



GOVERNO DO ESTADO DO RIO DE JANEIRO
SECRETARIA DE ESTADO DE CIÊNCIA E TECNOLOGIA
FUNDAÇÃO DE APOIO À ESCOLA TÉCNICA
CENTRO DE EDUCAÇÃO PROFISSIONAL EM TECNOLOGIA DA INFORMAÇÃO
FACULDADE DE EDUCAÇÃO TECNOLÓGICA DO ESTADO DO RIO DE JANEIRO
FAETERJ/PETRÓPOLIS

Tecnologias de desenvolvimento de aplicações em tempo real

João Pedro Souza Homem

Petrópolis - RJ

Março, 2017

João Pedro Souza Homem

***Tecnologias de desenvolvimento de aplicações em
tempo real***

Trabalho de Conclusão de Curso apresentado à
Coordenadoria do Curso de Tecnólogo em Tec-
nologia da Informação e Comunicação da Fa-
culdade de Educação Tecnológica do Estado do
Rio de Janeiro Faeterj/Petrópolis, como requi-
sito parcial para obtenção do título de Tecnó-
logo em Tecnologia da Informação e Comuni-
cação.

Orientador:

Marlan Kulberg

Petrópolis - RJ

Março, 2017

Folha de Aprovação

Trabalho de Conclusão de Curso sob o título “*Tecnologias de desenvolvimento de aplicações em tempo real*”, defendida por João Pedro Souza Homem e aprovada em 6 de Março de 2017, em Petrópolis - RJ, pela banca examinadora constituída pelos professores:

Prof. MSc. Marlan Kulberg
Orientador

Prof. MSc. Victor Thomaz
Faculdade de Educação Tecnológica do Estado
do Rio de Janeiro Faeterj/Petrópolis

Prof. MSc. Douglas Ericson Marcelino de
Oliveira
Faculdade de Educação Tecnológica do Estado
do Rio de Janeiro Faeterj/Petrópolis

Declaração de Autor

Declaro, para fins de pesquisa acadêmica, didática e técnico-científica, que o presente Trabalho de Conclusão de Curso pode ser parcial ou totalmente utilizado desde que se faça referência à fonte e aos autores.

João Pedro Souza Homem
Petrópolis, em 6 de Março de 2017

Agradecimentos

Agradeço a todos que de forma direta ou indireta contribuíram para a elaboração deste trabalho através do investimento em minha capacitação, compartilhamento de conhecimento, disponibilização de ferramentas e componentes para desenvolvimento de *softwares*, entre outros não menos importantes.

Dentre todos os relacionados a este agradecimento, gostaria especialmente de agradecer a toda comunidade *open-source*, da qual pude usufruir de diversos recursos que viabilizaram a implementação da aplicação abordada neste documento. A iniciativa *open-source* abre muitos caminhos para a inovação e inclusão na área de desenvolvimento de *softwares* e deve ser incentivada e reconhecida, atribuindo-se os devidos méritos a cada um que adere a iniciativa em seus projetos.

Epígrafe

"Faça-se simples com ideias complicadas"

(JoPaseho)

Resumo

A evolução da tecnologia vem trazendo para as nossas vida diversas inovações que estão mudando a forma de lidar com inúmeras situações. Alcançamos novos patamares em como tratamos os negócios, relações pessoais, a busca por novos conhecimentos, entre outras coisas. Tudo isso nos levou a buscar entender um tipo de aplicação que tem se tornado cada vez mais presente nas ferramentas que utilizamos.

As aplicações de tempo real são o propósito da realização deste trabalho. Primeiramente, buscamos no mercado algumas referências de aplicações para embasar o desenvolvimento de uma aplicação de *chat* por localização geográfica. Nas referências, encontram-se diversos aspectos que foram considerados para a escolha de como e quais tecnologias usar para construir as funcionalidades.

Tendo como base todas as possibilidades para a implementação, o problema foi delimitado em construir uma ferramenta com tecnologias modernas e utilizadas no mercado por aplicação com fins semelhantes ao nosso, e que demonstrasse o funcionamento elemental da transmissão de dados em tempo real na *web*.

Sendo assim, subdividimos o desenvolvimento em duas partes principais, que são a aplicação servidor e cliente.

- A aplicação servidor desempenhará o papel de receber as informações da aplicação cliente, processá-las, e distribuí-las para os outros clientes através de uma conexão bidirecional, que nos permite implementar uma comunicação de tempo real entre os diversos usuários da aplicação.
- A aplicação cliente, resumidamente pode ser considerada a interface com a qual os usuários interagem com a aplicação. Através dela, o usuário poderá encontrar, por um mapa terrestre, um outro que esteja utilizando a aplicação no mesmo momento, que será identificado como um ponto marcado no local em que os dados de geolocalização coletados pelo servidor informam. Após a localização do usuário, este poderá ser contactado através de uma janela de comunicação por texto.

A arquitetura utilizada foi inspirada em alguns padrões que moldam a infraestrutura da aplicação para privilegiar ações de escalabilidade, portanto apresentamos parte desse esquema implementado em nosso sistema, e uma outra parte complementar de forma teórica.

Palavras-chave Aplicação de tempo real. Chat por localização. Escalabilidade. NodeJS. SocketIO. Redis. MongoDB. AngularJS. Google Maps.

Abstract

The evolution of technology has brought to our lives several innovations that are changing the way to deal with many situations. We reached new heights in how we treat business, personal relationships, the search for new knowledge, among other things. All this led us to seek to understand a type of application that has become increasingly present in the tools we use.

The real-time applications are the subject of this work. First of all, we seek some applications's references to support us with the development of a geographic location based chat. We found in our references several aspects that were considered for the choice of how and which technologies to use to build the functionalities.

Based on all the possibilities that we had to implement, we delimited the problem of building a tool with modern technology and used in the market for application purposes similar to ours, and to demonstrate the elemental operation of real-time data transmission in the web.

Thus, the development was subdivided into two main parts, which are the application server and client.

- The application server will play the role of receiving the client application information, process them, and distribute them to other clients through a bidirectional connection that allows us to implement a real-time communication between the application's users.
- The client application can be briefly considered the interface which users interact with the application. Through it, you can find, by a terrestrial map, another user who is using the application at the same time, it will be identified as a marked point where the geolocation data collected by the server report. After found the user location, it can be contacted by a text communication window.

The architecture used was inspired by some standards that shape the application infrastructure to favor scalability actions, therefore we present part of this scheme implemented in our system, and another complementary part theoretically.

Keywords Real-time Application. Chat by location. Scalability. NodeJS. SocketIO. Redis. MongoDB. AngularJS. Google Maps.

Lista de Figuras

2.1	Interface principal do Uber Mobile. Fonte: Referral SaaSquatch [5]	p. 17
2.2	Interface de pagamento do Uber Web. Fonte: Referral SaaSquatch [5]	p. 18
2.3	Interface principal do FanMappr. Fonte: Captura de tela	p. 20
2.4	Interface principal do RadiusIM Fonte: wwwwhats new [11]	p. 21
2.5	Janela de <i>chat</i> do RadiusIM Fonte: wwwwhats new [11]	p. 22
4.1	Visão geral do Sublime Text 3. Fonte: Captura de tela	p. 29
4.2	Visão geral do Visual Studio Code. Fonte: Captura de tela	p. 30
4.3	Fluxo simplificado de requisições no NodeJS. Fonte: maxroecker [32]	p. 31
4.4	Conexão entre servidor e cliente com SocketIO. Fonte: ifollow [36]	p. 34
4.5	Localização do Nginx na arquitetura de aplicações <i>web</i> . Fonte: Thunder-Boy [50] . .	p. 40
5.1	Estrutura de arquivos do projeto. Fonte: Captura de tela	p. 42
5.2	Representação UML (Unified Modeling Language) dos modelos persistidos no banco de dados.	p. 44
5.3	Demonstração de utilização do MongoDB pelo terminal do CentOS.	p. 45
5.4	Fluxo de acesso aos recursos com OAuth2. Fonte: jlabusch [59]	p. 46
5.5	Página principal. Fonte: Captura de tela	p. 51
5.6	Página de acesso ao sistema. Fonte: Captura de tela	p. 52
5.7	Menu lateral. Fonte: Captura de tela	p. 52
5.8	Janelas de <i>chat</i> . Fonte: Captura de tela	p. 53
5.9	Representação da arquitetura do sistema. Fonte: Autoria própria	p. 54

Lista de Abreviaturas

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheet

DOM Document Object Model

HAML HTML Abstraction Markup Language

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

I/O Input/Output

JRE Java Runtime Environment

JSON JavaScript Object Notation

JVM Java Virtual Machine

MEAN MongoDB, ExpressJS, AngularJS, NodeJS

MVW Model-View-Whatever

NPM Node Package Manager

NoSQL Not Only Structured Query Language

PUB/SUB Publish/Subscription

REST Representational State Transfer

RHEL Red Hat Enterprise Linux

RPM Red Hat Package Manager ou RPM Package Manager

SASS Syntactically Awesome Stylesheets

SEO Search Engine Optimization

SPA Single Page Application

SSL Secure Sockets Layer

UI User Interface

UML Unified Modeling Language

UTC Coordinated Universal Time

UX User Experience

XML Extensible Markup Language

Sumário

1	Introdução	p. 14
1.1	Motivação	p. 14
1.2	Objetivos	p. 15
2	Trabalhos Relacionados	p. 16
2.1	Uber	p. 16
2.2	Redis Publish-Subscribe (RedisMVA)	p. 18
2.3	Redis Pub-Sub (Rajaraodv)	p. 19
2.4	FanMappr e RadiusIM	p. 20
2.5	Snapchat, Happn, Tinder e afins	p. 22
3	Problematização	p. 24
3.1	Caracterização do Problema	p. 24
3.2	Solução Proposta	p. 25
4	Hardware e Software Utilizados	p. 27
4.1	Codificação e Testes	p. 27
4.1.1	Ambiente de Desenvolvimento	p. 27
4.1.2	Versionamento de Código	p. 28
4.1.3	Editores de Texto	p. 28
4.2	Back-end	p. 30
4.2.1	NodeJS	p. 30
4.2.2	NPM	p. 32

4.2.3	ExpressJS	p. 32
4.2.4	SocketIO	p. 33
4.2.5	MongoDB	p. 34
4.2.6	Redis	p. 35
4.3	Front-end	p. 36
4.3.1	Bower	p. 36
4.3.2	AngularJS	p. 36
4.3.3	Angular Material	p. 37
4.3.4	Gulp	p. 37
4.3.5	SASS	p. 38
4.3.6	HAML	p. 38
4.3.7	Webpack	p. 39
4.3.8	Babel	p. 39
4.3.9	Nginx	p. 39
5	Implementação	p. 41
5.1	Backend	p. 42
5.1.1	Persistência	p. 44
5.1.2	Autenticação e Autorização	p. 45
5.1.3	Rotas	p. 47
5.1.4	Socket	p. 47
5.2	Frontend	p. 49
5.3	Resultados	p. 50
6	Conclusão	p. 55
6.1	Trabalhos Futuros	p. 55
	Referências	p. 57

1 Introdução

Os avanços da tecnologia em conjunto com a internet têm ditado aos seus usuários novas formas de consumo de dados e informações. Além disto, um assunto tem ganhado cada vez mais destaque no provimento de dados na rede: a periodicidade. A inclusão dos meios tecnológicos no cotidiano das pessoas e o grande aumento do volume de dados gerados têm desenvolvido uma nova tendência de informações instantâneas, o que nos leva a elaborar meios para atender a esta nova demanda.

Alguns questionamentos aos modelos atuais de consumo de dados, apesar de bem sucedidos em diversas áreas, surgem como uma necessidade ainda maior pelo consumo de informações. O *e-mail*, tradicional meio de comunicação entre pessoas, oferecendo uma boa razão de confiabilidade e formalidade, já é questionado por *startups* [1] [2] que acreditam no futuro da *web* em tempo real.

1.1 Motivação

As aplicações de tempo real estão surgindo como requisitos de muitos dos novos sistemas em desenvolvimento, e buscam atender consumidores que já possuem em seus negócios a necessidade da informação imediata. Neste trabalho iremos desenvolver uma aplicação que explora de forma bem específica essa necessidade da troca de informações entre usuário e aplicação em tempo real.

A aplicação desenvolvida será um *web chat* de comunicação por texto baseado em localização geográfica. As duas principais características do projeto, comunicação por texto e localização geográfica, trazem para si aspectos modernos, que exigem a utilização de ferramentas e técnicas atuais de desenvolvimento para a *web*. Deve-se levar em consideração diversos conceitos, como compatibilidade, escalabilidade, infraestrutura de sistemas, entre outros. Tudo isto serve como motivação para atualização e aprendizado de diversos tópicos relacionados ao desenvolvimento de *softwares*.

1.2 Objetivos

A finalidade do projeto será demonstrar como pode ser feita a transferência de dados em tempo real entre servidores e clientes. Para isto, será feita a implementação de uma aplicação que irá realizar a comunicação por texto entre usuários, que serão localizados para comunicação através de localização geográfica em um mapa terrestre real.

O produto mínimo viável da aplicação será elaborado para um ambiente de testes local, não oferecendo um fluxo completo de funcionamento dos servidores em ambiente de produção, nem do fluxo de utilização de um usuário comum em sua interface. Abordaremos apenas alguns conceitos sobre escalabilidade e infraestrutura da aplicação em ambiente de produção, assim como apenas algumas funcionalidades necessárias para interação na interface gráfica.

Será explicado com mais clareza os problemas que acercam este projeto no tópico "Problematização", abordando especificamente quais são os ideais de implementação de uma solução em tempo real que tem características semelhante a desenvolvida neste trabalho, levando em conta aspectos como produto e serviço, já que ideologicamente um software é feito para realizar uma ou mais funções para um ou mais usuários, mas como citado, o foco é demonstrar fundamentalmente o funcionamento deste tipo de aplicação, e não oferecer um produto ou serviço. A problematização irá se perfazer em torno do estudo de uma forma de se implementar uma solução de tempo real para a *web* envolvendo duas funcionalidades de representatividade para a circunstância.

2 *Trabalhos Relacionados*

Com o objetivo de situar o presente trabalho em relação ao funcionamento da arquitetura e funcionalidades do sistema, analisamos alguns trabalhos que se assemelham em um ou mais aspectos da aplicação desenvolvida. Dentre estas, estão relacionadas propostas de implementação, que explicam o funcionamento da arquitetura de aplicações de tempo real, e também projetos que foram publicados como produto de mercado.

Será feita uma breve análise de cada uma das plataformas, sendo que algumas análises serão fundamentadas em sites que falam sobre o funcionamento destas, pois as mesmas não se encontram mais disponíveis para acesso. Contudo, são ferramentas que tentaram como produto final uma aplicação muito semelhante a desenvolvida como caso de estudo neste trabalho.

2.1 **Uber**

Iniciando a pesquisa sobre o desenvolvimento de uma aplicação de tempo real, foi buscada uma referência, que segundo estatísticas [3], é considerada uma aplicação de grande escala. O Uber [4] é uma aplicação *web* e *mobile* que oferece serviços de transporte urbano privado através de pessoas que se disponibilizam a serem motoristas e estarem disponíveis na plataforma para alguma oferta de demanda.

Encontrou-se um site [6] que fez-se de grande valia para o início da escolha das tecnologias que utilizamos para desenvolver a aplicação deste trabalho. A publicação fala sobre como era a arquitetura do sistema, para quais possibilidades ela foi pensada, e sobre a nova organização do sistema e quais tecnologias utilizam para suportar tamanha demanda [3].

É notório ao longo do trabalho que a arquitetura aplicada para desenvolvimento da aplicação não se aproxima em muitos pontos da aplicada no Uber, mas após o estudo da publicação, fomos fomentados a pesquisar algumas das tecnologias utilizadas e como aplicá-las em nosso projeto. A principal influência retirada deste projeto foi a utilização do Redis como banco de dados em memória, e não diretamente uma influencia, mas uma consolidação de ideias, a utilização do

NodeJS e um banco de dados NoSQL (Not Only Structured Query Language) para persistência dos dados.

As imagens 2.1 e 2.2 mostram brevemente a interface do Uber. Na imagem 2.1 podemos observar a interface principal, e é possível notar a utilização do mapa, o qual nestes são contidos os carros próximos a uma localização, estes com suas posições geográficas atualizadas em tempo real devido ao fato de estarem se movimentando. Estas características representam bem algumas funcionalidades que serão implementadas na aplicação.

A imagem 2.2 mostra a aplicação *web* do Uber. Nesta é possível fazer o gerenciamento das contas de usuário em um formato mais cômodo e adequado para análise dos dados gerados através da utilização do aplicativo *mobile*, como é mostrado na imagem 2.2 um histórico detalhado de uma corrida, com trajetória realizada, horário, preços, entre outros.

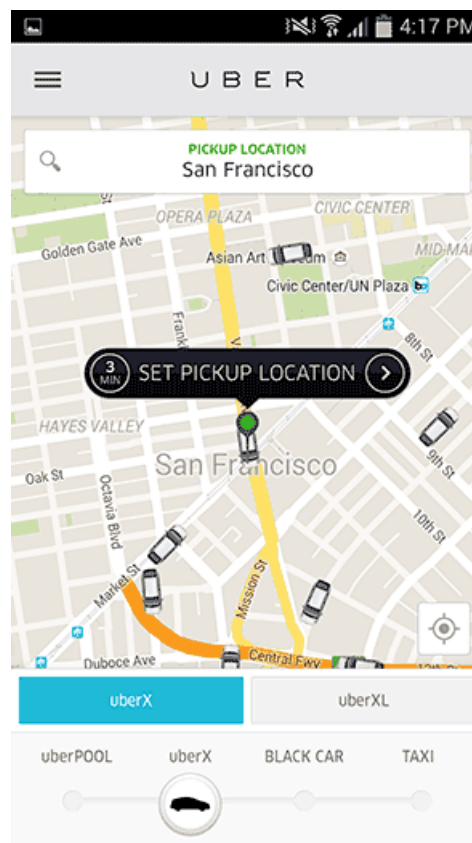


Figura 2.1: Interface principal do Uber Mobile. Fonte: Referral SaaSquatch [5]

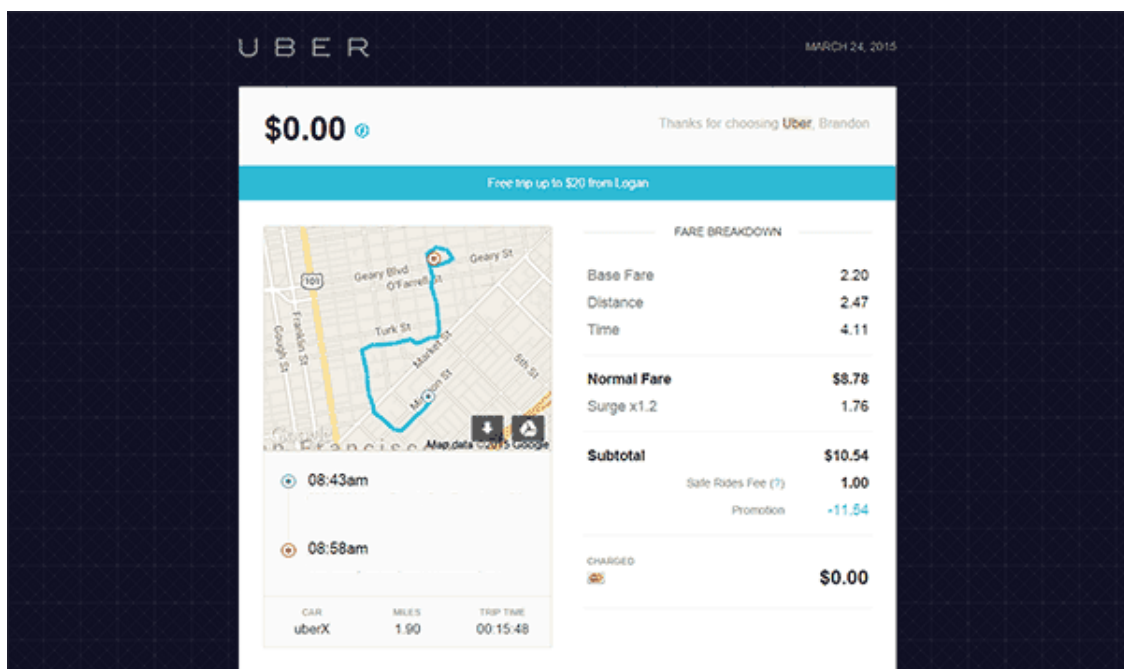


Figura 2.2: Interface de pagamento do Uber Web. Fonte: Referral SaaSquatch [5]

2.2 Redis Publish-Subscribe (RedisMVA)

Esta publicação [7] faz parte de um projeto demonstrativo da utilização do padrão de comunicação pub/sub (Publish/Subscription), com um banco de dados NoSQL, chamado Redis. É abordada a implementação em NodeJS de um *chat* de comunicação por texto, utilizando o padrão pub/sub para realizar a comunicação entre processos, onde os processos são as diversas instâncias que podem ser executadas de um mesmo servidor NodeJS distribuídos em diversos locais na internet.

O estudo desta publicação serviu como grande base de aprendizado para a elaboração da arquitetura do sistema que desenvolvemos, pois utilizava tecnologias atuais e que já tínhamos como plano para a implementação do nosso sistema, como citamos no caso do Uber. A utilização da linguagem de programação JavaScript, naturalmente traz um certo intimismo para quem já desenvolve para a *web*, dado que ela pode ser considerada a linguagem da *web* (*client-side*), pois todos os maiores navegadores *web* [8] utilizados no mundo interpretam a linguagem nativamente, sendo muito comum programadores voltados para a área já terem tido algum contato com a linguagem.

O autor da publicação disponibiliza todo o código fonte do projeto de demonstração com licença aberta, sendo assim, ele discorre sobre o tema de forma muito objetiva, apenas abordando em seu texto os pontos de interseção em que a implementação da comunicação pub/sub

se insere. O texto oferece uma didática teórica de boa qualidade, demonstrando com diagramas e imagens da aplicação real, o funcionamento de cada passo do qual explica no documento.

A demonstração dos fluxos de dados que ocorrem no sistema ficam bem representadas com os diagramas, embora alguns destes fluxos não estejam implementados no sistema, como o balanceador de carga, visto que não era o foco de esclarecimento. É simples entender como a interface envia os dados para o servidor, que distribui os dados para os outros clientes conectados.

2.3 Redis Pub-Sub (Rajaraodv)

Esta publicação [9] fala sobre o mesmo assunto da citação anterior, que é a implementação de um chat por texto utilizando NodeJS e banco de dados Redis, mas nesta publicação o autor é mais abrangente no quesito de infraestrutura e escalabilidade da aplicação, abordando todos os passos para uma implantação completa do sistema em um servidor.

O texto da publicação é muito explicativo, é possível até para os mais principiantes no assunto de aplicações de tempo real entenderem a implementação e a finalidade de cada passo relatado. O texto também conta com ilustrações dos fluxos de dados na aplicação e demonstrações do funcionamento da interface.

O que se pode extrair do conteúdo desta publicação além da anterior, é a implementação do balanceador de carga, em conjunto com o *proxy* reverso Nginx (aplicação que controla como será feito o fluxo de dados de fora para dentro da rede de servidores), o conceito de *sticky sessions* (técnica que grava nas requisições de um cliente informações sobre servidor de origem da sessão) para traçar requisições a um mesmo servidor, e a integração do sistema em um servidor na nuvem (conjunto de recursos computacionais na internet, dedicados a hospedar aplicações de diversos tipos).

Após o estudo da publicação deste autor, utilizamos em nossa implementação a utilização do Nginx. Entendemos a ideia de centralizar a distribuição de conteúdo estático da aplicação, como a interface, e canalizar as conexões aos servidores através de um ponto em comum, facilitando assim o gerenciamento para futuras implementações de *sticky sessions*, balanceamento de carga, reconexão em casos de *scale-up* ou *scale-out* dos servidores (aumento e diminuição do número de servidores), entre outros benefícios que este tipo de arquitetura pode prover para as aplicações modernas. Tudo isso aspira o rumo em que a tecnologia vem tomando com o crescimento exponencial de usuários, consumo de dados, globalização, entre outros, que contribuem para a necessidade de se construir sistemas que sejam distribuídos e escaláveis, alcançando altos

desempenhos.

2.4 FanMappr e RadiusIM

O FanMappr [10] e o RadiusIM [11] são aplicações baseadas em *chat* por localização geográfica, mas que foram disponibilizadas como produto, com a proposta de criarem relações sociais através da busca de usuários por localização, ou seja, são redes sociais e tinham isso como proposta final. Após uma análise mais técnica nos dois últimos tópicos em relação ao desenvolvimento da aplicação, agora iremos analisar ferramentas que implementam como um produto, aplicações que tem funcionalidades semelhantes as do nosso projeto.

Encontramos o FanMappr e RadiusIM após algumas poucas pesquisas que buscavam especificamente produtos que tivessem funcionalidades muito próximas das que implementamos em nossa aplicação. O FanMappr é uma aplicação com interface e funcionalidades aparentemente modestas, navegando entre os poucos menus que possui, percebemos que as funcionalidades se restringem somente ao *chat* por texto e a um mapa terrestre onde os usuários são localizados.

A imagem 2.3 é uma captura de tela da aplicação *web* do FanMappr. Nela é possível notar a simplicidade da aplicação, que não possui personalizações das ferramentas utilizadas, como o mapa, e não utiliza conceitos de responsividade, pois a captura de tela foi realizada em um monitor de 22 polegadas, e a interface da aplicação se manteve no mesmo formato utilizado para tela menores.

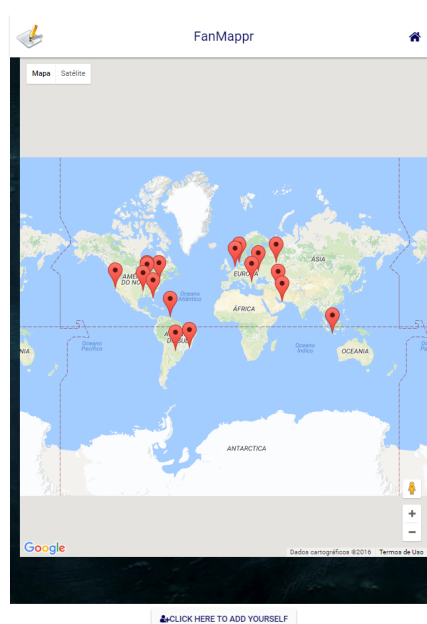


Figura 2.3: Interface principal do FanMappr. Fonte: Captura de tela

O RadiusIM não está mais disponível para acesso, mas encontramos um artigo [11] que possuía algumas imagens sobre a aplicação e uma breve explicação do seu funcionamento.

Foi possível notar que o RadiusIM também é muito semelhante com o FanMappr, apesar de parecer ter tido mais sucesso, nos anos 2000, presume-se que teve um investimento maior, pois possuía uma interface mais harmoniosa e funcionalidades que eram o de se esperar para sua época.

Podemos observar nas imagens 2.4 e 2.5 alguns detalhes de sua interface que demonstram um pouco da utilização do sistema. A imagem 2.4 mostra que de forma amigável ao usuário, a interface fornece os dados de sua localização, marcação no mapa, e uma série de pessoas nas proximidades e suas respectivas imagens de perfil. Na imagem 2.5 podemos ver a janela de chat



Figura 2.4: Interface principal do RadiusIM Fonte: wwwwhats new [11]



Figura 2.5: Janela de *chat* do RadiusIM Fonte: [www.whats new](http://www.whatsnew.com.br) [11]

É interessante perceber que estas aplicações não tiveram grande sucesso colocando como produto final uma ferramenta de comunicação com funcionalidades semelhantes a que iremos implementar, o que nos leva a interpretar que a *web* em tempo real não é uma inovação a qualquer custo. É importante considerar que as melhorias que se pode ter com informações em tempo real devem ser mensuradas, e ela é apenas mais uma opção que está ganhando mais importância com as demandas que estão surgindo.

2.5 Snapchat, Happn, Tinder e afins

Neste tópico iremos fazer uma análise, mais filosófica do que técnica, sobre a utilização de aplicações de tempo real na vida dos usuários, pois achamos interessante fundamentar onde estão se rompendo os limites para a necessidade de aplicações deste tipo, e gostaríamos de ressaltar que Snapchat [16], Happn [18] e Tinder [17] foram escolhas que achamos que representam um grupo de aplicações com características próximas, mas que para muitas outras aplicações existentes, caberiam-se a mesma análise.

A seguir foi feita a síntese de uma citação [12] que nos traz algumas importantes informações sobre a opinião de um usuário comum:

"Há uma diferença fundamental entre chat baseado em localização e aplicativos de chat,

como o Snapchat. O Snapchat mudou como as pessoas se comunicam, não mudou como elas encontram as pessoas para se comunicar.

A localização em si não define quem somos ou o que gostamos, as pessoas buscam se comunicar com pessoas que possuam outros tipos de afinidades, e estas é que devem ser exploradas para se criar um aplicativo que seja interessante."

(Síntese de "Do location based chat apps start trending again?"[12], NOTNOTCITRICS-QUID, 2012)

A resposta a pergunta do tópico expressa uma opinião que condiz com alguns pontos que pudemos notar ao longo do conteúdo apresentado até aqui, que se relacionam com como as aplicações tem obtido sucesso na inserção da experiência de tempo real na vida dos usuários mais comuns, e aplicações como o FanMapp e o RadiusIM não alcançaram o mesmo resultado. Snapchat, Happn e Tinder são aplicações que utilizam localização geográfica em muitas de suas funcionalidades, mas trabalham a experiência do usuário com este tipo de funcionalidade de uma forma que preserva conceitos como os abordados na citação anterior, e este é um ponto que acreditamos ser o diferencial para a aceitação das ferramentas deste tipo, pelo fato de não infringir algumas barreiras que existem na interação humana com o mundo virtual.

3 *Problematização*

3.1 Caracterização do Problema

A internet oferece diversas possibilidades de integração e distribuição de dados em sua rede mundial. A necessidade cada vez maior de se coletar dados que são gerados a todo momento, cria a iniciativa de se laborar novas maneiras de envio e recepção de dados na rede.

A transmissão de dados em tempo real não é uma necessidade recente, desde que se começaram a desenvolver sistemas críticos de controle na *web*, muitos destes já utilizavam desta forma de se transmitir dados para realizar suas tarefas. As técnicas e tecnologias utilizadas para alcançar tal feito eram diferentes e mais custosas, o que levava este tipo de característica da aplicação ser adotada apenas em casos específicos, onde deveria se contrabalancear com mais perícia a utilização.

Com os avanços das tecnologias de *hardware*, *software* e da internet como um todo, começaram a se criar novas possibilidades da implementação de *features* de tempo real em aplicações cada vez mais próximas do usuário comum. Com tudo isso, podemos observar que muitas das aplicações de sucesso do mercado usufruem desta tecnologia, que permite uma conexão direta e persistente entre cliente e servidor, viabilizando que se possa receber e enviar dados a qualquer momento, atualizando em tempo real todos os interessados em uma determinada informação.

A possibilidade da informação instantânea tem muito a contribuir em diversas áreas, oferecendo vantagem competitiva para quem está no mundo dos negócios, antecipação de tomada de decisões, entre outros benefícios. Este mundo do tempo real está a cada vez mais tomando conta do nosso presente como usuários comuns, e a tendência é que essa forma de compartilhamento instantâneo se torne mais natural.

Como as pessoas utilizam e disponibilizam dados, tem uma importância fundamental no impacto em que isso pode ter em suas vidas. Desde as necessidades no mundo dos negócios, às necessidades do cotidiano de cada um. Por parte de quem está inovando no setor, é preciso entender comportamentos e como não romper barreiras que irão fracassar a relação entre

tecnologia e pessoas.

A implementação de funcionalidades de tempo real definitivamente se tornou algo exequível para as aplicações atuais na *web*, como *push notifications* (notificações de eventos que são emitidas do servidor para o cliente), *stats* (status de dados), interações em jogos de navegadores, entre outros. Os *WebSockets* funcionam analogicamente como os *sockets*, e nos permite fazer a comunicação entre processos através da rede. Como o próprio nome diz, os *WebSockets* são especificamente feitos para este tipo de comunicação na *web*, possibilitando uma conexão contínua entre códigos JavaScript executados nos *browsers* e os servidores web. Estes serão base fundamental para a resolução do problema proposto neste trabalho, levando em conta que são a tecnologia mais atual.

3.2 Solução Proposta

Buscando explorar a usabilidade do tempo real na *web*, iremos formular nossa solução no desenvolvimento de uma aplicação de comunicação por texto, também podendo ser chamada de *chat*. Para que os usuários tenham conhecimento da existência um do outro na aplicação, existirá um mapa terrestre do mundo real que será marcado com pontos identificadores de cada usuário utilizando o sistema. Estes pontos terão como base a coordenada geográfica real do usuário que será fornecida quando o usuário entrar no sistema.

A escolha por uma aplicação deste modelo se deve a ideia de querer tornar o entendimento do emprego da tecnologia mais amigável para as pessoas que não possuem conhecimento técnico muito aprofundado. A troca mensagens explícita em uma conversa entre duas pessoas, essencialmente ilustra a ideia de transferência de dados em tempo real, e em conjunto com a localização geográfica, conseguimos dar um adendo de outra aplicação da tecnologia.

Iremos desenvolver uma interface com a pretensão de ser amigável ao usuário, utilizando conceitos de responsividade para a adaptação do conteúdo mostrado em telas de dispositivos diversos, inclusive móveis, tendo em vista que algumas estatísticas [19] apontam que dispositivos móveis já são responsáveis por pelo menos um quarto do todo o tráfego de dados gerado na internet, e com estimativas [20] [21] de aumento em cada ano. Mesmo não sendo o foco da aplicação ser um produto final, achamos que o aprendizado com este tipo de requisito seria conveniente. Em relação a organização da interface, a aplicação terá duas telas principais, sendo a primeira com as opções de cadastro através do fornecimento de alguns dados pessoais, e autenticação no sistema por *e-mail* e senha. A segunda tela será o ambiente principal da aplicação, onde o usuário poderá navegar por um mapa terrestre em busca de marcações que identifiquem

outro usuário, com o qual ele poderá interagir através de uma janela de *chat* por texto que surge neste mesmo ambiente.

O principal componente da solução será o servidor da aplicação. Este será desenvolvido sobre a arquitetura REST (Representational State Transfer), que transmite e recebe dados da interface no formato JSON (JavaScript Object Notation). Funcionalidades como cadastro de usuário e autenticação farão uso deste modelo. A comunicação em tempo real será provida por um *framework* que utiliza como base a tecnologia de WebSockets para conexões bidirecional de dados entre servidor e cliente. Apesar da tecnologia WebSocket ser a mais apropriada para este tipo de conexão, o próprio *framework* trata de usar outras técnicas de comunicação em tempo real, como *long polling*, em casos de falta de compatibilidade dos navegadores com a tecnologia de WebSocket. Tudo isso acontece de forma transparente ao usuário e ao desenvolvedor, pois a utilização do *framework* se dá através de API (Application Programming Interface) que oferece de forma simplificada todo o processo de conexão e comunicação.

A persistência dos dados será feita através de um banco de dados no NoSQL. A escolha por este tipo de banco de dados se deve as características da aplicação que exigem muitas operações de I/O (Input/Output) em dados com baixo nível de relacionamento com outros dados, onde este tipo de tecnologia NoSQL oferece melhor desempenho e praticidade.

A tecnologia utilizada como ambiente de execução no servidor, utiliza o JavaScript como linguagem padrão, além de oferecer características que também beneficiam o tipo de aplicação que iremos desenvolver, como *single-threaded* (aplicação executada em apenas uma thread, processando um comando por vez), *non-blocking IO* (operações de escrita e leitura de dados não bloqueiam a execução de código) e *asynchronism* (não existe ordem de execução dos blocos de código).

Considerando também alguns conceitos de infraestrutura, iremos adotar na arquitetura da aplicação algumas formas de permitir o escalonamento dos serviços através da distribuição dos servidores em diversas instâncias no mesmo servidor físico ou em outros de locais distintos. Para isso utilizaremos tecnologias para fazer a comunicação entre processos, com um banco de dados NoSQL centralizado e em memória, e interações *stateless*, onde o servidor não guarda dados de sessão do usuário.

4 *Hardware e Software Utilizados*

4.1 Codificação e Testes

4.1.1 Ambiente de Desenvolvimento

O ambiente para desenvolvimento da solução foi criado em um único computador, de caráter pessoal, do tipo notebook, com as seguintes especificações técnicas [22]:

- Processador Intel Core i5 3230M - 2.60 GHz (T-Max: 3.20 GHz) - 64 Bits - Smart Cache 3 MB L3
- Chipset Intel ® HM75 Express
- Memória RAM 8 GBs DDR3 SDRAM - 1600 MHz
- Disco rígido 500GB - 5.400 RPM
- Rede 10/100 e Wireless 802.11 b/g/n - Link de conexão com a internet de 15 Mbps

Especificamos apenas algumas informações que julgamos serem de relevância para alguma análise de desempenho da aplicação por parte do leitor. Em nossa solução não iremos realizar levantamentos de dados sobre desempenho, pois este não pode ser considerado um ambiente que simule a realidade em servidores de aplicações, do qual seria de relevância este tipo de medição.

O sistema operacional escolhido para nosso ambiente de desenvolvimento foi o CentOS 7. O CentOS é uma distribuição Linux baseada no Red Hat [23]. A Red Hat é uma empresa privada, que possui um sistema operacional chamado Red Hat Enterprise Linux (RHEL), e tem como proposta oferecer produtos *enterprise* baseados no GNU/Linux. Tendo em conta que toda distribuição baseada no *core* do Linux deve seguir uma série de regras contidas em sua licença,

a Red Hat concentra seus faturamentos em suporte as distribuições que ela construiu. Ter um sistema compatível com o RHEL traz alguns benefícios que ainda não foram alcançados sem que se tenha um time de pessoas dedicadas ao desenvolvimento de um *software* específico, como no caso da maioria dos sistemas open-sources, que é o alto padrão de qualidade em testes contra erros e problemas de segurança, *releases* mais curtos, correções de problemas em curto prazo, entre outros benefícios que o CentOS consegue absorver da comunidade Red Hat. Além disso, o CentOS já possui uma considerável comunidade ativa, que facilita na resolução de problemas mais específicos e no desenvolvimento de novas funcionalidades. O sistema também já possui grandes *cases* de sucesso, como Facebook [24] e Google [25], o que passa uma maior confiança para a adoção em outros negócios e uma perspectiva de crescimento contínua.

4.1.2 Versionamento de Código

O versionamento de código é uma tecnologia que permite traçar históricos de mudanças nos arquivos de código escrito através de um gerenciamento multi-usuário. Este gerenciamento permite que uma ou mais pessoas possam modificar um ou mais arquivos simultaneamente ou não, sendo controlados através de funcionalidades que permitem junção, revisão de conflitos, entre outros.

A tecnologia utilizada para versionamento de código neste projeto foi o Git. Este é um sistema de versionamento distribuído que atende a diversos tipos de *workflows* de trabalho. A escolha por esta tecnologia se deve ao fato de ser umas das mais difundidas no mercado, e onde se encontram os maiores e mais confiáveis repositórios de código. Embora o fluxo de desenvolvimento tenha sido muito simples, utilizamos um *workflow* chamado *Gitflow Workflow* [26] que divide o desenvolvimento basicamente em dois *branches* (subdivisões de versionamento de código) principais, um de produção e outro de desenvolvimento, podendo ser criado *branches* de cada *feature* sobre o *branch* de desenvolvimento.

4.1.3 Editores de Texto

Para o desenvolvimento majoritário do *frontend*, utilizamos o editor de texto Sublime Text 3 [27], apoiado de alguns *plugins* para auxílio de escrita e design, todos encontrados no repositório de pacotes packagecontrol.io [28]. A imagem 4.1 oferece uma visão geral do Sublime Text 3 com o tema Material Theme aplicado a sua interface, onde existe ao lado direito uma área que mostra a estrutura de diretórios do projeto e seus arquivos, e a esquerda o editor de texto propriamente dito, com *syntax highlighting*.

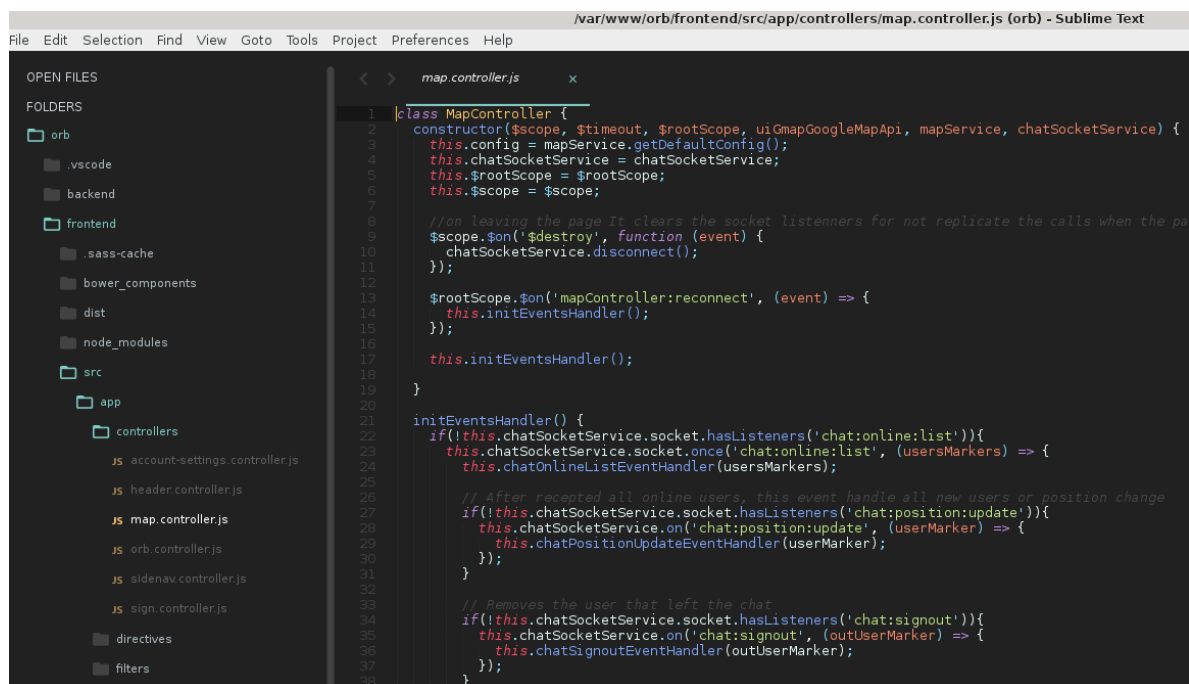


Figura 4.1: Visão geral do Sublime Text 3. Fonte: Captura de tela

Para o desenvolvimento do *backend* e parte do *frontend*, utilizamos o editor de texto Visual Studio Code 1.6 [29]. A escolha por este outro editor de texto se fez principalmente pela funcionalidade de *debug*, que apresentou melhores resultados em sua utilização, além de ser uma funcionalidade nativa da aplicação, assim como a ferramenta para versionamento de código. Também foi utilizado *plugins* semelhantes aos escolhidos no SublimeText para auxílio de escrita de código, todos encontrados no repositório de pacotes padrão do editor.

A imagem 4.2 mostra a visão geral do Visual Studio Code. A distribuição dos componentes se assemelham as do Sublime, mas é possível notar na imagem uma barra na extremidade esquerda, que possui alguns componentes nativos do editor, como o componente de listagem de arquivos e diretórios, pesquisa, controle de versão (GIT), *marketplace*, e principalmente a sua ferramenta de debug, que foi um dos principais motivos para adoção desse editor, como citado anteriormente.

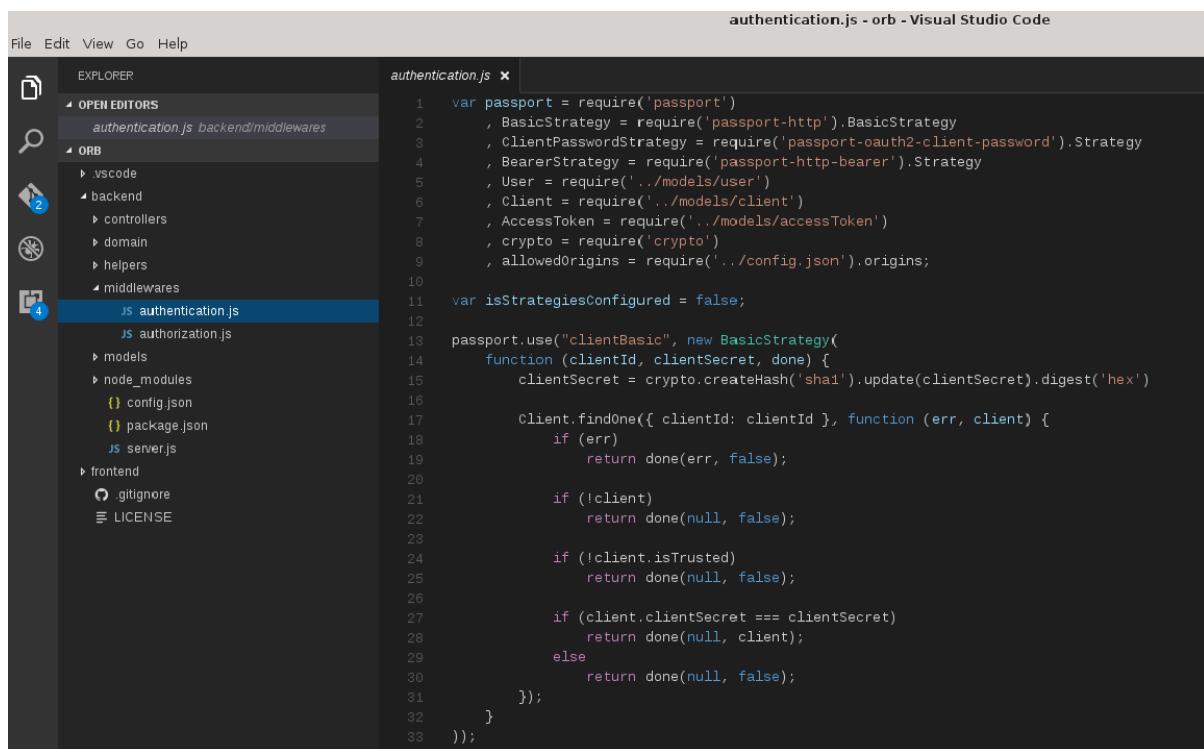


Figura 4.2: Visão geral do Visual Studio Code. Fonte: Captura de tela

4.2 Back-end

4.2.1 NodeJS

O NodeJS [30] é uma plataforma de desenvolvimento de alta performance construída sobre a *Chrome's Javascript runtime V8* [31]. A V8 é um interpretador Javascript *open source* desenvolvido em C++ pela Google e capaz de executar em sistemas operacionais Windows, Linux e iOS, consequentemente trazendo esta possibilidade de multi plataforma para o NodeJS. Comparando-se a uma tecnologia que está a mais tempo no mercado e se possui um maior entendimento, o funcionamento da *engine V8* é superficialmente semelhante ao fluxo de execução do conjunto JVM (Java Virtual Machine) e JRE (Java Runtime Environment) do Java.

A plataforma NodeJS possui uma arquitetura de funcionamento *single-threaded* com operações orientadas a eventos e I/O não-bloqueantes. A escolha pela linguagem Javascript se molda bem a este tipo de funcionamento, pois por ser uma linguagem dinâmica e trazer características da programação funcional, como *first-class functions* e *function composition*, que auxiliam fortemente no controle do fluxo de execução através *callbacks*, tornam a programação assíncrona mais fácil na plataforma, contribuindo para a criação de sistemas escaláveis e distribuídos.

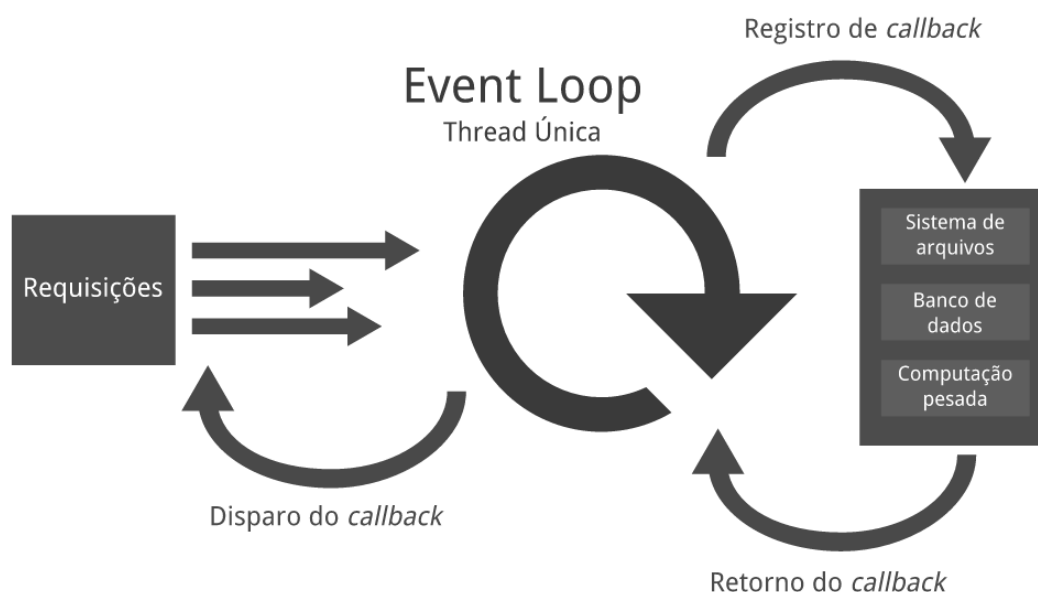


Figura 4.3: Fluxo simplificado de requisições no NodeJS. Fonte: maxroecker [32]

Como se pode observar na figura 4.3, diferentemente dos servidores *web* comuns, o NodeJS não cria um processo ou *thread* para cada requisição que chega ao servidor. Ao invés disso, cada instância NodeJS é um processo que funciona como um *event loop*, onde para cada operação de I/O requisitada, é registrada uma *callback* que será chamada posteriormente no *event loop* quando os dados estiverem disponíveis. Sendo assim, a *thread* principal nunca fica bloqueada, ela está sempre aguardando que um evento esteja para acontecer, que pode ser a chegada ou resposta de uma requisição, por exemplo. É possível que surjam questionamentos sobre como o NodeJS faz as requisições de I/O assíncronas e simultaneamente atende outras requisições, mas isso é abstraído ao desenvolvedor, pois o NodeJS utiliza uma biblioteca multi-plataforma, codificada em C, chamada libuv [33], que trata as operações de I/O assincronamente com utilização de *thread pool* e afins.

Tendo em mente todos esses conceitos do funcionamento da arquitetura do NodeJS, é possível entender o porque dele conseguir tratar muito mais conexões simultâneas do que as arquiteturas convencionais, pois a criação de processos e *threads* tende a ser um recurso custoso para o sistema operacional, além de reservas de recursos valiosos do sistema, como memória RAM. É importante ressaltar que a programação assíncrona e o ambiente criado pelo NodeJS tem uma curva de aprendizado maior e se faz necessário maiores cuidados por parte do desenvolvedor. Uma simples implementação incorreta pode acarretar no travamento da única *thread* da instância da aplicação, bloqueando totalmente o sistema de processar novas requisições.

Por outro lado, quando o entendimento é claro e o desenvolvimento é correto, a utilização

dos recursos naturalmente é escalável e distribuída. A própria singularidade da plataforma nos faz trabalhar conceitos de sistemas distribuídos, como por exemplo, para que possamos aproveitar de todos os recursos de um processador multi-core com NodeJS, temos que executar mais de uma instância da aplicação, o que nos leva a trabalhar conceitos de clusterização, comunicação de processos e etc. Tudo isso facilita o escalonamento horizontal da aplicação em múltiplos servidores, conceito importante para aplicações de alto desempenho.

4.2.2 NPM

O NPM (Node Package Manager) [34] é o gerenciador de pacotes do NodeJS. A função do NPM é oferecer um repositório central de qual se pode baixar diversos módulos distribuídos pela própria equipe que desenvolve o NodeJS ou que outros usuários tenham contribuído.

A evolução do NPM se tornou dominante, sendo considerado o maior repositório de registro de conteúdo Javascript do mundo. Devida a utilização do Javascript para outras finalidades além do desenvolvimento em NodeJS, muitos dos módulos disponibilizados pelo NPM não são parcialmente ou completamente desenvolvidos para a utilização com o NodeJS, sendo que muitos usuários e desenvolvedores utilizam e publicam conteúdo voltado para o desenvolvimento *frontend*, que também utiliza o Javascript como principal linguagem. Talvez este seja um dos grandes motivos de sucesso do NPM em números.

Sua utilização é feita basicamente via linha de comando, onde iniciamos um novo repositório quando criamos uma aplicação, e apartir deste comando é criado um arquivo em formato JSON, chamado `package.json`, do qual ficará registrado todas as dependências daquele projeto, através da especificação do nome e versão de cada módulo baixado do repositório oficial.

A utilização de um gerenciador de pacotes traz diversas vantagens, dentre estas, a facilitação no versionamento de código. Isto traz o benefício de não ser necessário manter uma cópia de um conteúdo que poderá ser baixado posteriormente de um local central, e o controle das versões de cada biblioteca que se está utilizando no sistema. Assim facilitando a resolução de problemas de compatibilidade e dependências uma das outras, pois é um local único para busca e contribuição de soluções na linguagem.

4.2.3 ExpressJS

O ExpressJS [35] é o principal *framework* NodeJS para desenvolvimento de aplicações *web* e APIs. A sua proposta é ser simples, performático e *loosely coupled* na construção de aplicações, sem interferir nas características padrões que o NodeJS oferece.

O desenvolvimento com o ExpressJS é bem intuitivo pela organização que ele propõe para a aplicação. Suas abstrações são apenas essenciais, que não interferem na utilização dos artifícios da própria linguagem nativa do NodeJS para a resolução dos problemas, sem muitas automatizações, sendo ainda mais conveniente oferecendo mecanismos de *middlewares* que possibilitam total manipulação das requisições e seus atributos, injeção de *template-engines* e outros *handlers* e *plugins* que podem ser baixados via NPM. A comunidade possui diversas soluções empregadas, o que também facilita na curva de aprendizado, produtividade e resolução de problemas. Rapidamente é possível instalá-lo na aplicação e se disponibilizar um servidor HTTP (Hypertext Transfer Protocol) pronto para receber, tratar e responder requisições como API ou UI (User Interface).

4.2.4 SocketIO

O SocketIO é uma biblioteca para a criação de aplicações de tempo real na *web*, que permite troca de mensagens em formato binário ou texto de forma bi-direcional e instantânea. Ele funciona através de duas bases de código, uma aplicada no servidor NodeJS e outra no *browser* do cliente. A biblioteca faz uso da tecnologia WebSocket para estabelecer uma comunicação bi-direcional e persistente entre servidor e cliente, mas em casos de problemas de compatibilidade, é utilizado o método de *long polling*, que sempre mantém uma requisição aguardando por uma nova mensagem do servidor, e com isso consegue alcançar este tipo de comunicação. As bibliotecas do SocketIO utilizadas no servidor e no cliente possuem chamadas semelhantes, o que facilita na realização da interação dos dois lados.

A figura 4.4 demonstra simplificada a forma de conexão entre servidor e cliente descrita anteriormente. No lado esquerdo da figura é representado uma aplicação NodeJS com um servidor *websocket* (socket.io) em funcionamento, e as conexões bi-direcionais representadas pelas setas que ligam ao lado direito da figura. Neste lado da figura está representado os clientes, que estão possivelmente executando a aplicação em seus *web browsers* com uma biblioteca javascript que realiza a conexão com o servidor através da tecnologia *websocket* por uma URL que identifica o servidor.

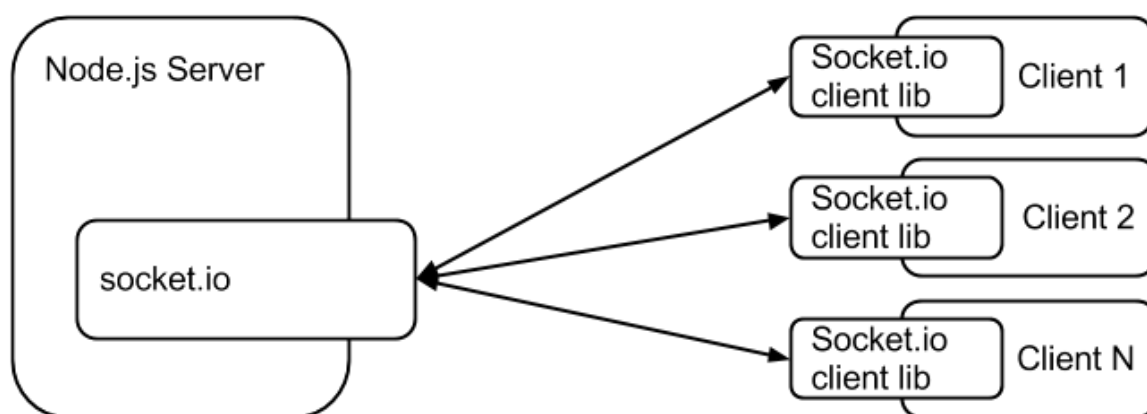


Figura 4.4: Conexão entre servidor e cliente com SocketIO. Fonte: ifollow [36]

Apesar do SocketIO ser um encapsulador da tecnologia *websocket*, as chamadas de sua API são muito semelhantes a utilização do *websocket* diretamente, onde a biblioteca apenas adiciona funcionalidades que são convenientes para o gerenciamento das conexões dos *sockets*, como *broadcast* de mensagens, atrelar informações específicas do usuário de cada *socket*, entre outros.

A tecnologia *websocket* é baseada em canais TCP, o que significa que os dados trafegados por ele tem a garantia de ordem e não perca de informações, o que torna a comunicação mais confiável e capaz de atender um abrangente número de necessidades. Também é preciso levar em conta os custos que canais TCP podem ter dependendo do tipo de dados que se quer transmitir, onde apesar da possibilidade de transmissão de conteúdo binário por *websockets*, muitas das vezes é mais eficiente a utilização de um canal UDP, como faz uso a tecnologia WebRTC, que é muito utilizada para *streaming* de vídeos, por exemplo.

4.2.5 MongoDB

O MongoDB é um banco de dados de alta performance, enquadrado nos conceitos de bancos de dados NoSQL. A escolha por este tipo de ferramenta se faz pelas características que o modelo de arquitetura escolhido traz para a aplicação. Os bancos de dados NoSQL foram criados para oferecer novas possibilidades para as demandas que exigem escalabilidade horizontal e performance na escrita e leitura de dados, o que se encaixa com os atributos das tecnologias da aplicação desenvolvida.

A forma como este tipo de banco de dados funciona remete diretamente aos ganhos de desempenho e escalabilidade, pois existe uma quebra de conceitos dos bancos de dados relacio-

nais, como o ACID (Atomicity, Consistency, Isolation, Durability), que trazem flexibilidade em como se tratam os dados armazenados para se ter os benefícios citados. O MongoDB além de adotar estes conceitos, é um banco de dados orientado a documentos, onde a persistência dos dados é realizada em formato JSON. Apesar do conceito NoSQL ser uma grande ruptura de paradigmas, a maior parte da popularidade do MongoDB se deve a sua forma de interação com o usuário, que é amigável a quem está migrando dos conceitos relacionais para o modelo NoSQL. Apesar de não existir a necessidade de criação de tabelas, definição de relações, entre outros, a API do MongoDB oferece a criação de esquemas que remetem a conceitos parecidos, e também habilitam outras possibilidades, como a indexação de campos para buscas mais eficientes. As chamadas de busca no banco também são amigáveis, utilizando palavras chaves como *SELECT*, *WHERE*, *ORDER BY*, entre outros. É importante salientar que algumas das funcionalidades citadas se devem ao uso de um modelador de dados chamado Mongoose [37], ferramenta da qual oferece um *set* de utilidades para se trabalhar com o MongoDB.

4.2.6 Redis

O Redis é um banco de dados NoSQL em memória, que implementa apenas alguns tipos de estruturas de dados, como *strings*, *hashes*, *lists*, *sets* e *sorted hashes*, e funcionalidades para gerência, busca e transmissão dos dados armazenados, o que o torna muito enxuto. Devido ao Redis ser um banco de dados que funciona em memória, ele possui uma alta performance, tendo em vista que a memória RAM é um dos componentes de armazenamento de dados mais rápidos em leitura, escrita e transmissão na arquitetura computacional. Por outro lado, a memória RAM é volátil, o que nos leva a buscar entender qual a proposta de um banco de dados em memória.

A utilização principal do Redis é para servir como *cache* e/ou *message broker* de informações em sistemas. Em implementações *stateless* o Redis pode servir como *cache* de dados muito requisitados, pois seu rápido desempenho em memória pode ser vital para atender requisições em tempo hábil. Outra utilidade do Redis, que é o caso deste projeto, é do compartilhamento de dados entre servidores distintos de uma aplicação, onde através das estruturas de dados fornecidas pelo Redis e algumas funcionalidades, como a do protocolo PUB/SUB, é possível que se armazene dados no Redis e outra instância da aplicação sendo executada em outro local possa buscar estes mesmos dados. Devido a simplicidade e robustez que o Redis possui, uma única instância é capaz de realizar milhares de operações, aproximadamente 1.5 milhões por segundo [39], podendo funcionar muito bem sendo este tipo de centralizador, pois o escalonamento de servidores de aplicação ocorre com muito mais frequência do que a de instâncias Redis.

4.3 Front-end

4.3.1 Bower

O Bower [40] é um gerenciador de pacotes, assim como o NPM, mas com conteúdo focado para o *frontend*. Através dele é possível fazer *download* de diversas bibliotecas, como AngularJS, jQuery e Bootstrap. Sua utilização se dá através de linha de comando, onde ao criar um novo projeto *frontend* o usuário deve iniciar o Bower no mesmo diretório, através de linha de comando, e este criará um arquivo chamado *bower.json*, que conterá informações do projeto e todas os nomes e versões de bibliotecas baixadas do seu repositório oficial, permitindo-se assim o gerenciamento de novos pacotes e resolução de dependências entre eles.

O NPM também oferece muitas das bibliotecas disponíveis no Bower, e o contrário muitas vezes não é válido, sendo assim, é comum a utilização dos dois gerenciadores de pacotes no projeto *frontend*, o Bower e NPM.

4.3.2 AngularJS

O AngularJS [41] é um *framework* Javascript para desenvolvimento de interfaces *web*. Mais do que apenas interfaces, o AngularJS auxilia na criação de aplicações complexas no lado cliente da comunicação. Com o avanço das tecnologias e formas de se construir sistemas, ferramentas como o AngularJS vieram para ajudar na organização e qualidade dos sistemas que estavam se desenvolvendo.

A abordagem do AngularJS na construção de aplicações *frontend* é completa, oferecendo suporte para manipulação do DOM (Document Object Model), *data binding*, gerenciamento de rotas, padrões de projeto (Model-View-Whatever, Model-View-ViewModel, Model-View-Controller), que são conceitos que visam a organização e reutilização de código, entre outros. Apesar da fragmentação de alguns recursos em módulos separados, como as rotas, o produto AngularJS consegue oferecer uma solução completa para o desenvolvimento de interfaces de usuário.

O desenvolvimento de uma aplicação AngularJS, normalmente segue as diretrizes de uma aplicação SPA (Single Page Application) com o estilo de programação declarativa, tendo em vista a forma com que se desenvolve interfaces para os *browsers*, as tecnologias envolvidas, como o próprio HTML (Hypertext Markup Language), em um ambiente orientado a eventos, se faz mais adequada este tipo de abordagem declarativa, que segue uma lógica mais comum as interfaces.

A arquitetura moldada ao um formato de API também é um fator determinante para a utilização de um *framework* como o AngularJS, pois ele já oferece módulos e serviços para o consumo de dados através de *endpoints* que forneçam dados em formato XML (Extensible Markup Language) ou JSON.

4.3.3 Angular Material

O Angular Material [42] é uma outra solução do produto Angular, também para interfaces, mas focado em conceitos e características de *design*, UX (área do desenvolvendo que trabalha por uma melhor experiência de uso da aplicação por parte do usuário), reutilização, acessibilidade, compatibilidade *cross-browsers*, responsividade, entre outros.

A ferramenta oferece diversos componentes visuais de interface com efeitos e estilizações próprios, levando em conta diversos conceitos abstratos de *design*, que enriquecem a experiência e agradabilidade do usuário ao ver e interagir com eles. Os componentes do Angular Material são totalmente compatíveis com as implementações AngularJS, facilitando na integração entre os componentes. Um grande problema que é evitado utilizando um *framework* como este, são os de problemas de compatibilidade entre os diversos navegadores no mercado, onde cada um possui especificidades que podem causar comportamentos inesperados nos comentários de interface. Acessibilidade e outras otimizações também são importantes para a inclusão do site e SEO.

Por último, um benefício importante que o Angular Material dispõe, é o seu *grid system*, baseado no CSS3 *flexbox*. Esta característica possibilita que todo o conteúdo apresentado no site se redimensione de acordo com o tamanho da tela do dispositivo que está exibindo o conteúdo, possibilitando assim uma experiência viável para dispositivos móveis, do qual já citamos a importância em números [19] [20] [21].

4.3.4 Gulp

O Gulp [43] é uma ferramenta de automatização de tarefas Javascript, comumente utilizado para automatizações de fluxos de trabalho no desenvolvimento *frontend*. Sua utilização se dá por linha de comando e um arquivo de configuração chamado `gulpfile.js`. Neste arquivo Javascript podemos criar *tasks* que irão realizar tarefas diversas, de acordo com a programação realiza, como por exemplo a execução de testes unitários, concatenação e minificação de arquivos, compilação de linguagens intermediárias, como o SASS, *autoprefixer*, *lint*, entre outros.

Existem diversos *plugins* disponíveis no NPM que executam tarefas específicas, como as

citadas anteriormente, bastando ao usuário baixá-las e chamá-las em suas *tasks*, que podem executar uma ou mais funcionalidades organizadas através de chamadas síncronas pelo método *pipe* de cada *task*.

4.3.5 SASS

O SASS [44] é uma linguagem de *script* que pode ser interpretada para CSS. O CSS tem uma sintaxe muito simples e com poucas funcionalidades para estruturação, reaproveitamento e modularização de código, onde o SASS oferece uma série de funcionalidades, como variáveis, operadores, herança e modularização, que podem ser aplicadas para a geração de folhas de estilos.

Todas as funcionalidades do SASS facilitam a construção de interfaces complexas, onde se exigem muitos componentes, classes, hierarquização de seletores, entre outros. A interpretação do código SASS para CSS pode ser feita por diversos interpretadores, inclusive o próprio Gulp oferece *plugins* para este fim.

4.3.6 HAML

O HAML (HTML Abstraction Markup Language) [45] é uma linguagem que tem a proposta de facilitar a escrita do HTML através de uma sintaxe mais curta, indentação de código e alguns conceitos de HTML dinâmico. Neste projeto utilizamos mais especificamente o HAMLJS [46], que é uma derivação da versão principal do HAML, que habitualmente é utilizado com Ruby on Rails. Essa derivação do HAML oferece uma implementação em Javascript, assim possibilitando sua utilização em diversas ferramentas que utilizam a linguagem. O funcionamento do HAML é parecido com o do SASS e a interpretação dos arquivos HAML para HTML pode ser feita por diversas ferramentas, inclusive *plugins* do próprio Gulp podem realizar esta tarefa.

Apesar da escolha de utilização de uma ferramenta transcrita de outra linguagem, existem diversas opções de *template engines* feita exclusivamente para a utilização com Javascript, como o Jade e o EJS, e que podem trazer melhor qualidade das funcionalidades e *releases* mais curtos com novas implementações.

4.3.7 Webpack

O WebPack [47] é um Javascript *module bundler*. Devido a linguagem Javascript não possuir um sistema para isolamento de escopo dos diversos blocos de código que podem existir em uma aplicação, o WebPack proporciona essa funcionalidade baseando-se nos principais padrões de módulos Javascript utilizados, como o AMD e CommonJS, onde ele é capaz de gerenciar todas as dependências entre módulos e gerar *assets* estáticos para a utilização.

A utilização do WebPack se faz principalmente no desenvolvimento *frontend*, pois o NodeJS, principal ambiente de desenvolvimento com Javascript do lado servidor, já adota o sistema de módulos CommonJS, o que aproxima ainda mais os conhecimentos de desenvolvimento *frontend* e *backend* com a adoção de módulos CommonJS nos dois lados, além dos benefícios na organização e isolamento de códigos. Também é possível a utilização de *plugins* intermediários com o WebPack, estes podem oferecer diversas funcionalidades, como interpretação de ECMAScript 2015 ou TypeScript.

4.3.8 Babel

O Babel [48] é um compilador Javascript que permite a utilização das novas funcionalidades do ES2015. Devido a esta grande atualização do Javascript não ser compatível com uma grande maioria dos navegadores do mercado, o Babel oferece a opção de se escrever o código Javascript com a sintaxe ES2015 e ter a geração de outro arquivo traduzido para o ES5, que é a implementação com maior compatibilidade entre navegadores.

A utilização do Babel pode ser feita utilizando diversas ferramentas, como o Gulp ou o WebPack. Quando se utiliza o WebPack no projeto, é preferível que se faça a integração do Babel com ele, pois como ele cuida da resolução de dependências entre os módulos, ele consegue tratar no momento certo a tradução do código em ES2015 para ES5, evitando problemas de interpretação e compatibilidade.

4.3.9 Nginx

O Nginx [49] é um servidor *web* de alta performance *single-threaded*, uma arquitetura parecida com a do NodeJS, com funcionalidades que o permitem atuar como *proxy* reverso e/ou *load balancer* de aplicações. Para demonstrar onde o Nginx se encaixa na infraestrutura de uma aplicação *web*, podemos observar a seguinte imagem:

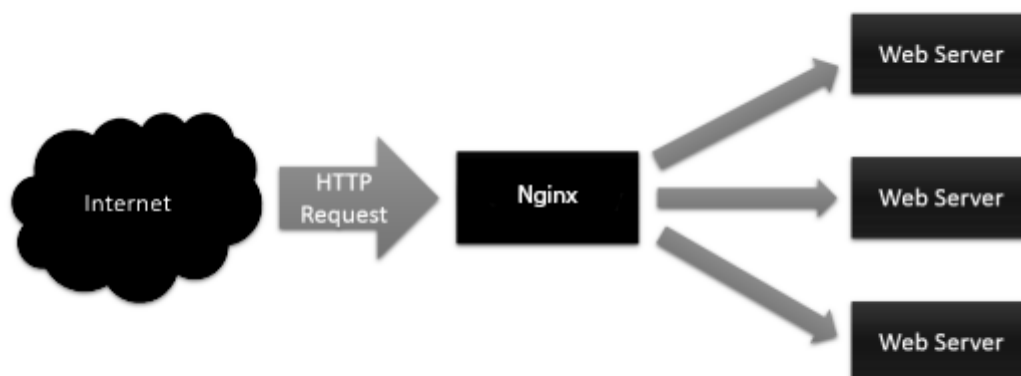


Figura 4.5: Localização do Nginx na arquitetura de aplicações *web*. Fonte: Thunder-Boy [50]

Na imagem 4.5 o fluxo representativo de dados é da esquerda para a direita, onde no exemplo é feita uma requisição HTTP através de internet para uma URL, a qual encaminha a requisição para o Nginx, o qual encaminha a requisição para um dos servidores da rede privada.

O conceito de *proxy reverso* se dá pela capacidade do Nginx encaminhar requisições de entrada para os servidores da rede privada, ao contrário do que ocorre com os *proxys* comuns, que encaminham o tráfego de saída da rede. Sendo assim, o Nginx está na frente dos servidores de aplicação e gerencia todas as conexões que chegam de um determinado domínio, possibilitando assim que também gerencie controles de *cache*, conexões SSL (Secure Sockets Layer), *streaming*, fornecimento de conteúdos estáticos da aplicação, como arquivos HTML, folhas de estilo, scripts, imagens, entre outros.

O Nginx também pode funcionar como um *load balancer*, tendo em vista que ele recebe todas as conexões e conhece todos os caminhos para os servidores, bastando apenas a implementação de um ou mais algoritmos de balanceamento de carga, como Round Robin, Least Traffic, Least Latency, URL, entre outros. Uma funcionalidade importante que também pode ser implementada com a ajuda do Nginx é o conceito de *sticky sessions*, que por exemplo, é uma opção para o funcionamento de conexões *websockets* em aplicações distribuídas em múltiplos servidores, devido a necessidade de se manter as requisições traçadas ao mesmo servidor que contém suas informações de conexão.

5 *Implementação*

A implementação do sistema segue todas as diretrizes de tecnologias e conceitos apresentados até este tópico. A descrição de como a implementação foi feita será dividida em duas partes principais: *backend* e *frontend*. Sendo a parte de *backend* a implementação da infraestrutura e servidor da aplicação, e na parte do *frontend* a implementação de interface com o usuário. No desenvolvimento elegemos um nome fictício para a aplicação, chamado Orb. Este nome será utilizado para remeter a aplicação em alguns textos, logos, implementações, entre outros.

O código-fonte da aplicação está hospedado no GitHub [51], como *software open-source*, liberado para utilização através da licença MIT [52]. A abordagem utilizada para a descrição da implementação feita fará referências pontuais a trechos do código-fonte que poderão ser consultados pelo leitor no repositório informado [51], evitando-se assim a exposição de código neste formato de texto que não é favorável a análise deste tipo de conteúdo.

A figura 5.1 mostra a estrutura de diretórios e como alguns arquivos da aplicação real estão alocados neles. A esquerda está representada a estrutura de diretórios da aplicação backend, seguindo o modelo de estruturação do ExpressJS e o diretório *domain*, criado para alocação de lógica de negócio da aplicação. A direita da imagem é representado a estrutura de diretórios do *frontend*, assim como a da aplicação Angular, que segue o padrão MVC.



Figura 5.1: Estrutura de arquivos do projeto. Fonte: Captura de tela

5.1 Backend

Primeiramente iremos descrever a criação do ambiente de desenvolvimento do projeto, onde inicialmente foi instalado o sistema operacional e todas as ferramentas que decidimos utilizar para a construção da aplicação, e posteriormente iniciamos o processo de codificação.

Iniciando-se pelo sistema operacional, foi criada uma partição separada no disco rígido, com volume de 60 GBs. O sistema operacional escolhido para o ambiente de desenvolvimento foi de uma distribuição Linux chamada CentOS [54], versão 7. A instalação do sistema operacional foi feita através de um *pendrive* e foram seguidas todas as configurações padrões do *wizard* de instalação do sistema.

Após a instalação do sistema operacional, foi feita a instalação de todas as ferramentas que foram escolhidas para o desenvolvimento da aplicação, como o NodeJS, MongoDB, Redis, edi-

tores de texto e navegadores. Todas estas ferramentas foram instaladas através do repositório de pacotes padrão do sistema CentOS 7, chamado YUM. Para algumas ferramentas foram necessário a atualização das fontes do repositório YUM e outras foram instaladas através de pacotes RPM (Red Hat Package Manager ou RPM Package Manager) disponibilizados no próprio site da ferramenta.

A implementação do *backend* iniciou-se com a criação de um diretório chamado backend na pasta raiz do projeto, do qual a partir dele foram seguidas recomendações de boas práticas [53] para a organização da estrutura de diretórios em aplicações ExpressJS. A seguinte listagem identifica o nome destes diretórios e suas respectivas funcionalidades:

controllers Este diretório contém os arquivos de código relacionados as rotas e suas lógicas de implementação. Rotas são os *endpoints* que podem ser acessados através de requisições HTTP enviadas por uma determinada URL que a identifica. Neste diretório contém os arquivos das rotas de cliente, usuário e *token*.

domain Este diretório contém os arquivos relacionados a lógica de negócio da aplicação. Os *handlers* que tratam as requisições *websocket* da aplicação estão implementados em arquivos contidos nesta pasta.

helpers Este diretório contém arquivos que implementam funcionalidades que podem ser utilizadas em diversos pontos do sistema, este diretório funciona como um *cross-cutting layer* da aplicação. Nesta pasta estão implementados alguns métodos que ajudam na geração de *random strings* e retorno de *datetime* em formato UTC (Coordinated Universal Time).

middlewares Este diretório contém arquivos relacionados a *middlewares* de tratamento de requisições que chegam ao servidor, como realização de autenticação e autorização de acesso as rotas da aplicação.

models Este diretório contém arquivos relacionados aos objetos que podem ser persistidos na aplicação. Devido ao Javascript ser uma linguagem dinâmica, estes objetos não representam um contrato de como estes eles devem existir na aplicação, mas como eles devem ser persistidos no banco de dados, que no caso desta aplicação é o MongoDB. Neste diretório serão encontrados os arquivos de persistência dos *tokens*, *chat*, *messages*, *users*, entre outros.

Após a definição da estrutura de diretórios da aplicação, foi realizada a instalação e inicialização do NPM, responsável por gerenciar todos os pacotes que iremos utilizar no desenvolvimento da solução.

5.1.1 Persistência

A persistência dos dados foi feita em um banco de dados MongoDB, onde para sua utilização foi feito o *download* do pacote Mongoose pelo NPM. A conexão com o banco foi feita no arquivo de inicialização da aplicação, através do método `connect` do próprio Mongoose, passando-se como parâmetros uma *connection string* com o endereço local do banco de dados. Por ser um ambiente de testes, não utilizamos autenticação com o banco de dados, mas caso fosse necessário, bastava-se apenas adicionar os dados de *username:password* na própria *connection string*.

No diretório *models* foram criados arquivos para cada objeto que fosse ser persistido no banco, como ilustra a imagem 5.2. As implementações da persistência utilizaram o Mongoose como modelo, exportando de cada módulo o objeto *schema* de cada persistência, podendo através deste realizar todas as operações de CRUD (Create, Read, Update, Delete).

A figura 5.2 ilustra a representação dos objetos persistidos no banco de dados:

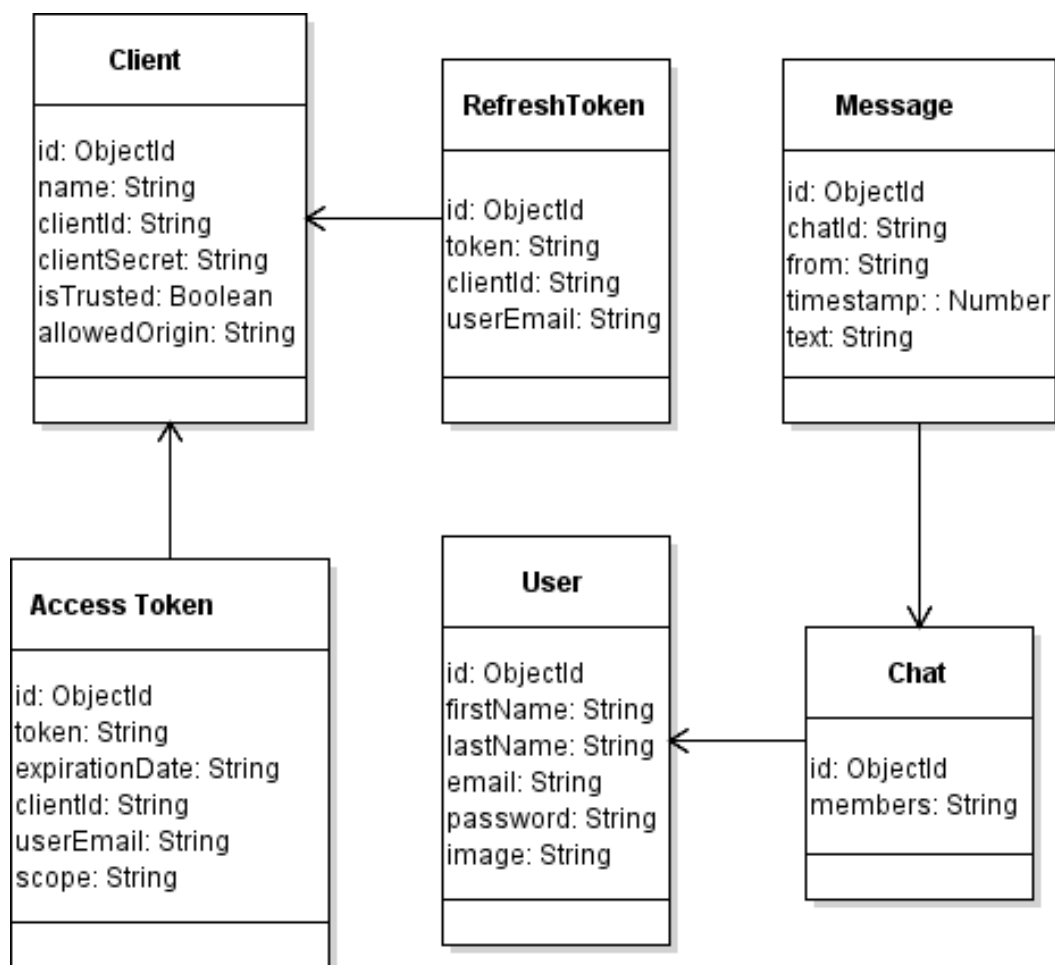
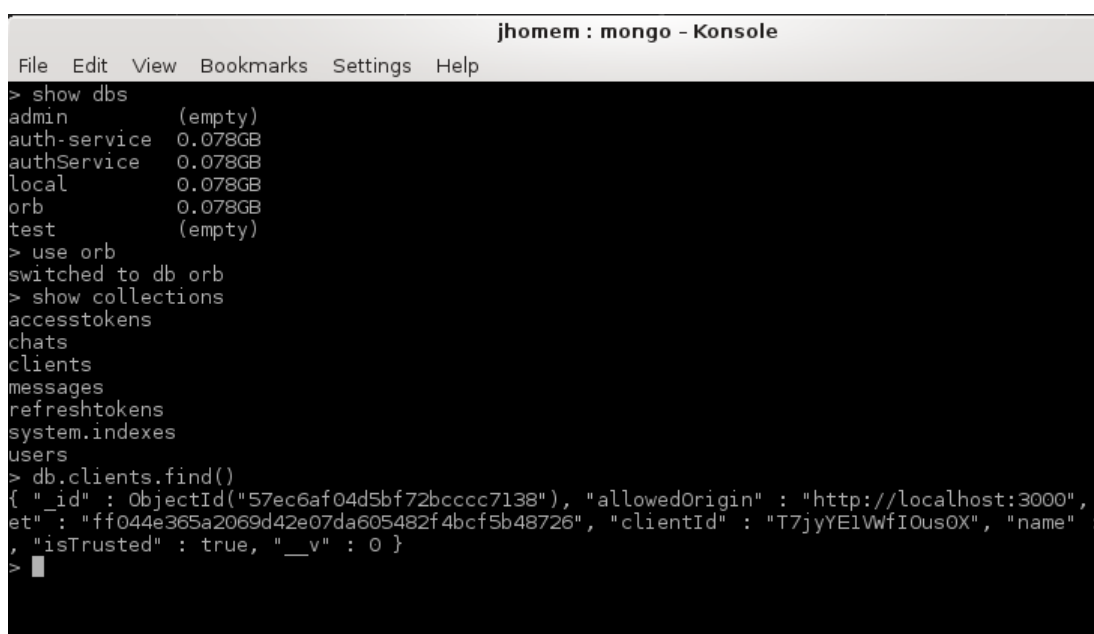


Figura 5.2: Representação UML (Unified Modeling Language) dos modelos persistidos no banco de dados.

Apesar de utilizar um banco de dados não relacional, a imagem 5.2 demonstra o relacionamento entre as entidades do sistema. A ideia de banco de dados não relacional vem da não aplicação de uma regra que deve ser respeitada na persistência dos dados, mas isso não quer dizer que não possamos ter referências para as outras entidades. O intuito desse tipo de banco é oferecer flexibilidade de acordo com a necessidade, sendo assim, podemos persistir documentos sem seguir uma série de princípios de bancos de dados relacionais. Na figura 5.3 demonstramos uma breve utilização do do MongoDB, onde são executados comandos para exibir os bancos existentes no servidor, seleção do banco de dados da aplicação, exibição das coleções de documentos, e finalmente a exibição de um registro em formato JSON da coleção *users*.



```
jhomem : mongo - Konsole
File Edit View Bookmarks Settings Help
> show dbs
admin (empty)
auth-service 0.078GB
authService 0.078GB
local 0.078GB
orb 0.078GB
test (empty)
> use orb
switched to db orb
> show collections
accesstokens
chats
clients
messages
refreshtokens
system.indexes
users
> db.clients.find()
{ "_id" : ObjectId("57ec6af04d5bf72bcccc7138"), "allowedOrigin" : "http://localhost:3000", "token" : "ff044e365a2069d42e07da605482f4bcf5b48726", "clientId" : "T7jyYE1VwfIOus0X", "name" : "jhomem", "isTrusted" : true, "__v" : 0 }
```

Figura 5.3: Demonstração de utilização do MongoDB pelo terminal do CentOS.

5.1.2 Autenticação e Autorização

Para a implementação do sistema de autenticação, cujos códigos estão no diretório *middlewares*, utilizamos três pacotes NPM, que funcionam como *middlewares* que extraem de um *request* as informações de autenticação, e através de uma *callback* é possível validar essas informações e sinalizar se as informações estão corretas para concretização da autenticação. Para o sistema de autorização, utilizamos o OAuth2 com algumas modificações em seu padrão para atender aplicações *trusted*, como o nosso *frontend app* desenvolvido em AngularJS, que é um tipo de aplicação que não necessita de autorização do usuário da conta para acessar suas informações confidenciais.

A figura 5.4 representa o fluxo de autenticação utilizado no sistema. Como estamos utili-

zando um sistema de *tokens*, para realizar acesso ao *endpoint* token (Authorization Server na figura), o qual é utilizado no sistema de *login* para se conseguir um Access Token e um Refresh Token através do fornecimento de *e-mail* e senha, é realizada a autenticação pelos padrões Basic [55] e Client [56]. Após esta autenticação, o usuário é encaminhado para um *exchange* do OAuth2 que irá gerar, persistir e retornar um Access Token e um Refresh Token para o solicitante, que poderá usá-lo para acessar os demais *endpoints* da aplicação (Resource Server na figura).

Como já introduzido, o restante das rotas da aplicação são protegidos através do padrão de autenticação Bearer [57], que procura em cada requisição feita um Access Token, que será verificado, e caso seja válido permitirá o acesso ao *endpoint* solicitado. Em caso do Access Token não ser válido, o servidor irá emitir uma resposta de erro, HTTP 401, informando o problema ocorrido, e se o problema for a expiração do Access Token, ele poderá utilizar o Refresh Token para conseguir um novo Access Token válido por mais um determinado tempo, evitando de ter que realizar toda o processo de autenticação com *e-mail* e senha novamente. É interessante ressaltar que o *Resource Server* e o *Authorization Server* foram implementados na mesma aplicação, diferentemente do que se pode entender com a figura 5.4 demonstrando elementos separados.

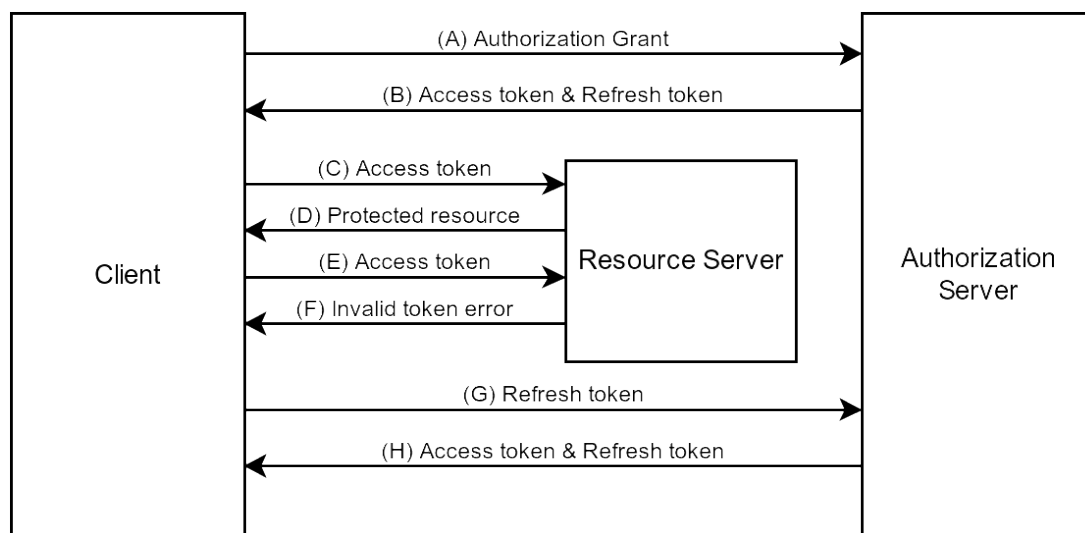


Figura 5.4: Fluxo de acesso aos recursos com OAuth2. Fonte: jlabusch [59]

Um detalhe importante que deve ser falado, é a utilização do método bcrypt [60] para criptografia da senha dos usuários e a utilização do crypt [61] para a criptografia do Access Token e Refresh Token. Como o bcrypt é um tipo de criptografia mais complexa, que leva em conta o custo de processamento, utilizamos-o em funcionalidades menos requisitadas, como o *login*, e utilizamos o crypt (SHA256) para a criptografia dos *tokens*, pois são requisitados na

maioria das rotas do sistema, consequentemente necessitando-se fazer a verificação de chaves a todo momento, sendo assim, a utilização de um método menos custoso se faz benéfico.

5.1.3 Rotas

As rotas de aplicação estão organizadas no diretório controller da aplicação, tendo sido implementadas de acordo com o tipo de persistência que elas representam. A seguir, uma listagem organizada pelos *endpoints* das rotas e uma descrição sobre sua funcionalidade:

client A rota client possui dois *endpoints* que podem ser acessados através de chamadas pelo método HTTP GET e HTTP POST. O método GET retorna uma lista em formato JSON de todos os clients cadastrados no sistema. O método POST recebe um client em formato JSON e cadastra-o no banco.

token A rota token possui apenas um *endpoint* chamado revoke, o qual é responsável por remover do banco de dados o Access Token e Refresh Token relacionados a um determinado cliente. Este *endpoint* é utilizado para realizar o *signout* da aplicação.

user A rota user possui quatro *endpoints*, sendo o primeiro podendo ser chamado pelo método GET e retornando todos os usuários persistidos. O segundo pelo método GET, passando-se como parâmetro o id do usuário que se quer retornar. O terceiro pelo método POST, passando-se como parâmetro o usuário em formato JSON que se quer persistir. O quarto pelo *endpoint* accesstoken e método GET, passando-se como parâmetro o Access Token do usuário que se quer retornar.

As rotas seguem o padrão de utilização dos métodos HTTP para realização do CRUD em APIs, sendo os *endpoints* citados funcionalidades da aplicação, e são protegidos de acordo com as especificações faladas no tópico Autenticação e Autorização. Estes *endpoints* são chamados pela aplicação *frontend*, que passa em cada *request* todas as informações de autenticação necessária, e trata os dados de sucesso ou erro que cada *endpoint* retorna.

5.1.4 Socket

O desenvolvimento da comunicação em tempo real do *chat* é a principal funcionalidade do *backend* da aplicação. Os arquivos de implementação estão localizados no diretório domain, onde três arquivos implementam todas as funcionalidades relacionadas.

Inicialmente foi feito o *download* pelo NPM das bibliotecas *socket.io* e *redis*. A inicialização do servidor de conexão do *socket.io* foi feita no arquivo de inicialização do servidor através da porta 1200. A conexão com o banco de dados *Redis* foi feita no arquivo que trava os eventos dos *sockets*.

Com a biblioteca devidamente configurada, o servidor já está pronto para receber conexões *socket*. No arquivo *socketBootstrap.js* criamos um sistema de inicialização que cria um *namespace* de conexão chamado *chat*, e também realiza a autenticação dos usuários que tentam estabelecer uma conexão por *socket* com o servidor. Para tratar a autenticação, utilizamos uma biblioteca [62] que oferece uma solução semelhante a explicada no tópico Autenticação e Autorização.

No arquivo *chatSocketHandler.js* implementamos todos os eventos de envio e recebimento de mensagens dos *sockets*. Visando a utilização do protocolo PUB/SUB, abrimos três conexões/canais com o nosso servidor *Redis*, o primeiro para *publish* de mensagens, o segundo para *subscription* e o terceiro para *storage* de informações dos usuários *online*.

A seguir uma lista dos eventos emitidos e recebidos no *namespace* *chat* e a descrição de seu funcionamento:

disconnect Este evento é emitido por um cliente que se desconectou do servidor. Neste evento tratamos de remover este usuário da lista de usuários *online* do *storage* e lançamos um evento *chat:signout*, comunicando a todos os usuários que aquele usuário se desconectou, logo devem o remove-lo de seu mapa.

chat:signout Este evento é emitido para todos os usuários, informando-os o id de um usuário que acabou de se desconectar do servidor. Quando este evento é recebido pelo cliente, ele procura e remove este usuário do seu mapa.

chat:new Este evento é emitido pelo cliente quando ele inicia um *chat* com outro usuário. Este evento procura no banco de dados a existência de uma conversa entre os dois usuários, caso exista, esta conversa é lida com as últimas 15 mensagens registradas. Caso não exista, é criada uma nova conversa entre os usuários.

chat:online:list Este evento é recebido pelo servidor e emitido por ele próprio com uma lista de informações de todos os usuários *online*, inclusive suas respectivas localizações geográficas.

chat:message:send Este evento é emitido pelo cliente quando ele clica no botão de enviar de um janela de *chat*. Neste evento é enviado os dados do *chat*, como a mensagem e o id

do chat, e o *handler* deste evento no servidor registra a mensagem no banco de dados e a envia para todos os membros da conversa.

chat:message Todo cliente é capaz de receber este evento. Quando o servidor recebe um evento `chat:message:send`, ele registra a mensagem no banco e envia para cada membro do *chat* a respectiva mensagem através deste evento.

chat:position:update Este evento é emitido pelo cliente, com latitude e longitude de sua localização geográfica, quando ele se conecta ao *chat* e quando ele se movimento pelo menos 100 metros em alguma direção. Após receber este evento o servidor emite o mesmo, juntamente com os dados do usuário para todos os outros clientes, assim eles podem atualizar a localização real do usuário em seus mapas.

chat:ready Este evento é emitido pelo servidor para um usuário que acabou de se conectar após todos os eventos relacionados aquele a ele tenham sido registrados.

É importante frisar a emissão de eventos por parte do servidor são feitas através da função `init`, que é registrada quando se executa o servidor, onde esta função utiliza o sistema de PUB/SUB do redis para receber as mensagens, procurar a existência do *socket* de destino em sua lista de *sockets* conectados a aquele servidor, e se encontrado a mensagem é enviada a ele. Este esquema de funcionamento é que permite a comunicação entre os diversos servidores que a aplicação pode ter, pois todos os servidores irão receber uma mensagem publicada no Redis por algum deles, podendo assim verificar se a mensagem pertence a algum de seus clientes e enviá-la ou não.

Toda a interação com o banco de dados é feita através do repositório criado no arquivo `chat-SocketHandler.js`. Neste arquivo estão implementados diversos métodos e serviços de CRUD para os eventos dos *sockets*.

5.2 Frontend

O desenvolvimento da aplicação *frontend* foi feito com AngularJS. O uso deste *framework* trouxe para a aplicação algumas características importantes para a construção de interfaces, como o conceito de SPA e um padrão de projeto, que quando se lançou a ferramenta a documentação declarava este padrão como MVVA, mas ao longo do tempo os desenvolvedores foram adquirindo novas formas de se trabalhar com o AngularJS, criando-se divergências sobre qual modelo realmente é empregado, sendo assim a documentação oficial declarou o modelo

MVW (Model-View-Whatever). Este modelo de *design pattern* tem algumas semelhanças com o aplicado no projeto de *backend*.

Para começar o desenvolvimento, criamos um diretório chamado *frontend* na pasta raiz da aplicação. Dentro deste diretório criamos duas pastas, *dist* e *src*, onde na pasta *dist* ficam todos os arquivos que estão prontos para serem distribuídos pelo servidor da aplicação, como imagens, bibliotecas de terceiros e fontes. Na pasta *src* contém todos os arquivos que serão processados por algum *plugin* do Gulp que gerará um arquivo correspondente na pasta *dist*, como os arquivos da pasta *app*, referente a aplicação Angular, arquivos SASS, referente a estilos, entre outros. Também inicializamos os gerenciadores de pacotes NPM e Bower, e o WebPack.

No diretório *app*, temos os arquivos relacionados ao projeto AngularJS. Na pasta *controller* deste diretório estão implementados os *controllers*, que controlam blocos de componentes HTML em uma página, eles são ligados ao DOM, e através deles é possível manipular os dados da *view* através dos *models* implementados, assim como vincular métodos com ações, entre outras funcionalidades. Os *controllers* foram utilizados para criar diversas interações na aplicação, como a abertura de novas janelas de *chat*, envio de mensagens, *handlers* dos eventos dos *sockets*, entre outros. O diretório *directives* possui pastas que contém implementações de diretivas AngularJS, como o painel de *chats*, a janela de *chat*, lista de contatos, entre outros. Cada pasta possui todos os arquivos relacionados a aquela diretiva específica, como *controllers*, *services* e *view*. O diretório *services* possui serviços AngularJS que implementam funcionalidades de lógica de negócio e de repositórios que fazem acesso aos *endpoints* da API para inserir ou ler dados.

Apesar da utilização do Angular Material em muitos dos componentes da interface, também implementamos alguns componentes próprios, como o *header*, painel de *chats*, a janela de *chat*, entre outros. O diretório *styles* possui todas as implementações em SASS das folhas de estilo.

5.3 Resultados

O principal objetivo do trabalho foi alcançado na implementação do sistema, que foi explorar o funcionamento de aplicações de tempo real. Com a ajuda de todas as ferramentas citadas, foi possível desenvolver uma aplicação *web* com alguns fluxos de utilização que facilitam o entendimento do que é e como pode ser aplicada esse tipo de tecnologia na construção de uma ferramenta de comunicação, além do estudo de uma arquitetura que pode ser utilizada para outros tipos de finalidade.

Para o projeto entregue foram alcançados a comunicação por mensagens de texto entre dois

usuários, demonstrado na imagem 5.5, a exibição e alteração da localização real do usuário, o sistema de autenticação *stateless* através de *tokens*, criação de novas contas de usuário, demonstrado na imagem 5.6, persistência de mensagens no banco de dados e utilização de conceitos de escalabilidade em aplicações.

As arquitetura do sistema seguiu conceitos de escalabilidade os quais não puderam ser testados em sua totalidade devidas a não extensão do tempo de elaboração do trabalho final, apenas demonstramos teoricamente a função de cada tecnologia em um ambiente que se possa ter diversas instâncias da aplicação distribuídas em uma rede de servidores. Assim como a proficiência em desempenho também não foi alvo de análise.

Alguns componentes de interface, apesar de existirem visualmente, não foram implementadas as funcionalidades para a execução das tarefas, como as alterações de informações da conta do usuário, lista de contatos, mostrados na imagem 5.7, e página de sobre. Apesar da utilização de um *framework* que provê compatibilidade e responsividade de componentes entre diversos *browsers*, alguns componentes foram desenvolvidos a parte, e não foram realizados testes *cross-browsers* para verificar o funcionamento correto de todos os componentes, sendo as funcionalidades apenas validadas no navegador de testes para desenvolvimento, o Google Chrome, versão 54.

A seguir podemos observar algumas imagens da interface depois de implementada:

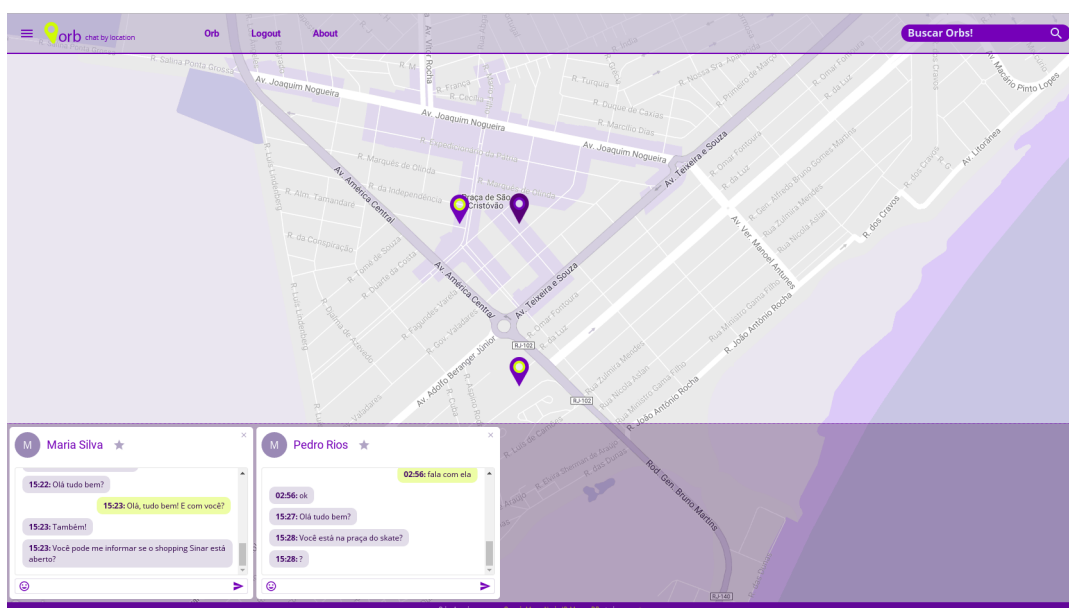



Figura 5.5: Página principal. Fonte: Captura de tela

orb chat by location Sign About Buscar Orbs!

Sign In or Sign Up




Sign In

Enter with your e-mail and password

E-mail

Password

SIGN IN



Sign up now and enjoy our community!

First Name Last Name

E-mail

Password Confirm Password

SIGN UP

Orb - Aplicativos por Google Maps, Facebook, Instagram e todos os outros

Figura 5.6: Página de acesso ao sistema. Fonte: Captura de tela

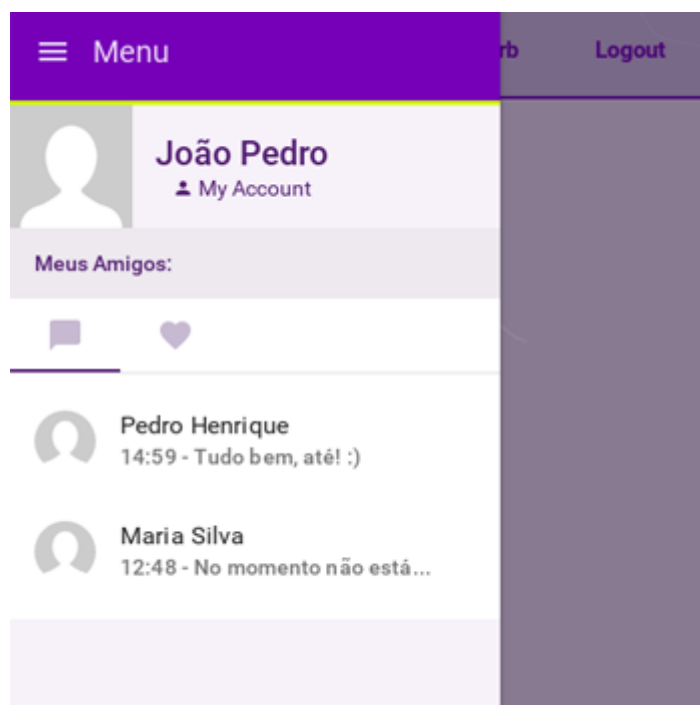


Figura 5.7: Menu lateral. Fonte: Captura de tela

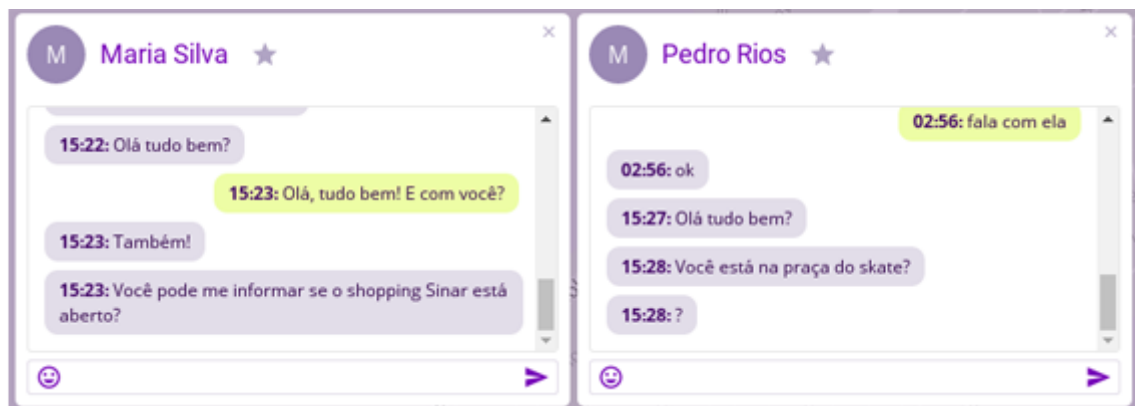


Figura 5.8: Janelas de *chat*. Fonte: Captura de tela

A figura 5.9 demonstra como ficou arquitetado o funcionamento do sistema. O Frontend, representado no topo da figura, é do escopo de utilização do usuário, onde a aplicação funciona no *web browser* dos clientes e oferece as interações necessárias para utilização do sistema. A aplicação *frontend* se comunica com a aplicação *backend* através de chamadas HTTP à API, e também conexões websockets.

As linhas que interligam os componentes da arquitetura representam o caminho pelo qual os dados trafegam na rede, onde estes são transmitidos e recebidos através de rede pública (internet) e privada (rede de comunicação interna entre as camadas do servidor), que tem em seu escopo o Nginx, servidores de aplicação NodeJS e banco de dados.

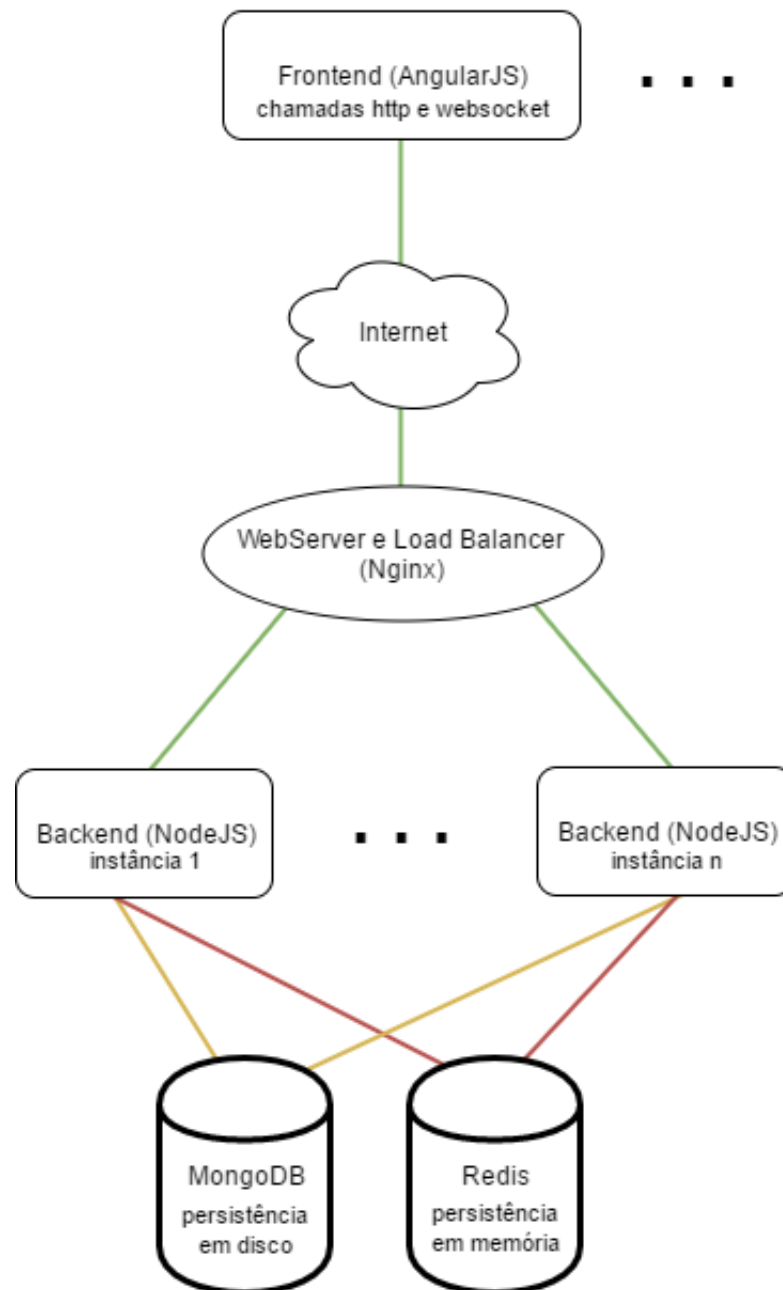


Figura 5.9: Representação da arquitetura do sistema. Fonte: Autoria própria

6 Conclusão

A elaboração deste projeto trabalhou diversas áreas do desenvolvimento de *softwares*, desde a infraestrutura à criação de interfaces. A aplicação de uma tecnologia que proporciona interação em tempo real, foco do nosso tema, foi facilmente introduzida devido as características que as tecnologias escolhidas já propunham, tendo em vista que a *stack* MEAN (MongoDB, ExpressJS, AngularJS, NodeJS) é muito utilizada para a construção de aplicações de tempo real, possibilitando uma pesquisa muito mais fácil devido as diversas soluções semelhantes já elaboradas por outros desenvolvedores.

A exploração deste tema nos proporcionou conhecer melhor muitas das aplicações deste tipo de tecnologia, e principalmente a experiência em se desenvolver uma aplicação deste tipo, tendo em vista a solução que este tipo de tecnologia pode oferecer para diversos problemas tecnológicos. Os trabalhos relacionados apresentam grandes ferramentas que utilizam uma ou mais tecnologias citadas e demonstram a relevância do tema para o tempo atual da tecnologia, levando a trabalhos cada vez maiores e que exploram as diversas complexidades que sistemas de tempo real podem alcançar para atender as necessidades demandadas. Acreditamos que os resultados alcançados atingiram as metas propostas pelo trabalho, e deixam abertas oportunidades para pesquisas mais específicas da tecnologia.

A ideia deste trabalho foi ser como um passo inicial para o entendimento do ambiente de tempo real em aplicações, consequentemente não aplicamos e nem nos aprofundamos em algumas boas práticas, como o desenvolvimento orientado a testes, o que tornou a utilidade da aplicação meramente demonstrativa, mas que com mais trabalho pode se tornar algo utilizável para outros futuros estudos na área em que abrange.

6.1 Trabalhos Futuros

A criação das funcionalidades de alterações de informações da conta do usuário e lista de contatos. assim como a validação dos componentes da interface em outros navegadores, como o

FireFox, Opera, Safari, Internet Explorer e Microsoft Edge, podem ser implementações rápidas, devido a toda a infraestrutura já estar pronta, assim como a interface, e tornar o sistema útil para algum fim específico que possa se beneficiar das características de comunicação do sistema.

Pensando na utilização por diversos usuários do sistema desenvolvido, existem implementações que devem ser refeitas devido a problemas aparentes de desempenho que a aplicação pode ter, principalmente nas implementações de interação com o mapa, pois a aplicação atual marca todos os usuários *online* no sistema no mapa de cada usuário, o que não é o ideal. Melhorias nesse sentido podem envolver uma pesquisa mais aprofundada nas APIs do Google Maps, para entender melhor as formas de utilização, assim como citamos em trabalhos relacionados, o qual o Uber pode utilizar ferramentas diferentes para a elaboração do seu mapa.

Outro ponto importante que envolve este problema de escalabilidade com o sistema de localização, pode ser a utilização das funcionalidades de geolocalização do banco de dados Redis, onde as informações ao invés de serem distribuídas uniformemente para todos os usuários, elas podem ser baseadas na localização do usuário requerente, onde por exemplo, os usuários obtenham informações apenas de outros usuários próximos a ele, fazendo-se cálculos para retorno de dados baseando-se em um determinado raio de distância.

A simulação de ambientes distribuídos e testes de estresse podem ser realizados na aplicação e trazerem resultados interessantes, que podem demonstrar a eficiência da arquitetura desenvolvida para a aplicação. Isto também implicará em novas implementações, como *sticky sessions* e *load balancer*, desmistificando muitos conceitos envolvidos em aplicações que suportam milhares de usuários simultâneos.

Referências

- [1] Liveclicker Delivers Dynamic Messaging Capability with RealTime Email Solution. Disponível em: <https://goo.gl/C4Z3pv>. Acesso em: 19 Out. 2016.
- [2] Liveclicker RealTime Email Solution. Disponível em: <http://www.realtime.email/>. Acesso em: 19 Out. 2016.
- [3] 50 Amazing Uber Statistics (October 2016). Disponível em: <http://expandedramblings.com/index.php/uber-statistics/>. Acesso em: 20 Out. 2016.
- [4] Uber. Disponível em: <https://www.uber.com/>. Acesso em: 20 Out. 2016.
- [5] Referral SaaSquatch. Disponível em: <https://www.referralsaasquatch.com/how-the-uber-referral-program-thinks-about-user-experience/>. Acesso em: 26 Nov. 2016.
- [6] How Uber Scales Their Real-Time Market Platform. Disponível em: <http://highscalability.com/blog/2015/9/14/how-uber-scales-their-real-time-market-platform.html>. Acesso em: 20 Out. 2016.
- [7] Redis Publish-Subscribe (RedisMVA). Disponível em: https://github.com/sayar/RedisMVA/blob/master/module6_redis_pubsub/README.md#redis-publish-subscribe. Acesso em: 20 Out. 2016.
- [8] Web browsers usage table. Disponível em: <http://caniuse.com/usage-table>. Acesso em: 20 Out. 2016.
- [9] Redis Pub/Sub (Rajaraodv). Disponível em: <https://github.com/rajaraodv/redispubsub>. Acesso em: 20 Out. 2016.
- [10] FanMappr. Disponível em: <http://h.fanapp.mobi/modules/fanmappr/fanmappr.php?fid=1056666>. Acesso em: 20 Out. 2016.
- [11] RadiusIM – Social Messenger Com Busca Por Geotagging. Disponível em: <http://br.wwwhatsnew.com/2008/09/radiusim-%E2%80%93-social-messenger-por-com-busca-por-geotagging/>. Acesso em: 20 Out. 2016.
- [12] NOTNOTCITRICSQUID. Do location based chat apps start trending again?. Disponível em: https://www.reddit.com/r/startups/comments/1tkc3f/do_location_based_chat_apps_start_trending_again/. Acesso em: 21 Out. 2016.

- [13] TigetFunk. Top 10 Applications For Singles. Disponível em: <http://tigerfunk.com/applications-for-singles/>. Acesso em: 26 Nov. 2016.
- [14] SmartnTechs. An Entire Account on Snapchat – How to Get Started, Find Friends, Block Person and Lot More. Disponível em: <https://goo.gl/06EnLI>. Acesso em: 26 Nov. 2016.
- [15] News. 10 tips on how to get a date using Tinder. Disponível em: <http://www.news.com.au/finance/business/tips-on-how-to-get-a-date-using-tinder/story-fn5lic6c-1226692193146>. Acesso em: 26 Nov. 2016.
- [16] Snapchat. Disponível em: <https://www.snapchat.com/>. Acesso em: 21 Out. 2016.
- [17] Tinder. Disponível em: <https://www.gotinder.com/>. Acesso em: 21 Out. 2016.
- [18] Happn. Disponível em: <https://www.happn.com/>. Acesso em: 21 Out. 2016.
- [19] How Much Web Traffic Is On Mobile Devices (Really)? Disponível em: <https://keriganmarketing.com/freelunch/view/how-much-web-traffic-is-on-mobile-devices/>. Acesso em: 22 Out. 2016.
- [20] Internet stats and facts for 2016. Disponível em: <https://hostingfacts.com/internet-facts-stats-2016/>. Acesso em: 22 Out. 2016.
- [21] Mobile Marketing Statistics compilation. Disponível em: <http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/>. Acesso em: 22 Out. 2016.
- [22] Notebook HP 1000-1460BR. Disponível em: <http://support.hp.com/br-pt/document/c03788977>. Acesso em: 24 Out. 2016.
- [23] Difference between CentOS, Fedora, and RHEL. Disponível em: <https://linux-audit.com/difference-between-centos-fedora-rhel/>. Acesso em: 27 Nov. 2016.
- [24] Facebook Linux, What Distro is it?. Disponível em: <http://www.internetnews.com/blog/skerner/facebook-linux-CentOS.html>. Acesso em: 24 Out. 2016.
- [25] Red Hat Enterprise Linux Atomic Host expands to Google Compute Engine. Disponível em: <https://cloudplatform.googleblog.com/2014/07/red-hat-enterprise-linux-atomic-host-expands-to-google-compute-engine.html?m=1>. Acesso em: 24 Out. 2016.
- [26] Gitflow Workflow. Disponível em: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>. Acesso em: 24 Out. 2016.
- [27] Sublime Text. Disponível em: <https://www.sublimetext.com/>. Acesso em: 24 Out. 2016.
- [28] Package Control. Disponível em: <https://packagecontrol.io/>. Acesso em: 24 Out. 2016.
- [29] Visual Studio Code. Disponível em: <https://code.visualstudio.com/>. Acesso em: 24 Out. 2016.

- [30] NodeJS. Disponível em: <https://nodejs.org/>. Acesso em: 24 Out. 2016.
- [31] Chrome V8. Disponível em: <https://developers.google.com/v8/>. Acesso em: 24 Out. 2016.
- [32] Introdução ao Node.js. Disponível em: <http://maxroecker.github.io/blog/introducao-ao-nodejs/>. Acesso em: 24 Out. 2016.
- [33] Libuv. Disponível em: <https://github.com/libuv/libuv>. Acesso em: 24 Out. 2016.
- [34] NPM. Disponível em: <https://www.npmjs.com/>. Acesso em: 24 Out. 2016.
- [35] ExpressJS. Disponível em: <https://expressjs.com/>. Acesso em: 24 Out. 2016.
- [36] Criando Salas com Socket.IO. Disponível em: <http://blog.ifollow.com.br/2015/06/01/criando-salas-com-socket-io/>. Acesso em: 25 Out. 2016.
- [37] Mongoose. Disponível em: <http://mongoosejs.com/>. Acesso em: 25 Out. 2016.
- [38] Redis. Disponível em: <http://redis.io/>. Acesso em: 25 Out. 2016.
- [39] Using New Relic to Understand Redis Performance: The 7 Key Metrics. Disponível em: <https://blog.newrelic.com/2015/05/11/redis-performance-metrics/>. Acesso em: 25 Out. 2016.
- [40] Bower. Disponível em: <https://bower.io/>. Acesso em: 25 Out. 2016.
- [41] AngularJS. Disponível em: <https://angularjs.org/>. Acesso em: 25 Out. 2016.
- [42] Angular Material. Disponível em: <https://material.angularjs.org/>. Acesso em: 25 Out. 2016.
- [43] Gulp. Disponível em: <http://gulpjs.com/>. Acesso em: 25 Out. 2016.
- [44] SASS. Disponível em: <http://sass-lang.com/>. Acesso em: 25 Out. 2016.
- [45] HAML. Disponível em: <http://haml.info/>. Acesso em: 26 Out. 2016.
- [46] HAMLJS. Disponível em: <https://github.com/creationix/haml-js>. Acesso em: 26 Out. 2016.
- [47] WebPack. Disponível em: <https://webpack.github.io/>. Acesso em: 26 Out. 2016.
- [48] Babel. Disponível em: <https://babeljs.io/>. Acesso em: 26 Out. 2016.
- [49] Nginx. Disponível em: <https://www.nginx.com/>. Acesso em: 26 Out. 2016.
- [50] NAT 1:1 ou Proxy Reverso, qual a melhor alternativa? Disponível em: <http://jff.eti.br/nat-11-ou-proxy-reverso-qual-a-melhor-alternativa/>. Acesso em: 26 Out. 2016.
- [51] Orb. Disponível em: <https://github.com/joaopsh/orb>. Acesso em: 26 Out. 2016.
- [52] MIT License. Disponível em: <https://opensource.org/licenses/MIT>. Acesso em: 26 Out. 2016.

-
- [53] Best practices for Express app structure. Disponível em: <https://www.terlici.com/2014/08/25/best-practices-express-structure.html>. Acesso em: 27 Out. 2016.
- [54] CentOS. Disponível em: <https://www.centos.org/>. Acesso em: 27 Out. 2016.
- [55] Passport-HTTP. Disponível em: <https://github.com/jaredhanson/passport-http>. Acesso em: 27 Out. 2016.
- [56] passport-oauth2-client-password. Disponível em: <https://github.com/jaredhanson/passport-oauth2-client-password>. Acesso em: 27 Out. 2016.
- [57] passport-http-bearer. Disponível em: <https://github.com/jaredhanson/passport-http-bearer>. Acesso em: 27 Out. 2016.
- [58] OAuth2orize. Disponível em: <https://github.com/jaredhanson/oauth2orize>. Acesso em: 27 Out. 2016.
- [59] OAUTH 2.0 SERVER. Disponível em: <http://jlabusch.github.io/oauth2-server/>. Acesso em: 27 Out. 2016.
- [60] bcrypt. Disponível em: <https://www.npmjs.com/package/bcrypt>. Acesso em: 28 Out. 2016.
- [61] crypto-js. Disponível em: <https://www.npmjs.com/package/crypto-js>. Acesso em: 28 Out. 2016.
- [62] socketio-auth. Disponível em: <https://github.com/facundoolano/socketio-auth>. Acesso em: 28 Out. 2016.

APÊNDICE A – Principais códigos do software

Arquivo authentication.js:

```

var passport = require('passport') , BasicStrategy = require('passport-http').BasicStrategy ,
ClientPasswordStrategy = require('passport-oauth2-client-password').Strategy ,
BearerStrategy = require('passport-http-bearer').Strategy , User = require('../models/user') ,
Client = require('../models/client') , AccessToken = require('../models/accessToken') , crypto
= require('crypto') , allowedOrigins = require('../config.json').origins;

var isStrategiesConfigured = false;

passport.use("clientBasic", new BasicStrategy( function (clientId, clientSecret, done)
clientSecret = crypto.createHash('sha1').update(clientSecret).digest('hex')

Client.findOne( clientId: clientId , function (err, client) if (err) return done(err, false);

if (!client) return done(null, false);

if (!client.isTrusted) return done(null, false);

if (client.clientSecret === clientSecret) return done(null, client); else return done(null, false);
); ));

passport.use("clientPassword", new ClientPasswordStrategy( function (clientId, clientSecret,
done) clientSecret = crypto.createHash('sha1').update(clientSecret).digest('hex');

Client.findOne( clientId: clientId , function (err, client) if (err) return done(err, false);

if (!client) return done(null, false);

if (!client.isTrusted) return done(null, false);

if (client.clientSecret == clientSecret) return done(null, client); else return done(null, false); );
));

passport.use("accessToken", new BearerStrategy( function (accessToken, done) var
accessTokenHash = crypto.createHash('sha1').update(accessToken).digest('hex');
```

```
AccessToken.findOne( token: accessTokenHash , function (err, token) if (err) return done(err);
if (!token) return done(null, false)
if (new Date() > token.expirationDate) done(null, false);
else User.findOne(username: token.userId, function (err, user) if (err) return done(err);
if (!user) return done(null, false);
Client.findOne( clientId: token.clientId , function(err, client) if(err) return done(null, false);
if(!client) return done(null, false);
return done(null, user, scope: '*' ); ); ); ); );
```

Arquivo authorization.js:

```
var oauth2orize = require('oauth2orize') , passport = require('passport') , crypto =
require('crypto') , utils = require("../helpers/utils") , bcrypt = require('bcrypt') , User =
require('../models/user') , AccessToken = require('../models/accessToken') , RefreshToken =
require('../models/refreshToken');

// create OAuth 2.0 server var server = oauth2orize.createServer();

//Resource owner password server.exchange(oauth2orize.exchange.password(function (client,
email, password, scope, done) User.findOne( email: email , function (err, user) if (err) return
done(err);
if (!user) return done(null, false);
bcrypt.compare(password, user.password, function (err, res) if (!res) return done(null, false);
var newAccessToken = utils.uid(256); var newRefreshToken = utils.uid(256);
var newAccessTokenHash = crypto.createHash('sha1').update(newAccessToken).digest('hex');
var newRefreshTokenHash =
crypto.createHash('sha1').update(newRefreshToken).digest('hex');
// 10 minutes token var expirationDate = new Date(new Date().getTime() + (1000 * 60 * 200));
var accessToken = new AccessToken();
accessToken.token = newAccessTokenHash; accessToken.expirationDate = expirationDate;
accessToken.clientId = client.clientId; accessToken.userEmail = email;
AccessToken.remove( clientId: client.clientId, userEmail: email , function(err) if(err) return
done(err); );
```

```
accessToken.save(function (err) if (err) return done(err);

var refreshToken = new RefreshToken();

refreshToken.token = newRefreshTokenHash; refreshToken.clientId = client.clientId;
refreshToken.userEmail = email;

RefreshToken.remove( clientId: client.clientId, userEmail: email , function(err) if(err) return
done(err); );

refreshToken.save(function(err) if (err) return done(err);

done(null, newAccesstoken, newRefreshToken, expires_in: expirationDate ); ); ); ); ); );

//Refresh Token server.exchange(oauth2orize.exchange.refreshToken(function (client,
refreshToken, scope, done) var refreshTokenHash =
crypto.createHash('sha1').update(refreshToken).digest('hex');

RefreshToken.findOne( token: refreshTokenHash , function (err, foundRefreshToken) if (err)
return done(err);

if (!foundRefreshToken) return done(null, false);

if (client.clientId !== foundRefreshToken.clientId) return done(null, false);

var newAccesstoken = utils.uid(256); var newRefreshToken = utils.uid(256);

var newAccessTokenHash = crypto.createHash('sha1').update(newAccesstoken).digest('hex');
var newRefreshTokenHash =
crypto.createHash('sha1').update(newRefreshToken).digest('hex');

// 10 minutes token var expirationDate = new Date(new Date().getTime() + (1000 * 60 * 200));

var accessToken = new AccessToken();

accessToken.token = newAccessTokenHash; accessToken.expirationDate = expirationDate;
accessToken.clientId = client.clientId; accessToken.userEmail =
foundRefreshToken.userEmail;

AccessToken.remove( clientId: client.clientId, userEmail: foundRefreshToken.userEmail ,
function(err) if(err) return done(err); );

accessToken.save(function (err) if (err) return done(err);

var refreshToken = new RefreshToken();

refreshToken.token = newRefreshTokenHash; refreshToken.clientId = client.clientId;
```



```

refreshToken.userEmail = foundRefreshToken.userEmail;

RefreshToken.remove( clientId: client.clientId, userEmail: foundRefreshToken.userEmail ,
function(err) if(err) return done(err); );

refreshToken.save(function(err) if (err) return done(err);

done(null, newAccesstoken, newRefreshToken, expires_in: expirationDate ); ); ); ); );

// token endpoint exports.token = [ passport.authenticate(['clientBasic', 'clientPassword'],
session: false ), server.token(), server.errorHandler() ]

```

Arquivo chatSocketHandler.js:

```

var config = require('../config.json') , redis = require('redis') , storage =
redis.createClient(config.redis.port, config.redis.host) , sub =
redis.createClient(config.redis.port, config.redis.host) , pub =
redis.createClient(config.redis.port, config.redis.host) , utils = require('../helpers/utis') , Chat
= require('../models/chat') , repo = require('../chatSocketHandlerRepository');

storage.on('connect', function() if(process.env.NODE_ENV === 'dev')
storage.del('chat.onlineList');

storage.del('subscribers');

console.log('Redis clients connected on HOST ' + config.redis.host + ':' + config.redis.port);
);

//It's limited to ten listeners. Zero means infinity sub.setMaxListeners(0);

var _init = function(chatNamespace) sub.on('message', function(channel, message)
switch(channel) case 'chat:position:update': var data = JSON.parse(message);

chatNamespace.emit('chat:position:update', id: data.id, email: data.email, firstName:
data.firstName, lastName: data.lastName, coords: latitude: data.latitude, longitude:
data.longitude ); break; case 'chat:signout': chatNamespace.emit('chat:signout',
JSON.parse(message)); break; case 'chat:message': var parsedMessage =
JSON.parse(message);

parsedMessage.chat.members.forEach(function(member) for (var property in
chatNamespace.connected) if (chatNamespace.connected.hasOwnProperty(property))

if(chatNamespace.connected[property].user.id === member.id)
chatNamespace.connected[property].emit('chat:room', parsedMessage); break; );

```

```
break;

);

//Global list of channels subscriptions (It should subscribes only once) var subs = [];

//handler var _handler = function(socket) if(subs.indexOf('chat:position:update') === -1)
subs.push('chat:position:update'); sub.subscribe('chat:position:update');

if(subs.indexOf('chat:signout') === -1) subs.push('chat:signout');
sub.subscribe('chat:signout');

if(subs.indexOf('chat:invitation') === -1) subs.push('chat:invitation');
sub.subscribe('chat:invitation');

if(subs.indexOf('chat:message') === -1) subs.push('chat:message');
sub.subscribe('chat:message');

//receptors socket.on('chat:position:update', function(data) // Add or update the user in the
online list storage.hmset('chat.onlineList' , socket.user.email , JSON.stringify( id:
socket.user.id, firstName: socket.user.firstName, lastName: socket.user.lastName, coords:
latitude: data.latitude, longitude: data.longitude ) );

data.id = socket.user.id; data.email = socket.user.email; data.firstName =
socket.user.firstName; data.lastName = socket.user.lastName;

pub.publish('chat:position:update', JSON.stringify(data)); );

socket.on('chat:message:send', function(message) message.from = socket.user.id;

repo.addMessageAndReturnService(message).then(function(message)
pub.publish('chat:message', JSON.stringify(message));

, function(err) throw new Error(err); );

);

// Creates a chat room and invites the target user socket.on('chat:new', function(invitedUsers)
if(!Array.isArray(invitedUsers)) throw new Error('invitedUsers variable in chat:new event is
not an array.');
```

```
var invitedUsersIds = [socket.user.id];

invitedUsers.forEach(function(invitedUser) invitedUsersIds.push(invitedUser.id); );

// Get an existing chat or create a new one, then gets the full user objects
```

```

repo.getChatAndMembersAndMessagesService(invitedUsersIds).then(function(chat)
socket.emit('chat:new', chat); );

);

// Get online users socket.on('chat:online:list', function(data) storage.hgetall('chat.onlineList',
function(err, onlineList) if(onlineList === null) onlineList = ;

var result = []; var keys = Object.keys(onlineList);

for(var i = 0; i < keys.length; i++) var userInfoParse = JSON.parse(onlineList[keys[i]]);

result.push( id: userInfoParse.id, email: keys[i], firstName: userInfoParse.firstName,
lastName: userInfoParse.lastName, coords: latitude: userInfoParse.coords.latitude, longitude:
userInfoParse.coords.longitude );

socket.emit('chat:online:list', result); ); );

// When disconnect socket.on('disconnect', function() storage.hdel('chat.onlineList',
socket.user.email);

pub.publish('chat:signout', JSON.stringify( id: socket.user.id )); );

socket.emit('chat:ready', );

exports.handler = _handler; exports.init = _init;

```

Arquivo chatSocketHandlerRepository.js:

```

var Q = require('q') , Chat = require('../models/chat') , Message =
require('../models/message') , User = require('../models/user') , helper =
require('../helpers/utills') , mongoose = require('mongoose');

var _createChatAndReturn = function(members) var deferred = Q.defer();

new Chat( members: members ).save(function(err) if(err) deferred.reject(err);

_findChatByUsers(members).then(function(chat) deferred.resolve(chat);

, function(err) deferred.reject(err);

);

);

return deferred.promise

var _findChatByUsers = function(members) var deferred = Q.defer();

```

```
Chat.findOne( $and: [ members: $all: members , members: $size: members.length ] ,
'-__v' , function(err, chat) if(err) deferred.reject(err);

deferred.resolve(chat); );

return deferred.promise;

var _getUsersByIds = function(ids) var deferred = Q.defer();

var mongoIds = [];

ids.forEach(function(id) mongoIds.push(new mongoose.Types.ObjectId(id)); );

User.find( _id: $in: mongoIds , '-__v -password' , function(err, users) if(err)
deferred.reject(err);

deferred.resolve(users); );

return deferred.promise;

var _getMessagesByChatId = function(chatId) var deferred = Q.defer();

Message.find( chatId: chatId , '-__v').sort( timestamp: 'desc' ).limit(15).exec(function(err,
messages) if(err) deferred.reject(err);

// The older messages should be in the initial array positions, like a push natural behavior.
messages.reverse();

deferred.resolve(messages); );

return deferred.promise;

var _getChatOrCreateAndReturn = function(members) var deferred = Q.defer();

_findChatByUsers(members).then(function(chat) if(!chat)
_createChatAndReturn(members).then(function(chat) deferred.resolve(chat);

, function(err) deferred.reject(err); );

else deferred.resolve(chat);

, function(err) deferred.reject(err); );

return deferred.promise;

var _addMessageAndReturn = function(message) var deferred = Q.defer();

var timestamp = helper.getUTCTimestamp();

new Message( chatId: message.chatId, from: message.from, timestamp: timestamp, text:
```

```
message.text ).save(function(err) if(err) deferred.reject(err);

Message.findOne( $and: [ timestamp: timestamp , chatId: message.chatId ] , '-__v',
function(err, message) if(err) deferred.reject(err);

deferred.resolve(message); );

);

return deferred.promise;

var _getChatById = function(id) var deferred = Q.defer();

Chat.findOne( _id: new mongoose.Types.ObjectId(id) , '-__v', function(err, chat) if(err)
deferred.reject(err);

deferred.resolve(chat); );

return deferred.promise;

//services (services are everything that handles data to something. Eg. multiple db calls, obj
structure changes and so on. var _getChatAndMembersAndMessagesService =
function(members) var deferred = Q.defer();

_getChatOrCreateAndReturn(members).then(function(chat)
_getMessagesByChatId(chat._id).then(function(messages)
_getUsersByIds(members).then(function(users) var handledUsers = [];

users.forEach(function(user) handledUsers.push( id: user._id.toString(), email: user.email,
firstName: user.firstName, lastName: user.lastName ); );

deferred.resolve( id: chat._id.toString(), members: handledUsers, messages: messages );

, function(err) return deferred.reject(err); ); , function(err) return deferred.reject(err); ); ,
function(err) return deferred.reject(err); );

return deferred.promise;

var _addMessageAndReturnService = function(message) var deferred = Q.defer();

_addMessageAndReturn(message).then(function(message)
_getChatById(message.chatId).then(function(chat)
_getMessagesByChatId(message.chatId).then(function(messages)
_getUsersByIds(chat.members).then(function(users) var handledUsers = [];

users.forEach(function(user) handledUsers.push( id: user._id.toString(), email: user.email,
```

```

firstName: user.firstName, lastName: user.lastName ); );

deferred.resolve( chat: id: chat._id.toString(), members: handledUsers, messages: messages ,
message: id: message._id.toString(), chatId: message.chatId, from: message.from,
timestamp: message.timestamp, text: message.text );

, function(err) deferred.reject(err); );

, function(err) deferred.reject(err); );

, function(err) deferred.reject(err); );

, function(err) deferred.reject(err); );

return deferred.promise;

exports.createChatAndReturn = _createChatAndReturn; exports.findChatByUsers =
_findChatByUsers; exports.getUsersByIds = _getUsersByIds; exports.getMessagesByChatId =
_getMessagesByChatId; exports.getChatOrCreateAndReturn = _getChatOrCreateAndReturn;
exports.addMessageAndReturn = _addMessageAndReturn; exports.getChatById =
_getChatById;

exports.getChatAndMembersAndMessagesService =
_getChatAndMembersAndMessagesService; exports.addMessageAndReturnService =
_addMessageAndReturnService;

```

Arquivo socketBootstrap.js:

```

var auth = require('socketio-auth') , crypto = require('crypto') , User =
require('../models/user') , AccessToken = require('../models/accessToken') ,
chatSocketHandler = require('./chatSocketHandler');

var socketBootstrap = function(io) //creates a namespace called "chat" var chat = io.of('/chat');

chatSocketHandler.init(chat);

//chat authentication auth(chat, authenticate: function (socket, data, callback)
if(!data.credentials || !data.credentials.access_token) return callback(new Error("Invalid access
token"));

var accessTokenHash =
crypto.createHash('sha1').update(data.credentials.access_token).digest('hex');

AccessToken.findOne( token: accessTokenHash , function(err, accessToken) if(err) return
callback(new Error("Internal server error."));

```

```
if(!accessToken) return callback(new Error("Access token not found."));

if (new Date() > accessToken.expirationDate) return callback(new Error("Expired access
token."));

User.findOne( email: accessToken.userEmail , function (err, user) if (err) return callback(new
Error("Internal server error."));

if (!user) return callback(new Error("User not found."));

//bind the user informations with the socket socket.user = id: user._id.toString(), email:
user.email, firstName: user.firstName, lastName: user.lastName

return callback(null, true); ); );

, postAuthenticate: function(socket, data) chatSocketHandler.handler(socket); , timeout: 1000
* 20 );

module.exports = socketBootstrap;
```

Arquivo userController.js:

```
var express = require('express') , router = express.Router() , crypto = require('crypto') , User =
require('../models/user') , AccessToken = require('../models/accessToken') , mongoose =
require('mongoose') , passport = require('passport');

router.get('/', passport.authenticate('accessToken', session: false ), function(req, res)
User.find(, '-password -__v', function(err, users)

var results = users.map(function(user) return id: user.id, firstName: user.firstName, lastName:
user.lastName, email: user.email ;

);

return res.json( status: 'OK', users: results ); );

);

router.get('/:id', passport.authenticate('accessToken', session: false ), function(req, res)
User.findOne( _id: req.params.id , '-password -__v', function(err, user) if(err) return
res.status(500).json( status: 'ERROR', message: 'Internal Server Error.' );

if(user) return res.json( status: 'OK', user: id: user.id, firstName: user.firstName, lastName:
user.lastName, email: user.email );

else res.status(400).json( status: 'ERROR', message: 'User does not exist.' );
```

```
);  
  
);  
  
router.get('/accesstoken/:accessToken', passport.authenticate('accessToken', session: false ),  
function(req, res) if(!req.params.accessToken) return res.status(400).json( status: 'ERROR',  
message: 'Missing access token parameter.');
```



```
var accessTokenHash =  
crypto.createHash('sha1').update(req.params.accessToken).digest('hex');
```



```
AccessToken.findOne( token: accessTokenHash , function(err, accessToken) if(err) return  
res.status(500).json( status: 'ERROR', message: 'Internal Server Error.');
```



```
if(!accessToken) return res.status(400).json( status: 'ERROR', message: 'Invalid access  
token.');
```



```
User.findOne( email: accessToken.userEmail , '-password -__v', function(err, user) if(err)  
return res.status(500).json( status: 'ERROR', message: 'Internal Server Error.');
```



```
if(user) return res.status(200).json( status: 'OK', user: id: user.id, firstName: user.firstName,  
lastName: user.lastName, email: user.email );
```



```
else return res.status(400).json( status: 'ERROR', message: 'User does not exist.' );
```



```
);  
  
);  
  
);
```



```
router.post('/', function(req, res) req.checkBody('user.firstName', 'First name is  
required.').notEmpty(); req.checkBody('user.lastName', 'Last name is required.').notEmpty();  
req.checkBody('user.email', 'E-mail is required.').notEmpty();  
req.checkBody('user.password', 'Password is required.').notEmpty();
```



```
var errors = req.validationErrors();
```



```
if(errors) return res.status(400).json(errors);
```



```
User.findOne( email: req.body.user.email , function(err, foundUser) if(err) return  
res.status(500).json( status: 'ERROR', message: 'Internal Server Error.');
```



```
if(foundUser) res.status(400).json( status: 'ERROR', message: 'User already exists.' );
```



```
else
```



```
var user = new User();

user.firstName = req.body.user.firstName; user.lastName = req.body.user.lastName; user.email
= req.body.user.email; user.password = req.body.user.password;

user.save(function(err) if(err) if(err.name == 'ValidationError') return res.status(400).json(
status: 'ERROR', message: 'Validation Error.');
```

```
return res.status(500).json( status: 'ERROR', message: 'Internal Server Error.');
```

```
else return
res.status(200).json( status: 'OK', user: id: user._id, firstName: user.firstName, lastName:
user.lastName, email: user.email );

);

); );

module.exports = router;
```

Arquivo user.js:

```
var mongoose = require('mongoose') , bcrypt = require('bcrypt');
```

```
var userSchema = new mongoose.Schema( firstName: type: String, required: true, , lastName:
type: String, required: true, , email: type: String, required: true, unique: true, , password:
type: String, required: true, , image: type: String, default: '/dist/images/profile-default.jpeg' ,
);

userSchema.pre('save', function(next) var user = this;

if(!user.isModified('password')) return next();

bcrypt.genSalt(10, function(err, salt) if(err) return next(err);

bcrypt.hash(user.password, salt, function(err, hash) if(err) return next(err);

user.password = hash;

return next(); ); ); );

var User = mongoose.model('User', userSchema);

module.exports = User;
```

Arquivo server.js:

```
var config = require('./config.json') , express = require('express') , app = module.exports =
express() , io = require('socket.io')(config.app.socketioPort) , chatSocketHandler =
require('./domain/socketBootstrap')(io) , mongoose = require('mongoose') , bodyParser =
```

```
require('body-parser') , passport = require('passport') , expressValidator =
require('express-validator') , authentication = require('./middlewares/authentication') , oauth =
require('./middlewares/authorization');

//controllers import var userController = require('./controllers/userController') ,
clientController = require('./controllers/clientController') , tokenController =
require('./controllers/tokenController');

//set environment process.env.NODE_ENV = 'dev';

mongoose.connect(config.mongodb);

//middlewares register app.use(bodyParser.urlencoded( extended: true ));
app.use(bodyParser.json()); app.use(expressValidator()); app.use(passport.initialize());

app.use(function(req, res, next) res.set('Access-Control-Allow-Headers', 'Content-Type,
Authorization'); res.set('Access-Control-Expose-Headers', 'WWW-Authenticate');
res.set('Access-Control-Allow-Origin', config.allowedOrigins);

if(req.method === 'OPTIONS') res.sendStatus(200); else return next();

);

//routes register app.post('/oauth/token', oauth.token); app.use('/oauth',
passport.authenticate('accessToken', session: false ), tokenController); app.use('/user',
userController); app.use('/client', passport.authenticate('accessToken', session: false ),
clientController);

app.listen(config.app.apiPort, function() console.log('Orb API running on PORT ' +
config.app.apiPort); );
```