UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

FELIPE RITTER

DESENVOLVIMENTO EM ASP.NET MVC UTILIZANDO ENTITY FRAMEWORK

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

FELIPE RITTER

DESENVOLVIMENTO EM ASP.NET MVC UTILIZANDO ENTITY FRAMEWORK

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – CSTADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Msc. Everton Coimbra de Araújo.

MEDIANEIRA
2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

Desenvolvimento em ASP.NET MVC Utilizando Entity Framework

Por

Felipe Ritter

Este Trabalho de Diplomação (TD) foi apresentado às 14:40h do dia 13 de junho de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O acadêmico foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Msc. Everton Coimbra de Araújo
UTFPR – *Campus* Medianeira
(Orientador)

Prof. Diego Stiehl
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Msc. Cláudio Leones Bazzi
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Msc. Juliano Lamb
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

A folha de aprovação assinada encontra-se na Coordenação do Curso.

“Grandes realizações não são feitas por impulso,
mas por uma soma de pequenas realizações.”

Vincent Van Gogh, pintor, 1853-1890

AGRADECIMENTOS

Ao professor orientador Everton Coimbra de Araújo, pela disponibilidade para auxiliar na solução de dúvidas durante a elaboração deste trabalho e ao professor Juliano Lamb, responsável pela atividade dos trabalhos de diplomação, pela disponibilidade para resolver problemas que surgiram no decorrer do mesmo.

RESUMO

A cada dia que passa, cresce a demanda por sistemas *Web* e, paralelamente a isso, exige-se cada vez mais das aplicações. Com isso, os programadores necessitam de ferramentas que auxiliem no desenvolvimento desses sistemas, para que aumente a produtividade e melhore o desempenho dos mesmos. O ASP.NET MVC é um *framework* da plataforma .NET que fornece um desenvolvimento dividido em camadas, promovendo total controle da aplicação e, juntamente com o ADO.NET Entity Framework, é possível realizar de forma transparente e rápida o acesso ao banco de dados, além de simplificar o código das classes responsáveis pelas regras de negócio da aplicação. Um estudo de caso mostra as facilidades que os *frameworks* proporcionam aos desenvolvedores durante a criação dos sistemas, como o acesso à base de dados sem preocupação com abertura e fechamento de conexões e a transparência no acesso, sem se preocupar com comandos SQL, além da maneira com que funcionam as classes de um projeto ASP.NET MVC.

Palavras-Chave: *ASP.NET MVC, ADO.NET Entity Framework.*

ABSTRACT

To each day that passes, the demand grows for Web systems and, in parallel to this, it is demanded more and more of the applications. With that, the programmers need tools that help them in the development of these systems, so that it increases the productivity and improves the performance of same. The ASP.NET MVC is a framework of the .NET platform what supplies a development divided in layers, promoting total control of the application and, together with the ADO.NET Entity Framework, is possible to do transparently and quick access to the database, besides simplifying the code of the classes responsible for the business rules of the application. A case study shows the facilities that the frameworks provide to the developers during the creation of the systems, like the access to the base of data without preoccupation with opening and closure of connections and the transparency in the access, without be preoccupying by commands SQL, and how the classes of a project ASP.NET MVC work.

Keywords: *ASP.NET MVC, ADO.NET Entity Framework.*

LISTA DE SIGLAS

API	-	*Application Programming Interface*
CRUD	-	*Create, Retrieve, Update, Destroy*
DAO	-	*Data Access Object*
DER	-	*Diagrama Entidade-Relacionamento*
DDL	-	*Data Definition Language*
EDM	-	*Entity Data Model*
EF	-	*Entity Framework*
EF4	-	*Entity Framework 4*
HTML	-	*HyperText Markup Language*
HTTP	-	*Hypertext Transfer Protocol*
IDE	-	*Integrated Development Environment*
LINQ	-	*Language Integrated Query*
MVC	-	*Model-View-Controller*
POCO	-	*Plain Old CLR Object*
RTM	-	*Release to Manufacturing*
SSDL	-	*Store Schema Definition Language*
SQL	-	*Structured Query Language*
SSL	-	*Secure Sockets Layer*
URL	-	*Uniform Resource Location*

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

É grande o número de aplicações web nas quais, durante o seu desenvolvimento, os programadores têm muita preocupação com comandos SQL (*Structured Query Language*) e, principalmente, com a conexão com a base de dados (RAYMUNDO, 2010).

Isso causa vários obstáculos no decorrer do projeto e pode acabar atrasando o mesmo, ou então acumulando atividades, o que "congestiona" as tarefas dos desenvolvedores.

Muitas dessas aplicações têm, ainda, problemas com o redirecionamento entre uma página e outra quando necessitam passar algum objeto por parâmetro ou realizar alguma chamada de método para execução de regras de negócio do sistema.

Algumas vezes, também, o programador se depara com uma situação em que é necessário mais de um *form* HTML (*Hypertext Markup Language*) na mesma página. Isso gera um enorme problema e, muitas vezes, a solução está em outra forma de desenvolvimento. Isso é possível com ASP.NET MVC, onde todo o HTML gerado é o programador quem controla. Mesmo nos métodos de auxílio, quem decide se vai utilizá-los é o programador (BASSI, 2008).

1.1 OBJETIVO GERAL

Este trabalho objetiva demonstrar o aumento no desempenho para desenvolver aplicações ASP.NET utilizando o padrão ASP.NET MVC juntamente com Entity Framework.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são:

- Realizar uma pesquisa bibliográfica sobre as tecnologias ASP.NET MVC e ADO.NET Entity Framework;
- Demonstrar a arquitetura, modelo de programação e funcionamento do ASP.NET MVC;
- Desenvolver uma aplicação web em ASP.NET MVC utilizando ADO.NET Entity Framework.

- Demonstrar as facilidades no desenvolvimento de aplicações ASP.NET com o uso de ASP.NET MVC e ADO.NET Entity Framework;

1.3 JUSTIFICATIVA

O ADO.NET Entity Framework permite realizar a modelagem e o acesso às informações de uma base de dados através de um modelo conceitual de banco de dados, deixando as regras de negócio de forma transparente ao desenvolvedor. Uma das vantagens disso é que o mapeamento dos objetos do banco e o próprio banco de dados podem ser alterados sem prejuízo para o desenvolvimento do sistema, a camada de aplicação torna-se independente da camada de acesso aos dados.

Outra vantagem do ADO.NET Entity Framework é o suporte ao uso de *LINQ (Language Integrated Query) to Entities*, fazendo com que o desenvolvedor não se preocupe com o SQL, deixando o próprio Framework fazer isso da melhor maneira que ele encontrar (RAYMUNDO, 2010).

Trabalhando paralelamente com o ADO.NET, o padrão *Model-View-Controller* (MVC) de arquitetura separa uma aplicação em três componentes principais: o modelo, a visão e o controlador (PALERMO, 2010). A estrutura do ASP.NET MVC fornece uma alternativa ao padrão ASP.NET Web Forms para criar MVC baseada em aplicações web. O ASP.NET MVC é leve e possui uma estrutura de apresentação altamente testável, que é integrada com os recursos existentes do ASP.NET, tais como páginas mestras e autenticação baseada em membros (HANSELMAN, 2010).

O ASP.NET MVC auxilia em várias partes do desenvolvimento, desde a parte de redirecionamento de páginas, juntamente com as regras de negócio, passagem de objetos por parametro e chamadas de métodos de outras páginas, que são controlados pelas controllers, até a parte de testes do sistema.

O padrão também torna fácil a divisão do sistema em módulos, o que facilita o desenvolvimento em equipe.

Com o constante crescimento no desenvolvimento web com ASP.NET, faz-se necessário um estudo aprofundado sobre as tecnologias existentes e os benefícios que o ADO.NET Entity Framework e o ASP.NET MVC podem trazer para o desenvolvimento, os quais podem fazer diferença na criação de uma aplicação de qualidade.

1.4 ESTRUTURA DO TRABALHO

O trabalho é composto por cinco capítulos, o primeiro apresenta a introdução sobre o assunto a ser abordado, com os objetivos gerais e específicos e a justificativa para o tema, também fazendo uma introdução ao *.NET* F*ramework,* além da ferramenta de desenvolvimento *Visual Studio* e do banco de dados *SQL Server*. O Capítulo dois descreve o ASP.NET MVC e seus principais conceitos. No capítulo três, encontram-se os conceitos sobre o ADO.NET Entity Framework. Já o quarto capítulo mostra exemplos práticos da utilização do ASP.NET MVC juntamente com o *Entity framework*, através de um estudo de caso, demonstrando as facilidades do uso de amos no desenvolvimento *Web*. O Capítulo cinco ressalta as considerações finais sobre o estudo de caso realizado.

1.5 .NET FRAMEWORK

1.5.1 ASP.NET

É uma plataforma de desenvolvimento criada pela Microsoft para geração de códigos do lado do servidor (*server-side*). É o sucessor da linguagem ASP, entretanto foi escrito a partir do zero e não é compatível com o ASP clássico (TECHNET, 2011).

ASP.NET é a maior parte do framework .NET da Microsoft.

Algumas vantagens de sua utilização são:

- ASP.NET reduz drasticamente a quantidade de código necessária para construir grandes aplicações;
- As páginas ASP.NET são fáceis de escrever e manter, porque o código-fonte e o HTML estão juntos;
- O código fonte é executado no servidor. As páginas têm muito poder e flexibilidade;
- O código fonte é compilado pela primeira vez que a página é solicitada, o que torna a execução mais rápida no servidor *Web*. O servidor guarda a versão compilada da página para usar na próxima vez que a página é solicitada;
- O HTML produzido pela página ASP.NET é enviado de volta para o navegador. O código-fonte da aplicação escrita não é enviado e não é facilmente roubado;

- O servidor *Web* monitora continuamente as páginas, componentes e aplicativos rodando sobre ele. Se notificar vazamentos de memória, loops infinitos, software ilegal ou outras atividades, para perfeitamente essas atividades e reinicia-se;
- ASP.NET valida as informações (controles de validação) digitadas pelo usuário sem escrever uma única linha de código;
- Trabalha facilmente com ADO.NET usando ligação de dados e recursos de formatação de página (TEMPLEMAN & VITTER, 2007).

1.5.2 Microsoft Visual Studio

O Microsoft Visual Studio é um conjunto de ferramentas desenvolvido pela Microsoft para desenvolvimento com o uso do *framework* .NET e a sua versão mais recente é o Microsoft Visual Studio 2010. É um grande produto de desenvolvimento para web, com a plataforma ASP.NET, com suporte às linguagens VB.NET (Visual Basic .NET) e C# (MICROSOFT, 2010d).

O *Visual Studio* é uma suite de ferramentas de desenvolvimento baseado em componentes e outras tecnologias para a construção de poderosas aplicações de alto desempenho. Além disso, é otimizado para o projeto em equipe, desenvolvimento e implantação de soluções corporativas (TEMPLEMAN & VITTER, 2007).

A **Figura 1** mostra a tela da ferramenta em uso, durante o desenvolvimento de um sistema.

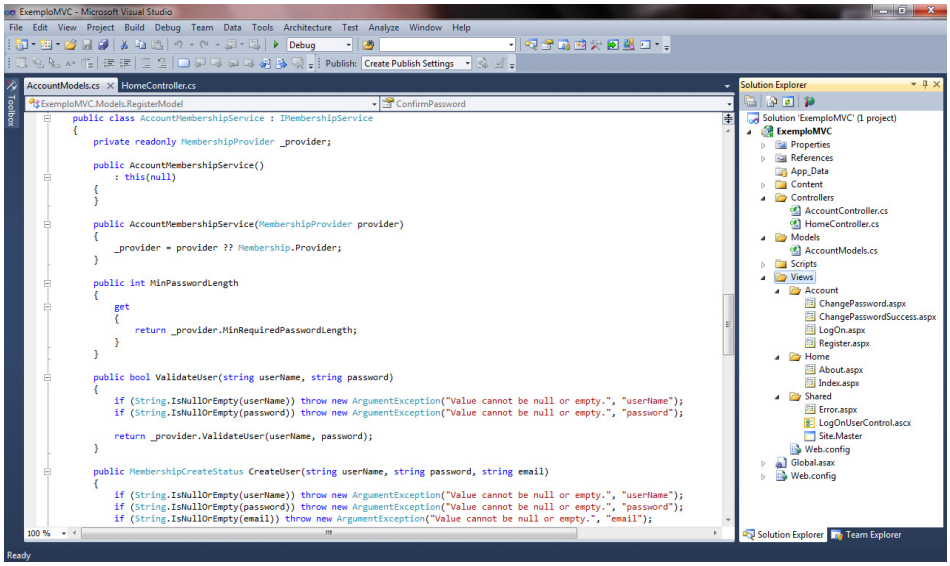

Figura 1 - Visual Studio 2010

1.5.3 Microsoft SQL Server

O SQL Server é o sistema gerenciador de banco de dados (SGBD) desenvolvido pela Microsoft em parceria com a Sybase, sendo versão mais atual a SQL Server 2008 (OFICIALDANET, 2011).

Fornece uma plataforma de dados confiável, produtiva e inteligente que permite a execução de aplicações de missão crítica mais exigentes, reduz o tempo e o custo com o desenvolvimento e o gerenciamento de aplicações e entrega percepção que se traduz em ações estratégicas (HIRT, 2010).

Segundo Hirt (2010), o SQL Server é um banco de dados robusto, usado por sistemas corporativos dos mais diversos portes.

A **Figura 2** mostra a ferramenta de gerenciamento da base de dados do SQL Server.

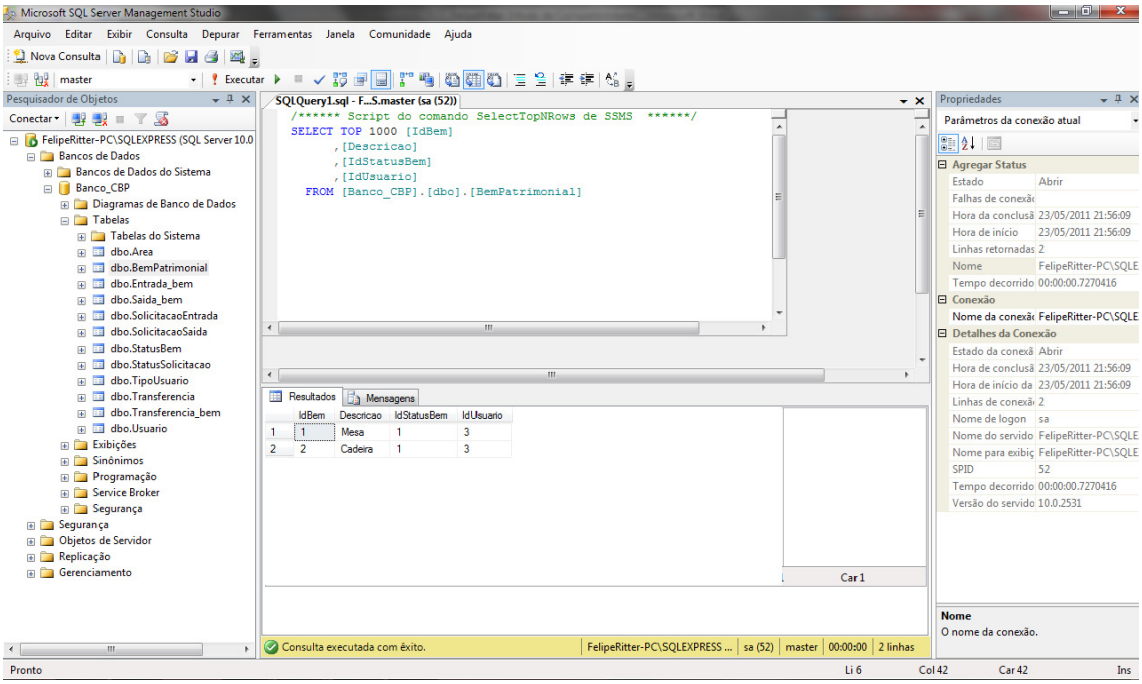

Figura 2 - SQL Server Management Studio 2008

1.5.4 C#

C# é uma linguagem de programação desenvolvida pela Microsoft e está incluída no *framework* .NET (TECHNET, 2010).

É a linguagem símbolo do .NET por ter sido criada praticamente do zero, especificamente para a plataforma (TECHNET, 2010).

O C# possui várias características diversificadas. Por exemplo, não existe herança múltipla, a classe não pode herdar mais de uma classe, no entanto, existe uma maneira de simular heranças múltiplas, que é com o uso de interfaces.

A linguagem só suporta ponteiros com o uso da palavra reservada *unsafe*. Sua implementação não é aconselhável e blocos de códigos que o usam geralmente requisitam permissões mais altas de segurança para poderem ser executados (TECHNET, 2010).

2 ASP.NET MVC

2.1 SURGIMENTO E EVOLUÇÃO

Até outubro de 2007, quando foi apresentado um primeiro *preview*[1] do ASP.NET MVC, a comunidade ASP.NET possuía apenas o ASP.NET Web Forms como opção de desenvolvimento para *Web*. Apesar da sua existência, muitos desenvolvedores procuravam um framework para trabalhar com o padrão MVC, além de uma forma mais simples de gerar suas páginas, uma maneira que fosse livre do complexo ciclo de vida das aplicações com o padrão *Web Forms*. Além disso, muitos destes desenvolvedores procuravam também uma forma de aproximar suas aplicações da forma como a *Web* e o HTTP (*Hypertext Transfer Protocol*) funcionam: sem estado (QUAIATO, 2011).

É neste cenário que o ASP.NET MVC começa a encontrar mercado. Trabalhar com o padrão MVC não é apenas uma questão de gosto, este padrão de arquitetura ajuda os desenvolvedores a trabalhar com "*separation of concerns*" (separação de conceitos) em suas aplicações. Possibilita e facilita a testabilidade, propiciando ainda o desenvolvimento utilizando TDD (*Test Driven Development*) (QUAIATO, 2011).

Em março de 2009 o ASP.NET MVC chega a sua versão 1.0. A comunidade já vinha acompanhando o ASP.NET MVC desde sua versão CTP, e no lançamento do *release* 1.0 a Microsoft também liberou o código fonte do ASP.NET MVC para que a comunidade pudesse estudá-lo e possivelmente aprimorá-lo, ainda que não fazendo "*commits*[2]" para o repositório do projeto (QUAIATO, 2011).

Em março de 2010 é anunciado o lançamento do ASP.NET MVC 2, apenas um ano após o lançamento anterior. Foi um ciclo de vida bastante curto, mas que conseguiu evoluir muito a ferramenta. O ASP.NET MVC passa a tornar-se mais utilizado e ganhar mais mercado e adeptos. Livros já foram publicados e as empresas começaramm a adotar o ASP.NET MVC sem muito receio de ser um projeto que não vai dar certo (QUAIATO, 2011).

Em julho de 2010, apenas três meses após o lançamento da versão 2 do ASP.NET MVC é lançado um primeiro *preview* do ASP.NET MVC 3. Este já trouxe algumas das novidades mais interessantes do que estaria por vir no MVC 3: *Razor View Engine*, uma nova *engine* para escrita das *Views* no ASP.NET MVC (QUAIATO, 2011).

---

[1] *preview* – Pré-lançamento.

[2] *commit* – Comando que efetiva a ação que está sendo executada no banco de dados.

Em outubro de 2010 foi anunciado o primeiro beta do ASP.NET MVC 3, passados apenas sete meses do lançamento da última versão RTM (*Release to Manufacturing*). De fato a equipe do ASP.NET MVC está trabalhando arduamente para disponibilizar um framework robusto e que atenda às necessidades do mercado e dos desenvolvedores que estão trabalhando com o mesmo (QUAIATO, 2011).

2.2 ARQUITETURA

O padrão de arquitetura MVC (*Model-View-Controller*) separa a aplicação em três partes principais: modelo (*models*), visão (*views*) e controladores (*controllers*).

O *framework* MVC para ASP.NET providencia uma alternativa para o padrão *Web Forms*, pois permite a criação de aplicações web baseadas em MVC .

Segundo Hanselman (2010), o *framework* MVC não substitui os *Web Forms* do ASP.NET. Deve-se pensar cuidadosamente sobre quando criar um projeto utilizando MVC. Contudo, é possível utilizar tanto MVC quanto *Web Forms* na mesma aplicação.

A **Figura 3** mostra os componentes do *framework* MVC:

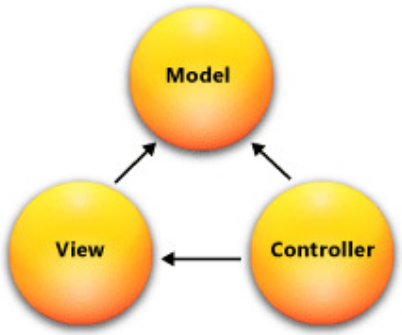

Figura 3 - Componentes do ASP.NET MVC
Fonte: Hanselman (2010)

- **Model:** é a parte da aplicação onde se encontram as regras de negócio e comunicação com banco de dados.
- **View:** é o componente da aplicação onde se encontra a interface do usuário. Normalmente é criada baseada em um objeto modelo.
- **Controller:** é o componente que interage com o usuário. Trabalha em conjunto com o *model* e, por fim, seleciona a página ideal para a qual o usuário será redirecionado. Em uma aplicação MVC, as páginas (*views*) somente possuem

informações; são os controladores (*controllers*) que manuseiam e respondem às solicitações do usuário (MICROSOFT, 2010c).

O ASP.NET MVC também faclita o teste dos aplicativos com relação ao *Web Forms.* Por exemplo, em uma aplicação ASP.NET baseada em *Web Forms*, uma única classe é responsável tanto pelo controle da exibição da saída de dados, quanto pela entrada de dados do usuário. Escrever testes automatizados para aplicações *Web* baseada em formulários pode ser complexo, porque para testar uma página individual, deve-se instanciar a classe da página, todos os seus controles filho e mais classes dependentes da aplicação. Pelo fato de muitas classes serem instanciadas para executar a página, pode ser difícil escrever testes que se concentrem exclusivamente em partes individuais da aplicação (MSDN, 2010c).

O *framework* MVC separa os componentes e faz uso pesado de interfaces, o que torna possível testar os componentes individuais isoladamente do resto da aplicação (MICROSOFT, 2010c).

Ele é construído como uma série de componentes independentes - satisfazendo uma interface .NET ou construído em uma classe abstrata - para que se possa facilmente substituir o sistema de roteamento, o motor das *Views*, a criação das *Controllers*, ou qualquer componente de outro *framework*, com uma implementação própria de maneira diferente (SANDERSON, 2010).

A estrutura do ASP.NET MVC oferece as seguintes vantagens:

- Fica mais fácil para gerenciar a complexidade dividindo um aplicativo em modelo, visão e controlador.
- Usa um padrão *Front Controller* que processa as solicitações de aplicativos *Web* em um único controlador. Isso permite criar um aplicativo que oferece suporte a uma rica infra-estrutura de roteamento.
- Funciona bem para aplicativos da *Web* que são suportados por grandes equipes de desenvolvedores e para os *Web designers* que precisam de um alto grau de controle sobre o comportamento do aplicativo (HANSELMAN, 2010).

Toda requisição em uma aplicação ASP.NET MVC, passa primeiramente por um objeto *UrlRoutingModule*, que consiste em um módulo HTTP. Este módulo analisa a solicitação e realiza a seleção do caminho e seleciona o primeiro objeto da rota que coincide com a solicitação realizada pelo usuário.

Em um site desenvolvido com o uso de ASP.NET MVC, uma URL (*Uniform Resource Location*) normalmente serve como um mapa para um arquivo armazenado em

disco (geralmente um arquivo *.aspx*). Esses arquivos possuem marcações e códigos que são processados de forma a atender as solicitações realizadas (MSDN, 2010a).

O ASP.NET MVC não utiliza o modelo *postback*[3] utilizado nos *Web Forms* para interações com o servidor. Em vez disso, todas as interações do usuário final são encaminhados para a classe controladora. Isso mantém a separação entre a interface e a lógica de negócio, ajudando a testabilidade (MICROSOFT, 2010a).

Se existe um projeto ASP.NET Web Forms e o desenvolvedor deseja migrá-lo para o MVC, as duas tecnologias podem coexistir no mesmo aplicativo ao mesmo tempo. Isso dá uma oportunidade de migrar a aplicação aos poucos, principalmente se ela já estiver dividida em camadas com o modelo de domínio ou de lógica de negócios realizada separadamente das páginas *Web Forms*. Em alguns casos se pode até mesmo deliberadamente criar um aplicativo para ser um híbrido das duas tecnologias (SANDERSON, 2010).

2.3 TESTABILIDADE

Arquitetura MVC fornece um grande auxílio ao simplificar a manutenção e a testabilidade da aplicação, porque o desenvolvedor irá separar diferentes problemas de um sistema em independentes partes do mesmo (SANDERSON, 2010).

Para suportar o teste de unidade, o *framework* possui um design orientado a componentes e cada peça separada é idealmente estruturada para atender as exigências (e superar as limitações) dos testes de unidade e ferramentas de simulação. Além disso, o *Visual Studio wizards* cria projetos de teste de unidade de partida[4] (integração com ferramentas open source de teste de unidade como NUnit e xUnit, bem como MSTest da Microsoft), por isso, mesmo que nunca tenha escrito uma unidade de teste antes, o desenvolvedor terá todo o auxílio necessário por parte do *framework* (SANDERSON, 2010).

Testabilidade não é apenas uma questão de testes de unidade. Aplicações ASP.NET MVC trabalham bem com as ferramentas de automação de teste da interface do usuário também. É possível escrever scripts que simulam as interações do usuário sem ter que adivinhar o que os elementos HTML, CSS ou IDs do *framework* irão gerar.

---

[3] *postback* – Quando uma página realiza um pedido para ela mesma, esse pedido é chamado de *postback*.

[4] *Teste de unidade* – É toda a aplicação de teste nas assinaturas de entradas e saídas de um sistema e consiste em validar dados via entrada/saída, sendo aplicado por desenvolvedores e analistas de teste.

2.4 API MODERNA

O ASP.NET MVC foi desenvolvido para as versões 3.5 e 4 do *.NET Framework*. Não se preocupa com problemas de incompatibilidade com versões anteriores, para ter todas as vantagens no uso das tecnologias mais recentes, incluindo métodos de extensão e tipos anônimos – todas as partes do LINQ (*Language Integrated Query*). Muitos dos métodos da API do *Framework* MVC e padrões de codificação seguem uma composição mais limpa, mais expressiva do que era possível quando as plataformas anteriores foram inventadas (SANDERSON, 2010).

2.5 SISTEMA DE ROTEAMENTO

Hoje, os desenvolvedores da *Web* reconhecem a importância do uso de URLs limpas. Não é bom fazer uso de incompreensíveis URLs como: /App_v2/User/Page.aspx?action=show%20prop&prop_id=82742. É muito mais profissional utilizar algo como: /Aluguel/Chicago/2303 (SANDERSON, 2010).

Os motores de busca dão um peso considerável às palavras-chave encontradas em uma URL. Uma busca por "aluguel em Chicago" é muito mais provável que encontre a última URL. Em segundo lugar, muitos usuários da web são espertos o suficiente para entender uma URL e apreciar a opção de navegar digitando na barra de endereço do navegador. Em terceiro lugar, não expõem-se inutilmente os detalhes técnicos, pasta e nome do arquivo, estrutura de sua aplicação a toda a rede pública, tendo, o desenvolvedor, liberdade para alterar a implementação subjacente sem desfragmentar todos os *links* recebidos (SANDERSON, 2010).

URLs limpas eram difíceis de implementar em *framework* anteriores, mas o ASP.NET MVC usa a facilidade do System.Web.Routing para limpar URLs por padrão. Isso dá controle total sobre o esquema de URL e o mapeamento para os controladores e ações, sem a necessidade de obedecer a qualquer padrão pré-definido.

Um exemplo prático para detalhar o mapeamento na URL em uma aplicação é uma chamada a “http://servidor:porta/Home/Index”, onde “servidor:porta” consiste no endereço da máquina que hospeda o site, “/Home” é o controlador (*controller*) responsável pela integração entre a página (*view*) e todas as regras de negócio relacionadas a ela e, por fim, “/Index”

consiste no método dentro do controlador, responsável por realizar a chamada da página em questão (no caso, *Index.aspx*).

Cada aplicação ASP.NET MVC precisa de pelo menos um caminho para definir como o aplicativo deve manipular as solicitações, mas geralmente vai acabar com vários. É concebível que uma aplicação muito complexa poderia ter dezenas de rotas ou mais.

As definições das rotas começam com a URL, que especifica um teste padrão que o percurso irá corresponder. Junto com a URL de rota, as rotas também podem especificar valores padrão e as restrições para as várias partes da URL, proporcionando um controle rígido sobre a forma como a rota corresponde a solicitação de entrada URL (GALLOWAY & HANSELMAN, 2010).

A rota consiste de vários segmentos URL, cada qual contém um espaço reservado delimitadado utilizando barras. Estes marcadores são referidos como parâmetros de URL.

Esta é uma regra de correspondência de padrão usado para determinar se esta rota se aplica a uma solicitação de entrada. Neste exemplo, esta regra irá corresponder a qualquer URL com três segmentos, porque um parâmetro de URL, por padrão, corresponde a qualquer valor não vazio. Quando ele corresponde a uma URL com três segmentos, o texto no primeiro segmento dessa URL corresponde ao parâmetro da URL {primeiro}, o valor no segundo segmento da URL que corresponde ao parâmetro URL {segundo}, e o valor no terceiro segmento corresponde ao {terceiro} (GALLOWAY & HANSELMAN, 2010).

Quando uma solicitação chega, analisa-se o encaminhamento de pedido de URL em um dicionário (especificamente um *RouteValueDictionary* acessível através do *RequestContext*), usando os nomes de parâmetro URL como as chaves e subseções do mesmo na posição correspondente aos valores (GALLOWAY & HANSELMAN, 2010).

2.6 VIEWS

A *View* é responsável por fornecer a *User Interface* (Interface de Usuário) para o usuário do sistema. É dada uma referência para o modelo, que o transforma em um formato pronto para ser apresentado ao usuário. O ASP.NET MVC examina o *ViewDataDictionary* entregue a ele pela *Controller* (acessado através da propriedade ViewData) e transforma isso em HTML.

No caso de *strongly-type View*, o *ViewDataDictionary* tem um tipo mais forte de modelo de objeto que o transforma em *View*. Esse modelo pode representar o objeto de

domínio real, como uma instância de um objeto, ou pode ser uma apresentação do modelo de objeto específico para a *View* (GALLOWAY & HANSELMAN, 2010).

Tomando como exemplo a **Figura 4**, temos:

```
    Home Page
</asp:Content>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">
    <h2><%: ViewData["Message"] %></h2>
    <p>
        To learn more about ASP.NET MVC visit <a href="http://asp.net/mvc"
  title="ASP.NET MVC Website">http://asp.net/mvc</a>.
    </p>
</asp:Content>
```

Figura 4 – View
Fonte: Galloway e Hanselman (2010)

Este é um exemplo extremamente simples de uma *View*, mas é útil para apontar alguns dos elementos essenciais das *Views* em ASP.NET MVC. Uma das primeiras coisas se pode notar é que, na superfície, ela parece com um *Web Form*. O ASP.NET MVC permite que se troque os diferentes motores das *Views*, mas o motor padrão da View é um *WebFormsViewEngine*.

Tecnicamente, este não é um *Web Form*, porque não inclui a *tag* <*form runat="server"*>. É apenas uma página ASP.NET. *Views* em ASP.NET MVC derivam de uma classe base comum, *System.Web.Mvc.ViewPage*, que por sua vez deriva de *System.Web.UI.Page*. *Strongly-type Views* derivam da *ViewPage*<*T*> genérica (GALLOWAY & HANSELMAN, 2010).

Essas páginas podem utilizar uma página-mãe (*Master Page*), que é criada automaticamente quando gerado o projeto ASP.NET MVC.

2.7 FILTROS

ASP.NET MVC inclui os seguintes filtros de ações:

- *Authorize*;
- *HandleError*;
- *OutputCache*;
- *RequireHttps*.

Os seguintes itens irão descrever cada um.

2.7.1 Authorize

O *AuthorizeAttribute* é a autorização de filtro padrão incluído no ASP.NET MVC. Usa-se para restringir o acesso a um método de ação. Aplicar este atributo para um controlador é uma abreviação para aplicá-lo a todos os métodos de ação (GALLOWAY & HANSELMAN, 2010).

Ao aplicar este atributo é possível especificar uma lista separada por vírgula, de funções ou usuários. Se especificada uma lista de funções, o usuário deve ser membro de pelo menos um desses papéis em ordem para o método de ação executar. Da mesma forma, especificada uma lista de usuários, o nome do usuário atual deve estar nessa lista.

2.7.2 RequireHttps

O *RequireHttpsAttribute* pode ser aplicado à métodos de ação e aos controladores para impedir o acesso não-SSL (*Secure Sockets Layer*). Se um não-SSL (HTTP) GET é recebido, ele será redirecionado para a URL SSL (HTTPS) equivalente (GALLOWAY & HANSELMAN, 2010).

No caso de uma solicitação HTTP usando um método diferente de GET, um redirecionamento não é apropriado, porque o corpo do pedido e o verbo não podem ser reproduzidos corretamente. Por exemplo, redirecionar a partir de um formulário de solicitação de POST não preserva os valores dos elementos contidos no formulário. Neste caso, o filtro *RequireHttpsAttribute* gera um *InvalidOperationException* para garantir que o verbo solicitado e corpo sejam propagados corretamente.

RequireHttpsAttribute tem um propósito muito simples, portanto, não expõem todos os outros parâmetros de ordem, que herdam da classe base *FilterAttribute*. Isso torna o uso muito simples (GALLOWAY & HANSELMAN, 2010).

2.7.3 OutputCache

O *OutputCacheAttribute* é usado para armazenar a saída de um método de ação. Esse atributo é um gancho para o *built-in ASP.NET output cache feature* e fornece grande parte da API e do comportamento que se recebe quando usa-se *@OutputCache* na página (GALLOWAY & HANSELMAN, 2010).

Com o MVC, a ação da *Controller* executa antes de uma *View* ser selecionada. Assim, colocando cache de saída em uma *View*, vai realmente armazenar em cache a saída dessa página, o método de ação propriamente dito continua a executar em cada pedido, negando assim a maioria dos benefícios do cache de saída (GALLOWAY & HANSELMAN, 2010).

Ao aplicar esse atributo para um método de ação, o filtro pode então determinar se o cache é válido e deixar de chamar o método de ação e vai direto para o retorno do conteúdo do cache.

2.7.4 Exception Filter (HandleError)

O HandleErrorAttribute é o filtro de exceção padrão incluído no ASP.NET MVC. Usa-se para especificar um tipo de exceção para tratar em uma *View* (e *Master View*, se necessário) para mostrar se um método de ação gera uma exceção sem tratamento ou é derivado do tipo de exceção especificada (GALLOWAY & HANSELMAN, 2010).

Por padrão, se nenhum tipo de exceção é especificado, então o filtro manipula todas as exceções. Se nenhuma página é especificada, o filtro padroniza o erro para uma *View* nomeada. O padrão de projeto ASP.NET MVC inclui uma *View* chamada *Error.aspx* dentro da pasta compartilhada.

2.7.5 Como o AuthorizeFilter Interage Com o OutputCache

O ASP.NET MVC oferece suporte a cache de saída através do seu filtro *OutputCache*. Isto funciona como o cache de saída do ASP.NET Web Forms, em que armazena em cache a resposta do todo para que possa ser reutilizado imediatamente na próxima vez que a mesma URL é solicitada. Nos bastidores, [OutputCache] é de fato implementado utilizando a tecnologia da plataforma ASP.NET de cache de saída, o que significa que se houver uma entrada de cache para uma determinada URL, esta será executada sem invocar qualquer parte do ASP.NET MVC (nem mesmo os filtros de autorização) (SANDERSON, 2010).

se combinar um filtro de autorização com [*OutputCache*], na pior das hipóteses, corre-se o risco de um usuário autorizado primeiramente visitar a sua ação, fazendo com que a ação seja executada e armazenada em cache, seguido por um usuário não autorizado, que recebe a saída em cache mesmo que eles não são autorizados. Felizmente, a equipe do ASP.NET MVC antecipou esse problema, e tem lógica especial para *AuthorizeAttribute* para lidar bem com o

cache de saída do ASP.NET. Ele usa uma API *OutPutCaching* pouco conhecida para registar-se para ser executado quando o módulo de cache de saída prestes a servir uma resposta do cache. Isso impede que usuários não autorizados tenham acesso ao conteúdo em cache (SANDERSON, 2010).

2.8 ASP.NET MVC E ASP.NET WEB FORMS JUNTOS

Muitas pessoas perguntam se é possível trabalhar com o ASP.NET MVC e ASP.NET Web Forms da mesma aplicação web. A resposta, felizmente, é bastante curta: Sim, é possível trabalhar com ambas as plataformas em um único aplicativo. A razão pela qual se pode fazer isso é porque eles são duas camadas distintas na parte superior do núcleo do ASP.NET que não interferem uma na outra (GALLOWAY & HANSELMAN, 2010).

Existem várias razões pelas quais um desenvolvedor pode desejar desenvolver um projeto com as duas plataformas:

- Web Forms é muito bom para encapsular a lógica da *View* em componentes. Pode-se ter uma necessidade de mostrar uma página de informação complexa, ou talvez uma página da portal, e poderia utilizar componentes *flex* conhecidos pelo desenvolvedor.
- ASP.NET MVC é muito eficiente para testar a lógica do aplicativo *Web*. A empresa pode ter uma política de cobertura de código que determina que noventa por cento de todo o código escrito deve ser coberto por um teste de unidade. MVC pode auxiliar nisso.
- É possível migrar uma aplicação *Web Form* existente para o ASP.NET MVC e não fazer isso tudo em um só ciclo de desenvolvimento. Ter a capacidade de adicionar ASP.NET MVC para a aplicação e, lentamente, observar o enorme benefício (GALLOWAY & HANSELMAN, 2010).

3 ADO.NET ENTITY FRAMEWORK

Como atualmente a maioria dos aplicativos são escritos em cima de bases de dados relacionais, mais cedo ou mais tarde eles terão que lidar com os dados representados em um formato relacional. Mesmo se houvesse um modelo de alto nível conceitual utilizado durante o projeto, esse modelo é tipicamente não diretamente "executável", por isso precisa ser traduzido em uma forma relacional e aplicado a um esquema de banco de dados lógico e ao código do aplicativo (KLEIN, 2010).

Respondendo a essa crescente tendência, a *Microsoft* criou uma camada intermediária, denominada *Entity Framework*, que tem a habilidade de transformar objetos de negócio em dados relacionais e vice versa, permitindo uma maior facilidade para a utilização de linguagens de programação orientadas a objeto e bancos de dados relacionais (MSDN, 2010b).

O *Entity Framework* controla a camada de negócio da aplicação responsável pela persistência dos objetos na base de dados.

Seu foco principal não é o banco de dados, mas o modelo de negócios e dessa forma uma de suas tarefas é gerar o modelo conceitual a partir do modelo de banco de dados, ou vice-versa, onde o banco de dados é gerado a partir das classes do sistema. Feito este serviço o desenvolvedor não tem que lidar mais diretamente com o banco de dados, mas com o modelo conceitual e o modelo de entidades (MACORATTI, 2011).

O ADO.NET Entity Framework 4, também chamado de EF4, é um *Middleware* com capacidades de modelagem e acesso a dados, sendo uma evolução do pacote de tecnologias ADO.NET.

O componente utilizado para realizar o mapeamento objeto/relacional é o *Entity Data Model* e as linguagens utilizadas para consulta são o *Entity SQL* e o *LINQ to Entities.*

A fim de fornecer um mecanismo para armazenar dados modelados pela EDM em bancos de dados relacionais, o ADO.NET Entity Framework abriga uma infra-estrutura cliente-views poderosa projetada para gerenciar as transformações entre o esquema do banco de dados lógico que está presente no armazenamento relacional e conceitual da EDM utilizado pelo aplicativo (JENNINGS, 2008).

Com isso, é facilmente feito o acesso a dados, já que não há necessidade de abertura e fechamento de conexões com o banco.

Todas as consultas resultam num *IQueryable*, que contem os dados vindos do banco e é facilmente percorrida com um comando *foreach* ou então filtrada com o uso de *LINQ* (BALTIERI, 2010).

O EF4 inclui o provedor de dados *EntityClient*. Este provedor gerencia conexões, traduzindo consultas a entidades para consultas de dados de fontes específicas, e retorna um leitor que o *Entity Framework* usa para materializar os dados em objetos.

Quando o objeto de materialização não é necessário, o provedor *EntityClient* também pode ser usado como um padrão de provedor de dados ADO.NET, permitindo as aplicações executar consultas *Entity SQL* (KLEIN, 2010).

A **Figura 5** ilustra a arquitetura do *Entity Framework* para acessar dados.

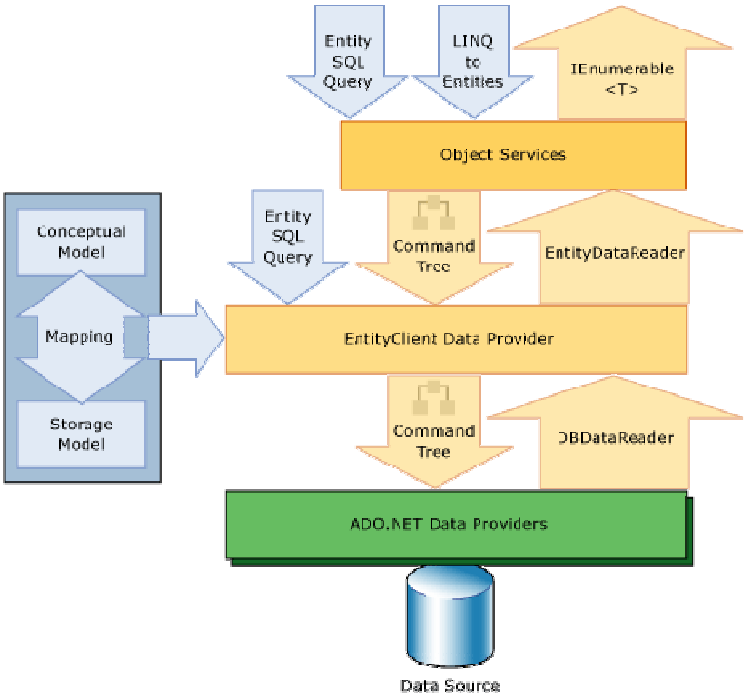

Figura 5 - Arquitetura do Entity Framework
Fonte: MSDN (2010b)

O EF4 é, por padrão, configurado com a forma *Lazy Load* de consulta nos dados persistidos, o que significa que só os objetos filhos do pesquisado serão carregados quando houver a necessidade.

Uma grande adição a versão 4.0 do *Entity Framework* é o suporte para tipos complexos. Os tipos complexos são como entidades que consistem de uma propriedade escalar ou uma ou mais propriedades do tipo complexo. Assim, eles são propriedades não

escalares de entidades que habilitam propriedades escalares a serem organizadas no âmbito das entidades (KLEIN, 2010).

O outro modo de consulta é o *Eager Load* que, automaticamente, traz junto aos objetos pais todas as listas de objetos filhos carregadas quando é realizada uma pesquisa na base de dados. Esse método é útil, por exemplo, quando se faz necessário o uso desses objetos para serem exibidos em uma tabela de consulta (BALTIERI, 2010).

3.1 SUPORTE A POCO

Uma das características mais novas e poderosas do Entity Framework é a capacidade de adicionar e usar suas próprias classes de dados personalizado em conjugação com o seu modelo de dados. Isto é alcançado usando-se os objetos CLR, vulgarmente conhecidos como "POCO" (*Plain Old CLR Objects*). A vantagem vem na forma de não precisar fazer modificações adicionais para as classes de dados (KLEIN, 2010).

A flexibilidade de estender essas classes parciais significa maior controle sobre a funcionalidade do núcleo entidade-objeto. Esta é uma enorme vantagem que os desenvolvedores podem, agora, aproveitar para preservar a lógica do negócio.

3.2 SUPORTE BACK-END

A grande vantagem do *Entity Framework* é que ele não se preocupa com o banco de dados a partir do qual os dados estão sendo consultados. Ele não precisa. O tipo de banco e o esquema em si são completamente desconhecidos para o Entity Framework, e eles não terão nenhum impacto em seu modelo (KLEIN, 2010).

O *Entity Framework* conta com dois provedores:

- *EntityClient Provider*: Usado por aplicativos Entity Framework para acessar os dados descritos na EDM (*Entity Data Model*). Esse provedor usa o *.NET Framework Data Provider* para *SQL Server* (*SqlClient*) para acessar um banco de dados *SQL Server*.
- *.NET Framework Data Provider* para *SQL Server* (*SqlClient*): Suporta o *Entity Framework* para uso com o banco de dados *SQL Server* (KLEIN, 2010).

O Entity Framework é independente de banco de dados, pode-se criar provedores personalizados para acessar outros bancos. Por exemplo, por meio de provedores de terceiros é possível acessar:

- Oracle;
- MySql;
- PostgreSQL;
- SQL Anywhere;
- DB2;
- Informix;
- U2;
- Ingress;
- Postgress;
- Firebird;
- Synergy;
- Virtuoso.

A grande vantagem desta característica é que o provedor faz todo o trabalho sobre a reformulação de consultas. O desenvolvedor é responsável por fornecer as informações de conexão para o armazenamento de dados específicos, mas o provedor se encarrega do resto ao trabalhar com o *Entity Framework*. Usa-se a sintaxe de consulta do *Entity Framework*, tais como o *LINQ to Entities* ou *Entity SQL* e renuncia-se à dor de cabeça de lembrar as diferenças de banco de dados (KLEIN, 2010).

3.3 CHAVES ESTRANGEIRAS E RELACIONAMENTOS

A criação de relacionamentos por parte do *Entity Framework* não é trivial, e segue regras específicas de DDL (*Data Definition Language*) para gerar cada associação.

Em um *Entity Data Model*, as relações parecem muito com relações lógicas no nível de esquema de banco de dados e, portanto, são as conexões lógicas entre as entidades. Cada entidade que participa de uma associação é chamada de um “ fim”. Cada extremidade tem um atributo chave que nomeia e descreve cada extremidade do relacionamento (ou em outras palavras, especifica as entidades relacionadas pela associação). Associações têm o que é um atributo chamado multiplicidade, que especifica o número de instâncias que cada extremidade pode participar da associação (KLEIN, 2010).

Uma explicação melhor pode ser visualizada no seguinte exemplo: um vendedor pode ter várias vendas, mas apenas uma venda pode ser relacionada a um único vendedor. As propriedades *onDelete* especificam uma ação a ser tomada quando uma entidade no “fim” correspondente será excluída. Em termos de banco de dados, isso é chamado de "delete em cascata" e fornece uma maneira de apagar um registro-filho quando o registro-pai for excluído, impedindo a criação dos chamados registros-filho "órfãos".

3.4 CARREGANDO ENTIDADES E PROPIEDADES DE NAVEGAÇÃO

O *Entity Framework* fornece um rico ambiente de modelagem que representa uma visão conceitual dos objetos subjacentes e relações em armazenamento de dados.

O comportamento padrão do *Entity Framework* é carregar somente as entidades acessadas diretamente pelo seu aplicativo. Em geral, isto é exatamente o que o desenvolvedor deseja. Se o EF carregasse todas as entidades ligadas através de uma ou mais associações, provavelmente iria acabar carregando entidades além do necessário. Isso aumentaria o consumo de memória do aplicativo, além de deixá-lo mais lento (TENNY & HIRANI, 2010).

Em *Entity Framework* é possível controlar o carregamento de entidades ligadas e otimizar o número de consultas de banco de dados a serem executadas. Gerir cuidadosamente quando as entidades relacionadas são carregados pode aumentar o desempenho e simplificar o seu código (TENNY & HIRANI, 2010).

Este tipo de carregamento, também chamado de *eager loading*, é usado tanto para reduzir o número de idas ao banco de dados quanto para controlar mais precisamente quais as entidades relacionadas são carregadas (TENNY & HIRANI, 2010).

Às vezes pode haver necessidade de adiar o carregamento de algumas entidades relacionadas, pois elas podem ser “pesadas” para carregar ou não são usadas com muita freqüência. O método *load*() pode ser usado para controlar precisamente quando carregar uma ou mais entidades relacionadas.

3.5 RELACIONAMENTOS MANY-TO-MANY

Para mapear uma tabela de associação de um relacionamento muitos para muitos no *Entity Framework*, ela não pode ser carregada no EDM. Se a mesma conter informações adicionais, o EF deve criar uma terceira entidade para que se possa acessar propriedades

adicionais e também ser capaz de inserir e atualizar esses valores para a tabela de ligação. Portanto, quando se importa o modelo usando o assistente de atualização, o EF reconhece que a tabela de ligação não contém quaisquer informações e a remove automaticamente, representando relação entre as entidades como muitos para muitos (HIRANI, 2010).

Ambas as classes serão geradas contendo em seus *Navigation Properties* uma *Collection* da entidade relacionada como *many-to-many.* Assim, uma consulta irá retornar os dados de uma delas e, junto a cada um eles, a lista dos contidos na tabela relacionada.

3.6 HERANÇA

Entity Framework oferece suporte a três diferentes modelos de herança.

- *Table per Hierarchy* (herança simples);
- *Table per Type*;
- *Table per Concrete Class* (HIRANI, 2010).

De todos os modelos de herança com suporte, o mais simples e mais fácil de implementar é o *Table per Hierarchy*. Para implementar essa herança, pode-se armazenar todos os tipos concretos em uma tabela. Em *Entity Framework,* para a identidade de uma linha como um tipo concreto específico, define-se uma coluna discriminadora que define para qual tipo concreto uma linha específica é mapeada (HIRANI, 2010).

Se a partir da perspectiva do banco de dados, o modelo não parecer privilegiar uma abordagem limpa, é porque todos os diferentes tipos concretos estão armazenados em uma única tabela (HIRANI, 2010).

Para realizar a flexibilidade ao nível da tabela, é preciso marcar todas as colunas que são específicas para sua implementação concreta como permitir nulos.

Alguns desenvolvedores de banco de dados podem ver nesta abordagem uma má solução, porque não se faz uso eficiente de espaço em disco. Por outro lado, *Table per Hierarchy* oferece bom desempenho, porque encontra um tipo concreto, não sendo necessário aplicar *joins* a uma outra tabela, o que pode ser "pesado" se a tabela for muito grande (HIRANI, 2010).

Como todos os tipos são armazenados em uma tabela, é possível aplicar o índice sobre a coluna discriminadora para permitir buscas mais rápidas com base no tipo concreto que se está procurando. Para mapear esta estrutura como uma tabela por hierarquia no modelo de dados da entidade, deve-se definir a coluna na qual o *Entity Framework* pode usar identidade

de cada tipo, sendo esta, uma coluna discriminadora. Em seguida, é necessário mover o campo específico para cada tipo da classe base para a sua própria entidade (HIRANI, 2010).

No modelo *Table per Type*, define-se uma tabela base que contém campos em comum em todos os tipos. Em seguida, cria-se uma tabela para cada tipo que contém campos que são específicos para esse tipo. Além disso, a coluna de chave primária definida na tabela derivada é também a chave estrangeira para a tabela base. Para mapear essa forma de estrutura de tabela em *Table per Type* do EDM, cada tipo de indivíduo precisa herdar o tipo de base, onde o tipo base é mapeado para a tabela de base definido no banco de dados. Cada tipo derivado deve ser mapeada para sua tabela específica no banco de dados. Além disso, é necessário excluir a propriedade de chave primária na entidade derivada gerada pelo *designer* e mapear a coluna de chave primária na entidade derivada da chave entidade definida na classe base (HIRANI, 2010).

No modelo *Table per Concrete Class*, cada tabela representa a entidade inteira. Não é necessário que duas tabelas que participam na *Table per Type* tenham o mesmo número de colunas. As colunas que são específicas para uma tabela que não está em outra tabela que participam na *Table per Type*, acabaria como uma propriedade sobre a entidade derivada. O restante das colunas seriam colocadas como propriedades da entidade de base. O tipo *Table per Concrete Class* não é totalmente suportado no *designer* para que se comece com a importação do modelo e criar o modelo conceitual, mas para utilizar a *Table per Concrete Class* deve-se editar manualmente o arquivo *WebConfig.xml*. Uma das razões pelas quais se cria a *Table per Concrete Class* é retratar os dados provenientes de várias tabelas como sendo uma única entidade recuperando dados de uma única tabela. Isto significa que a chave primária ou entidade chave no modelo conceitual não pode ser repetida. Não se deve ter uma chave primária na tabela 1 e chave primária de 1 na tabela 2, porque isso faria o *Entity Framework* jogar restrição de violação de chave primária (HIRANI, 2010).

3.7 MAPEANDO OPERAÇÕES CRUD EM STORED PROCEDURES

Quando uma entidade é arrastada sobre o *designer* EDM, ela fica fora do suporte de a inserções, atualizações e exclusões. No entanto, se as exigências forem escrever procedimentos armazenados para executar operações CRUD, o *Entity Framework* fornece várias opções para mapear as operações definidas para *stored procedures* no banco de dados (HIRANI, 2010).

O seguinte exemplo pode ser levado em consideração para melhor entender o mapeamento: foram criados *stored procedures* para inserir, atualizar e excluir categorias no banco de dados. É necessário ter certeza de que quando uma categoria é inserida, atualizada ou excluída usando o EDM, o *Entity Framework*, em vez de gerar uma instrução SQL dinâmica para realizar a operação, deve usar procedimentos armazenados para executar a operação.

A solução é usar instruções SQL dinâmicas geradas pelo EF, pois este é menos um código que há necessidade de se manter, porque não deixa o *framework* identificar o CRUD. Os *stored procedures* contrariam ao oferecer melhor desempenho, pois eles são compilados. Além disso, um aplicativo pode não ter os privilégios diretos para inserir, atualizar ou excluir diretamente na tabela e requer permissão DBA para o uso de *stored procedures* para executar operações em uma tabela (HIRANI, 2010).

Uma das outras razões para usar uma *stored procedure* é para aplicar a segurança do banco de dados de nível onde, dependendo dos privilégios de um usuário, ele pode não ter privilégios para excluir itens de uma tabela, mas teria a capacidade de inserir. Para utilizar os *stored procedures* definidos no banco de dados para efetuar CRUD, é necessário importar os procedimentos para o modelo SSDL e usar o diálogo de mapeamento de *stored procedures* para inserir mapas, atualizações e exclusões para uma entidade (HIRANI, 2010).

Atualmente o EF exige que todas as operações de CRUD sejam feitas usando *stored procedures* ou deixa que o *framework* lide com o CRUD. Não há meio termo, onde se pode usar *stored procedures* para inserções e atualizações e o *delete* ser feito usando Entity Framework (HIRANI, 2010).

3.8 LINQ TO ENTITIES

LINQ to Entities é um dialeto do *LINQ* (*Language Integrated Query),* e permite que o usuário realize consultas nas tabelas mapeadas pelo modelo de entidades com o uso da mesma linguagem utilizada para desenvolver a lógica de negócio da aplicação. Uma grande vantagem do seu uso é a não necessidade de conhecimento dos nomes das tabelas, nem de seus atributos na base de dados, já que basta utilizar as classes geradas pelo *Entity Data Model* (EDM) (JENNINGS, 2008).

Para executar uma consulta *LINQ to Entities* no Entity Framework, a consulta deve ser convertida para uma árvore de comando que pode ser executada.

LINQ to Entities é composto de operadores de consulta *LINQ* padrões (como *Select, Where e GroupBy*) e expressões (x > 10, List.Last(), e assim por diante).

Os tipos *IQueryable<T>* e *IOrderedQueryable<T>* são o resultado de uma consulta LINQ que suporta polimorfismo, otimização e geração de consulta dinâmica e, ao contrário do que muitos imaginam, eles não são listas. No entanto, é possível invocar o método *IQueryable<T>.ToList()* para converter a seqüência de um tipo *List<T>*. O tipo *IorderedQueryable<T>* é necessário para suportar as expressões de consulta que incluem *orderby* (JENNINGS, 2008).

Operadores *LINQ* não são definidos por uma classe, mas são métodos em uma classe. Expressões podem conter qualquer coisa permitida por tipos no namespace *System.Linq.Expressions* e, por extensão, qualquer coisa que pode ser representada em uma função. Este é um superconjunto das expressões que são permitidas pelo *Entity Framework*, que, por definição, são restritos a operações permitidas no banco de dados (MSDN, 2010b).

3.9 ENTITY SQL

Entity SQL é o outro dialeto suportado pelo *Entity Framework* para consultas na base de dados.

Desde que o EDM (*Entity Data Model*) introduziu o modelo de mapeamento baseado no modelo de entidades e relacionamentos, é necessária a existência de uma linguagem que permita escrever consultar em termos de abstrações do *Framework*. O *Entity SQL* foi projetado para atender essa necessidade (SHIELDS & VALBUENA, 2007).

Ele proporciona uma transição natural para desenvolvedores SQL. Mesmo sendo o *Entity SQL* fortemente influenciado pelo SQL, ele não é puramente SQL. Vale ressaltar que EntitySQL não impede ninguém de escrever consultas SQL clássicas (JENNINGS, 2008).

O *Entity SQL* utiliza basicamente as consultas SQL tradicionais (*select, from, where, group by, order by*). As expressões foram obtidas a partir do SQL com poucas diferenças, mas a estrutura geral é semelhante. Uma das primeiras coisas que pode-se notar é que *Entity SQL* não tem "SELECT *" - expressões *Select* devem ser declaradas explicitamente (SHIELDS & VALBUENA, 2007).

4 ESTUDO DE CASO

Para apresentar algumas técnicas quanto ao desenvolvimento de uma aplicação acessando o banco de dados o SQL Server 2008, que utilize e coloque em prática todo o estudo feito até aqui, foi elaborado um estudo de caso.

4.1 CONFIGURAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento da aplicação utiliza as seguintes ferramentas:

- Ferramenta de desenvolvimento *Microsoft Visual Studio 2010 Ultimate*, para implementação da aplicação utilizando ASP.NET MVC.
- Microsoft SQL Server 2008, para a criação da base de dados a ser mapeada pelo *Entity Framework.*

4.2 CONTEXTUALIZAÇÃO DA APLICAÇÃO

Optou-se pelo desenvolvimento de uma aplicação ASP.NET MVC 2, que faça um controle de bens patrimoniais de uma determinada empresa, utilizando-se o Microsoft Visual Studio 2010 e com a base de dados SQL Server 2008.

4.3 DESENVOLVIMENTO DA APLICAÇÃO

4.3.1 Geração da Base de Dados no SQL Server 2008

O *Visual Studio 2010* fornece uma ferramenta para criação de tabelas para armazenamento de dados no *SQL Server.*

Como exibido ver na **Figura 6**, a geração do DER (Diagrama de Entidades/Relacionamentos) é o primeiro passo na criação das tabelas do banco de dados.

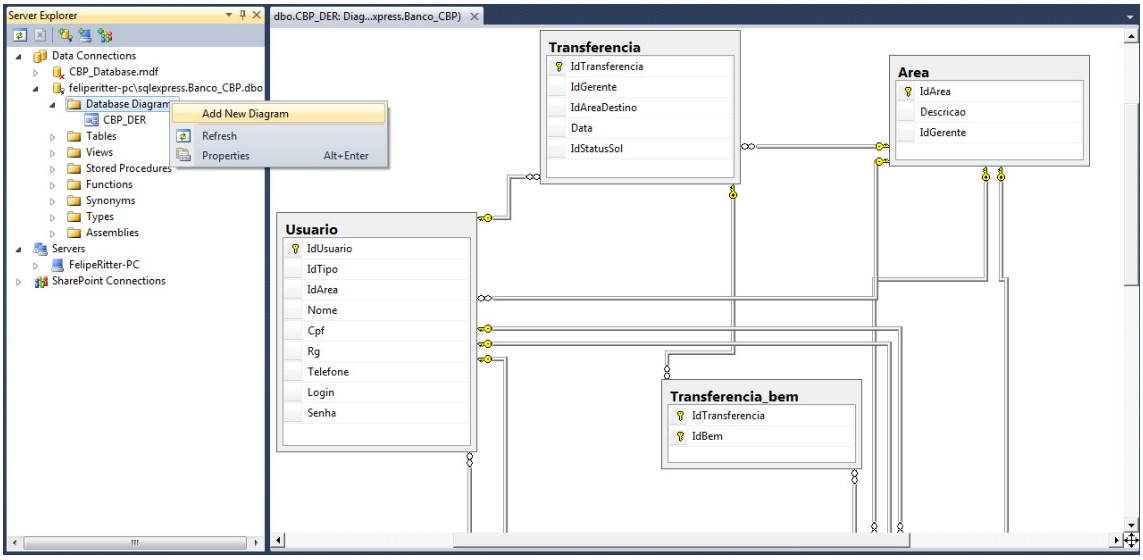

Figura 6 - Criação do DER

O *SQL Server* "entende" os diagramas gerados como sendo um mapa para a criação de tabelas, e automaticamente, cria a base de dados de acordo com o DER, como pode ser visto na **Figura 7.**

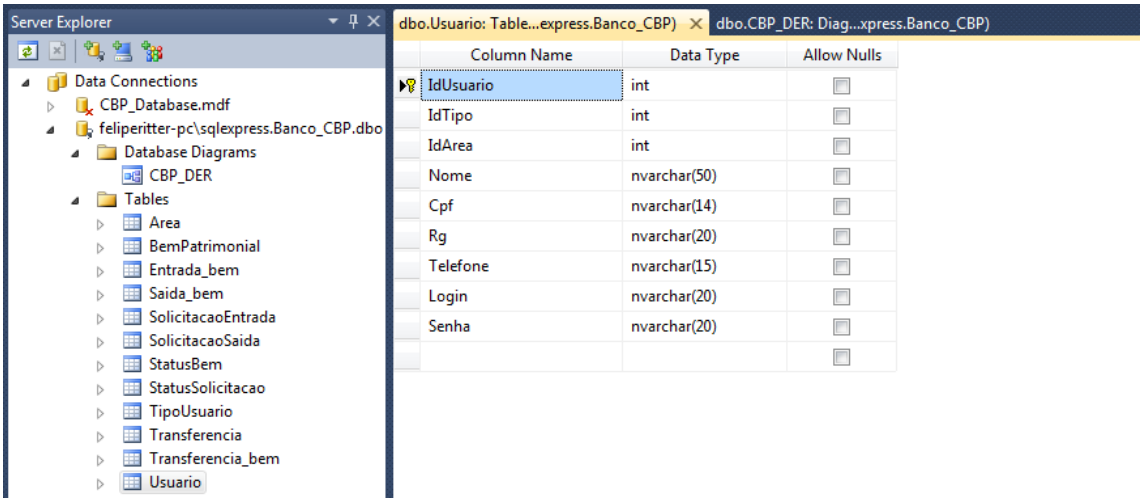

Figura 7 - Tabelas Geradas

Todas as chaves primárias, os relacionamentos, os tipos de cada coluna e a validação de dados nulos, são automaticamente criados pelo *SQL Server*.

4.3.2 Aplicação ASP.NET MVC 2 no Visual Studio 2010

A ferramenta de desenvolvimento *Visual Studio 2010* gera toda a arquitetura de pastas de um projeto ASP.NET MVC 2, e o deixa pronto para desenvolvimento, poupando boa parte do serviço do programador.

Neste estudo de caso, a aplicação divide-se em duas partes: Negócio e *Web*.

A parte *Web* compreende aos controladores (*controllers*) e às páginas (*views*), enquanto o Negócio compreende a todas as regras de negócio e acesso à dados do sistema, substituindo, assim, a parte do modelo (*model*).

A partir do momento em que os projetos são criados, a principal configuração a ser feita é referenciar o projeto Negócio no projeto *Web*.

A **Figura 8** mostra como fica a arquitetura de um projeto ASP.NET MVC 2 divido em duas partes.

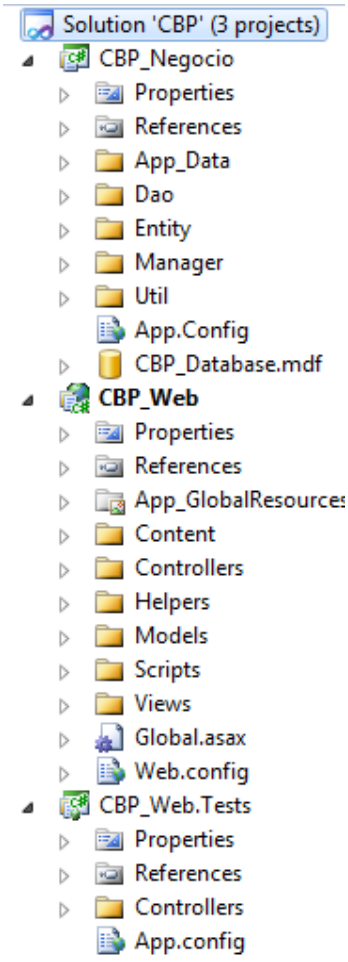

Figura 8 - Arquitetura da Aplicação

4.3.3 Entity Data Model

Ao criar um arquivo do ADO.NET Entity Framework na aplicação o sistema mapeia o banco de dados criado no *SQL Server* e gera as classes referentes às respectivas tabelas, para que a aplicação possa utilizá-las.

A base de dados utilizada é selecionada no momento da criação do arquivo e o resultado é mostrado nas **Figuras 9 e 10**, que é o arquivo *Entity Data Model* (.edmx), além do arquivo (*.Designer.cs*) que contém as classes mapeadas**.**

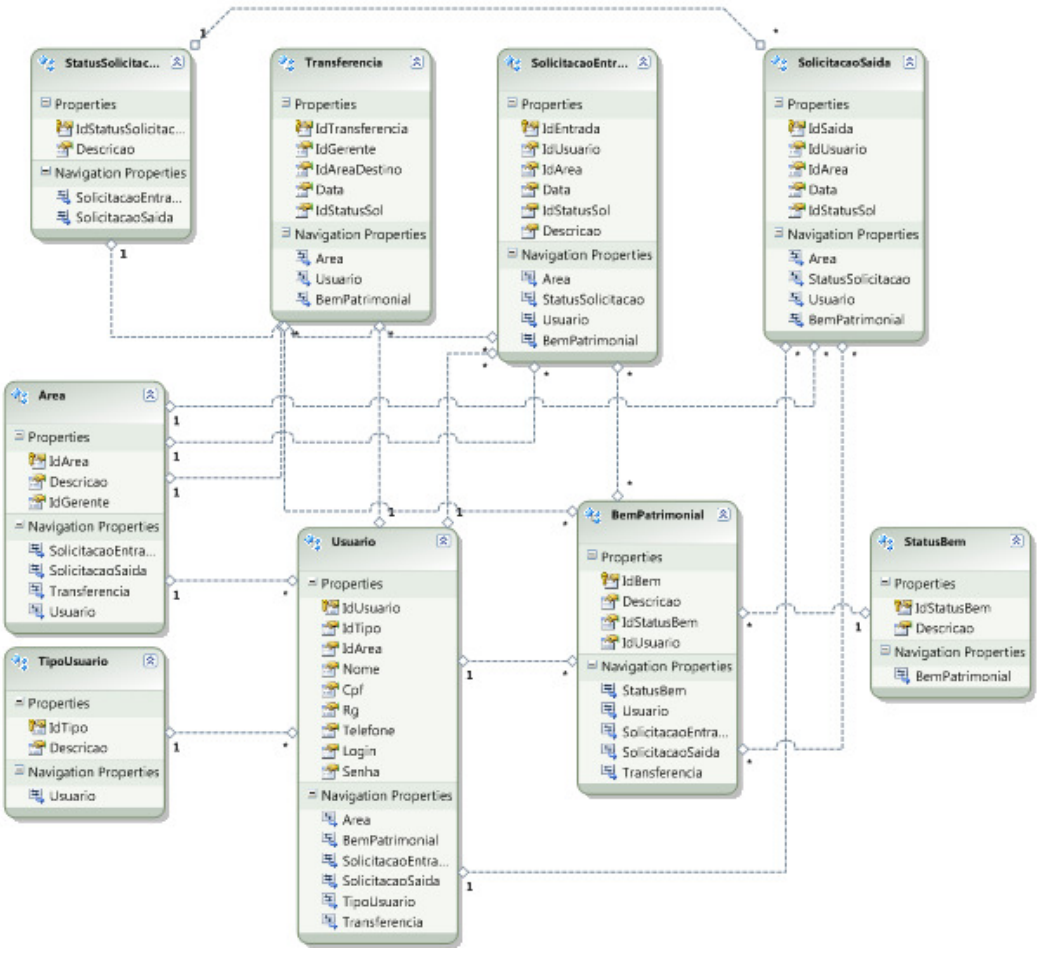

Figura 9 - Entity Data Model

```
[assembly: EdmSchemaAttribute()]
EDM Relationship Metadata

namespace CBP.Entity
{
    #region Contexts

    public partial class CBP_DatabaseEntities : ObjectContext
    {
        Constructors

        Partial Methods

        ObjectSet Properties
        AddTo Methods
    }
    #endregion

    #region Entities

    /// <summary>
    /// No Metadata Documentation available.
    /// </summary>
    [EdmEntityTypeAttribute(NamespaceName="Entity", Name="Area")]
    [Serializable()]
    [DataContractAttribute(IsReference=true)]
    public partial class Area : EntityObject...
```

Figura 10 - Classe Designer do EDM

Como pode ser notado, o número de tabelas mostradas no EDM é inferior ao número no *SQL Server*. Isso acontece porque o Entity Framework entende as classes associativas, aquelas que são criadas entre duas tabelas onde há um relacionamento *many-to-many*, como inúteis para a aplicação, já que na orientação a objetos esse relacionamento é possível de ser utilizado.

Os relacionamentos entre tabelas no banco de dados são transformados em relacionamentos entre objetos na aplicação. Como exemplo, o relacionamento mostrado no DER da **Figura 6** entre *Usuario* e *Area*, o EF4 monta o objeto *Usuario* contendo uma *Area* como atributo dele e, ao montar o objeto *Area*, cria uma lista de objetos *Usuario* como atributo.

Além das classes e relacionamentos mapeados com o banco de dados, a classe *Designer* gera um *ObjectContext* que será o objeto utilizado pela aplicação para o acesso à base de dados.

4.3.4 Classes de Acesso a Dados (DAO)

As classes conhecidas como DAOs, são responsáveis, em um sistema, pela conversação entre aplicação e base de dados.

Essas classes realizam todas as operações de um CRUD e seus métodos são única e exclusivamente criados para tais funções.

Com o uso do ADO.NET Entity Framework 4, os métodos dessas classes se tornam muito simples de se desenvolver. Temos como exemplo os cadastros do sistema de bens patrimoniais.

Três das quatro operações de um CRUD, são idênticas para qualquer tipo de objeto em uma aplicação com EF4, então, para facilitar, é geralmente criado um DAO genérico, que só conhece o tipo de entidade que irá lidar no momento em que é instanciado. Como mostra, em linguagem C#, o **QUADRO 1.**

```
/// <typeparam name="TEntity">Entidade cuja classe Dao interage</typeparam>
/// <typeparam name="TObjectContext">ObjectContext do EDM gerado</typeparam>
public abstract class GenericDao<TEntity, TObjectContext> : IGenericDao<TEntity>
    where TEntity : EntityObject
    where TObjectContext : ObjectContext
{
    /// <summary>
    /// Objeto que representa o banco de dados
    /// </summary>
    protected TObjectContext _objectContext;
```

QUADRO 1 - Generic DAO

Como se pode notar, a classe, quando declarada, recebe um *EntityObject,* que nesse sistema pode ser, por exemplo, um *Usuario*, ou um *BemPatrimonial*, e um *ObjectContext,* que é gerado no arquivo *designer* criado junto ao *Entity Data Model*. O construtor da classe é exibido no **QUADRO 2.**

```
/// <param name="objectContext">ObjectContext gerado no EDM</param>
/// <param name="entitySetName">Nome da entidade do banco de dados</param>
public GenericDao(TObjectContext objectContext, ObjectSet<TEntity> entity)
{
    _entitySetName = entity.EntitySet.Name;
    _objectContext = objectContext;
    _containerName = objectContext.DefaultContainerName;
    _fullEntitySetName = ((new StringBuilder())
                          .Append(_containerName)
                          .Append(".").Append(_entitySetName)).ToString();
}
```

QUADRO 2 - Construtor da Classe DAO

Esse construtor, seta as variáveis da classe de acordo com o *EntityObject* e o *ObjectContext* passado no momento que a classe é instanciada, como é o caso da variável *_entitySetName*, que recebe o nome da tabela na base de dados que é mapeada pelo objeto entidade.

No caso de um *Insert*, cuja função é gravar o objeto na base de dados relacional, teremos o método exibido no **QUADRO 3.**

```
/// <summary>
/// Metodo para inserir um registro no banco de dados
/// </summary>
/// <param name="entity">Objeto para inserir</param>
/// <returns>Objeto inserido</returns>
public TEntity Insert(TEntity entity)
{
    try
    {
        _objectContext.AddObject(_entitySetName, entity);
        _objectContext.SaveChanges();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }

    return entity;
}
```

QUADRO 3 - Método Insert

Estudando o código, é notável que basta adicionar o objeto ao *ObjectContext* e salvar as alterações com o método *SaveChanges*() que ele é automaticamente gravado no banco.

Os métodos *Update* e *Delete* são bem parecidos ao *Insert,* a única diferença são os métodos chamados. O **QUADRO 4** mostra ambos.

```
/// <summary>
/// Metodo para atualizar um registro no banco de dados
/// </summary>
/// <param name="entity">Objeto para atualizar</param>
/// <returns>Objeto atualizado</returns>
public TEntity Update(TEntity entity)
{
    try
    {
        _objectContext.ApplyCurrentValues(_entitySetName, entity);
        _objectContext.SaveChanges();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
```

```
    return entity;
}
/// <summary>
/// Metodo para excluir um registro no banco de dados
/// </summary>
/// <param name="entity">Objeto para excluir</param>
public void Delete(TEntity entity)
{
    try
    {
        _objectContext.AttachTo(_entitySetName, entity);
        _objectContext.DeleteObject(entity);
        _objectContext.SaveChanges();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
}
```

QUADRO 4 - Métodos Update e Delete

Essas formas de realizar operações CRUD facilitam muito o trabalho dos desenvolvedores e, principalmente, não há preocupação em abrir e fechar conexões com o banco de dados.

Existe, porém, o caso de um *Insert* onde há um relacionamento *many-to-many*, como é o caso, nesse sistema, de um usuário desejar solicitar a entrada de um conjunto de bens patrimoniais com o fim de utilizá-los em seu trabalho.

Como um objeto do tipo *BemPatrimonial* possui um conjunto de objetos *SolicitacaoEntrada*, e vice-versa, na base de dados existe uma tabela entre esse relacionamento, mas no EDM não.

O **QUADRO 5** demonstra quais são os passos a se seguir para que seja possível inserir um objeto *SolicitacaoEntrada* contendo uma lista de objetos do tipo *BemPatrimonial* no banco de dados.

```
public SolicitacaoEntrada solicitarEntrada(SolicitacaoEntrada entity,
List<BemPatrimonial> bens)
{
    try
    {
        this.Verifica(entity);

        using (TransactionScope transaction = new TransactionScope())
        using (CBP_DatabaseEntities e = new CBP_DatabaseEntities())
        {
                SolicitacaoEntradaDao solicitacaoDao = new SolicitacaoEntradaDao(e);
                if (entity.IdEntrada > 0)
                {
                    SolicitacaoEntrada solAux = solicitacaoDao.FindById(entity);
```

```
                    solAux = entity;
                    solAux.IdStatusSol = 1;

                    if (solAux != null)
                        entity = solicitacaoDao.Update(solAux);
                }
                else
                {
                    entity.IdStatusSol = Solicitacao.ENVIADA;
                    entity = solicitacaoDao.Insert(entity);
                }


                BemPatrimonialDao bemManager = new BemPatrimonialDao(e);


                foreach (var bem in bens)
                {

                    entity.BemPatrimonial.Add(bemManager.FindById(bem));
                    entity = solicitacaoDao.Update(entity);
                }


                transaction.Complete();

            }
        }
        catch (Exception e)
        {
            throw new Exception(e.Message);
        }

        return entity;
}
```

QUADRO 5 - Insert com Relacionamento Many-to-Many

Esse método realiza tanto o *Insert* quanto o *Update* na tabela *SolicitacaoEntrada* e como visto, é necessário primeiramente inserir ou atualizar a solicitação, e depois percorrer a lista de bens (*List<BemPatrimonial>*) realizando uma pesquisa em cada bem, mesmo sabendo que os bens em questão existem na base de dados, pois isso os colocará no mesmo contexto (*ObjectContext*) da solicitação, podendo assim, adicionar esses bens na lista e realizar o *Update* no objeto *SolicitacaoEntrada*.

Feito isso, o *Entity Framework* atualiza automaticamente a tabela associativa desse relacionamento na base de dados, facilitando muito o trabalho, se comparado ao feito sem o uso desse artifício.

Outro benefício do método mostrado no **QUADRO 5** é o uso de uma *TransactionScope*.

Essa classe abre uma transação para que o EF4 realize as modificações no banco de dados e caso ocorra algum problema durante o procedimento, os dados não ficam

corrompidos no banco, pois só acontece o co*mmit* para salvar as alterações quando é completada a transação (*transaction.Complete*()).

Os métodos acima citados fazem parte da classe genérica para acesso ao banco de dados, mas existe uma operação do CRUD que não consta nela: o *Select (Retrieve).*

Essa operação não pode ser genérica, por existir dados específicos de objetos que podem fazer parte das consultas e, para isso, são criadas classes que herdam da classe genérica, para poderem lidar com essas diferenças.

Com o uso de *LINQ to Entities* é possível fazer várias formas de pesquisa na base de dados e a mais básica delas, é aquela que não recebe parâmetro algum, ou seja, busca todos os objetos de uma determinada tabela.

Uma pesquisa desse tipo pode ser feita de maneiras diferentes, os **QUADROS 6 e 7** mostram duas delas, com a entidade *Usuario*.

```
public override List<Usuario> FindAll()
{
    try
    {
        IQueryable<Usuario> query = _objectContext.Usuario.AsQueryable<Usuario>();

        return query.ToList();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
}
```

QUADRO 6 - Método FindUsuario Tipo 1

```
public override List<Usuario> FindAll()
{
    try
    {
        return (from Usuario u in _objectContext.Usuario
                select u).ToList();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
}
```

QUADRO 7 - Método FindUsuario Tipo 2

As duas maneiras retornam o mesmo resultado, a diferença entre ela é que, enquanto a segunda é mais simples de se utilizar nesse contexto, a primeira facilita quando existem parâmetros para a pesquisa, e esses parâmetros nem sempre são obrigatórios.

Assim, faz-se necessária uma validação e a adição de cláusulas *Where* na consulta e, utilizando o método do **QUADRO 6**, temos o resultar exibido no **QUADRO 8**, onde a intenção é retornar uma lista de usuários (*List<Usuario>*), onde os objetos da lista respondam a algumas condições que são passadas ao método em um objeto do tipo *Usuario*.

```
public override List<Usuario> Find(Usuario entity)
{
    try
    {
        IQueryable<Usuario> query = _objectContext.Usuario.AsQueryable<Usuario>();

        if (entity.IdUsuario > 0)
            query = query.Where<Usuario>(u => u.IdUsuario == entity.IdUsuario);
        if (!String.IsNullOrEmpty(entity.Nome))
            query = query.Where<Usuario>(u => u.Nome.Contains(entity.Nome));
        if (!String.IsNullOrEmpty(entity.Cpf))
            query = query.Where<Usuario>(u => u.Cpf.Contains(entity.Cpf));
        if (!String.IsNullOrEmpty(entity.Rg))
            query = query.Where<Usuario>(u => u.Rg.Contains(entity.Rg));
        if (!String.IsNullOrEmpty(entity.Telefone))
            query = query.Where<Usuario>(u => u.Telefone.Contains(entity.Telefone));
        if (!String.IsNullOrEmpty(entity.Login))
            query = query.Where<Usuario>(u => u.Login.Contains(entity.Login));
        if (entity.IdArea > 0)
            query = query.Where<Usuario>(u => u.IdArea == entity.IdArea);

        return query.ToList();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
}
```

QUADRO 8 - Método FindUsuário com Parâmetros

Como visto, existem várias validações para os parâmetros passados para o método e, caso um não seja nulo, ele é adicionado à variável *IQueryable* com o método *Where*().

Outra coisa notável no método do **QUADRO 8,** é que na declaração da variável que recebe a consulta, existe um operador do tipo *Include*("*Area*"). Isso faz com que o objeto *Area* que existe como atributo do objeto *Usuario* venha da pesquisa preenchido com a respectiva *Area* relacionada a ele no banco de dados.

Às vezes, porém, existem consultas na base de dados que esperam parâmetros obrigatórios para serem realizadas, ou seja, caso seja passado um valor nulo, a mesma não retornará valor algum.

Neste caso, é mais aconselhável a utilização da pesquisa na forma como é feita no **QUADRO 7**, com adição da cláusula *Where*, já que o código do **QUADRO 8** busca, primeiramente, todos os dados da tabela, para depois filtrá-los de acordo com os parâmetros passados para o método. Assim, é possível ganhar muito em desempenho, pois existem ocasiões que uma tabela possui um grande volume de dados a ser carregado.

O **QUADRO 9** mostra como é feita uma pesquisa de usuários onde o parâmetro *Login*, que é um atributo do objeto *Usuario*, é obrigatório.

```
public Usuario FindByLogin(Usuario entity)
{
    try
    {
        return (from Usuario u in _objectContext.Usuario
                where (u.Login == entity.Login)
                select u).SingleOrDefault();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
}
```

QUADRO 9 - Consulta com Parâmetros Obrigratórios

Lembrando que uma consulta dessa forma retorna um objeto do tipo *IQueryable*. Mas como o campo *Login* é único para cada usuário, sabe-se que essa consulta somente retornará um objeto, por isso existe um operador do tipo *.SingleOrDefault*(), que, ao invés de retornar uma lista de resultados, retorna somente um. Esse método gera um Exception quando a consulta retorna mais de um objeto, então o desenvolvedor deve tomar cuidado ao utilizá-lo.

Existe, porém, um caso específico nas passagens de parâmetros para as consultas, que é quando se tem um relacionamento *many-to-many*.

Nesse caso, um exemplo seria uma transferência de bens de uma área para outra.

Um objeto *BemPatrimonial* pode pertencer a vários objetos *Transferencia*, e vice-versa. Então cada um possui uma *List*<> do outro como atributo.

Uma consulta que retorne todos os objetos da tabela *BemPatrimonial*, deverá percorrer os objetos *Transferencia* das listas de cada bem, procurando um dado que corresponda ao parâmetro especificado.

Aparentemente seria trabalhoso escrever um código em C# que fizesse isso, mas o **QUADRO 10** mostra o quanto simples é realizar esse tipo de consulta.

```
public List<BemPatrimonial> FindByTransferencia(Transferencia transferencia)
{
    try
    {
        IQueryable<BemPatrimonial> query = _objectContext.BemPatrimonial
                                           .Include("Transferencia")
                                           .AsQueryable<BemPatrimonial>();

        query = query.Where<BemPatrimonial>(a => a.Transferencia
                                           .Any(s => s.IdTransferencia
                                           == transferencia.IdTransferencia));

        return query.ToList();
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
}
```

QUADRO 10 - Consulta Many-to-Many

O método *Any*() retorna um valor *boolean* (verdadeiro ou falso) e percorre a variável que contém a lista de objetos *Transferencia* de acordo com um determinado parâmetro, como no **QUADRO 10**, em que serão retornados somente os bens patrimoniais que possuírem uma transferência cujo código (*IdTransferencia*), que é a chave primária na tabela, seja igual ao passado por parâmetro para o método *FindByTransferencia*().

4.3.5 Controllers

As classes *Controllers* são o "cérebro" da aplicação, já que é por elas que passa todas as requisições das páginas, e é nelas também que estão as chamadas para classes que possuem as regras de negócio do sistema.

As *Controllers* estão localizadas no projeto *Web,* na pasta *Controllers* e o **QUADRO 11** mostra uma dessas classes.

```
public class BemController : Controller
{
    public ActionResult Inserir(){
        return View();
    }
    public ActionResult Consultar()
    {
```

```
        return View();
    }

    [HttpPost]
    public JsonResult Consultar(GridSettings grid, BemPatrimonial vBemp)...

    [HttpPost]
    public JsonResult Salvar(BemPatrimonial bem)...

    public ActionResult Editar(int id)
    {
        IManager<BemPatrimonial> bemManager = new BemPatrimonialManager();
        BemPatrimonial bem = new BemPatrimonial();
        bem.IdBem = id;
        bem = bemManager.FindById(bem);
        UsuarioManager usuarioManager = new UsuarioManager();
        Area area = new Area();
        area.IdArea = ((Usuario)Session["UsuarioLogado"]).IdArea;
        IEnumerable<Usuario> usuarioList =
usuarioManager.FindByArea(area).AsEnumerable<Usuario>();
        ViewData["UsuarioList"] = new SelectList(usuarioList, "IDUSUARIO", "NOME");
        return View(bem);
    }
}
```

QUADRO 11 - BemController

Como visto, essa é a *Controller* responsável pela parte que envolve os bens patrimoniais no sistema.

Analisando o código, pode-se definir que os métodos *Inserir(), Consultar()* e *Editar(int id)* são os chamados métodos *GET*, por não possuírem anotação e por retornarem um *ActionResult*, ou seja, todos esses métodos, quando chamados, exibem suas respectivas *Views* (páginas).

Em ASP.NET MVC, as *Views* normalmente tem um modelo, que nada mais é do que um objeto, cujos valores dos atributos podem preencher campos na tela, ou então, esses campos podem preenchê-los, como é o caso de uma página de cadastro.

Os métodos *Inserir*() e *Consultar*(), como mostrados no **QUADRO 11**, não enviam nenhum dado para a *View*, ou seja, o modelo dessa página não é preenchido quando ela é exibida ao usuário.

Por outro lado, o método *Editar(int id)* interage com o usuário na sua primeira chamada, já que ele espera um parâmetro e este é usado para preencher o modelo da *View* de acordo com as regras de negócio chamadas pelo método.

Existem duas maneiras de passar para a página o valor do modelo, uma delas é passando o objeto na chamada da *View* (*return View(Object)*), e a outra é instanciando o

atributo *Model* contido no objeto *ViewData* (*ViewData.Model = Object*), que, traduzindo para o português, são os dados da página.

Além do modelo, o método *Editar*() envia outros dados para a sua respectiva *View*, que, de acordo com o código, são passados através do objeto *ViewData["UsuarioList"]*.

Os métodos que possuem a anotação *[HttpPost]*, são aqueles que, quando chamados, não necessariamente retornam uma *View,* mas executam regras de negócio dentro da própria página, como é o caso do método *Consultar* anotado, que, mesmo com o mesmo nome do anterior, ao invés de redirecionar para a página *Consultar.aspx*, realiza as consultas necessárias para exibir os dados da tabela na tela.

Como todas as ações realizadas nas *Views* passam pelas suas respectivas *Controllers*, é fácil controlar o que acontece na aplicação e, basicamente, o que se deve controlar é a *URL* que é passada no *POST*. Como exemplo, temos o **QUADRO 12**, que mostra o código do método *Salvar*() da *Controller* de usuários, cuja ação do botão "salvar" gera um *submit* passando a seguinte *URL*: "http://servidor:porta/Usuario/Salvar". Essa ação significa que será chamado o controlador *UsuarioController* e o método Salvar() anotado com *[HttpPost]*.

```
[HttpPost]
public JsonResult Salvar(Usuario usuario)
{
    try
    {
        IManager<Usuario> usuarioManager = new UsuarioManager();
        usuario.IdTipo = TipoUser.FUNCIONARIO;


        usuario = usuarioManager.Save(usuario);
    }
    catch (Exception e)
    {
        Response.Write(e.Message);
        return new JsonResult();
    }

    return new JsonResult();
}
```

QUADRO 12 - Método Salvar Usuário

Na *View*, todos os campos são do tipo *Html.TextBoxFor* que obrigatoriamente devem receber o valor de um atributo do modelo da página.

Como o evento do botão "salvar" gera um *submit*, que deve chamar o método anotado por *[HttpPost]* na *Controller*. Esse método espera receber um objeto *Usuario* como parâmetro

e como os campos da página possuem os valores de atributos dessa classe, o objeto passado ao método é automaticamente carregado com eles.

O método *Salvar*() retorna um *JsonResult*, porque a ação do botão na página foi escrita com o uso de *JQuery*[5], e esse método não gera um *redirect*.

Esse controlador realiza somente o que é de seu dever, como se pode ver, ele só controla o fluxo entre *View* e *Model*, as regras de negócio só serão aplicadas no projeto Negócio.

4.3.6 Criação das Views

O ASP.NET MVC facilita muito na criação das *Views*, dando ao programador o auxilio necessário para rapidamente desenhar as telas necessárias para a aplicação.

Após criadas as *Controllers*, basta alguns cliques com o mouse para gerar a página no modelo que o desenvolvedor deseja, seja ela para um cadastro (*create*), uma edição (*edit*), uma consulta (*list*), uma página que mostre os detalhes do objeto (*details*) ou uma página em branco (*empty*), onde cabe ao desenvolvedor montá-la do zero. E a única preocupação que se tem é o estilo, para "embelezar" as telas de acordo com o desejo do cliente ou então adicionar certas funcionalidades à página conforme preferir.

Além disso, o desenvolvedor pode, também, optar por criar as páginas por conta própria, gerando o pacote com o nome do controlador dentro do pacote das *Views* e adicionando uma *View* com o mesmo nome contido no método da *Controller.*

A **Figura 11** mostra a geração de uma *View* a partir da *Controller*, enquando no **QUADRO 13** é possível ver a página gerada pelo ASP.NET MVC.

---

[5] *JQuery* – É um *framework* para auxiliar no desenvolvimento com *Javascript* (SILVA, 2008).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using CBP.Entity;
using CBP_Web.Models.Grid;
using CBP.Manager;
using CBP.Util;

namespace CBP_Web.Controllers
{
    public class UsuarioController : Controller
    {
        public ActionResult Create()
        {
            IManager<Area> areaManager = new AreaMana

            IEnumerable<Area> areaList = areaManager.
            ViewData["AreaList"] = new SelectList(are

            return View();
        }

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

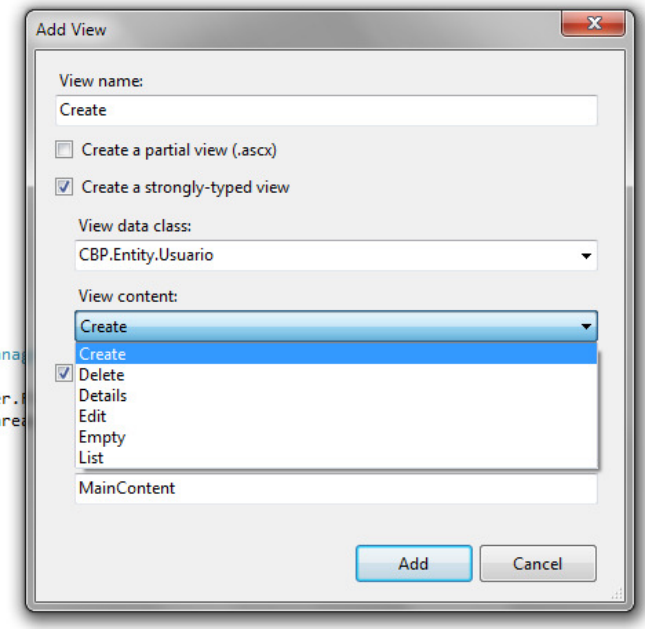

Figura 11 - Criação da View

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Create</h2>
    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>
        <fieldset>
            <legend>Fields</legend>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.IdUsuario) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.IdUsuario) %>
                <%: Html.ValidationMessageFor(model => model.IdUsuario) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.IdTipo) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.IdTipo) %>
                <%: Html.ValidationMessageFor(model => model.IdTipo) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.IdArea) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.IdArea) %>
                <%: Html.ValidationMessageFor(model => model.IdArea) %>
            </div>
```

QUADRO 13 - View Gerada a Partir da Controller

Observando o código gerado, é possível ver que a *View* já é criada de acordo com o objeto selecionado para ser o seu modelo (*ViewData.Model*) e cada campo é criado de acordo com os respectivos atributos dele com o uso de *Html.TextBoxFor*.

O ASP.NET MVC também gera automaticamente validações de campos obrigatórios. Isso se faz durante a criação do MER, no banco de dados, pois um campo definido como *not null* é dado como obrigatório pelo *Entity Framework*.

É possível ver na página criada as *tags* *Html.ValidationMessageFor*, que, automaticamente, geram uma mensagem de erro caso os campos a serem validados não tenham sido preenchidos.

Outro exemplo é a criação de uma *View* de consulta, onde, como no **QUADRO 13**, o modelo é um usuário.

Esse tipo de *View* é chamado de *List* e, no momento da criação, o *framework* MVC gera uma página que espera, ao invés de um só objeto, uma lista. Com isso é montado uma tabela contendo um comando do tipo *foreach* que percorre os dados enviados pela *Controller* e os exibe para o usuário, como mostra o **QUADRO 14.**

```
<% foreach (var item in Model) { %>

    <tr>
        <td>
            <%: Html.ActionLink("Edit", "Edit", new { id=item.IdUsuario }) %> |
            <%: Html.ActionLink("Details", "Details", new { id=item.IdUsuario })%> |
            <%: Html.ActionLink("Delete", "Delete", new { id=item.IdUsuario })%>
        </td>
        <td>
            <%: item.IdUsuario %>
        </td>
        <td>
            <%: item.IdTipo %>
        </td>
        <td>
            <%: item.IdArea %>
        </td>
        <td>
            <%: item.Nome %>
        </td>
        <td>
            <%: item.Cpf %>
        </td>
        <td>
            <%: item.Rg %>
        </td>
        <td>
            <%: item.Telefone %>
        </td>
```

QUADRO 14 - List View

Esse tipo de *View* pode ser aprimorado adicionando campos que servirão de parâmetros para uma consulta, o que alterará a lista de objetos no modelo da página.

A **Figura 12** mostra, em execução, a *grid* de consulta de bens patrimoniais preenchida pela lista enviada pela *Controller*.

Figura 12 - Grid de Consulta

Analisando a *grid*, nota-se como os campos são preenchidos através do comando *foreach* gerado na *List View*.

Como se pode ver, o ASP.NET MVC auxilia de várias maneiras na geração das *Views* de acordo com o desejo do programador. Toda página gerada está pronta para uso e só necessita de alterações se este desejar adicionar funcionalidades a ela.

As páginas do tipo *Strongly-type View*, auxiliam muito no desenvolvimento do sistema, já que o desenvolvedor mapeia facilmente o objeto desejado, podendo realizar operações CRUD de forma simples e rápida.

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

O *Framework .NET* oferece várias soluções para os problemas encontrados durante o desenvolvimento, não só em aplicações *Web*, e com a utilização do ASP.NET MVC juntamente com o ADO.NET Entity Framework, pode-se notar benefícios como:

- O *Entity Framework* auxilia na transparência da comunicação entre aplicação e base de dados, fazendo automaticamente a conexão com o banco, além de criar as classes que mapeiam as tabelas do banco, ou vice-versa, caso o desenvolvedor possua o código das classes e não haja a estrutura de tabelas na base de dados.
- O *Entity Framework* também gera um padrão de desenvolvimento que aumenta a produtividade e facilita a manutenção da aplicação, pelo fato de o desenvolvedor não necessitar se preocupar com comandos SQL ou então com o tipo de banco de dados que está sendo utilizado.
- O ASP.NET MVC divide o sistema em camadas e utiliza o padrão *Front Controller,* facilitando a testabilidade e o controle de todas as requisições que ocorrem na aplicação.
- O ASP.NET MVC facilita bastante o redirecionamento de páginas e as passagens de parâmetros entre elas por utilizar um sistema de roteamento de simples entendimento e implementação.

5.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

A sugestão para trabalhos futuros consiste em realizar uma pesquisa sobre tecnologias a serem aplicadas nas *Views* que auxiliem para o aumento no desempenho das mesmas durante a entrada de dados do usuário e lidar com os resultados vindos das *Controllers*, além da possibilidade de implementar medidas de segurança contra invasão a um sistema ASP.NET MVC.

REFERÊNCIAS BIBLIOGRÁFICAS

BALTIERI, André. Descubra o ADO.NET Entity Framework. Disponível em: < http://www.linhadecodigo.com.br/artigo/2920/Descubra-o-ADONET-Entity-Framework.aspx>. 2011. Acesso em: 01/05/2011.

BASSI, Giovani. ASP.NET MVC: Quando Usar. Disponível em: < http://unplugged.giggio.net/post/ASPNet-MVC-Quando-Usar.aspx>. Acesso em: 01/05/2011.

GALLOWAY, J.; HANSELMAN, S. Professional ASP.NET MVC 2. Indianápolis, IN: Wiley, 2010.

GUTHRIE, Scott. ASP.NET MVC Framework (Part 2): URL Routing. Disponível em: < http://weblogs.asp.net/scottgu/archive/2007/12/03/asp-net-mvc-framework-part-2-url-routing.aspx>. 2007. Acesso em: 14/05/2011.

HADDAD, Renato. Por Que Utilizar o Visual Studio 2010?. Disponível em: < http://www.linhadecodigo.com.br/artigo/3064/Porque-adotar-o-Visual-Studio-2010.aspx>. Acesso em: 01/05/2011.

HANSELMAN, Scott. ASP.NET MVC 2 Basics. Disponível em: < http://channel9.msdn.com/blogs/matthijs/aspnet-mvc-2-basics-introduction-by-scott-hanselman>. Acesso em: 28/04/2011.

HIRANI, Zeeshan. Entity Framework Learning Guide. Disponível em: < zeeshanjhirani@gmail.com>. 2010. Acesso em: 29/05/2011.

HIRT, Alan. Pro SQL Server 2008: Failover Clustering. New York, NY: Apress, 2010.

JENNINGS, Roger. ADO.NET 3.5 with LINQ and the Entity Framework. Wiley Publishing, Inc. 2008.

KLEIN, Scott. Pro Entity Framework 4.0. New York, NY: Apress, 2010.

MACORATTI, José Carlos. Usando o Entity Framework 4 com um serviço WCF (C#)**.** Disponível em: < http://www.macoratti.net/11/01/svl4_ef4.htm>. 2010. Acesso em: 10/05/2011.

MICROSOFT. ASP.NET MVC Overview. Disponível em: < http://www.asp.net/mvc/tutorials/asp-net-mvc-overview-cs>. 2010a. Acesso em: 01/05/2011.

MICROSOFT. ASP.NET MVC Routing Overview. Disponível em: < http://www.asp.net/mvc/tutorials/asp-net-mvc-routing-overview-cs>. 2010b. Acesso em: 12/05/2011.

MICROSOFT. Understanding Models, Views and Controllers. Disponível em: < http://www.asp.net/mvc/tutorials/understanding-models-views-and-controllers-cs>. 2010c. Acesso em: 12/05/2011.

MICROSOFT. Visual Studio 2010. Disponível em: < http://www.microsoft.com/business/smb/pt-br/servidores-e-ferramentas/visual-studio-pro.mspx>. 2010d. Acesso em: 01/05/2011.

MSDN. Filtering in ASP.NET MVC. Disponível em: < http://msdn.microsoft.com/en-us/library/gg416513%28VS.98%29.aspx>. 2010a. Acesso em: 03/05/2011.

MSDN. LINQ To Entities. Disponível em: <http://msdn.microsoft.com/en-us/library/bb386964.aspx>. 2010b. Acesso em: 12/05/2011.

MSDN. MVC Framework and Application Structure. Disponível em: < http://msdn.microsoft.com/en-us/library/dd410120%28VS.98%29.aspx>. 2010c. Acesso em: 03/05/2011.

OFICIALDANET. SQL Server. Disponível em: < http://www.oficinadanet.com.br/artigo/501/sql_server>. Acesso em: 01/05/2011.

PALERMO, Jeffrey. ASP.NET MVC Em Ação. São Paulo, SP: Novatec, 2010. 432 p. ISBN 978-85-7522-221-8.

QUAIATO, Vinicius. ASP.NET MVC 3 – Parte 1. Disponível em: < http://www.devmedia.com.br/post-18964-Asp-net-mvc-3-parte-1--net-magazine-81.html>. 2011. Acesso em: 28/05/2011.

RAYMUNDO, Felipe Pedroti. ADO.NET Entity Framework 4. Disponível em http://feliperaymundo.com.br/?p=262. Acesso em 08/03/2011.

SANDERSON, Steven. Pro ASP.NET MVC 2 Framework, Second Edition. New York, NY: Apress, 2010.

SHIELDS, M.; VALBUENA, F. Entity SQL. Disponível em: < http://blogs.msdn.com/b/adonet/archive/2007/05/30/entitysql.aspx>. 2007. Acesso em: 01/05/2011.

SILVA, Maurício Samy. JQuery, A Biblioteca do Programador JavaScript. São Paulo, SP: Novatec, 2008.

TECHNET. Arquitetura ASP.NET. Disponível em: < http://technet.microsoft.com/pt-br/library/cc737863%28WS.10%29.aspx>. 2010. Acesso em: 01/03/2011.

TEMPLEMAN, J.; VITTER, D. Visual Studio .NET: The .NET Framework Black Book. Scottsdale, Arizona: Coriolis, 2007.

TENNY, L.; HIRANI, Z. Entity Framework 4.0 Recipes: A Problem-Solution Approach. New York, NY: Apress, 2010.