



Universidade Federal
do Rio de Janeiro
Escola Politécnica

UM ESTUDO SOBRE PADRÕES E TECNOLOGIAS PARA O DESENVOLVIMENTO WEB - FRONT-END

Pedro Menezes Ribeiro de Souza

Projeto de Graduação apresentado ao Curso de Engenharia Eletrônica e de Computação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Heraldo Luis Silveira de Almeida

Rio de Janeiro

Setembro de 2016

pedromrsouza@poli.ufrj.br

DRE: 110073292

UM ESTUDO SOBRE PADRÕES E TECNOLOGIAS PARA O DESENVOLVIMENTO WEB - FRONT-END

Pedro Menezes Ribeiro de Souza

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRÔNICO E DE COMPUTAÇÃO

Autores:

Pedro Menezes Ribeiro de Souza

Orientador:

Heraldo Luis Silveira de Almeida, D.Sc.

Examinador:

Flávio Luis de Mello, D.Sc.

Examinador:

Aloysio de Castro Pinto Pedroza, Dr.

Rio de Janeiro – RJ, Brasil

Setembro de 2016

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Escola Politécnica – Departamento de Eletrônica e de Computação

Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária

Rio de Janeiro – RJ CEP 21949-900

Este exemplar é de propriedade da Universidade Federal do Rio de Janeiro, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es).

Meu profundo agradecimento,

Ao meu orientador, Heraldo, que apoiou e acolheu o tema do meu trabalho

Ao meu grande amigo, Vitor, que me auxiliou durante toda a elaboração do trabalho

A todos os meus amigos que sempre estiveram comigo, me dando apoio, amizade e amor

Aos amores da minha vida, que são a razão de tudo o que faço: June, Bira & Gabriel

RESUMO

O desenvolvimento de sistemas web vem expandindo não só a sua importância como também a sua complexidade. O número de dispositivos conectados à Internet vem crescendo de forma expressiva, fazendo com que seja de extrema importância a criação de sistemas flexíveis.

Com o tempo, as tecnologias para desenvolvimento web *front-end* evoluíram consideravelmente, a fim de proporcionar ao usuário experiências cada vez melhores de forma eficiente. A web atual é produto do esforço de uma comunidade aberta, que vem surgindo com diversas tecnologias e conceitos que tornam este crescimento sustentável. Neste projeto serão expostas e comparadas diversas tecnologias e conceitos presentes neste novo mundo de desenvolvimento web front-end. Como prova de conceito, será implementada uma aplicação simples com as ferramentas apresentadas.

Palavras-Chave: JavaScript, *Front End*, Desenvolvimento, *Software*, Web, Internet, MVC.

ABSTRACT

The development of web systems are expanding, not only in importance, but also in complexity. The number of different devices connected to Internet have been notoriously increasing, so that the creation of flexible systems has become extremely important. As time went by, technologies for the development of web front-end evolved substantially, with the purpose to provide the user even better experiences in an effective way. The current web is a product of the effort of an open community, which has created several new technologies and concepts that make this development sustainable. In this project, many technologies and concepts that compose this new world of web front-end development will be exposed and compared. The implementation of a simple application with the use of the techniques presented in this work will be presented as concept proof.

Key-words: JavaScript, *Front End*, Development, *Software*, Web, Internet, MVC.

SIGLAS

- **AJAX** - Asynchronous Javascript and XML
- **API** - Application programming interface
- **CSS** - Cascading Style Sheets
- **CVE** - Common Vulnerabilities and Exposures
- **DOM** - Document Object Model
- **HTML** - HyperText Markup Language
- **HTTP** - Hypertext Transfer Protocol
- **JSON** - JavaScript Object Notation
- **SPA** - Single-page application
- **TDD** - Test Driven Development
- **UI** - Interface do usuário
- **URL** - Uniform Resource Locator
- **XP** - Extreme Programming
- **XXS** - Corss-site scripting

LISTA DE FIGURAS

Figura 1 Fluxo da arquitetura cliente-servidor	6
Figura 2 Ciclo de vida das aplicações web tradicionais	7
Figura 3 Ciclo de vida de uma SPA.....	9
Figura 4 Fluxo do desenvolvimento orientado a testes (TDD).....	12
Figura 5 Arquitetura de aplicações AngularJS	18
Figura 6 Fluxo de dados em uma aplicação AngularJS.....	19
Figura 7 Fluxo de uma aplicação desenvolvida com Backbone.js	20
Figura 8 Fluxo de uma aplicação desenvolvida em Ember.js	22
Figura 9 Gráfico que relaciona a porcentagem de sites que utilizam as três ferramentas	32
Figura 10 No grunt, precisamos criar arquivos intermediários em disco	52
Figura 11 No Gulp nós podemos canalizar os arquivos intermediários na memória para serem usados por outros fluxos	52
Figura 12 Criação da estrutura do projeto com o Yeoman (Angular Generator)	63
Figura 13 Estrutura de diretórios criada pelo Angular Generator (Yeoman)	64
Figura 14 Estrutura da pasta app	65
Figura 15 Aplicação inicial criada pelo Yeoman	66
Figura 16 Testes do serviço de comunicação com o servidor falhando	72
Figura 17 Testes do serviço de comunicação com o servidor sendo bem sucedidos	74
Figura 18 Testes do controlador de adição de usuários falhando.....	79
Figura 19 Testes de unidade do controlador de adição de usuários sendo executado com sucesso.....	82
Figura 20 Botão "Adicionar" desabilitado devido ao preenchimento incorreto do formulário	84

Figura 21 Botão "Adicionar" habilitado devido ao preenchimento correto do formulário	85
Figura 22 Testes do controlador de exibição da lista de usuários falhando	89
Figura 23 Testes obtendo sucesso após implementação do controlador de exibição de lista de contatos	90
Figura 24 Visão da listagem de contatos	92
Figura 25 Listagem de contatos filtrada	93
Figura 26 Teste fim-a-fim executado com sucesso	101
Figura 27 Estrutura de pastas do projeto em desenvolvimento	102
Figura 28 Estrutura de pastas da aplicação pós build	102

LISTA DE TABELAS

Tabela 1 Dados gerais a respeito da comunidade das ferramentas. (As consultas foram realizadas em 27 de Agosto de 2016).....	24
Tabela 2 Tabela que compara o tamanho em Kilobytes dos frameworks AngularJS, Backbone.js, Ember.js	26
Tabela 3 Comparação entre aspectos relacionados a segurança das aplicações.....	29
Tabela 4 Comparação entre os tres frameworks no que diz respeito ao seu histórico a mantenedores.....	30
Tabela 5 Interesse de pesquisa sobre as LESS e Sass no decorrer do tempo	42

Sumário

1. Introdução	1
1.1 Tema.....	1
1.2 Finalidade/Justificativa	1
1.3 Metodologia	2
2. Referencial Teórico	3
2.1 HTML 5	3
2.2 CSS 3	4
2.3 JavaScript	4
2.4 JQuery.....	5
2.5 Arquitetura Cliente-Servidor	5
2.6 Aplicações Web Tradicionais.....	6
2.7 Aplicações Web Modernas.....	8
2.8 SPA (Single-page Applications).....	9
2.9 O Padrão de Arquitetura MVC.....	10
2.10 TDD - Test Driven Development	11
3. Desenvolvimento de Aplicações Web <i>Front-End</i>	16
3.1 Frameworks e bibliotecas <i>JavaScript</i> MVC.....	16
3.1.1 Vantagens da utilização de frameworks e bibliotecas JavaScript MVC	17

3.1.2 AngularJS.....	17
3.1.3 Backbone.js.....	20
3.1.4 Ember.js	22
3.1.5 - Estudo comparativo	23
3.2 Pré-processadores de CSS.....	33
3.2.1 - Sass	34
3.2.2 - LESS.....	34
3.2.3 - Aspectos das linguagens.....	34
3.2.4 Estudo Comparativo	41
3.3 Gerenciadores de Dependência.....	44
3.3.1 Bower.....	44
3.4 Automatizadores de Tarefas	45
3.4.1 Gerenciamento de fluxo de trabalho	45
3.4.2 Build.....	46
3.5 Testes em aplicações web	55
3.5.1 Teste de unidade	56
3.5.2 Testes de integração.....	57
3.5.3 Testes fim-a-fim.....	57
3.5.4 - Karma - Execução de testes.....	58
3.5.5 - Jasmine - Escrita de testes	60
3.5.6 - Protactor	60
4. Exemplo de Aplicação.....	61

4.1 - Configurando o ambiente de desenvolvimento	62
4.2 - Criando a estrutura da aplicação utilizando o Yeoman (Angular Generator).....	62
4.2.1 Módulos do AngularJS	66
4.3. Criando o serviço de comunicação com o servidor.....	68
4.3.1 - O serviço \$http	68
4.3.2 - Criando testes de unidade para o nosso serviço	69
4.3.3 Criando o serviço	72
4.4 Implementando a adição de novos usuários.....	74
4.4.1 Criando a estrutura da funcionalidade	74
4.4.2 - Criação da rota.....	75
4.4.3 - Criando os testes de unidade para o controlador	76
4.4.4 Criando o controlador	80
4.4.5 Criando a Visão	82
4.5 - Implementando a listagem dos usuários	85
4.5.1 - Criando a estrutura da funcionalidade	86
4.5.2 - Criação da rota.....	87
4.5.3 - Criando o teste de unidade para o controlador	88
4.5.4 - Criando o controlador.....	89
4.5.5 - Criando a visão	90
4.6 - Utilização de interceptadores	93
4.7. Criando testes fim-a-fim com Prtractor	96

4.7.1 instalando o Protractor	96
4.7.2 - Criando a configuração do Protractor.....	97
4.7.3. Criando o teste fim-a-fim.....	98
4.7.4 - Executando o teste fim-a-fim	100
4.8 - Build utilizando o Grunt	101
5 Conclusões	103
Referências	105

1. Introdução

1.1 Tema

Este projeto tem como objetivo apresentar as diversas tecnologias e conceitos que foram inseridos no desenvolvimento web *front-end* nos últimos anos. Serão apresentados estudos comparativos entre as diversas ferramentas de código aberto presentes no mercado e o desenvolvimento de uma aplicação simples utilizando as escolhidas.

1.2 Finalidade/Justificativa

O desenvolvimento de aplicações web vem crescendo não só em importância como também em complexidade. Com o tempo, as tecnologias e ferramentas para desenvolvimento web *front-end* evoluíram consideravelmente, a fim de proporcionar ao usuário experiências cada vez melhores de forma eficiente.

Para isso, a comunidade de desenvolvedores web *front-end* vem criando novas tecnologias e desenvolvendo novas ferramentas que auxiliam a criação de sistemas cada vez mais robustos e escaláveis. Esta nova era do desenvolvimento web *front-end* ainda é muito desconhecida no meio acadêmico, porém cada vez mais reconhecida e requisitada pelo mercado.

Este trabalho tem como objetivo apresentar e comparar as novas tecnologias e ferramentas que vêm tornando o desenvolvimento web *front-end* cada vez mais complexo e desafiador, de modo a auxiliar estudantes e profissionais no desafio de escolher, em meio a um grande número de opções, tecnologias adequadas para o desenvolvimento do *front-end* de suas aplicações.

1.3 Metodologia

Primeiramente serão abordados conceitos necessários para o entendimento do estado atual do desenvolvimento de sistemas *front-end*. Feito isso, serão apresentadas diversas tecnologias e ferramentas de código aberto presentes no mercado, juntamente com um estudo comparativo entre ferramentas alternativas semelhantes. Baseado nesse estudo comparativo, algumas ferramentas serão selecionadas e melhor detalhadas em um exemplo de uma aplicação web *front-end* simples, como prova de conceito.

2. Referencial Teórico

Nesta seção serão abordados conceitos necessários para o entendimento do estado atual do desenvolvimento de sistemas *front-end*.

2.1 HTML 5

O *HTML* foi inicialmente projetado para descrever semanticamente documentos científicos. Hoje em dia é uma tecnologia chave da Internet, sendo uma linguagem para estruturação, apresentação e conteúdo para a *World Wide Web* (WWW).

O *HTML 5* (PILGRIM, 2009), que surgiu a partir de um consórcio entre a W3C (World Wide Web Consortium) e a WHATWE (Web Hypertext Application Technology Working Group), trouxe melhorias significativas com novas funcionalidades de semântica e acessibilidade, ao mesmo tempo melhorando o suporte aos mais recentes tipos de conteúdo multimídia.

As principais novas características da versão 5 do *HTML* estão ligadas às necessidades de suporte aos novos formatos de conteúdo multimídia, novas funcionalidades de semânticas e acessibilidade. São elas:

- **Inclusão do elemento canvas para desenho:** O elemento canvas permite desenhar gráficos em uma página web sem instalação de nenhum plug-in externo.
- **Inclusão de elementos de vídeo e áudio para reprodução multimídia:** O *HTML5* dá suporte nativo para a reprodução de áudio e vídeo sem a necessidade de utilizar mecanismos externos.
- **Melhor suporte para armazenamento local:** São oferecidos 2 novos objetos para armazenar dados locais: *sessionStorage* e *localStorage*.

- **Inclusão de novos elementos de conteúdo específico:** A nova versão traz novos elementos para proporcionar aos usuários uma melhor estrutura, desenho e conteúdo multimídia.
- **Inclusão de novos controles para formulário:** Foram adicionados novos controles para formulários, como `<datalist>`, `<keygen>` e `<output>`.
- **Suporte total ao CSS3:** Deste modo as páginas webs podem receber os mais variados tipos de estilos que são oferecidos pela versão 3 das Cascading Style Sheets, descrita na próxima subseção.

2.2 CSS 3

Cascading Style Sheet, ou *CSS* (SCHMITT, 2009) , é uma linguagem de folhas de estilo que formata a informação entregue pelo *HTML*, nos possibilitando realizar a devida separação entre o modo de apresentação e o conteúdo do documento. Entende-se como informação qualquer elemento criado, como imagem, texto, áudio ou vídeo. Essa informação na maioria das vezes é visual, mas nem sempre. Com o *CSS Aural* também se pode manipular o áudio que é apresentado ao usuário, regulando seu som, profundidade etc. O *CSS* prepara todos esses tipos de informação para que sejam exibidos da melhor forma para o usuário.

2.3 JavaScript

O *JavaScript* (OSMANI, 2016) é uma linguagem de programação dinâmica de alto nível e é adequada ao estilo de programação funcional, estando presente em diversos navegadores e permitindo a melhoria da comunicação entre o usuário e as aplicações web.

A linguagem surgiu em 1995 e sua principal tarefa era realizar a validação de dados de formulários, que anteriormente era realizada do lado do servidor. Com o passar do tempo veio ganhando popularidade e se tornou uma linguagem completa.

Essa popularidade veio junto com a criação de chamadas *AJAX*, com o surgimento de bibliotecas robustas, como o *JQuery* e posteriormente com o surgimento de frameworks e bibliotecas para o desenvolvimento de aplicações web *client-side* (lado do cliente) que utilizam o padrão de arquitetura MVC (ou MV*).

2.4 JQuery

O *JQuery* (OSMANI, 2016) é uma biblioteca *JavaScript* de código aberto que facilita a interação entre o documento *HTML* e a *DOM*. Esta biblioteca possui funcionalidades como manipulação da *DOM* (HAROL, 2002), resposta a eventos, criação de animações e realização de chamadas *AJAX* de forma simples. A primeira versão estável foi lançada em 2006. O *JQuery* foi muito importante para resolver o problema de inconsistência (diferença de comportamento) entre os navegadores ao executar código *JavaScript*, garantindo um comportamento igual entre eles.

2.5 Arquitetura Cliente-Servidor

BATTISTI (2001,p.27) define o modelo cliente-servidor como uma arquitetura onde o processamento da informação é dividido em processos ou módulos distintos. Desta forma, um processo é responsável pela obtenção dos dados (cliente) e o outro pela manutenção da informação (servidor).

Outra definição feita por Battisti (2001,p. 39)

“Sistema inovador surgido nos anos 90 e muito utilizado no meio corporativo, baseado em três componentes principais: gerenciamento de banco de dados, que funcionam como servidores; redes, que funcionam como meio de transporte de dados e, finalmente, softwares para acesso aos dados: Clientes.”

É nos servidores onde se encontram as regras de negócio da aplicação, sendo ele responsável pela manipulação das informações e pela persistência no banco de dados. Podemos visualizar o fluxo dessa arquitetura na Figura 1.

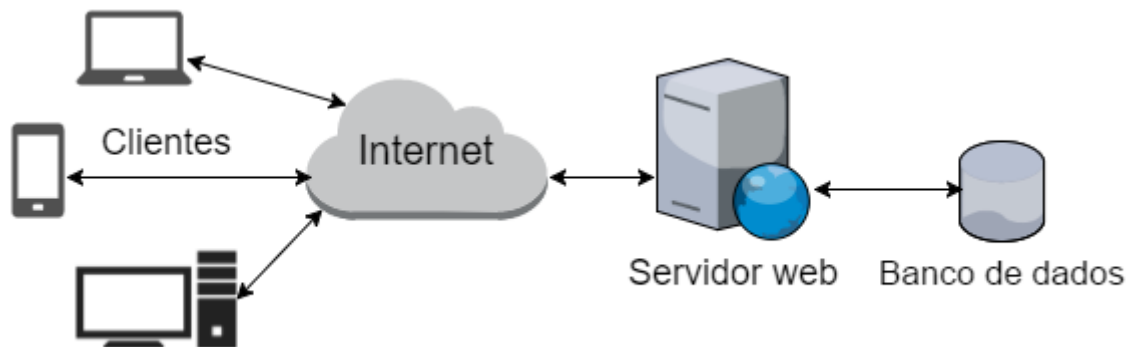


Figura 1 Fluxo da arquitetura cliente-servidor

Essa arquitetura nos proporciona diversas vantagens, como a atualização mais fácil das regras de negócio, já que estas devem ser atualizadas apenas no lado do servidor e não no lado dos clientes (VASKEVITCH, 1995). Além disso, proporciona maior segurança e controle de acesso aos dados.

2.6 Aplicações Web Tradicionais

Com o surgimento dos *web servers* (servidores web), que recebem as requisições e devolvem o conteúdo requisitado para o cliente, as páginas da web passaram a se tornar dinâmicas. (FINK; FLATOW, 2014, p. 3)

O ciclo de vida destas aplicações pode ser visualizado na Figura 2:

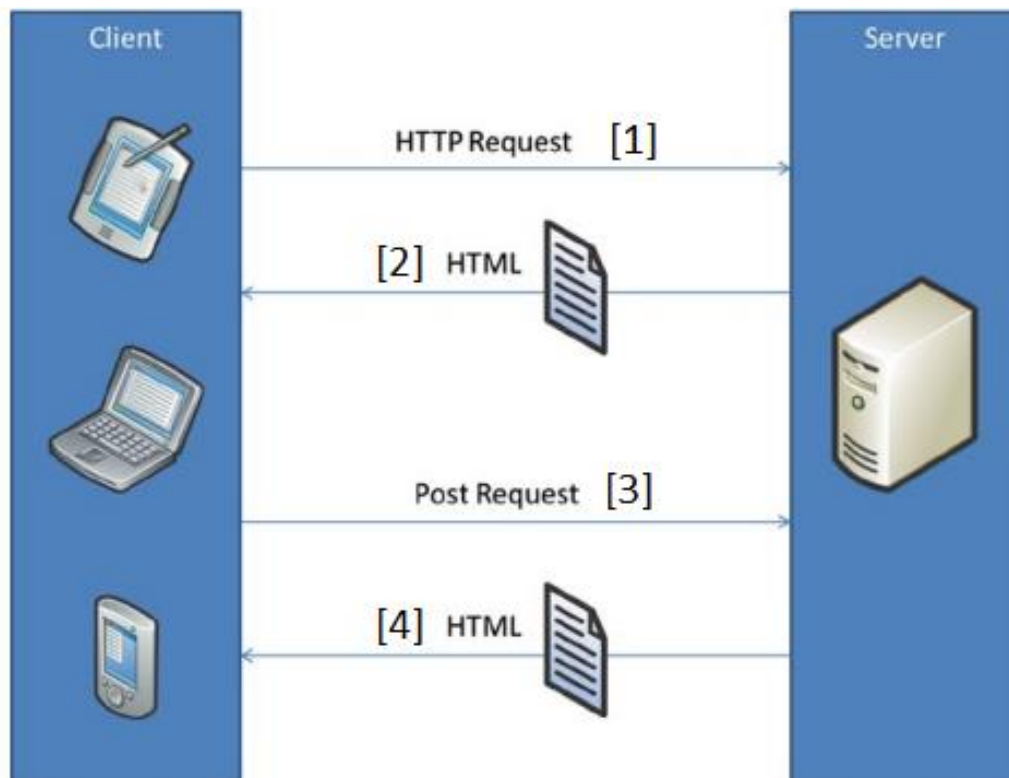


Figura 2 Ciclo de vida das aplicações web tradicionais

Fonte: FINK; FLATOW, 2014

Este fluxo pode ser traduzido da seguinte maneira:

1. Inicialmente o cliente realiza uma requisição *HTTP* ao servidor.
2. O servidor responde com uma página *HTML*
3. Através da submissão de um formulário, por exemplo, o usuário realiza uma requisição *HTTP POST* para o servidor.
4. O mesmo retorna novamente uma página *HTML*, fazendo com que a página inteira seja recarregada e o novo conteúdo seja exibido para o usuário.

Em toda interação entre o usuário e a página são enviadas requisições *HTTP* ao servidor, que é o responsável por controlar a aplicação.

Neste tipo de aplicação a cada recarga na página, todos os arquivos *HTML*, *CSS* e *JavaScript* são baixados novamente, gerando um *overhead* (sobrecarga) desnecessário de

cabeçalhos e requisições ao servidor. A lentidão gerada por esta abordagem levou a criação do modelo de desenvolvimento moderno de aplicações web.

2.7 Aplicações Web Modernas

Ao contrário do desenvolvimento tradicional, o estilo de desenvolvimento moderno trabalha de forma assíncrona, buscando não bloquear a UI (interface do usuário) durante as requisições. Para isto, são utilizadas linguagens *client-side* (do lado do cliente) nativas dos navegadores e extensamente suportadas, como o *JavaScript*, para a manipulação do documento. Estas linguagens não necessitam da instalação de nenhum *plug-in* e são multiplataforma, uma vez que basta o dispositivo possuir um navegador que as interprete.

Este estilo de desenvolvimento trabalha de forma independente da linguagem *server-side*, utilizando *HTML5* e chamadas *AJAX* (WOYCHWSKY, 2006). Desta forma o servidor passa apenas a receber requisições feitas pelas chamadas *AJAX* e a responder com objetos *JSON* (OSMANI, 2016).

O *AJAX* teve seu potencial visto por grandes empresas como a Google, tornando-se assim um padrão para criação de aplicações web modernas e contribuindo para a grande evolução da linguagem *JavaScript*. A sua capacidade de atualizar apenas uma parte da página sem precisar recarregá-la, utilizando chamadas assíncronas, revolucionou a experiência dos usuários.

Mais tarde surgiram bibliotecas *JavaScript* para manipulação do *HTML* como *JQuery*. Novos padrões foram inseridos, como o *HTML5* e *CSS3*. Frameworks e bibliotecas MVC *JavaScript* foram criados, como o AngularJs (SESHADRI, 2014), Backbone.js (OSMANI, 2011) e Ember.Js (CRAVENS, 2014). Além de diversos outros artifícios, trazendo cada vez mais programadores para o novo cenário do desenvolvimento web front-end.

2.8 SPA (Single-page Applications)

Junto a esse novo mundo do desenvolvimento web *front-end*, vieram as *Single-page Applications (SPA)*, também conhecidas como *aplicações de página única*. A *SPA* consiste em uma técnica de desenvolvimento web que utiliza uma única página *HTML* como base para todas as outras páginas da aplicação. (FINK; FLATOW, 2016, p. 3).

Podemos ilustrar o ciclo de vida de uma SPA da seguinte forma:

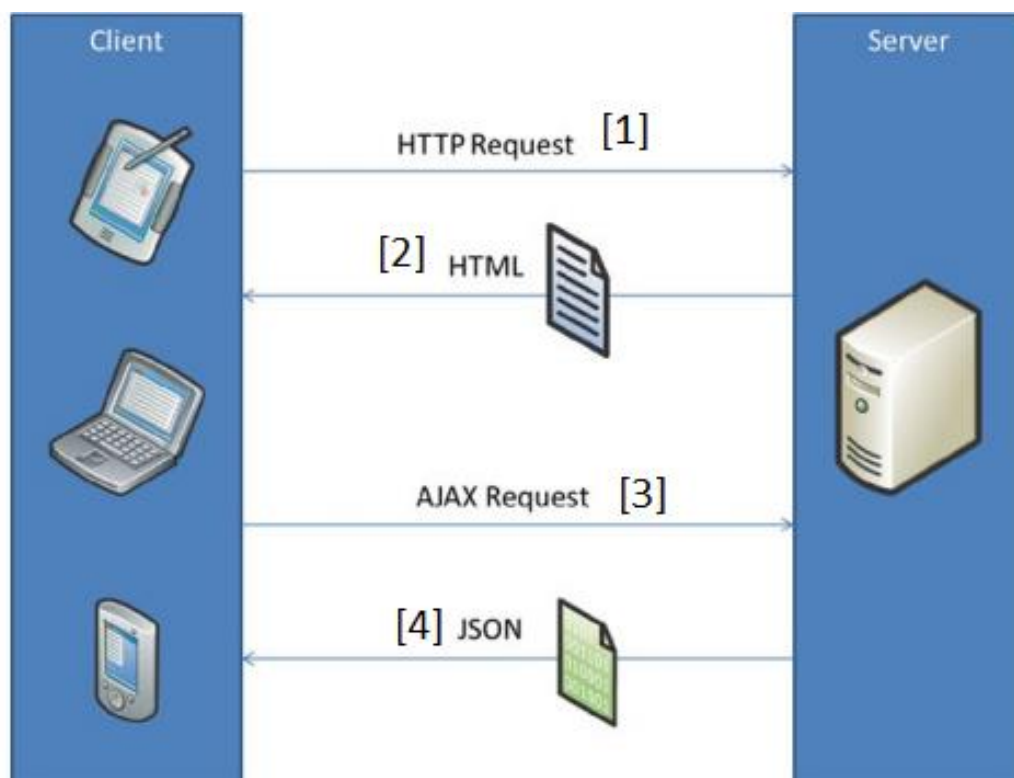


Figura 3 Ciclo de vida de uma SPA

Fonte: FINK; FLATOW, 2014

Este fluxo pode ser traduzido da seguinte maneira:

1. O cliente faz um requisição *HTTP* ao servidor.

2. O servidor retorna um documento *HTML* e todos os arquivos *JavaScript* e *CSS* que compõem a aplicação. Desta forma, este documento *HTML* é exibido pelo navegador e nunca é recarregado pelo servidor.
3. Todas as próximas interações do usuário com a aplicação realizarão requisições *AJAX* para o servidor.
4. O servidor, por sua vez, retornará todos os dados necessários via *JSON* ou partes de *HTML* pré-renderizados, que serão atualizadas na *DOM* (*Document Object Model*) e exibidas para o usuário.

A *SPA*, que faz parte do modelo moderno de desenvolvimento web, diminui de forma considerável o *overhead* (sobrecarga) de requisições ao servidor, fazendo com que a aplicação fique muito mais rápida. Ao mesmo tempo garante melhor usabilidade para o cliente, uma vez que as requisições *AJAX*, por serem assíncronas, não bloqueiam a interface do usuário. A maioria das *SPA* são desenvolvidas do lado do cliente (*client-side*), não necessitando a recarga constante das páginas inteiras pelo servidor.

2.9 O Padrão de Arquitetura MVC

O padrão de arquitetura MVC (BECK, 2010), Model-View-Controller, ou Modelo-Visão-Controlador, tem como objetivo a separação de responsabilidades e de unidades lógicas em aplicações de grande porte. Este padrão auxilia a decisão dos desenvolvedores sobre como dividir as responsabilidades, separando a aplicação em três partes modulares. São elas:

Modelo (*model*)

Os modelos são onde os dados da aplicação são armazenados, sendo geralmente acessados pelo servidor. Toda interface do usuário (*UI*) é derivada de um modelo ou de um conjunto deles.

Visão (*view*)

É através das visões que o usuário visualiza os dados e interage com a aplicação. Ela é dinâmica, sendo gerada de acordo com os modelos da aplicação.

Controlador (*controller*)

Os controladores são os tomadores de decisão, que conectam os modelos às visões. Representam, assim, a camada de apresentação da aplicação. Eles são responsáveis pela comunicação com o servidor para acessar e enviar os modelos, realizar a lógica de negócios, decidir que partes do modelo serão exibidas e realizar validações e tratamento de dados, dentre outras tarefas. Por serem responsáveis pela decisão de que partes dos modelos devem ser exibidas, também podem ser chamados de visões do modelo (*viewmodels*).

2.10 TDD - Test Driven Development

Test Driven Development, ou TDD, é um conjunto de técnicas associadas a métodos ágeis e *Extreme Programming* (XP). BECK (2000) afirma que o desenvolvimento orientado a testes tem como principal objetivo o desenvolvimento de “*Clean code that works*” (Código limpo que funciona). O TDD é um processo de desenvolvimento em que o primeiro passo sempre é a escrita de um teste que cubra alguma parte da especificação do sistema a ser desenvolvido. Esses passos garantem que o código terá uma boa cobertura de testes, além de fornecerem rápidos *feedbacks* sobre o código implementado, acerca de sua funcionalidade e design.

O *Test-driven development* obriga o desenvolvedor a planejar o que deve ser desenvolvido, já que é necessário traçar um objetivo e definir qual código será necessário para resolver o problema. Deste modo, um bom desenvolvimento orientado a testes nunca vai

conter código que não será executado, por exemplo. Além disso, ao utilizar o TDD, o desenvolvedor se concentra na criação de componentes com responsabilidades individuais, criando um código desacoplado e que honre com o *Single Responsibility Principle* (Princípio da Responsabilidade Única).

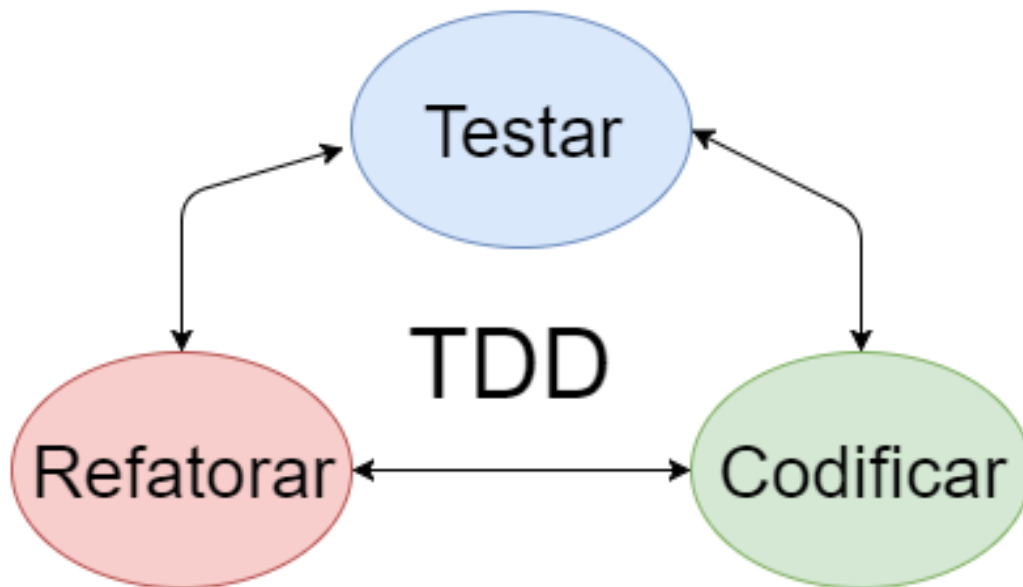


Figura 4 Fluxo do desenvolvimento orientado a testes (TDD)

O Processo de desenvolvimento

O TDD é um processo iterativo que consiste em 4 passos:

1 - Escrever um teste

O primeiro passo do desenvolvimento orientado a testes é selecionar uma *feature* (funcionalidade) que deve ser implementada e criar um teste unitário para ela. Um bom teste unitário deve ser enxuto e focado no comportamento de uma única função (ou método).

2 - Executar todos os testes (observar o novo teste falhar)

Há diversas razões para executarmos todos os testes antes de começar a codificar. A mais importante é garantir que o próprio teste foi codificado corretamente. Para isso verificamos se o teste está falhando, uma vez que a funcionalidade ainda não foi implementada. Caso o teste passe, ou o teste foi implementado de forma incorreta, ou o sistema já suporta o que iria ser adicionado.

3 - Fazer o teste passar

Uma vez que confirmamos que o teste falhou, podemos começar a implementação. Neste ponto, o TDD nos diz que devemos desenvolver a solução mais simples que nos garanta que o teste passe.

4 - Refatorar o teste

A última fase é a mais importante no que diz respeito a escrever um código limpo. Quando escrevemos a quantidade necessária para fazer com que o teste passe, é hora de revisarmos o que foi feito e realizarmos os ajustes necessários, seja arrumando a estrutura e design do código, removendo duplicações, aplicando padrões projeto etc.

Vantagens do TDD

- **Garante que o código funciona corretamente**

O maior benefício do TDD é a garantia de que todo código possui o comportamento esperado. Além disso, uma vez que os testes unitários cobrem cada uma uma pequena parte do

código, conseguimos encontrar mais facilmente onde o *bug* está, já que os testes que falharam nos apontarão qual parte específica do código que falhou.

Nagappan (2006) realizou um estudo de caso na Microsoft e na IBM e os resultados indicaram que o número de defeitos de quatro produtos diminuiu entre 40% e 90% em relação à projetos similares que não usaram TDD. Por outro lado, o mesmo estudo mostrou também que o tempo inicial de desenvolvimento obteve aumentos entre 15% e 35%.

- **Honra o princípio da responsabilidade única**

Quando desenvolvemos componentes isolados, garantimos um código de baixo acoplamento, honrando assim o princípio da responsabilidade única.

Em um de seus trabalhos, Steinberg (2001) apresentou que códigos produzidos com TDD são mais coesos e possuem menor acoplamento.

- **Força o desenvolvimento consciente**

O desenvolvimento orientado a testes força o desenvolvedor a pensar e planejar melhor o código que deve ser escrito antes de começar a codificar. Ao se pensar a fundo sobre o problema que precisa ser resolvido, aumentam as chances de que seja produzida uma solução mais sólida. Além disso, ao se descrever um caso de uso representativo para cada funcionalidade que será desenvolvida, o código tende a permanecer menor, sem a introdução de funcionalidades desnecessárias.

Langr (2010) apresentou em seu trabalho que o TDD aumenta a qualidade do código, a sua manutenibilidade e são produzidos cerca de 33% mais testes quando comparado a abordagens tradicionais.

3. Desenvolvimento de Aplicações Web *Front-End*

Neste capítulo serão apresentadas algumas das principais ferramentas e bibliotecas de código aberto utilizadas na criação de aplicações web *front-end* atualmente.

3.1 Frameworks e bibliotecas *JavaScript* MVC

O *JavaScript* evoluiu de uma linguagem de script, voltada para validações de dados simples, para uma linguagem de programação robusta, que atualmente é usada tanto no lado do cliente quanto no lado do servidor. Junto a isso, surgiram importantes bibliotecas *JavaScript*, como a *JQuery*, que oferecia uma *API* (SAHNI, 2016) estável, que garantia compatibilidade com diversos navegadores e uma fácil interação com a *DOM*.

Porém, com o aumento da complexidade e do tamanho das aplicações web, a *JQuery* já não conseguia mais oferecer um *framework* sólido que permitisse criar aplicações modulares, testáveis e bem estruturadas.

Com a finalidade de resolver estes problemas, surgiram os *frameworks* e bibliotecas MVC para *JavaScript*, disponibilizando uma camada sobre a *JQuery* (e, portanto, sobre a *DOM*). Estes frameworks e bibliotecas ajudam a reduzir drasticamente a quantidade de código *boilerplate* (repetido) e a organizar a aplicação de forma mais estruturada e de mais fácil manutenção. Utilizando um *framework* destes, os desenvolvedores podem começar a implementar de maneira adequada o projeto desde o seu início, sempre pensando no layout e arquitetura da aplicação. É possível, com eles, concentrar a atenção na aparência e nas funcionalidades da aplicação, reduzindo a preocupação com códigos redundantes.

3.1.1 Vantagens da utilização de frameworks e bibliotecas JavaScript MVC

Esses *frameworks* e bibliotecas permitem seguir conceitos que estão no centro do desenvolvimento das aplicações web atualmente, como por exemplo:

- Seguir o princípio da responsabilidade única, uma vez que conseguimos garantir a modularidade e a separação de responsabilidades de nossa aplicação, dividindo-a em partes pequenas e reutilizáveis.
- Desenvolver uma aplicação altamente testável, garantindo a integridade e funcionalidade de nossa aplicação.
- Utilizar *programação declarativa*, onde se diz “o que” deve ser feito, cabendo ao framework decidir como fazer, em oposição à *programação imperativa*, onde é necessário dizer “como” a tarefa deve ser realizada.
- Utilizar a *programação orientada a dados*, onde o objetivo é manipular o modelo e deixar com que o framework faça o trabalho de renderização da *UI*.

Veremos a seguir três dos *frameworks MVC JavaScript* mais famosos: *AngularJS*, *Backbone.js* e *Ember.js*.

3.1.2 AngularJS

O *AngularJS* (SESHADRI, 2014) foi criado em 2009 como parte de um produto comercial chamado *GetAngular*. Misko Hevery, um dos engenheiros que criou a ferramenta, reescreveu o *Google Feedback* utilizando este *framework*. Esta aplicação web, que possuía 18 mil linhas de código e que demorou 6 meses para ser desenvolvida, foi reescrita em apenas 3 semanas usando *GetAngular*, possuindo por volta de 1500 linhas.

A Google então viu o potencial do então *GetAngular* e começou a patrocinar o projeto, tornando o *AngularJS* uma aplicação de código-aberto. O *AngularJS* foi lançado possuindo

importantes inovações, como o *two-way data binding* (conceito que será explicado mais à frente), injeção de dependência, código facilmente testável e extensão do *HTML* por meio de diretivas.

Modelo arquitetural

As aplicações criadas com *AngularJS*, em sua grande maioria, se organizam da forma ilustrada na Figura 5:

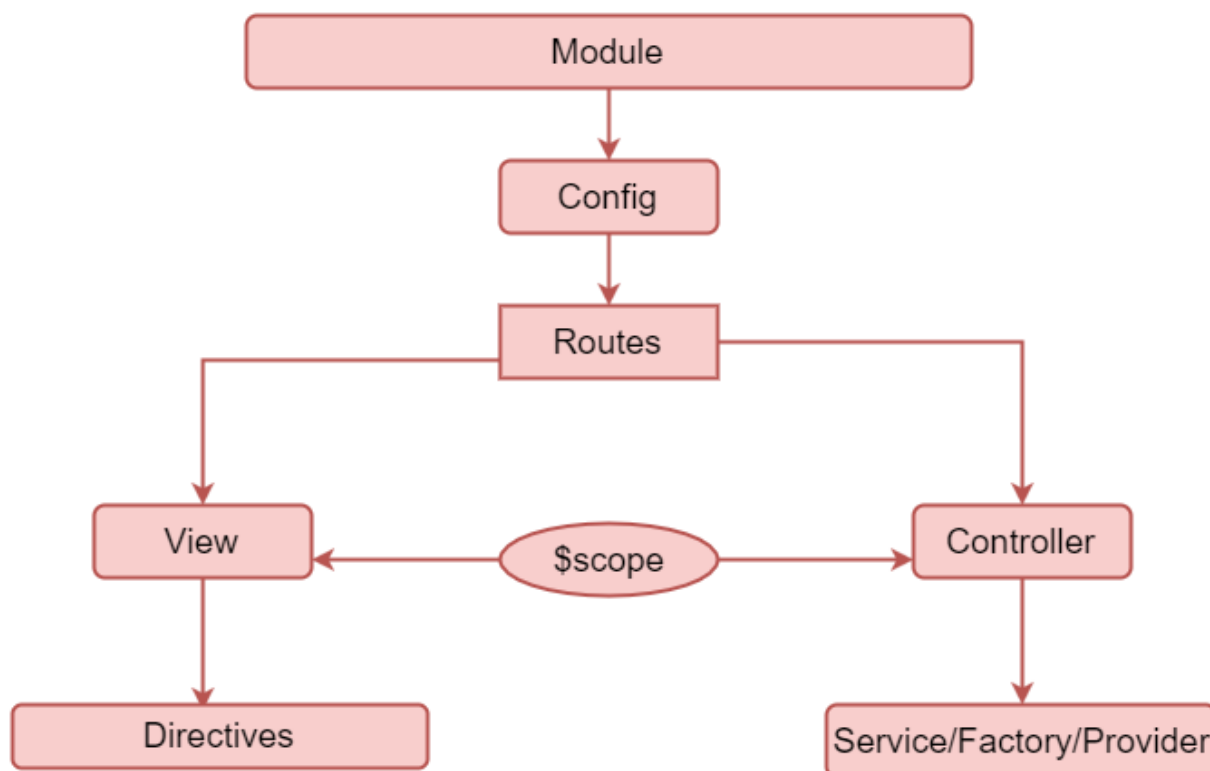


Figura 5 Arquitetura de aplicações AngularJS

A arquitetura do AngularJS tem como pilar a criação de **Modules** (módulos), que podem trabalhar de forma separada ou conjunta. Os módulos são responsáveis pela inicialização da aplicação, além de atuar como um *container* para diferentes partes da aplicação, como *views* (visões), diretivas, serviços, dentre outros. A ideia é que cada módulo tenha suas próprias configurações, rotas e lista de dependências.

Quando um usuário acessa determinada *URL* da aplicação, a rota controlada pelo módulo carrega na tela o *template* da *view* e o seu respectivo controlador. A partir daí o usuário pode manipular os *templates*, sendo tarefa do controlador realizar as trocas de informações necessárias com o servidor. Este fluxo é ilustrado na Figura 6:

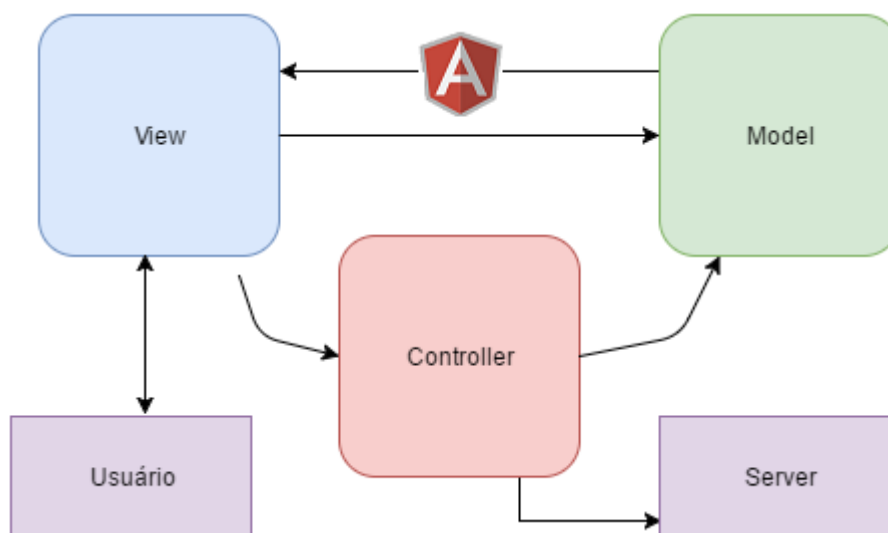


Figura 6 Fluxo de dados em uma aplicação AngularJS

O usuário interage com a **View** (visão). Esta, por sua vez, está vinculada ao controlador através das diretivas do *AngularJS*, e se encontra dentro de um *template* que é carregado na inicialização da aplicação. Quando alguma ação do usuário necessita da troca de informação com o servidor, quem o faz é o **Controller** (controlador). Para isso o controlador utiliza algum **Service** (serviço) do *AngularJS*, como por exemplo *\$http*, que abstrai a interação com o **Server** (servidor). A resposta do servidor, na maioria das vezes no formato *JSON*, é espelhada para o modelo e renderizada na visão através da técnica de *two-way data binding*. (STROPEK,2013)

3.1.3 Backbone.js

O *Backbone.js* é uma biblioteca *JavaScript* leve que dá estrutura à aplicação *front-end*, uma vez que segue o padrão de arquitetura MV* (semelhante ao MVC, porém não possui um controlador bem definido). Com ele é fácil gerenciar e desacoplar responsabilidades da aplicação. Isso torna o código mais sustentável ao longo do tempo.

O *Backbone.js* oferece métodos que facilitam a consulta e manipulação de dados, abstraindo códigos *boilerplate* (repetido) de *JavaScript*. Necessita de outras bibliotecas e *frameworks* para seu funcionamento completo, como o *Underscore.js* (que será abordado na próxima sessão), *Zepto* ou *jQuery*.

Modelo arquitetural

A Figura 7 mostra a manipulação de uma requisição típica realizada para o *framework* Backbone.js.

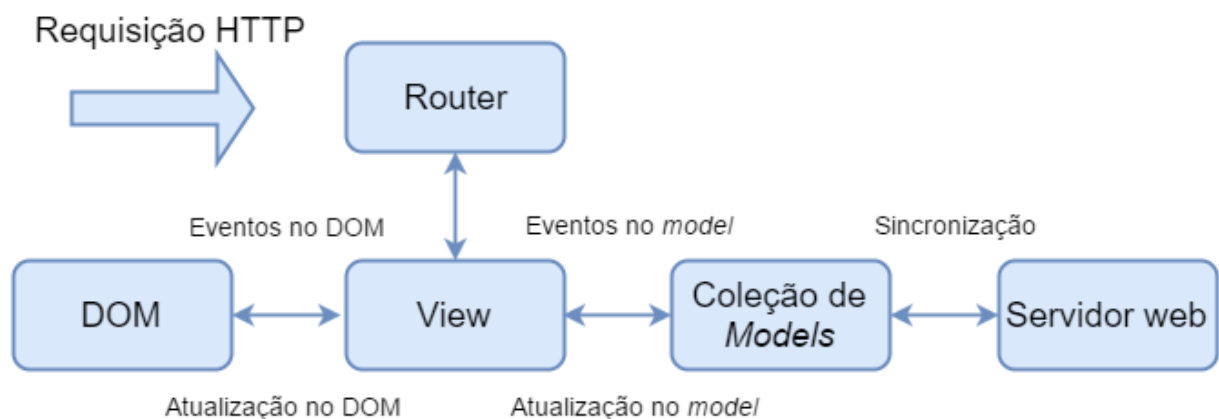


Figura 7 Fluxo de uma aplicação desenvolvida com Backbone.js

A *SPA* é carregada através de uma solicitação *HTTP* ao servidor. De acordo com a interação do usuário o **Router** intercepta as *URLs* e chama as lógicas presentes no *client-side* (lado do cliente), em vez de enviar uma nova requisição ao servidor.

O roteamento *URL*, os eventos da *DOM* (como cliques do mouse) e eventos no *Model* (modelo), são os gatilhos para a manipulação lógica na *View* (visão). Essas manipulações lógicas, que representam as regras de negócio e apresentação, são responsáveis por atualizar as visões e os modelos.

Um modelo pode ter diversas visões o observando. Isso quer dizer que as visões registram interesse em serem avisadas sempre quando um modelo sofrer alguma alteração. Isso garante que o que está sendo exibido na tela está sincronizado com o estado atual do modelo. Para o modelo não faz diferença se uma ou mais visões o estão observando, ele simplesmente anuncia quando algum dado seu é alterado.

No *Backbone.js* os modelos são agrupados no que são chamadas de *Collections* (coleções). Agrupar modelos permite desenvolver lógicas de aplicação baseadas em notificações de um grupo, mesmo quando apenas um modelo que integra esse grupo sofre alteração. Isso evita ter que observar cada modelo individualmente.

O usuário interage com as visões, lendo ou editando as informações dos modelos. Para isto nós definimos a utilidade *render()* na nossa visão, que será responsável por renderizar o conteúdo do modelo utilizado. Para isso o *Backbone.js* utiliza o *Underscore.js*, que é um motor *JavaScript* de *templates*. Feito isso, assinamos nosso *render()* a um modelo, fazendo com que a visão seja acionada sempre que este sofrer alguma alteração. (OSMANI, 2013)

Quando o usuário interage com a tela, como por exemplo clicando em algum elemento, não é responsabilidade da visão saber o que deve ser feito em seguida, e sim do controlador. No *Backbone.js* não há um controlador bem definido. Alcança-se isto adicionando um *event listener* (ouvinte de eventos) para o elemento, que irá delegar a manipulação do clique para um *event handler* (manipulador de eventos). (OSMANI, 2013)

3.1.4 Ember.js

O *Ember.js* é um *framework* de código aberto para criação de aplicações web do lado do cliente. Tem como foco a criação de *Single-page Applications* (SPA) e utiliza o padrão de arquitetura *MVC*.

Este *framework* elimina de maneira expressiva a quantidade de código *boilerplate* (repetido) e promove uma estrutura sólida de arquitetura de aplicação.

Modelo arquitetural

A Figura 8 ilustra como as **Routes** (rotas), **Controllers** (controladores), **Views** (visões), *templates* e **Models** (modelos) interagem.

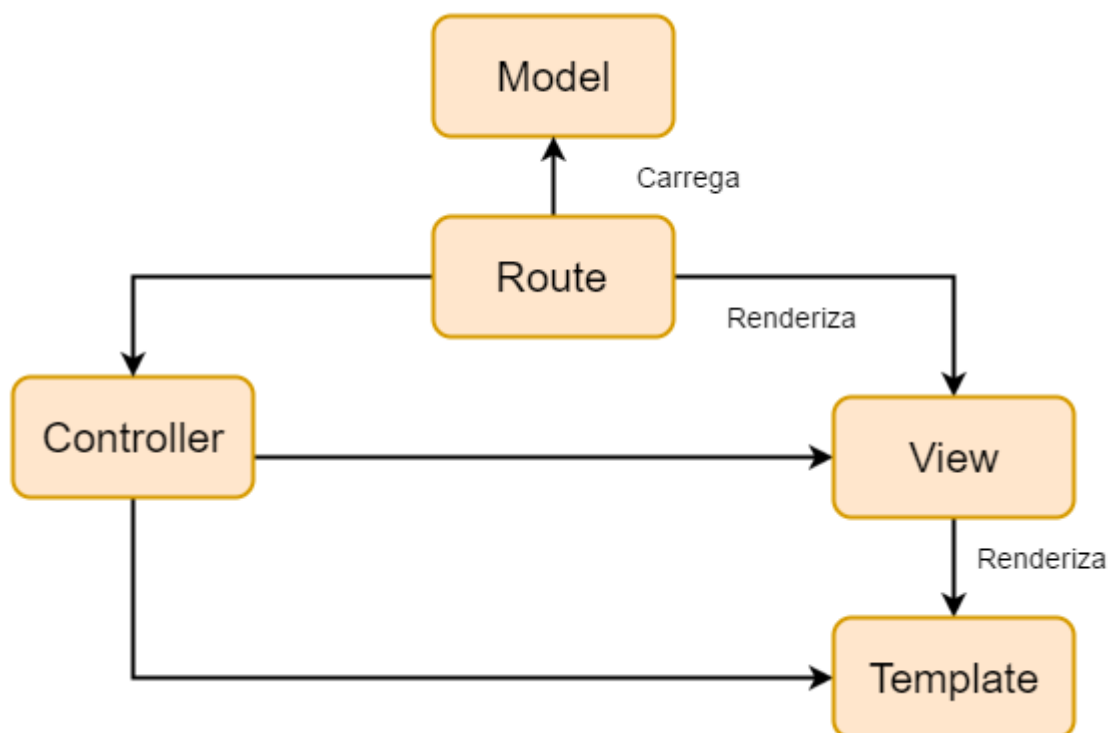


Figura 8 Fluxo de uma aplicação desenvolvida em Ember.js

Os **Models** (modelos), que são onde os dados ficam armazenados, são inseridos ou atualizados através das **Routes** (rotas), seja utilizando chamadas *AJAX* ou utilizando o *Ember-Data*, que é uma biblioteca acoplada ao *Ember.js* para facilitar a comunicação com servidores *RESTful* (REST APIs).

O **Route** (rota) representa as rotas da aplicação. As rotas são representações *URL* de objetos da nossa aplicação. As rotas têm como objetivos consultar os modelos e torná-los disponíveis para os controladores e *templates*. São responsáveis também por definir propriedades nos controladores, executar ações ou conectar um determinado *template* a um controlador.

Um **Controller** (controlador) recebe um modelo através de uma rota. É responsável por realizar a conexão entre o modelo e a visão (ou *template*). É nele que estão presentes as lógicas de negócio da aplicação.

As **Views** (visões) são responsáveis pela apresentação dos dados ao usuário. Toda visão é associada a um controlador, a um *template* e a uma rota.

O **Template** é uma marcação no *HTML* das visões. Ele é responsável por exibir na tela as informações do modelo e automaticamente se atualizar quando o modelo mudar. O *Ember.js* utiliza o *Handlebars*, que é um motor de *templates* mantido pelo próprio time *Ember*. Ele possui *template* de lógicas, como *if* e *loops*, além de *helpers* para formatação da apresentação dos dados

3.1.5 - Estudo comparativo

Escolher um *framework* ou biblioteca para um determinado projeto tem um grande impacto no tempo de entrega do produto e na manutenibilidade do mesmo. Deve-se buscar um *framework* sólido e estável, que nos garanta as funcionalidades necessárias.

Todas as três ferramentas apresentadas são de código aberto e lançados sob a licença MIT, que permite a reutilização do *software* licenciado em programas livres ou proprietários. Foram desenvolvidas com o objetivo de resolver o problema de criação de *Single-page Applications* (aplicações de página única) utilizando o padrão de arquitetura MVC (ou MV*, como é o caso do Backbone). Todas as três apresentam conceitos de visão, modelos, eventos e roteamento.

Comunidade

A comunidade é um importante fator a ser considerado quando escolhemos um framework de código aberto. Uma grande comunidade nos garante mais questões solucionadas, mais material criado sobre o assunto, mais módulos de terceiros, mais tutoriais, etc.

Métrica	AngularJS	Backbone.js	Ember.js
Stars no Github	51,763	25,423	16,749
Módulos de terceiros	1,981	275	2,800
Perguntas no StackOverflow	194,741	28,008	26,509
Resultados no youtube	176,000	27,700	27,400
Contribuidores no Github	1,512	290	613
Issues abertas no Github	690	39	183
Issues fechadas no Github	7,314	2,232	4,590

Tabela 1 Dados gerais a respeito da comunidade das ferramentas. (As consultas foram realizadas em 27 de Agosto de 2016)

Analisando estes dados fica claro que o *AngularJS* é o *framework* mais ativo entre os três, uma vez que é o terceiro projeto com mais *Stars* no Github (indica o número de usuários interessados no projeto), possui mais contribuidores e consideravelmente mais resultados no *StackOverflow* e no *Youtube* que as outras duas ferramentas.

Os contribuidores são as pessoas que contribuem para o código do projeto. Mais usuários do Github contribuindo para o projeto indica que há mais pessoas revisando os códigos, mais pessoas desenvolvendo para a melhoria da ferramenta e menor a chance de o projeto ser abandonado.

Vale ressaltar que, para a quantidade de contribuidores ter a sua devida relevância, o projeto deve ter suas devidas políticas para *commits* do código-fonte e lançamento de atualizações. Além disso, devem possuir certos requerimentos e instruções que sirvam para controlar o número e a qualidade dos desenvolvedores que contribuem para o projeto.

Além dessas métricas, na Figura X podemos analisar o avanço da popularidade do *AngularJS* comparado ao *Ember.js* e ao *Backbone.js*. Esta consulta, realizada na ferramenta *Google Trends*, nos mostra o nível de interesse dos usuários de acordo com as buscas realizadas na plataforma de pesquisa da Google e outros fatores.

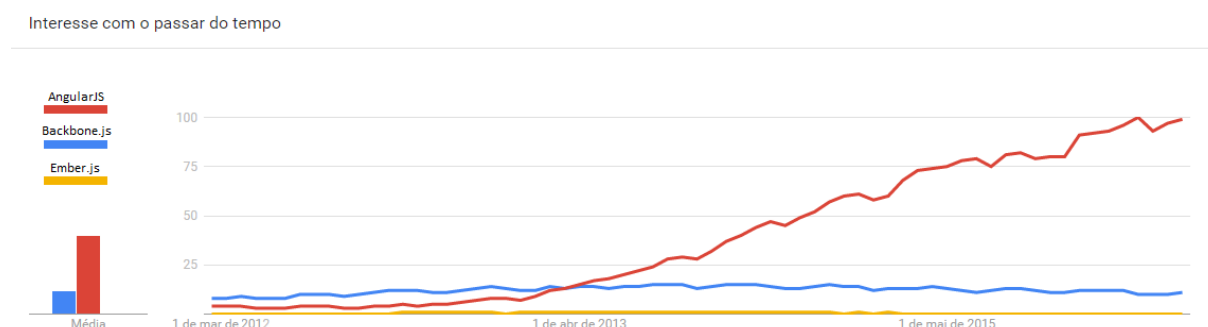


Figura 9 Interesse de pesquisa sobre as aplicações ao decorrer do tempo.

De acordo com o site do Google Trends (<https://www.google.com.br/trends/>) os números representam o interesse de pesquisa relativo ao ponto mais alto no gráfico de uma determinada região em um dado período. Um valor de 100 é o pico de popularidade de um termo. Um valor de 50 significa que o termo teve metade da popularidade. Da mesma forma, uma pontuação de 0 significa que o termo teve menos de 1% da popularidade que o pico.

Tamanho das ferramentas

Estudos (NAH, 2004) comprovam que o tempo de carregamento é uma característica crucial para o sucesso de uma aplicação web. Existem dois fatores importantes a serem considerados quando avaliamos o impacto de um *framework* (ou biblioteca) no tempo de carregamento de uma página: o tamanho dele e o tempo que este leva para inicializar.

Os arquivos *JavaScript* são geralmente fornecidos minimizados e comprimidos. Faremos uma comparação entre as 3 ferramentas, lavando em consideração não só o tamanho delas como também o de suas dependências.

Framework (ou biblioteca)	Tamanho	Tamanho com as dependências necessárias
AngularJS 1.2.22	39.5kb	39.5kb
Backbone.js 1.1.2	6.5kb	43.5kb (jQuery + Underscore) 20.6kb(Zepto + Underscore)
Ember.js 1.6.1	90kb	136.2kb (jQuery + Handlebars)

Tabela 2 Tabela que compara o tamanho em Kilobytes dos frameworks AngularJS, Backbone.js, Ember.js

Templating

O AngularJS e o Ember.js incluem um motor de templates. Já o Backbone.js, nos deixa livre para utilizar o motor de *templates* de nossa escolha.

Para o melhor entendimento do que é um *template* e futura comparação, utilizaremos um exemplo de formatação de uma lista de objetos no HTML. Neste exemplo iremos exibir o nome e telefone de todos os objetos *pessoa* contidos dentro de uma *lista de pessoas*.

AngularJS

O motor de *templates* do Angular consiste simplesmente em adicionar *bindings expressions* (expressões de ligação) em nosso HTML, como podemos ver no exemplo a seguir:

```
1. <ul>
2.   <li ng-repeat="pessoa in pessoas" title="{{pessoa.nome}}">
3.       {{pessoa.telefone}}
4.   </li>
5. </ul>
```

Backbone.js

Apesar de o Backbone.js poder se integrar com diversos motores de *templates*, o padrão é o Underscore.js. Por outro lado, o Underscore é uma ferramenta muito básica, havendo a necessidade de se implementar código Javascript no *template*.

```
1. <ul>
2.   <% _.each(pessoas, function(pessoas) { %>
3.       <li title="<%- pessoa.nome%>">
4.           <%- pesssoa.telefone %>
5.       </li>
6.   <% }); %>
7. </ul>
```

Ember.js

O *Ember.js* atualmente utiliza o motor de *templates* **Handlebars**, que é uma extensão de um famoso motor de *templates* chamado *Mustache*.

Seguem um exemplo:

```
1. {{#each m in model}}
2.   <tr>
3.     <td>
4.       <a href="{{m.links.[0].uri}}">{{m.name}}</a>
5.     </td>
6.   </tr>
7. {{/each}}
```

Segurança

Todas as 3 ferramentas tratam os problemas de segurança nas aplicações do lado do cliente, porém de maneiras diferentes e com diferentes *trade-offs*. Abordaremos então a questão da segurança entre os três.

	AngularJS	Backbone.js	Ember.js
Dependências	Não possui	Underscore.js e jQuery	jQuery e Handlebars
CVEs	0	0	5
Retire.js	5	1	15
Política de Segurança	Não	Não	Sim
Contato de Segurança	Sim	Não	Sim
Documentação de Segurança	Sim	Não	Não

Features de Segurança	Sim	Não	Não
-----------------------	-----	-----	-----

Tabela 3 Comparação entre aspectos relacionados a segurança das aplicações

CVE, *Commom Vulnerabilities and Exposures*, é uma base de dados internacional e pública para troca de informações entre produtos sobre falhas de segurança.

Retire.js é uma ferramenta que escaneia aplicações web ou aplicações em Node.js buscando suas vulnerabilidades.

AngularJS

O AngularJS atualmente não possui nenhuma *CVE* associada ao seu nome. *Retire.js* tem recomendação para atualizar versões diferentes baseado em vulnerabilidades encontradas. O Angular possui uma página de documentação específica para manter o código seguro e possui um contato específico para questões de segurança, porém não oferece política de segurança.

O *AngularJS* implementa sua própria versão de expressões *JavaScript*, sendo restritas e executadas em um *sandbox*, chamado *Sandbox Expression*. O *sandbox* é utilizado para manter uma devida separação de responsabilidades na aplicação. Porém isso não aumenta muito a segurança, uma vez que, por meio das *sandboxes*, podem ser realizados ataques *XXS* na aplicação.

O ataque de Cross-site scripting (*XSS*) consiste em uma vulnerabilidade causada pela falha nas validações dos parâmetros de entrada do usuário e resposta do servidor na aplicação web. Este ataque permite que código *HTML* seja inserido de maneira arbitrária no navegador do usuário alvo.

Backbone.js

O *Backbone.js* também não possui nenhum *CVE* registrado em seu nome. *Retire.js* tem apenas uma versão a ser atualizada para a biblioteca.

O *Backbone.js* não possui política de segurança e nem documentação a respeito disso. Ele possui dependências como o *Underscore.js* e o *Jquery*. O *JQuery* tem diversas *CVEs* em seu nome.

No *Backbone.js* não existe o conceito de *expression sandbox* por seu escopo ser muito simples. As expressões em *JavaScript* ficam por conta do desenvolvedor e se localizam nos *templates*.

Ember.js

O Ember possui cinco *CVEs* em seu nome: *CVE-2013-4170*, *CVE-2014-0013*, *CVE-2014-0014*, *CVE-2014-0046*, *CVE-2015-1866*.

Retire.js tem quinze versões a serem atualizadas. O *Ember.js* por sua vez possui política de segurança e contato para este fim. Também não utiliza *Sandbox Expressions*.

Estudo de mercado

Veremos na tabela a seguir a relação dos três *frameworks* no que diz respeito ao ano que foram introduzidos, suas origens, seus principais contribuidores e uma visão geral dos sites mais populares que os utilizam:

	Introdução	Origem/Base	Principal contribuidor	Sites mais populares
Ember.js	2007	SpoutCore / Handlebars	Yehuda Katz	Yahoo, Groupon
Angular.js	2009	GetAngular	Brat Tech LLC, Google	Youtube (PS3), Sky Store
Backbone.js	2010	Underscore.js	Jeremy Ashkenas	Wordpress, NYTimes

Tabela 4 Comparação entre os três frameworks no que diz respeito ao seu histórico a mantenedores

As três ferramentas são utilizadas ou mantidas por empresas de grande reconhecimento.

Utilizando a ferramenta chamada *SimilarTech* (<https://www.similartech.com>), podemos fazer uma análise sobre a quantidade de sites que utilizam as três ferramentas.

Tivemos os seguintes resultados:

- AngularJS: 344,072 sites
- Backbone.js: 233,451 sites
- Ember.js: 1,891 sites

Esta ferramenta não fornece a quantidade absoluta de sites que utilizam a aplicação, mas dá uma noção da estimativa média. Somando o total de sites utilizados pelas 3 ferramentas, obtém-se o gráfico apresentado na Figura 10.

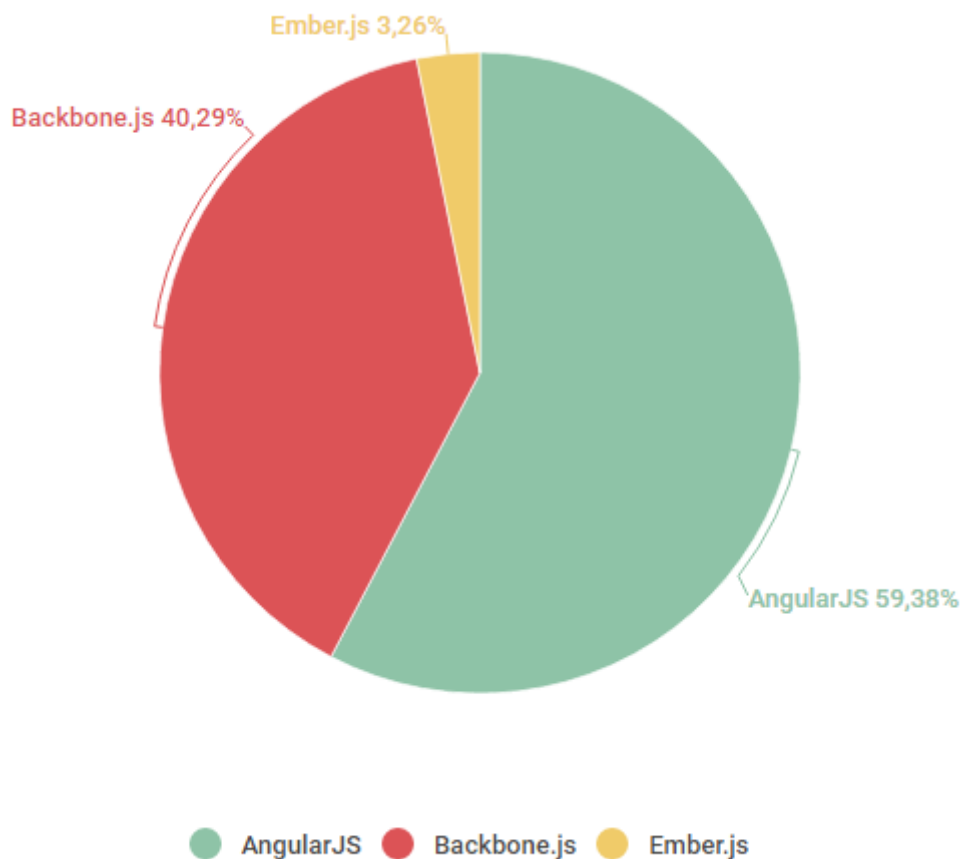


Figura 10 Gráfico que relaciona a porcentagem de sites que utilizam as três ferramentas

Somando todos os sites que utilizam as três ferramentas, temos que o AngularJS está presente em 59,38% delas. Em segundo lugar vem o Backbone.js, com 40,29% e por último o Ember.js com apenas 3.26%.

Conclusão

O *AngularJS* se sobressai sobre o *Ember.js* e o *Backbone.js* em diversos aspectos e por isso foi a ferramenta escolhida para o aplicativo de exemplo apresentado neste trabalho. Um dos principais fatores é a sua popularidade crescente. Ele possui uma comunidade extremamente ativa, o que no mundo de desenvolvimento de aplicações é extremamente importante. O número de interessados e de contribuidores é um grande fator que contribui para a evolução da plataforma.

Além disso, além de mais nova que o Backbone.js, o AngularJS possui mais sites que o utilizam na web, comprovando-se como tendência.

3.2 Pré-processadores de CSS

Nos últimos anos, devido ao aumento de dispositivos com os mais variados tamanhos de tela (celulares, tablets etc.), que exigem um design mais arrojado que os websites, a complexidade dos requisitos e necessidades a respeito do CSS vem crescendo.

Hoje em dia, os profissionais front-end têm que lidar com folhas de estilo cada vez mais complexas e extensas, tornando o desenvolvimento mais lento e o sistema mais difícil de dar manutenção.

Devido a isso, foram criados os pré-processadores de CSS. Eles possuem uma linguagem bem próxima do CSS, porém com o uso de variáveis, funções, importação de código, dentre outras.

Uma vez que se tenha o CSS dominado, há várias vantagens para o uso de pré-processadores. Um dos principais motivos é a aplicação do conceito DRY (Don't repeat yourself ou Não repita a si mesmo). Pode-se citar outras vantagens, como:

- Código limpo, com variáveis e pedaços reutilizáveis.
- Código fácil de manter com *snippets* e bibliotecas
- Utilização de cálculos e lógicas
- Código mais organizado e de fácil configuração

Nesta seção serão abordados os dois pré-processadores mais utilizados no mercado: LESS e Sass.

3.2.1 - Sass

O Syntactically Awesome Stylesheets, conhecido como Sass (POPLADE, 2013) é uma linguagem de folhas de estilo dinâmica de código aberto que foi inicialmente desenvolvida por Hempton Catlin, Natalie Wezenbaum e Chris Eppstein, utilizando a linguagem de programação *Ruby* (HARTL, 2014).

O Sass possui dois tipos de sintaxe. A sintaxe original é a “sintaxe indentada”, que como o nome já diz, utiliza indentação para separar blocos e caracteres *newline* para separar regras. A segunda e mais nova sintaxe é a “SCSS”, que usa formatação de blocos assim como o CSS, utilizando chaves para separar blocos de código e ponto e vírgula para separar linhas em um bloco.

3.2.2 - LESS

O LESS (Lopes, 2012) é uma linguagem de folha de estilo dinâmica de código aberto criada por Alexis Sellier e Dmitry Fadeyev. Foi inicialmente desenvolvida em Ruby (HARTL, 2014) , porém mais tarde teve seu código substituído por *JavaScript*.

A sintaxe do LESS é uma metalinguagem aninhada, logo, um código válido em CSS também é válido em LESS. Esta linguagem influenciou a criação da nova sintaxe do “SCSS” do Sass.

3.2.3 - Aspectos das linguagens

SAAS e LESS são ambas poderosas extensões de CSS. Pode-se pensar nelas como linguagens de programação projetadas para manter o CSS mais sustentável, personalizável e extensível. Tanto o Sass quanto o LESS são compatíveis com versões anteriores, bastando alterar o arquivo de extensão *.css* para *.scss* ou *.less*, respectivamente.

Instalação

Será apresentado como realizar a instalação pela linha de comando:

Sass

Para rodar os Sass, precisaremos do *Ruby* (<http://rubyinstaller.org/downloads/>) instalado. As novas versões do Linux e OSx já vem com o Ruby pré-instalado. Com o Ruby instalado, pode-se utilizar o seguinte comando para instalação: `sudo gem install sass`

Less

Como o LESS é escrito em *JavaScript*, necessita-se do NodeJS (<https://nodejs.org/en/download/>) instalado para rodá-lo.

No linux pode-se realizar a instalação do LESS executando o seguinte instrução na linha de comando: `npm install -g less`

No Windows, após a instalação do NodeJS installer, deve-se executar a seguinte instrução na linha de comando: `npm install less`

Editores de texto - Realce da Sintaxe

Após a instalação é interessante que se utilize alguma extensão em seu editor de textos para o realce da sintaxe. Segue uma tabela com os *plug-ins* mais populares:

Editor de Texto	Sass	Less
SublimeText	<i>Sass Bundle</i>	<i>Less-sublime</i>
Notepad++	<i>Notepad-plus-plus</i>	<i>Less-for-Notepad-plusplus</i>
VisualStudio	<i>SassyStudio</i>	<i>CSS is Less</i>
TextMate	<i>SCSS.tmbundle</i>	<i>Less.tmbundle</i>
Eclipse	<i>Eclipse platform</i>	<i>Eclipse for Less</i>
Coda	<i>Coda Sass</i>	<i>Coda 2 Less</i>

Tabela 5 Plug-ins de sintaxe para editores de texto, para Sass e LESS

Variáveis e operações

Um das principais vantagens na utilização de pré-processadores de CSS é o uso de variáveis. Nelas pode-se armazenar qualquer valor para ser reutilizado, como fontes e cores ,por exemplo. Um outro artifício interessante é o uso de operações matemáticas com variáveis e constantes.

As variáveis são declaradas de forma semelhante em ambas. Em Sass é utilizado \$ antes do nome da variável, em LESS é utilizado @.

Exemplo em LESS:

```
1. @danger-red: #FF0000;
2. @light-green: @first-blue + #111;
3. #header {
4.   color: @light-green;
5. }
```

Saída em CSS:

```
1. #header {
2.   color: #1FE9966;
3. }
```

Mixins

Mixins são utilizados para agruparmos propriedades ou declarações.

Sass

Em Sass é utilizado @*mixin* para definir o agrupamento e @*include* para incluí-lo.

Segue um exemplo para a definição para os raios de uma borda (border-radius):

```
1. @mixin border-radius($radius) {
2.   -ms-border-radius: $radius;
3.   -border-radius: $radius;
```

```
4. -webkit-border-radius: $radius;
5. -moz-border-radius: $radius;
6. }
7. .box { @include border-radius(20px); }
```

Less

Em Less basta que se defina a classe e a encapsulamos no bloco desejado. Segue um exemplo:

```
1. .bordered {
2.   border-top: dotted 1px black;
3.   border-bottom: solid 2px black;
4. }
5. #side-menu a {
6.   color: #111;
7.   .bordered;
8. }
```

Saída em CSS:

```
1. #side-menu a {
2.   color: #111;
3.   border-top: dotted 1px black;
4.   border-bottom: solid 2px black;
5. }
```

Nesting

Nesting é uma grande vantagem dos pré-processadores, uma vez que criam uma hierarquia visual, similar à encontrada no HTML. Com os exemplos fica claro como há uma menor repetição de classes e *divs* quando utilizada esta abordagem em cascata.

O *Nesting* é realizado de forma semelhante em ambos.

Exemplo para Sass e Less:

```
1. nav {
2.   ul {
3.     margin: 0;
4.     padding: 0;
5.     list-style: none;
6.   }
7.   li { display: inline-block; }
8.   a {
9.     display: block;
10.    padding: 6px 12px;
11.    text-decoration: none;
12.  }
13. }
```

Saída em CSS:

```
1. nav ul {
2.   margin: 0;
3.   padding: 0;
4.   list-style: none;
5. }
6. nav li {
7.   display: inline-block;
8. }
9. nav a {
```

```
10. display: block;
11. padding: 6px 12px;
12. text-decoration: none;
13. }
```

Importação

O CSS possui uma opção de importação que possibilita dividir as folhas de estilo em arquivos menores e em porções mais sustentáveis. O problema disso é que cada vez que se utiliza *@import* em nosso CSS, cria-se uma nova requisição HTTP. O Sass e o LESS também utilizam o *@import*, porém trabalham de maneira diferente. Em vez de criar novas requisições HTTP, eles combinam um arquivo a outro, similar a uma concatenação, entregando apenas um arquivo CSS ao browser.

O Sass e o LESS trabalham de forma semelhante, segue um exemplo de importação:

Exemplo para Sass e LESS

Supondo que temos um arquivo *visual_one.scss*. Em nosso arquivo *base.scss* basta incluirmos no topo do arquivo:

```
1. nav a {
2.   display: block;
3. }
4. @import 'visual_one.css';
5. nav a {
6.   display: block;
7. }
```

OBS.: Com o Sass a extensão não é necessária. Já com o LESS, a extensão só não é necessária para arquivos *.less*.

Extend (Herança)

Um outro caso a ser abordado é quando uma classe deve possuir todos os estilos de outra classe, além de seus próprios estilos. Isso é chamado de herança. Está é uma característica muito interessante dos LESS e do Sass. Utilizando herança, o CSS torna-se muito menos repetitivo, respeitando o conceito DRY (Don't repeat yourself, ou Não repita a si mesmo).

A herança é utilizada da mesma forma tanto em Sass quanto em LESS, porém com sintaxes diferentes. Em Sass utilizamos `@extend` e em LESS utilizamos `:extend`.

Exemplo em Sass:

```
1. .message {
2.   border: 1px solid #ccc;
3.   padding: 10px;
4.   Color: #333;
5. }
6. .success {
7.   @extend .message;
8.   border-color: green;
9. }
10. .error {
11.   @extend .message;
12.   border-color: red;
13. }
```

Saída em CSS:

```
1. .message, .success, .error, {
2.   border: 1px solid #cccccc;
3.   padding: 10px;
4.   color: #333;
```

```
5.  }  
6.  .success {  
7.    border-color: green;  
8.  }  
9.  .error {  
10.   border-color: red;  
11. }
```

OBS.: Note a diferença entre *Mixins* e *Extends*. Em *Mixins* o código é copiado de uma classe para a outra no arquivo de saída CSS, já o *Extends* realiza a herança de classes na saída CSS.

3.2.4 Estudo Comparativo

Sintaxe e features

Como foi analisado na seção anterior, ambos possuem a sintaxe muito semelhante.

Documentação

Ambas as documentações são robustas. A documentação do LESS é mais orientada a código, possui muitos exemplos a serem explorados e explicações claras e objetivas quando necessárias. Já a documentação do Sass é mais orientada a textos, porém também completa em relação a exemplos.

Curvas de aprendizado

Os dois pré-processadores possuem sintaxe simples e vasta documentação, fazendo com que a curva de aprendizado de ambas seja semelhante.

Frameworks e bibliotecas

O Sass destaca-se neste aspecto, uma vez que possui o *Compass* e o *Bourbon*, que são um framework e uma biblioteca, respectivamente. Ambos são bastante usados no mercado, por facilitarem o desenvolvimento com os Sass.

Segue uma breve explicação sobre eles:

- **Compass:** Se descreve como um um *Framework de autoria para Sass*. É mantido por Chris Eppstein (um dos dois mantenedores do Sass). Este *framework* fornece diversas ferramentas e *mixins* para o Sass.
- **Bourbon:** É uma biblioteca que disponibiliza diversos *Mixins* para o Sass.

Comunidade

Google trends

Ao analisarmos o interesse com o passar do tempo pelo *Google Trends*, que avalia a procura pelo termo na plataforma de busca *Google*, vemos que o Sass é amplamente mais procurado que o Less.

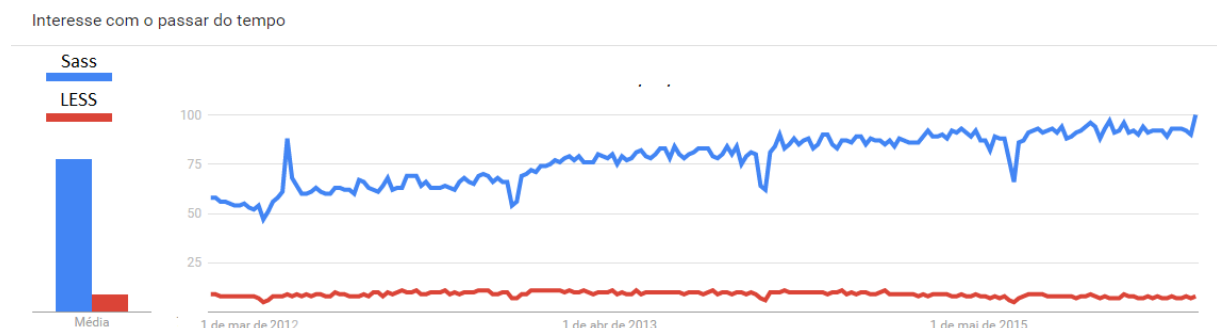


Tabela 6 Interesse de pesquisa sobre as LESS e Sass no decorrer do tempo

Número de Downloads

Outro aspecto interessante de ser analisado é a quantidade de downloads de cada plataforma. De acordo com o site do npm (<https://www.npmjs.com/>), o LESS teve um total de 30,375,332 downloads. Já o Sass, pelo site do Ruby (<https://www.ruby-lang.org/pt/>), teve um total de 67.723.847. (Ambas as consultas foram realizadas em 27 de Agosto de 2016)

Número de perguntas na plataforma StackOverflow

Analisando o número de perguntas na plataforma StackOverflow, que é um dos maiores sites de perguntas e respostas relacionadas à programação, existe um total de 27,432 perguntas relacionadas ao LESS e 38,024 relacionadas ao Sass. (Ambas as consultas foram realizadas em 27 de Agosto de 2016)

Repositório GitHub

Analisando o repositório GitHub de cada um, o Sass possui *8152 Stars* (indica o número de usuários interessados no projeto) e *177 Contribuidores*. Já o LESS possui *13921 Stars* e *211 Contribuidores*. Isto mostra que o projeto do LESS está um pouco mais ativo que o do SASS. (Ambas as consultas foram realizadas em 27 de Agosto de 2016)

Os contribuidores são as pessoas que contribuem para o código do projeto. Mais usuários do GitHub contribuindo para o projeto indica que há mais pessoas revisando os códigos, mais pessoas desenvolvendo para a melhoria da ferramenta e menor a chance de o projeto ser abandonado.

Vale ressaltar que para a quantidade de contribuidores ter a sua devida relevância, o projeto deve ter suas devidas políticas para *commits* do código-fonte e lançamento de atualizações. Além disso, devem possuir certos requerimentos e instruções que sirvam para controlar o número e a qualidade dos desenvolvedores que contribuem para o projeto.

Conclusão

Será utilizado o Sass na aplicação de exemplo que será apresentada no final do projeto. Uma das maiores vantagens são os frameworks disponíveis para ele. O *Compass* vem sendo amplamente utilizado pela comunidade e traz diversas ferramentas que auxiliam no desenvolvimento e manutenção do código CSS. Apesar do projeto do LESS se encontrar mais ativo que o do Sass no GitHub, a diferença não se mostra tão grande e relevante.

3.3 Gerenciadores de Dependência

Junto com o aumento da complexidade das aplicações web, aumentam também o número de bibliotecas de terceiros que são utilizadas. Em grandes projetos, tornasse difícil o gerenciamento delas, principalmente quando se trata das versões e compatibilidades.

Para isso foram criados os Gerenciadores de Dependência, que automatizam todas as tarefas referentes à administração e atualização das bibliotecas utilizadas no projeto.

3.3.1 Bower

O *Bower* (JOUBRAN, 2014) foi criado para facilitar a administração, a atualização e o controle de versão das dependências de terceiros da aplicação. É uma das ferramentas de gerenciamento de dependências mais utilizadas no mercado e focada em desenvolvimento *front-end*.

O *Bower* funciona da seguinte maneira:

- Por ser um pacote NodeJS, para a instalação deve-se utilizar o comando: `npm install bower`.
- Após disso, define-se todas as dependências de terceiros no arquivo de configuração do *Bower*, chamado por padrão de *bower.json*. Neste arquivo

JSON encontram-se todas as bibliotecas de terceiros das quais a aplicação depende e suas respectivas versões.

- Após isto, ao executarmos o comando `bower`, a ferramenta automaticamente, obtém as dependências das versões especificadas para uma determinada pasta do projeto.

Isso torna mais fácil a configuração do projeto em uma nova máquina, já que todas as dependências estão explicitadas no arquivo *bower.json*. Além disso, o *Bower* verifica a compatibilidade entre frameworks, alertando sobre as versões a serem utilizadas.

3.4 Automatizadores de Tarefas

Neste capítulo serão abordadas *workflow management tools* (ferramentas para gerenciamento de fluxo de trabalho) baseadas em JavaScript. São ferramentas que se responsabilizam pela execução de tarefas repetitivas, porém essenciais, presentes no desenvolvimento *front-end*, como minificação, otimização, processamento de CSS, deploy etc. *Task Runners*, como também são chamadas, aumentam consideravelmente a produtividade do desenvolvimento, tornando o time mais eficiente, reduzindo os custos e o tempo de desenvolvimento. Além disso, promovem o aumento da qualidade do software, uma vez que diminuem a quantidade de erros humanos.

3.4.1 Gerenciamento de fluxo de trabalho

3.4.1.1 Yeoman

O *Yeoman* (SPRATLEY, 2014) é uma ferramenta de *scaffolding* e gerenciamento de fluxo de trabalho. Ela automatiza diversas tarefas rotineiras e necessárias em projetos de variados tipos. O *AngularJS* por si só já automatiza boa parte do código *boilerplate* (repetido),

porém ainda assim existem tarefas manuais, como a criação e gerenciamento da estrutura de arquivos e diretórios, referênciação de arquivos, criação do esqueleto dos testes de unidade, criação dos arquivos de controladores, *views*, diretivas etc.

Generator (geradores) são os *plug-ins* do *Yeoman*. Existem diversos geradores relacionados ao AngularJS que automatizam todas as tarefas anteriormente descritas. Um gerador que é muito utilizado e que será explorado no exemplo de aplicação é o *Angular Generator*(<https://github.com/yeoman/generator-angular>). Ele cria e gerencia a estrutura de diretórios, se responsabiliza pela criação dos arquivos do projeto, pelas referências necessárias e diversas outras tarefas.

Vale ressaltar que o *Yeoman*, juntamente com o *plug-in Angular Generator*, se integra com diversas ferramentas que foram e serão abordadas neste trabalho, como:

- Pré-processadores de CSS: Sass (e Compass) e LESS
- Gerenciadores de dependência: *Bower*
- Automatizadores de tarefas: *Grunt* (PILLORA, 2014) e *Gulp* (MAYNARD, 2015)
- Frameworks de testes: *Karma* (BRAITWAITE, 2016), *Jasmine* (MORGAN, 2016) e *Protractor* (AMORIM, 2016).

3.4.2 Build

Ao se trabalha com o AngularJS (ou qualquer outro *framework*/biblioteca MVC), é importante entendermos quais componentes e itens são fundamentais para a implementação das tarefas rotineiras que irão compor a *build* do projeto.

Existem ferramentas (*task runners*) que nos disponibilizam diversos *plug-ins* para realização de tarefas de build como:

- Renomear arquivos

- Concatenar arquivos
- Mover e copiar arquivos
- Minificar o CSS e o *JavaScript*
- Executar testes

Um exemplo básico de rotina de build pode ser descrito pelas seguintes tarefas:

1. Execução de testes utilizando o *Karma* e o *Protractor*, validando o estado da aplicação para decidir se o processo de build deve ou não continuar.
2. Concatenação de todos os arquivos *JavaScript* em um único arquivo.
3. Execução do *JavaScript Compiler* para a remoção de espaços e redução do tamanho do nosso arquivo *JavaScript* (Minificação).
4. Compilação CSS em um arquivo menor.
5. Mover o HTML, JavaScript e CSS para um diretório separado de *Release*.

Disponibilizar um único arquivo JavaScript

Quando se trata de um projeto em *AngularJS* (ou qualquer outro framework/biblioteca JavaScript MVC), é importante que se tenha um projeto modulado, com diversos arquivos *JavaScript* e pastas separadas para suas determinadas funções, garantindo maior facilidade na leitura, implementação e manutenção do código. Por outro lado, existirem centenas de arquivos *JavaScript* em uma aplicação tornará certamente o site mais lento para carregar.

Um navegador normalmente possui uma limitação na quantidade de chamadas GET paralelas que pode realizar para um mesmo domínio. Suponha-se que um navegador possa acessar somente 5 arquivos ao mesmo tempo. Deste modo, as centenas de arquivos *JavaScript* terão que ser divididas em blocos de 5 arquivos para serem acessados, fazendo com que o processo se torne extremamente lento.

A maneira mais correta de disponibilizar arquivos *JavaScript* em produção é reunir todos os arquivos em um só. Com isso, se garante que haverá apenas uma solicitação do navegador para o carregamento do *JavaScript* da aplicação.

O mesmo deve ser feito para os arquivos CSS. Não é recomendado que haja mais do que três a cinco solicitações em paralelo em um mesmo instante, evitando que o navegador fique bloqueado e serialize as solicitações.

Minificação

Ao realizar a concatenação dos arquivos, diminui-se consideravelmente a quantidade de requisições em paralelo feitas. Porém, o tamanho total e os dados continuam os mesmos. Para garantir que a aplicação seja rápida e compacta, é necessária a redução da quantidade de *bytes* do arquivo de *JavaScript*.

O arquivo *JavaScript* pode ser minificado através de um minificador, como o *UglifyJS* (<https://github.com/mishoo/UglifyJS>) ou o Google Closure JavaScript (<https://developers.google.com/closure/compiler>). Essas ferramentas percorrem o arquivo *JavaScript* e removem tudo que for desnecessário, como espaços e comentários, reduzindo o seu tamanho. Além disso, ainda podem ser configurados para renomear os nomes das variáveis, tornando o código ainda menor e incompreensível, promovendo também maior segurança.

Vale ressaltar que, quando se trata de aplicações *AngularJS*, deve-se prestar atenção em sempre se seguir o estilo seguro de injeção de dependência, para que a aplicação funcione normalmente após o processo de minificação. Existe para isso também uma biblioteca de código aberto chamada *ng-annotate* (<https://github.com/olov/ng-annotate>) que faz a conversão de um código não seguro para um código seguro para minificação.

Códigos CSS também passam pelo mesmo processo de minificação, removendo espaços e comentários, além de identificar maneiras de reescrever o código para que este seja otimizado.

A seguir serão apresentados os dois mais populares *task runners* baseados em *JavaScript*: *Grunt* e *Gulp.js*.

3.4.2.1 - *Grunt*

O *Grunt* (PILLORA, 2014) foi liberado por Ben Alman em março de 2012. O projeto tem como título *The JavaScript Task Runner* e foi desenvolvido com o objetivo de automatizar tarefas rotineiras relacionadas ao desenvolvimento em *JavaScript*.

Com a popularização do *JavaScript* e junto com o grande aumento da demanda de aplicações web no mundo, o *Grunt* sempre fez um grande sucesso na comunidade de código aberto.

O *Grunt* é uma ferramenta de linha de comando. Uma vez instalado, pode-se executar o comando `grunt` no diretório que se encontra a nossa aplicação. Este comando faz com que o *Grunt* procure pelo *Gruntfile.js*. Este arquivo é o ponto de entrada da *build*, onde pode-se definir as tarefas a serem executadas, carregar tarefas de arquivos externos, configurar as tarefas etc.

Segue um exemplo simples de um *Gruntfile.js* onde é realizada a tarefa de minificação:

```
1. //Exemplo de grunt file - Minificação
2. module.exports = function(grunt) {
3.   // Carrega o plugin que define a tarefa de minificação
4.   grunt.loadNpmTasks('grunt-contrib-uglify');
5.   // Projeta a configuração
6.   grunt.initConfig({
7.     uglify: {
8.       target1: {
```

```

9.         src: 'foo.js',
10.        dest: 'foo.min.js'
11.    }
12.  }
13. });
14. // Define a tarefa padrão
15. grunt.registerTask('default', ['uglify']);
16. };

```

Como podemos ver, no *Gruntfile.js* são declarados os *plug-ins* e as tarefas que serão seguidas. Nesse exemplo, carregamos o plug-in *grunt-contrib-uglify*, definimos a configuração de minificação e registramos ela como tarefa.

3.4.2.2 - Gulp

O *Gulp* (MAYNARD, 2015) foi lançado no início de 2014 como uma alternativa ao *Grunt*. É também uma ferramenta para automatização de tarefas rotineiras de aplicações *JavaScript*, porém possui alguns diferenciais.

O *Gulp*, é baseado em código *JavaScript* e não em configurações, como o *Grunt*. Ele utiliza a *API de Stream* do *Node.js*, promovendo builds mais rápidas, uma vez que utiliza a memória para armazenar os dados temporários e não o disco rígido.

Segue um exemplo simples de um *Gulpfile.js* onde é realizada a tarefa de minificação:

```

1. var gulp = require('gulp');
2. var uglify = require('gulp-uglify');
3. gulp.task('uglify', function(){
4.   gulp.src('foo.js')
5.     .pipe(uglify())
6.     .pipe(gulp.dest('foo.min.js'));
7. });
   gulp.task('default', ['uglify']);

```


Com este *Gulpfile.js*, o Gulp realiza a mesma tarefa descrita anteriormente para o Grunt. Por meio de código, nós definimos o *plug-in* de configuração, definimos a tarefa de *uglify* (minificação) e a registramos no final.

3.4.2.3 - Estudo Comparativo

Documentação

O *Grunt* possui uma documentação formatada e bem apresentada em seu site. Já o *Gulp.js* possui sua documentação apenas no site do GitHub, sendo apresentada de uma maneira um pouco mais complexa.

Curvas de aprendizado

No *Grunt* são declarados *plug-ins* e tarefas que serão seguidas. Já no Gulp utiliza conceitos da *API de Stream* do *Node.js* (MCLAUGHLIN, 2011), que devem ser dominados para a utilização completa da ferramenta. Desta forma, o *Grunt* necessita de menos conhecimento prévio específico para começar a ser utilizado.

Plug-ins

Uma característica importante a respeito dos dois automatizadores de tarefas é a quantidade de *plug-ins* que eles nos disponibilizam utilizar. O *Grunt* possui, hoje, 5,873 *plug-ins*. Já o Gulp oferece 2606 *plug-ins*.

Apesar da diferença entre a quantidade de *plug-ins*, o Gulp ainda possui uma quantidade expressiva. Além disso, a qualidade dos *plug-ins* a serem utilizados devem também ser avaliadas.

Velocidade

Em sua execução o *Grunt* realiza I/O, utilizando o disco rígido. Ao passo que as tarefas vão sendo executadas arquivos temporários são gerados. Além disso, o Grunt funciona de forma sequencial, onde uma tarefa é executada após a outras.

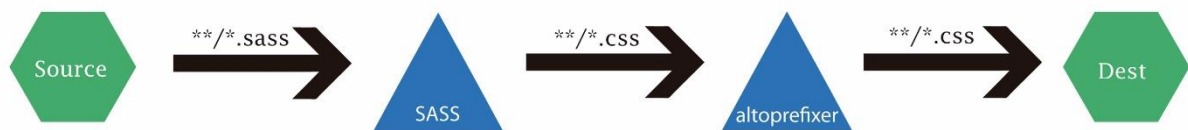


Figura 11 No *Grunt*, precisa-se criar arquivos intermediários em disco

Já o *Gulp*, por usar a API de *Stream* do *Node.js*, armazena todas as informações em memória, não utilizando o disco. Além disso, as tarefas podem ser executadas em paralelo, caso sejam configuradas desta maneira.

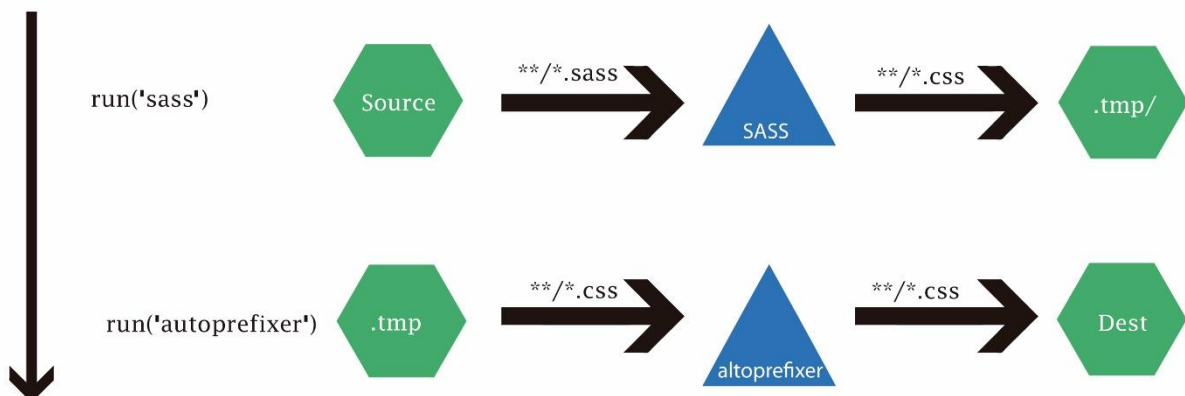


Figura 12 No *Gulp* pode-se canalizar os arquivos intermediários na memória para serem usados por outros fluxos

Comunidade

Interesse - Google trends

Analisando ao longo do tempo o interesse de pesquisas realizadas no Google por cada uma dessas ferramentas, pode-se notar que a partir da metade de 2012 o *Grunt* veio dominando o mercado sozinho, uma vez que o *Gulp* ainda não havia sido lançado. No início de 2014 *Gulp* começa a tomar força, ultrapassado o *Grunt* por volta de agosto de 2015. A partir daí o nível de interesse do *Grunt* vem caindo e o do *Gulp* crescendo.

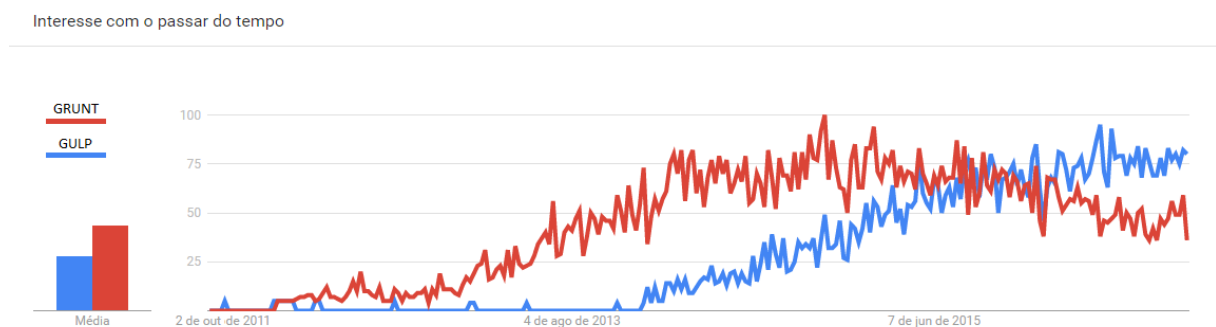


Figura 13 Interesse de pesquisa sobre as ferramentas *Gulp* e *Grunt* no decorrer do tempo

Número de Downloads

Um aspecto interessante de ser analisado é que mostra o potencial de crescimento de cada framework é o número de downloads. De acordo com o site do *npm* (www.npmjs.com) o *Grunt* teve 1.848.636 downloads no último mês, contra 2.321.906 do *Gulp*. (Ambas as consultas foram realizadas em 28 de Agosto de 2016)

Número de perguntas na plataforma StackOverflow

Analisando o número de perguntas plataforma StackOverflow, que é um dos maiores sites de perguntas e respostas relacionadas à programação do mundo, temos um total de 22,654

perguntas relacionadas ao *Grunt* e 22,086 relacionadas ao *Gulp*. (Ambas as consultas foram realizadas em 27 de Agosto de 2016)

Repositório GitHub

Analisando o repositório GitHub de cada um, temos que o *Grunt* possui *11021 Stars* (número de usuários interessados no projeto) e *64 Contribuidores*. Já o *Gulp* possui *22,849 Stars* e *175 Contribuidores*. Isto mostra que o projeto do *Gulp* mais ativo que o do *Grunt*.

(Ambas as consultas foram realizadas em 27 de Agosto de 2016)

Os contribuidores são as pessoas que contribuem para o código do projeto. Mais usuários do GitHub contribuindo para o projeto indica que há mais pessoas revisando os códigos, mais pessoas desenvolvendo para a melhoria da ferramenta e menor a chance de o projeto ser abandonado.

Vale ressaltar que para a quantidade de contribuidores ter a sua devida relevância, o projeto deve ter suas devidas políticas para *commits* do código-fonte e lançamento de atualizações. Além disso, devem possuir certos requerimentos e instruções que sirvam para controlar o número e a qualidade dos desenvolvedores que contribuem para o projeto.

Conclusão

Apesar de ser mais novo, o *Gulp* tem uma quantidade expressiva de plug-ins e tópicos no StackOverflow. Apesar de demandar mais conhecimento prévio, uma vez que utiliza a *API de Stream* do Node.js, o *gulp* se mostra mais rápido. Por outro lado, o *Grunt* demanda menos conhecimento prévio. Por ser baseado em configurações, a implementação de suas rotinas é mais fácil. Se encontra a mais tempo no mercado, ainda sendo considerada uma ferramenta mais sólida que o *Gulp*.

Apesar do Gulp se mostrar como tendência, o Grunt atende muito bem às necessidades de automatização de build, além de ser totalmente suportado pelo *Angular Generator*, que é o plug-in do Yeoman que será utilizado para o desenvolvimento da aplicação de exemplo.

3.5 Testes em aplicações web

Sempre esteve presente na vida dos desenvolvedores *front-end* a maçante rotina de testes manuais. O repetitivo processo consistia em codificar algumas linhas, atualizar o navegador e verificar se o comportamento correspondia ao esperado. Esses tipos de teste além de consumirem muito tempo, são muito propensos a erro e irreproduzíveis. Somado ao fato de existirem diversos navegadores diferentes, com diferentes comportamentos, realizar o teste manualmente em todos eles se tornava uma tarefa extremamente custosa. Além de aumentar muito o tempo e custo de desenvolvimento, não garante que o resultado final da aplicação possua uma confiabilidade satisfatória.

Nos últimos anos, houve o surgimento de ferramentas de *debug* para *JavaScript* e diversas IDEs mais poderosas, que nos mostram erros de digitação etc. Porém, ainda se mantém o processo manual, demorado e propenso a erros de *debugging*.

Testes automatizados são a solução para acabar com o processo manual de testes. Diversas ferramentas para automatização de testes foram criadas para facilitar o trabalho dos desenvolvedores web. Como ferramentas para escrita de testes unitários e de integração e ferramentas para execução automática destes testes em diversos tipos de browsers.

As vantagens são claras. Uma vez que agora existe uma maneira conveniente de execução de testes, podendo estabelecer procedimentos e programações específicas para a execução destes, não é mais necessária a interação manual.

Ao contrário de linguagens fortemente tipadas, que possuem segurança de tipos e compiladores, o *JavaScript* não possui artifícios nativos que garantam que o código não possui erros. Daí a importância de se utilizar o TDD para o desenvolvimento de aplicações web que utilizem frameworks JavaScript MVC.

3.5.1 Teste de unidade

Os testes de unidade têm como conceito verificar se uma função ou parte do código está funcionando como esperado, através de provações e asserções.

A execução automática desses testes no decorrer de todo desenvolvimento e manutenção do código, garante que após determinadas mudanças a aplicação continuará funcionando de acordo com o esperado.

Este tipo de teste é comumente utilizado em aplicações do lado do servidor e não do cliente. Sendo esse um ponto onde o *AngularJS* se destaca, uma vez que é muito bem integrado com ferramentas de testes. Logo, na aplicação de exemplo será utilizado o framework *Jasmine* para a criação de testes de unidade.

Por que devem ser executados

Seguem algumas das justificativas para a execução de testes de unidade, principalmente quando se trata do desenvolvimento de aplicações em *JavaScript*:

- O JavaScript não possui um compilador que nos mostre que algo não funciona corretamente. Logo, erros serão alertados pelos navegadores, que por serem diferentes podem apresentar erros distintos. Como os testes unitários são executados durante todas as etapas do desenvolvimento, pode-se identificar os erros antes da nossa aplicação chegar ao navegador. Deste modo, também se aumenta a velocidade de desenvolvimento.

- Em projetos mais complexos, geralmente vários desenvolvedores trabalham no mesmo projeto, gerando modificações que podem afetar o código dos outros. Os testes de unidade sempre mostrarão quando determinada alteração afetar alguma pressuposição esperada. Deste modo, evitaremos regressões.
- Como os testes escritos com *Jasmine* possuem uma sintaxe de elevado nível, podendo ser lidos como textos simples, os testes de unidade podem atuar também como uma especificação fluida e ativa do código da aplicação.

3.5.2 Testes de integração

Os testes de unidade possuem por sua vez algumas limitações. Eles fazem suposições sobre os demais pontos de integração entre os módulos do código, como controladores e serviços no *AngularJS*. O teste de unidade supõe que cada módulo integrado a ele se comporte de determinada maneira ou retorne determinado resultado. Porém, essas suposições podem não ser verdadeiras. Os testes de integração verificam se os diferentes módulos da aplicação estão devidamente configurados e associados.

3.5.3 Testes fim-a-fim

Os testes fim-a-fim envolvem abrir um navegador, acessar a aplicação web *front-end* e realizar cliques e digitações, da mesma maneira que um usuário real o faria. Testam o fluxo fim-a-fim a aplicação, simulando testes de usuário no nível real. Desta maneira conseguimos garantir que o servidor está se comportando da maneira correta e garantir que o nosso HTML foi formatado e exibido da maneira correta.

Ao mesmo tempo que são necessários, não é recomendável substituir os testes de unidade e de integração pelos testes fim-a-fim. Um dos motivos é que para cobrir toda a

aplicação, seria necessária uma quantidade exponencial de testes. Além disso, estes testes não oferecem as informações necessárias sobre o que está errado, somente que algo está com problema.

O objetivo destes testes é garantir que os recursos e fluxos mais básicos da aplicação estão funcionando corretamente. Isto pode ser feito com um conjunto pequeno e determinístico de testes.

De acordo com SESHADI (2014, p.327):

“Uma separação comum e aceitável no que diz respeito às quantidades relacionadas aos três tipos de teste em sua aplicação é 70:20:10, em que 70% de seus testes serão testes de unidade, 20% serão testes de nível de integração e 10% serão testes de cenário fim-a-fim. Esta é a proporção ideal que devemos ter como meta”

Para aplicações desenvolvidas em *AngularJS*, podemos utilizar o Protactor (AMORIM, 2016).

3.5.4 - Karma - Execução de testes

O *Karma* (BRAITWAITE, 2016) é uma ferramenta de execução automática de testes, que utiliza NodeJS* e SocketIO*. Ela é responsável por encontrar todos os testes de unidade no código, abrir os navegadores, executá-los e capturar os resultados. Tudo isso de forma automática e rápida.

Plug-ins do Karma

O *Karma* possui, de modo geral, quatro grupos de *plug-ins* que podem ser instalados.

São eles:

- **Plug-ins para inicialização de navegadores**

Este *plug-in* é responsável por iniciar automaticamente determinado navegador na hora de execução dos testes. Existem *plug-ins* deste tipo para diversos navegadores, como Google Chrome, Mozilla Firefox, Internet Explorer etc. Na aplicação de exemplo utilizamos o *karma-chrome*.

- **Plug-ins para frameworks de escrita de testes**

Com este tipo de *plug-in* é selecionado o *framework* que será utilizado para escrever os testes de unidade. Existem *plug-ins* para diversas ferramentas, como o *Jasmine*, que será utilizada na aplicação web de exemplo.

- **Plug-ins informantes**

Estes *plug-ins* se referem à exibição dos resultados dos testes. O *Karma*, atualmente, já vem com um *plug-in* padrão instalado. Porém, existe a opção de disponibilizar esses resultados em arquivos *junit.xml*, bastando instalar o *plug-in* necessário para tal.

- **Plug-ins para integração**

São os *plug-ins* utilizados para integrar o Karma com outros frameworks *JavaScript*, como o *RequireJs* ou o *Closure* do Google.

Configuração do Karma

O *Karma* possui um arquivo de configuração que por padrão é chamado de *karma.conf.js*. Neste arquivo é descrito para o framework como ele deve trabalhar. O arquivo possui seções nas quais os desenvolvedores definem as suas preferências.

O *Karma* possui uma ferramenta muito interessante para a criação deste arquivo de configuração. Basta executar o comando `karma init` no interpretador de linha de comando sobre a pasta da aplicação. A execução deste comando exibe um *shell* interativo que faz diversas perguntas relacionadas às preferências a respeito do modo que o *Karma* deve agir. Com as perguntas respondidas, o arquivo de configuração é automaticamente gerado.

3.5.5 - Jasmine - Escrita de testes

Jasmine (MORGAN, 2016) é um *framework* para a escrita de testes que utiliza o estilo orientado a comportamento (*BDD - Behavior Driven Development*) para a criação dos testes. Ou seja, em vez de se escrever conjuntos de funções e asserções, descreve-se os comportamentos e define-se as expectativas.

3.5.6 - Protactor

O *Protactor* (AMORIM, 2016) é uma ferramenta para a configuração de testes fim-a-fim. A primeira tentativa de criar testes de cenário fim a fim com o *AngularJS* foi feita com o *AngularJS Scenario Runner* (<https://code.angular.org/1.2.16/docs/guide/e2e-testing>). Ela simulava ações do usuário, como cliques e digitações, por meio de JavaScript. Notou-se, porém, que esse método não representava realmente o fluxo de um usuário real.

Devido a isso, o *Protactor* foi criado. Baseando-se no *Selenium WebDriver* (<http://docs.seleniumhq.org/projects/webdriver/>), que funciona a nível do sistema operacional. O *Protactor* foi implementado para trabalhar com o browser realizando cliques e digitações do mesmo modo que um usuário real o faria.

Existe ainda outro problema em testes fim-a-fim com aplicações AJAX, que é esperar pela página ser carregada. Em uma SPA (Single-Page Application), mesmo havendo apenas uma página a ser carregada, dados costumam ser carregados assincronamente. Deste modo, não seria uma tarefa fácil saber o momento certo para verificar se um elemento foi corretamente carregado.

O *Protactor*, além de ter sido desenvolvido com base no *WebDriver*, tem conhecimento do *AngularJS*. Desta forma, após de qualquer interação realizada com a interface, ele sabe esperar a resposta do servidor antes de continuar com o teste. Desta forma, pode-se confiar que o teste será executado exatamente da maneira que um usuário o faria.

4. Exemplo de Aplicação

Neste capítulo será apresentado o passo a passo para o desenvolvimento front-end de uma aplicação web simples. Ela servirá como prova de conceito para os assuntos abordados neste trabalho.

Será desenvolvida uma agenda telefônica simples. Consistirá em uma aplicação de página única (SPA) *client-side* (do lado do cliente). Será seguido o padrão de arquitetura MVC e abordadas as funcionalidades de criação de novos contatos e listagem de todos os registros.

A aplicação consumirá o serviço MinhaAgenda.Back, que é uma API RESTful desenvolvida em C#. Ela nos disponibilizará os serviços necessários e se encontra no repositório do Bitbucket (<https://bitbucket.org/loscamaradas/minhaagenda.back>) .

Para o desenvolvimento da aplicação, serão utilizadas as seguintes tecnologias:

- **AngularJS:** Framework JavaScript que nos auxiliará a criar uma SPA (Single-page Application, ou Aplicação de página única) utilizando o padrão de arquitetura MVC.
- **Karma, Jasmine e Protractor:** Ferramentas para automatização e criação de testes. Com elas será possível seguirmos a técnica de desenvolvimento de software TDD (Teste Driven Development, ou Desenvolvimento guiado por testes).
- **Yeoman:** Ferramenta de *scaffolding* e gerenciamento de fluxo de trabalho. Ela automatiza diversas tarefas rotineiras.
- **Grunt:** Automatiza o processo de *build* da aplicação.
- **Sass e Compass:** Pré-processador de CSS, que auxilia na criação dos arquivos de CSS para estilização das nossas páginas.

A aplicação será desenvolvida utilizando-se o sistema operacional Windows 10, a IDE de código-aberto *Visual Studio Code* v1.2.1 e o prompt de comando *PowerShell*. Além disso, serão usadas as últimas versões de todos os frameworks, bibliotecas e ferramentas até 20 de Agosto de 2016.

4.1 - Configurando o ambiente de desenvolvimento

Primeiramente deve-se instalar o NodeJS (<https://nodejs.org/en/>). Com isso pode-se utilizar o comando `npm install` para instalar os frameworks necessários.

Para configurar o ambiente de desenvolvimento deve-se abrir o *PowerShell* e seguir os seguintes passos:

- **Primeiro passo:** Instalar o *Sass* e o *Compass*. Para isto deve-se instalar o Ruby (<http://rubyinstaller.org/downloads/>) na máquina. Com o Ruby instalado pode-se instalar o *Compass* utilizando o seguinte comando: `gem install compass`
- **Segundo passo:** Agora serão instalados o *Yeoman*, o *Grunt*, o *Karma Generator* (que vem com o *Karma* e com o *Jasmine*) e o *Angular Generator* (que automaticamente instala o *AngularJS*). Para isso é utilizado o seguinte comando: `npm install -g grunt-cli bower yo generator-karma generator-angular`

Após configurado o ambiente de desenvolvimento, pode-se criar a estrutura da aplicação com o *plug-in Angular Generator* do *Yeoman*.

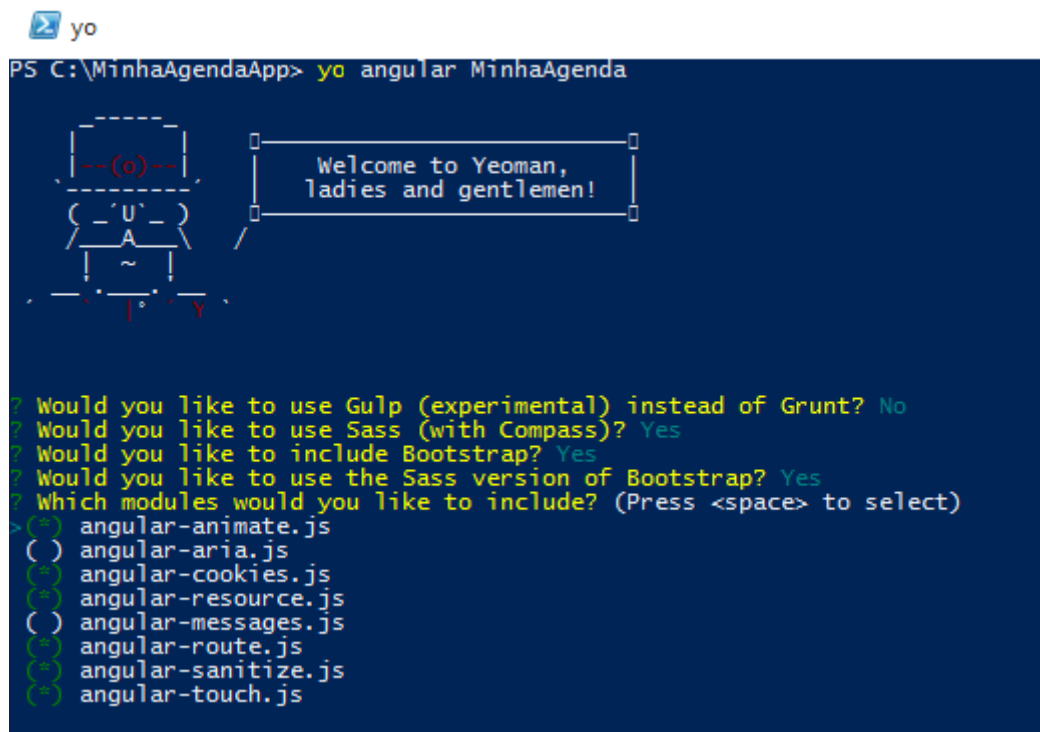
4.2 - Criando a estrutura da aplicação utilizando o Yeoman (Angular Generator)

Após instalados os pacotes necessários, pode-se começar a desenvolver a aplicação.

Primeiramente se criar uma pasta para o nosso projeto. Depois de criada, deve-se navegar até ela utilizando o *PowerShell*. Desta forma pode-se pedir para o *Yeoman* (Angular Generator) criar o esqueleto da aplicação, que será chamada de *MinhaAgenda*, utilizando o seguinte comando:

```
yo angular MinhaAgenda
```

Como mostra a figura na Figura 14, o Yeoman nos faz algumas perguntas. Respondemos que gostaríamos de usar o Grunt, o Sass com o Compass, a versão Sass do Bootstrap e que gostaríamos de incluir alguns módulos..



```
PS C:\MinhaAgendaApp> yo angular MinhaAgenda

Welcome to Yeoman,
ladies and gentlemen!

? Would you like to use Gulp (experimental) instead of Grunt? No
? Would you like to use Sass (with Compass)? Yes
? Would you like to include Bootstrap? Yes
? Would you like to use the Sass version of Bootstrap? Yes
? Which modules would you like to include? (Press <space> to select)
> ( ) angular-animate.js
  ( ) angular-aria.js
  (x) angular-cookies.js
  (x) angular-resource.js
  ( ) angular-messages.js
  (x) angular-route.js
  (x) angular-sanitize.js
  (x) angular-touch.js
```

Figura 14 Criação da estrutura do projeto com o Yeoman (Angular Generator)

Feito isso, a estrutura da aplicação é criada, como mostra a Figura 15.

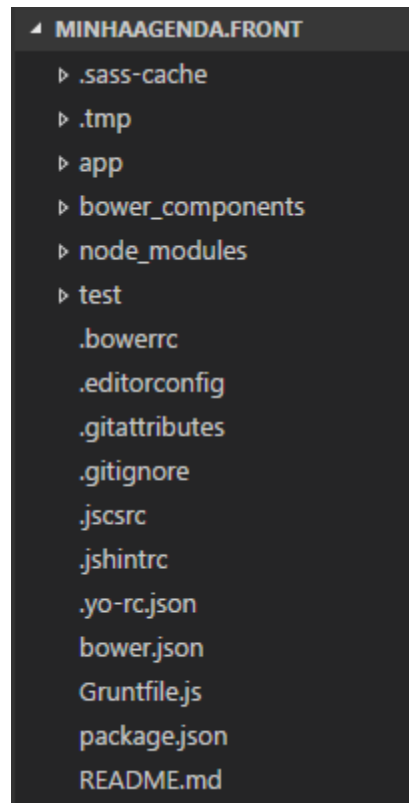


Figura 15 Estrutura de diretórios criada pelo Angular Generator (Yeoman)

O Yeoman (Angular Generator) gerou automaticamente toda a estrutura básica para o projeto.

Na pasta raiz, pode-se destacar os seguintes arquivos e pastas:

- *Gruntfile.js*: O *Yeoman* (*Angular Generator*) já elaborou o arquivo de configuração Grunt, com toda a automatização de build configurada.
- *Bower.json* e *bower_components*: Também foi criado o arquivo de configuração do Bower, referenciando todos os pacotes básicos que serão utilizados, assim como a pasta onde ficaram armazenados estes pacotes
- *Package.json*: Este é o arquivo de configuração do *npm*, que é um gerenciador de pacotes. Nele estará descrito todos os frameworks que estão sendo utilizados, como Yeoman, Grunt, Sass etc.
- *.gitignore* e *.gitattributes*: São alguns arquivos de configuração do *Git*, que é um sistema de controle de versão distribuído e um sistema de gerenciamento de código fonte.

- *test*: Foi criada também a pasta com toda a estrutura para criação e execução de testes, utilizando Karma e Jasmine.
- *app*: Esta pasta contém o projeto AngularJS em si. Como podemos ver na Figura 16, nesta pasta é onde se encontram os controladores, *views*, imagens, arquivos de estilo, nosso *index.html*, e aonde serão criadas automaticamente as pastas de serviços, configuração, filtros, diretivas etc.

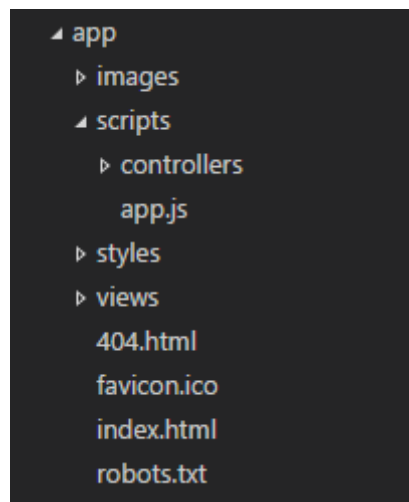


Figura 16 Estrutura da pasta app

O Yeoman (Angular Generator) já disponibiliza uma aplicação minimamente funcional. Para compilá-la e executá-la, será utilizado o Grunt.

Para isto, basta que se navegue até o diretório da aplicação utilizando o PowerShell e se execute o seguinte comando: `grunt serve`

Este comando irá realizar a build do projeto e executá-lo no navegador. Deste modo, o Grunt irá identificar sempre que for modificado qualquer arquivo do projeto e irá atualizá-lo no navegador automaticamente.

A aplicação inicial criada pelo Yeoman é demonstrada na Figura 17.

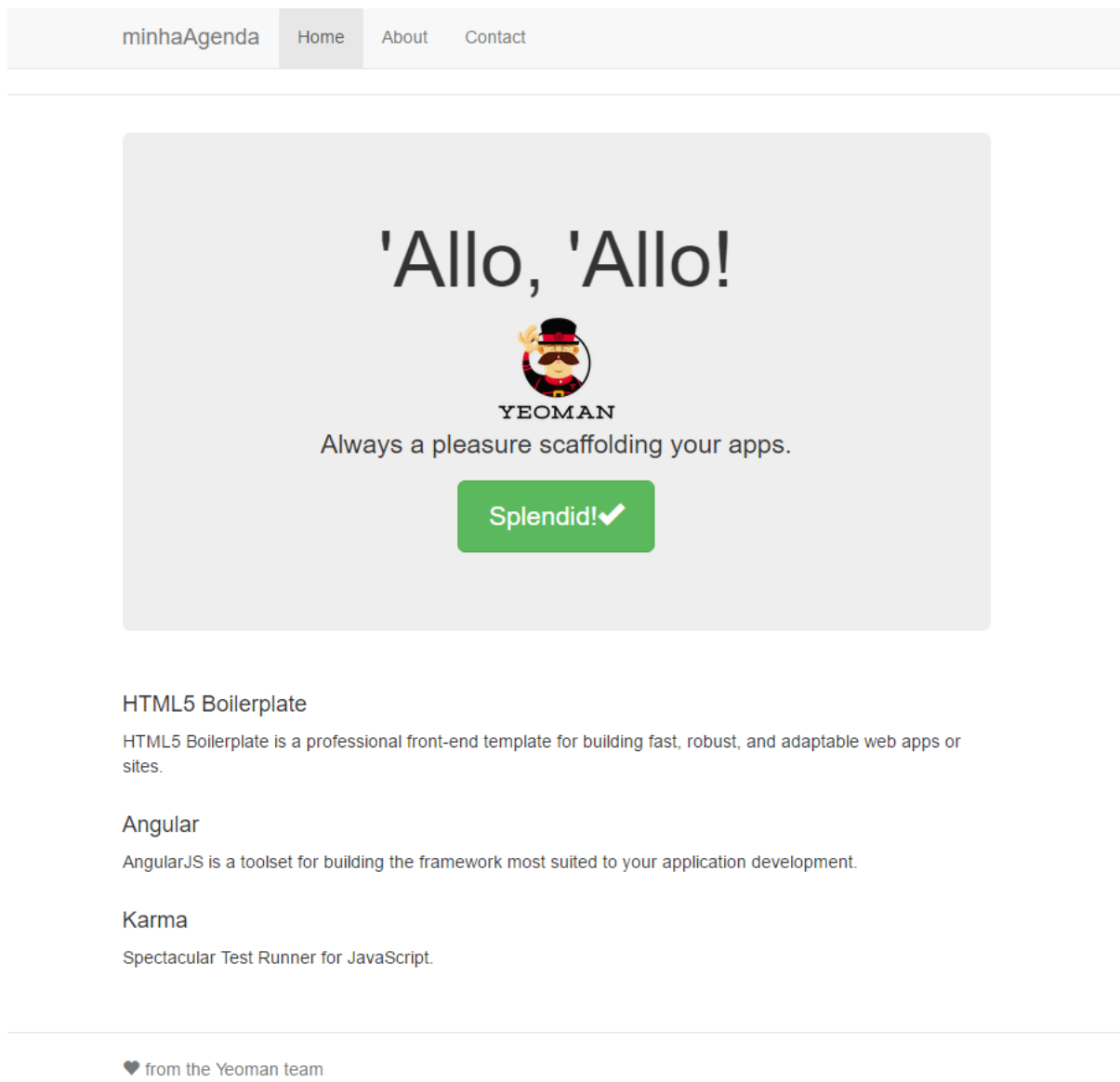


Figura 17 Aplicação inicial criada pelo Yeoman

4.2.1 Módulos do AngularJS

Os módulos funcionam como contêineres no *AngularJS*, um modo de empacotar códigos JavaScript relevantes utilizando um único nome.

Em um módulo podem ser definidos controladores, factories, serviços e diretivas. Estes, que serão abordados mais profundamente no decorrer deste trabalho, são funções e códigos que podem ser acessados em todo o módulo.

O módulo pode também ter como dependências outros módulos. Ao ser instanciado, o *AngularJS* buscará o módulo dependente, utilizando seu nome particular, garantindo que todos os seus serviços, controladores etc. fiquem disponíveis para o módulo principal.

O *AngularJS* também utiliza o módulo para inicializar uma aplicação. Para isso, devemos passar o nome do módulo para a diretiva *ng-app* em nosso arquivo *index.html*, definindo assim o módulo principal da aplicação.

Na aplicação de exemplo, o *Yeoman* (Angular Generator) gerou automaticamente a definição do módulo dentro do arquivo *app/app.js*. Nosso módulo principal é descrito da seguinte maneira:

```
1. angular
2. .module('minhaAgendaApp', [
3.     'ngAnimate',
4.     'ngCookies',
5.     'ngResource',
6.     'ngRoute',
7.     'ngSanitize',
8.     'ngTouch'
9. ])
```

O primeiro argumento da função define o *nome do módulo*. O segundo argumento é um *array de nomes de módulos* dos quais o módulo principal depende.

Para se carregar um módulo que já foi previamente definido, deve-se utilizar a função *angular.module()* com apenas o primeiro argumento.

```
1. angular.module('minhaAgendaApp')
```

Deste modo, pode-se referenciar um módulo em vários arquivos para que seja utilizado, adicionado ou modificado.

Os módulos devem sempre ser carregados previamente no arquivo *index.html*. Isso também foi feito automaticamente pelo *Yeoman* (Angular Generator) da seguinte maneira:

1. `<script src="scripts/app.js"></script>`

4.3. Criando o serviço de comunicação com o servidor

Nesta seção serão abordados alguns temas importantes relacionados ao *AngularJS*, como por exemplo:

- Serviços
- Testes

O *AngularJS* não necessita de nenhum tipo específico de *back-end*. Qualquer linguagem pode ser usada para o desenvolvimento do lado do servidor, como C#, Java, Ruby etc. Precisa-se, porém, haver uma maneira de comunicação com esse servidor, preferencialmente por meio de XHR (solicitação HTTP XML) ou sockets.

O cenário ideal é aquele que os servidores tenham pontos de conexão com *REST* ou *APIs* que disponibilizem valores JSON. Porém, isto não limita o *AngularJS*, uma vez que existem diversos modos de fazê-lo se comunicar com outros formatos.

4.3.1 - O serviço \$http

O serviço *\$http* que o *AngularJS* disponibiliza nativamente facilita bastante a comunicação com *servidores remotos HTTP* utilizando objetos *XMLHttpRequest* ou *JSONP*.

Este serviço oferece os seguintes métodos para comunicação com os servidores REST:

- *\$http.get*
- *\$http.head*
- *\$http.post*

- `$http.put`
- `$http.delete`
- `$http.patch`
- `$http.jsonp`

Estes métodos retornam instancias de *Promise*, que dão acesso ao resultado das solicitações quando completadas.

Não é considerada boa prática realizar chamadas *\$http* diretamente do controlador. Ao encapsularmos as chamadas em serviços, estamos garantindo a reusabilidade de código e maior escalabilidade da aplicação,

4.3.2 - Criando testes de unidade para o nosso serviço

Como discutido neste trabalho, no TDD é importante que se comece pelos testes para depois partir para a implementação. Quando aplicado o desenvolvimento orientado a testes, pode-se definir como o trecho de código testado deve se comportar.

Logo, segue o comportamento esperado do nosso serviço que é responsável pela comunicação com o servidor:

- O serviço deve possuir um método para adição de um novo contato. Este método deve enviar uma requisição POST para determinada URL, contendo um objeto JSON com as informações de um contato.
- O serviço deve possuir um método para listagem de contatos. O método deve enviar uma requisição GET para determinada URL e receber como resposta uma lista de contatos.

Definido deste modo, segue a implementação comentada dos testes:

```
1. describe('Service: contatosServiceClient', function () {
2.
```

```

3.  // Carrega o módulo do controlador
4.  beforeEach(module('minhaAgendaApp'));
5.
6.  //Define as variáveis
7.  var contatosServiceClient,
8.  mockBackend;
9.
10. // Este bloco será executado antes da execução de cada teste
11. // As dependências são injetadas e as variáveis definidas
12. beforeEach(inject(function (_contatosServiceClient_, $httpBackend) {
13.     contatosServiceClient = _contatosServiceClient_;
14.     mockBackend = $httpBackend;
15. })));
16.
17. //Este teste verifica se o método adicionarContato está realizando
18. //um POST ao servidor
19. it('deve realizar um POST de adicao de contato para o servidor', function () {
20.
21.     //Diz qual a URL que deve ser chamada, qual objeto deve ser enviado e qual a
22.     //resposta que o MOCK irá retornas
23.     mockBackend.expectPOST('http://localhost:3412/Contatos',{nome: 'Pedro', telefone:
        '99999-8888'}).respond(200,'');
24.
25.     //Define o objeto contato
26.     var contato = {nome: 'Pedro', telefone: '99999-8888'}
27.
28.     //Executa o método que realiza a chamada ao servidor
29.     contatosServiceClient.adicionarContato(contato)
30.     .then(
31.         function (resposta) {

```

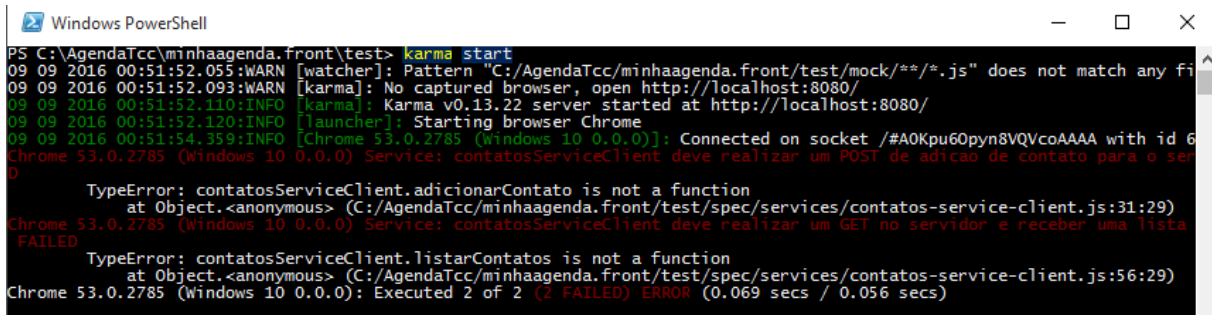
```

32.         expect(resposta).toBeDefined(); //Espera que o status da resposta seja de
           sucesso
33.     }
34. )
35. //Dispara a resposta do MOCK
36.     mockBackend.flush();
37. });
38.
39. //Este teste verifica se o método listarContatos está realizando
40. //um GET no servidor e recebendo uma lista de contatos
41. it('deve realizar um GET no servidor e receber uma lista de contatos', function () {
42.
43.     //Define a lista de contatos
44.     var contatos = [{nome: 'Pedro', telefone: '99999-8888'},
45.                     {nome: 'Gabriel', telefone: '99999-7777'}]
46.
47.     //Diz qual a URL que deve ser chamada e que este irá
48.     //retornar a lista de contatos
49.     mockBackend.expectGET('http://localhost:3412/Contatos').respond(200, contatos);
50.
51.     //Executa o método que realiza a chamada ao servidor
52.     contatosServiceClient.listarContatos()
53.     .then(
54.         function (resposta) {
55.             expect(resposta.data).toEqual(contatos); //Espera-se que seja retorna uma
               lista de contatos
56.         }
57.     )
58.
59.     //Dispara a resposta do MOCK
60.     mockBackend.flush();
61. });

```

```
62. });
```

Para executar os testes basta que se navegue até o diretório de *test/* e execute o comando *karma start*. Como o controlador ainda não foi implementado, os dois testes irão falhar.



```
Windows PowerShell
PS C:\AgendaTcc\minhaagenda.front\test> karma start
09 09 2016 00:51:52.055:WARN [watcher]: Pattern "C:/AgendaTcc/minhaagenda.front/test/mock/**/*.js" does not match any fi
09 09 2016 00:51:52.093:WARN [karma]: No captured browser, open http://localhost:8080/
09 09 2016 00:51:52.110:INFO [karma]: Karma v0.13.22 server started at http://localhost:8080/
09 09 2016 00:51:52.120:INFO [launcher]: Starting browser Chrome
09 09 2016 00:51:54.359:INFO [Chrome 53.0.2785 (Windows 10 0.0.0)]: Connected on socket /#A0Kpu60pyn8VQVcoAAAA with id 6
Chrome 53.0.2785 (Windows 10 0.0.0) Service: contatosServiceClient deve realizar um POST de adicao de contato para o ser
D
  TypeError: contatosServiceClient.adicionarContato is not a function
    at Object.<anonymous> (C:/AgendaTcc/minhaagenda.front/test/spec/services/contatos-service-client.js:31:29)
Chrome 53.0.2785 (Windows 10 0.0.0) Service: contatosServiceClient deve realizar um GET no servidor e receber uma lista
  FAILED
  TypeError: contatosServiceClient.listarContatos is not a function
    at Object.<anonymous> (C:/AgendaTcc/minhaagenda.front/test/spec/services/contatos-service-client.js:56:29)
Chrome 53.0.2785 (Windows 10 0.0.0): Executed 2 of 2 (2 FAILED) ERROR (0.069 secs / 0.056 secs)
```

Figura 18 Testes do serviço de comunicação com o servidor falhando

4.3.3 Criando o serviço

Os serviços no AngularJS são funções que proporcionam funcionalidades para a nossa aplicação. Também podem ser usados como objetos que armazenam dados que podem ser acessados por todo o sistema. Todos os serviços são instanciados apenas uma vez, logo, todas nossa aplicação tem acesso à mesma instância dele. Serviços são utilizados para *caches*, *factories*, comportamentos repetitivos, estados compartilhados, dentre outros.

O AngularJS possui 3 tipos de serviços, os *services*, *providers* e *factories*. Cada um dos 3 possui suas peculiaridades. A *factory* será utilizada para a criação do serviço que se comunicará com o servidor.

Segue seu código implementado e comentado:

1. *//Define o serviço que se comunicará com o back-end, chamado de 'contatosServiceClient'*
2. `angular.module('minhaAgendaApp')`
3. `.factory("contatosServiceClient", function ($http, configValue) {`
- 4.

```

5.      //Função que realiza uma requisição GET para a URL do servidor
6.      var _listarContatos = function () {
7.          return $http.get(configValue.contatosServiceBaseUrl + "/Contatos");
8.      };
9.
10.     //Função que realiza uma requisição POST a URL do servidor
11.     var _adicionarContato = function (contato) {
12.         return $http.post(configValue.contatosServiceBaseUrl + "/Contatos", contato);
13.     };
14.
15.     //Expõe os métodos para quem consumir o serviço
16.     return {
17.         listarContatos: _listarContatos,
18.         adicionarContato: _adicionarContato
19.     };
20. });

```

Esse serviço tem como função basicamente encapsular as chamadas *\$http* ao servidor e retornar a *promise* de cada uma. Cabendo então aos outros controladores ou serviços a manipulação das possíveis respostas.

Vale ressaltar que é válido armazenar as URLs em um arquivo de configuração, pois deste modo é garantida também maior escalabilidade da aplicação. Para isso é criado um arquivo chamado *config-value*, utilizando o comando:

```
yo angular:value config-value
```

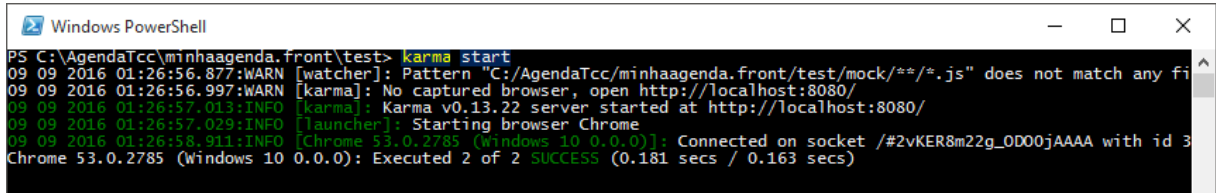
Segue o código do arquivo de configuração:

```

1. angular.module('minhaAgendaApp')
2.   .value('configValue', {
3.       contatoServiceBaseUrl: "localhost:8080/"
4.   });

```

Após a implementação realizada, executa-se novamente nossos testes, utilizando o comando `karma start`.



```
PS C:\AgendaTcc\minhaagenda.front\test> karma start
09 09 2016 01:26:56.877:WARN [watcher]: Pattern "C:/AgendaTcc/minhaagenda.front/test/mock/**/*.js" does not match any fi
09 09 2016 01:26:56.997:WARN [karma]: No captured browser, open http://localhost:8080/
09 09 2016 01:26:57.013:INFO [karma]: Karma v0.13.22 server started at http://localhost:8080/
09 09 2016 01:26:57.029:INFO [launcher]: Starting browser Chrome
09 09 2016 01:26:58.911:INFO [Chrome 53.0.2785 (Windows 10 0.0.0)]: Connected on socket /#2vKER8m22g_0D00jAAAA with id 3
Chrome 53.0.2785 (Windows 10 0.0.0): Executed 2 of 2 SUCCESS (0.181 secs / 0.163 secs)
```

Figura 19 Testes do serviço de comunicação com o servidor sendo bem sucedidos

4.4 Implementando a adição de novos usuários

Nesta seção implementaremos a adição de novos contatos. Com isso, serão expostos diversos tópicos importantes a respeito do *AngularJS*, como:

- Rotas
- Visões (views)
- Formulários e validações
- Controladores (controllers)

4.4.1 Criando a estrutura da funcionalidade

Para implementar a funcionalidade de adicionar contatos na aplicação, primeiro se deve planejar como será feito. Seguem destacados em tópicos os procedimentos que devem ser seguidos:

- Criar a Visão (*view*) e seu respectivo Controlador (*controller*).
- Referenciar o arquivo *JavaScript* do Controlador no *index.html*, para que o *AngularJS* saiba que ele exista.

- Definir a rota da Visão (*/adicionar-contato*) no configuração *\$routeProvider* do módulo principal.
- Por se tratar de um desenvolvimento orientado a testes (TDD), antes de começar a implementar o controlador, cria-se testes de unidade para ele.

O *Yeoman* (Angular Generator) automatiza a criação de toda a estrutura base para esta implementação. Para isso, deve-se navegar no PowerShell até a pasta da aplicação e executar o seguinte comando:

```
yo angular:route adicionar-usuario
```

Deste modo, o *Yeoman* (Angular Generator) realizou as seguintes tarefas:

1. Referenciou o arquivo JavaScript do Controlador (*app/scripts/controllers/registrar-contato.js*) no *Index.html*
2. Criou o arquivo JavaScript do Controlador : *app/scripts/controllers/registrar-contato.js*
3. Preparou o esqueleto de testes para o Controlador : *test/spec/controllers/registrar-contato.js*
4. Cria a rota da nossa View (Visão) na configuração *\$routeProvider* do Módulo principal, referenciando o controlador à ela.
5. Cria a Visão: *app/views/registrar-contato.html*

4.4.2 - Criação da rota

No *AngularJS* as rotas tem a função de, quando a URL for modificada, realizar o carregamento e apresentar as páginas (*templates*), como também instanciar o controlador (*controller*) que trará contexto a este *template*.

Ao se executar o comando para criação de rota, automaticamente o *Yeoman* (Angular Generator) define a rota na configuração do serviço *\$routeProvider* do módulo. Este serviço

trabalha da seguinte maneira: quando as modificações no *location.path()* correspondem a alguma mapeamento realizado, a view (visão) correspondente é carregada na tela.

Está configuração encontra-se no arquivo *app/app.js* e apresenta-se da seguinte forma:

```
1. .config(function ($routeProvider) {  
2.     $routeProvider  
3.         .when('/adicionar-contato', {  
4.             templateUrl: 'views/adicionar-contato.html',  
5.             controller: 'AdicionarContatoCtrl',  
6.             controllerAs: 'adicionarContato'  
7.         })  
8.         .otherwise({  
9.             redirectTo: '/adicionar-contato'  
10.        });
```

Na linha 3 temos o nosso segundo *when()*. Ele nos diz que quando a *URL* for *'localhost/#/adicionar-contato'*, nossa visão *adicionar-contato.html* será renderizada, Diz também que o controlador ligado à essa rota é o *AdicionarContatoCtrl* e que este será chamado de *adicionarContato* na nossa visão.

Na linha 13tem-se o *otherwise*. Qualquer outra *URL* que for digitada será direcionado para *'localhost//adicionar-contato'*.

4.4.3 - Criando os testes de unidade para o controlador

Com a estrutura feita, pode-se criar o primeiro teste de unidade para o controlador. Como discutido neste trabalho, no TDD é importante que comecemos pelos testes para depois partir para a implementação.

Segue a definição do comportamento que espera-se que o controlador tenha:

- O controlador será responsável por receber o modelo *contato* da visão, e executar uma função que realizará uma requisição POST ao servidor, para que este faça a persistência do contato na base de dados.
- Caso a requisição obtenha uma resposta de sucesso, deve-se redirecionar o usuário para a rota da página de listagem de contatos (/listagem-contatos).
- Caso a requisição obtenha uma resposta de falha, será exibido um alerta e o usuário continuará na mesma página onde se encontra.

Decidido este comportamento, segue o teste implementando e comentado:

```

1. describe('Controller: AdicionarContatoCtrl', function () {
2.
3.     // Carrega o módulo do controlador
4.     beforeEach(module('minhaAgendaApp'));
5.
6.     //Define as variáveis
7.     var adicionarContatoCtrl,
8.         $loc,
9.         contatosServiceClientMock,
10.        scope,
11.        q;
12.
13.    // Este bloco será executado antes da execução de cada teste
14.    beforeEach(inject(function ($controller, $rootScope, $location,
15.        $q, contatosServiceClient) {
16.        adicionarContatoCtrl = $controller('AdicionarContatoCtrl', {});
17.        scope = $rootScope;
18.        $loc = $location;
19.        q = $q;
20.        contatosServiceClientMock = contatosServiceClient;
21.        //Simulamos uma resposta para o método adicionarContato
22.        //do nosso serviço de comunicação com o servidor.

```

```

22.
23.   });
24.
25.   //Este teste simula uma chamada ao método de adição de contatos do controlador.
26.   //A resposta do servidor deve ser de sucesso e o controlador redireciona o usuário
27.   //para a rota /main
28.   it('deve ser direcionado para a $location.path() /main', function () {
29.       //Simulamos uma resposta de sucesso para o método adicionarContato
30.       //do nosso serviço de comunicação com o servidor.
31.       spyOn(contatosServiceClientMock, "adicionarContato").and.callFake(function() {
32.           var deferred = q.defer();
33.           deferred.resolve();
34.           return deferred.promise;
35.       });
36.       //Verificamos que a $location.path() está vazia
37.       expect($loc.path()).toEqual('');
38.       //Criamos o objeto contato
39.       adicionarContatoCtrl.contato = {nome: 'Pedro', telefone: '99999-8888'}
40.       //Chamamos o método do controlador
41.       adicionarContatoCtrl.adicionar();
42.       //Dispatchamos a promise se sucesso
43.       scope.$apply();
44.       //Verificamos se a $location.path() foi alterada pelo controlador
45.       expect($loc.path()).toEqual('/main');
46.   });
47.
48.   //Este teste simula uma chamada ao método de adição de contatos do controlador.
49.   //A resposta do servidor deve ser de falha e o controlador não irá manter o usuário
50.   //na mesma rota
51.   it('deve se manter na mesma $location.path()', function () {
52.       //Simulamos uma resposta de falha para o método adicionarContato
53.       //do nosso serviço de comunicação com o servidor.

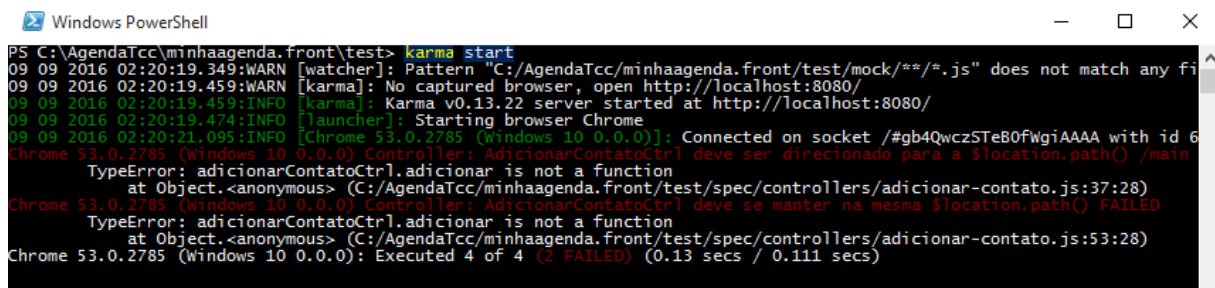
```

```

54.     spyOn(contatosServiceClientMock, "adicionarContato").and.callFake(function() {
55.         var deferred = q.defer();
56.         deferred.reject();
57.         return deferred.promise;
58.     });
59.     //Verificamos que a $location.path() está vazia
60.     expect($loc.path()).toEqual('');
61.     //Criamos o objeto contato
62.     adicionarContatoCtrl.contato = {nome: 'Pedro', telefone: '99999-8888'}
63.     //Chamamos o método do controlador
64.     adicionarContatoCtrl.adicionar();
65.     //Dispachamos a promise se sucesso
66.     scope.$apply();
67.     //Verificamos se a $location.path() continua vazia
68.     expect($loc.path()).toEqual('');
69. });
70.
71. });

```

Para executarmos os testes que se navegue até o diretório de *test/* e que seja executado o comando `karma start`. Como o controlador ainda não foi implementado, os dois testes irão falhar.



```

Windows PowerShell
PS C:\AgendaTcc\minhaagenda.front\test> karma start
09 09 2016 02:20:19.349:WARN [watcher]: Pattern "C:/AgendaTcc/minhaagenda.front/test/mock/**/*.js" does not match any fi
09 09 2016 02:20:19.459:WARN [karma]: No captured browser, open http://localhost:8080/
09 09 2016 02:20:19.474:INFO [launcher]: Starting browser Chrome
09 09 2016 02:20:21.095:INFO [Chrome 53.0.2785 (Windows 10 0.0.0)]: Connected on socket /#gb4QwczSTeB0fWgiAAAA with id 6
Chrome 53.0.2785 (Windows 10 0.0.0) Controller: AdicionarContatoCtrl deve ser direcionado para a $location.path() /main
TypeError: adicionarContatoCtrl.adicionar is not a function
    at Object.<anonymous> (C:/AgendaTcc/minhaagenda.front/test/spec/controllers/adicionar-contato.js:37:28)
Chrome 53.0.2785 (Windows 10 0.0.0) Controller: AdicionarContatoCtrl deve se manter na mesma $location.path() FAILED
TypeError: adicionarContatoCtrl.adicionar is not a function
    at Object.<anonymous> (C:/AgendaTcc/minhaagenda.front/test/spec/controllers/adicionar-contato.js:53:28)
Chrome 53.0.2785 (Windows 10 0.0.0): Executed 4 of 4 (2 FAILED) (0.13 secs / 0.111 secs)

```

Figura 20 Testes do controlador de adição de usuários falhando

4.4.4 Criando o controlador

Os controladores são responsáveis basicamente por controlar todo o fluxo de dados da aplicação, executando a maior parte do trabalho referente à UI (Interface do usuário).

Em uma aplicação *AngularJS*, eles têm a função acessar os dados no servidor, realizar a lógica de apresentação e controlar as interações com o usuário. Deste modo, os controladores ditam que dados devem ser exibidos, que partes da interface do usuário serão mostradas, o que acontece quando os usuários clicam em algo etc.

Os controladores atuam como um *gateway* entre os nossos modelos, que são os dados que ditam comportamento da nossa aplicação e a visão, que corresponde à UI. Deste modo, um controlador deve sempre ser utilizados pela interface do usuário.

Definindo um controlador

Todo módulo do AngularJS possui uma função *controller*, que é utilizada para definirmos um controlador.

O Yeoman (Angular Generator) criou o controlador quando a rota foi definida. Segue o código do controlador já implementado para que os testes passem:

```
1. angular.module('minhaAgendaApp')
2.   .controller('AdicionarContatoCtrl', function (contatosServiceClient, $location) {
3.
4.       //Métodos que chama o nosso serviço de comunicação
5.       //com o servidor e trata a sua resposta
6.       this.adicionar = function () {
7.           contatosServiceClient.adicionarContato(self.contato)
8.           .then(
9.               function (resposta) {
10.                  $location.path('/listagem-contatos');
```

```

11.     },
12.     function (respostaErro) {
13.         alert("Erro ao adicionar usuário");
14.     }
15. )
16. }
17. });

```

Pode-se ver que a função *controller* recebe como primeiro argumento o nome do controlador: `'AdicionarContatoCtrl'`. O segundo é um *array*, que aceita como primeiros argumentos todas as dependências do controlador em forma de variáveis do tipo *string*. O último argumento do array corresponde à função do controlador.

As variáveis definidas com a palavra-chave `this` são acessíveis na visão, já as variáveis locais e internas do controlador não. Deste modo, a função `adicionar()` do controlador estará disponível na visão.

O objeto `this.contato` será criado automaticamente assim que seus valores forem sendo definidos na visão. A atualização destes valores se dá em tempo real, devido ao *data binding* bidirecional (*two-way data binding*) presente no *AngularJS*.

Na *Linha 4* começa a definição do método `adicionar()`. Ele chama o serviço que faz comunicação com o back-end e trata suas respostas através do método `then()` que tem como primeiro argumento a função que será executada em caso de *sucesso* e a função que será executada em caso de *falha*.

Se o contato for adicionado com sucesso, redirecionamos o usuário para a *path* (`/listagem-contatos`), caso haja um erro ao adicionar, manteremos o usuário na mesma *path* e disparamos um alerta na tela do usuário comunicando sobre o erro.

Com o controlador implementado, pode-se executar os testes utilizando o Karma para verificar se ele está se comportando da forma desejada.

```
Windows PowerShell
PS C:\AgendaTcc\minhaagenda.front\test> karma start
09 09 2016 02:22:34.242:WARN [watcher]: Pattern "C:/AgendaTcc/minhaagenda.front/test/mock/**/*.js" does not match any fi
09 09 2016 02:22:34.290:WARN [karma]: No captured browser, open http://localhost:8080/
09 09 2016 02:22:34.310:INFO [karma]: Karma v0.13.22 server started at http://localhost:8080/
09 09 2016 02:22:34.323:INFO [launcher]: Starting browser Chrome
09 09 2016 02:22:36.009:INFO [Chrome 53.0.2785 (Windows 10 0.0.0)]: Connected on socket /#rT7uIcx-J3ryxYMwAAAA with id 8
Chrome 53.0.2785 (Windows 10 0.0.0): Executed 4 of 4 SUCCESS (0.252 secs / 0.225 secs)
```

Figura 21 Testes de unidade do controlador de adição de usuários sendo executado com sucesso

4.4.5 Criando a Visão

Com o arquivo *adicionar-contato.html* já criado pelo Yeoman (Angular Generator), deve-se implementar a visão para adição de contatos. Para isto, foi definido como será a sua estrutura e funcionamento:

- A visão deve conter um formulário com dois campos: Nome e telefone.
- O preenchimento do campo nome é obrigatório.
- O campo telefone deve ter no máximo 9 caracteres.
- O botão de Adicionar estará bloqueado até que o formulário esteja preenchido corretamente.
- O botão Adicionar deverá chamar a função *adicionar()* do controlador.
- Deve haver um link para o usuário ser direcionado para página de listagem de contatos.

Definido isso, segue a implementação da visão:

1. `<div class="jumbotron">`
2. `<h2>Adicionar contato</h2>`
3. `<form name="novoContatoForm" ng-submit="adicionarContato.adicionar()">`
- 4.
5. `<div class="form-group">`


```

6.         <input type="text" name="nome" id="nome" class="form-control"
placeholder="Nome" ng-model="adicionarContato.contato.nome" required />
7.     </div>
8.
9.     <div class="form-group">
10.         <input type="text" name="telefone" id="telefone" class="form-control"
placeholder="Telefone" ng-model="adicionarContato.contato.telefone" ng-minlength="8"
ng-maxlength="9" required/>
11.     </div>
12.
13.     <div class="form-actions">
14.         <button id="adicionar" type="submit" ng-
disabled="novoContatoForm.$invalid" class="btn btn-primary">Adicionar</button>
15.         <a ng-href="#/listagem-contatos" class="btn btn-link">Ver contatos</a>
16.     </div>
17. </form>
18. </div>

```

Como foi definido no roteamento, o controlador será referenciado pelo nome *adicionarContato* na visão.

Na linha 3 definimos o formulário *novoContatoForm*. Nele, foi colocada a diretiva `ng-submit="adicionarContato.adicionar()"`. Ela diz que quando o formulário for submetido, a função *adicionar()* do controlador será chamada.

Na linha 5 foi definido o *input* do referente ao *Nome*. Ao ser adicionada a diretiva `ng-model="adicionarContato.contato.nome"`, define-se que o valor daquele input será o valor da propriedade *Nome* do objeto *Contato* do controlador. Ao colocarmos a diretiva `required`, o campo torna-se obrigatório e enquanto ele não for preenchido, o formulário é inválido.

Na linha 10 é definido o *input* do referente ao *Telefone*. Ao ser adicionada a diretiva `ng-model="adicionarContato.contato.telefone"`, define-se que o valor daquele input será o valor da propriedade *Telefone* do objeto *Contato* do nosso controlador. Ao se colocar as diretivas `ng-`

`maxlength="9"` e `required` define-se que o comprimento máximo do telefone é de 9 dígitos e que o preenchimento é obrigatório. Até que as duas condições sejam atendidas, o formulário é considerado inválido.

Na linha 14 temos a definição do botão para submeter o formulário. A diretiva `ng-disabled="novoContatoForm.$invalid"` diz que o botão deve permanecer desabilitado até que o formulário seja válido.

Na linha 15 está definida a diretiva `ng-href="#/listagem-contatos"`, que gera um *link* que redireciona usuário para a rota de listagem de usuários.

Este trecho de código nos mostra a *programação declarativa* que o AngularJS (e outros frameworks JavaScript MVC) proporciona. Apenas declara-se o comportamento que a visão deve ter. Fica a cargo do AngularJS definir como isso será feito e realizar as tarefas descritas.

A Figura 22 mostra que o botão *Adicionar* está desabilitado, já que a visão não está preenchida corretamente.

Figura 22 Botão "Adicionar" desabilitado devido ao preenchimento incorreto do formulário

Um fato interessante é que as validações são realizadas a tempo real. Assim que o formulário for preenchido corretamente, o botão se torna habilitado, como mostra a Figura 23. Em nenhum momento há a recarga da página.

A imagem mostra uma interface web com uma barra de navegação no topo contendo os links 'minhaAgenda', 'Home', 'About' e 'Contact'. Abaixo, há um formulário centralizado com o título 'Adicionar contato'. O formulário possui dois campos de entrada: o primeiro contém o nome 'Gabriel Menezes' e o segundo contém o número '123456789'. Abaixo dos campos, há dois botões: 'Adicionar' (em azul escuro com um ícone de cursor) e 'Listar contatos' (em azul claro). No rodapé, há uma mensagem '♥ from the Yeoman team'.

Figura 23 Botão "Adicionar" habilitado devido ao preenchimento correto do formulário

4.5 - Implementando a listagem dos usuários

Nesta seção será implementada a listagem de todos os contatos e uma busca que filtra em tempo real os contatos pelo nome. Com isso, serão expostos dois novos tópicos importantes do AngularJS: as Diretivas (directives) e os Filtros (filters).

Diretivas

De acordo com Branas (2014, p.18), “Uma diretiva é uma extensão do vocabulário *HTML* que permite ao desenvolvedor a criação de novos comportamentos. Essa tecnologia permite a criação de componentes reutilizáveis para toda a aplicação. ”. É através das diretivas que os dados modificados são espelhados na visão e por onde os controladores recebem as ações realizadas pelo usuário.

Filtros

Utilizados nas *visões*, os filtros são aplicados para formatar os dados apresentados ao usuário. Quando um filtro é definido em um módulo, ele pode ser utilizado por todas as visões e controladores daquele módulo. O *AngularJS* possui filtros nativos, como por exemplo o *currency*, que recebe um número e formata para dólar, como também permite a criação de filtros personalizados.

4.5.1 - Criando a estrutura da funcionalidade

Do mesmo modo que foi feito anteriormente para a adição de novos contatos, deve-se o seguinte comando no Yeoman (Angular Generator):

```
yo angular:route listagem-contatos
```

Deste modo, o Yeoman (Angular Generator) realizou as seguintes tarefas:

6. Referenciou o arquivo JavaScript do nosso Controlador (`app/scripts/controllers/listar-contatos.js`) no `Index.html`
7. Criou o arquivo JavaScript do Controlador : `app/scripts/controllers/listar-contatos.js`
8. Preparou o esqueleto de testes para o Controlador : `test/spec/controllers/listar-contatos.js`

9. Cria a rota da View (Visão) na configuração `$routeProvider` do Módulo principal, referenciando nosso controlador à ela.
10. Cria a Visão: `app/views/listar-contatos.html`

4.5.2 - Criação da rota

Como foi visto na seção de adição de usuário, o *Yeoman* (Angular Generator), automaticamente cria a nova rota na configuração `$routeProvider` do nosso módulo.

A rota então fica definida da seguinte maneira no arquivo `app/app.js`:

```
1. .when('/listagem-contatos', {
2.   templateUrl: 'views/listagem-contatos.html',
3.   controller: 'ListagemContatosCtrl',
4.   controllerAs: 'listagemContatos',
5.   resolve: {
6.     contatos: function (contatosServiceClient) {
7.       return contatosServiceClient.listarContatos();
8.     }
9.   }
10. })
```

Na linha 1 encontra-se o primeiro `when()`. Ele define que quando a *URL* for `'localhost/#/listagem-contatos'`, a visão `listagem-contatos.html` será renderizada. Diz também que o controlador ligado a essa rota é o `ListagemContatosCtrl` e este será chamado de `listagemContatos` na visão.

Na linha 5 encontra-se o `resolve`. Assim que a rota é chamada, ele executa o método `listarContatos()` do serviço `contatosServiceClient` que se comunica com a API. Este método devolve um array de contatos para o objeto `contatos`, que estará disponível na visão.

4.5.3 - Criando o teste de unidade para o controlador

Com a estrutura feita, pode-se criar o primeiro teste de unidade para o controlador.

Como discutido neste trabalho, no TDD é importante que comecemos pelos testes para depois partir para a implementação.

Foi definido o comportamento que o controlador deve ter:

- Após o controlador receber a lista de contatos através do resolve da rota, ele deve disponibilizar esta lista para que a visão consiga acessá-la.

Decidido este comportamento, segue a implementação do teste com seus comentários:

```
1. describe('Controller: ListagemContatosCtrl', function () {
2.
3.     // Carrega o módulo principal
4.     beforeEach(module('minhaAgendaApp'));
5.
6.     // Define as variáveis
7.     var ListagemContatosCtrl,
8.         scope;
9.
10.    // Inicializa o controlador
11.    beforeEach(inject(function ($controller, $rootScope) {
12.        scope = $rootScope.$new();
13.        ListagemContatosCtrl = $controller('ListagemContatosCtrl', {
14.            $scope: scope,
15.            //Inicializa o objeto listaDeContatos, como seria feito pela rota
16.            listaContatos: {data: [{nome: 'Bira', telefone: '99999-8888'},
17.                                   {nome: 'Gabriel', telefone: '99999-7777'}]}}
18.        ));
19.    }));
20.
```

```

21. //Este teste verifica se o modelo "contatos" contém uma
22. //lista de contatos quando está é inserida em seu scope
23. it('o modelo "contatos" deve estar preenchido com um lista de contatos', function ()
    {
24.     //Verifica se os dados presentes no objeto "contatos" do controlador
25.     //foram preenchidos corretamente
26.     expect(ListagemContatosCtrl.contatos)
27.     .toEqual([
28.         {nome: 'Bira', telefone: '99999-8888'},
29.         {nome: 'Gabriel', telefone: '99999-7777'}
30.     ]);
31. });

```

Para executar os testes basta que se navegue até o diretório de *test/* e se execute o comando `karma start`. Como o controlador ainda não foi implementado os testes irão falhar.

```

PS C:\AgendaTcc\minhaagenda.front\test> karma start
09 09 2016 03:18:19.848:WARN [watcher]: Pattern "C:/AgendaTcc/minhaagenda.front/test/mock/**/*.js" does not match any fi
09 09 2016 03:18:19.926:INFO [karma]: Karma v0.13.22 server started at http://localhost:8080/
09 09 2016 03:18:19.926:INFO [launcher]: Starting browser Chrome
09 09 2016 03:18:22.285:INFO [Chrome 53.0.2785 (Windows 10 0.0.0)]: Connected on socket /#fv-1X1za9TetSxFoAAAA with id 1
ALERT: 'Erro ao adicionar usuário'
Chrome 53.0.2785 (Windows 10 0.0.0) Controller: ListagemContatosCtrl o modelo "contatos" deve estar preenchido com um l
atos FAILED
Expected undefined to equal [ Object({ nome: 'Bira', telefone: '99999-8888' }), Object({ nome: 'Gabriel', telefo
7777' }) ].
at Object.<anonymous> (C:/AgendaTcc/minhaagenda.front/test/spec/controllers/listagem-contatos.js:27:6)
Chrome 53.0.2785 (Windows 10 0.0.0): Executed 5 of 5 (1 FAILED) (0.157 secs / 0.124 secs)

```

Figura 24 Testes do controlador de exibição da lista de usuários falhando

4.5.4 - Criando o controlador

Com o comportamento definido e o teste escrito, o controlador pode ser implementado.

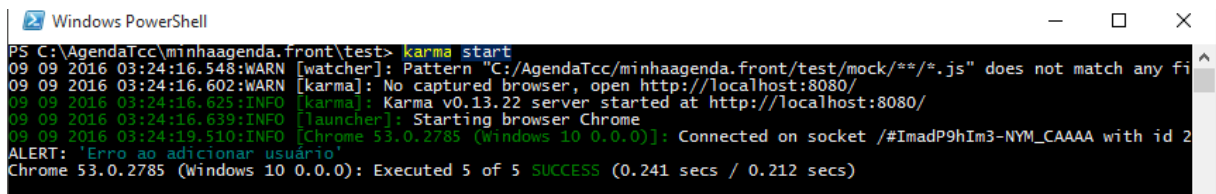
Segue o código do controlador comentado:

```

1. angular.module('minhaAgendaApp')
2.   .controller('ListagemContatosCtrl', function (listaContatos, $scope) {
3.     //Disponibiliza o modelo "contatos" para a visão
4.     this.contatos = listaContatos.data;
5.   });

```

Com o controlador implementado, o Karma pode ser executado para verificação dos testes de unidade.



```
PS C:\AgendaTcc\minhaagenda.front\test> karma start
09 09 2016 03:24:16.548:WARN [watcher]: Pattern "C:/AgendaTcc/minhaagenda.front/test/mock/**/*.js" does not match any fi
09 09 2016 03:24:16.602:WARN [karma]: No captured browser, open http://localhost:8080/
09 09 2016 03:24:16.625:INFO [karma]: Karma v0.13.22 server started at http://localhost:8080/
09 09 2016 03:24:16.639:INFO [launcher]: Starting browser Chrome
09 09 2016 03:24:19.510:INFO [Chrome 53.0.2785 (Windows 10 0.0.0)]: Connected on socket /#ImadP9hIm3-NYM_CAAAA with id 2
ALERT: 'Erro ao adicionar usuário'
Chrome 53.0.2785 (Windows 10 0.0.0): Executed 5 of 5 SUCCESS (0.241 secs / 0.212 secs)
```

Figura 25 Testes obtendo sucesso após implemetação do controlador de exibição de lista de contatos

4.5.5 - Criando a visão

Com o arquivo `listagem-contatos.html` já criado pelo Yeoman (Angular Generator), deve-se implementar a visão. Para isto, foi definido como será a sua estrutura e funcionamento:

- A visão deve receber uma lista de contatos e exibi-los em uma lista
- Haverá um campo de busca, pelo qual o usuário poderá filtrar os contatos pelos seus nomes
- Haverá um link para redirecionar o usuário para a rota de adição de usuários.

Definido isso, segue a implementação da visão:

1. `<div class="jumbotron">`
2. `<input class="form-control" type="text" ng-model="criterioDeBusca" placeholder="0`
`que você está buscando?"/>`
- 3.
4. `<table ng-show="listagemContatos.contatos.length > 0" class="table table-striped">`
5. `<tr>`
6. `<th>Nome</th>`
7. `<th>Telefone</th>`
8. `</tr>`
9. `<tr ng-repeat="contato in listagemContatos.contatos | filter:criterioDeBusca ">`
10. `<td>{{contato.nome}}</td>`
11. `<td>{{contato.telefone}}</td>`


```
12.         </tr>
13.     </table>
14.     <hr/>
15.     <a class="btn btn-primary btn-block" href="#/adicionar-contato">Novo Contato</a>
16. </div>
```

Na Linha 2 foi definido um *input* que será o critério de busca, no caso o nome do contato. Nele há a diretiva `ng-model="critérioDeBusca"`. O modelo definido nessa diretiva terá como valor o que for digitado no *input*.

Na Linha 4, utilizando a diretiva `ng-show="listagemContatos.contatos.length > 0"`, define-se que a tabela só será exibida caso exista algum contato na lista.

Na Linha 9, ao ser utilizada a diretiva `ng-repeat="contato in listagemContatos.contatos | filter:critérioDeBusca "` é enviado um comando para o Angular percorrer toda a lista de contatos. Ao mesmo tempo foi aplicado um filtro chamado *filter*, que filtra a lista e acordo com um critério. Deste modo, só serão exibidos os contatos cujos nomes contenham a cadeia de caracteres presente no modelo `critérioDeBusca`.

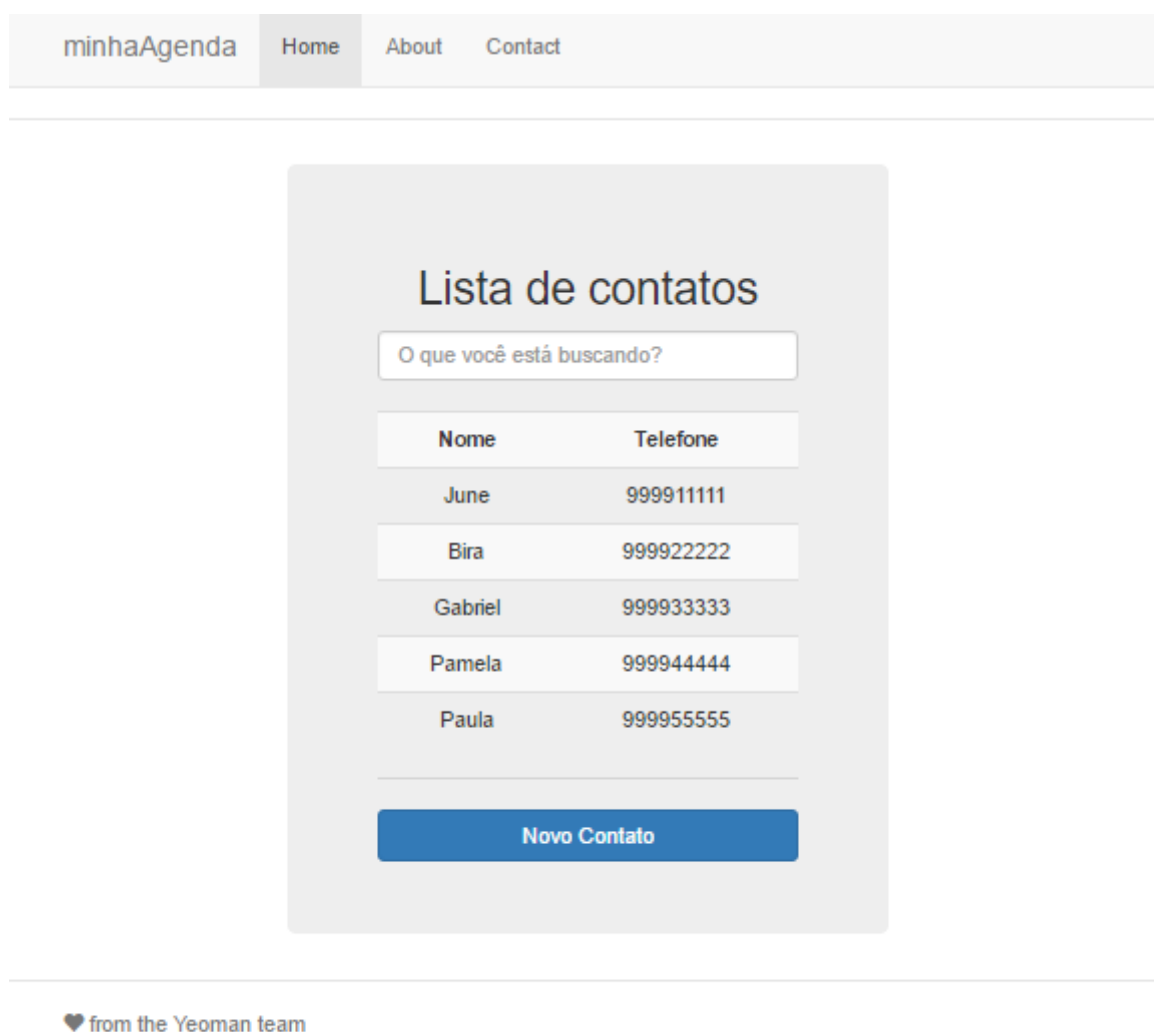


Figura 26 Visão da listagem de contatos

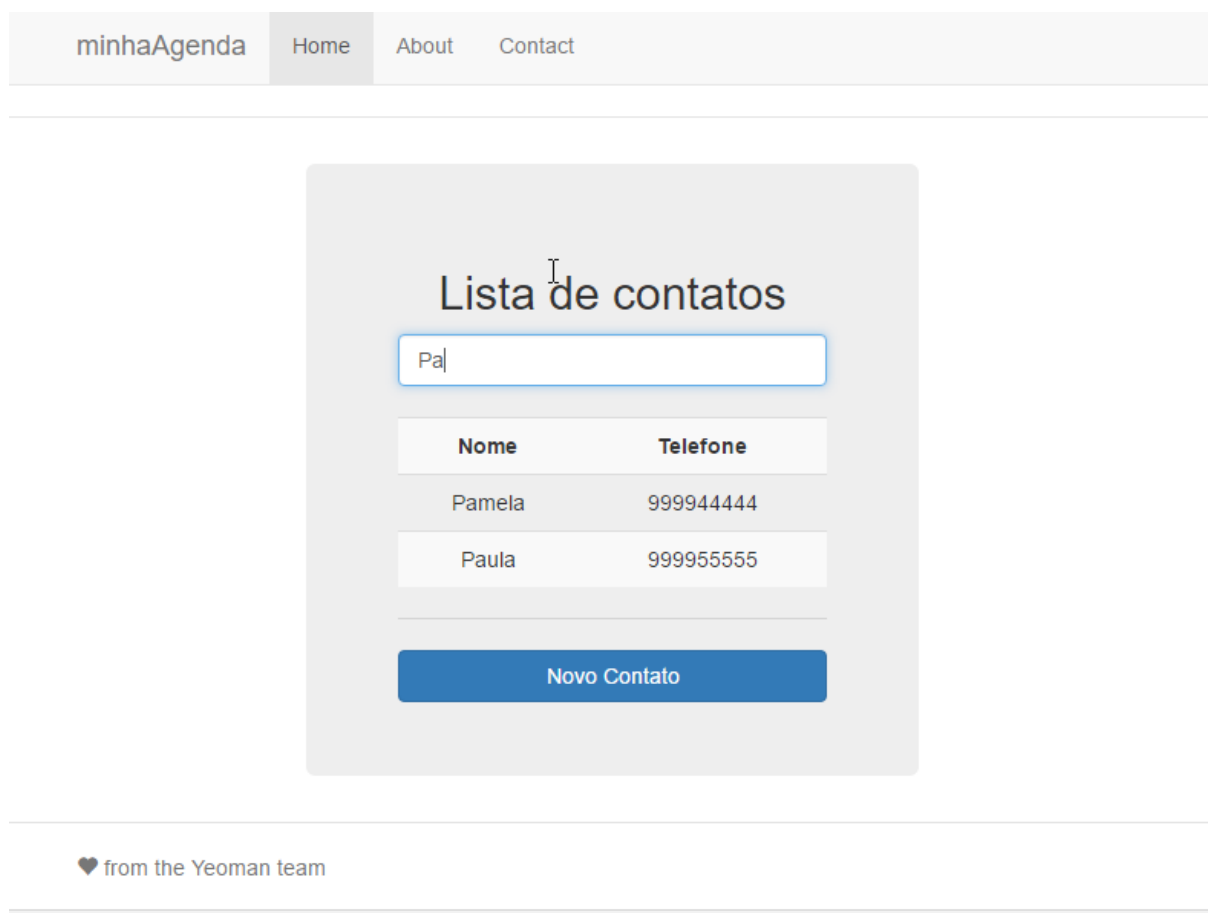


Figura 27 Listagem de contatos filtrada

4.6 - Utilização de interceptadores

O AngularJS simplifica muito o tratamento de ações a nível de solicitações, como *logging*, verificação de autenticação tratamento de respostas. Utilizando o *\$httpProvider*, conseguimos criar interceptadores, que permitem efetuar um *hook*, verificando todas as solicitações e respostas para realizar devidos tratamentos.

De uma maneira resumida, os interceptadores podem interceptar as solicitações *\$http* antes que sejam enviadas ao servidor, assim como as respostas antes que sejam recebidas pelo chamador.

Será utilizado como exemplo o *LoadingInterceptor* da aplicação. Ele é responsável por definir o parâmetro global *\$loading* , definindo como *verdadeiro* logo antes da requisição ser enviada e como *falso* assim que a resposta for recebida. A partir desse parâmetro *booleano* pode-se definir se uma tela de loading será exibida, por exemplo.

LoadingInterceptor

```
1. angular.module('minhaAgendaApp')
2.   .factory('loadingInterceptor', function ($q, $rootScope) {
3.     return{
4.
5.       request: function (config) {
6.         $rootScope.loading = true;
7.         return config;
8.       },
9.
10.      requestError: function (rejection) {
11.        $rootScope.loading = false;
12.        return $q.reject(rejection);
13.      },
14.
15.      response: function (response) {
16.        $rootScope.loading = false;
17.        return response;
18.      },
19.
20.      responseError: function (rejection) {
21.        $rootScope.loading = false;
22.        return $q.reject(rejection);
23.      }
24.    };
25.  });
```

Os interceptadores são implemetados como *factories*, que retornam um objeto que pode retornar até 4 métodos específicos: *request*, *requestError*, *response* e *responseError*.

Segue uma visão geral de cada um deles:

request

Toda solicitação de saída passa por esse método. Ele recebe o parâmetro *config*, que é a configuração com a qual a requisição foi feita, permitindo acessar e manipular toda a informação da solicitação. Caso se queira que solicitação prossiga, deve-se retornar o argumento *config*. Para impedir que ela seja relizada, deve-se retornar *\$q.reject*.

requestError

É disparada caso algum outro interceptador rejeite a requisição. Ele recebe o parâmetro *\$q.reject*.

response

Esta função é chamada sempre que o servidor retornar alguma resposta. Recebe como parâmetro o objeto *response*, que contém a configuração, o código de status, os cabeçalhos e os dados.

responseError

Se o servidor retornar algum código que seja diferente de algum número entre 200-299, este método é chamado. Ou seja, sempre que o status da resposta não for de sucesso, pode-se acessar e manipular o argumento *reject*, realizando os devidos tratamentos.

Com os interceptadores definidos, precisa-se adicioná-los ao *\$httpProvider*, utilizando a configuração do *módulo*. Vejamos o exemplo de aplicação de exemplo:

```
1. angular.module('minhaAgendaApp')
2.   .config(function ($httpProvider) {
3.     $httpProvider.interceptors.push("loadingInterceptor");
4.   });
```

O ideal é que os interceptors sejam criados com responsabilidades individuais, garantindo maior modularidade do código e mais fácil manutenção.

4.7. Criando testes fim-a-fim com Protractor

Nesta seção será configurado um teste fim-a-fim para a agenda. Estes testes abrem o navegador, acessam uma versão ativa em execução da nossa aplicação e efetuam cliques, digitações e verificações do mesmo modo que um usuário real o faria. Para este fim, foi utilizada a ferramenta Protractor. Primeiro foi criada uma configuração para este framework. Posteriormente o teste foi criado e executado.

4.7.1 instalando o Protractor

O Protractor (AMORIM, 21666) é um pacote *NodeJS* e deve ser instalado a partir do seguinte comando:

```
npm install -g protractor
```

Deste modo o Protractor é instalado junto com todas as suas dependências como um pacote global.

O Protractor instalado fornece os scripts necessários para a instalação do *WebDriver*, que é utilizado para iniciar e controlar os navegadores em que os testes serão executados.

Para instalá-lo, deve-se executar o seguinte comando:

```
webdriver-manager update
```

Desta forma o ambiente está configurado para o uso da ferramenta de testes fim-a-fim.

4.7.2 - Criando a configuração do Protractor

O Protractor (AMORIM, 2016) necessita de um arquivo JavaScript de configuração para executar os testes. Este arquivo contém diversos elementos de configuração para o funcionamento da ferramenta, como por exemplo em quais navegadores os testes devem ser executados, em qual endereço o *Selenium WebDriver* se encontra, em qual endereço o servidor a ser testado se encontra etc.

O arquivo de configuração foi chamado de *protractor.conf.js*. Segue sua implementação comentada.

```
1. //Exemplo de arquivo de configuração do Protractor
2. exports.config = {
3.
4.     //O endereço que o Selenium WebDriver está rodando
5.     seleniumAddress: 'http://localhost:4444/wd/hub',
6.
7.     //O endereço que a aplicação a ser testada esta rodando
8.     baseUrl: 'http://localhost:9000/',
9.
10.    // Os testes serão executados no Chrome
11.    capabilities: {
12.        'browserName': 'chrome'
13.    },
14.
```

```

15. //Arquivo onde o teste de encontra
16. specs: ['spec.js'],
17.
18. //Opções a serem passadas ao Jasmine
19. jasmineNodeOpts: {
20.   showColors: true,
21.   defaultTimeoutInterval: 30000
22. }
23. };

```

4.7.3. Criando o teste fim-a-fim

Os testes no Protractor (AMORIM, 2016) utilizam a mesma sintaxe estrutural do Jasmine, que foi utilizado para a criação de testes de unidade. Logo, existe o mesmo bloco *describe*, para um conjunto de testes que são definidos nos blocos *it*.

Antes de escrever o teste foram definidas as ações que este deve realizar:

- Abrir o navegador Chrome
- Navegar até a URL para adição de contatos
- Verificar se está na URL correta
- Digitar um nome e um telefone para o novo contato
- Clicar no botão de adicionar contato
- Verificar se o usuário foi redirecionado para pagina de listagem de contatos
- Pegar o último contato da lista de contatos
- Verificar se os dados do contato adicionado batem com o último contato da lista

Com o comportamento bem definido, foi criado o arquivo de teste chamado *spec.js*. Segue a sua implementação comentada:


```

1. // Arquivo 'spec.js.' com a descrição do teste fim-a-fim
2. describe('Testes fim-a-fim', function(){
3.
4.     //O teste adiciona um contato, vai até a pagina de listagem e
5.     //confere de o contato foi adicionado corretamente
6.     it('Realiza a adicao de um contato e confere se foi adicionado', function () {
7.
8.         //Acessa a URL para adição de novos contatos
9.         browser.get('#/adicionar-contato');
10.
11.        //Garante que o usuário foi redirecionado para a
12.        //página de adição de usuário
13.        expect(browser.getCurrentUrl())
14.            .toEqual('http://localhost:9000/#/adicionar-contato');
15.
16.        var nome = element(
17.            by.model('adicionarContato.contato.nome'));
18.
19.        var telefone = element(
20.            by.model('adicionarContato.contato.telefone'));
21.
22.        //Digita um novo contato (nome e telefone)
23.        nome.sendKeys('Pamela');
24.        telefone.sendKeys('999944444');
25.
26.        //Clica no botão de adicionar
27.        element(by.id('adicionar')).click();
28.
29.        //Garante que o contato foi redirecionado para a

```

```

30.      //página de listagem de contatos
31.      expect(browser.getCurrentUrl())
32.          .toEqual('http://localhost:9000/#/listagem-contatos');
33.
34.      //Pega a quantidade atual de contatos
35.      var qtdContatos = element.all(by.repeater('contato in listagemContatos'));
36.
37.      //Pega o nome do último contato da lista
38.      var nomeUltimoContato = element.all(
39.          by.repeater('contato in listagemContatos').column('contato.nome'))
40.          .last();
41.
42.      //Pega o telefone do último contato da lista
43.      var telefoneUltimoContato = element.all(
44.          by.repeater('contato in listagemContatos').column('contato.telefone'))
45.          .last();
46.
47.      //Verifica se o nome e telefone do último contato da lista
48.      //confere com o último contato adicionado
49.      expect(nomeUltimoContato.getText()).toEqual('Pamela');
50.      expect(telefoneUltimoContato.getText()).toEqual('999944444');
51.
52.  })
53. });

```

4.7.4 - Executando o teste fim-a-fim

O teste foi escrito como se um usuário estivesse interagindo com a aplicação. Cabe ao *Protractor* e ao *AngularJS* realizar estas interações e as verificações.

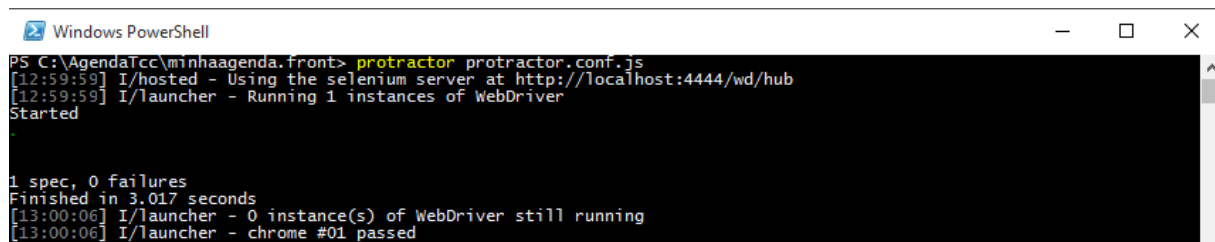
Para executar os testes deve-se seguir os seguintes passos:

1. Garantir que o servidor da nossa aplicação está no ar.

2. Colocar o serviço do Selenium no ar executando o comando `webdriver-manager start`
3. Navegar até a pasta que contém o arquivo de configuração e as especificações e executar o comando `protractor protractor.conf.js`

Após executar este último comando, o Protractor abrirá o navegador e realizará os testes.

A Figura X nos mostra o resultado do teste.



```
PS C:\AgendaTcc\minhaagenda.front> protractor protractor.conf.js
[12:59:59] I/hosted - Using the selenium server at http://localhost:4444/wd/hub
[12:59:59] I/launcher - Running 1 instances of WebDriver
Started
.
1 spec, 0 failures
Finished in 3.017 seconds
[13:00:06] I/launcher - 0 instance(s) of WebDriver still running
[13:00:06] I/launcher - chrome #01 passed
```

Figura 28 Teste fim-a-fim executado com sucesso

4.8 - Build utilizando o Grunt

O *Yeoman* (Angular Generator) cria automaticamente um arquivo de configuração robusto do Grunt. Este arquivo possui todos os principais procedimentos de *build* para uma aplicação AngularJS, como:

- Renomear arquivos
- Concatenar arquivos
- Move e copiar arquivos
- Minificar o CSS e o JavaScript
- Executar testes

Para realizar o *build* da aplicação, basta executar o comando `grunt build` na pasta do projeto. Feito isso, o Grunt será executado e criará os arquivos finais de *build*, colocando-os em uma pasta chamada *dist*.

Na Figura 29 pode-se visualizar a estrutura de pastas da nossa aplicação em desenvolvimento.

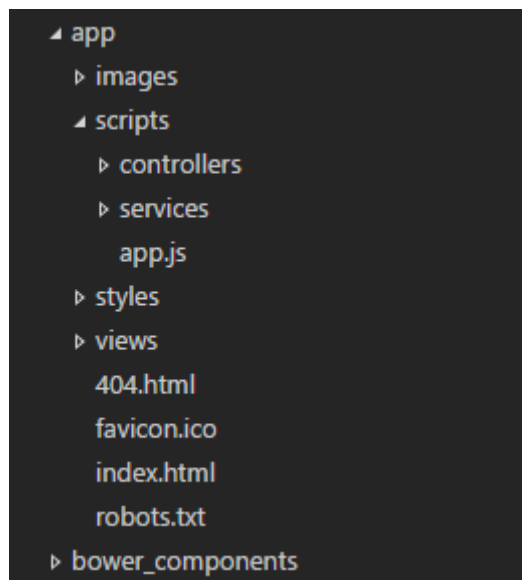


Figura 29 Estrutura de pastas do projeto em desenvolvimento

Na Figura 30, temos a estrutura de pastas na aplicação pós *build*.

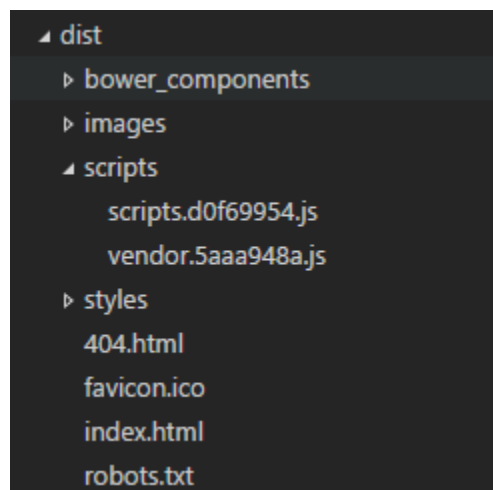


Figura 30 Estrutura de pastas da aplicação pós build

5 Conclusões

Neste trabalho foram apresentados diversos conceitos, tecnologias e ferramentas que possuem extrema importância no desenvolvimento web *front-end* moderno. Deste modo, o projeto pode ser utilizado como um guia para quem deseja se atualizar ou ingressar no novo mundo de desenvolvimento front-end.

Inicialmente foram expostos conceitos básicos sobre o tema. Feito isso, foi apresentado o estado atual em que se encontra o meio de desenvolvimento web front-end. Para tal, foram introduzidas as aplicações de página única (SPA), o padrão de arquitetura MVC e demonstrada a importância do desenvolvimento orientado a testes (TDD) ao desenvolvermos aplicações web front-end modernas.

Para adentrar o desenvolvimento dessas aplicações, foram analisados os três mais importantes *frameworks* e bibliotecas JavaScript MVC: *AngularJS*, *Backbone.js* e *Ember.js*. Foram apresentadas suas arquiteturas e realizado um estudo comparativo entre elas. Após uma análise detalhada de diversas características de cada uma, optou-se pela utilização do AngularJS para o desenvolvimento da aplicação de exemplo, principalmente por este estar sendo cada vez mais adotado pelo mercado e reconhecido pela comunidade de desenvolvedores front-end.

Devido ao aumento de dispositivos mais variados, que exigem um design mais arrojado que os websites, a complexidade dos requisitos e necessidades a respeito do CSS vem crescendo. Com isso, foram apresentados também os pré-processadores de CSS mais famosos: Sass e LESS. Foi demonstrada a utilização básica de cada um e realizado também um estudo comparativo entre eles. Por fim, optou-se pela utilização do Sass, que possui o *framework* Compass, que é bastante utilizado e auxilia mais ainda a manipulação das folhas de estilo.

O aumento da complexidade das aplicações web, aumentam também o número de bibliotecas de terceiros que são utilizadas, tornando-se difícil o gerenciamento destas. Devido a isso, foi apresentado o gerenciador de dependência chamado *Bower*, que automatiza todas as tarefas referentes à administração e atualização das bibliotecas utilizadas no projeto.

Foram apresentadas também ferramentas JavaScript para automatização de tarefas. Primeiramente o Yeoman e seu plug-in *Angular Generator*, que facilitam de forma expressiva o gerenciamento de projetos de médio e grande porte em AngularJS. Também foram compradas as ferramentas para automatização de processos de *build*, como o Grunt e o Gulp. Optou-se por utilizar o *Grunt* na aplicação de exemplo principalmente por ser mais simples e suportado pelo plug-in *Angular Generator*.

Foram apresentadas também diversas ferramentas que auxiliam o desenvolvedor a seguir o desenvolvimento orientado a testes em aplicações JavaScript MVC. Para a execução dos testes foi apresentada a ferramenta chamada Karma. Para a escrita de testes de unidade e integração foi apresentado o framework Jasmine. Para a criação de testes fim-a-fim o Protractor.

Por fim, foi implementada uma aplicação web simples utilizando o AngularJS e as outras ferramentas escolhidas. Com isso foi demonstrada a utilização básica de diversas ferramentas e tecnologias apresentadas no trabalho, assim como introduzidas diversas funcionalidades do *AngularJS*.

Referências

- AMORIM, Daniel. **Protractor: Testando Aplicações AngularJs com uma Solução Integradora**. Disponível em: <https://www.thoughtworks.com/pt/insights/blog/testing-angularjs-apps-protractor>. Acesso em: 24, ago. 2016.
- BATTISTI, Júlio. **SQL Server 2000: Administração e Desenvolvimento – Curso Completo**. 2. ed. Rio de Janeiro: Axcell Books, 2001.
- BECK, Kent, **Test-Driven Development By Example**. Addison Wesley, 2000.
Estados Unidos
- BRANAS, Rodrigo, **AngularJS Essentials**. 1. ed. UK, Birmingham, 2014.
- BRAITWAITE, Bradley. **Getting started with Karma for AngularJS Testing**.
Disponível em: <http://www.bradoncode.com/blog/2015/05/19/karma-angularjs-testing/>
Acesso em: 01, set, 2016.
- CRAVENS, Jesse. BRADY, Q. Thomas. **Building Web Apps with Ember.js**. 1. ed.
California: Sebastopol, 2014.
- FINK, Gil; FLATOW, Ido. **Pro Single Page Application Development: Using Backbone.js and ASP.NET**. 1. Ed Nova Iorque: Apress Media LLC., 2014.
- HARTL, Michael. **Ruby on Rails Tutorial: Learn Web Development with Rails**. 3. Ed. 2014.
- HAROLD, R. Elliotte. **Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX**. (2002)

JOUBRAN, Jad. **Beginners Guide: Getting Started with Bower Package Manager.**

Disponível em: <https://www.codementor.io/bower/tutorial/beginner-tutorial-getting-started-bower-package-manager>. Acesso em: 01 set. 2014.

LANGR, J., **Evolution of Test and Code Via Test-First Design**, 02.12.2010

LOPES, Sergio. **CSS fácil, flexível e dinâmico com LESS.** Disponível em:

<http://blog.caelum.com.br/css-facil-flexivel-e-dinamico-com-less/>. Acesso em: 10. Jul, 2016.

MAYNARD, Travis. **Getting Started with Gulp.** 1. ed. UK, Birmingham, 2015.

MCLAUGHLIN, Brett. **What Is Node? JavaScript Breaks Out of the Browser.**

Estados Unidos, 2011

MORGAN, Adam. **Testing AngularJS with Jasmine and Karma (Part 1).**

Disponível em: <https://scotch.io/tutorials/testing-angularjs-with-jasmine-and-karma-part-1>

Acesso em: 20, ago, 2016.

NAGAPPAN, N., Bhat, T. **Evaluating the efficacy of test- driven development: industrial case studies.** Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering.

OSMANI, Addy. **Developing Backbone.js Applications.** 1. Ed. (2011)

OSMANI, Addy. **Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide.** 1. Ed. (2012)

PILGRIM, Mark. **HTML5: Up and Running.** 1. Ed (2010)

PILLORA, Jaime. **Getting Started with Grunt: The JavaScript Task Runner.** 1. ed. UK, Birmingham, 2014.

POPLADE, Thaiana. **O que é Sass? Entenda esse outro método de escrever CSS.**

Disponível em: <http://tableless.com.br/sass-um-outro-metodo-de-escrever-css/>. Acesso em: 15, set, 2016.

SAHNI, V., **Best Practices for Designing a Pragmatic RESTful API**. Disponível em: <<http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>>. Acesso em: 01 set. 2016.

SCHMITT, Christopher. **CSS Cookbook**. 3. Ed. (2009)

SESHADRI, Shyam. GREEN, Brad. **Desenvolvendo com AngularJS**. Tradução de Lúcia A kinoshita. 1. Ed. São Paulo, 2014.

SPRATLEY, Jonnathan. **Learning Yeoman**. 1. ed. UK, Birmingham, 2014.

STEINBERG, D. H. **The Effect of Unit Tests on Entry Points, Coupling and Cohesion in an Introductory Java Programming Course**. XP Universe, Raleigh, North Carolina, USA, 2001.

STROPEK, Rainer. **AngularJS with TypeScript and Windows Azure Mobile Services, 2013**. Disponível em: <http://goo.gl/O2NENn> . Acesso em 24 jul. 2016

VASKEVITCH, David. **Estratégia Cliente/Servidor: um guia para a reengenharia da empresa**. São Paulo: Berkeley, 1995.

WOYCHWSKY, Edmond. **AJAX: Creating Web Pages with Asynchronous JavaScript and XML**. 1. Ed. (2006)