Desenvolvimento de softwares usando ferramentas front-end baseadas em componentes

Lucas Frota Marinho[1], Mario Angel Praia Garcia[2]

[1]Fundação Centro de Pesquisa e Inovação Tecnológica – FUCAPI
Caixa Postal 69075-351 – Manaus – AM – Brasil

[2]Departamento de Engenharia da Computação – Faculdade FUCAPI
Manaus, AM – Brasil

{lucasfrotam,marioangelpg}@gmail.com

Abstract. *This article has the objective of highlight the relevance and demonstrate the best metric out of three component-based front-end tools that best fit the user-proposed system. From tests and performance analysis performed with each of the tools, and their reuse by other systems, we get a validation about the use of components in front-end tools. Lastly, increasing the quality of the final product from the precepts defended by Malcolm Douglas McIlroy in his article published in Germany (1968).*

Resumo. *Este artigo tem como objetivo destacar a relevância e demonstrar a melhor métrica dentre três ferramentas front-end baseadas em componentes que se melhor adequam ao sistema proposto pelo usuário. A partir de testes e análises performáticas realizadas com cada uma das ferramentas, além da sua reutilização por outros sistemas, obtém-se uma validação a cerca do uso de componentes em ferramentas front-end. Por fim, aumentando a qualidade do produto final a partir dos preceitos defendidos por Malcolm Douglas McIlroy em seu artigo publicado na Alemanha (1968).*

1. Introdução

O final dos anos de 1950 foi a era da computação, principalmente para as instituições de pesquisas e universidades que puderam ter acesso a computadores, desde os mais simples aos mais completos em questão de performance, além de também estarem presentes na engenharia e nas ciências naturais. O mundo moderno tornou-se um grande aliado ao *software* ao ponto de a maior parte dos produtos eletrônicos incluir um sistema embarcado no qual o controla [Sommerville 2011].

Segundo Pressman (2011) ”devido ao avanço e a complexidade dos *softwares* a serem implementados, fez-se necessário desenvolver uma nova técnica que ficou conhecida como CBD (*component-based development*), desenvolvimento baseado em componentes, para economizar tempo, aumentar a produtividade, seguir padrões de qualidade e com isso ampliar a escalabilidade do seu produto final” [Pressman 2011]. Malcolm Douglas McIlroy, em 1968, foi a primeira pessoa a falar sobre este método de desenvolvimento em seu artigo *Mass produced software components* [McIlroy 1968], publicado na Alemanha, seu foco era em uma boa técnica, tornando eficaz o

desenvolvimento e o reuso do componente com o intuito de proporcionar aos desenvolvedores a possibilidade de escolher quais componentes utilizar conforme as suas necessidades

Para o desenvolvimento do *front-end* de aplicações Web existem uma série de ferramentas disponíveis no mercado para otimizar seu tempo de trabalho. Para CSS existem pré-compiladores como SASS5 e LESS6 que ajudam na reutilização de CSS. Para HTML, uma serie de *frameworks* dentre eles o Bootstrap, que é o mais conhecido. Logo, para a otimização do JavaScript foi criado o Angular, um *framework*, e o React, uma biblioteca. Ambos têm a mesma finalidade, porém trabalham de formas distintas com seus pontos positivos e negativos dependendo da aplicação proposta. Logo, neste artigo será analisado e discutido o desempenho de três ferramentas *front-end* em um mesmo cenário de teste para que se obtenha uma melhor métrica a cerca da ferramenta que se melhor adequa ao sistema proposto pelo usuário.

2. Fundamentação Teórica

Inicialmente, Malcolm Douglas McIlroy, em 1968, em seu artigo *Mass produced software components* foi pioneiro sobre componentização no desenvolvimento de *software* e defendeu o reuso de *software* como sendo o uso de componentes de *software* (produtos de prateleira) como blocos para a construção de grandes sistemas, assumindo que esses componentes são constituídos de um código-fonte. Em 1980 foi estabelecido o primeiro projeto de pesquisa universitário no tema de reutilização, na Universidade da Califórnia, coordenado por Peter Freeman. Ele definiu reuso como o uso de produtos não executáveis para o desenvolvimento de *software*, indicando que apenas usar o código-fonte não era suficiente, já que tal prática ainda exige muito esforço dos desenvolvedores nas atividades de análise e projeto, cujos produtos também podem ser reutilizados.

Em meados de 2000, Werner e Braga, ambas professoras da UFRJ, apresentam outra definição bastante interessante de que ”os componentes reutilizáveis são artefatos autocontidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras em conformidade com um dado modelo de arquitetura de *software*, documentação apropriada e um grau de reutilização definido”[Werner e Braga 2000]. Nesta definição, é evidenciado que o componente desempenha de forma completa a função a que se destina e claramente identificável de que o componente é facilmente localizável. A questão das interfaces também é ressaltada visto que elas são cruciais para o mecanismo de composição de componentes.

A reutilização de componentes, com seu fluxo de criação representado na Figura 1 [Werner e Braga 1999], tem sido encarada na Engenharia de *software* como uma opção na tentativa de otimizar o processo de desenvolvimento de sistemas. Simples casos demonstram essa necessidade, como quando encontramos uma solução para resolver um problema, replicamos para situações similares. No cenário da programação quando o trecho de código usado para solucionar uma adversidade, por exemplo, é utilizado de maneira exaustiva, ele se torna um padrão, aceito, generalizado e já validado pelos funcionários. Isso não é característica da implementação de *software*s em si, mas em qualquer situação que o problema que já tenha ocorrido seja qual for a área e se obteve uma solução eficaz, torna-se provável uma reexecução em uma nova ocorrência.

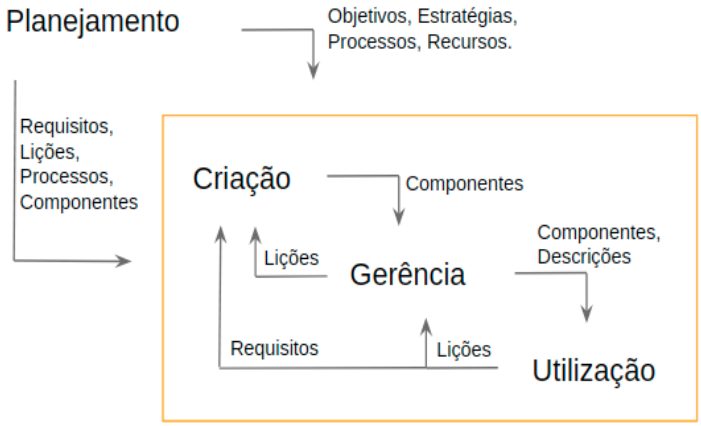


Figure 1. Fluxo para criação de um componente reutilizável

Nesse processo de planejamento e criação de componentes reutilizáveis é preciso ter atenção em fatores relacionados aos processos e atividades básicas que compõem o ciclo de vida do *software* como descrito na norma NBR ISO/IEC 12207 [ABNT 2009] na qual é um modelo de referência para capacitação de processos, além de ser uma norma internacional utilizada por muitas organizações para alcançar um diferencial competitivo. Tal norma define um *framework* para os processos de um *software*, os quais serão aplicados nas atividades de desenvolvimento, operação e manutenção de produtos, logo, consegue estabelecer uma arquitetura de alto nível para o ciclo de vida do *software*. Vale ressaltar, que essa diretriz não possui nenhuma ligação com métodos, métricas ou tecnologias quando aplicada, consequentemente pode ser usada com qualquer linguagem de programação. Essa flexibilidade é um fator importante pois seu foco é em 'o que fazer' e não 'como fazer'.

Os processos são agrupados, por uma questão de organização, de acordo com a sua natureza e o seu objetivo principal é de manter a estabilidade do *software*. Esse agrupamento resultou em 3 diferentes classes de processos que são: processos fundamentais, processos de apoio e processos organizacionais. A execução desses processos citados vai depender diretamente de como está o nível de conhecimento do time em relação a cada etapa, podendo ser descartada alguma delas [Lima 2014].

Os processos fundamentais são as atividades principais do ciclo de vida de um *software*, além de ser a base para que seja desenvolvido e executado. Dentre esses processos estão: aquisição, fornecimento, desenvolvimento, operação e manutenção. Já os processos de apoio são usados para garantir a qualidade do *software*, porém não são primordiais, neste caso, a documentação, gerência de aplicação, garantia de qualidade, verificação, validação, revisão conjunta, auditoria e resolução de problemas. Por fim, os processos organizacionais possuem a finalidade de organizar, implementar e controlar uma estrutura de processos sendo a gerência, infraestrutura, melhoria e treinamento.

Nos processos de apoio um dos pontos é a documentação, instrumento importante para o desenvolvimento baseado em componentes, visto que sua composição pode ser realizada de forma inteligível, logo podendo ser utilizada como referência para o uso correto do componente. Vale ressaltar que além de documentar o sistema, também é utilizada por futuros programadores, que realizarão a manutenção no produto, para que possam entender todo o fluxo do desenvolvimento. Além disso, diversos

frameworks e linguagens de programação atualmente usam somente sua documentação no seu site oficial para passar toda a sua metodologia de desenvolvimento, como é o caso do Angular, React, Vue, etc.

Para Parnas (1995), a documentação deve ser tida como pelo menos tão importante quanto o *software*, "se existe uma boa documentação, o produto de *software* pode ser revisado e substituído com relativa facilidade; sem uma boa documentação, o produto de *software* tem seu valor questionável a longo prazo"[Parnas 1995]. Uma boa documentação, bem elaborada e feita de forma legível pode trazer muitos benefícios como:

- Redução de tempo e esforço despendidos na construção do *software*.
- Facilidade de localização das informações e de compreensão das estruturas do *software*.
- Permite que atividades, como modificação e demonstração, possam ser realizadas sem maiores dificuldades.
- Facilidade e eficiência no uso do *software* pelos usuários.
- Permite que pessoas que não tiveram participação no projeto ou na construção do *software* tenham noção de como foi desenvolvido, além de como usar o componente de forma adequada ou simplesmente aproveitar a experiência que nela foi retratada

Segundo Phoha (1997), "a padronização da documentação pode reduzir o tempo e o esforço despendido na construção do *software*, aumentar sua portabilidade e auxiliar os usuários na sua compreensão"[Phoha 1997]. Um modelo americano para documentação científica bastante usado chamado ANSI/ANS 10.3 *Standard for Documentation of Computer Software* [ANSI 1995], no entanto não existem padrões universais aceitos para formatar uma documentação de uma aplicação. Este padrão faz recomendações quanto ao conteúdo e categorias propostas na documentação. Dentre esses padrões estão:

1. Resumo: Contém um resumo dos principais propósitos e capacidades do componente.
2. Informação da Aplicação: Resume o problema que o componente se propõe a solucionar e específica como utilizá-lo.
3. Definição do Problema (Definição Funcional): Esta categoria descreve todos os modelos matemáticos e algoritmos utilizados no programa. Também contém uma descrição detalhada de como o problema pode ser resolvido pelo componente, incluindo limites de capacidade e processamento de dados.
4. Informação do Programa: Relata a linguagem em que o componente foi desenvolvido, detalhes relativos à sua portabilidade, descreve os tipos de dados utilizados, os requisitos de implementação e os comandos para o uso adequado deste componente, entre outras informações.

Com o surgimento dessa ideia de componentes reutilizáveis, ciclo de vida, reutilização de código e até mesmo o aquecimento do mercado para entregas contínuas com prazos curtos, diversas técnicas de programação mais complexas e trabalhosas foram menos utilizadas como a Arquitetura Orientada a Serviços, a Programação Orientada a Objetos e a Programação Orientada a Aspectos [Freeman 2018]. Muito se

discute atualmente sobre essas metodologias pois toda tecnologia tem seu aspecto positivo em relação a outra, nenhuma supre totalmente tudo o que outra oferece.

Entretanto, com o aquecimento do mercado para entregas contínuas com prazos curtos, foi desenvolvido o AngularJS em 2009, que segundo Freeman (2018), "é o *framework* pioneiro com uma documentação marcante para desenvolvedores seguirem passo a passo na implementação do *software*, focado diretamente na metodologia de desenvolvimento baseado em componentes e de código aberto em JavaScript mantido pelo Google que auxilia, principalmente, na execução de *single-page applications*"[Freeman 2018]. Esta ferramenta foi a pioneira neste segmento de componentização, o que fez com que passasse por diversas mudanças em suas versões seguintes. Seu ponto mais positivo é incluir diversas funções, que possivelmente faria o desenvolvedor buscar bibliotecas de terceiros, de forma nativa. Entretanto, isso também traz aspectos negativos como o tamanho da aplicação final que atualmente é o principal obstáculo do Angular [Freeman 2018].

O Angular adota o padrão semelhante ao MVC (*Model View Controller*) [Freeman 2018] para estruturar qualquer aplicação da sua natureza. Os dados para apresentar ao usuário correspondem ao modelo em um projeto Angular, que, em sua maior parte, é composto de dados puros e é representado por meio de objetos JSON. Tal ferramenta adota essa estratégia pois esclarece a separação de responsabilidades entre as diversas partes de sua aplicação, com estrutura e padrões bem definidos.

Com o passar do tempo houve o surgimento de outras linguagens para esse mesmo segmento de desenvolvimento baseado em componentes, neste caso, o React, que segue a mesma linha de ter código aberto em JavaScript porém foi criado pelo Facebook. Quando comparado ao Angular leva vantagem em diversos pontos, dentro de suas limitações visto que é uma biblioteca e não um *framework*. Segundo Anthony et al. (2017), "o React trabalha com virtual-DOM, que é uma técnica de manipular o objeto e não o DOM, dessa forma quando o objeto é atualizado um algoritmo calcula a diferença entre a virtual-DOM e DOM real, alterando somente os pedaços de DOM que estiverem diferentes, logo, isso o torna favorito entre as tecnologias de *front-end*"[Anthony et al. 2017].

Entre as principais características do React estão:

- Declarativo: projeta diferentes páginas para cada estado da aplicação, que serão renderizadas e atualizadas de forma eficiente.
- Baseado em componentes: componentes de compilação, que gerenciam seu próprio estado, e os estruturam juntos nas situações mais complexas.
- Mantem uma representação interna da página corrente (virtual DOM), que processa apenas os elementos alterados.

Apesar de Angular e React parecerem bem diferentes, ambos podem executar a mesma função de criar um sistema simples, limitado, ou um mais robusto com diversos usuários, tudo depende da estrutura proposta para desenvolvimento. O Angular é um *framework* que tem uma estrutura mais completa e que favorece a construção de sistemas maiores, principalmente se o sistema proposto for um ERP (*Enterprise Resource Planning*) que possivelmente abrange módulos, muitas dinâmicas de permissões e validações, mas isso não significa que o React com seu jeito *clean code*

não atenda aos requisitos desejados e muito menos se torne inferior às demais tecnologias.

Atualmente em aplicações *low memory* para dispositivos com baixo nível de hardware, o React é a primeira opção dos desenvolvedores, visto que seu *bundle*, que é o tamanho do pacote final gerado para a aplicação, é menor que de uma aplicação feita em Angular [Anthony et al. 2017]. Na seção de resultados será apresentado por meio de tabelas e gráficos testes de performance dessas duas tecnologias diante de uma arquitetura em JavaScript, além do tempo de renderização de páginas web em cada uma das linguagens utilizadas e até mesmo uma média do tempo de desenvolvimento para cada caso utilizando uma amostragem de desenvolvedores.

3. Metodologia

O primeiro momento se destina, conforme apresentado na seção anterior, a um estudo sobre a componentização e sua evolução. Dentro desse contexto, fez-se uma análise da norma NBR ISO/IEC 12207 [ABNT 2009] que trata sobre ciclo de vida de um *software* no qual está diretamente ligado ao desenvolvimento baseado em componentes, visto que um componente pode ou não ter um ciclo de vida curto para reutilização dependendo de como foi implementado.

Dentro acordo com essa norma, um dos processos engloba a documentação de uma aplicação e dentro desse âmbito fez-se uma análise sobre as tecnologias de desenvolvimento baseado em componentes que mais são exigidas pelo mercado profissional atualmente, pois grande parte dessas tecnologias investem em uma documentação ampla ao passo de que o desenvolvedor não precise de outras fontes para construir sua aplicação.

Para atingir o objetivo do trabalho de destacar a relevância e demonstrar a melhor métrica dentre três ferramentas front-end baseadas em componentes que se melhor adequam ao sistema proposto pelo usuário foram feitos testes de carga, ou seja, testes nos quais o componente proposto recebe um *array* de valores padrão para que seja analisado o seu comportamento diante de cada métrica, além de testes de performances para medir seu tempo de renderização até que a aplicação final seja totalmente visível para o usuário. Outro ponto importante que deve-se ressaltar é a respeito do tempo de desenvolvimento para cada componente em sua respectiva ferramenta proposta, além do grau de complexidade e a curva de aprendizado de cada desenvolvedor. Logo, será esclarecido a usabilidade (capacidade de aprendizado, tempo de desenvolvimento), custo computacional, eficiência (desempenho, tamanho) e outros aspectos exclusivos de cada ferramenta *front-end*.

4. Resultados

Inicialmente foi analisado o tempo de desenvolvimento de um *grid* comum, no qual recebe um *array* de dados, por 3 desenvolvedores nos quais nunca tiveram contato com nenhuma dessas tecnologias, então foi proposto para cada um que implementasse esse *grid* nas linguagens propostas, neste caso, Angular, React e JavaScript, da forma que o código ficasse o mais legível possível como representado na tabela 1. Além dessa medida de tempo de desenvolvimento, também foi avaliado a curva de aprendizado de cada desenvolvedor durante todo esse processo de desenvolvimento.

Tabela 1. Comparação geral do tempo de desenvolvimento

	Pessoa 1	Pessoa 2	Pessoa 3	Nível de dificuldade
JavaScript	2 horas e 47 minutos	2 horas e 58 minutos	3 horas e 12 minutos	Alta
Angular	1 hora e 43 minutos	1 hora e 30 minutos	1 hora e 56 minutos	Médio
React	2 horas e 10 minutos	1 hora e 23 minutos	1 hora e 6 minutos	Médio

Como visto na tabela acima, as maiores dificuldades encontradas foram em se adaptar aos padrões do JavaScript, visto que o tempo de desenvolvimento sem *framework*/biblioteca foi o maior de todos. Consequentemente, os tempos com a ajuda das ferramentas Angular e React foram menores, pois o nível de conhecimento do ambiente proposto era maior do que inicialmente.

Segundo Freeman (2018), "o Angular tem uma estrutura que faz com que o desenvolvedor tenha que se adequar rigidamente a uma maneira específica de programação"[Freeman 2018]. O Angular impõe suas regras e todas estão descritas na sua documentação o que consequentemente faz com que o seu tempo de desenvolvimento e aprendizagem seja maior que o do React, que é mais fácil de aprender, pois existem conceitos e estruturas menos complexas para o entendimento dos desenvolvedores devido à sua natureza fracamente opinativa. O JavaScript [Duckett 2016] por ter sido a linguagem na qual os desenvolvedores começaram todo o aprendizado logo levou mais tempo para o entendimento de cada um, além da sua estrutura exigir mais código. Consequentemente, levou mais tempo para a adaptação de estrutura de código ao *grid* proposto.

Outro ponto analisado foi o desempenho de cada aplicação no *browser* Google Chrome. Foi feito uma carga de um *array* com 500 posições, neste caso, gerando um *grid* de 500 linhas. O algoritmo de teste calcula o tempo desde o início da geração desse *array* de valores até total renderização do conteúdo no *browser*, ou seja, até que tudo esteja visível para o usuário. Para cada execução foi medido esse intervalo de tempo e esse algoritmo de teste foi executado 3 vezes. Por fim, foi feito uma média dos resultados encontrados sendo representada na tabela 2.

Tabela 2. Comparação de tempo de renderização

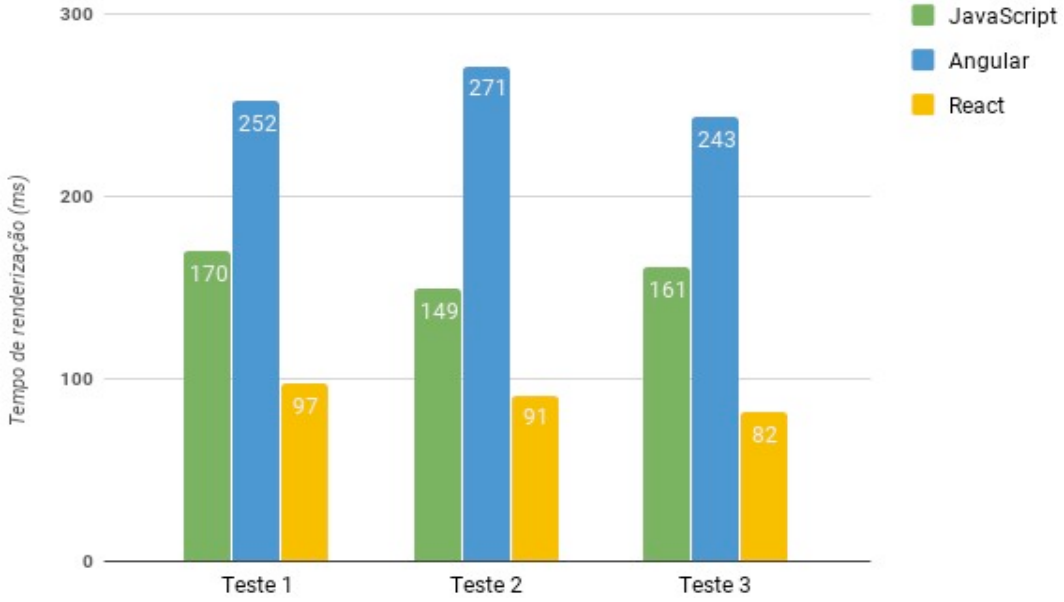
	Teste 1	Teste 2	Teste 3	Tempo Médio

JavaScript	170 ms	149 ms	161 ms	160 ms
Angular	252 ms	271 ms	243 ms	255 ms
React	97 ms	91 ms	82 ms	90 ms

De acordo com a tabela 2, é possível observar que o React é o menor pelo fato de ser apenas uma biblioteca, logo, seu *bundle* é menor então o tempo de renderização se torna menor. O Angular, por ser um *framework*, carrega diversos módulos dentro do seu *bundle*, que muitas das vezes nem são usados pelo desenvolvedor, o que o torna mais lento no processo de renderização, porém mais completo que o React no quesito de funcionalidades nativas, que neste caso, o React precisaria de bibliotecas de terceiros para atender algumas demandas. Além dessas duas ferramentas, também é possível perceber que o JavaScript ficou em segundo lugar, principalmente pelo fato de ter que gerar muitas linhas de código para conseguir executar todo o *grid* proposto.

Para melhor visualização dos resultados, foi gerado o gráfico conforme mostrado na figura 2 com cada tempo e sua respectiva linha colorida de acordo com as 3 medições feitas. Com isso foi possível elaborar este gráfico de colunas com as situações propostas. Um grande ponto a destacar é que todos os testes foram feitos no *browser* Google Chrome, ou seja, caso sejam executados em outros *browsers* haverá uma leve mudança de valores, visto que cada um interpreta JavaScript de uma maneira, neste caso o Google Chrome usa o Blink [Buckler 2013], logo isso influencia diretamente no tempo de execução e renderização de cada página.

Figura 2. Desempenho/renderização da aplicação em cada situação proposta



A forma como cada ferramenta lida com a manipulação da DOM (*Document Object Model*) [Anthony et al. 2017], que é a interface visual gerada pelo HTML da aplicação, influencia muito no desempenho final do *software* como pode ser observado no gráfico confome mostrado na figura 2. O Angular manipula a DOM real, enquanto o React, como dito anteriormente, cria e manipula uma DOM virtual evitando que simples mudanças de comportamento da aplicação sejam transmitidas para DOM real, o que geraria uma renderização de toda a página, logo influenciando diretamente na performance do *software*. A DOM virtual consegue coletar várias mudanças e aplicá-las de uma só vez, no entanto também trata simples mudanças de forma separada executando-as antes mesmo da renderização de toda a páginia evitando o acúmulo de mudanças na fila de renderização.

A quantidade de código que é carregado pelo *browser* também interfere no desempenho de uma aplicação. Neste caso, o Angular [Freeman 2018] tem uma estrutura mais padronizada ao ponto de forçar o desenvolvedor a seguir seu padrão de codificação e até mesmo suas metodologias de design. Já o React [Anthony et al. 2017], por ser uma biblioteca, contém uma coleção de classes e métodos que podem ser estendidos de outras bibliotecas de terceiros e reutilizadas. O JavaScript é o que mais tem linhas de códigos quando esse é o ponto de comparação, visto que tudo nele é feito de forma mais 'braçal', nele é preciso implementar funções nas quais a maioria já estão dentro do pacote do Angular ou React, porém um grande fator de usar somente o JavaScript é que não há necessidade de adicionar código extra para a aplicação renderizar em algum *browser* específico, visto que todos interpretam JavaScript nativamente.

5. Conclusão

O desenvolvimento deste trabalho possibilitou uma análise mais profunda sobre aplicações web e suas ferramentas de *front-end*. As diferenças encontradas nos resultados a partir dos testes feitos são significativas, visto que para a mesma implementação houveram diferentes resultados de acordo com o *framework* ou biblioteca usada. Além disso, foi possível observar o desempenho dessas aplicações, como tempo de renderização, e também o tempo de desenvolvimento usando cada ferramenta com um cenário de três desenvolvedores.

Foi identificado o quanto a manipulação da DOM atua diretamente no desempenho final da aplicação. O React se mostrou mais eficaz em todos os testes feitos pelo fato de ser apenas uma biblioteca e trabalhar com DOM virtual. O JavaScript mostrou um desempenho superior ao Angular, visto que os *browsers* modernos renderizam essa linguagem de forma nativa sem precisar de ferramentas externas para conversão de alguma base de código, como representado no gráfico conforme mostrado na figura 2, porém por não possuir funções que podem ser estendidas, como por exemplo uma biblioteca que aplica uma máscara em um *input* de valores na moeda Real que faz com que sua aplicação tenha muitas linhas de código pois toda essa tratativa, como neste caso, precisa ser feita no corpo do código, o que torna um código fonte grande e consequentemente um dos principais fatores que podem atrapalhar na manutenção de um *software* de larga escala. O Angular, apesar de ser categorizado como mediano de acordo com os testes elaborados, é uma das ferramentas mais completas por trazer muitas funções nativas, o que faz com que o *software* tenha pouca ou até mesmo nenhuma dependência de biblioteca de terceiros.

O reuso de código e uso de ferramentas que trabalham com componentização têm sido debatidos por diversas organizações como a solução para uma entrega mais rápida, com os requisitos atendidos e planejados de acordo com o cliente. Logo, os desenvolvedores de linguagens mais recentes criam especificações de componentização de acordo com as suas necessidades. A documentação de diversas linguagens, além das já citadas anteriormente, assim como as VueJS, SvelteJS e LitElement por exemplo, já contemplam essa forma de abordagem, ensinando técnicas de desenvolvimento e focando sempre em componentizar tudo que for possível para facilitar e potencializar a velocidade de entrega do produto final.

Referências

ABNT (2009). Norma NBR ISO/IEC 12207. https://www.abntcatalogo.com.br/norma.aspx?ID=38643. Acessado em: 27/08/2019.

ANSI (1995). *Documentation of computer software*. American Nuclear Society.

Werner, C., Braga, R. (1999). Infra-estrutura de reutilização baseada em modelos de domínio. https://www.researchgate.net/publication/232219627_Odyssey_Infraestrutura_de_Reutilizacao_baseada_em_Modelos_de_Dom'inio. Acessado em: 06/09/2019.

Werner, C., Braga, R. (2000). Desenvolvimento baseado em componentes. https://www.researchgate.net/publication/42799911_Desenvolvimento_baseado_em_Componentes. Acessado em: 06/09/2019.

Lima, A. S. (2014). Norma NBR ISO/IEC 12207. http://micreiros.com/norma-nbr-isoiec-12207/. Acessado em: 08/09/2019.

McIlroy, M. D. (1968). *Mass produced software components*. https://www.cs.dartmouth.edu/ ~doug/components.txt. Acessado em: 16/08/2019.

Parna, D. (1995). *Functional documents for computer systems*. Elsevier.

Phoha, V. A. (1997). *A standard for software documentation*. IEEE Computer Society.

Pressman, R. (2011). Engenharia de Software: uma abordagem profissional. Pearson Makron Books. 7º ed.

Sommerville, I. (2011). Engenharia de software. Pearson Prentice Hall. 9º ed.

Buckler, C. (2013). *Blink: Chrome's New Rendering Engine*. https://www.sitepoint.com/blink-rendering-engine-google-chrome/. Acessado em: 28/08/2019.

Anthony, A. Nathaniel, M. Lerner, A. (2017). *Fullstack React: The Complete Guide to ReactJS and Friends*. Fullstack.io. 1º ed.

Freeman, A. (2018). *Pro Angular 6*. Apress. 3º ed.

Duckett, J. (2016). Javascript & Jquery desenvolvimento de interfaces web interativas. Alta Books. 1º ed.