FACULDADES INTEGRADAS DE CARATINGA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

FRAMEWORKS DE SINGLE PAGE APPLICATION ANGULARJS E EMBER.JS – UMA ANÁLISE DE QUALIDADE INTERNA A PARTIR DA ISO/IEC 9126

ROMÁRIO JOSÉ HUEBRA HENRIQUE

CARATINGA

2015

ROMÁRIO JOSÉ HUEBRA HENRIQUE

FRAMEWORKS DE SINGLE PAGE APPLICATION ANGULARJS E EMBER.JS – UMA ANÁLISE DE QUALIDADE INTERNA A PARTIR DA ISO/IEC 9126

Monografia apresentada ao Curso de Ciência da Computação do Instituto Doctum de Educação e Tecnologia como requisito parcial para obtenção do título de Bacharel em Ciência da Computação orientada pela Professor Glauber Luiz da Silva Costa.

CARATINGA

2015

FACULDADES INTEGRADAS DE CARATINGA

TRABALHO DE CONCLUSÃO DE CURSO

TERMO DE APROVAÇÃO

TÍTULO DO TRABALHO

FRAMEWORKS DE SINGLE PAGE APPLICATION ANGULARJS E EMBER.JS - UMA ANÁLISE DE QUALIDADE INTERNA A PARTIR DA ISO/IEC 9126

por

Romário José Huebra Henrique

Este Trabalho de Conclusão de Curso foi apresentado perante a Banca de Avaliação composta pelos professores Glauber Luiz da Silva Costa, Gilberto Pacheco e Wanderson N. Miranda, às 19 horas do dia 14 de dezembro de 2015 como requisito parcial para a obtenção do título de bacharel. Após a avaliação de cada professor e discussão, a Banca Avaliadora considerou o trabalho aprovado, com a qualificação: ÓTIMO.

Trabalho indicado para publicação: (X)SIM ( )NÃO

Caratinga, 14 de dezembro de 2015

Professor Orientador e Presidente da Banca

Professor Avaliador 1

Professor Avaliador 2

Aluno(a)

Coordenador (a) do Curso

AGRADECIMENTOS

Primeiramente agradeço aos meus pais, que sempre confiaram em mim, estiveram do meu lado desde sempre, em todos momentos, me dando uma oportunidade que eles não tiveram. Se sou alguém hoje, devo isso a eles.

Agradeço também a minha namorada pela compreensão, incentivo e ajuda, e também a toda família dela que sempre me apoiou e me ajudou quando foi preciso.

Agradeço meu orientador Glauber, que sempre se mostrou disposto a ajudar, disponibilizando seu tempo para criação desse trabalho, sem ele não seria possível criar esse trabalho com qualidade.

Agradeço aos meus professores que me mostraram o caminho do conhecimento e o seu poder para transformar vidas.

Agradeço ainda a todos os meus colegas de sala que mesmo com grande bom humor sempre nos ajudamos uns aos outros.

RESUMO

A *web* é uma área de grande importância para o desenvolvimento e utilização de *software* atualmente. Essa plataforma hoje possui vários modelos de *software*, com o objetivo de melhorar a qualidade do *software* para o usuário, um desses modelos é o *single page application*(SPA). Com o modelo SPA é possível construir aplicações com as características de aplicações nativas e aplicações *web* tradicionais. Para facilitar a criação de aplicações nesse modelo e reduzir o retrabalho começaram a surgir *frameworks*, que auxiliam os desenvolvedores durante a elaboração de uma *single page application*.

Uma grande variedade de *frameworks* gera dificuldades aos desenvolvedores que querem escolher uma ferramenta de qualidade, levando isso em consideração, esse trabalho visa avaliar os dois principais *frameworks* de *single page application* do mercado, o AngularJS e o Ember.js, utilizando como base para avaliação as métricas de qualidade interna da ISO/IEC 9126-3.

Para realização dos testes para obtenção dos resultados, foram selecionadas 10 métricas da ISO/IEC 9126-3, as métricas foram selecionadas de acordo com o objeto de estudo, algumas métricas foram aplicadas ao *framework*, mas como o mesmo não é um *software* executável, foi usado também uma aplicação para outras métricas.

Dessa forma foi possível concluir que o AngularJS se saiu melhor em 7 métricas, o Ember.js se demostrou mais eficiente em 2 métricas e houve empate e uma métrica. A partir dos resultados foi possível concluir que o AngularJS é melhor planejado, usando menos recursos técnicos em relação ao Ember.js. Mesmo assim esse trabalho não chega a um veredito final, ficando por conta do desenvolvedor analisar os resultados de acordo com os requisitos de seu *software*.

Palavras-chave: framework, qualidade, ISO/IEC 9126, métrica, desenvolvimento.

ABSTRACT

The web is an area of great importance for the development and utilization of current software. This platform now has a range of software, in order to better the quality of the software to the user, one of these models is the single page application (SPA). With the SPA model is possible to build applications with the characteristics of native and traditional web applications. To facilitate the creation of applications that model and reduce rework began to emerge frameworks that help developers during the development of a single page application.

A wide variety of frameworks creates difficulties for developers who want to choose a quality tool, taking this into consideration, this study aims to evaluate the two main frameworks single page application market, the AngularJS and the Ember.js, using as a basis for evaluating the internal quality metrics ISO/IEC 9126-3.

For carrying out the test to obtain the results, we selected 10 metric of the ISO/IEC 9126-3, the metrics were selected according to the object of study, some metrics were applied to the framework, but how it is not an executable software, It was also used for an application other metrics.

Thus it was concluded that the AngularJS fared better 7 metrics, Ember.js be demonstrated more efficient in 2 metrics and there was a tie and a metric. From the results it was concluded that the AngularJS is better planned, using less technical resources relative to Ember.js. Yet this work does not reach a final verdict, staying for developer account analyze the results according to the requirements of your software.

Keywords: framework, quality, ISO/IEC 9126, metrics, development.

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

LISTA DE SIGLAS

AJAX - *Asynchronous Javascript and XML*
XML - *eXtensible Markup Language*
CSS - *Cascading Style Sheets*
HTML - *HyperText Markup Language*
SPA - *Single Page Application*
PHP - *PHP: Hypertext Preprocessor*
JSON - *JavaScript Object Notation*
API - *Application Programming Interface*
URL - *Uniform Resource Locator*
REST - *Representational State Transfer*
MVC - *Model-view-controller*
MVVM - *Model View ViewModel*
MVP - *Model-View-Presenter*
MVW - *Model View Whatever*
TI - Tecnologia da Informação
ISO - *International Organization for Standardization*
IEC - *International Electrotechnical Commission*
MB - *MegaByte*
GB - *GigaByte*
CRUD - *Create, Read, Update and Delete*
RAM - *Random-access memory*
OS - *Operating system*
SSD - *Solid-state drive*
W3C - *World Wide Web Consortium*

SUMÁRIO

INTRODUÇÃO

A preocupação com a qualidade dos *softwares* tem crescido de acordo com o que os mesmos vem sendo usados em nossas vidas. Segundo Pressman (2011), a qualidade de *software* pode ser definida como um processo eficaz de um *software*, aplicado de uma maneira que cria um *software* útil que oferece valor mensurável para quem o produz e quem use-o. A qualidade de *software*, é alcançada através da aplicação de métodos de engenharia de *software*, práticas de gestões sólidas e controle de qualidade, tudo com apoio de uma infraestrutura que garanta uma base já de qualidade. Hoje é cada vez mais desejável fabricar *softwares* de melhor qualidade com menor custo.

Desenvolver um *software* usando todas as metodologias de engenharia de *software*, práticas de gestão e controle de qualidade, nada irá adiantar, se os desenvolvedores usarem ferramentas sem qualidade e com falhas. Uma ferramenta sem qualidade gera um *software* sem qualidade. Em outubro de 2001 a CIO Magazine, uma revista comercial, publicou um artigo no qual dizia que as empresas chegam a desperdiçar US$ 78 bilhões por ano com *softwares* de má qualidade. Qualquer tipo de sistema está sujeito a falhas, por isso a necessidade de usar um *software* de qualidade.

Os sistemas *web* estão cada vez ganhando mais espaço, graças a sua disponibilidade, multiplataforma. Para facilitar o desenvolvimento dessas aplicações vão surgindo diversas ferramentas, cada uma com seu padrão de *software* e sua arquitetura. Essas ferramentas estão sujeitas a falhas e podem ter baixa qualidade. Dentre essas ferramentas as mais comuns são os *frameworks*, que servem como base para o desenvolvimento de uma aplicação.

Um modelo de sistema web é o *single page application* (SPA), é um modelo relativamente novo, que de acordo com Elliott (2014) em 2004 o Gmail foi a primeira aplicação a usar o modelo, que possui mais 160 *frameworks* de acordo com o GitHub.

Para Pereira (2014), SPA é uma aplicação que carrega uma única página, por meio da mesma o restante do conteúdo é gerado dinamicamente. Esse conceito é

altamente utilizado em aplicações modernas pois reflete uma experiência fluída para o usuário, comportando-se de forma semelhante a aplicações nativas do sistema operacional.

As aplicações web que seguem o modelo SPA, na maioria dos casos fazem uso de *frameworks*, para evitar retrabalho, aumentando a produtividade e diminuindo custos. Em uma busca feita no site de desenvolvimento colaborativo GitHub, usando o termo *single page application framework* foram retornados 165 resultados de *frameworks*. É muito trabalhoso para o desenvolvedor escolher o *framework* que garanta a qualidade adequada ao seu projeto, são inúmeros fatores envolvidos no processo de escolha de um *software*.

Tendo como base a capacidade das aplicações *single page application* e os dois *frameworks* mais populares para esse modelo, esse estudo tem a finalidade de analisar a qualidade interna dos *frameworks* AngularJS e Ember.js e com base nos resultados realizar uma comparação entre ambos, determinando qual é o *framework* mais indicado para determinados casos. Utilizando um método de avaliação definido por uma organização mundialmente respeitada, esse trabalho faz uso da ISO/IEC 9126. Esse método oferece um conjunto de métricas de avaliação de *software*, das quais 10 foram usadas nesse trabalho. A relevância desse trabalho justifica-se pela grande quantidade de *frameworks* disponíveis no mercado e pela baixa qualidade de alguns *softwares* existentes no mesmo.

Esse trabalho pode determinar fatores importantes para garantir a qualidade de *frameworks* e aplicações no modelo SPA. Os resultados levantados nesse trabalho servem como base para desenvolvedores realizarem a análise dos *frameworks* com base em seus requisitos de *software*. Com isso desenvolvedores, empresas e usuários podem ter acesso a *frameworks* capazes de gerar *softwares* de maior qualidade, reduzindo custos e retrabalho, levando aumento de produtividade aos usuários.

Os *frameworks* AngularJS e o Ember.js foram escolhidos por serem os *frameworks* para criação de SPA mais populares entre os desenvolvedores, de acordo com o site GitHub (2015). Para realização das análises, foram selecionadas algumas métricas da ISO/IEC 9126-3 e após isso, as mesmas foram aplicadas aos *frameworks* e a aplicações desenvolvidas com base nos *frameworks*, que foram aplicadas de acordo com as exigências de cada métrica.

Com os resultados obtidos através da utilização das métricas, foi possível

realizar comparações entre os *frameworks*, conhecer a importância da qualidade de *software* no desenvolvimento de uma aplicação e de um *framework* e quais impactos isso pode gerar em um projeto real utilizado para obtenção dos resultados de algumas métricas.

Esse trabalho foi dividido em seis capítulos, onde o segundo capítulo é o referencial teórico, que sustenta essa pesquisa, explicando a evolução das aplicações *web* até chegarem as SPAs, a qualidade de *software* e o modelo de qualidade de *software* ISO/IEC 9126. O terceiro capítulo apresenta a metologia usada para obter os resultados e analisar os mesmos. O quarto capítulo apresenta os resultados da análise, o quinto capítulo possui uma conclusão sobre o trabalho, no sexto capítulo foram apresentadas ideias de trabalhos futuros, para dar continuidade a esse trabalho.

1 REFERENCIAL TEÓRICO

Esta seção tem como objetivo, introduzir todos os conceitos necessários para o entendimento do trabalho, detalhando a evolução do ambiente *web* até chegar as *single page applications* (SPA), detalhando as tecnologias usadas nesse modelo de aplicação. Além disso, é demonstrado a importância da qualidade de software e como uma análise de software pode evitar prejuízos durante o desenvolvimento de um sistema.

1.1 A WEB

O ambiente *web* tem tornado-se cada vez mais a escolha das empresas que desejam desenvolver um novo *software.* Isso acontece graças à facilidade de acesso a esses sistemas, que em alguns casos têm como único requisito um navegador de internet instalado no sistema operacional. Outro fator importante é a disponibilidade das aplicações *web*, que podem ser acessadas de qualquer lugar que possua acesso à internet.

Segundo Flanagan (2011) as principais linguagens de desenvolvimento para *web* e suas responsabilidades são: o HTML (*HyperText Markup Language*) que é usado para especificar o conteúdo das páginas *web*, CSS que é usado para especificar a apresentação de páginas *web* e o JavaScript que é usado para especificar o comportamento de páginas *web*.

Segundo Mazza (2012) estamos em uma época revolucionária para o desenvolvimento *web*. Graças a avanços como a banda larga é possível acessar a internet por aparelhos de TV, *smartphones*, *tablets* e *videogames*. Com essas novas possibilidades, surgiu a necessidade de aumentar a interatividade dessas aplicações para que eles possam concorrer de igual para igual com as aplicações nativas desses dispositivos. Ainda de acordo com Mazza (2012) diversas dessas melhorias se devem a empresas que investem em novas tecnologias *web*, dentre elas podemos citar Google, Facebook, Microsoft e Apple.

Cannings, Dwivedi e Lackey (2008) dizem a *web* 2.0 foi criada depois da

Bolha da Internet no final década de 1990, os usuários passaram a usar a internet de uma forma diferente, usando sites onde o usuário gera seu próprio conteúdo, como por exemplo os blogs. Os desenvolvedores estão vendo um conjunto de tecnologias e soluções para enriquecer a experiência do usuário na Internet. Graças a essa evolução, hoje é possível ver *sites* da internet executando tarefas que somente programas nativos do sistema operacional eram capazes de fazer a dez anos atrás, como por exemplo os editores de texto, editores de planilhas, editores de apresentação, aplicativos de agenda de compromissos, contatos, entre outros.

Ainda de acordo com Cannings, Dwivedi e Lackey (2008) a *web* 2.0 é composta por novas tecnologias que são usadas para trazer mais interatividade para aplicações *web*, com o uso de tecnologias como AJAX (*Asynchronous JavaScript and XML*), CSS (*Cascading Style Sheets*), Flash, XML (*EXtensible Markup Language*) e uso de JavaScript avançado.

1.2 JAVASCRIPT

Segundo Ogeden (2015) JavaScript é uma linguagem de programação interpretada pelo navegador, que começou como uma maneira de fazer páginas *web* mais interativas, dando a possibilidade de executar *scripts* sem a necessidade de acessar o servidor, assim é possível controlar o navegador, executar requisições assíncronas e alterar o conteúdo que é exibido pelo navegador. Ela é padronizada pela especificação de linguagem ECMAScript, que atualmente se encontra na versão 6.

O JavaScript é executado em mais lugares do que apenas os navegadores da *web*. Existem plataformas que permitem executar JavaScript em servidores, em aplicativos de celulares, consoles de jogos e até mesmo em robôs. Flanagan (2011) diz que JavaScript é a linguagem de programação mais ubíqua da história.

Para Elliott (2014) JavaScript não só é a linguagem dominante do lado do cliente na *Web*. É uma das linguagens de programação mais avançadas e expressivas desenvolvidas até a data. Esse trabalho poderia ser feito sobre qualquer outra linguagem, porém o JavaScript é a linguagem de programação que tem obtido maior destaque ultimamente, sendo dita por vários autores como a linguagem que

mais se expande para outras plataformas.

De acordo com Elliott (2014) em 2004 o Gmail estava tirando vantagem da nova tecnologia AJAX, criando um *single page application*, rápido e ágil que mudaria para sempre a maneira que os aplicativos *web* são projetados.

O AJAX é uma maneira de usar o JavaScript para efetuar busca de dados no servidor depois que uma página já tiver sido carregada, ou seja de maneira assíncrona, desse modo a interface do usuário continua disponível para outras interações e quando a busca de dados é concluída a aplicação executa uma função, sendo assim possível remodelar uma parte ou toda a aplicação com base nos dados recebidos, é o que diz Messenlehner e Coleman (2014), isso evita a necessidade de recarregar a aplicação *web* para buscar uma nova informação, essa é uma base importante para as *single page applications*.

O AJAX aumentou as possibilidades para os desenvolvedores e com a evolução do JavaScript abrindo a possibilidade da criação de sistemas *web* mais interativos, com animações, armazenamento de dados em servidores sem recarregar a página, busca de dados assíncrona, assim as aplicações *web* ganharam espaço para concorrer com as aplicações nativas.

Como dito, a Google foi uma das primeiras empresas a compreender o potencial do AJAX, posteriormente utilizando a tecnologia em outras aplicações, transformando ela em um padrão de desenvolvimento para a *web,* o *single page application*.

1.3 SINGLE PAGE APPLICATION

De acordo com Fink e Flatow (2014), existem três tipos de aplicações mais comuns, as aplicações *web* tradicionais, as aplicações nativas e as SPA. As aplicações *web* tradicionais são executadas em servidores com linguagens *script* como PHP, a cada página requisitada esses servidores executam um programa que retorna uma página HTML que será exibida ao usuário pelo navegador, nesse caso a dinamicidade da aplicação está no servidor, sendo assim essas aplicações precisam realizar uma requisição de página durante a mudança de uma página para outra, durante essa mudança de páginas as aplicações ficam inutilizáveis, isso torna a

aplicação lenta e pouco fluída.

As aplicações nativas são executadas diretamente sobre o sistema operacional, sendo assim existe a necessidade de se instalar a aplicação no sistema operacional. As aplicações nativas são desenvolvidas usando um conjunto de ferramentas oferecidas para o sistema operacional específico, então a aplicação só poderá ser executada para tal sistema operacional. Além disso as aplicações nativas não possuem a necessidade de estarem conectadas a um servidor *web* para funcionarem e possuem uma ótima fluidez entre as páginas, é o que diz Fink e Flatow (2014).

De acordo com Fink e Flatow (2014), o modelo de aplicações SPA usa apenas uma página HTML como base para carregar todas as páginas do sistema, controlando as interações com o usuário através de JavaScript, HTML e CSS. Nesse caso as SPA podem ainda usar um servidor *web* para buscar apenas dados em formato JavaScript *Object Notation*(JSON) ou trechos de códigos HTML e não páginas inteiras, tornando a requisição menor, sendo assim a aplicação continua utilizável durante a mudança de páginas. Isso passa ao usuário a impressão de utilizar uma aplicação nativa do sistema operacional, além de reduzir o consumo de processamento e memória do servidor e o consumo de rede do servidor e do usuário.

As aplicações SPA possuem o desenvolvimento e comportamento parecido com o de uma aplicação nativa, já em sua execução elas não se assemelham, pois as SPAs são executas em navegadores enquanto as aplicações nativas são executadas sobre o sistema operacional, é o que diz Fink e Flatow (2014).

Tabela 1: Comparação entre tipos de aplicações

Funcionalidade	**Web Tradicional**	**Nativa**	**SPA**
Multiplataforma	V	X	V
Estado da aplicação	X	V	V
Sem instalação	V	X	V

Fonte: Fink e Flatow, 2014.

A tabela 1 faz uma comparação entre os três tipos de aplicações, as SPA se destacam por manter o estado da aplicação – não perde os dados ao alterar a página da SPA e se o usuário sair da aplicação ainda é possível armazenar informações no navegador para quando o usuário retornar –, ter capacidade multiplataforma – pode ser executado em qualquer sistema operacional com

navegador de internet instalado – e não precisa de instalação para ser usada.

O modelo SPA pode ser aplicado em qualquer plataforma que execute JavaScript, ele pode ser aplicado em programas para Android, iOS, Windows Phone entre outros, graças ao Cordova que executa código-fonte JavaScript em aplicativos *mobile*. Pode ser aplicado também a complementos para o navegador Google Chrome, também ao Chrome OS e também à plataforma *web* por meio dos navegadores. Com isso tendo um código-fonte escrito para uma plataforma pode ser facilmente reutilizado ou portado para outra plataforma.

Hoje o HTML está em sua versão 5, e várias novidades em comunicação, gráficos, multimídia, novos elementos para exibição de áudio e vídeos sem a necessidade de ter outros programas instalados, mais de 50 *Application Programming Interface*(API), *Web Sockets* para comunicação bidirecional, dentre outros, trouxeram mais possibilidades para as aplicações *web*, trazendo ainda mais possibilidades para as SPA.

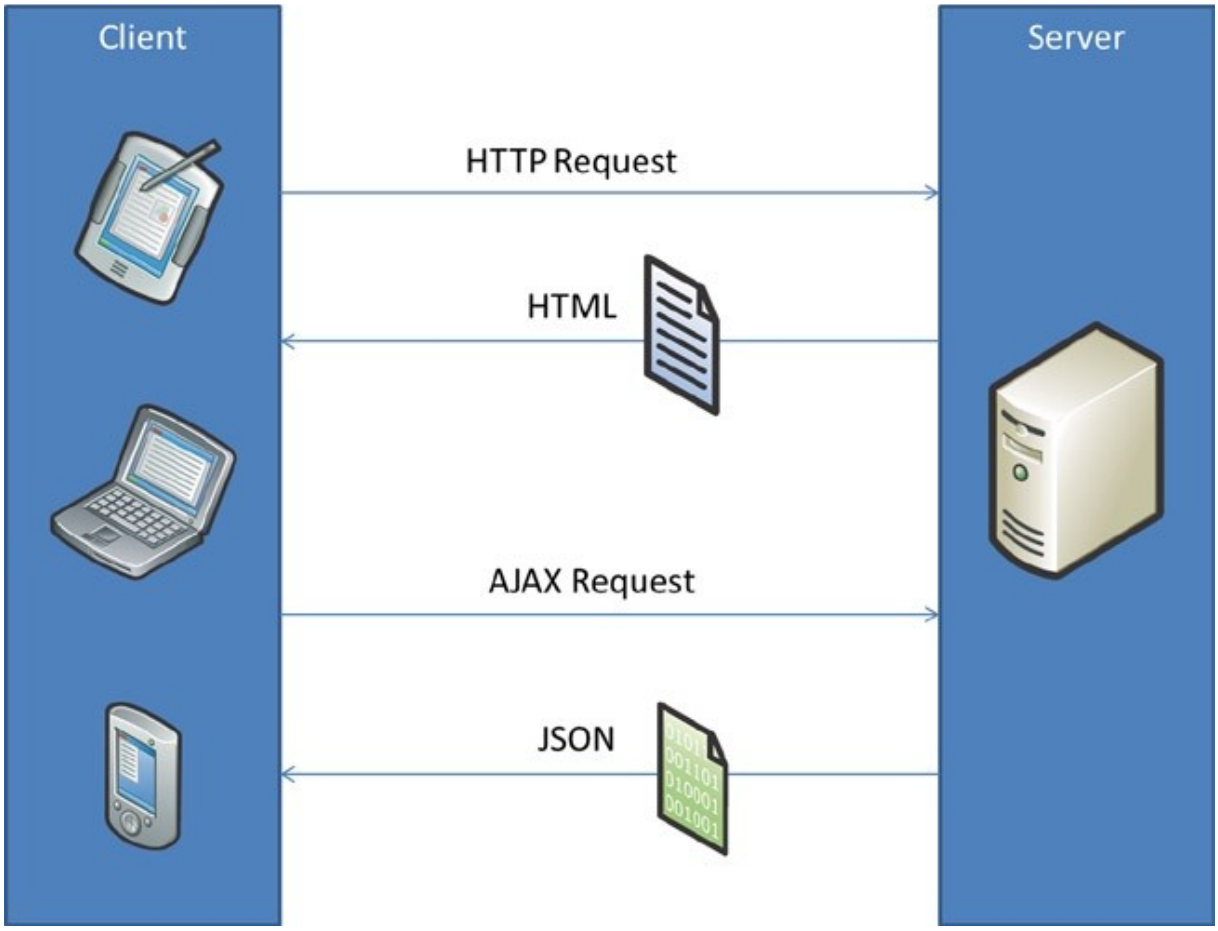

Figura 1: Ciclo de vida de uma *single page application*. Fonte: Fink e Flatow, 2014.

A figura 1 mostra o ciclo de vida de uma SPA, primeiramente o cliente realiza uma requisição de página para o servidor, então o mesmo retorna uma página HTML

que será exibida no navegador, após isso o usuário fará a navegação entre páginas, sendo que o conteúdo exibido nas páginas será requisitado para o servidor por meio de uma requisição assíncrona e o servidor retornará o conteúdo em formato JSON ou então pedaços de código HTML. Uma vez que os dados chegam ao cliente ele renderiza a página HTML, relembrando que as SPA não precisam buscar todos arquivos para renderizar uma nova página, permitindo que o usuário interaja com a aplicação durante a requisição, neste caso temos uma aplicação centralizada no cliente.

De acordo com Fink e Flatow (2014), as tecnologias descritas a seguir são as mais usadas pelas SPAs:

- As bibliotecas e *frameworks* JavaScript: Pode ser feito o uso de bibliotecas ou *frameworks* para facilitar na criação das aplicações.
- Rotas: As rotas são responsáveis por identificar as páginas e fornecer a navegação entre páginas em uma SPA. A maior parte do *frameworks* e bibliotecas possuem mecanismos para facilitar a criação e manipulação de rotas. De certo modo as rotas são os endereços das páginas (URLs).
- Motor de *templates*: Os motores de *templates* são responsáveis por criar visual de uma página. *Frameworks* para o desenvolvimento de SPA normalmente já trazem algum motor de *templates*, por exemplo, o *framework* Ember.js usa o Handlebars como motor de *templates*.
- HTML5: O HTML5 veio com uma série de novidades que permitem tornar as SPA mais interativas, dentre as novidades temos, acesso recursos multimídia, armazenamento, comunicação, dentre outros.
- API *Backend* e *Representation State Transfer* (REST): O servidor oferece uma *web* API que será consumida pela SPA, nenhuma página deve ser renderizada pelo servidor.
- Ajax: Todas interações entre o servidor são executadas de maneira assíncrona utilizando Ajax. Quando uma resposta chega ao SPA ela é parcialmente renderizada se necessário.

Ainda de acordo com Fink e Flatow (2014), as SPAs combinam o melhor das aplicações *web* tradicionais e das aplicações nativas, como descrito anteriormente.

Nas SPAs a lógica de negócio deve ser implementada no cliente, enquanto o servidor é usado como API para autenticar, validar e gravar dados em bancos de

dados. Por a lógica ser criada no cliente a aplicação responde de maneira rápida para o usuário, pois não há a necessidade de esperar o servidor realizar a lógica e retornar a resposta para a SPA. De acordo com Fink e Flatow (2014) por esses fatores o modelo SPA é perfeito para criar a próxima geração de aplicações *web*.

Implementar o uso dessas tecnologias em aplicações é um trabalho repetitivo, a cada aplicação criada as mesmas ferramentas são refeitas, trabalhando nesse conceito surgiram os *frameworks* para desenvolvimento de SPAs.

1.4 FRAMEWORKS

Assim como vários modelos de desenvolvimento possuem *frameworks* que facilitam o seu desenvolvimento, as aplicações no modelo SPA também possuem *frameworks*. Segundo Minetto (2007) um *framework* é uma ferramenta que serve como *software* base para o desenvolvimento de novos *softwares* maiores e mais específicos. Trazendo uma coleção de código-fonte, classes, funções, técnicas e metodologias para diminuir o trabalho necessário para desenvolver um sistema, aumentar a reusabilidade, facilitar a manutenção, aumentar a qualidade do *software* final e assim reduzir custos.

Existem no mercado diversos *frameworks* JavaScript que trazem facilidade no desenvolvimento de sistemas no modelo SPA, dentre eles temos o Durandal, AngularJS, Ember.js, scaleApp, Mithril e muitos outros. Cada *framework* trabalha em uma arquitetura de desenvolvimento diferente como o MVC (*Model-view-controller*), MVVM (*Model View ViewModel*), MVP (*Model–view–presenter*) e alguns que possuem seus próprios padrões. Essa diversidade de ferramentas traz algumas dificuldades ao desenvolvimento de sistemas e uma escolha errada pode levar o desenvolvimento de um *software* ao fracasso.

1.4.1 AngularJS

Pereira (2014) diz que a primeira versão do AngularJS foi lançada em 2012 e

que o início do seu desenvolvimento se deu em 2009. O projeto é mantido e atualizado pela Google e seu código-fonte está disponível no endereço da *web* https://github.com/angular/angular.js.

Seshadri e Green (2014) dizem que o AngularJS é um *framework* JavaScript que faz uso do modelo SPA e que usando ele terá de se que escrever menos código-fonte se comparado com uma solução de JavaScript puro. O AngularJS permite ter mais foco na lógica de negócio e nas funcionalidades do núcleo da aplicação, e criar componentes que podem ser reutilizados em outras partes da aplicação, código-fonte mais limpo também é um dos benefícios levantados por Seshadri e Green (2014). Demais benefícios foram levantados nos resultados desse trabalho.

Pereira (2014) diz que “o AngularJS é um *framework* JavaScript que simplifica o desenvolvimento de aplicações *web* dinâmicas robustas, viabilizando a implementação do conceituado modelo MVC (*Model-View-Controller*).”

Ainda de acordo com Pereira (2014) o AngularJS apresenta as seguintes características:

- Produtividade.
- Desempenho.
- Fácil customização.
- É testável e extensível.
- Implementa o fantástico conceito de diretivas ampliando o vocabulário HTML.
- Suporta o desenvolvimento de módulos.
- Implementa o revolucionário conceito de *Two-Way Data Binding*.
- Tem fácil integração com diversos *frameworks* e ferramentas JavaScript.

Como diz Pereira (2014) o AngularJS utiliza o modelo MVC, que significa modelo, visão e controlador, sendo que o modelo é responsável por realizar operações nos dados da aplicação, a visão são as páginas visualizadas pelo usuário, que são formadas por HTML e CSS, e o controlador é responsável por ligar as camadas de modelo e visão. O AngularJS ainda faz uso de um outro conceito o MVW (*Model-View-Whatever*), que significa modelo, visão e qualquer outra coisa que se fizer necessária. Neste caso o “qualquer outra coisa que se fizer necessária” faz referência ao conceito *Two-Way Data Binding*, que é utilizado pelo AngularJS,

esse conceito permite realizar uma ligação entre uma variável no modelo e uma outra variável na visão, sendo que essas fazem referência a uma mesma informação, através desse conceito é possível alterar a variável na visão e automaticamente a mesma será atualizada no modelo e vice-versa, sem a necessidade de utilizar um controlador para realizar a alteração nas duas variáveis ao mesmo tempo. Isso também é chamado por Pereira (2014) de uma alteração automática de “duas vias”.

De acordo com Pereira (2014) o conceito *Two-Way Data Binding* é responsável por uma grande economia de linhas de código-fonte em programas que usam o AngularJS.

1.4.2 Ember.js

De acordo com Selle et al. (2014) a primeira versão do Ember.js foi lançada em 2011, e ele descende do SproutCore, um *framework* muito sustentado pela Apple, hoje o Ember.js e o SproutCore seguem direções diferentes. Atualmente o Ember.js possui dez pessoas responsáveis pelo seu desenvolvimento, o código-fonte do Ember.js está disponível no endereço da *web* https://github.com/emberjs/ember.js.

De acordo com Agboado (2014) o Ember.js é projetado para ajudar os desenvolvedores a construir aplicações *web* ambiciosamente grandes e que concorram com aplicativos nativos. O Ember.js é concorrente direto do AngularJS, tendo como foco também a criação de sistemas no modelo SPA. Agboado (2014) diz que as URLs – URLs são os endereços usados para que sistemas *web* possam ser encontrados na internet – são o foco do Ember.js.

De acordo com a documentação oficial do Ember.js, as suas características centrais são:

- Os *templates*: Um *template* é um modelo de documento, que no Ember.js é responsável por criar a interface da aplicação, ou seja, o correspondente a visão do modelo MVC. A linguagem usada para a criação dos *templates* do Ember.js é a Handlebars, que possui diversas expressões que são usadas para ligar uma informação do modelo ao *template*, percorrer um vetor de

variáveis, dentre outros.

- Os componentes: Os componentes são a principal maneira de organizar as interfaces no Ember.js. Eles são parte de uma visão que podem ser reaproveitados, eles são formados por um *template* e um arquivo JavaScript que define o componente.
- Os controladores: Os controladores são responsáveis por ligar os *templates* ao modelo, porém no Ember.js eles se parecem muito com os componentes, por tanto a documentação recomenda que substitua-se os controladores por componentes, pois os controladores serão removidos no futuro para dar lugar totalmente aos componentes.
- Modelo: O modelo é responsável pelos dados da aplicação.
- Rotas: As rotas direcionam a aplicação para um controlador e um *template* específico, os mesmos ainda podem utilizar um ou mais modelos para exibição de dados no *template*.
- O Roteador: O roteador mapeia uma URL para uma rota.

A documentação usa essas características para enfatizar que as URLs são a parte mais importante da aplicação, pois elas descrevem o estado da aplicação, determinam qual rota será usada, controlador, *template* e modelos. Sendo assim as rotas são consideradas os componentes mais importantes do Ember.js.

1.5 QUALIDADE DE SOFTWARE

Tendo em vista a grande variedade de *frameworks*, o desenvolvedor se encontra diante do problema da qualidade dos mesmos. São centenas de *frameworks* disponíveis em plataformas de desenvolvimento colaborativo. O uso de ferramentas sem qualidade e que não seguem padrões mundiais, podem comprometer todo um projeto, levando prejuízos em dinheiro e em casos graves até vidas.

Em 2001 a CIO Magazine publicou um artigo com o título “Chega de desperdiçar US$ 78 bilhões por ano”, sobre o fato de que as empresas americanas gastavam bilhões em *softwares* que não faziam o que supostamente deveriam fazer.

A ComputerWorld também lamenta que:

> *Software* de má qualidade está em praticamente todas as organizações que usam computadores, provocando horas de trabalho perdidas durante o tempo em que a máquina fica parada, dados perdidos ou corrompidos, oportunidades de vendas perdidas, custos de suporte e manutenção de TI elevados e baixa satisfação do cliente.
> (Hildreth, 2005)

De acordo com Pressman (2011) a qualidade de *software* não pode ser definida em apenas uma expressão, são vários os fatores que trazem um *software* de qualidade, como o projeto do *software*, o empenho da equipe, o tempo disponibilizado para o desenvolvimento do projeto e as ferramentas de desenvolvimento. *Software* de qualidade é o fator mais importante dentro de uma empresa, um *software* de qualidade consome menos recursos humanos, financeiros e técnicos de uma empresa.

> A qualidade de um projeto de *software*, se refere as características definidas pelos projetistas a um produto. A qualidade dos materiais, as tolerâncias e as especificações de desempenho, todos são fatores que contribuem para a qualidade de um projeto.
> (Pressman, 2011, p. 359)

Para Pressman (2011) a análise e gestão de risco são ações que ajudam a equipe a trabalhar com a incerteza. Muitos problemas podem ocorrer com um projeto, o risco é um deles, ele pode ou não ocorrer.

De acordo com Sommerville (2011) existem pelo menos 6 tipos de riscos que podem ser incluídos em um projeto de desenvolvimento de *software*:

- Risco de tecnologia: Riscos que derivam das tecnologias de *software* ou *hardware* que são usadas para desenvolver um sistema.
- Riscos de pessoas: Riscos relacionados a pessoas que compõem a equipe do projeto.
- Riscos organizacionais: Riscos relacionados ao ambiente organizacional onde o *software* está sendo desenvolvido.
- Riscos de ferramentas: Riscos relacionados a ferramentas de *software* e outros *softwares* de suporte, usados para desenvolver o sistema.
- Riscos de requisitos: Riscos relacionados a mudanças nos requisitos do sistemas e no gerenciamento de mudanças do sistema.

- Riscos de estimativas: Riscos que derivam das estimativas de gerenciamento dos recursos necessários para construir o sistema.

O risco de tecnologia, está diretamente ligado a importância da escolha das ferramentas de desenvolvimento. Isso demonstra a importância de utilizar o *framework* certo no desenvolvimento de um *software*.

A escolha das ferramentas para o desenvolvimento de um *software* afetam todas as etapas de um projeto, desde o planejamento de custos de um *software*, passando pelo treinamento de profissionais, manutenção de código-fonte, reúso de código-fonte e chegando até o usuário final como um *software* de qualidade ou não.

Surgiram então os padrões de qualidade, que além de ajudar os desenvolvedores a escolher as suas ferramentas de trabalho, ajudam usuários a escolher o melhor programa para determinada tarefa, os padrões ainda servem como base para o desenvolvimento de *frameworks* de qualidade. A principal organização a definir padrões de qualidade é a ISO (*International Organization for Standardization*), que é descrita na seção 1.6.

1.6 NORMA DE QUALIDADE DE SOFTWARE

Denomina-se ISO a organização internacional para a padronização. Ela foi estabelecida em 1947, como uma organização mundial não governamental e conta atualmente com mais de 100 organizações nacionais de padronizações, possui representantes em mais de 130 países, responsáveis por mais de 95% da produção industrial mundial. Sua sede fica em Genebra, Suíça, tem como principal atividade a elaboração de padrões para especificações e métodos de trabalho nas mais variadas áreas. O principal objetivo da ISO é desenvolver padrões mundiais, com o objetivo de facilitar o intercâmbio internacional de produtos e serviços e a criar uma cooperação intelectual, científica, econômica e técnica, é o que diz Guerra e Colombo (2009).

Guerra e Colombo (2009) dizem que, a IEC (*International Electrotechnical Commission*), fundada em 1906, é a organização mundial que publica normas e padrões internacionais relacionadas com eletricidade, eletrônica e áreas afins, tendo

a participação de mais de 50 países.

De acordo com Guerra e Colombo (2009), a ISO, em conjunto com a IEC, elaboraram um conjunto de normas que tratam, especificamente, da atual padronização mundial para a qualidade de softwares.

As normas criam modelos de qualidade e processos de medição de qualidade, que são seguidos na criação e avaliação de *software*. O uso de tais normas e processos podem levar um *software* de maior qualidade até uma empresa, instituição ou usuário através da avaliação de *softwares*. Isso abre caminho para que os projetistas de *software* avaliem as ferramentas a serem usadas em seus sistemas, como os *frameworks*, assim antes mesmo de se começar a desenvolver o projeto, já é possível pensar em sua qualidade. Grande parte de um *software* que usa um *framework* é o próprio *framework*, isso mostra a importância da avaliação de dessas ferramentas.

1.6.1 ISO/IEC 9126-1: Modelo de qualidade para qualidade interna

A ISO/IEC 9126-1 é um modelo de qualidade de *software*, composto por duas partes, qualidade interna e qualidade externa e qualidade em uso, permitindo que a qualidade do *software* seja medida em diferentes perspectivas pelos envolvidos com aquisição, requisitos, desenvolvimento, uso, avaliação, apoio, manutenção, garantia de qualidade e auditoria de *software*.

Existem três modelos de qualidade de software, a qualidade em uso, a qualidade interna e a qualidade externa. O modelo de qualidade em uso possui quatro características de qualidade em uso e não possui subcaracterísticas. Os modelos de qualidade interna e qualidade externa possuem seis características para qualidade interna e externa, que por sua vez são subdivididas em subcaracterísticas. As características definidas nos modelos podem ser aplicadas em qualquer tipo de *software*, incluindo programas de computador, sistemas *web* e *frameworks*.

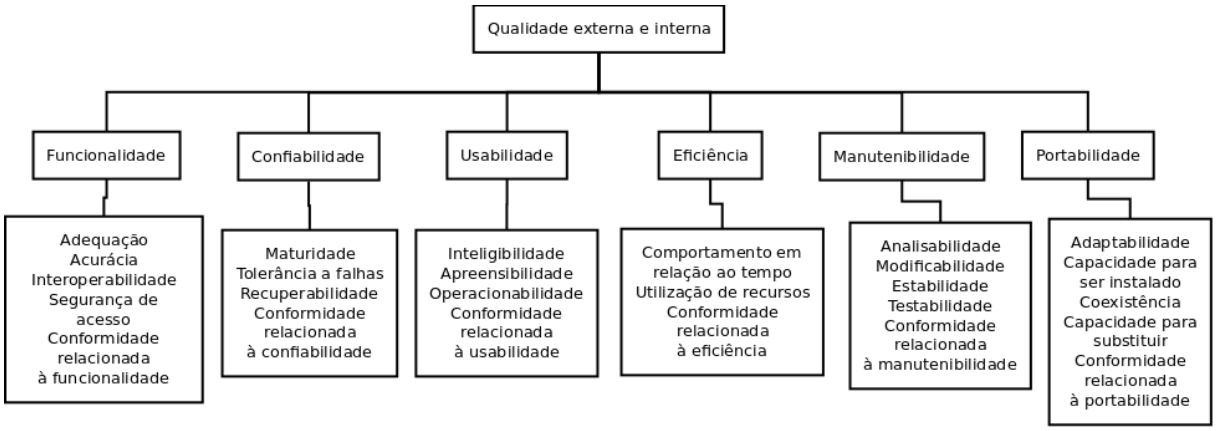

Figura 2: Atributos de qualidade interna e externa. Fonte: ISO/IEC 9126-3, 2002.

A figura 2 acima categoriza os atributos de qualidade interna e externa de *software* em seis características as quais são por sua vez, subdivididas em subcaracterísticas. As subcaracterísticas podem ser medidas por meio de métricas externas e internas. De acordo com a ISO/IEC 9126-1 as características são:

- Funcionalidade: Capacidade do produto de software de prover funções que atendam às necessidades explícitas e implícitas, quando o software estiver sendo utilizado sob condições especificadas.
- Confiabilidade: Capacidade do produto de software de manter um nível de desempenho especificado, quando usado em condições especificadas.
- Usabilidade: Capacidade do produto de software de ser compreendido, aprendido, operado e atraente ao usuário, quando usado sob condições especificadas.
- Eficiência: Capacidade do produto de software de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas.
- Manutenibilidade: Capacidade do produto de software de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do software devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais.
- Portabilidade: Capacidade do produto de software de ser transferido de um ambiente para outro.

As métricas de qualidade de *software* da ISO/IEC 9126 são divididas em três partes, a ISO/IEC 9126-2 que possui as métricas de qualidade externa, a ISO/IEC 9126-3 que possui as métricas de qualidade interna e a ISO/IEC 9126-4 que possui

as métricas de qualidade em uso. Através das métricas é possível medir a qualidade de um *software*, elas também fornecem uma estrutura para realizar comparações entre *softwares*.

1.6.2 ISO/IEC 9126-3

Métricas internas podem ser aplicadas a um produto de *software* não executável, tais como uma especificação ou código-fonte, durante o projeto e a codificação. É indicado que, no desenvolvimento de um de *software*, os produtos intermediários desse *software*, sejam avaliados utilizando-se métricas internas, as quais medem propriedades intrínsecas, incluindo aquelas que podem ser derivadas de um comportamento simulado. O propósito básico das métricas internas é garantir que a qualidade externa e a qualidade em uso requeridas sejam alcançadas. Métricas internas oferecem a usuários, avaliadores, executores de teste e desenvolvedores os benefícios de poderem avaliar a qualidade do *software* e considerar questões relativas à qualidade bem antes do *software* tornar-se executável.

Métricas internas medem atributos internos pela análise das propriedades estáticas de *softwares* intermediários ou preparados para entrega. Da mesma forma, as métricas internas podem ser indicadoras de atributos externos. As medições de métricas internas utilizam números ou frequências de elementos que compõem o *software* e que aparecem, por exemplo, em declarações de códigos-fonte, no gráfico de controle e nas representações de fluxo de dados e de transição de estados.

A ISO/IEC não atribui um intervalo de valores desejáveis para as métricas, pois cada *software* possui um objetivo e se propõe a atingir diversas necessidades diferentes entre os usuários, para um *software* determinado valor pode ser aceitável para outro não.

O usuário da ISO/IEC 9126-3 pode modificar as métricas de qualidade definidas e também pode adicionar métricas de qualidade de acordo com sua necessidade. Ao criar ou modificar uma métrica o usuário precisa relacionar a mesma ao modelo de qualidade ISO/IEC 9126-1 ou a outro modelo de qualidade que está sendo usado.

O usuário da ISO/IEC 9126-3 deve selecionar as características de qualidade e subcaracterísticas – referentes a ISO/IEC 9126-1 – a serem avaliadas, identificar as métricas relevantes e então interpretar o resultado da medição de uma maneira objetiva.

2 METODOLOGIA

Esse trabalho, tem como objetivo conhecer qual é o melhor *framework* para criação de SPA em cada uma das características que foram avaliadas, características essas que estão descritas na seção 2.4. As características descritas na seção 2.4 foram retiradas, modificadas e/ou criadas com base na ISO/IEC 9126-3, que traz um conjunto de métricas para avaliação de qualidade interna de *softwares*.

Os *frameworks* são produtos de *software* não executáveis, eles são usados como base para construção de outras aplicações, eles funcionam como o esboço de uma aplicação, ou seja, que não podem ser usados diretamente por um usuário e sim por desenvolvedores, a ISO/IEC 9126-3 foi escolhida por ser indicada para aplicação em produtos de *software* não executáveis.

Como será especificado na seção 2.4 nem todas as métricas da ISO/IEC 9126-3 foram utilizadas, pois são ao todo 70 métricas de qualidade interna, avaliar todas tornaria essa análise dos *frameworks* muito extensa e inviável, desse modo foram selecionadas 10 métricas de acordo com as características dos *frameworks*, para obtenção de resultados mais precisos algumas sofreram adaptações.

Para aplicação de determinadas métricas seriam necessários o uso de documentos como a especificação de requisitos dos *frameworks*, projeto e relatório de teste, esses não se encontram disponíveis para uso em trabalhos de acadêmicos. Para contornar esse problema foram usadas duas aplicações modificadas, que foram explicadas na seção 2.2, uma utilizando o AngularJS e outra utilizando e Ember.js, assim foram analisadas essas aplicações. As métricas que usaram as aplicações estão especificadas na seção 2.4. As demais métricas foram aplicadas sobre os próprios *frameworks* e também estão especificadas na seção 2.4.

Os *frameworks* AngularJS e Ember.js, foram escolhidos por sua popularidade entre os desenvolvedores, a seção 2.1 explica como foi o processo de escolha dos mesmos.

2.1 ESCOLHA DOS FRAMEWORKS

Os *frameworks* escolhidos para esse trabalho foram o AngularJS e Ember.js, a escolha foi realizada por meio de uma ferramenta do GitHub que permite medir a popularidade de projetos de *software* hospedados na plataforma, vale salientar que a maioria dos *frameworks* populares do mercado encontram-se hospedados no GitHub, isso elimina o uso de uma outra ferramenta para medir a popularidade dos *frameworks*, além disso foram usados outros parâmetros técnicos para escolha dos mesmos.

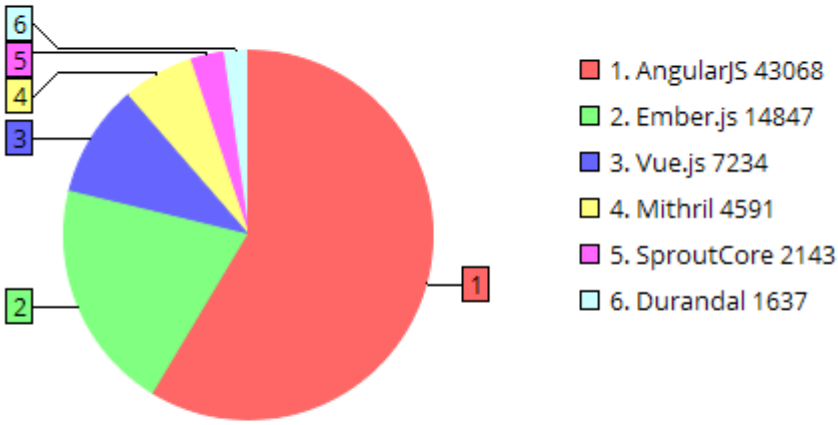

Gráfico 1: Popularidade de frameworks para single page application. Fonte: GitHub (2015).

O GitHub é um site de desenvolvimento de *software* colaborativo, para projetos que usam o controle de versionamento Git. Ele foi criado em 2008 por Tom Preston-Werner, Chris Wanstrath e PJ Hyett, em 2015 o site possui mais de 12 milhões de usuários e 30 milhões de projetos, em sua maior parte os projetos são de código-fonte aberto.

O gráfico 1 mostra dados coletados no site GitHub no dia 29 de setembro de 2015, referente a popularidade dos *frameworks*, de acordo com os desenvolvedores.

O gráfico 1 mostra a popularidade dos *frameworks* escolhidos de acordo com o GitHub. O GitHub é uma rede de desenvolvimento colaborativo, grande parte dos projetos de *software open-source* estão disponíveis no GitHub para os desenvolvedores, com isso o GitHub possui uma ferramenta que permite marcar uma estrela em um projeto de *software* que o desenvolvedor goste ou apoie, então um número maior de estrelas indica um *software* mais popular, dessa forma foi possível determinar os *frameworks* para SPAs mais populares de acordo com o gosto dos desenvolvedores.

No gráfico 1, o AngularJS aparece em azul e até o dia 29 de setembro de 2015 possuía 43068 estrelas, o que representa 59% do total de estrelas representadas no gráfico, o Ember.js possuía 14847 até o mesmo dia, o que representa 20% do total de estrelas representadas no gráfico, os demais *frameworks* representam apenas 21% do total. Sendo assim, o AngularJS e o Ember.js são os *frameworks* mais utilizados de acordo com a preferência de desenvolvedores no site GitHub.

Além da popularidade, outro fator que levou a escolha dos *frameworks* AngularJS e Ember.js, foi que ambos utilizam a arquitetura de MVC para suas aplicações, além disso, ambos possuem um motor de *templates* e possuem funcionalidades próprias para controle de rotas.

Para realizar as análises dos *frameworks* foram usadas a métricas da ISO/IEC 9126-3, alguns das métricas foram aplicadas sobre uma aplicação, que tem como base os *frameworks* AngularJS e Ember.js, a aplicação encontra-se descrita na sessão a seguir.

2.2 APLICAÇÃO

Foi necessário o uso de uma aplicação durante a análise dos *frameworks*, pois em determinadas métricas precisa-se realizar consumo de recursos de *hardware*, como descrito na seção 2.4. Os *frameworks* não podem ser executados no navegador, eles precisam ser usados como base de uma aplicação que terá seus resultados analisados, sendo que a aplicação deve usar os recursos do *framework*.

A aplicação usada nesta análise é a aplicação TODOMVC, que é uma lista de tarefas, onde o usuário cria tarefas que ele realizará, existe ainda a possibilidade de editar tarefas, excluir tarefas, marcar tarefas como prontas e filtrar as tarefas que estão sendo exibidas. Essa aplicação está disponível no GitHub, no endereço https://github.com/tastejs/todomvc, ela possui a mesma aplicação de lista de tarefas construída em vários *frameworks* diferentes, seu slogan é “Ajudando você a escolher um *framework* MV*”, todos os *frameworks* são escritos na linguagem JavaScript e são usados na criação de SPAs.

Essa aplicação foi escolhida para a análise neste trabalho pois ela trabalha com os aspectos mais importantes das SPAs. Faz uso do conceito de CRUD (*Create*, *Read*, *Update* e *Delete*), isso permite analisar o comportamento do *framework* respectivamente ao criar, ler, atualizar e excluir dados. As rotas descritas na seção 1.4.1, também são usadas nessa aplicação outro conceito básico em uma SPA, as rotas são usadas nessa aplicação para acessar tarefas com diferentes estados. Os *templates*, descritos na seção 1.4.1, que cada *framework* usa, também são usados dentro dessa aplicação, para a geração da interface das diferentes páginas. Além disso a aplicação armazena os dados no navegador, essa é uma funcionalidade nova que veio com a versão 5 do HTML.

Para realização deste trabalho, a aplicação sofreu algumas modificações, onde as tarefas são ordenadas primeiramente por estado – completa ou incompleta – e em seguida por nome em ordem alfabética e as tarefas concluídas que aparecem em cor amarelo claro e além disso, a aplicação foi traduzida para o português.

A aplicação descrita nos parágrafos anteriores pode ser vista na figura 3.

Figura 3: Aplicação TodoMVC modificada. Fonte: Próprio autor.

A figura 3 exibe a tela principal da aplicação, já com as modificações realizadas no trabalho.

Sendo que a maior parte do código-fonte da aplicação é composta pelo *framework*, foi possível alcançar os resultados esperados. A aplicação foi usada como base da análise, apenas quando foi inviável ou quando não foi possível usar o *framework*.

Todos os testes foram executados em um notebook, como será descrito na seção 2.3.

2.3 AMBIENTE DE TESTES

Para análise das aplicações e dos *frameworks*, foi utilizado um notebook com as seguintes especificações técnicas:

- Modelo Dell Inspiron 14 n4050.

- Processador Intel core i5 de segunda geração.
- 4 GB de memória RAM.
- Sistema operacional Elementary OS 0.3.1 Freya de 64 bits.
- Navegador Chromium versão 45.0.2454.101.
- Armazenamento SSD 120 GB.

Esse notebook apresenta boas especificações técnicas para a execução de *single page applications*, isso impede que travamentos decorrentes de problemas de *hardware* venham a interferir nos resultados das análises.

Para a realização dos testes foram usados o *framework* AngularJS em sua versão 1.4.3 e o Ember.js em sua versão 1.10.3.

Ambos *frameworks* foram submetidos aos métodos de análise descritos nas métricas da ISO/IEC 9126-3, tais métricas serão apresentadas na seção 2.4.

2.4 MÉTRICAS

As métricas descritas nessa seção foram retiradas, modificadas e/ou criadas com base na ISO/IEC 9126-3, que traz um conjunto de métricas para avaliação de qualidade interna de *softwares*. Essas métricas foram usadas para analisar os *frameworks* e as aplicações descritas na seção 2.2, que foram criadas como base nos mesmos.

Como dito anteriormente, as métricas internas são divididas em 6 características, funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. Porém, nem todas essas características se enquadram no objeto de estudo desse trabalho, com isso surgiu a necessidade de selecionar as características a serem analisadas nos *frameworks*, além disso algumas métricas sofreram algumas alterações, essa é uma das possibilidades que a ISO/IEC 9126-3 propícia.

As métricas foram escolhidas com base nas características que as SPAs possuem. As métricas que foram escolhidas para realização da análise estão descritas nas seções seguintes.

2.4.1 MÉTRICAS DE CONFIABILIDADE

As métricas de confiabilidade medem o quão confiável determinado software é, medindo a capacidade do produto de *software* de manter um nível de desempenho especificado, com o mínimo de falhas, quando usado em determinadas condições.

A tabela a 2 apresenta o nome da métrica, o seu propósito, a fórmula usada para obter o resultado e como esse resultado deve ser interpretado.

Tabela 2: Descrição das métricas de confiabilidade.

Nome da métrica	**Propósito**	**Fórmula**	**Interpretação do resultado**
Remoção de falhas	Qual é a proporção de falhas removidas?	X = A / B A = Número de falhas do *framework* corrigidas. B = Número de falhas do *framework* detectadas.	0 <= X <= 1 Quanto mais próximo de 1, melhor. (mais falhas removidas)

Fonte: Tradução do próprio Autor. Fonte: ISO/IEC 9126-3, 2002.

A tabela 2 apresentou a métrica remoção de falhas, para tal métrica foi utilizada a seguinte metodologia, foram recolhidos dados dos repositórios oficiais dos *frameworks* no GitHub. O GitHub oferece uma ferramenta para registrar falhas, das quais qualquer usuário pode sugerir uma correção para a mesma. A coleta dos dados foi realizada no dia 30 de outubro de 2015, com a métrica remoção de falhas será possível determinar qual *framework* possui um melhor gerenciamento para correção de falhas, um *framework* com menos falhas irá gerar uma aplicação com menos falhas, levando um produto de maior qualidade para o usuário.

Outra característica avaliada é a usabilidade que é descrita na seção 2.4.2 do presente trabalho.

2.4.2 MÉTRICAS DE USABILIDADE

A usabilidade é capaz de determinar se um *software* é agradável de ser usado por um usuário, se ele possui uma interface bonita, produtiva e que pode ser bem compreendida.

De acordo com a ISO/IEC 9126-3 uma métrica de usabilidade, mede a capacidade do produto de *software* de ser compreendido, aprendido, operado e agradável aos usuários, quando usado sob condições especificadas.

A tabela a seguir descreve a métrica de acessibilidade física, que determina se o software pode ser acessado por pessoas com deficiências físicas.

Tabela 3: Descrição das métricas de usabilidade.

Nome da métrica	**Propósito**	**Fórmula**	**Interpretação do resultado**
A acessibilidade física.	A ferramenta oferece algum mecanismo para garantir a acessibilidade de pessoas com deficiências físicas?	X = Sim, se o *framework* possui algum mecanismo que garante a acessibilidade de pessoas com deficiências físicas. Não, se o *framework* não possui algum mecanismo que garante a acessibilidade de pessoas com deficiências físicas.	X = Sim, significa que adequa a métrica. X = Não, significa que não se adequa a métrica.

Fonte: Tradução do próprio Autor. Fonte: ISO/IEC 9126-3, 2002.

A tabela 3 descreve como deve ser aplicada a métrica acessibilidade física, a mesma foi adaptada pelo próprio autor do trabalho, com ela foi medido se o *framework* possui algum mecanismo que garante a acessibilidade de pessoas com deficiências físicas, neste caso são botões maiores, fontes maiores, cores com maior contraste, dentre outros. Um mecanismo é considerado válido se o mesmo estiver contido na documentação do *framework*. Com ela será possível determinar se uma aplicação construída com qualquer um dos *frameworks* terá uma boa

usabilidade para usuários com deficiências visuais, deficiências físicas e mentais. Os dados foram obtidos no dia 31 de outubro de 2015.

Além da usabilidade, foram analisadas também características referentes a eficiência, a seção 2.4.3 descreve todas as métricas de eficiência que foram usadas durante esse trabalho.

2.4.3 MÉTRICAS DE EFICIÊNCIA

As métricas de eficiência tem como objetivo medir se o *software* em análise apresenta um bom desempenho, sendo capaz de executar várias tarefas e de maneira rápida, além disso consumir poucos recursos de *hardware* e de *software*, realizando isso sem comprometer as funcionalidades descritas pelo *software*.

De acordo com a ISO/IEC 9126-3 as métricas de eficiência medem a capacidade do produto de *software* de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas.

A tabela 4 descreve o primeiro grupo de métricas relacionadas a eficiência, que é composto por duas métricas usadas para medir tempo de execução.

Tabela 4: Descrição das métricas de eficiência(Grupo 1).

Nome da métrica	**Propósito**	**Fórmula**	**Interpretação do resultado**
Tempo de resposta.	Qual é o tempo estimado para completar uma tarefa específica?	X = tempo (calculado ou simulado)	Quanto menor, melhor.
Tempo de processamento.	Qual é o número estimado de tarefas que podem ser realizadas através de uma unidade de tempo?	X = Número de tarefas executadas por unidade de tempo.	Quanto maior melhor.

Fonte: Tradução do próprio Autor. Fonte: ISO/IEC 9126-3, 2002.

Para calcular a métrica tempo de resposta, foi executada a funcionalidade de criação de tarefa, exclusão da tarefa e edição de uma tarefa, em ambas aplicações. Para realizar a medição foram inseridas linhas de código-fonte, essas linhas de

código imprimem no console do navegador o tempo com precisão de milionésimo de segundo, o tempo foi impresso antes e depois de cada tarefa executada. Para saber o tempo gasto foi subtraído o tempo obtido depois da execução pelo tempo obtido antes da execução. Para atingir um resultado mais preciso essa tarefa foi executa por 100 vezes, cada funcionalidade teve o resultado de todas as execuções exibidos em um gráfico, e para aplicação da métrica ISO/IEC 9126-3 foi realizada uma média dos valores.

Para realizar a medição da métrica de tempo de processamento, todos os dados das aplicações foram removidos, a aplicação executou a funcionalidade de criação de tarefas por um segundo, após isso foram contadas as quantidades de tarefas que foram criadas. Para calcular a quantidade de edições e exclusões que as aplicações são capazes de realizar em um segundo, foram inseridos 5000 tarefas, no caso da edição de tarefas foram contadas quantas tarefas foram editadas durante esse período de tempo, já métrica de exclusão de tarefas foi subtraído a quantidade restante de tarefas da quantidade de total de tarefas, para se obter o resultado.

As métricas relacionadas ao tempo de execução, foram capazes de determinar qual *framework* é o mais rápido ao executar uma determinada funcionalidade e qual *framework* é capaz de executar o maior número de tarefas em um determinado tempo.

As métricas do segundo grupo de métricas de eficiência foram capazes de determinar qual *framework* é mais eficiente no gerenciamento de consumo de memória em geral, as mesmas se encontram descritas na tabela 5.

Tabela 5: Descrição das métricas de eficiência(Grupo 2).

Nome da métrica	**Propósito**	**Fórmula**	**Interpretação do resultado**
Utilização de memória de armazenamento.	Qual é o espaço de memória de armazenamento que o *software* vai ocupar após completar uma tarefa específica?	X = tamanho em *bytes* ocupado (calculado ou simulada)	Quanto menor, melhor.
Utilização de memória RAM.	Qual é o espaço de memória RAM que o	X = tamanho em *bytes* ocupado (calculado ou	Quanto menor, melhor.

	software vai ocupar para completar uma tarefa específica?	simulada)	
Utilização de transmissão de dados.	Qual é a quantidade estimada de utilização de recursos de transmissão de dados?	X = *bytes* / tempo (calculado ou simulada)	Quanto menor, melhor.

Fonte: Tradução do próprio Autor. Fonte: ISO/IEC 9126-3, 2002.

As métricas descritas na tabela 5 são responsáveis por medir consumo de memória em geral. A métrica utilização de memória de armazenamento que mede a quantidade de armazenamento que foi utilizada após executar 5 vezes a funcionalidade de criação de tarefas, essa medição foi realizada com a ferramenta console das ferramentas do desenvolvedor do Chromium, digitando o comando *localStorage.getItem('todos-angularjs')* são retornados os dados da aplicação feita com o AngularJS, digitando o comando *localStorage.getItem('todos-emberjs')* são retornados os dados da aplicação feita com o Ember.js, o valor que será aplicado a métrica é o valor utilizado após a inserção da quinta tarefa.

A métrica utilização de memória RAM mediu a quantidade máxima de memória RAM consumida pela aplicação ao executar a funcionalidade de criação de tarefas, após isso a aplicação e o navegador foram fechados, foi executada a funcionalidade de edição de uma tarefa e medido o consumo de memória RAM, após isso a aplicação e o navegador foram fechados, então foi executada a funcionalidade de exclusão de tarefa e medido o consumo de memória RAM da mesma, o consumo de memória RAM foi medido através da ferramenta gerenciador de tarefas do Chromium.

A métrica utilização de transmissão de dados, mediu a quantidade de *bytes* transmitidos até que a aplicação estivesse completamente carregada no navegador de internet, para tal medição foi utilizada a ferramenta *network,* que pode ser encontrada no menu de ferramentas do navegador Chromium.

Com o segundo grupo de métricas de eficiência foi possível determinar qual *framework* é capaz de executar em computadores com menor quantidade de memória RAM, algo que é extremamente válido levando em conta grande quantidade de smartphones com pouca memória RAM disponíveis no mercado, com a métrica de utilização de memória de armazenamento foi possível determinar qual

framework consome menos espaço de memória para armazenar seus dados, e com a métrica de utilização de transmissão de dados foi possível determinar qual *framework* consome menor banda de internet, esse é um fator importante se levarmos em conta que a maioria dos planos de internet para smartphones hoje em dia são limitados ao consumo de alguns *megabytes* por mês.

A métrica do terceiro grupo de métricas de eficiência calcula quantas dependências de outro software o *framework* possui, a descrição de tal métrica está na tabela 6.

Tabela 6: Descrição das métricas de eficiência(Grupo 3).

Nome da métrica	**Propósito**	**Fórmula**	**Interpretação do resultado**
Número de dependências de *software*.	Medir a quantidade de dependências que um *software* possui.	X = Quantidade de dependências de *software*.	Quanto menor, melhor.

Fonte: Tradução do próprio Autor. Fonte: ISO/IEC 9126-3, 2002.

A métrica número de dependências de *software*, foi criada pelo próprio autor deste trabalho, com o objetivo de medir qual é a quantidade de dependências que o *framework* possui. As dependências são outros software necessários para que a aplicação funcione de maneira correta, essa métrica foi medida com base na aplicação desenvolvida nesse trabalho, ou seja, o número de dependências necessárias para que a aplicação TODOMVC funcione, com cada *framework*.

Um *framework* que possui dependências de muitos outros *softwares*, mostra-se pobre em número de funcionalidades, pois precisa de outros *softwares* para complementar suas funções, isso pode tornar a aplicação como um toda mais complexa e assim consumindo mais recursos de *hardware*, desse modo essa métrica pode justificar os resultados obtidos nas métricas de eficiência descritas anteriormente.

A seção 2.4.4 descreve as métricas relacionadas a manutenibilidade, com a manutenibilidade foi possível aprimorar a eficiência do *framework*.

2.4.4 MÉTRICAS DE MANUTENIBILIDADE

Através das métricas de manutenibilidade será possível determinar a facilidade de se prestar manutenções, melhorias e adaptações ao *software*, um código-fonte de fácil compreensão é um fator de grande importância, de acordo com o que é descrito pela ISO/IEC 9126-3.

De acordo com a ISO/IEC 9126-3, métricas de manutenibilidade medem a capacidade do produto de *software* de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do *software* devido a mudanças no ambiente e nos seus requisitos ou nas especificações funcionais. Na tabela 7 está descrita a métrica mudança de código-fonte.

Tabela 7: Descrição das métricas de manutenibilidade.

Nome da métrica	**Propósito**	**Fórmula**	**Interpretação do resultado**
Mudança no código-fonte.	Em alterações nos módulos do *software*, as mesmas recebem comentários?	X = A / B A = Número de mudanças no código-fonte que receberam comentários. B = número total de funções que foram alteradas.	0 <= X <= 1 Quanto mais próximo de 1, melhor.

Fonte: Tradução do próprio Autor. Fonte: ISO/IEC 9126-3, 2002.

De acordo com a tabela 7, a métrica mudança no código-fonte é capaz de medir quantas alterações no código-fonte do *framework* receberam comentários para assim facilitar o entendimento de quem for ler o código-fonte posteriormente. Para realizar essa medição foi realizado o uso do GitHub, em cada um dos *frameworks* foram analisadas as dez últimas alterações de código-fonte feitas até o dia 1 de novembro de 2015, e verificadas se as mesmas possuíam comentários. Não foram conferidas todas as alterações, pois a quantidade de alterações realizadas nos *frameworks* é muito alta, o que inviabilizaria a utilização da métrica.

Além de todos as características descritas até o momento, também foram aplicadas por último as métricas de portabilidade, que estão descritas na seção

2.4.5.

2.4.5 MÉTRICAS DE PORTABILIDADE

As métricas de portabilidade tem como objetivo determinar se um *software* pode ser usado em vários ambientes e em condições diferentes.

De acordo com a ISO/IEC 9126-3 as métricas de portabilidade medem a capacidade do produto de *software* de ser transferido de um ambiente para outro. As métricas de portabilidade estão descritas na tabela 8.

Tabela 8: Descrição das métricas de portabilidade.

Nome da métrica	**Propósito**	**Fórmula**	**Interpretação do resultado**
Flexibilidade de Instalação.	O quão flexível e personalizável pode ser a instalação do *software*?	X = possibilidades de instalação.	Quanto maior, melhor.

Fonte: Tradução do próprio Autor. Fonte: ISO/IEC 9126-3, 2002.

A métrica flexibilidade de instalação descrita na tabela 8, foi personalizada para se adaptar as necessidades do atual objeto de estudo, foram contadas todas as possibilidades de instalação do *framework* descritas na documentação dos mesmos. Com tal métrica, foi possível determinar em quantos ambientes diferentes o *framework* pode ser instalado, isso é importante pois torna o *framework* flexível, se adaptando as necessidades do desenvolvedor, além disso, quanto maior as possibilidades maior deve ser a quantidade de desenvolvedores que o *framework* irá atrair.

Todas as métricas descritas na seção 2.4, foram aplicadas aos *frameworks* ou as aplicações, de acordo com o que foi especificado. Com as métricas selecionadas, será possível determinar com grande precisão qual é o *framework* mais indicado para cada caso, levando em consideração as características mais importantes de um *software* de acordo com o modelo de qualidade da ISO/IEC 9126-1, que foi descrito na seção 1.6.1.

Tendo as características e métricas como base, foram aplicadas as

metodologias em ambiente de teste, os resultados obtidos foram apresentados e analisados na seção 3.

3 RESULTADOS

Os dados que se seguem, tem por objetivo apresentar os resultados e análise dos mesmos com base nas métricas e metodologias descritas na seção 2 deste trabalho.

Os resultados foram exibidos em formatos de tabelas e gráficos, logo após cada tabela ou gráfico se seguirá uma análise dos mesmos.

3.1 MÉTRICAS DE CONFIABILIDADE

Os resultados apresentados a seguir, são referentes às características de confiabilidade, determinando o quão confiável o *framework* mostra-se, jugando aspectos como número de falhas corrigidas.

3.1.1 Remoção de falha

A primeira métrica a ser analisada foi a métrica que mede a remoção de falhas no *framework*. Através dos dados obtidos no repositório de cada *framework*, até o dia 30 de setembro de 2015, os resultados foram calculados de acordo com esta métrica, os resultados são mostrados na tabela 9.

Tabela 9: Dados sobre a remoção de falhas.

Framework	**Falhas encontradas**	**Falhas corrigidas**	**Resultados**
AngularJS	6977	6041	0.8658
Ember.js	4147	3869	0.9329

Fonte: Próprio autor.

A tabela 9 mostra os resultados de ambos os *frameworks*. O AngularJS possui uma proporção menor de falhas corrigidas em relação a falhas reportadas pelos desenvolvedores, com um valor de aproximadamente 0.86, já o Ember.js demostra-

se melhor nesse quesito tendo um resultado mais próximo de um com o valor de aproximadamente 0.93, quanto mais próximo de um, melhor. Desse modo, no aspecto de correção de falhas e de acordo com os resultados, o Ember.js mostra-se como melhor opção.

Desse modo, os resultados obtidos pelo presente estudo, mostra que o Ember.js mostra-se como melhor *framework* para características de confiabilidade. A seguir estão os resultados obtidos nos testes das métricas de usabilidade.

3.2 MÉTRICAS DE USABILIDADE

Os resultados a seguir medem a capacidade do *framework* de ser compreendido, aprendido, operado e atraente aos usuários, avaliando aspectos como acessibilidade física.

3.2.1 Acessibilidade física

Os resultados da tabela 10 são referentes a métrica de acessibilidade física, os dados usados foram obtidos na data de 31 de outubro de 2015, através da documentação de ambos os *frameworks*.

Tabela 10: Dados sobre a acessibilidade física

Framework	**Disponibilidade de mecanismo que garante a acessibilidade de pessoas com deficiências físicas**
AngularJS	Sim
Ember.js	Sim

Fonte: Próprio autor.

A tabela 10 mostra resultados de ambos os *frameworks*. Ambos os *frameworks* possuem mecanismos que facilitam a acessibilidade de pessoas com deficiências físicas. Outro fato importante é que ambos seguem os padrões de acessibilidade definidos pela *World Wide Web Consortium* (W3C). A W3C é uma

comunidade internacional que desenvolve padrões abertos para a internet. Sendo assim ambos os *frameworks* apresentam interfaces agradáveis a pessoas com deficiências físicas. Conclui-se que no aspecto de acessibilidade física ambos os *frameworks* apresentam bons resultados.

Para as métricas de usabilidade, ambos os *frameworks* apresentaram bons resultados, sendo assim capazes de levar uma interface agradável, de fácil compreensão e fácil de ser operada.

A seção 3.3 possuem os resultados e análises referentes às métricas de eficiência.

3.3 MÉTRICAS DE EFICIÊNCIA

Os resultados a seguir mostram a capacidade do produto de *software* de apresentar desempenho apropriado, relativo a quantidade de recursos utilizados.

3.3.1 Tempo de resposta

O gráfico 2 representa os resultados de tempo de resposta referente a funcionalidade de criação de tarefa, o mesmo foi gerado com dados coletados durante a execução das aplicações no ambiente de testes.

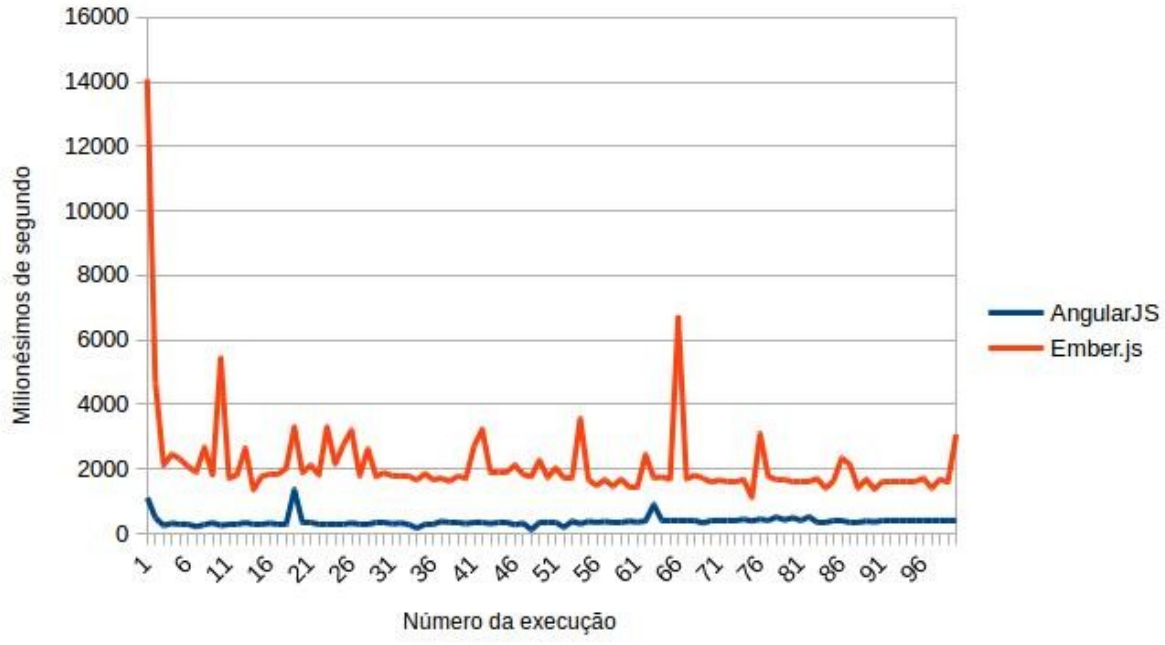

Gráfico 2: Tempo de resposta para funcionalidade de criação de tarefa. Fonte: próprio autor.

O gráfico 2 mostra que durante a primeira execução da funcionalidade, ambos *frameworks* apresentaram um tempo de execução superior em relação a maioria das demais execuções, isso acontece por que os *frameworks* fazem cacheamento do seu código-fonte após a primeira execução, fazendo com que a funcionalidade de criação de tarefas execute de maneira mais rápida nas execuções seguintes. Durante a execução a aplicação com Ember.js apresentou muitas variações em seu tempo de execução, isso pode ser justificado por uma perda de cacheamento, mesmo assim a aplicação que usa Ember.js foi diminuindo o tempo necessário para executar a funcionalidade gradativamente. A aplicação desenvolvida com AngularJS apresentou poucas variações em seu tempo de execução, mostrando eficiência ao gerenciar o cacheamento do seu próprio código-fonte porém, após algumas execuções o mesmo apresentou um pequeno aumento no tempo de execução da tarefa, mesmo assim, o AngulaJS durante todas as execuções apresentou um tempo de execução menor que o tempo gasto pela aplicação desenvolvida em Ember.js.

A tabela 11 exibe o tempo médio gasto por cada aplicação para executar a funcionalidade de criação de tarefas, a média dos foi obtida através dos dados do gráfico 2.

Tabela 11: Média relativa a tempo de resposta para funcionalidade de criação de tarefa.

	AngularJS	**Ember.js**

Milionésimos de segundo	372	2149

Fonte: Próprio autor.

A tabela 11 exibe a média dos valores de todas execuções para a funcionalidade de criação de tarefas. O AngularJS apresenta-se em média 477% mais rápido que o Ember.js, consumindo em média 1777 milionésimos de segundo a menos para executar a mesma tarefa. De acordo com a métrica para tempo de resposta o AngularJS é o melhor *framework*, para desenvolvedores que procuram um *framework* capaz de inserir registros em uma aplicação com melhor velocidade.

O gráfico 3 exibe os resultados de tempo de resposta referente a funcionalidade de edição de tarefa.

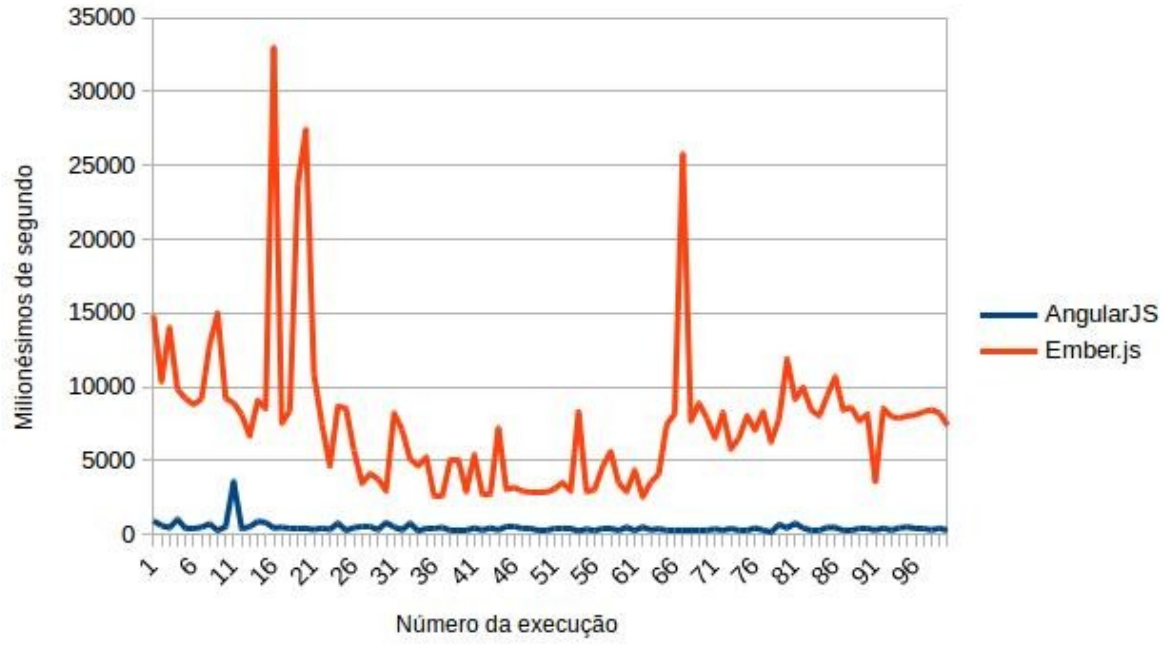

Gráfico 3: Tempo de resposta para funcionalidade de edição de tarefa. Fonte: próprio autor.

De acordo com o que é observado no gráfico 3, na funcionalidade de edição de tarefas nenhuma das aplicações apresentou sinais de cacheamento. A aplicação desenvolvida com AngularJS na décima primeira execução apresentou o maior tempo ao executar a funcionalidade, consumindo 3589 milionésimos de segundo para executar a funcionalidade, durante as outras execuções o tempo necessário para executar as funcionalidades se manteve estável. A aplicação desenvolvida com Ember.js, gastou o seu maior tempo na décima sexta execução, onde o tempo foi de 32969 milionésimos de segundos, além disso o Ember.js apresentou uma grande variação de tempo necessário para executar a funcionalidade, essa grande variação no tempo pode fazer com que a aplicação apresente travamentos inesperados,

quando a mesma trabalhar com uma grande quantidade de informação.

A tabela 12 exibe o tempo médio gasto por cada aplicação ao executar a funcionalidade de edição de tarefa, a média foi obtida através dos dados presentes no gráfico 3.

Tabela 12: Média relativa a tempo de resposta para funcionalidade de edição de tarefa.

	AngularJS	**Ember.js**
Milionésimos de segundo	471	7628

Fonte: Próprio autor.

A tabela 12 mostra que o AngularJS se apresenta em média 1520% mais rápido que o Ember.js, esse valor corresponde ao que foi observado no gráfico 3, a grande variação no tempo de execução do Ember.js fez com que a média dos resultados ficasse alta, isso pode fazer com que a aplicação desenvolvida com Ember.js seja mais lenta. De acordo com a métrica de tempo de resposta o melhor *framework* é o AngularJS, sendo o mais indicado para desenvolvedores que querem um bom desempenho ao editar registros.

O gráfico 4 exibe os resultados de tempo de resposta referente a funcionalidade de exclusão de tarefa.

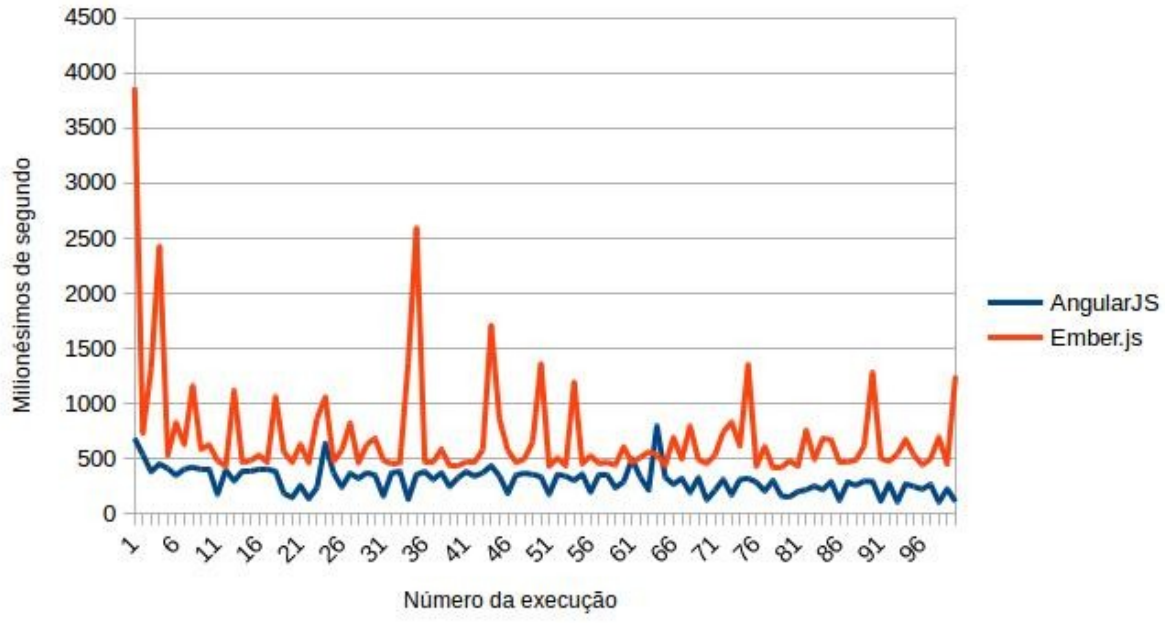

Gráfico 4: Tempo de resposta para funcionalidade de edição de tarefa. Fonte: próprio autor.

Como foi mostrado no gráfico 4, na funcionalidade de exclusão de tarefa, ambas aplicações realizaram cacheamento de seu código-fonte, podemos perceber que a primeira execução da aplicação em Ember.js demorou 3869 milionésimos de

segundo para executar, sendo que nenhuma outra execução demorou tanto quanto a mesma. O AngularJS também apresentou cacheamento, porém na 64ª execução a aplicação demorou 800 milionésimos de segundos para executar a funcionalidade enquanto na primeira execução foram gastos 689 milionésimos de segundo, essa variação pode ser justificada por uma perda de cacheamento. Durante todo o gráfico em ambas aplicações pode-se perceber grande variação dos valores, isso é justificável pela perda de cacheamento, em aplicações de grande porte isso pode causar travamentos, principalmente na aplicação que usa Ember.js, onde as variações são maiores. Apenas nas execuções 61 e 64, a aplicação desenvolvida em AngularJS se apresentou mais lenta que a aplicação desenvolvida em Ember.js.

A tabela 13 mostra o tempo médio que cada aplicação demorou para executar a funcionalidade de exclusão de tarefa, a média foi obtida através dos dados presentes no gráfico 4.

Tabela 13: Média relativa a tempo de resposta para funcionalidade de exclusão de tarefa.

	AngularJS	**Ember.js**
Milionésimos de segundo	309	709

Fonte: Próprio autor.

A tabela 13 mostra os resultados mais próximos em relação as tabelas 11 e 12, mesmo estando com valores mais próximos, o AngularJS apresentou um tempo de execução 129% menor que o Ember.js. Esses resultados foram os melhores para ambas as aplicações em relação as tabelas 11 e 12. Com base nesses resultados o melhor *framework* de acordo com a métrica de tempo de resposta é o AngularJS, sendo ele mais indicado para aplicações que precisam de velocidade ao excluir dados de um sistema.

Desse modo, com a aplicação da métrica de tempo de resposta nas três funcionalidades diferentes, o AngularJS se mostrou como a melhor opção, tanto para inserir, editar ou excluir dados. Sendo ele capaz de fazer um bom cacheamento de código-fonte, permitindo que suas funcionalidades executem de maneira mais rápida e eficiente, sem grandes variações em seu tempo de resposta

A seção 3.3.2 exibe os resultados obtidos com o uso da métrica de tempo de processamento.

3.3.2 Tempo de processamento

Nesta seção, serão exibidos os resultados referentes a métrica de tempo de processamento, que indica quantas tarefas podem ser executadas no período de um segundo. Essa medição foi feita para as funcionalidades de criação, edição e exclusão de tarefas.

Os resultados mostrados na tabela 14 mostram quantas vezes a funcionalidade de criação de tarefas foi executada por cada aplicação no tempo de um segundo, os dados da tabela 14 foram obtidas através da execução da aplicação.

Tabela 14: Dados relativos a tempo de processamento para a funcionalidade de criação de tarefa

	AngularJS	**Ember.js**
Quantidade	1512	3330

Fonte: Próprio autor.

Em um primeiro momento os resultados apresentados na tabela 14 mostram-se contraditórios aos dados obtidos na métrica de tempo de resposta para a mesma funcionalidade, porém os resultados podem ser explicados da seguinte forma, na métrica de tempo de resposta cada tarefa foi executada de maneira manual, havendo um tempo até a execução da próxima, já nessa métrica as inserções são feitas de maneira automática, com isso é possível perceber que o Ember.js perde cacheamento mais rapidamente (isso explica as grandes variações em seus resultados para a métrica de tempo de resposta) porém, quando as funcionalidades são executadas imediatamente uma após a outra, seu sistema de cacheamento se demonstra mais eficiente, conseguindo inserir dados 120% mais dados que o AngularJS, desse modo o Ember.js se mostra como uma melhor opção para a métrica de tempo de processamento.

Os resultados mostrados na tabela 15 mostram quantas vezes a funcionalidade de edição de tarefas foi executada por cada aplicação no tempo de um segundo.

Tabela 15: Dados relativos a tempo de processamento para a funcionalidade de edição de tarefa

	AngularJS	**Ember.js**

Quantidade	402	5927

Fonte: Próprio autor.

Novamente de acordo com os dados exibidos na tabela 15, o Ember.js mostrou realizar um bom cacheamento para executar funcionalidades uma imediatamente após a outra, sendo assim, ele apresenta um mal cacheamento para realizar funcionalidades um tempo maior após a outra. Para a funcionalidade de edição de tarefas o Ember.js editou 1374% mais dados que a mesma aplicação em AngularJS. Desse modo na métrica de tempo de processamento, o Ember.js se mostra como o melhor framework, sendo mais indicado para desenvolvedores que querem editar muitos dados ao mesmo tempo.

Os resultados mostrados na tabela 16 mostram quantas vezes a funcionalidade de exclusão de tarefas foi executada por cada aplicação no tempo de um segundo.

Tabela 16: Dados relativos a tempo de processamento para a funcionalidade de exclusão de tarefa

	AngularJS	**Ember.js**
Quantidade	530	1610

Fonte: Próprio autor.

Novamente nesta análise, o Ember.js se mostrou mais rápido, conseguindo excluir 204% mais registros que a mesma aplicação desenvolvida em AngularJS. Desse modo, o Ember.js se mostrou novamente mais eficiente em realizar o cacheamento de códigos-fonte que são executados um seguidamente após o outro. Sendo assim, de acordo com a métrica de tempo de processamento, o Ember.js se mostrou como uma melhor opção para desenvolvedores que querem que suas aplicações excluam uma quantidade grande de dados ao mesmo tempo.

O Ember.js apresentou os melhores resultados em todos os testes para a métrica de tempo de processamento, por realizar um melhor cacheamento de seus códigos-fonte. Sendo assim, o Ember.js é mais indicado para desenvolvedores que procuram um *framework* capaz de realizar a mesma tarefa consecutivas vezes e apresentar um bom desempenho.

A terceira métrica de eficiência exibe os resultados referentes a utilização de memória de armazenamento, os mesmos são exibidos na seção 3.3.3.

3.3.3 Utilização de memória de armazenamento

Os resultados da tabela 17, mostram a quantidade de memória de armazenamento consumida pelas aplicações desenvolvidas com ambos os *frameworks*. Os resultados foram obtidos após todas cinco execuções da funcionalidade de criação de tarefas.

Tabela 17: Dados relativos a utilização de memória de armazenamento

	AngularJS	**Ember.js**
Quantidade em *bytes* na primeira execução	41	85
Quantidade em *bytes* na segunda execução	81	148
Quantidade em *bytes* na terceira execução	121	211
Quantidade em *bytes* na quarta execução	161	274
Quantidade em *bytes* na quinta execução	201	337

Fonte: Próprio autor.

De acordo com os dados apresentados na tabela 17, o AngularJS consome uma quantia consideravelmente menor de memória de armazenamento que o Ember.js. Na primeira execução, o Ember.js já estava ocupando mais que o dobro da memória consumida pelo AngularJS. Na quinta execução, ou seja, na quinta tarefa adicionada, o Ember.js consumia 40% mais memória de armazenamento que o Angular.JS, isso pode representar um grande problema em aplicações de grande porte. Dessa forma de acordo com a métrica o AngularJS se demonstrou ser melhor que o Ember.js no quesito de consumo de memória de armazenamento.

A quarta métrica de eficiência mede a utilização de memória RAM, os resultados obtidos através da aplicação da mesma são encontrados na seção 3.3.4.

3.3.4 Utilização de memória RAM

Os resultados apresentados pela métrica de utilização de memória RAM, foram medidos logo após a execução das funcionalidades de criação, edição e exclusão de tarefas.

A tabela 18, exibe resultados referentes a consumo de memória RAM da aplicação, após a mesma ter executado a funcionalidade de criação de tarefas.

Tabela 18: Dados sobre utilização de memória RAM na funcionalidade de criação de tarefa

	AngularJS	**Ember.js**
Quantidade em *bytes*	46696000	52948000

Fonte: Próprio autor.

De acordo com os dados apresentados na tabela 18, o AngularJS consumiu uma quantidade menor de memória RAM, sendo assim, o AngularJS é um *framework* melhor para desenvolvedores que procuram por *frameworks* que tenham a capacidade de inserir dados consumindo pouca memória RAM, sendo assim, conseguem ser executados em computadores mais defasados. Porém, isso não compromete o desempenho de aplicações que usem o Ember.js, pois a diferença de consumo de memória RAM é de apenas 12%. Levando também em consideração a quantidade de memória RAM disponível nos computadores ambos *frameworks* apresentaram bom consumo de memória RAM. Desse modo o AngularJS é o melhor *framework* de acordo com a métrica de utilização de memória RAM.

A tabela 19, exibe resultados referentes a consumo de memória RAM da aplicação, após a mesma ter executado a funcionalidade de edição de tarefas.

Tabela 19: Dados sobre utilização de memória RAM na funcionalidade de edição de tarefa

	AngularJS	**Ember.js**
Quantidade em *bytes*	36232000	60240000

Fonte: Próprio autor.

A tabela 19 mostra que o AngularJS consumiu menos memória RAM em relação ao Ember.js, representando 66% a menos. Esse resultado mostra que o Ember.js deverá consumir uma grande quantidade de memória RAM, podendo gerar travamentos e esgotamento de memória em aplicações grandes sendo executadas

em computadores com pouca quantidade de memória RAM. Desse modo o AngularJS se mostra como melhor *framework* de acordo com a métrica de utilização de memória RAM, sendo mais indicado para desenvolvedores que procuram um *framework* capaz editar dados consumindo pouca memória RAM.

A tabela 20, exibe resultados referentes a consumo de memória RAM da aplicação, após a mesma ter executado a funcionalidade de exclusão de tarefas.

Tabela 20: Dados sobre utilização de memória RAM na funcionalidade de exclusão de tarefa

	AngularJS	**Ember.js**
Quantidade em *bytes*	30540000	54544000

Fonte: Próprio autor.

De acordo com os dados exibidos na tabela 20, o Ember.js consumiu 79% mais memória RAM em relação a mesma aplicação desenvolvida em AngularJS, isso mostra novamente que a aplicação desenvolvida em Ember.js pode apresentar travamentos ao excluir muitos dados em computadores que possuam pouca memória RAM. Desse modo, o AngularJS é o melhor *framework* de acordo com a métrica de utilização de memória RAM, sendo mais indicado para desenvolvedores que procuram um *framework* capaz excluir dados consumindo pouca memória RAM.

Além disso, o AngularJS apresentou menor consumo de memória RAM nos três testes realizados, sendo assim ele é o *framework* que menos consome memória RAM, apresentando os melhores resultados de acordo com a métrica de utilização de memória RAM, isso pode resultar em aplicações executando de maneira melhor e conseguindo trabalhar com uma quantidade maior de dados sem apresentar travamentos.

A seção 3.3.5 possui os resultados e a análise dos mesmos em relação a métrica de utilização de transmissão de dados.

3.3.5 Utilização de transmissão de dados

Os resultados apresentados na tabela 21 mostram a quantidade de dados que são transmitidos pela rede até o navegador de internet, todos os dados transmitidos são necessários para o funcionamento correto da aplicação.

Tabela 21: Dados sobre utilização de transmissão de dados

	AngularJS	**Ember.js**
Quantidade em *bytes*	1173040	2386120

Fonte: Próprio autor.

Como é mostrado na tabela 21, o AngularJS apresentou resultados muito superiores em relação ao Ember.js, enquanto o Ember.js consumiu 2.3 MB de transmissão de dados, o AngularJS consumiu apenas 1.1 MB. Esse é um fator de muita importância se tratando de aplicações SPA, com o advento dos smartphones e com internet de baixa qualidade, uma baixa utilização de transmissão de dados, significa menos tempo para que a aplicação comece a funcionar corretamente, pois é necessário esperar até que o navegador baixe todos os arquivos para que a aplicação funcione corretamente. De acordo com Nielsen (2011), 47% dos usuários esperam que a aplicação carregue em 2 segundos e 40% vão deixar a aplicação se ela demorar mais de 3 segundos para carregar, ou seja, com esses valores uma aplicação AngularJS demora 51% menos para carregar do que uma mesma aplicação desenvolvida em Ember.js. O AngularJS é a melhor opção para desenvolvedores que querem que sua aplicação carregue em menos tempo.

Os resultados apresentados na seção 3.3.6, pertencem a última métrica relacionada a eficiência, a métrica de número de dependências de *software*.

3.3.6 Número de dependências de *software*

Os resultados apresentados na tabela 22, são referentes a número de dependências a outros *softwares* para que as aplicações em ambos os *frameworks* funcionem, o próprio *framework* não está incluído ao resultado.

Tabela 22: Dados sobre dependências de *software*

	AngularJS	**Ember.js**
Quantidade	2	4

Fonte: Próprio autor.

Como é mostrado na tabela 22, o AngularJS necessita de metade da quantidade de dependências que o Ember.js precisa. Isso justifica os resultados encontrados nas métricas de utilização de memória RAM e utilização de transmissão

de dados, pois, uma maior quantidade de softwares como dependência consumirão mais memória RAM, além disso para transferir essas dependências será necessário mais transmissão de dados. Isso mostra que o Ember.js precisa de mais *softwares* que sejam capazes de complementar suas funções, desse modo o AngularJS se mostra mais completo em número de funcionalidades, por conseguir executar uma mesma aplicação sem a necessidade de muitos *softwares* para complementar suas funções. De acordo com a métrica número de dependências de *software*, o AngularJS é melhor por precisar de um menor número de dependências.

De todas as métricas de eficiência analisadas, o AngularJS apresentou resultados superiores em cinco, enquanto o Ember.js apresentou bons resultados apenas na métrica de tempo de processamento, desse modo com base nos resultados obtidos nesse trabalho o AngularJS é o melhor framework, referente a característica de eficiência.

A seção 3.4 apresentará os resultados das métricas que estão relacionadas a característica de manutenibilidade.

3.4 MÉTRICAS DE MANUTENIBILIDADE

De acordo com a ISO/IEC 9126-3, os resultados dessa característica medem a capacidade do produto de *software* de ser modificado com facilidade por outros desenvolvedores.

3.4.1 Mudança no código-fonte

Essa métrica tem como objetivo medir o quão entendível são as alterações no código-fonte do *framework*.

Os resultados apresentados abaixo foram obtidos no site GitHub, foram usadas as dez últimas alterações no código-fonte do *framework* para calcular os resultados da métrica, que são apresentados na tabela 23.

Tabela 23: Dados sobre mudança de código-fonte

	AngularJS	**Ember.js**
Quantidade de alterações com comentário	6	3

Fonte: Próprio autor.

Os resultados apresentados na tabela 23 foram capturados no dia 1 de novembro de 2015. Das últimas dez alterações, o AngularJS possui 100% mais alterações comentadas do que o Ember.js, esse é um fator importante para futuras alterações no código-fonte do *framework*. Existe uma possibilidade de que, a próxima pessoa a realizar uma alteração no mesmo trecho de código-fonte não comentado, não compreenda a utilidade do código-fonte alterado e venha a removê-lo ou alterá-lo de maneira incorreta, gerando problemas para outros desenvolvedores que usam o código-fonte que foi alterado. Desse modo, o AngularJS apresentou um melhor resultado em relação ao Ember.js, porém o ideal é que ambos possuam comentários em 100% de suas alterações.

Na seção 3.5 serão apresentados resultados referentes as métricas de portabilidade.

3.5 MÉTRICAS DE PORTABILIDADE

De acordo com a ISO/IEC 9126-3, os resultados apresentados pelas métricas de portabilidade, mostram a capacidade do *framework* de ser instalado ou transportado para vários ambientes diferentes.

3.5.1 Flexibilidade de instalação

Os resultados da métrica de flexibilidade de instalação são capazes de determinar qual *framework* apresentam um número de possibilidades de instalação. Os resultados na tabela 24 apresentam as possibilidades de instalação do

framework de acordo com o fornecido na documentação.

Tabela 24: Dados sobre flexibilidade de instalação

	AngularJS	**Ember.js**
Possibilidades	3	2

Fonte: Próprio autor.

De acordo com o que foi apresentado na tabela 24, o AngularJS pode ser instalado de três formas diferentes, através de download do seu arquivo principal, através de um projeto base, que pode ser usado para criar outras aplicações e também realizando a inclusão do arquivo principal do *framework* que fica armazenado em um servidor externo.

Como mostra a tabela 24, o Ember.js oferece apenas duas formas de instalação, uma é instalando o Ember CLI que é uma ferramenta disponibilizada pelo *framework,* e que possibilita a automatização de algumas tarefas, a outra forma pode ser encontrada em uma versão antiga da documentação, que é instalar o Ember.js através do Bower – o Bower é um gerenciador de pacotes para a *web*, que permite não só a instalação do Ember.js, mas de vários outros *frameworks* e ferramentas - mesmo se encontrando em uma versão antiga da documentação, essa ainda é uma forma alternativa de se instalar o *framework*.

Existem diversas outras formas de se instalar ambos *frameworks* porém as mesmas não estão descritas na documentação, o que não garante o seu funcionamento correto. Ambos *frameworks* apresentam um pequeno número de possibilidades de instalação, porém, AngularJS apresenta uma maior possibilidade de instalação, sendo assim de acordo com a métrica o AngularJS é o melhor.

A conclusão do presente trabalho apresenta uma conclusão geral sobre os resultados obtidos através das métricas da ISO/IEC 9126-3.

CONCLUSÃO

Com base nos resultados desse trabalho, que teve como objetivo avaliar e comparar os *frameworks* de *single page application* AngularJS e Ember.js, de acordo com as métricas de qualidade interna da ISO/IEC 9126-3, foi possível concluir, que das métricas aplicadas, o Ember.js apresentou bons resultados apenas nas métricas de remoção de falhas, acessibilidade física e tempo de resposta, já o AngularJS apresenta bons resultados em acessibilidade física, tempo de resposta, utilização de armazenamento, utilização de memória RAM, utilização de transmissão de dados, número de dependências de software, mudança no código-fonte e flexibilidade de instalação.

As diferenças mais expressivas foram encontradas nas análises de eficiência. Como foi comprovado, o AngularJS mostra-se mais rápido ao executar tarefas separadamente, já ao executar várias tarefas consecutivas o Ember.js mostra-se como melhor, uma inferência que pode ser feita sobre esses resultados é que o Ember.js é menos eficiente ao gerar cacheamento de seu código-fonte. Com isso, o AngularJS é mais indicado a aplicações que executem poucas tarefas ao mesmo tempo, normalmente aplicações dedicadas a usuários domésticos, já o Ember.js torna-se mais indicado a aplicações que trabalham com muitas operações de dados ao mesmo tempo, normalmente aplicações profissionais e especialistas, que precisam gerar gráficos e relatórios, trabalham com dados acumulados por um longo período de tempo.

A dificuldade do Ember.js em gerar cacheamento do próprio código-fonte, se deve a ele possuir um grande número de dependências de *software*, como foi verificado na métrica número de dependências de *software*, o Ember.js possui 100% mais dependências de outros *softwares* em relação ao AngularJS, isso torna a execução do Ember.js mais complexa, pois ele possui muitos outros *softwares* para gerenciar, além de si mesmo.

Outro fator que corroborou para aumentar a dificuldade em gerar cacheamento da aplicação que usa Ember.js, é que nos resultados das métricas de utilização de transmissão de dados, utilização de armazenamento e consumo de memória RAM, o Ember.js consumiu 103%, 40% e 79% mais recursos,

respectivamente, em relação ao AngularJS, com mais recursos de memória sendo consumidos, sobram menos recursos para geração de cacheamento, além de sobrar menos recursos, o cacheamento da aplicação construída em Ember.js consome mais memória.

Os resultados apresentados pelas métricas de consumo de memória RAM, utilização de transmissão de dados, utilização de armazenamento e dependências de software, mostram que o Ember.js possui um código-fonte mais complexo para realizar as mesmas tarefas em relação ao AngularJS, além disso ele precisa de outros *softwares* para complementar suas funcionalidades, o que torna o *framework* ainda mais complexo, já o contrário pode ser verificado no AngularJS, que precisa de menos recursos para executar as mesmas tarefas. De acordo com os resultados obtidos com esse trabalho, o AngularJS possui mais funcionalidades implementadas em seu código-fonte e mesmo assim consome menos recursos de memória e transmissão de dados.

Com as métricas de remoção de falhas e mudança de código-fonte, percebe-se que mesmo o AngularJS removendo uma menor proporção de falhas, as suas alterações no código-fonte, que incluem novas funcionalidades e remoções falhas, são comentadas, o que facilita o intendimento do código-fonte para outros desenvolvedores que forem realizar outras alterações.

As métricas de acessibilidade física e flexibilidade de instalação, fazem com que as aplicações que usam os *frameworks* estudados, possam ser levados a uma quantidade maior de ambientes, no caso da acessibilidade física ambos *frameworks* podem ser usados em aplicações destinadas a pessoas com deficiências físicas, porém, o AngularJS apresenta maior flexibilidade de instalação, podendo ser levado a uma quantidade maior de ambientes.

Sendo assim, de acordo com os resultados obtidos nesse trabalho, o AngularJS apresenta os melhores resultados nas características de eficiência, manutenibilidade e portabilidade, já o Ember.js apresenta melhores resultados na característica de confiabilidade, além disso, ambos os *frameworks* apresentaram bons resultados na característica de usabilidade.

Desse modo, de acordo com os resultados obtidos nesse trabalho, o *framework* que apresenta melhores resultados em um número maior de métricas e características é o AngularJS. Porém, para que o desenvolvedor escolha o melhor *framework* ele deve analisar os resultados desse trabalho, então de acordo com os

seus requisitos de *software* escolher o *framework* que melhor se adapta ao seu projeto.

Com isso podemos concluir que, esse trabalho foi eficiente em realizar a comparação nas métricas selecionadas, atingindo os objetivos esperados em seu planejamento. Portanto, a qualidade se mostra como um fator determinante no sucesso e na popularidade de um *framework*.

TRABALHOS FUTUROS

Como sugestão de trabalhos futuros, seria interessante a aplicação de mais métricas de qualidade interna contidas na ISO/IEC 9126-3. Mais métricas avaliadas, servirão para que outros desenvolvedores tenham ainda mais aspectos para levar como base no momento de escolher o *framework* para o desenvolvimento de uma aplicação.

Outra sugestão, seria a criação de um método de avaliação dedicado aos *frameworks* de *single page applications*, por mais que a ISO/IEC 9126 possa ser aplicada a qualquer tipo de *software*, algumas adaptações foram necessárias. A criação de um método de avaliação poderia gerar resultados ainda mais precisos e específicos.

Mais uma sugestão seria avaliar uma gama maior de *frameworks* para *single page application* ou então avaliar outros *frameworks* de *single page application*, com o objetivo de descobrir se mesmo com menor popularidade os demais *frameworks* também possuem uma qualidade equiparável ao AngularJS e ao Ember.js.

REFERÊNCIAS

AGBOADO, Precious Jahlom. **Ember.js Guides**. 1ª ed. Vancouver: Leanpub, 2014.

ANGULAR. **angular-seed — the seed for AngularJS apps**. Disponível em: <https://github.com/angular/angular-seed>. Acesso em 8 de out. de 2015.

CANNINGS, Rich; DWIVEDI, Himanshu; LACKEY, Zane. **Hacking Exposed – Web 2.0**. 1ª ed. New York: McGraw-Hill Osborne Media, 2008.

CONSORTIUM, World Wide Web. **Accessible Rich Internet Applications (WAI-ARIA) 1.0**. Disponível em: <http://www.w3.org/TR/wai-aria/>. Acesso em 1 de nov. de 2015

ELLIOTT, Eric. **Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries**. 1ª ed. Sebastopol: O'Reilly Media, 2014.

EMBER.JS. **Ember.js - A framework for creating ambitious web applications**. Disponível em: <http://emberjs.com/>. Acesso em 8 de out. de 2015.

EMBER.JS. **Ember.js - Ember.AriaRoleSuport**. Disponível em: <http://emberjs.com/api/classes/Ember.AriaRoleSupport.html>. Acesso em 1 de nov. de 2015.

EMBER.JS. **Issues - emberjs/ember.js**. Disponível em: <https://github.com/emberjs/ember.js/issues>. Acesso em 1 de nov. de 2015.

FINK, Gil; FLATOW, Ido. **Pro Single Page Application Development**. 1ª ed. New York: Apress, 2014.

FLANAGAN, David. JavaScript: **The Definitive Guide**. 6ª ed. Sebastopol: O'Reilly Media, 2011.

GITHUB. **GitHub**. Disponível em: <https://github.com>. Acesso em: 21 de jun. de 2015.

GOOGLE. **AngularJS – Superheroic JavaScript MVW Framework**. Disponível em: <https://angularjs.org/>. Acesso em 8 de out. de 2015.

GOOGLE. **AngularJS: Developer Guide: Accessibility**. Disponível em: <https://docs.angularjs.org/guide/accessibility>. Acesso em 1 de nov. de 2015.

GOOGLE. **Issues - angular/angular.js**. Disponível em:

<https://github.com/angular/angular.js/issues>. Acesso em 1 de nov. de 2015.

GUERRA, Ana Cervigni; COLOMBO, Regina Maria Thienne. **Tecnologia da Informação: qualidade de produto de software**. 1ª ed. Brasilia: PBQP, 2009.

HILDRETH, Sue. **Buggy Software: Up From a Low-Quality Quagmire**. Disponível em: <http://www.computerworld.com/article/2557403/app-development/buggy-software-up-from-a-low-quality-quagmire.html>. Acesso em: 26 de mar. 2015.

LEVINSON, Meridith. **SOFTWARE DEVELOPMENT - Let's Stop Wasting $78 Billion a Year**. Disponível em: <http://www.cio.com/article/2441228/enterprise-software/software-development---let-s-stop-wasting--78-billion-a-year.html>. Acesso em 26 de mar. 2015.

MAZZA, Lucas. **HTML5 e CSS3**.1ª ed. São Paulo: Casa do Código, 2012.

MESSENLEHNER, Brian; COLEMAN, Jason. **Criando Aplicações Web com WordPress**. 1ª ed. São Paulo: Novatec, 2004.

MINETTO, Luís Elton. **Frameworks para Desenvolvimento em PHP**. 1ª ed. São Paulo: Novatec, 2007.

NIELSEN, Jakob. **How Long Do Users Stay on Web Pages?**. Disponível em: <http://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/>. Acesso em 1 de nov. de 2015.

OGEDEN, Max. **JavaScript for Cats**. Disponível em: <http://jsforcats.com/>. Acesso em: 7 de out. de 2015.

OSMANI, Addy, et al. **TodoMVC**. Disponível em: <http://todomvc.com/>. Acesso em 1 de nov. de 2015.

PEREIRA, Michael Henrique R. **AngularJS: Uma abordagem prática e objetiva**. 1ª ed. São Paulo: Novatec, 2014.

PRESSMAN, Roger S. **Engenharia de software, uma abordagem profissional**. 7ª ed. New York: The McGraw-Hill Companies, 2011.

ROBBINS, Charlie. **http-server: a command-line http server**. Disponível em: <https://github.com/indexzero/http-server>. Acesso em: 8 de out. de 2015.

SELLE, Pam, et al. **Choosing a JavaScript Framework**. 1ª ed. Santa Rosa: Bleeding Edge Press, 2014.

SESHADRI, Shyam; GREEN, Brad. **AngularJS: Up ang Running**. 1ª ed. Sebastopol: O'Reilly Media, 2014.

SOMMERVILLE, Ian. **Engenharia de Software**. 9ª ed. São Paulo: Pearson

Prentice Hall, 2011.

STANDARDIZATION, INTERNETIONAL ORGANIZATION FOR. **ISO/IEC 9126-1:2002**. 1ª ed. Genebra: INTERNETIONAL ORGANIZATION FOR STANDARDIZATION, 2002.

STANDARDIZATION, INTERNETIONAL ORGANIZATION FOR. **ISO/IEC 9126-3:2002**. 1ª ed. Genebra: INTERNETIONAL ORGANIZATION FOR STANDARDIZATION, 2002.