ANTONIO MENEGHETTI FACULDADE - AMF
CURSO DE SISTEMAS DE INFORMAÇÃO

CARLOS EDUARDO PEDRASSANI

UMA SOLUÇÃO EM NODEJS E REACT NATIVE PARA BUSCA E OFERTA DE EMPREGO

RESTINGA SECA/RS

2018

CARLOS EDUARDO PEDRASSANI

UMA SOLUÇÃO EM NODEJS E REACT NATIVE PARA BUSCA E OFERTA DE EMPREGO

Trabalho de Conclusão do Curso de Sistemas de Informação
Orientador: Profº. Dr. Rafael Pereira

RESTINGA SECA/RS

2018

FACULDADE ANTONIO MENEGHETTI

Carlos Eduardo Pedrassani

UMA SOLUÇÃO EM NODJES E REACT NATIVE PARA BUSCA E OFERTA DE EMPREGO.

Trabalho de Conclusão de Curso-Monografia, apresentado como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação, Curso de Graduação em Sistemas de Informação, Faculdade Antonio Meneghetti-AMF.

Orientador: Prof. Dr. Rafael Teodósio Pereira

Prof. Dr. Rafael Teodósio Pereira
Orientador do Trabalho de Conclusão de Curso
Antonio Meneghetti Faculdade

Profª Dr. Felipe Becker Nunes
Membro da Banca Examinadora
Antonio Meneghetti Faculdade

Profª Ms. Gabriel Marchesan
Membro da Banca Examinadora
Antonio Meneghetti Faculdade

Restinga Sêca, RS, 28 de novembro de 2018.

AGRADECIMENTOS

Aos meus pais **Adeleir Pedrassani** e **Jane Pedrassani** que me foram professores das coisas mais importantes da vida e de todos os ensinamentos que fora me passado. Todas as palavras aqui escritas serão poucas ao tamanho da gratidão que sinto por vocês. Aos dois que abriram mão de realizarem os seus desejos pessoais para ajudarem a alcançar este objetivo.

A todos os **familiares** e **amigos** que de alguma maneira me contribuíram para que eu realizasse esse projeto, aos meus colegas de trabalho que dedicaram um pouco de tempo para me ajudarem em pesquisas e estudos para alcançar a finalização deste documento.

A todos os meus **professores** do curso de Sistemas da Informação, que tive o prazer de ter aulas ou de ouvir os ensinamentos. Obrigado a todos pela dedicação em transmitir os conhecimentos que possuem, incentivando a busca por conhecimento constante.

Ao meu professor e orientador **Rafael Pereira** que com a sua experiência e conhecimento me auxiliou para a realização desse projeto. Obrigado por toda a paciência, dedicação e aprendizado.

Aos professores da disciplina **FOIL** que passaram conhecimentos que contribuam diretamente na minha vida profissional, me tornando um profissional diferenciado em relação a outros.

Ao patrono desta instituição, **Acad. Prof. Antonio Meneghetti**, que apesar de não estar referenciado neste trabalho, contribuiu de forma significativa na minha formação acadêmica e vida pessoal.

A todos, meu muitíssimo obrigado!

“As oportunidades multiplicam-se à medida que são agarradas. ”

Sun Tzu

RESUMO

A busca de oportunidades de emprego, nem sempre são fáceis para quem necessita de urgência, empresas podem levar meses até encontrarem alguém capaz de suprir uma determinada vaga e, pessoas que se encontram disponível no mercado as vezes não conseguem encontrar oportunidades de trabalho. Em face disso, a criação de um sistema que seja capaz de diminuir essa distância entre empregadores e possíveis funcionários, pode estreitar esta comunicação. Diante deste contexto, o objetivo deste trabalho é estudar as melhores tecnologias disponíveis para criar uma web API e um aplicativo nativo em Android e iOS e, por conseguinte, realizar a criação de um sistema que seja capaz de registrar possíveis empregados e empregadores, juntamente com as vagas que serão disponibilizadas por cada empregador. Foi então, desenvolvida uma web API RESTful, criada utilizando a tecnologia NodeJS, contendo os métodos e funções necessárias para todos as ações CRUD (*Create*, *Read*, *Update* e *Delete*) e iniciado o desenvolvimento de um aplicativo feito utilizando a tecnologia React Native, capaz de criar um aplicativo nativo multiplataforma e esse capaz de efetuar as requisições para a web API.

Palavras-chave: NodeJS; React Native; aplicativo nativo; Api RESTful.

ABSTRACT

The search for employment opportunities is not always easy for those in need of urgency, it may take companies for months to find someone who can fill that vacancy, and people who are available in the market sometimes can not find opportunities, much less in their area specialization. In the face of this, the creation of a system that is capable of bridging this distance between employers and potential employees, and can open a short distance to get communication. Given this context, the goal of this work is to study the best technologies available to create a web API and a native application on Android and iOS and, therefore, create a system that is able to register possible employees and employers along with the vacancies that will be made available by each employer. A RESTful API web was developed, using the NodeJS technology, containing the necessary methods and functions for all CRUD actions and started the development of an application made using React Native technology, able to create a native multi platform application and capable to make the *request*s to the web API.

Keywords: NodeJS; React Native; Native Aplication; RESTful.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

JS - JavaScript

IDE - *Integrated Development Environment*

JSON - *JavaScript Object Notation*

CSS - *Cascading Style Sheets*

HTML - *Hypertext Markup Language*

XML - *Extensible Markup Language*

API - *Application Programming Interface*

REST - *Representational State Transfer*

MVC - *Model-View-Controller*

HTTP - *Hypertext Transfer Protocol*

URL - *Uniform Resource Locator*

RN - React Native

JSX - *JavaScript eXtension*

NPM - *Node Package Manager*

VM - *Virtual Machine*

SDK - *Software Development Kit*

LAN - *Local Area Network*

URI - *Uniform Resource Identifier*

SOAP - *Simple Object Access Protocol*

CRUD - *Create, Read, Update e Delete*

BD - Banco de Dados

UUID - *Universally Unique IDentifier*

DaaS - *Database as a Service*

SUMÁRIO

1. INTRODUÇÃO

A tecnologia vem se expandindo cada vez mais, grandes empresas estão investindo muito para o seu crescimento, assim tornando cada dia mais fácil algumas tarefas humanas. Com a tecnologia também surgiu algo inovador que está sempre em evolução, que é chamado de telefone inteligente. O *smartphones* atualmente é um dispositivo indispensável, para fazer qualquer tarefa deve-se levá-los junto. Ele tem como principal objetivo facilitar o dia-a-dia, fazendo ligações, mandar mensagens instantâneas, efetuar cálculos, marcar agendamentos no calendário, gravar uma hora para despertar, entre outras funções e inúmeros aplicativos que podem ser utilizados.

Com a escassez de oportunidades de trabalho juntamente com a falta de mão de obra qualificada, muitas pessoas têm buscado trabalhos temporários em restaurantes, bares, pizzarias, etc..... Muitas empresas precisam desses funcionários em finais de semana ou em dias especiais, eventos ou até em dias comuns. Um facilitador na hora desses dois mundos se comunicarem seria de grande valia e aumentaria a produtividade, já que diminuiria o tempo de procura em ambos os casos. Num mundo onde o tempo e o dinheiro são cruciais para o crescimento profissional, onde tudo deve ser feito com agilidade e rapidez, um *smartphone* pode ser um ajudante nesses momentos.

Para que se possa encontrar funcionários, com disponibilidade e com interesse, pode-se demorar muitas horas em telefonemas ou em mensagens, para funcionários que procuram oportunidades, leva-se tempo excessivo que não é necessário, muitas vezes não encontram nada e perdem todo esse tempo que poderiam estar fechando a vaga com outra empresa. Criar um aplicativo que facilitaria e agilizaria esse processo de buscar e aceitação de empregos, é uma forma de auxiliar empresas e funcionários a não perderem tanto tempo nesse processo que em excesso torna-se chato. Dando a responsabilidade ao marketplace do aplicativo, que fará o processo de buscar os cadastros de funcionários e de empresas e suas respectivas oportunidades de empregos, assim mostrando em forma de feed para os perfis correspondentes.

1.1 OBJETIVOS

Nesta seção serão descritos os principais objetivos deste trabalho, como o de criar um aplicativo que auxilie no processo de busca e oferecimento de vagas de trabalhos temporários, em que candidatos possam buscar oportunidades e empregadores possam oferecer vagas e aceitar possíveis funcionários.

1.1.1 Objetivo principal

O presente trabalho tem como objetivo geral, estudar as tecnologias de desenvolvimento web e nativas para a criação de uma web API e início do desenvolvimento de um aplicativo e que possa aproximar pessoas que estão fora do mercado de trabalho, ou apenas, estão buscando um trabalho temporário, de oportunidades que possam surgir em empresas cadastradas no sistema. Permitir uma rápida busca e interação entre candidato e empregador, facilitando e diminuindo o processo de contratação e de busca de oportunidades.

1.1.2 Objetivos específicos

a) Utilizar as tecnologias e *frameworks* baseados em JavaScript mais atuais no mercado em seu ambiente de desenvolvimento nativo, pois são ferramentas com um alto índice de aceitação pelas comunidades.
b) Aplicar o padrão de arquitetura capaz de utilizar camadas de visualização feitas em React Native[1], consumindo uma *Application Programming Interface* (API*) Representational State Transfer* (REST), que será a camada de controle e modelo do projeto, assim, deixando mais dinâmico o desenvolvimento e com mais disponibilidade de requisições multiplataformas.
c) Criar um aplicativo multiplataforma capaz de fazer requisições para uma *API REST* e ela ser capaz de efetuar as principais funcionalidades, tais como: cadastro, alteração, busca e exclusão de dados.

[1] Disponível em: <https://facebook.github.io/react-native/>. Acessado em: 23 de agosto de 2018.

d) Utilizar o banco de dados não relacional, orientado a documentos *JavaScript Object Notation* (JSON)[2], chamado MongoDB[3]. Desta forma, a aplicação fica responsável por todo o manuseio dos dados, capaz de definir como os documentos se relacionam e quais atributos possuem, assim deixando o aplicativo rápido, escalável e simples.
e) Iniciar a criação de um layout amigável, de fácil utilização, fácil manutenção e que seja capaz de fazer com que pessoas que necessitem de emprego, se aproximem ou se relacionem com possíveis empregadores.

[2] Disponível em: <https://www.json.org/>. Acessado em 24 de agosto de 2018.
[3] Disponível em: <https://www.mongodb.com/>. Acessado em 10 de outubro de 2018

1.2 JUSTIFICATIVA

O interesse pelo tema se dá pela curiosidade do autor em desenvolver sistemas para aplicativos móveis e sobre a imensa gama de formas de se desenvolver aplicativos nativos. Além disso, de criar um aplicativo que não se encontra no mercado, tendo como objetivo principal, criar uma forma de aproximar empregadores, que possuem vagas de empregos, de funcionários que sejam capazes de se candidatarem a mesma, tornando ágil o processo de busca e oferecimento de oportunidades de emprego, já que, segundo o IBGE, o número de desempregados no Brasil nos três primeiros meses de 2018 foi de 13,7 milhões de pessoas[4].

Este trabalho tem também o objetivo de facilitar o dia-a-dia de quem possui um *smartphone* e se encontra sem tempo para ficar pesquisando na internet, sendo possível cadastrar uma vaga de emprego ou procurar uma vaga de emprego em qualquer lugar que esteja. O interesse em aprender e utilizar futuramente as tecnologias que serão utilizadas no projeto, também é uma motivação para o seu desenvolvimento. Todas as ferramentas utilizadas nesse projeto serão estudadas no seu desenvolvimento, já que são pouco utilizadas diariamente pelo autor.

A motivação do autor em utilizar JavaScript para a criação desse projeto vem do aumento expressivo em que esta tecnologia teve em relação as demais. Segundo o indicie de maiores tecnologias utilizadas, disponibilizada pelo GitHub[5], o JavaScript está na quinta posição, com isso tornou-se possível utilizar esta tecnologia em diversas plataformas e diversos tipos de projetos.

[4] Disponível em: < https://economia.uol.com.br/empregos-e-carreiras/noticias/redacao/2018/04/27/desemprego-pnad-ibge.htm?cmpid=copiaecola >.
[5] Disponível em: <https://octoverse.github.com/2017/>. Acessado em 17 de setembro de 2018.

2. REFERENCIAL TEÓRICO

Esse capítulo apresentará uma descrição teórica das ferramentas utilizadas e os principais motivos para a escolha de segui-las durante o desenvolvimento do projeto. Na seção 2.1 demonstrará a arquitetura que será seguida para o desenvolvimento da aplicação. Na seção 2.2 será mostrado as ferramentas que foram utilizadas para constituir o *front end* do aplicativo, juntamente com sua explicação. Na seção 2.3 será abordado as ferramentas utilizadas no *back end* da aplicação, contendo suas explicações. Na seção 2.4 será descrito o banco de dados não relacional que será utilizado

2.1 MODELO ARQUITETURAL

O glossário do site oficial *Software Engineering Institute* (SEI)[6], descreve que arquitetura de software é como uma estrutura ou estruturas de um sistema, essas estruturas possuem elementos, cada elemento é visível ao todo e todos outros elementos possuem a capacidade de verem seus atributos. Sommerville (2011, p. 105) define que um projeto de arquitetura é um processo criativo que é possível projetar uma organização de sistema capaz de satisfazer os requisitos pré-es*tab*elecidos. Ele também se refere a arquitetura como um conjunto de decisões ao invés de uma sequência lógica de atividades.

A arquitetura de software define o que é o sistema em termos de componentes computacionais e, os relacionamentos entre estes componentes, os padrões que guiam a sua composição e restrições (SHAW; GARLAN, 96).Além da escolha dos algoritmos e estruturas de dados, a arquitetura envolve: decisões sobre as estruturas que formarão o sistema, controle, protocolos de comunicação, sincronização e acesso a dados, atribuição de funcionalidade a elementos do sistema, distribuição física dos elementos escalabilidade e desempenho e outros atributos de qualidade; e seleção de alternativas de projeto (SHAW; GARLAN, 96).

Segundo a definição encontrada no site oficial[7] da ISO do padrão ISO/IEC 42010: 2007, uma arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução.

[6] Disponível em: <https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=513807>. Acessado em: 15 de abril de 2018.
[7] Disponível em: <https://www.iso.org/standard/45991.html>. Acessado em: 10 de agosto de 2018.

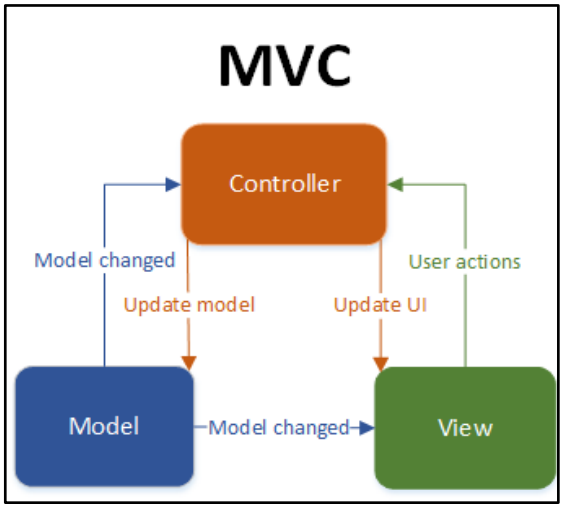
Um dos aspectos de que se faz necessário o uso de uma arquitetura é garantir um aumento na produtividade, pois com uma boa arquitetura possui maior capacidade de aumentar o desempenho, dando conta do aumento da escala e do volume de dados gerados na aplicação. Outro ponto importante da utilização de uma boa arquitetura é a garantia de escalabilidade, isso facilita o processo de implementação e evita o risco de algum tipo de mau funcionamento (PICOLI, 2018).

No projeto será adotada a arquitetura MVC *(Model-View-Controller*), que é uma arquitetura que auxilia no desenvolvimento de aplicações separando seus principais componentes, a manipulação e armazenamento dos dados, as funções que irão trabalhar com as entradas dos dados e a visualização do usuário. A arquitetura MVC específica onde cada tipo de lógica deve estar localizado na aplicação (SANTOS, I., et al., 2010).

2.1.1 Arquitetura MVC

A arquitetura que será adotada para o projeto será o padrão *Model-View-Controller* que separa a modelagem do banco de dados, a apresentação e as ações com base na entrada do usuário, em três classes separadas (STEVE, 2009). A camada *model* manipula os dados com comandos de acesso a base de dados e responde às instruções para alterar o estado.A camada *view* gerencia as informações que serão exibidas em tela. E por fim, a camada *controller* é responsável por controlar e interpretar as informações recebidas e manipulando qual *view* ou *model*, deverão ser usados.

Figura 1. Camadas utilizadas no MVC



Fonte: BELLAVER (2017)

As *views* devem gerenciar o espaço na tela e exibir formas de texto ou gráficas nessa camada. Os *controllers* devem garantir que seja interpretada a entrada de comandos. Essas duas camadas, junto com suas subclasses, fornecem uma variedade de comportamentos que seus aplicativos requerem poucas etapas para realizar uma entrada de comando e comportamento de saída interativa. Em contraste, a *model* não pode ser estilizada. Restrições no tipo de objetos permitidos para funcionar como a *model* limitam o intervalo útil de aplicativos dentro do paradigma MVC. Necessariamente, qualquer objeto pode ser um modelo (BURBECK, 2009, p.2).

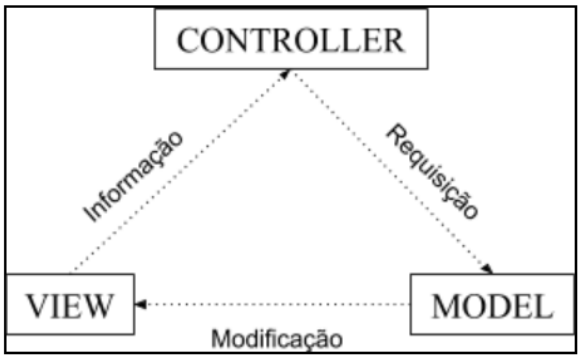
A *model* contém a comunicação com os dados armazenados que serão visualizados na *view*, podendo estar armazenado em um banco de dados. É somente na camada *model* que as operações de cadastro, edição, busca e exclusão, acontecem. O *model* representa a informação das *views* e as regras correspondente a manipulação das regras de negócio, ele mantém o estado persistente e fornece ao *controller* uma maneira de acessar as funcionalidades da aplicação, o *model* é a parte mais importante de toda a aplicação. O *model* apenas acessa a base de dados e deixa com que o *controller* distribua as rotas e transfira os dados para as *views* corretas (BAPTISTELLA, 2004).

A *controller* é a camada responsável por administrar todo o fluxo de passagem de dados, de comandos, e armazenada a maior parte da lógica de programação, trabalha com os dados

recebidos pela *view* e seleciona a operação que será utilizada da camada *model* (BASTOS, 2011). Sem a *controller* tornaria extremamente confuso o desenvolvimento, ela é a camada que intermedia a camada *model* e *view*, responsável por controlar e coordenar o envio de requisições feita entre essas duas camadas. O *controller* define o comportamento da aplicação, interpretando as requisições e seguindo fluxos, mapeando e gerenciando as ações dos usuários.

A *view* é a camada responsável por garantir que a apresentação seja o reflexo do estado do *model*, quando mudam os dados da classe *model*, a *view* deve-se atualizar-se, gerenciar e exibir formas de textos ou gráficos. Quando uma entrada de informação é recebida pela *view*, então é passado para o *controller* decidir para qual *model* encaminha a requisição, após isso, o *model* retorna para a *view*, e essa transforma os dados em uma tela para que o usuário consiga ver e interagir (BAPTISTELLA, 2004).

Figura 2. Interação entre as camadas MVC



Fonte: Elaborada pelo autor

O padrão MVC separa *views* e *models* es*tab*elecendo um protocolo de assinatura e notificação entre eles. Uma *view* deve assegurar que sua aparência reflita o estado do *model*. Sempre que os dados do *model* são alterados, ele notifica os modos de exibição que dependem dele. Em resposta disso, cada *view* tem a oportunidade de se atualizar. Esse comportamento permite que várias *views* sejam anexadas a um *model* para fornecer apresentações diferentes. Também é possível criar uma nova *view* e anexar a um *model* existente, sem precisar que ele seja reescrito. (DAVIS, 2004)

Outro recurso do MVC é permitir a alteração da maneira de como uma *view* responde à entrada do usuário, sem alterar sua apresentação em tela. Pode-se alterar a maneira como ela responde ao teclado, por exemplo, ou usar um menu, um pop-up em vez de teclas de comando. O MVC encapsula o procedimento de resposta em um objeto *controller*. Existe uma hierarquia de classes controladoras, facilitando a criação de um novo controlador como uma variação de um já existente (DAVIS, 2004).

A arquitetura que será seguida para o decorrer do projeto, será voltada para uma aplicação web. Conforme Avraham Leff e James T. Rayfield (2001), o padrão MVC orientado a aplicação web, consta com um problema: esses tipos de aplicações são internamente particionadas entre o cliente e o servidor. A *view*, é claro, é exibida no cliente; mas o *model* e o *controller* podem ser distribuídos de várias maneiras entre o cliente e servidor (LEFF; RAYFIELD, 2001). Para suprir esse problema, serão feitas adaptações na arquitetura MVC, essas alterações, refletem a natureza fundamental do estilo de arquitetura cliente-servidor subjacente à web. A plataforma de visualização será fisicamente separada dos outros componentes e, portanto, a camada de visualização ficará menos vinculada ao controlador e ao modelo na estrutura e na operação. Ao mesmo tempo, a camada *view* será responsável pela parte de fornecer uma interface de usuário mais responsiva. A responsabilidade adicional do controlador é fundamental devido à necessidade de gerenciar o fluxo de controle inerente à experiência do usuário (GROVE; OZKAN, 2011).

Segundo Grove e Ozkan (2011), os principais aspectos da utilização de um modelo MVC adaptado para a web que diferem do modelo MVC original, são:

a) Nenhuma dependência específica entre *views* e *models*: o *model* não notifica a *view* quando há alterações. Em vez disso, o *controller* determina o comportamento da *view*, dependendo do resultado do processamento da transação do *model*.
b) Nenhuma restrição do controlador de exibição: cada elemento de visualização pode ter vários controladores e cada controlador podem ter vários elementos de visualizações.
c) Padrão *front controller*: um único elemento controlador é responsável pelo roteamento de solicitações HTTP de entrada, com base na URL solicitada e nos dados de configuração.

d) O controlador especifica a dinâmica da visualização: O *controller* que decide qual *view* segue cada ação, com base na resposta da requisição. Isso equivale ao que é essencialmente uma dependência de *view* e *controller*. Os elementos do controlador são responsáveis pela validação de dados: A validação do parâmetro de transação é essencialmente uma função do modelo. A lógica de validação pode ser inserida no componente do modelo, mas a responsabilidade pela execução da função de validação ainda está no controlador.
e) O modelo como controlador e gerenciador: o modelo se encarrega de gerenciar dados, gerenciar lógicas de aplicativos e autenticações, porém não há uma definição para o modelo e muito menos para separá-lo do controlador.
f) As classes de modelo são instanciadas sob demanda: em vez de um componente de modelo persistente, conforme previsto no MVC original;são instanciados modelos conforme necessário para a manipulação de transações e encapsular as entidades de domínio. No entanto, o banco de dados ou a camada de persistência de dados do aplicativo pode ser um componente de modelo persistente.

2.2 FRONT-END

Na web, o front-end trabalha no lado do cliente, no navegador. Os links, as imagens, os botões e os textos são o resultado de uma mistura de dois códigos escritos em duas linguagens: oHyper Text Markup Language (HTML)[8] e o Cascading Style Sheets (CSS)[9]. O HTML organiza todo o conteúdo que há dentro de uma página e ajuda o navegador a saber como exibir essasinformações. O CSS é utilizado para especificar estilos visuais e ainda, é responsável por toda a parte estética (CHAN, 2016).

Essas duas linguagens, geralmente, se unem com outra linguagem capaz de tornar o site interativo: a linguagem JavaScript. Essa linguagem é permite criar diversas ações e interações com o usuário. Sendo assim, o HTML é o esqueleto enquanto o CSS é o que deixa com boa aparência, o JavaScript é o que dá vida ao corpo (CHAN, 2016).

Figura 3. Linguagens utilizadas no Front End

Fonte: PEREIRA (2018)

Além do trio mais conhecido do mercado front-end (HTML, CSS e JavaScript), essa área ainda conta com centenas de *frameworks*, bibliotecas e tecnologias auxiliares que ajudam no desenvolvimento do front-end das aplicações. A tecnologia que será utilizada para ajudar nesse processo de desenvolvimento, é um framework baseado em JavaScript capaz de gerar código nativo para iOS e Android.

[8] Disponível em: <https://www.w3.org/html/>. Acessado em: 10 de julho de 2018
[9] Disponível em: <https://www.w3.org/Style/CSS/Overview.en.html>. Acessado em: 10 de julho de 2018

2.2.1 React Native

React Native (RN) é um projeto desenvolvido pelos engenheiros do Facebook que consiste em uma série de ferramentas que juntas possibilitam a criação de aplicativos móveis nativos para android e iOS, utiliza as ferramentas de front-end mais modernas e o desenvolvimento é baseado em JavaScript (CABRAL, 2016).

Com RN é possível utilizar JavaScript; CSS Flexbox, capaz de deixar a tela visível em diversas resoluções e dispositivos diferentes (EIS, 2012); JSX, capaz de utilizar-se de *tags*HTML e JavaScript, em um mesmo arquivo (MILFONT, 2017); diversos pacotes do NPM dentro de muitas outras ferramentas. Ele também permite fazer debug na mesma *Integrated Development Environment* (IDE) utilizada para o desenvolvimento nativo com essas plataformas (CABRAL, 2016).

O RN utiliza chamadas nativa para renderizar esses elementos. Como o desenvolvimento é baseado em javascript, para realizar a comunicação entre o setor javascript e o setor nativo da aplicação, existe uma ponte que é responsável por esse processo (CÂMARA, 2018). Para o projeto não será necessário saber o funcionamento da ponte, uma vez que, ela já é integrada com o funcionamento do RN.

O React Native funciona a partir dessa ponte que permite a realização de chamadas nativas ao JS e chamadas do JS ao nativo. No android é incluído uma JS VM (JavaScriptCore), no iOS a JavaScriptCore já é uma integração do sistema. O RN é um código JS rodando em umamáquina virtual, controlado por uma interface de usuário nativa, o código gerado por javascript não é compilado ou convertido para a linguagem nativa do dispositivo. Tudo acontece através dessa ponte entre o JS e o ambiente nativo do dispositivo (MONTEIRO, 2017).

As principais vantagens da utilização desse framework são: conter uma experiência do usuário muito mais fluida, uma vez que, ele gera códigos nativos; os carregamentos e requisições são mais rápidos, pois, não necessita conter uma *webview* para intermediar esse processo; possuiuma melhor integração entre funções do dispositivo, como câmera, gps, giroscópio, etc.; possui maior segurança em relação a aplicativos *web mobile* e uma performance em geral relativamente superior. O desenvolvimento é extremamente mais rápido, possuindo um reaproveitamento de código gigantesco que não existiria se fosse desenvolvido em object-c para iOS e java para android (KUPKA, 2017).

O RN também conta com tipos de sintaxes fáceis de entender e de reutilizar, suas principais formas se baseiam em tags HTML e tags específicas para tipos de dispositivos. Ele também possui uma forma de descrever estilos *inline*, utilizando javascript, muito diferente dos demais *frameworks* que costumam utilizar estilos separadamente. A partir disso, existem abordagens que fazem com que não seja necessário estilizar cada um dos itens, assim, é possível criar um objeto de estilo que pode ser reutilizado em diversos elementos (KUPKA, 2017). Outra vantagem do RN é a customização de layouts que é feita completamente por meio deum modelo unidimensional (COLARES, 2018).

Uma funcionalidade interessante do RN, que aumenta a praticidade e a produtividade, é o *reload* automático após cada alteração. Com ele o programa é capaz de ficar rodando em desenvolvimento, e a cada mudança no código uma nova versão é injetada na aplicação, levando menos de 1 segundo para atualizar. Em outros frameworks nativos, com apenas uma modificação, a aplicação precisa ser recompilada inteiramente, levando muito mais tempo. Outra função que essa tecnologia possui é a possibilidade de depurar a aplicação via Google Chrome, como se fosse uma aplicação web padrão, facilitando o desenvolvimento. Outra característica interessante é a possibilidade de mesclar códigos Javascript com códigos nativos (objective-c ou java), caso seja necessário utilizar componentes prontos, ou otimizar alguns aspectos da aplicação (KUPKA, 2017).

Além de tudo, o React Native é contemplado com uma comunidade extremamente colaborativa, caso ocorra problemas com desenvolvimento, com plugins, incompatibilidade, etc, torna-se extremamente mais rápida e rápida a forma de encontrar soluções. Além disso, muitas vezes a comunidade disponibiliza componentes prontos para serem utilizados, e mesmo que não sejam encontrados, a criação de um componente nativo pode ser feita uma única vez e reutilizada posteriormente (OLIVEIRA, 2018).

Ao criar um aplicativo usando react native, é completamente necessário tratar o estado de maneira eficaz e segura. Especialmente, porque escrevendo em javascript, pode-se lançar erros em qualquer lugar da aplicação e que não dá a devida segurança das linguagens tipadas estaticamente. Essa é uma das principais razões pelas quais a comunidade javascript e react native em geral, aconselham a utilização de algum tipo de estrutura que seja capaz de gerenciar o estado, possibilitando a modelagem de aplicativos que contenham um estado complexo, o Redux é um deles e será descrito na próxima subseção.

2.2.2 Redux

Dan Abramov (autor do Redux) descreveu Redux enquanto trabalhava em sua palestra no React Europe chamada “Hot Reloading with Time Travel”. O seu objetivo era criar uma biblioteca de gerenciamento de estado usando uma API mínima, mas possuindo um comportamento completamente previsível. Segundo a documentação oficial[10], com o Redux é possível implementar logis, recarregamento automático, viagem temporal, aplicativos universais, gravações e replays, sem qualquer participação do desenvolvedor.

A documentação do redux descreve que se concentre a lógica de atualização de modelo em uma certa camada da aplicação. Em vez de permitir que o código do aplicativo sofra uma mutação direta dos dados, descrevendo cada mutação como um objeto simples chamado de “ação”[11]. Garantir que todas as alterações sejam descritas como uma ação, permite ter uma compreensão clara do que está acontecendo no aplicativo. Se algo mudou, sabe-se porquê mudou.

Para unir estados e ações, é declarada uma função chamada redutor. Esta função é capaz de tornar o estado e ação em argumentos e retorna estes argumentos no próximo estado do aplicativo. É possível começar com um único redutor e, à medida que o aplicativo vai ganhando volume, é preciso dividi-lo em redutores menores que gerenciam partes específicas da árvore de estados do Redux. Na documentação oficial, encontram-se diversos exemplos de utilizações dessa função[12].

O Redux tenta tornar as mutações de estado previsíveis, impondo certas restrições sobre como e quando as atualizações podem acontecer. Essas restrições são refletidas nos três princípios do Redux, conforme descrito na documentação oficial[13]:

a) Única fonte de verdade

O estado de todo o aplicativo é armazenado em uma árvore de objetos em um único armazenamento facilitando a criação de aplicativos universais, já que o estado do servidor pode ser iniciado e mantido no cliente sem nenhuma codificação extra. Uma única árvore de estado também facilita a depuração ou inspeção de aplicativos, permitindo também a persistência do estado do aplicativo em desenvolvimento, para um ciclo de desenvolvimento mais rápido.

[10] Disponível em: <https://redux.js.org/>. Acessado em: 24 de outubro de 2018.
[11] Disponível em: <https://redux.js.org/introduction/priorart>. Acessado em: 24 de outubro de 2018.
[12] Disponível em: <https://redux.js.org/introduction/coreconcepts>. Acessado em: 24 de outubro de 2018.
[13] Disponível em: <https://redux.js.org/introduction/threeprinciples>. Acessado em: 24 de outubro de 2018.

b) Estado é somente leitura

A única maneira de fazer alterações em um o estado é emitindo uma ação, um objeto que descreva o que foi alterado. Essa atitude garante que nem as exibições nem os retornos de chamada serão gravados diretamente no estado. Em vez disso, eles mostram a intenção de transformar o estado. Como as ações são apenas objetos simples, elas podem ser registradas, serializadas, armazenadas e reproduzidas para fins de depuração ou teste.

c) Alterações são feitas com funções puras

As alterações são feitas com funções puras. Para especificar como a árvore de estados é transformada por ações, é necessário a escrita de redutores puros.

Basicamente, o Redux ajuda no desenvolvimento de aplicativos para que eles se comportem de maneira consistente, executados em diferentes ambientes (cliente, servidor e nativo) e com fácil maneira de execução de testes. Além disso, proporciona uma ótima experiência de desenvolvedor, como edição de código em tempo real combinada com um depurador de viagem no tempo[14].

O projeto de desenvolvimento resultará em uma aplicação nativa para os dois tipos de plataforma, iOS e Android, portanto, para emular essa aplicação em um dispositivo físico, sem precisar de uma ferramenta que faça isso virtualmente, o aplicativo será criado utilizando o Expo. Ele foi criado exclusivamente para aplicações React Native e utilizando o expo para a criação do projeto se torna possível executá-lo em qualquer uma das duas plataformas sem precisar de um emulador.

2.2.3 Expo

Expo é uma ferramenta utilizada para criar-se aplicações React Native, que permite um fácil acesso a partes nativas dos dispositivos, sem a necessidade de instalar dependências no código fonte do projeto. O grande ponto da utilização do expo está em desenvolver aplicativos sem precisar instalar SDK do Android ou XCode para Mac, isso tudo se dá ao motivo de que o Expo possui um aplicativo instalável nas lojas das plataformas (Google Play e Apple Store) que contém todo o código nativo essencial, assim só é necessário mexer na parte JavaScript do React Native, possibilitando ter-se acesso ao aplicativo nas duas plataformas (FERNANDES, 2018).

[14] Disponível em: <https://redux.js.org/>. Acessado em 24 de outubro de 2018.

Os aplicativos criados utilizando o Expo, contam com o Expo SDK. O SDK é uma biblioteca nativa e em JS que fornece acesso à funcionalidade do sistema do dispositivo, tornando o projeto puro JavaScript, deixando-o mais portátil, pois pode ser executado em qualquer ambiente nativo que contenha o Expo SDK[15]. Quando um projeto é executado utilizando Expo, ele cria uma instância do projeto e disponibiliza localmente, para ter acesso a essa instância através da rede, deve-se utilizar o Expo Client. Com ele, é possível solicitar ao computador uma cópia local do projeto (através de localhost, LAN ou um túnel), baixar essa cópia e executá-la, com essa ferramenta, também pode-se depurar a aplicação, ter logs de dispositivos, elementos de inspeção, carregamento de módulos ativos e simultâneos a cada salvamento dos códigos fontes[16].

[15] Disponível em: <https://docs.expo.io/versions/latest/>. Acessado em: 27 de outubro de 2018.
[16] Disponível em: <https://docs.expo.io/versions/latest/introduction/project-lifecycle.html>. Acessado em: 27 de outubro de 2018

2.3 BACK-END

O back-end é o código que roda do lado do servidor, também conhecido como *server-side*. Ao contrário do *front-end*, o *back-end* é responsável por regras de negócio, *webservices* e, no nosso caso, *APIs*. Uma API é uma interface que permite com que dois sistemas, criados em tecnologias diferentes possam se comunicar através de uma linguagem em comum. Em outras palavras, é uma interface capaz de fazer com que diferentes softwares, aplicações, banco de dados e serviços possam se conectar e interagir através de um canal em comum, sem a necessidade de programações complexas (CHINAGLIA, 2018). O projeto será arquitetado tendo em vista o padrão MVC, com uma API implementada seguindo a arquitetura REST, sendo a mesma responsável pelo componente modelo e controlador da aplicação.

REST é uma arquitetura que os dados e funcionalidade são considerados recursos, e esses recursos são acessados usando *Identificadores Uniformes de Recursos* (URIs) (MATRAKAS, 2016). Os recursos são acionados usando um conjunto de operações simples e bem definidas. A arquitetura REST é baseada fundamentalmente na arquitetura cliente-servidor, e é projetada para usar um protocolo de comunicação sem estado (*stateless*). Na arquitetura REST, clientes e servidores trocam representações de recursos usando uma interface padronizada e protocolada. Com isso, os aplicativos baseados nesse modelo arquitetural são simples, leves e com alto desempenho (MENIYA, 2012).

Quando se possui uma API que aplica todos conceitos da arquitetura REST, então é dito que é uma API RESTful, pois o seguinte termo explica que a API foi feita inteiramente seguindo a arquitetura REST, tendo em vista que pode haver casos em que nem todos os conceitos são aplicados ou quando são, podem ser aplicados de maneira errônea (DIAS, 2016).

Existe um protocolo, que especifica o formato de dados para a troca de informações entre diferentes serviços, com um padrão para que haja interoperabilidade entre eles. Esse protocolo é chamado de SOAP, abreviação para *Simple Object Access Protocol* (DANTAS, 2007). Esse protocolo utiliza *Extensible Markup Language* (XML) para a transferência de objetos entre as aplicações e utiliza o protocolo HTTP para o transporte das informações (ROZLOG, 2013). A Tabela 2 ilustrada no trabalho de Meniya (MENIYA, 2012), mostra os resultados de *benchmarking* dos serviços web de concatenação de palavras, adição de números inteiros e números decimais. A Tabela também mostra o tamanho da mensagem em bytes para os serviços

web. O tamanho da mensagem do serviço web RESTful é extremamente inferior ao do serviço SOAP convencional. O tempo de resposta do RESTful é menos do que a metade do que o serviço SOAP levaria. A partir do resultado, mostra a maior vantagem de usar o serviço da web RESTful. O intervalo é muito grande entre os dois serviços web, o RESTful é largamente superior ao SOAP.

Tabela 1. Diferença de tempo entre RESTful e SOAP

Nome do serviço	Tamanho da mensagem (bytes)		Tempo (milliseconds)	
	SOAP	**REST**	**SOAP**	**REST**
Concatenação de palavras	148	19	257	126
Adição inteiros	268	27	478	247
Adição de decimais	363	43	789	376

Fonte: MENIYA (2012)

Em relação às vantagens de utilização do serviço RESTful, ao invés de SOAP, leva-se em consideração a facilidade de desenvolvimento, de aproveitamento de código e de um esforço de aprendizado pequeno, pois o modo como é arquitetado é extremamente simples. Ainda sobre o artigo de Meniya (MENIYA, 2012), também constam alguns dos principais objetivos de se utilizar uma API RESTful, que são:

a) Capaz de fornecer uma resposta melhor e um carregamento de dados do servidor mais rápidos devido ao suporte para armazenamento em *cache*.
b) Melhora a escalabilidade do servidor reduzindo a necessidade de manter o estado de comunicação.
c) Requer menos software do lado do cliente para ser escrito, um único navegador é capaz de acessar qualquer tipo aplicação e qualquer recurso.
d) Não requer um mecanismo de descoberta de recursos separada, devido ao uso de hiperlinks no conteúdo.
e) Compatibilidade com documentos HTML, mesmo com diferentes versões.
f) Capacidade de adicionar suporte para recursos e novos conteúdos sem deixar cair ou reduzir o suporte para tipos de conteúdo mais antigos.

Por ser um protocolo ainda muito utilizado, ainda existem serviços que o ideal seria a utilização de SOAP, e também pode-se utilizar os dois ao mesmo tempo, porém nosso trabalho não necessita e não há motivos para a sua utilização.

O tipo de serviço RESTful é recomendado para projetos que necessitem de ações não muito complexas, esse serviço expõe recursos por meio de URIs e esses recursos usam os quatro principais métodos HTTP para chamar ações CRUD (Create, Read, Update e Delete) (MATRAKAS, 2016). A *tab*ela abaixo mostra o mapeamento de métodos HTTP para as ações CRUD.

Tabela 2. Métodos HTTP e suas operações CRUD

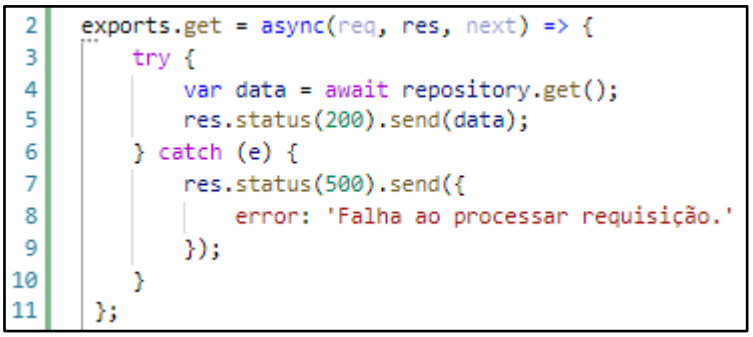
Métodos	**Operações**
GET	Buscar
POST	Criar
PUT	Atualizar
DELETE	Excluir

Fonte: MENIYA (2012)

Exemplos de utilizações de métodos HTTP na linguagem JavaScript:

a) Método GET: o método GET, é responsável por buscar os arquivos no repositório, como pode ser visto na Figura 4, linha 4 do código listado abaixo. Nesse exemplo não é utilizado parâmetros de pesquisa, portanto o método irá retornar todos os dados do documento sem restrições. Caso houver problemas na busca das informações, esse método irá retornar uma mensagem informativa, conforme mostrado na linha 8.

Figura 4. Script do método GET

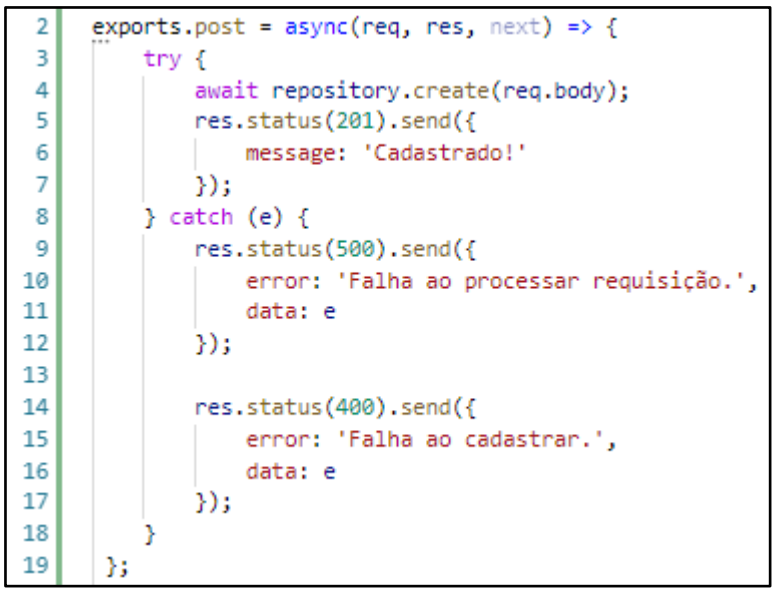


```
2   exports.get = async(req, res, next) => {
3       try {
4           var data = await repository.get();
5           res.status(200).send(data);
6       } catch (e) {
7           res.status(500).send({
8               error: 'Falha ao processar requisição.'
9           });
10      }
11  };
```

Fonte: Elaborada pelo autor

b) Método POST: o método POST, é encarregado por enviar dados para o repositório. Conforme a linha 4 da Figura 5, o método é utilizado para criar um novo registro, porém poderia ser utilizado para autenticação também. Na linha 6, o método retornará uma mensagem caso o cadastro tenha sido feito com sucesso e, na linha 10 e 15 se houveram problemas nesse processo.

Figura 5. Script do método POST

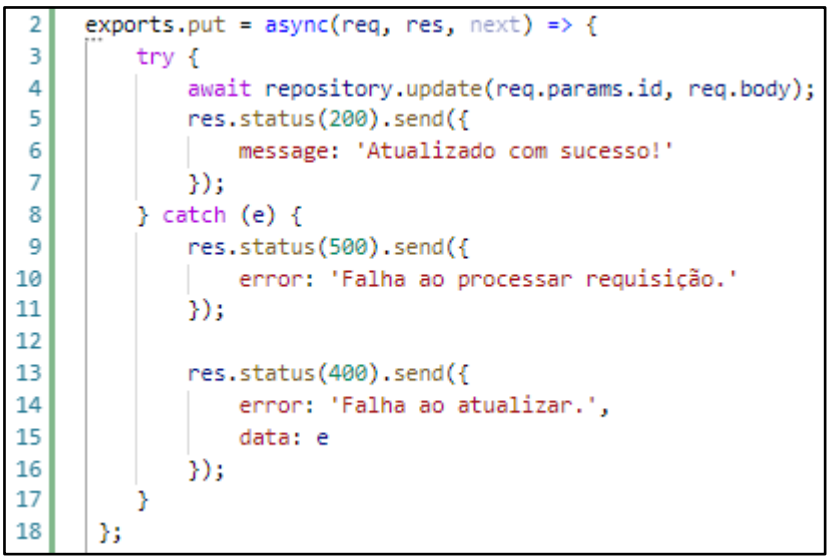


```
2   exports.post = async(req, res, next) => {
3       try {
4           await repository.create(req.body);
5           res.status(201).send({
6               message: 'Cadastrado!'
7           });
8       } catch (e) {
9           res.status(500).send({
10              error: 'Falha ao processar requisição.',
11              data: e
12          });
13
14          res.status(400).send({
15              error: 'Falha ao cadastrar.',
16              data: e
17          });
18      }
19  };
```

Fonte: Elaborada pelo autor

c) Método PUT: o método PUT, é responsável por executar a alteração de um registro, neste exemplo, mostrado na linha 4 da Figura 6 é utilizado dois parâmetros que são passados através da variável *req*, esses parâmetros geralmente são o id do registro que se queira editar e o *token* gerado a partir do *login*, porém não há restrições de parametrizações. Na linha 6, o código retorna uma mensagem informativa caso a alteração tenha ocorrido com sucesso e nas linhas 10 e 14, mensagem que informam se houveram problemas durante esse processo.

Figura 6. Script do método PUT

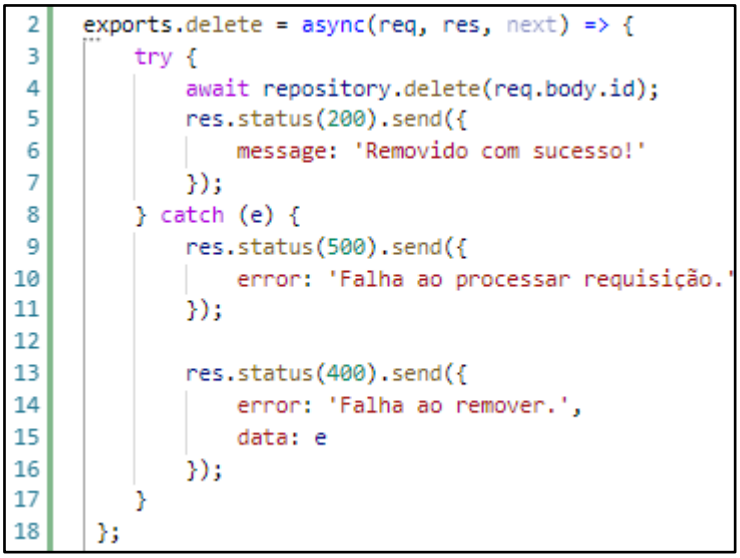


```
2   exports.put = async(req, res, next) => {
3       try {
4           await repository.update(req.params.id, req.body);
5           res.status(200).send({
6               message: 'Atualizado com sucesso!'
7           });
8       } catch (e) {
9           res.status(500).send({
10              error: 'Falha ao processar requisição.'
11          });
12
13          res.status(400).send({
14              error: 'Falha ao atualizar.',
15              data: e
16          });
17      }
18  };
```

Fonte: Elaborada pelo autor

d) Método DELETE: o método DELETE, é o responsável por executar a deleção dos registros. Na linha 4 da Figura 7 foi-se utilizado como parametrização apenas o id do registro, nesses casos de exclusões, não são necessárias muitas parametrizações. Geralmente se utiliza apenas o id e o *token* gerado no *login*. Se o registro for excluído com sucesso, na linha 6 mostra a mensagem que será mostrada, nas linhas 10 e 14 são retornadas mensagens informativas em caso de falhas no processo de exclusão.

Figura 7. Script do método DELETE



```
exports.delete = async(req, res, next) => {
    try {
        await repository.delete(req.body.id);
        res.status(200).send({
            message: 'Removido com sucesso!'
        });
    } catch (e) {
        res.status(500).send({
            error: 'Falha ao processar requisição.'
        });

        res.status(400).send({
            error: 'Falha ao remover.',
            data: e
        });
    }
};
```

Fonte: Elaborada pelo autor

Por ser uma área muito abrangente, existem diversas linguagens, *frameworks* e tecnologias para desenvolver o lado do servidor, desenvolver uma API e serem capazes de executar esses 4 métodos. Existem as tecnologias de *back-end* voltadas para web, e outras voltadas para soluções desktop. Cada uma delas tem suas vantagens e desvantagens perante ao seu desenvolvimento (VIANA, 2017).

Para o projeto será utilizado uma tecnologia voltada para a web, baseada em javascript. Com essa tecnologia será possível montar todas as rotas e *endpoints* da API, fazer a transição e manipulação dos dados no back-end, fazer todas as operações CRUD, e em forma de *response*, retornar os dados para cada *request* feita pelo front-end da aplicação.

2.3.1 Node.js

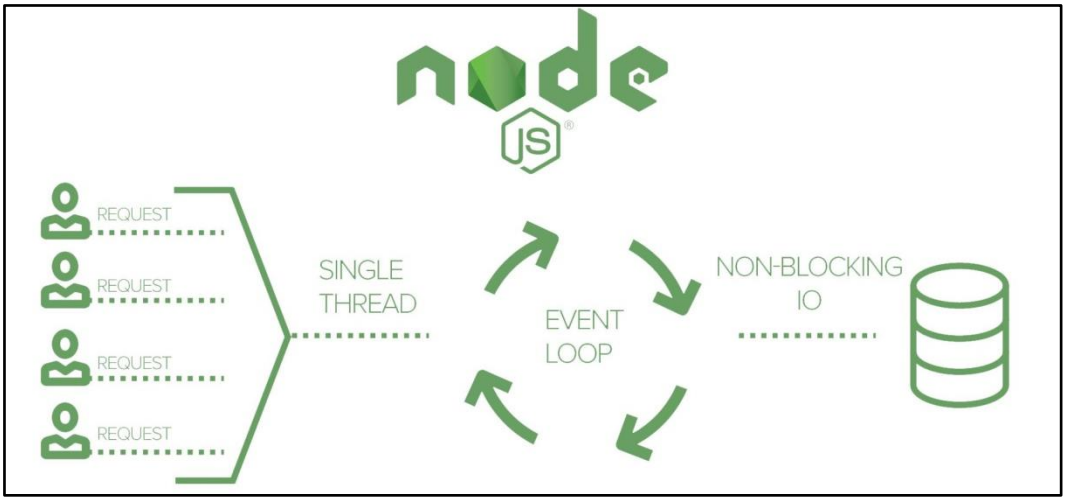
Node.js é um *runtime* orientado a eventos assíncronos desenvolvido sobre o motor JavaScript V8 do Google Chrome, o Node foi projetado para criar aplicativos de rede escalonáveis[17]. Além disso, ele usa uma arquitetura voltada a eventos não bloqueáveis. As requisições são interpretadas dentro de um *loop* de eventos em uma única thread e, isso é feito de

[17] Disponível em: <https://nodejs.org/en/about/>. Acessado em: 19 de agosto de 2018.

forma assíncrona não bloqueável, o que torna a aplicação mais rápida e eficiente, principalmente quando se tem um número extenso de requisições (SANTOS, G., 2016).

O *loop* de eventos é contemplado com diversas requisições, por exemplo, buscar um dado no banco de dados, ler arquivos, etc. Após ele executar uma das ações, não se espera uma resposta de retorno, ele vai para a execução da próxima requisição. Quando uma das requisições que estão sendo executadas é retornada, um evento e um *callback* é disparado para a requisição, no caso, uma função que deve ser executada como resultado da resposta é posta em uma fila para que seja executada (SANTOS, G., 2016). A Figura 8 representa o *loop* de eventos que é executado pelo NodeJS.

Figura 8. Evento de *loop* do NodeJS



Fonte: HELLER (2017)

Node.js é direcionado a eventos e o processamento de requisições de entrada e saída não são bloqueantes, garantindo es*tab*ilidade e pouco consumo de recursos do sistema. O desenvolvimento da tecnologia teve como objetivo "fornecer uma maneira fácil para construir aplicações escaláveis", conforme o site oficial (GOSS, 2017).

Em outras linguagens como Java e PHP, cada nova conexão cria um nova *thread*, adicionando 2 MB de memória com ela. Em um servidor que contenha 8 GB de memória RAM o número máximo de conexões são de 4.000 usuários simultâneos. Quando o número de conexões aumenta, para que seja possível suportar mais usuários, deve-se aumentar a quantidade de

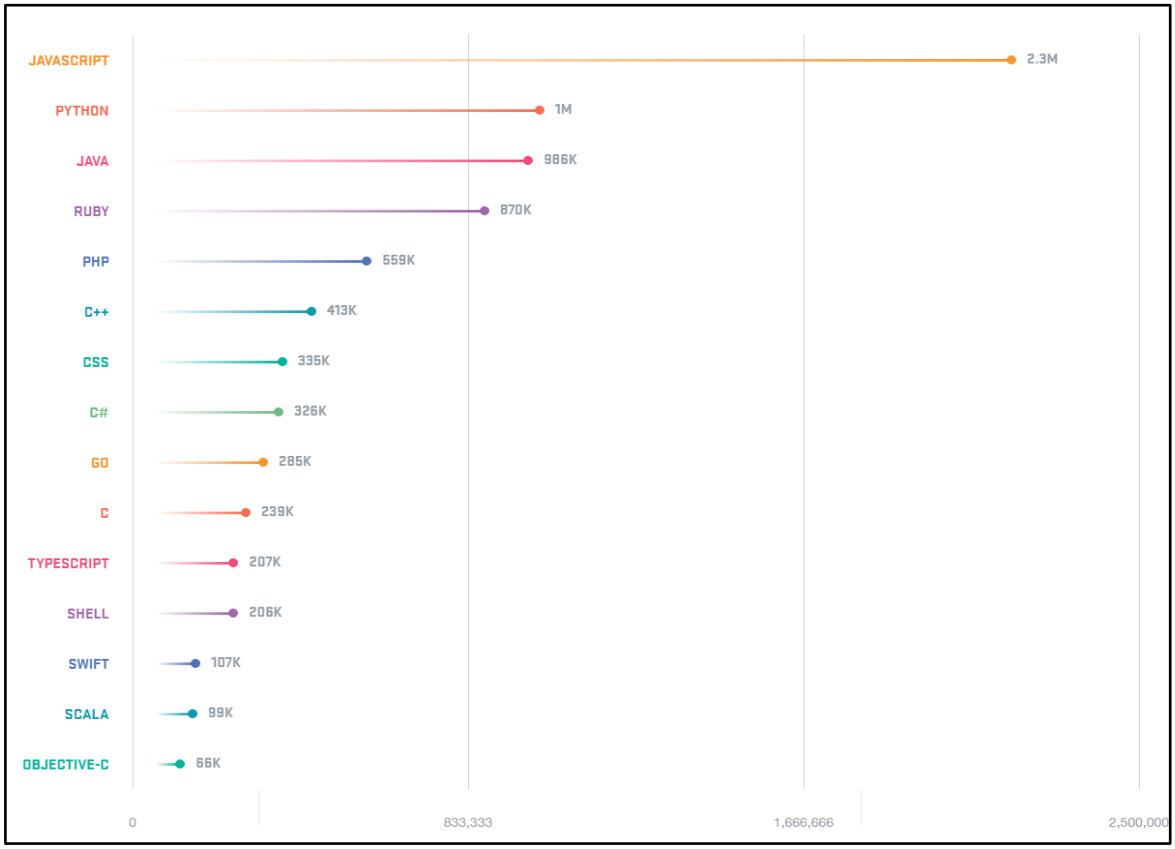
servidores, aumentando o custo. Somado a esse custo vem problemas técnicos que incluem, por exemplo, um usuário poder usar diferentes servidores para cada uma das requisições feitas por ele, com isso, cada recurso é compartilhado com todos os servidores[18]. O gargalo que a arquitetura da aplicação pode suportar é referente a quantidade de conexões simultâneas que o servidor consegue manipular.

O NodeJS consegue contornar esse problema mudando a maneira de como é tratada a requisição no servidor. Ao invés de criar uma *thread* a cada nova conexão e alocar memória junto a ela, cada nova conexão dispara um evento que é executado dentro de um conjunto de processos do NodeJS. Não existem bloqueios diretos para chamadas de entrada e saída, assim impossibilitando dar *deadlocks* no sistema. Com um servidor NodeJS rodando é capaz de suportar dezenas de milhares de conexões simultâneas[19].

Outra vantagem interessante de se utilizar NodeJS é o fato de ser baseado em javascript, podendo utilizar essa linguagem no *server-side*, diminuindo a curva de aprendizagem e o tempo de desenvolvimento, pois, será a mesma linguagem utilizada no *client-side*[20]. Além disso, com o NodeJS é possível executar aplicaçõesem qualquer plataforma, por exemplo: pode-se escrever todo o código no Mac OS, e em outros ambientes de produção executá-lo em ambiente Windows (FELIX, 2016).

[18] Disponível em: <http://nodebr.com/o-que-e-node-js/>. Acessado em 27 de setembro de 2018.
[19] Idem.
[20] Disponível em: <https://udgwebdev.com/6-motivos-para-usar-nodejs/>. Acessado em: 05 de outubro de 2018.

Figura 9. Linguagens mais populares no github 2017



Fonte: GitHub Octoverse 2017[21]

A Figura 9 demonstra o quão a linguagem JavaScript é popular no ambiente de repositórios GitHub. Outro ponto interessante é que, o node já oferece muitos pacotes, através do seu gerenciador de pacotes NPM (*Node Package Manager*). Com ele é capaz ter-se um controle de pacotes do NodeJS e de terceiros. Junto com isso, o NodeJS conta com uma comunidade grande e ativa, junto a ela se pode encontrar diversos tutoriais, vídeos explicativos, componentes e exemplos prontos. A comunidade ativa é uma das principais vantagens, pois através dela é possível encontrar mais de 70.000 módulos nos sites oficiais do NPM[22].

Com a utilização do NodeJS para a criação de uma API, é necessário que haja uma boa estrutura de roteamento de requisições, assim não havendo falhas e nem falta de respostas para cada requisição. No projeto será utilizado um framework capaz de auxiliar em todo esse processo

[21] Disponível em: <https://octoverse.github.com/2017/>. Acessado em 17 de setembro de 2018.
[22] Disponível em: <https://www.npmjs.com/>. Acessado em: 17 de setembro de 2018.

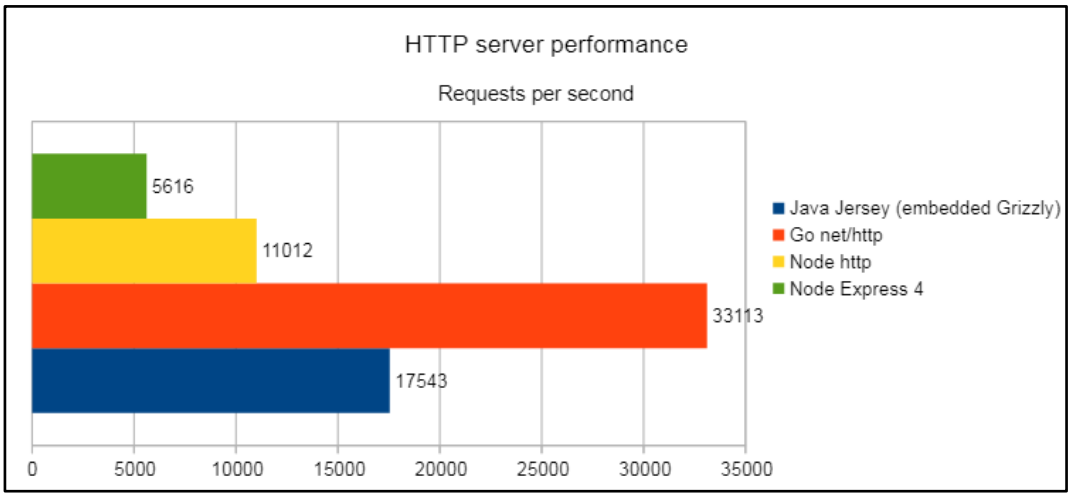
através de *middlewares*, ele conta com uma grande massa de métodos e funções capazes de facilitar esse gerenciamento, ele também é baseado em JavaScript, e foi feito exclusivamente para atender aplicações NodeJS. O *framework* em questão se chama *Express* e será descrito a seguir.

2.3.2 Express

Express é um *framework* que possui uma camada de recursos extras para aplicativos e aplicações web. Ele fornece uma série de métodos e utilitários para o roteamento HTTP e um *middleware* para facilitar na hora do roteamento das requisições[23]. Esses *middlewares* são pacotes que podem ser utilizados para resolver diferentes problemas no desenvolvimento da aplicação web. Há pacotes capazes de trabalhar com cookies, sessões, *login* de usuários, parâmetros de URL, dados em requisições POST, cabeçalho de segurança e entre tantos outros. No site oficial, é possível encontrar diversos pacotes mantidos pela equipe Express, juntamente com muitos outros pacotes subjacentes[24]. O roteamento é como um aplicativo responde a uma requisição do cliente/usuário, que é um caminho e um método de solicitação HTTP (GET, POST, etc.), para cada rota, pode-se ter uma ou mais funções que manipulam os dados após a rota ser correspondida[25].

[23] Disponível em: <https://expressjs.com/>. Acessado em: 17 de setembro de 2018.
[24] Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introdu%C3%A7%C3%A3o>. Acessado em: 18 de outubro de 2018.
[25] Disponível em: <https://devdocs.io/express/>. Acessado em: 18 de setembro de 2018.

Figura 10. Performance se servidores HTTP



Fonte: MÜTSCH (2016)

O Express é um dos frameworks JavaScript mais populares que fornece diversos mecanismos para gerenciar requisições e rotas URLs, para gerenciar respostas inserindo-as diretamente no *model* e renderizando a *view*, para definir configurações comuns em aplicações web, e a porta a ser usada na conexão. A principal funcionalidade do Express é a capacidade de inserir *middlewares* em qualquer ponto da requisição, o que deixa possível interceptar, processar ou pré-processar e tratar a mesma [26].

Para fazer os testes de requisições e funcionalidades utilizando-se o *Postman*, uma ferramenta que é capaz de simular requisições em formato JSON para aplicações que utilizam NodeJS (RODRIGUES, 2010). Com o *Postman* não é necessário escrever uma linha de código para conseguir verificar se a API está com o comportamento adequado (SANTIAGO, 2017). A partir dessa ferramenta, criou-se um volume de dados e diversos testes para as ações CRUD, todos os testes foram executados correspondentemente aos parâmetros informados.

[26] Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introdu%C3%A7%C3%A3o>. Acessado em: 18 de outubro de 2018

2.4 BANCO DE DADOS

BD (Banco de dados) é um local capaz de armazenar dados que posteriormente sejam consumidos por algum programa e por um ou diferentes usuários. Portanto, os dados são acoplados a uma espécie de rede, capaz de reunir as informações, assim surgindo o nome banco (PILLOU, 2017).

Um BD permite com que sejam inseridos, consultados ou alterados dados disponíveis, um banco de dados pode ser local, utilizando um dispositivo por usuário, ou distribuído, podendo armazenar dados através de dispositivos remotos e disponibilizados por uma rede. A maior vantagem da utilização de um banco de dados consiste na possibilidade de poder acessar os dados por diversos usuários simultaneamente (PILLOU, 2017).

Existem diferentes tipos de bancos de dados relacionais, porém, para oprojeto será utilizado um modelo de banco de dados não relacional, baseado em documentos. Um banco de dados orientados a documentos possui como característica manter todas as informações importantes em um documento, não possuir esquemas, possuir *Universally Unique IDentifier* (UUID), possibilitar a consulta de documentos através de métodos avançados de agrupamento e filtragem (MapReduce) e também permitir redundância e inconsistência (MEDEIROS, 2014). O banco que será utilizado para seguir esse modelo será o MongoDB. O MongoDB é o banco não relacional mais utilizado no ambiente de desenvolvimento, além de ser *open-source* e baseado em C++ o que o torna portátil para diferentes sistemas operacionais (DUARTE, 2017).

2.4.1 MongoDB

Segundo o site DB-Engines[27], o MongoDB é o 5º banco de dados mais utilizado no mundo e para o seu modelo, orientado a documentos e multiplataforma, é o 1º. Esse banco de dados é escrito em C++ e é *open-source*, é um modelo de banco de dados que armazena dados em documentos JSON com um esquema dinâmico (SOAREZ, 2016), ou seja, diferente de bancos relacionais, ele não possui uma restrição e necessidade de possuir *tab*elas e colunas criadas previamente, permitindo que o documento represente todas as informações necessárias(DAKAR, 2017).

[27] Disponível em: <https://db-engines.com/en/ranking>. Acessado em: 17 de agosto de 2018

Todos os documentos são agrupados em *collections*, um conjunto de *collections* forma o banco de dados (BOAGLIO, 2015). O MongoDB permite que esse banco de dados seja replicado para outros servidores, assim deixando em cada servidor uma cópia da*database*. Ele também conta com um recurso mais avançado que é chamado de *sharding*, que se usa para dividir os dados de uma *collection* entre mais de um servidor. Para *collections* grandes, esse recurso é usado para não sobrecarregar um único servidor (DAKAR, 2017).

O ponto positivo de se utilizar o MongoDB é a flexibilidade, com a estrutura orientada a documentos, que permite gravar dados da melhor forma para a aplicação, diferente de um banco relacional, onde há casos em que devesse mudar a estrutura da aplicação para se adequar ao banco de dados (DAKAR, 2017). Outra característica do mongoDB é a forma prática e simples para alterar a estrutura de um documentos, assim, pode-se mudar documentos simplesmente adicionando novos campos ou excluindo existentes. Com essa capacidade significativa, a representação de relações hierárquicas, para armazenamento de matrizes, e estruturas mais complexas torna-se extremamente simples em relação ao modelo relacional (SOAREZ, 2016).

Para ter acesso aos dados do banco a partir da aplicação, será utilizado o Mongoose [28]do MongoDB. O Mongoose é capaz de modelar os dados baseando-se em esquemas. Com ele será possível criar consultas, criar validações, converter tipos. Essa biblioteca fornece um mapeamento de objetos para classes *models* (NODEJS…, 2016). Portanto, o Mongoose traduz os dados do banco de dados para objetos JavaScript, assim consegue-se utilizá-los na aplicação, além disso o mongoose é capaz de suportar um ambiente assíncrono[29]. Um ambiente assíncrono é capaz de enviar diversas mensagens, sem ser necessária uma resposta imediata, caso haja uma resposta não é necessária a visualização imediata da mesma, isso torna-se uma comunicação assíncrona (FILHO, 2015).

Para gerenciar os dados, será utilizado um serviço capaz de oferecer uma automatização na hora de buscar, inserir e alterar os dados. Essa instância de banco de dados pode ser utilizada rapidamente necessitar possuir um conhecimento avançado em sistemas de gerenciamento de banco de dados. Esta ferramenta é feita exclusivamente para o banco de dados MongoDB e possui uma ótima conectividade. Essa ferramenta de serviço é online e gratuita, chamada de mLab.

[28] Disponível em: <https://mongoosejs.com/>. Acessado em 23 de setembro de 2018.
[29] Disponível em: <https://www.npmjs.com/package/mongoose>. Acessado em 27 de setembro de 2018.

2.5.2 mLab

A ferramenta mLab é um *Databaseas a Service* (DaaS) em nuvem, totalmente gerenciado, que oferece um dimensionamento automatizado de bancos de dados MongoDB, um backup e recuperação, um monitoramento 24 horas por dia e 7 dias por semana, ferramentas de gerenciamento baseadas na web e um suporte especializado[30]. Para uma aplicação que utiliza MongoDB, o mLab fornece um ótimo serviço de conectividade ao banco de dados, já que é apenas necessária a inserção de uma string e de qualquer lugar, já que possui serviço online.

O conceito de DaaS foi criado com o intuito de prover uma instância de banco de dados na nuvem, configurada para ser utilizada rapidamente, ou seja, com mLab não se torna necessário algum tipo de conhecimento avançado sobre Serviços de Gerência de Banco de Dados (SGDB) para iniciar a utilização dos serviços, ao contrário de muitas ferramentas de gerenciamento que é necessário uma longa etapa de configurações e instalações. Após criar uma conta, o serviço será automaticamente configurado, apenas será necessário informar o nome, senha e privilégios (na maioria dos casos) do banco de dados (HOROCHOVEC, 2016).

[30] Disponível em: <https://mlab.com/company/>. Acessado em: 27 de setembro de 2018.

3. METODOLOGIA

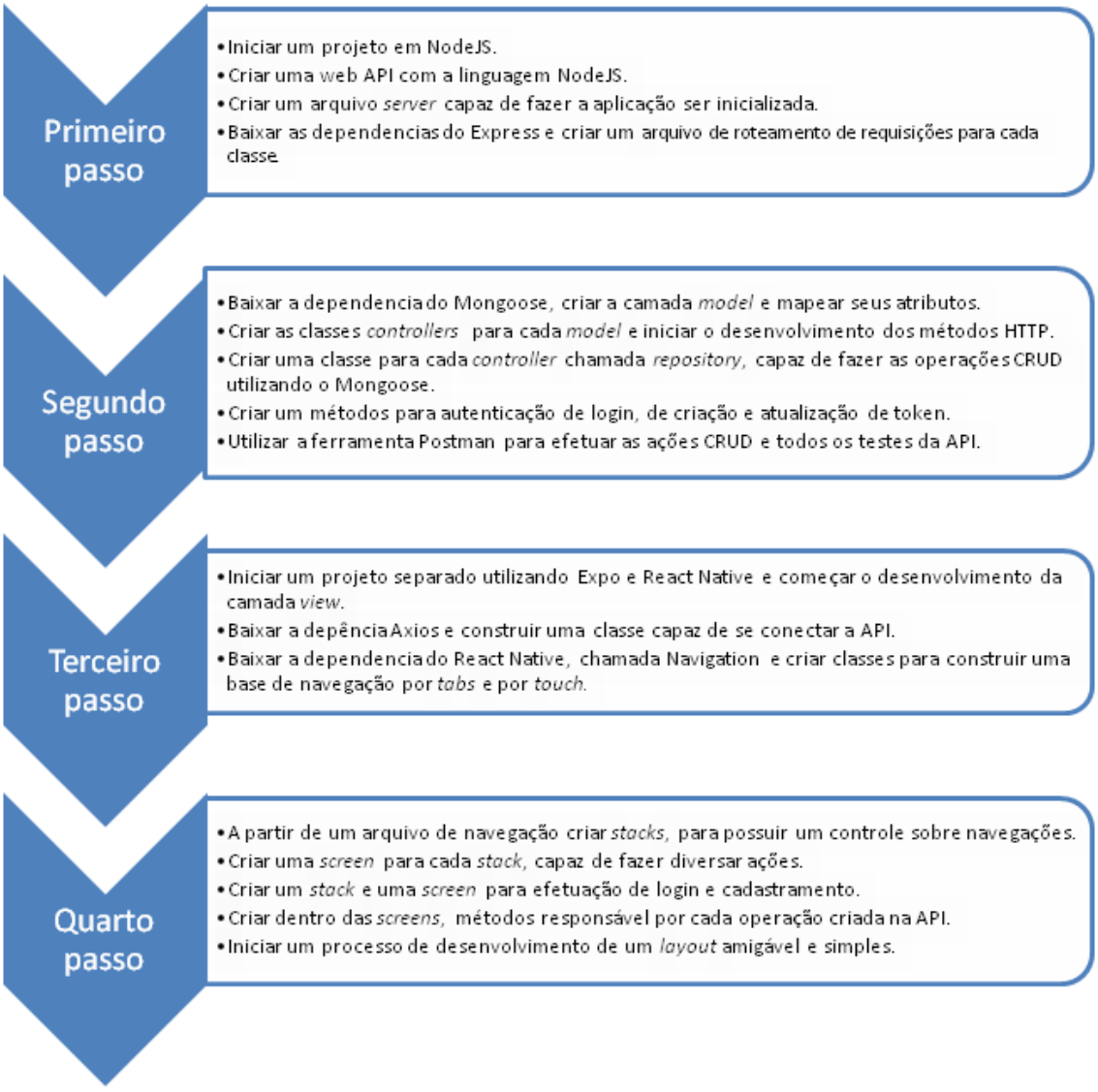
O presente trabalho é caracterizado por iniciar um desenvolvimento de um software multiplataforma para dispositivos móveis, utilizando uma série de ferramentas e métodos que auxiliarão nesse processo. Também é uma característica, a obtenção de conhecimentos necessários sobre as ferramentas em tempo de criação desse projeto, procurando a melhor forma de compreender, estruturar, manipular e desenvolver uma web API capaz de disponibilizar todos os dados necessários para o desenvolvimento do aplicativo. O contexto desse projeto de desenvolvimento baseia-se na construção de uma web API utilizando-se da linguagem JavaScript e o início da criação de um aplicativo de busca e disponibilização de vagas de empregos não-fixos.

A fim de alcançar os objetivos aqui propostos foi criada uma web API que contém todos os métodos e funcionalidades para as ações CRUD, contendo todas as rotas e mapeamentos necessários para que o aplicativo possa efetuar requisições e receber as respostas adequadas. Esses métodos e funcionalidades, juntamente com a comunicação entre o banco de dados e a aplicação foram feitas as requisições em formato JSON, utilizando a ferramenta *Postman*.

Iniciou-se o desenvolvimento de um aplicativo localmente utilizando a ferramenta Expo, com ele o aplicativo foi emulado para um dispositivo móvel que utiliza o sistema operacional iOS. Sempre quando houver alterações no processo de desenvolvimento do aplicativo, o expo gerará um novo *build* para que se tenha sempre a versão mais atualizada.

O aplicativo e a API foram desenvolvidos e testados localmente, após finalizados, serão testados por mais pessoas em mais locais, afim de agregar maior fluidez ao funcionamento. Para a elaboração do projeto, as etapas foram realizadas seguindo-se um fluxo de desenvolvimento, esse fluxo foi divido em 4 etapas de desenvolvimento. O fluxograma da imagem 11 ilustra os passos seguidos para a construção do projeto e a as principais etapas de construção.

Figura 11. Fluxograma de passos e atividades



Fonte: Elaborada pelo autor

Os arquivos de configurações principais de ambos os projetos, foram criados automaticamente na sua inicialização. O projeto foi inteiramente dividido em duas partes, uma delas o desenvolvimento da *web api* e em outro momento o início de um desenvolvimento de um aplicativo capaz de fazer as requisições.

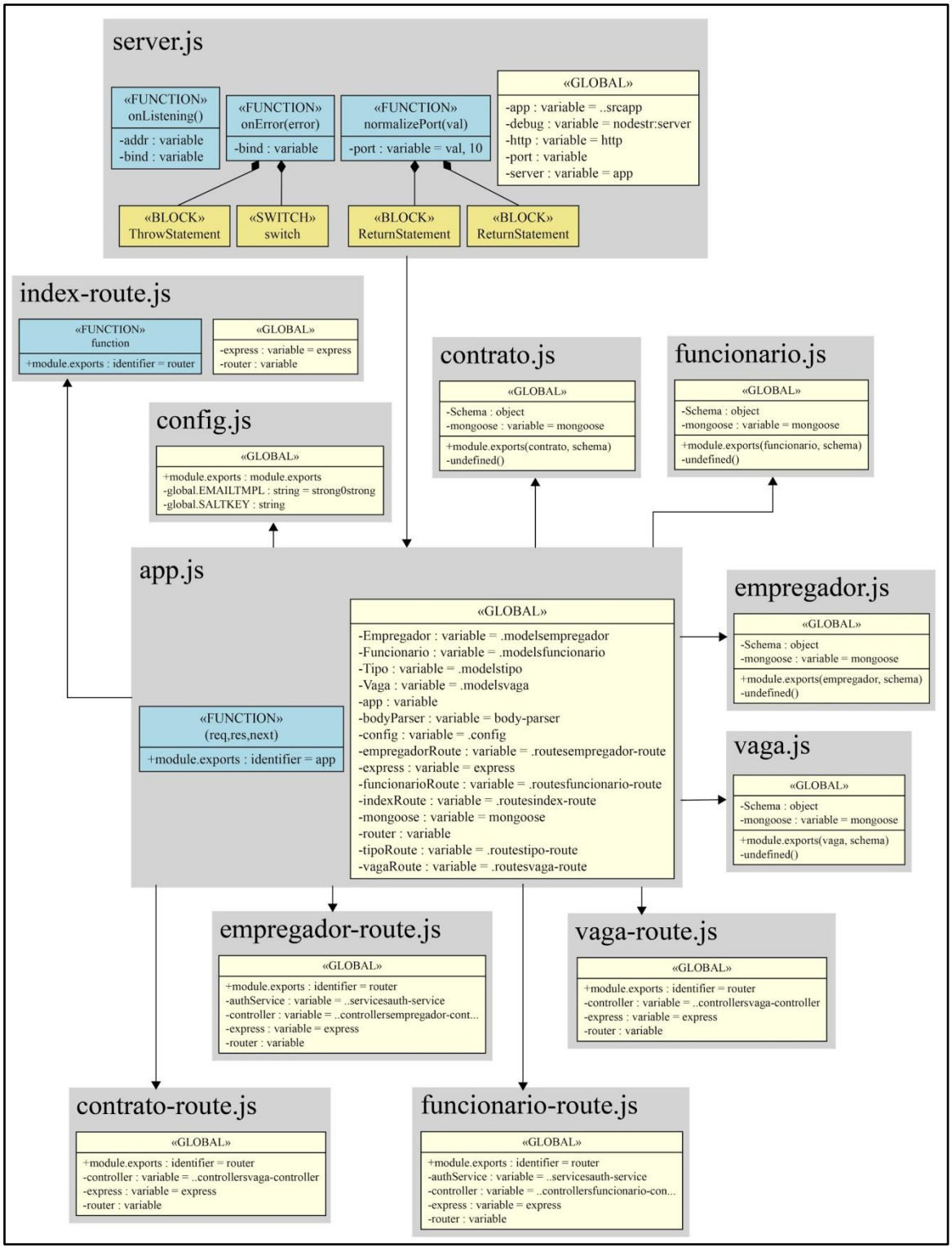
4. DESENVOLVIMENTO

Após ter estudado as melhores tecnologias para a criação desse projeto, foram criados dois projetos localmente, utilizando as tecnologias e melhores formas relatadas neste documento. O processo de desenvolvimento desses projetos, foi dividido em duas etapas principais: a primeira etapa é a construção de uma web API, possuindo todas as classes e funções necessárias para a construção da camada *model* e *controller*. Ademais, esta camada inicial irá possuir todos os métodos para as ações CRUD e tendo um volume de dados criada com a utilização da ferramenta *Postman*. A segunda etapa se dará ao início do desenvolvimento de um aplicativo nativo, capaz de executar algumas chamadas a web API. As partes foram chamadas de *back-end* e *front-end*, respectivamente.

4.1 BACK-END

O back-end é baseado em uma web API que é a responsável pelas duas camadas da arquitetura MVC, a *model* e a *controller*. Com essas duas camadas sendo as principais da API, foram desenvolvidas classes auxiliares capazes de efetuar o roteamento de requisições, controle de autenticação de *login*, criação e alteração de web *token*s e criptografia de senhas. O back-end foi construído de maneira simples e possuindo uma fácil manutenção e refatoração. Levando em consideração que o desenvolvimento da web API foi feito em javascript orientado a eventos e documentos, dividiou-se um diagrama de classes em pequenos módulos. A Figura 12 é um diagrama de um módulo de inicialização do sistema, contendo as classes que são instanciadas e os métodos de exportações dessas classes.

Figura 12. Diagrama do módulo de inicialização



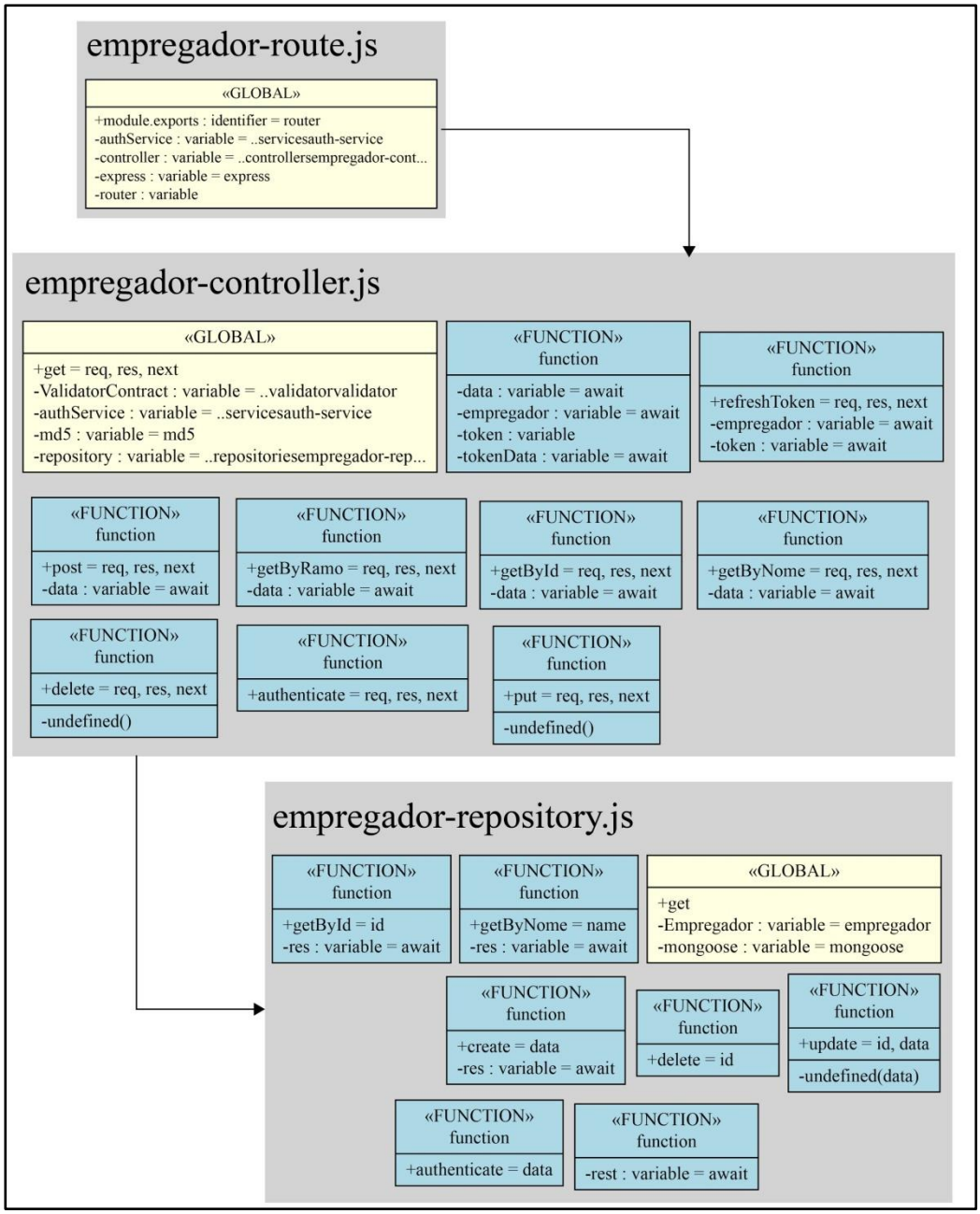
Fonte: Elaborada pelo autor

Conforme mostrado na imagem, primeiramente foi desenvolvida uma classe de inicialização da aplicação, contendo funções úteis para que isso ocorra, após isso, é chamada a classe responsável pela instanciação de classes que serão usadas na aplicação. São 8 classes que serão chamadas, 4 classes *models*, nomeadas conforme serão espelhadas nos documentos do banco de dados, e outras 4 classes de roteamento das requisições chamadas *routes*.

A partir de cada classe *route*, é criada uma classe *controller*. Na classe *controller*, foram desenvolvidos métodos capazes de executarem as ações necessárias para contemplar os métodos HTTP de cada *route*. No momento em que os métodos das classes *controller* estão sendo executados, esses chamam uma classe *repository*, onde foram implementados todos os métodos que fazem acesso a base de dados. Essa classe foi incrementada para diminuir o acoplamento de informações na classe *controller*. A classe *controller* fica mais organizada e com menos tarefas, assim a classe *repository* fica com o critério de executar todas as ações CRUD e a classe *controller* apenas efetua as manipulações de dados. As figuras 13, 14, 15 e 16 são as quatro classes *routes* com seus respectivos *controllers* e *repositories.*

A Figura 13 demonstra o diagrama de classes referente aos arquivos de rotas *empregador-route.js*, com as ligações das suas classes subjacentes e os métodos correspondentes.

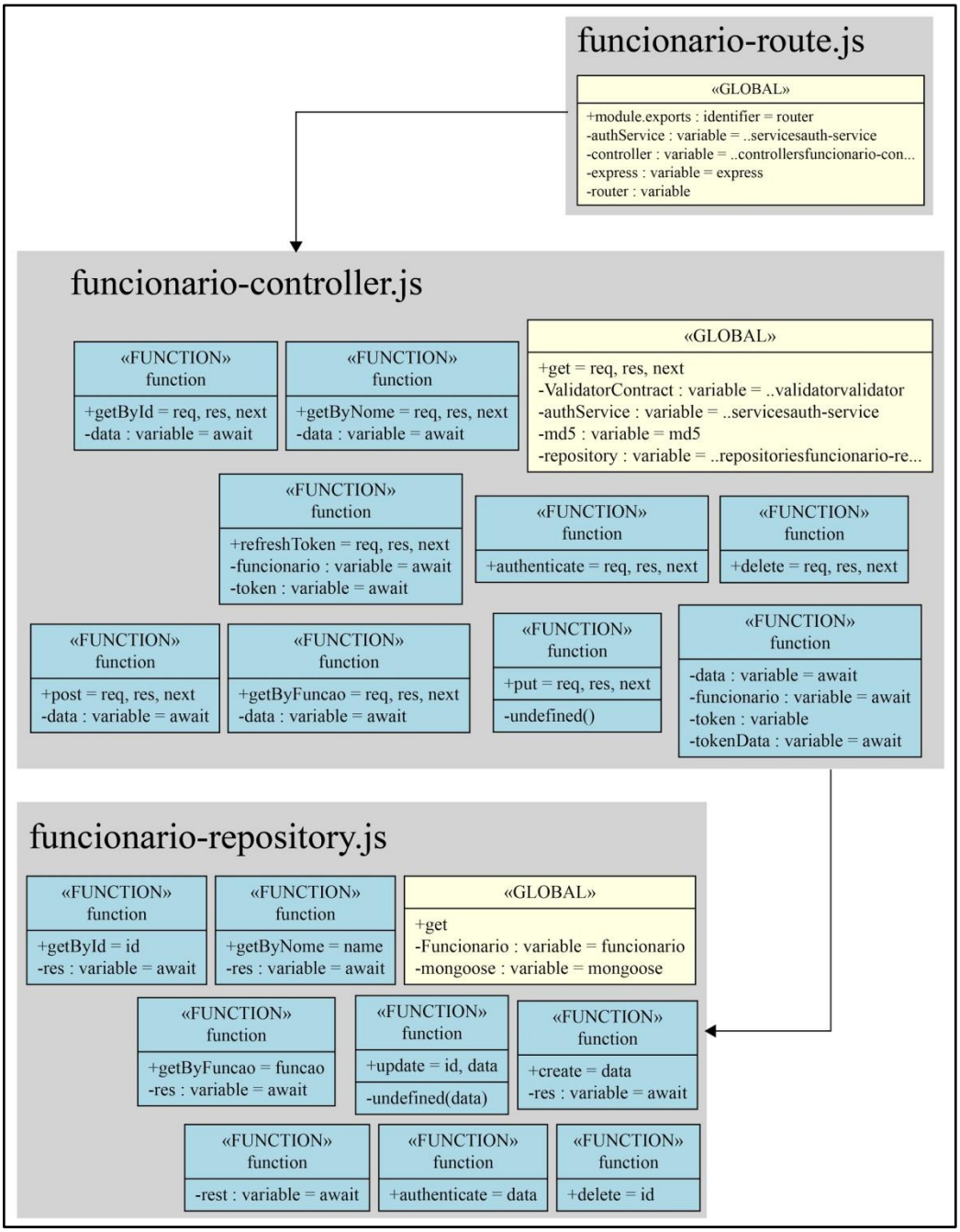
Figura 13. Diagrama do módulo empregador



Fonte: Elaborada pelo autor

O diagrama da Figura 14 é referente a classe de rotas *funcionário-route.js*. Este diagrama é semelhante ao diagrama feito na Figura 13. Estas classes são as responsáveis por todas as requisições e manipulações dos dados de possíveis empregados.

Figura 14. Diagrama do módulo funcionário

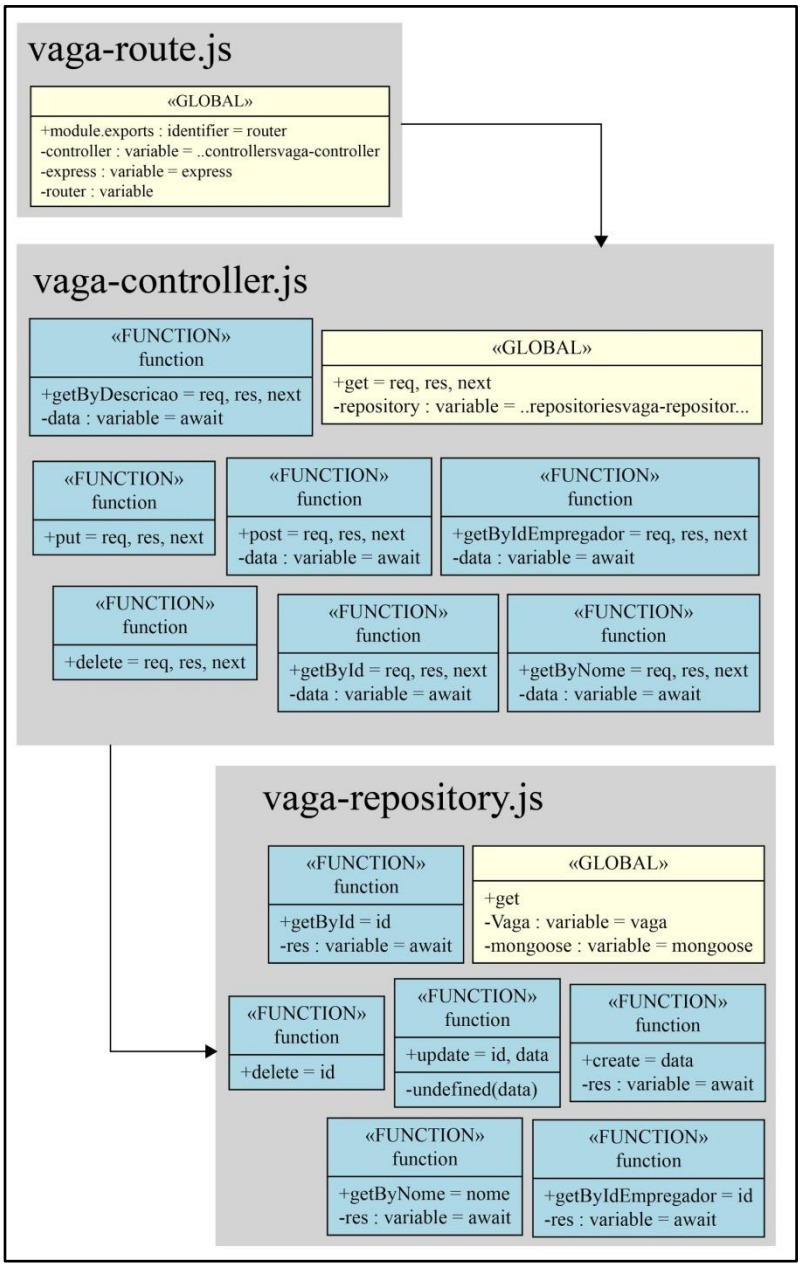


Fonte: Elaborada pelo autor

A Figura 15, correspondente ao diagrama feito para a classe *vaga-route.js*, é semelhante aos arquivos anteriores, porém possui menos métodos em cada uma das suas classes. Estas

classes são as responsáveis por todas as requisições e manipulações de vagas de empregos criadas a partir de um empregador cadastrado.

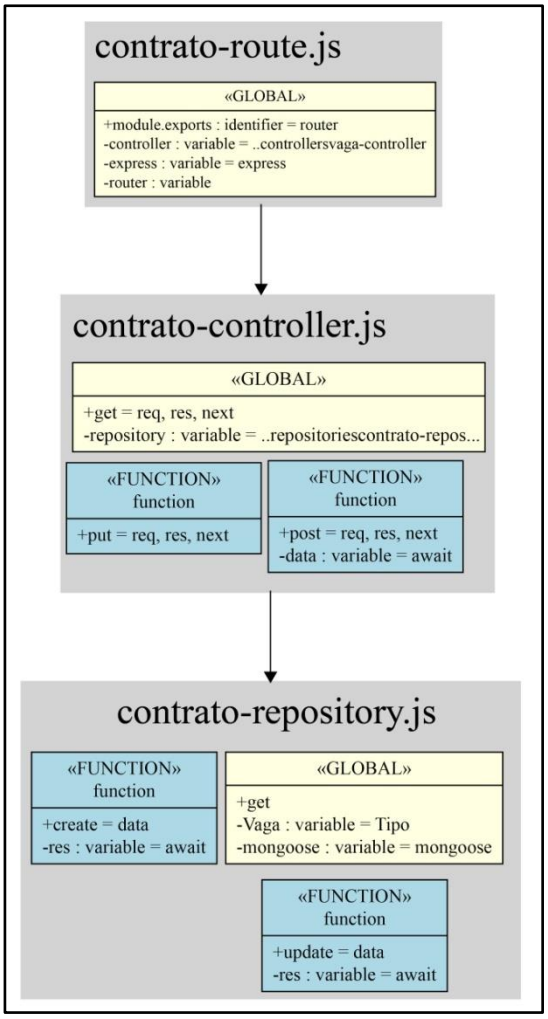
Figura 15. Diagrama do módulo vaga



Fonte: Elaborada pelo autor

O diagrama da Figura 16 é o da classe *contrato-route.js*, este é o último diagrama de classes para arquivos de rotas criados para o projeto. Estas classes são responsáveis por fazer a conexão entre empregadores, que se candidataram para alguma vaga, e os empregadores que criaram a vaga.

Figura 16. Diagrama do módulo Contrato



Fonte: Elaborada pelo autor

Conforme salientado acima, a criação dessa aplicação baseia-se em um arquivo de configurações de servidor, chamado *server.js*, na imagem a seguir é mostrado os métodos principais criado dentro da classe *server.js*, na linha 18 da Figura 17 encontra-se o método

responsável pela normalização da porta, na linha 32 tem-se uma função criada para caso houver problemas referente a permissões e utilização do endereço. Se não houver nenhum erro, durante o processo de criação do endereço, que se encontra na linha 11, a aplicação será iniciada e será direcionada para a classe inicial do sistema.

Figura 17. Estrutura da classe *Server.js*

```
5   const app = require('../src/app');
6   const http = require('http');
7   const debug = require('debug')('nodestr:server');
8
9   const port = normalizePort(process.env.PORT || '3000');
10  app.set('port', port);
11  const server = http.createServer(app);
12
13  server.listen(port);
14  server.on('error', onError)
15  server.on('listening', onListening)
16  console.log('Aplicação rodando na porta: ', port);
17
18  function normalizePort(val) {
19      const port = parseInt(val, 10);
20
21      if (isNaN(port)){
22          return val;
23      }
24
25      if (port >= 0){
26          return port;
27      }
28
29      return false;
30  }
31
32  function onError(error){
33      if (error.syscall != 'listen'){
34          throw error;
35      }
36
37      const bind = typeof port === 'string' ? 'Pipe ' + port : 'Port ' + port;
38
39      switch (error.code) {
40          case 'EACCES':
41              console.error(bind + ' Requer permissão mais elevada');
42              process.exit(1);
43              break;
44          case 'EADDRINUSE':
45              console.error(bind + ' já está em uso');
46              process.exit(1);
47              break;
48          default:
49              throw erros;
50      }
51  }
52
53  function onListening() {
54      const addr = server.address();
55      const bind = typeof addr === 'string' ? 'pipe ' + addr : 'port ' + addr.port;
56      debug('Escutanto ' + bind)
57  }
```

Fonte: Elaborada pelo autor

Para a inicialização da API é utilizado o seguinte comando *nodemon bin\server.js*. Após a inicialização da web api o serviço estará disponível localmente para ser utilizado acessando pelo endereço *http:/localhost:* colocando-se a porta que foi inicializada juntos ao arquivo *server.js*, que no nosso caso é a porta 3000, conforme a linha 9 da Figura 17. Utilizou-se para auxiliar e alavancar o desenvolvimento a dependência *nodemon*, que é capaz de, em cada salvamento do arquivo, reinicializar o serviço sem a necessidade de executar esse processo manualmente, então fica a critério do *nodemon* manter a API sempre atualizada constantemente mesmo se houverem melhorias e correções durante a sua criação.

Quando a porta é normalizada e a aplicação finalmente foi inicializada, então é acionada a classe principal do sistema, que é chamada de *app.js*. A imagem a seguir é a classe app.js da aplicação, essa classe é a responsável por instanciar toda api, na linha 13 da imagem 18 é utilizada a dependência do mongoose para criar uma conexão com o banco de dados, da linha 15 até a linha 19 são inicializadas as classes *models*, na linha 21 até a linha 25 são criadas as rotas a partir de arquivos criados utilizando o Express, da linha 27 a 39 possui métodos para configurações de requisições, como: tamanho de imagens permitidas, se a url será codificada, acessos a controles de *requests*, dentre outros. Por último, da linha 41 a 45, são configuradas rotas que a aplicação utilizará passando como parâmetros os objetos de rotas instanciados anteriormente.

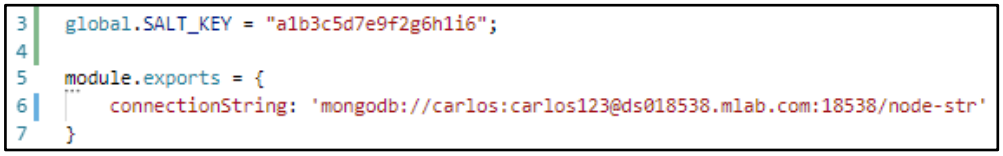
Figura 18. Estrutura da classe *App.js*

```
5   const express = require('express');
6   const bodyParser = require('body-parser');
7   const mongoose = require('mongoose');
8   const config = require('./config');
9
10  const app = express();
11  const router = express.Router();
12
13  mongoose.connect(config.connectionString);
14
15  const Funcionario = require('./models/funcionario');
16  const Empregador = require('./models/empregador');
17  const Vaga = require('./models/vaga');
18  const Tipo = require('./models/tipo');
19  const Contrato = require('./models/contrato');
20
21  const indexRoute = require('./routes/index-route');
22  const funcionarioRoute = require('./routes/funcionario-route');
23  const empregadorRoute = require('./routes/empregador-route');
24  const vagaRoute = require('./routes/vaga-route');
25  const contratoRoute = require('./routes/contrato-route');
26
27  app.use(bodyParser.json({
28      limit: '5mb'
29  }));
30  app.use(bodyParser.urlencoded({
31      extended: false
32  }));
33
34  app.use(function (req, res, next) {
35      res.header('Acces-Control-Allow-Origin', '*');
36      res.header('Acces-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept, x-access-token');
37      res.header('Acces-Control-Allow-Headers', 'GET, POST, PUT, DELETE, OPTIONS');
38      next();
39  });
40
41  app.use('/', indexRoute);
42  app.use('/funcionario', funcionarioRoute);
43  app.use('/empregador', empregadorRoute);
44  app.use('/vaga', vagaRoute);
45  app.use('/contrato', contratoRoute);
46
47  module.exports = app;
```

Fonte: Elaborada pelo autor

Essa classe é responsável por chamar uma classe de configuração externa, chamada de *config.js*. A imagem ilustra as configurações criadas na classe de configuração, na linha 3 da Figura 19, foi criada uma string para melhorar o processo de criptografia de senhas, na linha 6 foi criada uma string com a url de conexão a base de dados.

Figura 19. Arquivo de configuração config.js



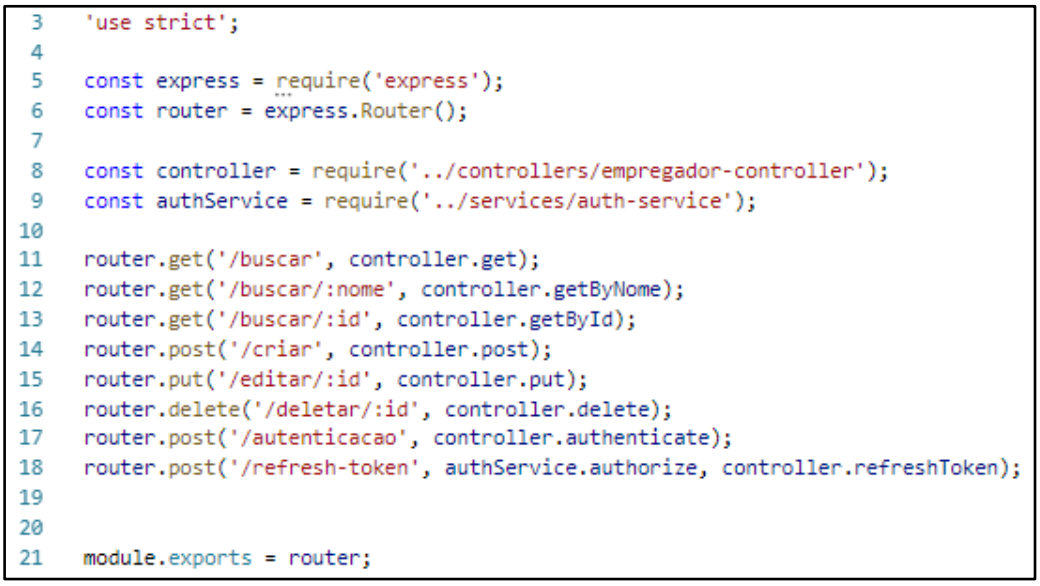
```
3  global.SALT_KEY = "a1b3c5d7e9f2g6h1i6";
4
5  module.exports = {
6      connectionString: 'mongodb://carlos:carlos123@ds018538.mlab.com:18538/node-str'
7  }
```

Fonte: Elaborada pelo autor

Então, quando a API está completamente inicializada, deve-se entender o caminho que cada requisição fará no sistema, para isso será montado um passo-a-passo em relação às classes e métodos que cada requisição passará, portanto, a API se comporta da seguinte maneira:

1. Primeiramente, com a utilização do *express*, é possível designar diferentes URIs para cada método do controlador, e por essas URIs viram todas as requisições do *front-end*. Para que todas ações saibam por onde tenham que chegar, foram criadas classes que são responsáveis por armazenar todos os *endpoints*, essas classes foram chamadas de *routes*. O exemplo a seguir é da classe *empregador-route.js*.

Figura 20. Estrutura da classe *empregador-route.js*



```
3   'use strict';
4
5   const express = require('express');
6   const router = express.Router();
7
8   const controller = require('../controllers/empregador-controller');
9   const authService = require('../services/auth-service');
10
11  router.get('/buscar', controller.get);
12  router.get('/buscar/:nome', controller.getByNome);
13  router.get('/buscar/:id', controller.getById);
14  router.post('/criar', controller.post);
15  router.put('/editar/:id', controller.put);
16  router.delete('/deletar/:id', controller.delete);
17  router.post('/autenticacao', controller.authenticate);
18  router.post('/refresh-token', authService.authorize, controller.refreshToken);
19
20
21  module.exports = router;
```

Fonte: Elaborada pelo autor

Com a utilização da classe *Router()* do *express*, criou-se diferentes ações CRUD, utilizando os métodos HTTP para cada URI diferente, como observa-se nas linhas 11 a 18 da Figura 20, esses métodos estão sendo chamados da classe *empregador-controller.js*, inicializada anteriormente na linha 8. Torna-se extremamente simples a distribuição de rotas utilizando a dependência do express. Resumidamente a requisição chegará a uma classe *route*, e essa distribuirá essas requisições para diferentes métodos dentro de uma classe *controller.*

2. A partir do momento que uma das rotas são chamadas pelo *front-end*, ou por qualquer ferramenta que seja possível simular requisições, ela é designada para um método criado dentro de uma classe *controller*, cada um desses métodos possui uma função específica, podendo ser de inserção, listagem, edição, deleção, listar com parâmetros, dentre outro. A imagem a seguir é uma parte da classe *empregador-controller.js*, responsável por conter todos os métodos que são chamados na classe *empregador-route.js*.

Figura 21. Alguns métodos da classe *empregador-controller.js*

```
3   'use strict';
4
5   const ValidatorContract = require('../validator/validator')
6   const repository = require('../repositories/empregador-repository');
7   const authService = require('../services/auth-service')
8   const md5 = require('md5');
9
10  exports.get = async(req, res, next) => {
11      try {
12          var data = await repository.get();
13          res.status(200).send(data);
14      } catch (e) {
15          res.status(500).send({
16              error: 'Falha ao processar requisição.'
17          });
18      }
19  };
20
21  exports.getByNome = async(req, res, next) => {···
30  }
31
32  exports.getById = async(req, res, next) => {···
41  }
42
43  exports.post = async(req, res, next) => {
44      let contract = new ValidatorContract();
45      contract.hasMinLen(req.body.nome, 3, 'O nome deve conter pelo menos 3 caracteres');
46      contract.isEmail(req.body.email, 'Deve ser um e-mail válido');
47      contract.hasMinLen(req.body.senha, 8, 'A senha deve conter no mínimo 8 caracteres');
48
49      if (!contract.isValid()){
50          res.status(400).send(contract.errors()).end();
51          return;
52      }
53
54      try {
55          await repository.create({
56              nome: req.body.nome,
57              cnpj: req.body.cnpj,
58              ativo: req.body.ativo,
59              email: req.body.email,
60              senha: md5(req.body.senha + global.SALT_KEY)
61          });
62          res.status(201).send({
63              message: 'Empregador cadastrado!'
64          });
65      } catch (e) {
66          res.status(500).send({
67              error: 'Falha ao processar requisição.',
68              data: e
69          });
70
71          res.status(400).send({
72              error: 'Falha ao cadastrar o empregador.',
73              data: e
74          });
75      }
76  };
```

Fonte: Elaborada pelo autor

Quando se chega em uma *controller*, é possível fazer várias manipulações com os dados que são recebidos através das requisições. Foram criados métodos capazes de autenticar os dados, utilizar criptografias, criar *tokens*, renovar *tokens*. A classe *controller* foi desenvolvida para ser a responsável por toda essa manipulação de dados entre o *front-end* do aplicativo e os repositórios da API, na linha 10 da Figura 21 observa-se o método para efetuar a busca de registros, contendo toda uma manipulação caso não houver registros. Na linha 43 é criado um método criado para criar um novo registro, neste método pode-se observar que foram criadas validações de dados, conforme linhas 45, 46 e 47, foram também feitas validações caso for efetuado o cadastro com sucesso ou não, conforme as linhas 62, 66 e 71.

No método de criação, também utilizou-se a dependência md5, a partir desta, consegue-se criptografar a senha utilizando criptografia MD5, na linha 60, ainda da Figura 21, é possível observar essa utilização. Para deixar ainda mais completa a criptografia, foi criada uma variável global chamada *SALT_KEY* que se encontra no arquivo de configuração, *config.js*, mostrado anteriormente, nela consta uma string aleatória criada pelo desenvolvedor.

Um outro exemplo de métodos pertencentes a classe *controller*, é o que faz a autenticação por e-mail e senha, esse método foi criado para se ter acesso a um *token*, e a partir dele receber todo o resto das funcionalidades. Na Figura 22, na linha 126 criou-se uma variável capaz de armazenar a resposta da busca por e-mail e senha, recebidos por parâmetros, do registro. Caso a busca na base de dados por esses dois parâmetros, possua um valor, é criado um *token*, contendo diversas informações, conforma a linha 137. Na linha 144 é o momento em que o *token* é retornado via *response* da função. Caso a busca não retorne registros, na linha 131 é criada essa validação para que a *response* retorne uma mensagem informativa.

Figura 22. Método responsável por efetuar a autenticação

```
124  exports.authenticate = async(req, res, next) => {
125      try {
126          const empregador = await repository.authenticate({
127              email: req.body.email,
128              senha: md5(req.body.senha + global.SALT_KEY)
129          });
130
131          if (!empregador) {
132              res.status(404).send({
133                  message: 'Empregador não encontrado',
134                  data: e
135              })
136          }
137          const token = await authService.generateToken({
138              nome: empregador.nome,
139              senha: empregador.senha,
140              email: empregador.email,
141              id: empregador._id
142          });
143          console.log(token)
144          res.status(201).send({
145              token: token,
146              data: {
147                  email: empregador.email,
148                  nome: empregador.nome
149              }
150          })
151
152      } catch (e) {
153          res.status(500).send({
154              error: 'Falha ao processar requisição.',
155              data: e
156          });
157      }
158  }
```

Fonte: Elaborada pelo autor

O método responsável por fazer a autenticação, juntamente com todos os outros métodos das ações CRUD, chama um método que é disponibilizado pela dependência *mongoose*, responsável por toda a comunicação entre a API e o MongoDB, dentro de uma classe auxiliar chamada *repository*, conforme consta na linha 126 da Figura 22.

3. Quando os dados foram manipulados pela classe *controller*, então essa é a responsável por fazer a comunicação com a camada *model*, porém foi desenvolvida uma classe auxiliar, capaz de conter todos os métodos que o *mongoose* mapeou automaticamente. O *mongoose* adapta o mapeamento de ações CRUD para ações em que o MongoDB

consiga saber em qual documento faz determinada ação. Esses métodos possuem nomenclatura padrão e não é possível alterá-los. A demonstração a seguir, da Figura 23, é dos métodos CRUD que foram criados dentro da classe *empregador-repository.js*.

Figura 23. Alguns métodos contidos na classe *empregador-repository.js*

```
3   'use strict';
4
5   const mongoose = require('mongoose');
6   const Empregador = mongoose.model('empregador');
7
8   exports.get = async () => {
9       const res = await Empregador.find({
10          ativo: true
11      }).populate('vagas');
12      return res;
13  }
14
15  exports.getByNome = async (name) => {···
22  }
23
24  exports.getById = async (id) => {
25      const res = await Empregador.findById(id);
26      return res;
27  }
28
29  exports.create = async (data) => {
30      var empregador = new Empregador(data);
31      await empregador.save();
32  }
33
34  exports.update = async (id, data) => {
35      await Empregador
36          .findByIdAndUpdate(id, {
37              $set: {
38                  nome: data.nome,
39                  cnpj: data.cnpj,
40              }
41          });
42  }
43
44  exports.delete = async (id) => {
45      await Empregador.findByIdAndRemove(id);
46  }
47
48  exports.authenticate = async (data) => {
49      const rest = await Empregador.findOne({
50          senha: data.senha,
51          email: data.email,
52      });
53      return rest;
54  }
```

Fonte: Elaborada pelo autor

É possível observar que é instanciado a classe *empregador* através do método *model()* do*mongoose*, na linha 6 da Figura 23. Essa classe será a responsável por efetuar todas as

ações, que o *mongoose* disponibiliza, diretamente no banco de dados, conforme observa-se nas linhas 8 o método *get* correspondente ao método Read, na linha 29 o método *create*, na linha 34 o método *update* e na linha 44 o método delete, assim efetuando todas as ações CRUD.

A *model* é a classe que contém todos os atributos e formatos em que o documento será gerado no banco. A classe repository foi desenvolvida para não sobrecarregar a classe *model* e nem a classe *controller*, nessa ficaram os métodos enxutos, deixando com uma aparência mais organizada. A classe *controller* está com a maior parte de uso das informações e a classe *model* é a que menos possui manipulação de dados.

4. Enfim, como foi utilizado um banco de dados não relacional, baseado em documentos JSON, não foi criada uma estrutura possuindo diversas *tab*elas e ligações entre elas, por isso a utilização de uma classe modelo. A Figura a seguir é a representação de uma classe *models*, denominada *empregador.js.* Na linha 8 da Figura 24, utiliza-se o *mongoose* para criar um *Schema*, ondepode-se restringir atributos requeridos, definição do tipo de atributo, criar lista de atributos e de *models*, designar uma mensagem caso algum campo requerido não seja preenchido, como está na linha 11, dentre outras funcionalidades. Resumidamente, a classe model da API é designada para ser um reflexo do que são os documentos gravados na base de dados, possuindo alguns avisos em casos de inconsistência.

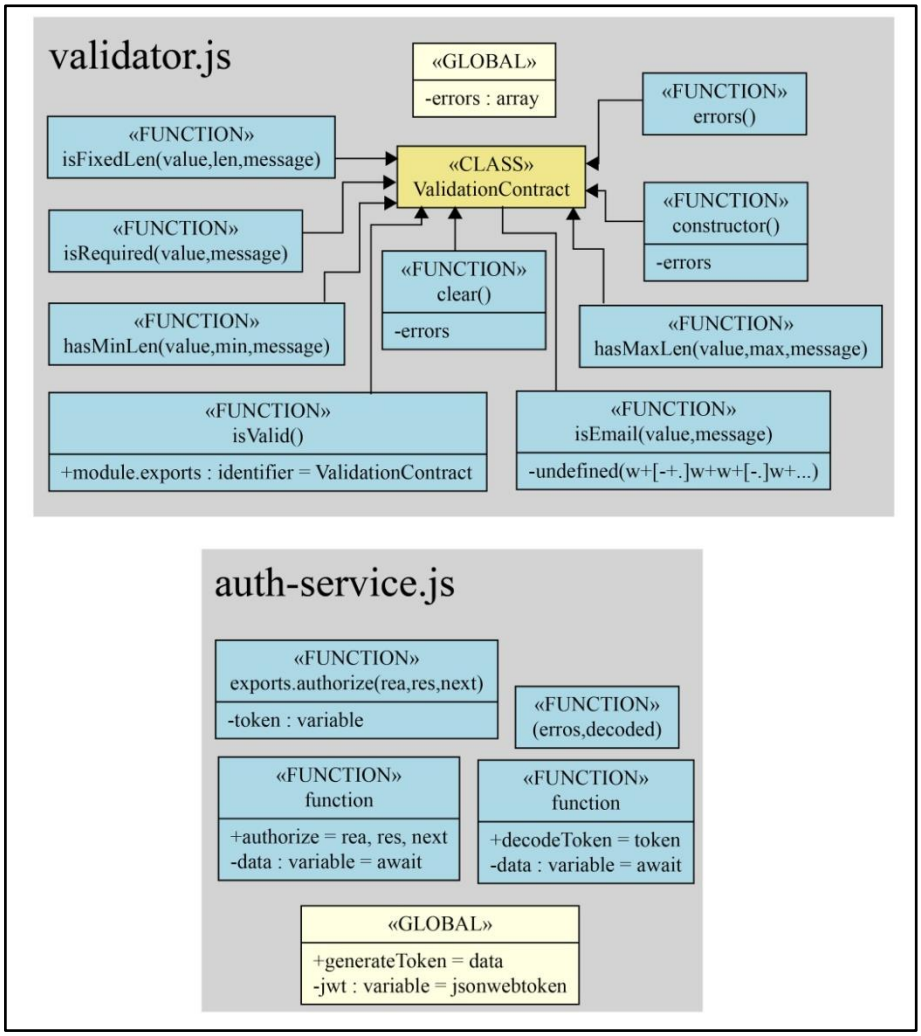
Figura 24. Estrutura da classe modelo *empregador.js*

```
'use strict'

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const schema = new Schema({
    nome: {
        type: String,
        required: [true, 'O nome é obrigatório']
    },
    cnpj: {
        type: Number,
        required: [true, 'O CNPJ é obrigatório'],
        trim: true
    },
    ativo: {
        type: Boolean,
        required: true,
        default: true
    },
    senha: {
        type: String,
        required: [true, 'A senha é obrigatória'],
        trim: true
    },
    email: {
        type: String,
        required: true,
        trim: true
    },
});

module.exports = mongoose.model('empregador', schema);
```

Fonte: Elaborada pelo autor

Além do arquivo de configuração *config.js* que foi criado, criou-seo mais dois arquivos que auxiliam no processo de autenticação e de validação de dados. Criou-se um diagrama que ilustra esses dois arquivos, como mostra a Figura 25.

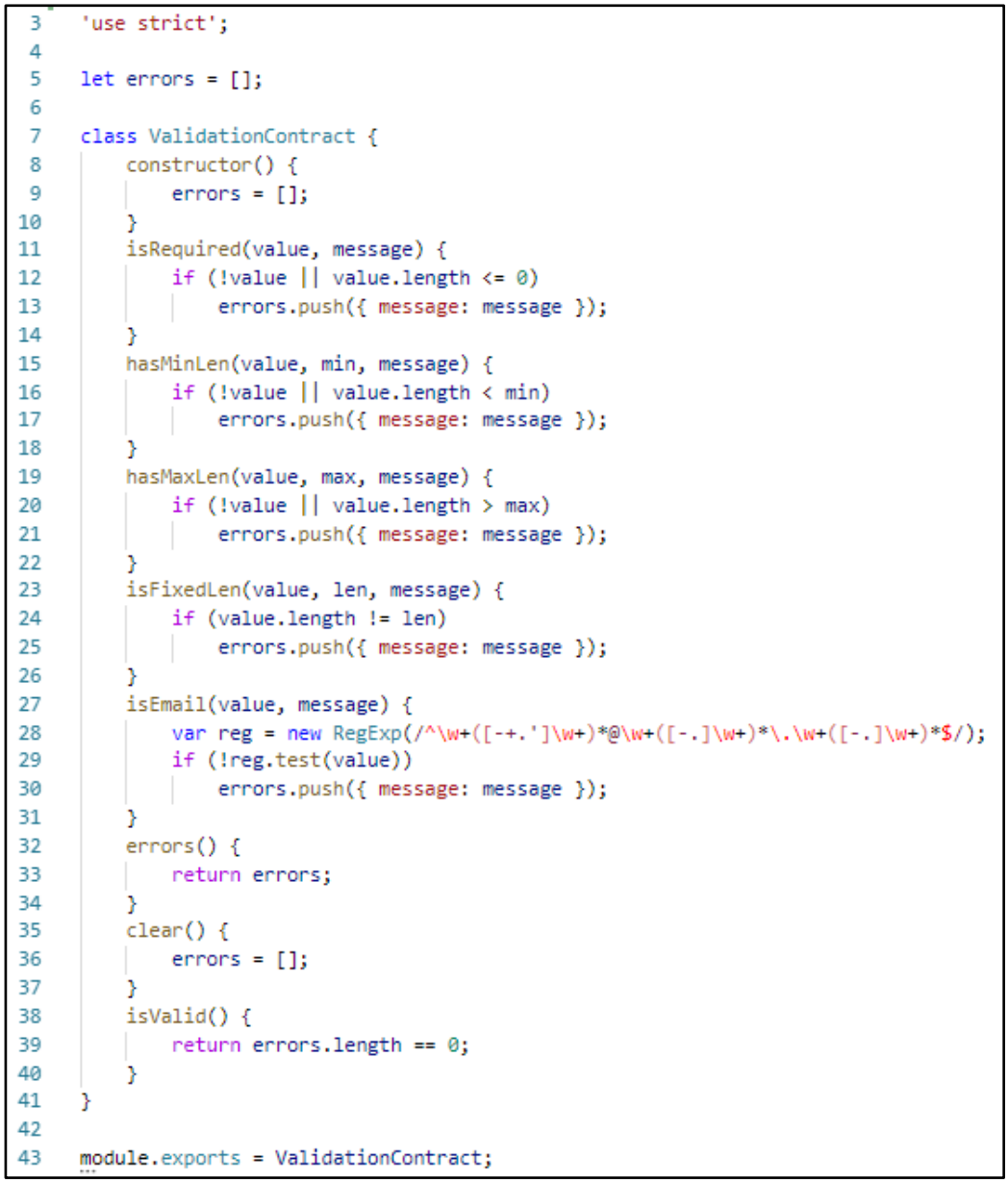
Figura 25. Diagramas da classe *validator.js* e *auth-service.js*



Fonte: Elaborada pelo autor

O arquivo de validação é chamado de *validator.js*, nele foram implementadas funções em que se pode saber se um e-mail é válido, se o campo foi preenchido, se possui tamanho mínimo e máximo, dentre outros. Esse arquivo foi gerado para não ficar apenas a critério do *front-end* fazer validações de informações, possuindo maior segurança referente as informações no momento em que se gera e altera novos registros. A Figura 26 ilustra todos os métodos utilitários criados dentro da classe de validação.

Figura 26. Estrutura da classe auxiliar *validator.js*



```
'use strict';

let errors = [];

class ValidationContract {
    constructor() {
        errors = [];
    }
    isRequired(value, message) {
        if (!value || value.length <= 0)
            errors.push({ message: message });
    }
    hasMinLen(value, min, message) {
        if (!value || value.length < min)
            errors.push({ message: message });
    }
    hasMaxLen(value, max, message) {
        if (!value || value.length > max)
            errors.push({ message: message });
    }
    isFixedLen(value, len, message) {
        if (value.length != len)
            errors.push({ message: message });
    }
    isEmail(value, message) {
        var reg = new RegExp(/^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/);
        if (!reg.test(value))
            errors.push({ message: message });
    }
    errors() {
        return errors;
    }
    clear() {
        errors = [];
    }
    isValid() {
        return errors.length == 0;
    }
}

module.exports = ValidationContract;
```

Fonte: Elaborada pelo autor

O arquivo que facilita na autenticação e geração de *token* foi nomeado de *auth-service.js*. Neste arquivo, com a utilização da dependência que auxilia na manipulação de web *tokens*, chamada de *jsonwebtoken*, foram criados métodos capazes de gerar um novo *token* e de decodificar o mesmo, foi desenvolvido também um método que é capaz de autorizar caso o usuário possua o *token* e de negar caso sejainválido ou inexistente.

Figura 27. Estrutura da classe auxiliar *auth-service.js*

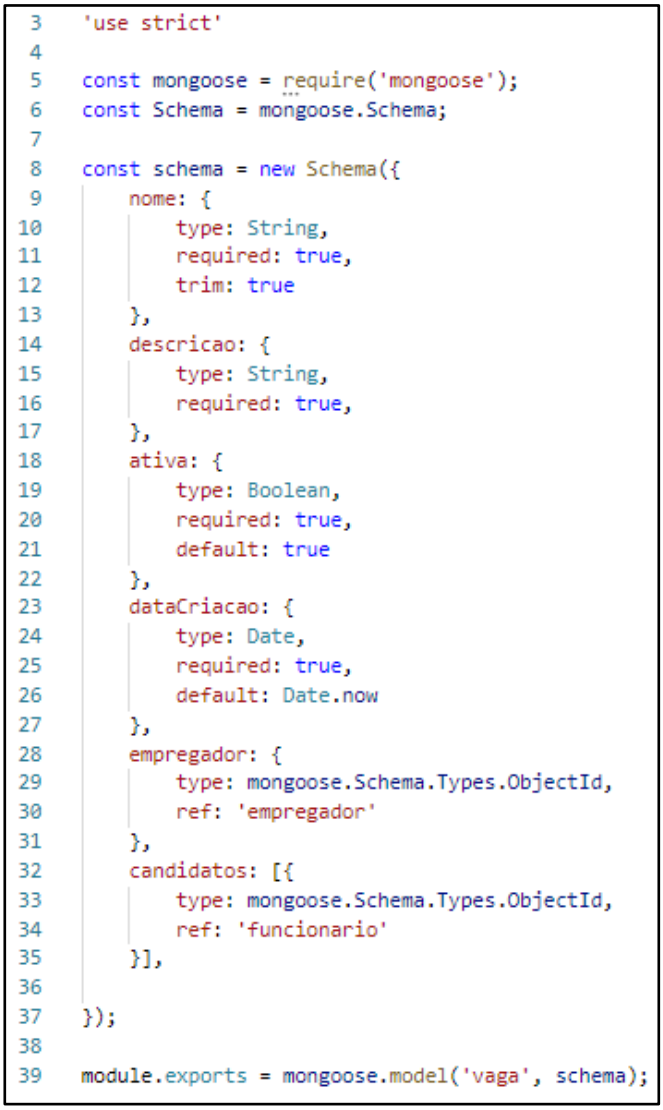
```
3   'use-district'
4
5   const jwt = require('jsonwebtoken');
6
7   exports.generateToken = async (data) => {
8       try {
9           var data = await jwt.sign(data, '', { expiresIn: '1d' });
10      } catch(e) {
11          console.log(e);
12      }
13      return data
14  }
15
16  exports.decodeToken = async (token) => {
17      var data = await jwt.verify(token, '');
18      return data;
19  }
20
21  exports.authorize = function (rea, res, next)  {
22      var token = req.body.token || req.query.token || req.headers['x-acces-token'];
23      console.log('entrou no authorize');
24      if (!token) {
25          res.status(401).json({
26              message: 'Acesso Restrito'
27          });
28      } else {
29          jwt.verify(token, '', function (erros, decoded) {
30              if (erros) {
31                  res.status(401).json({
32                      message: 'Token inválido'
33                  });
34              } else {
35                  next();
36              }
37          });
38      }
39  }
```

Fonte: Elaborada pelo autor

Observa-se que a ilustração demonstra o funcionamento dessas validações, ao receber o parâmetro de *token*, são designadas as funcionalidades. Na função da linha 21 da Figura 27, percebe-se que foram feitas 3 etapas. A primeira etapa é a linha 24, onde é feita validação de existência de *token*, caso não haja retornará uma mensagem informativa. A segunda e a terceira etapa são as linhas 20 e 24, onde é a validação do *token*, na linha 35 poderá seguir com a requisição caso seja um *token* válido, caso for inválido, será retornada uma mensagem informativa, conforme a linha 32.

A aplicação foi desenvolvida em torno de uma série de classes principais que foram denominadas de *vaga-controller.js*, *vaga-repository.js* e *vaga.js*. Essas classes são as responsáveis por buscar as vagas para os empregados e de gerar e excluir novas vagas para os empregadores. Na classe *model*, foram adicionadas uma lista de candidatos que é atualizada a cada entrada de um novo, junto com ela possuí o empregador responsável pela vaga. Como está sendo utilizado um banco não relacional, as *tab*elas não se relacionam por identificadores únicos, porém pode-se criar referências a outros documentos. Segue abaixo uma imagem referente a classe *model*, *vaga.js*.

Figura 28. Estrutura da classe modelo *vaga.js*



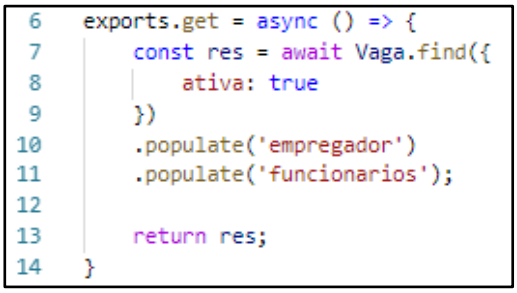
```
3   'use strict'
4
5   const mongoose = require('mongoose');
6   const Schema = mongoose.Schema;
7
8   const schema = new Schema({
9       nome: {
10          type: String,
11          required: true,
12          trim: true
13      },
14      descricao: {
15          type: String,
16          required: true,
17      },
18      ativa: {
19          type: Boolean,
20          required: true,
21          default: true
22      },
23      dataCriacao: {
24          type: Date,
25          required: true,
26          default: Date.now
27      },
28      empregador: {
29          type: mongoose.Schema.Types.ObjectId,
30          ref: 'empregador'
31      },
32      candidatos: [{
33          type: mongoose.Schema.Types.ObjectId,
34          ref: 'funcionario'
35      }],
36
37  });
38
39  module.exports = mongoose.model('vaga', schema);
```

Fonte: Elaborada pelo autor

Observa-se que ao final dos atributos, na Figura 28 nas linhas 28 e 32, existem dois que fazem referência, com a utilização do mongoose, a um objeto ID de outro documento. No momento em que se buscam essas informações, criou-se um método na classe *repository* que é capaz de montar um novo objeto, com todos os atributos e ainda com os atributos referenciados criados na classe *model*, além da própria classe. Assim, quando forem retornados pela a *response*, o objeto estará completo, contendo todas as informações de todos os documentos envolvidos. A imagem a seguir mostra um dos métodos criado que faz essa população, na Figura 29, as linhas

10 e 11 são as responsáveis pela população dos dois outros documentos referenciados como atributos.

Figura 29. Método que busca e preenche os documentos

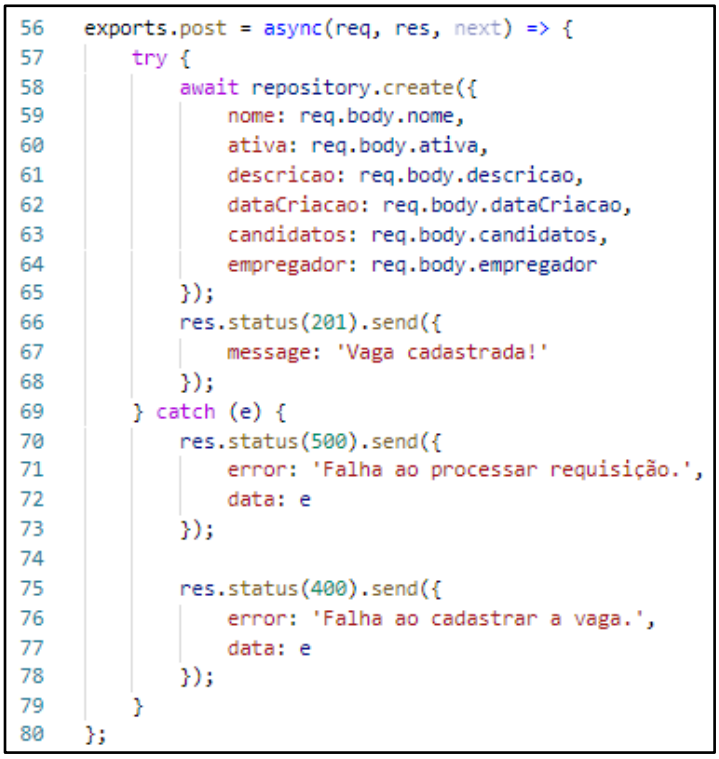


```
6   exports.get = async () => {
7       const res = await Vaga.find({
8           ativa: true
9       })
10      .populate('empregador')
11      .populate('funcionarios');
12
13      return res;
14  }
```

Fonte: Elaborada pelo autor

A criação de uma nova vaga de emprego é parcialmente parecida ao exemplo utilizado anteriormente da classe *controller*, *empregador-controller.js*. Na classe *vaga-controller.js*, foram criados diversos métodos parametrizados, e todos os métodos CRUD. O método POST ficou simples, pelo fato de que o *mongoose* interpreta as informações e as usa de forma coerente aos atributos criados no *model*. A Figura 30 mostra o método POST responsável pela criação de uma nova vaga de emprego.

Figura 30. Método de criação de uma nova vaga

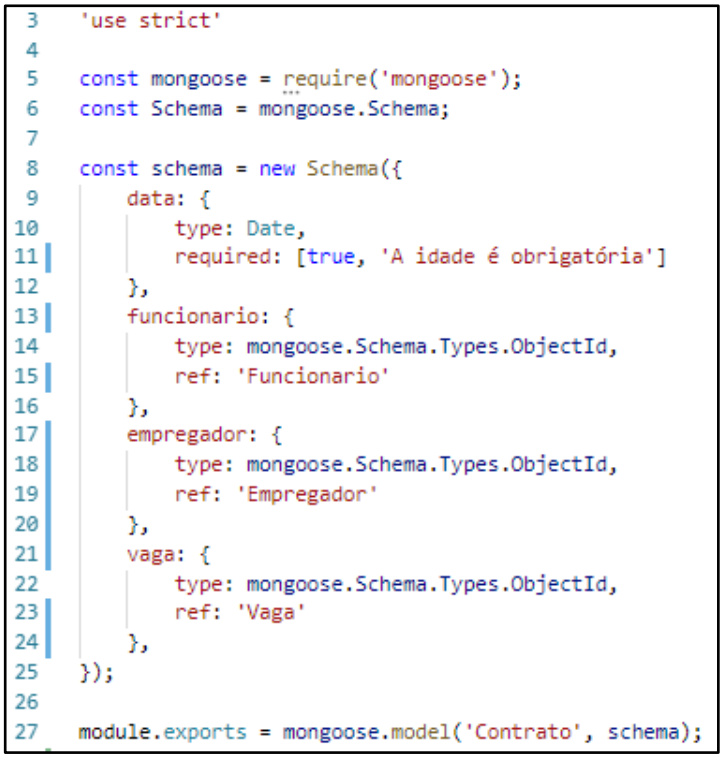


```
56  exports.post = async(req, res, next) => {
57      try {
58          await repository.create({
59              nome: req.body.nome,
60              ativa: req.body.ativa,
61              descricao: req.body.descricao,
62              dataCriacao: req.body.dataCriacao,
63              candidatos: req.body.candidatos,
64              empregador: req.body.empregador
65          });
66          res.status(201).send({
67              message: 'Vaga cadastrada!'
68          });
69      } catch (e) {
70          res.status(500).send({
71              error: 'Falha ao processar requisição.',
72              data: e
73          });
74
75          res.status(400).send({
76              error: 'Falha ao cadastrar a vaga.',
77              data: e
78          });
79      }
80  };
```

Fonte: Elaborada pelo autor

Para finalizar a web API, criou-se uma classe *model*, capaz de armazenar o candidato que foi escolhido para uma determinada vaga de um empregador, essa classe foi chamada de *contrato.js*. A classe contrato.js se baseia em um model capaz de armazenar três atributos referenciados da classe *vaga.js*, *empregador.js* e *funcionario.js.* Assim, deixando mais simples a forma em que se busca ou se es*tab*elecem uma ligação entre um empregado e um empregador. A classe é construída conforme a Figura 31, nas linhas 13, 17 e 21 pode-se identificar a referência, utilizando o *mongoose*, às classes mencionadas anteriormente.

Figura 31. Estrutura da classe modelo *contrato.js*



```
'use strict'

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const schema = new Schema({
    data: {
        type: Date,
        required: [true, 'A idade é obrigatória']
    },
    funcionario: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Funcionario'
    },
    empregador: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Empregador'
    },
    vaga: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Vaga'
    },
});

module.exports = mongoose.model('Contrato', schema);
```

Fonte: Elaborada pelo autor

Foram testados todos os métodos de todas as classes relatadas neste tópico utilizando a ferramenta *Postman*. Com essa ferramenta, além da execução dos testes de rotas, métodos (busca, inserção, exclusão, edição), retornos, parâmetros e funções, criou-se um grande volume de dados que contempla todas as dependências necessárias para uma estrutura de objetos suficiente para a utilização da mesma.

4.2 FRONT-END

O front-end é a camada *view* da nossa arquitetura MVC, nessa camada foi iniciado o processo de desenvolvimento de um aplicativo nativo em React Native. Esse aplicativo foi iniciado a partir de uma estrutura de modelagem criada pelo Expo. O aplicativo foi divido em pequenos componentes a fim de deixar o desenvolvimento mais dinâmico e possuir um maior reaproveitamento de código. Inicialmente, quando se cria um projeto utilizando o Expo, já se monta uma arquitetura de classes e pastas padrão referente ao tipo de projeto que é iniciado.

Primeiramente foi-se criada uma classe inicial que contém uma série de métodos que são executados após e durante a abertura do aplicativo. Essa classe é denominada *App.js*. Cada classe feita em React Native deve ser exportada como um componente, para se exportar uma classe deve-se ter uma forma de renderizar aquele componente, então, para a classe *App.js*, renderizou-se em JSX um componente de navegação de telas, conforme mostrado na imagem a seguir.

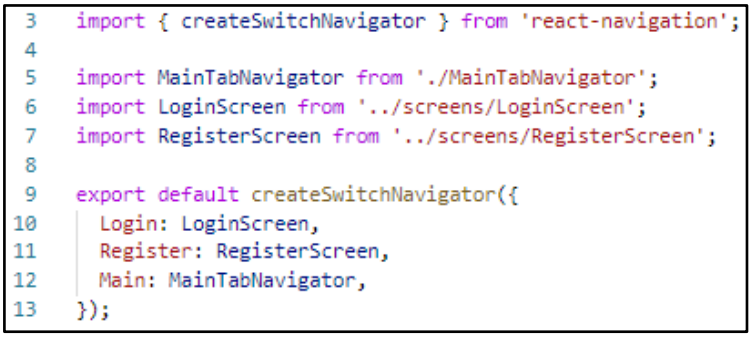
Figura 32. Estrutura da classe *App.js*

```
3   import React from 'react';
4   import { Platform, StatusBar, StyleSheet, View } from 'react-native';
5   import { AppLoading, Asset, Font, Icon } from 'expo';
6   import AppNavigator from './navigation/AppNavigator';
7
8   export default class App extends React.Component {
9     state = {
10      isLoadingComplete: false,
11    };
12
13    render() {
14      if (!this.state.isLoadingComplete && !this.props.skipLoadingScreen) {
15        return (
16          <AppLoading
17            startAsync={this._loadResourcesAsync}
18            onError={this._handleLoadingError}
19            onFinish={this._handleFinishLoading}
20          />
21        );
22      } else {
23        return (
24          <View style={styles.container}>
25            {Platform.OS === 'ios' && <StatusBar barStyle="default" />}
26            <AppNavigator />
27          </View>
28        );
29      }
30    }
```

Fonte: Elaborada pelo autor

Observa-se que na linha 13 da Figura 32 é o momento em que a classe é renderizada a partir do método render(), e então é criado em formado JSX uma chamada a outro componente de uma classe capaz de iniciar o processo de navegação, encontrado na linha 26. Esse componente foi criado dentro da classe *AppNavigation.js*, nesta foram exportadas todas as telas do aplicativo, utilizando uma dependência do React Native, chamada *React Navigation*. A dependência faz com que todas as telas iniciadas a partir dessa classe consigam navegarem entre si, disponibilizando um fluxo de navegação simples através de camadas. A imagem a seguir mostra como foi criada a classe *AppNavigation.js.*

Figura 33. Estrutura da classe *AppNavigation.js*



```
3   import { createSwitchNavigator } from 'react-navigation';
4
5   import MainTabNavigator from './MainTabNavigator';
6   import LoginScreen from '../screens/LoginScreen';
7   import RegisterScreen from '../screens/RegisterScreen';
8
9   export default createSwitchNavigator({
10    Login: LoginScreen,
11    Register: RegisterScreen,
12    Main: MainTabNavigator,
13  });
```

Fonte: Elaborada pelo autor

Verifica-se que na linha 9 da Figura 33, foram exportadas, com a utilização de um método de dependência *React Navigation*, as classes e seus nomes que serão utilizados no momento de navegação. A partir do momento em que a tela de *Login* é chamada, ela entra no topo de uma pilha e só sai no momento em que outra tela é chamada. A classe de *login*, *LoginScreen.js*, foi criada com métodos capazes de efetuar o *login*, buscar um *token*, validar se o *token* já está sendo utilizado, dentro outros métodos auxiliares. A imagem a seguir é a ilustração do método responsável por efetuar o processo de *login* da aplicação.

Figura 34. Método que efetua o processo de *login*

```
47  login = async () => {
48    if (this.validacao()) {
49      try {
50        const response = await api.post("empregador/autenticacao", {
51          email: this.state.email,
52          senha: this.state.senha
53        });
54
55        if (response !== "undefined") {
56
57          const data = [response.data.data];
58          const token = [response.data.token];
59
60          AsyncStorage.multiSet([
61            ["@api:token", JSON.stringify(token)],
62            ["@api:data", JSON.stringify(data)]
63          ]);
64
65          this.props.navigation.navigate("Home");
66        }
67      } catch (response) {
68        this.setState({ errorMessage: "E-mail ou senha incorretos" });
69      }
70    }
71  };
```

Fonte: Elaborada pelo autor

Observa-se, na linha 50, da Figura 34, que foi criada uma variável responsável por armazenar a resposta de uma requisição via método POST para a web API, passando como parâmetros o e-mail e a senha do registro. Após o retorno do método POST, essa variável é submetida a um teste para saber se retornou registros ou não, conforme mostra a linha 55. Caso o método retorna registros válidos, na linha 60, ainda da Figura 34, foi criado uma função capaz de gravar no armazenamento do dispositivo as informações retornadas para que se possa utilizar posteriormente, e na linha 65 é chamada uma tela inicial. Em base ao armazenamento no dispositivo, foi desenvolvido um método, que é inicializado junto com a abertura do aplicativo, capaz de verificar se existe um *token* armazenado no dispositivo, caso haja, o processo de *login* é simplificado é já muda para uma tela inicial do aplicativo.

Na tela *Register*, *RegisterScreen.js*, foi desenvolvida para ser a tela de cadastro da aplicação, nela constam métodos capazes de efetuar a validação dos campos e a utilização de método POST para efetuar a criação de novos registros. Para a criação dessa tela foi utilizado uma pequena dependência chamada *tcomb-form-native*, com ela é possível efetuar manipulações

em formulários, deixando mais simples a passagem de valores e verificações de preenchimento dos mesmos. A imagem a seguir é um exemplo do método responsável por efetuar a criação de novos registros.

Figura 35. Método de criação de um registro

```
59      _handleAdd = () => {
60        const value = this.refs.form.getValue();
61
62        if (value) {
63          const data = {
64            nome: value.nome,
65            email: value.email,
66            ativo: true,
67            senha: value.senha,
68            cpf_cnpj: value.cpf_cnpj,
69          }
70
71          const json = JSON.stringify(data);
72
73          fetch('empregador/criar', {
74            method: 'POST',
75            headers: {
76              'Content-Type': 'application/json',
77              Accept: 'application/json'
78            },
79            body: json
80          })
81          .then((response) => response.json())
82          .then(() => {
83            alert('Registrado com sucesso.');
84            this.props.navigation.navigate("Login");;
85          })
86          .catch((error) => {
87            alert('Houve erro durante a execução.');
88          })
89          .done()
90        } else {
91          alert('Por favor preencha os campos e tente novamente.')
92        }
93      }
```

Fonte: Elaborada pelo autor

Na linha 60 da Figura 35, é chamada uma função capaz de obter os valores do formulário e inseri-los em uma variável, na linha 62 essa variável é submetida a um teste, caso existam valores, a variável é desmontada e armazenada em um objeto data, caso não existam valores, é retornada uma mensagem informativa para o usuário, conforme linha 90. Na linha 71, ainda da

mesma figura, é transformado o objeto data em um objeto JSON e, enfim, na linha 73 é iniciada a chamada ao método POST, passando como parâmetro a variável json. Se o registro for efetuado com sucesso é mostrada uma mensagem informativa, como mostra a linha 83 e, na linha 84 é chamada a tela de *login*.

Na tela *Main*, ou tela inicial do aplicativo encontra-se uma navegação por *tabs,* a primeira *tab* irá conter todas as oportunidades de emprego disponíveis, e a outra *tab* é a configuração de perfil do usuário. A classe *MainTabNavigator.js*, foi criada para manter todas as *tabs* envolvidas, para cada uma das *tabs*, foram exportadas variáveis denominadas *stacks* referentes a uma *screen*. Ao final dessa classe são exportadas todas as variáveis *stacks*, dentro de cada variável existe um método de configurações disponibilizado pelo *React Navigation*, onde contém uma manipulação de imagens, palavras, ícones. Resumindo, essa classe é uma barra que contém 2 *tabs*, a primeira denominada *Home* e a segunda de *Settings* e, ao clicar em alguma delas irá abrir uma tela correspondente.

Figura 36. Estrutura da classe *MainTabNavigation.js*

```
3   import React from 'react';
4   import { Image, StyleSheet } from 'react-native';
5   import { createStackNavigator, createBottomTabNavigator } from 'react-navigation';
6
7   import HomeScreen from '../screens/HomeScreen';
8   import SettingsScreen from '../screens/SettingsScreen';
9
10  const HomeStack = createStackNavigator({
11    Home: HomeScreen,
12  });
13
14  HomeStack.navigationOptions = {
15    tabBarOptions: {
16      showLabel: false
17    },
18    tabBarIcon: () => (
19      <Image
20        source={require('../assets/images/magnifying-glass.png')}
21        style={styles.icon}
22      />
23    ),
24  };
25
26  const SettingsStack = createStackNavigator({···
28  });
29
30  SettingsStack.navigationOptions = {···
40  };
41
42  const styles = StyleSheet.create({···
48  });
49
50  export default createBottomTabNavigator({
51    HomeStack,
52    SettingsStack,
53  });
```

Fonte: Elaborada pelo autor

Na linha 10 da Figura 36, observa-se a criação de uma variável *HomeStack*, que será a *tab* de navegação capaz de armazenar uma *screen* e, na linha 14 são feitas algumas configurações referentes a *tab HomeStack.* Na linha 50, ainda da Figura 36, então, são exportadas todas as *tabs* (*Stacks*) da tela inicial.

Conforme descrito anteriormente, foram desenvolvidas *screens*, referentes a cada *tab* do aplicativo. A *screen home* é a parte inicial do aplicativo, nesta classe foram desenvolvidas as funcionalidades de busca de vagas e os candidatos para cada vaga. Desenvolveu-se uma *scrollview*, dentro desta é criado um componente capaz de pegar cada uma das vagas disponíveis e fazer uma listagem de suas informações.

Figura 37. Método da classe *HomeScreen.js*

```
23    async componentDidMount() {
24      this.loadVagas();
25    }
26
27    loadVagas = async () => {
28      const jsonToken = await AsyncStorage.getItem('@api:token');
29      const token = JSON.parse(jsonToken);
30      const response = await api.get('vaga/buscar/empregador/' + token[0].id);
31
32      this.setState({
33        vagas: response.data
34      });
35
36      await AsyncStorage.setItem('@api:vaga', JSON.stringify(this.state.vagas));
37    };
38
39    render() {
40      return (
41        <View style={styles.container}>
42          <ScrollView contentContainerStyle={styles.vagaList}>
43          {this.state.vagas.map(vaga =>
44              <Vaga key={vaga._id} data={vaga} />
45            )}
46          </ScrollView>
47        </View>
48      );
49    }
50  }
```

Fonte: Elaborada pelo autor

A imagem anterior, mostra os principais métodos da classe *HomeScreen.js*, observa-se que na linha 23, da Figura 37 criou-se um método responsável por, toda a vez que se entrar nesta tela, carregar a função de buscar vagas disponíveis. Na linha 27, ainda da Figura 37, desenvolveu-se a função capaz de efetuar uma requisição GET para a web API passando como parâmetro ID encontrado dentro do objeto *token*, que foi armazenado no dispositivo no momento do *login*. Ao retornar do método HTTP, conforme a linha 32, a resposta é armazenada em uma variável do estado do React Native, essa variável vai ser atualizada toda a vez que a tela processar um *reload*. Na linha 43 desenvolveu-se um método que faz um *loop* na variável armazenada no objeto *state* e, toda a vez que o *loop* der uma volta, é criado um componente, chamado Vaga, como mostra a linha 44. Esse componente é o responsável pela listagem dos atributos da vaga que se encontra naquela volta do *loop*.

Figura 38. Método render do componente Vaga

```
36      render() {
37          moment.locale("pt-br");
38          return (
39              <View style={styles.vagas}>
40                  <View style={styles.vagaInfo}>
41                      <View style={styles.container}></View>
42                      <View style={styles.vagaInfoHeader}>
43                          <Text style={styles.vagaNome}>{this.props.data.nome}
44                              {this._renderButton('Ver', () => this.setState({ visibleModal: true }))}
45                          </Text>
46                          <Text style={styles.vagaDta}>
47                              Anunciada {moment(this.props.data.dataCriacao).fromNow()}
48                              <Text style={styles.qtdCandidatos}>
49                                  {" "} - {this.props.data.candidatos.length} candidatos
50                              </Text>
51                          </Text>
52                      </View>
53                      <Text style={styles.vagasDescricao}>{this.props.data.descricaoMini}</Text>
54                  </View>
55
56                  <Modal
57                      isVisible={this.state.visibleModal}
58                      animationIn={'slideInLeft'}
59                      animationOut={'slideOutRight'}
60                      swipeArea={20}
61                      swipeThreshold={50}>
62                          {this.state.visibleModal === true ? this._renderModalContent() : ""}
63                  </Modal>
64              </View>
65          );
66      }
```

Fonte: Elaborada pelo autor

Neste componente foi-se desenvolvido uma estrutura em JSX que contém as informações da vaga disponível conforme mostra a imagem anterior. Neste componente também foi criado um método GET capaz de buscar as informações dos funcionários que se candidataram para vaga em questão. Toda a vez que a vaga for selecionada, irá mostrar os candidatos que estão disponibilizados para a mesma.

Foram criados outro métodos capazes de renderizar um modal, renderizar botões, entre outros, assim, deixando o componente ainda mais dividido em métodos para facilitar a leitura e entendimento. Na linha 56 da Figura 38, por exemplo, encontra-se implementada a chamada a um componente modal, dentro dele, no momento em que é visível, foi chamada uma função capaz de carregar o seu conteúdo, conforme a linha 62. Ao renderizar o conteúdo do modal, é chamada uma função que faz a busca de funcionários candidatos para aquela vaga em questão. A imagem a seguir mostra a função criada para executar essa busca dos funcionários. Na linha 20, ainda da

mesma figura, foi desenvolvida a requisição GET, passando como parâmetro o ID do funcionário para que se possam buscar as informações referentes ao mesmo.

Figura 39. Método que busca os funcionários

```
19      getCandidato = async (cand) => {
20          const candidato = await api.get('funcionario/buscar/id/' + cand);
21          return candidato.data;
22      }
```

Fonte: Elaborada pelo autor

Na linha 21 do código mostrado na Figura 39, a requisição *request* é retornada em formato de objeto, sendo ela positiva ou não, se não existirem registros, será mostrada uma lista vazia, caso haja registros irá mostrar todas as suas informações.

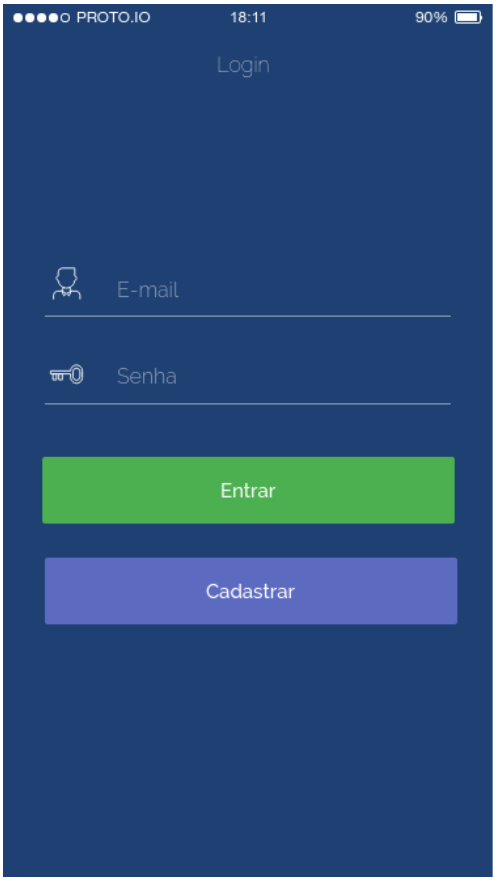
Para finalizar, além da *tab* que lista as vagas como mencionado acima, foi desenvolvida uma aba em que mostra o perfil do usuário, contendo as informações cadastradas por ele. O perfil de usuário, além das suas informações irá conter botões para configurações da vaga para empregadores e informações das vagas em que empregados se candidataram. O método de geração de uma nova vaga, e atualização da mesma, juntamente com a adição de novos candidatos é extremamente parecido com o método mostrado na imagem 35, porém é adicionado o *token* ao enviar os parâmetros. Pode-se enviar informações repetidas para a web API, ela é capaz de identificar quais as informações foram alteradas, utilizando como mediador desse processo o *token* e as informações contidas nele.

4.3 RESULTADOS PARCIAIS

Como resultados parciais a criação de layouts para as telas que foram desenvolvidas, auxiliando o manuseio do aplicativo pelo usuário. As telas que foram desenvolvidas são basicamente a tela de *login*, cadastro e listagem de vagas disponíveis. Essas telas foram desenvolvidas possuindo as interações e um fluxo de navegação entre as mesmas. Como auxilio foram desenvolvidos protótipos que se espelham nas telas desenvolvidas.

O primeiro protótipo, observado na Figura 40, baseia-se em uma tela de *login* contendo os campos de preenchimento de e-mail e senha, e dois botões para efetuar o *login* e para a criação de uma nova conta.

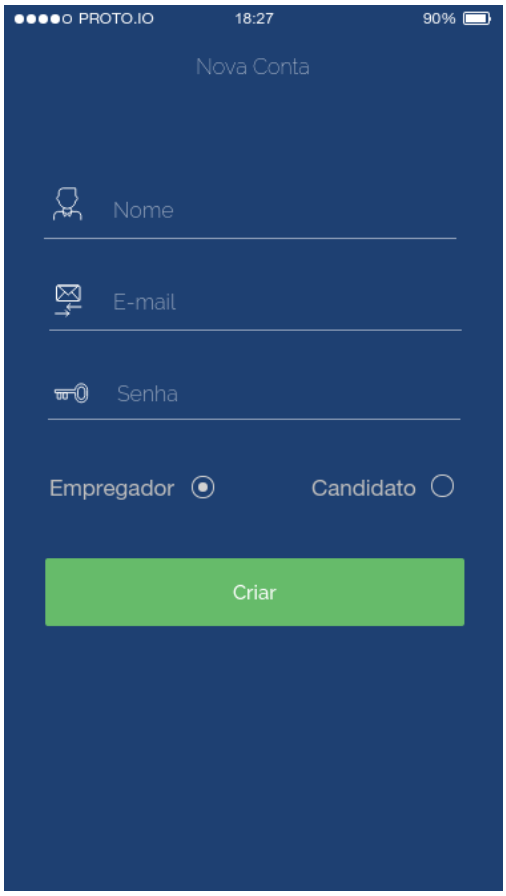
Figura 40. Protótipo de layout para a tela de *login*



Fonte: Elaborada pelo autor

A Figura 41 mostra o próximo protótipo, feito para a criação de uma interface de cadastro de novos registros, o protótipo contém campos de preenchimento referente a informações do novo usuário, e um botão que efetua a execução dos métodos.

Figura 41. Protótipo para a tela de criação de usuário



Fonte: Elaborada pelo autor

O protótipo de configurações do perfil, observado na Figura 42, foi elaborado baseado em uma lista de informações do usuário e uma imagem, que será a foto que o usuário poderá fazer *upload*. Este protótipo contém informações referente a vaga e a geração de novas vagas de emprego, caso for um usuário do tipo empregador. Foi criado também um item que efetua o *logout* da aplicação.

Figura 42. Protótipo para a tela de configurações

Fonte: Elaborada pelo autor

Todas a telas foram capazes de executar as requisições para a web API, com diversas parametrizações e uma manipulação dos dados retornados pelas *requests*. Todas estas telas possuem um ótimo fluxo de navegação utilizando a ferramenta React Navigator. Como mostrado nas figuras, todas as telas foram padronizadas em cores, fontes e nomenclaturas, assim deixando a interface do usuário mais amigável e intuitiva.

5. CONSIDERAÇÕES FINAIS

Conforme apresentado no presente trabalho, a utilização de uma web API feitas em NodeJS, juntamente com um aplicativo em React Native, é simples de ser implementada. Tendo em vista que a maioria das tecnologias que utiliza-se para criar web API são mais acopladas e têm uma curva de aprendizado mais vantajosa. A utilização dessas tecnologias, juntamente com suas dependências e ferramentas vem crescendo todos os dias no mercado, possuindo grandes comunidades que são especializadas em aprimorar as suas funcionalidades.

Neste trabalho foi focado em estudar as tecnologias e escolher a melhor forma para que se possa desenvolver a criação sistemas diferentes implementados utilizando apenas JavaScript. A criação do primeiro sistema se baseia em uma web API RESTful, capaz de efetuar todas as validações das informações recebidas e de se conectar a um banco de dados não relacional baseado em documentos JSON, chamado MongoDB. A utilização de bancos não relacionais vem aumentando consideravelmente no mercado, a sua utilização neste projeto foi de grande valia para aprimorar os conhecimentos do autor em como se comportam aplicações feitas utilizando esse tipo de banco de dados.

A segunda parte deste projeto foi o início de um aplicativo feito utilizando uma tecnologia capaz de ser executada em diferentes dispositivos, com a ajuda do Expo, ferramenta capaz de gerar *build* através de um *client* sem a necessidade de usar uma IDE para efetuar esse processo. Esta aplicação foi responsável por fazer requisições para a web API, juntamente com a utilização do *Postman*, capaz de efetuar as requisições antes mesmo do aplicativo ter sido inicializado. A utilização de uma tecnologia criada baseada em JavaScript foi um grande facilitador no processo de desenvolvimento, pois é simples e possuem diferentes bibliotecas disponíveis para serem utilizadas.

Por fim, conclui-se que a união dessas duas aplicações foram feitas seguindo todas as melhores formas de se desenvolver utilizando as tecnologias estudadas na fundamentação teórica deste projeto. Assim, concluindo o objetivo principal de criar uma forma de facilitar a comunicação entre possíveis empregados e empregadores, que possuem uma vaga de emprego disponível. Os desenvolvimentos das duas aplicações, além de todos os estudos e conhecimentos obtidos durante o processo de desenvolvimento desse projeto, foram de grande valia para que o

autor possa seguir utilizando esses conhecimentos em projetos futuros e possivelmente se especializando na área de desenvolvimento de aplicativos móveis utilizando JavaScript.

5.1 TRABALHOS FUTUROS

Os protótipos das telas foram todos elaborados, porém o que não elaborou-se em tempo de criação desse projeto foram as integrações entre todas as *views* e os recursos do *back-end*. Embora não tenha sido possível concluir todo o projeto conforme previsto inicialmente no planejamento do presente trabalho, foi possível desenvolver por completo o serviço da aplicação web API. Deixo como trabalho futuro a realização desse *review* dos códigos fontes, para que seja realizado da melhor maneira possível todas as interações entre as camadas e entre o fluxo de navegações e fluxo de requisições. Ademais, para a concretização deste trabalho, várias tecnologias web foram estudadas em paralelo ao desenvolvimento do mesmo. Sendo assim, acredita-se que agregou-se novos conhecimentos e foi possível concluir este trabalho de forma satisfatória.

6. REFERÊNCIAS

Repositório Digital da Universidade de Carnegie Mellon. Instituto de Engenharia de Software. Disponível em: <https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=513807>. Acessado em: 15 de abril de 2018.

SANTOS, Isaias et al. **Possibilidades e limitações da arquitetura mvc (model –view – controller) com ferramenta ide (integrated development environment).**2010. 56f. Trabalho de Conclusão de Curso (Graduação em Ciências daComputação) - Universidade José do Rosário Vellano, Alfenas, Mg.

BURBECK, Steve. **Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)** - Disponível em: <http://www.dgp.toronto.edu/~dwigdor/teaching/csc2524/2012_F/papers/mvc.pdf>. Acessado em: 16 de abril de 2018.

DEVELOPER NETWORK. **Model-View-Controller** - Disponível em: <https://msdn.microsoft.com/en-us/library/ff649643.aspx>. Acessado em: 16 de abril de 2018.

SHAW, M.; GARLAN, D. **Software Architecture. Perspectives on an Emerging Discipline**. Editora: Prentice Hall, 1996.

SHAW, M.; GARLAN, D.**An introduction to software architecture**. Publicado por: World Scientific, 1994.

VERGILIO, Silvia Regina. **Arquitetura de Software**. Disponível em: <http://www.inf.ufpr.br/andrey/ci163/IntroduzArquiteturaAl.pdf>. Acessado em:

BASTOS, Daniel. **O que é Model-view-controller (MVC)?**. 2009. Disponível em: < https://www.oficinadanet.com.br/artigo/desenvolvimento/o_que_e_model-view-controller_mvc>. Acesso em: 7 de Junho de 2013.

CHAN, Iana. **O que é front-end e back-end?**. 2016. Disponível em: <https://www.programaria.org/o-que-e-front-end-e-back-end/>. Acessado em: 29 de Março de 2018.

CABRAL, Carlos. **React Native: Construa aplicações móveis nativas com JavaScript**. 2016. Disponível em: <https://*tab*leless.com.br/react-native-construa-aplicacoes-moveis-nativas-com-javascript/>. Acessado em: 6 de maio de 2018.

CÂMARA, Rafael. **O que você deve saber sobre o funcionamento do React Native**. 2018. Disponível em: <https://*tab*leless.com.br/o-que-voce-deve-saber-sobre-funcionamento-react-native/>. Acessado em: 6 de maio de 2018.

MONTEIRO, Filipe. **React Native: Webview ou realmente nativo?**. 2017. Disponível em: <https://medium.com/nutripad/react-native-webview-ou-realmente-nativo-4e30a37ae020>. Acessado em: 6 de maio de 2018.

BATTISTELLI, Juliana. **Os principais conceitos de back-end para começar a desenvolver para web**. Disponível em: <https://blog.mastertech.tech/tecnologia/os-principais-conceitos-de-back-end-para-comecar-desenvolver-para-web/>. Acessado em: 7 de maio de 2018.

CHINAGLIA, Larissa. **Como começar em web back-end do zero?**. Disponível em: <http://blog.geekhunter.com.br/como-comecar-em-web-back-end-do-zero/>. Acessado em: 7 de maio de 2018.

VIANA, Daniel. **O que é front-end e back-end?**. 2017. Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-front-end-e-back-end/>. Acessado em: 7 de maio de 2018.

SANTOS, Guilherme. **Node.js—O que é, por que usar e primeiros passos**. 2016. Disponível em: <https://medium.com/thdesenvolvedores/node-js-o-que-%C3%A9-por-que-usar-e-primeiros-passos-1118f771b889>. Acessado em: 9 de maio de 2018.

SOAREZ, Jhonathan. **O que é MongoDB e porque usá-lo?**. 2016. Disponível em: <https://codigosimples.net/2016/03/01/o-que-e-mongodb-e-porque-usa-lo/#prettyPhoto>. Acessado em: 30 de maio de 2018.

DAKAR, Romulo. **MongoDB – o que é e para que serve?**. 2017. Disponível em: <http://desenvolvedor.ninja/mongodb-o-que-e-e-para-que-serve/>. Acessado em: 30 de maio de 2018.

MEDEIROS, Hygor. **Introdução ao MongoDB**. 2014. Disponível em: <https://www.devmedia.com.br/introducao-ao-mongodb/30792>. Acessado em: 30 de maio de 2018.

LEFF, A; RAYFIELD, J. **Web-Application Development Using the Model/View/Controller Design Pattern.** 2001. Disponível em: <https://domino.watson.ibm.com/library/cyberdig.nsf/papers/696CFBA5D4B1E68985256A1E00626E27/$File/rc22002.pdf>. Acessado em: 14 de julho de 2018.

GROVE, R; OZKAN, E. **THE MVC-WEB DESIGN PATTERN.** 2011. Disponível em: <http://www.scitepress.org/Papers/2011/32969/32969.pdf>. Acessado em: 15 de julho de 2018.

BAPTISTELLA, Adriano José. **Abordando a arquitetura MVC, e Design Patterns: Observer, Composite, Strategy.** Disponível em: <http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx> Acessado em: 10 de agosto de 2018.

DAVIS, Ian. **What Are The Benefits of MVC?.** 2008. Disponível em: <http://blog.iandavis.com/2008/12/what-are-the-benefits-of-mvc/>. Acessado em: 10 de agosto de 2018.

PICOLI, Cristian. **SAIBA O QUE É ARQUITETURA DE SOFTWARE E QUAL A SUA IMPORTÂNCIA.** 2018.Disponível em: <https://blog.atmdigital.com.br/saiba-o-que-e-arquitetura-de-software-e-qual-a-sua-importancia/>. Acessado em: 10 de agosto de 2018.

MARQUES, Keise de Leone. **Back-end vs Front-end vs Full-Stack: qual é a melhor escolha?.** 2017. Disponível em: <https://becode.com.br/back-end-front-end-full-stack/>. Acessado em: 12 de agosto de 2018.

KUPKA, Fernando. **O que é React Native?.** 2017. Disponível em: <https://www.organicadigital.com/seeds/o-que-e-react-native/>. Acessado em: 12 de agosto de 2018.

COLARES, Thiago. **Conceitos básicos do flexbox.** 2018. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/CSS/CSS_Flexible_Box_Layout/Conceitos_Basicos_do_Flexbox>. Acessado em: 12 de agosto de 2018.

OLIVEIRA, Igor. **React Native é mesmo nativo?.** 2017. Disponível em: <http://programadorbr.com/blog/react-native-e-mesmo-nativo/>. Acessado em: 13 de agosto de 2018.

FERNANDES, Diego. **Expo: o que é, para que serve e quando utilizar?.** 2018. Disponível em: <https://blog.rocketseat.com.br/expo-react-native/>. Acessado em: 13 de agosto de 2018.

MENIYA, Arvind. **Next Generation Mobile Application in Cloud Computing using RESTful Web Services.** 2012. Disponível em: <https://pdfs.semanticscholar.org/b1b1/6034d0a9e9cdf176c3b1ddf081dc30a8ea66.pdf>. Acessado em 25 de setembro de 2018.

GOSS, Bruna. **Node.js: por que você deve conhecer essa tecnologia?.** 2017. Disponível em: <https://www.treinaweb.com.br/blog/node-js-por-que-voce-deve-conhecer-essa-tecnologia/>. Acessado em: 25 de setembro de 2018.

DIAS, Emílio. **Desmistificando REST com Java.** Publicado por: AlgaWorks, 2016.

O QUE É Node.js. 2016. Disponível em: <http://nodebr.com/o-que-e-node-js/>. Acessado em 27 de setembro de 2018.

6 MOTIVOS para usar Node.js. 2014. Disponível em: <https://udgwebdev.com/6-motivos-para-usar-nodejs/>. Acessado em: 05 de outubro de 2018.

FELIX, Waldyr. **5 motivos para começar a usar Node.js em seus projetos HOJE.** 2016. Disponível em: <https://waldyrfelix.com.br/5-motivos-para-come%C3%A7ar-a-usar-node-js-em-seus-projetos-hoje-678f4db7b93>. Acessado em: 05 de outubro de 2018.

PILLOU, Jean-François. **O que é um banco de dados.** 2017. Disponível em: <https://br.ccm.net/contents/65-bancos-de-dados>. Acessado em: 10 de outubro de 2018.

NODEJS e MongoDB – Introdução ao Mongoose. 2016. Disponível em: <http://nodebr.com/nodejs-e-mongodb-introducao-ao-mongoose/>. Acessado em 11 de outubro de 2018.

HOROCHOVEC, Stefan. **Webtask + mLab + Auth0.** 2016. Disponível em: <https://horochovec.com.br/webtask-mlab-auth0-b2636b2fb74f>. Acessado em: 11 de outubro de 2018.

SOMMERVILLE, Ian. **Software engineering.** Editora: Prentice Hall, 2011.

INTRODUÇÃO Express/Node. 2018 . Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introdu%C3%A7%C3%A3o>. Acessado em: 18 de outubro de 2018.

EIS, Diego. **Flexbox – Organizando seu layout**. 2012. Disponível em: <https://*tab*leless.com.br/flexbox-organizando-seu-layout/>. Acessado em 20 de outubro de 2018.

MILFONT, Christiano. **JSX, a resposta do React pra resolver definitivamente um problema**. 2016. Disponível em: <https://medium.com/@milfont/jsx-a-resposta-do-react-pra-resolver-definitivamente-um-problema-99f572316eb5>. Acessado em: 20 de outubro de 2018.

MATRAKAS, Miguel. **Web Services REStful**. 2016. Disponível em: <https://www.devmedia.com.br/introducao-a-web-services-restful/37387>. Acessado em 20 de outubro de 2018.

ROZLOG, Mike. **REST e SOAP: Usar um dos dois ou ambos?**. 2013. Disponível em: <https://www.infoq.com/br/articles/rest-soap-when-to-use-each>. Acessado em: 20 de outubro de 2018.

DANTAS, Daniel Chaves Toscano. **Simple Object Access Protocol (SOAP)**. 2007. Disponível em: <https://www.gta.ufrj.br/grad/07_2/daniel/>. Acessado em outubro de 2018.

DUARTE, Luiz. Boas práticas com MongoDB. 2017. Disponível em: <https://blog.umbler.com/br/boas-praticas-com-mongodb/>. Acessado em: 21 de outubro de 2018.

BOAGLIO, Fernando. **MongoDB: Construa novas aplicações com novas tecnologias**. 2015. Disponível em: <https://books.google.com.br/books?id=bWmCCwAAQBAJ&hl=pt-BR&source=gbs_navlinks_s>. Acessado em 21 de outubro de 2018.

BELLAVER, Thomas Milton. **Padrões de Arquiteturas MVC, MVP e Pipeline**. 2017. Disponível em: <http://micreiros.com/padroes-de-arquiteturas-mvc-mvp-e-pipeline/>. Acessado em: 21 de outubro de 2018.

RODRIGUES, Joel. 20-. Disponível em: <http://www.linhadecodigo.com.br/artigo/3712/testando-servicos-web-api-com-postman.aspx> . Acessado em: 21 de outubro de 2018.

PEREIRA, Fernanda. **HTML, CSS e Javascript – Entendendo melhor a base da programação Front-End.** 2018. Disponível em: <http://apexensino.com.br/html-css-e-javascript-entendendo-melhor-base-da-programacao-front-end/>. Acessado em: 21 de outubro de 2018.

HELLER, Martin. **What is Node.js? The JavaScript runtime explained**. 2017. Disponível em: <https://www.infoworld.com/article/3210589/node-js/what-is-nodejs-javascript-runtime-explained.html>. Acessado em: 22 de outubro de 2018.

MÜTSCH, Ferdinand. **Http performance Java (Jersey) vs. Go vs. NodeJS**. 2016. Disponível em: <https://ferdinand-muetsch.de/http-performance-java-jersey-vs-go-vs-nodejs.html>. Acessado em: 22 de outubro de 2018.