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

EDIVALDO NEGIR CHERUBINI

DESENVOLVIMENTO EM ASP.NET MVC, JQUERY E AJAX

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

EDIVALDO NEGIR CHERUBINI

DESENVOLVIMENTO EM ASP.NET MVC, JQUERY E AJAX

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 Everton Coimbra de Araújo, Msc.

MEDIANEIRA

2011

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

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Desenvolvimento ASP.NET MVC, JQuery e AJAX

Por

Edivaldo Negir Cherubini

Este Trabalho de Diplomação (TD) foi apresentado às 13:15 h do dia 15 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 candidato foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

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

Prof. Alan Gavioli
UTFPR – *Campus* Medianeira
(Convidado)

Prof[a]. Alessandra B. Garbelotti
UTFPR – *Campus* Medianeira
(Convidado)

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

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

AGRADECIMENTOS

Aos meus pais que transferiram sua alegria e coragem para vencer os desafios que a vida nos impõe, e os obstáculos que não cansam de transpor os nossos caminhos, e também pela paciência e compreensão pelas vezes que não estive presente nos melhores momentos partilhados entre família.

Ao Prof. Everton Coimbra de Araújo, pela dedicação e empenho na orientação deste trabalho e ao meu irmão e grande amigo Edivan Cherubini, por sua disponibilidade, dedicação e auxilio para resolver problemas que surgiram durante o trabalho.

Se você acreditar que consegue, ou que não consegue, em ambos os casos, voce estará absolutamente certo.

Henry Ford

RESUMO

Com o interesse de criar aplicações *web* mais interativas, ágeis e produtivas, cada vez mais se tem dado importância aos *frameworks* e *plugins.* O *JQuery*, por exemplo, além de simplificar o código *Javascript*, tem como objetivo fazer com que os desenvolvedores produzam mais em menos tempo. Tudo isso utilizando AJAX para que a aplicação *web* se aproxime cada vez mais de uma aplicação *desktop* e execute mais rápido que uma aplicação *web* normal. Mas o *JQuery* só faz o trabalho do *client*, quando precisa-se enviar os dados de formulário é necessário que se tenha uma arquitetura robusta que possa manter e manipular os dados com uma maior liberdade. O ASP.NET MVC, além de dividir a aplicação em 3 camadas facilitando a compreensão da lógica do sistema, possibilita que o desenvolvedor manipule protocolos HTTP com uma maior liberdade. Esse trabalho tem como foco descrever os conceitos e recursos das tecnologias ASP.NET MVC, JQuery e AJAX. E através de um estudo de caso demonstrar o ganho de produtividade e vantagens que as tecnologias oferecem no desenvolvimento de aplicações web.

Palavras-chave: MVC, ASP.NET, JQuery, AJAX.

ABSTRACT

With the interest of creating more interactive web applications, agile and productive, more importance has been given to the frameworks and plugins. The JQuery, for example, simplify the Javascript code, aims to get developers to produce more in less time. All this using the AJAX web application to be closer and closer to a desktop application and run faster than a normal web application. JQuery only makes the work of the client, when you need to send the form data is necessary to have a robust architecture that can maintain and manipulate the data with greater freedom. The ASP.NET MVC also divides the application into three layers facilitating the understanding of the logic of the system and allows that the developer handle HTTP protocols with greater freedom. This work focuses on describing the concepts and capabilities of the technologies ASP.NET MVC, jQuery and AJAX. And through a case study demonstrate the productivity gains and benefits that the technology offered in web application development.

Keywords: MVC, ASP.NET, JQuery, AJAX.

LISTA DE ABREVIATURAS E SIGLAS

AJAX	*Asynchronous Javascript And Xml*
API	*Application Programming Interface*
CSS	*Cascading Style Sheets*
JSON	*Javascript Object Notation*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
MVC	*Model - View – Controller*
RAD	*Rapid Application Development*
RSS	*Really Simple Syndication*
UI	*User Interface*
URL	*Uniform Resource Locator*
XML	*eXtensible Markup Language*

LISTA DE QUADROS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A grande maioria das aplicações *web* precisam fazer *refresh* da página ao submeter dados de formulários, fazendo com que a página precise ser recarregada e, portanto, aumentando a dificuldade em trabalhar com mensagens de formulário e em manter os dados para o resto da aplicação. Essas tarefas deverão ser desenvolvidas em *Javascript* e assim o código da aplicação poderá ficar muito extenso e de difícil reutilização. Além disso, é marcado por grandes incompatibilidades entre *browsers*, a maioria dos programadores aderiram ao uso de bibliotecas *Javascript* para evitar esses tipos de problemas (FLANAGAN, 2011).

Manipular tantos conteúdos, validações e eventos podem ser tarefas árduas para o programador, tornando a manutenção do código mais complexa. O *plugin JQuery* traz uma solução fácil e produtiva, além de utilizar *Ajax* para resolver tais problemas.

Outra dificuldade esta em redirecionar páginas que precisam dos dados dos formulários, principalmente quando se tem mais de um formulário na página. O ASP.NET MVC vem para facilitar esse quesito e também auxiliar nas regras de negócio da aplicação web, já que nessa arquitetura tudo é separado tornando mais fácil a compreensão do código para futuras manutenções.

1.1 OBJETIVO GERAL

Demonstrar que as tecnologias ASP.NET MVC, AJAX e *JQuery* possibilitam uma maior produtividade e flexibilidade de código, desenvolvendo uma aplicação web que demonstre os recursos que essas tecnologias oferecem.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são:

- Realizar um estudo bibliográfico sobre os conceitos e recursos das tecnologias ASP.NET MVC, Ajax e JQuery;
- Demonstrar a arquitetura, modelo de programação e funcionamento do ASP.NET utilizando a arquitetura MVC e Ajax;
- Desenvolver uma aplicação web em ASP.NET utilizando MVC, Ajax e *JQuery* para exemplificar o ganho de produtividade.

1.3 JUSTIFICATIVA

Existem vários *frameworks* disponíveis para as mais variadas linguagens de programação, não existindo um que seja o melhor de todos, portanto é importante saber escolher qual irá melhor atender as necessidades da aplicação, qual será mais produtivo, mais seguro e reutilizável, de acordo com o problema do projeto.

O *JQuery* é uma biblioteca *Javascript* que tem por objetivo simplificar o código e maximizar a produtividade no desenvolvimento web, por meio dele é possível manipular qualquer conteúdo com poucas linhas de código e sem toda a complexidade do *Javascript*. Também se obtém um ganho quanto à compatibilidade pois o programador não precisa se preocupar se a aplicação web vai rodar de acordo em todos os browsers.

Segundo John Resig, criador da biblioteca, citado por Silva (2008), "O foco principal da biblioteca *JQuery* é a simplicidade. Por que submeter os desenvolvedores ao martírio de escrever longos e complexos códigos para criar simples efeitos?". A interação da aplicação web com o usuário pode ser resolvida com AJAX e para tornar essa tarefa mais simples utiliza-se *JQuery*.

Ajax não é uma linguagem de programação, é uma maneira de se utilizar as tecnologias HTML e *Javascript* para criar aplicações Web melhores, mais rápidas e interativas. Permite submeter dados em formulários e manipular conteúdos sem precisar recarregar a página (REFSNES; HENTHORNE, E HENTHORNE, 2010).

O ASP.NET *Model-View-Controler* (MVC) é uma implementação da arquitetura MVC que tem por objetivo criar aplicações web no padrão MVC. A arquitetura consiste em três camadas, modelo, visão e controle. As regras de negócio estão na camada de modelo, as interfaces em visão e as de interação com o usuário na de controle. Programando neste padrão, testar, manter e atualizar a aplicação web se torna uma tarefa mais fácil e também possibilita programar em equipe, separando o sistema em módulos.

É certo que para utilizar um *framework*, *plugin* ou uma arquitetura diferente é preciso estudar como estes funcionam, como devem ser utilizados e tudo isso demanda tempo de estudo e treino, mas as vantagens são grandes e valem o esforço.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho é composto por 5 capítulos, o primeiro trata sobre os assuntos que serão abordados, introdução, objetivos gerais e específicos, justificativa e uma breve introdução ao ASP.NET. O segundo capítulo trata sobre o desenvolvimento em ASP.NET MVC e seus conceitos. O terceiro descreve as tecnologias *JQuery* e as técnicas de AJAX no desenvolvimento de aplicações mais interativas. O quarto capítulo descreve um estudo de caso que tem por objetivo mostrar a integração e o desenvolvimento das tecnologias ASP.NET MVC e *JQuery.* O quinto capítulo destaca as considerações finais sobre o assunto abordado no estudo de caso.

2 ASP.NET MVC

2.1 ASP.NET

ASP.NET foi desenvolvido em 2001 e abriu portas para muitos profissionais. A tecnologia também contribuiu para modificar o modelo de desenvolvimento de aplicações web (ESPOSITO, 2010).

O ASP.NET é considerado estável, maduro e uma plataforma altamente produtiva em desenvolvimento *Web*. A *Microsoft* foi refinando o ASP.NET ao longo dos anos e atualmente, a tecnologia inclui vários recursos como uma rica plataforma de desenvolvimento AJAX (ESPOSITO, 2010).

Por um longo período a tecnologia era conhecida apenas pela arquitetura *WebForms.* Mas em 2007 a *Microsoft* apresentou uma outra alternativa – o ASP.NET MVC que vem crescendo muito e amadurecendo rapidamente (ESPOSITO, 2010).

2.2 PADRÃO MVC

Inventado para ser utilizado na linguagem *Smalltalk* por Trygve Reenskaug nos anos 70, o padrão MVC foi se popularizar na web em 2003 com o surgimento do *Ruby on Rails* (PALERMO, et. al., 2010). Segundo Sanderson (2009, p.7), o motivo dessa popularização talvez seja pelos seguintes fatos:

- A interação com o usuário segue um ciclo natural. O usuário executa uma ação e como resposta a aplicação atualiza os dados do banco e/ou entrega uma representação da *view* atualizada. E assim o ciclo se repete várias vezes. O padrão é perfeito para aplicações *web* baseadas em HTTP *requests* e *responses*;
- Aplicações *web* precisavam de uma maneira de combinar as tecnologias existentes, normalmente divido em camadas.

Implementado em quase todas as plataformas em uso atualmente, a arquitetura consiste em três camadas, a de modelo (*model)*, controle (*controller*) e visão (*view)* (BERARDI; KATAWAZI; BELLINASO, 2009).

- Modelo: é a camada responsável pelas regras de negócio da aplicação. Também é o encarregado de guardar o estado dos objetos. Em outras palavras, essa camada manipula os dados do sistema, de maneira que os dados estejam prontos para que as camadas de controle e visão possam utilizá-las (BAPTISTELLA, 2009);
- Visão: é a representação visual do modelo (BERARDI; KATAWAZI; BELLINASO, 2009). Responsável por mostrar os dados de uma maneira que o usuário possa entender e pelos *inputs* de entrada de dados (BAPTISTELLA, 2009);
- Controle: fornece uma conexão entre a camada de modelo e visão. É o encarregado de organizar e direcionar as ações entre o usuário e o sistema. Em outras palavras, é quem interpreta quais métodos da camada de modelo serão utilizados e/ou qual objeto de visão será encarregado de renderizar essas informações (PALERMO, et. al., 2010).

Para ser mais exato, a camada de modelo faz todo o trabalho da aplicação, a camada de visão encarrega-se da interface do usuário e a camada de controle executa todas as ações da aplicação e faz a troca de mensagens entre o modelo e a visão. Para cada objeto de visão existe um objeto de controle, mas para cada modelo podem existir vários objetos de visão e controle (KRASNER; POPE, 1988).

Pode se dizer que as camadas de visão e controle são as encarregadas de apresentar a camada de modelo de uma maneira que usuários possam entender (BERARDI; KATAWAZI; BELLINASO, 2009).

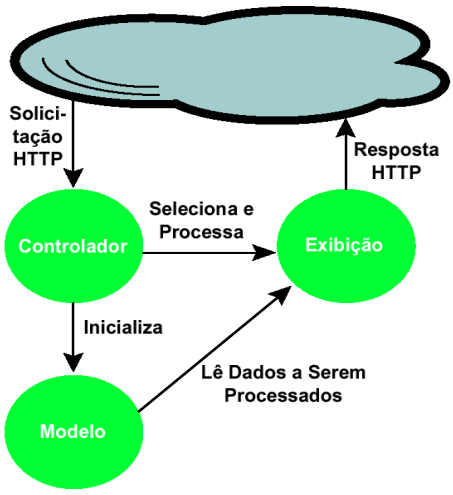


Figura 1 - Model-View-Controller
Fonte: (MSDN MAGAZINE, 2008).

A **Figura 1** mostra o funcionamento da arquitetura MVC. A camada de exibição faz uma solicitação HTTP e espera uma resposta que será interpretada pelo controlador que irá decidir se esta requisição será processada por outro objeto de visão ou pela camada de modelo. Caso requisitado, o objeto de modelo irá processar essas informações e devolver os dados prontos para a controladora enviar uma resposta à camada de exibição.

2.3 FRAMEWORK ASP.NET MVC

Apresentado em novembro de 2007, o ASP.NET MVC é um dos *frameworks* da *Microsoft* para o desenvolvimento de sistemas web. O *framework* já sofreu várias atualizações e utiliza o padrão de arquitetura *Model-View-Controller*. Este padrão, que já é muito popular, não veio para substituir o *WebForms,* mas para ser uma alternativa eficiente em desenvolvimento web (PALERMO, et. al., 2010).

Diferente do *WebForms*, o MVC é divido nas camadas: modelos (*model)*, ações (*controller)* e exibições (*view)* e não em eventos (CHIARETTA, 2010). Forçar a divisão da aplicação dessa maneira se apresentou ser muito útil. Separar os *views* de toda a lógica da aplicação permite refazer o *layout* sem encostar na lógica do sistema. A divisão ajuda a aplicar as habilidades específicas de cada membro da equipe de desenvolvimento de maneira mais adequada, pois o sistema está dividido entre a aparência, a lógica e o acesso ao banco de dados. Também vale ressaltar que os designers podem trabalhar na camada de exibição sem se preocupar se algo esta faltando nos controles ou nos modelos (WALTHER, 2010).

Além disso, a divisão dos *views* facilita em uma troca futura de tecnologia. Por exemplo, migrar de HTML para o *Siverlight* poderia ser algo complicado caso a lógica de negócio estivesse amarrada as *views* da aplicação. Colocar todas as ações nos *controllers* também é uma vantagem, pois freqüentemente é necessário mudar a maneira com que o usuário interage com o sistema e assim pode-se modificar sem encostar nas outras camadas. O mesmo vale para a camada de *model* que poderá trocar a tecnologia de acesso ao banco de dados sem precisar modificar o resto do sistema (WALTHER, 2010).

A arquitetura MVC possibilita uma aplicação altamente testável e sustentável porque a aplicação ficará naturalmente dividida entre independentes camadas de diferentes softwares. Assim pode-se criar um código limpo, com várias unidades de teste de cada ação da aplicação

usando falsas ou reproduções dos componentes do *framework* para simular qualquer cenário. O *framework* possibilita que o *Visual Studio* crie um projeto de testes, então mesmo que o desenvolvedor não tenha esse conhecimento, já terá um bom começo (SANDERSON, 2009).

Como o ASP.NET MVC faz parte do .NET *framework*, pode-se desenvolver em qualquer linguagem .NET e também acessar todas as bibliotecas disponíveis pela plataforma. Recursos prontos como *master page, Forms authentication, membership, roles, profiles* e *globalization* podem diminuir a quantidade de código que o desenvolvedor precisará escrever e é tão efetivo em projetos MVC quanto em *WebForms* (SANDERSON, 2009).

Pode-se dizer que o ASP.NET MVC é um bom exemplo de *RESTful*. Isso porque o *framework* trabalha com envio de requisições HTTP para recursos. Cada recurso é identificado com uma URL e esse recurso é o *controller* do MVC. E assim como toda aplicação *RESTful*, toda essa troca de informações é feita a partir de linguagens de marcação como XML e JSON (ESPOSITO, 2010).

2.3.1 Roteamento de URL

Atualmente os desenvolvedores se importam mais em trabalhar com URLs limpas, que não possuem informações extras. Primeiro, por facilitar os sistemas de busca. Segundo, pois assim impede que usuários que possuem um entendimento maior pelas URLs, naveguem pela aplicação *web* apenas alterando os valores do endereço HTTP. Terceiro, impossibilita que informações pessoais do usuário sejam passados pela URL. Quarto, não expõe na URL detalhes técnicos como estrutura de pastas e nome dos arquivos da aplicação (SANDERSON, 2009).

No ASP.NET MVC as URLs não correspondem as estruturas de pastas e nome de arquivos do servidor. Até porque isso não faria sentido, já que todas as requisições HTTP são interpretadas pela *controller* da aplicação (SANDERSON, 2009). Segundo Jakob Nielsen, citado por Galloway, et. al. (2010, p. 203), para que uma URL seja de alto nível é preciso seguir os seguintes requisitos:

- Deverá ter um nome de domínio fácil de lembrar e ler;
- URLs pequenas;
- URLs fáceis de serem escritas;

- URLs que refletem a estrutura do site;
- URLs que são *hackable* para permitir que usuários com conhecimento avançado possam hackear o final da URL;
- URLs persistentes, que não mudam.

Ao invés de ter uma URL que se relaciona diretamente com o arquivo em disco do servidor *web*, se tem uma URL que se relaciona com os métodos da *controller.* Para fazer esse relacionamento o ASP.NET MVC utiliza o padrão *front controller* (GALLOWAY, et. al., 2010).

Table 8-3. *How the Default Route Entry Maps Incoming URLs*

URL	Maps To
/	{ controller = "Home", action = "Index", id = "" }
/Forum	{ controller = "Forum", action = "Index", id = "" }
/Forum/ShowTopics	{ controller = "Forum", action = "ShowTopics", id = "" }
/Forum/ShowTopics/75	{ controller = "Forum", action = "ShowTopics", id = "75" }

Figura 2 - Roteador de URL.
Fonte: (SANDERSON, 2009).

Como mostra a **Figura 2**, o roteador *default* do ASP.NET MVC, mapeia o método e a *controller* seguindo o padrão {*controller*}/{*action*}/{*id*}. Quando não é passado pela URL o nome da *controller,* automaticamente o roteador buscará o método *index* do *controller Home.*

Segundo Walther (2010, p. 270), para configurar as rotas da maneira que desejar, deve-se alterar o arquivo *Global.asax* que vem junto quando cria-se um projeto ASP.NET MVC *Web Application*.

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

namespace MvcApplication1
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // Route name
                "{controller}/{action}/{id}", // URL with parameters
                new { controller = "Home", action = "Index", id = "" }  //
Parameter defaults
                );

        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
```

Figura 3 - Global.asax.
Fonte: (WALTHER, 2010).

O arquivo *Global.asax* (**Figura 3**) possui dois métodos chamados *Application_Start()* e *RegisterRoutes()*. O método *Application_Start()* é chamado apenas uma vez, quando a aplicação ASP.NET MVC é iniciada. Esse método utiliza o *Register_Router()* que possui todas as configurações de roteamento da URL (WALTHER, 2010).

Segundo Walther (2010), para configurar uma nova rota deve-se alterar o método *MapRoute()*. Esse método aceita os seguintes argumentos:

- *Name*: nome da rota;
- URL: a URL padrão da rota;
- *Defaults:* os valores padrão para os parâmetros da rota;
- *Constraints:* conjunto de *constraints* que limitam as requisições utilizadas pela rota;
- *Namespaces:* conjunto de *namespaces* que limitam as classes que utilizam esta rota.

2.3.2 Model

A camada de modelo (*model*) refere-se a lógica de negócio ou a objetos de domínio da aplicação. Esses objetos são responsáveis por persistir o estado da sua aplicação em um banco de dados ou utilizando outros meios. O *model* pode utilizar qualquer padrão, tecnologia ou metodologia para desenvolver uma camada de negócios (BERARDI; KATAWAZI; BELLINASO, 2010).

Uma classe *model* pode ser um *domain model* ou um *application model*. O primeiro tem por objetivo suportar e modelar apenas os problemas da aplicação. Já o segundo, é o objeto que sabe que as *views* existem e precisam de uma maneira de obter os dados da aplicação (DEACON, 1995).

Sem a camada de *model* a aplicação não terá nenhum valor para ser representado e ao mesmo tempo nenhum sentido. É extremamente importante criar um *model* que claramente expresse a realidade da aplicação e as soluções para os problemas desse domínio (PALERMO, et. al., 2010).

2.3.3 Controller

O foco do padrão MVC está na *controller.* É o responsável por interpretar todas as ações feitas pelo usuário e retornar uma *view.* Com ASP.NET MVC, todas requisições são manipuladas pela *controller* que implementa a *interface IController.* Para facilitar a criação de *controllers* a *Microsoft* fornece a classe *System.Web.MVC.Controller* (PALERMO, et. al., 2010).

Segundo Sanderson (2009), utilizando a classe *System.Web.MVC.Controller* se tem as seguintes vantagens:

- *Action methods:* as ações são os métodos que a *controller* possui. Cada ação ou método utiliza uma URL diferente e é invocado pelos parâmetros passados pela requisição;
- *Action results:* possui a opção de retornar um objeto que resulta em uma ação. As ações podem ser *views* ou podem redirecionar para outras URLs;
- *Filters:* um atributo de filtro que pode ser aplicado a uma ação do *controller.*

A partir dessa classe pode-se também utilizar algumas propriedades como o *ViewData* que serve para recuperar e modificar um dicionário de valores que podem ser criados pela *controller* e pelas *views* (ESPOSITO, 2010).

Os métodos da *controller* recebem os dados da requisição HTTP através de seus parâmetros. A *controller* converte os dados da requisição de acordo com os tipos de parâmetros aceitados pelo método (PALERMO, et. al., 2010).

Em ASP.NET MVC, os tipos de protocolos HTTP como GET, POST e PUT podem ser associados aos métodos através de anotações. Para isso, utiliza-se os atributos: *HttpGet, HttpPost* e *HttpPut,* respectivamente. Quando um método não possui nenhuma relação com essas anotações, por padrão, o método utiliza o protocolo GET. Como os métodos podem ser associados a protocolos diferentes, é possível utilizar métodos com nomes iguais desde que os protocolos sejam diferentes (ESPOSITO, 2010).

2.3.4 View

A camada de *views* é a representação gráfica do sistema. É responsável por renderizar as páginas HTML com que os usuários poderão interagir (WALTHER, 2010).

A maioria dos desenvolvedores afirmam que a *user interface* é melhor quando separada do resto da lógica da aplicação. O código de modelo e *interface* misturados tornará difícil a compreensão do que é o que. Uma simples modificação poderá virar em uma corrente de *bugs*, diminuindo a produtividade. MVC tenta resolver esses problemas forçando as *views* a serem simples e separadas do *model.* Nesse padrão a *view* é responsável apenas por utilizar lógicas simples de apresentação de dados para criar páginas HTML (SANDERSON, 2009).

Uma *view* não precisa ser necessariamente uma página HTML,pode ser qualquer outra forma de transmissão de dados para renderização de páginas, incluindo JSON, XML, RSS, ou qualquer outro protocolo (BERARDI; KATAWAZI; BELLINASO, 2010).

Segundo a *Microsoft* (2011), quando se cria uma *view* deve ser levado em consideração o nome da ação e da *controller* que retorna essa *view*. A estrutura da aplicação deverá seguir as seguintes regras:

- O nome da *view* deverá ser o mesmo nome da ação utilizado pela *controller*;
- A *view* deverá se encontrar na pasta que possui o nome da *controller,* sem o *Controller* no final;
- A pasta da *view* criada deverá ficar no diretório *views* do projeto.

Views são menos testáveis que *controllers* ou *models*. Por esse motivo as *views* as vezes nem são testadas pois espera-se que a lógica que realmente precise ser testada fique nos *controllers* ou *models* da aplicação (SANDERSON, 2009).

2.4 ASP.NET MVC VS ASP.NET WEBFORMS

O ASP.NET *WebForms* continuará existindo mesmo após o lançamento da arquitetura MVC. Não se pode afirmar qual arquitetura é melhor, cada um tem seus pontos fortes e são aconselhados dependendo da situação em que a equipe de desenvolvimento se encontra, mas é

fato que o padrão MVC vem conquistando muitos programadores (SANDERSON, 2009). Os tópicos a seguir irão listar os pontos fortes que cada arquitetura possui.

2.4.1 ASP.NET WebForms

Criado em janeiro de 2002, o *WebForms* foi a primeira tentativa da *Microsoft* de criar uma arquitetura robusta e flexível que pudesse suprimir a demanda da web na época. O padrão provou que poderia ser utilizado em desenvolvimento de pequenas e grandes aplicações. Como a arquitetura é baseada no *Windows Forms*, padrão baseado em eventos, uma vantagem importante pôde ser reaproveitada. A vantagem do designer não precisar compreender quase nada de HTML ou qualquer outra linguagem relacionada a apresentação de dados para poder desenvolver uma página utilizando *WebForms*. Tudo é feito com uma interface visual e isso nos proporciona mais produtividade (BERARDI; KATAWAZI; BELLINASO, 2010). Segundo Berardi, et. al. (2010), as vantagens e desvantagens são:

- Tecnologia madura, sofreu várias atualizações;
- Desenvolvimento em RAD;
- Gerenciamento de estados abstraído;
- Fácil de trabalhar;
- Suporta ricas bibliotecas da *Microsoft* e de terceiros;
- Reduz a necessidade de compreender HTTP, HTML, CSS e em alguns casos, *Javascript*;
- Programação familiar ao *Windows Forms*.

  Desvantagens:
- Controle sobre o HTML limitado;
- Dificuldade integração com *framework s Javascript* como *JQuery*;
- Não estimula o desenvolvimento em padrões, apesar de suportá-las;
- Dificuldade nos testes da aplicação;
- Controle de gerenciamento de estados torna as páginas de exibição muito grandes e algumas páginas desnecessárias.

2.4.2 ASP.NET MVC

A arquitetura foi muitas vezes ignorada por desenvolvedores modernos. Isso porque o desenvolvedor precisa entender HTML e outras tecnologias para poder desenvolver no padrão MVC, algo que não acontece com *WebForms*. Mas quando se trata de controle de HTML e divisão do sistema, o *WebForms* é muito limitado e é nessas ocasiões em que o MVC se destaca mais. Sistemas complexos que necessitam de uma divisão de tarefas, testes confiáveis e maior flexibilidade, são os pontos mais importantes da arquitetura MVC (BERARDI; KATAWAZI; BELLINASO, 2010).

Segundo Berardi, et. al. (2010), as vantagens e desvantagens são:

- Fornece bom controle sobre o HTML;
- Código HTML simples;
- Divisão clara da lógica do sistema;
- Possibilita testes mais confiáveis;
- Suporta vários tipos de linguagem de marcação, como o XSLT, *Brail*, *NHaml*, *NVelocity* e assim por diante;
- Fácil integração com bibliotecas *Javascript* como *JQuery* e *Yahoo* UI;
- Habilidade de mapear as URLs logicamente e dinamicamente, dependendo da situação;
- Interface RESTFul são utilizados por padrão;
- Sem *ViewState* ou *PostBack*;
- Suporta os principais recursos do ASP.NET, como autenticação, *cache* e assim por diante;
- Tamanho das páginas geralmente são menores, pois não precisam de *ViewState*.

Desvantagens:

- É mais complexo;
- Exige mais do desenvolvedor;
- Exige conhecimento de HTML, CSS, *Javascript* e assim por diante;
- Não é baseada em eventos então pode ser difícil para desenvolvedores acostumados com *WebForms* utilizarem a arquitetura.

2.4.3 Escolha da Arquitetura

Devem ser considerados pontos importantes como o conhecimento da equipe e também as necessidades da aplicação para decidir qual arquitetura utilizar (BERARDI; KATAWAZI; BELLINASO, 2010). Por exemplo, quando a aplicação já está concluída ou parte dela já esta feita em *WebForms*, o mais aconselhado é não misturar as arquiteturas e continuar com a atual. Em outras questões também como falta de conhecimento da equipe tanto para o padrão MVC quanto para o *WebForms*, a melhor opção é a segunda devido ao desenvolvimento com recursos visuais que a arquitetura oferece. Mas quando se trata de controle de HTML, ou necessidade em dividir a aplicação em módulos, testes unitários, maior liberdade sobre o *framework* e vantagens de plug-ins *Javascript* como *JQuery* e similares, com certeza o MVC é a melhor escolha (SANDERSON, 2009).

As duas arquiteturas possuem suas vantagens e desvantagens e os principais a serem pesados para a escolha provavelmente é o conhecimento da equipe e o tempo disponível para o desenvolvimento. O padrão MVC exige do desenvolvedor um conhecimento avançado ao contrário do *WebForms*, que possui recursos visuais para gerar o código. Mas quando se trata de aplicações complexas, o MVC é mais aconselhado, devido a divisão da lógica do sistema em módulos, testes mais confiáveis, extensão do *framework* e liberdade sobre o HTML. Além disso, o MVC nos possibilita modificar o sistema com uma facilidade maior, sem ter que modificar a lógica inteira do sistema para utilizar uma tecnologia diferente ou mudar a interação do usuário com o sistema (WALTHER, 2010).

3 JQUERY

Javascript é intencionalmente incompatível com vários *browsers*. Mesmo com as atualizações no Internet Explorer que minimizam muito as incompatibilidades, os programadores preferem utilizar *frameworks Javascript* ou *plugins* para fazer tarefas comuns e driblar estes problemas. O *JQuery* é uma dessas bibliotecas *Javascript* (FLANAGAN, 2011).

JQuery ficou extremamente popular entre os desenvolvedores de aplicação web por causa de sua simplicidade e fácil uso. A biblioteca possui suporte de uma comunidade de desenvolvedores e tem crescido muito ao longo dos anos desde a sua criação, em 2006, por John Resig (ALLANA, 2011). Segundo John Resig, criador da biblioteca, citado por Silva (2008), "O foco principal da biblioteca *JQuery* é a simplicidade. Por que submeter os desenvolvedores ao martírio de escrever longos e complexos códigos para criar simples efeitos?".

Com *JQuery* fica fácil manipular os elementos das páginas, adicionar e modificar atributos HTML e propriedades CSS, definir eventos e animações.Essa biblioteca também possui suporte a AJAX para fazer dinamicamente HTTP *requests* e outras funcionalidades gerais.Por estar sendo bastante utilizado, os desenvolvedores devem estar familiarizados com o *JQuery*, mesmo que não a utilizem no seu código, é bem provável que irá encontrar no código de outras pessoas (FLANAGAN, 2011).

Com a colaboração da *Microsoft* com *templates*, *datalink* e globalização de *plugins* para o *JQuery*, a biblioteca tem se tornado popular entre os desenvolvedores de ASP.NET. Além de ASP.NET, a biblioteca pode ser utilizada por outras linguagens como PHP e Java (ALLANA, 2011).

A equipe de desenvolvimento do *JQuery* teve muito cuidado para fazer com que a biblioteca fosse extensível. Ao fornecer o núcleo de recursos do *JQuery* a partir de um *framework* , conseguiram facilitar a extensão e criação de *plugins JQuery*. Felizmente, muitos desenvolvedores tiraram vantagem da capacidade de extensão do *JQuery* e atualmente já existem centenas de *plugins* excelentes e disponíveis no repositório do site do *JQuery* (CASTLEDINE; SHARKIE, 2011).

3.1 O BÁSICO DE JQUERY

Para utilizar a biblioteca deve-se referenciá-la utilizando a propriedade *src* do *script*:

```
<script type='text/Javascript ' src='jquery-1.4-min.js'></script>
```

A biblioteca *JQuery* possui uma única função chamada *jQuery*(). Essa função é tão freqüentemente utilizada que foi definido *$* como um atalho (FLANAGAN, 2011).Como o *JQuery* tem essa função que age com um *gateway* dificilmente irá acontecer conflitos com outras bibliotecas *Javascript* ou com o código que será escrito(CASTLEDINE; SHARKIE, 2010).

A **Figura 4** apresenta os comandos do *JQuery*:

selector	action	parameters
jQuery('p')	.css	('color', 'blue');
$('p')	.css	('color', 'blue');

Figura 4 - Comandos JQuery.
Fonte: (CASTLEDINE; SHARKIE, 2011).

Cada comando é feito em 4 partes: a função *JQuery* (ou atalho), seletores, ações e parâmetros. Primeiro é a função *JQuery* que já conhecemos. Depois vêm os seletores que servem para que possamos selecionar os elementos da página. Em seguida, a ação que desejamos aplicar para cada elemento selecionado. E por último, especificam-se os parâmetros para que o *JQuery* saiba exatamente como se pretende aplicar a ação no elemento selecionado (CASTLEDINE; SHARKIE, 2011).

No exemplo acima (**Figura 4**), foi demonstrado como alterar uma propriedade CSS de todos os elementos <*p*> de uma página HTML. Mas essa não é a única maneira de se utilizar os comandos *JQuery*. Ao invés de passar apenas uma propriedade CSS podem ser passadas várias ou em alguns casos, não seria necessário passar nenhum parâmetro e em alguns seria preciso passar outra função *Javascript* quando se define um evento. Também poderia ser selecionado apenas um elemento passando o atributo *id* ou *class* dele, ao invés de selecionar todos os elementos <*p*> de uma página HTML (CASTLEDINE; SHARKIE, 2011).

Antes de poder alterar os elementos de uma página HTML, precisa-se verificar se esses elementos estão prontos para o uso. No *Javascript* a única maneira de fazer isso é esperando toda a página ser carregada para poder rodar os *scripts*. Felizmente, o *JQuery*

possui um evento que executa o *script* assim que os elementos estão prontos para uso e este evento chama-se *$(document).ready()* (CASTLEDINE; SHARKIE, 2011).

3.2 EVENTOS

Uma das dificuldades em trabalhar com eventos em *Javascript* é que o *Internet Explorer* implementa um evento diferente da API dos outros *browsers.* Para resolver de vez esse problema, o *JQuery* possui uma API que é compatível com todos os navegadores (FLANAGAN, 2011).

Os eventos são ações e interações do usuário que ocorrem na página *web.* Quando um evento acontece, pode-se dizer que um evento foi disparado. E quando é escrito um código para manipular esse evento, pode-se dizer que o evento foi capturado (BIBEAULT; KATS, 2008).

Há milhares de eventos que podem ser disparados em uma página *web* e ao tempo todo. Quando um usuário move o *mouse* ou clica em um botão, ou quando a jánela do navegador é redimensionada, ou a barra de rolagem movida. Pode-se capturar e agir sobre qualquer um desses eventos (BIBEAULT; KATS, 2008).

Segundo Flanagan (2011), o *jQuery* declara métodos simples de registro de eventos. Para registrar um manipulador de eventos para “clicar” nos elementos, por exemplo, utiliza-se o método *click():*

```
// Ao clicar em qualquer elemento <p> muda-se a cor de fundo
do elemento
$("p").click(function() {
$(this).css("background-color", "gray");
});
```

Os eventos que o *JQuery* consegue registrar são: *blur(), focusin(), mousedown(), mouseup(), change(), focusout(), mouseenter(), resize(), click(), keydown(), mouseleave(), scroll(), dblclick(), keypress(), mousemove(), select(), error(), keyup(), mouseout(), submit(), focus(), load(), mouseover(), unload().*

A função *click* do exemplo acima não espera nenhum argumento e não retorna nenhum valor. É perfeitamente escrever eventos assim, mas o *jQuery* pode invocar todos os

eventos com um ou mais argumentos. A coisa mais importante que precisa-se saber é que cada manipulador de evento utiliza um objeto de evento *jQuery* como seu primeiro argumento. Os campos desse objeto proporcionarão detalhes como as coordenadas do ponteiro do *mouse* sobre o evento (BIBEAULT; KATS, 2008).

3.3 AJAX TRADICIONAL

Pode-se dividir as tecnologias web em duas categorias: a de cliente que comporta tecnologias de HTML, CSS, *Javascript* e outros; e, a de servidor que comporta, *Apache*, PHP, *MySQL*, etc. O AJAX não se encaixa em nenhuma das categorias pois utiliza uma linguagem *Javascript* que é uma linguagem de cliente e também envolve comunicação com linguagens de servidor (KEITH, 2007).

AJAX não é uma linguagem nova, mas uma nova maneira de utilizar as linguagens existentes para criar melhores, mais rápidas e interativas aplicações web. AJAX tem por objetivo trocar informações entre servidor e cliente, e manipular o conteúdo da página sem ter que recarregá-la (REFSNES; HENTHORNE, 2010).

Usando AJAX, as páginas e a aplicação apenas solicitam ao servidor o que realmente precisam, apenas as partes que precisam mudar, os dados que o servidor precisa fornecer. Isso significa menos tráfego, menos atualizações e menos tempo esperando uma página ser recarregada (RIORDAN, 2008).

Para utilizar AJAX é preciso ter conhecimento com *Javascript*, XML e HTML. A partir do objeto XMLHttpRequest do *Javascript* é possível fazer a troca de informações entre o cliente e o servidor, independente da linguagem de programação utilizada no servidor (REFSNES; HENTHORNE, 2010).

3.3.1 XMLHttpRequest

Para recuperar ou enviar uma informação para o banco de dados ou um arquivo do servidor, é preciso criar um formulário no HTML. Quando o usuário clicar no botão de

submit, as informações serão enviadas ao servidor que irá processar os dados e enviar uma resposta ao usuário. Enquanto o servidor processa esses dados, o usuário é obrigado a esperar até que a informação esteja pronta para ser retornada e recarregar a página (REFSNES; HENTHORNE, 2010).

Com AJAX, o *Javascript* comunica diretamente com o servidor através do objeto XMLHttpRequest. Dessa maneira, uma página consegue se comunicar com o servidor sem precisar fazer *refresh*. Continuando na mesma página o usuário nem perceberá que a informação foi processada por um servidor, dando a impressão de que a aplicação web está interagindo com o usuário da mesma maneira que uma aplicação *desktop* (REFSNES; HENTHORNE, 2010).

3.3.2 Desenvolvendo AJAX

Para entender como funciona será criado um pequeno algoritmo em AJAX. A aplicação terá dois *buttons* que ao serem clicados irão buscar dados no servidor e alterar o conteúdo HTML sem recarregar a página.

Primeiro cria-se um pequeno HTML com uma *tag div*. Essa *div* será utilizada para mostrar o conteúdo recuperado pelos botões. Para identificar a *div* que possuirá o conteúdo será colocado um “teste” como atributo de id:

```
<html>
<body>
<div id=”teste”>
<h2>Clickto let AJAX change this text</h2>
</div>
<body>
</html>
```

Agora será adicionado dois *buttons*. Quando eles forem clicados chamarão a função loadXMLDoc() passando por parâmetro o arquivo TXT que o servidor deverá buscar os dados:

```
<button type="button" onclick="loadXMLDoc(‘test1.txt’)">Click Me</button>
<button type="button" onclick="loadXMLDoc(‘test2.txt’)">Click Me</button>
```

Por último adiciona-se o código AJAX para declarar a função loadXMLDoc() em *Javascript* . Então será adicionado no *head* da página o seguinte *script*:

```
<script type="text/Javascript ">
     function loadXMLDoc(url) {
          if (window.XMLHttpRequest)
               {
               // code for IE7+, Firefox, Chrome, Opera, Safari
               xmlhttp=new XMLHttpRequest();
               }
          else {
               // code for IE6, IE5
               xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
               }
     xmlhttp.open("GET",url,false);
     xmlhttp.send(null);
     document.getElementById('test').innerHTML=xmlhttp.responseText;
     }
</script>
```

Quadro 1 - Exemplo de AJAX.

Como pode-se ver, a função loadXMLDoc() identifica se a versão do *browser* que esta sendo utilizado é posterior a versão 6 do Internet Explorer, caso seja, cria um objeto XMLHttpRequest, senão um objeto ActiveXObject. Essa identificação serve para resolver problemas de incompatibilidades de *browsers*, pois as versões posteriores ao Internet Explorer 6 e outros navegadores utilizam o XMLHttpRequest para fazer suas chamadas remotas ao servidor, ao contrario das versões anteriores que utilizavam o ActiveXObject.

Apos criar o objeto utiliza-se o método *open* passando como parâmetro o protocolo HTTP a ser utilizado, a *url* e o tipo de conexão, que pode ser assíncrono ou não. Envia-se a *request* utilizando o método *send* e mudamos o conteúdo da *div* passando os dados que vieram do servidor a partir da propriedade *responseText*.

Esse é apenas um pequeno exemplo do que o AJAX é capaz, poderia ter sido utilizado também outras funções e propriedades como o *readyState* que consegue informar a aplicação se a requisição já foi iniciada, configurada, enviada e processada.

3.4 AJAX EM JQUERY

Como as técnicas de AJAX são tão úteis para as aplicações *web* atuais, o *JQuery* inclui essas utilizadas na biblioteca para simplificar o desenvolvimento. O *JQuery* utiliza um

método de alto nível e mais 4 funções para trabalhar com AJAX. Todas essas funções são baseadas no método *jQuery.ajax()* (FLANAGAN, 2011).

Um dos usos mais comuns do AJAX é pegar um pedaço de conteúdo do servidor e colocá-lo em algum elemento DOM. O elemento DOM pode ser um fragmento da página HTML que se tornará em outro elemento ou poderá ser apenas um texto simples (BIBEAULT; KATS, 2008).

A grande diferença de simplicidade entre o AJAX tradicional e o *JQuery* pode ser percebida mesmo com um exemplo simples. O equivalente para o código mostrado no exemplo do item 3.2.2 que possui várias linhas de código, em *JQuery* ficaria:

```
$('#someContainer').load(url);
```

O método *load()* é o mais simples de todas as utilidades *JQuery:* passa-se apenas uma URL, que irá carregar o conteúdo de forma assíncrona e em seguida, inserir esse conteúdo no elemento da página selecionado substituindo o conteúdo que havia antes (FLANAGAN, 2011).

Esse método também pode ser utilizado para carregar apenas um elemento da páginas HTML passada pela URL caso seja de interesse. Nesse caso, é passado a URL do documento HTML a ser carregado e o *id* do elemento que deverá ser carregado, separando com um espaço entre a URL e o *id* (FLANAGAN, 2011). Exemplo:

```
// Load the temperature section of the weather report
$('#temp').load("weather_report.html #temperature");
```

Segundo Bilbeault e Kats (2008), o método *load()* suporta os seguintes parâmetros:

- URL: a *url* que contém o recurso que será utilizado pela página;
- Parâmetros: Parâmetros a serem passados para a requisição. Se especificado, o pedido é feito utilizando protocolo POST, caso contrário, é utilizado o GET;
- *Callback:* uma função *callback* é chamado após o termino do carregamento do conteúdo do servidor.

Outros utilitários de alto nível para técnicas de AJAX são funções, não métodos, e eles são chamados diretamente através do método *jQuery*, ou *$*. O *jQuery.getScript()* carrega e executa arquivos de código *Javascript. jQuery.getJSON()* carrega uma URL, analisa como JSON, e passa o objeto resultante para o retorno de chamada especificada. Ambas utilizam o

jQuery.get(). E finalmente o *jQuery.post()* que funciona como um *jQuery.get()* mas utiliza o protocolo HTTP POST ao invés de um GET. Todas essas funções fazem requisições assíncronas, retornam para a aplicação antes de serem carregadas nas páginas e podem utilizar métodos de *callback* (FLANAGAN, 2011).

Segundo Bibeault e Kats (2008), muitos desenvolvedores de *web* têm utilizados os métodos GET e POST sem dar a atenção em como esses protocolos fazem as requisições. As intenções de cada método são as seguintes:

- Requisições GET: o resultado é idempotente, sempre é preciso. A mesma requisição pode ser feita várias vezes e o resultado será o mesmo;
- Requisições POST: o resultado não é idempotente, pode variar. Podem ser utilizados para mudar o estado do modelo da aplicação, por exemplo, adicionar novos registros ao banco de dados.

Essas funções e comandos que foram vistos até agora podem ser convenientes para alguns casos, mas pode haver momentos em que é preciso detalhar mais a requisição AJAX. Para esses momentos em que precisa-se desenvolver um AJAX com um nível de detalhe mais refinado, utiliza-se uma função de funcionalidade geral chamado *$.ajax()* (BIBEAULT; KATS, 2008).

A função aceita um único argumento: um objeto de opções contendo as propriedades do AJAX em detalhes para executar as requisições (FLANAGAN, 2011).

Segundo Bibeault (2008), essas opções são várias:

- *url*: a url para a requisição;
- *type*: o tipo de protocolo HTTP, GET ou POST;
- *data*: os dados que serão enviados;
- *datatype:* o tipo de dados que se espera para a resposta da requisição. Pode ser XML, JSON, HTML, JSONP, *script* ou *text;*
- *timeout:* estipulado um prazo para a requisição em milissegundos. Se a requisição não terminar no prazo estipulado é enviado uma mensagem de erro pelo método de *callback*;
- *contentType:* o tipo de conteúdo a ser especificado pela requisição. Caso não utilizado, o padrão é *application/x-www-form-urlencoded;*
- *sucess:* é invocado uma função caso a requisição retorne com sucesso;
- *error:* é invocado uma função caso a requisição retorne um status de erro;
- *complete:* uma função é invocada quando a requisição é completada;

- *async:* caso configurado como *false*, utiliza conexão síncrona;
- *processData:* se for definido como *false,* impede que os dados passados sejam processados pelo formato URL *Encoded*.
- *ifModified:* verifica se a requisição utiliza o mesmo *Header* que a ultima requisição feita.

3.5 JQUERY UI

O *JQuery User Interface* (*JQuery UI*) é uma extensão da biblioteca *JQuery* que permite adicionar interação, animação, aplicar temas e efeitos avançados nas aplicações web. Ao invés de desperdiçarmos tempo desenvolvendo complicados efeitos utilizando a biblioteca *JQuery*, podemos utilizar o *JQuery UI* para facilitar e assim produzir mais em menos tempo (CASTLEDINE; SHARKIE, 2010).

Para maximizar ainda mais a produtividade, o *JQuery UI* fornece em seu site, temas prontos e um gerador de temas (**Figura 5**) para customizar os atuais de acordo com a preferência do designer.

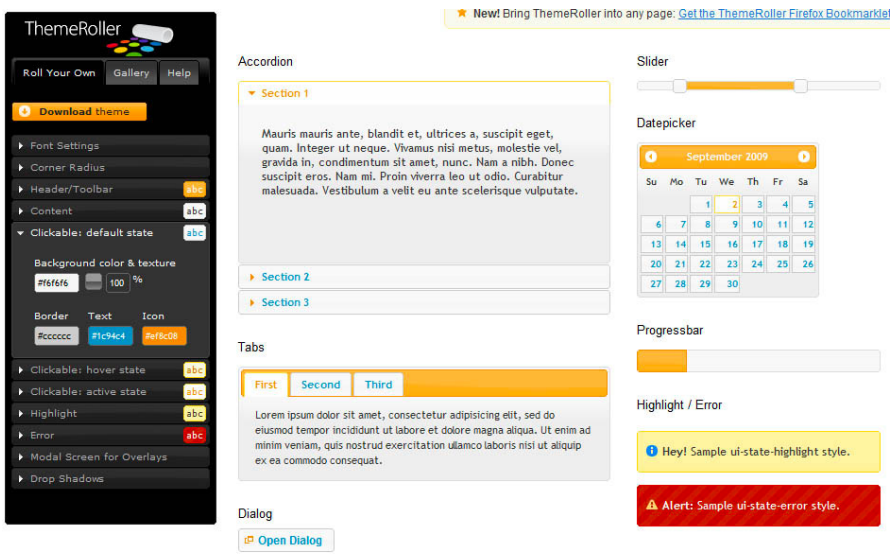


Figura 5 - Gerador de Temas JQuery UI.
Fonte: (JQUERY UI, 2011).

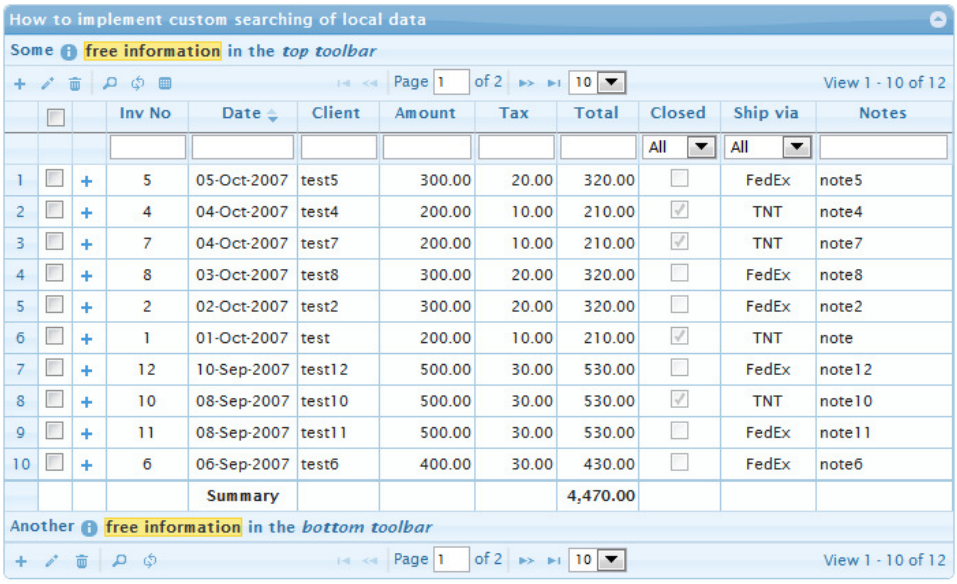
O gerador se encarregará de criar o código CSS de acordo com os padrões, garantindo que o tema funcionará da mesma maneira em diferentes *browsers* (CASTLEDINE; SHARKIE, 2011).

O CSS *Framework* do *JQuery UI* oferece várias classes prontas para o uso. Essas classes podem ser cabeçalhos, áreas para conteúdo ou estados dos elementos. O *framework* encapsula todo o código em apenas um arquivo chamado *ui.theme.css*. Esses códigos apenas incluem atributos que alteram a aparência da aplicação, então não existe a possibilidade do *framework* entrar em conflito com outro *plugin JQuery* (JQUERY, 2011).

3.6 JQGRID

Desenvolvido por Tony Tomov, pela *Trirand*, o *jqGrid*, assim como o *JQuery UI*, é uma extensão da biblioteca *JQuery*. O *plugin* utiliza como padrão a folha de estilos disponibilizada pelo *JQuery UI*. Tomov teve a idéia quando deparou com um problema em um de seus projetos. O projeto precisava representar os dados do banco de dados rapidamente e independentemente da linguagem ou banco de dados utilizado pelo servidor. A versão do *jqGrid*, desenvolvida por Tomov, é *open-source*, mas possui outras versões que são comercializadas pela *Trirand* (TRIRAND, 2011).

Como o *jqGrid* utiliza técnicas AJAX para carregar e manipular os dados, o *plugin* pode ser utilizado com qualquer linguagem de servidor como PHP, ASP.NET, *Java Servlets*, JSP, *ColdFusion* e *Perl* (TRIRAND, 2011).



		Inv No	Date	Client	Amount	Tax	Total	Closed	Ship via	Notes
								All	All	
1		5	05-Oct-2007	test5	300.00	20.00	320.00		FedEx	note5
2		4	04-Oct-2007	test4	200.00	10.00	210.00	✓	TNT	note4
3		7	04-Oct-2007	test7	200.00	10.00	210.00	✓	TNT	note7
4		8	03-Oct-2007	test8	300.00	20.00	320.00		FedEx	note8
5		2	02-Oct-2007	test2	300.00	20.00	320.00		FedEx	note2
6		1	01-Oct-2007	test	200.00	10.00	210.00	✓	TNT	note
7		12	10-Sep-2007	test12	500.00	30.00	530.00		FedEx	note12
8		10	08-Sep-2007	test10	500.00	30.00	530.00	✓	TNT	note10
9		11	08-Sep-2007	test11	500.00	30.00	530.00		FedEx	note11
10		6	06-Sep-2007	test6	400.00	30.00	430.00		FedEx	note6
			Summary				4,470.00			

Figura 6 - Exemplo de jqGrid.
Fonte: (TRIRAND, 2011).

A **Figura 6** demonstra a quantidade de recursos para customizar a *grid* que o *plugin* oferece. Pode-se adicionar, editar e deletar os registros da *grid*, fazer pesquisas utilizando filtros avançados, alterar o local dos botões, adicionar *toolbars*, habilitar *subgrids*, entre outras opções (TRIRAND, 2011).

O *plugin* é compatível com os *browsers Internet Explorer* 6.0+, *FireFox* 2.0+, *Safari* 3.0+, *Opera* 9.2+ e *Google Chrome* e suporta mais de 20 idiomas (TRIRAND, 2011).

4 ESTUDO DE CASO

Para demonstrar as vantagens em desenvolver uma aplicação em ASP.NET MVC utilizando *JQuery* e AJAX, foi elaborado um estudo de caso. A aplicação desenvolvida para o estudo conta com os pontos citados nos capítulos 2 e 3, e foi elaborada em conjunto com outro desenvolvedor que trabalhou apenas na camada de *model*, enquanto que, as camadas de *controller* e *view* serão descritas no presente trabalho, possibilitando a implementação de *JQuery* e o trabalho em equipe com o padrão MVC.

4.1 CONFIGURAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento da aplicação utiliza as seguintes ferramentas:

- Ferramenta de desenvolvimento *Visual Studio* 2010, para implementação da aplicação utilizando ASP.NET MVC 2;
- *JQuery* 1.5.2, biblioteca *Javascript*, responsável por permitir o desenvolvimento de Javascript utilizando comandos *JQuery*;
- *JQuery UI ThemeRoller* 1.8.1, responsável por customizar e disponibilizar um tema para o layout da aplicação;
- *JqGrid*, *plugin* do *JQuery*, para implementação de *DataGrids* utilizando *Javascript*;
- *Framework* para desenvolvimento de aplicações .NET *Framework* 4.0.

4.2 CONTEXTUALIZAÇÃO DA APLICAÇÃO

Com o uso da ferramenta *Visual Studio* 2010 foi elaborado uma aplicação para demonstrar o desenvolvimento em ASP.NET MVC. A aplicação conta com a biblioteca *JQuery* para a implementação das técnicas de AJAX e simplificação do código *Javascript*, além de auxiliar na customização do layout a partir do uso do *plugin JQuery User Interface* e manipulação de dados com *JqGrid*.

4.3 REQUISITOS DA APLICAÇÃO

Foi optado pelo desenvolvimento de uma aplicação para controle de bens patrimoniais de uma empresa. A aplicação deverá atender os seguintes requisitos:

- Manter os dados de todos os bens da empresa, bem como seus responsáveis;
- Cada usuário deverá ter um registro, com *login* e senha; e dados que serão vinculados aos bens;
- Os usuários poderão ser funcionário ou gerente;
- O gerente possui permissões para cadastrar novos funcionários, bens, solicitar transferências de bens para outras áreas e também aceitar ou recusar solicitações de bens dos funcionários a seu encargo;
- Os usuários poderão apenas ver os bens pelos quais são responsáveis e solicitar novos bens;
- O usuário administrador é o único usuário que possui permissão para cadastrar um novo gerente.

4.4 DESENVOLVENDO A APLICAÇÃO

4.4.1 Criando Um Projeto ASP.NET MVC

Cria-se um projeto ASP.NET MVC *Web Application* selecionando a opção *New Project* do *Visual Studio 2010*. Com a criação do projeto, pode-se desenvolver em ASP.NET MVC e utilizar as bibliotecas *JQuery* que vem por padrão no ambiente. A **Figura 7** mostra a tela de criação de projetos.

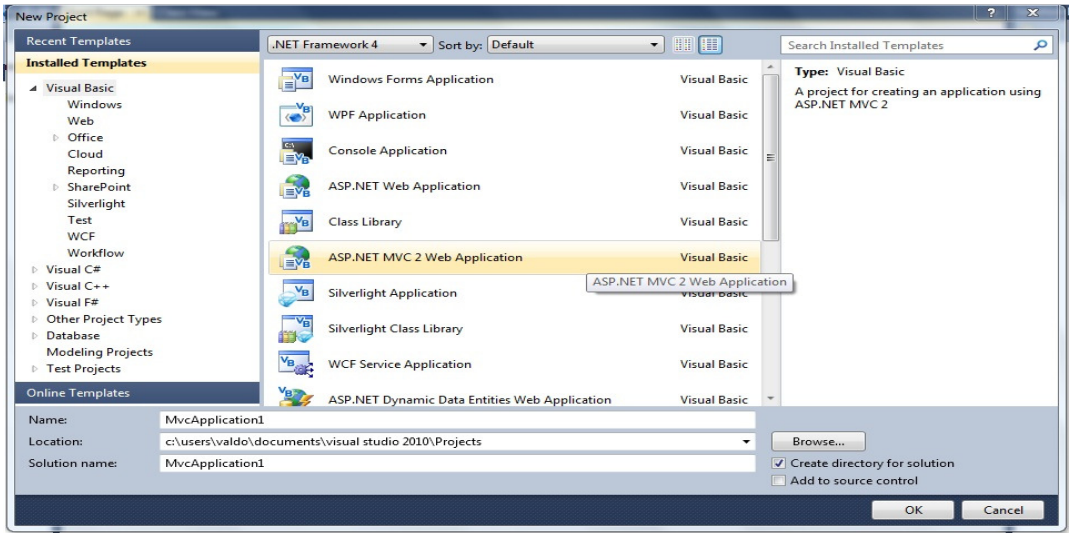


Figura 7 - Criando projeto no Visual Studio 2010.
Fonte: Autoria Própria.

O *Visual Studio 2011* cria por padrão algumas pastas contendo *scripts, contents, views, controllers* e *models* como aplicação exemplo de ASP.NET MVC, conforme demonstra a **Figura 8**.

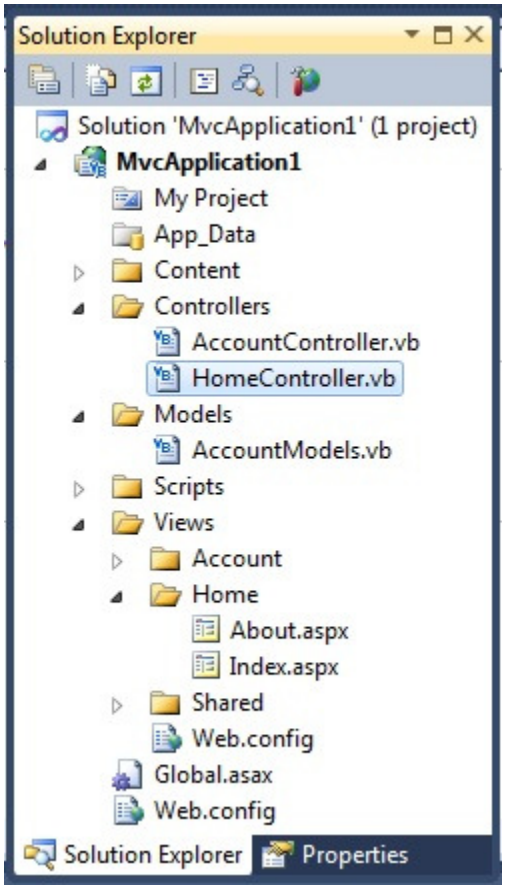


Figura 8 - Solution Explorer.
Fonte: Autoria Própria.

4.4.2 Configurando Projeto e Bibliotecas

Foi optado por desenvolver a lógica de negócio da aplicação em um projeto separado do projeto *web*. Para que os projetos trabalhem em conjunto, deverá ser referenciado no projeto *web* o projeto que contém a lógica de negócio. Para isso, deve-se ir nas opções de referencia do projeto, clicando com o botão direito do *mouse* em *references* e selecionando a opção *Add Reference*. As opções de *references* do projeto deverão ser configuradas como mostra a **Figura 9**.

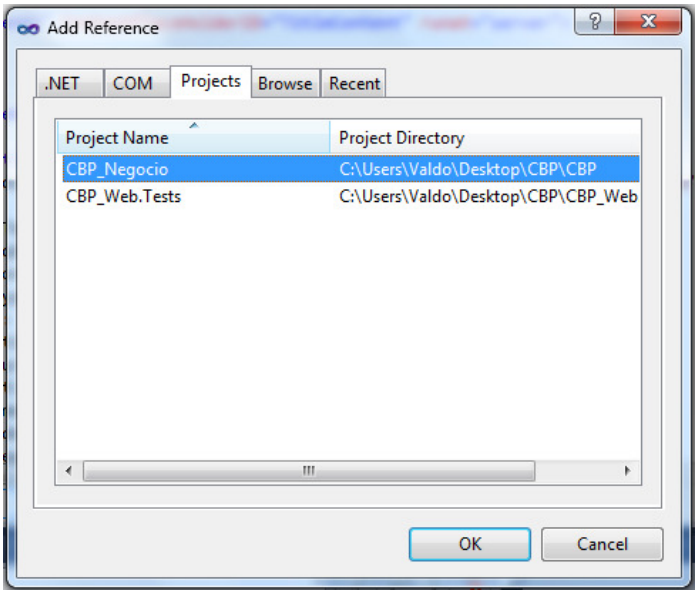


Figura 9 - Adicionar referencia.
Fonte: Autoria Própria.

Deve-se configurar também para que o projeto aceite comandos *JQuery* nas *views*. Para isso, é adicionado no *Site.Master* do projeto *web*, os seguintes comandos:

```
1 <link href="/Scripts/css/custom-theme/jquery-ui-1.8.13.custom.css" rel="stylesheet"
2 type="text/css" />
3 <script src="/Scripts/js/jquery-1.5.1.min.js" type="text/javascript"></script>
4 <script src="/Scripts/js/jquery-ui-1.8.13.custom.min.js"
5 type="text/javascript"></script>
6 <script src="/Scripts/js/util.js" type="text/javascript"></script>
7 <!-- JQGRID -->
8 <link href="/Scripts/css/custom-theme/ui.jqgrid.css" rel="stylesheet"
9 type="text/css" />
10 <script type="text/javascript" src="../../Scripts/js/grid.locale-pt-
11 br.js"></script>
12 <script type="text/javascript"
13 src="../../Scripts/js/jquery.jqGrid.min.js"></script>
```

Quadro 2 - Configurando JQuery.

O **Quadro 2** demonstra a inclusão das bibliotecas *JQuery* no projeto. Cada linha referencia um arquivo *script* ou CSS:

- Linha 1: insere o arquivo "*jquery-ui-1.8.13.custom.css*" do *plugin JQuery UI*, assim pode-se utilizar todas as propriedades de folha de estilo que foram customizadas pelo gerador de temas do site da *JQuery UI*;
- Linha 3: adiciona na aplicação, as bibliotecas do *JQuery*;
- Linha 4: insere o arquivo "*jquery-ui-1-8-13.custom.js*" para que a aplicação tenha todas as vantagens do *plugin JQuery UI*;
- Linha 8: fornece a folha de estilos para o *plugin* do *JqGrid*;
- Linha 10: fornece o *script* de idiomas do *JqGrid*, nesse caso, o idioma português do Brasil, arquivo "grid.locale-pt-br.js";
- Linha 12: fornece o *script* para a execução do *plugin JqGrid*.

Com o projeto devidamente configurado, pode-se começar o desenvolvimento da aplicação.

4.4.3 Criando Controllers

Para criar um *controller*, basta clicar com o botão direito do *mouse* na pasta *controller* e posteriormente na opção *add controller*. Por padrão o *Visual Studio* gera automático um código padrão para a *controller* contendo apenas um método que serve para chamar uma *view* chamada *index*.

```
[HandleError]
    public class AreaController : Controller
    {

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

        public ActionResult Editar(int id)...

        public ActionResult Consultar()...

        [HttpPost]
        public JsonResult Consultar(GridSettings grid, Area vArea)...
```

```
        [HttpPost]
        public JsonResult Salvar(Area area)...

        public ActionResult Excluir(int id)...
}
```

Quadro 3 - AreaController.

A **Quadro 3** demonstra os métodos da classe *AreaController*. Os métodos *Inserir(), Consultar(), Editar()* e *Excluir()*retornam um *ActionResult*. Nesses casos, os métodos utilizam GET como protocolo HTTP e retornam uma *view* como resposta. A *view* que o método deverá retornar segue as seguintes regras:

1. O nome do método deverá ser o mesmo nome da *view*.
2. O nome da pasta aonde a *view* se encontra deverá ter o nome da *controller* sem o *Controller* no final.

O método *Inserir()* da classe *AreaController* deverá retornar uma *view* chamada *Inserir.aspx* que se encontra na pasta *Area*do projeto e o método *Consultar*, retorna uma *view* chamada *Consultar.aspx* e que se encontra na mesma pasta.

```
public ActionResult Editar(int id)
{

    IManager<Area> areaManager = new AreaManager();

    Area area = new Area();
    area.IdArea = id;
    area = areaManager.FindById(area);

    IManager<Usuario> usuarioManager = new UsuarioManager();

    IEnumerable<Usuario> gerentelist =
usuarioManager.FindAll().AsEnumerable<Usuario>();
    ViewData["GerenteList"] = new SelectList(gerentelist, "CODIGO", "NOME");

    return View(area);
}
```

Quadro 4 - Método Editar.

Diferente dos métodos *Inserir()* e *Consultar()*, os métodos *Editar()* e *Excluir()*, recebem o parâmetro *id*. Ao invés de apenas retornar para uma *view*, o método *Editar()* (**Quadro 4**), recupera um registro no banco de dados utilizando o *id* e envia este objeto para a *view*. Os objetos podem ser enviados pela *ViewPage.Model*, pela *ViewData* ou passando pelo retorno da *view* (*return view(area)).*

O método *Excluir()* deleta um registro do banco de dados utilizando o *id* passado como parâmetro e redireciona a página para a *view Consultar.aspx* utilizando o método *RedirectToAction("Consultar")*como retorno *ActionResult*.

Além dos métodos que retornam *ActionResult*, a *controller* possui outros métodos.Estes métodos possuem a anotação *[HttpPost]* e retornam um *JsonResult.*Métodos que possuem esta anotação aceitam apenas requisições de protocolo HTTP POST e não precisam necessariamente retornar uma *view.*

```
[HttpPost]
public JsonResult Salvar(Area area)
{
    try
    {
        IManager<Area> areaManager = new AreaManager();

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

    return new JsonResult();
}
```

Quadro 5 - Método Salvar.

O método *Salvar()* (**Quadro 5**)é um dos métodos anotados pelo *[HttpPost]* e espera um objeto da classe Area.Os dados que serão recebidos do cliente serão inseridos no banco de dados utilizando o método *Save()* da classe *AreaManager* do projeto de lógica de negócio, camada de modelo do projeto. Para que a aplicação cliente saiba que o método *Save()* funcionou corretamente, é enviado um *new JsonResult()*. Este método é utilizado tanto para inserir novos registros quanto para editar antigos.Isso é possível porque o método apenas insere os dados no banco se o objeto não tiver um *id* definido ainda, caso contrário, busca o registro que possui o *id* daquele objeto e atualiza os atributos modificados.

```
[HttpPost]
public JsonResult Salvar(SolicitacaoEntrada sol, String ids)
{
    try
    {
        SolicitacaoEntradaManager solManager = new SolicitacaoEntradaManager();
        SolicitacaoEntrada solEntrada = new SolicitacaoEntrada();
        solEntrada.Data = DateTime.Now;
        solEntrada.IdStatusSol = Solicitacao.ENVIADA;
        solEntrada.Descricao = sol.Descricao;

        var id = ids.Split(',');

        var qtdade = id.Count();

        List<BemPatrimonial> bemPatrimoniais = new List<BemPatrimonial>();
        IManager<BemPatrimonial> bemManager = new BemPatrimonialManager();

        for (int i = 1; i < qtdade;i++)
        {
```

```
            BemPatrimonial b = new BemPatrimonial();
            b.IdBem = System.Convert.ToInt32(id.GetValue(i));
            b = bemManager.FindById(b);
            b.IdStatusBem = Status.EM_USO;
            bemManager.Save(b);
            bemPatrimoniais.Add(b);
        }

        var usuario = (Usuario)Session["UsuarioLogado"];
        solEntrada.IdUsuario = usuario.IdUsuario;
        solEntrada.IdArea = usuario.IdArea;

        solEntrada = solManager.solicitarEntrada(solEntrada, bemPatrimoniais);

    }
    catch (Exception e)
    {
        Response.Write(e.Message);
        return new JsonResult();
    }

    return new JsonResult();
}
```

Quadro 6 - Método Salvar SolicitacaoEntradaController.

O método *Save()* da *SolicitacaoEntradaController* (**Quadro 6**) deverá se preocupar não apenas em inserir uma nova solicitação de entrada mas também em relacionar essa solicitação aos bens patrimoniais selecionados pelo usuário. Para fazer essa relação, a *view* envia para o método *Save()* os *ids* dos bens patrimoniais que foram selecionados na *grid*. O método deverá fazer uma busca para recuperar os bens. Feito isso, deverá passar como parâmetro os bens e a solicitação de entrada no método *solicitarEntrada()* da classe *SolicitacaoEntradaManager* do projeto de negócio da aplicação.

```
[HttpPost]
public JsonResult Consultar(GridSettings grid, Area vArea)
{
    IManager<Area> areaService = new AreaManager();

    var query = areaService.Find(vArea).AsQueryable();

    //count
    var count = query.Count();

    //paging
    var data = query.Skip((grid.PageIndex - 1) *
grid.PageSize).Take(grid.PageSize).ToArray();

    var usuarioManager = new UsuarioManager();
    //converting in grid format
    var result = new
    {
        total = (int)Math.Ceiling((double)count / grid.PageSize),
        page = grid.PageIndex,
        records = count,
        rows =  (from varea in data
```

```
                select new
                {
                    idArea = varea.IdArea.ToString(),
                    dsDescricao = varea.Descricao,
                    dsGerente = varea.IdGerente
                }).ToArray()
    };

    return Json(result, JsonRequestBehavior.AllowGet);
}
```

Quadro 7 - Método Consultar AreaController.

Quando a *JqGrid* da *view* requisita os dados do banco de dados é chamado o método *Consultar()* (**Quadro 7**) que possui a anotação *[HttpPost]*. O método busca todos os registros do banco de dados e prepara essas informações para serem exibidas na *grid* implementada pelo *JqGrid* da *view Consultar.aspx.* Para preparar os dados para a *grid*, primeiro o método utiliza o objeto *GridSettings* que recebeu por parâmetro para identificar qual é o index atual da *grid* e a quantidade de registros permitido por página. Essas informações são utilizadas para filtrar apenas os registros que deverão ser usados pela *view*. Feito isso, é declarado um *object* que recebe:

- *total*: número total de páginas que a *grid* deverá ter;
- *page:* primeira página da *grid* a ser exibida;
- *records:* quantidade de registros que a *grid* possui;
- *rows:* insere um *array* de Area que foi recuperada pelo banco de dados.

Por fim, é retornado um *JsonResult*, criado a partir do *object result* que possui todos os dados da consulta.

4.4.4 Criando Views

View é a camada mais próxima do usuário e responsável por exibir as informações do banco de dados e interagir com o usuário. Nesse estudo de caso, a *view* encarregada de fazer a inserção de novos registros no banco de dados faz uma requisição AJAX através do *JQuery* para enviar os dados dos formulários para uma *controller*. A *controller* interpretará a ação e deverá chamar os métodos da camada de modelo que são necessários para inserir os dados enviados pela *view* no banco de dados da aplicação.

```
<fieldset class="ui-widget-content">
```

```
    <div class="divLinha">
        <%= Html.Text("Descricao", Html.Resource("bundle, area_descricao") + ":",
15)%>
        <%= Html.TextBoxFor(model => model.Descricao, new { id = "Descricao", style =
"width: 10%;" })%>
    </div>
</fieldset>

<div class="divLinha">
    <button id="btnAdd">Cadastrar Area</button>
</div>
<br />

<div id="dialog-message" title="Cadastro de Área" style="display: none;">
    <p>
        <span class="ui-icon ui-icon-circle-check" style="float:left; margin:0 7px
50px 0;"></span>
        Área cadastrada com sucesso
    </p>
</div>
```

Quadro 8 - **Corpo da página HTML Inserir.**

A **Quadro 8** demonstra o conteúdo do corpo da páginas HTML criada para inserir novos registros da classe Area. O corpo da página possui alguns métodos que são chamados de HTML *helpers*. Esses métodos vieram por padrão a partir da versão 2 do ASP.NET MVC. Os HTML *Helpers* auxiliam a escrever *tags* HTML como *inputs*, *buttons*, entre outros. No caso da **Quadro 8**, há três *helpers*:

- *Html.Resource*: utiliza-se este método para acessar um arquivo de propriedades chamado "*Bundle.resx*" que contem todas as descrições da aplicação;
- *Html.Text*: retorna uma tag *label* no código HTML;
- *Html.TextBoxFor:*retorna uma tag *input* de entrada de dados gerado a partir de um *model*, nesse caso, a partir do modelo de Area.

Além de existirem outros vários, é possível criar novos *helpers*.

A **Quadro 8** também mostra que a *view* possui uma div com id "*dialog-message*" que está configurado com a propriedade *display:none* e um *button* com id "*btnAdd*" que deverá ser utilizado para enviar os dados dos formulários para a *controller*.

```
$("#btnAdd").click(function () {
        $.ajax({
            url: url + "/Area/Salvar",
            type: "post",
            data: {
                "Descricao": $("#Descricao").val(),
                "IdGerente": $("#IdGerente").val()
            },
            dataType: "json",
            error: function (xhr, ajaxOptions, thrownError) {
                showErrorMessage(xhr.responseText);
            },
```

```
            success: function (data, textStatus, XMLHttpRequest) {
                $("#dialog-message").dialog({
                    modal: true,
                    buttons: {
                        Ok: function () {
                            $(this).dialog("close");
                            redirect(url+"/Area/Consultar");
                        }
                    }
                });
                $("#dialog-message").dialog("show");
            }
        });
});
```

Quadro 9 - Script AJAX.

Para enviar os dados para a *controller*, utiliza-se o protocolo HTTP. O *script* da **Quadro 9** demonstra como utilizar as técnicas de AJAX através do *JQuery* para fazer uma requisição POST para o *controller*. Esse *script* é acionado quando há um evento de *click* no botão "*btnAdd*". Para enviar os dados, primeiro, o AJAX utiliza a URL "HTTP://localhost:2504/Area/Consultar", onde o "*localhost:2504*" é o *root* do *link*, a "Area" é a *controller* encarregada de processar esta ação e o "Salvar" é o método da *controller* que deverá ser utilizado.Depois, é configurado através do atributo *type* se a requisição utilizará o protocolo HTTP POST ou GET. Em seguida, o atributo *data* recebe um *array* com os valores dos campos Descrição e IdGerente que são os dados da Area a ser inserida no banco. Por fim, o método faz a requisição e em casos de erros no servidor, o *script* lança um aviso padrão do *JQuery* e em casos de sucesso, manda exibir na tela a div "*dialog-message*", que estava com a propriedade *display:none*.

```
<fieldset class="ui-widget-content">

    <div class="divLinha">
    <%= Html.HiddenFor(model => model.IdArea, new { id = "IdArea", style = "width:
10%;" })%>
        <%= Html.Text("Descricao", Html.Resource("bundle, area_descricao") + ":",
15)%>
        <%= Html.TextBoxFor(model => model.Descricao, new { id = "Descricao", style =
"width: 10%;" })%>
    </div>
    <div class="divLinha">
        <%= Html.Text("IdGerente", Html.Resource("bundle, area_gerente") + ":", 15)%>
        <%: Html.DropDownList("comboGerente",
(IEnumerable<SelectListItem>)ViewData["GerenteList"], new { id = "idGerente",

style = "width: 20%;"

})%>
    </div>
</fieldset>
<div class="divLinha">
    <button id="btnAdd">Editar Area</button>
```

```
</div>
```

Quadro 10 - Editar Area.

Uma *view* de edição (**Quadro 10**) possui praticamente os mesmos códigos que uma *view* de inserção. Mas existem duas diferenças:

- Um objeto é recebido pela *controller* para preencher os *inputs* do formulário com os dados da Area;
- É adicionado um *Html.HiddenFor()* que serve para inserir o código da area que esta sendo editada. Esse *helper* esconde do usuário o *input* para que o valor do código da area não seja alterado.

```
$("#btnAdd").click(function () {
    $.ajax({
        url: url + "/Area/Salvar",
        type: "post",
        data: {
            "IdArea" : $("#IdArea").val(),
            "Descricao": $("#Descricao").val(),
            "IdGerente": $("#IdGerente").val()
        },
        dataType: "json",
        error: function (xhr, ajaxOptions, thrownError) {
            showErrorMessage(xhr.responseText);
        },
        success: function (data, textStatus, XMLHttpRequest) {
            $("#dialog-message").dialog({
                modal: true,
                buttons: {
                    Ok: function () {
                        $(this).dialog("close");
                        redirect(url + "/Area/Consultar");
                    }
                }
            });
            $("#dialog-message").dialog("show");
        }
    });

});
```

Quadro 11 - Script Editar Area.

O *script* (**Quadro 11**) da *view* de edição é praticamente igual ao de inserção. A diferença é que dessa vez é enviado o *IdArea* também para a *controller*.

```
function criaTabela(caption, colNames, colModels) {
    $("#solEntradaGrid").jqGrid({
        caption: caption,
        datatype: "json",
        mtype: "post",
        traditional: true,
        url: url + "/Consultar",
        height: 220,
        colNames: colNames,
        colModel: colModels,
```

```
        jsonReader: {
            root: "rows",
            page: "page",
            total: "total",
            records: "records",
            repeatitems: false,
            userdata: "userdata"
        },
        scroll: 1,
        rowNum: 100,
        gridview: false,
        subGrid: false,
        autowidth: true,
        sortname: "dsNome",
        sortorder: "asc",
        subGrid: true,
        subGridUrl: url + '/Detalhar',
        subGridModel: [{
            name: [
            'idBem',
            'Descricao',
            'Status',
            'Usuario'],
            width: [100, 100, 100, 100],
            params: ['IdEntrada']
        }],
        loadComplete: function () {
            var grid = $("#solEntradaGrid");
            var subGridCells = $("td.sgcollapsed", grid);
            $.each(subGridCells, function (i, value) {
                var rowData = grid.getRowData(i + 1);
                var colData = rowData["IdEntrada"];

                idEntrada = $(colData).html();
                if (idEntrada == 0) {
                    $(value).unbind('click').html('');
                }
            });
        },
        pager: '#solEntradaPager',
        viewrecords: true,
        gridComplete: function () {
        }
        // Este trecho desabilita a seleção de uma ROW na grid
        //beforeSelectRow: function () {
        //    return false;
        //}
    });
    $("#solEntradaGrid").jqGrid('navGrid', '#solEntradaPager', { edit: false, add:
false, del: false, search: false, refresh: false });
};
```

Quadro 12 - Grid Para Consulta de Area.

O *script* da *view Consultar* da *AreaController* (**Quadro 12**) utiliza o *JqGrid* para criar uma *grid* com dados da Area. Para isso, quando a página é carregada, utiliza-se a função *criarTabela()* que irá criar uma *grid* utilizando os dados do *model*.

Para utilizar *JqGrid* é preciso criar duas *div* na página HTML, uma *div* será o *grid* e a outra será o rodapé da *grid*. Quando acionado, a função *criarTabela()* utiliza *JQuery* para selecionar a *div* "*#areaGrid*" e em seguida utiliza *jqGrid* para setar os valores necessários para criar uma *grid*. Nesse caso, utilizam-se as propriedades:

- *caption*: título da *grid*;
- *datatype*: tipo de dados que serão recebidos do servidor, XML ou JSON;
- *mtype*: tipo de protocolo HTTP utilizado;
- *url*: URL do servidor;
- *height*: altura do *grid*;
- *colNames*: nome das colunas;
- *colModel*: modelo das colunas;
- *jsonReader*: mapeamento dos dados JSON que serão recebidos da *controller*;
- *scroll*: se o *grid* deverá ter barra de rolagem;
- *rowNum*: numero de linhas;
- *gridview*: utilizado para quando se espera muitos registros do servidor, nesses casos, é aconselhado a utilização dessa propriedade para aumentar a velocidade da *grid* mas é preciso levar em consideração que, caso ativado, a *grid* não poderá mais utilizar as opções de *subGrid*, *treeGrid* ou evento *afterInsertRow*;
- *subGrid*: *boolean*, permite que cada registro possua uma *subgrid*;
- *autowith*: para redimensionar automaticamente o comprimento da *grid*;
- *pager*: *id* da *div* que será o rodapé da *grid*;
- *sortname*: com qual coluna deverá ser ordenado por padrão os registros da *grid*;
- *sortorder*: se a ordenação é crescente ou decrescente;
- *viewrecords*: exibe a quantidade de registros que estão sendo exibidos por páginas;
- *gridComplete*: para executar um algoritmo ao termino da *grid*;
- *beforeSelectRow*: quando o usuário selecionar um registro da *grid* esta função será executada.

Por último, após o termino da montagem da *grid*, é utilizado o método *jqGrid* para definir quais botões serão exibidos na barra de navegação da *grid*.

```
$(document).ready(function () {

    $("#tabs").tabs();

    $("#btnNovo").click(function () {
        redirect(url + "/Inserir");
```

```
    });

    var caption = '<%:Html.Resource("bundle, sol_entrada_grid")%>';
    var colNames = ['','',
        '<%:Html.Resource("bundle, sol_entrada_usuario")%>',
        '<%:Html.Resource("bundle, sol_entrada_area")%>',
        '<%:Html.Resource("bundle, area_descricao")%>',
        '<%:Html.Resource("bundle, sol_entrada_data")%>',
        '<%:Html.Resource("bundle, sol_entrada_status")%>',
        '',
        ''
    ];

    var colModels = [
        { name: 'IdEntrada', index: 'IdEntrada', hidden: true },
        { name: 'idTipo', index: 'idTipo', hidden: true },
              { name: 'Usuario', index: 'Usuario', width: 180, formatter: gridDado,
align: "left" },
              { name: 'Area', index: 'Area', width: 80, formatter: gridDado, align:
"left" },
              { name: 'Descricao', index: 'Descricao', width: 80, formatter: gridDado,
align: "left" },
              { name: 'Data', index: 'Data', width: 80, align: "left" },
         { name: 'Status', index: 'Status', width: 80, formatter: gridDado, align:
"left" },
         { name: 'editar', width: 35, formatter: gridBotaoEditarAceitar, sortable:
false },
         { name: 'excluir', width: 35, formatter: gridBotaoExcluir, sortable: false }
    ];

    criaTabela(caption, colNames, colModels);
});
```

Quadro 13 - **Grids Para Consulta de Area Parte 2.**

O *script* (**Quadro 13**) declara três funções que serão utilizados pela *grid*. Os três métodos estão esperando os parâmetros:

- *cellValue*: será o valor de cada célula da *grid*;
- *options*:são as opções que foram definidas para a célula;
- *rowObject*: é um objeto que possui os dados de cada linha da *grid*;

  Cada função possui um objetivo diferente:

- *gridDado()*: apenas insere os dados da célula em uma div para caso seja de interesse mudar as propriedades de estilo das células;
- *gridBotaoEditar():* cria um botão para editar cada registro da *grid*, o botão referencia o *link* para a edição do objeto na *controller* e também adiciona no *link* o *id* da area, usando o *rowObject.idArea*;
- *gridBotaoExcluir():* faz quase o mesmo procedimento que o *gridBotaoEditar()* mas, ao invés de editar, exclui os registros da *grid*.

Em seguida vem o *$(document).ready* que é a função que será chamada quando a páginas estiver carregada. Esta função irá criar um evento de *click* no botão "*btnNovo*" para referenciar o método *Inserir()* da *AreaController* que, como explicado anteriormente, irá retornar uma *view*.

Para poder utilizar o método *criaTabela()* e criar a *grid*, antes, é declarado 3 variáveis:

- *var caption*: *string* que irá conter o título da *grid,* no caso da **Quadro 13**, também é utilizado o *Html.Resource* para adquirir a descrição do título a partir do *Bundle.resx*;
- *var colNames*: *array* contendo o cabeçalho da *grid*, também utiliza o *Bundle.resx*;
- *var colModels*: fornece o modelo de dados e o numero de colunas para o modelo.

Podem ser definidas várias propriedades para o *colModel*s, no caso da **Quadro 13**, é definido:

1. *name* e *index*: o nome utilizado na *controller* para identificar as colunas dos registros no arquivo JSON;
2. *hidden*: apenas utilizado na primeira coluna, serve para esconder o valor dessa célula, no caso da **Quadro 13**, não é interessante que o usuário saiba qual o valor do id de cada Área;
3. *width:* comprimento da célula;
4. *align:* alinhamento da célula;
5. *formatter:* é aonde definimos a formatação de cada coluna. Existem várias formatações padrões no *jqGrid* mas no caso da **Quadro 13**, foi utilizado as funções *gridDado()*, *gridBotaoEditar()* e *gridBotaoExcluir().*

Uma *grid* normal é o suficiente para listar todos os registros da *Area* mas quando se tem uma relação de muitos para muitos entre duas classes como o *SolicitarEntrada* tem com os *BensPatrimoniais*, é necessário ir além de uma simples *grid.* Para poder listar as solicitações de entrada de bem e ao mesmo tempo listar quais bens estão sendo solicitados em cada registro, foi utilizado *subGrid*. As configurações da *jqGrid* deverão seguir o mesmo conceito mostrados no exemplo de *grid* de *Area.* A única propriedade que deverá ser modificada é a propriedade de *subGrid*.

```
subGrid: true,
subGridUrl: url + '/Detalhar',
subGridModel: [{
    name: [
    'idBem',
    'Descricao',
    'Status',
    'Usuario'],
    width: [100, 100, 100, 100],
```

```
    params: ['IdEntrada']
}],
loadComplete: function () {
    var grid = $("#solEntradaGrid");
    var subGridCells = $("td.sgcollapsed", grid);
    $.each(subGridCells, function (i, value) {
        var rowData = grid.getRowData(i + 1);
        var colData = rowData["IdEntrada"];

        idEntrada = $(colData).html();
        if (idEntrada == 0) {
            $(value).unbind('click').html('');
        }
    });
},
```

Quadro 14 - Exemplo de SubGrid.

A **Quadro 14** demonstra como deve ser feita a configuração das propriedades da *subGrid*. Primeiro, deve-se adicionar a propriedade *subGridUrl* que será a URL para chamar o método *Detalhar()* da *SolicitacaoEntradaController*. Depois se define o nome das colunas através da propriedade *name*, o tamanho de comprimento da *subGrid (width)* e o parâmetro que deverá ser enviado ao método *Detalhar()* para identificar os bens de qual solicitação a aplicação precisa.

Por último, é preciso configurar a função *loadComplete* da *grid* para que quando seja clicado no botão "+", a *subGrid* receba o parâmetro *IdEntrada* da linha selecionada. A **Figura 10** mostra o funcionamento da *subGrid.*

Figura 10 - Exemplo de SubGrid.
Fonte: Autoria Própria.

Mas não é apenas a *view* de consulta que precisa ser diferente. As *views* para os métodos *Inserir()* e *Editar()* da *SolicitacaoEntradaController* também precisam ser repensados.

A *view* que insere novos registros de *SolicitacaoEntrada* precisa inserir também os bens patrimoniais relacionados a essas solicitações. Para isso é preciso criar uma *grid*

contendo todos os bens patrimoniais disponíveis em depósito para que o usuário possa escolher qual ou quais bens deverá ser solicitado.

```
$("#btnAdd").click(function () {

    var s;
    s = jQuery("#bemGrid").jqGrid('getGridParam', 'selarrrow');
    var i;
    i = 0;
    var ids;
    ids = "";
    while (i < s.length) {
        var myrow = jQuery('#bemGrid').jqGrid('getRowData', s[i]);
        ids += "," + myrow.idBem;
        i++;
    }
    $.ajax({
        url: url + "/SolicitacaoEntrada/Salvar",
        type: "post",
        data: {
            "Descricao": $("#Descricao").val(),
            "ids": ids
        },
        dataType: "json",
        error: function (xhr, ajaxOptions, thrownError) {
            showErrorMessage(xhr.responseText);
        },
        success: function (data, textStatus, XMLHttpRequest) {
            $("#dialog-message").dialog({
                modal: true,
                buttons: {
                    Ok: function () {
                        $(this).dialog("close");
                        redirect(url + "/SolicitacaoEntrada/Consultar");
                    }
                }
            });
            $("#dialog-message").dialog("show");
        }
    });

});
```

Quadro 15 - Script AJAX SolicitacaoEntrada.

O *script* (**Quadro 15**) demonstra como ficaria para enviar os dados para a *controller.* Foram adicionados no *script* os códigos que estão entre as linhas 64 e 74. Esse algoritmo identifica quais foram os bens patrimoniais selecionados pelo usuário na *grid* e coloca todos os *ids* em um *string* separando cada *id* por uma vírgula “,”, assim o *controller* poderá separar essa *string* e identificar quais bens patrimoniais deverão ser relacionados a classe *SolicitacaoEntrada.*

```
function selectRows() {
    var grid = jQuery("#bemGrid");
    var rowId = new Array();
    var i = 0;
```

```
    <%
        List<CBP.Entity.BemPatrimonial> bem = new List<CBP.Entity.BemPatrimonial>();
        bem = (List<CBP.Entity.BemPatrimonial>) ViewData["BemList"];

        foreach(CBP.Entity.BemPatrimonial b in bem){
            %>
        rowId[i] = '<%:b.IdBem %>';
        grid.setSelection(rowId[i]);
        i++;
            <%
        }
    %>

}
```

Quadro 16 - Script SelectRows.

Para editar os valores de uma solicitação de entrada de bem (**Quadro 16**), é necessário que os bens patrimoniais sejam selecionados na *grid* da *view* do método *Editar()* da *controller*. Para selecionar os bens que possuem relação com a solicitação de entrada utiliza-se o método *selectRows()*.Utiliza-se o método quando a *grid* já estiver carregada, ou seja, na propriedade *gridComplete* do *JqGrid.* Também é necessário que a *controller* envie para a *view* através de um *ViewData*, uma *List* dos bens patrimoniais da solicitação de entrada que deverá ser editada. O método *selectRows()* irá percorrer a lista de bens patrimoniais e em cada laço de repetição, irá selecionar o bem patrimonial da *grid* que possui o *IdBem* do objeto atual da lista.

4.4.5 JQuery UI

Existem duas maneiras de utilizar as vantagens da biblioteca *JQuery UI.* Pode-se utilizar apenas as classes da folha de estilos CSS ou utilizar comandos *JQuery* para definir *buttons, widgets,* animações, efeitos, entre outros.

```
<div class="ui-widget-content ui-corner-all">
    <asp:ContentPlaceHolder ID="MainContent" runat="server" />

    <div id="footer">
    </div>
</div>
```

Quadro 17 - CSS Framework.

A **Quadro 17** mostra a utilização do *framework* de classes CSS do *JQuery UI.* Nesse caso, utiliza-se apenas duas classes:

- *ui-widget-content*: classe para ser atribuída a um container de *widgets*;

- *ui-corner-all:* utiliza-se essa classe para criar curvas nas pontas das bordas.

  As classes do *framework* podem ser:

- *Layout Helpers*: auxiliares para o layout;
- *Widget Containers*: classes utilizadas em widgets;
- *Interaction States:* classes para atribuir os estados dos elementos;
- *Interaction Cues:* classes para serem atribuídas a elementos de dicas;
- *Icons:* classes para inserir ícones;
- *Misc Values:* classes para fazer curvas nas bordas e sombras nas widgets;

```
function runEffect(button) {
    var options = {};
    //options = { to: { width: 100, height: 60} };
    $(button).effect("shake", options, 500, callback);
}

$(document).ready(function () {
    $(function () {
        $("#btnNovo").button();

        $("#btnNovo").click(function () {
            runEffect($(this));
        });
```

Quadro 18 - Efeitos com JQuery UI.

A **Quadro 18** demonstra a utilização do *plugin JQuery UI* para atribuir classes e efeito de "*shake*" a botões. Quando um elemento chama uma função *button()*, automaticamente define todas as propriedades CSS de *button* do *JQuery UI* para aquele elemento. No exemplo demonstrado na **Quadro 18** quando é acionado um evento de *click* no botão *btnNovo*, chama-se a função *runEffects()*. A função chamará o método *effects* do elemento e irá atribuir como efeito o "*shake*". O efeito "*shake*" faz com que o botão fique "balançando" por alguns segundos antes de executar qualquer outra operação.

```
<div id="tabs" class="fieldsetDados">
    <ul>
            <li><a href="#tabTransPendencias">Transferencias Pendentes</a></li>
            <li><a href="#tabTrans">Solicitações de Transferência</a></li>
      </ul>

    <div id="tabTrans">
        <div class="grid">
            <table id="transGrid"></table>
            <div id="transPager"></div>
        </div>
    </div>
    <div id="tabTransPendencias">
        <br />
        <div class="grid">
            <table id="transPendenciasGrid"></table>
            <div id="transPendenciasPager"></div>
```

```
        </div>
    </div>
</div>
```

Quadro 19 - Exemplo de Tabs.

Para criar botões parecidos com abas, utiliza-se a função *tabs()* do *JQuery* (**Quadro 19**). Para isso, primeiro cria-se uma *div* com o *id* "*tabs"* que será o container do conteúdo das abas. Feito isso, cria-se uma lista desordenada *ul* com dois *links*. Os *links* deverão referenciar os conteúdos da páginas que serão exibidos e não URLs. Para fazer essa referencia utiliza-se o *id* dos containers de cada conteúdo como *link href*. Com o corpo da página devidamente configurado, deve-se adicionar no *script Javascript* o comando *$("#tabs").tabs()* para transformar essas *divs* em um conteúdo manipulado por abas.

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

Não se pode afirmar que os *frameworks* e *plugins* analisados no presente trabalho são as melhores escolhas para desenvolver aplicações rápidas e robustas, mas é fato que, auxiliam e muito na produção de aplicações *web*.

A experiência do estudo de caso possibilitou uma maior compreensão do desenvolvimento de aplicações utilizando o *framework* ASP.NET MVC e a biblioteca *JQuery.* Percebe-se a importância de utilizar o *JQuery* para manipular o conteúdo das páginas de uma maneira mais fácil e interativa, e como a biblioteca torna simples o código *Javascript*.

O desenvolvimento em camadas do ASP.NET MVC torna muito mais fácil o entendimento da lógica do sistema, futuras manutenções e possíveis trocas de tecnologias, além de possibilitar uma maior liberdade sobre o protocolo HTTP.

Com o uso desses *frameworks* percebe-se que o desenvolvedor se preocupa menos com a lógica do sistema e mais em introduzir novas ideias.

5.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

O presente trabalho descreve as implementações de *JQuery* com a arquitetura ASP.NET MVC, trabalhando apenas nas camadas de *controller* e *view*. Mas uma arquitetura com tantas vantagens requer que seja realizado um estudo mais apropriado de tecnologias da camada de *model* como o *Entity Framework.*

REFERÊNCIAS BIBLIOGRÁFICAS

ALLANA, Sonal Aneel. **ASP.NET JQuery Cookbook.** Packt Publishing. Birmingham – Mumbai, 2011.

BAPTISTELLA, Adriano José. **Abordando a arquitetura MVC, e Desin Patterns: Observer, Composite e Strategy**. Disponível em <http://www.linhadecodigo.com.br/artigo/2367/Abordando-a-arquitetura-MVC-e-Design-Patterns-Observer-Composite-Strategy.aspx>. Acesso em 20/05/2011.

BERARDI, Nick; KATAWAZI, Al; BELLINASO, Marco.**ASP.NET MVC 1.0 Website Programming: Problem – Design – Solution**. Wiley Publishing, Inc – Indianapolis, 2009.

BIBEAULT, Bear; KATS, Yehuda. **JQuery in Action**. Manning Publications, 2008.

CASTLEDINE, Earle; SHARKIE, Craig. **JQuery: Novice to Ninja**. SitePoint Pty, Estados Unidos. Fevereiro, 2010.

CHIARETTA, Simone. **What's New In ASP.NET MVC 2**. Wiley Publishing, Inc – Indianapolis, 2010.

DEACON, John. **Model-View-Controller (MVC) Architecture**. Disponível em <http://www.jdl.co.uk/briefings/mvc.html>. Acesso em 27/05/11.

ESPOSITO, Dino. **Programming Microsoft ASP.NET MVC**. Microsoft Press – Washington, 2010.

ESPOSITO, Dino. **Programming Microsoft ASP.NET 4.** Microsoft Press – Washington, 2010.

FLANAGAN, David. **JQuery Pocket Reference**. O'Reilly Media, 2011.

GALLOWAY, Jon; HAACK, Phil; HANSELMAN, Scott; GUTHRIE, Scott; CONERY, Rob. **Professional ASP.NET MVC 2**. Wiley Publishing, Inc. – Indianapolis, 2010.

KRASNER, Glenn; POPE, Stephen. **A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System**. ParcPlace Systems, 1988.

KEITH, Jeremy. **Bulletproof Ajax**. New Riders – USA, 2007.

JQUERY. JQuery Documentation. Disponível em <http://jqueryui.com/docs/Getting_Started> Acesso em 25/05/11.

MICROSOFT. **Understanding Models, Views and Controllers (VB)**. Disponível em <http://www.asp.net/mvc/tutorials/understanding-models-views-and-controllers-vb>.Acesso em 27/05/11.

MSDN Magazine. **Criando Aplicativos Web Sem Web Forms.** Edição de Março de 2008.

PALERMO, Jeffrey; SCHEIRMAN, Ben; BOGARD, Jimmy; HEXTER, Erix; HINZE, Matthew.**APS.NET MVC 2 In Action**. Manning Publications – Stamford, 2010.

REFSNES, Hege, Stale, Kai Jim e Jan Egil; HENTHORNE, Kelly. **Learn Javascript and Ajax with w3schools**. Wiley Publishing, Inc. Indianapolis, Indiana – 2010.

RIORDAN, Rebecca M. **Head First Ajax**. O'Reilly Media, Inc. Estados Unidos, 2008.

SANDERSON, Steven. **Pro ASP.NET MVC Framework.** Apress – United States of America, 2009.

SILVA, Maurício Samy. **Introdução a JQuery**. Disponível em <http://www.linhadecodigo.com.br/artigo/2068/Introdu%C3%A7%C3%A3o-%C3%A0-jQuery.aspx>. Acesso em 07/03/2011.

TRIRAND. **JqGrid.** Disponível em <http://www.trirand.com/jqgridwiki/doku.php>. Acesso em 25/05/11.

WALTHER, Stephen. **ASP.NET MVC Framework Unleashed**. Pearson Education, 2010.