UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO
INSTITUTO MULTIDISCIPLINAR

MATHEUS ZACHÉ GONÇALVES

Construção de um *chat web* para comunicação em tempo real em uma plataforma para egressos

Prof. Filipe Braida do Carmo, D.Sc.
Orientador

Nova Iguaçu, Junho de 2024

Construção de um *chat web* para comunicação em tempo real em uma plataforma para egressos

Matheus Zaché Gonçalves

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto Multidisciplinar da Universidade Federal Rural do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Apresentado por:

_____

Matheus Zaché Gonçalves

Aprovado por:

_____

Prof. Filipe Braida do Carmo, D.Sc.

_____

Prof. Bruno José Dembogurski, D.Sc.

_____

Prof. Marcel William Rocha da Silva, D.Sc.

NOVA IGUAÇU, RJ - BRASIL

Junho de 2024

Agradecimentos

Em primeiro lugar gostaria de agradecer a Deus, em cuja fé redescobri durante meu período na graduação. Sem Ele nada seria possível, principalmente graças às pessoas que agiram em minha vida em Seu nome. Dentre essas pessoas, cabe um agradecimento especial ao Padre Roberto e ao Padre Paulo, os sacerdotes na época de meu retorno à Igreja.

Agradecimentos também são devidos à minha família. Aos meus pais, Marcos e Márcia, pela paciência e pelo apoio financeiro. À minha avó, Ozeína, pelos serviços domésticos muito além da sua obrigação ou mesmo necessidade. Aos meus outros dois avós, Arlete e Paulo, e meu tio, Marlon, pela constante lembrança e preocupação. E finalmente obrigado ao meu irmão, Phelipe, pela companhia.

Também quero agradecer aos meus amigos pessoais, a maioria me acompanha muito antes do ingresso na universidade. Particularmente, muito obrigado a Pedro Caetano por não apenas ter dividido comigo seu sonho dar vida às suas histórias, mas por ter me convidado a construí-lo ao seu lado. Hoje, acredito que esse sonho seja nosso.

Muito obrigado aos meus amigos espalhados pelo Brasil, que até hoje compartilham seu tempo comigo através dos jogos de luta. Alguns já tive o prazer de conhecer pessoalmente, mas até para os que ainda não encontrei face a face, ter com quem jogar, competir e trocar experiências continua sendo imprescindível anos depois.

Agradeço a Gabriel e ao Alexandre por terem investido e acreditado nas minhas capacidades como desenvolvedor *web*. Em um período pessoal de dúvidas a respeito da minha permanência na área, eles me ofereceram oportunidade, flexibilidade e

compreensão para que eu pudesse trabalhar da melhor forma.

A todo corpo docente do DCC, agradeço pelo aprendizado durante todo esse tempo. Em especial, muito obrigado ao meu orientador, Filipe Braida, pelo direcionamento, paciência e confiança.

RESUMO

Construção de um *chat web* para comunicação em tempo real em uma plataforma para egressos

Matheus Zaché Gonçalves

Junho/2024

Orientador: Filipe Braida do Carmo, D.Sc.

Ao longo dos anos, tem surgido a demanda por acompanhamento de estudantes egressos tanto nas universidades quanto em instituições interessadas, sejam elas públicas ou privadas. Alcançar os egressos exige que os dados deixados nas universidades estejam constantemente atualizados, anos depois de suas partidas. Esse desafio oferece a oportunidade para a criação de plataformas para que os egressos interajam entre si e com a própria universidade, incentivando a manutenção de seus dados. O sistema proposto neste trabalho visa construir um *chat* com comunicação em tempo real para uma plataforma *alumni* em desenvolvimento, utilizando as tecnologias *web* e os princípios de aplicações de dados. Este trabalho também propõe estudar as alternativas na *web* para comunicação em tempo real, entendendo como os protocolos da *internet* transferem dados, além de ponderar as vantagens e desvantagens de cada solução para troca de mensagens bidirecional.

ABSTRACT

Construção de um *chat web* para comunicação em tempo real em uma plataforma para egressos

Matheus Zaché Gonçalves

Junho/2024

Advisor: Filipe Braida do Carmo, D.Sc.

Through the years, there has been a demand for tracking graduate students from both universities and institutions, whether they are private or public. Reaching out to graduates depends on the data they left at said universities being constantly updated, years after their departure. This challenge creates an opportunity for the development of platforms that allow graduates to interact with each other and with their university, thereby incentivizing the maintenance of their data. The system proposed in this paper seeks to build a real-time chat for an alumni platform that is already in development, utilizing web technologies and the principles of data applications. This paper also intends to study the alternatives for real-time communication on the web, understand how internet protocols transfer data, and weigh the advantages and disadvantages of each solution for bidirectional message exchange.

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviaturas e Siglas

HTTP *Hyper Text Transfer Protocol*

IP *Internet Protocol*

TCP *Transmission Control Protocol*

WS *WebSockets*

SME Sistema de Monitoramento de Egressos

OSI *Open Systems Interconnection*

UDP *User Datagram Protocol*

API *Application Programming Interface*

GUID *Globally Unique Identifier*

ORM *Object Relational Mapping*

JSON *JavaScript Object Notation*

XML *eXtensible Mark-up Language*

SME Sistemas de Monitoramento de Egressos

GTS *Graduate Tracking Systems*

HTML *Hyper Text Mark-up Language*

CSS *Cascade Style Sheets*

JS *JavaScript*

CHEERS *Carrer after Higher Education: a European Research Study*

MVC *Model View Controller*

SGBD Sistema Gerenciador de Banco de Dados

ACID *Atomicity, Consistency, Isolation, Durability*

URL *Uniform Resource Locator*

SPA *Single Page Application*

CORS *Cross-Origin Resource Sharing*

SMS *Short Message Service*

SDK *Standart Development Kit*

REST *Representational State Transfer*

Sumário

Capítulo 1

Introdução

A população brasileira tem, nas últimas décadas, impulsionado seu número de graduados a nível superior. Segundo INEP (2023), entre 2002 e 2021, esse número saltou de 3,5 para 8,9 milhões. Dentro desse grupo, somente no último ano, se formaram um milhão. Esse fenômeno não é exclusivo ao Brasil, Paul (2015) relata que países como França os Estados Unidos atravessaram, e atravessam, semelhante processo. A crescente demanda por formação ensino superior foi descrita por Sipos (2017) como a transição para uma economia baseada em conhecimentos.

Cada vez mais, a universidade deixa de ser seu próprio ambiente, com própria prática e cultura, para se comunicar com as exigências de fora. De acordo com dados de ABMES (2023), 75% dos egressos estão empregados em até um ano após a sua graduação. A maior parte, 83%, exercem um cargo condizente à sua formação. Entretanto, Sipos (2017) afirma que não necessariamente o conhecimento acadêmico prevalece no mercado. As áreas do conhecimento estão em constante mudança, criando uma distância entre teoria e prática.

Uma forma de amenizar o problema foi o surgimento de Sistemas de Monitoramento de Egressos (SME). Assim que os estudantes se formam, as instituições onde cursaram fazem um levantamento de seu estado posterior. SME são de interesse de toda a sociedade, existindo como iniciativas públicas e privadas. O processo visa construir uma relação de retorno entre estudante e instituição, de forma que a

universidade esteja sempre se atualizando à demandas novas do mercado. Por outro lado, o vínculo entre os responsáveis pela pesquisa e os egressos pode ser efêmero. Egressos mudam de *e-mail*, endereço, número de telefone muito depois de seu tempo de formado. Não há como a universidade atualizar seu banco de dados sem um pedido direto, o qual pressupõe prévio conhecimento de como acessar o egresso.

Os institutos responsáveis por essas pesquisas necessitam de mitigar esse cenário. Michael et al. (2012) traçam um paralelo das necessidades de um SME com um de uma rede social e, portanto, deve haver um meio de recompensar o egresso por seu empenho. Portanto, quanto mais próximo da experiência de uma rede social, maior adesão a SME os usuários teriam. Especialmente no Brasil, as redes sociais que permitem comunicação direta entre os usuários possuem um prestígio singular. Atualmente, *Whatsapp* é a rede social mais utilizada em território nacional, seguida pelo *Instagram* e *Facebook* (BIANCHI, 2023). A comunicação proporcionada por esses aplicativos, exige que as informações dos usuários sejam constantemente atualizadas. Logo, a integração de um *chat* para essas plataformas, se mostra como uma solução emergente para esse problema. Nesse momento, na Universidade Federal Rural do Rio de Janeiro, há uma plataforma para egressos em desenvolvimento, o *Alumni*, sobre a qual o desenvolvimento de um sistema de comunicação em tempo real se faz oportuno.

1.1 Objetivo

A proposta deste trabalho é um sistema comunicação em tempo real, por meio de um*chat* para o ambiente *web*, para o SME em desenvolvimento da universidade. Sua comunicação ocorrerá por meio do protocolo *WebSockets*. O sistema permite envio de mensagens de texto, áudio, imagem ou qualquer outro arquivo, em tempo real. Ademais, o sistema também possibilita conversas entre mais de dois usuários, através da reunião destes usuários em grupos. Essa aplicação visa aproximar as partes interessadas no *Alumni*, tanto egressos quanto empresas e profissionais da instituição de ensino superior. Ao mesmo tempo, a possibilidade de comunicação é o incentivo aos egressos atualizarem suas informações com maior frequência ao mesmo

tempo que se tornam mais acessíveis a oportunidades de emprego ou pesquisas.

1.2 Organização do Trabalho

O trabalho está organizado da seguinte forma:

- **Capítulo 2:** serão discutidos os pilares e fundamentos de uma aplicação de dados, como um sistema de comunicação em tempo real. Alternativas e técnicas contribuem para a elaboração da proposta do trabalho serão discutidas a fundo.
- **Capítulo 3:** será dada uma breve explicação de como a transferência e apresentação de conteúdos funciona na *internet*. Depois, uma explicação aprofundada de como o protocolo *WebSockets* opera nesta infraestrutura a fim de promover uma comunicação de duas vias em tempo real.
- **Capítulo 4:** busca aplicar os conceitos de aplicações de dados e *websockets* em uma modelagem das entidades, banco de dados e serialização para o sistema, além de justificar as decisões que serão tomadas durante a implementação.
- **Capítulo 5:** serão detalhadas as tecnologias empregadas na construção do *chat*. Também serão apresentados como se deram a implementação e integração dos ambientes da aplicação *back-end* e do *front-end*. Ademais, cada tela e funcionalidade receberão um guia a respeito do visual, além da utilização.
- **Capítulo 6:** virão as considerações finais a respeito da aplicação, ressaltando limitações do projeto inicial, além de uma discussão breve sobre propostas de avanço em seu desenvolvimento.

Capítulo 2

Aplicações *Data-Intensive*

Desenvolvimento de *software* é um processo com padrões de projeto definidos, de acordo com as características da aplicação que se deseja construir. Aplicações podem ser caracterizadas de duas formas: *compute-intensive* ou *data-intensive*. Essa classificação está relacionada aos limites impostos pelo domínio da aplicação. Aplicações *compute-intensive* são limitadas pela potência do processamento, ao passo que *data-intensive* diz respeito à manipulação dos dados. A partir de agora, essa classe será escrita pela sua tradução: aplicação de dados.

Kleppmann (2017) define aplicações de dados como um termo generalizado para contemplar um conjunto de práticas a respeito de dados. Dados são armazenados, indexados ou processados, seja um lote ou tempo real. O propósito desta seção é avaliar, entre as práticas mais comuns, as alternativas para implementar uma aplicação de dados. Na seção 2.1 será discutido os princípios de uma aplicação deste porte. Debaixo desses fundamentos, serão estudadas alternativas de armazenamento, na seção 2.2, e serialização, na seção 2.3.

2.1 Pilares da Aplicação de Dados

Segundo Kleppmann (2017), aplicações de dados seguem três princípios ou pilares para seu desenvolvimento. Estes pilares são **Confiabilidade**, **Manutenibilidade** e

Escalabilidade. Cada escolha no desenvolvimento de uma aplicação de dados tem esses pilares em mente. Por outro lado, cada aplicação possui contextos diferentes, de modo que um pilar se torna mais caro que outro. Resta compreender o que estes pilares significam por si mesmos, como também um para o outro.

Confiabilidade define um sistema capaz de funcionar a despeito de erros. Um sistema pode apresentar falhas ou erros. Heimerdinger et al. (1992) definiram a falha como a perda completa de um serviço ao usuário. Em uma escala menor, o erro se trata de um componente menor, desviando de sua função. Erros sucessivos criam o cenário de falha. Falhas devem ser evitadas a todo custo, enquanto erradicar todos os erros é uma tarefa praticamente impossível. Um sistema confiável é tolerante a erros. Erros de *hardware* têm sua origem na falha dos componentes físicos do projeto. Eles podem envolver, mas não limitados a, perdas inesperadas de energia, armazenamento insuficiente ou mal funcionamento de uma peça. Mais sutis que os erros de *hardware*, erros no *software* podem permanecer escondidos por muito tempo. Eles podem ser *bugs* que se propagam para outras funcionalidades, ou impedem acesso a recursos compartilhados. O ser humano também é uma possível origem do erro de uma aplicação de dados. Humanos são erráticos, e, portanto, são capazes de preencher dados incorretamente ou usar a aplicação de jeitos inesperados.

Um sistema com **Escalabilidade** persiste em sua boa performance, apesar da carga crescente sobre ele. Carga significa diferentes problemas para uma aplicação de dados. No contexto da *web*, pode ser a quantidade de requisições em um período de tempo. No entanto, dependendo do tipo de aplicação, a carga esperada varia. Mais ainda, se a carga se tornar um problema apenas em casos extremos, a solução pode mudar. Portanto, convém definir o que seria uma performance aceitável com base no domínio da aplicação deste trabalho. Na *web*, um rápido tempo de resposta é imprescindível. Um tempo de resposta aceitável pode significar a mediana de todos os acessos, ou uma performance melhorada nos piores casos. As opções para escalar um sistema são horizontais ou verticais. **Escalabilidade** vertical significa mudar para máquinas mais potentes, seja com mais memória, tempo de resposta mais veloz ou maior armazenamento. Já **Escalabilidade** horizontal, distribui a carga entre várias máquinas.

O pilar da **Manutenibilidade** envolve o custo de conservação do software. O propósito é construir o *software*, de modo que sua correção siga três principais princípios: operabilidade, simplicidade e evolução. O primeiro diz respeito à facilidade com que os problemas cotidianos são resolvidos. Uma equipe pode efetuar esses processos, porém há meios para o *software* tornar essa tarefa menos árdua, *e.g.*, ter um comportamento previsível. Simplicidade está relacionada à redução das complexidades no código escrito. Vale ressaltar que um código menos complexo não implica em um código com menos funcionalidade. Deseja-se evitar a noção de complexidade acidental, como descrita por Moseley e Marks (2006). Nela, complexidade surge não de uma característica fundamental da funcionalidade, mas de como a funcionalidade está implementada no código. Abstração é uma prática capaz de esconder problemas sofisticados em soluções de fácil entendimento. Por último, evolução é o princípio que depende dos outros dois. Os requisitos de um *software* nem sempre são estáticos. Logo, quanto mais simples o entendimento do sistema, mais fácil será modificá-lo.

2.2 Armazenamento de Dados

Modelar o banco de dados de uma aplicação é um processo que afeta todo o sistema. No tocante à aplicações *web*, banco de dados definem quais serão as informações que recebemos e apresentamos ao usuário. A forma como tratamos essa questão diz respeito a alguns pilares da aplicação de dados. A escolha por um deles afeta como os dados são armazenados e indexados, fatores que contribuem para a **Escalabilidade**. A modelagem do banco também age na escrita do código. Aplicações de dados são desenvolvidas com ajuda de *frameworks*, que consistem em grandes abstrações de código feitas para facilitar a concepção de um sistema. *Frameworks* possuem *Object Relational Mapping* (ORM), responsável por traduzir a modelagem do banco para entidades no código. Contudo, mesmo esta tradução não está livre de problemas, principalmente em caso de alteração, seja no banco ou no próprio código. Portanto, o pilar da **Manutenibilidade** também cumpre um papel nesta etapa.

Os dois paradigmas a serem discutidos são: banco de dados relacional e não-

relacional, definidos por Korth, S. e A (2019). Um banco de dados possui uma estrutura definida, chamada de *schema*. A diferença entre os dois paradigmas está em como no reforço do *schema*. Bancos relacionais exigem obediência ao *schema* no momento da escrita. Isso significa que cada entrada nova precisa obedecer a uma estrutura pré-estabelecida. Bancos não-relacionais, ou baseados em documento, não reforçam uma estrutura pré-definida para a escrita de suas linhas. A persistência dos seus dados é classificada como poliglota, conforme Sadalage e Fowler (2012). Em outras palavras, somente no momento de apresentá-los que a aplicação vai estruturar o que foi escrito.

Em um banco de dados, entidades se relacionam cardinalmente. Uma entidade pode estar relacionada a uma única, numa relação 1:1, como também para várias, 1:N. Analogamente, várias entidades podem relacionar-se entre si, simultaneamente, descrito com uma relação N:N, muitos para muitos. Um banco relacional constrói esta relação por meio de identificadores únicos. Cada entidade registrada possui o seu, de modo que pode ser referenciado no registro de outras entidades e vice-versa. Bancos não-relacionais não possuem este capacidade, graças a sua ausência de uma estrutura rígida na escrita. Relacionamentos são escritos por extenso, diretamente na entidade, em um processo denominado denormalização.

No âmbito da **Escalabilidade**, bancos relacionais crescem tanto vertical quanto horizontalmente. Ao mesmo tempo que uma máquina é capaz de guardar mais dados, é possível distribuir este armazenamento em várias máquinas. Por outro lado, bancos não-relacionais expandem horizontalmente. Uma vez que todos os dados entram em um único documento, aumentar seu armazenamento consiste em disponibilizar mais documentos a serem escritos. Esta divergência afeta o relacionamento entre os dados em cada paradigma. A denormalização do paradigma não-relacional determina que registros duplicados se tornem recorrentes em um documento. De um lado, a recuperação de um registo não-relacional é mais rápida, já que não realiza buscas encadeadas. Do outro, a quantidade de informações repetidas pode se tornar um problema para sistemas maiores, porque o banco relacional possui apenas um registro referenciado em vários lugares.

Esta característica também se torna um fator determinante na **Manutenibilidade**. Registros duplicados em um banco não-relacional, implicam em cada mudança na modelagem ser uma alteração em várias partes do documento. Caso contrário, dados se tornarão inconsistentes ao longo de toda a aplicação. Em um banco relacional, identificadores, chamados de chaves primárias, são imutáveis. Se a demanda por mudança em uma entidade surgir, somente um registro será alterado. A escolha de um paradigma precisa considerar o domínio da aplicação, a fim de escrever um código mais simples e uma performance melhor. Para aplicações cujos dados estão organizados de maneira hierárquica, *e.g.*, uma árvore, uma estrutura de documentos, não-relacional é preferencial por evitar buscas sucessivas. Na ocasião dos dados precisarem de relacionamentos frequentes entre várias entidades,

2.3 Serialização

Kleppmann (2017) afirma que programas tem duas formas de utilizar dados. A primeira são dados em memória, *i.e.*, objetos, árvores, vetores ou variáveis. A segunda são dados para escrita ou envio para rede, se houver. Este segundo envolve a conversão do que temos em memória para uma sequência auto-contida de *bytes*. Este processo é denominado *encoding*, porém, daqui em diante, será referido como serialização. Serializar os dados possui vantagens para a comunicação entre sistemas diferentes, assim como para o armazenamento em bancos de dados. Partindo da premissa onde várias partes de um sistema podem mudar a qualquer instante, a serialização é uma forma de tornar o código resiliente às alterações. É um meio unificado e encapsulado de comunicação, visando principalmente o pilar da **Manutenibilidade**, mais especificamente seu princípio de evolução.

Essa necessidade é especialmente importante na *web*. Mais a frente, veremos como protocolos de rede trabalham para reduzir a carga dos envios dos pacotes em tempo real. Mais ainda, o formato do corpo que é enviado e recebido por cliente e servidor se torna um problema separado. Na prática, são sistemas diferentes, possivelmente com linguagens de programação distintas trocando informações. Os dados entrando e saindo precisam ser serializados de modo que as duas pontas possam decodificá-los.

Tradicionalmente, o HTTP se comunica em formato de texto, apesar de versões mais recentes suportarem compactação (M.BISHOP; AKAMAI, 2022). As notações em texto seguem um padrão de escrita hierárquico, legível para humanos, de modo que sua construção no código seja simples. Como desvantagem, envios em texto exigem mais espaço no pacote do que uma serialização binária.

Nesta classe de serialização em texto temos *JavaScript Object Notation* (JSON) [1] e *eXtensible Mark-up Language* (XML) [2]. O formato JSON tem suporte a maioria dos navegadores por ser uma parte do *JavaScript*. Em contrapartida, sua implementação em códigos é menos estruturada que o XML. Formatos de texto possuem o problema da ambiguidade em seus dados, principalmente valores numéricos. Não está especificado no JSON se um valor é inteiro ou real, muito menos sua precisão. O XML já possui recursos para representar valores numéricos de vários tipos, porém sua escrita é muito mais verborrágica quando comparada ao JSON. Consequentemente, uma notação em XML consome mais espaço do que JSON.

[1] <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/JSON>

[2] <https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction>

Capítulo 3

Comunicação Em Tempo Real na *Web*

Os pilares de uma aplicação de dados fornecem uma base para pensar sistemas de grande porte. Na prática, como Kleppmann (2017) destaca, não há um padrão definitivo, tanto quanto um conjunto de práticas recorrentes durante desenvolvimento. O sistema de comunicação em tempo real é uma aplicação de dados, que depende da sincronia de envio e recebimento das mensagens para gerar **Confiabilidade** na aplicação. Para tal, o fluxo de dados entre as pontas precisa chegar rapidamente e sem necessidade de recarregamento. Suprir essa demanda tem sido um dos desafios dos protocolos que regem a *internet*. Neste capítulo será discutido em detalhes o funcionamento destes protocolos, sobretudo do *WebSockets*.

Na seção 3.1, será abordado o funcionamento do HTTP. Primeiro, um breve resumo sobre os protocolos que o auxiliam, TCP e *Internet Protocol* (IP). Depois, um histórico das tentativas de solucionar o problema da comunicação em tempo real. Então na seção 3.2, será tratado como *WebSockets* funciona por cima das estruturas definidas da internet, de modo a superar as limitações das alternativas anteriores. Finalmente, *WebSockets* serão comparadas às soluções históricas do HTTP.

3.1 HTTP

Tanenbaum (2003) descreve rede de computadores como um conjunto de computadores independentes, porém conectados entre si através de uma tecnologia única. Os computadores de uma rede são chamados de nós, que trocam informações no meio através de pacotes. Para mediar esta comunicação, existem os chamados protocolos. Protocolos operam em diversas áreas da rede, da física até a lógica. Pode-se dizer que os diversos protocolos montam uma pilha hierárquica de camadas. A mediação se dá por uma sequência de *bytes* presente em cada pacote, denominada cabeçalho.

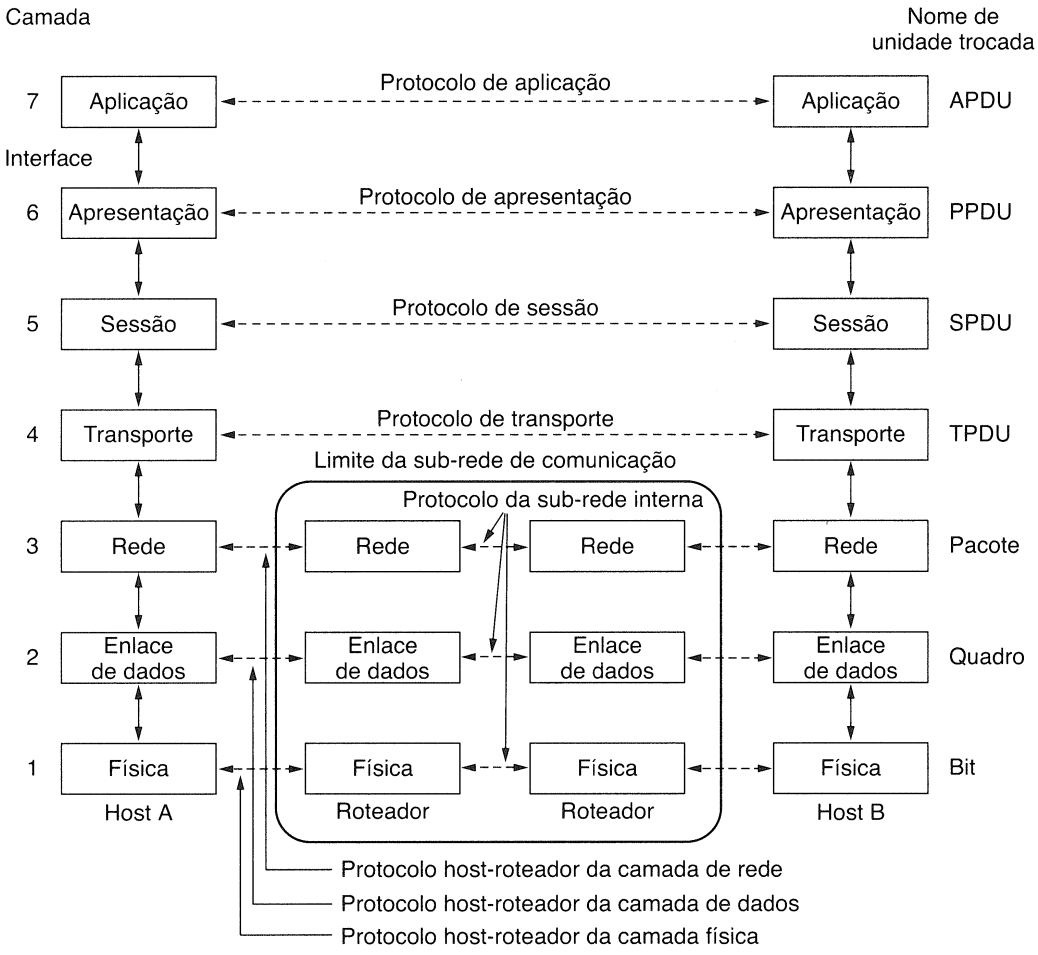

Figura 3.1: Pilha de Camadas do Modelo OSI retirado de Tanenbaum (2003)

Na Figura 3.1 está a distribuição das camadas de protocolos, de acordo com o modelo OSI, padronizado por Day e Zimmermann (1983). Protocolos são ordenados do maior ao menor nível de abstração, de cima para baixo. Um pacote criado tem

seu cabeçalho preenchido a partir da camada mais alta e descendo. Analogamente, o cabeçalho um pacote recebido começa na camada mais baixa fazendo seu caminho ao topo. A presença destas camadas varia de acordo com o escopo da rede. Em redes mais complexas, como a *internet*, todas as camadas se fazem presentes na comunicação. Já nós na fronteira da rede possuem protocolos de alto nível exclusivos a eles, enquanto intermediários, como roteadores, responsáveis por repassar pacotes aos *hosts* da rede, executam protocolos de nível mais baixo.

Visto que a *internet* é uma rede mais complexa, os protocolos mais relevantes para seu funcionamento operam nas camadas altas, a partir da camada de transporte. Nela, dois protocolos mais utilizados são o TCP e o *User Datagram Protocol* (UDP). Para o contexto desta seção, O TCP será o objeto de nossa análise. Por conta do volume do tráfego e a cobertura de grandes distâncias, a *internet* pode ter episódios de perda ou danificação de pacotes. O protocolo TCP foi concebido para controlar estas perdas na rede. Eddy (2022) afirma que o objetivo do protocolo consiste na detecção da perda de pacotes, como também em uma confiável retransmissão deles.

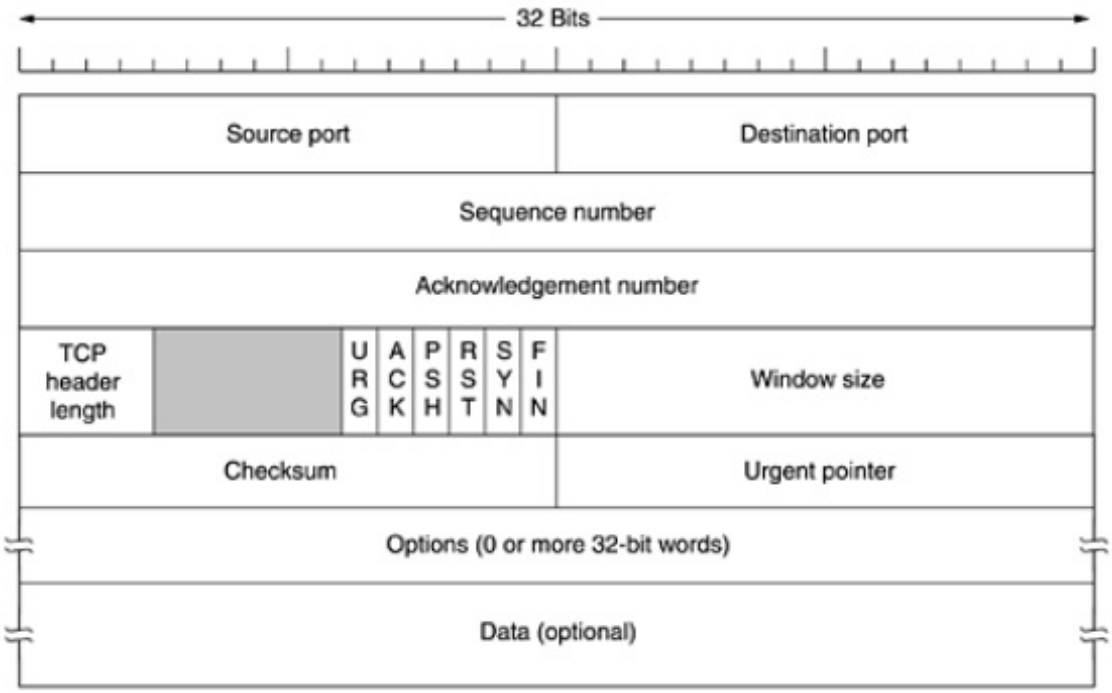

Figura 3.2: Cabeçalho de uma Requisição TCP, retirado de Tanenbaum (2003)

No cabeçalho de um pacote TCP, visto na Figura 3.2, é possível compreender

como seus dados de controle são relevantes para seu propósito. O *checksum* é uma sequência de *bytes* complemento a um do valor do pacote, de modo que a sua integralidade possa ser verificada por meio da soma dos dois valores. O *bit* FIN é responsável pela transmissão em partes do pacote. Um pacote com FIN setado em 0, traz consigo a informação de que há mais dados por vir. O receptor identifica o final deste envio por meio do último pacote, que vem com o *bit* FIN setado em 1.

O TCP também opera em conjunto com o IP, um protocolo a nível de camada de rede, para rotear os pacotes. O IP é responsável pelo endereçamento dos nós da rede. Contudo, um mesmo endereço é capaz de hospedar diferentes aplicações ao mesmo tempo. A fim de entregar os pacotes aos destinos corretos, os destinatários de uma rede são identificados por uma combinação de endereço IP e porta. É possível ver os campos *Source port* e a *Destination port* no cabeçalho TCP. Respectivamente eles representam a porta do emissor e do receptor para facilitar essa comunicação. A combinação do endereço com a porta, cria o que chamamos *socket*. Um par de *sockets*, um em cada nó, estabelece uma conexão.

Outro campo revelante é o ACK, um *bit* de confirmação da chegada do pacote enviado. Uma conexão TCP é estabelecida através do *three-way handshake*. Dados dois nós, o primeiro nó envia um segmento com número de sequência e o *bit* SYN igual a 1. SYN é uma *flag* de confirmação da requisição e de aceitação. Depois deste envio, o segundo nó responde com o seu próprio segmento com número sequência, ACK e SYN, confirmando o recebimento. Somente então, este mesmo primeiro nó responde com o ACK da chegada do número de sequência do segundo nó e a conexão está estabelecida. Um *bit* 0 no ACK é uma ocasião para o nó descartar o número de confirmação. No entanto, o envio do ACK pode ultrapassar o tempo de espera definido pelo nó. Em casos como este, ou na chegada de ACKs duplicados, o pacote deverá ser retransmitido.

Por conta da garantia de entrega dos dados, o TCP é a conexão utilizada pelo protocolo principal da *internet*, o *Hyper Text Transfer Protocol* (HTTP). De acordo com Nielsen et al. (1999), o HTTP atua na camada de aplicação, acima do TCP. Sua comunicação é uma dinâmica cujo nó, cliente, faz uma requisição a outro, chamado

servidor. Este servidor responde com seu pacote, atendendo às especificações do cliente.

Da mesma forma que outros protocolos, ele incrementa o cabeçalho do pacote com informações referentes ao propósito da requisição. Na Figura 3.3, podemos ver um exemplo de requisição HTTP, capturada com o *software Wireshark* [1]. Seu cabeçalho guarda informações sobre as características da transmissão. Em primeiro lugar, temos o seu método, descritos como verbos, *i.e.*, ações a serem aplicadas a um determinado do servidor. Todos os servidores devem suportar os métodos GET, HEAD e POST. Ambos os métodos são de requisição direta ao servidor. GET requisita um recurso, ou entidade, específica, que vem no corpo da resposta, ao passo que o HEAD requisita dados do cabeçalho e o POST envia dados para o servidor no corpo da requisição. O restante dos métodos, como o PUT, são opcionais.

Em segundo lugar temos o caminho do *site*, ou *path*, que é a localização específica do recurso dentro do *host* e a versão do protocolo HTTP utilizado. Todos estes campos são importantes para o endereçamento do pacote, já discutido anteriormente. *Connection* é um campo que diz respeito ao tipo de conexão estabelecida, mais adiante isso será detalhado. *Content-Type* e *Content-Length* guardam informações do corpo da requisição. O nó que recebe uma requisição precisa saber o formato do corpo, *e.g.*, texto ou binário, e seu tamanho.

```
GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.1
Host: gaia.cs.umass.edu
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/12
4.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0
.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate
Accept-Language: pt-PT,pt;q=0.9,en-US;q=0.8,en;q=0.7
If-None-Match: "80-61735f344d8ea"
If-Modified-Since: Mon, 29 Apr 2024 05:59:02 GMT
```

Figura 3.3: Exemplo de uma Requisição HTTP, capturada com *Wireshark*

A relação cliente-servidor do HTTP é classificada como *half-duplex*. A nível de enlace, os nós podem se comunicar unidirecional, chamado *simplex*, ou bidirecio-

[1] <https://www.wireshark.org/>

nalmente *duplex*. Uma comunicação *duplex* pode permitir comunicação simultânea entre os nós, *full-duplex*, ou ordenada *half-duplex*. O HTTP foi construído para uma comunicação *half-duplex*. O servidor responde somente quando cliente o requisita, tanto quanto o cliente não envia dados enquanto espera uma resposta do servidor. Os protocolos baseados no TCP também permitem esta característica. Contudo, mesmo que o servidor servidor sofra alguma alteração, ele não pode tomar a iniciativa no canal. Partindo da intenção emular uma comunicação *full-duplex* dentro da dinâmica cliente-servidor, começaram a surgir soluções para comunicações iniciadas pelo servidor.

3.1.1 *Polling*

Polling foi uma tentativa inicial promover uma comunicação *full-duplex* na *web* (SAINT-ANDRE et al., 2011). Seu modelo consiste no cliente enviar requisições HTTP ao servidor intercaladamente. O tempo maior de espera cria um cenário onde o servidor age sem ser explicitamente requisitado na ocasião. No entanto, esta espera uma hora acaba, forçando o servidor a responder com um corpo vazio. Somente após um breve aguardo, que o usuário faz o próximo *polling*.

Na seção 3.1, foi discutido como o protocolo TCP utiliza extensos cabeçalhos e o *three-way handshake* para criar uma conexão confiável. Para cada resposta do servidor, uma nova conexão precisa ser estabelecida com todos os cabeçalhos e verificações. Por conta disso, a latência do *polling* é alta, embora haja parâmetros no cabeçalho para mitigar essa repetição. O parâmetro *keep-alive* permite que cliente e servidor entrem em acordo, preservando a conexão aberto mesmo após o envio do pacote. Isso evita múltiplos *handshakes*, porém o esgotamento de tempo do *polling* continua existindo. Esse cenário pede a contínua troca de mensagens vazias e novas requisições, cujo cabeçalho maior do HTTP prejudica a velocidade de resposta.

O *polling* ainda passou por melhorias. Saint-Andre et al. (2011) acrescentam que o *Long Polling* permite que os servidores mantenham as chamadas dos clientes até haver modificações. Sem a obrigação de respostas vazias e uma conexão duradoura, o cliente pode enviar o próximo *long poll* logo após a resposta. No entanto, *polling*

ainda está restrito ao HTTP e, portanto, carrega desvantagens. O conhecido *overhead* de processamento do cabeçalho afeta mensagens menores, já que o grande cabeçalho do HTTP passa a ocupar a maior parte do tamanho do quadro. Em outra via, no contexto de uma conexão TCP, perda de pacotes e retransmissão é uma realidade inevitável. Na melhor das hipóteses, o tempo de devolução do servidor passa por três estágios: sua primeira resposta, um novo *poll* do cliente e então outra resposta. Mais ainda, a imprevisibilidade do *long polling*, transforma o cabeçalho *keep-alive* em um risco de *timeout*.

3.1.2 HTTP *Streaming* e *Server-Sent Events*

Uma outra alternativa para comunicação bidirecional veio na forma de HTTP *Streaming*, e sua variação *Server-Sent Events*. Roome e Yang (2020) definem HTTP *Streaming* como uma conexão indefinida fim a fim. Ela é análoga ao *long polling*, por ser uma requisição prolongada. A diferença está no reconhecimento de que todos os quadros são partes de um quadro maior. Esse comportamento é definido no cabeçalho *transfer-encoding* com o valor *chunked*. Entretanto, assim como *polling*, este método sofre com alta latência.

Server-Sent Events oferecem outra solução, enviando estas partes no disparo de eventos. Os dados servidor são chegam em pares de chave e valor, em eventos assíncronos chamados *server-push*. Esse modelo possibilita uma comunicação iniciada pelo servidor, porém esses eventos não operam em conjunto com intermediários de conexão, *e.g.*, *proxies*. Este problema também implica na dificuldade de separar o quadro em pedaços, uma vez que os *proxies* costumam fazer isso. Soluções como *Long polling* não possuem esse dificuldade, por abrirem múltiplas requisições ao custo de uma latência alta.

3.2 O protocolo *WebSockets*

WebSockets (WS) objetiva uma comunicação *full-duplex* (MELNIKOV; FETTE, 2011). É importante não confundir com o conceito de *socket*, definido anteriormente.

WebSockets é um protocolo, cujo posicionamento no modelo OSI, estaria uma camada acima do TCP. Uma vez que depende do estabelecimento da conexão via *three-way handshake*, o WS também precisa do HTTP para processar sua requisição. Em outras palavras, WS funciona com ajuda do mecanismo dos *sockets* de rede, assim como o próprio HTTP, já que o serviço é hospedado em um par de endereço e porta. A maior contribuição do WS é permitir, numa mesma conexão, que cliente e servidor recebam e enviem quadros simultaneamente um para o outro.

As subseções seguintes vão detalhar o padrão WS mais a fundo. Na subseção 3.2.1, o foco está na requisição WS e como ela se difere do HTTP, tanto na abertura quanto fechamento da conexão. Já na subseção 3.2.2, fala-se sobre o pacote WS e seu cabeçalho. Comparações do WS com outras soluções, encontram-se na subseção 3.2.3. E a subseção 3.2.4 comenta sobre as práticas e erros mais comuns na sua implementação.

3.2.1 Conexão *WebSocket*

A abertura de conexão do WS, observada na Figura 3.4, divide semelhanças iniciais com um pedido de conexão HTTP. Melnikov e Fette (2011) explicam que, como o protocolo busca construir suas funcionalidade em cima das fundações do HTTP, campos como URI, *host*, servidor e *origin* são comuns a ambos. Na verdade, WS se trata de uma requisição de evolução do HTTP pré-estabelecido. Para iniciar uma conexão WS, o cliente requisita uma mudança no campo *Upgrade*.

Os campos específicos do WS dizem respeito à segurança e identificação dos partes envolvidas. A maioria destes campos são opcionais, exceto pelo *Sec-WebSocket-Key*. Partindo do princípio de que o servidor pode enviar e receber pacotes fora da origem, ele distingue seus clientes por meio deste campo. Na resposta do servidor, demonstrado na Figura 3.5, o campo *Sec-WebSocket-Accept* é um meio de autenticar o cliente através do *Sec-WebSocket-Key* recebido. O servidor primeiro concatena esse campo com o *Globally Unique Identifier* (GUID) (LEACH; SAIZ; MEALLING, 2005). Depois um algoritmo de *hashing*, SHA-1, é aplicado e codificando com *base64* para então ser devolvido no campo mencionado. O *handshake* entre as partes termina

```
GET /chat HTTP/1.1
    Host: server.example.com
    Upgrade: websocket
    Connection: Upgrade
    Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
    Origin: http://example.com
    Sec-WebSocket-Protocol: chat, superchat
    Sec-WebSocket-Version: 13
```

Figura 3.4: Requisição WS inicial do Cliente, retirado de Melnikov e Fette (2011)

assim que o cliente verifica o campo *Sec-WebSocket-Accept*. Caso o campo esteja preenchido, ele pode começar a enviar os seus dados. Caso contrário, *i.e.*, o campo está vazio ou ausente, a conexão não será aberta.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Figura 3.5: Resposta WS inicial do Servidor, retirado de Melnikov e Fette (2011)

Assim como na abertura, as partes costumam seguir papéis específicos no fechamento da conexão. Enquanto *handshake* inicial parte do cliente, o encerramento é iniciado geralmente pelo servidor, mesmo que qualquer uma das pontas possa fazê-lo. Caso o cliente requisite o fechamento, ele ainda precisa esperar a confirmação do servidor. Em casos extraordinários, o servidor também pode encerrar a conexão durante o *handshake* inicial. Para finalizar a conexão, servidor envia junto com seu último pacote um comando para fechar a conexão no *opcode*, um conjunto de *bits* no cabeçalho do pacote. O quadro então é respondido pelo cliente com o seu próprio comando de fechamento. Após essa última troca, qualquer outro quadro enviado será descartado. Adicionalmente, as partes podem explicar o motivo do fim

da comunicação no corpo das suas requisições.

3.2.2 O pacote *WebSocket*

Após a conexão estabelecida, os nós podem trocar quadros entre si. Melnikov e Fette (2011) descrevem semelhanças a outros protocolos na sua estrutura, possuindo corpo e cabeçalho de controle. As diferenças estão nos dados que promovem a comunicação bidirecional. Cliente e servidor distinguem seus pacotes através de uma máscara nos dados. No cabeçalho presente na Figura 3.6, temos o *bit Mask*, sem o qual o servidor não aceita quadros do cliente. Analogamente, o cliente não aceita quadros do servidor cujo campo esteja preenchido. Em ambas as ocorrências, a conexão é finalizada pela parte que reconhecê-la.

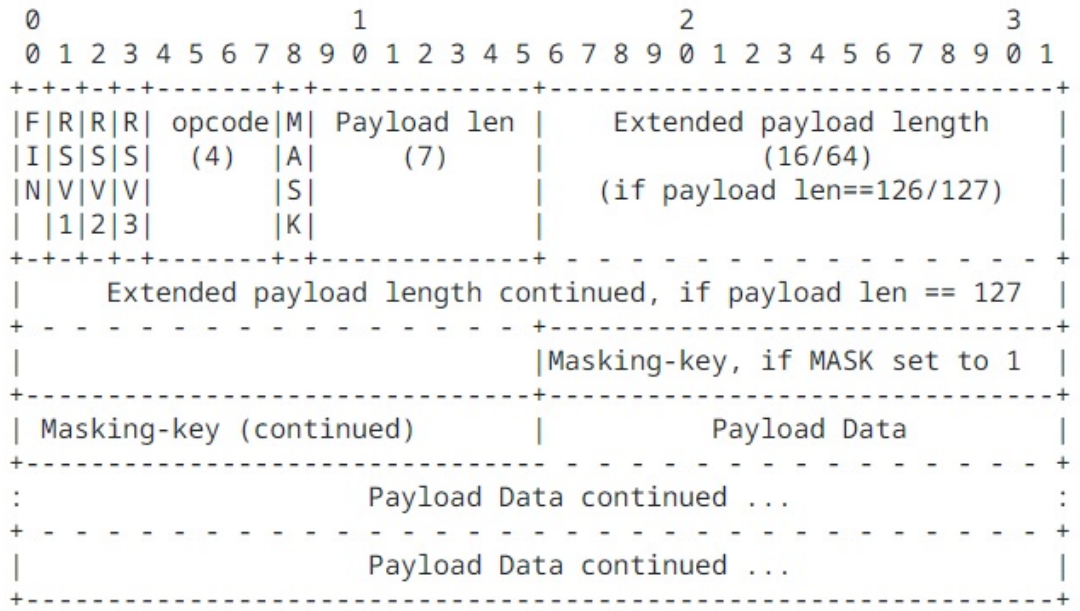

Figura 3.6: Exemplo de Quadro WS, retirado de Melnikov e Fette (2011)

Há outros *bits* individuais para representar estados da conexão. RSV1, RSV2 e RSV3 são *bits* para negociação de extensões do protocolo. Essas extensões também são calculadas no tamanho do pacote. Este tamanho é calculado em *Payload Length*. O *opcode* é um conjunto de quatro *bits*, responsável por interpretar o dado no corpo. Isto pode significar o tipo de dado, *e.g.*, binário ou texto, estado da conexão, aberta ou fechada, entre outras diversas características do pacote.

O cálculo da máscara depende de *Masking-Key*. Para o servidor, esse campo é necessário para decodificar o corpo do cliente. Para o cliente, é o contrário, o campo serve para codificar o corpo para envio. Esta codificação é feita através de uma operação XOR entre o octeto original, o *byte* a ser enviado, e a j-ésima chave da *Masking-Key*. O valor de j é o resto da divisão de i, o índice do octeto original, por quatro. Desfazer a máscara envolve a aplicação do mesmo cálculo ao octeto transformado.

Assim como no HTTP, um corpo de um requisição WS também podem ser grandes demais para um único envio. Portanto, o protocolo WS suporta fragmentos. Além do *bit* FIN, WS também conta com *opcode* para classificar fragmentos. Um quadro com FIN setado como 1 e um *opcode* diferente de zero não é um fragmento. Já uma mensagem com FIN e *opcode* definidos como 0, exceto pela última, é parte de uma mensagem maior. Outros dados de controle podem existir em mensagens de ambas as classes, embora não possam ser fragmentados, ou seja, todos cabeçalhos precisam portá-los.

3.2.3 Comparação entre as soluções

O trabalho de Murley et al. (2021) estudou a prevalência de diferentes soluções comunicação em tempo real na *web*. As tecnologias avaliadas foram *polling*, *Server-Sent Events* e *websockets*. Utilizando uma larga amostra de sites, as soluções foram ranqueadas de acordo com a frequência do seu uso, a origem das mensagens, como também quais tipos de aplicações utilizam mais cada solução.

A técnica de *polling* estava presente em 14% da amostra, sendo a maioria utilizando a sua forma regular ao invés do *long polling*. O cabeçalho de *long polling* esteve presente em apenas 4,5% das páginas. O autor destaca que não deixar os servidores esperando para entregar a requisição, pode ter sido um dos fatores que pode ter levado à maior adoção do *polling* tradicional. Igualmente, *Server-Sent Events* foram pouco contemplados pelos sites, cerca de 0,4% dentre os mil *sites* mais utilizados.

WebSockets estavam presentes em 6,3% dos *sites*. A maior parte deles estão envolvidos com mercados de ações, ocupando metade da amostra total. Abaixo deles

estão *sites* de compras, apostas e *chats*, todos com 10%. O autor atribui a preferência por WS ao uso de tecnologias de terceiros. Contando todos os *sites* que utilizam a *Application Programming Interface* (API), mais de 90% não constroem soluções próprias de comunicação em tempo real.

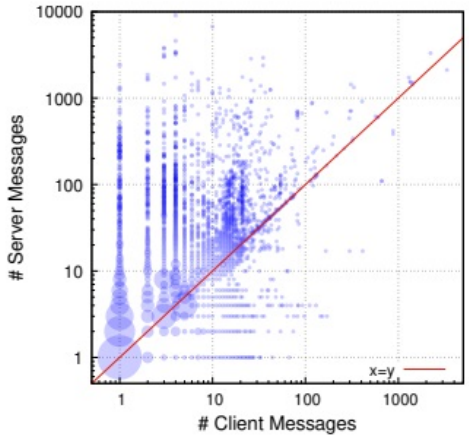

Figura 3.7: Comparação do total de envios feitos por Cliente e Servidor em WS, retirado de Murley et al. (2021)

O estudo também abordou o fluxo dos pacotes de WS nos *sites*. A Figura 3.7 indica que a maioria das mensagens trocadas no protocolo seguem na direção do servidor para o cliente, apesar da característica *full-duplex*. Em termos de comunicação de cliente para servidor, o estudo encontrou registros do seu uso para recuperação de dados direto do servidor. Os autores comentam que esta é uma maneira eficiente do ponto de vista da largura de banda. Contudo, ele julga solução redundante, uma vez que o HTTP possui a requisição GET. Alguns *sites* também usaram WS no envio mensagens de *ping* e *pong*, ainda que protocolo já ofereça suporte para essas mensagens sem necessidade de implementação. Essas ocorrências sugerem que os desenvolvedores *web* destes *sites* não possuíam conhecimento abrangente do protocolo.

Paralelamente, Murley et al. (2021) também avaliaram o ganho de largura de banda entre as tecnologia. O ganho do WS nesses experimentos em relação às alternativas foi cerca de 16Mbps. Isso pode ser atribuído ao tamanho padrão dos quadros dos protocolos. Os cabeçalhos HTTP de envio possuem 184,9 *bytes* de tamanho, enquanto os de resposta têm mais que o dobro do tamanho, em 403,1 *bytes*.

Um pacote WS comum tem cerca de 82 *bytes*, sendo o cabeçalho 8 *bytes* e o corpo 74 *bytes*. Vale lembrar que também não há requisições sucessivas do cliente como no HTTP. Em caso de *long polling*, a mesma requisição custaria 499 *bytes* para cada atualização.

Na época da escrita do seu estudo, Murley et al. (2021) afirmam que 45% dos sites analisados usavam o HTTP 2.0. Esta versão deprecia a funcionalidade de *server-push*, característica da comunicação *Server-Sent Events*. Diante da constante adesão de versões mais evoluídas dos protocolos, e dos problemas apresentados por soluções como o *polling*, WS se torna a solução mais prevalente na *web*.

3.2.4 Segurança e Más Práticas

A maior adesão de WS no mundo *web* também carrega desvantagens. Principalmente no âmbito da comunicação bidirecional, o protocolo possui riscos de implementação e uso malicioso, classificados por Murley et al. (2021). Por meio de WS, é possível acessar informações sensíveis construindo um registro do rastro digital dos usuários. Isso é feito interconectando *cookies* de diferentes páginas em um único perfil. A comunicação direta do servidor para uma página específica também pode abrir brechas para o envio de *spams* e *malwares*, embora essa vulnerabilidade não seja exclusiva desta tecnologia.

Além do uso perigoso de WS, um sistema baseado nele também pode apresentar vulnerabilidades. Na seção 3.2.3, comentamos como a maior parte das aplicações utilizam soluções prontas, *e.g.*, *socket.io* e *socketjs*. A princípio, elas dão acesso ao servidor a partir de qualquer origem por padrão, apesar de terem funcionalidades embutidas para limitar isso. Permitir acessos externos de qualquer origem pode ser também uma estratégia de coleta de dados, mas também abre para perigos de origem desconhecida. A escolha por essa permissão depende do escopo da aplicação. *Sites* de *marketing* e rastreamento entendem a permissão do *Cross-Origin Resource Sharing* (CORS) como uma boa prática. Por outro lado, jogos e mineração de criptomoedas protegem seus sites contra origens estrangeiras.

A conclusão é de que estas vulnerabilidades estão associadas à falta de informação.

Sem conhecimento sobre soluções prontas, desenvolvedores são obrigados a aprenderem por vias informais. Exemplos deste uso descuidado podem ser encontrados na exposição de 14.1% dos *sites* ao pedido de *upgrade* da requisição externos. Alguns destes sequer utilizam *WebSockets*, mas mantém um nível extra de exposição desnecessária. Aprofundar os conhecimentos não apenas do protocolo, como também das soluções prontas que o implementam, pode ser o caminho para evitar o lançamento de aplicações vulneráveis na web.

Capítulo 4

Proposta de uma Arquitetura para *Chat* utilizando *WebSockets*

Neste capítulo será discutida a proposta de construção do *chat* para um SME. A princípio, a seção 4.1 explica a motivação por trás do desenvolvimento da aplicação. Depois, a seção 4.2, outras construções de *chat* serão analisadas, dada sua arquitetura e abordagem para comunicação bidirecional. A seção 4.3, serão definidos os requisitos e o escopo da aplicação. As decisões tomadas a respeito dos pilares da aplicação de dados, assim como suas justificativas estão na seção 4.4. A estrutura e organização desses fatores também serão explicadas com detalhes na seção 4.5.

4.1 Motivação

Conforme descrito por Sipos (2017), SME, ou sua versão na língua inglesa, *Graduate Tracking Systems* (GTS), é uma iniciativa para acompanhar o desempenho de estudantes universitários após a conclusão de seus respectivos cursos. Esse processo pode ser conduzido por instituições públicas, privadas ou organizações internacionais. No passado, o ensino superior era um espaço onde poucos ingressavam. Mesmo dentro, o lugar tinha seus próprios ritos e formação cultural, distantes da realidade de fora. Sipos (2017) descreve esse cenário na metáfora de uma torre de marfim. Nas últimas décadas, universidades passaram por uma mudança de paradigma. A

crescente demanda por formação superior pede que as instituições busquem mudanças para fora de si. Um dos interessados neste processo foi o próprio governo. Por parte do Estado, a expansão de instituições de ensino superior privadas criou uma demanda por um controle de qualidade unificado. SMEs nasceram para encurtar a distância entre a experiência acadêmica e a de trabalho.

Dependendo da instituição encarregada, as características de um SME mudam. A cultura dos países envolvidos desempenha um papel importante também. Países que possuem uma tradição de coleta de dados dos seus egressos, como Estados Unidos e Nova Zelândia, têm pesquisas direcionadas pelo governo. O objetivo dessas pesquisas é avaliar o retorno do investimento em ensino superior. O tempo da pesquisa também é um fator importante para a medição. Institutos podem rastrear o primeiro ano após a formação, os próximos cinco, ou somente dez ou vinte anos depois. Numa via diferente, Paul (2015) descreve o *Carrer after Higher Education: a European Research Study* (CHEERS), conduzido em países europeus, como um rastreamento do processo de transição dos egressos. As perguntas envolveram a primeira fase da carreira, semelhanças entre trabalho e universidade, além do grau de satisfação com seu atual estado.

Essas pesquisas são conduzidas majoritariamente por questionários em *e-mails*. Durante seu tempo na universidade, estudantes deixam seu contato nas redes internas da instituição. Nos primeiros anos após a sua partida, esses dados guardam alguma relevância. Contudo, na medida em que as demandas por um acompanhamento prolongado surgem, a possibilidade de que os canais salvos sejam ainda ativos diminui. Egressos podem trocar de celular, endereço, conta nas redes sociais, entre outros meios de comunicação. A manutenção deste contato com os alunos ao longo dos anos é o grande problema dos SMEs. Michael et al. (2012) comentam que:

> Exceto pelo comprometimento pessoal e altruísta de alguns egressos, a questão central sempre foi garantir o interesse permanente dos alunos em suas respectivas instituições. [...] Se ao menos eles recebessem outros benefícios, como informações sobre eventos, oportunidades de continuar sua formação ou vagas de emprego, além de trocar mensagens com

> antigos pares, há uma maior probabilidade de que eles respondam aos questionários. (MICHAEL et al., 2012, p.43, tradução própria)

Redes sociais como *Facebook* e *LinkedIn* também desempenham papéis nesse cenário. Através de grupos de trabalho ou promessas de *networking*, órgãos e instituições encarregadas da pesquisa atraem os egressos a contribuir com a atualização de seus dados. Uma vez que a relação entre redes sociais e SMEs é separada, unir esses dois ambientes é imprescindível. Uma plataforma da mesma natureza está sendo desenvolvida para a Universidade Federal Rural do Rio de Janeiro. A fim de manter os egressos engajados, se faz necessário o desenvolvimento de um meio para os usuários se comunicarem na plataforma. Tal qual as redes sociais de seus cotidianos, a plataforma precisa suportar um sistema de comunicação em tempo real entre os membros desta plataforma. Por meio deste sistema, a abrangência de interesses do egresso se torna considerável. A plataforma, uma vez completa e com este recurso, seria capaz de proporcionar comunicação até para além dos discentes. Além disso, o sistema serviria também para trocas entre outras instituições de pesquisa, ou empresas ofertando vagas.

4.2 Trabalhos Relacionados

Sistemas de comunicação em tempo real são ofertados e desenvolvidos de inúmeras formas. As soluções para empresas interessadas nessa integração em na forma de plataformas de *Chat as Service*. O *Sendbird* [1] é um exemplo desse tipo de sistema. Seus recursos abrangem tanto uma aplicação completa de *chat*, quanto a integração com outros sistemas via API ou biblioteca de interface. Serviços adicionais como *chatbots* e moderação de conteúdo também estão inclusos. O *Sendbird* também age como intermediário para outros recursos, como *Whatsapp*, *KakaoTalk* ou *Short Message Service* (SMS). Semelhante ao *Sendbird*, o *Comet* [2] também é um *chat* como serviço. Ele possui a capacidade de traduzir, sugerir respostas automáticas ou mencionar mensagens individualmente. Seus recursos para moderação envolvem

[1] <https://sendbird.com/>
[2] <https://www.cometchat.com/>

detecção de imagens inseguras e sentimento por trás do texto escrito por meio de Inteligência Artificial. A nível de usuário, além de opções para reportar, bloquear, é possível filtrar mensagens com determinadas palavras de risco.

Ao mesmo tempo, soluções próprias ainda podem ser desenvolvidas. Yusoff e Selliah (2017) identificaram uma demanda por atendimento em tempo real no cliente no setor de serviço. Seu trabalho desenvolve um sistema de *chat web* com *Meteor* [3] e *Yeoman* [4]. Os dados são salvos no *MongoDB* [5], sistema gerenciador de banco de dados não-relacional. Já Obadjere (2020) desenvolveu um *chat* focado na experiência da língua franca na Nigéria. Apesar de ter sido desenvolvido com ferramentas semelhantes ao trabalho de Yusoff e Selliah (2017), como *JavaScript* e *MongoDB* para o *back-end*, o desenvolvimento da aplicação do cliente foi orientado para celulares, utilizando *Flutter* [6].

4.3 Requisitos Funcionais e Não-Funcionais

A proposta deste trabalho é um sistema de comunicação em tempo real, sobre o contexto de uma plataforma para egressos já em desenvolvimento, chamado *alumni*. Atualmente, *alumni* possui interfaces para empresas e egressos. Esses usuários possuem seus próprios perfis, onde empresas inserem e divulgam vagas de emprego na plataforma. O egresso, por sua vez, tem a experiência ilustrada pela Figura 4.1, onde seu perfil é capaz de pesquisar por essas vagas de emprego e se candidatar a elas. A partir desta classe de usuários egressos que o sistema de comunicação em tempo real será construído, através de um *chat*.

Para construir o *chat*, primeiro se faz necessário estabelecer seus requisitos. Sommerville (2011) define requisitos como a definição do que o sistema deve fazer tal como suas limitações. Requisitos podem ser funcionais ou não-funcionais. Requisitos funcionais declaram o que o sistema deve ou não fazer, abrangendo as exigências das mais até as menos específicas. Requisitos não-funcionais são restrições gerais

[3] <https://www.meteor.com/>
[4] <https://yeoman.io/>
[5] <https://www.mongodb.com/>
[6] <https://flutter.dev/>

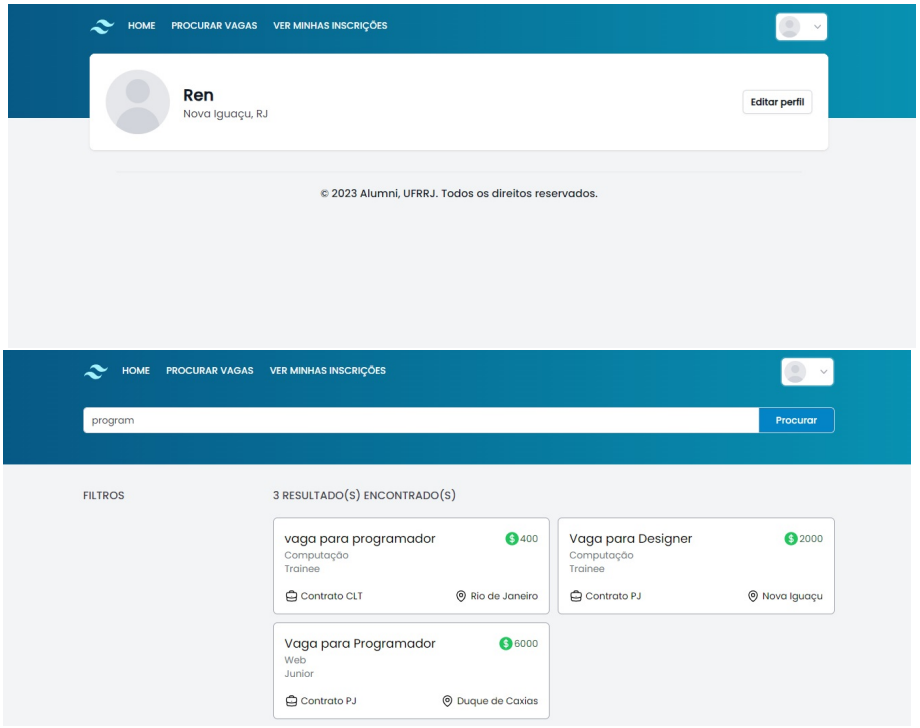

Figura 4.1: Perfil do usuário egresso e Busca por Vagas de Emprego

impostas ao comportamento do sistema. Especificamente para o sistema do *chat*, convém definir os requisitos que respeitem os pilares da aplicação de dados.

Entre os requisitos funcionais, o *chat* deve permitir o envio e recebimento de mensagens. Concomitante a essa característica, o sistema deverá permitir mensagens além do texto escrito. O usuário poderá anexar arquivos, áudios ou imagens. Todas essas mensagens serão enviadas e recebidas através de conversas, um canal específico entre duas ou mais pessoas. Usuários deverão ser capazes de acompanhar as suas conversas, por meio de uma lista de contatos. Esta lista de contatos deverá permitir que seus usuários criem seus próprios canais de comunicação, denominados grupos, adicionando qualquer usuário já existente nesta lista. Paralelamente, o sistema deve permitir a busca e filtragem por contatos e grupos previamente salvos.

Requisitos não-funcionais, por se tratarem do comportamento do sistema, estão mais ligados com as demandas dos pilares da aplicação de dados. Sommerville (2011) descreve-os como propriedades emergentes da aplicação. A **Confiabilidade** do *chat* depende que o envio e recebimento de mensagens seja em tempo real. Ademais, as mensagens devem chegar somente para os usuários correspondentes e na ordem em

que são enviadas. Analogamente, a inclusão de um usuário em qualquer grupo deverá ser notificada também em tempo real. A nova conversa deverá aparecer na lista de contatos, igualmente ordenada com base em quão recentemente ela foi atualizada. Em termos de performance e **Escalabilidade**, a latência para o envio, registro e exibição das mensagens deverá ser baixa, dada a característica simultânea dos canais.

4.4 Modelagem do Sistema

Os requisitos funcionais do sistema possibilitam que abstrações acerca do *chat* sejam feitas. As abstrações por sua vez, serão compreendidas como entidades deste sistema. Em primeiro lugar, existe o **Usuário**, *i.e.*, o ator principal do sistema. Esta entidade representa a mesma classe do sistema *Alumni*. O diagrama de classes, visto na Figura 4.2, se trata de uma extensão do projeto anterior, exceto por alguns atributos do **Usuário** no projeto original que foram omitidos por questão de clareza. Será por meio do **Usuário** e para ele que as conversas acontecem. Em segundo lugar, o **Usuário** precisa se comunicar por meio de mensagens. Uma **Mensagem**, enquanto entidade, é definida pela sua variedade de formatos, *e.g.*, textual, áudio ou arquivo. Por sua vez, **Mensagens** são ligadas por um canal, contendo dois ou mais **Usuários**. Este canal será uma entidade chamada **Conversa**.

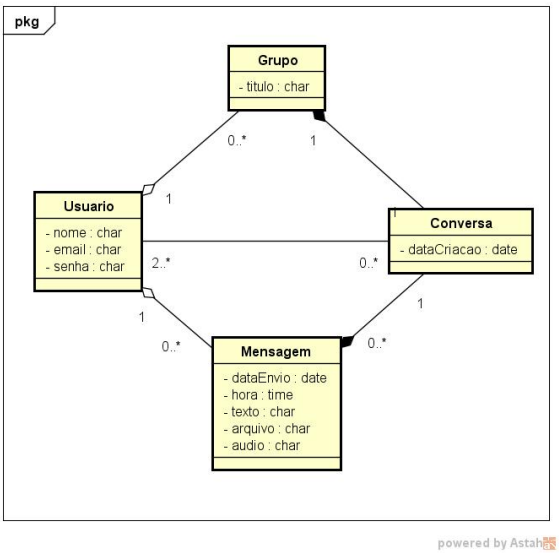

Figura 4.2: Diagrama de Classes do Sistema de *Chat*

A entidade **Usuário** guarda dados credenciais, como "*e-mail*" e "senha", como também seu "nome". **Conversa**, enquanto abstração, permite pelo menos duas possíveis modelagens. Uma ideia inicial seria a entidade representar um relação N:N, *i.e.*, muitos para muitos, entre **Usuários**. Na prática, a **Conversa** seria uma entidade intermediária, nascida da cardinalidade da relação entre **Usuários**. Entretanto, a escolha para este sistema foi de tornar **Conversa** uma entidade em si própria. O motivo está no fator limitante de abstraí-la como uma entidade intermediária. Neste caso, o relacionamento N:N entre **Usuários**, os forçaria a se relacionariam aos pares. Essa característica estaria contra nosso requisito funcional de conversas entre múltiplos **Usuários**. Enquanto entidade independente, **Conversa** salva sua data de criação, no campo "dataCriacao", e os **Usuários** envolvidos.

Informações adicionais de uma **Conversa**, *e.g.*, seu título, não estão explicitamente inseridas na entidade, porque conversas dois a dois serão referidas pelo nome do contato. No entanto, conversas em grupo pedem nomes diferentes. Graças a isso, uma entidade de **Grupo** foi pensada para compor a entidade **Conversa**. **Grupo** é uma entidade auxiliar, um atributo opcional de **Conversa**. Portanto a relação entre ambas as entidades é 1:1. Um **Grupo** guarda o "título" da conversa, além de agregar seu **Usuário** criador. Assim como **Conversa**, o **Usuário** também possui uma relação 1:1 com **Grupo**. Com exceção destes atributos, os canais de comunicação individuais ou coletivos são a mesma entidade no sistema. Logo, o requisito da lista de contatos será cumprido, trazendo as conversas atreladas ao **Usuário** logado.

Dada uma **Conversa**, uma **Mensagem** guarda a data e hora de envio e o texto escrito. Estes dados estão representados por "dataEnvio", "hora" e "texto", respectivamente no diagrama. Arquivos de imagem ou outras terminações tem a referência de seus arquivo salvos, guardada no atributo "arquivo". Esta referência é um caminho para a localidade destes arquivos nas pastas do projeto. Áudios são guardados da mesma forma, embora em uma variável diferente. Em termos de relações, **Mensagem** possui relação 1:N tanto com **Usuário** quanto **Conversa**. Este foi outro fator determinante para a decisão de modelar **Conversa** como uma entidade independente. No contexto em que **Conversa** é a relação N:N de **Usuários**, referenciar uma **Mensagem** em seu escopo exigiria a constante inserção do par

de chaves únicas de ambos os **Usuários** envolvidos. Essa modelagem não apenas tornaria parte dos registros da base redundante, mas também sua escrita um pouco maior. Estes fatores poderiam ferir o pilar da **Escalabilidade**, como também de **Manutenibilidade**, caso futuras expansões exigissem mais relacionamentos para **Conversa**.

4.5 Arquitetura do Sistema

De posse da modelagem do sistema, se faz necessário planejar como será sua arquitetura. A aplicação é um *chat* em *web*, portanto seus recursos serão fornecidos através da dinâmica cliente e servidor. Sua *interface* será uma página *web*, capaz de requisitar outros recursos do servidor. A proposta de arquitetura, observado na Figura 4.3, descreve esta separação em dois serviços principais: o *chat* e a API. A *interface* do *chat* constitui o *front-end* da aplicação, enquanto a API, o *back-end*. Nesta seção, será estudada todas as partes deste sistema e as decisões envolvidas nelas. A seção 4.5.1 trata de como a modelagem estará representada no *back-end*, principalmente no banco de dados. Depois, na seção 4.5.2, será discutida a *interface* do *chat* e como ela trata as informações vindas do *back-end*.

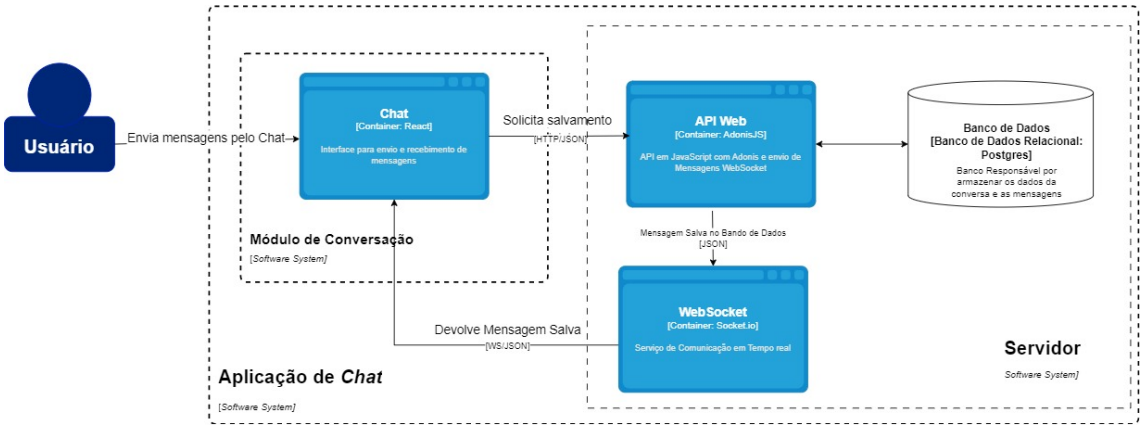

Figura 4.3: Diagrama da Arquitetura do Sistema

4.5.1 Servidor *Web*

O fluxo dos dados da aplicação começa no *front-end*, para ser processado no *back-end*. O chat foi modelado como um *web-service* do tipo *Representational State Transfer* (REST) (FIELDING, 2000). Esse padrão utiliza os métodos do protocolo HTTP, *e.g.*, GET, POST e PUT, vistos na seção 3.1, para criar uma *interface* de manipulação de dados através do estado das entidades. Cada entidade terá sua rota de acesso aos seus recursos, cujas operações estarão atreladas aos respectivos métodos. O método GET servirá para busca e recuperação de informação. Um dos requisitos não-funcionais, que diz respeito ao ordenamento das **Mensagens** e **Conversas** será realizado na resposta a uma requisição GET. Os métodos POST e PUT serão para salvamento ou atualização da entidade. Por último, DELETE é o método para a destruição da entidade.

Essas operações são todas realizadas no banco de dados. O estado que manipulamos são os registros que podem ser salvos de duas formas. Na seção 2.2, foram apresentadas alternativas de bancos relacionais ou não-relacionais. Nossa modelagem possui algumas relações de composição, isto é, entidades dentro de outras, sem as quais sua existência perde o sentido. Não há **Mensagem** sem o contexto da **Conversa**. Da mesma forma, não há um **Grupo** se primeiro, não houver uma **Conversa**. O relacionamento entre **Mensagem** e usuário também segue uma estrutura hierárquica, 1:N. Essas características não impedem a escolha de um banco não-relacional, porém a escolha por **Conversa** ser uma entidade própria cria uma relação N:N com o **Usuário** favorece o modelo relacional. Relações N:N provocam uma maior duplicidade de informação em bancos não-relacionais, provocando múltiplas rescritas em cada mudança, por menor que seja. Bancos relacionais, com sua capacidade de unir tabelas e referenciar por chaves, simplificam o processo de registro e atualização e unificam o processo de busca. Há apenas um registro, referenciado em vários lugares. Devido a prevalência de relacionamentos, especialmente N:N, foi optado por um *schema* relacional.

Outro possível motivo para adesão do banco não-relacional seria sua flexibilidade. Uma **Mensagem** vem em vários formatos, mudando não apenas seu registro como

apresentação. Essa característica combina com a capacidade do banco não-relacional de não forçar a estrutura do *schema* durante a escrita. Contudo, as demais entidades do sistema, **Conversa** e **Usuário**, possuem *schemas* bem definidos, cuja imprevisibilidade é baixa. No mais, o *chat* está sendo desenvolvido em um sistema pré-existente. A plataforma *alumni* construída na *web* já registra seus usuários relacionalmente. Havia a possibilidade de uma implementação híbrida, porém a implementação e a **Manutenibilidade** se tornaram mais simples com o reaproveitamento das estruturas existentes.

Além das requisições HTTP, o servidor também será responsável por enviar pacotes do WS. A principal característica que desejamos no *chat* é a comunicação *full-duplex* de baixa latência em tempo real, ambos requisitos não-funcionais. Para obtê-la, o servidor precisa notificar a todos os **Usuários** presentes numa **Conversa** da nova **Mensagem**. Esta **Mensagem** vem na forma de uma requisição POST, porém a única máquina a receber a resposta dela será a que enviou o pacote. Na seção 3.2, vimos como *WS* nos permite criar um canal na direção oposta, do servidor ao cliente. Portanto, além de responder às requisições do cliente, o servidor deve disparar notificações do *chat* aos usuários, de acordo com os requisitos funcionais. Isso inclui outros eventos, *e.g.*, criação e alteração de *Grupo*, além de inserção e deleção de **Usuários** nele.

4.5.2 Módulo de Conversação

O *front-end* se encarrega de iniciar requisitos funcionais da aplicação. Portanto, as funcionalidades de criar **Conversas**, enviar **Mensagens** ou modificar **Grupos** começam por este módulo. A *interface* do *chat* se comunica com a API, de modo a persistir as operações do sistema. Contudo, o módulo de conversação precisa ser utilizado em tempo real. Usando HTTP como exemplo, se para cada nova mensagem, enviada ou recebida, uma nova requisição da página fosse necessária, o *front-end* estaria constantemente recarregando. Dependendo da quantidade de trocas, é possível que o usuário fique mais tempo esperando a página pronta do que lendo ou enviando mensagens. Para a transferência das mensagens, em vez de

toda a página, já será usado WS. Mas para carregar novos recursos, este módulo será implementado em *Single Page Application* (SPA). Uma sistema classificado como SPA nunca é recarregado completamente. Suas atualizações são sob demanda, partindo do cliente, com atualizações pontuais.

Na seção 2.3, foi comentado sobre os módulos não precisam necessariamente estar escritos na mesma linguagem de programação. Portanto, a representação das entidades do sistema precisam estar serializadas. Há duas alternativas de serialização na *web*: JSON e XML. O *chat* desenvolvido trabalha com objetos, com dados em formato textual. **Mensagens** salvam seu texto, ocasionalmente *bytes* de arquivos. Valores não textuais, serão identificadores no banco que consistem em números inteiros. A ambiguidade dos dados em formato JSON não será um problema para este domínio. Por outro lado, a estrutura mais complexa e verborrágica do XML torna a sua escrita e conversão custosa, principalmente para dados menores como os que vamos transportar. Além disso, o sistema no qual estamos construindo o *chat* já possui um *back-end* em *JavaScript*. Uma vez que JSON é parte do *JavaScript*, não haverá custo adicional de código para a conversão.

A comunicação em WS também será feita com JSON. No caso do envio de uma mensagem, o cliente cria um objeto JSON e faz uma requisição POST para o servidor. A princípio, a resposta desta requisição seria salva na linha do tempo do *chat*. No entanto, convém levar em conta as mensagens que chegariam neste meio tempo via *WS*. Por causa disso, mensagens chegam pela via única da subscrição de um evento no protocolo *WS*. O protocolo WS desempenha um papel importante na manutenção das entidades mais recentes disponíveis na *interface*, assim como na característica de SPA do *front-end*. Um **Usuário** pode ser inserido ou excluído de um **Grupo** por um terceiro, ou o título do mesmo do **Grupo** ser modificado sem o seu conhecimento. Em todos estes casos, a requisição inicial é feita através da API, enquanto a atualização em cada *interface* é disparada por um evento *WS*. A função do *front-end* destacar essas atualizações nas listas de contatos e *chats*, preservando o requisitando não-funcional de ordenação. Da mesma forma, as **Mensagens** são agrupadas por **Usuário**, dia e hora, também pelo *front-end* na medida em que chegam.

Capítulo 5

Implementação

O objetivo deste capítulo é detalhar a implementação do sistema de comunicação em tempo real. Esta é uma aplicação de *chat web*, portanto há uma série de opções para construí-la, dado os desafios específicos de tal ambiente. O primeiro desafio são as tecnologias escolhidas. Na seção 5.1, será exibida as escolhas das ferramentas e a razão por trás delas. Depois, na seção 5.2, será discutida a arquitetura de pastas e arquivos do código, em especial no *back-end*. Do lado do *front-end*, a seção 5.3 explicará a implementação das interfaces, aprofundando-se na apresentação de telas, suas funcionalidades, bem como o cumprimento dos requisitos funcionais.

5.1 Tecnologias

São necessárias tecnologias para implementar a proposta do capítulo anterior. Tanto no *back-end* quanto no *front-end*, a escolha de uma linguagem de programação ou biblioteca podem ser a diferença na **Manutenibilidade** do código. A escolha pelo *framework* do *back-end* encontra-se na subseção 5.1.1. O Sistema Gerenciador de Banco de Dados também será abordado na subseção 5.1.3. O *framework* do *front-end* será explicado na subseção 5.1.4. Ainda na visualização, a subseção 5.1.5 discutirá a biblioteca de estilização, já que se trata de uma aplicação *web*. A subseção 5.1.6 elabora mais sobre a tecnologia que integra ambos os módulos do *site*.

5.1.1 *Adonis*

Adonis [1] é um *framework* em *TypeScript* [2], uma linguagem de programação construída em cima do *JavaScript* [3], para *Node.js* [4]. Sua estrutura *Model View Controller* (MVC) permite a criação de uma aplicação *web* completa, desde a parte de comunicação com o banco de dados até a renderização visual. Este último, o *framework* oferece até mesmo *templates* para reutilização de recursos visuais, porém, para o caso do *chat*, *Adonis* estará relegado ao *back-end*. Será neste *framework* em que a API estará implementada, baseada no redirecionamento das requisições HTTP para aos serviços adequados ao modo REST*ful*. Por conta disso, os recursos de gerenciamento do *Adonis* será bastante caro à API. Na Figura 5.1, é possível ver um exemplo da definição das rotas. Cada rota está associada a um controlador e a um nome a ser usado de referência. Campos onde um ":" o antecede representam parâmetros esperados na rota.

```
41  Route.group(() => {
42    Route.get('/audio/:id', 'MensagemsController.getAudio').as('mensagens.audio')
43    Route.post('/:id', 'MensagemsController.store').as('mensagens.store')
44    Route.put('/:id', 'MensagemsController.update').as('mensagens.update')
45    Route.get('/conversa/:id/:limit?/:offset?', 'MensagemsController.findLimitOffset').as('mensagens.find')
46    Route.get('/anexo/:id', 'MensagemsController.getImage').as('mensagens.image')
47  })
48    .prefix('mensagens')
49    .as('mensagens')
50    .middleware(['auth'])
```

Figura 5.1: Exemplo de Rotas no *Adonis*

A opção por este *framework* veio por duas principais razões. A começar pela sua arquitetura MVC, *Adonis* possui recursos de ORM discutidos anteriormente, o que faz dele capaz de integrar com o banco de dados de maneira sólida. Depois, e mais importante, o projeto do *chat* é uma evolução de uma plataforma *alumni* em desenvolvimento utilizando *Adonis*. Por uma questão de **Manutenibilidade**, tornou-se oportuno aproveitar do código pré-existente para implementar este projeto.

[1] <https://adonisjs.com/>
[2] <https://www.typescriptlang.org/docs/>
[3] <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
[4] <https://nodejs.org/en>

5.1.2 *Socket.io*

O servidor em *Adonis* processa as requisições, porém a comunicação bidirecional entre servidor e cliente é feita por WS. *Socket.io* [5] é uma biblioteca compatível com *JavaScript* capaz dessa comunicação bidirecional com baixa latência, além de baseada em eventos. Apesar do nome, a biblioteca não implementa exatamente o protocolo WS. Na verdade, *socket.io* seleciona automaticamente a melhor conexão disponível dependendo da circunstância, seja *long-polling* ou *web-sockets*. A biblioteca também fornece dados adicionais no cabeçalho dos pacotes, de modo que promova maior controle sobre a conexão, sem prejudicar a performance.

5.1.3 *Postgres*

Apesar de testes iniciais realizados no *SQLite* [6], a escolha pelo Sistema Gerenciador de Banco de Dados (SGBD) culminou *Postgres* [7]. Ambos os SGBD são capazes de persistir as informações do chat, porém fatores como **Escalibilidade** e **Confiabilidade** balizaram a escolha. No âmbito da *Confiabilidade*, *Postgres* cria um ambiente tolerante à falhas, cumprindo os requisitos *Atomicity, Consistency, Isolation, Durability* (ACID), conforme descritos por Härder e Reuter (1983). No lado da **Escalabilidade**, o suporte ao *sharding* permite dividir a tabela em partes menores entre servidores. Isto é uma característica importante para a aplicação caso ela evolua no futuro com um número crescente de usuários.

5.1.4 *React*

Existem inúmeras tecnologias para a implementação de um SPA. O *Alpine* [8] pode carregar elementos na tela em tempo, inserindo *scripts* dentro das *tags Hyper Text Mark-up Language* (HTML). No entanto, por conta da comunicação tempo real, a aplicação demanda por comportamentos mais detalhados. O *Angular* [9] oferece

[5] <https://socket.io/>
[6] <https://www.sqlite.org/>
[7] <https://www.postgresql.org/>
[8] <https://alpinejs.dev/>
[9] <https://angularjs.org/>

uma estrutura baseada em componentes para *web*, com a possibilidade de expandir para classes de serviço para tratar os requisitos. Mas a solução encontrada para implementar o *front-end* foi o *React* [10], um *framework* em *JavaScript* assim como o *Angular* e o *Alpine*. Assim como *Adonis*, esses *frameworks* são voltados para o desenvolvimento *web*, porém com a diferença de ser orientado para o *front-end*. Mais especificamente, eles focam na programação de componentes. A maior parte das aplicações de grande porte utilizam a arquitetura MVC, sobre a qual ainda será falada. Contudo, *React* e *Angular* separam suas funcionalidades a nível dos componentes visuais da aplicação. Em vez de abstrair as funcionalidades de uma entidade, *e.g.*, usuário ou item, *React* propõe uma abstração para a função de botões, listas ou telas.

A decisão pelo *React* veio pela sua abordagem, que promove maior **Manutenibilidade**. Os códigos são menos extensos, por estarem pulverizados em componentes específicos, reduzindo a complexidade da escrita. Por consequência, o rastreio de erros se torna facilitado. Além disso, a possibilidade de importar e adaptar soluções prontas de outras bibliotecas torna o tempo de planejamento e construção de futuras melhorias menor. Durante a pesquisa para essas tecnologias, bibliotecas foram comparadas para identificar qual serviria às necessidades do sistema. Foi encontrado um *Standart Development Kit* (SDK) para *chat* do *Angular* [11], além de bibliotecas que emulam o *Messenger* do *Facebook* [12], mas foi no *Chatscope* em que a melhor solução foi encontrada.

O Chatscope [13] é uma biblioteca com vários componentes para interface de um *chat*, *e.g.*, caixas de texto, mensagem ou linha do tempo de conversas. Todos estão propriamente encapsulados e com seus principais eventos abstraídos. Ao mesmo tempo, seus recursos possibilitam a personalização de seus comportamentos e visuais. Dentro de um mesmo componente, no código 5.1.4, somos capazes de implementar vários outros pré-concebidos do *Chatscope*.

Apesar de suas inúmeras utilidades, *Chatscope* não possui tudo o que nossos

[10] <https://react.dev/>
[11] <https://www.npmjs.com/package/stream-chat-angular>
[12] <https://github.com/sejr/react-messenger>
[13] <https://chatscope.io/>

requisitos pedem. Por outro lado, sua documentação e escrita possibilitou a implementação de extensões. Duas bibliotecas foram adicionadas. A primeira foi o *React Simple Image Viewer* [14], uma biblioteca capaz de gerar uma visualização de uma imagem. Através dela foi possível o usuário ver a imagem mais de perto, sem sair da página do *chat*. A outra foi a biblioteca *React-Mic* [15] que forneceu a capacidade do *chat* gravar mensagens de voz.

```
<ChatContainer>
      <ConversationHeader>
        ...
      </ConversationHeader>
      <MessageList ...>
        ...
      </MessageList>
      <div as={MessageInput}>
        ...

        <MessageInput
          disabled={this.desativarInput()}
          placeholder="Type message here"
          onFocus={this.typing.bind(this)}
          onBlur={this.stopTyping.bind(this)}
          ...
          value={this.state.msg}
        />
        <InputToolbox>
          <AttachmentButton onClick={this.attach.bind(this
              )} />
          <SendButton onClick={this.talk.bind(this)} />
        </InputToolbox>
        </div>
      </div>
```

[14] <https://www.npmjs.com/package/react-simple-image-viewer>
[15] <https://www.npmjs.com/package/react-mic>

```
25      </ChatContainer>
```

Código 5.1: Exemplo de uso do *Chatscope*

5.1.5 *Tailwind*

Componentes em *React* precisam não apenas do código, mas também de estilização. Apesar do *Chatscope*, existem funcionalidades que necessitam de recursos não oferecidos pela biblitoteca. Portanto, se tornou necessário o uso de um *framework* de *Cascade Style Sheets* (CSS) como *Tailwind* [16]. Como um *framework*, *Tailwind* fornece estilos pré-definidos e mapeados em classes, que podem ser utilizadas por diversos componentes simultaneamente. Assim como *Chatscope*, sua escolha se deu principalmente pela facilidade na criação de páginas mais amigáveis ao usuário e na escrita do código. Outro fator é o fato de que a própria aplicação *alumni*, sobre o qual o *site* estava escrito, também utiliza *Tailwind*.

5.1.6 *Webpack*

Até agora, *back-end* e *front-end* estão se comunicando com a mesma linguagem de programação. Porém a estrutura de ambos os módulos não poderia ser mais diferente. Já cobrimos a comunicação dos pacotes por meio do JSON, mas a integração do código precisa ser feita de outra forma. A princípio, *Adonis* reproduz telas *web* com seus próprios *templates*, com terminação *.edge*. A ideia é renderizar uma aplicação *React* completa dentro de um arquivo *edge*. Para tal, precisamos de um meio para reconhecer o código *React* como tal e renderizá-lo corretamente, apesar dele existir no contexto do *Adonis*. Para isto, utilizamos *webpack*. O *webpack* [17] é um gerador de código web estático. Na *web*, a forma como o código é escrito, com espaçamentos e tabulações, pode fazer a diferença no tamanho do arquivo transferido e performance. *Webpack* trabalha para gerar o arquivo mais simplificado, ao mesmo tempo que compila seus recursos externos. Portanto, é possível configurá-lo para interpretar nossos arquivos *.jsx*, a terminação de um componente *React*.

[16] <https://tailwindcss.com/>
[17] <https://webpack.js.org/>

React e *Adonis* possuem motores diferentes para renderizar seus códigos e sintaxes. A função do *webpack* é pré-compilar nosso *front-end* em *React*, transformando arquivos *.jsx* e suas estilizações em CSS em arquivos *JavaScript* (JS) estáticos. Estes arquivos, por não possuírem qualquer alteração sintática do JS compreendido por navegadores, serão facilmente interpretados pelo *Adonis*. Por último, resta apenas referenciá-los em uma página do *template edge*.

No caso específico de arquivos *jsx*, foi acrescentado um pacote desenvolvido para o *webpack* chamado *Laravel Mix* [18]. Ele compila o código usando pré-processadores CSS e JS, além de entregá-los às pastas públicas da aplicação *Adonis*. Os arquivos da biblioteca do *Tailwind* também foram processadas pelo *Laravel Mix*. Era papel desta biblioteca também vigiar alterações no *React*, gerando código novo em tempo de execução. Este código novo estava em *JavaScript* nativo, para ser servido pelo *back-end* estaticamente. Enquanto isso, o comando de ativação do servidor esperava mudanças para reiniciá-lo, a fim de entregar o comportamento mais atualizado. Para o desenvolvimento pleno da aplicação, foi necessário rodar ambos os comandos paralelamente.

5.2 MVC

O padrão arquitetural implementado pelo *Adonis* é o MVC. Schmidt et al. (1996) definem a arquitetura como uma aplicação separada em três áreas: entrada, saída e processamento. A entrada dos dados está contemplada pelo *Controller*, ao passo que a saída é representada pela *View*. Entre os dois há o *Model*, a camada responsável pelo processamento. O fluxo de uma aplicação *web* em MVC começa pela entrada da *Uniform Resource Locator* (URL). O MVC está organizado, de modo que as mudanças na camada de processamento se propaguem para as múltiplas camadas. Um mesmo modelo de dados pode ser representado de várias maneiras.

Tomando como exemplo o processo de envio de uma mensagem, na Figura 5.2, é possível ver mais claramente o papel de cada camada. A *View* começa da interação do

[18] <https://laravel.com/docs/11.x/mix>

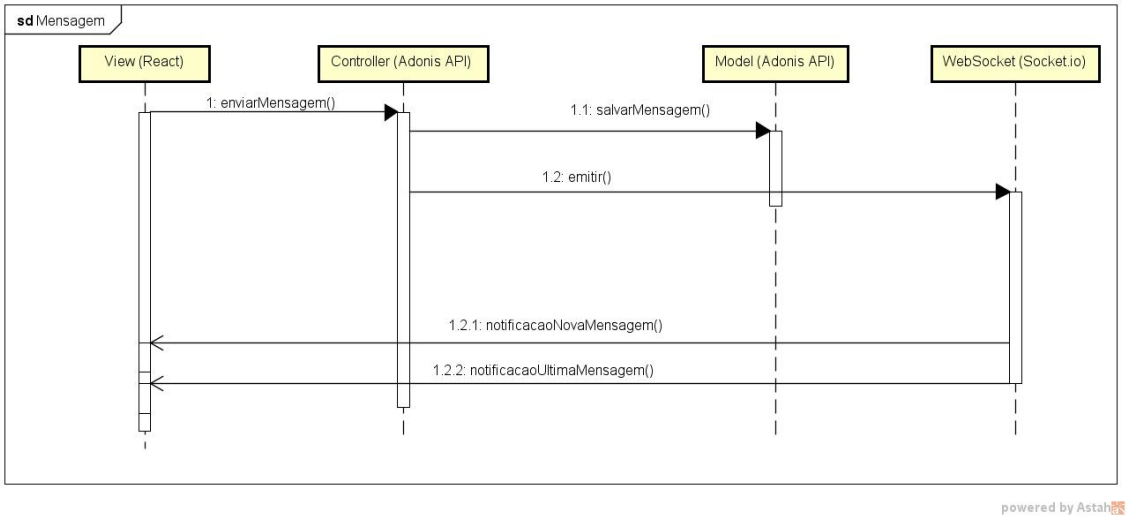

Figura 5.2: Diagrama de Sequência do Envio de Mensagem

usuário, fazendo uma requisição ao *Controller* para salvar a mensagem. Ao receber a requisição *Controller* armazena os dados com o *Model*. Entretanto, o retorno dos nosso dados se dá pelo canal WS ao invés do HTTP. Essa diferença permite uma comunicação assíncrona *full-duplex* dos nossos requisitos. Vários requisitos da aplicação operam sob esse modelo, já que as ações de um usuário em uma conversa podem ser vistas por todos nela.

Esta seção fará um breve resumo de cada componente desta arquitetura no contexto da aplicação *web*. Na seção 5.2.1, será apresentada por onde chegam as entradas e como o redirecionamento para os recursos de processamento são feitos. Já a seção 5.2.2 define como o processamento ocorre, especialmente na implementação das regras de negócio. E a seção 5.2.3 discute os meios pelos quais a aplicação visual, *i.e.*, o módulo *React*, compõe esta arquitetura.

5.2.1 *Controller*

Em uma aplicação *web* em MVC, todas as rotas em cima da URL base estão associadas a um recurso nos *Controllers*. Mais ainda, uma requisição HTTP permite associar rotas parecidas com recursos diferentes. O sistema deste trabalho, por exemplo, foi desenvolvido de modo que uma **Mensagem** possui um *Controller*

próprio que responde à requisições de busca, salvamento e atualização. Dessa forma, define-se busca através do método GET, salvamento do POST e atualização do PUT, conforme o padrão REST. O *Controller* também é responsável por intermediar as outras duas camadas, *View* e *Model*. Idealmente, dada uma URL recebida, o *Controller* designa o *Model* adequado ao recurso, ao passo que escolhe a *View* correspondente.

Como a arquitetura MVC propaga suas mudanças entre as camadas, o *Controller* desempenha o papel de passar essas mudanças adiante. Alterações que nascem no *Model* chegam a *View* como resposta à requisição a princípio. No entanto, como o *chat* exige troca de mensagens em tempo real, a chegada dos dados acontece fora do sentido cliente-servidor. A comunicação com essas duas pontas da aplicação se dá, principalmente, por meio de WS. É função do *Controller* propagar as notificações WS. As chamadas do protocolo estão inseridas dentro das operações dos *Controllers*. Em outras palavras, o *Controller* não chega a designar uma *View* para a página *web*, uma vez que a proposta deste sistema é um SPA do lado do cliente. Na prática, o comportamento do *Controller* é de uma API, cuja comunicação com o cliente se dá pelos dados serializados em JSON que transmite e recebe tanto através do WS quanto do HTTP.

5.2.2 *Model*

A camada *Model* serve como uma interface com o banco de dados. o processo de ORM ocorre nos *Models*, cada objeto associado a uma tabela no banco de dados. O *Adonis* permitiu que determinados processamentos pudessem ser feitos antes e depois da recuperação de um registro. O trecho de código 5.2.2 demonstra um exemplo para **Mensagem**. Como as mensagens são separadas e agrupadas temporalmente, um tratamento imposto ao *Model* foi a adição de um campo escrito de data, que muda de acordo com a proximidade do dia atual. O pós-processamento trata uma **Mensagem** do mesmo dia como data "Hoje", enquanto uma salva no dia anterior possui uma data "Ontem".

```
1 export default class Mensagem extends BaseModel {
```

```
...
@computed()
  public get dataEscrita() {
    let dataBd = new Date(this.dataEnvio)
      ...
    let today = new Date()
      ...
    let yesterday = new Date()
    yesterday.setDate(yesterday.getDate() - 1)
      ...
    if (dataBd === today) {
      return 'Hoje'
    } else if (yesterday === dataBd) {
      return 'Ontem'
    } else {
      return dataBd
    }
  }
}
```

Código 5.2: Trecho de código do *Model* de *Mensagem* no método de serialização da data de envio

5.2.3 *View*

A *View* constitui o *front-end* de uma aplicação. Para *web* isto implica nos arquivos de marcação, HTML, estilização, CSS, e comportamento, JS. Como o sistema separa *back-end* em uma API REST, o *front-end* não é consiste em arquivos separados dessa forma. O modelo adotado é de uma SPA em *React*, dentro do *Adonis*. Todos esses recursos são servidos no *template edge* do *framework*, de modo que as alterações cheguem e sejam processadas em tempo real graças ao *webpack* e *Laravel Mix*. Os dados fornecidos em *JSON* pelo *Controller* junto com renderização dos arquivos da aplicação desenvolvida em *React*. operam em conjunto para criar uma aplicação

cujos recursos chegam sem recarregamentos.

O processo de atualização é obtido com ajuda de WS. Enquanto o *Controller* dispara as atualizações, a *View* responde esses disparos de notificação do WS. Partindo do fato de que *React* está organizado em componentes, cada um deles foi desenvolvido de modo a se subscrever aos eventos adequados. Ao chegar uma alteração, os componentes subscritos ao evento estão configurados para executarem a sua rotina de tratamento, simultaneamente. Para uma Lista de Contatos, isto pode significar aumentar o contador de mensagens não lidas. Para uma conversa ativa, significa exibir uma mensagem nova após a última enviada. Este encapsulamento é muito importante para o rastreio de erros da aplicação. Os componentes devem operar de maneira independente. Logo não apenas o *site* mantém uma tolerância a erros, mas também torna a identificação da região necessitada de mudanças no código de fácil busca. De uma única vez, promove-se os pilares de **Confiabilidade** e **Manutenibilidade**.

5.3 Sistema de Conversação

No *front-end*, *React* é separado a nível de componentes, em vez do padrão MVC. Componentes do *site* podem ser específicos, como botões e campos de formulário, ou mais abstratos, como uma tela de cadastro ou lista de itens. Eles podem ser classes, que herdam de um componente generalizado, ou uma função que retorna um HTML com regras de implementação pontuais. Dessa forma, componentes podem existir tanto individualmente quanto combinados um no outro. Assim como seus atributos visuais, seus comportamentos também podem ser parametrizados de acordo com o contexto. Cada componente possui sua classe *.jsx* e um arquivo CSS. O *framework* também permite a implementação de métodos de ciclo de vida. Os componentes escritos como classes herdam comportamentos do componente geral, o qual são ativados quando são atualizados, removidos ou inseridos na página *web*. Todas essas características dão ao *React* a qualidade de reutilização do código e rastreamento dos erros do qual o sistema precisa.

A aplicação busca construir um *chat* de comunicação em tempo real. Ao mesmo tempo ela importa interfaces visuais e abstrações do *Chatscope*, além de realizar com operações novas com a base da biblioteca. Nesta seção serão discutidos como esses componentes auxiliaram na construção das interfaces principais do sistema em conjunto com as chamadas a API do *back-end*. A princípio, a seção 5.3.1 aborda a interface de criação e visualização dos mais variados tipos de mensagens. E depois, a seção 5.3.2 explica as telas referentes às conversas e grupos.

5.3.1 Mensagens

O envio de mensagens se dá por uma barra posicionada no fundo da conversa. Na Figura 5.3 é possível ver o campo para texto, além de botões para anexo de arquivos e gravação de áudio. No extremo direito do componente, há um botão de enviar, que se ativa quando há uma mensagem pronta para envio.

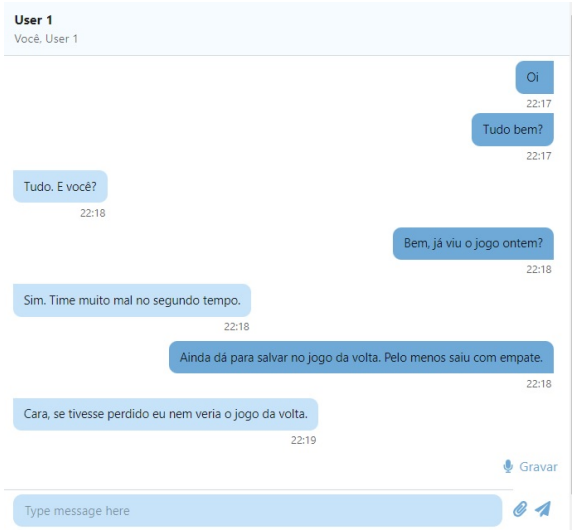

Figura 5.3: Exemplo Conversa

Mensagens na conversa, independente do tipo, exibem informações de contextualização. Para o *chat* essas informações são tanto visuais quanto textuais. Visualmente, as cores dos balões e o posicionamento são indicativos do pertencimento da mensagem ao usuário logado ou seus contatos. Balão azul escuro na direita são mensagens enviadas, enquanto balões claros na esquerda são mensagens recebidas. Textualmente, mensagens carregam o horário de seu envio, como também são agrupadas de acordo

com a data da conversa.

O componente da caixa de texto e botão de enviar estão dentro da mesma abstração, o componente *Chat*. Nele, há a lista de mensagens da conversa atual, os usuários envolvidos e as funções de resposta aos disparos WS. Em relação ao *back-end* o componente *Chat*, a tabela 5.1 descreve quais são as rotas destino. A primeira rota é a de salvamento da mensagem, cujo processo será detalhado para cada formato. No geral, mensagens são salvas via POST para o resultado chegar pelo WS para todos os usuários. Já a segunda, envolve a forma como as mensagens são apresentadas ao usuário. Para ganhar na performance, nem todas as mensagens são carregadas imediatamente nas conversas. Uma quantidade mínima é trazida no primeiro carregamento, porém o restante está programado para vir com a rolagem vertical do componente da lista de mensagens. O disparo desse evento já veio encapsulado do *Chatscope*, bastou implementar a função. Sempre que a rolagem atinge o limite da conversa, o código busca no *back-end* mais mensagens até completar o histórico. Como este recurso é sob demanda do cliente e nem sempre será chamado, foi decidido que ele chegaria como uma requisição HTTP comum.

Rota de Envio	Método	Retorno
/mensagens	POST	WebSocket
/mensagens/conversa/:id/:limit/:offset	GET	HTTP

Tabela 5.1: Tabela de rotas para o serviço de Mensagem do *back-end*

Existem dois eventos disparados pelo *Controller* de mensagens. O primeiro, o evento de nova mensagem, chega para o componente *Chat*, a fim de que seja ordenada e posicionado junto com as outras. O segundo, o evento de última mensagem enviada, chega para o componente de Lista de Contatos. Para Lista de contatos, é importante que a mensagem seja exibida como a última postada. Em conversas não ativas, o ícone de não-lida também precisa ser exibido.

5.3.1.1 Texto

Uma mensagem de texto é a forma mais comum de dois usuários se comunicarem. Após digitar toda a mensagem, o usuário a envia com *Enter* ou um clique no ícone de

enviar. Tal evento inicia uma requisição no servidor para salvar seu conteúdo. Caso a operação seja bem-sucedida, a mensagem será enviada de volta para a aplicação *React* através de *websockets*.

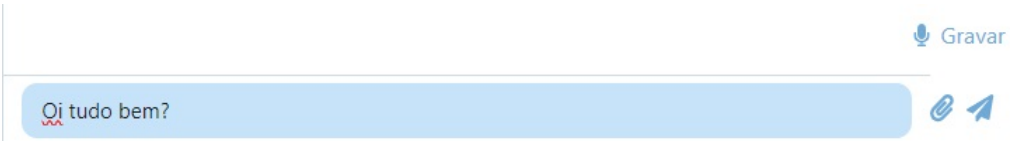

Figura 5.4: Campo para entrar com a mensagem

Quando uma mensagem chega em uma conversa ativa, ela pode ser exibida individualmente ou agrupada com outras. O critério para determinar a organização das mensagens é a ordem de envios. A Figura 5.5 ilustra ambos os cenários. Se a última mensagem antes da nova for do mesmo usuário, ela é agrupada. Caso contrário, ela é exibida de maneira individual.

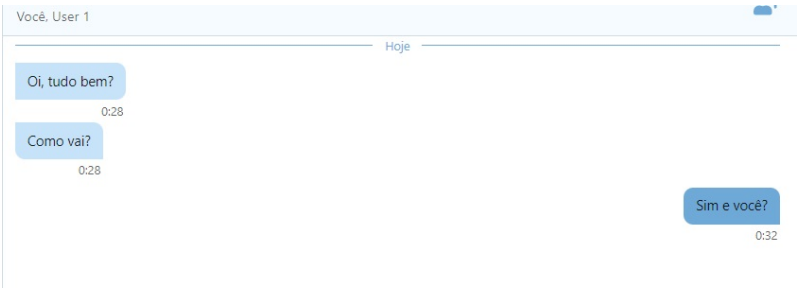

Figura 5.5: Exemplo mensagens agrupadas e não agrupadas

5.3.1.2 Arquivos e Fotos

Durante uma conversa, usuários devem ser capazes de trocar mais do que texto. A transferência de arquivos é um recurso que se divide em duas partes. A primeira é o envio de imagens, já a segunda é o envio de um arquivo de qualquer origem. Universalmente, ao clicar no ícone de um clipe de papel, o sistema abre os diretórios de arquivos do computador pessoal do usuário. Nesse momento, o usuário escolhe qual arquivo deseja enviar na conversa. Antes do envio, a Figura 5.6 mostra uma prévia do arquivo com seu nome, tamanho e formato.

Analogamente à mensagem de texto, o arquivo é enviado para o servidor, onde seu arquivo é salvo no diretório e sua referência, no banco de dados. Depois dessas

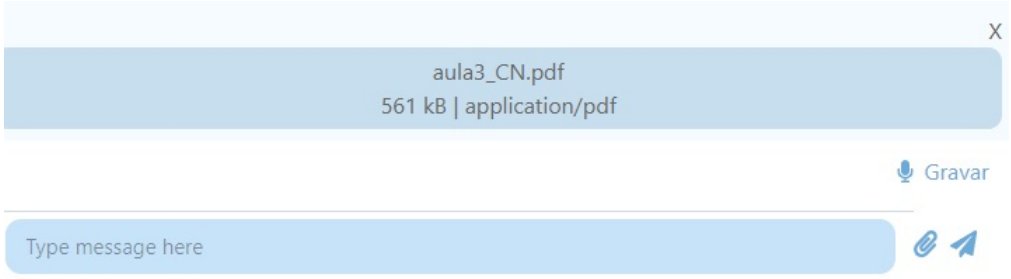

Figura 5.6: Prévia do arquivo a ser anexado

duas operações o evento *websockets* é disparado ao cliente. Dependendo do formato do arquivo, sua exibição muda. Um arquivo que não seja uma imagem, *e.g.*, um *pdf*, é agrupado com as mensagens anteriores somente com a opção para baixá-lo, como visto na Figura 5.7. Esta foi uma das adições feitas ao *chatscope* durante o desenvolvimento. A biblioteca não possui componentes de interface para exibir informações dos arquivos nem gerar *links*. Portanto, criou-se um componente novo, combinando os recursos da biblioteca para manter o mesmo visual.

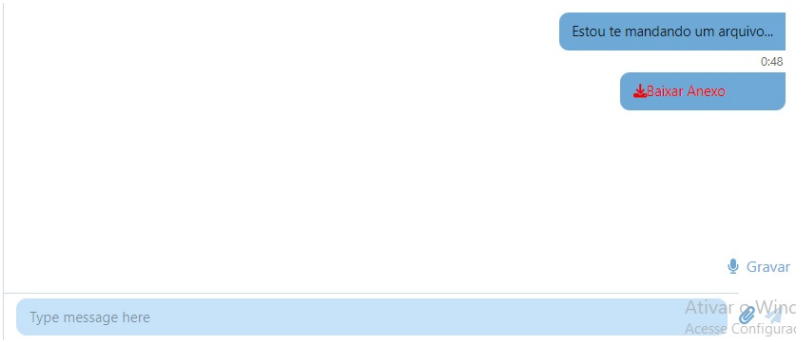

Figura 5.7: Exemplo arquivo no *chat*

Arquivo de extensão *jpg* ou *png* são tratados como imagens. Portanto, em vez de exibidos como um *link*, as imagens aparecem integralmente na conversa. O usuário também pode expandir a imagem para vê-la mais de perto. Essas duas funções, exemplificadas pela Figura 5.9, também não existiam inicial no *Chatscope*. Entretanto, foi integrado a biblioteca *React Simple Image Viewer* para criar esta tela adicional ao clicar na imagem. O resultado final, encontrado no código 5.3.1.2, é uma combinação dos componentes de ambas as bibliotecas.

```
1         <div as={Message.ImageContent}>
2
```

```
3         <div onClick={() => {
4             openImageViewer(index)
5         }}>
6           <Message.CustomContent className='mensagem'>
7               <Message.ImageContent
8                   src={src}
9                   ...
10              />
11              ...
12          </Message.CustomContent>
13        </div>
14
15      {isViewerOpen && (
16            <ImageViewer
17              src={ images }
18              ...
19              closeOnClickOutside={ true }
20              onClose={ closeImageViewer }
21            />
22        )}
23        </div>
```

Código 5.3: Trecho do Componente imagem utilizando *ImageViewer* e *Chatscope*

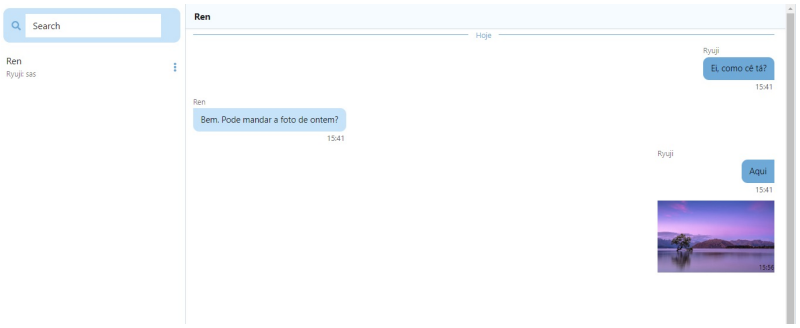

Figura 5.8: Exemplo de Imagem no *chat*

Figura 5.9: Visualização da Imagem

5.3.1.3 Áudio

Mensagens por áudio apresentam um agravamento das dificuldades dos exemplos anteriores. Similar às mensagens com arquivos, o *Chatscope* também não possui qualquer interface para mensagens de áudio. A gravação de um áudio envolve mais do que um simples anexo. A mensagem em áudio é um arquivo gerado em tempo real, onde o usuário deve manipular o começo e fim da gravação. Mais ainda, a exibição do arquivo de áudio é diferente das imagens, porque necessita dos meios para executar e pausar o arquivo. Nenhuma dessas funcionalidades é trivial por si só. Logo, assim como nas mensagens por imagem, foi importada uma biblioteca auxiliar, chamada *React-Mic*. O código 5.3.1.3 mostra a implementação de um componente híbrido, como o da mensagem com foto, entre *Chatscope* e *React-Mic*. O *React-Mic* forneceu o encapsulamento dos eventos de ciclo de vida de uma gravação, como iniciar, interromper ou pausar para implementar o código necessário. Ao final, a biblioteca também oferece os métodos para salvar ou descartar o arquivo gerado.

```
1       <div id="recorder">
2         <ReactMic
3           record={this.state.record}
4           mimeType="audio/webm"
5           className="sound-wave"
6           visualSetting="frequencyBars"
7           onStop={this.onStop.bind(this)}
8           ... />
```

```
9
10          <div id="botoes">
11          <Button
12            onClick={
13              this.state.record?this.stopRecording:this.
                    startRecording
14            } ...>
15              Gravar
16          </Button>
17
18            <Button
19                onClick={this.enviaMensagem.bind(this)}>
20              Enviar
21            </Button>
22
23            <Button onClick={this.descartaMensagem.bind(this)}>
24              Descartar
25            </Button>
26          </div>
27        </div>
```

Código 5.4: Trecho do Componente *AudioRecorder*

A mensagem por áudio se dá pelo clique no ícone de microfone, posicionado na barra acima do campo de texto. Imediatamente, o botão de gravar é substituído pelo de interromper, ao passo que as falas do microfone geram uma resposta visual na barra. O usuário pode gravar a mensagem por quanto tempo desejar, mas após seu fim, suas opções são enviar ou descartar o arquivo criado. Todo esse processo está exemplificado pela Figura 5.10. Esse processo de geração do áudio é diferente de todas outras mensagens, principalmente por necessitar de um componente separado. Logo, apesar de utilizar a mesma rota da API, uma função exclusiva para enviar a requisição POST foi implementada.

Exibir e tocar o arquivo de áudio foi resolvido sem auxílio de bibliotecas. Em vez

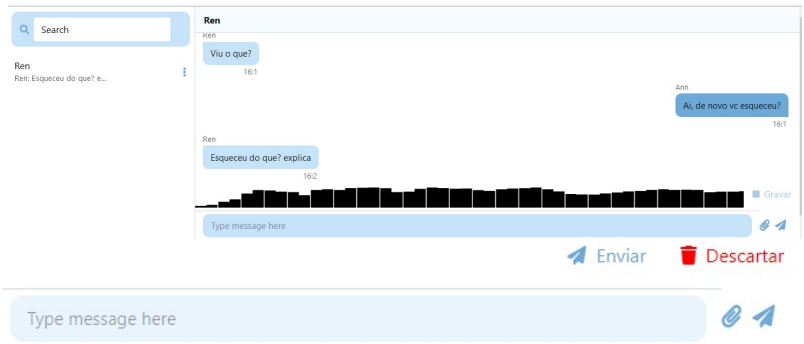

Figura 5.10: Processo de gravação e envio da mensagem de áudio

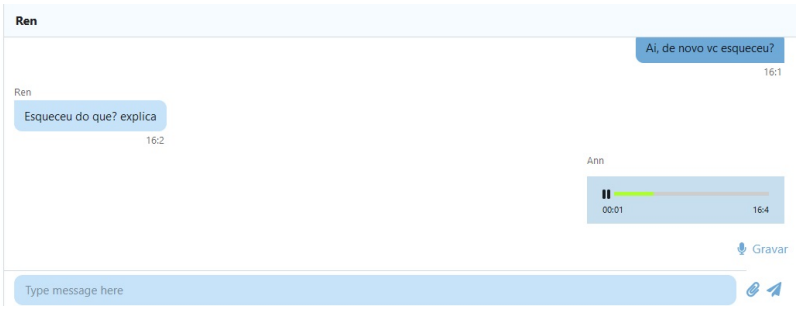

Figura 5.11: Mensagem em Áudio

disso, foi utilizado as próprias *tags* de leitura e execução de arquivos de áudio do HTML. O código 5.3.1.3 mostra uma função geradora do componente. As variáveis de estado para responder à execução do arquivo são *play* e *pause*. Os segundos de gravação ativa e a barra de progresso, são calculadas em tempo real. A integração com *chatscope* serviu apenas para ordená-lo e posicioná-lo com as outras mensagens, conforme exibido pela Figura 5.11. A estilização, para tornar o *player* de áudio parte da conversa, foi obtido com código CSS próprio.

```
export default function AudioMessage(props) {
    const audioRef = React.createRef()
    const [percentage, setPercentage] = useState(0);
    const [play, setPlay] = useState(false)
    const [icon, setIcon] = useState(faPlay)
    const [duration, setDuration] = useState('00:00')
    ...
    return (
        <div className="player" as={Message}>
            <div className="controls">
```

```
11                  <button onClick={handlePlayClick}>
12                      <FontAwesomeIcon icon={icon} />
13                  </button>
14                  <div className="background">
15                      <div style={{width: percentage+"%"}}
                           className="progress">
16                      </div>
17                  </div>
18              </div>
19              <div className="time">
20                  <div>{duration}</div>
21                  <div>{props.time}</div>
22              </div>
23              <audio style={{display: 'none'}} onTimeUpdate={()
                   => {...}} onEnded={handleAudioEnded} controls
                   ref={audioRef}>
24                  <source src={props.src} type="audio/webm"/>
25              </audio>
26          </div>
27      )
28  }
```

Código 5.5: Trecho da Função Geradora da Mensagem de Áudio

5.3.2 Grupo e Conversa

Durante a explicação sobre a modelagem do sistema, **Conversa** foi descrita como uma entidade separada, relacionando-se N:N com **Usuário**. Essa escolha ajudaria a cumprir o requisito da criação de **Grupos**, mas cria um problema para a criação da lista de contatos. Uma vez que a associação entre **Usuário** e **Usuário** não é direta, se faz necessário buscar os contatos por meio das **Conversas**. Porém, *Conversa* e *Grupo* a princípio são a mesma entidade, sendo **Grupo** apenas uma composição opcional. A distinção é feita pela ausência ou não desta entidade.

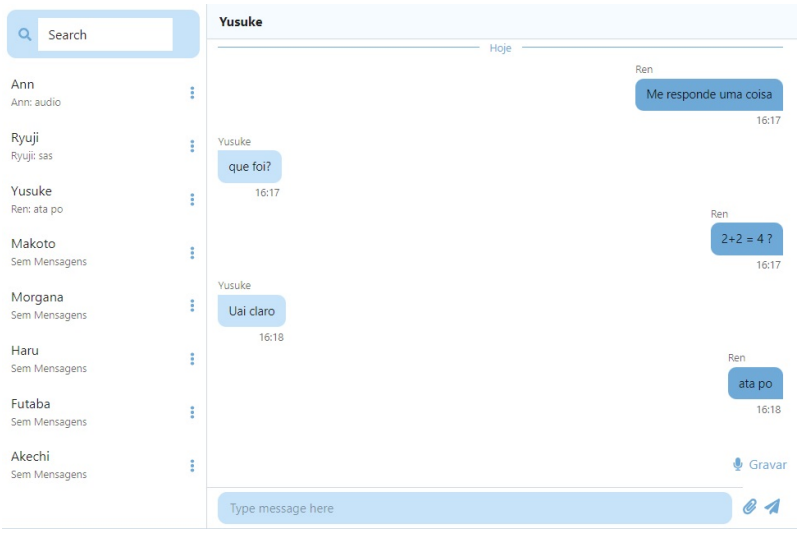

Figura 5.12: Lista de contatos ao lado do *chat*

A lista de contatos de um *Usuário*, como descrita na Figura 5.12, traz o *chat* para o centro e todas as **Conversas** disponíveis no canto. A barra superior à lista permite buscas dinâmicas nos contatos, por meio do nome. Na Figura 5.13, está exemplificado como uma letra é capaz de filtrar entre inúmeros contatos. A princípio, **Conversas** e **Grupos** são diferentes mas aparecem juntos. Contudo, a distinção a nível de código é importante, porque existem operações possíveis em um grupo cuja implementação foge ao escopo da conversa entre dois indivíduos apenas.

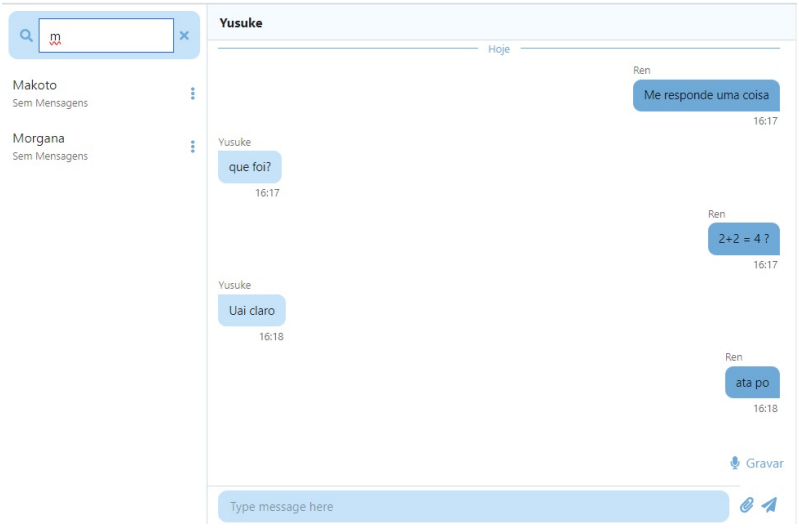

Figura 5.13: Busca de contatos por nome

A fim de preservar a característica SPA da *chat*, as funcionalidades foram desenvolvidas paratelas de diálogo, ou *pop-ups* por cima da tela principal. Esses recursos

também não existiam no *chatscope*, sendo construídos com auxílio do *tailwind*. A tela *pop-up*, ao qual será referida como *modal*, foi implementada através da criação de um componente universal parametrizado, com título, corpo e estilização. Ao mesmo tempo, cada funcionalidade também demandava regras de negócio muito distintas, exigindo componentes com seus próprios ciclos de vida específicos. Então a abordagem foi aninhar os componentes específicos no componente universal, algo que *React* nos possibilita.

As operações do *back-end* utilizadas por este módulo estão listadas na tabela 5.2. A maior parte das rotas retornam seus resultados via WS. A exceção é a rota com a lista de contatos, porque ela é chamada no momento em que o componente é carregado. Esse carregamento será feito apenas uma vez, já que as conversas modificadas durante o tempo de execução virão uma a uma pelo WS, optou-se por uma resposta síncrona em HTTP. Em um modelo de API REST*ful* convencional, modificar uma entidade consistiria em uma única rota PUT. Porém a decisão para nosso sistema foi pulverizar as operações relacionadas à edição da **Conversa** ou **Grupo**. Isso possibilitou uma escrita de funções menores, além de poupar o trabalho de verificação de qual disparo *WS* fazer. Para o caso de um **Usuário** sair de um **Grupo**, as notificações de saída individual ou remoção por parte do um terceiro também são diferentes em rota e retorno. Equilibrar essas duas distinções com outros disparos, como o de adição de um ou mais **Usuários**, aumentaria o tamanho do corpo da requisição e a complexidade do código.

Rota de Envio	Método	Retorno
/conversas/contatos	GET	HTTP
/conversas/sair/:id	POST	WebSocket
/conversas/add/:id	POST	WebSocket
/grupo/remove/:id	POST	WebSocket
/conversas/:id	PUT	WebSocket
/conversas/apagar/:id	POST	WebSocket

Tabela 5.2: Tabela com as rotas para o serviço de Conversa do *back-end*

5.3.2.1 Criar Conversa

Partindo do princípio de que o usuário já possui contatos, *i.e.*, conversas em dupla, o *chat* o permite criar uma conversa em grupo. Na sua lista lateral de contatos, há um botão para opções, onde está para criar um grupo. Ao clicar nele, um *modal* para a criação do grupo será exibido, como o da Figura 5.14. Os campos a serem preenchidos incluem o nome do grupo e quais serão os membros. Este último é preenchido marcando os usuários da sua lista de contatos que deseja incluir. Estes usuários são puxados da mesma rota de contatos usada para criar lista ao lado do *chat*, ou seja, os dados vêm através do HTTP. A diferença desses contatos para a lista está no fato de que o componente em questão filtra os contatos classificados como grupo ao chegarem no cliente. Essa filtragem é obtida através de um campo de pós-processamento no *Model* de **Conversa**. Há um atributo *isGrupo*, que é preenchido pela busca na tabela de grupos do banco de dados. Se uma linha for encontrada com o mesmo identificador da **Conversa**, então ela é um **Grupo**. Caso contrário, ela é uma conversa entre dois **Usuários**.

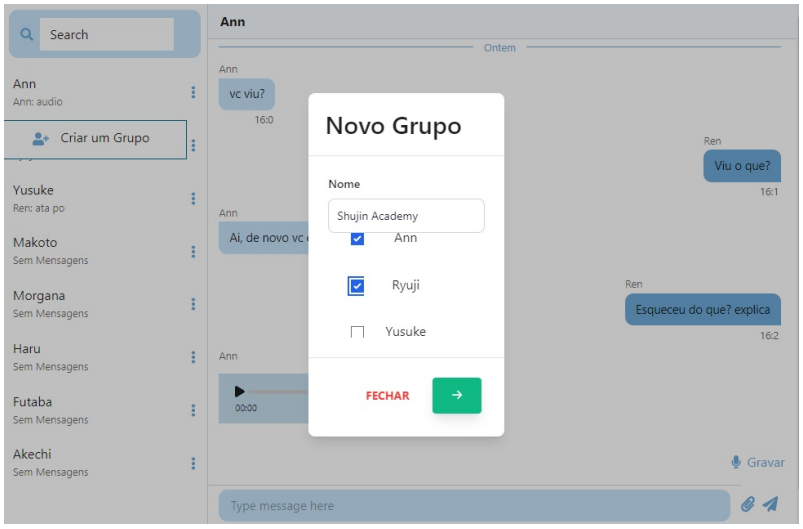

Figura 5.14: Criação de um grupo

As adições realizadas no componente universal envolvem a própria requisição para salvar a conversa. Assim como no envio de mensagens, salvar a nova conversa se dá por uma requisição HTTP retornada por *WS*. Esse disparo é subscrito pelo componente Lista de Contatos, de modo que tanto a conversa quanto as suas mensagens sejam exibidas. Outra adição foi no ciclo de vida do componente, onde a busca pelos

contatos pede uma nova requisição do servidor sempre que o componente se tornasse visível. Isso porque, entre uma criação de ou outra de **Grupo**, **Usuários** podem ser adicionados à lista de contatos. Portanto, a operação os contatos ofertados devem ser da lista mais atualizada no servidor.

5.3.2.2 Editar Conversa

Editar os dados do **Grupo** foi requisito funcional estabelecido na proposta. Essa operação é feita em duas etapas. A primeira é a edição do título do **Grupo**, disponível no ícone ao lado do topo da conversa. Clicar nesse ícone abre um *modal*, como o da Figura 5.15, cujo formulário possui apenas um campo de texto, que inicialmente vem preenchido pelo título mais recente.

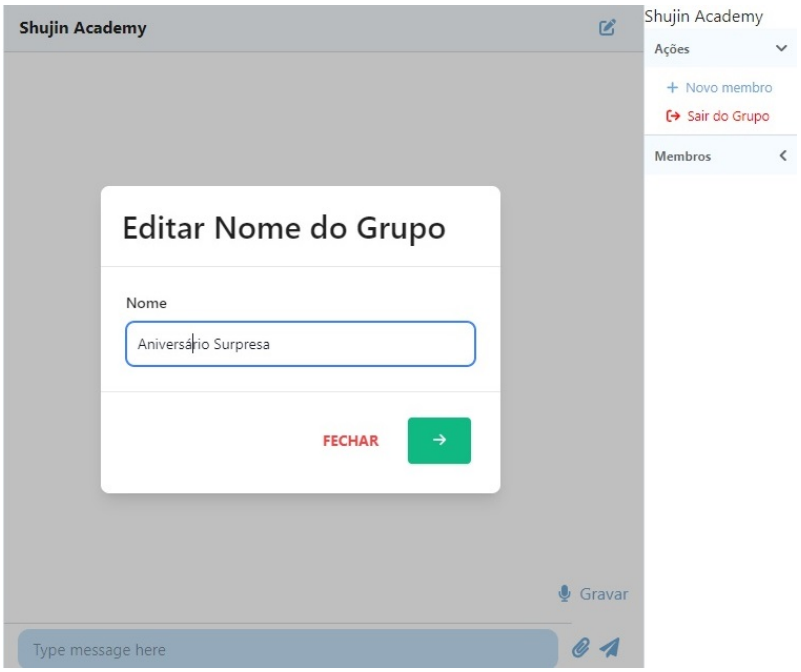

Figura 5.15: Editar título do Grupo

A segunda operação é incluir mais usuários. Presente na barra lateral do grupo, o ícone para novos membros exibe um *modal* com todos os usuários ainda não incluídos no grupo. Tal como a criação do **Grupo**, o usuário passa por uma lista, como a da Figura 5.16, marcando quais membros deseja incluir e confirma o salvamento. Tanto a inclusão de um **Usuário**, quanto a modificação do título do **Grupo**, disparam eventos WS subscritos pelo componente Lista de Contatos. A Lista de Contatos é

responsável por propagar esta mudança nos outros componentes, como os *pop-ups* de alteração ou *Chat*, caso seja a conversa ativa.

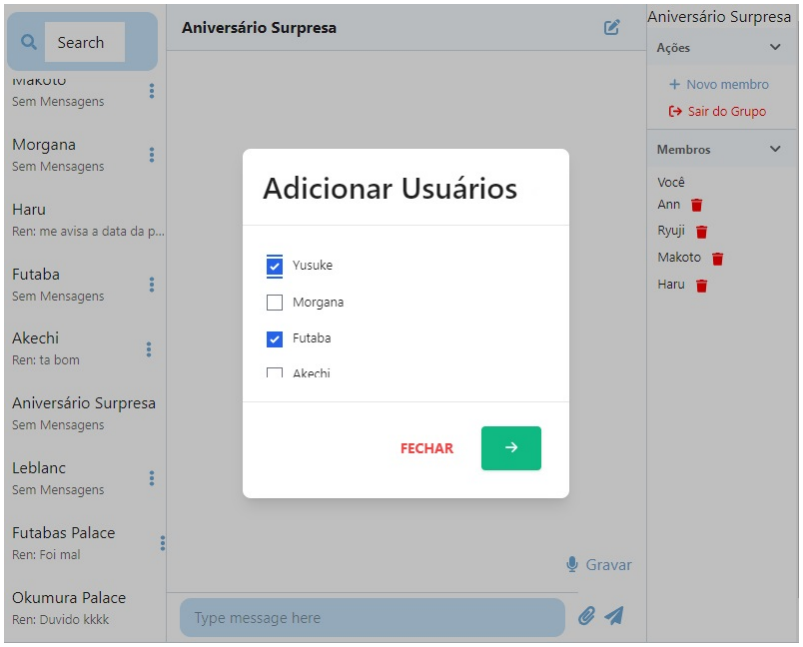

Figura 5.16: Incluir usuário em um grupo

5.3.2.3 Sair do Grupo ou Excluir Usuário

Um **Grupo** também é modificado pela saída de **Usuários**. No entanto, este não é um processo centralizado. O usuário pode iniciar a sua saída tanto quanto ele pode ser removido por outro. Dentro da interface do *Chat*, isso se dá de duas formas. Na barra lateral, como a Figura 5.17, o usuário pode se retirar do grupo escolhendo o ícone na cor vermelha no submenu de ações. O *modal* que aparece para ele pede uma confirmação que, ao ser concedida, faz a requisição para removê-lo daquela associação.

Depois de confirmada a saída, uma resposta em WS é subscrita pelos componentes Lista de Contatos e *Chat*. Ambos os componentes agem de modo a impossibilitar interações na conversa de onde o **Usuário** saiu. A Figura 5.18 mostra os campos de texto, áudio, arquivo desativados. As opções de inserir um novo membro, remover um existente ou alterar o título do grupo também se tornam indisponíveis. A relação entre **Usuário** e **Conversa** ainda permanece no banco de dados. A única ação possível é um novo botão, visto na Figura 5.19, que faz uma requisição para

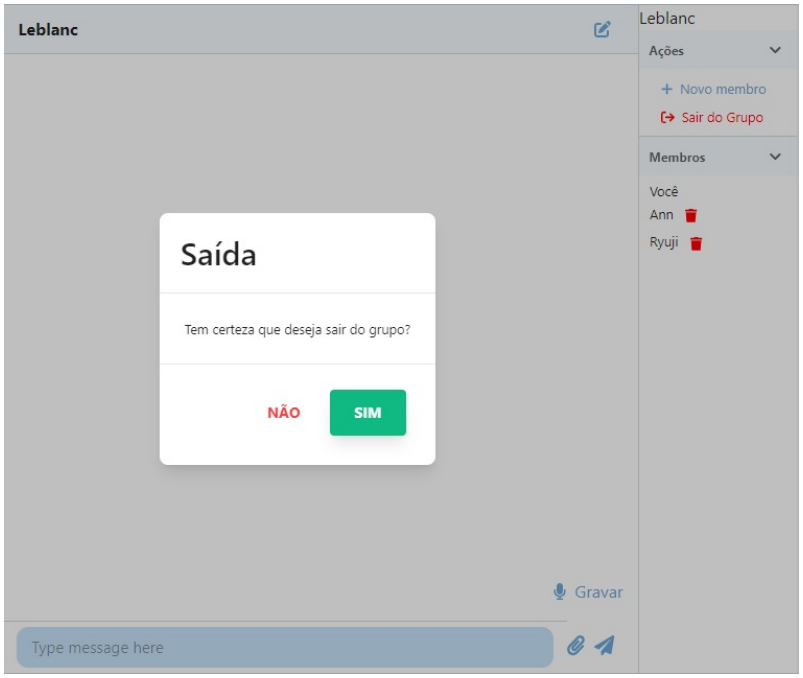

Figura 5.17: Remover Usuário do Grupo

desassociar as entidades definitivamente. O retorno em WS avisa ao cliente para deletar a **Conversa** do componente subscrito Lista de Contatos.

Figura 5.18: Campos de Mensagem Desativados

A remoção de usuários começa através de uma visita ao submenu de membros. Todos os membros do **Grupo** estão listados ali, fornecendo a opção para deletá-lo clicando no ícone de uma lixeira no canto da tela, visto na Figura 5.20. A confirmação é feita através do mesmo *modal* da saída, assim como a modificação é propagada aos outros usuários pelos mesmos componentes subscritos, *i.e.*, Lista de Contatos e *Chat*. As mesmas regras da saída voluntária se aplicam à remoção forçada. O usuário removido não pode enviar nem receber mensagens de qualquer tipo, tampouco realizar modificações no grupo.

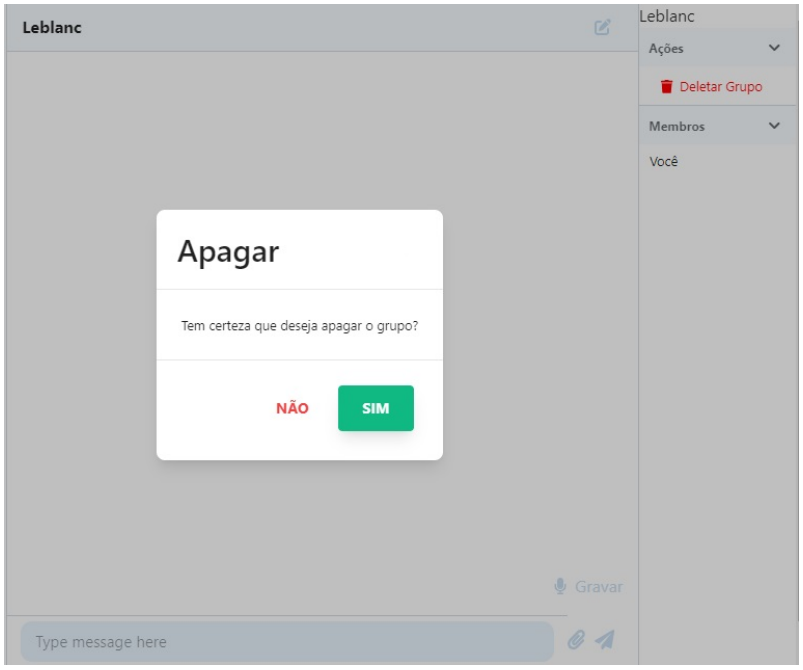

Figura 5.19: Apagar Grupo da Lista de Contatos

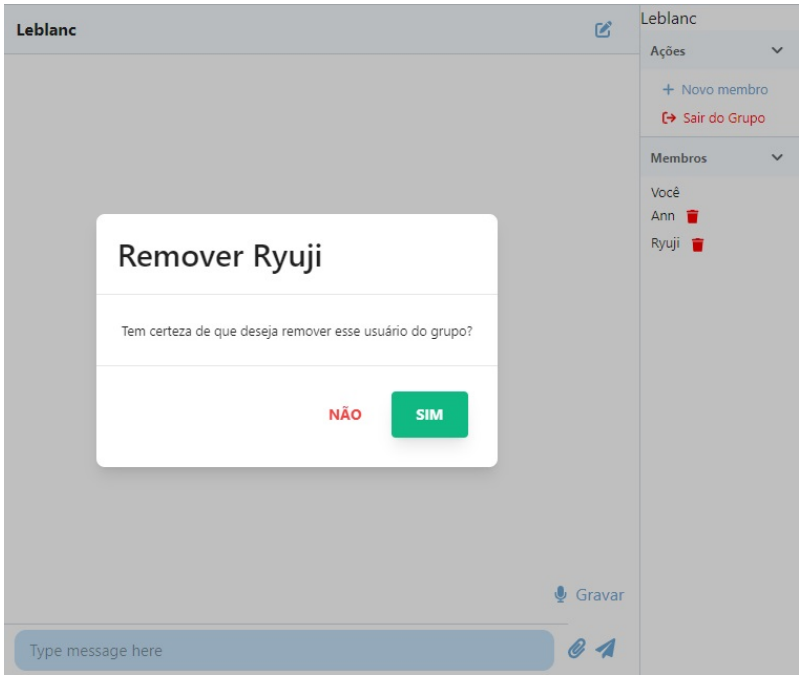

Figura 5.20: Remover Grupo da Lista de Contatos

Capítulo 6

Conclusão

Neste capítulo, serão feitos apontamentos finais a respeito do trabalho. As considerações finais a respeito da aplicação e seus objetivos encontram-se na seção 6.1. Após esse breve resumo, as atuais deficiências serão debatidas a fim de oferecer possíveis soluções para o futuro deste projeto estão presentes na seção 6.2.

6.1 Considerações finais

Este trabalho teve como principal foco o desenvolvimento de um sistema de comunicação em tempo real por meio de aplicação de *chat* para um ambiente *web*, pré-estabelecido pela plataforma *Alumni*. Sob a égide dos conhecimentos sobre aplicações de dados, foi modelado um sistema inicial, uma API em *Adonis*, capaz de suportar uma conversa de múltiplos usuários. Essa mesma conversa também permitia mensagens dos mais variados formatos, cuja apresentação e interatividade se adaptavam conforme. Tudo foi implementado integrando os recursos do *Alumni* de maneira escalável e manutenível.

Apesar disso, o *chat* necessitava de um tempo de latência rápido. Conversas precisam ser dinâmicas, o oposto da troca de mensagens de uma arquitetura cliente-servidor tradicional, especialmente quando se trata de um protocolo como HTTP, operado sobre o TCP. Os protocolos convencionais ofereciam meios de circundar

essa limitação, porém foi no *WebSockets* que a resposta foi encontrada. *WebSockets* permitiu que o servidor se comunicasse com seus clientes, invertendo a ordem cliente-servidor. Não apenas isso, a própria recepção e resposta dos clientes poderia ser personalizada. Notificações que não competiam a um cliente, puderam ser programadas para descarte na aplicação *React*.

Todas essas tecnologias e métodos convergiram em uma aplicação que cumpriu seu objetivo. O *chat* oferece um ótimo ponto de partida para a plataforma na qual está inserido. As mensagens chegam rapidamente, ao passo que são ordenadas e agrupadas pelos dias de envio, mesmo em tempo real. Por sua vez usuário possui a liberdade de se associar com quantos pares quiser. A criação, consulta e modificação dos canais é tão veloz e responsivo, quanto o próprio envio de uma mensagem. Embora haja funcionalidades predominantes em outros *chats*, que possam ser adicionadas para uma experiência mais fluida, a prioridade dos próximos passos para a evolução do *chat* depende de qual direção a plataforma *Alumni* deseja tomar.

6.2 Limitações e trabalhos futuros

O sistema desenvolvido neste trabalho deixou algumas funcionalidades inacabadas, e outras ainda por fazer. Uma delas consiste na alteração ou deleção das mensagens enviadas. Atualmente, uma mensagem na conversa jamais poderá ser modificada, uma funcionalidade presente na maior parte dos *chats* utilizados, como *Whatsapp*, *Facebook Messenger* ou *Discord*. Ainda no tópico de mensagens, o sistema ainda não é robusto o bastante para suportar oscilações de rede. Em caso de perda de conexão, *chats* costumam guardar localmente a mensagem enviada para tentar novamente, assim que a conexão estiver reestabelecida. O *chat* construído neste trabalho não possui esta capacidade.

Esta limitação também impossibilita notificações importantes ao usuário, como a de mensagem enviada, pendente ou visualizada. Outras notificações via WS que precisam de implementação envolvem a remoção ou adição de usuários em grupos. Em conversas, o estado dos usuários envolvidos são notificados na própria linha do

tempo de mensagens, uma funcionalidade incompleta no momento. O mesmo deveria ocorrer para mudanças no título do grupo, a fim de informar o usuário, ausente no momento da mudança, de que ainda é o mesmo grupo no qual estava inserido antes de antes.

Apesar da implementação de mensagens em áudio, imagem e arquivo, há outros formatos possíveis. Em especial, o formato de vídeo com a possibilidade de executar o arquivo dentro do *chat* é uma funcionalidade pertinente. Assim como os áudios, utilizar os recursos de câmera e gravação para gerar vídeos em tempo real também é uma possibilidade. Os formatos diferentes dos textuais também não carregam a opção de uma legenda ou um texto anexado, o que seria útil para deixar informações adicionais para o usuário que recebe.

Finalmente, já que o intento deste *chat* é que os egressos o utilizem com frequência, se faz necessário uma construção voltada para dispositivos móveis. Enquanto *framework*, *React* também possui sua versão para construção *mobile*, *React Native*. As funcionalidades principais da aplicação também estão previamente desenvolvidas e encapsuladas numa API em *Adonis*. Isso significa que qualquer aplicação pode consumi-las sem o retrabalho dos disparos em *websocket* e a implementação alguns requisitos.

Referências

ABMES. Avaliação de empregabilidade de graduados recentes. 2023.

BIANCHI, T. Whatsapp in brazil - statistics facts. 2023.

DAY, J. D.; ZIMMERMANN, H. The osi reference model. 1983.

EDDY, W. Transmission control protocol. 2022.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. [S.l.]: University of California, Irvine, 2000.

HÄRDER, T.; REUTER, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 1983.

HEIMERDINGER et al. *A Conceptual Framework for System Fault Tolerance*. [S.l.], 1992. Accessed: 2024-May-1. Disponível em: <https://insights.sei.cmu.edu/library/a-conceptual-framework-for-system-fault-tolerance/>.

INEP. Censo da educação básica 2022: notas estatísticas. *Brasília, DF: Inep*, 2023.

KLEPPMANN, M. *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. [S.l.]: Boston, Ma: O'Reilly Media, Inc., 2017.

KORTH, H.; S., S.; A, S. *Database System Concepts. McGraw-Hill Education*. [S.l.: s.n.], 2019.

LEACH, P. J.; SAIZ, R.; MEALLING, M. H. A universally unique identifier (uuid) urn namespace. 2005.

M.BISHOP; AKAMAI, E. Http/3. 2022.

MELNIKOV, A.; FETTE, I. The websocket protocol. 2011.

MICHAEL, G. et al. Tracking learners' and graduates' progression paths trackit. European University Association Bruselas, Bélgica, 2012.

MOSELEY, B.; MARKS, P. *"Out of the Tar Pit"*. [S.l.]: BCS Software Practice Advancement (SPA), 2006.

MURLEY, P. et al. Websocket adoption and the landscape of the real-time web. *University of Illinois*, 2021.

NIELSEN, H. et al. Hypertext transfer protocol – http/1.1. 1999.

OBADJERE, E. N. Design and implementation of a real time chat application. *Department of Computer Science Baze University*, 2020.

PAUL, J.-J. Acompanhamento de egressos do ensino superior: experiência brasileira e internacional. 2015.

ROOME, W.; YANG, Y. R. Application-layer traffic optimization (alto) incremental updates using server-sent events (sse). 2020.

SADALAGE, P. J.; FOWLER, M. *NoSQL Distilled. Addison-Wesley.* [S.l.: s.n.], 2012. ISBN 978-0-321-82662-6.

SAINT-ANDRE, P. et al. Known issues and best practices for the use of long polling and streaming in bidirectional. 2011.

SCHMIDT, D. et al. *A system of patterns: Pattern-oriented software architecture.* [S.l.]: Wiley New York, 1996.

SIPOS, N. Graduate career tracking system across the world - as information systems in higher education decision-making process. *University of Pécs Faculty of Business and Economics Pécs, Hungary*, 2017.

SOMMERVILLE, I. *Engenharia de Software.* [S.l.]: Pearson Education, 2011.

TANENBAUM, A. *Redes de Computadores - 4ª Edição.* [S.l.]: Editora Campus (Elsevier), 2003.

YUSOFF, N. H. M.; SELLIAH, D. A. The design and development of live chat support system for prossimo it solution using meteor. 2017.