UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCIAN STURIÃO RODRIGUES

Padrões de Projeto e Recomendação em Sistemas Web

Prof. Aloísio Carlos de Pina, D.Sc.
Orientador

Rafael Souza Nader, M.Sc.
Co-orientador

Rio de Janeiro, Fevereiro de 2015

Padrões de Projeto e Recomendação em Sistemas Web

Lucian Sturião Rodrigues

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Apresentado por:

Lucian Sturião Rodrigues

Aprovado por:

Prof. Aloísio Carlos de Pina, D.Sc.

Rafael Souza Nader, M.Sc.

Profa. Valeria Menezes Bastos, D.Sc.

Prof. Luis Mariano Peñaranda, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

Fevereiro de 2015

AGRADECIMENTO

Ao orientador Aloísio Carlos de Pina por sua disposição e ajuda durante a elaboração do projeto final;

Ao co-orientador Rafael Souza Nader por seu conhecimento, incentivo e dicas durante o desenvolvimento do projeto final;

Aos alunos da turma de Ciência da Computação 2010.2 da UFRJ, que ajudaram muito nos estudos para as diversas matérias do curso;

À família: Marcelo dos Santos Rodrigues, Nilma A Sturião Rodrigues, Patrick Sturião Rodrigues, cunhada Vanessa Simões Ferreira e minha namorada Isabele Gouvêa da Silva por todo apoio durante a graduação;

Aos colegas/amigos de trabalho da Stone Age por terem ajudado a desenvolver o conhecimento necessário para desenvolver sistemas Web e todas suas especificidades;

A Bruno Mattos e Luiz Felipe Perícolo por cederem o código base de sua monografia para consulta. A Bruna Xavier por ter feito uma base LaTeX tão intuitiva de ser modificada e completa, já com os padrões adotados pelo DCC [52].

- Lucian Sturião Rodrigues

Fevereiro/2015

RESUMO

Padrões de Projeto e Recomendação em Sistemas Web

Lucian Sturião Rodrigues

Fevereiro/2015

Orientador: Aloísio Carlos de Pina, D.Sc.

Este trabalho tem como objetivo documentar os melhores padrões de projeto para um sistema web. Será explicado o lado cliente e lado servidor da aplicação. Será também construído um sistema de recomendação e esclarecido o porque da sua crescente necessidade em sistemas online que dependem de sugestão de conteúdo. Serão analisados desde os métodos de desenvolvimento como também tecnologias e frameworks utilizados. Na concepção do sistema foram utilizados: .NET WebAPI 2.0, padrão de desenvolvimento MVC, AngularJS 1.3, Typescript, HTML5 e CSS3. Pretende-se construir um sistema de recomendação misto, usando as modalidades de base em conteúdo e filtragem colaborativa.

Palavras-chave: Desenvolvimento Web, Padrões de projeto, Sistemas de recomendação, Filtragem colaborativa.

ABSTRACT

Padrões de Projeto e Recomendação em Sistemas Web

Lucian Sturião Rodrigues

Fevereiro/2015

Advisor: Aloísio Carlos de Pina, D.Sc.

This work aims to document the best design patterns for a web system. Both client-side and server-side will be explained. A recommender system will be developed and why it is so important these days in online systems that depend on content suggestion will be elucidated. Here the development methods will be analyzed as well as technologies and frameworks used. In the system architecture were used: .NET WebAPI 2.0, MVC design pattern, 1.3 AngularJS, Typescript, HTML5 and CSS3. We intend to build a mixed recommender system, using content-based and collaborative filtering techniques.

Palavras-chave: *Desenvolvimento Web, Padrões de projeto, Sistemas de recomendação, Filtragem colaborativa.*

Keywords: *Web development, Design Patterns, Recommender System, Collaborative Filtering.*

Lista de Figuras

Lista de Códigos

Lista de Abreviaturas e Siglas

CSS	Cascading Style Sheets
DI	Dependency Injection
DOM	Document Object Model
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IOC	Inversion of Control
MVC	Model View Controller
MVP	Model View Presenter
MVVM	Model View ViewModel
MVW - MV*	Model View Whatever
SVN	Subversion
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience

Sumário

Capítulo 1

Introdução

Ao se desenvolver um sistema qualquer, muitos fatores devem ser pensados. As tecnologias utilizadas para se conseguir construir toda base necessária para o mesmo são muitas. Todas têm suas vantagens e desvantagens, que devem ser analisadas cuidadosamente.

Um framework que vem se destacando atualmente é o AngularJS [17]. Já é altamente adotado pela comunidade web e mobile para ser utilizado na construção do *client-side* ("lado do cliente", geralmente faz referência às linguagens e/ou frameworks que executam no navegador que acessa um sistema web) do sistema. O AngularJS é utilizado para construir HTML dinamicamente, permitindo interações e construções de conteúdo não possíveis sem o seu uso.

Vale a pena também citar o uso das tecnologias .NET no *server-side*. O Visual Studio [35] é uma IDE muito completa; auxilia desde a construção de código HTML até o acesso a banco de dados, com *plugins* e *patterns* de abstração, o que aumenta muito a produtividade. É considerada uma das melhores disponíveis no mercado[18]. As recentes políticas da Microsoft de disponibilizar variantes totalmente gratuitas do Visual Studio para uso comercial, e o fato de suas próximas versões serem de código aberto, foram fatores os quais motivaram a escolha deste para desenvolvimento do projeto.

1.1 Motivação e objetivos

A grande motivação do trabalho é, além de construir um sistema para ser utilizado por universitários, criar documentação que possa ser usada como referência de um projeto/sistema web por outros desenvolvedores. O objetivo basicamente é produzir algo com o conhecimento adquirido durante a faculdade de modo a ajudar as pessoas que ingressarão na mesma posteriormente. Pretende-se abordar tecnolo-

gias utilizadas no desenvolvimento, citar melhores práticas e detalhar os melhores padrões de design de um sistema web.

Alunos e universitários em sua grande maioria precisarão de um estágio ou iniciação científica para conclusão do curso, estes serão os maiores beneficiados. Além disto, pensando no tipo de implementação do projeto, ao se elaborar e desenvolver um sistema deve-se ter atenção a muitos itens de diversas áreas – desde qual dispositivo o usuário final vai usar, qual servidor vai hospedar a aplicação, quais linguagens utilizar, quais frameworks, quais tecnologias. Preocupações as quais valem a pena ser aprendidas e experimentadas pelo menos uma vez por um aluno de ciência da computação.

Arcaísmo dos sistemas de divulgação atuais

Uma curiosidade a ser citada é o atual estado dos sistemas de divulgação de vagas (iniciação científica, estágio, etc) das universidades. A maior parte da divulgação é feita por redes sociais, e-mail ou anúncios espalhados em murais pelos corredores dos departamentos. Este último é uma técnica muito importante em tempos onde não se existia a internet, mas agora já se torna uma prática arcaica e nem um pouco efetiva.

Não existe um sistema central onde ofertantes possam cadastrar suas vagas com pouca ou nenhuma burocracia. Grande esforço é necessário para que alunos saibam da existência de uma determinada oportunidade. O contrário também é verdade: quando um aluno está à procura de uma vaga deve-se procurar em diversos meios de comunicação. No final das contas a divulgação é feita ou por boca a boca ou através de redes sociais, como o grupo do Facebook [13] da turma/curso.

1.2 Metas

Sendo assim, foi feita a escolha do tema do sistema: um sistema de gerenciamento de vagas para universitários. Mas já exitem alguns, não especificamente com esse objetivo, então são necessários alguns diferenciais:

- Que o sistema seja intuitivo e simples.
- Que haja pouca ou quase nenhuma burocracia, assim como os murais nos corredores da faculdade.
- Que seja um sistema rápido e responsivo[8], para que de qualquer lugar o usuário possa fazer acesso – seja para procurar ou inserir uma nova vaga.

Somado ao fato de ser um sistema de gerenciamento de vagas, o projeto é um sistema de recomendação; para que ao se procurar as vagas estas sejam ordenadas de acordo com a preferência do usuário. A recomendação foi feita utilizando como base um algoritmo de filtragem colaborativa com cálculo de semelhança usando o índice/coeficiente de Jaccard. Usuários têm a possibilidade de dizer se gostaram ou não de um anúncio de vaga e isto é suficiente para construir o sistema de recomendação.

Um segundo sistema de recomendação ordena os anúncios que "empataram" na primeira leva de recomendação. Este segundo sistema é bem simples e faz a comparação entre requisitos que o usuário atende da vaga, distância até sua residência e salário anunciado. O sistema está disponível online para uso de qualquer um que desejar. Não é necessário o registro do usuário para serem vistas as vagas, somente para o cadastro de novas vagas e recomendação.

1.3 Composição do texto

Na construção da monografia serão abordados:

- As tecnologias estudadas previamente, citando sucintamente suas vantagens e desvantagens;
- Tecnologias utilizadas na parte cliente da aplicação (tudo que executa no navegador) – erros e acertos;
- Tecnologias utilizadas na parte servidor da aplicação, sua estrutura, testes e organização de código;

- Recente demanda por experiência de uso em sistemas Web e possíveis soluções e melhorias;
- Pesquisa de melhores práticas e padrões de projetos a ser utilizados neste tipo de sistema;
- Tipos de recomendação mais utilizados em computação, dificuldades encontradas na aplicação e recomendação utilizada no sistema.

Capítulo 2

Tecnologias estudadas

Este capítulo irá apresentar todas as tecnologias estudadas antes da escolha do que se usar na implementação final do sistema. Foram usadas .NET Razor View Engine, jQuery, a biblioteca KnockoutJS e o template .NET MVC 5 – posteriormente substituídos por AngularJS e o template .NET Web API 2.0. Serão comentadas suas vantagens e desvantagens e motivos pelos quais foram abandonados ou não escolhidos.

2.1 ASP.NET Razor View Engine

Razor[15] na verdade não é uma linguagem que executa no navegador, mas ajuda a construir HTML, que é exibido no lado cliente. Em conjunto com o template MVC[40] da Microsoft o Razor é interpretado no lado servidor onde executa suas operações e constrói o HTML que será retornado para o cliente, quando requisitado.

Código 2.1: Exemplo de código utilizando .NET Razor View Engine [15]

```
<!-- Guardando uma String -->
@{ var welcomeMessage = "Bem vindos, novos membros"; }
<p>@welcomeMessage</p>

 <!-- Guardando uma Data -->
@{ var year = DateTime.Now.Year; }

 <!-- Exibindo uma variavel -->
<p>Boas vindas aos nossos novos membros de @year!</p>
```

Para falar de Razor é necessária uma breve explicação sobre o Microsoft .NET MVC. Este é um template de auxílio de organização de código feito pela Microsoft.

Disponibiliza desde roteamento de páginas (provendo organização entre requisições de URL e páginas resposta), abstração para filtros de autenticação (controle de acesso a funções do servidor e páginas do aplicativo), entre outros.

O Razor funciona sendo interpretado sempre que uma página é requisitada. O .NET MVC controla quais páginas são retornadas ao se acessar determinada rota. Antes de toda página ser retornada ao navegador verifica-se a existência de código Razor: se presente, este código será interpretado e transformado em HTML que um navegador padrão consegue visualizar. O início do projeto foi construído com a ajuda de Razor.

Razor é muito útil quando se pretende construir páginas com pouca interação de usuário ou com pouca interatividade/dinâmica nas páginas. Com sua funcionalidade é possível preencher um HTML com o retorno de alguma função existente no servidor. É possível associar dados de formulários a variáveis do servidor, auxiliando a construção de páginas onde usuários irão fazer algum cadastro, por exemplo. Permite a divisão de uma página em diversas "sub páginas", chamadas *partials*, melhorando bastante a organização do HTML utilizado no aplicativo [24].

Mas sua facilidade termina aí. Não é possível fazer nenhuma interação com o código Razor sem que uma página seja re-submetida, porque seu código é interpretado exclusivamente no lado servidor. Então qualquer que seja a mudança desejada, seja mudar um título da página, mudar o número de itens em um menu, é necessário executar um *refresh* no navegador. Quando se deseja interatividade na página – por exemplo habilitar edição de algum campo somente depois de alguma ação ser executada - JavaScript[28] começa a se tornar necessário.

2.2 JavaScript e jQuery

Quando se começa a ter necessidade de qualquer interatividade, como explicado anteriormente, a primeira e lógica escolha é se utilizar JavaScript e jQuery[23]. jQuery é um biblioteca JavaScript rápida e rica em qualidades. Muito completa, permite manipulação de HTML (DOM), interceptação de eventos, cria possibilidade

de uso de animações, efetua chamadas Ajax[16] – Asynchronous Javascript And XML, geralmente usadas para comunicação cliente-servidor - além de ser suportada pela maioria dos browsers.

Com jQuery ou JavaScript pode-se, por meio da manipulação HTML, verificar estados e atribuir eventos a *inputs*, botões e qualquer elemento DOM de nossa página Web. Com esses estados e eventos consegue-se criar interatividade na aplicação que se está desenvolvendo. Com seu uso é possível, por exemplo: mostrar um campo somente quando X campos anteriores forem preenchidos, modificar a funcionalidade de um botão dependendo do que for escrito num formulário.

JavaScript puro ou jQuery podem servir bem, porém não fornecem nenhum auxílio para organização estrutural de um projeto. Por ser possível fazer "qualquer coisa", ser uma linguagem script que se colocada em qualquer lugar do HTML é interpretada, sem um estudo maior, o código começa a se tornar uma dor de cabeça para o desenvolvedor. Existem padrões de projeto a serem seguidos, porém fazer tudo "na mão" sem a explícita necessidade dessa organização para que as coisas funcionem não incentiva ninguém. Funções utilizadas em diferentes páginas começam a se concentrar num único arquivo, o que não ajuda em nada na manutenção. Algum padrão precisa ser rapidamente adotado para que tudo faça sentido. E por isso sua utilização *crua* não foi continuada.

2.3 KnockoutJS

Não foi diretamente utilizado no projeto, mas foi estudado em projetos paralelos durante a implementação inicial do sistema. KnockoutJS é uma biblioteca JavaScript que auxilia a implementação do padrão de projeto MVVM (Model-View-View-Model), que será explicado no próximo capítulo.

Ao se trabalhar com KnockoutJS[26] e outras bibliotecas/frameworks como EmberJS[22], Backbone[7], CanJS[9], AngularJS[17] e outros, ganha-se uma propriedade que é uma das mais importantes e que permite a melhor estruturação e organização do código no *client-side*: *Two-Way-Binding*.

Two-Way-Binding é uma expressão que define a característica de algum elemento na *view* – ou interface – estar associado a um modelo de uma *view-model* e o modelo da *view-model* estar associado, reciprocamente, ao mesmo elemento. Sendo assim, sempre que o elemento muda seu valor na *view*, o modelo é automaticamente atualizado. E sempre que o modelo tem seu valor atualizado, o elemento da *view* tem seu valor atualizado.

Isso torna possível que o código JavaScript não precise referenciar nenhum elemento da *view*, ou seja, o código construído não depende mais de como a interface com o usuário será implementada. O HTML por sua vez irá referenciar todas propriedades disponibilizadas pela *view-model* utilizando marcação própria do KnockoutJS.

Um exemplo de *view-model* (código que fica disponivel para utilização no HTML) é exibido em 2.2. Repare como nenhum elemento DOM é referenciado.

Código 2.2: Exemplo de código KnockoutJS na View-Model [26]

```
var SimpleListModel = function(items) {
    this.items = ko.observableArray(items);
    this.items = ko.observable("");
    this.addItem = function () {
        if(this.itemToAdd() != "") {
            this.items.push(this.itemToAdd());
            this.itemToAdd("");
        }
    }.bind(this);
};
ko.applyBindinds(new SimpleListModel([
    "Alpha",
    "Beta",
    "Gama"
]))
```

Um exemplo da sintaxe de utilização do KnockoutJS no HTML pode ser visto em 2.3.

Código 2.3: Exemplo de código KnockoutJS na View [26]

```
<form data-bind="submit: addItem">
    Novo Item:
    <input data-bind="value: itemToAdd,
        valueUpdate: 'afterKeyDown'"/>
    <button type="submit"
        data-bind="enable: itemToAdd().length > 0">
        Adicionar
        </button>
    <p> Seus itens: </p>
    <select multiple="multiple"
        data-bind="options: items"></select>
</form>
```

Um código não referenciar elementos DOM é uma grande conquista. Permite a implementação declarativa do aplicativo.

> "Uma implementação declarativa não trata o HTML ou JavaScript como um bug que precisa ser consertado. Em vez disso, funcionalidades são inseridas de uma forma tão natural que você nem vai acreditar no porque não pensou nisso antes."
>
> AngularJS for jQuery developers [41]

Grandes façanhas alcançadas apresentadas por "somente" uma biblioteca. O principal motivo por não se ter escolhido o KnockoutJS na implementação final foi a falta de um sistema de roteamento. Viu-se que seria desejável um sistema similar ao implementado pelo AngularJS: que somente parte da página seja carregada ao mudar

de rota na navegação do site. Garante grande fluidez na utilização da aplicação e também velocidade nas mudanças de página.

2.4 ASP.NET MVC 5

Foi a tecnologia usada no início do desenvolvimento do projeto, fez parte dos primeiros protótipos do mesmo. Foi escolhida no início por ser muito fácil começar a fazer um site utilizando a mesma. Este template da Microsoft já era conhecido pois já fora utilizado em outros projetos paralelos - fora da faculdade, no estágio, e durante a faculdade em algumas disciplinas.

O motivo de ser abandonado, apesar de todas facilidades, foi por obrigar o desenvolvedor a criar uma Action correspondente para toda página (.CSHTML, que é o correspondente a um .HTML quando se usa o template) que se deseja exibir. Somente retornada de uma Action (método que retorna uma *ActionView* dentro de um controller) uma página pode ser exibida. E, para ser retornada, o .CSHTML correspondente deve estar dentro de uma organização específica de pastas.

Por exemplo, se fosse necessário exibir a página *http://www.urldosite.com.br/home* os seguintes passos deveriam ser seguidos: criar um controller chamado *HomeController*, criar uma *Action* chamada obrigatoriamente *Index* que retornasse uma *ActionView* chamada *Index*, criar um arquivo *Index.cshtml* que esteja dentro da pasta */Views/Home*. É bom pelo motivo de se estabelecer um padrão a ser seguido, mas quando se usa AngularJS constata-se[43] que existem outros padrões de organização de arquivos mais adequados a este tipo de arquitetura (MV*[38]).

Além disso a obrigatoriedade de criar Actions para retorno de .HTML acaba poluindo os controllers, que também tem a função de fazer comunicação com o client-side via chamadas Ajax [16], e com isso o código fica cada vez mais desorganizado (conforme o projeto cresce). Por isso se optou utilizar o template .NET Web API 2.0, possibilitando que os *controllers* tenham a única responsabilidade - fazer a comunicação de *client-side* com o *server-side*, e permitindo o acesso de arquivos .HTML simplesmente acessando a URL do mesmo.

Capítulo 3

Tecnologias usadas no Client-Side

> "As linguagens do *Client-Side* são as que permitem a interação em uma página web. O código requerido para processar entradas do usuário é baixado e interpretado pelo navegador ou algum plugin. Um exemplo de interação *Client-Side* é a mudança de algum elemento da página quando se passa o mouse por cima do mesmo. JavaScript está incluída nas linguagens de script *Client-Side*."
>
> Definição de Client-Side – The Motive Web Design Glossary [31]

Este capítulo irá apresentar todas as tecnologias estudadas na implementação do lado cliente do sistema, lições aprendidas e padrão adotado na estrutura do mesmo.

3.1 AngularJS

AngularJS é um framework JavaScript. Foi desenvolvido em 2009 por Misko Hevery [19], funcionário da Google. Suas versões iniciais foram usadas internamente e em outubro de 2010 foi lançada sua primeira versão pública estável (0.9.0[37]). Para ser incorporado em uma página web somente é necessário que seu código seja importado por meio de uma *tag script*.

Ajuda a implementar uma arquitetura MVVM (que pode ser interpretada tanto como MVVM quanto MVC ou até MVP [38]) no *Client-Side*. Sua grande proposta é criar interfaces dinâmicas estendendo a funcionalidade do HTML, que foi originalmente criado para mostrar páginas estáticas [17]. O framework AngularJS disponibiliza novas *tags* - chamadas diretivas - que podem ser usadas no HTML.

Cada diretiva disponibilizada implementa uma nova função. Também é possível criar diretivas conforme a necessidade. Com isso é possível desenvolver verdadeiras aplicações web, mudando inteiramente o paradigma de utilização de uma página web, provendo interação do nível de aplicações desktop.

Framework que conquista uma experiência de uso incomum para aplicações web, também provém arquitetura baseada em injeção de dependência [3]. Uma grande melhora que o AngularJS traz para o desenvolvedor é a melhoria na qualidade de código gerado. Por sua estrutura ser por natureza baseada em injeção de dependência, o código é mais facilmente escrito de uma forma com que seja testável, que seja extensível, que possa ser manutenido. Este assunto será apresentado mais tarde no texto.

Recomenda-se utilizar como base o guia de estilos produzido por John Papa em conjunto com a equipe de desenvolvimento do AngularJS encontrado em [43].

3.1.1 Desenvolvimento

Para usar AngularJS em um projeto deve-se entender alguns conceitos básicos do mesmo: Diretivas, Módulo, Views, Controllers, Serviços, e Escopo.

Diretivas são marcadores em elementos DOM – podem ser atributos, nomes de elemento, comentários ou classes CSS – que dizem ao compilador HTML do AngularJS (*$compiler*) quais comportamentos especiais adicionar àquele elemento, ou até mesmo transformá-lo juntamente com seus elementos filhos [1]. Alguns exemplos de diretiva são: *ng-app*, *ng-controller*, *ng-model*, *ng-repeat*, etc. É possível criar diretivas customizadas.

Módulo é um *container* onde sua aplicação será construída. É onde todas suas configurações de aplicativo, controllers, serviços, diretivas, entre outros, serão armazenados. O código JavaScript em 3.1 mostra como definir um novo módulo usando AngularJS.

Código 3.1: Definição de novo módulo simples

```
angular.module('MeuModulo', []);
```

Caso o módulo tenha alguma dependência de outro módulo, na sua criação deve-se fazer a declaração 3.2.

Código 3.2: Definição de novo módulo com dependência

```
angular.module('MeuModulo', ['moduloDependencia1']);
```

No código em 3.2 explicita-se que o módulo 'MeuModulo' depende do módulo 'moduloDependencia1', então todas as funcionalidades implementadas por este agora estão disponívels para uso naquele. Geralmente se faz isto para utilizar diretivas criadas por outrem.

Após definir o módulo em seu código JavaScript é necessário declarar no HTML que o documento depende de um módulo, informando seu nome. Isso é feito utilizando a diretiva *ng-app*, geralmente utilizada na tag *<html>*, como visualizado em 3.3.

Código 3.3: Utilização de módulo no HTML

```
<!DOCTYPE html>
<html ng-app="MeuModulo">
    <body>
        <script src="angular.min.js"></script>
        <script>
            angular.module('MeuModulo', []);
        </script>
    </body>
</html>
```

Controller é uma função construtora responsável por disponibilizar funcionalidades ao **escopo** do AngularJS (*$scope*). Quando um controller é anexado a um elemento DOM via diretiva ng-controller, um novo objeto de controller é instanciado utilizando a função especificada.

No controller o estado inicial do $scope é configurado. Para isso propriedades são anexadas a este. Estas propriedades contém o *ViewModel* – modelo que será

representado pela *view*. Todas as propriedades do **escopo** serão disponibilizadas à interface a partir do elemento DOM onde o controller é registrado [6].

O código em 3.4 exemplifica a criação de um controller, registro do mesmo em uma * * e utlização do $scope para mostrar informação na interface.

Código 3.4: Definição e utilização de um controller

```
<!DOCTYPE html>
<html ng-app="MeuModulo">
    <body>
        <div ng-controller="MeuController">
            <input type="text" ng-model="nome">
        </div>
        <script src="angular.min.js"></script>
        <script>
            angular
                .module('MeuModulo' ,[])
                .controller('MeuController',
                    ['$scope', function ($scope) {
                        $scope.nome = 'Lucian';
                    }]
                )
        </script>
    </body>
</html>
```

O resultado de 3.4 é uma página contendo um input o qual está associado à propriedade “nome”, seu valor inicial foi definido como “Lucian”.

Serviços são objetos que têm objetivo de organizar e compartilhar código em seu aplicativo. Serviços são *singletons* [21], todos componentes que são dependentes de um serviço recebem uma referência a única instancia que foi gerada pela *fábrica de serviços* [2]. Sua construção é muito parecida com a de um controller. Para ser

utilizado é necessário explicitar sua dependência. Em 3.5 segue exemplo de definição de um Serviço e de sua utilização em um *controller*.

Código 3.5: Definição de serviço

```
angular
    .module('MeuModulo', [])
    .service('MeuServico'
                [function () {
                    this.MinhaFuncao = function () {
                        alert('teste');
                    }
                }]
            );
```

Em 3.5 foi definido um serviço de nome "MeuServico" que contém uma função chamada "MinhaFuncao". Esta poderá ser reutilizada em qualquer componente que depender deste serviço.

Código 3.6: Utilização de serviço em controller

```
1 angular
2     .module('MeuModulo')
3     .service('MeuController', ['$scope', 'MeuServico',
4                 function ($scope, MeuServico) {
5                     $scope.MinhaOutraFuncao = function () {
6                         MeuServico.MinhaFuncao();
7                     }
8                 }
9             ]);
```

No código 3.6 foi criado um controller que declara ser dependente dos serviços "$scope" e "MeuServico". Foi criada um função chamada "MinhaOutraFuncao" que encapsula a chamada à função "MinhaFuncao" do serviço criado. Sempre que a função "MinhaOutraFuncao" for chamada, um *alert* com o texto "teste" será mostrado

na tela – assim como definido no serviço.

Repare que na *linha 2* de 3.6 a chamada ao módulo "MeuModulo" é diferente da utilizada em 3.5. Isso ocorre porque em 3.5 o módulo é definido (método *set*) e em 3.6 o módulo somente é referenciado (método *get*) para ser criado o controller.

3.1.2 Injeção de dependência

> "Injeção de dependência (DI) é um padrão de projeto de software que trata como os componentes de uma aplicação declaram e obtêm suas dependências. O subsistema $injector do AngularJS é responsável pela criação de componentes, resolvendo suas dependências, e os provendo a outros componentes, conforme solicitado."
>
> Injeção de Dependência - Guia de desenvolvedor AngularJS [3]

Como pode-se observar em todas definições de componentes acima, injeção de dependência é um padrão adotado pela arquitetura na qual o AngularJS foi construído. Pode-se usá-la na definição de qualquer tipo de componente, e até mesmo em blocos de configuração e de inicialização do módulo [3].

Injeção de dependência é uma forma de implementação, ou um subconjunto, de um outro padrão de projeto de software chamado Inversão de Controle [27]. Existem formas diferentes de implementação da DI: injeção por construtor (objeto dependente recebe suas dependencias pelo construtor da classe), injeção por método *set* (objeto dependente recebe suas dependencias através de métodos), injeção por implementação de interface (objeto dependente implementa uma interface que o obriga a criar métodos *set* de forma específica) [47]. O AngularJS utiliza a implementação por construtor.

As vantagens notáveis na utilização de DI são:

- Desacoplamento de código - o objeto dependente não precisa mais saber como é a implementação daquilo que precisa usar. Isola o dependente de impactos em casos de refatoração de código [29];

- Princípio da responsabilidade única – com o desacoplamento de código pode-se definir estritamente a função de cada componente. Produz-se então reusabilidade, testabilidade e manutenibilidade de código;
- Redução de código de "setup" (boilerplate code [32]) – código que é repetido em todo lugar necessário para utilizar objetos os quais um componente é dependente. Essa redução se dá porque um componente provedor será o responsável por fazer a "entrega" das dependências [29]. No caso do AngularJS, quem tem essa responsabilidade é o *$injector*.
- Permite desenvolvimento paralelo de código. Mais de um desenvolvedor pode desenvolver duas *classes* que dependem mutuamente, somente sendo necessário que sejam conhecidas quais interfaces estas irão implementar.

3.1.3 MVVM

MVVM é um padrão de projeto de software derivado do MVC [34] – conhecido Model-View-Controller, geralmente utilizado em aplicações desktop. Sigla de Model-View-ViewModel. Quando se fala de AngularJS não existe um consenso se o framework ajuda a implementar MVVM, MVP ou alguma variação destes. Então quando for dito aqui ViewModel, interprete como quiser [38].

Quando se trabalha com MVVM o **ViewModel** é o principal constituinte. É quem carrega a lógica de apresentação, e quem possui maior parte do código no *client-side* [44]. O ViewModel se encarrega de atualizar a View com as informações vindas do modelo, quem tem essa função no AngularJS é o *$scope*.

Por meio de *Two-Way-Databiding* todas mudanças feitas na View são refletidas no ViewModel, e qualquer mudança no ViewModel é propagada na View. Então não é preciso se preocupar se as informações contidas no ViewModel estão atualizadas, porque isso sempre é verdade. O *$scope* pode se preocupar com sua real função: servir de fonte para a View, seja fonte de informação (tratando o modelo e manipulando-o para disponibilizar dado que a View sabe consumir) ou fonte de ações (disponibilizando quaisquer métodos necessários para a comunicação interface/servidor ou outros essenciais para o funcionamento do aplicativo) [42].

View, quando se utiliza Angular, é o próprio HTML. Através das diretivas pode-se consumir os métodos e propriedades contidos no *$scope* e, idealmente, toda lógica de visualização pode ficar na View. Quando alguma manipulação de elementos DOM não pode ser feita no próprio HTML, é recomendável que seja criada uma *diretiva customizada* para que isso seja feito.

Model é toda informação vinda do *server-side* da aplicação. O responsável por prover esses dados deve ser um Serviço Angular, nunca um Controller. Controllers somente devem conseguir acessar os dados do modelo declarando ser dependentes de algum Serviço Angular que forneça essa informação através de funções.

Um ponto a ser notado é que o *modelo* tem somente o papel de carregar informação, não modificá-la ou influenciar como o dado será visualizado. Formatar a informação é considerado lógica de negócio, então deve ser encapsulado pelo View-Model [42].

3.1.4 Single Page Applications

O principal objetivo ao se utilizar AngularJS em um projeto é a possibilidade de facilmente ser implementada uma aplicação de página única (também conhecida como SPA).

SPA é uma aplicação construída em somente uma página com o objetivo de ter uma experiência de uso diferenciada, similar a uma aplicação desktop. Mudanças de páginas em SPAs são feitas sem o conhecido – e indesejado – "reload" no navegador. Em uma SPA toda informação de troca de página é carregada dinâmicamente [10].

A criação de SPAs é possível graças a diretiva *ng-view*. Esta diretiva complementa o serviço *$route* incluindo o template renderizado da atual rota no layout principal, no local onde foi utilizada a diretiva. Sempre que a rota muda, a view inclusa muda de acordo com a configuração feita no serviço *$route* [4].

Segue exemplo de configuração do serviço *$route* em 3.7.

Código 3.7: Configuração de rotas em um módulo

```
angular.module('MeuModulo').config(['$routeProvider',
    function($routeProvider) {
        $routeProvider.
            when('/', {
                templateUrl: 'App_Angular/Home/Home.html',
                controller: 'HomeController'
            }).
            when('/Rota1', {
                templateUrl: 'App_Angular/Rotas/Rota1.html',
                controller: 'Rota1Controller'
            }).
            when('/404', {
                templateUrl: 'App_Angular/Shared/404.html'
            }).
            otherwise({
                redirectTo: '/404'
            });
}]);
```

As configurações acima declaram que:

- A rota inicial "/" exibirá o template encontrado em "App_Angular/Home/Home.html" que será controlado pelo *controller* "HomeController";
- Existe uma rota chamada "/Rota1" que exibirá o template encontrado em "App_Angular/Rotas/Rota1.html" o qual será controlado pelo *controller* "Rota1Controller";
- Existe uma rota chamada "/404" que exibirá o template encontrado em "App_Angular/Shared/404.html";
- Caso alguma rota não existente nesta definição seja requisitada, que o usuário seja redirecionado para a rota "/404".

Exemplo da utilização da diretiva ng-view em 3.8.

Código 3.8: Utilização da diretiva ng-view

```
<!DOCTYPE html>
<html ng-app="MeuModulo">
    <body>
        <ng-view></ng-view>
        <script src="angular.min.js"></script>
        <script src="MeuModulo-App.js"></script>
    </body>
</html>
```

3.2 TypeScript

TypeScript é um *superset* tipado de JavaScript. Compila para JavaScript, logo funciona em qualquer browser ou ambiente que dê suporte a este [49].

O Maior ponto positivo da sua utilização (em conjunto com o Visual Studio) é que ele é compilado. Consequentemente, sempre que se dá *build* em um projeto que faça uso de TypeScript, seus arquivos são compilados e convertidos para JavaScript, e qualquer erro de sintaxe que exista impede o projeto de ser executado. Pode parecer uma justificativa insignificante, mas em um projeto com grande quantidade de arquivos JavaScript (.js) ou TypeScript (.ts) isso faz grande diferença, melhorando significantemente a escalabilidade do mesmo.

Além de apontar erros de sintaxe, o TypeScript permite a criação de *classes, módulos, herança de classes, tipagem de variáveis* (funcionalidades que estão sendo implementadas no ECMAScript 6 [20]), fornece *autocomplete*, possibilita *navegação* de um arquivo a outro (de onde se está utilizando uma classe/função até sua definição) somente com um pressionar de tecla, entre muitas outras melhorias. Esses tipos de utilidades podem economizar muitas horas de um desenvolvedor. Não existe ponto negativo em sua utilização, somente vantagens (aumento de produtividade, por exemplo).

Seu uso foi feito em todo o projeto (todos os arquivos JavaScript existentes são compilados de arquivos TypeScript, exceto arquivos de terceiros). A funcionalidade mais explorada foi a de criação de Serviços Angular por meio de classes TypeScript. O motivo é que a única interação que se dá entre os arquivos .ts quando se utiliza Angular é o de um Controller depender de algum Serviço. Somente nessa hora um arquivo utiliza uma função vinda de outro arquivo. Apesar de ser um único momento, é um momento muito frequente. Ter *autocomplete* das funções que se tem disponibilizadas no Serviço economiza bastante tempo de trabalho ao longo do período em que se desenvolve o projeto.

Em 3.9, segue o exemplo de um Serviço Angular sendo definido por meio de uma classe TypeScript.

Código 3.9: Definição de serviço utilizando TypeScript

```
class MeuServico {

    private $q: ng.IQService;
    private $http: ng.IHttpService;

    public static $inject = ['$http', '$q'];
    constructor($http, $q) {
        this.$q = $q;
        this.$http = $http;
    }

    public GetProdutosExemplo() {
        var deferred = this.$q.defer();

        this.$http.get('/api/Example/GetProdutosExemplo')
            .success(function (data) {
                deferred.resolve(data);
            }).error(function (error) {
                deferred.reject(error);
            });

        return deferred.promise;
    }
}
angular
    .module('MeuModulo')
    .service('MeuServico', MeuServico);
```

Vale mencionar que existem as definições prontas de interface de diversas bibliotecas e frameworks populares, e é simples a sua utilização. Como observado acima,

foi incluído no projeto a interface dos serviços e funções AngularJS (ng.*).

Após a definição de um serviço utilizando uma classe TypeScript, a única alteração na definição de um controller que depende deste é a declaração do tipo do mesmo. Então o exemplo de definição de controller (3.6) somente seria alterado para o código exibido em 3.10.

Código 3.10: Utilização de serviço Typescript em controller

```
angular
    .module('MeuModulo')
    .service('MeuController', ['$scope', 'MeuServico',
                function ($scope, MeuServico: MeuServico) {

                }
            ]);
```

Ao receber a referência de "MeuServico", declara-se que este é do tipo "MeuServico" inserindo ": MeuServico". Somente com isso já se obtém todas as vantagens do TypeScript supracitadas.

3.3 Twitter Bootstrap

> "Existem inúmeros tamanhos de tela em diferentes celulares, phablets, tablets, computadores, consoles de videogame, TVs e até mesmo telas incorporadas em acessórios pessoais. Os tamanhos das telas estão em constante evolução, por isso é importante que seu site possa se adaptar a qualquer tamanho disponível hoje e no futuro."
>
> Princípios básicos de Web design responsivo [30]

Bootstrap [8] é considerado o framework HTML/CSS/JavaScript para criação de front-end de aplicações web mais utilizado. Possui templates baseados em HTML e CSS prontos para utilização. Tem tipografia, botões, widgets, layout de menu

entre muitas outras extensões. Inicialmente foi criado para servir como padrão de desenvolvimento dentro do Twitter. Sua grande vantagem é o diferencial de responsividade que proporciona.

Promete prover, com uso de seu sistema de *grids*, um design funcional tanto para desktops quanto para tablets, smartphones e telas de todos tamanhos – utilizando uma única página HTML para todos dispositivos. Esse sistema implementado é chamado layout Responsivo. O objetivo ao se utilizar esse tipo de técnica é fazer com que todo tipo de usuário tenha uma experiência agradável, independente de como seja feito o acesso ao site/aplicativo web [30].

Consegue ser descomplicado a ponto de quem não conhece muito CSS conseguir criar um site usando e adaptando os templates e documentação fornecidos. Tudo é feito por meio de classes CSS que podem ser inseridos em elementos DOM. Estas classes dividem a página em doze colunas que podem ser preenchidas com os elementos que se quer visualizar. Existem classes que atuam em tamanho de tela específico, então pode-se customizar para que determinados elementos ocupem, por exemplo, metade da página em um tamanho de tela grande e a página inteira em um tamanho de tela pequeno.

Além de ser simples para ser implementado integralmente pelo desenvolvedor, existem diversos temas gratuitos e pagos na internet que usam Bootstrap como base. Foi adotado um desses temas no design do site.

Capítulo 4

Tecnologias usadas no Server-Side

Linguagens *Server-Side* são aquelas executadas onde o site está sendo hospedado: o servidor web. Tem a função de aguardar solicitações de informações ou ações, e executá-las. Essas solicitações são geralmente usadas para construir sites interativos, que fazem troca de informações com um banco de dados central. A grande serventia das linguagens *Server-Side* é que o usuário não pode visualizar sua codificação, logo pode-se fazer customizações em suas respostas dados requisitos de usuário, direito de acesso, entre outros.

Como já citado anteriormente, algumas funções, por segurança, só poderiam ser responsabilidade de execução do próprio servidor. Este capitulo irá apresentar todas funcionalidades estudadas na implementação do lado servidor do sistema, lições aprendidas e padrão adotado na estrutura do mesmo. Será abordado comunicação com o banco de dados, interface de comunicação com o navegador e alguns frameworks adotados.

4.1 Entity Framework Code First e Migrations

Para falar de code first deve-se dar um breve explicação sobre o que é Entity Framework:

> "O Microsoft® ADO.NET Entity Framework é um framework do tipo ORM (Object/Relational Mapping) que permite aos desenvolvedores trabalhar com dados relacionais como objetos de domínio específico, eliminando a necessidade de maior parte dos códigos de acesso de dados que os desenvolvedores geralmente precisam escrever. Com o Entity Framework, os desenvolvedores podem lançar consultas usando LINQ, e depois recuperar e manipular dados como objetos fortemente tipificados. A implementação do ORM do Entity Framework fornece serviços como

> rastreamento de alterações, resolução de identidades, lazy loading e tradução de consultas para que os desenvolvedores possam se concentrar na lógica de negócios de seus aplicativos em vez dos princípios básicos de acesso a dados."
>
> Visão geral e breve análise do ADO.NET Entity Framework [36]

Basicamente, o Entity Framework introduz um série de facilidades que auxiliam o acesso ao banco de dados, seja ele qual for – MySQL, MariaDB, SQLServer, Oracle, Firebird, entre outros. Fornece uma série de ferramentas integradas ao Visual Studio para gerenciar modelos, definir pontos de acesso e até mesmo criar bancos de dados [36]. Existe uma ténica chamada "Code First" que cria e atualiza banco de dados a partir de classes que definem as propriedades do mesmo.

A abordagem Code First foi introduzida na versão 4.1 do Entity Framework. O Code First permite que o desenvolvedor não precise se preocupar com acesso a banco de dados. Somente é necessário cadastrar uma *string* de conexão, para informar ao contexto do banco de dados onde este está localizado e suas configurações, referenciar as devidas .dlls de providers do banco escolhido e habilitar as *Migrations* no projeto.

Após feitas as configurações iniciais, as propriedades de tabelas do banco de dados são criadas e modificadas através de classes especiais que são registradas no contexto de acesso ao banco de dados. De fato, trata-se os modelos gerados por estas classes como seus próprios banco de dados. Exatamente como os arquivos forem construídos, o banco será gerado.

Se for necessário estender a classe que descreve o banco de dados – para inserir propriedades computadas (*getters*), por exemplo – não é necessária nenhuma configuração externa, só precisa-se inserir as propriedades nas classes adequadamente (tendo-se atenção na sintaxe). Outra vantagem é que tem-se controle de como será a classe/objeto usado para fazer-se acesso ao banco de dados, todas suas propriedades, quais serão seus nomes e tipos, e pode-se usar quase todo tipo de variável disponível em C#.

A utilização do Code First permite outro ganho quase que único na utilização de um banco de dados: controle de versão. Isso é possibilitado graças a um sistema chamado "Code First Migrations". As Migrations do Entity Framework, além do versionamento, são as responsáveis pela atualização do seu banco de dados de acordo com as classes geradas em seu projeto. O versionamento é possível através de duas maneiras: modo pontual, onde criamos os pontos de versão manualmente – e pode-se escolher qual ponto quer-se ir em uma "linha do tempo" – e o modo automático, onde toda e qualquer modificação feita nas classes de modelo será correspondida no banco de dados (sempre que fizer-se acesso ao banco por alguma função durante a execução do projeto) [45].

Uma outra possibilidade criada com a utilização de Code First e Migrations é a utilização do método *Seed* disponibilizado em 4.1.

Este método permite popular o banco de dados com dados de teste ou até mesmo dados que devem sempre estar no banco. Pode-se garantir que no banco de dados sempre vá existir o usuário ***admin*** com determinada senha e permissões de acesso, por exemplo. Dados importantes e que não devem ser deletados nem modificados no banco de dados podem e devem ser inseridos no método *Seed*.

Foram enumeradas aqui várias vantagens na utilização do Entity Framework e método Code First. É mais interessante utilizar uma técnica que abstrai algumas camadas da aplicação e dá produtividade que perder tempo com criação de banco e manutenção do mesmo utilizando métodos convencionais. Pode ser que em um projeto pequeno não se veja a importância desse controle, mas quando se tem um projeto grande – onde diversos deploys e versões são liberados – o Code First mostra sua grande superioridade.

Código 4.1: Exemplo de implementaçao da função Seed

```
protected override void Seed(DBContext context)
{
    context.Authors.AddOrUpdate(x => x.Id,
      new Author() { Id = 1, Name = "Jane Austen" },
      new Author() { Id = 2, Name = "Charles Dickens" },
      new Author() { Id = 3, Name = "Miguel de Cervantes" }
    );

    context.Books.AddOrUpdate(x => x.Id,
      new Book() { Id = 1, Year = 1813, AuthorId = 1,
        Price = 9.99M, Genre = "Comedy of manners" },
      new Book() { Id = 2, Year = 1817, AuthorId = 1,
        Price = 12.95M, Genre = "Gothic parody" },
      new Book() { Id = 3, Year = 1850, AuthorId = 2,
        Price = 15, Genre = "Bildungsroman" },
      new Book() { Id = 4, Year = 1617, AuthorId = 3,
        Price = 8.95M, Genre = "Picaresque" }
    );
}
```

4.2 ASP.NET Web API 2.0

"HTTP não serve somente para servir páginas web. Também é uma poderosa plataforma que pode ser usada para criar APIs que expõem serviços e dados. HTTP é simples, flexível e universal. Quase todas plataformas que se pode pensar tem uma biblioteca HTTP, então serviços HTTP podem alcançar uma ampla gama de clientes, incluindo serviços, dispositivos móveis e tradicionais aplicações desktop."

Getting Started with ASP.NET Web API 2 (C#) [51]

De acordo com Rick Strahl em [46], ASP.NET Web API se diferencia das soluções anteriores da Microsoft para serviços HTTP porque foi construído inteiramente em torno do protocolo HTTP e suas semânticas de mensagem. Ao contrário de WCF REST ou ASP.NET AJAX com ASMX, é uma plataforma completamente nova em vez de ser criada com tecnologia que é funciona no contexto de um framework já existente. A força da nova ASP.NET Web API é que ela combina as melhores características das plataformas que vieram antes dela, para prover uma compreensiva e muito usável plataforma HTTP. Porque é baseada em ASP.NET e toma emprestado muitos conceitos do ASP.NET MVC, Web API deve ser imediatamente familiar e confortável para a maioria dos desenvolvedores ASP.NET.

Algumas características que valem ser mencionadas:

- Suporte para roteamento URL para produzir URLs limpas usando semântica familiar a do ASP.NET MVC
- Compatível com uma série de formatos de saída incluindo JSON, XML, ATOM
- Suporte padrão aos conceitos da semântica REST, mas são opcionais
- Testável usando conceitos de teste similares ao MVC

A plataforma ASP.NET Web API se comporta melhor em projetos onde o *back-end* tem a tarefa bem definida de servir chamadas AJAX e/ou implementar uma camada REST. Não é bem utilizada em aplicações onde o *back-end* também tem a função de servir HTML (ou qualquer tipo de "material" de interface de usuário).

Essa foi a utilização no projeto desenvolvido: funcionar como serviço para requisições AJAX e expor alguma funcionalidade REST desejada, porém com controle de acesso a determinados tipos de usuário. O beneficio de construir um serviço HTTP que só possui essa responsabilidade (que não esteja associado a nenhum tipo de interface) é que o *server-side* de sua aplicação pode ser reutilizado independente do tipo de visualização e independente até mesmo do tipo de aplicação que será desenvolvida.

Além de poder ser usado no site deste projeto, a Web API pode ser usada em qualquer site que queira visualizar as vagas cadastradas, e também poderá ser utilizada futuramente caso seja necessária a construção de um aplicativo mobile, por exemplo.

4.3 Arquitetura de N-Camadas

A arquitetura de N-camadas é provavelmente um dos modelos mais utilizados na indústria. E é usado tão frequentemente por ser escalável, extensível, segura e de fácil manutenção, segundo Darius Dumitrescu, em [12]. Sua principal finalidade é ajudar a definir claramente qual a função de cada parte do software.

Algumas propriedades devem ser consideradas para que o design do projeto continue sendo viável com o passar do tempo ([12]):

- Garanta que as camadas da aplicação não dependam umas das outras. Estas precisam continuar independentes para que novas camadas possam ser adicionadas ou tecnologia antiga possa ser modificada;
- Princípio da Responsabilidade Única. Tenha certeza que cada componente ou módulo é responsável por somente uma funcionalidade ou recurso;
- Princípio do Mínimo Conhecimento. Um componente ou objeto não deve saber sobre detalhes internos de outros componentes;
- Não repita a si mesmo. Uma função específica somente deve existir em um lugar. Esta não deve ser duplicada em nenhum outro componente;
- Estabeleça padrões de desenvolvimento para o projeto. Isso proporciona consistência de código e manutenibilidade;
- Mantenha funcionalidade compartilhada tal como *log* de exceções ou segurança em um lugar que possa ser acessado por qualquer nível da aplicação.

Se forem pesquisados vários autores será notada alguma semelhança e consenso

entre as camadas sugeridas por cada um. As principais camadas adotadas são: camada de apresentação, camada de serviços, camada de componentes de negócio, camada de acesso a dados, camada de entidades. Apesar do consenso, deve-se saber que cada projeto tem suas especificidades e estas camadas podem e devem ser questionadas. Observa-se também que nem sempre o nome adotado pelos autores é o mesmo, mas a funcionalidade das camadas é similar às que serão explicadas abaixo, segundo Dumitrescu, em [12].

4.3.1 Camada de Apresentação (ou UI)

Encarregada de hospedar a interface de usuário da aplicação. É a camada onde estão os arquivos HTML, CSS e Javascript. Nesta camada é onde ficam os frameworks front-end. É onde estão os arquivos utilizados no AngularJS.

Métodos que fazem acesso a informações do banco de dados somente devem ser referenciados diretamente a partir da camada de Componentes de Negócio.

4.3.2 Camada de Serviços (ou Web API)

É a camada que possibilita expor os métodos criados na camada de Componentes de Negócio a sistemas de terceiros. No caso do projeto desenvolvido, é a única camada que faz uso dos métodos da camada de Componentes de Negócio. No sistema desenvolvido é consumida diretamente pela camada de apresentação por meio de serviços AngularJS.

Somente a camada de apresentação deve ter conhecimento da existência dessa camada. Foi implementada usando ASP.NET Web API 2.0.

4.3.3 Camada de Entidades (ou Domínio)

Deve ser responsável por armazenar todos objetos e entidades de classe usados nas outras camadas do projeto – geralmente POCOs. Contém classes de mapeamento a banco de dados, objetos de transferência de dados, modelos criados para utilizações diversas.

4.3.4 Camada de Acesso a Dados (ou Repositório)

Contém toda funcionalidade de criar, retornar, atualizar, deletar itens no banco de dados. Geralmente é utilizado algum tipo de tecnologia para auxiliar o acesso ao banco. No projeto foi utilizado Entity Framework, como já citado anteriormente. Os objetos e classes de mapeamento às tabelas do banco de dados devem estar na camada de Entidades (ou Domínio).

A única camada que deve saber da existência dessa camada é a camada de Componentes de Negócio.

4.3.5 Camada de Componentes de Negócio (ou Lógica de Negócio)

Contém toda funcionalidade central da aplicação. Seu propósito é conter toda funcionalidade customizada que envolve a utilização dos métodos disponibilizados pela camada de acesso a dados. Toda lógica de negócio está nessa camada. Entidades e objetos necessários para essas operações serão referenciados da camada de Entidades e os métodos de acesso ao banco de dados serão referenciados da camada de Acesso a Dados.

Esta camada deve ser conhecida apenas pela camada de Apresentação e pela camada de Serviços.

4.3.6 Desenvolvimento

No sistema tanto a camada de apresentação quanto a camada de serviço foram desenvolvidas no projeto chamado *App* – por questões de praticidade, já que ambas camadas seriam desenvolvidas num projeto do tipo *Web*. A camada de Entidades foi desenvolvida no projeto chamado *Domain*. A camada de Acesso a Dados foi desenvolvida no projeto chamado *Repository*. A camada de Componentes de Negócio foi desenvolvida no projeto chamado *Service*. Todos projetos exceto o projeto *App* são projetos do tipo *Class Library*.

4.4 Unidade de trabalho e Padrão Repositório

> "O padrão Repositório é usado para criar uma camada de abstração entre a camada de acesso e a camada de lógica de negócio. Essa camada de abstração contém métodos para manipulação de dados que se comunicam com a camada de acesso para servir dados conforme as exigências da camada de lógica de negócios. O principal motivo para criar essa camada de abstração é para isolar a camada de acesso a dados para que mudanças não possam afetar a camada de lógica de negócios diretamente."
>
> Repository Pattern and Unit of Work with Entity Framework [25]

Os padrões Unidade de trabalho e Repositório andam juntos. O padrão Unidade de trabalho serve para gerenciar as transações feitas com o banco de dados. A unidade de trabalho representa um estado do banco de dados e fica encarregada de disponibilizar os objetos que estão no mesmo. Implementar esses padrões isola o projeto de mudanças no banco de dados e também auxilia na utilização de testes unitários – pode-se facilmente "mockar" uma unidade de trabalho com os objetos esperados para fazer testes específicos [25].

Por estarem isolados o acesso aos dados e a aplicação, e por utilizar Entity framework, como já comentado anteriormente, não existe informação no projeto nem especificidade de sintaxe de um tipo de banco de dados. Pode-se então modificar o banco utilizado com muito mais tranquilidade do que se não utilizados esses padrões e tecnologias.

Capítulo 5

Recomendação

As aplicações web que utilizam algum tipo de sistema de recomendação estão cada vez mais comuns. Existe uma necessidade crescente de conhecer o usuário e quais suas preferências. O objetivo é prever as escolhas do usuário e que com o mínimo possível de interações um possível "cliente" seja consolidado. Este capítulo irá explicar a base das técnicas que permite essas predições, como funciona cada um dos tipos e relatar quais e como foram implementadas no sistema.

5.1 Tipos de recomendação

Sistemas de recomendação utilizam tecnologias dos mais diversos tipos, mas é possível classificá-los em dois grandes grupos: sistemas *baseados em conteúdo* e sistemas de *filtragem colaborativa*. Alguns sistemas podem também utilizar uma mistura desses dois tipos básicos.

Sistemas baseados em conteúdo utilizam somente informações sobre o item a ser escolhido e o cliente em potencial. São analisadas informações que se tem sobre o possível cliente e sobre o item. Será sugerido o item que mais tem características que parecem favorecer a escolha pelo cliente. Por exemplo, se um usuário mora em Mesquita e existem duas vagas de emprego onde a única diferença entre elas é a localidade, sendo uma em Mesquita e outra na Barra da Tijuca, espera-se que o sistema mostre a vaga mais próxima do usuário.

Sistemas de filtragem colaborativa utilizam informações sobre outros cliente e/ou outros produtos similares para fazer a recomendação. Com isso existem duas possibilidades de um item ser recomendado a um usuário: o item é similar a um item já consumido pelo próprio usuário, ou o item já foi consumido por usuários similares ao usuário.

Repare que as palavras usuário, cliente, item, produto, consumido, entre outras, somente são utilizadas para ilustração. Cabe ao leitor interpretar a situação e encaixar no caso em que se deseja aplicar a recomendação.

5.2 Porque sistemas de recomendação são necessários

O fenômeno da cauda longa, que faz a utilização de sistemas de recomendação necessária, é explicado por Jure Leskovec, Anand Rajaraman e Jeffrey D. Ullman em *Mining of Massive Data-sets* p.309 [50] .

Sistemas de entrega e lojas físicas têm uma característica em comum: a falta de recursos e limitação no espaço físico. Só é possível para uma loja física armazenar um número limitado de produtos, não é possível dar ao cliente um número muito grande de escolhas. E por isso é fácil fazer a "recomendação". Sem muitas opções esse trabalho é muito simplificado e não existe dúvida na hora de fazer a compra. O cliente já sabe o que quer.

Já em lojas e sistemas de entrega online isso não é mais verdade. Não existe limitação física. A loja pode oferecer quantidade ilimitada de itens ao cliente. Não é necessário que o estoque esteja limitado a um lugar específico, o vendedor pode trabalhar com encomendas mais facilmente e o cliente então passa a ter "tudo" ao alcance de suas mãos (ou teclas). Como o número de opções cresce, comparando-se uma loja física a uma loja online, a indecisão aumenta proporcionalmente. Muitas vezes não é possível mostrar ao cliente todas opções existentes na loja.

Um cliente não tem tempo para visualizar milhares de páginas para encontrar o que quer, e mesmo se tivesse, não o faria. É dever do vendedor passar a descobrir o que o cliente quer comprar e sugerí-lo. Essa distinção entre o mundo físico e mundo virtual é chamada fenômeno da cauda longa, que é visualizado em 5.1. O eixo vertical representa a quantidade de vezes que um determinado item foi escolhido. Os item estão ordenados na horizontal conforme suas popularidades.

Lojas físicas disponibilizam somente o visualizado em laranja, na esquerda, que representa os itens mais populares. Lojas online disponibilizam tanto itens populares

quanto os itens não populares (amarelo), da cauda longa. Não é possível mostrar todos itens a um cliente, como já dito anteriormente, logo é necessário utilizar algum tipo de sistema de recomendação.

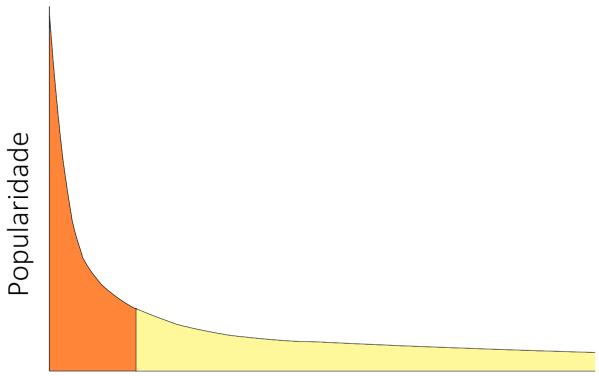

Figura 5.1: Cauda longa: lojas físicas somente disponibilizam produtos populares, lojas online podem disponibilizar todos

5.3 Desenvolvimento

O algoritmo de recomendação desenvolvido foi do tipo baseado em conteúdo. As características analisadas para recomendação de uma vaga são: cada um dos requisitos pedidos e seus níveis, salário e distância do usuário até o local da vaga.

O coeficientes inicialmente atribuídos para um usuário que nunca acessou o site foram estipulados através do uso de TDD com a comparação de valores em que um usuário padrão provavelmente iria preferir. Comparações entre nível de atendimento dos requisitos da vaga *versus* salário *versus* distância foram montados em forma de teste, gradativamente, e os pesos foram ajustados para que esses testes fossem atendidos.

O que deve-se observar é que, inicialmente, essa relação pré-estabelecida pode parecer satisfatória. Porém cada individuo tem suas preferências. Alguns podem dar mais importância a determinadas qualidades que outros. Pensando nisso, o sistema foi incrementado utilizando algoritmos baseados em filtragem colaborativa. A idéia é criar um sistema de recomendação misto. Desenvolvendo uma recomendação não "engessada", que pode ser aprimorada com o tempo, conforme o número de usuários do site aumenta.

Aprimorando o algoritmo base - Filtragem colaborativa

Após um usuário acessar um anúncio de vaga existe a opção de classificar a mesma clicando nos botões "gostei" ou "não gostei". Pode parecer simples, mas não era desejado adicionar nada que atrapalhe a utilização do site e este tipo de classificação já é o suficiente para implementar um algoritmo de filtragem colaborativa.

Com o auxílio da classificação de cada usuário pode-se montar uma matriz de utilidade, como a tabela 5.1 mostra.

	Vaga 1	Vaga 2	Vaga 3	Vaga 4	Vaga 5
Usuário 1		+1		-1	+1
Usuário 2			+1		-1
Usuário 3	+1	+1	-1		+1
Usuário 4		-1	+1	+1	

Tabela 5.1: Exemplo de Matriz de utilidade

A matriz de utilidade pode ser usada para fazer a comparação entre os usuários. A finalidade é encontrar usuários que sejam similares de acordo com suas preferências. Na matriz 5.1 o número "+1" representa que o usuário gostou do anúncio de uma vaga, o número "-1" significa que o usuário não gostou do anúncio de uma vaga. A partir daí pode-se utilizar um método de cálculo de distâncias entre usuários baseado no *coeficiente de Jaccard*, como explicado em [50] p. 322. Existem outros métodos para calcular esta distância, porém este método se mostrou eficaz para calcular distâncias de matrizes de sistemas de *like/dislike*, que é exatamente o caso do projeto. Algumas referências sobre utilização de Jaccard para medir similaridade são encontradas em [39], [11] e [5].

O algoritmo usado para medir-se a similaridade entre usuários é visualizado em 5.1. O algoritmo usado para medir-se a predição de pontuação de vaga é visualizado em 5.2.

Código 5.1: Similaridade entre usuários

```
 public double Similaridade(Usuario usuario, Usuario outro)
{
    double acordos = 0.0;
    double desacordos = 0.0;
    double totalAvaliacoes = 0.0;

    acordos += usuario.avaliacoes.Where(a =>
                    outro.avaliacoes.Any(b =>
                        b.vaga.ID == a.vaga.ID
                        && b.Gostou == a.Gostou))
                        .Count();
    desacordos += usuario.avaliacoes.Where(a =>
                    outro.avaliacoes.Any(b =>
                        b.vaga.ID == a.vaga.ID
                        && b.Gostou != a.Gostou))
                        .Count();

    totalAvaliacoes = usuario.avaliacoes
                    .Concat(outro.avaliacoes)
                    .GroupBy(a => a.vaga.ID)
                    .Count();

    return (acordos - desacordos) / totalAvaliacoes;
}
```

Código 5.2: Predição de pontuação de vaga

```
public double PredicaoDeVaga(Usuario usuario, Vaga vaga)
{
    double pontuacao = 0.0;

    IEnumerable<Avaliacao> avaliacoes =
                    vaga.avaliacoes.Where(a =>
                        a.usuario.ID != usuario.ID);

    double avaliadoPor = avaliacoes
                            .GroupBy(a =>
                                a.usuario.ID)
                                .Count();

    foreach (Avaliacao avaliacao in avaliacoes)
    {
        if (avaliacao.Gostou)
        {
            pontuacao +=
                Similaridade(usuario, avaliacao.usuario);
        }
        else
        {
            pontuacao -=
                Similaridade(usuario, avaliacao.usuario);
        }
    }

    return pontuacao / avaliadoPor;
}
```

Capítulo 6

Trabalhos Futuros

Este capítulo irá citar sugestões de possíveis futuras implementações e aprimoramentos no projeto desenvolvido.

O foco do desenvolvimento da monografia foi na criação da documentação e pesquisa de melhores práticas e padrões de projeto quando se faz um projeto web utilizando ASP.NET Web Api 2.0 no back-end e AngularJS no front-end. Com adição a isso, foi abordada a adoção de sistemas de recomendação para criar melhor experiência de uso em aplicações web, onde se faz necessário a sugestão de conteúdo para usuários.

Estes padrões e sistema de recomendação foram implementados num sistema de gerenciamento de vagas para universitários como forma de demonstração prática da utilização dos mesmos, porém alguns detalhes mostram-se úteis para melhor aproveitamento do site. Abaixo serão comendatas as ideias para futuras implementações/adições ao sistema.

6.1 Integração com mídias sociais

O sucesso de um sistema web depende da sua utilização. Sua utilização depende do quão bem divulgado este é. As mídias sociais estão "na moda". Deve-se saber aproveitar este potencial de marketing gratuito de forma eficiente. Alastair Thompson explica em [48] 10 benefícios ao se utilizar o marketing em mídias sociais. Entre eles estão: o aumento do reconhecimento da marca, maiores taxas de conversão, melhores resultados em ferramentas de busca, aumento de tráfego de entrada, entre outros.

Implementar integração com ao menos uma mídia social é indispensável para a sobrevivência do sistema. Os aspectos que se sugere ter como foco são login via

Facebook, compartilhamento de anúncios de maneira facilitada e incorporação de *meta tags* nos anúncios e site para utilização no Facebook. Sobre *meta tags* pode-se ler em [33] e [14].

6.2 Finalização de interações com usuário

Algumas interações ainda não foram concluídas no desenvolvimento do site. Pode-se indicar a finalização da implantação do sistema de confirmação de e-mail, criação de anúncio, edição de perfil de usuário, visualização de perfil de usuário, entre outros.

Capítulo 7

Conclusão

Muito trabalho e tempo foram necessários para adquirir confiança em dizer que uma determinada arquitetura é uma das melhores e que atende determinados requisitos como funcionalidade, confiabilidade, usabilidade, eficiência e manutenibilidade.

Nos quesitos *client-side* e *server-side* diversas tecnologias foram experimentadas. Projetos paralelos a este foram usados como experiências. Foram desenvolvidos sistemas web com as seguintes combinações de tecnologia: ASP.NET MVC, Razor e Javascript; ASP.NET MVC, Razor e KnockoutJS; ASP.NET MVC e AngularJS e por último ASP.NET Web API e AngularJS.

Sem dúvidas a melhor escolha, dentre estas, foi a de utilizar ASP.NET Web API em conjunto com AngularJS. Essas duas tecnologias se completam, não existe sobreposição de funcionalidades e sua integração é bem simples.

Depois de escolhidos os *frameworks*, desejou-se saber qual melhor estrutura para sua utilização. De nada adianta ter uma poderosa "arma" de desenvolvimento e não saber manuseá-la. A intenção da pesquisa foi em como adotar uma boa estrutura de organização de código tanto na manipulação da Web Api quanto na organização dos arquivos e componenetes utilizados pelo AngularJS.

No *server-side* foi utilizada o padrão de projeto de N-Camadas, sendo a camada de acesso ao banco constituída por uma "Unidade de trabalho", como explicado em capítulos anteriores. No *client-side* foi usado como base o guia de estilos para AngularJS do John Papa, encontrado em [43]. Este guia foi adaptado para a realidade do projeto. Na criação de serviços do AngularJS foi utilizado TypeScript, com intuito de aumentar a produtividade no desenvolvimento.

Vale ressaltar que estes padrões adotados só foram possíveis graças a escolha de frameworks/templates que dão suporte e liberdade à estruturação de arquivos/funcionalidades. Alguns destes, por mais que se queira, te obrigam a adotar determinadas

estruturas que impedem a implementação de padrões mais bem aceitos no mundo do desenvolvimento. Então além da escolha do padrão a ser adotado, a escolha da tecnologia base para sua aplicação é muito importante.

Sobre o sistema de recomendação notou-se que este é indispensável quando um sistema web precisa fazer sugestões a usuários. E, mesmo que não exista muita informação sobre o "cliente" em potencial, existem maneiras de implementar sistemas não complicados (tanto na utilização quanto implementação) que podem atingir o objetivo de tornar o conteúdo disponibilizado mais similar ao gosto de quem está fazendo o consumo da informação.

Bibliografia

[1] Ben McCann et al. *AngularJS - Creating Custom Directives*. Dez. de 2014. URL: https://docs.angularjs.org/guide/directive.

[2] Bryan Ford et al. *AngularJS - Services*. Dez. de 2014. URL: https://docs.angularjs.org/guide/services.

[3] Igor Minar et al. *AngularJS - Dependency Injection*. Dez. de 2014. URL: https://docs.angularjs.org/guide/di.

[4] Igor Minar et al. *ngView - AngularJS*. Jan. de 2015. URL: https://docs.angularjs.org/api/ngRoute/directive/ngView.

[5] Siwei Lai et al. *Hybrid Recommendation Models for Binary User Preference*. Jun. de 2012. URL: http://jmlr.csail.mit.edu/proceedings/papers/v18/lai12a/lai12a.pdf.

[6] Tobias Bosch et al. *AngularJS - Understanding Controllers*. Dez. de 2014. URL: https://docs.angularjs.org/guide/controller.

[7] *Backbone.JS*. Mar. de 2014. URL: http://backbonejs.org/.

[8] *Bootstrap*. Dez. de 2013. URL: http://getbootstrap.com/.

[9] *CanJS — Build better apps, faster*. Mar. de 2014. URL: http://canjs.com/.

[10] Mark J. Caplin. *Developing a Large Scale Application with a Single Page Application (SPA) using AngularJS*. Jan. de 2015. URL: http://www.codeproject.com/Articles/808213/Developing-a-Large-Scale-Application-with-a-Single.

[11] David Celis. *Collaborative filtering with likes and dislikes*. Jan. de 2015. URL: http://davidcel.is/blog/2012/02/07/collaborative-filtering-with-likes-and-dislikes/.

[12] Darius Dumitrescu. *N-Tier Architecture. ASP.NET Example*. Jan. de 2015. URL: http://dotnetdaily.net/tutorials/n-tier-architecture-asp-net/.

[13] Facebook. *Facebook*. Fev. de 2015. URL: http://facebook.com.

[14] Facebook. *Sharing Best Practices for Websites & Mobile Apps*. Fev. de 2015. URL: https://developers.facebook.com/docs/sharing/best-practices?locale=pt_BR.

[15] Tom FitzMacken. *Introduction to ASP.NET Web Programming Using the Razor Syntax (C-Sharp)*. Dez. de 2013. URL: http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-(c).

[16] The jQuery Foundation. *jQuery ajax API*. Jan. de 2015. URL: http://api.jquery.com/jquery.ajax/.

[17] Google. *AngularJS — Superheroic JavaScript MVW Framework*. Mar. de 2014. URL: https://angularjs.org/.

[18] Martin Heller. *Review: Visual Studio 2013 reaches beyond the IDE*. Dez. de 2014. URL: http://www.infoworld.com/article/2609091/application-development/review--visual-studio-2013-reaches-beyond-the-ide.html.

[19] Miško Hevery. *Miško Hevery - GitHub*. Dez. de 2014. URL: https://github.com/mhevery.

[20] Luke Hoban. *Overview of ECMAScript 6 features*. Dez. de 2014. URL: https://github.com/lukehoban/es6features.

[21] Caio Humberto. *Design Patterns – Singleton*. Dez. de 2014. URL: http://www.devmedia.com.br/design-patterns-singleton-parte-3/16782.

[22] Tidle Inc. *EmberJS — A Framework for creating ambitious web applications*. Mar. de 2014. URL: http://emberjs.com/.

[23] *jQuery - write less, do more*. Jan. de 2014. URL: http://jquery.com/.

[24] Josh JW. *Partial View in ASP.NET MVC 4*. Jan. de 2015. URL: http://www.codeproject.com/Tips/617361/Partial-View-in-ASP-NET-MVC.

[25] Amir Hamza Md. Kayes. *Repository Pattern and Unit of Work with Entity Framework in ASP.NET MVC*. Jan. de 2015. URL: http://www.codeproject.com/Articles/688929/Repository-Pattern-and-Unit-of.

[26] *KnockoutJS*. Dez. de 2013. URL: http://knockoutjs.com/.

[27] Shivprasad Koirala. *Dependency Injection (DI) vs. Inversion of Control (IOC)*. Dez. de 2014. URL: http://www.codeproject.com/Articles/592372/Dependency-Injection-DI-vs-Inversion-of-Control-IO.

[28] Teoli; kuni71. *About JavaScript*. Dez. de 2014. URL: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/About_JavaScript.

[29] Rod Johnson; Bob Lee. *JSR 330: Dependency Injection for Java*. Dez. de 2014. URL: https://jcp.org/en/jsr/detail?id=330.

[30] Pete LePage. *Princípios básicos de Web design responsivo*. Dez. de 2014. URL: https://developers.google.com/web/fundamentals/layouts/rwd-fundamentals/.

[31] Motive Ltd. *The Motive Web Design Glossary*. Dez. de 2014. URL: http://www.motive.co.nz/glossary/client-server.php.

[32] PC Magazine. *Boilerplate code definition - PC Magazine Encyclopedia*. Fev. de 2015. URL: http://www.pcmag.com/encyclopedia/term/62258/boilerplate-codeDI.

[33] Victor Matias. *Utilizando as metatags de OpenGraph*. Fev. de 2015. URL: http://tableless.com.br/utilizando-meta-tags-facebook/.

[34] Microsoft. *Model-View-Controller*. Fev. de 2015. URL: https://msdn.microsoft.com/en-us/library/ff649643.aspx.

[35] Microsoft. *Visual Studio*. Dez. de 2013. URL: http://www.visualstudio.com/.

[36] Microsoft. *Visão geral e breve análise do ADO.NET Entity Framework*. Jan. de 2015. URL: http://msdn.microsoft.com/pt-br/data/aa937709.aspx.

[37] Igor Minar. *AngularJS v0.9.0 - GitHub*. Dez. de 2014. URL: https://github.com/angular/angular.js/releases/tag/v0.9.0.

[38] Igor Minar. *MVC vs MVVM vs MVP. Post on Google+*. Jul. de 2014. URL: https://plus.google.com/+AngularJS/posts/aZNVhj355G2.

[39] Guy Morita. *Recommendation Racon*. Jan. de 2015. URL: https://github.com/guymorita/recommendationRaccoon.

[40] LLC Microsoft by Neudesic. *ASP.NET MVC 5*. Dez. de 2013. URL: http://www.asp.net/mvc/mvc5.

[41] Vlad Orlenko. *AngularJS for jQuery Developers*. Dez. de 2014. URL: http://www.artandlogic.com/blog/2013/03/angularjs-for-jquery-developers/.

[42] Addy Osmani. *Understanding MVVM – A Guide For JavaScript Developers*. Dez. de 2014. URL: http://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/.

[43] John Papa. *John Papa — Guia de estilo de código para AngularJS*. 2014. URL: https://github.com/johnpapa/angularjs-styleguide.

[44] Roy Peled. *An MVP guide to JavaScript – Model-View-Presenter*. Dez. de 2014. URL: http://www.roypeled.com/an-mvp-guide-to-javascript-model-view-presenter/.

[45] Carlos dos Santos. *Entity Framework Code First – Migrations*. Jan. de 2015. URL: http://msdn.microsoft.com/pt-br/library/jj856238.aspx.

[46] Rick Strahl. *Where does ASP.NET Web API Fit?* Jan. de 2015. URL: http://weblog.west-wind.com/posts/2012/Aug/07/Where-does-ASPNET-Web-API-Fit.

[47] Bhim Bahadur Thapa. *Dependency Injection (DI)*. Dez. de 2014. URL: http://www.codeproject.com/Tips/657668/Dependency-Injection-DI.

[48] Alastair Thompson. *Os 10 principais benefícios do marketing de mídia social*. Fev. de 2015. URL: http://www.webdesignflorianopolis.com.br/10-principais-beneficios-marketing-de-midia-social/.

[49] *Typescript Language Site*. Nov. de 2014. URL: http://www.typescriptlang.org/.

[50] Jure Leskovec; Anand Rajaraman; Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge, United Kingdom: Cambridge University Press, 2014, pp. 307–340. URL: http://infolab.stanford.edu/~ullman/mmds/book.pdf.

[51] Mike Wasson. *Getting Started with ASP.NET Web API 2*. Dez. de 2014. URL: http://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api.

[52] Bruna Xavier. *Modelo Latex para Monografias do TCC - DCC UFRJ*. Fev. de 2015. URL: https://github.com/dcc-ufrj/monografia-latex.