INSTITUTO FEDERAL DE SANTA CATARINA – IFSC
CAMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE SAÚDE E SERVIÇOS
CST EM GESTÃO DA TECNOLOGIA DA INFORMAÇÃO

LUIZ FELIPE ROSA DOS ANJOS

EVOLUÇÃO DO JAVASCRIPT EM APLICAÇÕES MULTIPLATAFORMA:

Como projetos podem se beneficiar dos *frameworks* e bibliotecas disponíveis.

Florianópolis – SC
2017

INSTITUTO FEDERAL DE SANTA CATARINA - IFSC
CAMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE SAÚDE E SERVIÇOS
CST EM GESTÃO DA TECNOLOGIA DA INFORMAÇÃO

LUIZ FELIPE ROSA DOS ANJOS

EVOLUÇÃO DO JAVASCRIPT EM APLICAÇÕES MULTIPLATAFORMAS:

Como projetos podem se beneficiar dos *frameworks* e bibliotecas disponíveis.

Trabalho de Conclusão de Curso apresentado ao Curso de Tecnologia em Gestão da Tecnologia da Informação do Instituto Federal de Santa Catarina como requisito à obtenção do grau de Tecnólogo em Gestão da Tecnologia da Informação

Orientador: Prof. Gilmar Carvalho

FLORIANÓPOLIS, NOVEMBRO DE 2017

Ficha de identificação da obra elaborada pelo autor.

DOS ANJOS, LUIZ FELIPE ROSA
EVOLUÇÃO DO JAVASCRIPT EM APLICAÇÕES MULTIPLATAFORMA
: como projetos podem se beneficiar dos frameworks e biblio-
tecas disponíveis / LUIZ FELIPE ROSA DOS ANJOS
; orientação de GILMAR CARVALHO. - Florianópolis,
SC, 2017.
86 p.
Trabalho de Conclusão de Curso (TCC) - Instituto Federal
de Santa Catarina, Câmpus Florianópolis. CST
em Gestão da Tecnologia da Informação. Departamento
Acadêmico de Saúde e Serviços.
Inclui Referências.

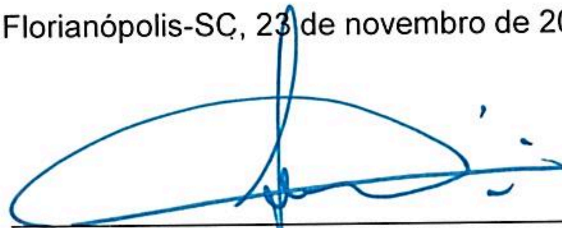
1. JAVASCRIPT. 2. REACT. 3. ANGULARJS. 4. FRAMEWORK.
5. BIBLIOTECA. I. CARVALHO, GILMAR. II. Instituto
Federal de Santa Catarina. Departamento Acadêmico de Saúde
e Serviços. III. Título.

TÍTULO DO TRABALHO: EVOLUÇÃO DO JAVASCRIPT EM APLICAÇÕES MULTIPLATAFORMAS

LUIZ FELIPE ROSA DOS ANJOS

Este trabalho foi julgado adequado para obtenção do Título de Tecnólogo em Gestão da Tecnologia da Informação e aprovado na sua forma final pela banca examinadora do Curso Superior de Tecnologia em Gestão da Tecnologia da Informação do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

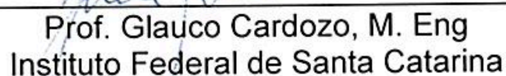
Florianópolis-SC, 23 de novembro de 2017.



Prof. Felipe Cantório Soares
Coordenador do CST em Gestão da Tecnologia da Informação
Instituto Federal de Santa Catarina

Banca Examinadora:

Prof. Gilmar Carvalho, M. Eng
Orientador
Instituto Federal de Santa Catarina



Prof. Glauco Cardozo, M. Eng
Instituto Federal de Santa Catarina

Herval Daminelli
Instituto Federal de Santa Catarina

RESUMO

No presente trabalho, realizou-se uma análise da evolução do JavaScript, abrangendo desde o seu propósito original de adicionar dinamismo às páginas da internet até os mais diversos usos possibilitados pelo advento do Node.js. A linguagem que tinha seu uso restrito aos navegadores hoje desempenha papéis em ferramentas diversas, como automatizador de tarefas e utilitários do sistema operacional. Sua popularização levou à criação de um leque muito amplo de *frameworks* e bibliotecas, e com o propósito de informar ao leitor quais dessas ferramentas se destacam, realizou-se uma pesquisa bibliográfica e experiencial - vivida pelo autor no desenvolvimento de suas atividades diárias -, dando enfoque a duas ferramentas muito utilizadas no *front-end*, nomeadamente AngularJS e React. Como resultado desta pesquisa, verificou-se que cada uma dessas ferramentas possui suas vantagens, sendo o React uma melhor escolha quando é necessário criar muitos componentes e ter uma renderização mais rápida, e o AngularJS mais indicado quando a aplicação visa escalabilidade e requer ser desenvolvida em um curto espaço de tempo. Além dessa comparação, a pesquisa revelou que a web caminha para uma crescente adoção de uma nova linguagem, chamada de WebAssembly, que visa ser uma linguagem de baixo nível e mais perto das instruções de máquina, o que trará performance para as aplicações que rodam nos navegadores.

Palavras-chave: JavaScript, React, AngularJs, WebAssembly, Biblioteca, *Framework*

ABSTRACT

This study performed an analysis of the evolution of JavaScript, comprising its original purpose of adding dynamism to web pages up until its recent diverse uses made possible by NodeJS. The language which was confined inside web browsers, today plays all sorts of roles that range from task automators to system utilities. Since its boom, a plethora of frameworks and libraries have been created, which makes providing the reader with a guide of what tools deserve their attention the driving force of this study. To do so, a bibliographic and experiential research took place, with a slight focus on two of the most used front-end tools, namely AngularJS and React. As result of this research, it became clear that each of these tools have its set of upsides when comparing to its counterpart, being React more appropriate when there is a need to create more granular components and to render the page faster, and AngularJS being the best option when there is less time to develop the application. Furthermore, research also indicates that the web faces an increasing adoption of WebAssembly, a language that aims to be a low level, closer to the metal alternative to JavaScript, which should improve the performance of applications that run in a web browser.

Keywords: JavaScript, React, AngularJs, WebAssembly, Library, Framework

LISTA DE FIGURAS

Figura 1 - Tecnologias utilizadas em cada área do desenvolvimento web 19
Figura 2 - Funcionamento de uma promise 34
Figura 9 - Classe por protótipos em ES5 37
Figura 5 - Os conjuntos da linguagem JavaScript da perspectiva do TypeScript 45
Figura 5 - Tecnologia mais amada segundo a pesquisa do site Stackoverflow........ 49
Figura 6 - Tecnologia mais temida segundo a pesquisa do site Stackoverflow 50
Figura 7 - Tecnologia que os desenvolvedores mais gostariam de usar segundo a pesquisa do site Stackoverflow 51
Figura 8 - Two-way data binding 57
Figura 9 - Modelo tradicional de interação entre aplicação e DOM 59
Figura 10 - Interação entre aplicação e DOM utilizando React................................ 59
Figura 11 - Fluxo de eventos até que a view seja atualizada 60
Figura 12 - Estrutura do DOM 61
Figura 13 - Interação entre os componentes do Flux.. 64
Figura 14 - Desempenho de cada componente no navegador Chrome 70
Figura 15 - Desempenho de cada componente no navegador Firefox 71
Figura 16 - Desempenho de cada componente no navegador Safari...................... 71

LISTA DE CÓDIGOS

Código 1 - Exemplo de objeto JSON 23
Código 2 - Função com AJAX 30
Código 3 - Função com atraso 31
Código 4 - Função Callback 32
Código 5 - Multiplos Callbacks indentados 33
Código 6 - Exemplo de Promise 35
Código 7 - Class declaration em ES6 37
Código 8 - Class Expression em ES6 38
Código 9 - Exemplo de declaração de variáveis usando let e var 39
Código 10 - Declaração de constantes 39
Código 11 - Função convencional e arrow function 40
Código 12 - Referência ao valor de this em funções seta 41
Código 13 - Exemplo de Strings 42
Código 14 - Exemplo com o operador Spread 43
Código 15 - Usos comuns do Destructuring 43
Código 16 - Código escrito em JSX e o resultado de sua compilação para JavaScript 47

LISTA DE SIGLAS E ABREVIATURAS

GTI - Gestão da Tecnologia da Informação
HTML - Hypertext Markup Language
HTTP - Hypertext Transfer Protocol
AJAX - Asynchronous JavaScript And XML
REST - Representational State Transfer
DOM - Document Object Model
CSS - Cascading Style Sheets
URI - Uniform Resource Identifier
URL - Uniform Resource Locator
API - Application Programming Interface
JS - JavaScript
JSON - JavaScript Object Notation
FTP - File Transfer Protocol
XML - Extensible Markup Language
SOAP - Simple Object Access Protocol
IDE - Integrated Development Environment
UC - Unidade Curricular

SUMÁRIO

1. INTRODUÇÃO **11**
1.1. PROBLEMA **13**
1.2. JUSTIFICATIVA **14**
1.3. OBJETIVOS **15**
1.3.1. Objetivo Geral 15
1.3.2. Objetivos Específicos 15
1.4. METODOLOGIA **15**
2. TECNOLOGIAS WEB **17**
2.1. FRONT-END **18**
2.2. HTML E CSS **19**
2.3. JAVASCRIPT **20**
2.4. BACK-END **21**
2.5 FORMATOS DE TRANSFERÊNCIA DE DADOS **21**
2.5.1. XML 22
2.5.2. JSON 22
2.6.1 Biblioteca 24
2.6.2 Framework 25
3. HISTÓRIA, MOMENTO ATUAL E NATUREZA DO JAVASCRIPT **27**
3.1 LANÇAMENTO **27**
3.2 CÓDIGO ASSÍNCRONO **28**
3.2.1 Promises e Callbacks 31
3.2.1.1 Callbacks 32
3.2.1.2 Promises 33
3.3. ES6 E ES5 ..... **35**
3.3.1 Classes 36

3.3.1.1 Definindo classes 36
3.3.1.1.1 Class declarations 37
3.3.1.1.2 Class expressions 38
3.3.2 Let 38
3.3.3 Const 39
3.3.4 Funções seta (*Arrow Functions*) 40
3.3.5 String 41
3.3.6 Spread *Operator* 42
3.3.7 Destructuring 43
3.4.1 TypeScript 44
3.4.2 JSX 46
4. JAVASCRIPT ONIPRESENTE 48
4.1 A POPULARIZAÇÃO DO JAVASCRIPT 48
4.2 NODEJS 52
4.3 PRINCIPAIS *FRAMEWORK*S E BIBLIOTECAS FRONT-END DIVIDIDOS EM GERAÇÕES 53
4.3.1 Primeira Geração 54
4.3.2 Segunda Geração 54
4.3.3 Geração Atual 55
4.4 CRIAÇÃO DO ANGULAR 55
4.5 CRIAÇÃO DO *REACT* 58
4.5.1 DOM e V-DOM 60
4.5.2 A Arquitetura Flux 62
5. COMPARATIVO ENTRE ANGULARJS E *REACT* NOS QUESITOS COMPONENTIZAÇÃO, DESEMPENHO, TRATAMENTO DE EVENTOS E *TEMPLATING*. 65
5.1 COMPONENTIZAÇÃO 66
5.2 TRATAMENTO DE EVENTOS (*EVENT HANDLING)* 67

5.3 TEMPLATING 68
5.4 DESEMPENHO... 69
6. PARA ONDE CAMINHA A WEB 73
7. CONCLUSÕES 75
7.1 RECOMENDAÇÕES PARA TRABALHOS FUTUROS 76
REFERÊNCIAS 78

1. INTRODUÇÃO

A web tem sido foco de interesse e pesquisa, em especial desde a última década, por conta do seu futuro imprevisível. Diversas ferramentas que antes necessitavam estar instaladas na máquina do usuário, hoje estão disponíveis majoritariamente apenas em versão web. Grande parte deste processo se deu graças a evolução dos browsers e da linguagem dominante da web, o JavaScript.

Apesar da fundação do desenvolvimento web ter se mantido a mesma há bastante tempo, com HTML (*Hypertext Markup Language*), CSS (*Cascading Style Sheet*) e JavaScript (JS), são atribuídas ao cliente (browser) cada vez mais responsabilidades. Inicialmente a ideia era apenas adicionar interatividade e tornar as páginas mais atraentes aos usuários, mas pouco a pouco adicionando segurança e aumento de performance, invertendo os pesos do modelo de desenvolvimento, que se baseava em “cliente magro”, ou seja, tendo a maior parte do processamento realizada pelo servidor, e tornando padrão o modelo “cliente gordo”, que realiza a maior parte das tarefas nos computadores dos usuários. Essas mudanças ocasionaram um crescimento exponencial no número de ferramentas criadas para auxiliar no desenvolvimento, tornando a escolha uma tarefa árdua, uma vez que as tendências mudam tão rápido quanto novas ferramentas aparecem.

> A noção de que “a tecnologia movimenta-se rapidamente” é um aforismo comumente utilizado, e por um bom motivo: a tecnologia de fato evolui rapidamente. (...)
> Inovação ocorre num passo quase febril dentro da comunidade JavaScript, que embora seja imensamente fascinante, traz também alguns desafios intrínsecos. O ecossistema do JavaScript de bibliotecas, *frameworks* e utilitários cresceu dramaticamente. (...). Como resultado, desenvolvedores encontram-se diante de uma tarefa cada vez mais difícil, a de

> escolher as ferramentas apropriadas de tantas, aparentemente, boas opções.
> (JavaScript Frameworks for Modern Web Dev, AMBLER; CLOUD, 2015, p.25, tradução nossa)

Por conta desse crescente número de ferramentas de desenvolvimento para JavaScript, este estudo visa demonstrar, apoiando-se nos conhecimentos adquiridos nas UCs relacionadas à programação, quais ferramentas se destacam e merecem atenção dos desenvolvedores que estão procurando se atualizar ou buscando outro caminho profissional. Baseando-se em pesquisa bibliográfica e experiencial vivida pelo autor, o trabalho faz uma breve narrativa acerca da história do JavaScript, analisando as versões da linguagem e elucidando as principais mudanças introduzidas desde o lançamento de sua última versão oficial. Além disso intenta apresentar um comparativo entre *React* e AngularJS, duas das ferramentas mais utilizadas para o desenvolvimento *front-end* com JavaScript atualmente.

> Criado em 1995 por Brendan Eich do Netscape, JavaScript foi de ‘linguagem de brinquedo’ usada apenas em navegadores, para uma das linguagens mais populares do mundo para desenvolvimento *full-stack*. Com o surgimento do Node.js, construído em cima da engine JavaScript V8 do Chrome, desenvolvedores agora são capazes de construir aplicações performantes e poderosas usando JavaScript. Com a adição de MongoDB, um banco de dados NoSQL, as aplicações podem utilizar JavaScript em todos os níveis. (Mastering JavaScript Single Page Application Development, KLAUZINSKI; MOORE, 2016, p. 1, tradução nossa)

O surgimento diário de novas alternativas, exige dos profissionais que se aprofundem no conhecimento desses recursos, posto que JS continuará sendo, por

ora, a linguagem predominante da web.

1.1. PROBLEMA

JavaScript em sua implementação original não previa tantas responsabilidades como hoje lhe são atribuídas. O propósito inicial da linguagem era apenas manipulação simples do DOM (*Document Object Model*), como animações e outras formas de automação. O cenário atual de desenvolvimento web, não limitado à, mas com ênfase em *front-end* nos mostra que há muito essa não é mais a única serventia da linguagem, visto o surgimento do AJAX, serviços RESTful[1] e a adoção em massa de transferência de dados via JSON (JavaScript *Object Notation*).

*Framework*s e bibliotecas como JQuery, React, AngularJS entraram em cena não só para facilitar o trabalho do desenvolvedor, mas também provendo funções prontas que auxiliam nessas tarefas e na padronização do estilo de desenvolvimento. Desenvolver utilizando JavaScript sem o auxílio de algum destes *frameworks* prova ser uma tarefa difícil quando o projeto visa escalabilidade, e alguns destes *frameworks* possibilitam (com ou sem o auxílio de outras bibliotecas) que se adote diferentes design *patterns*[2] na aplicação.

Para o profissional que trabalha com desenvolvimento web, obter um contexto comparativo dessas tecnologias e suas tendências é fundamental para determinar onde deve investir seus estudos.

[1] Serviços RESTful são serviços web que seguem a convenção REST de padronização de URLs e verbos HTTP e trafegam dados através de JSON.

[2] De acordo com o livro Design Patterns, Elements of Reusable Object-Oriented Software (Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm, pag 16, 1994), design *patterns* são descrições de objetos e classes que se comunicam e são customizados para resolver um problema de design geral em um determinado contexto.

1.2. JUSTIFICATIVA

Hoje JavaScript é a linguagem que permite o AJAX, manipulação de DOM, tratamento de eventos do browser e interação com o usuário. Sem ele as páginas seriam estáticas e monótonas, como nos estágios iniciais da web, trazendo apenas documentos e imagens, tendo pouca ou nenhuma interatividade. Assim sendo, essa linguagem vem sofrendo alterações em função de melhores práticas de desenvolvimento e aprimoramento de suas características fundamentais.

> Você pode facilmente ver evidências da popularidade do JavaScript quando olha para o GitHub. JavaScript é a linguagem número 1 quando se trata do número de repositórios. Sua proeminência também fica evidente no Livecoding.tv, onde membros criam mais vídeos em JavaScript do que em qualquer outro tópico. No momento desta escrita, o site hospeda 45.919 vídeos sobre JavaScript. (GARBADE, 2016, tradução nossa)

Por conta da infinidade de bibliotecas e *framework*s disponíveis, nem sempre é evidente que caminho tomar ao iniciar os estudos em novas tecnologias.

> A popularidade do JavaScript levou a um ecossistema vibrante de tecnologias, *framework*s e bibliotecas. Junto com toda essa diversidade e energia incríveis neste ecossistema, vem também um alto grau de confusão. Em que tecnologias você deve prestar atenção? Onde você deve investir seu tempo para obter o melhor retorno? Quais *tech stacks* as empresas estão buscando agora? Quais tem o melhor potencial para crescimento? (ELLIOT, 2016, tradução nossa)

A fim de auxiliar desenvolvedores de outras linguagens e interessados no

assunto, este estudo tenta esclarecer algumas dúvidas a respeito de como essas ferramentas se comparam entre si, que problemas elas visam resolver e de que forma abordam estes problemas, baseando-se no que há de mais recente em termos de publicações no assunto bem como em opiniões e indicações de desenvolvedores conceituados na comunidade JS.

1.3. OBJETIVOS

1.3.1. Objetivo Geral

- Analisar a evolução da linguagem JavaScript e suas principais ferramentas tendo por base os *framework*s AngularJs e React.

1.3.2. Objetivos Específicos

- Ampliar o conhecimento relativo às unidades do curso de GTI na área de engenharia de *software*;
- Mapear a evolução dos principais *framework*s JavaScript utilizados atualmente;
- Apontar tendências das aplicações multiplataforma;
- Estabelecer parâmetros que possam caracterizar a evolução dos *framework*s estudados.

1.4. METODOLOGIA

O presente trabalho tem como método de desenvolvimento as seguintes etapas:

1. Pesquisa bibliográfica baseada em literatura técnica atualizada acerca de ferramentas e processos de desenvolvimento utilizando JavaScript, bem como documentações oficiais disponibilizadas nos *sites* dos desenvolvedores.
2. Estabelecer uma base contextual a respeito das tecnologias utilizadas na *web.*
3. Apresentar as mudanças introduzidas na linguagem desde seu lançamento.
4. Explicitar os motivos que levaram ao crescimento na popularidade e utilização do JavaScript tanto dentro como fora dos navegadores.
5. Realização de um comparativo entre as tecnologias dominantes no ecossistema da linguagem.
6. Apontar as tendências para o futuro da *web* baseado nas tecnologias mais recentes.

2. TECNOLOGIAS WEB

Atualmente, a maioria das aplicações estão disponíveis no que chamamos de web. A web é uma miríade de recursos tecnológicos e serviços e foi originalmente criada para por Tim Berners-Lee, um cientista britânico, que viu no crescimento da internet uma possível solução para o problema de compartilhamento de informações enfrentado no laboratório de física de partículas em Genebra, na Suíça, onde cientistas do mundo todo realizavam experimentos, mas não dispunham de uma maneira simples de compartilhar suas descobertas.

> Naqueles dias, diferentes informações estavam em diferentes computadores, e você tinha que *logar* em diferentes computadores para ter acesso à elas. Além disso, muitas vezes você tinha que aprender um programa diferente em cada computador. Quase sempre era mais fácil simplesmente ir até a pessoa e perguntar o que se queria saber quando elas estavam tomando café... (BERNERS-LEE, tradução nossa)

Berners-Lee viria a ser conhecido como o "pai" da web justamente por ter desenvolvido muitos dos padrões e tecnologias usadas na web até hoje, como o HTML, as URI (*Uniform Resource Identifier*) usadas para especificar os endereços/caminhos dos recursos na internet e o protocolo HTTP (*Hypertext Transfer Protocol*).

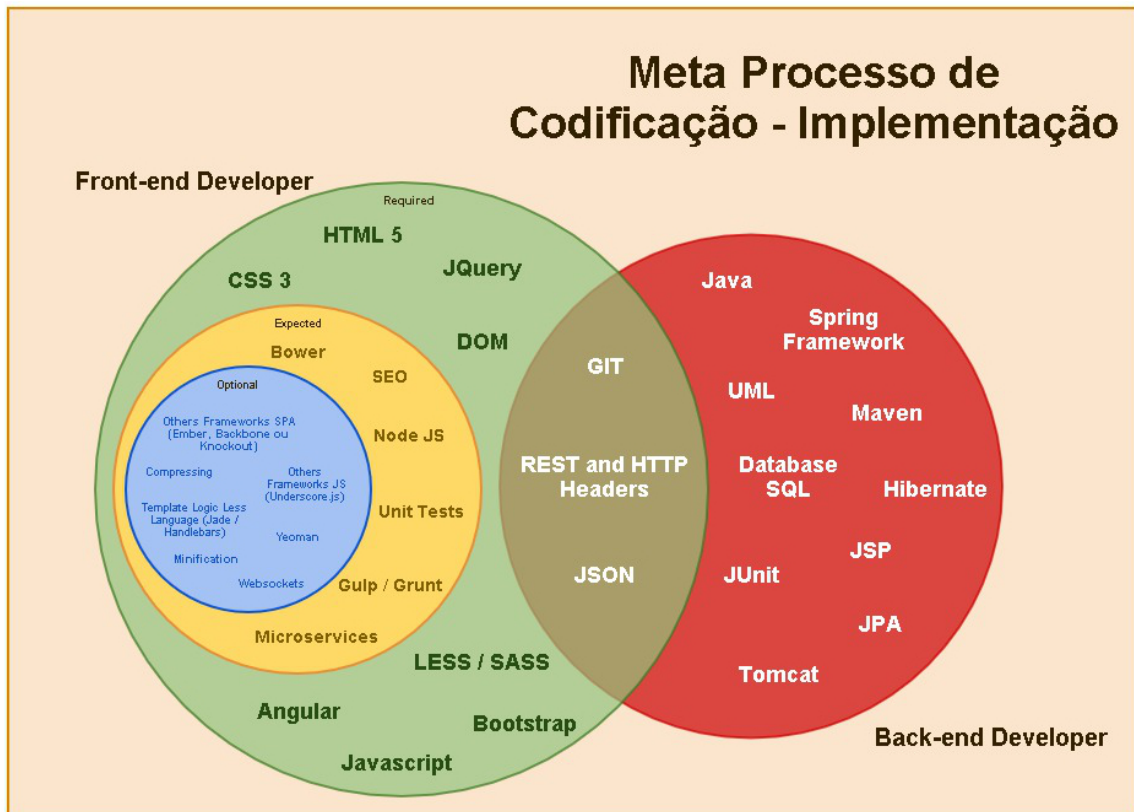
Neste capítulo será abordado, de forma sucinta, para fins elucidativos, as duas camadas essenciais do desenvolvimento de aplicações web. O *front-end*, que é a camada de apresentação e engloba HTML, CSS e JavaScript, e o *back-end*, que é a camada de acesso a dados, a qual possui um número bem maior de tecnologias envolvidas.

2.1. FRONT-END

O *front-end*, se encarrega de tudo o que tange o usuário/cliente de uma aplicação, o que inclui tudo o que engloba a interface de usuário. O desenvolvimento *front-end* na web é composto basicamente por: HTML, CSS e JS. Existem também as tecnologias comuns aos dois *ends* de uma aplicação, como Git, entendimento dos protocolos da web (HTTP, FTP, etc) e formatos de transferência de dados como XML e JSON.

O desafio do desenvolvedor *front-end* é fazer a comunicação do usuário com o *back-end* de uma maneira intuitiva, pensando em usabilidade e acessibilidade e garantindo a preservação da integridade dos dados. A figura a seguir apresenta algumas tecnologias que são facilmente encontradas no cotidiano de um desenvolvedor web, e também conceitos como SEO (search engine optimization) que consiste de tornar a pagina melhor indexada por mecanismos de busca, como o Google. É raro encontrar projetos grandes que não façam uso de alguma ferramenta de *build* e automatização de tarefas como o Gulp ou Grunt, pois é bastante comum que se utilize algum pré-processador de CSS (LESS ou SASS) que requer que arquivos de estilo sejam constantemente compilados.

Figura 1 - Tecnologias utilizadas em cada área do desenvolvimento web



Fonte: REIS (2016)

2.2. HTML E CSS

O HTML é uma linguagem de marcação que surgiu a partir de uma especificação criada por Tim Berners-Lee em meados de 1990. Originalmente consistia de cerca de 18 elementos e tinha como objetivo a estruturação das páginas da internet de forma semântica.

HTML é considerado o *backbone*, ou seja, a espinha dorsal de qualquer aplicação web, e é responsável por interpretar texto, renderizar imagens e diversos outros recursos, inclusive áudio e vídeo.

Quer o desenvolvedor escreva HTML ou não -é possível escrever

componentes em JavaScript, por exemplo, através de *strings* ou *JSX* - é essa linguagem que o navegador entende quando se trata de construir o leiaute de uma página.

A versão atual do HTML está em uso desde 2013 e sua sintaxe consiste basicamente de elementos escritos dentro de parênteses angulares - chevrons - (<>), que são chamados de *tags* e há uma tendência crescente de tornar esses elementos cada vez mais semânticos para que haja uma maior inclusão de deficientes visuais na internet. Da mesma forma que o navegador entende HTML para renderizar os elementos visuais, ele entende CSS para estilizar estes elementos. Todas as cores, formas e efeitos visuais ou de transição nas páginas são tarefas do CSS.

2.3. JAVASCRIPT

Por último, e de igual ou maior importância temos o JavaScript, que é a linguagem de script interpretada pelos navegadores e encarregado atualmente de uma ampla gama de responsabilidades, que vão desde requisições assíncronas para obtenção de dados a alterações dinâmicas de CSS. O JS é uma linguagem de alto nível, que há alguns anos tinha seu uso restrito aos navegadores, mas graças ao advento do Node.JS tem ganhado importância dentro e fora da web, sendo utilizada para vários fins como criação de micro serviços e ferramentas para automação de tarefas (Grunt, Gulp, Webpack, etc.), e até mesmo IDEs (ambientes integrados de desenvolvimento), como o Visual Studio Code da Microsoft, fato que impulsiona o crescimento da atualmente enorme comunidade, que é na sua maioria *open-source*, ou seja, de código aberto e licenciamento livre.
Segundo David Flanagan, autor do livro JavaScript The Definitive Guide:

> Não há definição de um programa em JavaScript no lado do cliente. Nós

> podemos dizer que um programa JavaScript consiste de todo o código JavaScript em uma página web (scripts inline, handlers de eventos HTML, e URLs javascript:) além de código JavaScript externo referenciado pelo atributo src de uma tag <script> (2011, tradução nossa)

Por esse motivo hoje contamos com tantas ferramentas que se propõem a organizar projetos JavaScript.

2.4. BACK-END

O *back-end* de uma aplicação ou sistema refere-se à parte do programa que o usuário não vê, mas que é vital para o funcionamento. Consiste geralmente de:

- Servidor
- Banco de dados
- Aplicação propriamente dita

A aplicação pode ser escrita em muitas linguagens de programação, e é no *back-end* onde encontramos as regras de negócio. Um processo comum que acontece no desenvolvimento de uma aplicação web é que os desenvolvedores do *back-end* disponibilizem endpoints[3] para que o *front-end* possa consumir os dados de um banco de dados e apresentá-los de maneira amigável ao usuário final.

2.5 FORMATOS DE TRANSFERÊNCIA DE DADOS

O *front-end* e *back-end* necessitam de uma maneira de compartilhar dados entre si. Para este fim temos duas opções de formatos de arquivo que são mais comumente utilizadas:

- XML
- JSON

[3] *endpoint* é a URL onde um serviço web pode ser acessado por uma aplicação cliente.

2.5.1. XML

De acordo com o World Wide Web Consortium (W3C), órgão que padroniza a web, XML significa *Extensible Markup Language* e é um formato simples, baseado em texto, de representação de informações estruturadas, como: documentos, configurações, livros, transações, entre outros. A sintaxe é muito similar à do HTML, tendo o mesmo princípio de *tags* e aninhamento.

As principais diferenças em relação ao HTML são que:

- Arquivos XML não são processados caso contenham erros, em vez disso, lançam um alerta de erro para que o autor corrija.
- Todos os elementos precisam estar devidamente fechados ou marcados como vazios. Isso quer dizer que uma *tag* XML sem conteúdo deve aparecer da seguinte forma:

<autor> </autor>

ou, de maneira simplificada:

<autor/>

Valores de atributos em XML também devem ser sempre cercados por aspas:

<animal tipo=”quadrupede” />

XML é utilizado como formato de transferência de dados por serviços web, através de um protocolo conhecido como *Simple Object Access Protocol* (SOAP). Atualmente a maioria dos serviços oferecem tráfego de dados por meio de JSON, já que é um padrão nativo à linguagem JS e também reduz bastante o tamanho dos arquivos, pois possui menos redundância que XML.

2.5.2. JSON

Apesar de o nome significar JavaScript Object Notation, JSON é um padrão que não depende de nenhuma linguagem, e só recebe este nome pois segue o mesmo estilo de criação de objetos JavaScript (chave e valor), com a diferença de que em JSON a chave é sempre cercada por aspas e o valor pode ou não ser

cercado por aspas, enquanto que nos objetos JS apenas o valor pode apresentar aspas.

Código 1 - Exemplo de objeto JSON

```json
{
  "nome": "Pedro",
  "sobrenome": "Silva",
  "idade": 25,
  "endereco": {
        "rua": "Rua Santos Saraiva",
        "cidade": "Florianopolis",
        "estado": "SC",
        "cep": "88070-000"
  }
}
```

Fonte: autor.

2.6 DISTINÇÃO ENTRE *FRAMEWORKS* E BIBLIOTECAS

A comparação entre AngularJs e React possui um nível elevado de complexidade, pois não se tratam apenas de duas ferramentas com propósitos razoavelmente distintos (o que não quer dizer que não possam ser intercambiáveis), mas também são duas ferramentas de categorias diferentes.

React é uma biblioteca originalmente concebida pela equipe do Instagram (adquirido posteriormente pelo Facebook) para ser uma ferramenta que lida apenas o V (*View*) de uma aplicação MVC, ou seja, não traz componentes para *Model* e *Controller*. Por si só, React não fornece nenhuma diretriz de como se deve organizar o projeto, assim como não tem diversas funcionalidades, como serviços prontos para requisições HTTP, que encontramos no Angular, que é um *framework* MVC completo.

Com o objetivo de levantar questões acerca do comparativo proposto e aprofundar os conhecimentos relativos às ferramentas o capítulo apresentará uma breve definição dos termos Biblioteca e *Framewor*ks.

2.6.1 Biblioteca

Por biblioteca entende-se uma coleção de objetos/funções/métodos (dependendo da linguagem). Segundo Zanette:

> É uma coleção de implementações de comportamentos escritos em uma linguagem e importadas no seu código. Nesse caso, há uma interface bem definida para cada comportamento invocado. Um bom exemplo é a biblioteca jQuery que implementa certos comportamentos, como por exemplo, a manipulação do HTML.

É um arquivo que contém código reutilizável, que pode ser compartilhado por diversas aplicações e não traz nenhuma determinação de como o usuário deve estruturar suas pastas ou nomear seus arquivos. Outra característica comum às bibliotecas é a impossibilidade de modificação para que o usuário “personalize” o código, seja adicionando funcionalidades ou mesmo alterando os retornos das funções.

Esse código é acessível através de uma API[4] (Interface de programação de aplicações), na qual o usuário fornece um input para determinada função da biblioteca, essa função executa seus procedimentos e devolve o controle ao usuário. Isso pode ser interessante pois não expõe variáveis temporárias no código principal.

2.6.2 Framework

Framework, que em português pode ser traduzido como “estrutura”, possui um conjunto de regras a serem seguidas. Diferentemente das bibliotecas, os *Frameworks*, como o nome sugere, geralmente ditam a arquitetura do projeto, o que ocorre por conta da inversão de controle. Zanette define como: " estrutura real, ou conceitual, que visa servir como suporte (ou guia) para a construção de algo (um produto, por exemplo)."

Caso o desenvolvedor opte por utilizar AngularJS em seu projeto, o *framework* se encarrega de disponibilizar as dependências declaradas para cada módulo, e a arquitetura deverá se moldar ao formato esperado pela ferramenta, cabendo ao desenvolvedor estruturar suas pastas e arquivos de acordo com o paradigma MVC[5] para que seus *scripts* sejam carregados de forma correta e o risco de inconsistências seja mitigado.

No caso do *React*, que não é um *framework*, mas uma biblioteca, para lidar com a “liberdade” de não estar amarrado a uma estrutura específica, o que pode ser

[4] No contexto de uma biblioteca, API é um conjunto de métodos públicos utilizados para interagir com outros objetos da aplicação principal.

[5] De acordo com o livro Design Patterns, Elements of Reusable Object-Oriented Software (GAMMA, Erich et al, pag 16, 1994), *design patterns* são descrições de objetos e classes que se comunicam e são customizados para resolver um problema de design geral em um determinado contexto.

prejudicial caso não se tenha uma ideia clara do tamanho final do projeto, o desenvolvedor conta com algumas opções de arquitetura, sendo uma das mais comuns o Flux, o qual impõe um fluxo de dados unidirecional na aplicação, auxiliando assim no processo de construção da estrutura de pastas e arquivos.

Bibliotecas e *Framework*s são escritos em JavaScript puro, também chamado de *Vanilla* JS, e têm como objetivo facilitar a vida do desenvolvedor, poupando o trabalho de pensar na estruturação e provendo código limpo, legível e de manutenibilidade facilitada. Algumas das bibliotecas mais comumente encontradas em diversos projetos são projetadas para lidar com problemas específicos, como é o caso do *lodash*, criado para lidar com iterações, modificações e avaliações de arrays e objetos, e também o *momentJS* que é extremamente difundido pois trata de uma das maiores dificuldades encontradas no JS: trabalhar com datas.

As principais vantagens de se utilizar um FW ou biblioteca são:

- Reduz o tempo e custo de desenvolvimento, uma vez que eles lidam com a parte burocrática de arquitetura, no caso dos *Framework*s;
- Torna mais fácil a entrada de novos desenvolvedores no projeto, já que as nomenclaturas são padronizadas;
- Diminui a discrepância de tempo no desenvolvimento do Front-end e Back-end.

Logicamente há o contratempo de se estar preso às limitações do *framework*/biblioteca, podendo o projeto ter algum quesito de avaliação prejudicado, seja performance, segurança ou funcionalidade.

3. HISTÓRIA, MOMENTO ATUAL E NATUREZA DO JAVASCRIPT

3.1 LANÇAMENTO

Em Dezembro de 1995 JavaScript foi anunciado pela Netscape em conjunto com a Sun Microsystems:

> Netscape e Sun anunciam JavaScript, a linguagem de script de objetos aberta e multi-plataforma para redes corporativas e a internet, 28 empresas líderes da indústria aprovam JavaScript como um complemento para Java para fácil desenvolvimento de aplicações online. (1995, tradução nossa)

A razão que levaria Marc Andreessen, fundador da *Netscape*, a pensar no desenvolvimento de uma linguagem para o browser era que a web precisava de uma "linguagem cola", que fosse fácil de usar por web designers e entusiastas da programação (amadores) para criar componentes e cujo código pudesse ser escrito diretamente na marcação da página. Isso levou Andreessen a recrutar Brendan Eich, que tinha inicialmente tinha como objetivo embutir a linguagem *Scheme* no *Netscape Navigator*. Antes que Eich pudesse começar, no entanto, houve uma colaboração da Netscape com a *Sun Microsystems* para que pudessem competir com a Microsoft no cenário do desenvolvimento de tecnologias web, e assim foi decidido que a linguagem usada no Netscape Navigator deveria ter a sintaxe similar ao Java da Sun, o que excluiria linguagens como Python, Perl e Scheme.

Para que a ideia de tal linguagem fosse escolhida frente às outras propostas, a empresa precisava de um protótipo, o qual Eich escreveu em dez dias. O nome JavaScript foi uma jogada de marketing para que desenvolvedores associassem JS ao Java, que na época era a linguagem do momento.

A década de 90 foi marcada pela "Guerra dos *Browser*s", sendo caracterizada pelos famosos sites "*best-viewed in Netscape*" (melhor visualizado no Netscape) ou *"best-viewed in* Internet Explorer" (melhor visualizado no Internet Explorer). Isto

ocorrera por causa das diferentes implementações da linguagem nos browsers. O Internet Explorer 3, por exemplo utilizava JScript, uma versão construída a base de engenharia reversa do JavaScript, já o Netscape utilizava a implementação original do JS o que fazia com que se tornasse difícil criar sites que renderizassem da mesma forma em todos os navegadores.

Como o nome JavaScript é uma marca registrada pela Sun Mircrosystems (hoje Oracle), em 1996 foi criada pela ECMA *international* (European Computer Manufacturers Association) uma versão padronizada do JavaScript, baseada no trabalho desenvolvido pela Netscape, chamada de ECMAScript. Desde então foram lançadas diversas atualizações do ECMAScript, sendo a segunda versão em 1998, a terceira em 1999, e a quinta em 2009. Uma quarta versão acabou sendo deixada de lado por fatores "políticos", o que atrasou bastante a evolução da linguagem.

Em Junho de 2011, foi lançado oficialmente o ECMAScript 5.1, versão que até o momento é a mais utilizada pelos browsers, apesar da crescente adoção do ES6 (ECMAScript 2015).

3.2 CÓDIGO ASSÍNCRONO

O pensamento na época do surgimento do JavaScript era de que a linguagem era voltada apenas para amadores, web-designers e entusiastas. O advento do AJAX, no entanto, mudou a opinião de muitos desenvolvedores, uma vez que houve repentinamente uma enorme demanda por profissionais que soubessem tirar proveito dessa especificação.

JavaScript é por natureza uma linguagem assíncrona, e é exatamente o que significa AJAX: *Asynchronous JavaScript and XML*. De uma maneira simplificada, a partir do AJAX não é mais necessário submeter o formulário todo e recarregar a página inteira para validar um único campo. Isso representa uma melhora na experiência do usuário, além de ser muito mais econômico em termos de recursos.

O que acontece, é que ao preencher um campo de um formulário, é possível

que o cliente (navegador) faça uma requisição assíncrona - não bloqueando a interface de usuário - ao *backend* e verifique a validade deste campo em específico. Por exemplo, ao se cadastrar em um serviço de e-mail, existe a possibilidade de que o endereço escolhido pelo usuário já esteja em uso, porém, ao invés de preencher todos os dados, submeter o formulário, deixar o servidor interpretar todos os dados para retornar uma resposta de erro, com AJAX apenas os campos que precisam de validação podem ser verificados enquanto o usuário os preenchem, sem que o formulário seja submetido. Caso seja um endereço válido o botão de enviar o formulário fica habilitado e então o processo é concluído sem chance de erros. O exemplo descrito é apenas um caso de uso básico, mas graças ao AJAX temos diversos serviços web disponíveis que vão desde consulta de endereço/CEP dos Correios, a mapas e acompanhamento de trânsito em tempo real.

Os dados trafegados via AJAX podem ser tanto XML ou JSON, dependendo apenas da implementação da API que faz as consultas. Atualmente boa parte dos *endpoints* ptam por utilizar JSON por ser um formato nativo à linguagem JavaScript. Para exemplificar uma requisição AJAX temos o seguinte código, que faz uso de Jquery:

Código 2 - Função com AJAX

```
$('#buscar').click(function(){
    $.ajax({
    url:'http://www.devmedia.com.br/api/cep/service/',
        type:'get',
        dataType:'json',
        crossDomain: true,
        data:{
            cep: $('#cep').val(),
      //pega valor do campo
            formato:'json',
            chave: 'SUA CHAVE'
        },
        success: function(res){
            $('#resultado')
            .append('<strong>RUA</strong>'+res.logradouro)
            .append('<strong>BAIRRO</strong>'+res.bairro)
            .append('<strong>CIDADE</strong>'+res.cidade+ ' -
'+res.uf);
        }
    })
});
```

Fonte: Autor

Este código faz uma requisição a uma API de consulta de CEP, e ao receber a resposta do servidor, de forma assíncrona, imprime e formata no elemento HTML de *id* “resultado” os dados da resposta.

Mas não apenas de AJAX consiste a natureza assíncrona do JavaScript. É muito comum também que se invoquem funções com *timeout*, ou seja, com um atraso na execução, conforme o exemplo a seguir:

Código 3 - Função com atraso

```
var timeoutID;

function alertAtrasado() {
  timeoutID = window.setTimeout(alertLento, 2000);
}
function alertLento() {
  alert('Foi bem demorado!');
}
```

Fonte: Autor

No código acima, a função *alertLento* será executada apenas após 2 segundos pela função *alertAtrasado*. Isso pode ser útil caso o desenvolvedor dependa de algum dado que não esteja disponível ainda, ou caso queira agendar um horário específico para fazer alguma chamada ao servidor.

3.2.1 Promises e Callbacks

Por muito tempo, a única forma de se compor funções assíncronas foi através de *callbacks*, que basicamente são funções passadas como argumento para outra função, e que são executadas em um momento posterior, determinado pela função que as recebem. *Promises*, por sua vez, apesar de servirem o mesmo propósito, funcionam de maneira diferente, dando apenas uma garantia (*token*) de que algum valor será entregue para que o código que as requisitam consiga concluir sua execução.

A ideia de *callbacks* e *promises* pode ser de difícil compreensão para quem vem de outras linguagens, já que são conceitos extremamente relacionados a programação assíncrona.

3.2.1.1 Callbacks

Callback é uma função tipicamente passada como argumento de outra função, e chamada quando um evento ocorrer, ou quando uma parte de código receber uma resposta de que estava à espera. Imagine uma função *Func1* que chama *Func2*. *Func2* realiza alguma operação assíncrona, como uma requisição AJAX. *Func1* precisa saber o resultado desta requisição, portanto, passará como parâmetro para *Func2* uma outra função, *Cb1*, a qual *Func2* executará uma vez que o AJAX estiver concluído.

Podemos pensar que os *callbacks* servem para informar à funções que dependem do resultado de operações assíncronas sobre a finalização destas, e que a partir do resultado da requisição poderá continuar com a execução do algoritmo com os dados necessários.

Código 4 - Função Callback

```
function callback(e) {
alert('aconteceu um evento' + e.type);
}
window.addEventListener('click', callback);
//Este código chama a função callback
//quando houver um evento de click
```

Fonte: Autor

Infelizmente *callbacks* se tornam difíceis de serem gerenciados quando fazemos várias chamadas AJAX e uma requisição depende de outra que está em execução. Uma maneira melhor de lidar com múltiplas requisições assíncronas são as *promises*. Utilizando *promises*, fica mais fácil evitar situações de código como o exemplo a seguir, onde existem muitos níveis de indentação e múltiplos tratamentos de erro:

Código 5 - Multiplos Callbacks indentados

```
fs.readdir(source, function(err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
     files.forEach(function(filename, fileIndex) {
       console.log(filename)
       gm(source + filename).size(function(err, values) {
         if (err) {
           console.log('Error identifying file size: ' + err)
         } else {
           console.log(filename + ' : ' + values)
           aspect = (values.width / values.height)
           widths.forEach(function(width, widthIndex) {
             height = Math.round(width / aspect)
             console.log('resizing ' + filename + 'to ' + height +
'x' + height)
             this.resize(width, height).write(destination + 'w' +
width + '_' + filename, function(err) {
                 if (err) console.log('Error writing file: ' + err)
                 })
             }.bind(this))
           }
         })
      })
    }
});
```

Fonte: CALLBACKHELL (2017).

3.2.1.2 Promises

Uma *promise* representa o eventual resultado de uma operação assíncrona. O nome desse recurso se dá justamente pelo papel que ele desempenha, já que é um valor que é prometido a alguém, neste caso à alguma outra parte do código.

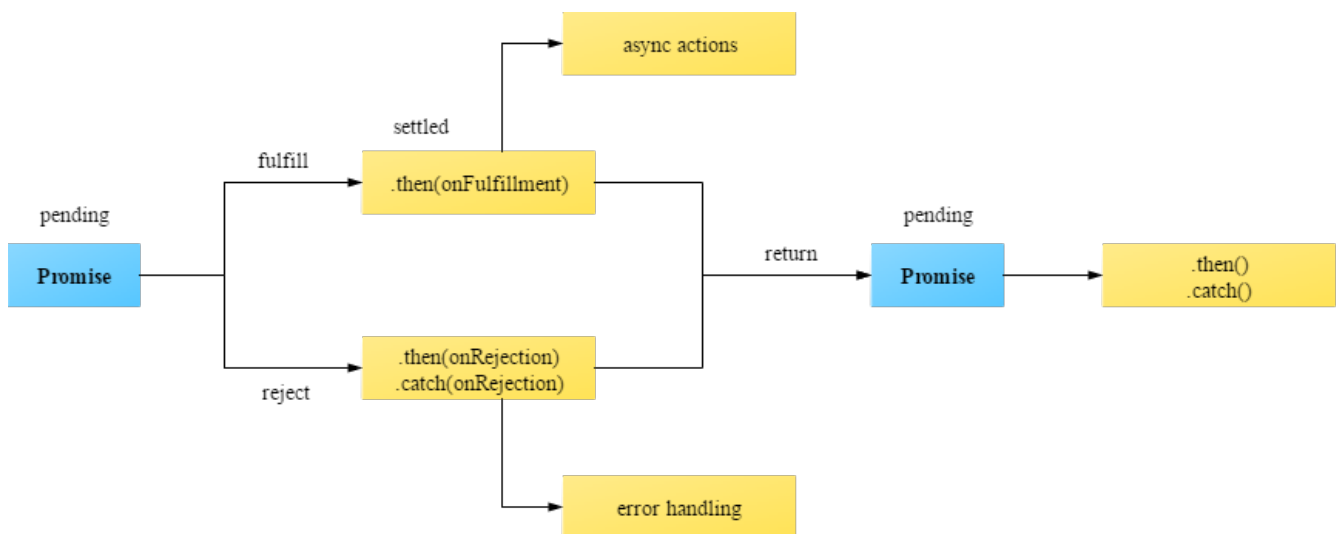
O objeto *promise* possui 3 estados:

- Pendente: a requisição está em andamento
- Resolvido: a operação completou com sucesso
- Rejeitado: a operação completou com erro

> Quando as usamos, o código executa um tipo de função chamada ***executor***, parecida com um *callback*, que recebe argumentos ***resolve*** e ***reject***. O *executor* requisita o dado externo, e te retorna a *promise* em status ***pending***. Quando a requisição é completada e o executor resolvido a *promise* muda para ***fulfilled***, o *token* é substituído pelo valor real do dado externo. Se o *executor* for rejeitado a *promise* muda para *promise rejected* e no valor é recebido o erro. (MONOBE, 2017)

O fluxo de uma promessa pode ser representado da seguinte maneira:

Figura 2 - Funcionamento de uma promise



Fonte: PROMISE (2017).

Através de promessas, o desenvolvedor consegue escrever código

assíncrono como se fosse síncrono, definindo através da função *.then()* o que ocorre uma vez que a promessa tenha completado com sucesso e através da função *.catch()* o fluxo caso tenha sido rejeitada ou finalizada com erro.

Código 6 - Exemplo de Promise

```
const geraPromiseFn = (msg) => () => {
    console.log(msg)
    return new Promise((resolve, reject) => setTimeout(resolve,
3000))
  }
  const primeiro = geraPromiseFn(`It's gonna be... wait for it...`)
  const segundo = geraPromiseFn('LE..')
  const terceiro = () => new Promise(() => { throw 'erro forçado' })
  const quarto = geraPromiseFn('DARY!')
  const tratar = (err) => console.log(`Tratando: ${err}`)
  const main = () =>
    primeiro()
    .then(segundo)
    .then(terceiro)
    .then(quarto)
    .catch(tratar)

main()
```

Fonte: MONOBE.

Neste exemplo do código acima forçamos um erro, que quando é lançado, interrompe o fluxo todo e faz com que a função *catch* seja executada para tratar o erro. Isso faz com que no console vejamos apenas a mensagem “Tratando: erro forçado”.

3.3. ES6 E ES5

Estamos atualmente em um período de transição entre versões da linguagem, que está indo do ECMAScript 5.1 para o ECMAScript 6 ou ES6. A nomenclatura das

versões do ECMAScript também está em uma fase de transição, pois a partir de 2015 elas passam a ser nomeadas de acordo com o ano de lançamento oficial, significando que ECMAScript 6, na verdade é oficialmente ECMAScript 2015, o que causa alguma confusão.

Essencialmente, as especificações da versão ES6 já estão finalizadas há algum tempo, porém, nem todos os navegadores suportam todas as mudanças implementadas, o que impede os desenvolvedores de tirar plena vantagem das novidades. Bibliotecas como o Babel, por sua vez, tornam código escrito em ES6 possível de ser interpretado por navegadores mais antigos e problemáticos.

Com o objetivo de demonstrar como as versões diferem, serão abordadas algumas das mudanças mais aguardadas pela comunidade, que pode participar ativamente do futuro da linguagem propondo melhorias e novas *features* através do repositório no GitHub https://github.com/tc39/proposals .

3.3.1 Classes

As classes em JavaScript introduzidas no ES6 são basicamente apenas "açúcar sintático"[6] sobre o já existente sistema de herança baseado em protótipo. A sintaxe de classe não traz consigo um novo modelo de herança orientado a objeto, apenas fornece uma maneira mais simples e limpa de criar objetos e definir seus protótipos.

3.3.1.1 Definindo classes

Anteriormente ao ES6 para criar uma "classe" era necessário seguir o seguinte procedimento, exemplificado no excerto a seguir:

[6] Em ciência da computação, açúcar sintático é uma sintaxe dentro da linguagem de programação que tem por finalidade tornar suas construções mais fáceis de serem lidas e expressas.

Figura 3 - Classe por protótipos em ES5

```
function Banana(tipo) {
    this.tipo = tipo;
    this.cor = "amarelo";
}
Banana.prototype.getInformações = function() {
    return 'banana'+ '' + this.tipo + ''+ this.cor;
     };
```

Fonte: Autor

Define-se uma função construtora Banana e, em seguida, dentro da propriedade *prototype*, presente em todos os objetos JavaScript, incluindo funções, definimos a função *getInformacoes*, dessa forma ela estará disponível da mesma maneira para qualquer instância de Banana criada.

Em ES6, as classes apresentadas, na verdade, são funções especiais, e análogo aos meios pelos quais declaramos funções, a sintaxe possui duas variações: *class expressions* e *class declarations*.

3.3.1.1.1 Class declarations

Uma maneira de definir classes é através da forma declarativa. Para declarar uma classe, usamos a palavra chave *class* com o nome da classe:

Código 7 - Class declaration em ES6

```
class Retangulo {
    constructor(altura, largura) {
        this.altura = altura;
        this.largura = largura;
    }
}
```

Fonte: Autor

3.3.1.1.2 Class expressions

Outra maneira de definir uma classe é através da class expression. Elas podem ser nomeadas ou não-nomeadas e o nome dado a uma *class expression* pertence apenas ao corpo da classe.

Código 8 - Class Expression em ES6

```
//não-nomeada
var Retangulo = class {
    constructor(altura, largura) {
        this.altura = altura;
        this.largura = largura;
    }
}
//nomeada
var Retangulo = class Retangulo {
    constructor(altura, largura) {
        this.altura = altura;
        this.largura = largura;
    }
      }
```

Fonte: Autor

3.3.2 Let

Até o ES5 as variáveis eram definidas através da palavra-chave *'var'*, e este era o único jeito de declará-las. O ES6 traz mais duas possibilidades para isso, as palavras-chave '*let*' e '*const*'. *var* é uma variável cujo escopo é ou global, ou o da função que a cerca, enquanto *let* é uma variável cujo escopo é o bloco no qual está inserida. Isso faz com que seja muito mais fácil reaproveitar nomes de variáveis, sem que elas sejam sobrescritas por outra função ou mesmo por iteradores ou condições.

Código 9 - Exemplo de declaração de variáveis usando let e var

```
function varTest() {
    var x = 1;
                                 if (true) {
      var x = 2;  // sobrescrevendo x!
      console.log(x);  // 2
    }
    console.log(x);  // 2
}
function letTest() {
    let x = 1;
    if (true) {
      let x = 2;  // variáveis diferentes!
      console.log(x);  // 2
    }
    console.log(x);  // 1
     }
```

Fonte: Autor

3.3.3 Const

Como o nome sugere, a palavra-chave *const* é usada para definir constantes, ou seja, referências para valores que são apenas de leitura. Para utilizar uma constante deve-se atribuir o valor no momento de sua criação, do contrário, uma exceção é lançada.

Código 10 - Declaração de constantes

```
// Constantes podem ser declaradas com letras maiúsculas
//ou minúsculas, mas é uma convenção comum usar apenas maiúsculas.

//definindo MY_FAV como const e dando a ela o valor 7
const MY_FAV = 7;

//Isto vai causar um erro! Não se pode redefinir consts
MY_FAV = 20;

//Isto causará um erro, pois constantes precisam ser inicializadas
//na declaração
const FOO;
```

Fonte: Autor

3.3.4 Funções seta (*Arrow Functions*)

Tradicionalmente funções em JavaScript tendiam a ser um pouco verbosas, especialmente por conta da palavra chave *function* necessária para declarar a função. Uma das mudanças mais aguardadas que o ES6 traz para os desenvolvedores é a possibilidade de declarar funções de uma maneira muito mais sucinta, utilizando apenas () =>, sendo este o motivo do nome *arrow function* (função seta).

Código 11 - Função convencional e arrow function

```
//Funções em ES5
var elementos = [‘hidrogenio’, ‘aluminio’, ‘helio’, ‘litio’]

elementos.map(function(elemento) {
    return elemento.length;
});
//retorna [10, 8, 5, 5]

//Em ES6
elementos.map((elemento) => {
    return elemento.length;
});
//retorna [10, 8, 5, 5]

//ou ainda mais simplificado em ES6
elementos.map(elemento => elemento.length);
     //retorna [10, 8, 5, 5]
```

Fonte: Autor

Outra diferença significativa nas funções seta do ES6 é que o *this* não é sobrescrito dentro corpo da função. Isso significa que não dependemos mais do método *bind()*, cuja serventia é justamente especificar o valor de *this*, ou de soluções convolutas e confusas para termos acesso ao *this* esperado, conforme o exemplo a seguir:

Código 12 - Referência ao valor de this em funções seta

```
// Em ES5
function Pessoa() {
    var that = this;
    that.idade = 0;  //that faz referência ao escopo global
    setInterval(function fazerAniversario() {
      // O callback referência o ‘that’ que foi o contexto
      //passado à declaração da função.
      that.idade++;
    }, 1000);
  }

  //EM ES6
  function Pessoa(){
    this.idade = 0;

    setInterval(() => {
      this.idade++; // |this| referencia o objeto pessoa
    }, 1000);
       }
```

Fonte: Autor

Vale ressaltar que várias das mudanças introduzidas pelo ES6, inclusive esta, já faziam parte do cotidiano de muitos desenvolvedores que utilizam *Coffeescript*, uma linguagem que é transpilada para JS e que inspirou muitas das especificações dessa versão do ECMAScript.

3.3.5 String

Essa é uma maneira limpa de escrever *strings* com interpolação de dados. Para escrevermos *strings* devemos cercar a *string* com crases (` `) em vez de aspas. Caso ela possua marcadores para interpolar expressões, estas deverão ser delimitadas usando um sinal de dólar seguido de chaves (${*expressão*}). Por meio de *strings* também é possível escrever *strings* multi-linha.

Código 13 - Exemplo de Strings

```
//EM ES5
console.log('texto linha 1\n' +
'texto linha 2');

//Em ES6
console.log(`texto linha 1
texto linha 2`);

//ambos resultam em
// "texto linha 1
// texto linha 2"
//Interpolação em ES5

var a = 5;
var b = 10;
console.log(Quinze é ' + (a + b) + ' e\nnão ' + (2 * a + b) +
'.');


//Interpolação em ES6
console.log(`Quinze é ${a + b} e
    não ${2 * a + b}.`);
//Ambos produzem:
// "Quinze é 15 e
// não 20."
```

Fonte: Autor

3.3.6 Spread *Operator*

O operador Spread traz praticidade para trabalhar com vetores. Com ele podemos, por exemplo, passar os elementos de um vetor de cinco posições para uma função que espera cinco argumentos. Ou compor um novo *array* usando como base todos os elementos de outro *array*.

Código 14 - Exemplo com o operador Spread

```
var arrayNumeros = [1, 2, 3, 4, 5]
function argFunction(a,b,c,d,e){
    console.log(a,b,c,d,e)
}
//podemos invocar a função argFunction passando
// todos os elementos de arrayNumeros atraves do operador …
argFunction(...arrayNumeros)  // 1, 2, 3, 4, 5

//podemos também criar um novo array a partir de arrayNumeros
var array2 = [...arrayNumeros, 6, 7]
//1, 2, 3, 4, 5, 6 ,7
```

Fonte: Autor

3.3.7 Destructuring

É possível utilizar o *Destructuring* de diversas maneiras. Com ele pode-se atribuir valores a mais de uma variável de uma só vez e também realizar a troca do valor de variáveis entre si. Além dessas funcionalidades, através dele podemos fazer a importação de componentes específicos exportados por outros arquivos, pois a partir do ES6 contamos com um sistema de exportação nomeada (*named export*) para que objetos, valores ou funções possam ser importados por nome por outros arquivos pelo nome.

Código 15 - Usos comuns do Destructuring

```
let [x, y] = [1, 2] // x = 1, y = 2

// para inverter o valor de x e y podemos fazer da seguinte forma

[x, y] = [y, x]
// Quando usando React é possível importar apenas o componente
//‘Component’ da biblioteca, em vez de ter que importar todos
// os componentes exportados por ela fazendo:

import { Component } from ‘react’;

    //dessa forma é possível usar Component em vez de
React.Component
```

Fonte: Autor

3.4 SUPERSETS

Supersets ou superconjuntos de JavaScript são linguagens que adicionam funcionalidades ou caracteristicas ao JS, mas cujas sintaxes aceitam também código escrito em vanilla JS. Sendo assim é possível integrar módulos e bibliotecas já existentes em JS à projetos que fazem uso de algum superconjunto.

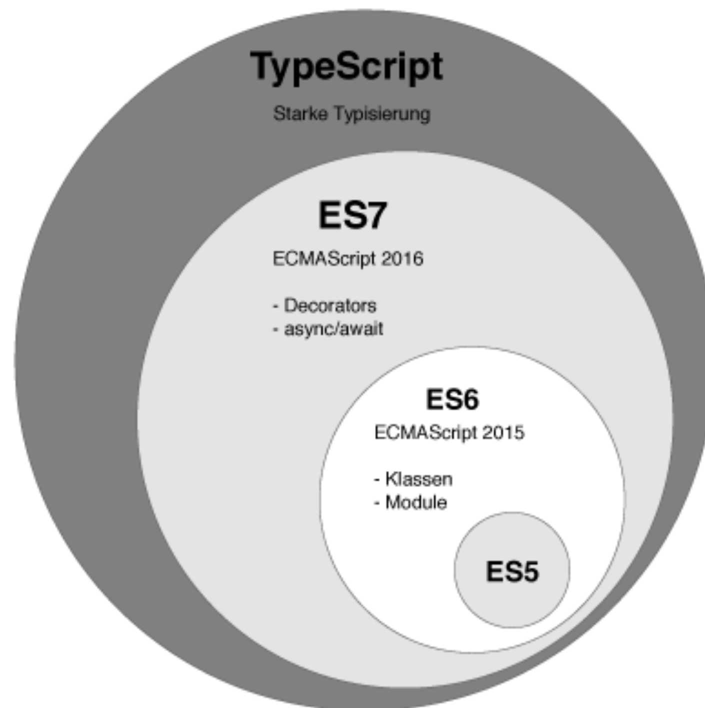
Das opções que se encontram atualmente disponíveis, TypeScript e JSX são duas das escolhas mais populares e isso em grande parte se deve a adoção massificada de ferramentas que recomendam o uso desses *supersets*.

3.4.1 TypeScript

Typescript é uma linguagem desenvolvida e mantida pela Microsoft, que adiciona várias características ao JavaScript, como a possibilidade de tipagem estática, que tradicionalmente não existe em JS.

Como *Typescript* é essencialmente JavaScript, é possível integrar qualquer módulo já existente em um projeto escrito em TS. Código escrito em TS é transpilado (traduzido) para JS para ser executado no ambiente alvo, seja o navegador ou *Node*, inclusive o próprio transpilador (tradutor) é escrito em TS.

Figura 4 - Os conjuntos da linguagem JavaScript da perspectiva do TypeScript



Fonte: ANGULAR - BUNCH

Apesar de recente - *Typescript* foi lançado ao público em 2012 -, a adoção à linguagem tem crescido exponencialmente, e grande parte do sucesso se deve ao lançamento do Angular 2, cuja própria equipe de desenvolvimento recomenda a utilização de TS em projetos construídos com o *framework*.

TypeScript surgiu da percepção das fraquezas do JavaScript para aplicações de larga escala. Códigos complexos com JavaScript levaram a uma demanda por ferramentas que facilitassem a criação de componentes com a linguagem. Sabendo que já existiam propostas para introdução de classes nas versões futuras de JS, os desenvolvedores por trás do TS seguiram essa linha ao elaborar suas especificações, o que de certa forma o torna uma prévia da versão 6 do ECMAScript.

Algumas das utilidades que o TS acrescenta ao ES5:

- anotação de tipo e checagem em tempo de compilação;
- inferência de tipo;

- interfaces;
- Enums;
- *mixin*;
- genéricos;
- *namespaces*;
- tuplas;
- *await*.

3.4.2 JSX

JSX é uma especificação de propriedade do Facebook que adiciona ao ECMAScript sintaxe similar ao XML e visa facilitar o desenvolvimento utilizando React. JSX necessita ser transpilado por pré-processadores (como o Babel), o qual realiza otimizações enquanto compila o código fonte para JS. O código gerado tende a rodar mais rápido do que código escrito diretamente em JS.

Exatamente como XML, as tags JSX tem nome, atributos e filhos/irmãos. Se o valor de um atributo estiver cercado por aspas, ele é então uma *string*, do contrário, é possível cercar o valor por chaves e torná-lo uma expressão JS.

O que acontece na prática é:

Código 16 - Código escrito em JSX e o resultado de sua compilação para JavaScript

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>

//É compilado para:

React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
      )
```

Fonte: Autor

O único propósito real de se utilizar JSX é deixar o código escrito usando React mais elegante e conciso, pois ele permite que marcação e lógica coexistam no mesmo arquivo.

4. JAVASCRIPT ONIPRESENTE

Apesar de ter seu uso massificado na parte de *frontend*, já que historicamente seu código era executado apenas nos navegadores, o número de ferramentas que utiliza JavaScript fora dos navegadores vem sofrendo um aumento significativo. Isso se deve integralmente ao Node.js, que nada mais é do que um interpretador de JS que roda no sistema operacional, o que possibilita que aplicações sejam feitas totalmente em JS.

4.1 A POPULARIZAÇÃO DO JAVASCRIPT

O site StackOverflow, criado e mantido por desenvolvedores para que usuários possam tirar dúvidas relacionadas à programação, realiza, anualmente, uma pesquisa para descobrir, dentre outras coisas, quais tecnologias os desenvolvedores gostariam de utilizar nos seus trabalhos. Com base nos dados levantados pela pesquisa de 2017 é possível tirar algumas informações interessantes a respeito do cenário atual do desenvolvimento web e de aplicativos. JavaScript aparece como linguagem de programação mais utilizada pelo quinto ano consecutivo em todas as categorias da pesquisa, o que inclui tecnologia mais utilizada por desenvolvedores profissionais.

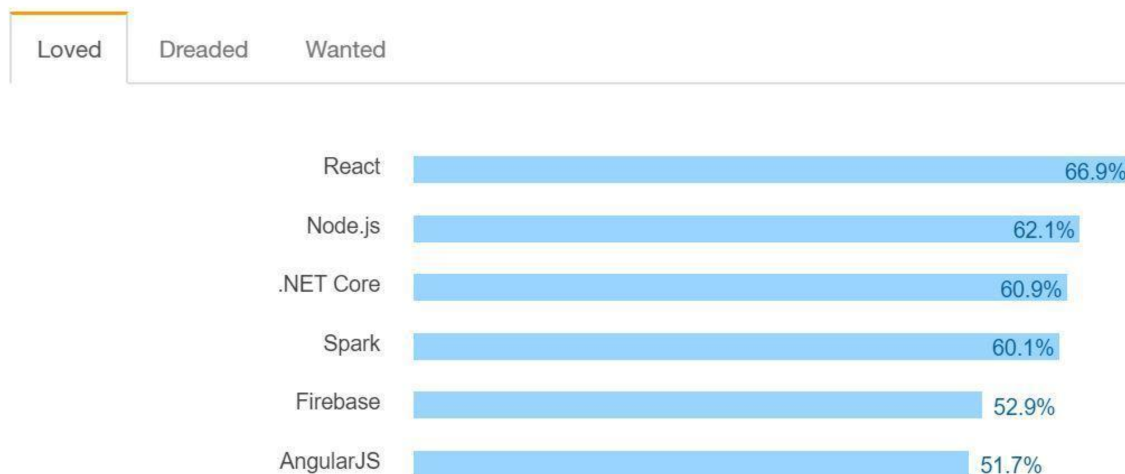
NodeJS aparece como ambiente mais utilizado quando a amostra é tanto desenvolvedores profissionais quanto pessoas que programam por qualquer outro motivo, enquanto AngularJS aparece como *framework* mais utilizado por desenvolvedores profissionais. Nessa categoria React aparece em 4º lugar. Essa pergunta é muito interessante pois abrange *framework*s de todas as linguagens, o que nos mostra como JS tem se tornado cada vez mais utilizado em todos os setores do desenvolvimento.

Durante os 5 anos em que a pesquisa tem sido realizada, JavaScript apenas

cresceu em popularidade. Ele aparece em segundo lugar quando a pergunta é sobre qual tecnologia os desenvolvedores mais gostariam de usar, ficando atrás apenas de *Python*. Já quando a pergunta é qual tecnologia é a mais temida, JS aparece em último lugar. E quando se trata da tecnologia mais amada, *Typescript* aparece em 3º lugar. Se levarmos em consideração que *Typescript* nada mais é do que um superconjunto de JS, temos mais um indicador da dimensão do crescimento da linguagem.

Quando se trata de quais *framework*s/bibliotecas os desenvolvedores mais amam, gostariam de utilizar ou temem utilizar os resultados são inconclusivos quanto a popularidade de algumas tecnologias, pois a pesquisa aponta React como biblioteca mais amada, quesito no qual Angular ocupa o 6º lugar.

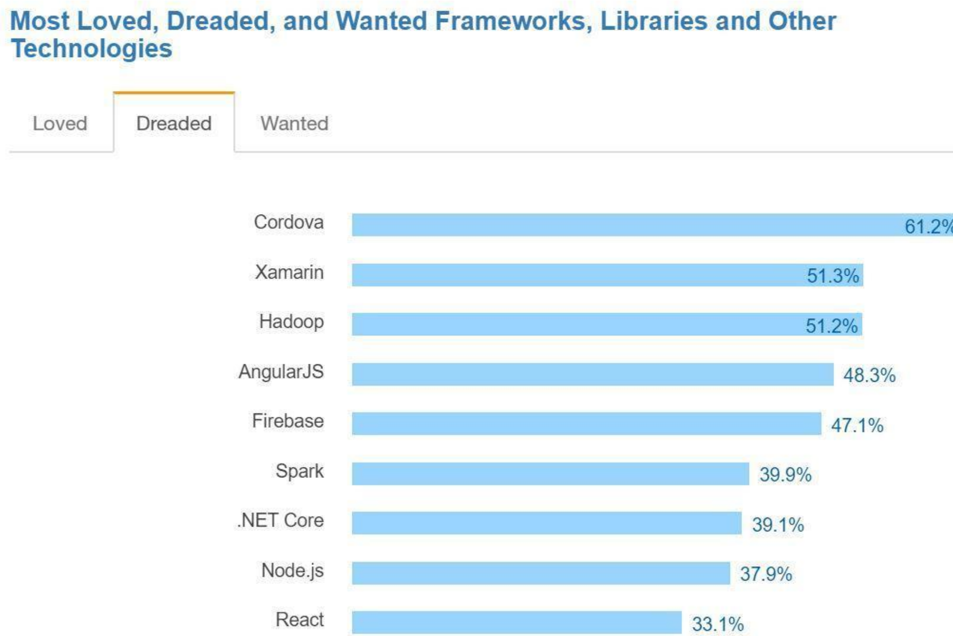
Figura 5 - Tecnologia mais amada segundo a pesquisa do site Stackoverflow



Fonte: **STACKOVERFLOW**

Já para o *framework* mais temido Angular fica em 4º lugar com um percentual expressivo, enquanto React fica em último lugar.

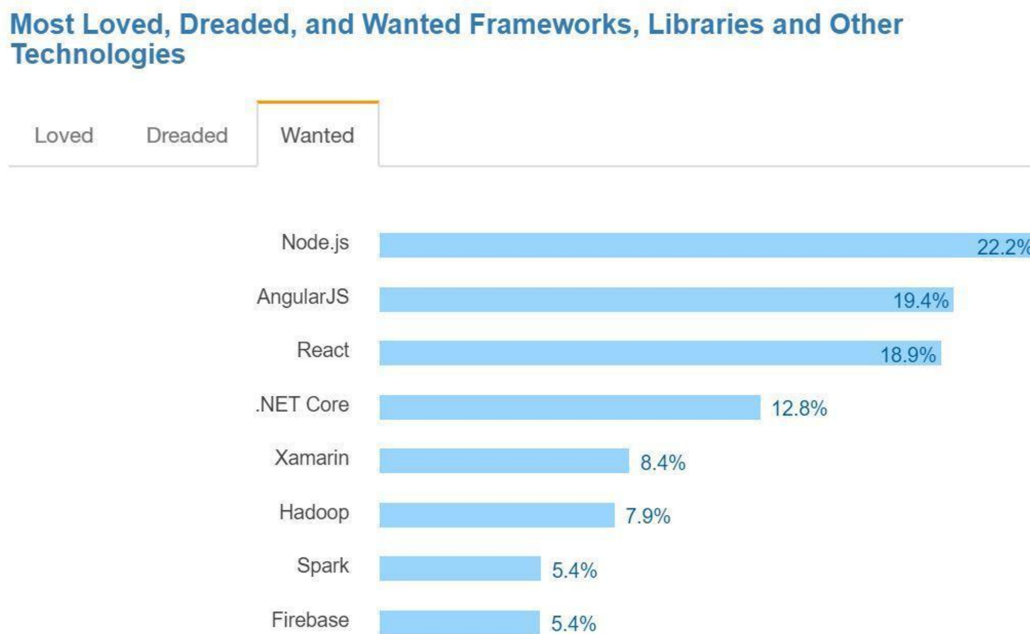
Figura 6 - Tecnologia mais temida segundo a pesquisa do site Stackoverflow



Fonte: **STACKOVERFLOW**

Surpreendentemente Angular aparece na frente de React quando o quesito é o *Framework* com o qual desenvolvedores gostariam de trabalhar.

Figura 7 - Tecnologia que os desenvolvedores mais gostariam de usar segundo a pesquisa do site Stackoverflow



***Fonte:* STACKOVERFLOW**

Em popularidade por ocupação, JavaScript é disparado a linguagem mais utilizada por desenvolvedores web, Cientistas de dados, especialistas DevOps e aparece em segundo lugar para desenvolvedores Desktop.

Sobre as tecnologias que pagam mais aos desenvolvedores Clojure é a linguagem que aparece em destaque no cenário mundial, pagando uma média de $72.000 dólares anuais para quem desenvolve nessa linguagem. JavaScript fica bem atrás, pagando uma média de $50.000 dólares anuais. Curiosamente *Coffeescript* aparece mais bem colocado no ranking, pagando cerca de $55.000 dólares por ano.

4.2 NODEJS

NodeJS foi ideia de um matemático chamado Ryan Dahl, que foi inspirado por uma barra de progresso enquanto tentava fazer upload de uma foto no site Flickr. Ele percebeu que o browser não sabia quanto do upload já havia sido concluído, e para obter essa informação necessitava requisitar ao servidor, o que pareceu-lhe muito ineficiente.

Dahl questionava as capacidades limitadas do que, na época, era o servidor web mais popular, o Apache HTTP *Server*, para lidar com muitas conexões concorrentes, e o fato de que a maneira mais comum de programar gerava problemas de bloqueio de processo (uma tarefa deveria ser concluída antes que outra pudesse ser iniciada) ou causava múltiplas pilhas de execução no caso de conexões simultâneas.

O projeto do Node foi apresentado no final de 2009, na primeira European JSConf. Node.js combinava o interpretador de JavaScript V8 do Google, um *loop* de eventos e uma API de baixo nível para input/output. Ao longo do tempo Node.js recebeu diversas atualizações importantes, mas a que merece maior destaque é com certeza o NPM.

O NPM - sigla para Node Package Manager - é uma maneira extremamente fácil de lidar com dependências entre módulos de uma aplicação. Utilizando o arquivo chamado *package.json*, gerado ao iniciar um projeto por meio do comando npm init, qualquer pessoa que quiser usar algum projeto existente pode, através do comando npm i, fazer o download e instalação de todas as dependências que o projeto utiliza já adequadas ao seu sistema operacional. O conceito do NPM em si não é exatamente novo nem único, pois outras linguagens e *frameworks* já contavam com isto, mas graças a ele os desenvolvedores JS passam a ter um repositório vasto de bibliotecas unificado e simples de usar.

A grande sacada do Node em relação aos concorrentes é que ele foi

projetado seguindo a natureza da linguagem na qual ele é escrito, para ser *non-blocking*, ou seja, ao contrário de servidores PHP, que ao executar um processo faz com que os demais processos fiquem parados até que o primeiro esteja concluído, as funções em Node.js executam em paralelo, utilizando-se de *callbacks* para apontar quando finalizam, seja com sucesso ou erro.

Node traz consigo a premissa de JavaScript onipresente, pois a partir dele é possível escrever aplicações completas com apenas uma linguagem, o que poupa um esforço considerável já que não é preciso se atentar à diferentes sintaxes, e traz também a vantagem de lidar naturalmente com JSON, já que este é um formato próprio à linguagem JS.

Alguns desenvolvedores ainda tem certo receio de preterir as linguagens dominantes do *back-end* em favor do JS, porém, a menor complexidade, a ampla opção de bibliotecas e o fato de que Node é novo, e por isso já aprendeu com os erros de outras linguagens, aos poucos vem quebrando essa resistência.

4.3 PRINCIPAIS *FRAMEWORK*S E BIBLIOTECAS FRONT-END DIVIDIDOS EM GERAÇÕES

Desde o surgimento da linguagem, seu propósito no browser sempre foi o mesmo: adicionar dinamismo e interatividade às páginas da internet. Por muito tempo essa tarefa foi dificultada por inúmeras razões, das quais cabe salientar duas em especial:

- As diferentes implementações da linguagem
- As confusas APIs do DOM

Eis que para lidar com as inconsistentes APIs do DOM surge JQuery, a primeira biblioteca popular para vanilla JS. JQuery abstraía as diferentes implementações do DOM e tornava mais fácil e menos arriscado fazer requisições AJAX e manipular nós do DOM. JQuery adicionava um objeto global $ que recebia um seletor - podendo este ser um *id*, *tag* html ou classe CSS - e a partir dele

executava um *callback*.

A biblioteca tornou-se e ainda é extremamente popular, porém, provou ser insuficiente a medida que mais e mais código era escrito, já que não fornece diretrizes para separação de responsabilidades. A partir daí tivemos três gerações de *framework*s que se propunham a mitigar o chamado código espagueti (código de difícil manutenção por falta de estrutura).

4.3.1 Primeira Geração

Temos então a primeira geração de *Framework*s JS, da qual o nome mais forte é possivelmente Backbone.js. Backbone trazia um pouco de ordem para a bagunça do JQuery, sendo que internamente ele próprio faz uso dessa biblioteca. Apesar de ter se firmado por um bom tempo como o *framework* preferido dos desenvolvedores, ainda havia necessidade de funcionalidades como *data-binding* (sincronização de dados entre *model* e *view*), presente em tecnologias RIA (Rich Internet Applications) como Flex e SilverLight, que na época competiam com JavaScript.

Nessa mesma geração aparece uma biblioteca que fornece *data-binding*, chamada de *Knockout. Knockout* trazia o conceito de “observáveis” para realizar atualizações de dados, porém sua popularidade nunca decolou, uma vez que não apresentava outras utilidades que facilitassem a criação de Single Page Applications (SPA).

4.3.2 Segunda Geração

A segunda geração de *framework*s traz como principal nome o AngularJS, que foi aberto ao público pelo Google em 2009. AngularJS é um *framework* MVC, com o estilo de código mais declarativo e um pacote completo de ferramentas para roteamento, *templating* e injeção de dependências. Ele elimina grande parte da dor de cabeça dos desenvolvedores de pensar na estruturação do projeto, já que carrega diretrizes fortes de como isso deve ser feito.

Cabe também uma menção ao Ember.JS, que tem muitos dos recursos encontrados no Angular, porém, com uma ênfase maior na parte de roteamento e preservação da

função do botão “voltar” dos navegadores, um problema comum em aplicações *single-page*, já que na barra de endereço a URL não muda.

4.3.3 Geração Atual

A geração atual é muito recente e ainda é cedo para determinar qual nome vai prevalecer como o mais proeminente, mas para fins instrucionais e comparativos podemos abranger Angular 2 (atualmente chamado também de Angular 4), React e Vue. Angular 2 e React disputam no momento a atenção da maior parte dos desenvolvedores *front-end*, como pode ser conferido na pesquisa feita pelo site *StackOverflow* abordada anteriormente, no entanto têm características totalmente distintas na sua implementação e proposta. Ambas fazem uso de algum superconjunto da linguagem JavaScript, sendo o *TypeScript* adotado pela equipe do Angular 2 e o JSX pelo React. Isso mostra que a evolução dos *framework*s é muito mais rápida do que a evolução da linguagem, uma vez que tornam-se necessários recursos mais modernos para garantir o desempenho e consistência do código desenvolvido com o uso destes *framework*s. Vue e React contam ainda com um recurso chamado de Virtual DOM (ou V-DOM), que abstrai o DOM dos navegadores para uma versão mais leve, a fim de tornar a atualização de dados mais performática.

Apesar de serem de gerações distintas, React disputa com AngularJS (Angular 1) o trono de ferramenta JS mais utilizada em projetos comerciais, ainda que o uso do Angular 2 esteja em plena ascensão.

4.4 CRIAÇÃO DO ANGULAR

AngularJS - também chamado de Angular 1 após o lançamento da segunda versão - foi criado em 2009, época da segunda geração de *framework*s, pelos sócios Miško Hevery e Adam Abrons, na Brat Tech LLC.

Hevery em seguida começou a trabalhar em um projeto no Google chamado

Google Feedback. Em 6 meses, com uma equipe de 3 desenvolvedores, escreveram mais de 17 mil linhas de código, e quanto mais o projeto crescia, mais difícil se tornava testar e fazer alterações. Foi por causa dessa frustração que Miško resolveu fazer uma aposta com seu supervisor, em que ele disse que seria possível reescrever a aplicação inteira em 2 semanas usando seu *framework*, chamado na época, *GetAngular*. Miško perdeu a aposta, mas conseguiu realizar o feito em não duas, porém, três semanas e reduziu de 17 mil linhas para aproximadamente 1500 linhas, o que foi considerado um grande sucesso, e levou seu supervisor a acelerar o desenvolvimento do *framework* dentro do Google.

Em uma entrevista à InfoWorld, site sobre tecnologia da informação, Hevery diz:

> Há muitos *frameworks* Web por aí. O que torna o Angular diferente são duas coisas. Em primeiro lugar, temos injeção de dependência, que é muito singular. Nenhum tem isso. Mas acho que o que realmente conquista as pessoas é que nós temos essa idéia de uma diretiva. Ao invés de escrever tudo dentro do JavaScript e, em seguida, ter um monte de s para gerar a UI (interface de usuário), você escreve uma grande quantidade de HTML, e HTML conduz a montagem da aplicação. É o inverso. É muito singular. Ninguém mais tem essa abordagem particular. (2013, tradução nossa)

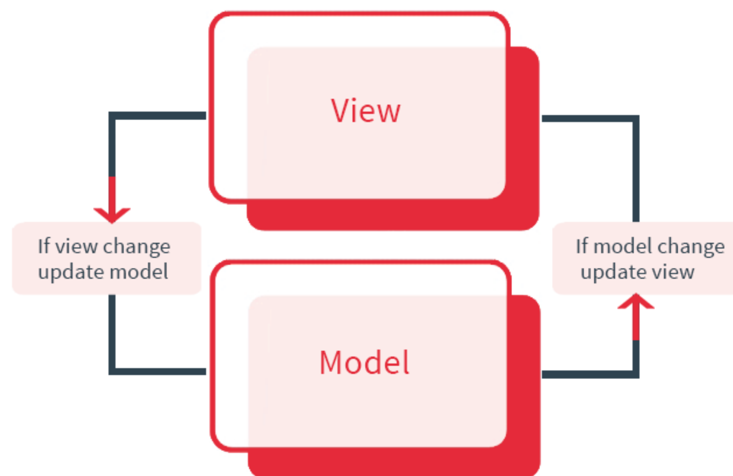
Diretivas são basicamente extensões do HTML, nas quais é possível definir aparência (caso seja uma diretiva do tipo elemento) e comportamento. Elas podem ser usadas tanto como *tags* quanto como atributos, e isso faz com que seja muito simples escrever a marcação da página, pois basta colocar o nome da diretiva criada entre chevrons, exatamente como *tags* HTML, e o definido na diretiva será renderizado na *view* principal da aplicação. Para criar um painel de fotos, por exemplo, basta definir a marcação deste painel, com efeitos e estilos CSS, declarar esta diretiva como uma diretiva do tipo Elemento (ou seja, que pode ser usada como uma tag HTML), definir seu escopo e que propriedades podem ser declaradas como atributos - neste caso é possível definir a fonte das imagens, pois assim este componente pode ser usado em diversas ocasiões mostrando imagens diferentes -

e pronto.

Angular traz em sua caixa de ferramentas alguns serviços, tais como o *$http* para realizar requisições AJAX, o *$timeout* para definir atrasos programáticos na execução de algum código, e também permite que se crie serviços customizados para a necessidade de cada aplicação. Para que o desenvolvedor possa alterar ou formatar os dados que vem na resposta da requisição, Angular possibilita a criação de filtros e *interceptors*.

Para gerenciar a abrangência de cada parte da aplicação, Angular possui escopos delimitados e acessíveis através do objeto *$scope*. Cada diretiva ou rota/estado define seu *$scope*, e elas podem compartilhar dados entre si. Para isso, AngularJS utiliza um sistema de *data-binding* baseado em eventos para automatizar o processo de atualização de dados compartilhados entre a *View* e o *Model*, pois havendo alteração em um, o outro é notificado e atualizado sem que o programador tenha que intervir.

Figura 8 - Two-way data binding



Fonte: CUSTODIA (2016).

AngularJS é suportado pela vasta maioria dos navegadores modernos, incluindo nessa lista versões do Internet Explorer a partir do 8. Ele foi construído

baseado na premissa de injeção de dependência, o que poupa bastante trabalho dos desenvolvedores, uma vez que se torna muito fácil construir módulos que dependam de outros. Além disso, foi pensado para aplicações *Single Page*, então tem também embutido um sistema de rotas bem versátil e robusto.

4.5 CRIAÇÃO DO *REACT*

React começou originalmente como uma adaptação do XHP, um *framework* para PHP criado pelo Facebook e utilizado para criação de componentes HTML reutilizáveis. XHP foi concebido para mitigar o risco de ataques por *cross-site-scripting*, que acontece inserindo scripts maliciosos em sites legítimos por meio de injeção de *payload* para que seja executado nos browsers de outros usuários. No entanto, XHP tem um problema, aplicações dinâmicas requerem muitas requisições ao servidor, e esse é o seu ponto fraco. Motivado por essa limitação, um desenvolvedor chamado Jordan Walke negociou com seu gerente para criar uma versão do XHP para JavaScript. Ele teve seis meses para realizar esta missão, e o resultado foi React.

A biblioteca fez sua aparição no *feed de notícia*s do Facebook em 2011 e mais tarde, em 2012, no Instagram, mas foi apenas em 2013 que ele se tornou aberto para os desenvolvedores.

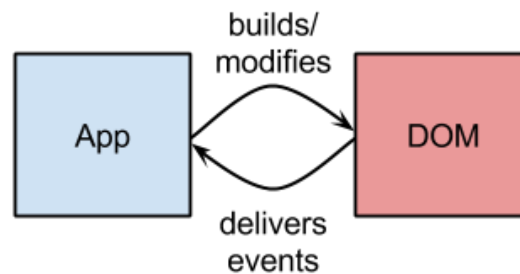
Como React é baseado em componentes, cada componente possui internamente uma propriedade *state*, e o funcionamento básico da mudança de estado ocorre da seguinte forma:

- A função *render()* pertence às propriedades *state* e *props*
- *Props* são atributos passados do componente pai para os seus filhos quando são renderizados
- O *state* não muda a não ser que o método *setState()* seja invocado de dentro do componente

- O props não muda a menos que componente pai seja renderizado novamente com valores diferentes para o props.

React funciona observando o estado de seus componentes, e manipula o DOM após realizar as comparações com o Virtual DOM apenas quando o estado é alterado. Diferentemente do que acontece em uma aplicação web tradicional, que interage extensivamente com o DOM

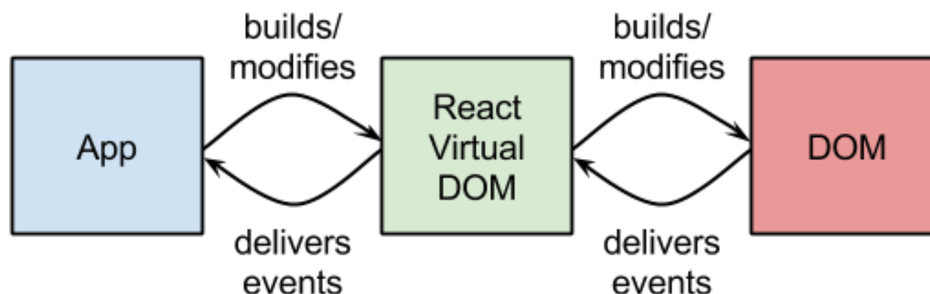
Figura 9 - Modelo tradicional de interação entre aplicação e DOM



Fonte: HABERMAN (2014)

Numa aplicação React este fluxo é diferente, passando a incluir uma etapa intermediária entre o cerne da aplicação e a interação com o DOM propriamente dita.

Figura 10 - Interação entre aplicação e DOM utilizando React

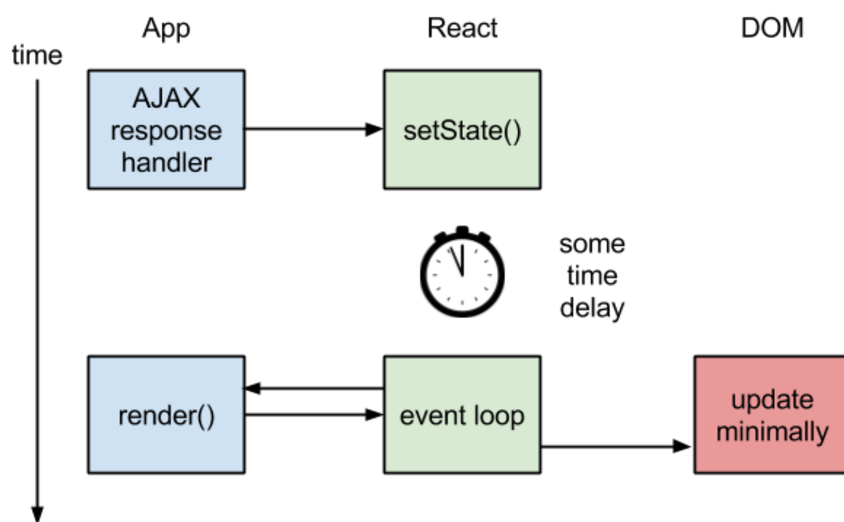


Fonte: HABERMAN (2014)

Em um primeiro momento, pode parecer que adicionar mais uma camada ao processo de atualização e sincronização de dados torna o processo mais custoso, no entanto, a semântica do Virtual DOM é diferente da encontrada no DOM.

Mudanças que ocorrem no V-DOM podem não surtir efeito imediatamente, pois elas aguardam o *loop* de eventos do React terminar sua execução para que as comparações sejam feitas entre o DOM e V-DOM e as mudanças aplicadas.

Figura 11 - Fluxo de eventos até que a view seja atualizada



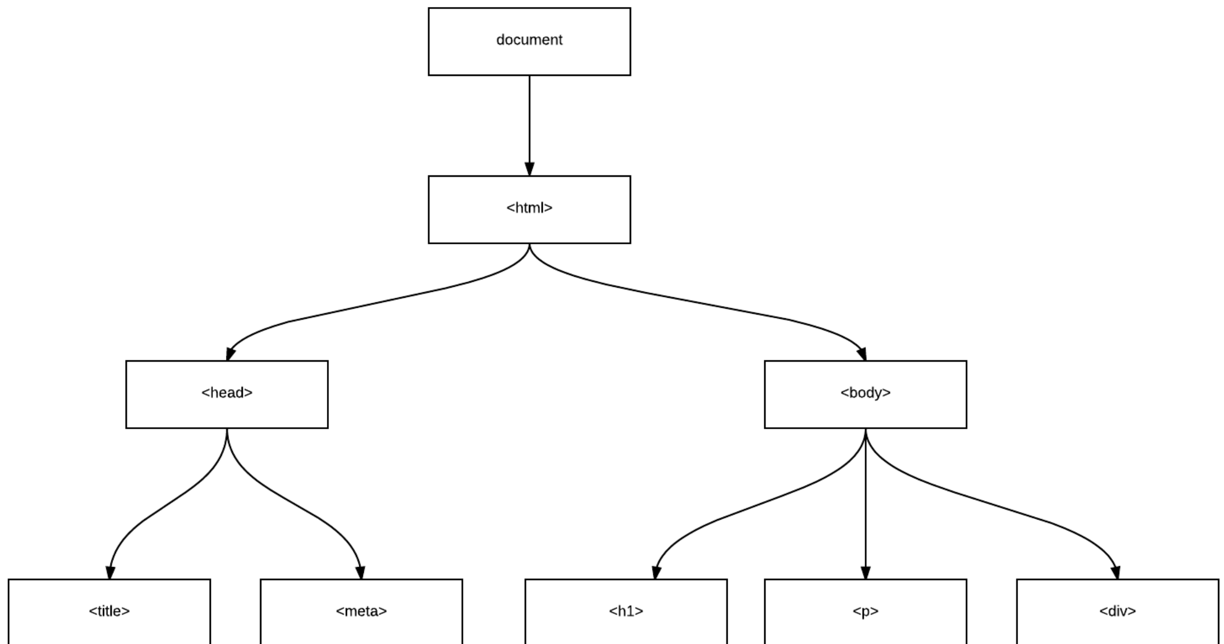
Fonte: HABERMAN (2014)

4.5.1 DOM e V-DOM

Antes que possamos falar sobre as diferenças entre o DOM e o Virtual DOM é necessário uma compreensão maior sobre o que é, e o que compõe o DOM.

O DOM é uma abstração da marcação HTML de uma página. Ele estrutura os elementos HTML em uma árvore de objetos e nós, mantendo as relações de pai e filho para elementos aninhados. Isso fornece uma API que nos permite navegar nós (elementos HTML) e manipulá-los de diversas formas.

Figura 12 - Estrutura do DOM



Fonte: Autor

Manipular o DOM tende a ser custoso e ineficiente, já que ele foi criado para páginas renderizadas pelo servidor e que não requeriam alterações dinâmicas. Quando ele atualiza, necessita modificar o nó e também reestilizar a página com o CSS e layout adequados. Com a popularização de aplicações *single-page*, componentes são cada vez mais frequentemente modificados e portanto responsáveis por ficar escutando atualizações e re-renderizando novos dados na interface gráfica.

Existem essencialmente duas maneiras de verificar alterações:

- *Dirty checking*: navegar todos os nós da página em determinado intervalo de tempo para avaliar se houve alguma modificação. Este método costuma ser lento e custoso;
- Observáveis: os componentes são responsáveis por escutar os

updates. Uma vez que os dados estão salvos no estado, os componentes apenas escutam eventos e, caso haja alguma atualização, ele re-renderiza os novos dados. React utiliza essa forma.

Já o V-DOM é uma abstração leve do DOM. É possível entendê-lo como uma cópia do DOM, que pode ser modificada sem afetá-lo diretamente. Ele possui todas as propriedades encontradas no DOM real, mas não tem a habilidade de renderizar na tela, e a cada atualização do DOM o V-DOM é recriado.

Quando updates são fornecidos para o V-DOM, React usa de um processo chamado de reconciliação - baseado em um algoritmo de *diff* que compara o DOM com o V-DOM para determinar quais mudanças ocorreram. React então atualiza apenas os elementos que mudaram, deixando todos os outros intactos.

Vale lembrar que React não inventou o V-DOM e outras bibliotecas já faziam o uso dele.

4.5.2 A Arquitetura Flux

Essa é uma arquitetura criada com o intuito de resolver um problema que acomete os *frameworks* MVC utilizados no *Front-end*, o excesso de comunicação dos *controllers* com as *views*. O JS possui uma diferença com relação às linguagens mais comumente utilizadas no *back-end*, que é o tempo de vida dos objetos. Enquanto no *back-end* os dados de uma requisição GET são destruídos logo após a requisição ser atendida, no JS esses objetos permanecem vivos. Para lidar com essa situação, *framework*s como AngularJS implementaram a solução conhecida por "*Two-way data binding"*, que basicamente consiste de observadores que a todo instante verificam se há mudanças no conjunto de dados, e, caso haja, os novos dados são reescritos na *View.* Isso apesar de ter sua utilidade, vem com alguns custos:

- Velocidade: Observadores consomem muitos recursos, causando lentidão a medida que a quantidade de dados aumenta.
- Inconsistência: A cada reescrita de dados fica mais difícil garantir a consistência dos mesmos, dada a natureza assíncrona da linguagem.

- Escalabilidade: Uma aplicação que tem dificuldades em lidar com grandes volumes de dados é uma aplicação que não escala

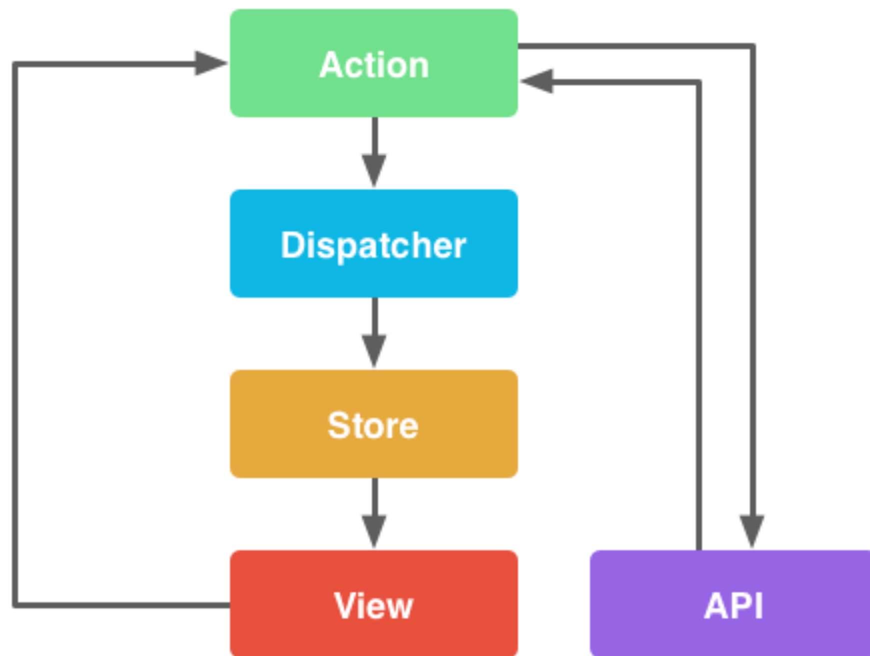
Destarte, o *Flux* foi concebido essencialmente para fornecer uma alternativa para os desenvolvedores não ficarem presos ao *two-way data binding*, justamente porque o Facebook lida com um volume absurdo de dados, pareceu o passo óbvio a se tomar. *Flux* é uma arquitetura utilizada internamente pela equipe do Facebook ao trabalhar com React. Ele não é um *framework* ou uma biblioteca, o que significa que não é um arquivo a ser importado na aplicação. O que os desenvolvedores do Facebook fizeram foi formalizar uma arquitetura de desenvolvimento com React baseada no conceito de fluxo unidirecional de dados. Tendo isso em mente, vale mencionar que o Facebook fornece um repositório que contém uma biblioteca *Dispatcher*, e este é o único código de terceiros necessário para desenvolver uma aplicação *Flux*.

O *Dispatcher* é um *handler* que registra os *callbacks* e emite eventos. Sendo o componente central de uma aplicação *Flux*, fica claro que deve haver apenas um *Dispatcher* para toda a aplicação.

Para entendermos a arquitetura *Flux*, precisamos saber que elementos a compõem. Eles são:

- *Action* - métodos auxiliares que facilitam o transporte de dados até o *Dispatcher*. São as ações que a aplicação realiza;
- *Dispatcher* - recebe ações e faz o *broadcast* de *payloads* aos *callbacks* registrados;
- *Store* - *stores* são responsáveis por manter os dados da aplicação, o que de certa forma se assemelha ao scope do AngularJS. A alteração dos dados de uma *Store* é feita por meio de *Actions* recebidas pelo *Dispatcher*. Ao ter seus dados alterados, a *Store* emite então um evento de *Change*;
- *View* - a *view* é a camada visual da aplicação, e pode ser tanto HTML normal, como web *components* ou mesmo uma função JS que gere HTML. Seu papel é mostrar os dados de uma *Store* e emitir *Actions* com interações do usuário.

Figura 13 - Interação entre os componentes do Flux



Fonte: WHEELER (2016)

5. COMPARATIVO ENTRE ANGULARJS E *REACT* NOS QUESITOS COMPONENTIZAÇÃO, DESEMPENHO, TRATAMENTO DE EVENTOS E *TEMPLATING*.

AngularJS e React têm sido as ferramentas de escolha de muitos projetos comerciais, sendo assim a demanda por profissionais que dominem essas tecnologias aumentou bastante nos últimos anos. Para que se possa fazer uma decisão mais acertada ao escolher que tecnologia empregar em um eventual projeto, temos abaixo alguns dados objetivos a respeito dos alvos da comparação.

Atributo	**AngularJS**	**React**
Versão	1.6.0-rc2	0.15
Autor	Google	Facebook
Linguagem	JavaScript/HTML	JSX
Tamanho	143k	151k
Contribuidores *no Github*	1,386	604
Ano de Lançamento	2009	2013

Este estudo vai utilizar-se de alguns critérios para avaliar qual tecnologia se sai melhor em determinadas situações. São os critérios: Componentização, Desempenho, *Event Handling* e *Templating*.

5.1 COMPONENTIZAÇÃO

Angular possui uma estrutura mais rígida quando se trata da criação de componentes, justamente porque é um *framework* MVC completo. Para escrevermos um componente em AngularJS, precisamos “quebrá-lo” em vários arquivos, sendo um arquivo com a marcação HTML, outro com o comportamento (JavaScript) e, dependendo de como a aplicação é construída, um arquivo de índice que faz as importações necessárias para o funcionamento do componente. No arquivo em que declaramos a diretiva definimos se ela possui um escopo próprio ou usa o do contexto no qual está inserida, seu nome e sua função link, que por sua vez será executada assim que o componente terminar de renderizar. Já no controlador definimos comportamento, e no *template*, a parte visual.

Uma vez definidos todas as partes de uma diretiva, podemos adicioná-la ao *template* principal da aplicação através de *tags* HTML ou de atributos, dependendo da restrição dada na criação do componente.

Já com React, formular componentes é como compor um programa JavaScript com funções. É possível criar classes e *templates* com React usando JavaScript puro, o que faz com que seja fácil portar aplicações já existentes em outros *framework*s para aplicações feitas em React. Porém, é preferível e aconselhado pelos desenvolvedores da biblioteca que se utilize JSX para que o retorno da função *render()* seja algo bem próximo do HTML.

Em termos de facilidade de criação de componentes, React leva uma grande vantagem. Torna-se muito fácil compor componentes bem granulares, já que lógica e marcação podem ser feitos no mesmo arquivo.

5.2 TRATAMENTO DE EVENTOS (*EVENT HANDLING)*

Em Angular, os eventos são tratados basicamente através de diretivas. Um exemplo claro de como isso acontece é a diretiva *ng-click*, que faz a ligação entre uma função *callback* e o evento de *click* em um elemento HTML que estejam no mesmo *$scope*.

Uma diretiva simples, para cumprimentar o usuário pode ser escrita da seguinte maneira:

Código 17 - Diretiva do Angular com tratamento do evento de clique

```
var app = angular.module("testApp", []);

app.controller("testController", function($scope) {
  $scope.nome = "Fulano de tal";
});

app.directive("helloWorld", function() {
    return {
        restrict: "E",
        template: "<button ng-
click='cumprimentar();'>Cumprimentar</button><br
/><div>{{saudacao}}</div>",
        scope: {
          nome: "@"
        },
        link: function(scope) {
          scope.saudacao = "";

          scope.cumprimentar = function() {
            scope.saudacao = "Olá, Fulano de tal!";
          }
        }
    };
  });
```

Fonte: Autor

Esta diretiva *helloWorld* possui um botão que ativa a função cumprimentar ao ser

clicado. Na função link podemos designar outros tratamentos de eventos e inicializar valores.

Tratamento de eventos em React funciona adicionando atributos customizados aos elementos HTML. Em vez de passar uma *string* para o *onClick* do botão, passamos uma expressão JavaScript.

Código 18 - Como react lida com tratamento de eventos

```
render: function() {
    return <div>
      <button onClick={this.cumprimentar}>Cumprimentar</button>
      <br />
      <span>{this.state.saudacao}</span>
    </div>;
}
});

     React.render(new  HelloWorld({  name:  "Fulano  de  tal"  }),
document.body);
```

Fonte: Autor

No exemplo acima temos uma situação corriqueira, que é o registro de um *callback* para um evento de clique em um botão.

Neste item da comparação AngularJS leva vantagem sobre React por apenas um motivo, lidar com eventos customizados em React exige muito mais código do que em AngularJS. Ao contrário do React, Angular disponibiliza um método *$on* de uso simplificado, que permite ao desenvolvedor registrar *callbacks* para eventos gerados em qualquer parte do código.

5.3 TEMPLATING

A verdade é que a maior parte do tempo gasto ao desenvolver uma aplicação

web, é criando a interface de usuário. Painéis, listas, menus, geralmente contém muita informação e não são reutilizáveis.

AngularJS possui embutidas várias diretivas padrão que facilitam a inserção de dados no DOM e também a criação de elementos repetidos, como a *ng-repeat*, que itera sobre uma lista e repete uma porção de HTML usando os dados de cada elemento do array. Já que React não traz recursos/componentes prontos, caso o desenvolvedor precise desse tipo de recurso, será necessário escrever um componente que desempenhe este papel. Apesar de não ser uma tarefa extremamente complexa, o fato de ter que criar componentes para atender algumas necessidades absolutamente básicas e conhecidas, torna React uma opção mais custosa, o que faz do Angular uma opção mais prática.

5.4 DESEMPENHO

Chris Harrington, desenvolvedor web e instrutor de JavaScript, descreveu, em um de seus artigos, um teste bem interessante a respeito das diferenças de desempenho entre React, AngularJS e KnockoutJS. O teste seguiu da seguinte forma:

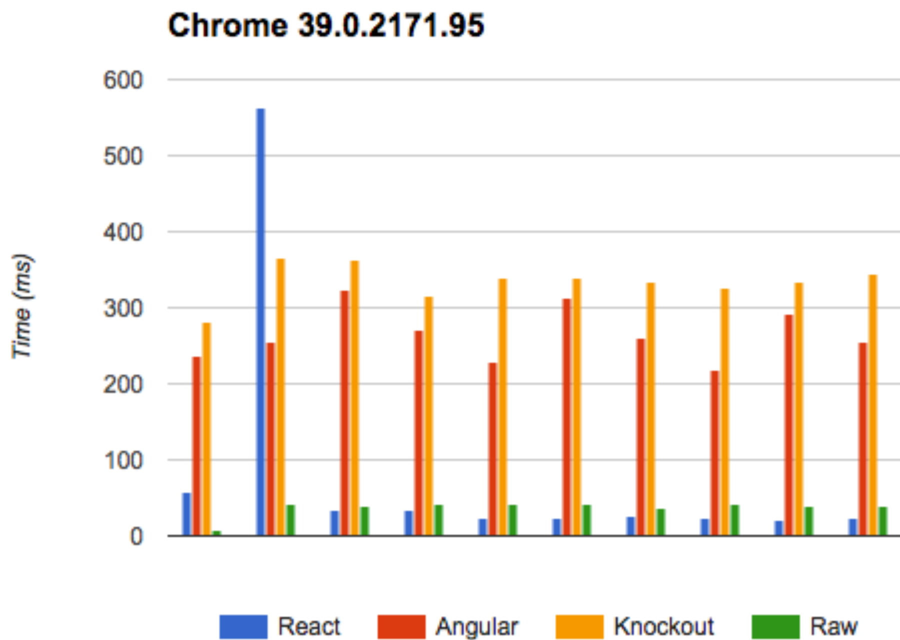
- Ele criou componentes usando as três ferramentas;
- Os componentes deveriam renderizar uma lista interativa com 1000 itens
- O vencedor seria o componente que levasse menos tempo para ser exibido pelo *browser.*

A construção de tais componentes neste caso é irrelevante, já que não está sendo avaliada a facilidade de realizar a tarefa, e sim o quão performático é o componente criado. Para estabelecer uma base comparativa, ele criou também a mesma lista utilizando apenas manipulação direta do DOM com jQuery, a qual ele esperava que obtivesse o melhor desempenho. O mesmo código foi testado em 3

navegadores: Chrome 39.0.2171.95, Firefox 34.0.5 e Safari 7.0.2, e o teste consistia em abrir a página teste 10 vezes em cada navegador e monitorar o tempo gasto.

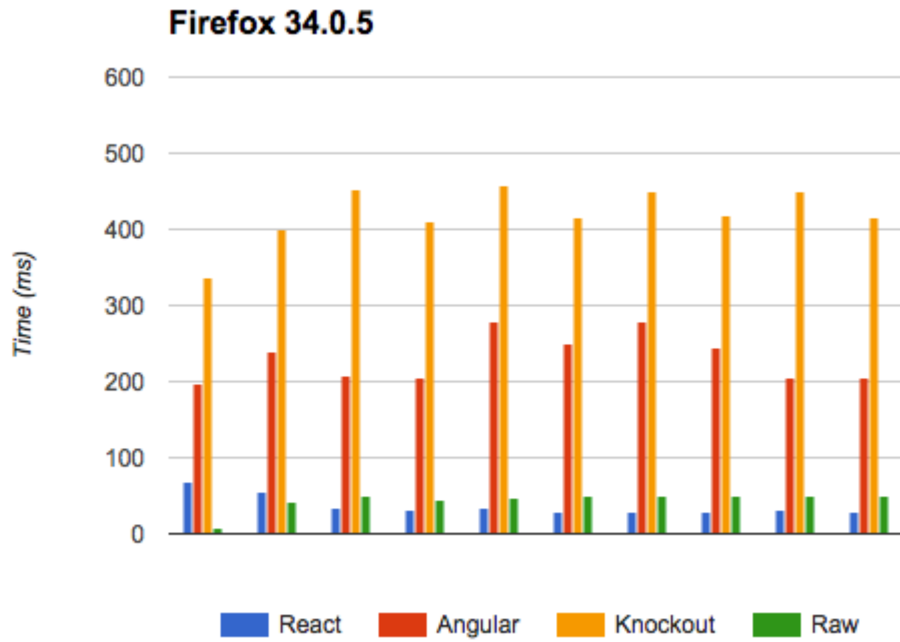
Os resultados foram surpreendentes. Os gráficos mostram o tempo em milissegundos para cada execução dos 4 componentes em cada um dos navegadores:

Figura 14 - Desempenho de cada componente no navegador Chrome



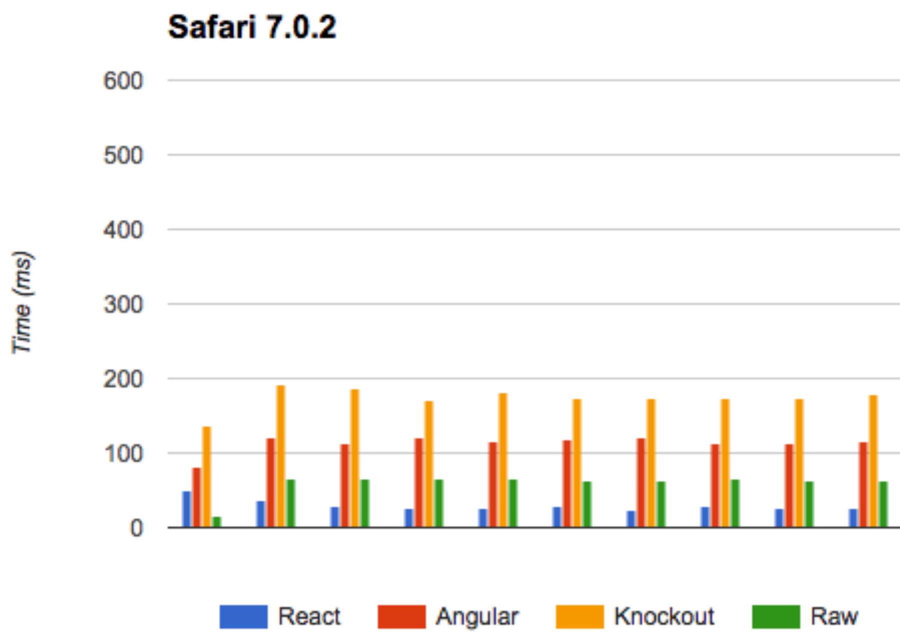
Fonte: Codementor (2017)

Figura 15 - Desempenho de cada componente no navegador Firefox



Fonte: Codementor (2017)

Figura 16 - Desempenho de cada componente no navegador Safari



Fonte: Codementor (2017)

A média de performance geral da tabela construída usando React foi melhor inclusive do que a tabela feita usando manipulação direta no DOM. KnockoutJS ficou em todos os testes com o pior desempenho. Angular apresenta um desempenho razoável, porém, neste teste, muito aquém do React. Outro fato interessante é que o navegador que leva menos tempo para renderizar a página, exceto quando utilizando a manipulação direta do DOM, é o Safari, desenvolvido pela Apple.

6. PARA ONDE CAMINHA A WEB

Muito tem sido debatido a respeito do futuro do desenvolvimento web, e apesar de todo o barulho, apenas uma coisa é certa: em um futuro próximo teremos uma gama de linguagens muito maior disponíveis para serem utilizadas na web. Isso se deve amplamente ao WebAssembly, que visa ser um código binário para web.

WebAssembly, também chamado de *wasm,* é o progresso de um trabalho que iniciou com o asm.js, que por sua vez é um subconjunto de JavaScript que adiciona certas características à linguagem e ao interpretador. Desenvolvedores poderão programar em qualquer linguagem, e através de um compilador, ter seu código traduzido para *wasm*, que é uma representação em baixo nível e rodará em uma espécie de máquina virtual do navegador. A grande vantagem é que os arquivos ficarão muito mais leves, terão alguns aspectos de segurança, como tipagem estática e o melhor de tudo: qualquer aplicação ficará imensamente mais fácil de ser portada para a web.

Um dos truques utilizados pelo asm.js para se tornar veloz é que em JS todos os números são do tipo *double*, já em asm.js uma adição é seguida imediatamente por uma operação *E bit-a-bit*, o que faz com que a CPU execute como uma operação de Inteiros simples. Portanto, asm.js faz com que seja mais fácil para as máquinas virtuais utilizar o potencial das CPUs, porém, fica limitado ao que é expressível em JavaScript. WebAssembly não é limitado desta maneira e permite que a CPU utilize ainda mais capacidades como:

- Operações com inteiros de 64-bits, que podem ser até 4 vezes mais rápidas. o que incorre em menos tempo para criar *hashes* ou para executar algoritmos de encriptação.
- Várias instruções de máquina como *popcount, copysign,* entre outras, tendo cada uma delas suas aplicações específicas.

Uma das indústrias que se beneficiará bastante dessa tecnologia é a de jogos, já que a grande maioria dos jogos atuais são desenvolvidos em C/C++, e por mais que já existam maneiras de ter código traduzido dessas linguagens para

JavaScript, com ferramentas como o *Emscripten*[7], o desempenho utilizando *wasm* promete ser ainda melhor.

JavaScript, porém, não está com os seus dias contados. WebAssembly vem apenas para preencher a lacuna que JS no momento preenche, a de ser a linguagem alvo de compiladores de outras linguagens para a web.

[7] *Emscripten* é uma ferramenta baseada em Low Level Virtual Machine (LLVM), uma tecnologia que visa ser a representação intermediária entre o código fonte e código binário, que compila C ou C++ para asm.js

7. CONCLUSÕES

Tendo o JavaScript ganhado cada vez mais importância, dentro e fora dos navegadores, mais e mais ferramentas são criadas para tornar as aplicações cada vez mais ricas e completas. Tantas opções podem gerar confusão na hora de escolher uma tecnologia nova para construir um projeto, e isso faz com que obter um contexto comparativo entre elas seja de grande serventia.

Com o propósito de mapear a evolução das bibliotecas e *framework*s utilizados em JavaScript e também de ampliar os conhecimentos adquiridos em engenharia de software nas Unidades Curriculares do curso de Gestão da Tecnologia da Informação, o presente trabalho tencionou demonstrar de forma bem estruturada como se modificaram, tanto a linguagem quanto suas ferramentas desde suas implementações originais. De maneira a tornar evidente, também, as mudanças nos papéis desempenhados pelo JavaScript e como suas ferramentas amadureceram no sentido de entregar cada vez mais funcionalidades, permitindo que diferentes *design patterns* sejam aplicados, bem como possibilitando a criação de aplicações com desempenho cada vez melhor.

O contato fornecido pelas disciplinas ministradas no curso de Gestão da Tecnologia da Informação foi essencial para que a compreensão se desse de maneira mais eficiente sobre como e porque as bibliotecas, *framework*s e superconjuntos foram criados, quais necessidades eles pretendem atender e onde erraram e acertaram.

Com base no comparativo feito entre AngularJS e React, fica claro que não existe uma ferramenta que seja a melhor em todas as situações, cabendo ao desenvolvedor elencar suas prioridades e adequá-las ao que cada opção oferece. Caso precise criar muitos componentes granulares, React parece ser a escolha mais acertada, mas se tempo for um fator limitante, AngularJS torna-se uma melhor opção.

À vista disso, os objetivos propostos pelo trabalho foram alcançados e por meio deste foi possível avaliar com sucesso a evolução de uma parte importante da web, a linguagem JavaScript. Este estudo torna-se mais um recurso para quem trabalha com desenvolvimento, e necessita aprofundar um pouco mais os conhecimentos sobre JavaScript, pois por mais que novas alternativas estejam surgindo, de acordo com a pesquisa realizada, ele vai continuar sendo a linguagem predominante da web por mais algum tempo. Assim sendo, mesmo desenvolvedores *back-end* ou desenvolvedores de jogos, caso queiram ter seus projetos portados para web, vão acabar lidando com JS de alguma forma.

Por fim, foi possível realizar uma previsão sobre o futuro das aplicações web. Ao que tudo indica o WebAssembly não vem para tomar o lugar do JS, mas sim preencher uma lacuna que atualmente ocupada pelo JavaScript com algumas deficiências. JS apesar de ser uma linguagem que oferece boa performance, não é a opção mais indicada para levar os desenvolvedores mais próximos do código binário, por assim dizer. O que significa que construir interfaces entre recursos de máquina utilizando JS, algo que vem sendo feito desde a expansão da “Internet das coisas”, é uma tarefa nem sempre possível e WASM vem para facilitar neste sentido.

7.1 RECOMENDAÇÕES PARA TRABALHOS FUTUROS

A web segue em plena expansão, e cada vez mais veremos ferramentas poderosas sendo oferecidas somente em versão online / aplicação móvel. Sendo assim, fica como indicação para pesquisas futuras verificar se as previsões feitas neste trabalho se concretizaram. Que rumo tomou a linguagem JavaScript e se teve sua participação diminuída após a adoção mais significativa do WASM pelos desenvolvedores. Com o passar do tempo certamente haverá mais material e projetos desenvolvidos utilizando WebAssembly, logo será possível medir o impacto tanto na performance das aplicações, quanto no espaço ocupado pelo JavaScript

Outra sequência possível para este trabalho seria acompanhar a evolução do Angular 2 e como ele se sai em uma comparação com Vue.js, que tem algumas similaridades com React. Ainda na linha de ferramentas da linguagem, existem ainda os *framework*s para JavaScript no lado do servidor que podem ser explorados.

Como este estudo trouxe um apanhado geral das mudanças introduzidas com a nova versão oficial do ECMAScript, seria interessante também realizar uma pesquisa sobre as versões vindouras, uma vez que as especificações da versão 7, ou ECMAScript 2016, já estão fechadas e uma oitava versão, chamada de ECMAScript 2017 está em desenvolvimento.

REFERÊNCIAS

AJAX. Disponível em: <https://developer.mozilla.org/en-US/docs/AJAX>. Acesso em: 14 de Set. De 2017.

AMBLER, Tim; CLOUD, Nicholas. **JavaScript Frameworks for Modern Web Dev**. Apress; 2015.

ANGULAR-BUCH. Disponível em: <https://angular-buch.com/>. Acesso 10 de Out. de 2017.

ATTRIBUTE Directives. Disponível em: <https://angular.io/guide/attribute-directives>. Acesso em: 17 de Out. De 2017.

A vocabulary and associated APIs for HTML and XHTML. Disponível em: <https://www.w3.org/TR/html5/>. Acesso em: 15 de Out. De 2017.

BERNERS-LEE, Tim. **Answers for Young People.** Disponível em: <https://www.w3.org/People/Berners-Lee/Kids.html>. Acesso em: 15 de Out. De 2017.

CALLBACK Hell. Disponível em: <www.callbackhell.com>. Acesso em: 10 de Out. de 2017.

CODESIDO, Ivan. **What is front-end development?.** Disponível em: <https://www.theguardian.com/help/insideguardian/2009/sep/28/blogpost>. Acesso em: 19 de Set. De 2017.

HARRINGTON, Chris. **CODEMENTOR.** Disponível em: <https://www.codementor.io/chrisharrington/react-vs-angularjs-vs-knockoutjs-a-performance-comparison-85hwzepb> Acesso em: 11 de Out. de 2017.

COLUNISTA PORTAL - INFORMATICA E TECNOLOGIA. Google Analytics. Disponível em: <http://www.portaleducacao.com.br/informatica/artigos/48358/google-analytics>. Acesso em: 3 de Jul. de 2017.

CUSTODIA, Henrique. **Two way data binding. Disponível em: <**https://medium.com/@henriquecustodia/two-way-data-binding-fd5d71712d28>. Acesso em: 10 de Out. de 2017.

DEVELOPER Survey Results 2017. Disponível em: <https://insights.stackoverflow.com/survey/2017>. Acesso em: 24 de Set. de 2017.

ECMAScript 2015 Language Specification. Disponível em: <http://www.ecma-international.org/ecma-262/6.0/> Acesso em: 22 de Set. de 2017.

ELLIOTT, Eric. **Top JavaScript Frameworks & Topics to Learn in 2017.** Disponível em : < https://medium.com/javascript-scene/top-javascript-*framework*s-topics-to-learn-in-2017-700a397b711> Acesso em: 10 de Out. De 2017.

GAMMA, Erich; VLISSIDIS, John; JOHNSON, Ralph; GAMMA, Richard Helm. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley Professional, 1994.

GARBADE, Michel. **15 JavaScript frameworks and libraries.** Disponível em: <https://opensource.com/article/16/11/15-javascript-*framework*s-libraries>. Acesso em: 11 de Out. de 2017.

GAZAROV, Petr. **What is an API? In English, please.** Disponível em: <https://medium.freecodecamp.org/what-is-an-api-in-english-please-b880a3214a82>. Acesso em: 3 de Jul. de 2017

GUIA JavaScript. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide> Acesso em: 22 de Set. de 2017.

HABERMAN, Josh. **React Demystified.** Disponível em: <http://blog.reverberate.org/2014/02/*react*-demystified.html>. Acesso em: 11 de Out. de 2017.

HARRINGTON, Chris. **React vs AngularJS – How the two Compare.** Disponível em: <https://www.codementor.io/*react*js/tutorial/*react*-vs-angularjs>. Acesso em: 22 de Set. De 2017.

HARRINGTON, Chris. **React vs AngularJS vs KnockoutJS: a Performance Comparison.** Disponível em: **<**https://www.codementor.io/chrisharrington/*react*-vs-angularjs-vs-knockoutjs-a-performance-comparison-85hwzepbz**>** Acesso em: 17 de Set. de 2017.

HISTORY of the Web. Disponível em: < https://webfoundation.org/about/vision/history-of-the-web/>. Acesso em: 10 de Out. de 2017.

HTML. Disponível em: <https://en.wikipedia.org/wiki/HTML>. Acesso em: 11 de Out. de 2017.

IN Depth Overview. Disponível em: <https://facebook.github.io/flux/docs/in-depth-overview.html#content>. Acesso em: 2 de Nov. De 2017.

KAPPERT, Lars. **ECMAScript 6 (ES6): What's New In The Next Version Of JavaScript.** Disponível em: < https://www.smashingmagazine.com/2015/10/es6-whats-new-next-version-javascript/ >. Acesso em: 10 de Out. De 2017

KAUFMAN, Amit. **Angular vs. React - the tie breaker.** Disponível em: <https://www.airpair.com/angularjs/posts/angular-vs-*react*-the-tie-breaker>. Acesso em: 22 de Set. de 2017.

KLAUZINSKI, Philip; MOORE, John. **Mastering JavaScript Single Page Application Development.** Birmingham Packt Publishing, 2016.

KRILL, Paul. **What's so special about Google's AngularJS.** Disnponível em: <https://www.infoworld.com/article/2612801/javascript/what-s-so-special-about-google-s-angularjs.html>. Acesso em: 17 de Set. de 2017.

LONG, Josh. **I Don’t Speak Your Language: Frontend vs. Backend.** Disponível em: <http://blog.teamtreehouse.com/i-dont-speak-your-language-frontend-vs-backend>. Acesso em: 9 de Out. De 2017.

MACGYVER. **Flux: Entenda como funciona a arquitetura Flux com React.** Disponível em: **<**https://tableless.com.br/flux-entenda-como-funciona-arquitetura-flux-com-*react*/ >. Acesso em: 19 de Set. de 2017.

MYINT, Win Nay. **TypeScript for Angular 2 - Part 1 (An Introduction).** Disponível em: <http://naywinmyint.com/typescript-for-angular-2-part-1/>. Acesso em: 2 de Nov. De 2017.

MONOBE, Felipe. **A Evolução do assíncrono no Node.js—Parte 3 (Promises).** Disponível em: < https://medium.com/@felipemonobe/evolucao-assincrono-nodejs-p3-60da38bb4dce>. Acesso em: 6 de Jul. de 2017.

NETSCAPE and Sun announce JavaScript, the open, cross-plataform object scriptinglanguage for enterprise networks and the internet. Disponível em: <https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html>. Acesso em: 3 de Jul. de 2017.

OSMANI, Addy. **Developing Backbone.js Applications.** O'Reilly Media; 1ª edição, 2013
SIMKHADA, Kumar. **Transitioning Angular 2 User Interface (UI) into *React*.** Helsinki Metropolia University of Applied Sciences, 2017.

PEYROTT, Sebastián. **7 Things you should know about WebAssembly.** Disponível em: <**https://auth0.com/blog/7-things-you-should-know-about-web-assembly/**>. Acesso em: 17 de Set. de 2017.

PROMISE. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>. Acesso em: 8 de Out. de 2017.

QUIN, Liam. **XLM Essentials.** Disponível em: <https://www.w3.org/standards/xml/core>. Acesso em: 8 de Nov. de 2017.

REACT. Disponível em: <https://*react*js.org/> Acesso em: 2 de Nov. de 2017.

REIS, André. **O que é um Front-End developer?** Disponível em: <http://www.matera.com/br/2016/02/04/o-que-e-um-front-end-developer/> Acesso em: 10 de Out. De 2017.

REIS, Vinicius. **Entenda de uma vez por todas o que é React.JS, Angular 2, Aurelia e Vue.JS.** Disponível em: < https://medium.com/by-vinicius-reis/o-que-e-*react*-ng2-auleria-vue-e34b0c77b5a1>. Acesso em: 19 de Set. De 2017.

TYPESCRIPT. Disponivel em: <https://www.typescriptlang.org/>. Acesso em : 3 de Jul. de 2017.

VLAD, V; FEDOR, L; SVIATOSLAV. A. **React vs AngularJS - A Popular JavaScript Library and a Powerful JavaScript Framework.** Disponível em: <https://rubygarage.org/blog/*react*-vs-angularjs >. Acesso em 9 de Set. de 2017.

VIANA, Daniel. **O que é front-end e back-end?.** Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-front-end-e-back-end/>. Acesso em 18 de Nov. de 2017.

WHAT Is a Front-End Developer?. Disponível em: <https://frontendmasters.gitbooks.io/front-end-handbook-2017/content/what-is-a-FD.html>. Acesso em: 17 de Out. De 2017.

WHAT is Back End programming? Disponível em: <https://help.codecademy.com/hc/en-us/articles/220958708-What-is-Back-End-programming->. Acesso em: 11 de Out. De 2017.

WHEELER, Ken. **Getting To Know Flux, the React.js Architecture.** Disponível em: <https://scotch.io/tutorials/getting-to-know-flux-the-*react*-js-architecture/>. Acesso em: 1 de Set. de 2017.

WODEHOUSE, Carey. **A Beginner's Guide to Back-End Development.** Disponível em: <https://www.upwork.com/hiring/development/a-beginners-guide-to-back-end-development/> Acesso em: 8 de Nov. De 2017.

WODEHOUSE, Carey. **Intro to APIs: What Are APIs and What Do They Do?.** Disponível em: <https://www.upwork.com/hiring/development/intro-to-apis-what-is-an-api/>. Acesso em: 12 de Nov. De 2017.

ZAKAI, Alon. **Why WebAssembly is Faster Than asm.js.** Disponível em: <https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/>. Acesso em: 22 de Set. De 2017.

ZANETTE, Alysson. **Framework x Biblioteca x API. Entenda as diferenças!** Disponível em: <https://becode.com.br/*framework*-biblioteca-api-entenda-as-diferencas/>. Acesso em: 6 de Out. de 2017.