UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

BACHARELADO EM ENGENHARIA DE SOFTWARE

LAÉRCIO GERMANO DE OLIVERA JÚNIOR

ATLOM.JS: UM FRAMEWORK NODE.JS PARA APLICAÇÕES WEB BASEADO EM COMPONENTES

QUIXADÁ – CEARÁ

2017

LAÉRCIO GERMANO DE OLIVERA JÚNIOR

ATLOM.JS: UM FRAMEWORK NODE.JS PARA APLICAÇÕES WEB BASEADO EM COMPONENTES

Monografia apresentada no curso de Engenharia de Software da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Engenharia de Software. Área de concentração: Computação.

Orientador: Dra. Carla Ilane Moreira Bezerra

Coorientador: Me. Victor Aguiar Evangelista de Farias

QUIXADÁ – CEARÁ

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

J1a Oliveira Júnior, Laércio Germano de.
ATLOM.JS: um Framework NODE.JS para aplicações Web baseado em componentes. / Laércio Germano de Oliveira Júnior. – 2017.
67 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2017.
Orientação: Prof. Dr. Carla Ilane Moreira Bezerra.
Coorientação: Prof. Me. Victor Aguiar Evangelista de Farias.

1. Framework (Programa de computador). 2. Aplicações Web. 3. Javascript (Linguagem de programação de computador). I. Título.

CDD 005.1

LAÉRCIO GERMANO DE OLIVERA JÚNIOR

ATLOM.JS: UM FRAMEWORK NODE.JS PARA APLICAÇÕES WEB BASEADO EM COMPONENTES

Monografia apresentada no curso de Engenharia de Software da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Engenharia de Software. Área de concentração: Computação.

Aprovada em:____/___/___.

BANCA EXAMINADORA

Dra. Carla Ilane Moreira Bezerra (Orientador)
Universidade Federal do Ceará – UFC

Me. Victor Aguiar Evangelista de Farias (Coorientador)
Universidade Federal do Ceará - UFC

Dr. Jefferson de Carvalho Silva
Universidade Federal do Ceará - UFC

A minha querida mãe, Hosana Valentim. Aos meus irmãos, Alysson Melo, Lorna Melo e Ian Melo. Ao meu avô, Antônio Valentim.

AGRADECIMENTOS

Primeiramente, quero agradecer à minha amada mãe, Hosana Valentim, não exclusivamente pela dedicação em me oferecer as melhores oportunidades e recursos, mas por ter sido a minha principal referência de vida, de mãe e de pai. Uma mulher símbolo de força, inteligência e honestidade. E que apesar do tempo e das batalhas travadas, ainda possui a beleza, o brilho e a pureza de uma criança.

Agradeço à Marcionília Souza, pelo carinho e amor recebidos, pelos seus cuidados que me ampararam quando precisei, e pelos dias felizes que compartilhamos juntos.

Agradeço à professora Carla Ilane, pela sua excelente orientação, que diante de todas as minhas dificuldades, conseguiu de forma sábia e paciente me conduzir minuciosamente em cada passo deste tão importante trabalho, tornando o sonho da graduação cada vez mais próximo.

Agradeço à professora Tânia Saraiva, por todas as nossas conversas, conselhos e cobranças, que de forma suave e sempre esbanjando um sorriso no rosto, me convenceu da importância da graduação, sendo primordial para que eu não desistisse e continuasse seguindo adiante.

Agradeço ao professor Carlos Diego, pela parceria e por todos os seus ensinamentos passados durante as suas aulas, e além disso, por ter sido mentor durante a coordenação do projeto de estágio, contribuindo significativamente para o desenvolvimento pleno deste projeto.

Agradeço ao professor Victor Farias, por ter aceitado o desafio de ser o meu co-orientador, contribuindo com seu tempo, experiência e conhecimento técnico no desenvolver deste trabalho. Agradeço à Laisa Morais, não exclusivamente por me auxiliar na correção do texto, mas pela sua amizade, conselhos e pelo interesse de me fazer enxergar a importância de concluir a minha trajetória acadêmica para o meu crescimento profissional.

Agradeço aos meus amigos Mário Falcão, Matheus Medeiros, Felype Tabosa, Pedro Sávio, Isabelly Damasceno, Paulo Júnior e Ramon Malveira, que de alguma forma contribuíram para para a minha formação acadêmica, e por todos os momentos memoráveis que passamos juntos.

"*Todo mundo é um gênio. Mas se você julgar um peixe pela sua habilidade de subir em árvores, ele viverá o resto de sua vida acreditando ser um tolo*"

(Albert Einstein)

RESUMO

Atualmente, existem muitos *frameworks* de aplicação Node.js criados com o intuito de otimizar o processo de desenvolvimento de *software*, onde cada *framework* possui as suas próprias tecnologias e padrões de implementação. Essas características adicionam uma maior complexidade ao sistema, acarretando em cadeia algumas consequências negativas, como a diminuição do número de adeptos, diminuição de componentes desenvolvidos para a plataforma e a diminuição do potencial reuso de *software*.Neste trabalho, é proposto o desenvolvimento de um *framework full-stack* Javascript baseado em componentes, denominado Atlom.js, que visa diminuir a curva de aprendizagem, bem como, otimizar o processo de reuso de *software*. Para alcançar tal desafio, foi proposto adaptar para o Node.js, tecnologia que atua no *back-end*, o modelo de injeção de dependências (DI) e inversão de controle (IoC) encontrados no *framework* AngularJS. Desta forma, é possível desenvolver uma aplicação completa utilizando a mesma linguagem e o mesmo modelo de desenvolvimento. O trabalho foi implementado com base nas funcionalidades dos *frameworks* mais atuais do mercado, utilizando conceitos e tecnologias mais conhecidas de Javascript. Seu principal desafio foi elaborar sua DI contendo os principais métodos de sua abordagem original. Com os resultados obtidos por meio de experimentos com especialistas, foi constatado que o *framework* diminui a curva de aprendizagem, otimiza o processo de reuso e proporciona a customização das tecnologias associadas.

Palavras-chave: Framework (Programa de computador). Aplicações Web. Javascript (Linguagem de programação de computador)

ABSTRACT

Currently many Node.js application frameworks with specific technologies and implementation standards are designed to optimize the software development process. Such unique characteristics add greater complexity to the system, resulting in a chain of negative consequences, such as the decrease in the number of users, decrease in the components developed for the platform and reduction of potential software reuse. In this work the development of a full-stack component- based framework called Atlom.js was proposed, aiming to reduce the learning curve, as well as to optimize the software reuse process. To achieve this goal, technologies operating in the back-end, in the dependency injection (DI) and in the control inversion (IoC) model found in the AngularJS framework, were proposed to adapt to the Node.js. In this way, it was possible to develop a complete application using the same language and the same development model. The work was implemented based on the functionality of the latest frameworks on the market, using well-known concepts and technologies of Javascript. The biggest challenge was to elaborate the DI containing the main methods of the original approach. Results obtained through experiments with specialists demonstrated the framework reduced the learning curve, optimized the reuse process and provided associated technologies customization.

Palavras-chave: Framework (Computer program). Web Application. Javascript (Computer programming language)

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

DBCDesenvolvimento Baseado em Componentes

MEANMongoDB, Express, Angular.js e Node.js

CRUDCreate, Read, Update and Delete

DIDependency Injection

IoCInversion of Control

I/OInput e Output

CPUCentral Process Unit

IoTInternet of Things

MVCModel-View-Controller

AJAXAsynchronous Javascript and XML

HTTPHyperText Transfer Protocol

HTMLHyperText Markup Language

DOMDocument Object Model

ORMObject Relational Mapping

URLUniform Resource Locator

JSONJavaScript Object Notation

SQLStructured Query Language

NoSQLNot Only SQL

NPMNode Package Manager

RESTRepresentational State Transfer

DBDatabase

VPSVirtual Private Server

IDEIntegrated Development Environment

SUMÁRIO

1 INTRODUÇÃO

Desde o princípio do desenvolvimento de software, pesquisadores e profissionais tem procurado por métodos, técnicas e ferramentas que permitam melhorias no custo, tempo e qualidade no desenvolvimento de software. Alguns avanços da Engenharia de Sofware, com foco no reuso, têm contribuido para que essas melhorias aconteçam, tais como a programação orientada a objeto, desenvolvimento baseado em componentes, linhas de produtos de software, entre outros (ALMEIDA et al., 2007).

O Desenvolvimento Baseado em Componentes (DBC) promove a construção de software a partir do reuso de componentes existentes, principalmente pela montagem e substituição dessas partes interoperáveis. Assim, um único componente pode ser reusado em muitas aplicações, reduzindo tempo, custo e esforço no desenvolvimento do projeto, além de minimizar a probabilidade de erros, uma vez que os componentes são testados em seu processo de criação, promovendo também o aumento da qualidade do sistema (TIWARI; CHAKRABORTY, 2015).

Alguns estudos sobre reuso tem mostrado que 40% a 60% do código pode ser reutilizado de uma aplicação para outra, 75% das funcionalidades dos sistemas são comuns para mais de um programa. Apenas 15% do código encontrado na maioria dos sistemas são únicos e novos para uma aplicação específica. As taxas atuais do potencial reuso variam entre 15% a 85% (ALMEIDA et al., 2007).

Para obter maiores taxas do reuso, o desenvolvedor deveria ser capaz de utilizar os melhores componentes disponíveis sem pensar em qual modelo de componentes foi utilizado para implementá-los. Para alcançar esse desafio, foi proposto o desenvolvimento de software utilizando *frameworks*, que padronizam o processo de implementação, otimizando a construção e o reuso efetivo de componentes (ALMEIDA et al., 2007).

Na indústria existem diversos *frameworks* web desenvolvidos em diferentes linguagens de programação, tais como: Ruby on Rails [1] (Ruby), Laravel [2] (PHP), Spring [3] (Java) , Django [4] (Python), entre outros. Com o surgimento do Node.js [5], plataforma construída sobre o motor Javascript do Chrome (V8), tornou-se possível criar sistemas web complexos

[1] http://rubyonrails.org/
[2] https://laravel.com/
[3] https://spring.io
[4] https://www.djangoproject.com/
[5] https://nodejs.org

utilizando a mesma linguagem em todos os ângulos do desenvolvimento, tais como: *back-end*, *front-end*, pré processadores, automatizadores de tarefas, dentre outros.

Assim, Node.js se tornou a tecnologia de desenvolvimento mais significante e com maior crescimento nos últimos anos, tornando-se uma plataforma universal para o desenvolvimento de aplicações web *real-time*, *mobile*, desktop, microsserviços e internet das coisas (JOYENT, 2012). A combinação de MongoDB [6], Express [7], AngularJS [8] e Node.js, popularmente conhecida como a *stack* MEAN, tornaram-se as principais tecnologias utilizadas neste contexto por apresentarem altos índices de escalabilidade comparadas as outras tecnologias (JOYENT, 2012).

Muitos *frameworks* de desenvolvimento de aplicações web Node.js surgiram nos últimos anos, entre os mais populares estão: Meteor [9], Sails.js [10] e Mean.io [11]. Todos esses possuem suas próprias características e padrões de desenvolvimento. Logo, reutilizar componentes desenvolvidos em *frameworks* distintos se torna uma tarefa inviável, uma vez que os componentes necessitarão passar por um processo de adaptação, elevando consideravelmente o nível de esforço no desenvolvimento.

O presente trabalho visou desenvolver um *framework* web Node.js baseado em componentes, denominado Atlom.js, que tem como foco padronizar o processo de desenvolvimento de novas aplicações *web*, utilizando os *frameworks* mais populares da atualidade como referência. Além disso, foi adaptado para o contexto *back-end* a injeção de dependências (DI) do AngularJS. Desta forma, será possível desenvolver aplicações web utilizando a mesma linguagem, bem como o mesmo padrão de modelo de componentes nos lados cliente e servidor.

Espera-se com este trabalho contribuir para a comunidade *open-source*, especificamente para os desenvolvedores Javascript, com um *framework* de desenvolvimento capaz de ser adaptado a diferentes contextos e necessidades, provendo um ambiente favorável a utilização, disponibilização e a construção de novos componentes, otimizando o processo de desenvolvimento de software.

---

6 https://www.mongodb.com/
7 http://expressjs.com/
8 https://angularjs.org/
9 https://www.meteor.com/
10 https://sailsjs.com/
11 http://mean.io/

2 TRABALHOS RELACIONADOS

Em Stanojevic´ et al. (2011) foi desenvolvido um *framework*, chamado de Oz, com o intuito de gerar aplicações desktop baseadas em um modelo *metadata* relacional. O *framework* deveria, com base em problemas arbitrários de domínios representados por um metamodelo, gerar esqueletos de aplicações usando uma arquitetura de 3 camadas, e então implementar um CRUD básico para um domínio específico. Na primeira parte do seu trabalho, são apresentadas as definições e as propriedades do processo de desenvolvimento do *framework*. Na segunda, as diretrizes para o planejamento e a sua implementação. O trabalho conclui com a sumarização dos dados coletados para criação de uma nova metodologia de desenvolvimento para *frameworks*.

As definições e propriedades do processo de desenvolvimento do *framework* vão servir de apoio para definirmos um processo semelhante e utilizarmos neste presente trabalho. Porém, o intuito deste é desenvolver um *framework* web voltado ao reuso de componentes.

Meteor, Sails.js e Mean.io, levando em consideração o número de *stars* (modelo de ranqueamento) do Github [1], são os *frameworks* Javascript *full-stack* mais populares da atualidade, cujo foco é o desenvolvimento de aplicações web modernas escalares. Eles incluem um conjunto de tecnologias para a criação de aplicações *real-time*, ferramentas de desenvolvimento e uma série de pacotes Node.js providos pela comunidade Javascript.

Meteor, assim como o Atlom.js, integra como padrão o banco de dados MongoDB em sua *stack*. Porém, o Meteor substitui o Express por um *framework* próprio como servidor HTTP. Além disso, também substitui o AngularJS como *framework front-end* por uma *view engine* própria denominada Blaze [2]. Desta forma, a curva de aprendizagem do usuário se torna elevada, uma vez que tecnologias restritas a um escopo específico necessitam ser estudadas, além do conhecimento obtido não ser reaproveitado em outros contextos.

Mean.io possui sua *stack* restrita a tecnologias específicas, impossibilitando a sua substituição. Diferente disso, o Atlom.js permitirá o usuário desenvolver e incorporar componentes de tecnologias diferentes para o projeto. Todas as partes do *framework* Atlom.js se comportarão como componentes interoperáveis. Desta forma, as tecnologias poderão ser facilmente substituidas por outras, tornando a *stack* do projeto customizável, desde o banco de dados, *frameworks* até *view engines*.

Sails.js possui uma feature, chamada de *Hooks*, que permite o programador

[1] https://github.com/
[2] http://blazejs.org/

desenvolver códigos reutilizáveis. Porém, possuem apenas regras de negócio restritas ao lado do servidor. Com o Atlom.js, por meio do seu modelo de componentes, será possível desenvolver componentes *full-stack* que encapsulam lógicas de programação dos lados cliente e servidor, otimizando ainda mais o reuso no processo de desenvolvimento de novas aplicações web.

Diferente dos demais trabalhos, Atlom.js visa trazer a injeção de dependênias do AngularJS, fortemente utilizada em aplicações *front-end*, também para o contexto do *back-end*. Assim, será possível padronizar o modelo de desenvolvimento, utilizando as mesmas tecnologias e conhecimentos em toda a aplicação, diminuindo o esforço e a curva de aprendizagem do usuário final.

3 FUNDAMENTAÇÃO TEÓRICA

Nesta Seção serão descritos os principais conceitos para o entendimento destem rabalho. Serão apresentados os principais conceitos sobre Desenvolvimento Baseado em Componentes: Componentes, *Frameworks* e a plataforma Node.js.

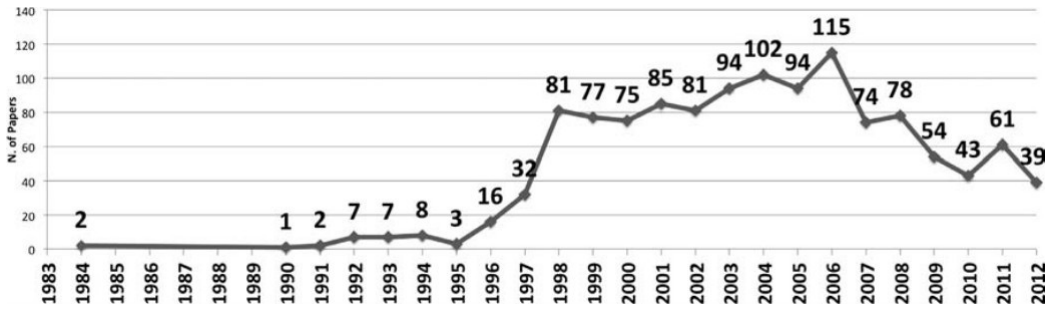
3.1 Desenvolvimento Baseado em Componentes

O Desenvolvimento Baseado em Componentes (DBC) promove o desenvolvimento de software a partir de componentes existentes, como também do desenvolvimento de novos componentes como entidades reusáveis; cuja evolução do sistema é realizada pela customização e substituição dessas partes interoperáveis (TIWARI; CHAKRABORTY, 2015).

Na literatura encontram-se diversas outras definições, para o DBC. D'Souza e Wills (1999) afirmam que é a técnica de desenvolvimento de software na qual todos os artefatos, desde o código executável até as especificações de interfaces, arquiteturas e modelos de negócio, podem ser construídos pela combinação, adaptação e interconexão de componentes existentes. Sametinger (1997), por sua vez, define como uma abordagem de desenvolvimento de software cujo objetivo é construir novas soluções por meio da combinação de partes já existentes, que foram concebidas para serem substituíveis, reusáveis e interoperáveis em uma infra-estrutura de execução específica.

O conceito do DBC não é novo, sua ideia é discutida desde os anos 60 pelo McIlroy (1968) que fornece uma ideia de produzir componentes comerciais similares aos encontrados em outros campos de engenharias. Entretanto, a maioria dos estudos relacionados ao DBC têm surgido somente nos últimos 20 anos. A motivação por trás do DBC tem natureza comercial e técnica, possuindo como principais interesses de estudo o aumento da produtividade, qualidade e a diminuição do tempo no processo de desenvolvimento de software. A Figura 11 ilustra o total de 1231 publicações espalhadas entre os anos 1984 e 2012 (VALE et al., 2016).

Figura 1 – Número de publicações por ano



Fonte: Vale et al. (2016)

Desenvolver software a partir de componentes existentes reduz tempo, custo e esforço no desenvolvimento do projeto, além de minimizar a probabilidade de erros, uma vez que os componentes são testados em seu processo de criação, promovendo o aumento da qualidade do sistema (TIWARI; CHAKRABORTY, 2015). O objetivo do DBC é prover a criação de componentes, assegurar a reutilização futura, dando suporte a sistemas desenvolvidos por meio da customização e substituição desses componentes (TAHIR et al., 2016).

Segundo Sommerville (2010), pode considerar no desenvolvimento baseado em componentes o desenvolvimento para reuso e o desenvolvimento com reuso. A primeira perspectiva se interessa no desenvolvimento de componentes ou serviços que podem ser reusados em outras aplicações, tendo domínio e acesso ao seu código fonte. A segunda perspectiva considera a utilização de serviços ou componentes existentes para a criação de novas aplicações, sem o conhecimento de sua disponibilidade, projetando o sistema para um reuso mais eficiente.

Por se tratarem de processos com objetivos distintos, suas atividades também se tornam diferentes. No desenvolvimento para reuso, a criação de componentes necessita da adaptação e a extensão dos componentes específicos da aplicação para criar versões mais genéricas, e portanto, mais reusáveis. Essa adaptação pode ter um custo elevado. Assim, é necessário levar em consideração se o componente é suscetível de ser reusado, e se a economia de custos do reuso futuro justifica o seu desenvolvimento. As adaptações necessárias para tornar um componente mais reusável inclue (SOMMERVILLE, 2010):

- Remoção de métodos específicos da aplicação;
- Formulação de nomes genéricos;
- Adicionar métodos para o fornecimento de uma cobertura funcional mais completa;

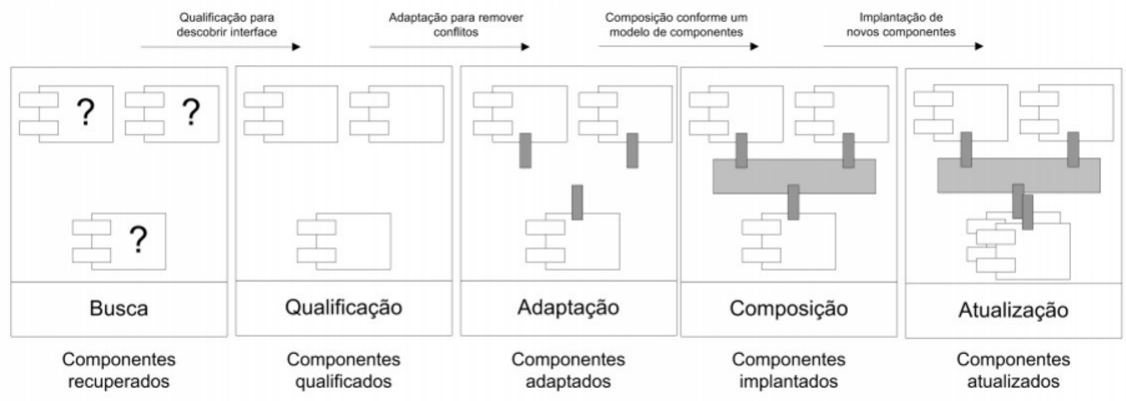
- Tornar o tratamento de exceções mais consistente nos métodos;
- Permitir que o componente possa ser adaptado por meio de uma interface de configuração.

Além do aspecto econômico, identificar o nível de reusabilidade do componente é um fator importante a ser considerado. Quanto mais genérico é um componente, maior suas chances de ser reusado. Porém, isso significa que ele apresentará mais métodos, será mais complexo, tornando mais difícil a sua compreensão e utilização (GIMENES; HUZITA, 2005).

Para se obter sucesso no desenvolvimento com reuso é necessário incluir atividades que encontrem e integrem componentes reusáveis. De um modo geral, esse processo ocorre em 5 etapas, conforme estabelecido por BROWN e K. (1997) e ilustrado na Figura 2:

- Busca: recupera os componentes existentes que atendem as necessidades do software em desenvolvimento;
- Qualificação: avalia detalhadamente os componentes encontrados na etapa de busca, de modo a definir qual será utilizado, levando em consideração a documentação disponível e a execução dos testes;
- Adaptação: realiza modificações nos componentes de modo a solucionar possíveis inconsistências entre os componentes que irão integrar o software;
- Implantação: realiza a implantação dos componentes no framework;
- Atualização: realiza a substituição de um componente por outro mais recente ou que apresente comportamento e interfaces equivalentes.

Figura 2 – Processo de desenvolvimento com reuso



Fonte: BROWN e K. (1997)

Neste trabalho, serão levados em consideração os principais fundamentos e boas práticas do DBC para a criação de um *framework* baseado em componentes com foco no reuso de

software, assumindo as atividades do processos de desenvolvimento com e para reuso conforme necessárias.

3.1.1 Componentes de software

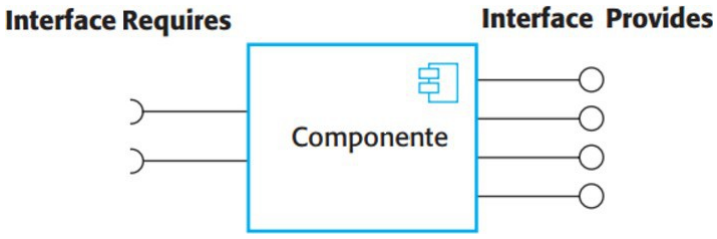
Considerado como o principal elemento do DBC, um componente de software pode ser definido como uma unidade de composição que encapsula sua implementação, oferecendo serviços ao ambiente externo por meio de interfaces bem definidas, podendo ser construído a partir de uma única classe ou ser tão complexo como um subsistema (GIMENES; HUZITA, 2005; SZYPERSKI; GRUNTZ; MURER, 2002).

Diversas definições de componentes são encontradas atualmente na literatura, e isso tende a dificultar acerca do que é um componente. Com o objetivo de auxiliar no entendimento da definição, Sommerville (2010) propôs algumas características essenciais que descrevem um componente de software, são elas:

- Padronizado: um componente precisa obedecer um modelo de componente padrão, podendo definir interfaces de composição, metadados de componentes, documentação, composição e implantação;
- Independente: deve ser possível implantar um componente sem a necessidade da composição de outros componentes específicos;
- Passível de composição: todas as interações externas devem ter lugar por meio de interfaces publicamente definidas;
- Implantável: deve ser capaz de operar como uma entidade autônoma em uma plataforma de componentes que obedeça a um modelo de componente;
- Documentado: os componentes devem ser devidamente documentados para que potenciais usuários possam decidir se satisfazem as suas necessidades.

Como mostra a Figura 3, os componentes podem fornecer dois tipos de interface: as interfaces fornecidas (*provides*), onde são definidos os métodos que podem ser chamados por outros componentes, e requeridas (*requires*), por onde o componente explicita suas dependências (TIWARI; CHAKRABORTY, 2015).

Figura 3 – Tipos de interfaces fornecidos pelo componentes.

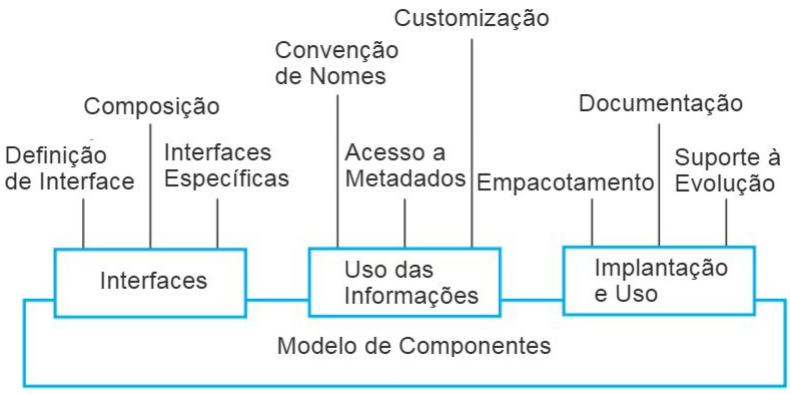


Fonte: Sommerville (2010), adaptado pelo autor.

Segundo Sommerville (2010), modelo de componentes é uma definição de padrões de implementação, documentação e implantação de componentes, cujo intuito é assegurar que os componentes possam interoperar entre si, além de oferecer suporte de infraestrutura, facilitando a integração entre eles. Os elementos básicos de um modelo de componentes, são:

- Interfaces: Especificação de como os elementos e as interfaces devem ser definidas, como nomes de operação, parâmetros e exceções, bem como a linguagem de programação adotada;
- Uso: Exclusividade de nomes ou identificadores associados, possibilitando a distribuição e acesso remoto dos componentes;
- Implantação: Especificação de como os componentes serão empacotados para a implantação como entidades independentes executáveis.

Figura 4 – Elementos básicos de um modelo de componentes



Fonte: Sommerville (2010), adaptado pelo autor.

Neste presente trabalho serão considerados os conceitos relacionados a componentes

de software com o intuito de desenvolver um modelo de componentes que respeite as definições e boas práticas descritas nesta seção, provendo uma interface sugestiva para o fácil compreendimento do usuário final.

3.2 Frameworks

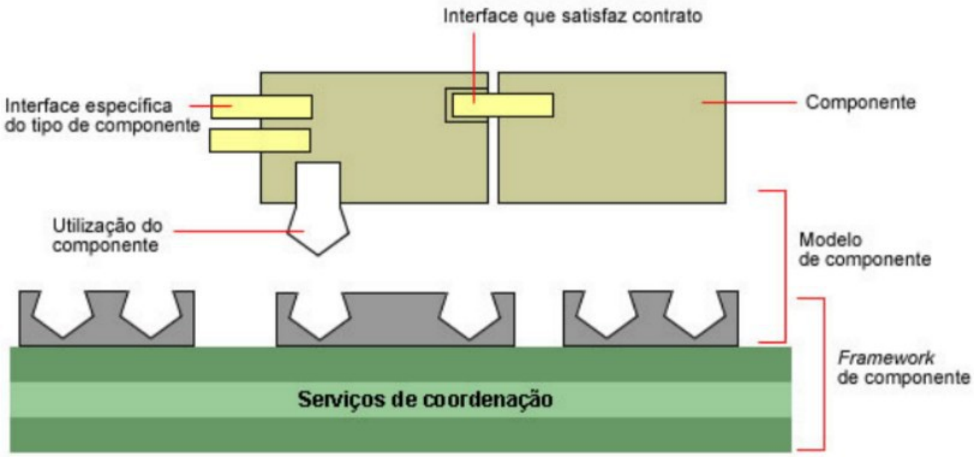
Um *framework* é uma aplicação semi-completa, construída a partir de uma coleção de componentes de softwares reusáveis que visam facilitar a criação de novas aplicações. É constituído por construtores básicos e pontos de extensão (*hot-spots*) que devem ser adaptados para o funcionamento específico de um produto (FAYAD; SCHMIDT, 1997).

Além disso, os *frameworks* possuem características que não podem ser modificadas por seus usuários. Esses pontos, chamados de *frozen spots*, são imutáveis e implementam suas decisões arquiteturais, estando sempre presentes em todas as instâncias do *framework* (MARKIEWICZ; LUCENA, 2001).

Existem diversas formas de se classificar os *frameworks*, dentre as principais estão os *frameworks* de aplicação orientado a objetos e *frameworks* de componentes (FAYAD; SCHMIDT, 1997), principal foco deste trabalho.

O *framework* de componente é uma infraestrutura que provê um conjunto de elementos de software, regras e contratos que governam a execução de componentes que estão em conformidade com um modelo (FAYAD; SCHMIDT, 1997). As definições estabelecidas pelo modelo de componentes devem ser suportadas e respeitadas pelo *framework*, tornando os conceitos complementares e fortemente relacionados (BACHMANN et al., 2000).

Figura 5 – A relação componentes, modelo e frameworks de componentes



Fonte: Bachmann et al. (2000)

Os componentes são gerenciados por meio de uma interface, chamada *lifecycle*, que permite aos desenvolvedores inicializar, executar e desativar os componentes (SCHWARZ; SOMMER; FARRIS, 2003). Eventualmente, componentes necessitam ser registrados antes de serem utilizados no projeto (HEINEMAN; COUNCILL, 2001).

As principais vantagens da utilização de *frameworks*, segundo Fayad e Schmidt (1997) decorrem da modularidade, reuso, extensibilidade e eventualmente da inversão de controle (IC), que assumem o controle da execução invocando métodos da aplicação quando necessário.

Por outro lado, existem desafios na criação e uso de *frameworks* que não podem ser ignorados. Instituições que necessitam construir ou usar *frameworks* devem atentar aos seguintes desafios: esforço de desenvolvimento, curva de aprendizagem, integração, eficiência, manutenção, validação e a falta de padrões (FAYAD; SCHMIDT, 1997).

Em relação ao desenvolvimento deste trabalho, os conceitos descritos nesta seção guiarão o desenvolvimento do *framework* proposto, levando em consideração as características e os desafios apontados.

3.2.1 Node.js

Node.js é uma plataforma construída sobre o motor Javascript do Google Chrome (V8) inicialmente planejada para o desenvolvimento de programas de rede escaláveis (IYER, 2013). Desenvolvida por Ryan Dahl em 2009 com colaboração de Joyent, trata-se de um

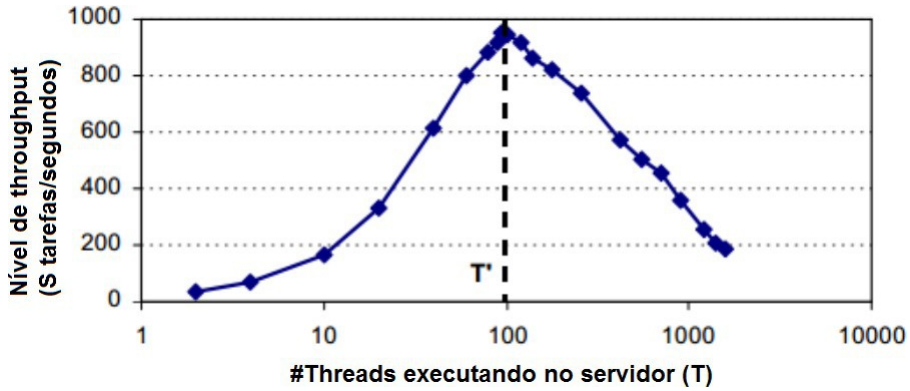
ambiente Javascript no lado do servidor.

Diferente de muitas outras plataformas modernas, Node.js é baseado em um modelo assíncrono orientado a evento I/O que suporta execuções concorrentes sem a necessidade de múltiplas *threads*, prevenindo a aplicação de ser bloqueada enquanto espera alguma operação I/O (TILKOV; VINOSKI, 2010).

Segundo Welsh et al. (2000), desenvolver sistemas concorrentes utilizando múltiplas *threads* é uma atividade custosa que exige muito recurso computacional. Independente de quão bem se trabalha com *threads* no servidor, o seu número tenderá a crescer, pois este modelo dedica uma *thread* separada a cada nova requisição. Desta forma, há um aumento significativo no consumo de memória e CPU, levando ao declínio a performance e a escalabilidade do sistema. Isso tem feito os programadores preferirem o modelo orientado a eventos.

A Figura 6 tipicamente mostra o máximo de número de *threads* T' que um sistema pode suportar sem que ocorra degradação de performance.

Figura 6 – Degradação do throuput no modelo de threads



Fonte: Iyer (2013), adaptado pelo autor.

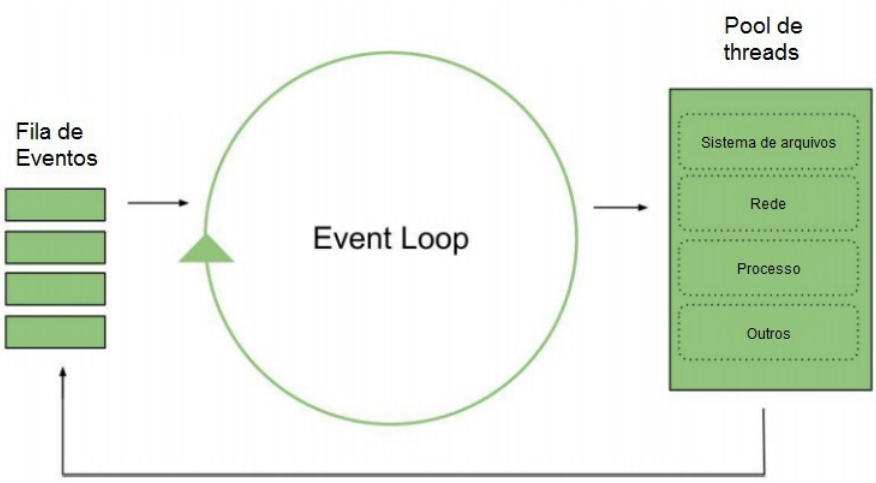
Uma das partes fundamentais da arquitetura do Node.js é o *event loop*, método que o Javascript utiliza para lidar com os eventos. O *event loop* permite que, ao invés do sistema operacional, o Node.js gerencie a mudança entre as tarefas a serem executadas (CANTELON; HOLOWAYCHUK, 2011).

Segundo Zeldovich et al. (2003), o *event loop* se trata de um *loop* principal responsável por esperar e despachar eventos que tem seus recursos disponibilizados. Esses

eventos são assinados na *thread* principal da aplicação passando funções como parâmetros, denominadas de *callbacks*, que são executadas quando esses eventos são disparados, levando

consigo o resultado da operação I/O. Ainda com Zeldovich et al. (2003), o Javascript é uma excelente linguagem para essa abordagem pois uma de suas principais features é o suporte a *callbacks*.

Figura 7 – Modelo *event loop*



Fonte: Joyent (2012), adaptado pelo autor.

Levando em consideração que o modelo orientado a evento tende a ser robusto, com baixa degradação em seu *throughput*, se a implementação dos eventos e o estado de empacotamento das tarefas forem eficientes, o pico de *throughput* será elevado (IYER, 2013). A Figura 8 apresenta o *throughput* alcançado pelo modelo orientado a evento. Seu número excede ao modelo de *threads*, porém, mais importante é o fato de não se degradar quando o número de conexões simultâneas aumenta.

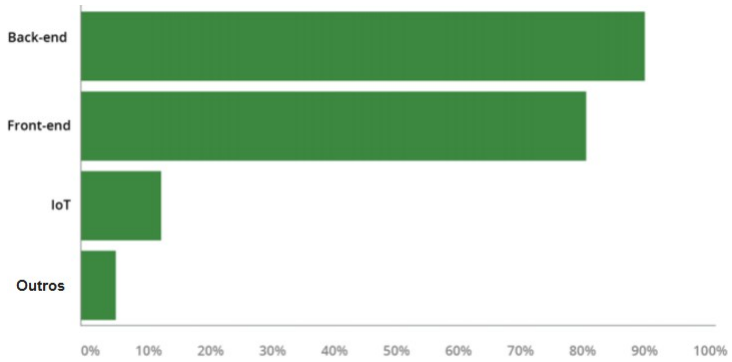
Figura 8 – *Throughput* do modelo orientado a evento

Fonte: Iyer (2013), adaptado pelo autor.

Com 3,5 milhões de adeptos e um crescimento anual de cem por cento, Node.js se tornou a tecnologia de desenvolvimento mais significante e com maior crescimento nos últimos anos, tornando uma plataforma universal para o desenvolvimento de aplicações web *real-time*, *mobile*, desktop, microsserviços e internet das coisas (JOYENT, 2012).

Sendo assim, o *full stack* não se resume mais ao *front-end* e *back-end*, mas sim ao *front-end*, *back-end* e dispositivos conectados, que é a combinação dos navegadores com todas as outras plataformas citadas anteriormente que, em sua maioria, podem executar Javascript coordenado pelo Node.js (JOYENT, 2012).

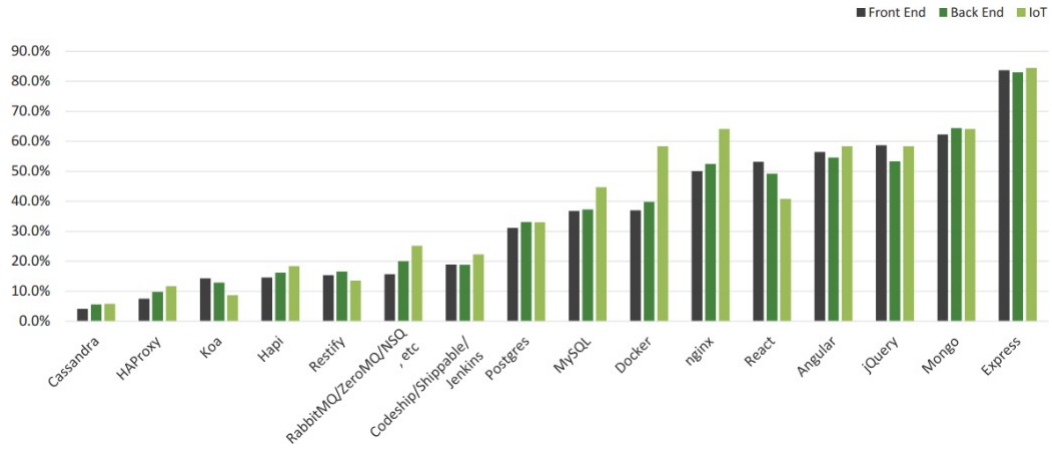
Figura 9 – Foco atual dos desenvolvedores



Fonte: Joyent (2012), adaptado pelo autor.

A popularidade de aplicações *real-time*, redes sociais e jogos interativos fez surgir a *stack* MEAN, combinação de MongoDB, Express, AngularJS e Node.js, é capaz de lidar com muitas conexões simultâneas de forma escalável (JOYENT, 2012). Como a Figura 10 apresenta, todas essas tecnologias são comumente utilizadas por desenvolvedores Node.js.

Figura 10 – Tecnologias mais populares relacionadas



Fonte: Joyent (2012), adaptado pelo autor.

Node.js, bem como tecnologias relacionadas como Express e AngularJS, serão utilizadas no processo de criação do *framework* proposto neste trabalho.

3.2.2 AngularJS

AngularJS é um *framework front-end* escrito em javascript e mantido pela Google que torna a implementação de páginas web bem organizada e estruturada utlizando o padrão MVC, possibilitando capturar e manipular os dados de entrada do usuário, além de controlar como os elementos serão mostrados no *browser* (DAYLEY, 2014). Com ele é possível desenvolver aplicações *web* que utilizam conceitos modernos, tais como *single-page* e *AJAX-style* de forma simples e eficiente (KOZLOWSKI, 2013).

AngularJS ganhou bastante atenção devido a sua forma de trabalhar com sistemas de *templates*, facilidade de desenvolvimento e suas práticas sólidas de engenharia, e de fato seu sistema de *templates* é única em alguns aspectos (KOZLOWSKI, 2013):

- Utiliza HTML como linguagem de *template*;
- Não necessita de atualização explícita do DOM, com ele é possível rastrear as ações do usuário, eventos do *browser* e mudanças dos dados para descobrir quando e qual *template* deve ser atualizado;
- Possui diversos componentes interessantes que tona possível ensinar como o *browser* deve interpretar as nova *tags* e atributos HTML;

Segundo Dayley (2014), outros *frameworks* javascript podem ser usados com a

plataforma Node.js, como Backbone [1], Ember [2] e Meteor, entretanto AngularJS possui o melhor *design* e as melhores *features*. Seguem alguns benefícios que o AngularJS promove:

- **Data binding**: AngularJS, utilizando um poderoso mecanismo de escopo, possui um método muito simples para manipular elementos HTML;
- **Extensibilidade**: A arquitetura do AngularJS permite facilmente estender quase todos os aspectos da linguagem para fornecer a sua própria implementação;
- **Limpo**: Angualarjs força o programador escrever códigos limpos e lógicos;
- **Reusável**: A combinação extensibilidade com o código limpo torna mais fácil escrever códigos reusáveis;
- **Suporte**: A Google investe bastante no AngularJS, ao qual dá ao projeto a vantagem sobre iniciativas similares que não tiveram êxito;
- **Compatibilidade**: AngularJS é desenvolvido em javascript e aniquilou a relação com a biblioteca jQuery. Isto torna fácil iniciar a integração do AngularJS e reusar partes de códigos existentes que possuem as mesmas estruturas do *framework*;

Figura 11 – DI AngularJS

```
var app = angular.module('MyApp', []);

app.factory('changeUrlTo', function ($location) {
    return function (destinationUrl) {
        $location.path(destinationUrl);
    };
});

app.controller('MyCntl', function (changeUrlTo) {
    changeUrlTo('/another-page');
});
```

Fonte: Jain, Mangal e Mehta (2015)

Além disso, AngularJS possui alguns tesouros escondidos, como a sua injeção de dependências (DI) que torna o código fortemente testável, além de possibilitar facilmente a construção de aplicações web por meio de pequenos componentes já existentes (KOZLOWSKI, 2013). Como a Figura 11 mostra, para acessar os serviços basta específicar em seus parâmetros que a DI irá detectar que o programador precisa do recurso e proverá a instância, fazendo com

[1] http://backbonejs.org/

[2] https://www.emberjs.com/

que o desenvolvedor se preocupe apenas com a lógica da aplicação (JAIN; MANGAL; MEHTA, 2015).

A comunidade é um dos fatores mais importantes na escolha de um *framework*, pois quanto maior a comunidade, mais dúvidas serão solucionadas, mais módulos serão desenvolvidos e mais tutoriais estarão disponíveis (JAIN; MANGAL; MEHTA, 2015). Como mostrado no quadro 1, o AngularJS possui os maiores números, sendo o sexto projeto com maior número de *stars* no Github e possui mais perguntas no StackOverflow do que Ember e Backbone juntos.

Quadro 1 – Quadro de comparação entre *frameworks*

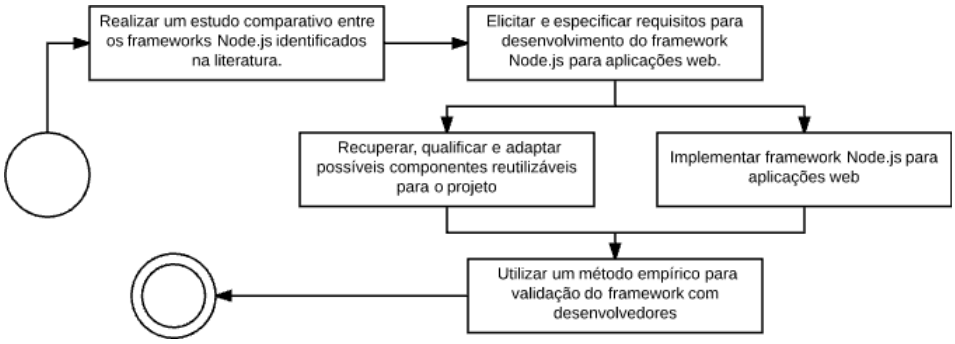
Metric	**AngularJS**	**BackBoneJs**	**Ember.js**
Stars no Github	27.2k	18.8k	11k
Módulos de Terceiros	800	236	21
Questões no StackOverflow	49.5k	11.9k	11.2k
Resultados no Youtube	75k	16k	6k
Contribuidores no GitHub	928	230	393
Extensões do Chrome que utilizam	150k	7k	38.3k

A maioria dos *frameworks* são desenvolvidos por hobby em comunidades *open source*, como Capuccino e Knockout; AngularJS foi desenvolvido e mantido por uma equipe dedicada de engenheiros da Google, isso significa que o programador não possui apenas a comunidade *open source*, mas também um grupo de engenheiros especializados que ajudam a solucionar as dúvidas e problemas encontrados (JAIN; MANGAL; MEHTA, 2015).

4 PROCEDIMENTOS METODOLÓGICOS

Esta Seção apresenta os procedimentos para a execução do trabalho proposto. A Figura 12 apresenta as etapas a seguir a partir do estudo comparativo entre os *frameworks* Node.js mais populares até a validação da implementação deste trabalho.

Figura 12 – Procedimentos para a execução do trabalho.



Fonte: Elaborada pelo Autor

4.1 Realizar um estudo comparativo entre os *frameworks* Node.js identificados na literatura

A primeira etapa tem como objetivo levantar os *frameworks* Node.js existentes mais populares que se enquadram na mesma categoria deste trabalho. Para isto, será utilizado com fonte o site Github, onde serão levados em consideração os projetos que apresentam os maiores números de stars. Desta forma, conseguiremos identificar os *frameworks* com maiores níveis de interesses entre a comunidade Javascript.

A segunda etapa visa identificar as funcionalidades mais recorrentes presentes na documentação dos *frameworks* recolhidos por ordem de popularidade. Assim, será possível enumerar um conjunto de *features* que serão avaliadas e incorporadas em procedimentos futuros no *framework* proposto por este trabalho.

4.2 Elicitar e especificar requisitos para desenvolvimento do *framework* Node.js para aplicações web

Este passo visa filtrar as funcionalidades recolhidas no processo anterior para serem inseridas no escopo de desenvolvimento do projeto, levando em consideração a prioridade de cada uma por meio de sua complexidade, dependências e ao tempo requerido de execução. Todos os requisitos serão devidamente documentados e inseridos no trabalho proposto para auxiliar no processo de implementação do *framework*.

4.3 Recuperar, qualificar e adaptar possíveis componentes reutilizáveis para o projeto

Este passo tem como objetivo adaptar o código do modelo de injeção de dependências encontrados no repositório do AngularJS para o contexto *server-side*, além de recuperar, qualificar e adaptar outros possíveis componentes reutilizáveis encontrados na comunidade *open-source* capazes de otimizar o processo de desenvolvimento deste trabalho.

4.4 Implementar *framework* Node.js para aplicações web

Para realizar o desenvolvimento do *framework*, será necessário utilizar as boas técnicas da Engenharia de Software para garantir a qualidade e o melhor atendimento dos requisitos levantados, levando em consideração o tempo disponível do projeto. Por se tratar de um projeto com requisitos rigorosamente bem definidos, será utilizado o modelo de desenvolvimento em cascata, onde as atividades serão definidas e executadas sequencialmente.

Por se tratar de uma ferramenta gratuita e muito popular, a ferramenta utilizada para gerenciar o controle de versão e mudança será o Github. Todas as importantes alterações de código serão comitadas em uma *branch* de desenvolvimento, e ao final de cada tarefa, um único *commit* será realizado na *branch* principal. Desta forma, será possível identificar quando e quais atividades foram realizadas, e se necessário, realizar modificações sem maiores riscos.

A primeira etapa consiste em recuperar todos os requisitos e as tecnologias descritas em fases anteriores. Após isto, será planejado como e quais os recursos serão utilizados na implementação de cada requisito. Desta forma, será possível separar as

responsabilidades em pequenos módulos e definir, por meio de suas dependências, qual sequência será seguida durante toda a fase da implementação do trabalho.

A segunda etapa será preparar o ambiente de desenvolvimento necessário para a execução plena do sistema. Serão levantadas as ferramentas e as tecnologias que serão utilizadas, instaladas e configuradas na máquina do desenvolvedor.

A terceira e última etapa será de implementação, na qual o sistema será codificado seguindo a risca a sequência dos módulos planejados. Desta forma, o sistema será finalizado quando todos os componentes estiverem completamente implementados e testados.

4.5 Utilizar um método empírico para validação do *framework* com desenvolvedores

Este passo consiste em aplicar um conjunto de atividades a programadores que já tiveram experiências com o desenvolvimento de aplicações Javascript utilizando o *framework* AngularJS. Após isso, será aplicado um questionário para avaliar se o produto final do trabalho proposto cumpriu com os resultados esperados.

5 PROJETO E DESENVOLVIMENTO DO *FRAMEWORK* ATLOM.JS

Nesta seção serão abordadas as atividades do projeto e desenvolvimento do *framework* Atlom.js. Inicialmente, são levantados todos os *frameworks full-stack* Javascript mais populares do âmbito Node.js com o intuito de classificar as suas *features* mais recorrentes. Desta forma, serão estabelecidos todos os requisitos que serão desenvolvidos para o presente trabalho. Além disso, será definida a sua arquitetura, onde serão determinados os componentes e as suas interações. E para finalizar, a codificação do *framework* Atlom.js.

5.1 Levantamento dos *frameworks* e suas *features*

O primeiro passo deste trabalho consistiu em buscar, por meio da ferramenta de pesquisa Google [1], as palavras chaves *"most popular frameworks nodejs"*. Nos resultados da pesquisa foram identificados estudos que indicavam os melhores e mais populares *frameworks* Node.js da atualidade. Foram levantados ao todo os 11 *frameworks* mais recorrentes indicando a sua popularidade no Github [2] e o seu tipo, conforme ilustrado no Quadro 2.

Quadro 2 – Levantamento dos *frameworks* Node.js.

Frameworks	***Stars* no Github**	**Tipo**
meteor	38,235	full-stack
socket.io	35,862	outro
express	33,979	rest API
koa.js	17,443	rest API
sails	17,770	full-stack
mean.io	10,331	full-stack
hapi	8,333	rest API
derby	4,278	rest API
mean.js	4,222	full-stack
total.js	3,355	full-stack
mojito	1,613	outro

A lista de *frameworks* foi organizada por ordem de popularidade, selecionando os projetos com maior número de *stars* no Github. A partir disso, foram analisadas as estruturas de cada projeto a fim de verificar quais os três melhores *frameworks* que se enquadram nas características deste trabalho, possuindo recursos de implementação *full-stack*. Os escolhidos

[1] https://www.google.com.br
[2] https://github.com/

foram Meteor [3], Sails.js [4] e Mean.io [5].

Na segunda etapa, foram analisados os códigos e a documentação de cada *framework* a fim de levantar as principais *features* que os projetos, em sua maioria, possuíam em comum, como:

- **Rotas customizáveis**: Habilidade de intepretar e mapear requisições HTTP's de URL's específicas aos seus respectivos controladores, possibilitando a intercepção dos dados, manipulação e resposta;
- **Modelos e ORM**: Serviços especializados em abstrair interações com o banco de dados, permitindo executar procedimentos de leitura e escrita de dados de forma simples;
- **Controladores**: Responsáveis por responder as requisições vindas dos navegadores *web*, aplicações *mobiles* ou qualquer outro sistema capaz de se comunicar com o servidor;
- **Views**: Marcação de templates compilados no servidor e entregues ao cliente em forma de página HTML;
- **Arquivos públicos**: Capacidade de expor arquivos estáticos (js, css, imagens e etc) do servidor que serão disponibilizados publicamente;
- **Configuração**: Interface de configuração que customiza os atributos padrões do projeto disponibilizados para toda a aplicação, permitindo sobrescrevê-los de acordo com as necessidads do sistema;
- **Middlewares**: Mediadores capazes de estabelecer procedimentos padrões configuráveis acionados pelas requisições HTTP's;
- **Sessão**: Conjunto de componentes que em conjunto persistem e compartilham informações do *user agent* entre as requisições;
- **Automatizadores de tarefas**: Ferramenta que permite automizar tarefas repetitivas, mas essencialmente para concatenação de *scripts*, minificação, testes e outras;
- **Socket**: Comunicação bi-direcional entre cliente e servidor capaz da construção de aplicações *real-time*;
- **Componentização**: Conjunto de interfaces que permitem a reutilização de outros componentes para a aplicação em desenvolvimento; e
- **Injeção de dependências**: Possibilita incorporar instâncias de serviços específicos em diferentes blocos de código espalhados na aplicação.

[3] https://www.meteor.com/
[4] https://sailsjs.com/
[5] http://mean.io/

O grande diferencial deste trabalho está na facilidade que o pogramador terá em desenvolver e manipular os recursos que irão compor o *framework* Atlom.js, permitindo adicionar, alterar ou remover componentes de forma simples e segura. Com isso, o programador utilizará em seu projeto apenas o que for necessário, descartando tecnologias agregadas que não serão aproveitadas pela aplicação em desenvolvimento. O módulo atlom, é responsável por implementar toda a lógica de DI (Dependency Injection) e IoC (Inversion of Control) do *framework* utilizando os conceitos da DI do AngularJS.

Além disso, padroniza o projeto, uma vez que o modelo de desenvolvimento estará presente no lado do cliente e servidor, diminuindo também a curva de aprendizagem do *framework* para os pogramadores que já estejam familiarizados com a ferramenta AngularJS.

5.2 Requisitos funcionais

Inicialmente, por meio de uma análise na documentação e no código dos *frameworks* selecionados, foram levantados todos os requisitos funcionais que serão desenvolvidos para o Atlom.js, levando em consideração o tempo de desenvolvimento e os recursos disponíveis. São eles:

- **[RF01] Gerenciar Configuração**: O *framework* deve permitir adicionar, alterar ou remover atributos de configuração do sistema, permitindo que a configuração seja executada de acordo com a variável de ambiente existente;
- **[RF02] Gerenciar Rotas**: O *framework* deve possuir um sistema de rotas que permita capturar requisições HTTP's por meio de URL's customizáveis, direcioná-las a controladores específicos e também permitir a adição de *middlewares*;
- **[RF03] Comunicação *Full-duplex***: O *framework* deve permitir a comunicação *full-duplex* entre cliente e servidor, podendo enviar dados de diferentes tipos, dentre eles: string, number, JSON e buffer;
- **[RF04] Configurar Arquivos de Execução**: O *framework* deve permitir escolher, seguindo uma ordem e uma sintaxe padrão, quais arquivos serão carregados e excutados pelo servidor;
- **[RF05] Servir Arquivos Estáticos**: O *framework* deve permitir escolher, seguindo uma ordem e uma sintaxe padrão, quais arquivos (css, js, imagens e etc) serão servidos estaticamente;
- **[RF06] Persistir Dados**: O *framework* deve permitir a persistência em um ou mais banco

de dados de diferentes tipos, dentre eles: NoSQL e SQL;

- **[RF07] Gerenciar Sessão**: O *framework* deve permitir adicionar, alterar ou remover informações do usuário e compartilhá-las entre as requisições utilizando a persistência em memória ou em um banco de dados;
- **[RF08] Renderizar *Views***: O *framework* deve permitir renderizar *views* utilizando uma linguagem de marcação compilada e entregue pelo servidor em forma de HTML;
- **[RF09] Minificar Arquivos Estáticos**: O *framework* deve permitir concatenar e minificar todos os arquivos estáticos (css e js) pré-configurados, transformando-os em um arquivo único para produção;
- **[RF10] Testes Unitários**: O *framework* deve suportar testes unitários de arquivos pré-configurados, podendo invocar todos os serviços disponíveis da aplicação para teste;
- **[RF11] Injeção de dependências**: O *framework* deve permitir a captura de instâncias de serviços específicos e injetá-los nos componentes da aplicação que requisitarem o recursos; e
- **[RF12] Componentização**: O *framework* deve permitir a construção de componentes reutilizáveis e incorporá-los em outros projetos.

5.3 Atributos de Qualidade

Este presente trabalho não visa criar funcionalidades que estão além das mais utilizadas no mercado. Em contrapartida, como diferencial, foram levantados os atributos de qualidade que serão propostos para o desenvolvimento do *framework* Atlom.js, são eles:

- **Facilidade de reuso**: O *framework* deve prover recursos necessários para a reutilização dos componentes desenvolvidos;
- **Adequação**: O *software* deve prover as funcionalidades necessárias para que os objetivos dos usuários e os requisitos funcionais sejam alcançados;
- **Compreensibilidade**: O *framework* deve facilitar a capacidade do programador de compreender o sistema, diminuindo a quantidade de conceitos prévios para o seu entendimento;
- **Curva de Aprendizado**: O *framework* deve permitir que o programador aprenda de forma rápida a utilização de todas as suas funcionalidades;
- **Modificabilidade**: O sistema deve permitir a realização de mudanças na implementação da aplicação sem que haja grandes impactos em seu funcionamento; e

5.4 Arquitetura

Para alcançar todos os requisitos e os atributos de qualidade levantados, a arquitetura foi planejada de forma a recuperar, qualificar e adaptar todos os componentes necessários para auxiliar na construção do *framework*. Os componentes foram selecionados por meio de um estudo de comparação aos quais foram escolhidas as tecnologias mais bem cotadas da comunidade Javascript.

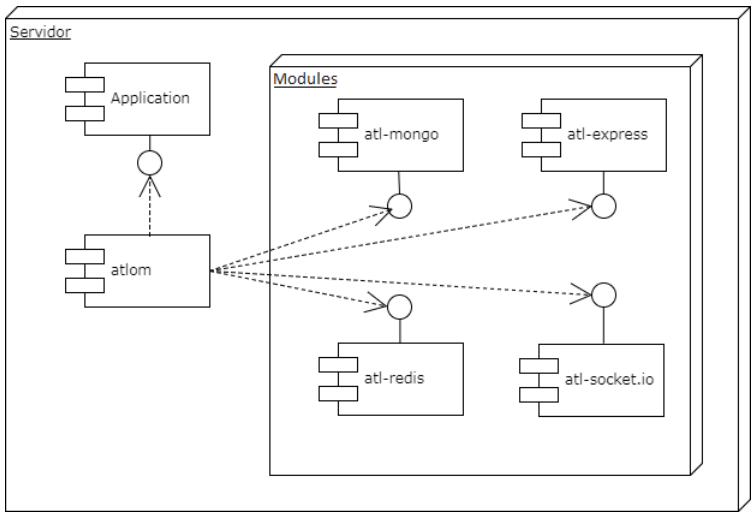
O trabalho proposto utilizará a arquitetura MEAN que possui como as principais tecnologias: MongoDB, Express, Angular e Node.js. Além disso, o *framework* utilizará o Redis[6] como uma segunda opção para a persistência de dados. Para a realização dos testes, foram escolhidos os módulos Mocha [7] e Should [8]. O Gulp[9] foi a ferramenta utilizada para a automação das tarefas, e o Bower[10] para organizar as dependências no lado do cliente. O Visual Studio Code [11] foi a IDE escolhida para a implementação do trabalho. Por fim, a biblioteca *ng-di*[12] foi escolhida para realizar a injeção de dependências e inversão de controle do módulo atlom. Todas as tecnologias foram devidamente instaladas e testadas na máquina do desenvolvedor para garantir o desenvolvimento pleno do sistema.

O *framework* foi dividido em duas camadas principais: cliente e servidor. A camada servidor possui três módulos básicos: (i) o módulo atlom: responsável por cuidar da injeção de dependências, relacionar e iniciar todos os outros componentes restantes da camada servidor, além de carregar todos os arquivos de configuração e da aplicação; (ii) módulo *modules*: responsável por oferecer recursos independentes reutilizáveis para a aplicação; e (iii) módulo *application*: contém a implementação das regras de negócio da aplicação no servidor. A Figura 13 ilustra como se estruturam os componentes da camada servidor.

---

6 https://redis.io/
7 https://mochajs.org/
8 https://shouldjs.github.io/
9 https://gulpjs.com/
10 https://bower.io/
11 https://code.visualstudio.com/
12 https://github.com/jmendiara/ng-di

Figura 13 – Arquitetura do servidor.

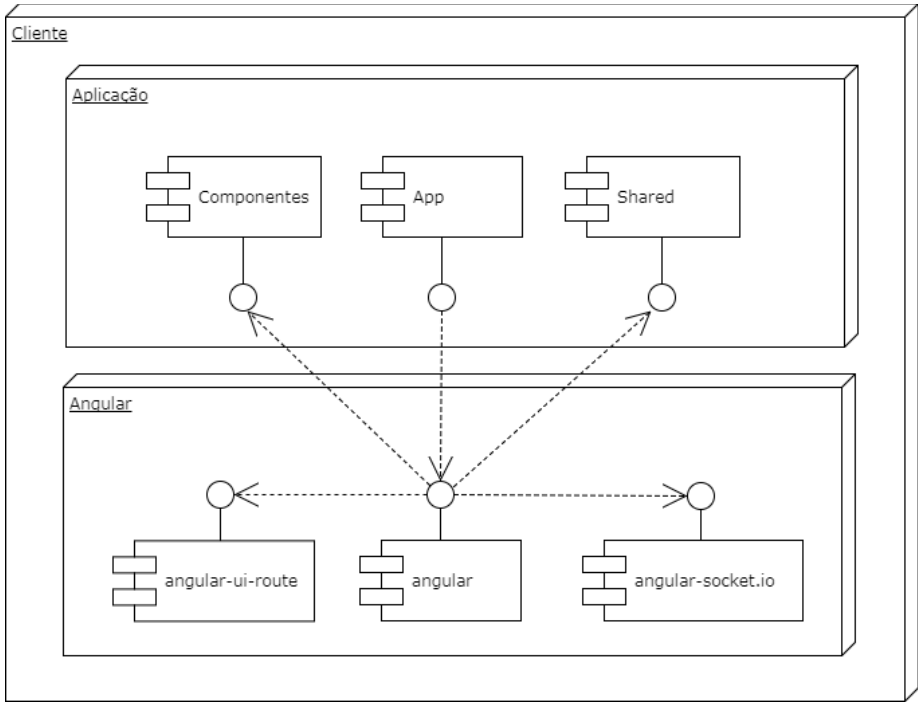


Fonte: Elaborada pelo Autor

Embora o *framework*, por meio de sua injeção de dependências, proveja facilmente a possibilidade da mudança dos componentes arquiteturais utilizados no sistema, o *framework* conterá inicialmente quatro bibliotecas básicas necessárias para alcançar os requisitos levantados nos passos anteriores: (i) biblioteca Atlom mongo: responsável por iniciar a conexão e implementar abstrações dos serviços de persistência do banco de dados MongoDB; (ii) biblioteca Atlom express: configura e implementa *middlewares* relevantes do *framework* Express; (iii) biblioteca Atlom socketIo: responsável por configurar e prover os serviços de socket da biblioteca socket.io; e (iv) biblioteca Atlom redis: responsável por conectar e implementar os serviços de persistência do banco de dados Redis.

Já a camada cliente possui dois elementos fundamentais: (i) módulo *application*: responsável por implementar todas as regras de negócio da aplicação no cliente, possuindo componentes de interface, modelos e serviços; e (ii) módulo Angular: contém componentes do *framework* AngularJS responsáveis por organizar e executar os componentes da aplicação, além de prover recursos auxiliares fundamentais no desenvolvimento de sistemas javascript. A Figura 14 ilustra como se estruturam os componentes da camada cliente.

Figura 14 – Arquitetura do cliente.



Fonte: Elaborada pelo Autor

O módulo *Application* possui três componentes básicos: (i) *components*: responsável por implementar os componentes visuais e suas interações; (ii) *app*: responsável por implementar lógicas de configuração e execução da aplicação; e (iii) *shared*: possui serviços compartilhados que os components e app irão requisitar.

O módulo Angular também possui também três componentes essenciais: (i) angular: responsável por organizar a aplicação no modelo MVC e prover recursos auxiliares comumente utilizados no ambiente de desenvolvimento javascript; (ii) angular-ui-route: módulo do AngularJS responsável por implementar sistemas de rotas para a construção de aplicações *single pages*; e

(iii) angular-socket.io: módulo do AngularJS responsável por abstrair recursos de comunicação socket da biblioteca socket.io.

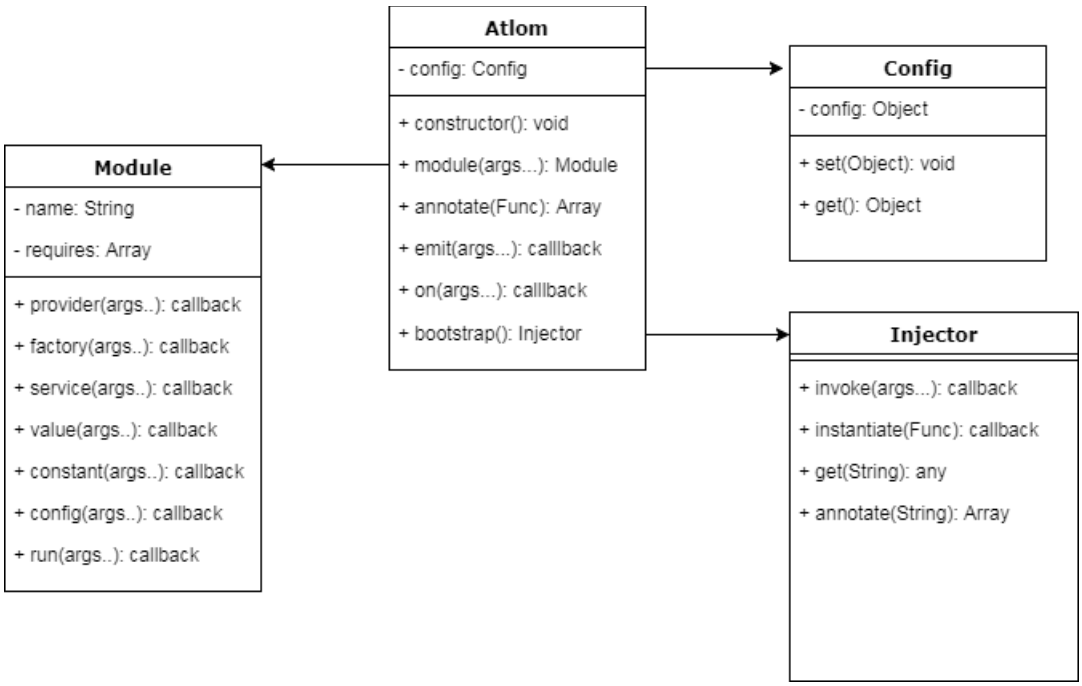
Desta forma, ficaram definidas as tecnologias que serão aproveitadas para o presente trabalho, além da projeção dos componentes e suas interações que servirão de base para a implementação do *framework* Atlom.js.

5.5 Implementação do *Framework* Atlom.js

Após terem sido definidos todos os requisitos funcionais, os atributos de qualidade, as tecnologias, os componentes e suas interações, foi realizado a implementação do *framework* Atlom.js. No processo de desenvolvimento, as atividades executadas foram: (i) desenvolvimento e publicação do módulo principal do projeto denominado com atlom; (ii) desenvolvimento dos módulos padrões Atlom; (iii) configuração do ambiente *client* e seus ativos; (iv) desenvolvimento das tarefas automatizadas, e para finalizar, (v) desenvolvimento de um CRUD básico.

Conforme o diagrama de classes define, ilustrado pela Figura 15, o módulo Atlom, além da classe principal, possui 3 classes relacionadas que desempenham responsabilidades únicas que compõem todo o funcionamento do módulo descrito anteriormente.

Figura 15 – Diagrama de classes do módulo Atlom.



Fonte: Elaborada pelo Autor

5.5.1 Desenvolvimento do Módulo Atlom

O módulo atlom, como explanado anteriormente, é o módulo principal do *framework*, responsável por conter todas as *features* necessárias para a montagem e execução dos componentes do sistema. Composto por quatro elementos básicos: (i) serviços de DI

(*Dependency Injection)*: responsável pela injeção de dependências utilizada no lado do servidor;

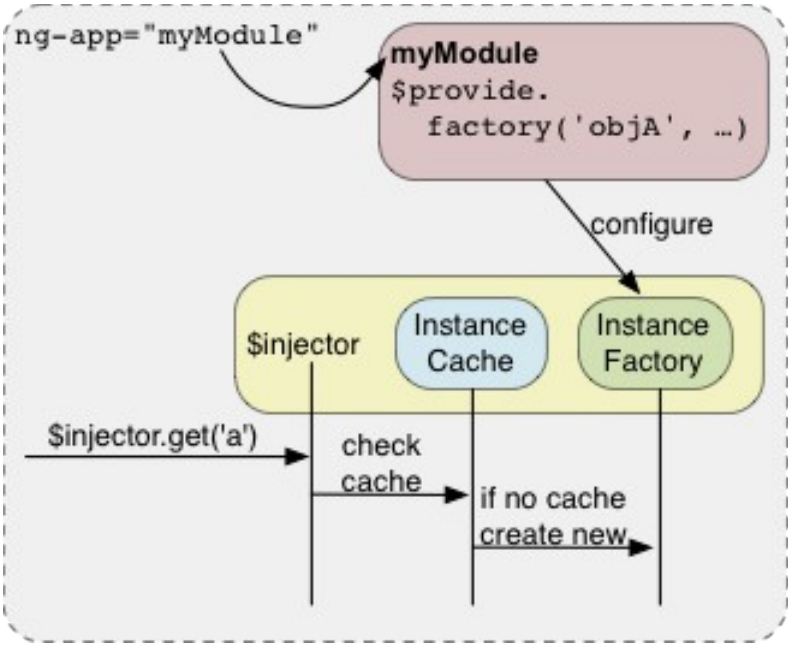
(ii) serviços de configuração: responsável por prover interfaces de configuração do sistema; (ii) serviços auxiliares: serviços que desempenham pequenas utilidades comumente utilizados no âmbito Node.js; (iii) serviços de inicialização: responsável pela montagem e execução dos componentes do sistema, bem como carregar e montar a configuração e execução dos arquivos da aplicação.

5.5.1.1 Desenvolvimento dos Serviços de DI

Inicialmente, no desenvolvimento dos serviços de DI do módulo atlom, foi realizado um longo estudo sobre os conceitos principais do funcionamento da DI do AngularJS, seus componentes e como eles se interagem. Assim, foram realizadas pesquisas com o intuito de levantar os módulos Node.js já desenvolvidos que propusessem a migração da DI do AngularJS para o Node.js. Foram encontradas duas bibliotecas: node-di e ng-di.

A possibilidade da utilização do módulo node-di foi rapidamente descartada por ser um módulo com métodos limitados, não contendo as principais *features* da DI do AngularJS. Em contrapartida, o módulo ng-di possui exatamente todos os métodos úteis da DI original no contexto do servidor, sendo a biblioteca escolhida para a componentização e o desenvolvimento da injeção de dependências do trabalho proposto, atendendo aos requisitos **RF11** e **RF12**. Isso evita o esforço da implementação de uma nova biblioteca do zero.

Figura 16 – Injeção de dependências do Atlom



Fonte: https://docs.angularjs.org/guide/di

Como a Figura 16 ilustra, os serviços implementados no módulo "myModule"poderão ser capturados posteriormente por um sistema de cache, tanto pelo mesmo módulo, como por outros que o especificarão como dependência. Desta forma, com o serviço de DI, será possível desenvolver uma aplicação completa por meio de pequenos módulos ou serviços, conforme o conceito do desenvolvimento baseado em componentes sugere.

5.5.1.2 Desenvolvimento dos Serviços de Configuração

No desenvolvimento dos serviços de configuração, foram levados em consideração os padrões dos métodos de configuração dos *frameworks* utilizados como referência deste trabalho, juntamente com os requisitos levantados na fase de planejamento.

A configuração do sistema necessita ser dinâmica de acordo com a variável de ambiente da aplicação. Atendendo ao requisito **RF01**, foi desenvolvido o método atlom.config.set que tem como intuito persistir em memória e disponibilizar as configurações setadas pelo sistema em uma variável de ambiente padrão (*default*) concatenadas em variáveis específicas (*development*, *production*, *test* ou qualquer outra). As configurações setadas em ambientes específicos detém prioridade sobre as configurações setadas em modo *default*.

5.5.1.3 Desenvolvimento dos Serviços Auxiliares

Como serviços auxiliares, até o presente momento, foram desenvolvidos os métodos atlom.emit e atlom.on, funcionalidades que manipulam a emissão e recebimento de eventos no contexto do servidor, herdados da classe Events, nativa do Node.js, possibilitanto a troca de mensagens entre o processo principal da aplicação.

Além disso, foi desenvolvido também o método atlom.requires, responsável por carregar determinados arquivos em tempo de execução de acordo com as regras estabelecidas como parâmetro na chamadas do método, geralmente utilizado para inicializar configurações dinâmicas ou implementações de *features* externas dos módulos atlom.

5.5.1.4 Desenvolvimento dos Serviços de Inicialização

Para a finalização do módulo atlom, foram desenvolvidos os serviços de inicialização dos módulos e dos arquivos de configuração e da aplicação do servidor. O método atlom.bootstrap foi desenvolvido com o intuito de inicializar os módulos definidos pela interface atlom.module e disponibilizar a instância injector para captação de recursos providos pelos módulos instanciados. Desta forma, o *framework* tem controle de quando e quais módulos serão executados e suas instâncias.

No momento em que o módulo atlom é iniciado pela aplicação, os arquivos contidos dentro da pasta config do projeto serão automaticamente carregados por meio da interface atlom.config.set. Os dados de configuração serão setados e disponibilizados para todo o sistema através do objeto config do módulo atlom.

Assim como os arquivos de configuração, os arquivos da aplicação também são iniciados no momento em que o módulo atlom é instanciado. Esses arquivos são pré configurados pelo requires.js, arquivo contido dentro da pasta config do projeto. Desta forma, conforme sugere o requisito **RF04**, é possível escolher quais arquivos serão carregados pelo *framework* e definir qual a ordem do carregamento.

5.5.1.5 Publicação do Módulo Atlom

Após a implementação de todos os seus recursos, o módulo atlom foi publicado no serviço de gerenciamento de pacotes oficial do Node.js (NPM). Desta forma, além de tornar o módulo um componente externo a aplicação, será possível controlar as versões e disponibilizar o

serviço para as instalações futuras do *framework* utilizando o comando **npm install atlom**.

5.5.2 Desenvolvimento dos Módulos Atlom

Após a construção do módulo atlom, o *framework* passou a conter as funcionalidades bases para a contrução do restante dos componentes planejados. Então, foi proposto o desenvolvimento dos módulos padrões que incorporam as tecnologias do servidor que serão fundamentais no desenvolvimento das aplicações que utilizarem o Atlom.js como *framework* do projeto. Essas tecnologias foram definidas na fase do planejamento da arquitetura, são elas: MongoDB, Express, Redis e Socket.io.

5.5.2.1 Desenvolvimento do Módulo Atlom Mongo

Para a construção do módulo Atlom mongo, responsável pela implantação do banco de dados MongoDB no projeto, de acordo com o requisito **RF06**, foi utilizado o módulo mongoose, bastante conhecido na comunidade Node.js, responsável oferecer um conjunto de métodos que abstraem as lógicas de manipulação dos dados de persistência. Foi setado no arquivo de configuração mongo do projeto as informações necessárias para a conexão do banco de dados em diferentes ambientes de execução, fundamental para que não haja conflitos dos dados em contextos diferentes como o de desenvolvimento, teste e produção.

A injeção de dependências do atlom não suporta execuções assíncronas, portanto foi necessária a utilização do módulo deasync para que a conexão do mongoose se tornasse síncrona, ou seja, o processo é bloqueado até que ocorra uma resposta, tornando as execuções restritamente sequenciais.

Após a finalização do módulo Atlom mongo, foi criado o módulo atlom app, responsável por conter a implementação das regras de negócio da aplicação em desenvolvimento, e declarado como uma de suas dependências o módulo Atlom mongo. Desta forma, a aplicação terá acesso a conexão do mongoose e seus métodos.

5.5.2.2 Desenvolvimento do Módulo Atlom Express

Embora seja uma biblioteca de grande importância para o projeto, contendo responsabilidades sobre as lógicas de implementação HTTP do servidor, o desenvolvimento

da biblioteca Atlom express se resumiu basicamente a implatação dos *middlewares* do módulo

express, levantados na fase de planejamento, seguindo os passos e as boas práticas descritas em suas documentações e as adaptando ao modelo de DI proposto por este trabalho.

Por conter *middlewares* que persistem informações no banco de dados MongoDB, a biblioteca Atlom express depende diretamente da biblioteca Atlom mongo. A instância app é o produto principal da biblioteca, é nela que estão contidas as interfaces que serão utilizadas para o acesso dos recursos, a configuração e a customização do módulo Express no projeto.

Os *middlewares* foram criados em arquivos diferentes, onde cada um contém uma responsabilidade específica. Esses arquivos são carregados pelo arquivo principal da biblioteca, dessa forma, poderão ser removidos ou adicionados novos *middlewares*, tornando o *framework* ainda mais customizável.

Seguindo o requisito **RF07**, para o gerencimento dos dados de sessão do usuário, foi utilizado o *middleware* express-session, onde oferece um conjunto de recursos que permitem a persistência das informações diretamente no banco de dados MongoDB, além de compartilhá-las entre as requisições HTTP.

Para a disponiblização dos arquivos estáticos da aplicação, em conformidade com o requisito **RF05**, foi desenvolvido o serviço *static*, onde os arquivos pré estabelecidos na configuração *assets* são definidos pelo módulo express como públicos, permitindo o seu acesso externo por meio de requisições HTTP. Este método é indicado apenas em ambientes de desenvolvimento, pois a tecnologia Node.js não é eficiente para esta tarefa. Em vez disso, é indicado em ambientes de produção a utilização de um serviço externo como o Nginx [13], capaz
de servir esses arquivos com alta performance sem interferir no funcionamento da aplicação. Esta opção também está disponível no arquivo de configuração do sistema.

Para alcançar os requisitos **RF02** e **RF08**, foram utilizados recursos do módulo express que permitem o gerenciamento de rotas, delegando as requisições para handlers específicos, e a renderização das *views* HTML como resposta HTTP. Foram desenvolvidos arquivos de configuração que auxiliam no funcionamento destes recursos.

5.5.2.3 Desenvolvimento do Atlom Redis

O módulo Atlom redis possui grande relevância para os recursos que necessitam de uma persistência de alta performance com dados de chave e valor, geralmente utilizada para o gerenciamento de sessões HTTP, clusterização de socket, cache, dentre outras necessidades. Em

[13] https://nginx.org

seu desenvolvimento, foi utilizado o módulo redis, responsável por abstrair a comunicação e a persistência no banco de dados Redis por meio dos seus métodos.

A instância client da conexão foi exposta para toda a aplicação utilizando o módulo atlom. Assim como a biblioteca Atlom mongo, foram setados também os seus atributos de configuração para diferentes ambientes de execução.

5.5.2.4 Desenvolvimento do Módulo Atlom SocketIo

A biblioteca Atlom socketIo, responsável por servir uma comunicação *full-duplex* entre cliente e servidor, foi desenvolvida utilizando como base o módulo socketIo. Foram utilizadas também recursos para a persistência dos dados gerados em tempo de execução pelo módulo e o compartilhamento das informações da sessão do usuário geridos pela instância app do express. A instância io também foi servida para toda a aplicação utilizando o módulo atlom.

Assim, foi alcançado o requisito **RF03**, e concluído o desenvolvimento dos módulos base que irão servir os recursos chaves escaláveis para o desenvolvimento de qualquer aplicação *web* no contexto Node.js. Com o módulo atlom, essa estrutura pode sofrer alterações sem comprometer o funcionamento geral da aplicação em desenvolvimento, podendo adicionar, remover ou até mesmo alterar os seus componentes existentes.

5.5.3 Configuração do Ambiente Client

O ambiente *client* permite que o desenvolvedor implemente os componentes visuais, comunicação cliente-servidor, serviços auxiliares e toda as regras de negócio que constituirão o sistema em desenvolvimento, além dos ativos de imagens, estilos dentre outros arquivos.

O AngularJS foi o *framework* escolhido para servir de base na construção da aplicação, possuindo os recursos necessários para o desenvolvimento dos componentes citados anteriormente. A sua instalação e de todas as outras bibliotecas foram delegadas ao módulo Bower, módulo do Node.js responsável por organizar e instalar as dependências *front-end* do projeto.

O arquivo bower.json, gerado pelo Bower, possui a descrição de todos as dependências que serão instaladas pelo comando **bower install**. Dessa forma, além de manter uma padronização, otimiza o processo de *deploy*, uma vez que os arquivos brutos não serão carregados para o servidor de produção.

O carregamento das dependências na página será feito utilizando o método externo,

ou seja, cada arquivo será manualmente linkado na *view index* do projeto. É importante separá-los dos demais arquivos, pois estes podem ser disponibilizados por outros serviços que utilizam técnicas de otimização em seus carregamentos.

Com as bibliotecas instaladas e carregadas pela aplicação, o próximo passo constituiu em configurar o módulo angular app, responsável por conter todas as regras de negócio do cliente. Foram criados os arquivos app.module.js, responsável por iniciar o módulo app e suas dependências; o arquivo app.route.js que contém a configuração das rotas *single pages* do projeto, delegando as requisições das URL's aos seus respectivos componentes e controladores; e o componente *app-root*, pai de todos os outros componentes *front-end* da aplicação.

Com os módulos e suas configurações finalizadas, foi criado o arquivo atlom.route, responsável por interceptar e entregar a *view index* do projeto, contento o HTML com a lógica de carregamento dos arquivos javascript e css da aplicação. Desta forma, foi concluída a configuração completa do ambiente *client*.

5.5.4 Configuração das Tarefas Automatizadas

O processo de automação otimiza o desenvolvimento realizando tarefas que são repetitivas para o programador, tais como: a concatenação e o processamento de arquivos, transpilações, realização de testes unitários e *deploy*.

Na comunidade Javascript são encontradas duas ferramentas poderosas que são populares nesse contexo, são elas: Grunt[14] e Gulp. Devido ao seu algorítimo lógico de funcionamento, utilizando conceitos de pipes em memória, o Gulp exerce uma performance superior ao Grunt, e por esse motivo foi a ferramenta escolhida para este projeto.

Foram implementadas ao todo duas tarefas automatizadas que são indispensáveis para o desenvolvimento de qualquer aplicação *web*: (i) a *task build*: responsável pela concatenação dos arquivos javascript e css; e (ii) a *task test*: responsável pela execução dos testes unitários da aplicação. As *tasks* são inicializadas pelo arquivo gulpfile.js com as terminações *.task.js;

Atendendo ao requisito **RF09**, na *task build*, os arquivos javascript e css, pré estabelecidos na configuração *assets*, são minificados em um arquivo único para cada extensão, tornando o código da aplicação menor, e consequentemente otimizando ainda mais o processo de carregamento da página. Os principais módulos Gulp utilizados foram gulp-uglify, gulp-babel e gulp-cssmin.

[14] https://gruntjs.com/

Os arquivos minificados foram denominados como aggregateds.js e aggregateds.css, ficarão contidos na pasta build dentro de *assets*. Esses arquivos são servidos estaticamente pelo módulo Atlom express, e linkados diretamente na *view index* do projeto quando o ambiente de execução for o de produção. Esta tarefa atende ao requisito **RF10**.

Para a *task test*, foi definido que os arquivos com as terminações ***.spec.js** serão os responsáveis pelos testes unitários de cada serviço. Esses arquivos são carregados apenas no ambiente de testes. Os serviços testados são facilmente recuperados utilizando o DI do módulo atlom. Os módulos Mocha e Should foram os escolhidos para a implementação das regras de teste deste passo.

5.5.5 Desenvolvimento de um To Do List Básico

Após a finalização da implementação de todos os recursos planejados, foi desenvolvida uma aplicação modelo que servirá de referência para os projetos futuros que utilizarão o framework Atlom.js. No processo de desenvolvimento do *To Do List*, fizeram parte as tarefas: (i) criação do componente *back-end todo*: Responsável por conter a lógica de implementação das rotas, controladores e persistência dos todo's; e (ii) criação do componente *front-end todo*: Responsável pelos componentes visuais, controladores e a comunicação REST com o servidor.

5.5.5.1 Criação do Componente Todo no Back-end

O componente Todo possui basicamente 4 funcionalidades fundamentais, são elas: (i) *create*: Responsável pela criação e persistência dos *todo's*; (ii) *getAll*: Responsável por capturar todos os *todo's* cadastrados no banco de dados; (iii) *update*: Responsável pela atualização do elemento *todo*; e (iv) *remove*: Responsável pela remoção do elemento *todo*.

Primeiramente, foi necessário o resgate da instância app Express do projeto que contém todas as funcionalidades de manipulação HTTP da aplicação que foram configuradas na implementação do módulo Atlom express. Para isto, foi utilizado o método *run* do módulo Atlom app requisitando o recurso descrito pela injeção de dependências.

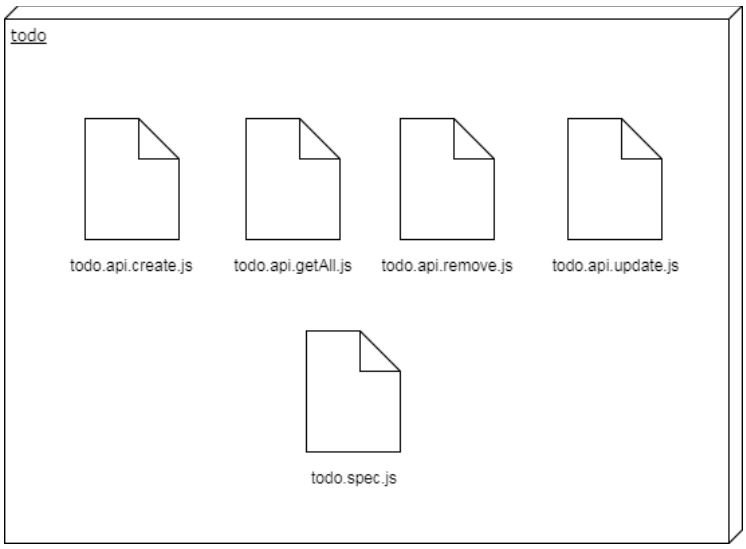
Antes da criação dos métodos, foi implementado o modelo TodoModel, utilizando a biblioteca mongoose, que será responsável por toda a persistência dos *todo's* no projeto. O esquema foi exposto como uma *factory* para ser utilizado em todo o módulo Atlom app.

Foi setada para cada rota dos métodos descritos, um *handle* que captura os parâmetros

de requisição que serão passados via formulário pelo componente *front-end*, contendo os dados necessários para cada operação. As operações serão processadas e delegadas para a factory TodoModel as responsabilidades de manipulação do banco de dados condizentes com cada procedimento.

Após a criação dos métodos, foram implementados as rotinas de testes que verificaram, por meio de requisições HTTP, se os dados vindos do servidor conferem com a resposta esperada de cada operação. O módulo Node.js *agent* foi utilizado para as requisições. A Figura 17 ilustra a organização dos arquivos que compõem o componente *todo* do *back-end*.

Figura 17 – Arquivos do componente todo no *back-end*



Fonte: Elaborada pelo Autor

5.5.5.2 Criação do Componente Todo no Front-end

Inicialmente, foi desenvolvido o serviço *todoService*, responsável por realizar a comunicação HTTP dos métodos de criação, remoção, atualização e captura dos *todo's*. A sua instância foi provida utilizando o método service para o módulo app. Desta forma, o serviço ficará disponível para os componentes que o requisitarem.

Foi criado o componente *todo*, contendo 3 arquivos básicos: (i) todo-list.css: Responsável pela estilização do componente; (ii) todo-list.ctrl.js: Responsável por conter a lógica de negócio; e (iii) todo-list.js: Responsável por definir o componente, inicializando o seu *template* HTML e o seu respectivo controlador.

O controlador expõe os 4 métodos descritos anteriormente para o seu template,

utilizando o objeto *scope* do angularJS. Assim, as ações acionadas pelo usuário serão captadas pelo controlador que decidirá qual procedimento executar. Os dados de resposta são automaticamente renderizados para usuário por meio do algoritmo *bidirecional bind*, que possibilita que as informações sejam atualizadas no momento da alteração do dado.

E para finalizar, foi declarada a diretiva *todo-list* dentro do componente *app-root*. Desta forma, o componente *todo* será inicializado quando a página for carregada. A Figura **??** ilustra a implementação do componente *todo* no *front-end*. O código completo da aplicação se encontra na página do Github: https://github.com/laerciogermano/atlom-mean-project.

6 VALIDAÇÃO DO *FRAMEWOK* ATLOM.JS

Nesta Seção será detalhada a validação do *framework* Atlom.js por meio de uma aplicação com programadores com experiências na linguagem Javascript. Será discutido o processo de planejamento, procedimentos, riscos à validade, execução e os resultados obtidos.

6.1 Planejamento do Experimento

O planejamento do experimento para a validação do *framework* é baseado nos seguintes tópicos:

- Objeto de estudo: *Framework* Node.js para aplicações web baseado em componentes;
- Propósito: Validação do *framework* Atlom.js;
- Foco do experimento: Analisar o alinhamento dos atributos de qualidade propostos pelo trabalho com o *framework* desenvolvido;
- Perspectiva do experimento: Acadêmico;
- Contexto do experimento: Programadores com experiências na linguagem Javascript;

Definidos os objetivos, as seguintes perguntas foram elaboradas para validar o experimento:

1. Os programadores que utilizaram o *framework* Atlom.js realizaram um maior número de atividades?
2. Os programadores que utilizaram o *framework* Atlom.js obtiveram uma maior capacidade de compreensão do sistema?
3. Os programadores que utilizaram o *framework* Atlom.js obtiveram uma maior facilidade para utilizar as *features* e desenvolver as atividades?
4. Os programadores que utilizaram o *framework* Atlom.js obtiveram uma maior facilidade para customizar e manipular as tecnologias utilizadas sem grandes impactos no comportamento do software?
5. Os programadores que utilizaram o *framework* Atlom.js obtiveram uma maior facilidade para a criação e reutilização dos componentes otimizando o processo de desenvolvimento do software?

Para avaliar as questões acima, foram utilizadas as seguintes métricas:

1. Quantidade de funcionalidades desenvolvidas: Avalia o número de tarefas finalizadas por cada programador;

2. Alinhamento entre as soluções e os requisitos do projeto: Avalia a qualidade de cada questão desenvolvida com relação ao objetivo descrito em cada passo;
3. Tempo médio gasto em pesquisa: Avalia o tempo médio gasto em pesquisas necessárias para a resolução das atividades;
4. Tempo médio gasto no desenvolvimento: Avalia o tempo de desenvolvimento necessário para a conclusão das atividades;e
5. Avaliação do *framework* pelos desenvolvedores: Permite uma avaliação do *framework* como uma ferramenta efetiva no desenvolvimento de *software*.

Com os objetivos e as métricas definidas, foram então consideradas as seguintes hipóteses.

- Hipótese H1: A utilização do *framework* Atlom.js não otimiza o processo de desenvolvimento de software conforme os atributos de qualidade planejados;
- Hipótese H2: A utilização do *framework* Atlom.js otimiza o processo de desenvolvimento de software conforme os atributos de qualidade planejados;

A hipótese H1 é a hipótese que desejou-se rejeitar, e a hipótese H2 é a que desejou-se aceitar. Desta forma, foi planejado o experimento que será aplicado aos participantes com intuito de validar o presente trabalho.

O experimento é composto por 4 programadores com experiências niveladas no âmbito Javascript e das tecnologias utilizadas pelo experimento. Os programadores são divididos em 2 grupos, onde as mesmas atividades deverão ser resolvidas individualmente utilizando dois *frameworks* distintos. Devido as similaridades de tecnologias, recursos e *scaffolding*, o Sails.js foi o *framework* de comparação escolhido.

O primeiro grupo, denominado como “Grupo Atlom.js”, ficou responsável pelo desenvolvimento das atividades utilizando o *framework* proposto neste trabalho. Já o segundo grupo, denominado como “Grupo Sails.js”, utilizou o *framework* concorrente. Essa divisão tem como objetivo possibilitar a análise do impacto da abordagem dos dois *frameworks* sobre os atributo de qualidade mencionados na fase de planejamento.

6.2 Execução do Experimento

Para a execução do experimento, foram escolhidos 4 programadores que possuíam experiências com a linguagem Javascript, bem como com as tecnologias relacionadas ao âmbito Node.js, tais como Express, MongoDB, RedisDB, Socket.io, dentre outros. Vale ressaltar que

todos são ex-alunos da Universidade Federal do Ceará, e atualmente encontram-se no mercado ou no mestrado. O experimento foi divido em 3 fases: treinamento, execução e questionário.

Na primeira fase (treinamento), foram dedicadas 2 horas exclusivas com cada participante, via Hangout [1], onde o autor do trabalho abordou de forma explicativa as tecnologias envolvidas, as restrições e os requisitos definidos em cada passo da atividade. Além disso, foi disponibilizado um guia de desenvolvimento com as questões implementadas sem a utilização de *frameworks*. O intuito do treinamento foi certificar que os níveis técnicos dos programadores estivessem equilibrados de forma que a não familiaridade com as tecnologias

interferisse no resultado do experimento.

Para auxiliar na fase de execução, os *frameworks* foram previamente configurados e testados para evitar quaisquer complicações de instalação na máquina do participante. Além disso, foi configurada uma VPS (*Virtual Private Server*) que disponibilizou, por meio de IP e portas, os bancos de dados MongoDB e RedisDB para serem aproveitados no procedimento, evitando também incompatibilidades ou ausência de recursos instalados nas máquinas dos programdores. Os *frameworks* e o guia de desenvolvimento foram disponibilizados no Github do autor.

Então, foram aplicadas 4 tarefas, ilustradas no Apêndice A, que pudessem abordar de forma incisiva os atributos de qualidade desejados por este trabalho, com ênfase na manipulação das tecnologias do projeto e no desenvolvimento de componentes, visando otimizar o processo de reuso de software na construção de novos sistemas.

Foi sugerido a cada participante que clonasse o respectivo projeto do seu grupo e criasse uma nova *branch*, de forma a melhorar na identificação das implementações e evitar possíveis conflitos de código. Após a finalização do teste, cada participante comitou suas alterações e atualizou o repositório do projeto no Github.

Todo procedimento foi realizado remotamente com a supervisão do autor do trabalho. Devido ao curto tempo disponível pelos participantes e a complexidade das questões, foi dado um prazo de 3 dias para a finalização das tarefas. O tempo de duração do teste foi provido por cada programador, com o auxílio dos *logs* gerados pela ferramenta *git*.

E por último, foi aplicado um questionário que abordou algumas questões objetivas com o intuito de medir as experiências dos participantes obtidas no decorrer do experimento, de forma a comparar os resultados de ambos os *frameworks* e validar se o Atlom.js de fato alcançou

[1] https://hangouts.google.com

as expectativas planejadas pelo trabalho. Além disso, também foram coletadas informações, por meio das observações do autor e análise do código final, com o objetivo de compreender de forma detalhada as dificuldades encontradas pelos programadores.

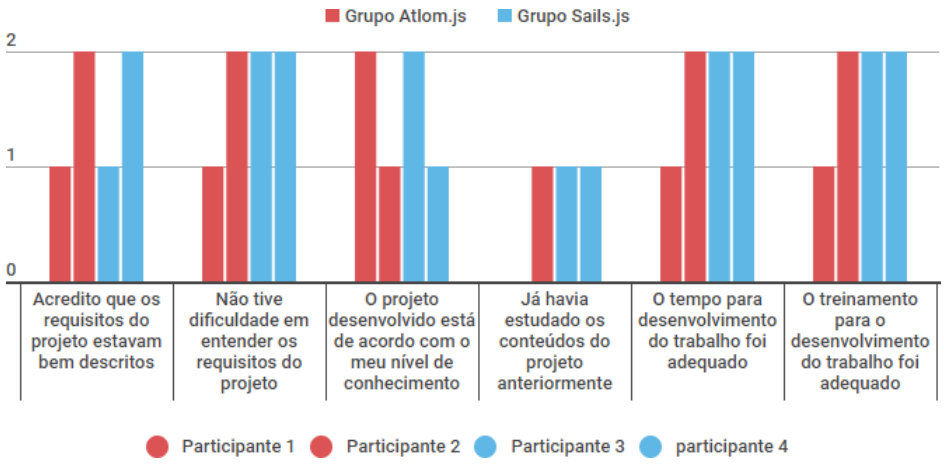
6.3 Resultados do Experimento

Após a aplicação das atividades, encontradas no apêndice A, e dos questionários, encontrados nos apêndices B e C, foi possível coletar os dados referentes as experiências dos programadores com seus respectivos *frameworks*. Todas as questões possuíam caráter objetivo em formatos de afirmações, onde o participante definia o grau de concordância utilizando a escala *Likert* que variava entre -2 (discordo fortemente) a +2 (concordo fortemente). Para uma melhor visualização, todas as informações foram compiladas em formato de gráficos.

6.3.0.1Resultados da Avaliação do Questionário

A seguir, serão apresentadas as informações coletadas pelos participantes referentes ao questionário dos requisitos do projeto. Foram avaliadas os sentimentos dos alunos com relação a descrição, compreensão, familiaridade com as tecnologias requisitadas, dentre outros pontos. A Figura 18 ilustra as respostas de cada participante e seu respectivo grupo.

Figura 18 – Resultado das experiências dos participantes referentes aos requisitos do projeto



Fonte: Elaborada pelo Autor

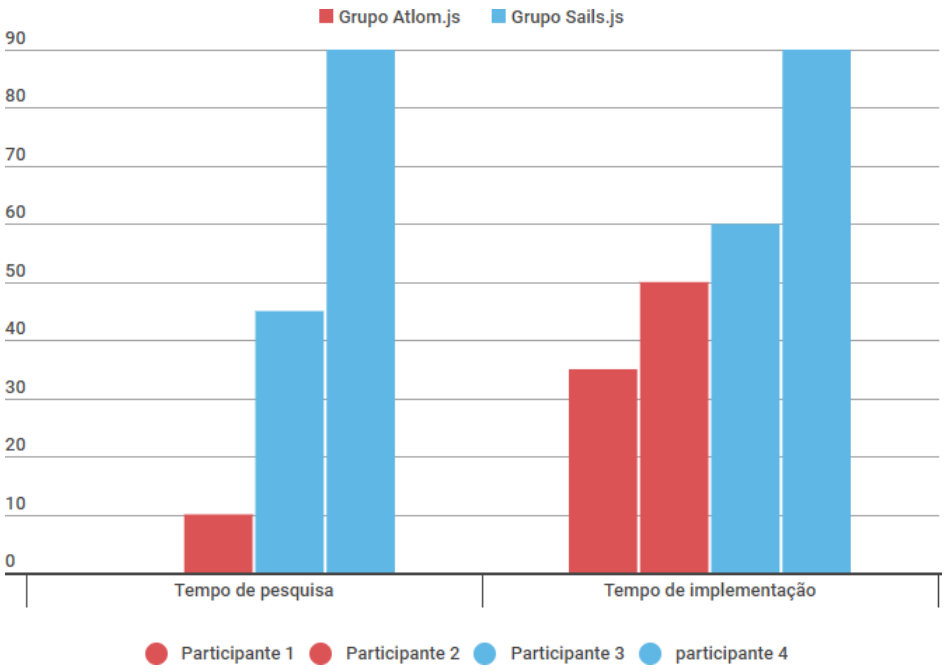
De acordo com a visualização do gráfico, pode-se perceber que os requisitos foram

bem descritos, onde os programadores não obtiveram dificuldades em compreendê-los, bem como o nível das questões estava apropriado ao conhecimento de cada participante. Além disso, o tempo e o treinamento para o desenvolvimento do trabalho também foram satisfatórios. Vale ressaltar que o "Grupo Sails.js"melhor avaliou a dificuldade em compreender os requisitos, o tempo e o treinamento cedido para o desenvolvimento das atividades.

6.3.1 Resultados do Tempo de Pesquisa e Desenvolvimento

A seguir, serão apresentados os dados referentes ao tempo de pesquisa e desenvolvimento necessários para a resolução completa das atividades. A Figura 19 ilustra em forma de gráfico o tempo que cada participante coletado e seu respectivo grupo.

Figura 19 – Resultado do tempo de pesquisa e desenvolvimento do projeto



Fonte: Elaborada pelo Autor

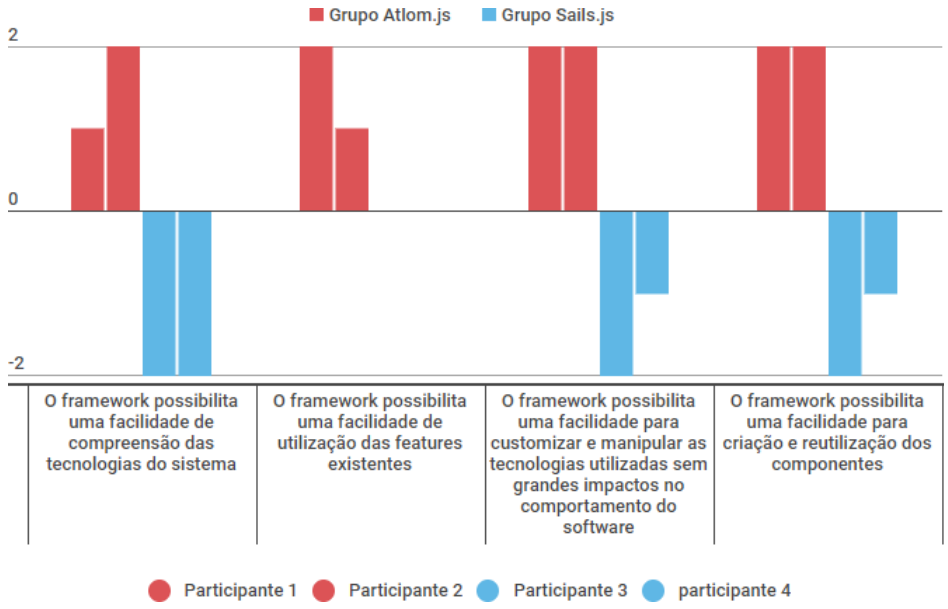
De acordo com as informações coletadas, pode-se perceber claramente que o tempo de pesquisa que o "Grupo Atlom.js"precisou para resolver as atividades foi inferior ao grupo concorrente, chegando a nulo com o participante 1, podendo concluir que o *framework* contribuiu para a diminuição da curva de aprendizagem, atingindo um dos seus principais objetivos. Em contra partida, o tempo de implementação de ambos os *frameworks* obteve

resultados equilibrados, não havendo superioridade excessiva por parte do "Grupo Atlom.js", com exceção do participante 4 que levou mais que 1 hora para a finalização dos exercícios. Tal resultado pode dever-se ao fato que a maior parte das atividades necessitou do manuseio de tecnologias associadas e não exclusivamente as *features* dos *frameworks*.

6.3.2 Resultados da Avaliação das Experiências com os Frameworks

A seguir, serão demonstradas as informações coletadas do questionário referente às experiências de cada participante com os seus respectivos *frameworks* durante o processo de implementação das atividades. A Figura 20 compila os dados em forma de gráfico para uma melhor visualização.

Figura 20 – Resultados da avaliação das experiências com os *frameworks*



Fonte: Elaborada pelo Autor

Pode-se perceber que o "Grupo Atlom.js"avaliou de forma positiva a facilidade de compreensão das tecnologias agregadas, podendo discernir de forma rápida, quais os recursos estão sendo utilizados e decidir se o *framework* atende as necessidades da aplicação a ser desenvolvida. Já o "Grupo Sails.js", discordou fortemente (-2) tal afirmação, alegando durante a fase de desenvolvimento que o *framework* abstraia em diferentes camadas a implementação base de cada tecnologia, levando a incompreensão do seu funcionamento. Desta forma, pode-se

concluir que outro objetivo importante do trabalho foi alcançado.

Com relação a facilidade de utilização das *features*, o *framework* Atlom.js obteve superioridade em seus resultados, podendo concluir que os requisitos do *framework* desenvolvidos, além de alinhados com o planejado, foram utilizados sem maiores complicações. Já o "Grupo Sails.js"avaliou de forma neutra tal afirmação.

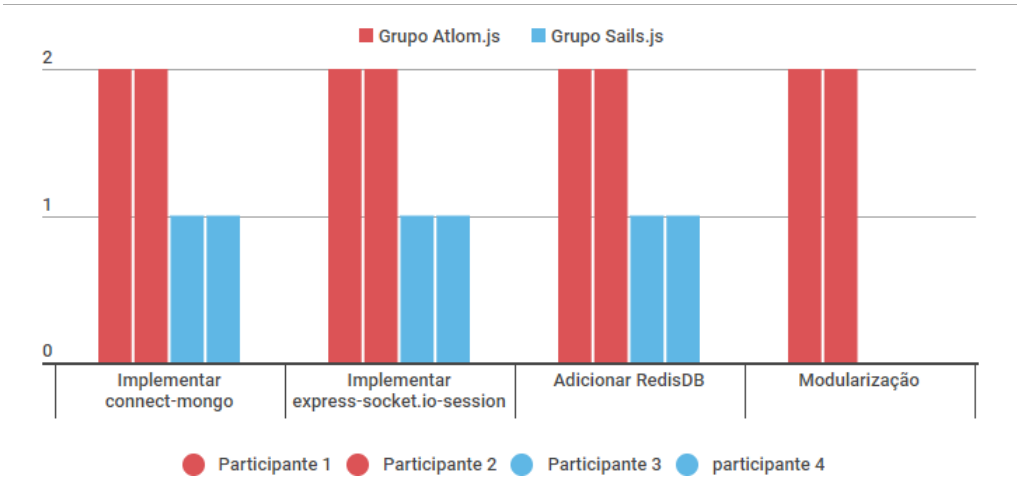
No aspecto voltado a manipulação das tecnologias sem grandes impactos, o "Grupo Atlom.js", por unanimidade, concordou fortemente (+2) que o *framework* Atlom.js possibilitou uma facilidade de customização dos recursos agregados, alegando que a abordagem utilizando a injeção de dependências do AngularJS de fato potencializou tal capacidade. O "Grupo Sails.js"discordou de tal afirmação. Além disso, durante a análise do código, foi observado que os participantes que utilizaram o *framework* Sails.js obtiveram uma maior dificuldade em resgatar as instâncias necessárias para o desenvolvimento fidedigno das atividades, sendo avaliados negativamente nos resultados referentes ao alinhamento dos requisitos posteriormente.

Na esfera do reuso de *software*, o "Grupo Atlom.js"também obteve a unanimidade de concordância (+2) com a afirmativa de que o *framework* possibilitou uma facilidade em criar e reutilizar novos componentes, alegando também que tal resultado foi alcançado devido a sua injeção de dependências. Já o "Grupo Sails.js"discordou com tal afirmação. Com base nos depoimentos dos participantes, pôde-se observar que o *framework* Sails.js possuía uma complexidade maior, tanto no código como na documentação, para criação de módulos nativos, impedindo com que os programadores concluíssem a última questão do experimento, sendo avaliados negativamente em resultados futuros.

6.3.3 Resultados da Avaliação das Questões Implementadas

A seguir, serão apresentados os dados referentes ao alinhamento das implementações com os seus respectivos requisitos, tais informações tem como objetivo avaliar a qualidade das soluções desenvolvidas de cada participante com relação aos objetivos descritos em cada atividade do experimento. Para levantar tais dados, foram executas análises no código final da avaliação pelo autor do trabalho. A Figura 21 aponta o resultado da análise de cada questão.

Figura 21 – Resultado da avaliação do alinhamento dos requisitos

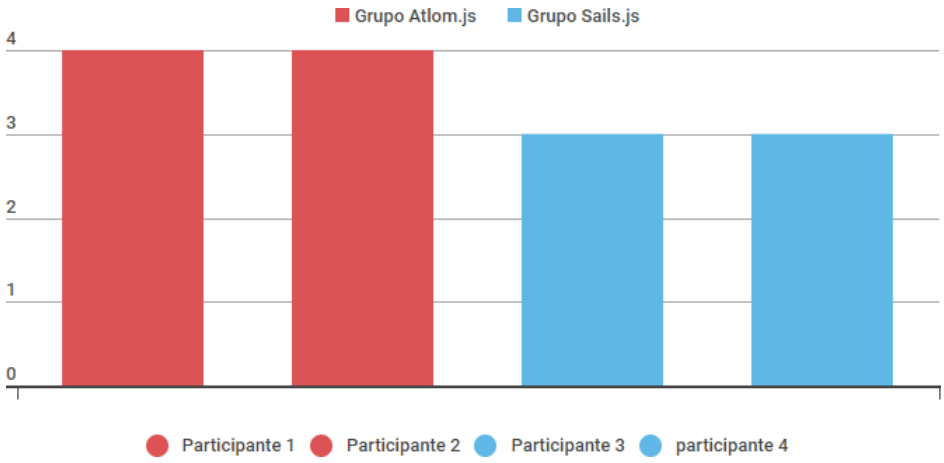


Fonte: Elaborada pelo Autor

Olhando para os gráficos, pode-se perceber que os participantes do "Grupo Atlom.js"conseguiram desenvolver todas as atividades com êxito, solucionando todos os seus respectivos desafios. Os participantes do "Grupo Sails.js"também obtiveram resultados positivos em suas implementações, com exceção da última questão que abordava a modularização. Entretanto, as soluções desenvolvidas pelo grupo azul não seguiram fielmente as características impostas por cada questão. Segundo os participantes, tal fato deu-se a dificuldade encontrada em compreender as técnicas de modularização impostas pela documentação do *framework* Sails.js. Desta forma, conforme citado anteriormente, alguns pontos foram descontados pelo desvio das soluções apresentadas com o resultado esperado.

E para finalizar, a Figura 22 ilustra o número de questões desenvolvidas por cada participante. Como no resultado anterior, tais questões também foram analisadas por meio do código final, verificando se cumpriam com os objetivos descritos, entretanto, neste resultado não foi levado em consideração se as soluções estavam dentro dos padrões esperados.

Figura 22 – Resultado da avaliação do número de questões concluidas



Fonte: Elaborada pelo Autor

Portanto, pode-se concluir com a análise dos resultados que o *framework* Atlom.js obteve uma boa avaliação por parte dos participantes como uma ferramenta que facilita o desenvolvimento de aplicações web, diminuindo a curva de aprendizagem, facilitando a utilização e a manipulação dos recursos agregados, além de propor um modelo de modularização que auxilia na criação de novos componentes, potencializando o reuso efetivo de *software*.

7 CONSIDERAÇÕES FINAIS

O presente trabalho propôs uma solução que otimizasse o processo de desenvolvimento de aplicações web com foco no reuso, baixa curva de aprendizagem e alto poder de manipulação de tecnologias agregadas. Para resolver tal desafio, foi proposto readaptar a injeção de dependências do AngularJS para o contexto *server-side*, bem como implantar padrões e conceitos modernos abordados no mercado, para criação do *framework* Atlom.js.

Inicialmente, foram levantados os *frameworks* existentes mais populares com o intuito de identificar as funcionalidades mais recorrentes presentes nas documentações de cada *framework*. Então, foram definidos os requisitos que seriam inseridos no escopo de desenvolvimento do projeto, levando em consideração a complexidade, número de dependências e ao tempo requerido de implementação.

Foram recuperadas e qualificadas as tecnologias que seriam utilizadas para o desenvolvimento dos requisitos, dentre elas, foi encontrado o módulo ng-di que otimizou o processo de adaptação dos métodos originais da injeção de dependências do AngularJS para o contexto *back-end*.

Durante a fase de implementação, foram seguidos sequencialmente a codificação de cada componente descrito durante a fase de planejamento. Desta forma, ao final do processo, todos os componentes foram implementados e testados, garantindo que todos os passos planejados foram devidamente seguidos.

Após a implementação, foram realizados experimentos com 4 programadores de nível técnico avançado em Javascript, separados em grupos de 2 integrantes: "Grupo Atlom.js"e "Grupo Sails.js", com o intuito de avaliar se o trabalho final obteve os resultados esperados.

Com os resultados obtidos, pôde-se perceber que, embora o tempo de implementação tenha sido equiparável, o tempo de pesquisa do "Grupo Atlom.js"para o desenvolvimento das atividades foi consideravelmente menor que o do grupo concorrente. Além disso, o "Grupo Atlom.js"obteve também melhores resultados na facilidade de compreensão, podendo discernir de forma rápida se o *framework* atende as necessidades do projeto, bem como na facilidade de utilização, facilidade na manipulação das tecnologias agregadas e no reuso de software.

Foram analisados os códigos finais de cada participante com o intuito de medir o alinhamento das soluções desenvolvidas com os seus respectivos requisitos, e pôde-se

perceber que os programadores que utilizaram o *framework* Atlom.js obtiveram não só um maior número de questões concluídas, como também uma melhor qualidade no código, atendendo ao que foi

solicitado pelas questões.

Devido ao alto nível técnico necessário para o manuseio de um *framework* que atua em um campo muito específico do desenvolvimento de *software*, foram encontrados poucos programadores que tivessem o perfil e a disponibilidade para a realização dos experimentos. Entretanto, a qualidade dos programadores escolhidos influenciou diretamente para que os resultados finais se aproximassem ao seu real contexto de aplicação (mercado).

Embora o modelo de componentes tenha sido definido neste trabalho, é necessário elaborar estratégias de compartilhamento de módulos para futuras reutilizações. Portanto, para trabalhos futuros é sugerida a criação de mecanismos de compartilhamento que possibilite o resgate e a publicação de novos componentes utilizando o NPM (*Node Package Manager*), para que futuramente possam ser integrados em projetos que utilizarem o Atlom.js como *framework* base de desenvolvimento.

REFERÊNCIAS

ALMEIDA, E. S.; ALVARO, A.; GARCIA, V. C.; BURÉGIO, V. A.; NASCIMENTO, L. M.; LUCRÉDIO, D. et al. **CRUISE. Component Reuse in Software Engineering**. CESAR e-book, 2007.

BACHMANN, F.; BASS, L.; BUHMAN, C.; COMELLA-DORDA, S.; LONG, F.; ROBERT, J.; SEACORD, R.; WALLNAU, K. **Technical Concepts of Component-Based Software Engineering**. 2nd. ed. Pittsburgh, PA, 2000. Disponível em: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5203>. Acesso em: 21 dez. 2018.

BROWN, A. W. . S.; K. On components and objects: The foundations of component-based development. **International Symposium on Assessment of Software Tools, IEEE Computer Society, Los Alamitos, CA, USA, v**, v. 0, p. 0112, 1997.

CANTELON, M.; HOLOWAYCHUK, T. **Node.js in action**. [S.l.]: S.n, 2011.

DAYLEY, B. **Node.js, MongoDB, and AngularJS Web Development**. Pearson Education, 2014. (Developer's Library). ISBN 9780133844344. Disponível em: <https://books.google.com.br/books?id=8kTCAwAAQBAJ>. Acesso em: 21 dez. 2018.

D'SOUZA, D. F.; WILLS, A. C. **Objects, Components, and Frameworks with UML**: The catalysis approach. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-31012-0.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. **Commun. ACM**, ACM, New York, NY, USA, v. 40, n. 10, p. 32–38, out. 1997. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/262793.262798>. Acesso em: 21 dez. 2018.

GIMENES, I. D. S.; HUZITA, E. **Desenvolvimento baseado em componentes: conceitos e técnicas**. Ciência Moderna, 2005. ISBN 9788573934069. Disponível em: <https://books.google.com.br/books?id=2_jvXwAACAAJ>. Acesso em: 21 dez. 2018.

HEINEMAN, G. T.; COUNCILL, W. T. (Ed.). **Component-based Software Engineering**: Putting the pieces together. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70485-4.

IYER, G. R. **Node.js**: Event-driven concurrency for web applications. 2013. Disponível em: <https://www.researchgate.net/profile/Ganesh_Iyer6/publication/280546121_Nodejs_Event-driven_Concurrency_for_Web_Applications/links/55b87f3e08ae9289a08d5dbf.pdf?origin=publication_detail>. Acesso em: 21 dez. 2018.

JAIN, N.; MANGAL, P.; MEHTA, D. Angularjs: A modern mvc framework in javascript. **Journal of Global Research in Computer Science**, v. 5, n. 12, p. 17–23, 2015.

JOYENT. **Node's goal is to provide an easy way to build scalable network programs'**. 2012. Disponível em: <http://nodejs.org/about>. Acesso em: 21 dez. 2018.

KOZLOWSKI, P. **Mastering Web Application Development with AngularJS**. Packt Publishing, 2013. (Community experience distilled). ISBN 9781782161837. Disponível em: <https://books.google.com.br/books?id=mZXjwz5X08EC>. Acesso em: 21 dez. 2018.

MARKIEWICZ, M. E.; LUCENA, C. J. P. de. Object oriented framework development. **Crossroads**, ACM, New York, NY, USA, v. 7, n. 4, p. 3–9, jul. 2001. ISSN 1528-4972. Disponível em: <http://doi.acm.org/10.1145/372765.372771>. Acesso em: 21 dez. 2018.

MCILROY, D. Mass-produced software components. In: NAUR, P.; RANDELL, B. (Ed.). **Proceedings of NATO Software Engineering Conference**. Garmisch, Germany: [s.n.], 1968. p. 138–155.

SAMETINGER, J. **Software Engineering with Reusable Components**. New York, NY, USA: Springer-Verlag New York, Inc., 1997. ISBN 3-540-62695-6.

SCHWARZ, J.; SOMMER, H.; FARRIS, A. **The ALMA Software System**. ASP Conf. Ser., Vol. 314 Astronomical Data Analysis Software and Systems XIII, Ochsenbein, F., Allen, M. Egret, D. (eds), San Francisco, ASP, v. 314, n. 643, 2003.

SOMMERVILLE, I. **Software Engineering**. 9.. ed. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1.

STANOJEVIC´ , V.; VLAJIC´ , S.; MILIC´ , M.; OGNJANOVIC´ , M. Guidelines for framework development process. In: IEEE. **Software Engineering Conference in Russia (CEE-SECR), 2011 7th Central and Eastern European**. [S.l.], 2011. p. 1–9.

SZYPERSKI, C.; GRUNTZ, D.; MURER, S. **Component Software**: Beyond object-oriented programming. ACM Press, 2002. (ACM Press Series). ISBN 9780201745726. Disponível em: <https://books.google.com.br/books?id=U896iwmtiagC>. Acesso em: 21 dez. 2018.

TAHIR, M.; KHAN, F.; BABAR, M.; ARIF, F.; KHAN, F. Framework for better reusability in component based software engineering. **the Journal of Applied Environmental and Biological Sciences (JAEBS)**, v. 6, p. 77–81, 2016.

TILKOV, S.; VINOSKI, S. Node. js: Using javascript to build high-performance network programs. **IEEE Internet Computing**, IEEE Computer Society, v. 14, n. 6, p. 80, 2010.

TIWARI, A.; CHAKRABORTY, P. S. Software component quality characteristics model for component based software engineering. In: IEEE. **Computational Intelligence & Communication Technology (CICT), 2015 IEEE International Conference on**. [S.l.], 2015. p. 47–51.

VALE, T.; CRNKOVIC, I.; ALMEIDA, E. S. de; NETO, S.; CAVALCANTI, Y. C.; MEIRAD, S. R. de L. Twenty-eight years of component-based software engineering. **Journal of Systems and Software January 2016, 11**, Elesvier, v. 111, n. 1, p. 128–148, January 2016. Disponível em: <http://www.es.mdh.se/publications/4272->. Acesso em: 21 dez. 2018.

WELSH, M.; GRIBBLE, S. D.; BREWER, E. A.; CULLER, D. **A Design Framework for Highly Concurrent Systems**. Berkeley, CA, USA, 2000.

ZELDOVICH, N.; YIP, A.; DABEK, F.; MORRIS, R. T.; MAZIERES, D.; KAASHOEK, F. Multiprocessor support for event-driven programs. **Proceedings of the 2003 USENIX**, 2003.

APÊNDICE A – TAREFAS DA VALIDAÇÃO

As atividades descritas serão aplicadas individualmente aos integrantes dos grupos Atlom.js e Sails.js. Algumas informações importantes devem ser consideradas durante a execução de cada atividade, são elas:

- **As tarefas devem ser desenvolvidas com foco no reuso de software. Portanto, é aconselhável utilizar o sistema de módulos/hooks do *framework* designado**;
- **Para a resolução das questões é necessário ter apenas o Node.js instalado na última versão v0.8.* e uma IDE de sua preferência**;
- **Para a conexão do banco de dados MongoDB e RedisDB, utilize o IP e a porta configurada na implementação dos exemplos passados para a realização do teste**;

1. connect-mongo é um *middleware Express* que persiste as informações da sessão HTTP no banco de dados MongoDB. Implemente o *middleware* na instância *app Express* do projeto existente e compartilhe a instância *session* para ser reutilizada em toda a aplicação.
2. express-socket.io-session é uma módulo Node.js que compartilha as informações da sessão HTTP com a instância io do Socket.io. Implemente o módulo na instância io do projeto existente, utilizando a instância *session* compartilhada na tarefa anterior, de forma que a configuração esteja presente em toda a aplicação.
3. Adicione o RedisDB como uma segunda alternativa de persistência utilizando o módulo Node.js redis. Exponha a instância da conexão para toda a aplicação.
4. Organize o código das tarefas anteriores, em pasta separada do projeto, de forma a reutilizá- lo em aplicações futuras.

APÊNDICE B – QUESTIONÁRIO DO PROJETO

As próximas questões são referentes ao projeto desenvolvido. Dificuldade, nível de descrição dos requisitos, etc. Todas as perguntas envolvem uma escala de -2 a +2, na qual o -2 está mais próximo do Discordo Fortemente e o +2 do Concordo Fortemente. Marque um número com um [x] de acordo com o quanto você concorda ou discorda de cada afirmação abaixo.

1. Acredito que os requisitos do projeto estavam bem descritos. [ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2
2. Não tive dificuldade em entender os requisitos do projeto. [ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2
3. O projeto desenvolvido está de acordo com o meu nível de conhecimento. [ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2
4. Já havia estudado os conteúdos do projeto anteriormente. [ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2
5. O tempo para desenvolvimento do trabalho foi adequado. [ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2
6. O treinamento para o desenvolvimento do trabalho foi adequado. [ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2

APÊNDICE C – QUESTIONÁRIO DO *FRAMEWORK*

As próximas questões são referentes ao *framework* utilizado. Dificuldade, nível de descrição dos requisitos, etc. Todas as perguntas envolvem uma escala de -2 a +2, na qual o -2 está mais próximo do Discordo Fortemente e o +2 do Concordo Fortemente. Marque um número com um [x] de acordo com o quanto você concorda ou discorda de cada afirmação abaixo.

1.O *framework* possibilita uma facilidade de compreensão das tecnologias do sistema.

[ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2

2.O *framework* possibilita uma facilidade de utilização das *features* existentes.

[ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2

3. O *framework* possibilita uma facilidade para customizar e manipular as tecnologias utilizadas sem grandes impactos no comportamento do software.

[ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2

4.O *framework* possibilita uma facilidade para criação e reutilização dos componentes.

[ ] -2, [ ] -1, [ ] 0, [ ] +1, [ ] +2