UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
ESCOLA DE INFORMÁTICA APLICADA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

VICTOR MAGALHÃES SILVA DE SOUZA

Lâmpada Mágica - *Framework para criação de ambientes de front-end*

Prof. Dr. José Ricardo da Silva Cereja
Orientador

Rio de Janeiro, Janeiro de 2016

Lâmpada Mágica - *Framework para criação de ambientes de front-end*

Victor Magalhães Silva de Souza

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Aprovado por:

____

Prof. Dr. José Ricardo da Silva Cereja (Orientador)

____

Prof. Ricardo Rodrigues Nunes

Rio de Janeiro, Janeiro de 2016

Dedico este projeto à minha família.

AGRADECIMENTOS

Gostaria de agradecer, primeiramente, à minha família e meus amigos mais próximos. Pessoas incríveis que me deram total apoio nessa caminhada que foi o aprendizado. Em especial, meus pais e avós, que me deram total suporte familiar para estar apto a estudar sem ter com o que me preocupar além da minha formação. Aos meus amigos e colegas de profissão que me fizeram ter apreço por tecnologia e me orientaram no que é hoje uma paixão para mim dentro da minha profissão: o *front-end*. Agradeço também a todos os mestres que me desafiaram a apresentar sempre algo melhor, não deixando assim que eu desista e mesmo me sentindo frustrado durante o percurso, me sinto aliviado e orgulhoso com o produto final desses anos de estudo.

RESUMO

O presente trabalho apresenta ferramentas de front-end e elabora um *framework* para criação de ambientes de *front-end*. Primeiramente, são explorados temas como CSS, HTML e Javascript afim de familizar o leitor com o que é *front-end*. Enfim é explorado o tema de *frameworks*, passando pelo seu conceito e aplicações de *front-end*. Então são apresentadas ferramentas atuais como SASS, CoffeeScript e Jade, visando sempre suas vantagens em relação às tradicionais e é explorada a forma de utilização em ambientes modernos de *front-end*. Então é explicado como o projeto "Lâmpada Mágica" utiliza essas ferramentas modernas.

Palavras-chave: Node.js, Sass, Yeoman.

“Magic Lamp” - An environment front-end framework

ABSTRACT

This paper presents front-end tools and prepares a simple environment front-end framework. First, it explores topics such as CSS, HTML and Javascript in order to get the reader familiar to what is front-end. Finally the theme of frameworks is explored, through it’s concept and some front-end applications. So are presented as current tools SASS, CoffeeScript and Jade, always seeking it’s advantages over traditional tools, it’s explored how to use in modern front-end environment. Then it’s explained how the “Magic Lamp” design uses these modern tools.

Keywords: Node.js, Sass, Yeoman.

LISTA DE ABREVIATURAS E SIGLAS

AJAX	*Asynchronous Javascript and XML*
API	*Application Programming Interface*
CSS	*Cascading Style Sheets*
CSSOM	*Cascading Style Sheets Object Model*
DHTML	*Dynamic HyperText Markup Language*
DOM	*Document Object Model*
DRY	*Don't repeat yourself*
HAML	*HTML Abstraction Markup Language*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
I/O	*Input/Output*
IP	*Internet Protocol*
JS	*Javascript*
NPM	*Node Pack Manager*
SASS	*Syntactically Awesome Style Sheets*
URL	*Uniform Resource Locator*
W3C	*World Wide Web Consortium*
XML	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO . . . 11
1.1 Objetivo . . . 11
1.2 Justificativa e Relevância . . . 11
1.3 Organização do texto . . . 12

2 O QUE É *FRONT-END*? . . . 13
2.1 Funcionamento dos *websites* . . . 14
2.2 HTML . . . 15
2.3 *Hyperlinks* . . . 17
2.4 CSS . . . 19
2.5 Interação entre CSS e HTML . . . 21
2.6 *Object Models* . . . 22
2.6.1 *Document Object Model* . . . 23
2.6.2 CSS *Object Model* . . . 24
2.6.3 *Browsers* e *Render Tree* . . . 25
2.7 Javascript e o processamento do lado cliente . . . 27

3 *FRAMEWORKS* DE *FRONT-END* . . . 28
3.1 HTML5 *Boilerplate* . . . 29
3.2 *Bootstrap* . . . 32
3.3 Comparativo entre os *frameworks* . . . 34

4 O *FRAMEWORK* LÂMPADA MÁGICA . . . 36
4.1 Node.js . . . 37
4.2 Yeoman . . . 38
4.3 SASS . . . 39
4.4 CoffeeScript . . . 42
4.5 Jade . . . 44
4.6 Gulp . . . 46

4.7 Como funciona o *framework* Lâmpada Mágica 49
4.7.1 Instalação 49
4.7.2 Compilador de CoffeeScript 50
4.7.3 Compilador de SASS 51
4.7.4 Compilador de Jade 52
4.7.5 Copiador de Imagens 52
4.7.6 Tarefas de limpeza 53
4.7.7 Tarefas de minificação 54
4.7.8 "Agrupador" dos compiladores 55
4.7.9 Monitorador 55

5 CONCLUSÃO 57
5.1 Principais contribuições 58
5.2 Trabalhos futuros 58

REFERÊNCIAS 59

1 INTRODUÇÃO

1.1 Objetivo

O objetivo deste estudo é discutir a base para o desenvolvimento do *framework* "Lâmpada Mágica". Para isso, é preciso entender o que é o *front-end* tradicional, entender como funcionam *frameworks* de *front-end* e como se pode desenvolver um *framework* para criação de ambientes de projetos utilizando ferramentas atuais de *front-end*.

1.2 Justificativa e Relevância

O *front-end* é uma área importante presente no desenvolvimento de *websites*; a partir dele define-se a parte visual dos sites e é indispensável que essa apresentação seja muito bem feita. As ferramentas que compõem o *front-end* tradicional possuem poucos recursos se comparadas com os recursos de linguagens de programação mais atuais. Dessa forma, faz-se necessário buscar soluções para problemas conhecidos que essas ferramentas apresentam durante o desenvolvimento utilizando-as.

Construir o próprio *framework* de projeto torna o desenvolvedor apto a adaptá-lo às futuras necessidades, com um esforço menor. O bom uso de ferramentas atuais tem grande importância na agilidade do processo de construção, pois o que se espera na criação de um produto é que haja pouco esforço na preparação do ambiente de desenvolvimento e o foco recaia sobre a elaboração do projeto em si.

1.3 Organização do texto

Este trabalho apresenta os seguintes capítulos:

- O que é *front-end*?: apresentação das ferramentas que compõem e definem o *front-end* tradicional, aprofundamento do funcionamento das mesmas e o relacionamento entre elas.
- *Frameworks* de *front-end*: definição de *framework*, apresentação e comparação entre *frameworks* importantes para o presente estudo.
- O *framework* Lâmpada Mágica: aprofundamento das tecnologias adotadas para desenvolver o "Lâmpada Mágica" e detalhamento de funcionalidades relevantes.

2 O QUE É *FRONT-END*?

O *front-end* é um conjunto de técnicas, utilizado em associação com ferramentas de desenvolvimento, que tem por objetivo criar e manipular a parte visual e estrutural das páginas *web*[1].

As ferramentas de desenvolvimento do *front-end* seriam suas linguagens principais e únicas interpretadas pelos *browsers* (cuja tradução é navegador *web*): HTML, CSS e Javascript.

Já que as ferramentas são de fácil manipulação e não têm muitas restrições em relação ao uso, as técnicas para utilizá-las são bastante variáveis. Isso quer dizer que inúmeras formas de resolver um problema podem ser adotadas e nem sempre haverá uma eleita como a melhor para casos gerais.

Ao tentar entender o que é o *front-end*, precisamos, antes de tudo, compreender alguma habilidades que fazem parte desse universo. Basicamente, para desenvolver plenamente em *front-end*, faz-se necessário entender os seguintes conceitos e ferramentas:

- Funcionamento dos *websites*
- HTML
- *Hyperlinks*
- CSS
- Interação entre HTML e CSS
- *Object Models*
- Javascript

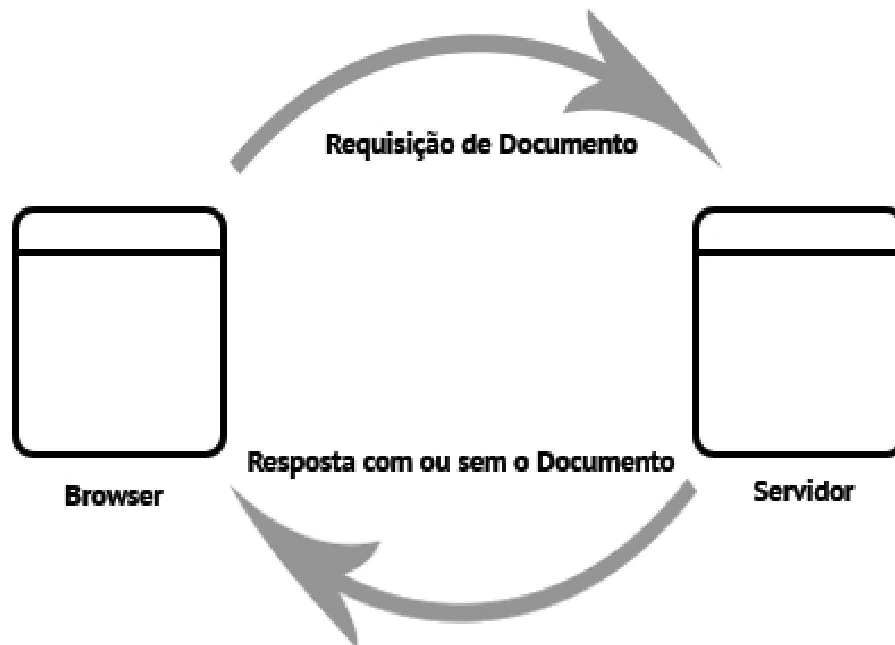
2.1 Funcionamento dos *websites*

Começamos o estudo apresentando o que é um *website* ou, simplesmente, *site*.

Segundo o *Wikipedia*, "Um *website* ou *site* é um conjunto de páginas *web*, isto é, de hipertextos acessíveis geralmente pelo protocolo HTTP na internet" [2]; ou seja, de modo simples, são páginas que ficam armazenadas em um servidor que possui conexão através de *hyperlinks* entre elas e são acessíveis por um endereço eletrônico (também conhecido como endereço IP), geralmente pelo protocolo HTTP.

Essas páginas são acessadas através de um *browser* em uma máquina cliente. Os *browsers* cuidam de toda a conexão da máquina cliente com o servidor e da transferência das páginas que compõem o *site*; o usuário apenas precisa ter em mãos o endereço (ou URL) e uma conexão com a internet ativa. De modo simplificado, a atuação do *browser* poderia ser definida deste modo[3]:

Figura 2.1: Modelo Cliente-Servidor



2.2 HTML

A sigla significa *HyperText Markup Language*, que, traduzida ao português seria *Linguagem de marcação de hipertextos*. Historicamente, o HTML foi criado em 1991 por Tim Berners-Lee, no CERN (*European Council for Nuclear Research*) na Suíça [4].

De início, foi criado para interligar redes de pesquisa de instituições próximas e compartilhar documentos entre elas. Em 1992, com a criação da *World Wide Web*, o HTML tornou-se a primeira linguagem de uso em escala global.

Desde então, ela passou por diversas mudanças que incluiram melhorias em suas

funcionalidades e a cada versão o padrão era revisto e tornava-se mais fácil de ser compreendido e utilizado. Também é importante ressaltar que certas funcionalidades cairam em desuso em decorrência da evolução de outras ferramentas como CSS e Javascript. Atualmente, o HTML encontra-se na versão 5.

O HTML não é uma linguagem de programação, mas, sim, uma linguagem de marcação baseada em *tags* com sintaxe própria.

Segue um exemplo da forma de escrita de uma *tag* HTML:

Figura 2.2: Exemplo de escrita de *tags* HTML

```
<a href="http://exemplo.url.com.br">Exemplo de um Link</a>
```

O HTML possui uma estrutura básica: através dela o *browser* consegue interpretar o documento com a extensão ".html".

Exemplo de estrutura básica de um documento HTML[5]:

Figura 2.3: Exemplo de Estrutura Básica HTML

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8">
5      <title>Document</title>
6  </head>
7  <body>
8
9  </body>
10 </html>
```

Linha 1: A definição do cabeçalho do documento serve para delimitar o tipo (ou versão) de documento HTML que o *browser* deve interpretar.

Linha 2: Abertura da *tag* <html>: é a *tag* que irá agrupar todas as outras, sendo necessária para o navegador entender que aquele é o local de um documento HTML.

Linha 3: Abertura da *tag* <head>, que é uma *tag* especial do HTML. Nada dentro dela é exibido pelo navegador na tela do usuário. Nela, ficam indicações para outros arquivos e meta-informações que são necessárias para o navegador processar o documento.

Linha 4: Meta-informação para o navegador que indica a codificação de caracteres do documento. Por consenso padrão, o documento tentará ser interpretado pela codificação que o *browser* assumir. Por esta razão, devemos incluir essa *tag* que informa ao *browser* qual a codificação a ser usada.

Linha 5: O conteúdo da *tag* <title> é o nome do documento ou da página, usado principalmente para a navegação entre as "abas" dos *browsers*.

Linha 6: Fechamento da *tag* <head>

Linha 7: Abertura da *tag* <body>. Tudo o que estiver dentro da *tag* <body> é o que será exibido ao usuário, no navegador.

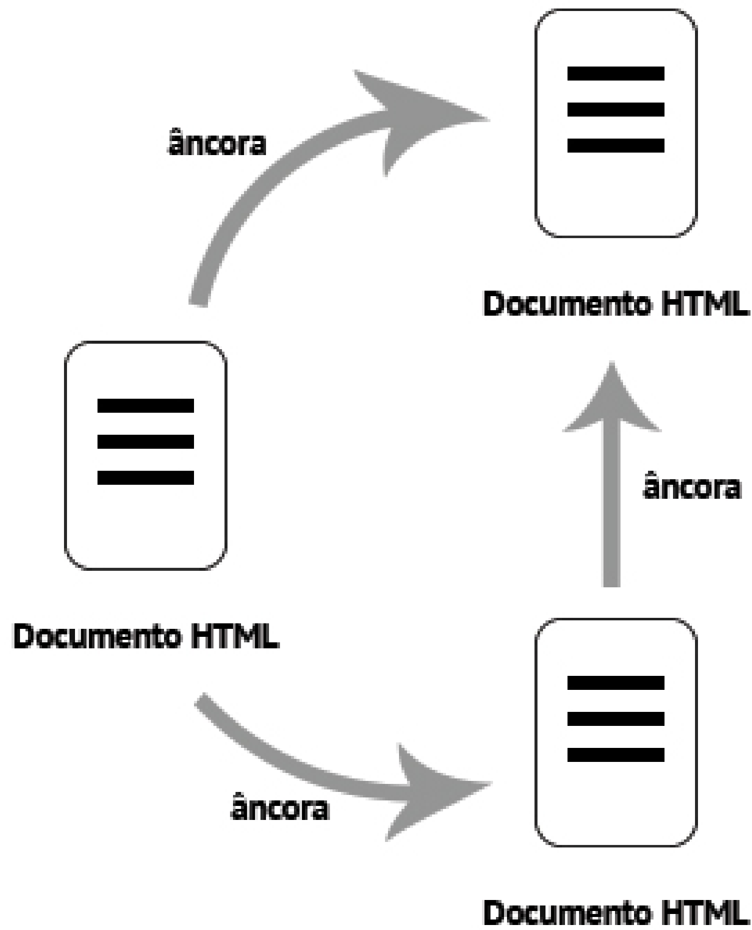
Linha 9 e Linha 10: Fechamento das *tags* <body> e <html>, respectivamente.

2.3 *Hyperlinks*

Os *hyperlinks* são, na verdade, os caminhos que ligarão um documento HTML a outro. Eles normalmente são conectados através das *tags* <a>, que são chamadas de "âncoras". Servem para a navegação entre os documentos, o que é essencial no

processo de preservação da arquitetura da informação.

Figura 2.4: Navegação entre Documentos



As *tags* <a> contêm um atributo chamado "*href*", cujo valor associado é o caminho relativo (ou absoluto) do documento ou da URL que se pretende acessar.

Exemplificando:

Figura 2.5: Exemplo *href*

```
7   <body>
8       <div class="menu">
9           <a href="/inicio/">Início</a>
10          <a href="/produtos/">Produtos</a>
11          <a href="/contato/contato.html">Contato</a>
12      </div>
13  </body>
```

Partindo do pressuposto de que a URL base do arquivo que contém esse código HTML é “http://lampamagica.com.br”, observamos que:

- O primeiro *hyperlink* faz conexão com “http://lampadamagica.com.br/inicio/”.
- O segundo *hyperlink* faz conexão com “http://lampadamagica.com.br/produtos/”
- O terceiro é diretamente conectado a um arquivo “contato.html”, que está dentro da estrutura “http://lampadamagica.com.br/contato/”.

Entendendo como funciona o HTML e os *Hyperlinks*, sabemos que o *front-end* é visto como a camada de “Visão” de uma página “web”, porém podemos assumir o HTML seria nossa camada de “Modelo” dentro do *front-end* por conter os dados que desejamos armazenar nas páginas *web*.

2.4 CSS

Sua sigla significa *Cascading Style Sheets* que, em português, seria traduzido como *Folhas de estilo em Cascata*. O CSS surgiu após a criação do HTML como solução para a formatação já existente no HTML, cujo sucesso do formato garantiu constantes evoluções no CSS que, atualmente, encontra-se na versão 3 [6].

O importante a se entender sobre o CSS é que ele serve para definir a parte visual das páginas e separar a camada de "Modelo" da camada de "Visão"; ou seja, tira toda a formatação visual dos dados da mesma estrutura em que estes se encontram. Em outras palavras: deixa o HTML preocupar-se somente em armazenar as informações e não em como irá apresentá-las.

O CSS tem um formato diferente do HTML. Para definir um estilo para um elemento ou mais, primeiramente precisa-se definir um seletor para ele. No caso das *tags* de HTML, seria o próprio nome da tag.

O HTML tem prontos alguns atributos que são muito utilizados nos seletores CSS: o "id" e o "class". Para definir um seletor de um elemento único, usa-se o "id" e o seu seletor é prefixado com uma tralha seguida do valor que foi atribuído ao documento HTML. Caso haja o desejo de que um grupo de elementos comporte-se visualmente de determinada forma, é recomendado utilizar o atributo "class" e prefixar no seletor com um ponto (".") seguido do valor associado ao HTML.

As *tags* de HTML têm eventos específicos que são disparados pelo *browser* quando alguma ação do usuário acontece, como o fato de colocar o cursor do mouse em cima do elemento (mouse over) ou clicar no elemento (click). Para esses estados que os elementos assumem, podemos atribuir um "pseudo-seletor", com sintaxes próprias do CSS para definir a estilização do elemento naquele estado. Segue um exemplo de estilização para vários elementos, com "id", "class" e "pseudo-seletores" [7]:

Figura 2.6: Seletor CSS

```
div, #elemento, .outro-elemento, a:hover, img[atributo="com-fundo"] {
    width: 100%;
    color: #666;
    background-image: url("imagens/fundo.png");
    font-size: 14px;
}
```

2.5 Interação entre CSS e HTML

Existem 3 formas de se aplicar CSS a um documento HTML.

A primeira delas é a mais simples e menos recomendada de todas: aplicar diretamente o CSS como um atributo "*style*" da *tag* de HTML e, dentro dele, definir as "chaves:valores" que definem o visual, como no exemplo abaixo:

Figura 2.7: A *tag style inline*

```
<body>
    <div class="menu" style="width: 940px; margin: 0 auto; background-color: #f0f0f0;">
        <ul class="lista">
            <li class="item">Início</li>
            <li class="item">Produtos</li>
```

Este exemplo apresenta uma aplicação inadequadado de CSS porque define um estilo *inline* somente para a *tag* que está com o atributo "*style*". A sua utilização vai contra boas práticas, como reutilização de código e "manutenibilidade".

O segundo jeito de aplicar CSS, que também não é recomendado, é definir os seletores de CSS dentro de uma *tag* HTML chamada <style> e, ali dentro, aplicar a formatação CSS.

Figura 2.8: O bloco *style*

```
<body>
    <style type="text/css">
        .menu {
            width: 940px;
            margin: 0 auto;
            background-color: #f0f0f0;
        }
    </style>
    <div class="menu">
        <ul class="lista">
            <li class="item">Início</li>
            <li class="item">Produtos</li>
```

O problema dessa forma de aplicação é que ela apenas aplica o CSS diretamente no arquivo HTML, tornando, assim, tudo o que for definido ali dentro, inacessível a outros documentos que não possuírem aquele trecho de código. Sua aplicação também vai contra boas práticas, porém é menos agressiva que a forma *inline*.

A terceira, e melhor forma de utilização do CSS, é através da criação de um arquivo com extensão ".css" e a chamada em cada documento que vá utilizá-lo. Essa forma possibilita que, ao fazer manutenção ou criar um elemento novo, ele possa ser atualizado em todas as páginas que incluírem o arquivo de CSS. Para incluir o arquivo, basta inserir na *tag* <head> uma chamada, da seguinte forma:

Figura 2.9: Inclusão de arquivo CSS em um documento

```
    <link rel="stylesheet" href="/estaticos/base.css">
</head>
<body>
    <div class="menu">
        <ul class="lista">
            <li class="item">Início</li>
```

E dentro do arquivo "base.css" definir o estilo do seletor:

Figura 2.10: Conteúdo de um arquivo CSS

```
.menu {
    width: 940px;
    margin: 0 auto;
    background-color: #f0f0f0;
}
```

2.6 *Object Models*

Para cada seletor CSS devemos definir um "caminho" ao qual ele pertence, no caso de utilizarmos as *tags* do HTML, ou o atributo *class* do HTML ou mesmo o atributo *id*.

Esse caminho é acessado através da DOM, ou *Document Object Model*, que contém a localização do elemento dentro da árvore de um documento HTML.

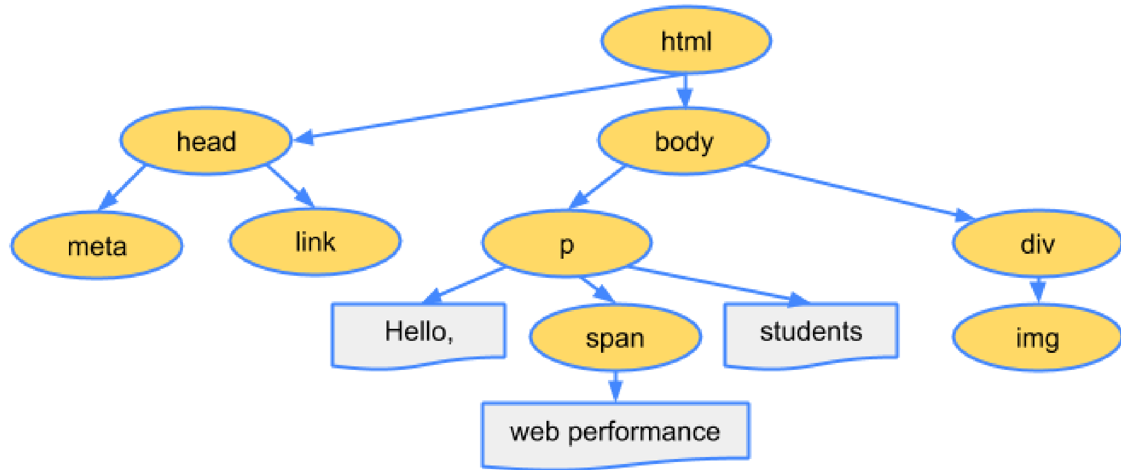
2.6.1 *Document Object Model*

A definição de DOM, pela W3C, é a seguinte: "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page"[8].

Isso quer dizer que a DOM não contém apenas os "caminhos" dos elementos como sugere a W3C ao defini-lo como interface neural, mas, também, contém propriedades importantes na atualização do conteúdo, estilo e estrutura dos elementos do documento.

Os navegadores costumam interpretar os documentos HTML seguindo a DOM; para isso, eles criam uma árvore com um "nó-raiz" chamado "*Document*" e, a partir dele, vão seguindo a estrutura do HTML. É assim que funciona na localização de um seletor de CSS, também. Abaixo, um exemplo de como seria a árvore de uma DOM sem o "nó-raiz" "*Document*".

Figura 2.11: Árvore DOM (*Document Object Model*)



Na figura acima, podemos observar a árvore da DOM e cada *tag* do HTML como um nó dentro dessa árvore. Se quiséssemos acessar qualquer elemento hipotético com a classe "menu", no CSS, por seu caminho completo na DOM, teremos:

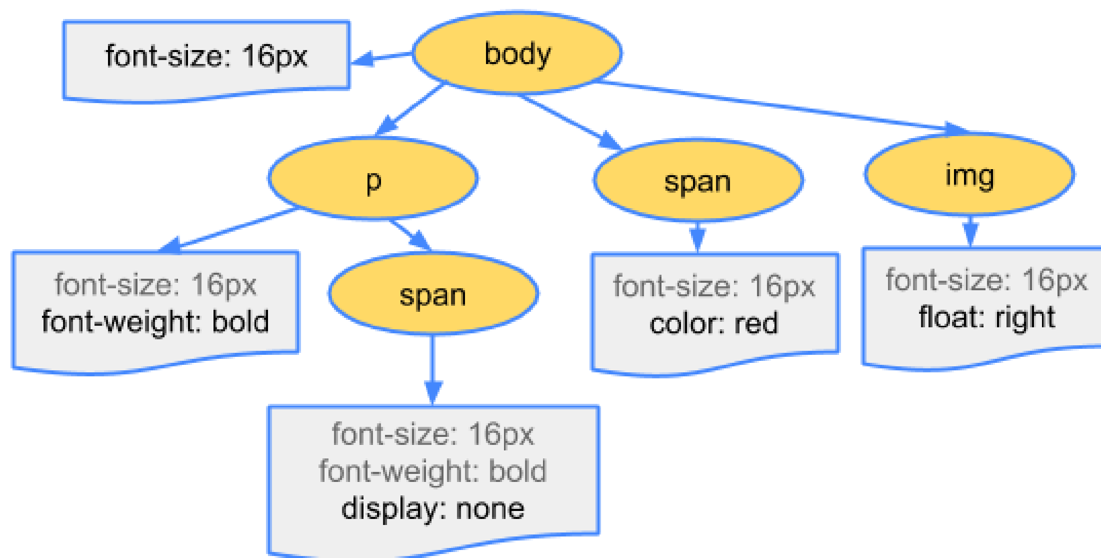
Figura 2.12: Árvore DOM (*Document Object Model*)

```
html body .menu {
    width: 940px;
    margin: 0 auto;
    background-color: #f0f0f0;
}
```

2.6.2 CSS *Object Model*

O CSS também possui um *Object Model*, chamado de CSSOM. Este funciona de forma semelhante ao DOM do HTML, porém, como os seletores de CSS possuem atributos chave e valor, o CSSOM será montado seguindo a lógica de percorrer todos os seletores e associando os valores aos objetos. Abaixo, um exemplo de CSSOM[9]:

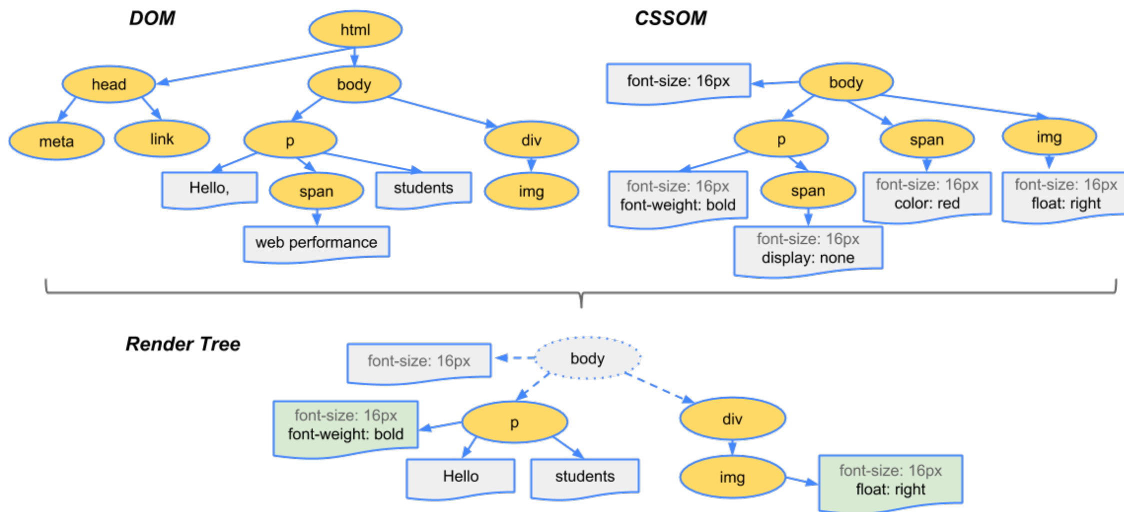
Figura 2.13: Árvore CSSOM (CSS *Object Model*)



2.6.3 *Browsers* e *Render Tree*

Durante o processo de "renderização" das páginas, os navegadores percorrem todos os elementos da DOM e vão montando uma estrutura chamada *Render Tree*, que é o resultado da junção da DOM com a CSSOM.

Figura 2.14: Junção de DOM com a CSSOM



O processo de montagem e interpretação da *Render Tree* pelo *browser* segue os seguintes passos:[10]

1. A partir da raiz da DOM, percorrer cada nó visível;
2. Para cada nó visível, procurar sua regra associada na CSSOM e aplicar a regra;
3. "Renderizar" o nó visível com seu conteúdo e regra associada.

É importante ressaltar que o *browser*, por padrão, não "renderiza" na tela todos os elementos de uma só vez; conforme percorre a *Render Tree*, ele "renderiza" os elementos.

2.7 Javascript e o processamento do lado cliente

O Javascript é uma linguagem inventada em 1995 por Brendan Eich, da *Netscape*, para o lançamento junto com a versão 2.0 do navegador *Netscape* e lançada com o nome *LiveScript*. Pouco tempo depois, a empresa Sun Microsystems lançou a versão 2.0B3 do *Netscape*, já com o nome Javascript [11].

O Javascript é uma linguagem interpretada, estruturada, funcional, com tipagem dinâmica e baseada em protótipos.

O objetivo do lançamento do Javascript foi fazer uma separação entre os processamentos do lado cliente e servidor, o que proporcionou mais liberdade aos servidores para processar informações e fazer acessos a dados, além de permitir que as páginas fossem mais dinâmicas, sem afetar as máquinas de servidores, criando, assim, o conceito de DHTML, que é *Dynamic HTML*.

O Javascript tem duas funções muito importantes no desenvolvimento de páginas *web*: manipulações e controle dos elementos na DOM; e comunicação assíncrona com o servidor através da técnica AJAX. Por esta razão, o Javascript poderia assumir um papel de camada de "Controle" na arquitetura do *front-end*.

3 *FRAMEWORKS* DE *FRONT-END*

Dentro da comunidade de desenvolvedores *front-end*, existem diversas ferramentas que auxiliam no desenvolvimento e, também, existem boas práticas para o uso de cada ferramenta e para o uso do *front-end*, de modo geral. Uma das melhores práticas é a utilização de *frameworks*.

"Um *framework* é uma arquitetura desenvolvida com o objetivo de atingir a máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização" [12].

Existem, na comunidade de desenvolvedores, diversos *frameworks* para ferramentas *front-end*; até mesmo empresas que desenvolvem *frameworks* como produtos. Para esse estudo, foram escolhidos os dois *frameworks* abaixo:

- HTML5 *Boilerplate*
- *Bootstrap*

Esses dois *frameworks* foram escolhidas porque são completamente diferentes uma da outra, de modo que atendem a situações diferentes. Sendo assim, é possível

observar como se comportam e, a partir disso, podemos definir um comparativo não só de quantidade de funcionalidades, mas, sim, de utilidade das funcionalidades em situações adversas.

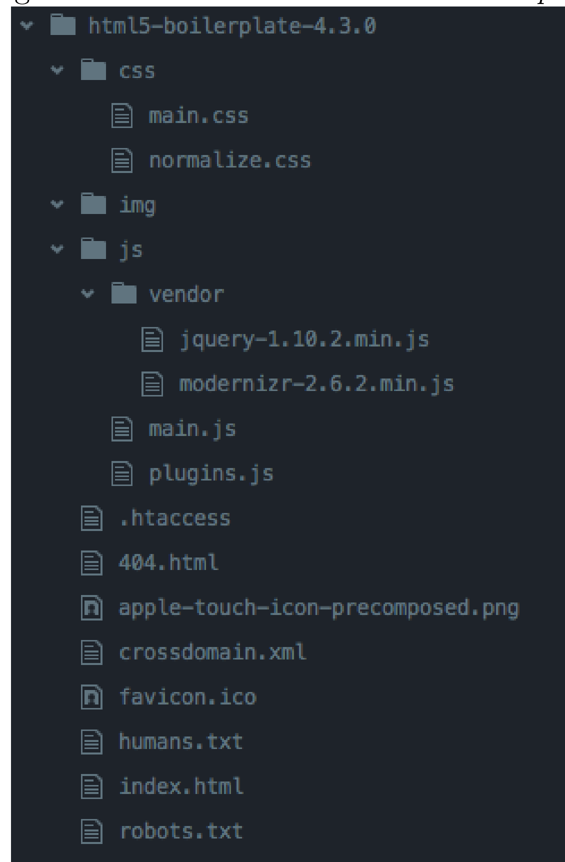
3.1 HTML5 *Boilerplate*

O HTML5 *Boilerplate* é um *framework* estrutural de *front-end*. Ao iniciar um trabalho com ele, cria-se uma estrutura padrão para o desenvolvimento de um projeto.

Figura 3.1: Logo do HTML5 *Boilerplate*

Abaixo, um exemplo da estrutura que o HTML5 *Boilerplate* cria. São basicamente 3 pastas: “css”, “img” e “js”. Elas são criadas para que os arquivo do mesmo gênero fiquem debaixo delas.

Figura 3.2: Estrutura do HTML5 *Boilerplate*



Parece uma ação simples organizar os arquivos desta forma, mas é uma excelente prática, sobretudo quando se tem arquivos com módulos de funções ou classes de CSS. Falando um pouco sobre os arquivos criados, temos o "main.css" e "normalize.css": "main.css" é o arquivo principal da folha de estilos, no qual o *framework* assume que todas as páginas terão um arquivo "main.css" incluso.

Já o "normalize.css" é um arquivo que serve como alternativa para os "resets" do CSS. O que ele faz é alinhar o entendimento dos *browsers* antigos para arquivos de CSS3 e HTML5, criando seletores para todas as regras de "renderização", que são diferentes nos *browsers*; é como se cada *browser* possuísse um CSS embutido e o "normalize.css"

alinhasse esse CSS a fim de que, em todos os *browsers*, a "renderização" fosse a mesma.

Sobre "main.js" e "plugins.js": da mesma forma como ocorre no CSS, a estrutura do *framework* entende que existirá um arquivo "main.js" que estará presente em todas as pastas, contendo regras de comportamento geral aos elementos. Já o arquivo "plugins.js" serve para os *plugins* jQuery que o usuário criar serem indicados ao *browser* e garantirem que não existirá nenhum conflito durante sua utilização.

A respeito dos arquivos da pasta *vendors*: este nome (*vendors*) é comumente utilizado pela comunidade de desenvolvedores para indicar a alocação de bibliotecas de terceiros. Dentro da pasta *vendors* da estrutura do HTML5 *Boilerplate*, ele inclui o jQuery, que é o principal *framework* de Javascript, e o Normalizr, que tem função semelhante aos "resets" do CSS: garantir o bom funcionamento de CSS3 e HTML5 em *browsers* antigos e novos.

O ".htaccess" e "crossdomain.xml", são arquivos de configuração da infraestrutura do projeto. O "humans.txt" é uma iniciativa da comunidade de desenvolvedores para obter *feedback* fácil dos usuários do *site* através do contato direto com os desenvolvedores; e o "robots.txt" faz com que o *site* fique visível nos mecanismos de busca. Finalizando, temos o "humans.txt" e o "index.html", "404.html". O "index.html" é a primeira página do *site* (todo *site* contém um arquivo chamado "index.html", que é o canal pelo qual o servidor exibirá a página quando esta ação for requisitada pela URL base). E o arquivo "404.html" é o mecanismo através do qual o servidor retorna ao *browser* quando não encontra o documento que o mesmo está requisitando.

Por último temos o "favicon.ico" e o "apple-touch-icon-precomposed-png", que são ícones para a aba do *browser* e para quando a ação de salvar nos favoritos dos dispositivos iOS for requisitada.

3.2 *Bootstrap*

O *Bootstrap* é um *framework* que possui soluções prontas para diversos elementos comumente utilizados em *websites* e sistemas *web*. Sua história é a seguinte: Foi criado em 2011, como um projeto interno da empresa *Twitter, Inc.*, para alinhar o padrão de arquitetura da informação com o padrão de código para soluções internas do *Twitter*.

Figura 3.3: Logo do *Bootstrap*

O sucesso da iniciativa do *Bootstrap* foi tão grande que os criadores o lançaram como um projeto livre e receberam milhares de contribuições da comunidade de desenvolvedores *front-end*. É conhecido como o *framework* mais completo, pois atende a todas as ferramentas de *front-end*: HTML, CSS e Javascript. Na verdade, o *Bootstrap* é um conjunto de elementos comuns em portais e páginas *web*, que possui soluções prontas de menus, rodapés, colunagem, formulários, alertas, destaques, dentre outras dezenas de elementos comuns na arquitetura da informação em *sites* que seguem padrões globais de usabilidade e acessibilidade.

Talvez seja este o grande ponto forte do *Bootstrap*: a agilidade para iniciar um projeto com ele e ter um *site* pronto com inúmeros elementos bem formatados e sem problemas de conflito entre ambos. Porém, em situações em que a customização do produto é essencial, o uso do *Bootstrap* é um complicador, já que ele funciona da seguinte maneira: os elementos HTML têm estrutura fixa (este é o segredo do

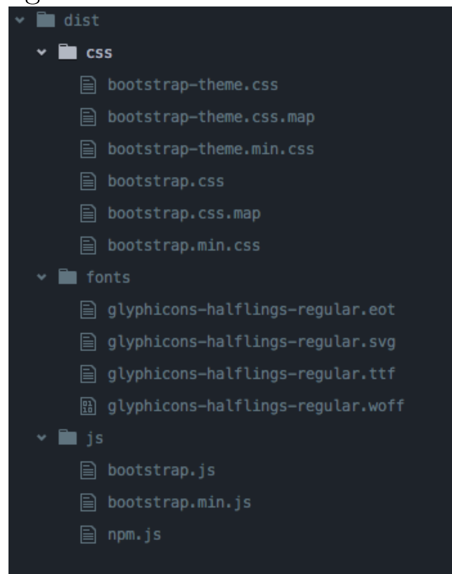
framework para o comportamento dos Javascripts, associados àquele *plugin* e ao estilo do CSS, para que funcionem perfeitamente em qualquer situação).

Os comportamentos de Javascript dos elementos são modelados por *plugins* jQuery, o que torna sua customização mais complicada, pois ele não é facilmente estendido. Na verdade, esses *plugins* não são feitos para isso; o *plugins* jQuery é uma forma mais eficiente de utilizar a memória e o processamento da máquina cliente.

Os estilos CSS são passíveis de extensão, mas, apenas, por substituição de regras em tempo de "renderização": o padrão dos elementos do *Bootstrap* é ser orientado a objetos, então, ao alterar no código-fonte da folha de estilos do *framework* a estilização de um elemento que utiliza uma classe qualquer, estaremos alterando todos que estendem aquele elemento ou classe. Isso foi feito para que as folhas de estilos do *Bootstrap* não fossem alteradas, preservando a "manutenibilidade" do código, permitindo que funcione como um núcleo intacto que pode ter sua versão sempre atualizada.

Esses 3 fatores (customização do produto, alteração dos *plugins* de Javascript e extensão do CSS) são os principais pontos fracos do *Bootstrap*, visto que objetivam a agilidade na criação das páginas e elementos, mas não contemplam a mesma agilidade na customização dos mesmos. Além disso, o *Bootstrap* não é indicado para ser "plugável" em um projeto já pronto; ou seja, tentar adaptar o projeto já existente para ser comportável com *Bootstrap* é altamente custoso. Por não ser um *framework* estrutural, ou seja, definir uma estrutura padronizada de projeto, o *Bootstrap* fornece os arquivos CSS, JS e de fontes.

Figura 3.4: Estrutura do *Bootstrap*



3.3 Comparativo entre os *frameworks*

	Bootstrap	HTML5 *Boilerplate*
É ágil na criação de um *site* novo com diversas funcionalidades	Sim	Não
Customização ágil em tempo de desenvolvimento	Não	Sim
É facilmente plugável em um projeto existente	Não	Sim
Possui suporte para diferenciar *debug* em ambiente de desenvolvimento e de produção	Não	Não
Suporte a tecnologias novas como SASS, Coffee e Jade	Parcial	Não
Trata questões de infraestrutura como *crossdomain* e configurações de servidor	Não	Sim
Trata configurações de mecanismos de buscas	Não	Sim
Possui elementos prontos e testados	Sim	Não
Possui *grid*	Sim	Não
Utiliza *plugins* jQuery para comportamento de elementos Javascript	Sim	Parcial
É frequentemente atualizado	Sim	Não
Monta a estrutura de um projeto *front-end*	Não	Sim
Possui suporte a dispositivos *mobile*	Sim	Não
É extensionável	Sim	Sim
Isola a camada de HTML de comportamento	Não	Sim
Elementos orientados à objetos	Sim	Sim

Acima, a tabela compara os dois *frameworks* em vários aspectos importantes na

escolha de um *framework*.

Os atributos de comparação selecionados englobam a agilidade na criação de um *site* novo e sua customização, pois em muitos casos um *framework* é escolhido para iniciar um projeto ou para customizá-lo. A capacidade de ser adaptado a um projeto já existente é um fator importante para saber se o impacto em trazer determinado *framework* para um projeto já existente será grande ou pequeno. Possuir suporte a tecnologias novas é um diferencial importante, quando se trata de inovação e quando se trata de agilidade a curva de aprendizado pode ser alta. Possuir arquivos de configuração de infraestrutura e mecanismos de busca pode ser um adicional para quem não tem familiaridade.

Quando se trata de elementos e ferramentas, o *grid* e os *plugins* jQuery são diferenciais para iniciar projetos, mas podem ser inúteis para projetos existentes. A frequência com que o *framework* é atualizado influencia muito no seu suporte a tendências de uso como os dispositivos *mobile*.

A capacidade se extencionável faz com que o *framework* possa ser utilizado para projetos customizados. Quanto ao fato de possuir uma camada de elementos HTML isolada de comportamento e elementos orientados à objetos é para garantir um projeto consistente e seguindo boas práticas de desenvolvimento.

4 O *FRAMEWORK* LÂMPADA MÁGICA

O *framework* “Lâmpada Mágica” é um projeto baseado na ideia do HTML5 *Boilerplate*. Tem por objetivos principais a inicialização de uma primeira estrutura simples e funcional, mas com diversas tecnologias agregadas para o desenvolvedor poder começar a desenvolver seu projeto sem ter de se preocupar com estrutura, focando no que realmente agregará valor ao projeto.

O *framework* “Lâmpada Mágica” foi feito utilizando as seguintes tecnologias:

- Node.js
- Yeoman
- SASS
- CoffeeScript
- Jade
- Gulp

4.1 Node.js

“Node.js é uma plataforma construída sobre o motor Javascript do Google Chrome para facilmente construir aplicações de rede rápidas e escaláveis. Node.js usa um modelo de I/O direcionada a evento não bloqueante que o torna leve e eficiente, ideal para aplicações em tempo real com troca intensa de dados através de dispositivos distribuídos” [13].

Figura 4.1: Node.js

O Node.js é um servidor de aplicações, apesar de ser diferente de servidores como *Apache* ou *Tomcat*. No Node.js existe um conceito de módulos de aplicações que estendem o núcleo do Node.js, fazendo com que agregue funcionalidades ao servidor. Esses módulos são bem simples de serem construídos e a comunidade de desenvolvedores contribui ativamente, já que produz milhares de módulos com diversas soluções para problemas conhecidos.

O que faz o Node.js ser uma opção interessante em relação aos outros servidores de aplicações é o fato de rodar em cima do Javascript V8, que é o motor do Google Chrome. O V8 possui um interpretador escrito em C++ que interpreta códigos escritos em Javascript. Dessa forma, do lado do servidor pode-se codificar em Javascript, mantendo, assim, uma uniformidade de linguagem entre *back-end* e *front-end*.

Outra questão importante quando se trata de usar Javascript no *back-end* é o uso de programação orientada a eventos. O Javascript é ótimo na questão de tratar eventos,

pois a implementação de *callbacks* é muito simples e prática. Desenvolvedores *front-end* que usam jQuery estão acostumados a trabalhar com orientação por eventos como cliques, movimentos de *mouse*, foco em campos de formulários, etc.

Esses eventos existem de forma semelhante no Node.js, mas sem uma interface de interação com usuário; portanto, o que é capturado é um evento como: conexão aberta, dados entrando por essa conexão, dados sendo enviados para o destino, etc. Dessa forma, assim que o servidor Node.js estiver rodando, toda vez que um evento for disparado, um tratamento pode ser feito para ele.

Para publicar um módulo, utiliza-se o NPM, que significa *Node Package Manager*, que nada mais é do que um gerenciador de pacotes que atua como conexão com o repositório oficial de pacotes e módulos do Node.js. Através do NPM, pode-se publicar o próprio módulo ou instalar módulos de terceiros.

4.2 Yeoman

Inspirado no gerador de *templates* do Ruby on Rails, o Yeoman é uma aplicação feita em Node.js, que é um *framework* para criação de *templates* de projeto, funcionando da seguinte maneira: através da API do Yeoman, é possível especificar a nomenclatura dos arquivos e o tipo de conteúdo, gerar opções de bibliotecas que vão compor a estrutura e dependências que farão parte do projeto final.

Figura 4.2: Yeoman

O produto final de uma codificação que usa Yeoman é um gerador de *template*, mas, acima disso, é um módulo em Node.js. Isso surgiu da necessidade de agrupar várias tecnologias novas em um mesmo projeto.

Cada desenvolvedor tem sua forma de organizar a sua estrutura. Por essa razão, é importante manter funcional e simples um "esqueleto" de projeto, para que o desenvolvedor possa ganhar tempo ao desenvolver o produto focado no usuário final. Este fato, a longo e médio prazo, é muito bom, pois se perde pouquíssimo tempo configurando ambiente e há sempre uma evolução conforme a necessidade, visto que o gerador é independente do projeto em si.

O Yeoman possui um vasto repositório de geradores e, como é *open source*, qualquer desenvolvedor pode contribuir com seu próprio gerador ou melhorar algum gerador já existente. Alguns geradores são feitos até mesmo pelos produtores responsáveis pelo Yeoman (tidos como oficiais). Os geradores são executáveis através de linha de comando, então, no local em que o usuário quiser gerar um *template*, basta rodar "yo NOME-DO-GERADOR" [14].

4.3 SASS

SASS é uma sigla para "*Syntactically Awesome Style Sheets*" que, em uma tradução livre, seria algo como "folhas de estilo sintaticamente impressionantes". O SASS é um pré-processador de CSS, o que quer dizer que é uma linguagem tal qual o CSS; por isso os desenvolvedores codificam em SASS e esse código é processado para outra folha de estilo no formato CSS, preservando todos os elementos e hierarquia, como se tivesse sido escrito em CSS.

Figura 4.3: SASS

Existem várias formas de instalar o SASS. Inicialmente foi desenvolvido em Ruby, contudo, com a expansão de usuários adeptos foram sendo criadas versões do processador de SASS para CSS em diferentes linguagens de programação e tecnologias distintas. Uma delas é um módulo do Node.js [15].

O SASS é muito semelhante ao CSS, na verdade, pode-se escrever SASS como se fosse CSS, e irá funcionar normalmente, mesmo que o processamento deixe o código gerado idêntico. Esse procedimento é feito porque os *browsers* não "entendem" SASS; eles só "entendem" CSS como folha de estilo.

O que torna o SASS uma ótima ferramenta que substitui o CSS tradicional são as melhorias que ele proporciona [16]. Algumas delas são:

1. A criação de variáveis e atribuição de valores a elas: as variáveis podem ser usadas como valores dos atributos dos elementos CSS e, até mesmo, como valores a serem passados por *mixins* e nomenclatura para classes e *ID's*;
2. "Aninhamento" de hierarquia dos elementos (talvez a melhoria mais usada e mais prática do SASS): baseado na estrutura do HTML, o SASS permite que os elementos possam ser aninhados, garantindo, assim, que a estilização de uma classe dentro de um elemento CSS, ao ser gerado, pertença somente àquele elemento; ou seja, quanto mais aninhado o elemento, mais específica é a sua

estilização. Isso é importante pois reduz a quantidade de código repetido e torna a compreensão do mesmo mais fácil;

3. Partições e inclusões: com o SASS, é possível dizer qual arquivo será gerado (isso é uma convenção que o SASS usa para *import's*). Quando um arquivo é uma partição de algo, utiliza-se o underscore antes do nome do arquivo e, quando ele for importado pelo "@import" em outro arquivo SASS, no arquivo CSS gerado ele estará presente, mas pertencendo ao arquivo que o importou. Outra diferença do "@import" do SASS para o do CSS é que, como ainda não foi processado para se tornar CSS, não há necessidade de utilizar uma requisição HTTP para carregar o arquivo, já que a inclusão acontece no processamento para o CSS. Essa medida é importante para reduzir a quantidade de requisições de CSS da página para o servidor, tornando, assim, mais rápido o carregamento dessa página;

4. *Mixins*: são como funções para o CSS. Dentro, é possível definir um bloco de atributos CSS com seus valores e os *mixins* também aceitam parâmetros. Onde for chamado aquele *mixin*, o bloco de atributos e valores, dentro, será copiado. Isso é muito útil para criar atributos de CSS3 com seus prefixos para *browsers* antigos;

5. Herança e extensão: com o SASS, é possível utilizar herança não só através do método tradicional no CSS, como também usando "@extend". Através desse método, é possível escrever bem menos código que no CSS tradicional, no qual só seria possível se reescrevesse o estilo de uma determinada "classe-pai". Usando o "@extend" nas "classes-filhas", o código fica limpo e reutilizável, pois, se alterasse na "classe-pai", as filhas também seriam alteradas, além de manter isso totalmente separado da camada de Javascript e da camada de HTML;

6. Operadores em valores: outra vantagem é poder utilizar operadores de adição, subtração, multiplicação e divisão nos valores dos atributos CSS. E isso

vai além da multiplicação de números simples; o SASS é capaz de entender uma soma de "10px" com "35px" e transformar em "45px", por exemplo. Essa funcionalidade torna-se extremamente útil quando se quer fazer uma página responsiva.

4.4 CoffeeScript

"CoffeeScript é uma pequena linguagem que compila para JavaScript. Por baixo de todas essas chaves e ponto-e-vírgulas inconvenientes, JavaScript sempre teve um lindo modelo de objeto em seu coração. CoffeeScript é uma tentativa de expor as partes boas de JavaScript de uma maneira simples" [17].

Figura 4.4: CoffeeScript

O CoffeeScript simplifica inúmeras situações que, em Javascript, são complexas. A forma como se codifica em CoffeeScript é mais "amigável", fazendo uso de recursos existentes em linguagens mais atuais, como Ruby e Python. As mudanças mais relevantes do Javascript para o CoffeeScript são [18]:

1. Remoção de chaves e ponto-e-vírgula: a estrutura do CoffeeScript é baseada na "identação", tal como Ruby, Python e Swift. Diferentemente de Swift, colocar chaves e ponto-e-vírgula no CoffeeScript é considerado errado, impedindo a compilação. Essa estratégia serve para manter a linguagem "pura" e padronizada;

2. O uso de vírgula em definição de objetos: o uso de vírgula em objetos é

opcional, porém, se esse objeto for definido com todas as chaves na mesma linha, é necessário usar vírgula;

3. Interpolação de *strings*: é um recurso muito útil que existe em linguagens mais atuais e foi implementado no CoffeeScript para facilitar a inclusão de uma variável no âmbito do texto, bastando utilizar #{VARIÁVEL} no meio do texto;

4. Intervalos: muitas vezes, em uma lista de objetos, é necessário obter apenas os objetos de uma determinada posição até outra; dessa forma, a facilidade de uso dos intervalos no CoffeeScript poupa algumas linhas de código ao desenvolvedor e o torna mais simples na leitura;

5. Compreensão de listas: é um recurso originário do Python e, no CoffeeScript, apresenta-se da mesma forma. É possível percorrer um dicionário de objetos ou uma lista com um "*for each*" associando variáveis para chaves e valores definidos na própria chamada do "*for*"; portanto, o desenvolvedor não precisa se preocupar em associar nada dentro do "*for*"; deve, apenas, utilizar os dados na forma das variáveis definidas. O recurso de intervalos pode ser utilizado em conjunto, como "objeto" a ser percorrido;

6. Operador existencial: é uma forma mais "enxuta" de verificar a existência de determinada variável utilizando "?", após a mesma, em qualquer estrutura condicional;

7. Classes e herança: o Javascript é uma linguagem orientada a protótipo que, na verdade, é uma forma mais primitiva da orientação a objetos presente em linguagens como Java, C++, Ruby e Python. O CoffeeScript possibilita às classes serem definidas num formato mais próximo às linguagens atuais, utilizando a palavra reservada "class", para definir uma classe, e "extends", quando for uma classe herdada de outra.

O Javascript teve seu lançamento em 1995, já o CoffeeScript teve sua primeira versão em 2009. De lá para cá, o ritmo de atualizações no CoffeScript foi notório, enquanto o Javascript manteve-se parado desde 2011, período em que houve uma grande atualização. Por essa razão, o CoffeeScript tem se tornado uma solução para os desenvolvedores que se mantêm atualizados com as tecnologias mais recentes; o ritmo de atualizações é frequente e praticamente acompanha linguagens modernas [19].

O importante é perceber que as alterações de Javascript para CoffeeScript não são funcionalidades novas nas linguagens de programação e, sim, adaptações especialmente de funcionalidades do Ruby e do Python para o Javascript, que é o objetivo do CoffeeScript.

4.5 Jade

Pela definição do autor da linguagem, "Jade é uma linguagem concisa para escrever *templates* HTML" [20]. Em outras palavras, Jade é um pré-processador de HTML feito em Node.js e funciona tal como o SASS para o CSS e o CoffeeScript para o Javascript. O Jade é influenciado pelo HAML, um pré-processador feito em Ruby [21].

Figura 4.5: Jade

Seguindo o mesmo princípio do SASS e do CoffeeScript, o Jade acrescenta recursos

que o HTML tradicional não possui. Suas principais funcionalidades são:

1. Sintaxe enxuta seguindo o padrão DRY (*Don't repeat yourself*): o Jade é baseado em "identação" e as *tags* de HTML não são definidas baseadas no padrão XML existente no HTML tradicional; não se utiliza "<" e ">" para abrir e fechar os nós de cada *tag*. Isso aproxima o Jade de linguagens modernas como Ruby e Python;

2. Utilização de variáveis: é possível definir variáveis e atribuir valores a elas. Variáveis definidas em escopos "pai" valem para *templates* inclusos, a fim de melhorar a "manutelibilidade" do código e diminuir o mesmo;

3. Acréscimo de estruturas condicionais e *loops*: o Jade torna o HTML uma linguagem de programação com esse recurso, tal como o SASS faz com o CSS. O "*if*" pode ser utilizado testando valores de variáveis definidas, assim como o "*each*" e o "*while*", que são as duas estruturas de iteração presentes no Jade. Esses são recursos que acrescentam muito à linguagem, pois tiram grande parte de processamentos feitos em Javascript;

4. Definição de blocos e extensão de *template*: assim como no Python, é possível que um *template* estenda de outro, tanto sobrescrevendo todo o conteúdo como substituindo ou complementando o conteúdo de blocos especificamente definidos para tal função. Esse recurso é interessante para manter organizados *templates* de *websites* com muitas páginas diferentes;

5. Utilização de filtros: talvez este seja o recurso mais interessante do Jade. Através dos filtros, pode-se definir códigos em CoffeScript, Markdown ou qualquer outra linguagem que possa ser instalada pelo NPM do Node.js, bastando que esteja instalada. Funcionam como uma *tag* <style> ou <script> do HTML tradicional (o que for definido dentro deles é compilado na linguagem que se

está marcando). Por exemplo: para criar um filtro de CoffeeScript, utilizar-se-ia o ":coffe-script" e, dentro, o código CoffeeScript "identado". O Jade, ao processar o código para HTML, também processaria esse bloco para Javascript através do processador de CoffeeScript instalado pelo NPM. A única limitação é em relação à utilização de códigos dinâmicos no bloco: um bloco não deve depender de um valor externo [22];

6. Partições e inclusão de *templates*: é possível partir um *template* HTML em elementos e incluí-los em outros arquivos. Este é um recurso existente em *templates* de *frameworks* de linguagens web, tal como o Django, que é feito em Python, e o Ruby on Rails, que foi feito em Ruby, cujo objetivo de aplicação é seguir o padrão DRY;

7. *Mixins*: funcionam como os *mixins* do SASS, porém, no caso do SASS, o conteúdo do *mixin* é formado por elementos CSS ou blocos de atributos e valores. Já no Jade, o conteúdo dos *mixins* funciona como uma inclusão de estrutura de elementos, diferenciando-se pelo fato de poder passar parâmetros no *mixin*.

4.6 Gulp

"Gulp é uma ferramenta de automação de tarefas feita em JavaScript e rodando em cima do Node.js" [23]. Em outras palavras, em sistemas com ambientes de desenvolvimento complexo, o Gulp torna diversas tarefas, como minimizar arquivos, processar de SASS para CSS, processar de CoffeeScript para Javascript, processar de Jade para HTML, etc, em tarefas automáticas do sistema, sem que o desenvolvedor precise se preocupar em realizar esses passos manualmente enquanto estiver trabalhando naquele projeto. Isso minimiza falhas em produções e documenta o funcionamento do ambiente de desenvolvimento do projeto, tornando fácil a manu-

tenção e a integração contínua no mesmo.

O módulo do Gulp possui algumas funções que são a base para o funcionamento de todos os automatizadores criados. Através dessa base, os desenvolvedores podem implementar a automação de determinada tarefa específica, como em [24]:

1. "gulp.src(globs[, options])": simplesmente procura por todos os arquivos definidos no parâmetro "glob", que é uma *string* contendo a localização dos arquivos em diretórios no mesmo formato que o terminal do *UNIX* utiliza para encontrar arquivos [25]. O *options* altera o funcionamento padrão da função, aumentando os recursos da mesma para situações específicas. É um parâmetro opcional, pois possui valores "*default*";

2. "gulp.dest(path[, options])": faz o processo inverso do "gulp.src(globs[, options])": desloca os arquivos para um determinado diretório definido como *string*, no parâmetro "path". O parâmetro *options*, além de opcional, tem a mesma finalidade de alterar o comportamento padrão da função;

3. "gulp.task(name[, deps], fn)": Aqui é definida a tarefa a ser automatizada. O primeiro parâmetro ("*name*") é apenas um nome para a tarefa. O segundo, opcional, é composto por outras tarefas que devem ser executadas e concluídas anteriormente. O último parâmetro ("fn") é a função que será executada para aquela tarefa, seguindo os padrões de definição de funções do Javascript;

4. "gulp.watch(glob[, opts], tasks)": essa função atua como um observador do projeto. Caso haja alguma mudança em qualquer arquivo definido no caminho do parâmetro "glob", a função executa as tarefas definidas no parâmetro "*tasks*", em ordem. Esse parâmetro deve ser definido na forma de um *array* com o nome das funções. Ademais, o parâmetro "opts" altera o comportamento padrão da função para situações variadas;

5. "gulp.watch(glob[, opts, cb])": este funciona do modo semelhante ao "gulp.watch(glob[, opts], tasks)". O que muda, aqui, é a forma de definir o que será feito em cada alteração. O parâmetro "cb" funciona como um retorno, que será chamado toda vez que ocorrer uma alteração em algum arquivo.

Os passos para começar a usar o Gulp são bem simples [26]:

1. Instalar, utilizando o NPM via linha de comando: "npm install -g gulp";

2. Definir arquivo "gulpfile.js": criação de um arquivo "gulpfile.js" na raiz do projeto (ou onde estiver localizado o arquivo "packages.json", que normalmente está na raiz dos projetos);

3. Instalar módulos das tarefas: por ser um módulo do Node.js, o Gulp utiliza o NPM para instalar os módulos de tarefas que os desenvolvedores criam a fim de automatizar determinadas tarefas. Para a maioria das tarefas comuns em um sistema complexo, existe um módulo feito utilizando o Gulp, que pode ser instalado pelo NPM da mesma forma como ocorre em qualquer outro módulo Node.js;

4. Importar o Gulp: dentro do arquivo "gulpfile.js", importa-se o módulo do "Gulp" responsável por todas as funções que os módulos de tarefas utilizam para seu funcionamento;

5. Implementar os módulos de tarefa: cada módulo de tarefa tem sua documentação e sua implementação; o que não muda é a utilização das funções da API do Gulp. Todos os módulos de tarefas são feitos utilizando a API do Gulp com o objetivo de terem uma função dentro de tarefas customizadas que o desenvolvedor irá criar;

6. Executar no terminal os comandos do Gulp: cada tarefa criada no "gulpfile.js" pode ser executada no terminal através do formato "gulp <NOME-DA-TAREFA>"; o mesmo serve para a função "gulp.watch()", como "gulp watch". Esse procedimento produzirá um servidor Node.js no diretório do projeto com a finalidade de monitorar as alterações nos arquivos.

4.7 Como funciona o *framework* Lâmpada Mágica

4.7.1 Instalação

Para instalar o *framework* é necessário ter pré-instalado o Yeoman e o Node.js. Após a instalação de ambos, basta rodar, em um terminal, "yo lampada" no diretório onde se deseja instalar o Lâmpada Mágica. Esse comando gerará a estrutura do Lâmpada Mágica.

O segundo passo é instalar as dependências para o funcionamento do Lâmpada Mágica, que estão definidas no arquivo "packages.json". Na raíz da pasta gerada pelo Yeoman, basta rodar "npm install" e o Node.js instalará todas as dependências em uma pasta chamada "node modules".

Esses passos garantem a instalação do Lâmpada Mágica. O comando para monitorar a alteração dos arquivos do projeto é "gulp monitorar". Abaixo é demonstrado com mais clareza como funcionam as tarefas do Gulp presentes no "Lâmpada Mágica".

4.7.2 Compilador de CoffeeScript

Figura 4.6: Tarefa para processar de CoffeeScript para Javascript

```
1  gulp.task('coffee', function() {
2      // runSequence('limpar-js');
3      gulp
4          .src(parametros.scripts + '/*.coffee')
5          .pipe(plumber())
6          .pipe(coffee({ bare: true}))
7          .pipe(plumber.stop())
8          .pipe(gulp.dest(parametros.destino + '/js'))
9          .on('error', function() { gutil.log(); });
10 });
```

Linha 1: Definição de nome da tarefa e definição da função.

Linha 4: Pesquisa por todos os arquivos com a extensão ".coffee" dentro do diretório que o desenvolvedor definir como o diretório dos arquivos CoffeeScript.

Linha 5: A função "plumber()" pertence ao módulo "gulp-plumber" e garante que o comando "gulp watch" não pare de executar a sequência de tarefas ao encontrar um erro de compilação, em qualquer tarefa.

Linha 6: Aqui é executada a tarefa que, efetivamente, compila o CoffeeScript, transformando-o em arquivos Javascript. Com os arquivos obtidos na linha 4, são gerados arquivos novos no padrão do Javascript. Já o parâmetro "bare: true", pertence à linguagem CoffeeScript e serve para indicar que as variáveis declaradas serão criadas e declaradas em tempo de execução.

Linha 7: Destruição da proteção criada na linha 5, com a finalidade de otimizar a performance da monitoração.

Linha 8: Os arquivos Javascript compilados são movidos para a pasta "js" no dire-

tório que o desenvolvedor definir. Caso a pasta não exista, esta será criada.

Linha 9: Tratamento de erro: caso aconteça qualquer tipo de erro durante o processo, deve-se sinalizar no *log* do terminal.

4.7.3 Compilador de SASS

Figura 4.7: Tarefa para processar o SASS em CSS

```
gulp.task('sass', function() {
    // runSequence('limpar-css');
    gulp
        .src(parametros.estilos + '/*.scss')
        .pipe(plumber())
        .pipe(sass())
        .pipe(autoprefixer({
            browsers: ['last 2 versions'],
            cascade: false
        }))
        .pipe(plumber.stop())
        .pipe(gulp.dest(parametros.destino + '/css'))
        .on('error', function() { gutil.log(); });
});
```

O processo é muito parecido com a tarefa do CoffeeScript, com apenas algumas mudanças:

Linha 4: Os arquivos obtidos são do formato do SASS (".scss").

Linha 6: Essa tarefa pertence ao módulo "gulp-sass" e, através dela, o SASS é processado e são criados novos arquivos no formato CSS.

Linha 7 até a linha 10: Após a criação dos arquivos CSS, essa função irá percorrê-los e acrescentará prefixos aos atributos CSS3, com base na disponibilidade, em até duas versões dos navegadores atuais.

4.7.4 Compilador de Jade

Figura 4.8: Tarefa para processar o Jade em HTML

```
gulp.task('jade', function() {
    // runSequence('limpar-html');
    gulp
        .src(parametros.templates + '/*.jade')
        .pipe(plumber())
        .pipe(jade({ pretty: true}))
        .pipe(plumber())
        .pipe(gulp.dest(parametros.destino))
        .on('error', function() { gutil.log(); });
```

Também é semelhante às outras duas, com apenas algumas mudanças:

Linha 4: Os arquivos obtidos são do formato do Jade (“.jade”).

Linha 6: Essa tarefa pertence ao módulo “gulp-jade” e, através dela, os arquivos “.jade” são processados e cria-se novos arquivos HTML. O parâmetro “pretty” indica que os arquivos HTML, resultantes do processamento, são “identados” utilizando tabulação.

Linha 8: Os arquivos gerados são copiados para a raiz da pasta de destino.

4.7.5 Copiador de Imagens

Figura 4.9: Tarefa para copiar imagens de “origem” para “destino”

```
// Imagens
gulp.task('imagens', function() {
    // runSequence('limpar-imagens');
    gulp
        .src(parametros.imagens + '/*')
        .pipe(gulp.dest(parametros.destino + '/imagens'))
        .on('error', function() { gutil.log(); });
});
```

A tarefa de copiar imagens existe por uma questão de comodidade ao desenvolvedor, partindo do princípio de que, caso se esteja organizando os arquivos em um diretório, então as imagens devem estar ali, também.

Não existe nenhum processamento das imagens; apenas copia-se da pasta de origem e cola-se na pasta "imagens" na pasta de destino.

4.7.6 Tarefas de limpeza

Figura 4.10: Tarefas que excluem arquivos de determinados destinos

```
1  // Deletar os arquivos da pasta destino/css
2  gulp.task('limpar-css',function() {
3      del([parametros.destino + '/css/*.css']);
4  });
5
6  // Deletar os arquivos .html da pasta destino
7  gulp.task('limpar-html',function() {
8      del([parametros.destino + '*.html']);
9  });
10
11 // Deletar os arquivos da pasta destino/js
12 gulp.task('limpar-js',function() {
13     del([parametros.destino + '/js/*.js']);
14 });
15
16 // Deletar os arquivos da pasta destino/imagens
17 gulp.task('limpar-imagens',function() {
18     del([parametros.destino + '/imagens/*']);
19 });
```

Cada tarefa dessas tem uma função Node.js em comum: a "del([glob])". Essa função exclui todos os arquivos que são encontrados no formato "Glob", passado como parâmetro.

Deste modo, existe uma tarefa que exclui cada tipo de formato gerado, para facilitar a manutenção.

4.7.7 Tarefas de minificação

Figura 4.11: Tarefas que comprimem os arquivos

```
1  // Minificar CSS
2  gulp.task('minificar-css', function() {
3      // runSequence('limpar-css');
4      gulp.src(parametros.destino + '/css/*.css')
5          .pipe(plumber())
6          .pipe(minifyCSS())
7          .pipe(plumber.stop())
8          .pipe(gulp.dest(parametros.destino + '/css'))
9          .on('error', function() { gutil.log(); });
10 });
11
12 // Minificar JS
13 gulp.task('minificar-js', function() {
14     // runSequence('limpar-js');
15     gulp
16         .src(parametros.destino + '/js/**.js')
17         .pipe(plumber())
18         .pipe(uglify({ outSourceMap: true}))
19         .pipe(plumber.stop())
20         .pipe(gulp.dest(parametros.destino + '/js'))
21         .on('error', function() { gutil.log(); });
22 });
```

A compressão de arquivos é uma estratégia que os desenvolvedores utilizam para otimizar a performance no processamento do lado do cliente, fazendo com que as requisições de arquivos CSS, JS e HTML busquem por arquivos menores, resultando, assim, em um carregamento mais rápido das aplicações *web*.

A tarefa "minificar-css" "pega" os arquivos CSS da pasta de destino, na linha 4, e utiliza o módulo "gulp-minify-css", na linha 6, para comprimir todos os arquivos. Do mesmo modo, a tarefa "minificar-js" "pega" os arquivos Javascript, na linha 16, e utiliza o módulo "gulp-uglify" para comprimir os arquivos Javascript, na linha 18.

4.7.8 “Agrupador” dos compiladores

Figura 4.12: Tarefa que executa sequência de tarefas

```
1  // Compilar origem
2  gulp.task('compilar-origem', function() {
3      runSequence('coffee', 'sass', 'jade', 'imagens');
4
5      if (parametros.minificar === true) {
6          runSequence('minificar-css', 'minificar-js');
7      }
8  });
9
```

Essa tarefa nada mais é do que um “agrupador” de todas as tarefas que processam CoffeeScript, SASS, Jade e “copiar imagens”. Na linha 3, o comando “runSequence” pertence à biblioteca do Node.js e executa funções em sequência.

Na linha 5, existe um condicional que verifica se o sistema irá comprimir arquivos, ou não, permitindo ao desenvolvedor configurar essa funcionalidade através do arquivo “parametros.js”. Caso o condicional seja verdadeiro, o sistema, além de rodar a sequência de tarefas, irá rodar a “minificação” do CSS e Javascript, posteriormente.

4.7.9 Monitorador

Figura 4.13: Tarefa de monitoração

```
1  gulp.task('monitorar', function() {
2      runSequence('limpar-css', 'limpar-js', 'limpar-html', 'limpar-imagens', 'compilar-origem');
3      gulp.watch(parametros.scripts + '/*.coffee', ['coffee']);
4      gulp.watch(parametros.estilos + '/*.scss', ['sass']);
5      gulp.watch(parametros.templates + '/*.jade', ['jade']);
6      gulp.watch(parametros.imagens + '/*', ['imagens']);
7
8  });
```

A tarefa de monitoração também agrupa diversas outras tarefas. Antes de qualquer processamento, é necessário limpar todas as tarefas de limpeza e o “agrupador” de

compiladores. Isso se faz necessário para manter a pasta de destino sempre atualizada. Como foi dito antes, a tarefa de monitoração nativa do Gulp ("gulp.watch(glob, [task]") cria um servidor "Node.js" em cada pasta que o caminho "Glob" associa.

Dessa forma, o objetivo dessa tarefa é observar o diretório em que estão os arquivos de origem, além de executar tarefas correspondentes ao tipo de arquivo modificado, que está representado entre as linhas 3 e 6. A função "gulp.watch(glob, [task])" recebe como parâmetros o "Glob", para observar, e a tarefa associada, para executar em cada alteração.

5 CONCLUSÃO

Este estudo explorou a criação do *framework* "Lâmpada Mágica", *framework* para criação de ambientes de *front-end* em projetos, baseando-se em técnicas atuais da comunidade Node.js, e permitindo ao desenvolvedor que for utilizá-lo para elaborar um projeto seu, possa ter um ambiente de desenvolvimento para *front-end* funcional. Para isto, foram agrupadas ferramentas de código aberto de grande aceitação na comunidade *front-end*, como Gulp, Yeoman, Node.js, Sass, CoffeeScript e Jade, de maneira que a compreensão do modo de funcionamento do "Lâmpada Mágica" seja plena, tornando-o, assim, facilmente adaptável ao projeto no qual se trabalhará devido à facilidade que é encontrar documentação das ferramentas na *web*.

A forma como o projeto é apresentado faz com que se tenha uma ideia da evolução do *front-end* tradicional para o que o *framework* "Lâmpada Mágica" possui, em termos de tecnologia e recursos. Isso se dá, muito do fato, da comunidade Node.js ter evoluído de forma que os próprios desenvolvedores possam criar soluções à limitações e problemas conhecidos do *front-end* tradicional.

5.1 Principais contribuições

A grande importância deste estudo é dar destaque à ferramentas desenvolvidas recentemente pela comunidade *front-end* que agregam ao *front-end* tradicional muita tecnologia. Também tem importância o fato de ser uma pesquisa aplicada na área de sistemas de informação voltada exclusivamente ao *front-end*. O estudo também permite que profissionais, e pesquisadores, da área de tecnologia possam compreender como é o processo de criação de um *framework* de *front-end* e dessa forma, sejam capazes de desenvolver outros *frameworks*.

O *framework* "Lâmpada Mágica" propõe ao mercado de tecnologia que é possível criar *frameworks* automatizados, adaptáveis e flexíveis.

5.2 Trabalhos futuros

A evolução do "Lâmpada Mágica" atende a oferta das tecnologias que a comunidade Node.js cria, e também às necessidades de projetos. Um complemento interessante ao *framework* seria a implementação de um meio de captar os dados de uso das tecnologias agregadas no "Lâmpada Mágica", isto é, achar um meio de receber *feedbacks* dos usuários para descobrir se determinada ferramenta está caindo em desuso. Dessa forma, o *framework* poderia se tornar mais flexível e personalizado ainda. A ideia é cada vez melhorá-lo, tendo em vista que a manutenção depende das ferramentas, e as dependências são totalmente plugáveis.

REFERÊNCIAS

[1] Pedro Marins. O que é front-end?, January 2014. Disponível em: <http://mobgeek.com.br/blog/o-que-e-front-end>. Acesso em Abril 12, 2015.

[2] Wikipédia. Site, June 2015. Disponível em: <https://pt.wikipedia.org/wiki/Site>. Acesso em Abril 16, 2015.

[3] Thiago Vinícius Varallo Palmeira. Como funcionam as aplicações web. Disponível em: <http://www.devmedia.com.br/como-funcionam-as-aplicacoes-web/25888>. Acesso em Março 22, 2015.

[4] Wikipédia. Html, June 2015. Disponível em: <https://pt.wikipedia.org/wiki/HTML>. Acesso em Abril 12, 2015.

[5] Yuri Pacievitch. Html. Disponível em: <http://www.infoescola.com/informatica/html/>. Acesso em Março 27, 2015.

[6] Wikipédia. Cascading style sheets, June 2015. Disponível em: <https://pt.wikipedia.org/wiki/Cascading_Style_Sheets>. Acesso em Abril 12, 2015.

[7] Maurício Samy Silva. A regra css e sua sintaxe, March 2011. Disponível em: <http://www.maujor.com/tutorial/sintaxetut.php>. Acesso em Maio 1, 2015.

[8] W3C. Document object model (dom), January 2015. Disponível em: <http://www.w3.org/DOM/>. Acesso em Maio 10, 2015.

[9] Ilya Grigorik. Constructing the object model, 2014. Disponível em: <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model>. Acesso em Maio 9, 2015.

[10] Ilya Grigorik. Render-tree construction, layout, and paint, 2014. Disponível em: <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction>. Acesso em Maio 9, 2015.

[11] Wikipédia. Javascript, June 2015. Disponível em: <https://pt.wikipedia.org/wiki/JavaScript>. Acesso em Abril 12, 2015.

[12] Celso Gomes Barreto Junior. Agregando frameworks de infra-estrutura em uma arquitetura baseada em componentes: Um estudo de caso no ambiente aulanet, July 2006. Disponível em: <http://www.maxwell.vrac.puc-rio.br/8623/8623_3.PDF>. Acesso em Junho 15, 2015.

[13] Rafael Henrique Moreira. O que é node.js?, January 2013. Disponível em: <http://nodebr.com/o-que-e-node-js/>. Acesso em Junho 3, 2015.

[14] The Yeoman Team. Writing your own yeoman generator, 2013. Disponível em: <http://yeoman.io/authoring/index.html>. Acesso em Junho 3, 2015.

[15] The SASS Team. libsass. Disponível em: <http://sass-lang.com/libsass>. Acesso em Junho 3, 2015.

[16] The SASS Team. Sass basics. Disponível em: <http://sass-lang.com/guide>. Acesso em Junho 3, 2015.

[17] Loop Infinito. Coffeescript, 2012. Disponível em: <http://coffeescript.loopinfinito.com.br>. Acesso em Junho 4, 2015.

[18] Almir Filho. Uma xícara de coffeescript, 2012. Disponível em: <http://loopinfinito.com.br/2012/09/18/uma-xicara-de-coffeescript/>. Acesso em Junho 5, 2015.

[19] The CoffeeScript Team. Change log, 2009. Disponível em: <http://coffeescript.org/#changelog>. Acesso em Junho 5, 2015.

[20] Forbes Lindesay. Language reference. Disponível em: <http://jade-lang.com/1.9.2/reference>. Acesso em Junho 5, 2015.

[21] Forbes Lindesay. Readme.md, 2015. Disponível em: <https://github.com/jadejs/jade/blob/master/README.md>. Acesso em Junho 5, 2015.

[22] Taswar Bhatti. Jade node.js template engine, filters mixins, July 2014. Disponível em: <http://taswar.zeytinsoft.com/2014/07/28/jade-node-js-template-engine-filters-mixins-part-5/>. Acesso em Junho 6, 2015.

[23] Leonardo Souza. Bye bye grunt.js, hello gulp.js!, January 2014. Disponível em: <http://blog.caelum.com.br/bye-bye-grunt-js-hello-gulp-js/>. Acesso em Junho 8, 2015.

[24] The Gulp Team. gulp api docs, November 2014. Disponível em: <https://github.com/gulpjs/gulp/blob/master/docs/API.md>. Acesso em Junho 8, 2015.

[25] Isaac S. Glob, March 2011. Disponível em: <https://github.com/isaacs/node-glob>. Acesso em Junho 8, 2015.

[26] Guilherme Kalani. Gulp: O novo automatizador, January 2014. Disponível em: <http://tableless.com.br/gulp-o-novo-automatizador/>. Acesso em Junho 8, 2015.