INTERFACE TECNOLÓGICA

ANÁLISE DAS SEMELHANÇAS E DIFERENÇAS ENTRE UTILIZAR JSX OU JAVASCRIPT PURO AO CONSTRUIR INTERFACES COM REACT

ANALYSIS OF SIMILARITIES AND DIFFERENCES BETWEEN USING JSX OR PURE JAVASCRIPT WHEN CONSTRUCTING INTERFACES WITH REACT

Anderson de Sousa Pereira – anderson-sousa-pereira@hotmail.com
Erick Eduardo Petrucelli – erick.petrucelli@fatectq.edu.br
Faculdade de Tecnologia de Taquaritinga (FATEC) – SP – Brasil

RESUMO

O mercado tecnológico está em constante transformação. A renovação e a inovação são características marcantes no desenvolvimento *Web* e estar pronto para mudanças faz com que o imprevisível esteja distante. *Frameworks*, bibliotecas, paradigmas e conceitos novos surgem com uma frequência assustadora e manter-se informado é dever de todo bom desenvolvedor. Desta forma, o presente artigo objetivou abordar duas formas de construção de interfaces com a biblioteca React: a utilização do JSX ou instanciação de elementos com JavaScript puro. Para tal, foram comparadas as diferenças de sintaxe, questões de desempenho, conhecimentos prévios necessários e a produtividade oferecida no desenvolvimento, utilizando como metodologia a revisão bibliográfica através de livros, artigos e documentações oficiais, os quais agregaram à experiência vivenciada empiricamente com tais tecnologias. Ao final do artigo, baseando-se na argumentação decorrida, apresenta-se as considerações vislumbradas.

Palavras-chave: React. JSX. JavaScript. Comparativo.

ABSTRACT

The technological market is constantly changing. Renewal and innovation are striking characteristics in Web development and being ready for change makes the unpredictable to be distant. Frameworks, libraries, paradigms and new concepts emerge with a frightening frequency and keeping itself informed is the duty of every good developer. Thus, this article aimed to address two ways of constructing interfaces with the React library: the use of JSX or the instantiation of elements with pure JavaScript. To achieve this, the differences in syntax, performance, previous knowledge needs and the productivity offered in the development were compared, using bibliographical revision methodology through books, articles and official documentations, which added to the experience lived empirically with such technologies. At the end of the article, based on the argumentation, the considerations glimpsed are presented.

Keywords: React. JSX. JavaScript. Comparison.

1 INTRODUÇÃO

O JavaScript que conhecemos hoje evoluiu bastante para chegar em sua posição atual. De acordo com Rauschmayer (2014), a primeira versão da linguagem se chamava Mocha e foi criada em 1995 por Brendan Eich, a pedido da Netscape, empresa a qual este programador era funcionário. Muitas versões surgiram desde então, bem como paradigmas, *frameworks*, bibliotecas e diferentes conceitos foram sendo estabelecidos. E o que começou apenas como um protótipo feito por um programador em dez dias, hoje se tornou um ecossistema de desenvolvimento vívido, com uma gama gigantesca de fãs ao redor do globo.

O avanço de seu ecossistema proporcionou aos desenvolvedores cada vez mais versatilidade em suas mãos. Com tantas ferramentas à disposição, desenvolver para a *Web* passou a exigir atualização constante e muito bom senso. Analisar minuciosamente o cenário se tornou imprescindível. O que em um projeto poderia ser fabuloso, dependendo da necessidade e da finalidade, em outro não seria a melhor opção. E foi levando em conta a inevitabilidade de uma boa análise que este artigo foi pensado. Nos capítulos a seguir, serão abordadas as questões sobre o desenvolvimento das interfaces com a biblioteca React, utilizando a sua sintaxe JSX ou o JavaScript puro tradicional.

2 METODOLOGIA

A realização do presente trabalho se baseou metodologicamente principalmente em revisão bibliográfica dos assuntos abordados, focada principalmente em livros, artigos e documentações oficiais. Para tal, foi realizada uma pesquisa inicial para identificação e listagem de fontes relevantes, seguida de análise de materiais pertinentes ao assunto.

Durante a construção do texto, o processo empírico foi eventualmente empregado, através das experiências vivenciadas pelos autores sobre as tecnologias em questão, como mecanismo de corroboração dos conteúdos pesquisados.

3 REACT

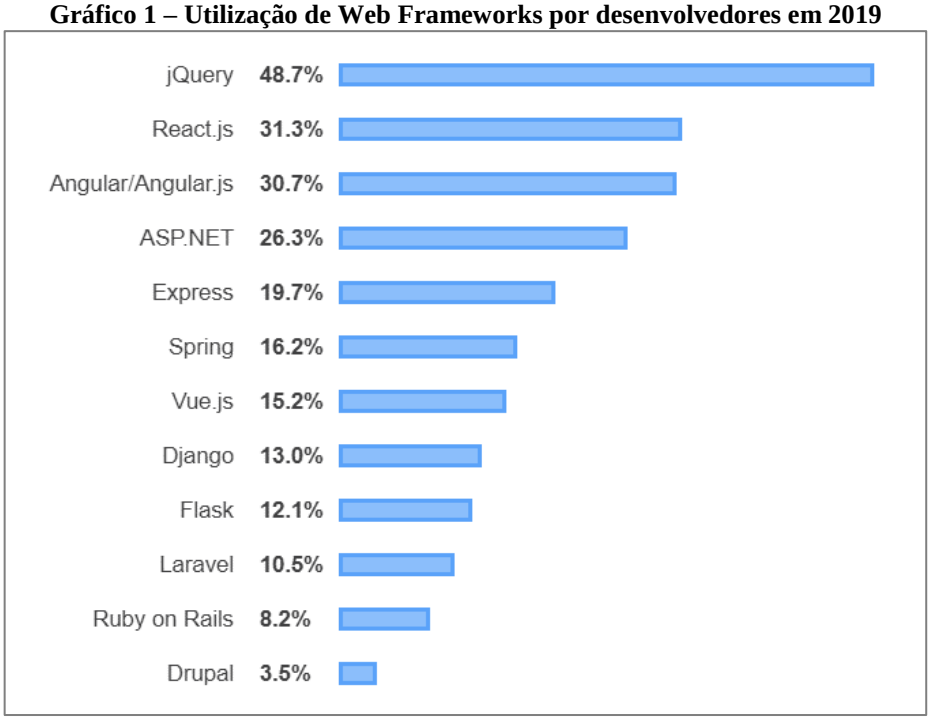
De acordo com Kostrzewa (2018) e Pandit (2018), o React foi criado em 2011 pela equipe de desenvolvedores do Facebook, como uma forma de resolver um problema que lhes causava considerável dor de cabeça: a necessidade frequente de atualização dos elementos que compõem a *timeline* da plataforma durante a navegação do usuário. Assim, desenvolveram uma

biblioteca cujo propósito principal era renderizar, de maneira mais eficaz, as mudanças ocorridas na *view*, ou seja, a camada visual da aplicação. O restante do modelo MVC (*Model-View-Controller*), arquitetura utilizada em aplicações *Web* e muito consolidada, se tornou responsabilidade de bibliotecas, *bundlers*, *transpilers*, *plug-ins* e ferramentas externas.

Foi apenas em 2013 que a equipe do Facebook decidiu tornar o React um projeto *open source* (GACKENHEIMER, 2015). Desde então, muitas coisas evoluíram, algumas graças à comunidade de desenvolvedores e outras à própria empresa detentora da rede social. Assim, mesmo sem ser um *framework* completo, tal biblioteca se transformou em um ecossistema de desenvolvimento de aplicações *Web* completo e consistente, a partir de diversas bibliotecas de apoio, algumas oficiais (do próprio Facebook) e outras criadas pela comunidade.

Além de ser um ecossistema de desenvolvimento *Web* consistente, tornou-se um dos mais conhecidos e utilizados, conforme o Gráfico 1 demonstra, extraído de pesquisa respondida por 63.585 desenvolvedores de todo o mundo, conduzida pela plataforma StackOverflow.

Gráfico 1 – Utilização de Web Frameworks por desenvolvedores em 2019



Fonte: StackOverflow (2019)

Há várias características do React frequentemente elogiadas, as quais serviram de inspiração para outros projetos, como: reatividade dos dados; eficiente implementação de DOM virtual; projeto derivado para dispositivos móveis, o React Native. Entretanto, uma das características do React é polêmica: a sintaxe conhecida como JSX.

3.1 JSX

JSX funciona como uma extensão para o ECMAScript (a especificação da linguagem JavaScript), gerando códigos JavaScript através de *transpilation*, oferecendo uma sintaxe similar ao XML/HTML. Possui algumas poucas diferenças na estruturação de *tags*, como, por exemplo, a substituição das propriedades *for* e *class*, por serem palavras reservadas do JavaScript, as quais precisam ter outros nomes nesta sintaxe. O seu princípio fundamental é juntar JavaScript e *layout* HTML no mesmo código (ANTONIO, 2015).

A similaridade com a linguagem de marcação HTML, mas dentro do próprio JavaScript, e ainda oferecendo algumas divergências, fez com que muitos desenvolvedores conservadores se sentissem incomodados. E, por um bom tempo, mesclar a estruturação do documento com o JavaScript foi alvo de críticas. Atualmente, a ideia é mais aceita e as discussões a respeito são menos frequentes (KOSTRZEWA, 2018).

Sendo o JSX uma extensão criada para utilização junto a *transpilers*, obrigatoriamente exige a presença do Babel ou outra ferramenta deste tipo para o trabalho de conversão em JavaScript tradicional. Desta forma, os exemplos de código JSX apresentados neste trabalho utilizaram o Babel 7.3.0 como *transpiler*.

4 JSX VS. JAVASCRIPT PURO

É comum, em pesquisas sobre React, encontrar termos como Webpack, Babel, ES6, JSX, GraphQL, Redux, dentre outros. Assim, é natural que os desenvolvedores sintam quase que uma obrigação em utilizar todas essas tecnologias no código com React. Porém, de acordo com referências como Facebook (2019b; 2019c), Ricoy (2018) e Nelson (2018), nenhuma delas é de fato um requisito. Como o React foi desenvolvido para que pudesse ser apenas um agente que manipula a camada *view*, todos esses termos são acréscimos que oferecem recursos adicionais à biblioteca base, mas não obrigatórios para o desenvolvimento React.

Sendo assim, as três únicas coisas que realmente são necessárias para desenvolver com a biblioteca são: um editor de texto qualquer, um navegador *Web*, e a biblioteca base do React importada na página. Com isto em mãos, o ambiente desenvolvimento já está preparado. Portanto, se as interfaces forem construídas com JavaScript puro, isto basta.

Já para a utilização do JSX, é necessário configurar o ambiente de desenvolvimento de forma mais complexa. De acordo com Buna (2016), além dos itens citados, é necessária a existência do *transpiler* que faça a conversão do código JSX em JavaScript puro. Este trabalho

não possui o foco em discorrer sobre tal configuração, uma vez que a referências como Facebook (2019a) já abordam detalhadamente como fazer.

Nos tópicos a seguir, detalha-se a comparação entre JSX e JavaScript puro.

4.1 Conhecimentos prévios necessários

Antes de começar a renderizar componentes React, existem alguns conceitos que devem ser entendidos, independentemente da abordagem utilizada. Com JavaScript puro, de acordo com Facebook (2019b), a criação é realizada através da função `createElement()`. Sua compreensão é de suma importância para a utilização da abordagem, mas, internamente em seu código, basta instanciar elementos usando JavaScript puro e ir montando uma hierarquia em memória, similarmente como os desenvolvedores já estão acostumados a trabalhar com a criação de elementos no DOM sem nenhuma biblioteca. Portanto, desenvolvedores JavaScript habituais podem iniciar rapidamente com React sem aprender uma nova sintaxe antes.

Já com o JSX, por ser uma sintaxe similar ao HTML, como descrito por Hudson (2016), é possível reaproveitar o conhecimento prévio sobre a linguagem de marcação, sendo que há algumas poucas diferenças entre as duas tecnologias, como a inserção de códigos JavaScript dentro da estruturação e algumas nomenclaturas de propriedades cujos nomes foram alterados, para que não houvesse incompatibilidade por conta de palavras reservadas. Ainda que de forma próxima ao HTML, é uma sintaxe adicional além do JavaScript, o que exige alguns conhecimentos prévios básicos antes da utilização.

4.2 Diferenças de sintaxe

Como já brevemente abordado, a sintaxe para se criar um elemento com JavaScript puro depende basicamente da função `createElement()`. De acordo com Facebook (2019b), esta recebe como parâmetro três entradas: o elemento a ser criado (ou seja, o nome da *tag* HTML desejada), os atributos que ele possui (ou `null`, caso não tenha atributos a informar) e seus elementos filhos (ou um texto literal, caso não tenha elementos filhos). A seguir, na Figura 1, podemos conferir um exemplo simples de um menu, criado através dessa abordagem.

Figura 1 – Código da estrutura de um menu em JavaScript puro

```
var rootElement = React.createElement("div", null, React.createElement("ul", null,
React.createElement("li", null, React.createElement("img", {
    src: ""
}), "Item 1"), React.createElement("li", null, React.createElement("img", {
    src: ""
}), "Item 2"), React.createElement("li", null, React.createElement("img", {
    src: ""
}), "Item 3"), React.createElement("li", null, React.createElement("img", {
    src: ""
}), "Item 4"), React.createElement("li", null, React.createElement("img", {
    src: ""
}), "Item 5")));
```

Fonte: Os autores (2019)

Por sua vez, considerando-se que o processo de "transpilação" com Babel foi previamente configurado corretamente, a abordagem com JSX é baseada simplesmente em escrever a estrutura de *tags* desejada de forma próxima ao HTML tradicional, diretamente na atribuição da variável que armazenará o componente construído. Na Figura 2, observa-se o mesmo exemplo descrito anteriormente, porém utilizando o JSX.

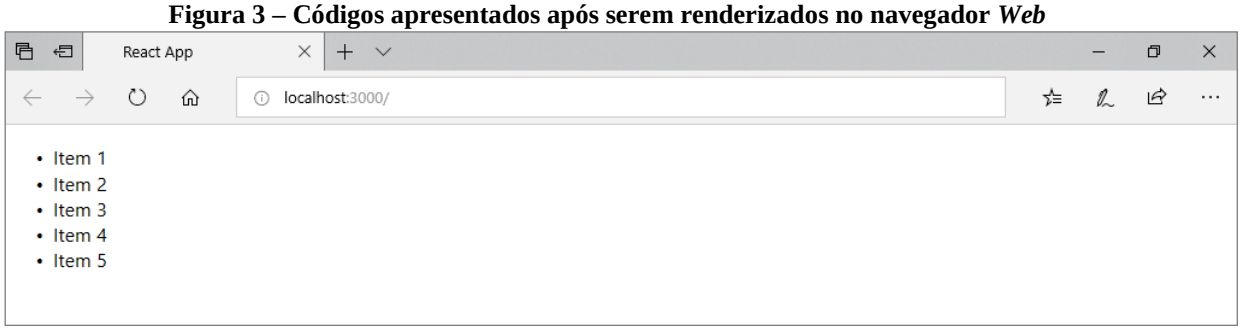
Figura 2 – Código da estrutura de um menu em JSX

```
var rootElement = (<div>
        <ul>
            <li><img src=""/>Item 1</li>
            <li><img src=""/>Item 2</li>
            <li><img src=""/>Item 3</li>
            <li><img src=""/>Item 4</li>
            <li><img src=""/>Item 5</li>
        </ul>
    </div>);
```

Fonte: Os autores (2019)

Como trata-se de componentes com o mesmo *layout* final, apenas escritos de forma diferente, na Figura 3 encontra-se o resultado retornado pelo navegador *Web* nos dois casos.

Figura 3 – Códigos apresentados após serem renderizados no navegador *Web*



Fonte: Os autores (2019)

Neste sentido, evidencia-se que a sintaxe JSX é consideravelmente mais legível, principalmente para desenvolvedores *Web* já habituados com HTML, enquanto a sintaxe com JavaScript puro se assemelha à manipulação direta do DOM e pode ser mais compreensível para desenvolvedores não acostumados com a estruturação dos *layouts* das páginas.

4.3 Produtividade no desenvolvimento

Preocupar-se com a produtividade é indispensável em qualquer tipo de trabalho. Cada linha de código escrita representa uma linha a mais que deverá ser testada e mantida. Portanto, escrever um código enxuto e legível é fundamental para a economia de recursos (BOSWELL; FOUCHER, 2012), o que o torna um fator relevante a considerar nesta comparação.

Já foi abordado no tópico anterior, que as sintaxes para o desenvolvimento com React, embora não apresentem grandes desafios, são diferentes em complexidade e dimensão. Enquanto JavaScript puro é uma sintaxe mais verbosa, a variação com JSX se mostra menos burocrática e mais intuitiva, ao menos na teoria, para a maioria dos desenvolvedores *Web*.

De acordo com Martin (2009), o nível de produtividade e de organização são proporcionais. Quanto mais limpo e organizado estiver o código trabalhado, melhor será a produtividade da equipe de desenvolvimento.

Portanto, ao comparar o código da Figura 1 com o código da Figura 2, fica evidente a diferença entre as duas abordagens no quesito limpeza de código. As funções encadeadas presentes na abordagem com JavaScript puro tendem a se tornarem confusas conforme aumenta o encadeamento dos elementos, algo natural em interfaces reais, com muitos elementos. Por outro lado, a disposição dos elementos em *tags* propiciada pelo JSX se mostra mais organizada e, portanto, tende a oferecer a melhor produtividade durante o desenvolvimento, além de proporcionar manutenibilidade mais facilitada.

4.4 Desempenho em execução

A *Web* 2.0 foi uma mudança gradativa na forma como a Internet opera. O termo retrata uma alteração de conceito gerada com o tempo, sem possuir qualquer correlação com mudanças realmente técnicas na forma como a *Web* funciona. Os produtos, que antes eram vistos e fornecidos através de pacotes, passaram a estar disponíveis através de serviços (O'REILLY, 2005). A preocupação com a velocidade de entrega de conteúdo foi se tornando cada vez mais

relevante, até se tornar imprescindível. Atualmente, um sistema *Web* que não ofereça bom desempenho dificilmente chamará a atenção e manterá usuários satisfeitos.

De acordo com Nielsen (1993), um único segundo é o tempo máximo para gerar resposta ao usuário sem que exista a crescente sensação de estar sendo interrompido. É dito também que é possível manter um usuário esperando em um sistema sem *feedback* por dez segundos.

Levando-se em consideração tal afirmação sobre a experiência do usuário em relação ao desempenho da aplicação, bem como o fato de que não é apenas a velocidade de transmissão de dados que pode impactar no desempenho, mas também o próprio tempo desprendido para renderização do resultado no navegador *Web*, evidencia-se a relevância de comparar o desempenho em execução para renderização entre ambas as abordagens.

Para tal, foi conduzido um teste simples de desempenho com cada tipo de sintaxe, utilizando os mesmos códigos apresentados anteriormente, com quatro chamadas consecutivas a cada um dos códigos, porém com o acréscimo de invocações à função `Date.now()` nativa do JavaScript, podendo-se assim calcular o tempo decorrido antes da função ser chamada, até o término da renderização. Na Tabela 1, é possível conferir os resultados, apresentados em milissegundos, obtidos para cada invocação realizada de renderização:

Tabela 1 – Tempo transcorrido a cada invocação de renderização (em milissegundos)

MOMENTO	JSX	JAVASCRIPT PURO
1ª invocação	1363.4350000065751	350.67500005243346
2ª invocação	1276.0399999679066	171.3549999985844
3ª invocação	1267.1750000445172	158.22000004118308
4ª invocação	1254.735000024084	164.844999962952

Fonte: Os autores (2019)

Analisando-se estes números, observa-se que tanto com JSX quanto com JavaScript puro, a primeira invocação do código para renderização demanda um tempo relativamente maior do que nas outras vezes. Isto pode ser explicado pelo fato do cache do navegador *Web* ainda não ter sido gerado nesta primeira execução. De acordo com Smith (1982), um cache tem a função de armazenar dados frequentemente utilizados, para assim fornecer um desempenho otimizado em execuções posteriores. Além disso, fica evidente que há um tempo natural para que o React construa o Virtual DOM e reflita sua estrutura no navegador *Web*, o qual precisa ser transcorrido mesmo com a implementação com JavaScript puro.

Quanto as invocações posteriores, observa-se que a versão com JavaScript puro é consideravelmente mais rápida do que a versão com JSX. Isto se deve ao fato de que o JSX precisa ser "transpilado" em JavaScript antes da execução, o que não ocorre quando o conteúdo já está informado como JavaScript puro.

5 CONSIDERAÇÕES FINAIS

Sabe-se que desenvolver para a *Web* requer muita pesquisa e reflexão, para proporcionar as melhores escolhas tecnológicas para cada projeto. Com tantas opções de tecnologias e diferentes abordagens em cada tecnologia, uma boa forma de se ter mais clareza é testando e comparando as opções existentes. Neste contexto, o presente artigo teve como objetivo a demonstração e o estudo de duas sintaxes possíveis para construção de interfaces de componentes com a biblioteca React.

Após a breve revisão sobre React e sua sintaxe JSX, foram realizadas comparações entre este e o JavaScript puro, abordando os conhecimentos prévios necessários, a sintaxe, a produtividade no desenvolvimento e o desempenho em execução, reunindo-se informações de diferentes referências, bem como realizando-se alguns testes de código próprios deste trabalho, sendo que ambas as abordagens podem entregar resultados similares.

Assim, é possível concluir que o objetivo do código a ser criado é decisivo para a escolha da abordagem. Quando o foco principal for a produtividade no desenvolvimento, a praticidade para manutenção ou reaproveitamento de conceitos conhecidos do HTML, a sintaxe JSX pode ser a melhor opção. Quando o foco for o melhor desempenho a qualquer custo, sem preocupar-se com os outros tópicos abordados, o JavaScript puro pode ser a escolha ideal. Além disso, quando não houver o interesse, ou a viabilidade técnica, de se utilizar um *transpiler*, evidencia-se que os outros fatores são ignorados e o JavaScript puro passa a ser a única opção.

REFERÊNCIAS

ANTONIO, C. **Pro React**: Build Complex Front-End Applications in a Composable Way With React. Apress, 2015.

BOSWELL, D; FOUCHER, T. **The Art of Readable Code**: Simple and Practical Techniques for Writing Better Code. Estados Unidos: O'Reilly Media, 2012.

BUNA, S. **React Succinctly**. Estados Unidos: [s.n], 2016. Disponível em: <www.syncfusion.com/ebooks/reactjs_succinctly>. Acesso em: 12 de abril de 2019.

FACEBOOK (2019a). **React**: Getting Started. React Docs, 2019. Disponível em: <reactjs.org/docs/react-api.html>. Acesso em: 13 de março de 2019.

FACEBOOK (2019b). **React Without ES6**. React Docs, 2019. Disponível em: <reactjs.org/docs/react-without-es6.html>. Acesso em: 10 de março de 2019.

FACEBOOK (2019c). **React Without JSX**. React Docs, 2019. Disponível em: <reactjs.org/docs/react-without-jsx.html>. Acesso em: 10 de março de 2019.

GACKENHEIMER, C. **Introduction to React**: Using React to Build scalable and efficient user interfaces.[s.i.]: Apress, 2015.

HUDSON, P. **Hacking with React.** 2016. Disponível em: <www.hackingwithreact.com/read/1/3/introduction-to-jsx>. Acesso em: 13 março 2019.

KOSTRZEWA, D. **Is React.js the Best JavaScript Framework in 2018?** 2018. Disponível em: <hackernoon.com/is-react-js-the-best-JavaScript-framework-in-2018-264a0eb373c8>. Acesso em: 12 março 2019.

MARTIN, R. **Clean Code:** A Handbook of Agile Software Craftsmanship. Estados Unidos: Prentice Hall, 2009.

NELSON, J. **Learn React's Fundamentals Without the Buzzwords?** 2018. Disponível em: <jamesknelson.com/learn-react-fundamentals-sans-buzzwords>. Acesso em: 12 março 2019.

NIELSEN, J. **Response Times**: The 3 Important Limits. 1993. Disponível em: <www.nngroup.com/articles/response-times-3-important-limits>. Acesso em: 10 de março de 2019.

O'REILLY, T. **What is Web 2.0**: Design Patterns and Business Models for the Next Generation of Software. 2005. Disponível em: <www.oreilly.com/pub/a/web2/archive/what-is-web-20.html#mememap>. Acesso em: 10 março 2019.

PANDIT, N. **What Is ReactJS and Why Should We Use It?** 2018. Disponível em: <www.c-sharpcorner.com/article/what-and-why-reactjs>. Acesso em: 12 março 2019.

RAUSCHMAYER, A. **Speaking JavaScript**: An In-Depth Guide for Programmers. Estados Unidos: O'Reilly Media, 2014.

RICOY, L. **Desmitificando React:** Uma Reflexão para Iniciantes**.** 2018. Disponível em: <medium.com/trainingcenter/desmitificando-react-uma-reflex%C3%A3o-para-iniciantes-a57af90b6114>. Acesso em: 13 março 2019.

SMITH, A. **Cache Memories**. ACM Computing Surveys (CSUR), New York, v.14, p. 473-530. 1982.

STACKOVERFLOW. **Most Popular Technologies**: Web Frameworks. Developer Survey Results, StackOverflow, 2019. Disponível em: <insights.stackoverflow.com/survey/2019#technology>. Acesso em: 13 abril 2019.