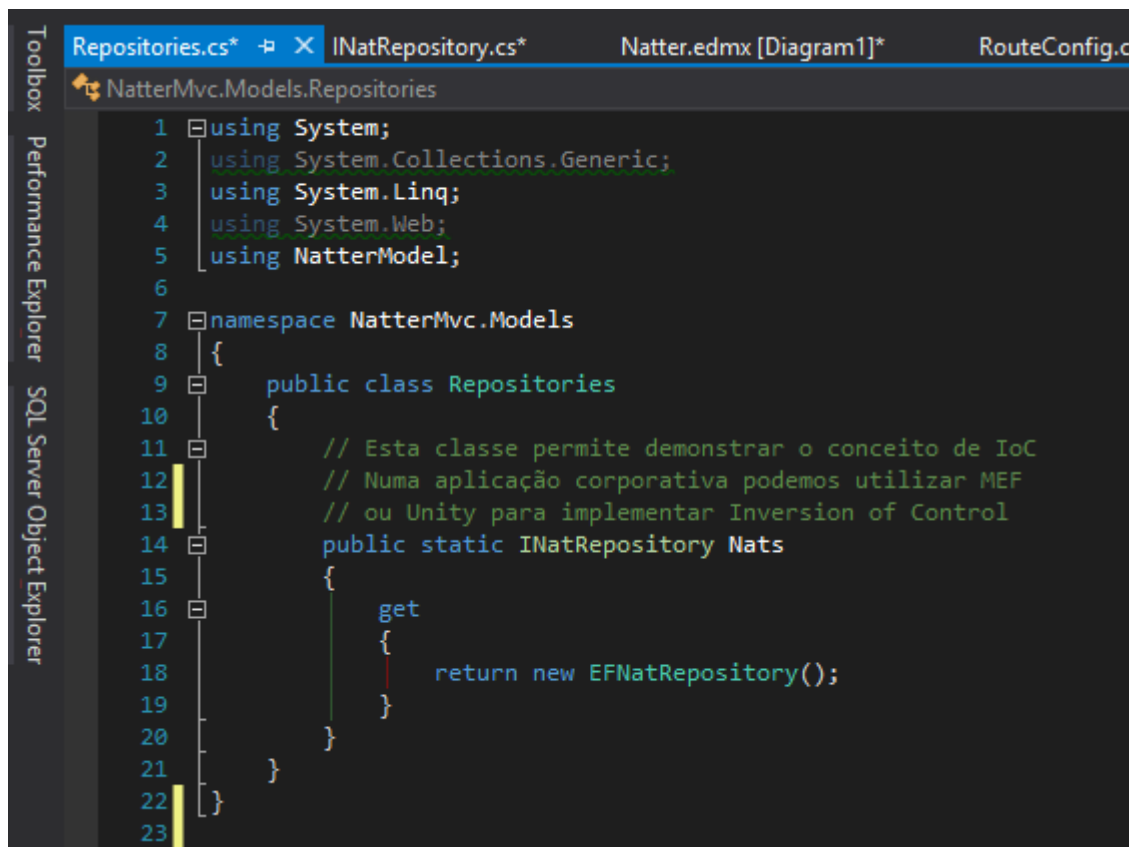


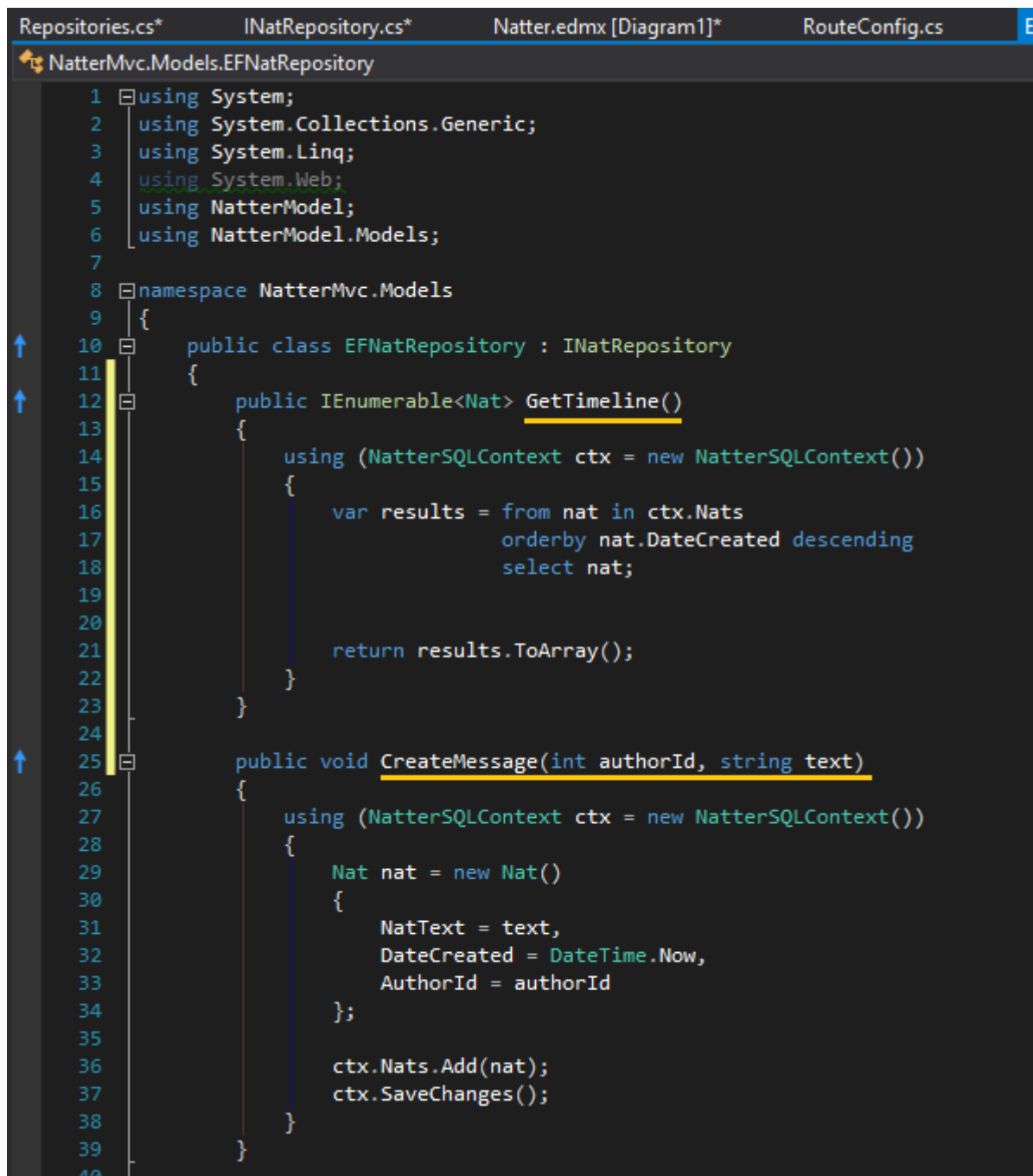
Figura 24: A classe Repositories

O Repositório EFNatRepository

Finalmente vamos implementar nosso repositório. Dentro da mesma pasta **Models**, crie uma nova classe denominada **EFNatRepository**. Este será nosso repositório. É nesta classe que o desenvolvedor implementará os métodos para interagir com a camada de acesso a dados.

Os dois primeiros métodos, apresentados na **Figura 25**, são **GetTimeLine()** e **CreateMessage()**. O primeiro é responsável por recuperar todas as mensagens gravadas no banco e exibi-las na linha do tempo da aplicação. O segundo método se encarrega de inserir novas mensagens no banco de dados. Ambos utilizam o objeto **DbContext** chamado **NatterSQLContext**.

Este objeto representa nosso banco de dados. As classes associadas ao contexto foram criadas automaticamente pelo **Entity Framework**. Dependendo do cenário da sua aplicação ou das práticas adotadas pela sua empresa, você pode mapear *stored procedures* do banco de dados para realizar estas tarefas utilizando o **Entity Framework**, ou mesmo criar um repositório que utilize o classes do **ADO.NET** para implementar os mesmos métodos. Lembre-se que seu repositório precisa apenas respeitar o contrato da Interface **INatRepository**, dito isto, você pode implementar os métodos usando **ADO.NET**, **Oracle ODP.NET**, **Entity Framework**, **Collections** ou o que você preferir para persistir os dados da sua aplicação. Este é o benefício de trabalharmos com repositórios.



```
Repositories.cs*    INatRepository.cs*    Natter.edmx [Diagram1]*    RouteConfig.cs    E
NatterMvc.Models.EFNatRepository
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using NatterModel;
6  using NatterModel.Models;
7
8  namespace NatterMvc.Models
9  {
10     public class EFNatRepository : INatRepository
11     {
12         public IEnumerable<Nat> GetTimeline()
13         {
14             using (NatterSQLContext ctx = new NatterSQLContext())
15             {
16                 var results = from nat in ctx.Nats
17                             orderby nat.DateCreated descending
18                             select nat;
19
20                 return results.ToArray();
21             }
22         }
23
24         public void CreateMessage(int authorId, string text)
25         {
26             using (NatterSQLContext ctx = new NatterSQLContext())
27             {
28                 Nat nat = new Nat()
29                 {
30                     NatText = text,
31                     DateCreated = DateTime.Now,
32                     AuthorId = authorId
33                 };
34
35                 ctx.Nats.Add(nat);
36                 ctx.SaveChanges();
37             }
38         }
39     }
40 }
```

Figura 25: Os métodos `GetTimeLine()` e `CreateMessage()`

Observe que cada método deste repositório utiliza um bloco *using* para executar suas tarefas. Este bloco *using*, envolvendo as ações, garante que todos os recursos utilizados pelo objeto **ctx** (**NatterSQLContext**) serão liberados imediatamente após concluir seu objetivo.

Uma particularidade do método **GetTimeLine()** refere-se a chamada ao método auxiliar **ToArrayList()** para retornar o resultado da consulta. Isto é necessário, porque as consultas **LINQ** são objetos do tipo *lazy evaluated*, o que significa que esses objetos representam apenas estruturas de consultas até que um método como **ToArrayList()**, **ToArrayList()** ou **ToDictionary()** seja invocado para que a consulta seja efetivamente executada contra o banco e os registros recuperados.

Otimizando LINQ Queries

O método **DeleteMessage()** demonstra uma forma de otimização para evitarmos *roundtrips* desnecessárias até o banco de dados. A **Figura 26** que apresenta a primeira versão do método **DeleteMessage()**. Observe que a linha 46 executa uma consulta ao banco de dados para recuperar o objeto **Nat** que será marcado para exclusão com o método **Remove** do **DbContext**. Isto significa que antes de executar a exclusão deste registro, nossa aplicação literalmente vai até o banco de dados para recuperar o objeto que corresponde ao Id que se deseja excluir. Este *roundtrip* pode ser evitado.

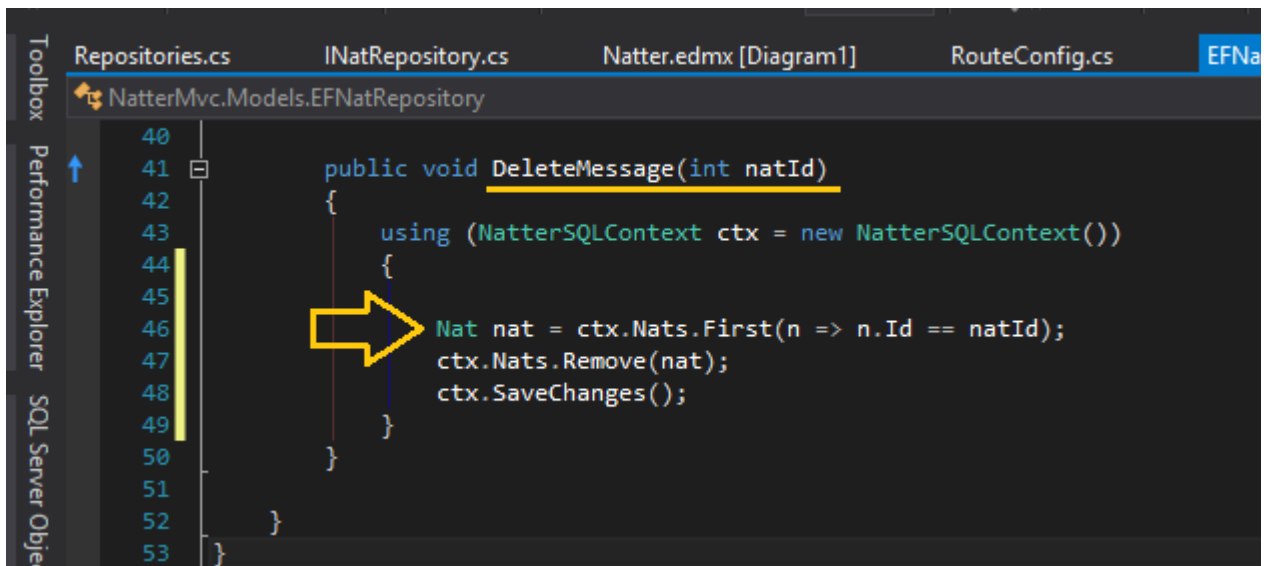


Figura 26: O método DeleteMessage() com um roundtrip desnecessário

Observe agora uma implementação diferente para o mesmo método. O uso do método **Attach** do objeto **DbContext** é um recurso muito útil para este cenário, e ele evita que a aplicação vá até o banco para recuperar o objeto que deseja-se excluir. O próprio método **Attach** se encarrega de marcar o objeto correto para exclusão.

A construção é simples, mas cabe aqui uma explicação. Observe com atenção que o método **DeleteMessage()**, desta vez, espera dois argumentos **natId** e **authorId**, ambos números inteiros que correspondem ao código da mensagem de texto e código do autor desta mensagem. Mas, o desenvolvedor pode estar perguntando por que vamos passar o código do autor se o já temos o Id da mensagem?

O que acontece aqui, é que o **EntityFramework** trata os relacionamentos como objetos de primeira classe, ou seja, para você excluir um registro de uma entidade, você deve conhecer as *chaves-estrangeiras* desta entidade.

Quando utilizamos a construção anterior (**Figura 26**), a aplicação vai até o banco e recupera um objeto **Nat**, que já contém este relacionamento. Para evitarmos esta viagem até o banco, com o objetivo de melhorar a performance da nossa aplicação, precisamos informar corretamente o objeto que desejamos excluir. Por este motivo é que o código do autor da mensagem deve ser incluído no método **DeleteMessage()**. Confira o código na **Figura 27**.

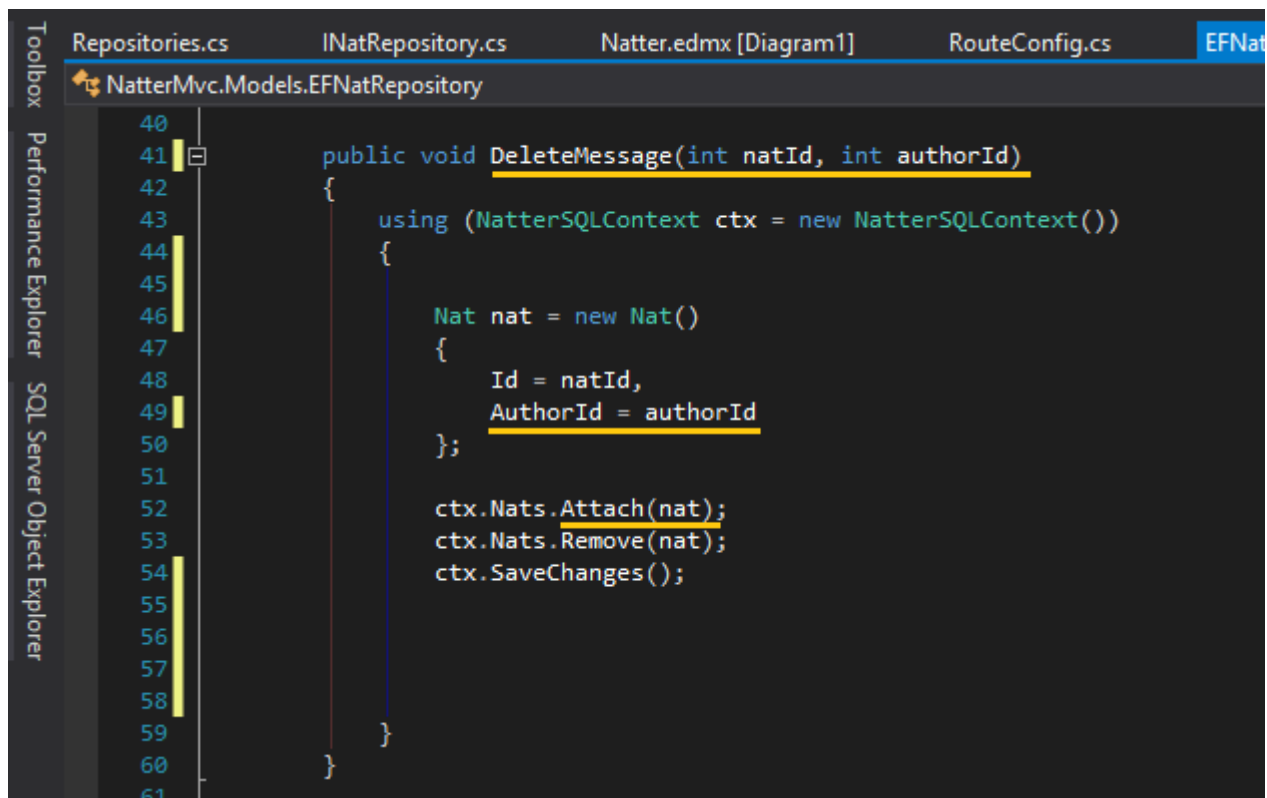


Figura 27: O método DeleteMessage() utilizando DbContext.Attach()

Implementando UnitOfWork

O conceito de **UnitOfWork** está baseado na capacidade da aplicação de isolar ações específicas, como demonstrado nos métodos do nosso repositório acima. Quando utilizamos os blocos **Using** para delimitar o processamento da ação, estamos na verdade criando **UnitsOfWork**.

Neste modelo, o desenvolvedor pode criar várias **UnitsOfWork** (ações), processá-las isoladamente, e apenas submetê-las contra o banco de dados quando desejado. Pense neste modelo de **UnitsOfWork** como uma transação.

Codificando nosso HomeController

Agora que já implementamos nosso repositório, interfaces e modelos, precisamos nos preocupar com os objetos **HomeController** e a *View* correspondente. Para implementar nosso *Controller*, vá para a pasta **Controllers**, clique com o botão direito do mouse sobre a pasta. No menu de contexto, selecione a opção **Add -> Controller**. E escolha o template **Empty MVC controller**, como mostra a **Figura 28**.

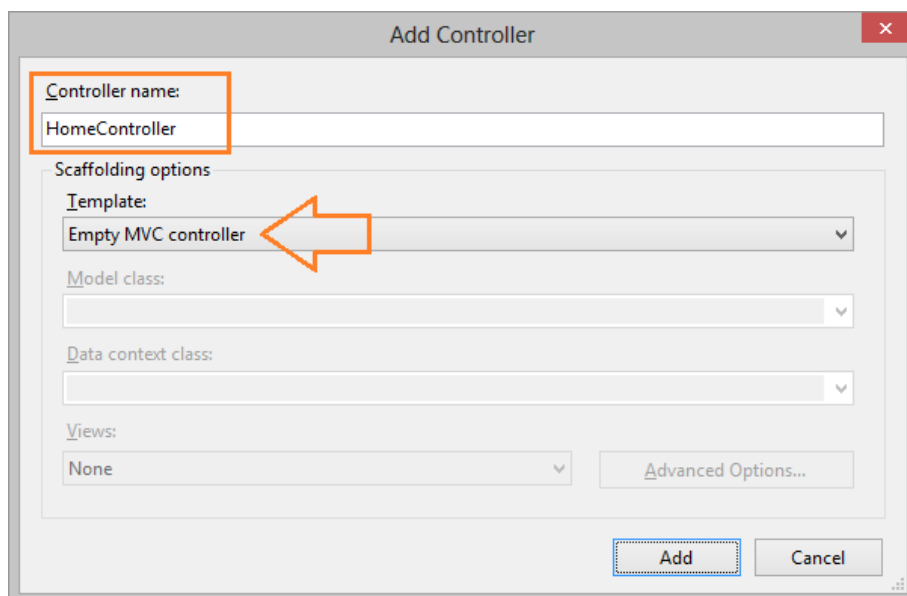
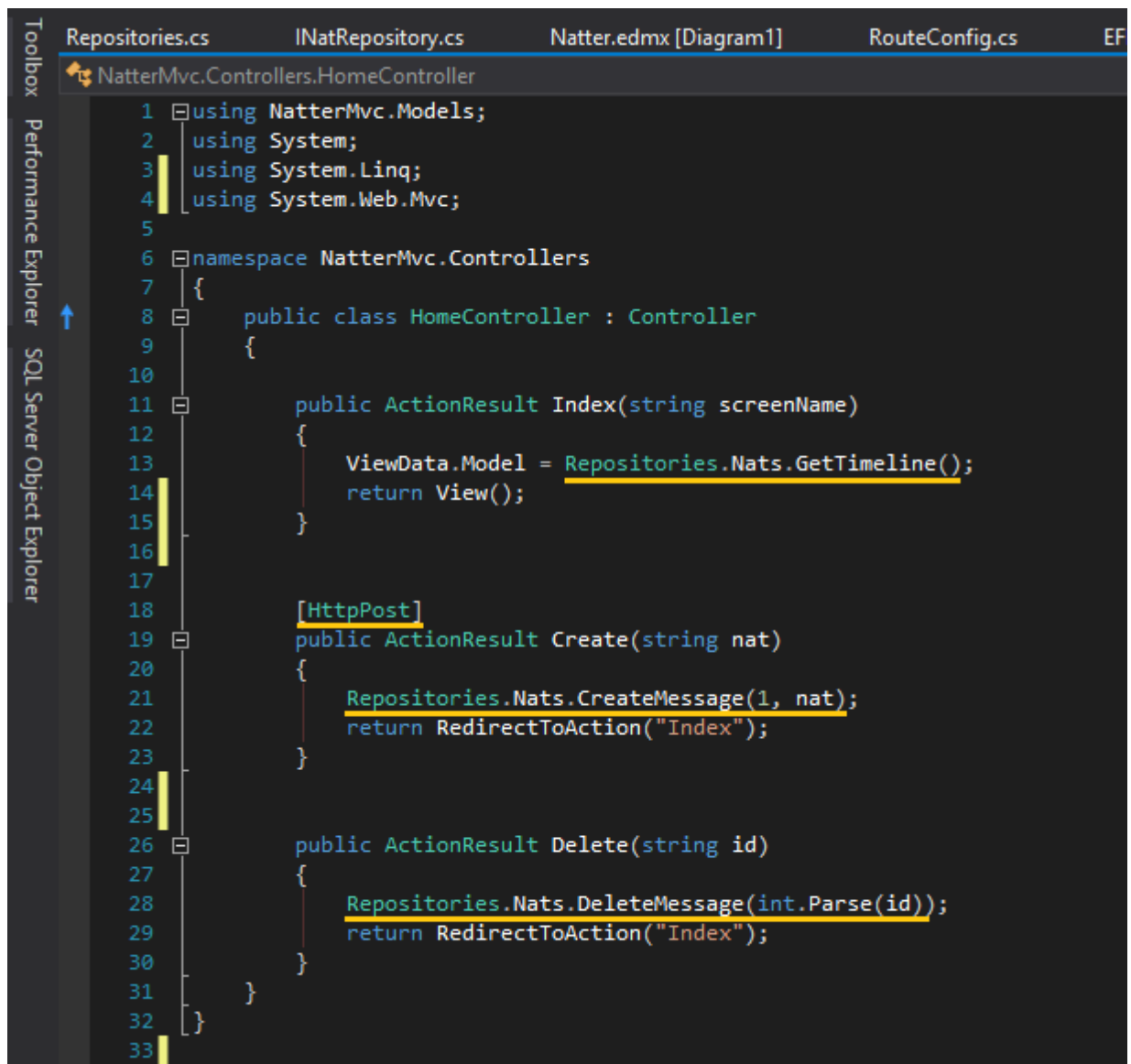


Figura 28: A janela Add Controller

Modifique o código deste *controller* para que fique exatamente como apresentado na **Figura 29**. Observe como o código do nosso *controller* ficou limpo, temos apenas as chamadas para os métodos do nosso repositório. A classe **Repositories** é responsável por direcionar as chamadas para o repositório desejado. Nosso *controller* tem apenas três ações: **Index** (utilizada para invocar o método **GetTimeLine()** e exibir as mensagens na página), **Create** que utiliza o método **CreateMessage()** para inserir um novo registro de mensagem no banco de dados e a ação **Delete** que dispara o método **DeleteMessage()** do nosso repositório para excluir uma mensagem já cadastrada.

Observe que o método **Create()** possui um atributo **[HttpPost]** associado a ele. Isto é necessário para que este método aceite *posts* enviados pela *View*. Este mesmo atributo poderia ser escrito utilizando a notação **[AcceptVerbs(HttpVerbs.Post)]**, sem alteração em seu comportamento.



```
1 using NatterMvc.Models;
2 using System;
3 using System.Linq;
4 using System.Web.Mvc;
5
6 namespace NatterMvc.Controllers
7 {
8     public class HomeController : Controller
9     {
10
11         public ActionResult Index(string screenName)
12         {
13             ViewData.Model = Repositories.Nats.GetTimeline();
14             return View();
15         }
16
17         [HttpPost]
18         public ActionResult Create(string nat)
19         {
20             Repositories.Nats.CreateMessage(1, nat);
21             return RedirectToAction("Index");
22         }
23
24         public ActionResult Delete(string id)
25         {
26             Repositories.Nats.DeleteMessage(int.Parse(id));
27             return RedirectToAction("Index");
28         }
29     }
30 }
31
32
33
```

Figura 29: A classe HomeController

A View Home

E como última peça da nossa aplicação de exemplo, temos a **View/Home**. Para criar esta *View*, vá para a pasta **Views**, crie uma pasta chamada **Home** (se não existir). Clique com o botão direito do mouse sobre a pasta **Home**, e no menu de contexto, selecione **Add -> View**.

Na janela que será exibida informe o nome da *view* como **Index**, certifique-se que o *View engine* está definido como **Razor (CSHTML)**, marque a opção **Create a strongly-typed view** e selecione a classe **Nat (NatterModel.Models)** para a propriedade **Model Class**. A janela **Add View**, permite ao desenvolvedor escolher entre os *scaffold templates* **Create**, **Delete**, **Details**, **Edit**, **Empty** e **List**. Cada um oferece um *layout* diferente para a *view*. O objetivo aqui é apenas de agilizar a tarefa de construção da *view*.

Contudo, numa aplicação real, muitas vezes, o desenvolvedor criará uma *view* vazia (*Empty template*) para adequar seu *layout* ao padrão criado por designers e aprovado pelo usuário.

Add View

View name:
Index

View engine:
Razor (CSHTML)

☒ Create a strongly-typed view

Model class:
Nat (NatterModel.Models)

Scaffold template:
List

☒ Reference script libraries

☐ Create as a partial view

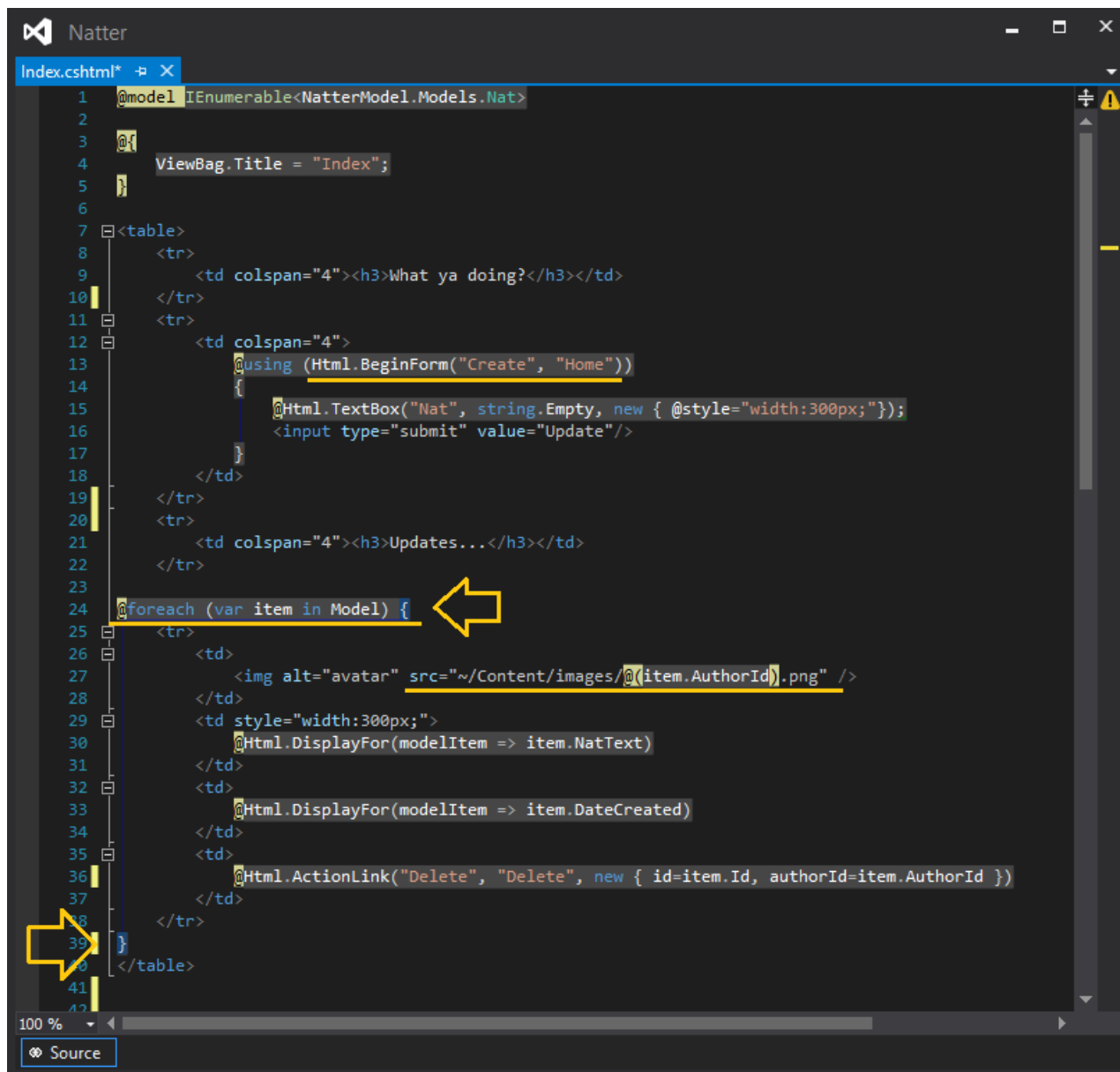
☒ Use a layout or master page:
...
(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:
MainContent

Add Cancel

Figura 30: A janela Add View

Uma vez criada a *View* **Home/Index**, modifique seu conteúdo para que fique exatamente como apresentado na **Figura 31**.



```
1 @model IEnumerable<NatterModel.Models.Nat>
2
3 @{
4     ViewBag.Title = "Index";
5 }
6
7 <table>
8     <tr>
9         <td colspan="4"><h3>What ya doing?</h3></td>
10    </tr>
11    <tr>
12        <td colspan="4">
13            @using (Html.BeginForm("Create", "Home"))
14            {
15                @Html.TextBox("Nat", string.Empty, new { @style="width:300px;" });
16                <input type="submit" value="Update"/>
17            }
18        </td>
19    </tr>
20    <tr>
21        <td colspan="4"><h3>Updates...</h3></td>
22    </tr>
23    @foreach (var item in Model) {
24        <tr>
25            <td>
26                
27            </td>
28            <td style="width:300px;">
29                @Html.DisplayFor(modelItem => item.NatText)
30            </td>
31            <td>
32                @Html.DisplayFor(modelItem => item.DateCreated)
33            </td>
34            <td>
35                @Html.ActionLink("Delete", "Delete", new { id=item.Id, authorId=item.AuthorId })
36            </td>
37        </tr>
38    }
39 </table>
40
41
42
```

Figura 31: A View Home/Index

Analisando o código apresentado na **Figura 31**, o desenvolvedor poderá observar a principal vantagem do **Razor engine** – a sintaxe. Simples, limpa e fácil de usar. O caracter **@** indica o início de um bloco de processamento no servidor, e você pode combinar código **HTML** com código processado no servidor, como mostra a **linha 27** onde o nome da imagem é formado pelo **Id** do **Autor** concatenado com o *path* da pasta e a extensão (***.png**) do arquivo. Outro exemplo é a própria tabela criada para apresentar as mensagens de texto. O conteúdo do elemento **<table>** está envolvido pelo bloco **@foreach** que inicia na **linha 24** e termina na **linha 39**. Mais um exemplo nesta *view* é o formulário criado com o método **Html.BeginForm** na **linha 13** que combina um controle **TextBox** gerado pelo objeto **Html.TextBox** com um botão criado com o código **HTML <input type="submit">**. O método **Html.BeginForm** requer dois parâmetros: Ação (*Action*) e *Controller*. Ao submeter o formulário, a aplicação enviará os dados para o *controller*/ação informados nos parâmetros. O *controller* por sua vez, processará o *request*.

Finalizando a aplicação de exemplo

Após concluir a **View Home/Index**, nossa aplicação de exemplo estará pronta para ser executada. Sua aparência deve ser semelhante a **Figura 20** deste tutorial. Neste exemplo simples, apresentamos o funcionamento de uma aplicação **MVC**, aplicamos os conceitos baseados no **Model 2** (uma variação do padrão **MVC**), incluindo o uso de repositórios, **Inversion of Control (IoC)** e **UnitOfWork**.

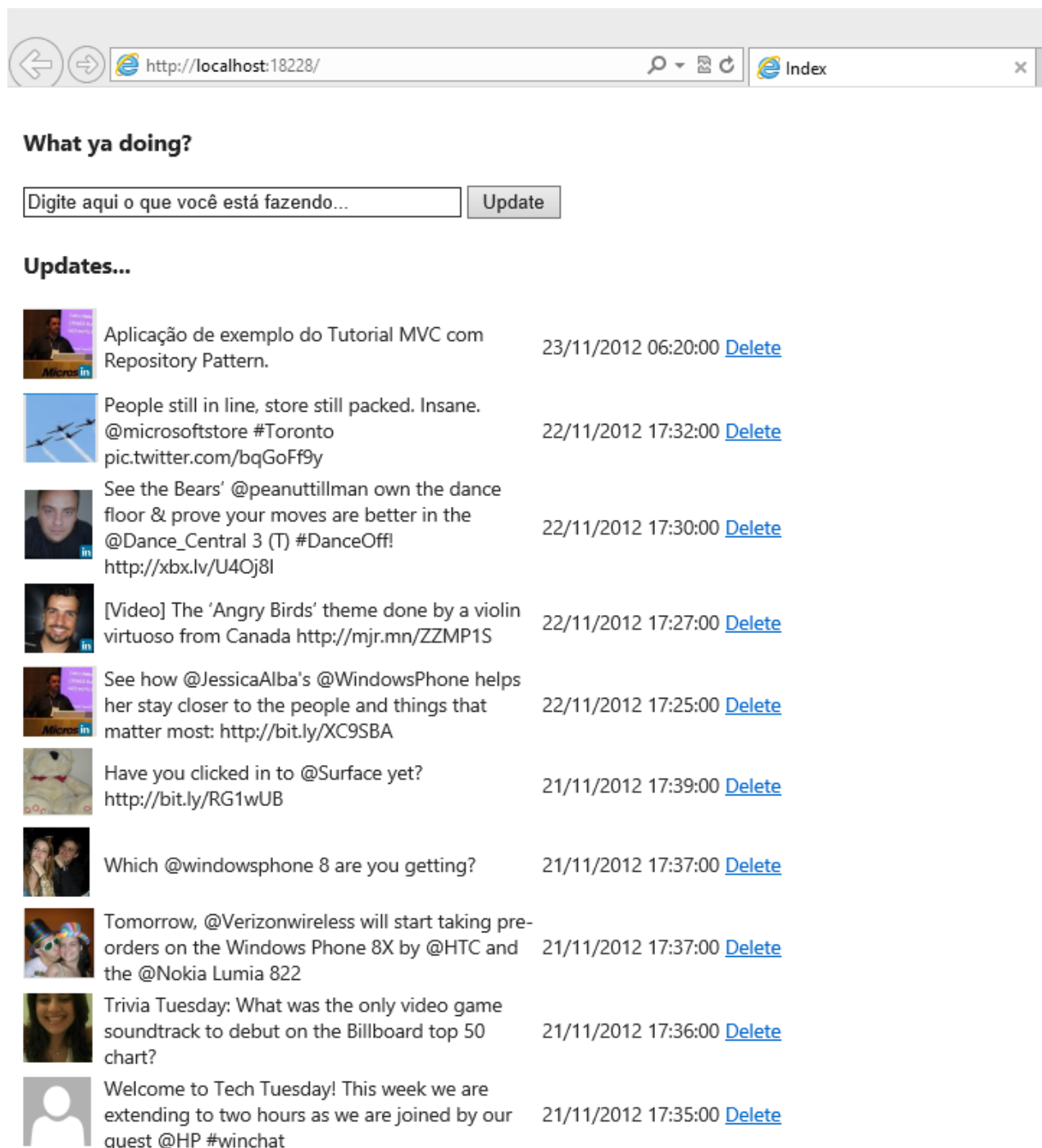


Figura 20: A aplicação de exemplo em tempo de execução.

Práticas comuns (e recomendadas) para aplicações MVC

Nos próximos parágrafos, vamos demonstrar algumas das práticas mais comuns nos cenários de aplicações web, e como elas se diferenciam quando adotamos o padrão **MVC**. Por exemplo, a forma como adicionamos um controle do tipo **DropDownList** numa aplicação **MVC** é diferente da forma utilizada quando trabalhamos com **ASP.NET WebForms**.

São tarefas comuns, que utilizamos com frequência no dia-a-dia, como validação de entrada de dados, paginação de registros, controles do tipo lista em cascata, exibir notificações para o usuário, que se diferenciam no modelo **MVC** e é preciso entendê-las antes de implementá-las em sua aplicação.

Validação com DataAnnotation Classes

No tópico **Regionalização das Strings e Mensagens do SimpleMembershipProvider**, na **página 13** deste tutorial, você já teve uma introdução ao mecanismo de validação disponível através do *namespace* **System.ComponentModel.DataAnnotations**. Este recurso gera automaticamente a regra de validação na UI, ou seja, você não precisa se preocupar com *client scripts* para este tipo de validação, o *framework* se encarrega disso. A seguir, vamos abordar alguns aspectos específicos deste recurso.

Validando Tipos de Dados específicos

Uma forma simples de validar tipos de dados específicos é utilizarmos o atributo de validação **[DataType]**, que nos permite definir dois argumentos nomeados, o tipo de dados (propriedade *DataType*) que o campo espera receber, e a mensagem de erro (*ErrorMessage*) que será exibida para o usuário caso ele informe um tipo inválido para este campo. Os tipos de dados disponíveis neste atributo estão apresentados na **Tabela 3**.

Atributo	Tipos de Dados
<i>DataType</i>	<i>Currency</i> <i>Custom</i> <i>Date</i> <i>DateTime</i> <i>Duration</i> <i>EmailAddress</i> <i>Html</i> <i>ImageUrl</i> <i>MultilineText</i> <i>Password</i> <i>PhoneNumber</i> <i>Text</i> <i>Time</i> <i>Url</i>

Tabela 3: Tipos de Dados do Atributo de Validação [DataType]

Obviamente, os formatos para Data e Número de Telefone estão, por padrão, no formato americano e devem ser customizados para atender nossas necessidades. A **Figura 32** mostra o atributo **DataType** sendo utilizado para a propriedade **Email** de uma classe.


```

63
64     [Required(ErrorMessage="O campo E-mail é obrigatório.")]
65     [Display(Name="E-mail",Description="E-mail(s) do Consumidor.")]
66     [DataType(DataType.EmailAddress,ErrorMessage="E-mail informado é inválido.")]
67     public string Email { get; set; }
68

```

Figura 32: O Atributo de Validação **DataType**

Validando Campos Requeridos

A validação para campos requeridos é simples. Basta adicionarmos o atributo de validação **[Required]** sobre a propriedade que desejamos tornar obrigatória. Assim como todos os atributos de validação, o desenvolvedor pode utilizar o argumento nomeado **ErrorMessage** para definir a mensagem que será exibida para o usuário quando esta regra não for atendida. A **Figura 32**, na **linha 64** temos o exemplo do atributo de validação **[Required]**.

Validação com RegularExpressions

Também é possível implementar a validação utilizando **RegularExpressions**. A **Figura 33** apresenta um exemplo de validação da mesma propriedade **Email** utilizando uma **RegularExpression**.

```

[Required(AllowEmptyStrings=false, ErrorMessage="O e-mail do amigo é obrigatório.")]
[Display(Name="E-mail",Description="E-mail(s) do Amigo.")]
[RegularExpression(@"^[a-zA-Z0-9_\-\.]+@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,3}$",ErrorMessage="O e-mail informado é inválido.")]
public string Email { get; set; }

```

Figura 33: Validação com Regular Expressions

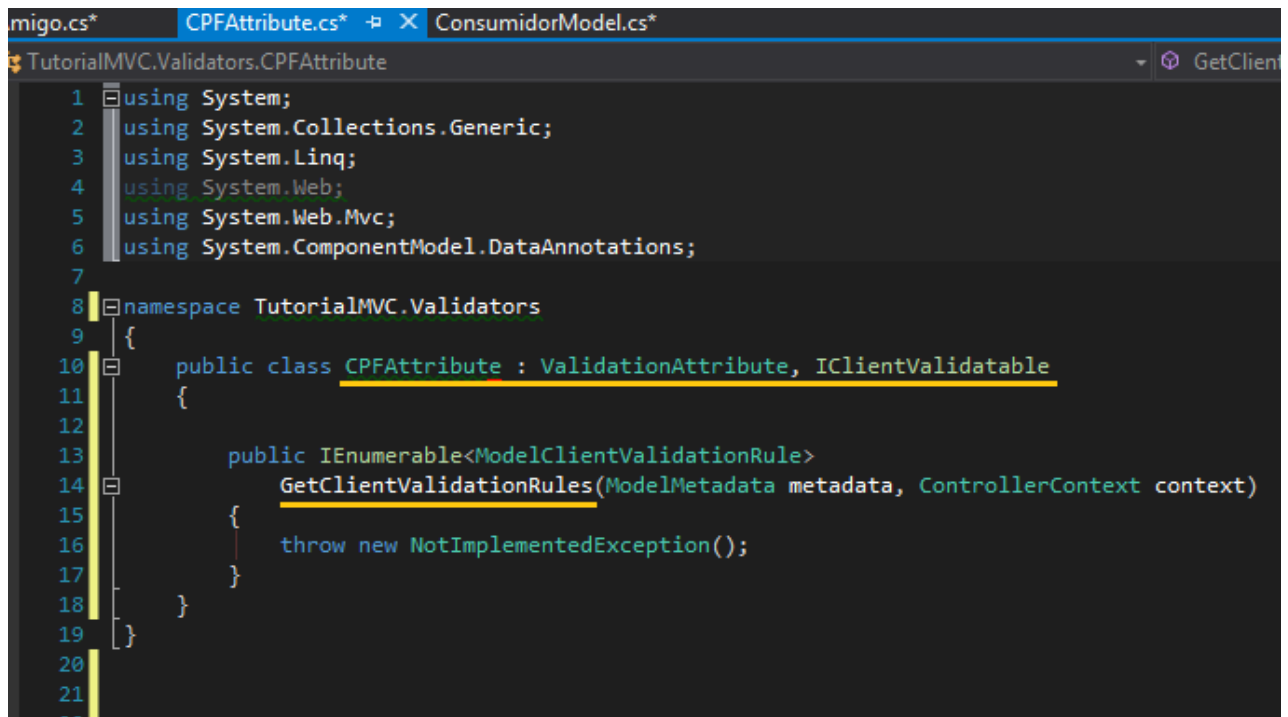
O atributo **[Display]**

Este é sem dúvida o atributo mais utilizado. O atributo **[Display]** oferece dois argumentos nomeados, *Name* e *Description*. O primeiro define o texto que será exibido como *label* do controle associado à propriedade ao qual o atributo está vinculado. Por exemplo, para uma propriedade denominada **AuthorId**, o desenvolvedor pode definir o argumento *Name* como “Código do Autor”, quando o engine do **MVC** processar a *View* correspondente, o texto exibido ao lado do controle associado à propriedade **AuthorId** será “Código do Autor”. O segundo argumento nomeado, *Description*, pode ser utilizado como complemento, a informação colocada neste argumento será exibida como **ToolTip** no controle associado à propriedade.

Custom ValidationAttributes

Não podemos esquecer da validação de propriedades como **CNPJ** e **CPF**. Para casos como estes, podemos implementar atributos de validação customizados. Os validadores customizados devem herdar a classe **ValidationAttribute** e o contrato da Interface **IClientValidatable**. Esta Interface requer o método **GetClientValidationRules**. Uma informação importante, refere-se ao nome do atributo. Os atributos de validação deverão sempre apresentar o sufixo **Attribute** imediatamente após sua denominação. Por exemplo, o atributo **CPF** deverá chamar-se **CPFAttribute**, para **CNPJ** o mesmo **CNPJAttribute**. Por outro lado, a notação com **DataAnnotationClasses** suprime o sufixo **Attribute**, permitindo que o desenvolvedor adicione seu atributo apenas com o **[CPF]** ou **[CNPJ]**.

A **Figura 34** apresenta a estrutura para o atributo de validação de **CPF**. A classe completa deste validador, pode ser encontrada nos arquivos do projeto de exemplo que estão disponíveis junto com este tutorial.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Web;
5 using System.Web.Mvc;
6 using System.ComponentModel.DataAnnotations;
7
8 namespace TutorialMVC.Validators
9 {
10     public class CPFAttribute : ValidationAttribute, IClientValidatable
11     {
12
13         public IEnumerable<ModelClientValidationRule>
14             GetClientValidationRules(ModelMetadata metadata, ControllerContext context)
15         {
16             throw new NotImplementedException();
17         }
18     }
19 }
```

Figura 34: Atributo de Validação para CPF (apenas estrutura)

Máscaras de entrada de dados

Outra necessidade comum no desenvolvimento de aplicações é o uso de máscaras para entrada de dados. Neste exemplo não estamos utilizando nenhum componente de terceiros, como **Telerik**, **Infragistics**, **DevXpress**, bastante conhecidos no mercado. Esses componentes, em sua maioria, oferecem vários recursos que facilitam a vida do desenvolvedor.

Desta forma, precisamos recorrer a outros mecanismos que proporcionem resultados similares. Uma prática comum, neste caso, é o uso de scripts **JQuery** para aplicar as máscaras de entrada.

Sem muito esforço, o desenvolvedor pode concluir esta tarefa. A **Figura 35** apresenta o código **JQuery** utilizado para aplicar máscaras de Telefone, CPF, Data e CEP. Para que este *script* funcione corretamente, o desenvolvedor deve certificar-se de ter adicionado o arquivo **jquery.maskedinput-1.3.min.js** ao projeto.

```

75
76     var $meujquery = jQuery.noConflict();
77     $meujquery(document).ready(function () {
78
79         $meujquery("#Telefone").mask("(99) 9999-9999?9");
80         $meujquery("#Celular").mask("(99) 9999-9999?9");
81         $meujquery("#CPF").mask("99999999999");
82         $meujquery("#CEP").mask("99999-999");
83         $meujquery("#DataNascimento").mask("99/99/9999");
84
85     });
86

```

Figura 35: Função JQuery para aplicar máscaras de entrada de dados

Exibindo notificações para o usuário

Manter seu usuário informado sobre o processamento da sua aplicação é fundamental. Para não estendermos este tutorial demasiadamente, gostaria de compartilhar o modelo de notificação que tenho adotado para aplicações **MVC**. Você pode testar este modelo, na prática, acessando este endereço: <http://demo.taiga.nl/notification/>. Este modelo é de autoria do Holandês **Martijn Boland**, e está disponível para download no **GitHub**. Para maiores informações, acesse o este endereço: <http://blogs.taiga.nl/martijn/2011/05/03/keep-your-users-informed-with-asp-net-mvc/>.

Trabalhando com Controles de Lista

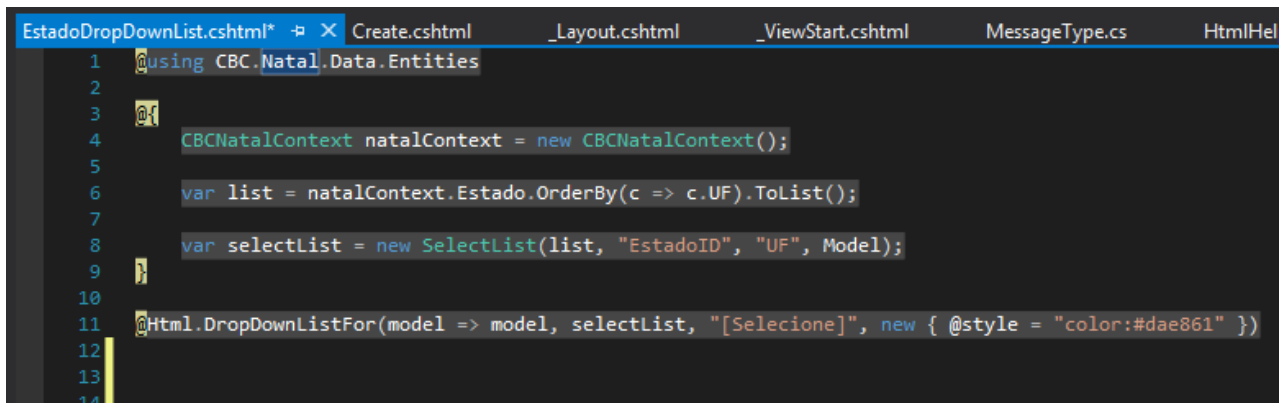
Carregar controles do tipo lista, como **DropDownLists** ou **ListBoxes** é uma tarefa simples e comum, mas também é diferente para as aplicações **MVC** quando comparamos com **WebForms**.

Carregando DropDownLists

Considere o cenário de uma aplicação comercial, onde você precisa exibir **DropDownLists** para selecionar **Estado** e **Cidade**. Os dados já estão armazenados no banco de dados, e as entidades expostas pelo **DbContext** criado pelo **EntityFramework**. Como podemos carregar esses controles numa aplicação **MVC**?

Na verdade, existem diversas formas para alimentarmos controles do tipo lista em aplicações **MVC**. A forma apresentada neste tutorial é prática e apresenta uma implementação muito simples.

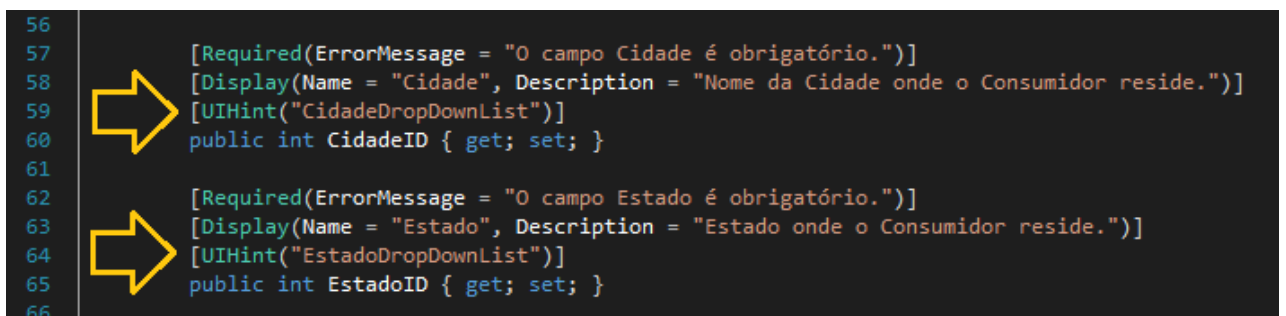
O primeiro passo é criarmos uma **View** que encapsulará o controle de lista. Vá para pasta **Views**, **Shared** e crie uma pasta chamada **EditorTemplates** (se não existir). Dentro desta pasta, crie uma **View** utilizando o **Empty Template** (vazio). No arquivo criado, modifique o código para que fique exatamente como apresentado na **Figura 36**. Este arquivo será responsável por informar a **UI** que o editor para a propriedade **EstadoId** será um controle **DropDownList** que exibirá todos as unidades da federação e estados gravados no banco de dados.



```
1 @using CBC.Natal.Data.Entities
2
3 @if
4 {
5     CBCNatalContext natalContext = new CBCNatalContext();
6
7     var list = natalContext.Estado.OrderBy(c => c.UF).ToList();
8
9     var selectList = new SelectList(list, "EstadoID", "UF", Model);
10
11 }
12
13 @Html.DropDownListFor(model => model, selectList, "[Selecione]", new { @style = "color:#dae861" })
14
```

Figura 36: O EditorTemplate para propriedade EstadoID

Uma vez criado este *template*, com a ajuda do atributo [UIHint] associado à propriedade **EstadoID**, e o engine do **ASP.NET MVC** se encarregará de exibir o controle **DropDownList** devidamente carregado. Observe a **Figura 37** que utiliza o atributo [UIHint] para associar o **EditorTemplate** às propriedades **CidadeID** e **EstadoID**.



```
56
57 [Required(ErrorMessage = "O campo Cidade é obrigatório.")]
58 [Display(Name = "Cidade", Description = "Nome da Cidade onde o Consumidor reside.")]
59 [UIHint("CidadeDropDownList")]
60 public int CidadeID { get; set; }
61
62 [Required(ErrorMessage = "O campo Estado é obrigatório.")]
63 [Display(Name = "Estado", Description = "Estado onde o Consumidor reside.")]
64 [UIHint("EstadoDropDownList")]
65 public int EstadoID { get; set; }
66
```

Figura 37: Utilizando o Atributo [UIHint]

DropDownLists em Cascata com JQuery

Outra tarefa comum neste mesmo cenário, refere-se a capacidade de filtrarmos os registros apresentados em um controle **DropDownList** com base no registro selecionado pelo usuário em outro controle do mesmo tipo.

Por exemplo, quando o usuário selecionar o **Estado** desejado no controle explicado no tópico anterior, nossa aplicação deverá carregar o controle de lista associado à propriedade **CidadeID** com as cidades pertencentes ao estado selecionado.

Com algumas linhas de código adicionais, um pouco de **JQuery + JSON**, e esta tarefa será facilmente concluída. Observe a ação **GetCidades()** adicionada à classe **ConsumidorController**. Esta ação será invocada a partir de uma função **JQuery** publicada na *View* **Consumidor/Create**. A **Figura 38** apresenta o código que implementa a ação **GetCidades()**.

```

314
315 [HttpGet]
316 [AllowAnonymous]
317 public ActionResult GetCidades()
318 {
319     int? estadoId = null;
320     if(Request.QueryString.HasKeys())
321         estadoId = Convert.ToInt32(Request.QueryString.Get("estado"));
322
323     var listaCidades = from c in natalContext.Cidade
324                        where c.EstadoID == estadoId
325                        select new
326                        {
327                            CidadeID = c.CidadeID,
328                            NomeCidade = c.NomeCidade
329                        };
330
331     var oSerializer = new System.Web.Script.Serialization.JavaScriptSerializer();
332     string sJSON = oSerializer.Serialize(listaCidades.ToList());
333     return Json(sJSON, JsonRequestBehavior.AllowGet);
334 }
335

```

Figura 38: A ação GetCidades da classe ConsumidorController

A função **jQuery**, para implementar a carga “em cascata” do controle **DropDownList CidadeID**, também é simples. A **Figura 39** apresenta o código necessário para esta função.

```

17
18 $(document).ready(function () {
19     displayMessages();
20
21     //bind evento onselecindechange
22     $("#EstadoID").bind("change", function () {
23         //chamada ajax para controler(consumidorController) metodo GetCidades
24         $.getJSON("GetCidades", { estado: $("#EstadoID").val() }, function (result)
25         {
26             //limpa combo
27             $("#CidadeID").html("");
28             $("#CidadeID").append($("<option></option>").val(null).html("[Selecione]"));
29             data = $.parseJSON(result);
30             //para cada valor retornado adiciona uma option no combo
31             for (var i = 0; i < data.length; i++) {
32                 //item corrente da lista
33                 var item = data[i];
34                 $("#CidadeID").append($("<option></option>").val(item.CidadeID).html(item.NomeCidade));
35             }
36         });
37     });
38 }
39

```

Figura 39: A função JQuery para carregar o controle DropDownList CidadeID

Paginação de Registros no MVC

A paginação de registros também é uma tarefa muito comum e necessária para aplicações web. No caso das aplicações **MVC**, temos uma alternativa disponível através do **NuGet Gallery**, que é o controle **PagedList**, encontrado neste endereço: <http://nuget.org/packages/PagedList.Mvc>.

Implementando Paginação para o Grid

Para utilizarmos o controle **PagedList** em nosso projeto, em primeiro lugar, é preciso instalar este controle a partir do **Package Manager Console**. O comando para instalação é simples, basta digitar **Install-Package PagedList.Mvc** na linha de comando do **Package Manager Console** (**Menu Tools -> Library Package Manager**).

No topo classe *Controller*, onde utilizaremos o **PagedList**, é necessário acrescentar a declaração: **using PagedList**. Uma vez adicionada esta declaração, vamos modificar o conteúdo da nossa ação **Index** para acrescentarmos os parâmetros necessários para o controle **PagedList**. Ao final deste processo, o código da nossa ação **Index** deverá apresentar construção semelhante à apresentada na **Figura 40**.

```
public ActionResult Index(string sortOrder, string currentFilter, string searchString, int? page)
{
    ViewBag.Message = "Consulta Consumidores";

    ViewBag.CurrentSort = sortOrder;
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "NomeConsumidor desc" : "";
    ViewBag.DateSortParm = sortOrder == "DataCadastro" ? "DataCadastro desc" : "DataCadastro";

    if (Request.HttpMethod == "GET")
    {
        searchString = currentFilter;
    }
    else
    {
        page = 1;
    }

    int pageSize = int.Parse(ConfigurationManager.AppSettings["AdmPageSize"].ToString());
    int pageNumber = (page ?? 1);
    string nomeConsumidor = string.Empty;
    string numeroCPF = string.Empty;

    return View(ListaConsumidores().ToPagedList(pageNumber, pageSize));
}
```

Figura 40: A ação **Index** modificada para o controle **PagedList**.

Observe na **Figura 40**, a linha em destaque, o método **ToPagedList(pageNumber, pageSize)**, este é o método responsável por implementar efetivamente a paginação dos registros. No exemplo acima, o tamanho da página (**pageSize**) é definido do arquivo **Web.config** e recuperado com o auxílio do objeto **ConfigurationManager**. A **View** que apresentará os registros paginados, também deve sofrer as alterações necessárias. A primeira delas é a declaração da classe **Model** que deve ser derivada da **Interface IPagedList**, como mostra a **Figura 41**.

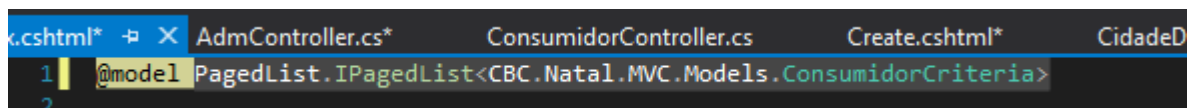


Figura 41: A declaração da classe Model

E finalmente, acrescentamos o código que exibirá os links para navegação através das páginas. Este código, deve ser colocado logo abaixo do bloco *foreach* utilizado para listar os registros. O código na íntegra está disponível com a aplicação de exemplo distribuída junto com este tutorial. A Figura 42 apresenta o fragmento do código necessário para o **PagedList**.

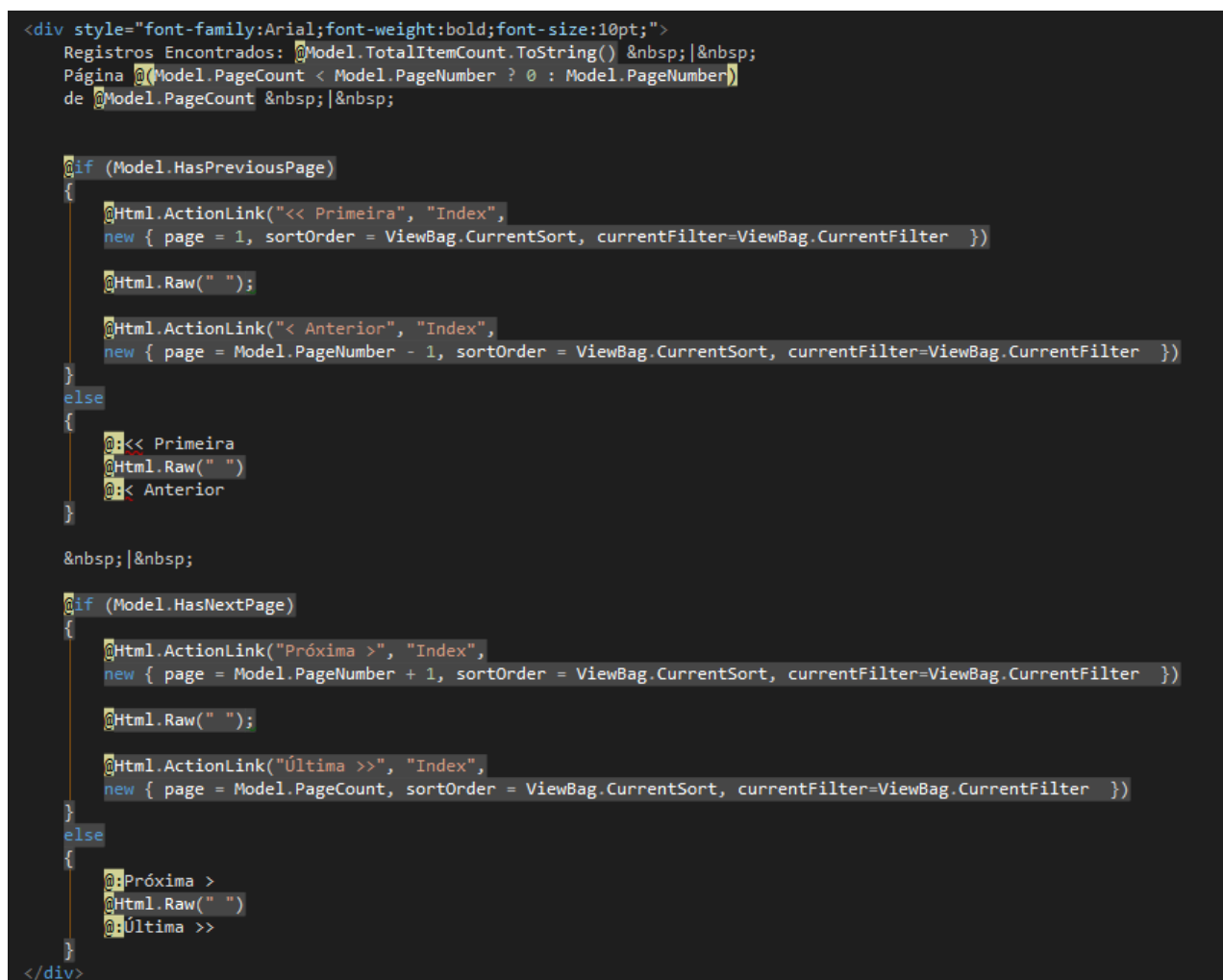


Figura 42: O código para os links de navegação do PagedList.

Conclusão Final

Há muitos e diferentes modelos para construirmos uma aplicação **MVC**. O modelo apresentado neste tutorial baseia-se em práticas recomendadas para o mercado. Sugiro que o desenvolvedor aprofunde seus conhecimentos visitando o portal www.asp.net/mvc.

Referências

Adam Tuliper's Blog

<http://completeddevelopment.blogspot.com.br/>

Alex James Blog

<http://blogs.msdn.com/b/alexi/>

Channel 9

<http://channel9.msdn.com/>

Entity Framework Developer Center

<http://msdn.com/ef>

Portal ASP.NET MVC

<http://www.asp.net/mvc/>

NuGet Gallery PagedList

<http://nuget.org/packages/PagedList.Mvc>

Martijn Boland's Blog

<http://blogs.taiga.nl/martijn/>