



**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO CEARÁ
IFCE CAMPUS ARACATI
COORDENADORIA DE CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

IVOMAR DE ARRUDA SANTOS

**UM GUIA DE BOAS PRÁTICAS PARA O DESENVOLVIMENTO DE
APLICAÇÕES *WEB* NO *FRONT-END***

**ARACATI-CE
2018**

IVOMAR DE ARRUDA SANTOS

UM GUIA DE BOAS PRÁTICAS PARA O DESENVOLVIMENTO DE
APLICAÇÕES *WEB* NO *FRONT-END*

Trabalho de Conclusão de Curso (TCC) apresentado ao curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia do Ceará - IFCE - Campus Aracati, como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Reinaldo Bezerra Braga

ARACATI-CE
2018

Dados Internacionais de Catalogação na Publicação

Instituto Federal do Ceará - IFCE

Sistema de Bibliotecas - SIBI

Ficha catalográfica elaborada pelo SIBI/IFCE, com os dados fornecidos pelo(a) autor(a)

S237g Santos, Ivomar de Arruda.

UM GUIA DE BOAS PRÁTICAS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB
NO FRONT-END / Ivomar de Arruda Santos. - 2018.

61 f. : il. color.

Trabalho de Conclusão de Curso (graduação) - Instituto Federal do Ceará, Bacharelado
em Ciência da Computação, Campus Aracati, 2018.

Orientação: Prof. Dr. Reinaldo Bezerra Braga.

Coorientação: Prof. Thiago Felipe de Lima Bandeira.

1. Aplicações Web. 2. Desenvolvimento. 3. Otimização. I. Título.

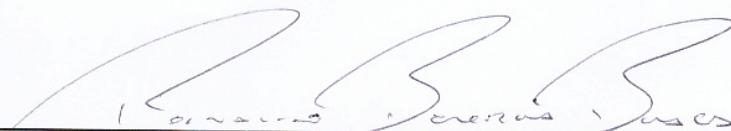
IVOMAR DE ARRUDA SANTOS

UM GUIA DE BOAS PRÁTICAS PARA O DESENVOLVIMENTO DE
APLICAÇÕES WEB NO *FRONT-END*

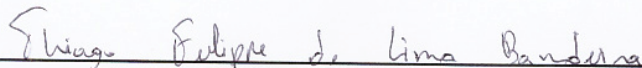
Trabalho de Conclusão de Curso (TCC)
apresentado ao curso de Bacharelado em
Ciência da Computação do Instituto Fede-
ral de Educação, Ciência e Tecnologia do
Ceará - IFCE - Campus Aracati, como re-
quisito parcial para obtenção do Título de
Bacharel em Ciência da Computação.

Aprovado em 18 de Abril de 2018

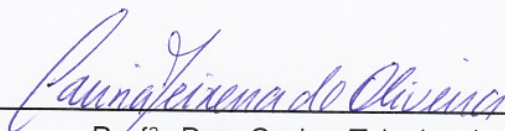
BANCA EXAMINADORA



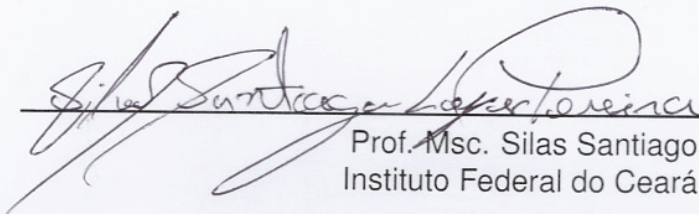
Prof. Dr. Reinaldo Bezerra Braga (Orientador)
Instituto Federal do Ceará (IFCE)



Prof. Bsc. Thiago Felipe de Lima Bandeira (Co-orientador)
Instituto Federal do Ceará (IFCE)



Prof^a. Dra. Carina Teixeira de Oliveira
Instituto Federal do Ceará (IFCE)



Prof. Msc. Silas Santiago Lopes
Instituto Federal do Ceará (IFCE)

DEDICATÓRIA

À minha persistência.

Ao meu orgulho.

Aos meus pais e especialmente irmãos. Também aos Amigos com A maiúsculo pela motivação e por serem pé no saco, obrigado!

Aos meus amigos que compartilham do meu empenho e sacrifício e que se mostraram presentes até mesmo quando busquei ausentar-me. Dedico-lhes o meu sucesso.

À minha esposa que não existe, minha filha que não nasceu e a toda minha família imaginária que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida.

Ao CS:GO e às pessoas com quem convivi nesses espaços ao longo desses anos. A experiência de uma produção compartilhada na comunhão com amigos nesses espaços foram a melhor experiência da minha formação acadêmica. Em especial ao meu clã, Berglândia, Grego e Cheira.

AGRADECIMENTOS

Ao LAR e IFCE pelo ambiente criativo e amigável que proporciona.

Ao Prof. Dr. Mauro Oliveira, pelo discurso no meu primeiro dia de aula, o que contribuiu para a minha permanência nesta instituição.

Ao Prof. Dr. Reinaldo Braga, pela orientação, apoio, confiança e principalmente pela amizade.

À Prof^a. Dra. Carina Teixeira, por ser implacável e motivacional, além de ser fundamentalmente inspiradora para a elaboração deste trabalho.

Ao Professor Thiago Bandeira, pelo suporte e pela dedicação no pouco tempo que lhe coube para auxiliar e co-orientar.

Aos professores, Silas, Felipe, Raquel, Alberto, Evandro, por terem deixado valores, algo mais importante que apenas a grade curricular.

A todos que direta ou indiretamente fizeram parte da minha formação profissional, o meu muito obrigado.

RESUMO

Atualmente, é perceptível a crescente utilização e difusão dos serviços existentes na rede mundial de computadores, também conhecida pela sigla WWW (em inglês *World Wide Web*) ou simplesmente *Web*, que é um sistema de documentos interligados e que são aplicados na Internet. Sua arquitetura é dividida principalmente em duas partes interconectadas, chamadas de cliente e servidor. Neste modelo, as responsabilidades são repartidas e juntas garantem a execução de serviços e sistemas de redes. O lado servidor, como o nome explicita, é responsável por servir seus clientes, disponibilizando o serviço para responder a uma demanda do cliente. Por outro lado, o cliente requisita o serviço, ou seja, inicia a interação a partir de uma requisição do usuário, através de um navegador (*browser*). Desta forma, o usuário pode realizar requisições de conteúdo dinâmico com o servidor, através de diversos protocolos de comunicação, sendo o HTTP (*Hypertext Transfer Protocol*) o mais comum. Além disso, as tecnologias empregadas para o perfeito funcionamento de uma aplicação *Web* também são subdivididas. É comum observar que, geralmente, toda interação visual está localizado no cliente. Já no servidor, localiza-se todo o serviço de armazenamento de dados e regras de negócio da aplicação *Web*. Com isto, é sabido que o serviço rodando no lado cliente serve como interface direta com o utilizador. Consequentemente, sistemas mal estruturados e incompletos podem fornecer ao usuário uma péssima primeira impressão. Portanto, é responsabilidade do desenvolvedor garantir que erros não ocorram, tais como: Ausência de retorno visual das interações; Tratamento correto de dados e validações; Aparência da aplicação e incompatibilidade entre dispositivos. Em outras palavras, é importante evitar erros que impactam o funcionamento do sistema e geram desconforto para a usabilidade. Neste contexto, este trabalho apresenta boas práticas de desenvolvimento de serviços *Web* e explora técnicas de otimização da experiência do usuário ao navegar através de aplicações *Web*. São utilizados padrões definidos por entidades reguladoras (W3C) e grandes grupos atuantes (WHATWG).

Palavras-chave: Aplicações *Web*; Desenvolvimento; Otimização.

ABSTRACT

Nowadays, it's noticeable the growing use and dissemination of existing services on the World Wide Web, also known by the acronym WWW or simply Web, which is a system of interconnected documents and that are applied on the Internet. Its architecture is divided mainly into two interconnected parts, called client and server. In this model, the responsibilities are shared and guarantee the execution of services and systems of networks. The server side, as the name explicitly, is responsible for serving its customers, making the service available to respond to a customer's demand. On the other hand, the client requests the service through a browser, initiating the interaction from a user request. In this way, the user can perform dynamic content requests with the server, through several communication protocols, being HTTP (Hypertext Transfer Protocol) the most common. In addition, the technologies employed for the perfect functioning of a Web application are also subdivided. It is common to note that, generally, every visual interaction is located on the client side. Even in the server side, it locates the service of data storage and business rules of the Web application. Thus, it is known that the service running on the client side serves as a direct interface with the user. Consequently, poorly structured and incomplete systems can provide the user a poor first impression. Therefore, it is a developer's responsibility ensure that errors do not occur, such as: absence of visual feedback of interactions; Correct treatment of data and validations; Appearance of the application and incompatibility between devices. In other words, it is important to avoid errors that impact the functioning of the system and generate discomfort for usability. In this context, this paper presents a set of good Web service development practices, exploring user experience optimization techniques during the use of Web applications. Standards defined by regulators (W3C) and large working groups (WHATWG) are used.

Keywords: Web Applications; Development; Optimization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Requisição HTTP.	19
Figura 2 – Árvore DOM.	22
Figura 3 – Exemplo de Código HTML.	25
Figura 4 – Exemplo de Código CSS.	29
Figura 5 – Modelo de Caixa no CSS.	30
Figura 6 – Exemplo de Código JS.	36
Figura 7 – Documento HTML com CSS e JS.	37
Figura 8 – <i>Flash</i> de conteúdo não estilizado.	50
Figura 9 – Aparência Inicial Esperada.	51
Figura 10 –Desempenho antes do <i>Critical CSS</i>	54
Figura 11 –Desempenho depois do <i>Critical CSS</i>	55
Figura 12 – <i>Payload</i> antes da otimização.	57
Figura 13 – <i>Payload</i> depois da otimização.	57

LISTA DE TABELAS

Tabela 1 – Operadores de Precedência.	35
Tabela 2 – Comparativo entre ESLint e JSHint.	44
Tabela 3 – Comparativo entre Pré-Processadores CSS e PostCSS.	46

LISTA DE CÓDIGOS

2.1	Estrutura Básica de um código HTML.	24
2.2	Exemplo de um código CSS.	28
2.3	Inserir CSS externo no HTML.	29
2.4	Exemplo de um código JS.	34
2.5	Documento HTML com CSS e JS.	35
2.6	Exemplo de um código JS.	36
2.7	Exemplo de um arquivo JSON.	38
2.8	Exemplo de um arquivo JSON.	39
3.1	CSS Crítico.	49

LISTA DE ABREVIATURAS E SIGLAS

WWW	<i>World Wide Web</i>
HTML	<i>HyperText Markup Language</i>
CSS	<i>Cascading Style Sheets</i>
JS	<i>JavaScript</i>
W3C	<i>World Wide Web Consortium</i>
HTTP	<i>Hypertext Transfer Protocol</i>
DOM	<i>Document Object Model</i>
API	<i>Application Programming Interface</i>
URL	<i>Uniform Resource Locator</i>
URI	<i>Uniform Resource Identifier</i>
EOF	<i>End of File</i>
OO	<i>Orientação a Objetos</i>
SO	<i>Sistema Operacional</i>
AJAX	<i>Asynchronous JavaScript and XML</i>
JSON	<i>JavaScript Object Notation</i>
UI	<i>User Interface</i>
WS	<i>Web Service</i>
SOAP	<i>Simple Object Access Protocol</i>
REST	<i>Representational State Transfer</i>
WSDL	<i>Web Services Description Language</i>
RPC	<i>Remote Procedure Call</i>
IDE	<i>Integrated Development Environment</i>
NPM	<i>Node Package Manager</i>

SUMÁRIO

1	INTRODUÇÃO	15
1.1	Motivação	15
1.2	Objetivos	16
1.2.1	Objetivo Geral	16
1.2.2	Objetivos Específicos	16
1.3	Organização do Trabalho	16
2	FUNDAMENTAÇÃO TEÓRICA	17
2.1	Ambiente de Programação para a <i>Web</i>	17
2.1.1	<i>Web</i>	17
2.1.2	Protocolo HTTP	18
2.1.3	Navegador	20
2.2	Tecnologias Básicas de uma Aplicação <i>Web</i>	23
2.2.1	HTML	23
2.2.2	CSS	26
2.2.3	JavaScript	30
2.3	Elementos Básicos para uma Aplicação <i>Web</i> Moderna	37
2.3.1	AJAX	37
2.3.2	JSON	38
2.4	Arquitetura de uma Aplicação	40
2.4.1	MVC <i>versus</i> VMP <i>versus</i> MVVM	40
2.4.2	A importância de um serviço <i>Web</i>	41
2.5	Ferramentas Avançadas para o Desenvolvimento <i>Web</i>	42
3	PROPOSTA	43
3.1	Padrão de Estilo de Código	43
3.1.1	Babel	43
3.1.2	ESLint	44
3.1.3	Prettier	44
3.1.4	PostCSS	45
3.1.5	Webpack	47
3.1.6	Git	47
3.2	Técnicas para Otimização	47
3.2.1	Tela de Abertura	48
3.2.2	Critical CSS	49
3.2.3	Reduzir o <i>Payload</i>	52

4	RESULTADOS	54
5	CONCLUSÃO	59
5.1	Trabalhos Futuros	59
	REFERÊNCIAS	60

1 INTRODUÇÃO

Com os avanços nas tecnologias de rede e comunicações, a *Web* se tornou o meio mais utilizado para a distribuição e acesso de informações, inclusive sobre a própria *Web*. Neste processo, tecnologias e linguagens foram projetadas para suprir essa demanda mais crescente de soluções na Internet. Este fato está ligado diretamente às necessidades em tornar os ambientes computacionais, cada vez mais, conectados, interativos e acessíveis.

Ao se pensar no desenvolvimento de um projeto *Web*, é extremamente importante ter uma base sólida de conhecimento sobre as tecnologias disponíveis, pois isto minimiza os problemas inerentes ao processo de criação de uma aplicação para Internet. Portanto, o primeiro passo que deve-se realizar para a criação de uma boa estrutura de *software* é entender, de forma detalhada, as principais tecnologias, respeitando as exigências fundamentais de uma arquitetura confiável para *Web*, sendo esta segura, de fácil manutenção, de alto desempenho, etc. Sendo assim, é impossível desenvolver uma aplicação *Web* de qualidade e escalável sem, antes, compreender e planejar os principais desafios enfrentados durante a fase de desenvolvimento da solução.

Tendo em vista tais desafios, este trabalho aborda um conjunto de técnicas aprovadas e consolidadas pelas comunidades de desenvolvimento *Web*. Estas técnicas têm como objetivo otimizar o desempenho das aplicações *Web*, através da utilização das boas práticas de desenvolvimento no *Front-end*.

No entanto, dominar uma tecnologia não é simplesmente saber como desenvolver uma aplicação, mas saber como esta aplicação se comporta sempre que um recurso é requisitado. Por isso, o guia proposto aborda os conceitos fundamentais de como funciona uma aplicação *Web*, revisando alguns métodos do protocolo HTTP e técnicas de programação assíncrona com JavaScript.

1.1 Motivação

É sabido que, considerando a grande quantidade de informação disponível sobre as tecnologias criadas para a *Web*, como as linguagens HTML, CSS e JavaScript, bem como a grande quantidade de bibliotecas e *frameworks* que oferecem suporte ao desenvolvimento de aplicações usando essas tecnologias, o aprendizado sobre como utilizá-las de forma correta nem sempre é uma tarefa fácil para desenvolvedores.

Para minimizar este problema de aprendizagem, este trabalho apresenta, de

forma simples e objetiva, um guia sobre como utilizar essas tecnologias em qualquer aplicação Web para o *front-end*.

O uso correto destas tecnologias torna o desenvolvimento de aplicações Web mais fácil e produtivo, do ponto de vista da qualidade das aplicações desenvolvidas, devido à sua padronização de código. A padronização também possibilita uma melhor manutenção de código.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho tem como objetivo geral apresentar, de forma sistemática, um guia sobre como utilizar as principais técnicas disponíveis na atualidade. A ideia é apresentar a proposta para atingir o máximo de performance no desenvolvimento de qualquer aplicação Web no *Front-end*.

1.2.2 Objetivos Específicos

Em particular, o presente trabalho propõe a implementação de técnicas e boas práticas para obter otimizações, tais como a diminuição do *payload* e o aumento da velocidade de renderização em páginas Web, tornando a aplicação mais fluida e assim proporcionando uma melhor experiência para o usuário.

1.3 Organização do Trabalho

Este trabalho está organizado como se segue. No Capítulo 2 é feita a fundamentação teórica, detalhando as tecnologias que são necessárias para o entendimento do restante do trabalho, mostrando o histórico de cada tecnologia, bem como sua evolução e funcionamento. No Capítulo 3 é detalhada a proposta, visando potencializar o desempenho de aplicações Web modernas, através do uso de boas práticas de desenvolvimento, além de explanar as técnicas e introduzir ferramentas complementares para auxiliar no desenvolvimento utilizando *code-pattern*. O Capítulo 4 descreve os resultados dos experimentos realizados a partir da abordagem proposta, além de detalhar as técnicas utilizadas para a obtenção dos resultados após a aplicação das boas práticas mencionadas no trabalho para a otimização do desempenho, comprovando assim a efetividade das técnicas no decorrer dos testes. Por fim, no Capítulo 5 é realizada a conclusão deste trabalho, dando direcionamentos para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Ambiente de Programação para a Web

2.1.1 Web

Com o aparecimento da Internet, o significado de *Web* deixou de representar apenas uma rede interligando pontos e ganhou um outro sentido. A *Web* passou a designar uma rede capaz de conectar computadores e pessoas em todo lugar, passando a ser mundialmente conhecida como a *World Wide Web* (WWW). Ou seja, a *Web* significa um sistema de informações ligadas através de hipermídia (hiperligações em forma de texto, vídeo, som e outras animações digitais) que permitem ao usuário acessar uma infinidade de conteúdo por meio da Internet. Estes conteúdos são estruturados através de uma linguagem de marcação como o *HyperText Markup Language* (HTML), personalizados com uma linguagem de estilização *Cascading Style Sheets* (CSS), que podem se tornar dinâmicos com uma linguagem de programação em execução no *Front-End* (i.e. executados no lado cliente), como JavaScript (JS). Após tal processo, o conteúdo pode ser interpretado por um navegador (*browser*) onde são visualizados os conteúdos disponíveis. Desta forma, as páginas podem apresentar informação em diferentes mídias, bem como estarem associadas a dados de estilo ou disponibilizarem funcionalidades interativas ou não. Dentre os diversos navegadores, podem ser destacados os seguintes: Mozilla Firefox, Brave, Opera, Vivaldi, Internet Explorer, Google Chrome, Safari e etc.

O processo de desenvolvimento da *Web* se iniciou em 1980 quando um projeto proposto pela Organização Europeia para a Investigação Nuclear (CERN), chamado de ENQUIRE, foi criado com o intuito de armazenar e reconhecer associações de informações em documentos. Neste projeto, cada documento existente deveria ligar-se a um novo documento. Como resultado, Tim Berners-Lee, funcionário do CERN e inventor do ENQUIRE em 1989, redigiu uma proposta de criação de um grande banco de dados com hiperligações entre documentos. Ou seja, foi apresentada uma solução para resolver o problema de apresentação de informações quando cientistas de várias partes do mundo necessitavam compartilhar dados utilizando diferentes plataformas.

"Em Agosto de 1984 escrevi um artigo ao Chefe do Grupo SW (do CERN), Les Robertson, para descrever um projeto piloto a fim de instalar e avaliar o protocolo TCP/IP em algumas máquinas não Unix do CERN [...] Cerca de 1990, o CERN tinha se tornado o maior sítio da Internet da Europa [...] e do mundo. Um resultado

chave de todos estes fatos foi que em cerca de 1989 a rede Internet do CERN estava a tornar-se a medida a partir da qual Tim Berners-Lee viria a criar a *World Wide Web* como uma ideia verdadeiramente ideal..." Segal (1995)

Tim Berners-Lee também foi o criador das ferramentas necessárias para o pleno funcionamento da *Web*. Ele desenvolveu o Protocolo de Transferência de Hipertexto (HTTP), a Linguagem de marcação de hipertextos (HTML), além do primeiro navegador e servidor *Web*. Toda a descrição do projeto foi escrita por ele e divulgada na *Web* (BERNERS-LEE, 1992). Atualmente, o projeto encontra-se disponível no sítio oficial da *World Wide Web Consortium* (W3C).

A partir daí, a *Web* tem evoluído constantemente. No momento de sua criação, a expectativa foi de apenas publicação de documentos (i.e. publicação de conteúdo sem interação). Ou seja, ela serviria apenas para informar e trocar informação estática, onde o usuário apenas lia e não haveria interação com os elementos da página. Com o passar dos anos, mais especificamente em 2002, surge o conceito de *Web 2.0* onde o usuário tem um papel fundamental na criação e personalização do conteúdo. Nesta vertente da interatividade, os blogs foram bastante difundidos, pois a interação do usuário em poder comentar e gerar conteúdo ajudou na expansão exponencial do volume de dados que hoje trafegam na rede. Já na terceira geração da *Web*, ou *Web 3.0*, o foco está para as aplicações *Web*, bem como para o uso da computação gráfica, semântica e inteligência artificial. Portanto, é possível observar um crescimento na publicidade onde o conteúdo exibido para o usuário é personalizado com base nos seus interesses. Ou seja, é uma versão mais inteligente do proposto inicialmente na *Web 1.0*, onde não havia preocupação semântica e visual com as páginas, recursos bastante explorados nas próximas versões.

2.1.2 Protocolo HTTP

Como apresentado, o *Hypertext Transfer Protocol* (HTTP) foi um dos primeiros protocolos a serem propostos para o modelo WWW. Ele é o protocolo de transferência de hipertexto adotado em toda *Web* e definido nos *Request for Comments* (RFC) de número 1945 e 2616, (BERNERS-LEE; FIELDING; FRYSTYK, 1997) e (GROUP et al., 1999) respectivamente. O protocolo está implementado na camada de Aplicação da Internet e é responsável pelo funcionamento da *Web*. Este protocolo foi desenvolvido com uma arquitetura cliente-servidor, onde toda a comunicação é realizada através de requisições. O cliente abre uma conexão com o servidor HTTP através do TCP e envia uma requisição (*request*) de conteúdo, que é prontamente retornado por meio de uma resposta (*response*) para o cliente.

Figura 1 – Requisição HTTP.



Fonte: [Kurose e Ross \(2017\)](#)

Como observado na (Figura 1), é possível perceber dois clientes se conectando a um único servidor. Cada dispositivo utiliza um navegador diferente em sistemas operacionais e arquiteturas distintas, porém, realizando a requisição HTTP (*request*) e obtendo uma pronta resposta (*response*) do *Servidor Web* em ambas as requisições. O protocolo HTTP funciona independentemente de plataforma, já que, como citado no parágrafo anterior, ele é implementado na camada de Aplicação.

Em cada requisição enviada de um cliente para um servidor, são enviadas também informações detalhadas sobre a requisição, podendo estas irem junto ao corpo (*body*) ou cabeçalho (*header*) da requisição. Podem ser informações sobre o tipo de dado a ser retornado, idioma da resposta ou filtros de dados. Além disso, é passado também um verbo (*HTTP Verb*), que é padrão do protocolo HTTP e define uma série de métodos de requisição responsáveis por indicar a ação a ser executada na representação de um determinado recurso. Cada um deles implementa uma diferente função sendo que alguns recursos são comuns entre todos os verbos. Por exemplo, qualquer método de requisição pode ser do tipo *safe*, *idempotent*, ou *cacheable*. Abaixo pode-se observar a lista completa de verbos existentes como também seus detalhamentos:

- **GET** — esse método é utilizado para solicitar uma representação de um recurso específico, Requisições utilizando o Método GET devem retornar apenas dados;
- **HEAD** — solicita uma resposta de forma idêntica ao processo que ocorre no tipo GET, porém sem um corpo *body* contendo o recurso;

- **POST** — utilizado para submeter uma entidade a um recurso específico, às vezes causando uma mudança no estado do recurso ou solicitando alterações do lado do servidor;
- **PUT** — substitui todas as atuais representações de seu recurso alvo pela carga de dados da requisição;
- **DELETE** — remove um recurso específico;
- **CONNECT** — estabelece um túnel para conexão com o servidor a partir do recurso alvo;
- **OPTIONS** — usado para descrever as opções de comunicação com o recurso alvo;
- **TRACE** — executa uma chamada de loopback como teste durante o caminho de conexão com o recurso alvo;
- **PATCH** — utilizado para aplicar modificações parciais em um recurso.

Atualmente, o protocolo HTTP está passando por uma evolução, saindo da versão 1.1 para o HTTP/2 padronizado no RFC 7540 (([BELSHE; THOMSON; PEON, 2015](#))). Em sua mais nova versão, pode-se observar um ganho de performance, visto que ele não é mais um protocolo criado para ser entendido por humanos, mas lido por máquinas. Por esse motivo, é mais eficiente e traz alguns benefícios que consertam os problemas atuais da *Web*. Otimizado, o HTTP/2 usa multiplexação, um nome complicado para dizer que o navegador abre uma única conexão para baixar múltiplos arquivos. Desta forma, as requisições e respostas são paralelas e assíncronas, ou seja, o seu navegador pede vários arquivos ao mesmo tempo e recebe-os assim que eles estiverem prontos, na mesma conexão. Já no HTTP/1.1 (RFC 2616 e versão mais utilizada atualmente) o navegador abre uma conexão para baixar um único arquivo. Se essa conexão ficar ocupada por muito tempo, seja porque o arquivo é muito grande ou porque o servidor está lento para responder, o carregamento da página simplesmente trava no meio do processo. Há como amenizar esse problema abrindo múltiplas conexões, mas isso é apenas uma solução paliativa e, não uma solução permanente. Este trabalho, entretanto, está direcionado a transmitir apenas boas práticas de programação e dicas de otimização de código e performance, evitando soluções paliativas.

2.1.3 Navegador

Responsável por toda interação visual do lado cliente (i.e. arquitetura *client-server*), o *browser* ou navegador é quem renderiza as páginas *Web*. Além disso, o

navegador trata o DOM (*Document Object Model*), realiza requisições e cuida dos eventos de interação do usuário. Ou seja, sem o navegador, a Internet não teria se desenvolvido tanto quanto vem crescendo nos últimos anos. Isso porque ele deixa tudo mais fácil e intuitivo. Dentre as várias partes de um navegador moderno e funcional, devem ser citado e destacado o motor de renderização, também chamado de mecanismo de renderização, que é quem transforma linguagem de marcação (i.e. HTML, XML, etc.) e informações de formatação (i.e. CSS, XSL, etc.) em um conteúdo formatado para ser exibido em uma tela.

De acordo com o Decálogo da *Web*, definido pelo W3C, qualquer pessoa tem o direito de acessar a informação de forma igualitária. Não importa se o acesso de determinado site é realizado via *tablet*, *desktop*, *smartphone* ou até mesmo a partir de uma TV. Todos têm direito de acessar todas as informações como se estivessem em um *desktop* de última geração. Ainda de acordo com a documentação da W3C, cada um destes meios de acesso utilizam um determinado *browser* para navegar na *Web*.

Todo *browser* carrega apenas um motor de renderização, para o usuário que apenas faz uso do mesmo ao navegar pela *Web*, o processo de faz de forma transparente. No entanto, existem diversos mecanismos de renderização e cada um funciona de maneira distinta para cada dispositivo/equipamento, podendo haver semelhanças ou incompatibilidades entre um e outro. Para o desenvolvedor que é quem constrói um sistema *Web*, não é viável desenvolver um sistema exclusivamente para determinado motor (*engine*). Para tal, a W3C provê uma documentação única e os motores de renderização implementam suas funcionalidades com base nesta documentação a fim de se obter uma padronização ou, no mínimo, um resultado visual eficiente. Portanto, o desenvolvedor precisa apenas utilizar as boas práticas para obter o máximo de produtividade possível e diminuir a quantidade de conflitos por incompatibilidade entre *engines*. Atualmente os dois motores mais utilizados são o *Webkit* e *Gecko* que correspondem aos navegadores: Opera, Safari, Google Chrome, Vivaldi e Brave (*Webkit*, porém, portando para o *Blink*); Mozilla Firefox (*Gecko*, porém, portando para o *Servo*); Além do Internet Explorer (*Spartan*).

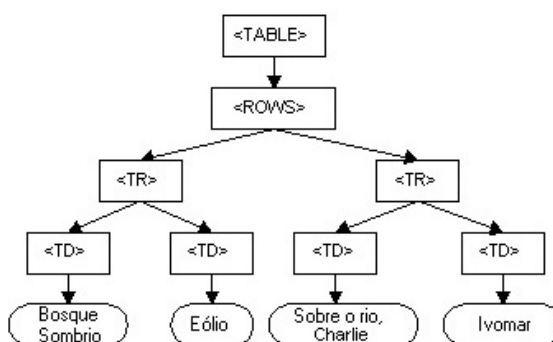
Além da renderização o navegador também contém um interpretador JavaScript, também chamado de motor JavaScript, que é um *software* especializado que interpreta e executa JavaScript ou ECMAScript. Assim como os motores de renderização, também existem diversos interpretadores JavaScript. Dentre eles, os mais impactantes são:

- *SpiderMonkey* — O primeiro interpretador, criado por Brendan Eich, criador do JavaScript, para o Netscape;
- *Squirrelfish Extreme* — Desenvolvido para *Webkit* e desenvolvido pela Google;

- *TraceMonkey* — Desenvolvido exclusivamente para o Mozilla Firefox e mais veloz que o concorrente *Webkit*;
- *V8* — Réplica da Google, uma nova tentativa de se tornar o melhor navegador com o Chrome 2.0. Atual *engine* do navegador.

Pode-se observar que há uma disputa por velocidade e otimização entre todos os motores de renderização, interpretadores e afins. Tudo por uma *Web* mais veloz e cada vez mais padronizada (*Web Standards*). A cada dia, novas versões, novos motores e até linguagens surgem para tornar mais veloz a navegação. No momento em que este trabalho foi escrito, destaca-se a existência de implementações do *Web Assembly*, porém, infelizmente, ainda não há versão estável. Portanto, estas implementações não serão citadas neste guia para o desenvolvimento de aplicações *Web*.

Figura 2 – Árvore DOM.



Fonte: Retirado de [Wood et al. \(2004\)](#)

Não pode-se esquecer do DOM, criado pelo W3C, que é uma multi-plataforma que representa como as marcações em HTML, XHTML e XML são organizadas e lidas pelo navegador. Uma vez indexadas, estas marcações se transformam em elementos de uma árvore que podem ser manipuladas via API. Em outras palavras, pode-se utilizar o DOM para alterar funcionalidades de uma página, tais como o conteúdo, a estrutura ou o estilo.

A (Figura 2) apresenta a representação gráfica de uma árvore DOM após a leitura de uma tabela HTML. Como percebe-se, o DOM é a base para uma outra árvore que é o que realmente um *browser* monta na tela, a Árvore de Renderização (*Render Tree*). A base para todos os nós da árvore DOM é a classe base chamada **Node.h**. Ela possui várias categorias, e as relevantes para renderizar código no navegador são os nós de documentos, elementos e texto.

- **Documentos** é o nó mais importante do DOM, com três classes diferentes: *Document*, que é usado por todos os documentos XML e outros que não sejam

SVG (que também é um XML, porém com marcação já padronizada), *HTML-Document* que como o nome diz, cuida de documentos HTML e *SVGDocument*, responsável pelos documentos SVG e também por outros documentos herdados da classe *Document* (Como o `Document.h` e o `HTMLDocument.h`).

- **Elementos** são todas as *tags* que estão em arquivos HTML ou XML se transformam em elementos da árvore DOM. Considerando a renderização do navegador, um elemento é um nó com uma *tag* que pode ser usada para fazer subclasses específicas, que podem ser processadas de acordo com as necessidades da árvore de renderização (`Element.h`).
- **Texto** é propriamente o texto que vai entre os elementos. Todo o conteúdo das *tags*.

2.2 Tecnologias Básicas de uma Aplicação Web

"HTML é como o pai engenheiro, ele é responsável por estruturar as coisas, garantir que haja um sólido. CSS se parece com a mãe arquiteta, ela embeleza a estrutura, dá brilho e glamour para a criação e fornece o apreço visual pela obra. Já o JS é o filho nerd que brinca com o *Arduino* pela casa, ele faz tudo funcionar, é ele quem faz a mágica acontecer". Fábio Magnoni.

2.2.1 HTML

HTML é a sigla em inglês para *HyperText Markup Language*, que, em português significa linguagem para marcação de hipertexto ¹. O HTML é o construtor de blocos mais básico da *Web*. Ela serve para descrever e definir o conteúdo de um documento. Desde a invenção da *Web* por Tim Berners-Lee, a linguagem HTML evoluiu por oito versões, que são: HTML; HTML +; HTML 2.0; HTML 3.0; HTML 3.2; HTML 4.0; HTML 4.01; HTML 5 ² (versão atual).

Oficialmente, a W3C, principal organização de padronização da *World Wide Web*, considera apenas 5 versões. Isso deve-se ao fato de duas terem sido criadas antes mesmo da W3C, além do HTML 3.0, nunca lançado oficialmente.

"Tim Berners-Lee acreditava que seria possível interligar hipertextos em computadores diferentes com o uso de links globais, também chamados de hiperlinks. Ele

¹ Hipertexto é uma referência aos *links* que interligam páginas, independente se estão localizadas localmente (i.e. mesmo *Website*) ou externamente.

² A WHATWG trabalha na versão do HTML5 (observe os espaços), enquanto a W3C se baseia no HTML5 para documentar o HTML 5

desenvolveu um software próprio e um protocolo para recuperar hipertextos, denominado HTTP. O formato de texto que criou para o HTTP foi chamado de HTML. Tim tomou como base para a criação da HTML a especificação SGML, que é um método internacionalmente reconhecido e aceito, contendo normas gerais para a criação de linguagens de marcação." [Silva \(2011\)](#)

Desde sua criação, em 1991, a HTML tem evoluído bastante tanto em sua especificação quanto em seus recursos e escrita. Entretanto, desde o início, ela utiliza marcação (*markup*) para envolver os conteúdos renderizados pelo navegador, tais como textos, imagens e miscelânea. A base do funcionamento está diretamente ligada às *tags* ou etiquetas e também aos atributos. No trecho de Código 2.1, é importante observar a estrutura básica de um arquivo HTML, chamado de `index.html`

Código 2.1 – Estrutura Básica de um código HTML.

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3   <head>
4     <meta charset="utf-8">
5     <title >Monografia </title >
6   </head>
7   <body>
8     <h1>Aqui escrevo minha monografia </h1>
9
10    <p>Em <strong>2008</strong> conheci a HTML.</p>
11  </body>
12 </html>
13 <!-- Salve engano -->
```

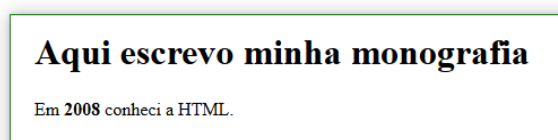
Após uma breve observação do código, segue abaixo a explicação detalhada de sua estrutura básica:

- **Linha 1:** É declarado o tipo do documento em questão, no caso um arquivo HTML.
- **Linha 2:** É iniciada/aberta a *tag* principal. Logo após, há um atributo chamado *lang* que informa o idioma deste documento HTML.
- **Linhas 3 a 6:** Este é o cabeçalho da página e dentro dele estão contidas informações referentes à página, como codificação e título do documento.
- **Linhas 7 a 11:** Chegamos ao corpo do documento, onde o conteúdo está disponível.
- **Linha 8:** Um elemento de título foi criado e um texto foi inserido dentro deste elemento.

- **Linha 10:** Um elemento de texto `<p>` é criado, ele é um parágrafo. Seu conteúdo tem um texto simples e um destaque dentro de um elemento `` que, por padrão, deixa a fonte em negrito.
- **Linha 12:** Finalização/Fechamento da *tag* principal e fim do código.
- **Linha 13:** Exemplo de comentário.
- **Linha 14:** Apesar de não ser visível, EOF ou Fim do Arquivo, é comum em ambientes UNIX a existência de uma linha em branco em arquivos não vazios. Esta é uma boa prática de desenvolvimento que é adotada neste trabalho.

O Resultado do Código 2.1 pode ser observado na Figura 3.

Figura 3 – Exemplo de Código HTML.



Fonte: Elaborado pelo autor.

É natural delimitar o fechamento de uma *tag* com uma barra "/" seguido do restante do nome da *tag* aberta. Elementos desta natureza são chamados de elementos de bloco. Há também os elementos vazios que não necessitam de fechamento de *tag*. No exemplo do Código 2.1, pode-se perceber a *tag* `<meta>`.

Para fins nominais, a terminologia prevê, no exemplo da Linha 10 do Código 2.1, que há em `<p>` uma Abertura de *Tag* e em `</p>` seu fechamento. O que está entre as duas *tags* deste elemento em bloco `<p>` é o conteúdo, e todo o conjunto se denomina Elemento. Elementos podem conter atributos, que são declarados logo após a abertura da *tag* e separados por um espaçamento. Além disso, os atributos podem ser múltiplos ou inexistentes dentro de um elemento e possuem o padrão de Chave-Valor, onde a chave é declarada inicialmente, seguido por um sinal de igualdade, e o valor é declarado dentro de aspas duplas "valor". Ainda na mesma linha 10, observa-se um exemplo de Aninhamento, que é o fenômeno que ocorre quando um elemento é declarado dentro de outro elemento.

A HTML traz para as suas *tags* uma tipografia *case insensitive* (i.e. não diferencia caracteres maiúsculos e minúsculos), ou seja, é possível declarar `<section>` ou `<SECTION>` ou até mesmo `<SeCtIoN>`. No entanto, para garantir um código limpo e fácil de ler e entender por toda a equipe de desenvolvimento, recomenda-se como uma boa prática de desenvolvimento, manter o nome da *tag* em letras minúsculas (ex.: `<section>`). Adicionalmente, caso um dia seja necessário declarar um elemento com

nome composto, a recomendação é utilizar o hífen para separar o nome do elemento, ou seja, declara-se `MeuElemento` como `<meu-elemento>`.

É possível aprender HTML de uma maneira muito fácil, pois há diversas fontes de conteúdo na Internet. A fonte mais completa e constantemente atualizada é o diretório HTML em [Mozilla \(MDN\)](#), mantida pela comunidade ativa da Mozilla.

2.2.2 CSS

CSS (*Cascading Style Sheets*) ou folhas de estilos em cascata é uma linguagem de estilo utilizada para personalizar a apresentação de um documentos *Web*, ou seja, CSS é um padrão de formatação (*Web Standards* ³) para páginas que permite ir além das limitações impostas pela HTML. Quando se deseja garantir uma formatação homogênea e uniforme em todas as páginas de um site as folhas de estilo em cascata facilitam muito o trabalho de criação.

Quando a HTML foi finalmente implementada, o esforço não foi de maneira alguma, formatar os dados, mas exibir o que previamente foi marcado com as *tags* conhecidas na Subseção 2.2.1. Com a popularização da HTML, houve a necessidade de implementar atributos de estilo para alterar algumas aparências da página. Isto fez com que o código ficasse muito complexo e difícil para se manter, visto que tudo estava misturado. Outro fator a se destacar foi a falta de padronização nas implementações dos navegadores daquela época, o que dificultava a visualização das páginas, exibindo conteúdos visualmente distintos entre *browsers*. Estes problemas infelizmente ainda acontecem, porém, pode ser minimizado com boas práticas, como separar o código HTML do CSS e utilizar abordagens mais genéricas e/ou bibliotecas CSS previamente testadas pela comunidade ativa de desenvolvedores. Isso permite evitar o retrabalho por parte do programador, pois o mesmo pode utilizar ferramentas existentes e focar apenas no produto, realizando apenas pequenas intervenções. Tudo isso permite a garantia de uma melhor experiência *cross-browser* ⁴ e *cross-platform* ⁵ para o usuário.

Em 1994, Håkon Wium Lie e Bert Bos propuseram a criação do CSS, após observarem todo este cenário caótico no desenvolvimento de Aplicações *Web*, uma maneira mais fácil de formatar a apresentação da página. Em 1995 a proposta foi

³ *Web Standards* é um conjunto de normas, diretrizes, recomendações, notas, artigos, tutoriais e afins de caráter técnico, e destinados a orientar fabricantes, desenvolvedores e projetistas para o uso de práticas que possibilitem a criação de uma *Web* acessível a todos, independentemente dos dispositivos usados ou de suas necessidades especiais.

⁴ *Cross-Browser* refere-se à habilidade de um site suportar múltiplos navegadores sem comprometer o estilo previamente implementado.

⁵ *Cross-Platform* é a denominação de uma Aplicação *Web* que foi desenvolvida para funcionar independente de plataforma.

apresentada ao *World Wide Web Consortium* (W3C), o qual é uma comissão que define os padrões de programação para a WWW. Recém criada, a W3C demonstrou interesse pelo projeto e montou uma equipe de desenvolvimento que viria a finalizar seu primeiro projeto em 1996. Foi assim que surgiu a primeira versão do CSS. A história do CSS é descrita em mais detalhes no capítulo 20 do livro *CSS: Projetando para a Web*, escrito pelos próprios criadores [Lie e Bos \(2005\)](#).

Atualmente, o CSS encontra-se em sua versão estável de número 3 (CSS 3) e em desenvolvimento da versão 4 (CSS 4), proposta por [Etemad e Stearns \(2015\)](#) nos diretórios da W3C. As evoluções do CSS trouxeram mais recursos e aumentaram a sua abrangência, porém, mantendo o mesmo princípio de formatar a aparência das páginas *Web*. Inicialmente, apenas cores eram mudadas. Após isso, o comportamento pode ser manipulado também. Então foi possível definir o tamanho dos elementos apenas com CSS e por fim, temos animações e efeitos em 3D, além de cada vez mais suporte à responsividade ⁶.

É possível inserir regras de estilo em cascata de três formas junto a um documento HTML. São elas:

- *Inline* — via atributo HTML, adicionando ao atributo `style` o seu código CSS. Desta forma, o código inserido refletirá sobre o elemento ao qual a propriedade foi declarada.
- *Embedding* — via *tag* `<style>`. Dentro do cabeçalho (`<head>`), é possível inserir trechos genéricos de código CSS. Podendo aplicá-los a todo o documento ou apenas a trechos específicos.
- *Linking* — utilizando um ou mais arquivos externos de folha de estilo em cascata com a extensão `.css`, importados dentro do documento HTML.

O *Linking* é considerado a melhor prática para se trabalhar com folhas de estilo em cascata, pois, além de ser a maneira mais recomendada, é também a estratégia mais organizada para descentralizar o estilo *inline* dos componentes. Em outras palavras, utilizando o atributo `<style>`. A vantagem de se utilizar uma importação *Linking* ao invés de *Embedding* se resume pelo fato de reduzir o tamanho de linhas no documento HTML, garantindo uma melhor leitura de código e separação das tecnologias, com o objetivo de facilitar a manutenção de código.

A estrutura de código CSS, assim como HTML é muito fácil de aprender. Juntamente ao arquivo `index.html` (Código 2.1), cria-se um outro arquivo chamado

⁶ Design Responsivo é uma técnica de estruturação HTML e CSS, em que o site se adapta ao browser do usuário sem precisar definir diversas folhas de estilos para cada resolução.

`style.css`, que contém o código CSS (Código 2.2). As regras de estilização podem ser observadas logo abaixo.

Código 2.2 – Exemplo de um código CSS.

```
1 /* h1 vai ser vermelho */
2 h1 {
3     color: red;
4 }
5
6 /* o paragrafo tera uma borda azul */
7 p {
8     border: 1px solid blue;
9 }
```

Como descrito através dos comentários do código acima, observam-se duas modificações visuais. A primeira é a cor vermelha no elemento de título "`<h1>`" e a segunda está relacionada ao parágrafo com a definição de uma borda na cor azul. Diferente do atributo de cor "`color`" que tem apenas uma propriedade, o atributo de borda "`border`" possui três propriedades, que são:

- **Tamanho** — O tamanho foi definido em *pixels*, mas existem diversas outras unidades de medida disponíveis no CSS como `px`, `rem`, `em`, `ch`, `%` dentre outras.
- **Estilo** — O tipo da borda escolhido foi a simples. Ou seja, uma borda sólida com a espessura previamente definida através do tamanho. Além disso, poderia ser uma borda pontilhada "`dashed`", dupla "`double`" ou outro tipo de borda.
- **Cor** — A cor pode ser definida pelo nome em inglês ou por meio de uma codificação em RGB "`rgb(0, 0, 255)`" ou hexadecimal "`#0000ff`", por exemplo.

Todos estes detalhes e valores das propriedades CSS podem ser consultadas no [Mozilla \(MDN\)](#) do diretório CSS.

É importante destacar a estrutura do bloco, inicialmente um seletor é declarado e logo após chaves " " são abertas e o código é inserido dentro delas. É possível perceber a semelhança com os atributos HTML pelo fato de existir um sistema de chave e valor, separados pelos dois pontos ":" e finalizando o comando com ponto e vírgula ";". Apesar deste ser um guia de boas práticas, não são utilizadas classes no código de exemplo. Para utilizar classes na HTML basta atribuir `class="titulo"` no elemento `<h1>` e, em seguida, alterar a linha 2 para `.titulo` no código CSS. O ponto é um identificador de classe ⁷.

⁷ O identificador ponto "." a frente do nome do seletor indica uma classe e o identificador cerquiha "#" referencia um identificador único (i.e. `id="titulo"`).

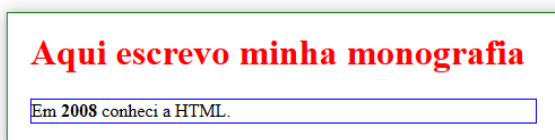
Para garantir o pleno funcionamento e ligar o código de estilo ao arquivo `index.html`, é necessário importá-lo. Para isso, basta adicionar dentro do cabeçalho do Código 2.1 o trecho de código disponível em 2.3, que é um elemento de *link*. Desta forma, seus atributos `rel` e `type` definem uma folha de estilo do tipo CSS e o atributo `href` informa o caminho onde está localizado o arquivo `style.css` em questão. Neste exemplo, o arquivo localiza-se no mesmo diretório do arquivo `index.html` criado.

Código 2.3 – Inserir CSS externo no HTML.

```
1 <link rel="stylesheet" type="text/css" href="style.css" />
```

Como resultado das implementações, é possível observar na Figura 4, uma página *Web* mais colorida.

Figura 4 – Exemplo de Código CSS.



Fonte: Elaborado pelo autor.

É importante salientar que as boas práticas existem para evitar erros e confusões no desenvolvimento do projeto. Se tratando de CSS, um dos maiores problemas está relacionado à precedência. Em outras palavras, é necessário evitar que duas regras de estilo sejam aplicadas para um mesmo elemento HTML especificado. Por exemplo, se houver diferentes modos de estilo (exemplo: *Inline* e *Linking*) as especificações podem entrar em conflito. Desta forma, o navegador é responsável por intermediar e decidir qual estilo conflitante deverá prevalecer baseado em quem tem maior ordem de precedência. A ordem de precedência adotada pelos navegadores felizmente é padronizada e sua definição é:

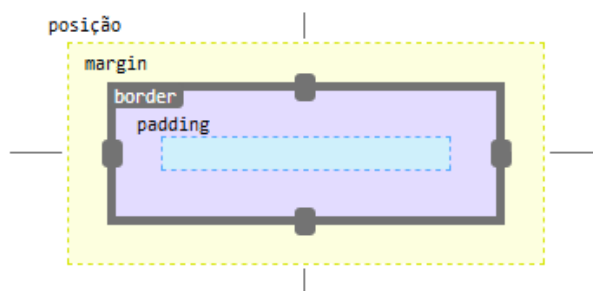
- *Inline* ou Estilo em linha — dentro de um elemento HTML.
- *Embedding* ou Folha de estilo incorporada — definida no cabeçalho do documento.
- *Linking* ou Folha de estilo externa — importada e referenciada.
- *Default* ou Estilo padrão do navegador — estilo predefinido.

É importante lembrar esta ordem de precedência para que se saiba qual valor de estilo tem maior prioridade. Assim sendo, um estilo *inline* tem a prioridade mais

elevada, o que significa que prevalece sobre aquele definido no cabeçalho do documento, que, por sua vez, é prioritário em relação ao definido em uma folha de estilo em cascata externa e esse sobre o formato que o navegador especifica ⁸.

Para os estilos *Embedding* e *Linking* onde o código é definido por blocos, há a possibilidade de se tratar a herança além de pseudo-classes ⁹. Por exemplo, quando o usuário passa o ponteiro do mouse sobre determinado elemento, este evento ou estado especial é chamado de *hover* e pode ser inserido `:hover` após o seletor CSS (exemplo: `h1:hover {}`). Isto faz com que o evento de *hover* seja acionado e um estilo, normalmente diferente do inicialmente renderizado, seja aplicado ao elemento que ativou o evento em questão. Existem diversas pseudo-classes que identificam diversos eventos de página, tanto em relação aos eventos causados pela interação do usuário com a aplicação (exemplo: `:hover`) quanto eventos de estado de elementos (exemplo: `:disabled`) e até herança (exemplo: `:first-child`). Portanto, as pseudo-classes para seletores CSS, tem tornado o desenvolvimento muito mais fácil e intuitivo.

Figura 5 – Modelo de Caixa no CSS.



Fonte: Developer Tools do Mozilla Firefox Nightly (61.0a1).

Outros detalhes importantes sobre CSS estão relacionados à orientação das margens (*margin*), preenchimentos (*padding*) e bordas (*border*). Na Figura 5, é possível observar cada uma das orientações. O espaçamento externo ao elemento definido por `margin` e interno definido por `padding`, que distancia o conteúdo do elemento das bordas, que, trivialmente é definido por `border`.

2.2.3 JavaScript

JavaScript, ou simplesmente JS, é uma linguagem de programação de alto nível, dinâmica e interpretada do lado cliente (i.e. processada pelo navegador), de-

⁸ É comum utilizar estilos CSS para reiniciar a aparência padrão do navegador, visto que, os navegadores apresentam pré-formatações de estilo distintas. Esta é uma boa prática.

⁹ Uma pseudo-classes CSS é uma palavra-chave adicionada aos seletores que especifica um estado especial do elemento a ser selecionado.

desenvolvida por Brendan Eich. O JavaScript, juntamente com HTML e CSS é um dos três pilares que atualmente sustentam a rede mundial de computadores (*World Wide Web*). No entanto, o JS é o principal responsável por tornar as páginas da *Web* interativas. Foi originalmente desenvolvido com o nome de Mocha. Posteriormente, teve seu nome modificado para LiveScript e, por fim, JavaScript. LiveScript foi o nome oficial da linguagem quando ela foi lançada pela primeira vez, na versão beta do navegador Netscape 2.0, em setembro de 1995. Entretanto, teve seu nome alterado em um anúncio conjunto com a *Sun Microsystems*, em dezembro do mesmo ano, quando foi implementado no navegador Netscape, versão 2.0B3.

Neste instante, a Netscape percebeu que a WWW precisava se tornar mais dinâmica, com o objetivo de realizar tarefas simples como verificar se os usuários inseriam valores corretos em um formulário. Algo dessa natureza precisaria enviar os dados para um servidor, para que esse interpretasse os dados e retornasse uma saída.

A linguagem é padronizada pela [ECMA® \(Ecma International\)](#), uma associação Europeia para a padronização de comunicação e informação. Esta versão padronizada de JavaScript, chamada ECMAScript, comporta-se da mesma forma em todas as aplicações que suportam o padrão. Ou seja, as empresas podem usar a linguagem de padrão aberto para desenvolver a sua implementação de JavaScript. O padrão ECMAScript é documentado na especificação ECMA-262 e encontra-se disponível em [Specification \(1999\)](#).

Com o JavaScript é possível criar efeitos especiais para nossas páginas na *Web*, além da possibilidade de explorar uma maior interatividade com nossos usuários. Além disso, o JavaScript é uma linguagem orientada a objetos, ou seja, ela trata todos os elementos da página como objetos distintos, facilitando a tarefa da programação também em multiplataforma.

A sintaxe da linguagem é bastante similar à linguagem C. Apesar de usar Java no nome, as duas linguagens são distintas. Outra curiosidade sobre o JavaScript, é que a linguagem apresenta recursos não disponíveis em C, C++ e em Java, já que ela teve fortes influências das linguagens de script, tais como Awk, Perl e Python. Os *scripts* desenvolvidos em JavaScript são muito populares e amplamente integrados em páginas *Web* devido à facilidade de interação com o *Document Object Model* (DOM). Atualmente, JavaScript é a principal linguagem para programação client-side em navegadores *Web*. Foi concebida para ser uma linguagem script com orientação a objetos baseada em protótipos, tipagem fraca e dinâmica e funções de primeira classe. Possui suporte à programação funcional e apresenta recursos como fechamentos e funções de alta ordem comumente indisponíveis em linguagens populares como Java e C++. É também a linguagem de programação mais utilizada do mundo

por ser a única existente para realizar as interações nos navegadores, o que explica facilmente sua fácil difusão. São características da linguagem:

- **Imperativa e Estruturada** — Suporta os elementos de sintaxe comuns em programação estruturada (exemplo: `if`, `while`, `switch`), com a exceção do escopo. Nativamente, o escopo em JavaScript é definido com base no nível de função, porém há suporte para escopo a nível de bloco através do comando `let`. Diferente da linguagem C, o JavaScript traz o uso do ponto-e-vírgula como opcional ao fim dos comandos. No entanto, é uma boa prática utilizá-lo.
- **Dinâmica** — Assim como na maioria das linguagens de *script*, a tipagem é dinâmica, não associando o tipo a variável, mas ao valor ¹⁰, o que garante uma grande tolerância a erros, uma vez que as conversões automáticas são realizadas durante operações. JavaScript é quase inteiramente baseada em objetos, sendo estes, *arrays* associativos, aumentados com protótipos. Em tempo de execução, as propriedades e seus valores podem ser adicionadas, mudadas, ou deletadas. Além disso, através do comando *eval*, é possível executar comandos da linguagem que estejam escritos em uma *string*. Um exemplo tradicional é quando uma variável possui um determinado valor inteiro em momento e um valor textual em outro instante.
- **Funcional** — As funções são de primeira classe. Isto é, são objetos que possuem propriedades e métodos, que podem ser passados como argumentos, serem atribuídos às variáveis ou retornados como qualquer outro objeto. Há suporte também para funções aninhadas ¹¹ e fechamentos ¹².
- **Baseada em Protótipos** — Para seu mecanismo de herança, a linguagem utiliza protótipos ao invés de classes. É possível simular diversas características de orientação a objetos (OO) baseada em classes com protótipos. Ou seja, não há distinção entre a definição de função e método. A distinção ocorre durante a chamada da função; função pode ser chamada como um método, neste caso, a *keyword* `this` é associada àquele objeto via tal invocação.

¹⁰ Em programação de computadores com linguagens de programação orientadas a objetos, *duck typing* é um estilo de tipagem em que os métodos e propriedades de um objeto determinam a semântica válida, em vez de sua herança de uma classe particular ou implementação de uma interface explícita.

¹¹ Funções 'internas' ou 'aninhadas' são funções definidas dentro de outras funções. São criadas cada vez que a função que as contém (externa) é invocada. Além disso, o escopo da função externa, incluindo constantes, variáveis locais e valores de argumento, se transforma parte do estado interno de cada objeto criado a partir da função interna, mesmo depois que a execução da função interna é concluída.

¹² JavaScript permite que funções aninhadas sejam criadas com o escopo léxico no momento de sua definição e possui o operador `()` para invocá-las em outro momento. Essa combinação de código que pode ser executado fora do escopo no qual foi definido, com seu próprio escopo durante a execução.

A segurança da linguagem é um ponto importante a discutir. A junção do JavaScript e DOM representam uma potencialidade para programadores maliciosos escreverem *scripts* para rodarem em clientes *Web*. Os navegadores são projetados para mitigar riscos. Por exemplo, o JavaScript utiliza uma *sandbox*¹³, onde apenas ações relacionadas à Internet podem ser executadas, não tarefas de propósito geral. Isso impede que o escopo de código acesse informações do Sistema Operacional (SO), bem como dos dados do usuário (exemplo: credenciais, arquivos). A maioria dos *bugs* em JavaScript relacionados à segurança são brechas de uma das regras implementadas em determinado cliente *Web*.

JavaScript tem uma biblioteca padrão de objetos, como: `Array`, `Date`, e `Math`, e um conjunto de elementos que formam o núcleo da linguagem, tais como: operadores, estruturas de controle e declarações. O núcleo do JavaScript pode ser estendido para uma variedade de propósitos, complementando assim a linguagem: *O lado cliente do JavaScript* — estende-se do núcleo da linguagem, fornecendo objetos para controlar um navegador *Web* e seu DOM. Por exemplo, as extensões do lado do cliente permitem que uma aplicação coloque elementos em um formulário HTML e responda a eventos do usuário, como cliques do mouse, entrada de formulário e de navegação da página. *O lado do servidor do JavaScript* — fornece objetos relevantes à execução do JavaScript em um servidor. Por exemplo, as extensões do lado do servidor permitem que uma aplicação comunique-se com um banco de dados, garantindo a continuidade de informações de uma chamada para a outra da aplicação, ou executar manipulações de arquivos em um servidor.

Há diversas expressões e operadores disponíveis na linguagem: Expressões primárias e *left-hand-side*; Incremento e decremento; Operadores unários, aritméticos, relacionais, de igualdade, de deslocamento bit a bit, lógicos binários, bit a bit binários, ternários e de atribuição, além do operador vírgula.

A sintaxe básica, como citada anteriormente, é produto da união de várias linguagens, incorporando recursos e estruturas. JavaScript é *case-sensitive*¹⁴ e utiliza um conjunto de caracteres Unicode¹⁵. No bloco de Código 2.4, pode ser visualizado um exemplo de operações básicas da linguagem. Atente ao fato das linhas 1 e de 4 a 6 representarem comentários, em linha e em múltiplas linhas, respectivamente. Além das declarações de variáveis nas linhas 2 e 7 utilizarem o operador `var`, sendo que existem também os operadores `let` e `const`, usados para declarar variáveis locais em escopo de bloco e constantes de apenas leitura. Ainda na linha 7, a *string*

¹³ O conceito do *Sandbox* é bem semelhante ao de criar uma máquina virtual, de fato, esse método é considerado um tipo de virtualização. Porém, esse sistema é muito mais focado em segurança.

¹⁴ *Case-sensitive* é um anglicismo que se refere a um tipo de análise tipográfica da informática onde há diferenciação entre maiúsculas e minúsculas.

¹⁵ Unicode é um padrão que permite aos computadores representar e manipular, de forma consistente, texto de qualquer sistema de escrita existente.

foi revestida com aspas duplas para representar o tipo *string*. Além deste caractere, as aspas simples ou crases também podem ser utilizadas para delimitar o conteúdo. A linha 9 traz uma declaração de função sem parâmetros de entrada, os parâmetros poderiam ser declarados entre os parênteses "()". Por fim, na linha 13 a função `exibirDados()`, previamente declarada é invocada, como resultado esperado teremos "Aracati, 2018". Os resultados podem ser visualizados no console do navegador, haja visto que foi utilizado a função nativa `console.log()` que imprime um log no console do cliente onde o *script* é executado.

Código 2.4 – Exemplo de um código JS.

```
1 // Declarar ano
2 var ano = 2018;
3
4 /* Declarar
5     cidade
6 */
7 var cidade = "Aracati";
8
9 function exibirDados() {
10     console.log(cidade, ano);
11 }
12
13 exibirDados();
```

O rascunho mais recente da ECMA-262 atualmente é o [Ecma \(2019\)](#) para 2019. Porém como no presente momento encontra-se em estado de *draft*, o padrão ECMA-262 *Edition 6* (ES6) ou como é oficialmente chamado, ECMAScript 2015, é a versão mais sólida e por esta razão foi escolhida como referência para este trabalho. O ES6 define sete tipos de dados para a linguagem JavaScript, os objetos "Object" e outros seis tipos que são:

- *Number* — Números inteiros ou de ponto flutuante.
- *String* — Cadeia de caracteres.
- *Boolean* — Valor binário entre `true` e `false`.
- *Symbol* — Tipo de dado com instâncias únicas e imutáveis.
- *undefined* — Propriedade de valor indefinido.
- *null* — Indicação de Valor nulo, existente, porém nulo.

A condição de existência é muito básica, em testes condicionais a linguagem entende `false`, `null`, `undefined`, zero "0" e *string* vazia "" como negações e qualquer valor distinto como afirmação (i.e. `true`). Tal comparação pode ser reproduzida

com o operador lógico NOT "!" que retorna falso caso o único operando possa ser convertido para verdadeiro; senão, retorna verdadeiro. Com isto, é possível executar um teste de existência utilizando o NOT duas vezes ¹⁶.

A precedência de operadores determina a ordem em que eles são aplicados quando uma expressão é avaliada. Assim como na álgebra, o usuário pode substituir a precedência dos operadores utilizando parênteses. É possível observar a descrição da precedência de operadores (Tabela 1) em ordem decrescente.

Tabela 1 – Operadores de Precedência.

Tipo de operador	Operadores individuais
membro	. []
chamada / criação de instância	() new
negação / incremento	! ~ - + ++ -- typeof void delete
multiplicação / divisão	* / %
adição / subtração	+ -
deslocamento bit a bit	<< >> >>>
relacional	< <= > >= in instanceof
igualdade	== != === !==
E bit a bit	&
OU exclusivo bit a bit	^
OU bit a bit	
E lógico	&&
OU lógico	
condicional	?:
atribuição	= += -= *= /= %= <<= >>= >>>= &= ^= =
vírgula	,

Fonte: Operadores de Precedência em [Mozilla Developer Network \(MDN\)](#).

Assim como nas subseções anteriores e pertencentes a Seção 2.2, esta subseção também utiliza o trecho de código HTML como base (Código 2.1), acrescentando as formações de estilo citadas ao longo da Subseção 2.2.2, bem como suas referências de arquivos criados (`index.html` e `style.css`) para fins de exemplo contínuo ao longo da seção. Como produto das modificações citadas anteriormente, pode-se observar a estrutura final HTML no trecho de Código 2.5. Além disso, há a necessidade de criar um novo arquivo chamado `app.js` para a inserção do código JavaScript que é responsável pela interação da página.

Código 2.5 – Documento HTML com CSS e JS.

```

1 <!DOCTYPE html>
2 <html lang="pt-br">
3   <head>

```

¹⁶ Este é um artifício pouco conhecido, entretanto, sua utilização é bastante fácil e útil para testar muitos casos lógicos com `!!`, funciona como um conversor booleano nativo da linguagem. Assim como utilizar `+`, converte para numérico.

```
4     <meta charset="utf-8">
5     <title >Monografia </title >
6     <link rel="stylesheet" href="style.css">
7 </head>
8 <body>
9     <p>Oi <strong class="name"></strong> tudo bem?</p>
10    <button onclick="rename(this)">Ronaldo</button>
11
12    <script src="app.js"></script>
13 </body>
14 </html>
```

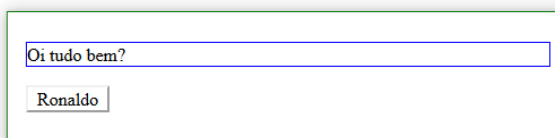
Neste ponto, observa-se um documento HTML modificado, com ligações para um arquivo de estilo CSS e um arquivo de *script* JS. Este *script* ainda inexistente, é descrito no bloco de Código 2.6, logo abaixo. Este arquivo objetiva prover uma interação na página, modificando-a através do DOM.

Código 2.6 – Exemplo de um código JS.

```
1 function rename(el) {
2     const strong = document.querySelector( '.name' );
3     strong.innerText = el.innerText;
4 }
```

O resultado pode ser observado nas figuras 6 e 7, onde a página mantém o mesmo arquivo *style.css*, imutável, adicionado um botão com um evento de *click* que altera parte do texto do elemento `<p>` de nosso HTML.

Figura 6 – Exemplo de Código JS.

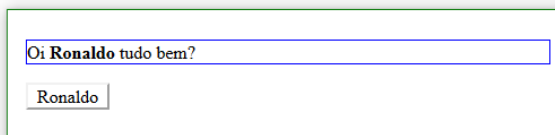


Fonte: Elaborado pelo autor.

No arquivo `app.js`, uma função chamada `rename()` recebe como parâmetro (`el`) o elemento que a invoca através do evento de `onclick=` do botão. Após isto, o elemento que contém a classe `.name` é renomeado para o mesmo texto de dentro do botão (i.e. Ronaldo). Como observado no na linha 9 do Código 2.5, dentro do elemento parágrafo "`<p>`" há um texto livre e dentro deste, um elemento `` com a classe `.name` que receberá o texto do botão. Na Figura 7 é possível observar o resultado desta interação. É importante ressaltar que o texto recém adicionado como retorno do evento está em negrito por formatação padrão do elemento `` no

navegador em questão e que esta aparência inicial pode ser personalizada com uma folha de estilo.

Figura 7 – Documento HTML com CSS e JS.



Fonte: Elaborado pelo autor.

Além do exemplo apresentado, onde foi realizada uma interação simples na página. É possível com a linguagem JavaScript: Criar validações em formulário, realizar troca de dados com o servidor, computar dados e até gerenciar recursos. Nesta subseção apenas exemplos de JavaScript para *Front-End* foram apresentados, porém, é possível executá-lo no *Back-End*, bem como, Banco de Dados e Sistemas Operacionais também podem ser criados com a linguagem. O grande diferencial da linguagem, além da exclusividade na *Web*, é a comunidade participativa muito responsável pela grande difusão e manutenção de novas soluções com a linguagem.

2.3 Elementos Básicos para uma Aplicação Web Moderna

2.3.1 AJAX

AJAX significa *Asynchronous JavaScript and XML*, uma técnica para criar aplicações *Web* melhores, mais rápidas e interativas com a ajuda de XML, HTML, CSS e JavaScript. Foi apresentado ao mundo em [Garrett et al. \(2005\)](#), descrevendo a técnica onde dados podem ser carregados em segundo plano sem necessidade de recarga de página. Isso resultou no ressurgimento do JavaScript, sustentado por novas comunidades, bibliotecas e *frameworks*.

Em aplicações *Web* convencionais, o *front-end* transmite informações para o *back-end* utilizando solicitações síncronas. Isso significa, por exemplo, que um formulário é preenchido, o botão de enviar é acionado e há um redirecionamento para uma nova página com novas informações retornadas do servidor. Com o AJAX, quando o formulário é submetido ao servidor, diferentemente do modo síncrono padrão, o JavaScript realiza uma solicitação ao servidor, interpreta os resultados e atualiza a tela atual. No sentido mais puro, o usuário nunca sabe que algo foi transmitido ao servidor.

AJAX é uma tecnologia *Web* independente de software e de servidor, baseada em *Open Standards* (i.e. Padrões Abertos). É comum o uso de XML como formato

para o recebimento de dados do servidor, embora qualquer formato, incluindo texto simples, possa ser usado. Utiliza o paradigma *data-driven* em vez de *page-driven*, o que permite que o usuário continue utilizando a aplicação enquanto o AJAX solicita informações do servidor, tudo em segundo plano e assíncrono através de objetos de requisição `XMLHttpRequest`, próprios do navegador. AJAX é capaz de realizar requisições em todos os verbos HTTP disponíveis na Subseção 2.1.2 e utiliza JavaScript para fazer tudo.

2.3.2 JSON

JSON é o acrônimo de *Javascript Object Notation* ou Objeto de Notação JavaScript. É um formato leve de intercâmbio de dados padronizado no RFC 4627 em [Crockford \(2006\)](#). Para seres humanos, é fácil de ler e escrever, para máquinas, é fácil de interpretar e gerar. Está baseado em um subconjunto da linguagem JavaScript, a notação, origem do nome. JSON é bastante similar ao XML, com a mesma utilidade, mesmo intuito, porém mais leve. Apesar do nome ser bem sugestivo, não necessariamente deve ser utilizado com JavaScript, é um formato de texto e completamente independente de linguagem. Muitas linguagens hoje em dia dão suporte ao JSON, é meio que um novo método, substituto do antigo e conhecido XML. Ele é muito usado para retornar dados vindos de um servidor utilizando requisições AJAX para atualizar dados em tempo real. JSON está constituído em duas estruturas:

- Uma coleção de pares nome/valor. Em várias linguagens, isto é caracterizado como um *object*, dicionário, *hash table* ou arrays associativas.
- Uma lista ordenada de valores. Na maioria das linguagens, isto é caracterizado como uma *array*, vetor, lista ou sequência.

Estas são estruturas de dados universais. Em JSON, os dados são apresentados desta forma: Um objeto é um conjunto desordenado de pares nome/valor. Um objeto começa com uma chave de abertura "{" e termina com uma chave de fechamento "}". Cada nome é seguido por dois pontos ":" e os pares nome/valor são seguidos por vírgula ",". Uma *array* é uma coleção de valores ordenados. O *array* começa com colchete de abertura "[" e termina com colchete de fechamento "]". Os valores são separados por vírgula. Por final, um valor pode ser uma cadeia de caracteres, um número, booleano, valor nulo, objeto ou até uma *array*. As estruturas podem estar aninhadas.

Código 2.7 – Exemplo de um arquivo JSON.

```
1 {  
2   "letras" : [  

```

```
3     { "letra": "A", "numeros": [ { "numero": 5 }, { "numero": 10 } ] },
4     { "letra": "B", "numeros": [ { "numero": 7 }, { "numero": 17 } ] },
5     { "letra": "C", "numeros": [ { "numero": 9 }, { "numero": 33 } ] }
6 ]
7 }
```

Nos códigos 2.8 e 2.7, é possível observar o quão menor é o arquivo JSON em comparação ao XML, visto que sua estrutura contribui por ser um formato de texto mais simplificado. Os blocos de código contêm, respectivamente para XML e JSON, 509 e 237 caracteres. Uma diminuição de aproximadamente 53% na quantidade de caracteres necessária para se chegar ao mesmo resultado. Optar pela utilização do JSON em relação ao XML é uma forma de otimização, visto a menor quantidade de dados a serem processados e posteriormente trafegados em rede.

Código 2.8 – Exemplo de um arquivo JSON.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <root>
3   <letras>
4     <letra>A</letra>
5     <numeros>
6       <numero>5</numero>
7       <numero>10</numero>
8     </numeros>
9   </letras>
10  <letras>
11    <letra>B</letra>
12    <numeros>
13      <numero>7</numero>
14      <numero>17</numero>
15    </numeros>
16  </letras>
17  <letras>
18    <letra>C</letra>
19    <numeros>
20      <numero>9</numero>
21      <numero>99</numero>
22    </numeros>
23  </letras>
24 </root>
```

2.4 Arquitetura de uma Aplicação

2.4.1 MVC versus VMP versus MVVM

O principal ponto de partida para decidir qual padrão arquitetural a escolher, é entender suas particularidades e semelhanças, comparando quais as melhores descrições em uma abordagem para o desenvolvimento de software.

Os padrões apresentam objetivos semelhantes, contudo o fazem de maneira distinta e estes objetivos almejam aumentar a modularidade, flexibilidade, testabilidade e manutenibilidade do código. Ainda sobre as semelhanças nas responsabilidades dentro da aplicação, devemos descrevê-las como:

- *Model* — Responsável pelo acesso aos dados, contém a lógica necessária para processar estes dados obtidos a fim de retorná-los na forma necessária para que as outras camadas possam utilizá-los.
- *View* — Tem todo o desenho e formatação da interface, assim como validações específicas e processa os dados obtidos na UI para disponibilizá-los para as outras camadas.

O que diferencia os três padrões de arquitetura no *front-end* é a comunicação entre as camadas e a forma como a terceira camada é organizada e executada. Como observado anteriormente, as camadas *Model* e a *View* são as mesmas para os padrões de arquitetura, distinguindo apenas na terminação dos acrônimos, sendo C, P e VM; Especificadas respectivamente abaixo:

- *Controller* — A peça central do MVC que desacopla o *Model* e o *View*. Responsável por toda a lógica de harmonização de dados e as relações entre entidades, bem como o fluxo de eventos por interação do usuário.
- *Presenter* — Uma evolução do MVC que torna a arquitetura ainda mais modular desacoplando as funções. A interação com o usuário é feita primariamente na *View* que delegará ao *Presenter* uma tarefa, porém nesta relação, o *Presenter* não pode delegar tarefas para a *View*. Devido ao desacoplamento, testar torna-se mais fácil. É possível vincular dados da *View* com o *Model* através de *data binding*. Isto ocorre na variação *Supervising Controller*, em oposição à variação *Passive View* onde a *View* essencialmente só possui o desenho da UI.
- *View-Model* — Diferente dos padrões apresentados anteriormente, este adiciona propriedades e operações ao *Model* para atender as necessidades do *View*, portanto ele cria um novo modelo para a visualização. O *data binding* é feito

entre a *View* e o *Model*, criando o *View-Model*. Com esse padrão é possível reduzir a quantidade de código para manter. Algumas automações são possíveis por ter todas as informações necessárias no *View-Model*.

Um determinado padrão não necessariamente é melhor que outra arquitetura, o critério de escolha sugerido é a compatibilidade com o projeto. Determinados *frameworks* implementam determinada arquitetura em sua própria estrutura, é recomendável utilizar tal padrão a fim de obter o máximo de desempenho deste *framework* em questão, não existindo assim, uma arquitetura melhor que outra por definitivo.

2.4.2 A importância de um serviço Web

Web service (WS) é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis. Os WS são componentes que permitem às aplicações enviar e receber dados.

As bases para a construção de um WS é o padrão REST ou SOAP. O transporte dos dados é realizado normalmente via protocolo HTTP. Os dados são transferidos em formato de texto. Muitas empresas temiam, no passado, prover funcionalidades na Internet devido ao medo de expor seus dados. Mas com advento dos WS elas podem publicar serviços de forma simples e que são totalmente isolados da base de dados. Os *Web services* permitem que a integração de sistemas seja realizada de forma compreensível, reutilizável e padronizada. É uma tentativa de organizar um cenário cercado por uma grande variedade de diferentes aplicativos, fornecedores e plataformas. Os dois padrões mais comumente utilizados em WS são:

- **SOAP** — Um protocolo de transferência de mensagens em formato XML para uso em ambientes distribuídos. O padrão SOAP funciona como um tipo de *framework* que permite a interoperabilidade entre diversas plataformas com mensagens personalizadas. Aplicando este padrão em *Web Services*, geralmente usa-se o WSDL para descrever a estrutura das mensagens SOAP e as ações possíveis em um *endpoint*. Uma das maiores vantagens disso é que várias linguagens e ferramentas conseguem ler e gerar mensagens facilmente. Várias linguagens de programação permitem a geração de objetos de domínio, *Stubs* e *Skeletons* a partir da definição do WSDL, permitindo a comunicação remota via RPC através de chamadas a métodos remotos, inclusive com argumentos complexos, como se fossem chamadas locais. O problema desse padrão, é que ele adiciona um *overhead* considerável, tanto por ser em XML quanto por adicionar

muitas *tags* de meta-informação. Além disso, a serialização e desserialização das mensagens pode consumir um tempo considerável.

- **REST** — Baseado no protocolo de hipermídia HTTP. Porém ele não impõe restrições ao formato da mensagem, apenas no comportamento dos componentes envolvidos. A maior vantagem do protocolo REST é sua flexibilidade. O desenvolvedor pode optar pelo formato mais adequado para as mensagens do sistema de acordo com sua necessidade específica. Os formatos mais comuns são JSON, XML e texto puro, mas em teoria qualquer formato pode ser usado. Isso nos leva a outra vantagem: quase sempre *Web Services* que usam REST são mais "leves" e, portanto, mais rápidos. O problema com o REST pode surgir justamente por causa de suas vantagens. Como a definição do corpo de dados fica totalmente a cargo do desenvolvedor, os problemas de interoperabilidade são mais comuns.

Em geral, SOAP é uma boa opção para instituições com padrões rígidos e ambientes complexos. Muitas ferramentas corporativas tiram vantagem do padrão e possibilitam filtrarem, enfileiramento, classificação e redirecionamento das mensagens trocadas entre sistemas. Praticamente todas as plataformas e linguagens modernas suportam esses conceitos e a solução final é muito mais simples do que o equivalente em SOAP. Além disso, integrações com alto volume de requisições são inviáveis em SOAP. REST é capaz de atender volume e complexidade sem dificuldades, exigindo apenas um mínimo de experiência do desenvolvedor para estabelecer e reforçar os padrões adequados.

2.5 Ferramentas Avançadas para o Desenvolvimento Web

Algumas ferramentas e extensões podem ser utilizadas para tornar o desenvolvimento mais ágil. Estes complementos são detalhados na próxima seção, porém, sua escolha é arbitrária e baseada em sua popularidade. Em outras palavras, as ferramentas mais utilizadas pela comunidade de desenvolvedores e que possuem um maior número de contribuidores. Estas ferramentas disponibilizam recursos que agregam mais funcionalidade ao desenvolvimento, tais como: entrada escrita de código de próxima geração e saída de código compatível com os navegadores; verificação de erros sintáticos; *minification* em arquivos; padronização de estilo de código. Estas extensões podem agregar mais funcionalidade para as três tecnologias base, HTML, CSS e JavaScript, renderizadas pelo navegador.

3 PROPOSTA

Este capítulo apresenta a descrição da proposta do trabalho, introduzindo ferramentas e técnicas que auxiliam no desenvolvimento de aplicações para Internet, disponibilizando *code patterns* que podem ser praticados a fim de manter um código único para toda a equipe, tornando a manutenção facilitada e evitando conflitos de sintaxe e formatação. A organização por seções inicia-se em 3.1, onde são abordadas técnicas de padronização de código bem como a utilização de *plug-ins* para facilitar a escrita de código. Por fim, em 3.2, são descritas técnicas visando a otimização da aplicação.

3.1 Padrão de Estilo de Código

Atualmente, existem diversos *plug-ins* disponíveis para garantir uma escrita padronizada de código, também conhecida por *style code pattern*. Neste trabalho, opta-se pela utilização de determinados *plug-ins* de código fonte aberto devido à sua popularidade com a comunidade de desenvolvedores, indicada pelo número de contribuidores ativos, além de suas documentações mais abrangentes e acessíveis. Estes podem ser utilizados para tornar o desenvolvimento mais padronizado, retirando a responsabilidade do desenvolvedor a garantia de um código limpo. Desta forma, o programador é capaz de se responsabilizar principalmente pelas regras de negócio, pois com o auxílio destes *plug-ins*, a padronização é realizada de forma automatizada.

3.1.1 Babel

Conversor de código e responsável por transpor códigos escritos de uma determinada sintaxe para outra, podendo também minimizar código removendo os comentários e espaçamentos desnecessários (*stripped*). Além disso, o Babel é capaz de renomear variáveis e objetos para uma escrita ilegível por humanos (*uglify*) em meio ao processo de conversão.

A grande maioria dos navegadores não implementa o padrão ES6 por completo, disponibilizando apenas o suporte oficial completo com a versão ES5 e pequenas *features* da versão mais recente. Com Babel é possível escrever o código inteiramente em uma versão mais moderna e obter um arquivo de código gerado em uma versão totalmente compatível com os navegadores mais antigos. Isso, torna mais fácil o processo de migração para o desenvolvedor, que pode escrever código de pró-

ximas gerações sem se preocupar com a compatibilidade e mantendo foco apenas no desenvolvimento.

3.1.2 ESLint

Como pode-se observar na Tabela 2, recomenda-se a utilização do ESLint devido sua popularidade com a comunidade de desenvolvedores, além de sua vasta documentação ¹. Além do ESLint, existem outros *plug-ins*, tal como o JSHint, que realizam tarefas similares. Todavia, além de realizar verificação léxica e sintática, o ESLint permite personalizar regras para erros, e oferece suporte completo ao ES6.

Tabela 2 – Comparativo entre ESLint e JSHint.

	ESLint	JSHint
Favoritos	11.056	7.905
<i>Forks</i>	1.906	1.646
Contribuidores	599	235
Licença	MIT	MIT
Criação	06/2013	11/2010

Fonte: Repositórios Oficiais no GitHub. Data de Acesso: 10 de Abril de 2018.

A função do ESLint neste conjunto com o Babel é a de garantir a padronização do código, sendo configurado em um arquivo `.eslintrc` para delimitar espaçamento, nomenclatura e até comentários, o que torna um código mais rígido e consequentemente único. Suas regras de configuração garantem o uso de boas práticas proibindo a utilização de determinados artifícios de linguagem e aconselhando a utilização de outros, por exemplo: a utilização obrigatória de identidade "===" ao invés de simples igualdade "==" a fim de garantir o mesmo valor e tipo de dado. Como o JavaScript não é uma linguagem compilada, os erros são observados em tempo de execução. Com o auxílio do ESLint, é possível perceber um futuro erro antes mesmo do código ser interpretado. No entanto, outros *plug-ins* podem complementar esta tarefa.

3.1.3 Prettier

Formatador de código para IDE, o *Prettier* embeleza o código utilizando as regras estabelecidas no ESLint (Subseção 3.1.2). Este *plug-in* analisa as regras de escrita de código e o padrão adotado, por exemplo: `standardjs`; `AirBnb`; dentre outros. Ele permite também que, ao salvar alterações no arquivo, o código seja formatado

¹ ESLint é um utilitário de *linting* plugável para JavaScript. Seu guia de uso está disponível em: <https://eslint.org/docs/user-guide/>; Acesso realizado em 9 de Abril de 2018.

automaticamente. Isso torna o desenvolvimento mais rápido e contribui para a diminuição de erros. A extensão oferece suporte para os seguintes editores e IDEs: Sublime Text; Atom; VS Code; dentre outros. Além disso, ele é compatível com as sintaxes: ES6; JSON; Vue; CSS3+; SCSS e derivados.

3.1.4 PostCSS

Assim como o JavaScript, o CSS também possui ferramentas desenvolvidas para oferecer uma gama maior de recursos não disponíveis na versão oficial ou suportados pelos navegadores atuais, como o PostCSS. Aliada a esta justificativa, existe a garantia de compatibilidade *cross-browser* provida pela ferramenta utilizada, que converte a linguagem estendida em linguagem de estilo em cascata nativa.

Existem diversas soluções de pré-processadores CSS, tais como SASS, LESS e Stylus. Estas ferramentas utilizam sintaxe própria, baseada ou compatível completamente com CSS, que implementam recursos avançados a fim de tornar o código mais reutilizável e o desenvolvimento mais ágil. A problemática destes pré-processadores se resume na forma com que implementam suas soluções, muitas vezes amarrando o desenvolvimento e gerando uma incompatibilidade entre pré-processadores. Devido as suas peculiaridades, torna-se, no mínimo, trabalhoso migrar um código escrito originalmente em LESS para Stylus por exemplo.

PostCSS é a solução mais versátil e minimalista, além de ser o projeto mais popular em número de colaboradores. Segundo a Tabela 3, pode-se ver um maior destaque de sua aceitação, além de ser o projeto mais recente. PostCSS também é modular, o que torna possível a instalação de diversos *plug-ins* para a resolução de problemas ou simplesmente por uma instalação limpa. A sua modularidade permite com que o desenvolvedor não seja responsável pela compatibilidade entre navegadores, mas tenha poder de escolher como é executada a solução. É possível, através do *plug-in* `cssnext`, escrever CSS de última geração sem se preocupar com os navegadores que ainda não implementaram as novas especificações de sintaxe. Também existem outros *plug-ins* específicos para cada necessidade, tais como:

- `autoprefixer` — Adiciona prefixos específicos de navegador às regras CSS, baseado nos valores disponíveis no Can I Use ².
- `css-modules` — Remove a necessidade de escrever classes com nomes extensos, basta utilizar o mais genérico, este módulo evita, automaticamente, conflitos em escopos de módulos distintos.

² O *Can I Use* (<https://caniuse.com/>) é uma importante base de dados online que disponibiliza consulta sobre a implementação de regras CSS nos diversos navegadores existentes. Acesso em: 10 de Abril de 2018.

- `stylelint` — A melhor solução para evitar erros de sintaxe em sua folha de estilos.
- `lost` — O *LostGrid* faz uso da função CSS `calc()` para criar grades impressionantes baseadas em frações que são definidas com configuração mínima.

Tabela 3 – Comparativo entre Pré-Processadores CSS e PostCSS.

	SASS	LESS	PostCSS
Favoritos	11.256	15.460	18.112
<i>Forks</i>	1.976	3.465	990
Contribuidores	186	216	249
Licença	MIT	Apache-2.0	MIT
Criação	06/2006	02/2010	09/2013

Fonte: Repositórios Oficiais no GitHub. Data de Acesso: 10 de Abril de 2018.

PostCSS é um parser de CSS, desenvolvido em JavaScript, capaz de criar uma árvore sintática abstrata e depois transformar isso em CSS novamente através de quatro etapas de atuação, que são:

- **Análise Léxica** — O código CSS passa por um processo de *tokenization*. Neste processo, são descartados caracteres como espaços extra, indentação e quebras de linha.
- **Criação da Árvore** — Um algoritmo capaz de processar o *array* de *tokens* e criar uma estrutura em árvore. Relacionando valores à propriedades, propriedades à seletores, seletores à *media queries* e etc.
- **Uso de *Plug-ins*** — A árvore criada no passo anterior é passada sequencialmente por uma lista de *plug-ins*, sofrendo alterações no caminho.
- **Retorno** — É de responsabilidade do *stringifier*; um algoritmo que recebe a árvore modificada e a transforma novamente em CSS.

É importante perceber que o PostCSS em instalação limpa não adiciona variáveis, *mixins*, *partials* ou quaisquer dessas funcionalidades. Tampouco especifica uma nova sintaxe. Se algo é uma alternativa aos pré-processadores conhecidos, são os *plug-ins*. Sem eles, o PostCSS apenas lê o CSS e reescrevê-lo exatamente como ele o encontrou. Os *plug-ins* são os responsáveis por todas as funcionalidades. E esse é o ponto mais importante desse novo ecossistema de processamento de CSS: A responsabilidade, a governança, e o mesmo código não estão centralizados em um único projeto.

3.1.5 Webpack

O *Webpack* é um empacotador de módulos (*bundler*). Seu principal objetivo é agregar arquivos JavaScript para uso em navegador, mas também é capaz de transformar, empacotar ou anexar qualquer recurso ou ativo.

Além de arquivos JavaScript, através da utilização de *plug-in*, pode-se configurar o *Webpack* para modularizar código, executar funções do Babel (Subseção 3.1.1), comprimir imagens, realizar funções de rotina e até iniciar um servidor HTTP para o desenvolvimento da aplicação. *Webpack* é responsável por realizar o *build* e gerar arquivos estáticos. Ele lista as dependências e gera um gráfico de dependência, permitindo que os desenvolvedores utilizem uma abordagem modular para seus propósitos de desenvolvimento. O *bundler* pode ser configurado usando um arquivo de configuração externo chamado `webpack.config.js` ou pode ser usado a partir da linha de comando.

3.1.6 Git

Git é um sistema de controle de versão distribuído e um sistema de gerenciamento de código fonte, com ênfase em velocidade. Bastante recomendado para o desenvolvimento de aplicações, principalmente quando há mais de um indivíduo envolvido na criação de código.

Não necessariamente contribui diretamente para a construção de código padronizado como sugere a Seção 3.1, porém permite uma gerência de código para a equipe, versionando e armazenando um histórico de alterações. Ou seja, ele fornece recursos indispensáveis para o desenvolvimento compartilhado de uma aplicação moderna. Por estas razões, Git localiza-se nesta seção, sendo categorizado como uma boa prática.

3.2 Técnicas para Otimização

Juntamente com a utilização de diversos *plug-ins* para o desenvolvimento ágil de aplicações *Web* modernas, técnicas podem ser aplicadas a fim de obter-se o máximo de eficiência em ambiente de produção tornando a experiência de execução para o usuário mais fluida e leve.

3.2.1 Tela de Abertura

Splash Screen é uma técnica bastante utilizada em aplicações compiladas que consiste em exibir uma imagem ou animação de carregamento enquanto o binário é carregado, passando a impressão ao usuário que a aplicação já está sendo carregada quando na verdade, muitas vezes, o processo já está sendo executado e algum recurso ainda não está pronto. Em aplicações *Web* também é possível criar telas de abertura.

Para aplicar a técnica em um ambiente *Web*, antes é necessário entender como funciona o carregamento de uma página bem como o funcionamento básico do DOM, citado na fundamentação teórica na Subseção 2.1.3, onde o navegador é descrito. Os navegadores baixam o HTML e o CSS do servidor e depois analisam e adicionam as *tags* HTML aos nós do DOM, criando uma árvore chamada árvore de conteúdo. Este processo consiste de três fases que são:

- **Fase 1** — Enquanto o HTML é analisado, o motor de renderização cria uma nova árvore chamada de árvore de renderização. Esta árvore representa os efeitos visuais com os quais os elementos serão exibidos.
- **Fase 2** — Após os processos acima, ambas as árvores passam pelo processo de *layout*, onde o navegador posiciona na área do documento cada nó (elemento). Isto é chamado pelo W3C de esquema de posicionamento, que instrui o navegador onde e como cada elemento deverá ser inserido, conforme três tipos: fluxo normal, *floaters* e posição absoluta.
- **Fase 3** — A fase final chamada *painting* (pintura). É o processo gradual onde o motor de renderização percorre a árvore de renderização aplicando todos os efeitos visuais, como tamanhos, cores etc. Esta fase pode ser observada quando se abre uma página em uma conexão mais lenta, podendo ver os estilos visuais sendo aplicados conforme a página é renderizada.

Como visto no item 3 (*painting*) da lista acima, na fase de pintura, a árvore de renderização é atravessada e o método de *painting* dos renderizadores é solicitado para exibir seu conteúdo na tela. A pintura utiliza o componente de infraestrutura da interface do usuário. É neste momento que deve ser exibido o conteúdo da página. Como é desejada uma tela de abertura, há a necessidade de se tardar o *script* principal para a tela durar determinado tempo previamente determinado ou simplesmente exibir primariamente a animação de carregamento e ocultar esta animação quando o conteúdo for completamente carregado, fazendo com que a página imprima um conteúdo visual estático antes de haver um conteúdo interativo.

O JavaScript dispõe de diferentes janelas de tempo que acontecem durante o carregamento de página. Essas janelas de tempo são delineadas pelos eventos `DOMContentLoaded` e `OnLoad`. O primeiro acontece quando a árvore DOM encontra-se pronta e disponível para manipulação. Já o último, é liberado apenas quando todo o conteúdo é carregado, bem como os estilos e as imagens. É no evento de `OnLoad` que a página não realiza mais requisição de conteúdo inicial, isto é, aquele indicador de carregamento na aba do navegador é finalizado.

3.2.2 Critical CSS

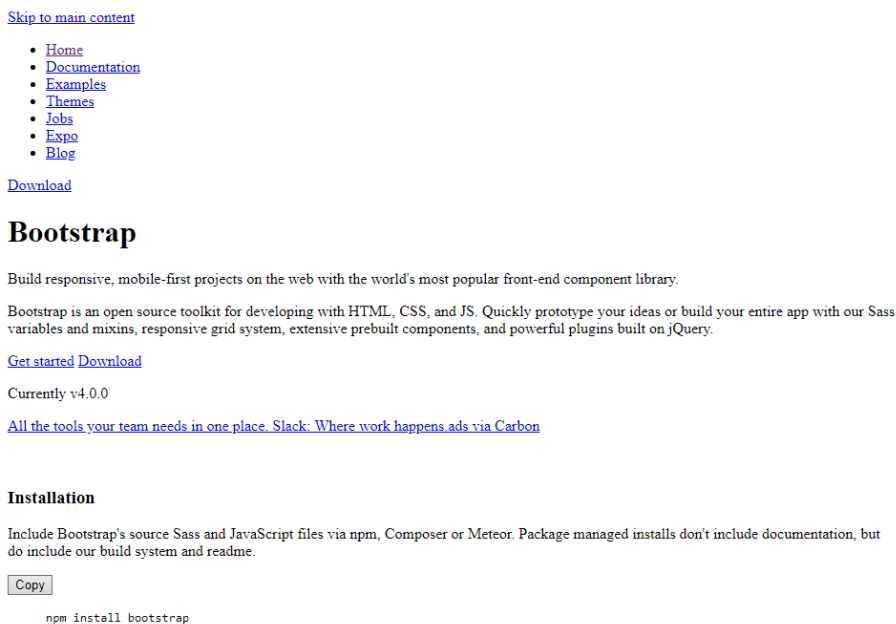
Esta técnica consiste em carregar apenas o conteúdo crítico ao estilo da página, desta forma, é reduzido o tempo para a visualização inicial, tornando uma experiência mais fluida e um carregamento mais rápido ao diminuir a quantidade de código carregado. Sendo assim, o CSS crítico é carregado primeiro e em seguida o não crítico.

Quando as dependências de um documento HTML são carregadas, há um bloqueio de renderização "*render-blocking*", que significa que o navegador não pode exibir a página até que o recurso seja baixado ou tratado de outra forma. Quando esta página é carregada por um navegador da *Web*, ela é lida de cima para baixo. Quando o navegador chegar à *tag* `<link>`, ele começa a baixar a folha de estilos imediatamente e não renderiza a página até que ela seja concluída. Quanto maior o tamanho do arquivo a ser carregado, maior o tempo de espera do usuário até que a transferência e leitura sejam realizadas. Mas não significa que um bloqueio de renderização seja algo ruim, somente deve-se saber utilizá-lo. Caso a folha de estilo não seja carregada no cabeçalho do documento, mas em seu corpo, acontece uma anomalia que pode ser visualizada na Figura 8 chamada de "*flash of unstyled content*".

O ponto ideal que queremos é onde renderizar a página com o CSS crítico necessário para estilizar a visualização principal, mas todo o CSS não crítico é carregado após a renderização inicial. Na Figura 9 pode ser observada a aparência que deveria ser apresentada após o carregamento inicial de página.

Determinados componentes visuais, considerados críticos, devem aparecer de prontidão bem como suas regras de estilo. Todavia, componentes secundários que não representam impacto para a visualização inicial do documento não são necessários para a exibição imediata. Tendo isto em mente, é necessário isolar o CSS crítico do CSS não crítico. Para isto, são criadas duas folhas de estilo: `critical.css` e `non_critical.css`, que separam os estilos crítico do não-crítico, respectivamente. Suas importações podem ser realizadas através do elemento `<link>`, juntamente com `<script>` ou arquivo externo contendo JavaScript.

Figura 8 – *Flash* de conteúdo não estilizado.



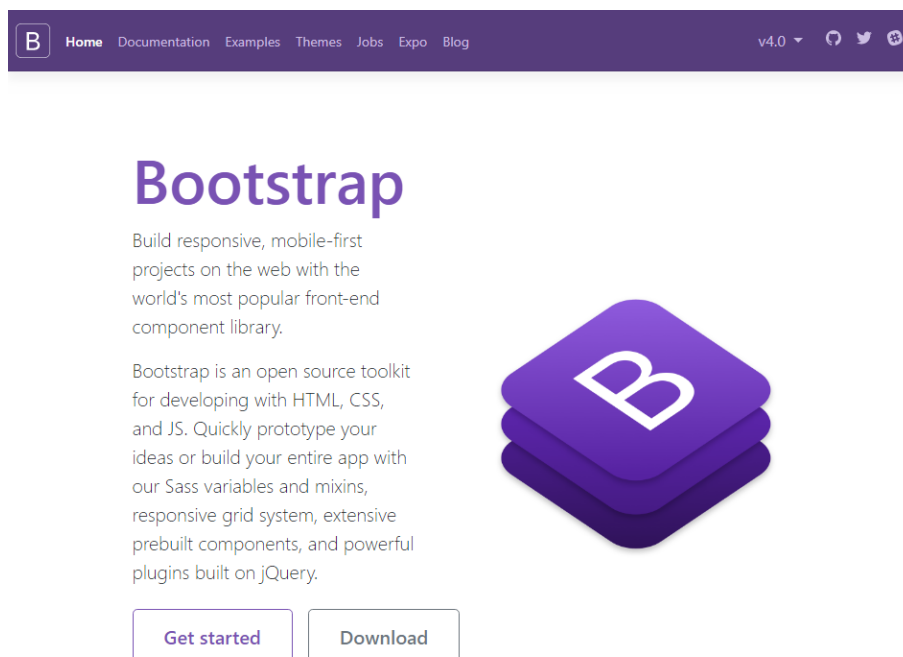
Fonte: Site do Bootstrap. Acesso em 9 de Abril de 2018.

Código 3.1 – CSS Crítico.

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3   <head>
4     <meta charset="utf-8">
5     <title >Monografia </title >
6     <link rel="stylesheet" href="critical.css">
7   </head>
8   <body>
9     <script >
10      require (" non_critical .css " );
11    </script >
12  </body>
13 </html>
```

O trecho de Código 3.1, escrito em ES6 para fins didáticos, ilustra como acontece a importação de uma folha de estilos em cascata crítica. Através do cabeçalho do documento e um arquivo não crítico, importado por meio de código JavaScript no corpo do arquivo HTML. Estas duas importações fazem com que o arquivo crítico seja lido inicialmente, no momento que a página é carregada e, posteriormente, quando a árvore DOM estiver montada no documento. Sendo assim, quando o arquivo de *script* for lido, o restante do estilo CSS é carregado junto ao documento também. É importante mencionar que o exemplo acima não funciona em navegadores atuais e, sua versão em ES5 foi omitida devido à grande extensão de linhas de código.

Figura 9 – Aparência Inicial Esperada.



Fonte: Site do Bootstrap. Acesso em 9 de Abril de 2018.

Como visto na Subseção 3.1.5, o *Webpack* é responsável por resolver programaticamente diversas atividades. Ele pode ser útil com o auxílio de alguns *plug-ins* para a identificação do CSS crítico automaticamente, visto que identificar manualmente pode ser uma tarefa árdua e repetitiva ao longo do período de desenvolvimento. Um destes *plug-ins* é o *critical*, disponível no NPM ³. Este módulo em JavaScript é responsável pela leitura de um documento HTML, para, posteriormente identificar o CSS crítico através de uma renderização de página no servidor. Para isso, é utilizado o PhantomJS ⁴, outro módulo JavaScript disponível no NPM, que, por sua vez, extrai as regras CSS aplicadas na página então renderizada pelo servidor com PhantomJS.

É importante ressaltar que esta solução descrita acima foi idealizada com base no cenário atual sob o protocolo HTTP/1.1, disponível na Subseção 2.1.2. Com o avanço do protocolo e a promessa de suportar multiplexação, o ideal é realizar requisições e respostas assíncronas e paralelas utilizando apenas uma conexão para baixar múltiplos arquivos. Isso pode fazer com que, através do protocolo HTTP/2 ou superior, seja possível obter primordialmente o código CSS crítico em uma modificação da técnica atual, através do uso da multiplexação em conjunto com *Critical CSS*.

No Capítulo 4, são apresentados os resultados deste experimento. Com o

³ *Node Package Manager* (NPM) é o maior gerenciador de pacotes para JavaScript, disponível em: <https://www.npmjs.com/>. Data de acesso: 10 de Abril de 2018.

⁴ Phantom JS é um *headless browser*, ou seja, um navegador que não renderiza as páginas. Ele apenas executa o código que está contido nas páginas, tornando-o uma ferramenta comum para um ambiente de integração contínua.

intuito de validar a otimização em questão, espera-se obter um retorno satisfatório com base nas métricas utilizadas na execução do teste. Além disso, contém as descrições dos resultados dos experimentos.

3.2.3 Reduzir o Payload

Em aplicações para rede de computadores e, principalmente voltadas para *Web*, é inevitável deparar-se com o *payload* ⁵. Uma boa prática de desenvolvimento para aplicações *Web* é tentar reduzir este volume de dados, uma vez que dados trafegam pela *Web* entre aplicações distribuídas. Para isso, há diversas técnicas que podem ser utilizadas para reduzir o *payload*.

Uma dessas técnicas é reduzir o número de requisições. Desenvolver serviços para Internet, pensando em baixas velocidades de conexão, bem como aplicando a estratégia do *mobile first* ⁶, são boas práticas para o desenvolvimento de aplicações *Web*. Determinadas requisições podem ser irrelevantes por completo ou apresentar conteúdo parcialmente desnecessário. Por exemplo, ao realizar uma requisição HTTP com o verbo GET, os dados retornados são: `id; name; birthday; address`. Sendo que, esta requisição será utilizada apenas para popular um `<select>` que, por sua vez, necessita apenas de um atributo `value`, referenciando o valor selecionado de uma opção com um texto indicativo. Os dados necessários para popular este elemento HTML podem ser resumidos a dois: `id` e o dado exibido. Por exemplo, uma lista de usuários necessitaria de `id` e `name`, apenas.

Outra técnica comum, é o agrupamento de arquivos do mesmo tipo, como arquivos de *script* e folhas de estilo, por exemplo. Apesar de aumentar o *payload*, com a transferência de um arquivo único de *bundle*, composto do agrupamento de todos os arquivos JavaScript, cujo o tamanho é a soma de todos os tamanhos dos arquivos aglomerados. A perda no aumento do *payload* é compensada devido ao comportamento atual do protocolo HTTP/1.1, onde não há multiplexação e as requisições são síncronas, ou seja, um arquivo é enviado após o outro. As respostas, no melhor caso, retornam para o cliente com o mesmo custo de tempo de envio de um único arquivo aglomerado. Entretanto, podem ocorrer complicações em rede, tais como: Falha na entrega de um dos arquivos, causado pela perda de pacotes; Atraso na Rede, devido à latência ou congestionamento de fluxo; dentre outros problemas comuns em redes de computadores. Sendo assim, justifica-se a criação de um arquivo composto da união de vários outros para o envio.

⁵ *Payload* refere-se à carga de uma transmissão de dados.

⁶ *Mobile First* é uma estratégia de desenvolvimento contrária ao histórico modelo de desenvolvimento de aplicações *Web*, onde a versão móvel era tratada de forma secundária.

Há também outro artifício que consiste em habilitar a compressão GZIP ⁷ no servidor. Esta técnica é suportada em todos os navegadores e servidores contemporâneos. Com o processo de compactação, todo o conteúdo textual (i.e. HTML, CSS, JS, etc) é comprimido antes de ser enviado para o cliente, contribuindo assim, para a diminuição do tráfego total em rede.

Em conjunto com a agregação de arquivos e redução do número de requisições, é importante também, comprimir os códigos gerados. Isto é, não basta escrever JavaScript elegante, bem documentado, organizado e padronizado. Estas características são agradáveis para se manter e manter, porém o navegador não entende linguagem humana não se importa com a organização do código. Enviar código padronizado via requisição HTTP para o lado cliente não acrescentará valor algum à performance do código. Para isto que deve-se utilizar um compressor de arquivos, por exemplo o módulo `uglify-js`, disponível no NPM. Após agregar todos os *scripts* em um único, tecnicamente, não há nada além de uma simples concatenação em um novo arquivo de *script*. Com uma ferramenta de otimização, como o `uglify-js`, o código agregado é processado e comprimido, a fim de diminuir o tamanho do arquivo gerado como saída. Neste processo, também chamado de *minification*, os espaçamentos são removidos, assim como a indentação, a nomenclatura dos objetos e variáveis é renomeada para uma versão minificada baseada em escopo. Em outras palavras, uma variável chamada de `option` pode ser renomeada para apenas `o` e suas correspondentes dentro do escopo da função seguem o mesmo processo de maneira que não haja conflitos. Todo esse processo é realizado automaticamente e o resultado produz um arquivo de agregação comprimido e ilegível para linguagem humana, apresentando um menor número de caracteres.

Imagens também podem ser otimizadas, no formato `.jpeg`, há uma série de metadados desnecessários. E no formato `.png`, as paletas de cores são definidas, porém, também não são necessárias. Estas informações extra consomem alguns KB de memória a mais, que não representam importância para sua reprodução no cliente. Além desta otimização, é importante utilizar os atributos `width` e `height` para contribuir com os cálculos de dimensões realizados pelo navegador durante a renderização de página. Entretanto, não devem ser utilizados os mesmos atributos para redimensionar o tamanho das imagens importadas, devem ser declarados os tamanhos reais. Por exemplo, se uma imagem nas dimensões `50px` por `50px` for exibida, não deve-se importar uma imagem de tamanho superior e compensar a proporção com os atributos `width` e `height`. O custo desta prática será a importação de uma imagem de tamanho superior e que não é utilizada, recomenda-se otimizar, também, a imagem em seu tamanho.

⁷ GZIP é um formato de compactação de arquivos, identificado pela extensão `.gz`. Descrito na RFC 1952 por Deutsch (1996).

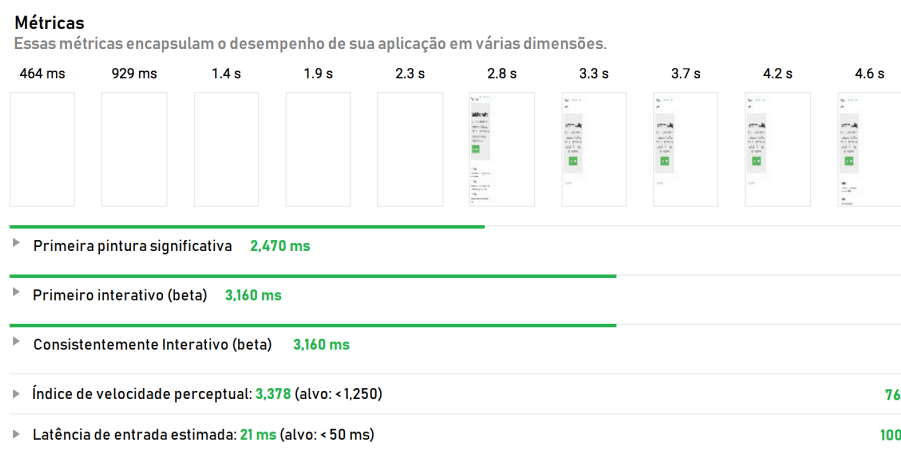
4 RESULTADOS

Neste trabalho, são consideradas as tecnologias *open-source* utilizadas para prover conteúdo na Internet, assim como, as boas práticas de desenvolvimento para aplicações *Web* modernas apresentadas ao longo deste trabalho. São descritos neste capítulo, os resultados dos experimentos propostos no Capítulo 3, que são: Estratégias para a redução do *payload*; Aplicação da técnica de *Critical CSS*.

Os resultados esperados para os experimentos têm como objetivo reduzir o consumo de tempo na renderização de páginas e na resposta das requisições HTTP. Portanto, são utilizados os conceitos explanados e aplicados às boas práticas introduzidas ao longo deste trabalho.

A primeira aplicação prática, proposta na Subseção 3.2.2, descreve a importância de haver separação do CSS crítico. Em outras palavras, há um conjunto de regras de formatação de página que são indispensáveis para a renderização inicial do documento HTML junto ao DOM. Em seguida, após a renderização do conteúdo crítico, os demais componentes visuais e folhas de estilo são requisitadas através de AJAX. Assim, o motor de renderização do navegador executa com mais fluidez a exibição da página devido à otimização no tamanho da árvore DOM a ser renderizada. Esta otimização é impulsionada pela aplicação da técnica de *Critical CSS*. Entretanto, os resultados não contêm dados de requisições realizadas após a renderização inicial. Na Figura 10, pode-se observar o gráfico de desempenho de renderização antes da técnica ser implementada.

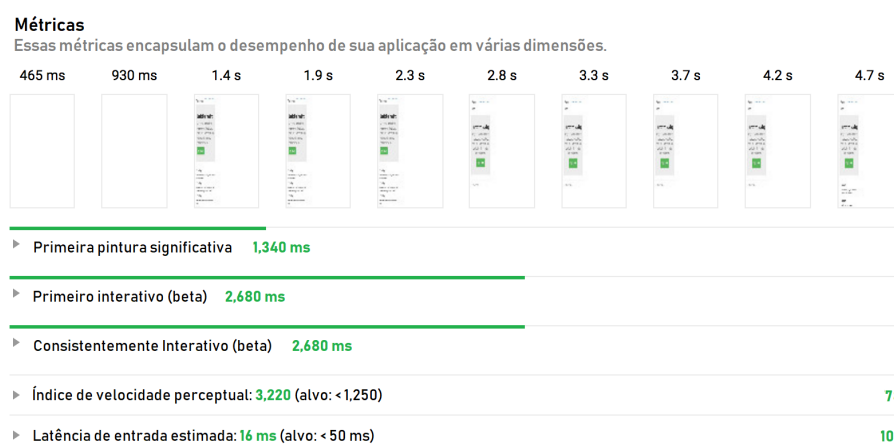
Figura 10 – Desempenho antes do *Critical CSS*.



Fonte: Gerado no *Lighthouse*, por Anthony Gore.

O desempenho da renderização, é medido através da ferramenta *Lighthouse*¹, uma extensão para navegador, automatizada e de código fonte aberto. A *Lighthouse* é desenvolvida e mantida pela mesma equipe responsável pelo Google Chrome, e a mesma é utilizada para auditar a qualidade de renderização das páginas *Web*. Através da utilização desta ferramenta, foi gerado um relatório, apresentado na Figura 10, que lista métricas utilizadas para medir o desempenho da renderização da página *Web*, tais como: Primeira pintura significativa²; Eventos de interação³; Índice de velocidade perceptual⁴ e Latência de entrada estimada⁵. Todavia, apenas o comportamento da primeira pintura significativa é avaliada, devido ao impacto causado pela técnica testada sobre a métrica. Em outras palavras, devido ao tempo que leva até que o usuário possa perceber conteúdo na página. Na Figura 11, pode-se perceber a melhoria do desempenho após o isolamento do CSS crítico do restante da folha de estilos em cascata.

Figura 11 – Desempenho depois do *Critical CSS*.



Fonte: Gerado no *Lighthouse*, por Anthony Gore.

Logo após a implementação da técnica de *Critical CSS*, é possível observar uma diminuição significativa no tempo necessário para a primeira pintura de página. Esta redução, foi de, aproximadamente, 54% em relação ao mesmo resultado antes da aplicação da técnica. É perceptível tal otimização, pois o tempo necessários para a apresentação primeiro efeito visual foi reduzido de 2470ms para 1340ms. Esta melhoria, proposta por Anthony Gore, reduziu, também, o tempo necessário para a liberação dos eventos de interação na página, que podem ser manipulados pelo usuário ou por animações. Além destes eventos de interatividade, as métricas de Latência de entrada estimada e Índice de velocidade perceptual, sofreram variações. Entretanto, não há

¹ Disponível em: <https://goo.gl/H8EpEJ>. Data de Acesso: 18 de Abril de 2018.

² Tempo necessário para que o conteúdo principal de uma página seja visível.

³ Tempo em que a página torna-se interativa.

⁴ Velocidade com que o conteúdo de uma página é visivelmente preenchida.

⁵ Tempo, em milissegundos, que a aplicação necessita para responder a entrada no usuário.

melhora significativa, para estas últimas métricas. Porém a aplicação da técnica não compromete o desempenho das outras métricas auditadas.

Os resultados foram obtidos a partir de um documento HTML com a adição de uma biblioteca CSS (*Bootstrap*), aproximadamente 217 *kilobytes* (que é um tamanho considerável para uma página simples de testes). A escolha da biblioteca é arbitrária e pode ser substituída por qualquer código CSS. Outras métricas podem ser obtidas a partir da extensão *Lighthouse*, porém, não agregam valor à proposta inicial deste trabalho. Devido a este fato, novos resultados não foram adicionados, restando apenas cinco otimizações que culminaram em sucesso após a aplicação da técnica.

O Capítulo 3, também apresenta outra otimização importante, que é a redução do *payload*. Essa estratégia utiliza um conjunto de técnicas voltadas à otimização do desempenho de renderização da página, que incluem: Redução do número de requisições realizadas; Redução da quantidade de dados enviados nas requisições; Aglomeração de arquivos do mesmo tipo; Compressão GZIP pelo servidor; Compressão do código gerado.

O ambiente de testes é fornecido pela Lux Teams ⁶, um sistema integrado de gerência de reuniões, acompanhamento de projetos e tarefas. Bem como, a utilização do *Firefox Developer Edition* ⁷, um navegador criado para atender as necessidades dos desenvolvedores *Web* em seus testes. Os experimentos propostos, utilizam duas versões do sistema, que são: Versão 1.0, desenvolvida sem *code-pattern* e não otimizada; Versão atual (2.1), apesar de não concluída, esta versão, está em fase avançada e os resultados de performance não são interferidos, visto que as métricas utilizadas estão estáveis. Além deste fato, a nova versão utilizou (desde o início do desenvolvimento), *code-pattern* e otimizações. A seguir, é detalhada a redução do *payload* nesta comparação.

Nas Figuras 12 e 13, podem ser observadas, o número de requisições realizadas a partir da entrada com credenciais no sistema (*Login*) até o carregamento completo da tela principal. Além do *payload* e o tempo de transferência necessário para o recebimento de todas as requisições. Os experimentos foram realizados múltiplas vezes para obter-se um resultado justo, com base na média. Além disso, a leitura de desempenho foi medida em horários de menor pico, de forma que, a interferência de rede seja mínima para o resultado final do teste realizado.

O *payload* antes da otimização, na Figura 12, possuía 5,13 *megabytes* correspondentes com as 511 requisições disparadas. O tempo de resposta de todas as requisições durou aproximadamente 71 segundos (1,18 minuto), o que é lento. Após

⁶ Disponível em: <https://app.teamslux.com/>. Data de Acesso: 12 de Abril de 2018.

⁷ Disponível em: <https://www.mozilla.org/pt-BR/firefox/channel/desktop/>. Data de Acesso: 18 de Abril de 2018.

Figura 12 – Payload antes da otimização.

Status	Method	File	Domain	Cause	Type	Transferred	Size	0 ms	20,48 s	40,96 s	1,02 min
200	GET	kanban-page.js	app.teamslux.com	script	js	2,16 KB	7,57 KB				→ 249 ms
200	GET	role-header.js	app.teamslux.com	script	js	1,17 KB	1,77 KB				→ 183 ms
200	GET	role-grid.js	app.teamslux.com	script	js	1,61 KB	3,36 KB				→ 180 ms
200	GET	role-detail.js	app.teamslux.com	script	js	2,80 KB	10,12 KB				→ 254 ms
200	GET	role.js	app.teamslux.com	script	js	1,96 KB	6,02 KB				→ 209 ms
200	GET	kanban-setup-header.js	app.teamslux.com	script	js	1,15 KB	1,64 KB				→ 191 ms
200	GET	kanban-setup-grid.js	app.teamslux.com	script	js	1,62 KB	3,46 KB				→ 187 ms
200	GET	combo-stage-role.js	app.teamslux.com	script	js	1,45 KB	3,21 KB				→ 237 ms
200	GET	combo-task-type.js	app.teamslux.com	script	js	1,40 KB	2,66 KB				→ 201 ms
200	GET	kanban-setup-detail.js	app.teamslux.com	script	js	7,32 KB	83,82 KB				→ 429 ms
200	GET	kanban-setup.js	app.teamslux.com	script	js	1,94 KB	6,27 KB				→ 233 ms
200	GET	dashboard-setup-header.js	app.teamslux.com	script	js	1,18 KB	1,99 KB				→ 175 ms
200	GET	dashboard-setup-grid.js	app.teamslux.com	script	js	2,86 KB	11,77 KB				→ 236 ms
200	GET	dashboard-setup.js	app.teamslux.com	script	js	1,81 KB	5,62 KB				→ 229 ms
200	GET	dashboards-header.js	app.teamslux.com	script	js	1,17 KB	1,89 KB				→ 200 ms
200	GET	dashboards-grid.js	app.teamslux.com	script	js	1,88 KB	5,43 KB				→ 247 ms
200	GET	dashboards.js	app.teamslux.com	script	js	1,80 KB	5,38 KB				→ 210 ms
200	GET	application.js	app.teamslux.com	script	js	6,05 KB	38,23 KB				→ 805 ms
200	GET	account-selection.js	app.teamslux.com	script	js	1,23 KB	1,87 KB				→ 181 ms
200	GET	login.js	app.teamslux.com	script	js	2,27 KB	7,13 KB				→ 203 ms
200	GET	lux-logo.svg	app.teamslux.com	img	svg	6,90 KB	6,66 KB				→ 196 ms
200	GET	logo-lux-teams.svg	app.teamslux.com	img	svg	6,93 KB	6,69 KB				→ 206 ms

511 requests | 5,13 MB / 1,21 MB transferred | Finish: 1,18 min

Filter output

Could not find style data in module named reset-style
 TypeError: this._data_ is undefined
 Campos de senha presentes em uma página insegura (http://). Este é um risco de segurança que permite que credenciais de login do usuário sejam roubadas.

Fonte: Disponibilizado pelo Autor.

implementar as modificações propostas para a redução do *payload*, pode-se observar na Figura 13 o resultado desta otimização.

Figura 13 – Payload depois da otimização.

Status	Method	File	Domain	Cause	Type	Transferred	Size	0 ms	5,12 s	10,24 s	15,36 s
200	GET	q=01yIOPocLackzWASUHTAKasiVwIICgIrnJmVwwo...	tonts.gstatic.com	tont	worlz	13,02 KB	13,02 KB				→ 105 ms
204	OPTIONS	users?id=7&page=1&pageSize=12	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 229 ms
204	OPTIONS	Timezones?getAll=true	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 227 ms
204	OPTIONS	roles?getAll=true	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 228 ms
204	OPTIONS	languages?getAll=true	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 226 ms
204	OPTIONS	taskTypes?getAll=true	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 226 ms
204	OPTIONS	pipelines?getAll=true	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 226 ms
204	OPTIONS	Timezones?getAll=true	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 226 ms
204	OPTIONS	WithAccount?getAll=true&id=7	teams-api-dev.azurewebsites.net	xhr	plain	346 B	0 B				→ 225 ms
200	GET	users?id=7&page=1&pageSize=12	teams-api-dev.azurewebsites.net	other	json	749 B	332 B				→ 280 ms
200	GET	Timezones?getAll=true	teams-api-dev.azurewebsites.net	other	json	5,24 KB	17,28 KB				→ 283 ms
200	GET	languages?getAll=true	teams-api-dev.azurewebsites.net	other	json	572 B	82 B				→ 280 ms
200	GET	roles?getAll=true	teams-api-dev.azurewebsites.net	other	json	755 B	579 B				→ 319 ms
200	GET	taskTypes?getAll=true	teams-api-dev.azurewebsites.net	xhr	json	1,13 KB	3,98 KB				→ 312 ms
200	GET	pipelines?getAll=true	teams-api-dev.azurewebsites.net	xhr	json	1,46 KB	4,42 KB				→ 327 ms
200	GET	Timezones?getAll=true	teams-api-dev.azurewebsites.net	xhr	json	5,24 KB	17,28 KB				→ 306 ms
200	GET	WithAccount?getAll=true&id=7	teams-api-dev.azurewebsites.net	xhr	json	641 B	173 B				→ 324 ms

30 requests | 132,95 KB / 109,89 KB transferred | Finish: 14,96 s | DOMContentLoaded: 4 ms | load: 1,07 s

Filter output

Fonte: Disponibilizado pelo Autor.

Ao final de todas as implementações, observa-se uma redução significativa em todas as métricas avaliadas na Figura 13. Antes da otimização, haviam 511 requisições. Já após a otimização, esse número foi reduzido para 30 requisições, 5.87% do número total de requisições. Além da diminuição da duração total da requisição para

apenas 15 segundos, 21.1% do tempo total de aproximadamente 71 segundos. Esta redução pode ser explicada pela necessidade de consumir o recurso da API REST apenas quando solicitado. Em outras palavras, foi observado que, várias requisições poderiam ser solicitadas por demanda, não havendo necessidade em requisitar todo o conteúdo após a entrada no sistema, o que impactava, inclusive, o carregamento da página inicial. Além desta métrica, o tamanho de dados a serem transferidos, bem como o tempo necessário para todas as requisições serem finalizadas foram reduzidos. Este comportamento deve-se principalmente à redução do número total de requisições. É pertinente observar que ao utilizar-se da Internet para trafegar os dados, a Aplicação *Web* está sucessível a erros e complicações de rede ligados diretamente ao modo que os protocolos que compõem a Internet operam. Isto é, lentidão na entrega causadas por congestionamentos de rede, perda de pacotes, latência na entrega devido a múltiplos saltos, tamanho das requisições e respostas. Todas essas problemáticas são inevitáveis e estão presentes em qualquer rede de computadores convencional ao modelo TCP/IP que é difundido por toda Internet. Entretanto, otimizações podem ser aplicadas, assim como neste trabalho, a fim de diminuir o impacto de um eventual problema de rede. Sendo assim, o usuário final observará um menor atraso na exibição de dados, o que contribui diretamente para melhorar a usabilidade da aplicação em questão.

5 CONCLUSÃO

Com a rápida evolução de soluções *Web*, surgem também diversas formas de desenvolvimento. No entanto, é importante definir quais são as melhores ferramentas e estratégias para alcançar os objetivos de uma aplicação *Web* de qualidade. Levando isso em consideração, este trabalho apresentou um conjunto de técnicas amplamente utilizadas para a otimização de aplicações *Web*, exemplificando através de *scripts* didáticos e de um sistema real em funcionamento.

A partir dos resultados, conclui-se neste trabalho que, a aplicação de boas práticas de programação durante o período de desenvolvimento, influenciam diretamente na performance da aplicação. Como é possível perceber nesta proposta, as otimizações descritas, quando implementadas corretamente, produzem significativos resultados. Esta percepção se faz possível ao utilizar técnicas de potencialização da aplicação focadas em fluidez e experiência do usuário.

5.1 Trabalhos Futuros

Podem ser apresentados, na sequência deste trabalho, a introdução de boas práticas para o desenvolvimento de aplicações *Web* utilizando novos conceitos referente às tecnologias citadas neste trabalho. São exemplos de trabalhos futuros: Técnicas de otimização de código e obtenção de maior desempenho na execução, além da abordagem de novos recursos para as futuras versões da HTML, da CSS e do JS; Adaptação na estrutura e arquitetura da aplicação para se obter o máximo de integração com as novas versões dos protocolos para transferência de arquivos, que causam impacto direto no desempenho das aplicações; Abordagem e inclusão de modelos e conceitos de aplicações *Web*, tais como *Single Page Application* (SPA) e *Progressive Web Apps* (PWA); Integração com novas tecnologias, como *Web Assembly*.

REFERÊNCIAS

- BELSHE, M.; THOMSON, M.; PEON, R. Rfc 7540. *Hypertext transfer protocol version*, v. 2, 2015. Citado na página 20.
- BERNERS-LEE, T. *WWW Project*. 1992. W3C. Disponível em: <<http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html>>. Acesso em: 27.3.2018. Citado na página 18.
- BERNERS-LEE, T.; FIELDING, R.; FRYSTYK, H. Rfc 1945: Hypertext transfer protocol—http/1.0, may 1996. *Status: INFORMATIONAL*, v. 61, 1997. Citado na página 18.
- CROCKFORD, D. Rfc 4627: The application/json media type for javascript object notation (json), july 2006. *Status: INFORMATIONAL*, 2006. Citado na página 38.
- DEUTSCH, L. P. Rfc 1952: Gzip file format specification version 4.3, may 1996. *Status: INFORMATIONAL*, 1996. Citado na página 53.
- ECMA, S. *ECMA-262 ECMAScript® 2019 Language Specification*. 2019. Disponível em: <<https://tc39.github.io/ecma262/>>. Citado na página 34.
- ECMA®. *Ecma International*. Ecma International. Disponível em: <<http://www.ecma-international.org/>>. Acesso em: 2.4.2018. Citado na página 31.
- EDEMAD, K. I. E. J.; STEARNS, A. *CSS Text Module Level 4*. 2015. Disponível em: <<http://www.w3.org/TR/css-text-4/>>. Acesso em: 29.3.2018. Citado na página 27.
- GARRETT, J. J. et al. *Ajax: A new approach to web applications*. 2005. Citado na página 37.
- GROUP, N. W. et al. Rfc 2616 hypertext transfer protocol—http/1.1. *R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee*, 1999. Citado na página 18.
- KUROSE, J.; ROSS, K. *Computer Networking: A Top-Down Approach*. [S.l.]: Pearson Education Limited, 2017. Citado na página 19.
- LIE, H. W.; BOS, B. *Cascading style sheets: designing for the Web*. [S.l.]: Addison-Wesley Professional, 2005. Citado na página 27.
- MOZILLA. *Mozilla Developer Network*. MDN. Disponível em: <<http://developer.mozilla.org/pt-BR/docs/Web/>>. Acesso em: 28.3.2018. Citado 2 vezes nas páginas 26 e 28.
- Mozilla Developer Network. *Operator Precedence*. MDN. Disponível em: <http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>. Acesso em: 3.4.2018. Citado na página 35.
- SEGAL, B. *A Short History of Internet Protocols at CERN*. 1995. CERN IT-PDP-TE. Disponível em: <<http://ben.home.cern.ch/ben/TCPHIST.html>>. Acesso em: 27.3.2018. Citado na página 18.

SILVA, M. S. *HTML 5-A Linguagem de Marcação que Revolucionou*. [S.l.]: NOVATEC, Rio de Janeiro, Brasil, 2011. Citado na página [24](#).

SPECIFICATION, E. L. Standard ecma-262. *ECMA Standardizing Information and Communication Systems*, v. 3, 1999. Citado na página [31](#).

WOOD, L. et al. *Document Object Model (DOM) level 3 core specification*. [S.l.]: W3C Recommendation, 2004. Citado na página [22](#).