UNIVERSIDADE PAULISTA – UNIP
Av. Alberto Benassi, 200 - Parque das Laranjeiras - Araraquara - SP
CEP 14804-300 - Tel.: (16) 3336-1800

CIÊNCIA DA COMPUTAÇÃO

APS – ARQUITETURA DE REDES DE COMPUTADORES **Desenvolvimento de uma Ferramenta para Comunicação em Rede**
Nome do Aluno Gabriel Carrascosa
RA: G744CE-2

PROF. NOEL MOREIRA

ARARAQUARA - SÃO PAULO

2025

APS – ARQUITETURA DE REDES DE COMPUTADORES

Desenvolvimento de uma Ferramenta para Comunicação em Rede

Atividades Práticas Supervisionadas para a obtenção de nota para a graduação em Ciência da Computação apresentado à Universidade Paulista – UNIP.

Orientador: Noel Moreira

ARARAQUARA - SÃO PAULO

2025

Biblioteca da Universidade Paulista - UNIP.

Guia de normalização para apresentação de trabalhos acadêmicos da Universidade Paulista: ABNT / Biblioteca da Universidade Paulista - UNIP.

– 2021.

52 p. : il. Color.

Normalização. 2. Trabalhos acadêmicos. 3. ABNT. I. Biblioteca da

Universidade Paulista – UNIP.

BANCA EXAMINADORA

_____/_____/_____

Prof. Noel Moreira

Universidade Paulista - UNIP

RESUMO

O projeto pode ser resumido na criação de um programa de comunicações em distância, usando de referência aplicativos como *“WhatsApp”*, *“Discord”*, *“Telegram”* ou outros que utilizam do protocolo *“WebSocket”* para seu funcionamento.

A aplicação utilizará de ferramentas modernas muito usadas no mercado de trabalho, como o *“Angular”* e *“JSON-Server”* para uma melhor modularização dos componentes inclusos no programa

SUMÁRIO

1. OBJETIVOS E MOTIVAÇÕES. 6
1.1 Objetivos gerais. 6
1.2 Objetivos específicos. 6

2. INTRODUÇÃO. 7

3. DESENVOLVIMENTO. 7

3.1 Fundamentos de comunicação de dados em rede. 7

3.2 Problemas quanto aos protocolos. 9

3.3 Plano de desenvolvimento da aplicação. 10

4. EXECUÇÃO DO PROGRAMA. 13

5. RELATÓRIO COM AS LINHAS DE CÓDIGO DO PROGRAMA. 17

6. BIBLIOGRAFIA. 27

1. OBJETIVOS E MOTIVAÇÕES

1.1 Objetivos gerais

Desenvolver uma aplicação web utilizando conceitos de comunicação com Sockets de Berkeley, conforme descrito no protocolo *"RFC 6455"*, que define o funcionamento do *"WebSocket" como* uma comunicação bidirecional em tempo real entre cliente e servidor usando apenas uma conexão *"TCP"*.

Essa abordagem é extremamente útil em aplicações que exigem interações de baixa latência, como chats de mensagens, jogos online ou sistemas de monitoramento em tempo real.

O projeto busca proporcionar uma compreensão prática dos fundamentos por trás da comunicação em tempo real, incentivando a utilização dessas tecnologias em aplicações modernas. A integração entre o *"Front-end"*, o *"Back-end"* e o banco de dados simulado reforçará a importância da arquitetura cliente com servidor em sistemas distribuídos.

1.1 Objetivos Específicos

Alguns tópicos devem ser destacados sobre o proceder do projeto:

- A aplicação será desenvolvida com auxílio da biblioteca de código aberto *"Socket.IO"*, que facilita o gerenciamento dessa conexão.
- Para a interface gráfica será utilizado o *"Angular"*, que auxilia na componentização e organização das páginas.
- O armazenamento dos dados dos usuários e suas mensagens será feito com o *"JSON-Server"*, que oferece uma estrutura simples de banco de dados.
- Terá um sistema de verificação de usuário, com telas de *"Login"* e Registro, ficando salvo no navegador local para futuros acessos de forma automática.
- Quando *"Logado"*, o usuário será capaz de adicionar amigos através de suas "*Tags*" e conversar com cada um deles de forma privada.

2. INTRODUÇÃO

A comunicação em redes não só vem se tornando, como já faz parte praticamente do cotidiano moderno, estando em praticamente tudo quando o assunto se trata de computadores, desde acessar sites para ver vídeos do seu astro favorito, até enviar relatórios confidenciais para o presidente.

Mas para que isto ocorra de forma correta, segura e eficiente, existem diversos protocolos que foram desenvolvidos ao decorrer das décadas, como o *“TCP/IP”* (Transmission Control Protocol/Internet Protocol ou Protocolo de Controle de Transmissão/Protocolo de Internet), um protocolo de envio e recebimento de informações e que garante que os dados sejam sempre recebidos por completo nos respectivos destinos, em ordem correta e sem falhas.

Outro protocolo amplamente utilizado é o *“HTTP”* (Hypertext Transfer Protocol ou Protocolo de Transferência de Hipertexto), que funciona sobre a conexão *“TCP”*, onde se estabelece uma comunicação entre o cliente (geralmente um navegador) e o servidor, utilizando de métodos já desenvolvidos, como o de buscar(*“GET”*), ou enviar(*"POST"*) informações para ambos os lados da conexão, o protocolo opera em um modelo de requisições e respostas, onde o cliente faz um pedido e o servidor responde, garantindo a entrega organizada dos dados.

Compreender e aplicar estes conceitos de comunicação em tempo real é essencial para qualquer profissional da área de tecnologia. O mercado atual valoriza soluções que sejam rápidas, eficientes e interativas, e muitos dos sistemas modernos dependem dessas tecnologias.

3. DESENVOLVIMENTO

3.1 Fundamentos de comunicação de dados em rede

Anteriormente foram citados exemplos de protocolos utilizados atualmente, mas o que são protocolos e o que fazem? Trata-se da definição de um formato que será utilizado na troca de informações entre dispositivos conectados, algo que ambos os lados concordam em seguir de forma correta. Neste padrão, existem uma série de

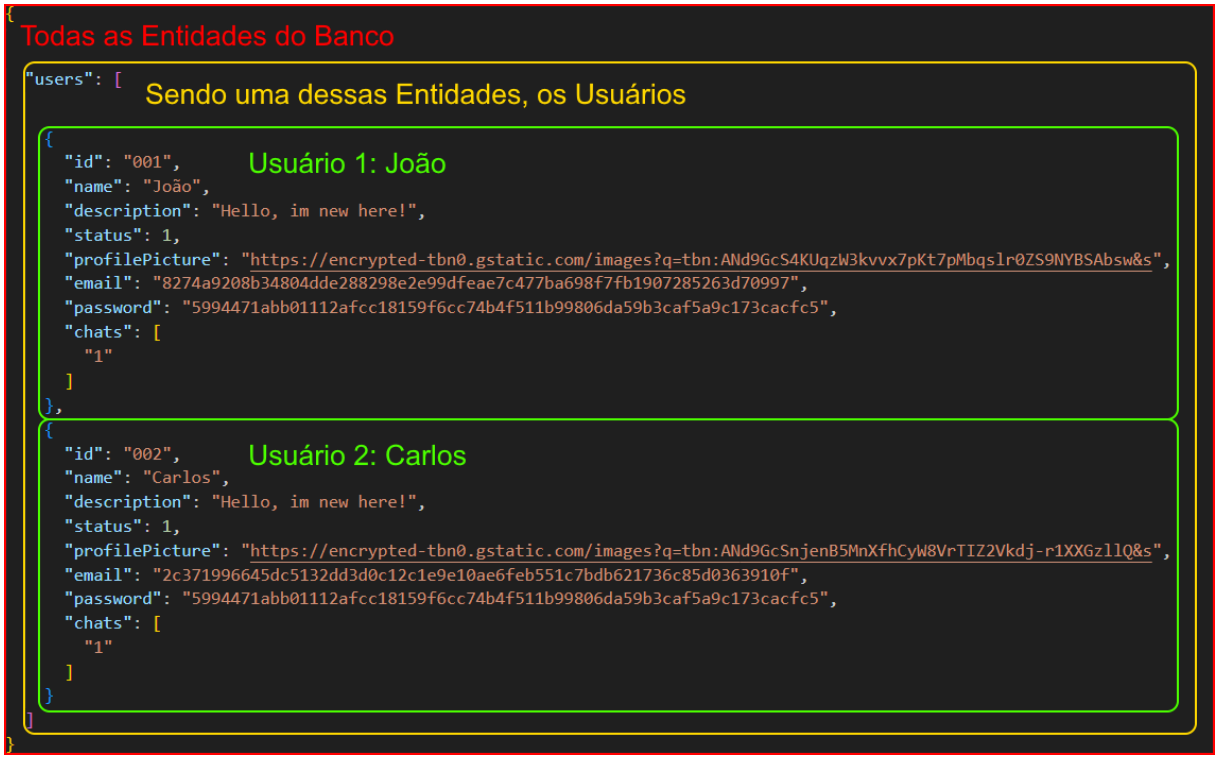
regras de troca de dados padronizadas, permitindo que diferentes *“hardwares”* e *“softwares”* possam acessar as informações que são divididas em pequenas partes, onde em cada uma contém um endereço da origem e do destino, sendo criado uma “língua” em que todos usam para se comunicar apropriadamente.

Esses protocolos funcionam em camadas, cada uma com responsabilidades e funcionalidades distintas:

1. Camada Física.
    - Sinais elétricos em uma placa mãe.
2. Camada de Enlace de dados.
    - Garante que dados cheguem corretamente entre dois computadores, essencialmente em uma mesma rede.
3. Camada de Rede.
    - Também garante que dados cheguem de forma correta, mas entre diferentes redes.
4. Camada de Transporte.
    - Garante que a entrega dos dados ocorra de forma confiável.
5. Camada de Sessão.
    - Gerencia as sessões de comunicação entre dois dispositivos.
6. Camada de Apresentação.
    - Formata os dados para que a próxima camada os interprete-os.
7. Camada de Aplicação.
    - O que um usuário comum tem acesso, um site ou aplicativo.

Na aplicação desenvolvida, o uso do *“WebSocket”*, que será abordado mais para frente, envolve diretamente o uso de algumas dessas camadas: Como a de Transporte, funcionando sobre o protocolo *“TCP”*, a de Sessão, que mantém uma conexão ativa entre cliente e servidor, a de Apresentação formatando arquivos *“JSON” (*JavaScript Object Notation ou Notação de objeto JavaScript*),* que é um dos arquivos mais utilizados para transportar dados, com uma estrutura que lembra a de tabelas em bancos de dados, mas como se separasse cada objeto de forma horizontal em sua própria tabela única, como demonstrado na imagem a seguir:

Figura 1 - Detalhes sobre o Formato *"JSON"*



Fonte: Autoria Própria

Posteriormente, o projeto irá se comunicar com estes arquivos, enviando ou recebendo, para que mais tarde, na camada de Aplicação, os mostre em forma de site, contendo o nome e foto do usuário *"Logado"*, seus amigos adicionados e suas respectivas mensagens diretas.

3.2 Problemas quanto aos protocolos

Todos esses protocolos se mostram ser extremamente úteis quando tratamos de aplicações que não necessitam ser em tempo real, como sites de vídeos, artigos ou de produtos, mas terá problemas quanto o assunto for o oposto, aplicações que requerem a menor latência no envio e recebimento de informações, como jogos ou aplicativos de bate papo entre duas ou mais pessoas.

E é exatamente aqui que surge uma das principais limitações do protocolo *"HTTP"*, forçando o cliente a sempre pedir algo para que o servidor o retorne com a informação desejada. Utilizar esse protocolo para criar um bate-papo exigiria que o cliente ficasse perguntando ao servidor de tempos em tempos se há novas

informações sobre sua atual conversa, gerando tráfego desnecessário, atrasos e uso excessivo de recursos.

Com a necessidade de algum novo protocolo que atenda a estes problemas, foi então desenvolvido em dezembro de 2011, o protocolo *"WebSocket"*, que se trata de uma forma de conexão persistente, sendo padronizado de forma oficial pela *"IETF"* (Internet Engineering Task Force ou Força-Tarefa de Engenharia de Internet) e documentado na *"RFC 6455"* (Request for Comments ou Solicitação de Comentários), uma série de documentos técnicos padronizados que descrevem protocolos, sistemas e práticas da *"Internet"*. O protocolo surgiu sendo uma proposta do grupo HyBi Working Group, com colaboração de empresas como Google e Mozilla.

Ao contrário do *"HTTP"*, onde cada requisição do cliente exige uma nova resposta do servidor, o *"WebSocket"* permite estabelecer uma conexão persistente, criada por meio de um *"Handshake"* inicial feito via *"HTTP"*, no qual o cliente solicita a atualização do protocolo com um cabeçalho especial *"Upgrade: Websocket" e após* aceito pelo servidor, a conexão é promovida para *"WebSocket"* e permanece aberta durante toda a sessão.

Com isso, tanto o cliente quanto o servidor podem enviar e receber dados a qualquer momento de forma bidirecional, assíncrona e tempo real, sem a necessidade de novas requisições.

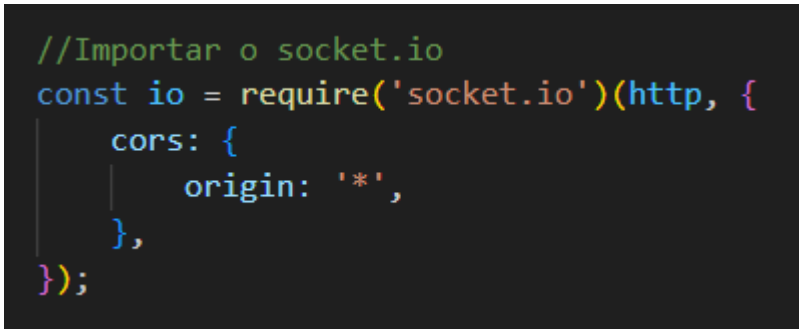
3.3 Plano de desenvolvimento da aplicação

Partindo para o projeto desenvolvido, pensando em tornar a aplicação simples o bastante, mas que ainda tenha um certo grau de liberdade para a criação de interfaces interessantes e dinâmicas, foi utilizada a biblioteca *"Socket.IO"*, que implementa o protocolo *"WebSocket"* de forma fácil, já que contém diversos códigos desenvolvidos e bem estruturados por outros programadores.

Com o auxílio do *"Angular"* para as interfaces, permitiu a capacidade de dividir as páginas em componentes reutilizáveis, o que torna o código menos repetitivo e intuitivo e fica responsável por toda a parte lógica de serviço da aplicação, com recebimento e envio de dados no protocolo *"HTTP"*, que ao receber, processa esses dados, identifica onde cada elemento será utilizado e retorna uma interface gráfica.

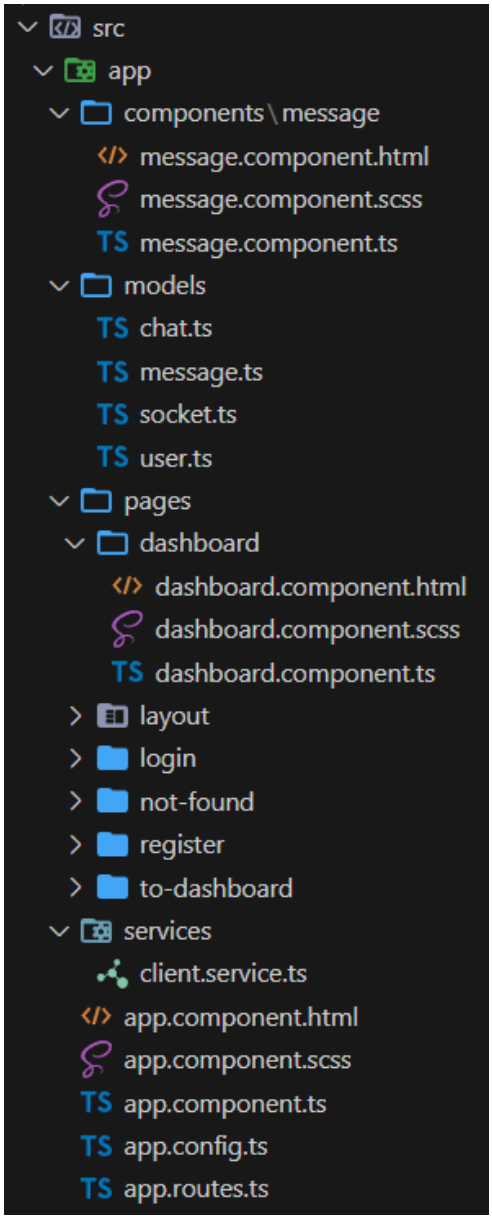
O “*JSON-Server*”, por sua vez, é responsável pelo armazenamento da aplicação, construindo um sistema funcional, com registro de usuários, envio de mensagens e uma comunicação fluida entre cliente e servidor.

Figura 2 - Importação do *“Socket.io”* para o Servidor do Projeto



Fonte: Autoria Própria

Figura 3 - Diretório padrão gerado pelo *"Angular"*



Fonte: Autoria Própria

Figura 4 – Arquivo *"JSON"* usado pelo *"JSON-Server"*

Fonte: Autoria Própria

4. EXECUÇÃO DO PROGRAMA

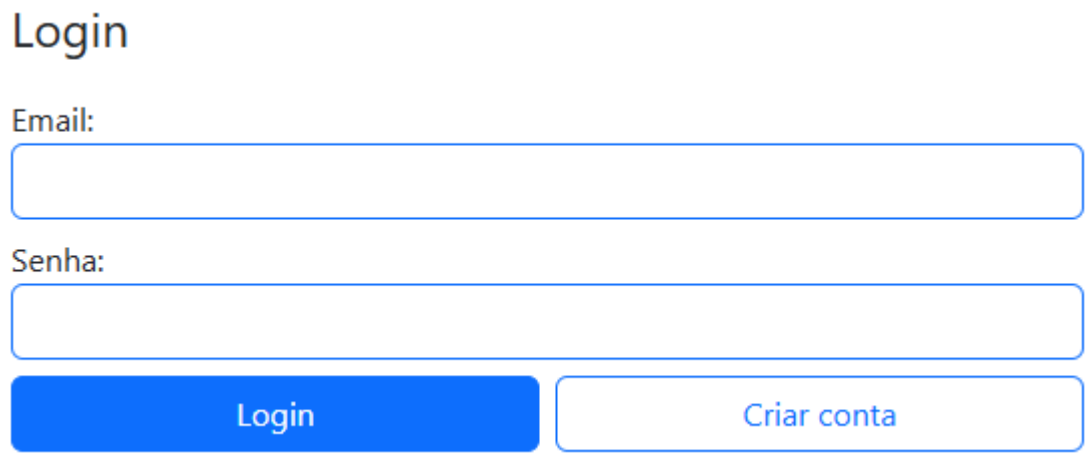
A execução da aplicação funciona em algumas etapas obrigatórias:

1. Estabelecer a conexão *"HTTP"* (*"HandShake"*).
2. Verificar se o usuário já esteve conectado, buscando informações salvas em *"Local Storage"* do navegador (Local que permite armazenar informações localmente no navegador) e conectando o usuário de forma automática.

3. Caso não tenha nenhum dado salvo, ou nenhum dado válido o usuário sempre será levado para a tela de *"Login"*, que posteriormente, caso não tenha uma conta, a tela de Registro.
4. A validação dos campos é feita criptografando o Email e Senha digitados e buscando alguém no banco de dados que contenham os mesmos dados, que já está salva de forma criptografada.

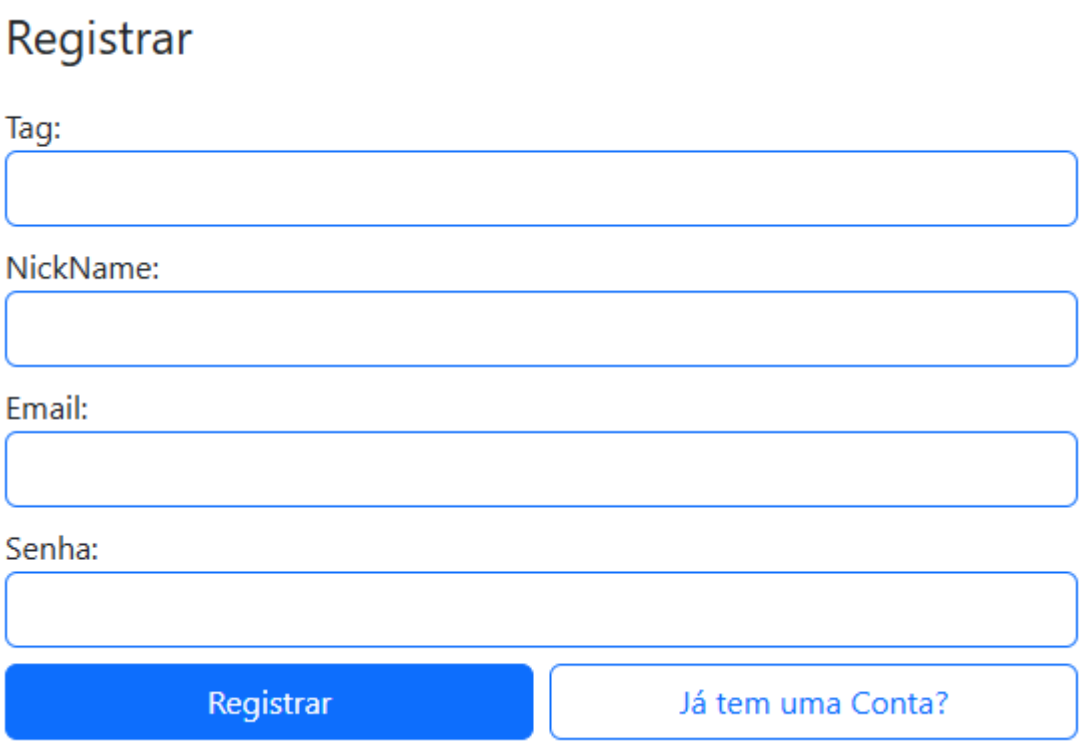
A partir desta validação inicial, o usuário tem liberdade de adicionar vários amigos através de suas *"Tags"* e conversar com qualquer um deles em tempo real

Figura 5 – Tela de *"Login"* da Aplicação



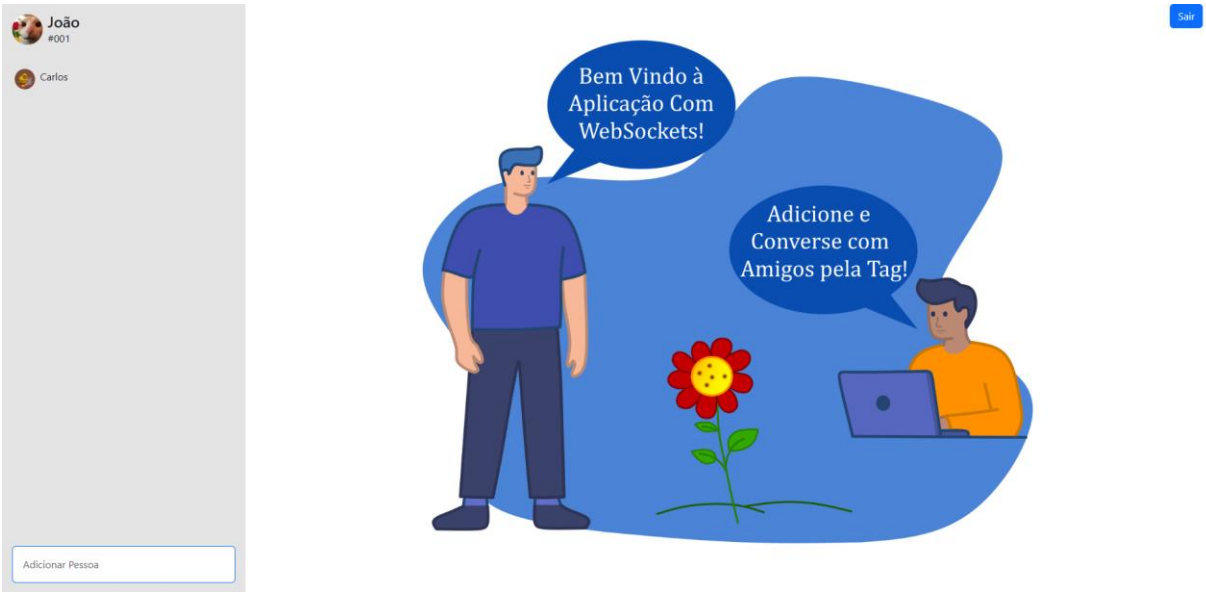
Fonte: Autoria Própria

Figura 6 – Tela de Registro *da Aplicação*



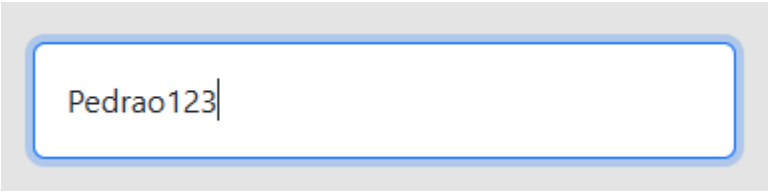
Fonte: Autoria Própria

Figura 7 – Tela inicial da Aplicação



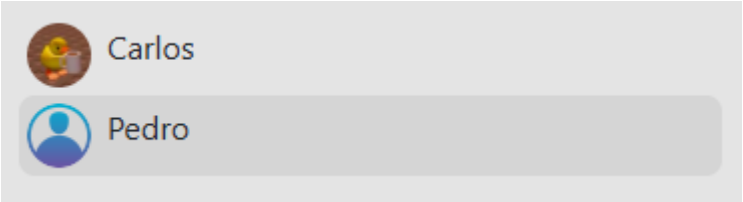
Fonte: Autoria Própria

Figura 8 – Adicionando amigos com suas *"Tags"*



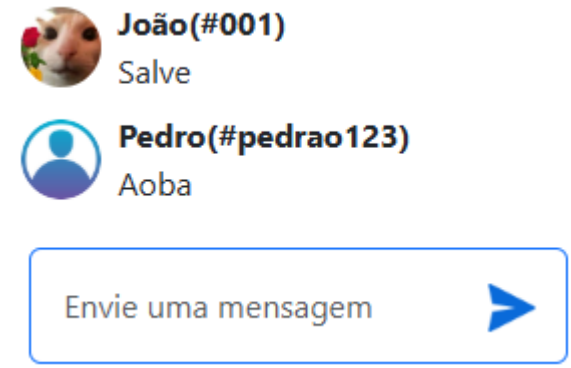
Fonte: Autoria Própria

Figura 9 – Amigo adicionado à Lista de Amigos na barra esquerda da tela



Fonte: Autoria Própria

Figura 10 – Conversando com o Amigo adicionado



Fonte: Autoria Própria

Foi de suma importância para a formação acadêmica, visão pessoal e profissional o trabalho aqui realizado, onde ensinamentos sobre o estudo de comunicação em rede, seguindo de forma correta seus protocolos, junto com a elaboração de códigos, formatação de arquivos formais e edição de imagens serão levados a futuros trabalhos acadêmicos e para o mercado de trabalho.

5. RELATÓRIO COM AS LINHAS DE CÓDIGO DO PROGRAMA

Código do Servidor da Aplicação, recebendo novas mensagens ou pedidos de amizade e enviando a informações para todos os dispositivos conectados:

```
const { log } = require('console');

//Criar um servidor HTTP básico
const http = require('http').createServer();

//Importar o socket.io
const io = require('socket.io')(http, {
    cors: {
        origin: '*',
    },
});

//Recebe a conexão do cliente
io.on('connection', (socket) => {
    log(typeof socket);
    console.log('New client connected');

    //Recebe a mensagem do cliente
    socket.on('message', (message) => {

//Envia a mensagem para todos os clientes conectados, será filtrado
        io.emit('message', message);
    });

    //Recebe pedidos de amizade
    socket.on('friendRequest', (friend) => {

//Envia o pedido para todos os clientes conectados, será filtrado
        io.emit('friendRequest', friend);
    });
})
```

```
http.listen(8080, () => {
    console.log('Server is listening on port 8080');
});
```

Regras de Negócio, funções de “Logar” o Usuário, adicionar amigos ou enviar mensagens, também há outras subfunções neste arquivo:

```
import { Injectable } from '@angular/core';
import { Message } from '../models/message';
import io from 'socket.io-client';
import { HttpClient } from '@angular/common/http';
import { ActivatedRoute, Router } from '@angular/router';
import { Observable } from 'rxjs';
import { User } from '../models/user';
import { Chat } from '../models/chat';

@Injectable({ providedIn: 'root' })
export class ClientService {
  private socket = io('ws://localhost:8080');
  private apiUrl = 'http://localhost:3000';

  //Usuário logado, chat de conversa e lista de amigos
  private user?: User;
  private chatID: string = '0';
  private friendsList: Array<{ chat: Chat, user: User; }> = [];

  constructor(private http: HttpClient, private getRouter: Router, private getActivatedRoute: ActivatedRoute) {
    //Recebe mensagens do servidor
    this.socket.on('message', (message: Message) => {
//com o message.userID dá pra pegar as informações que precisamos do usuário
//Atualizar a tela com a mensagem recebida, se bater o id do chat
      if (message.chatID == this.chatID) {
        //Busca o usuário pelo id
        this.getUserByID$(message.userID).subscribe({
          next: (user) => {
            if (user[0]) {
              this.friendsList.forEach(friend => {
                if (friend.chat.id == message.chatID) {

                  //Procurar qual chat adicionar a mensagem
                  this.friendsList.forEach((friend) => {
                    if (friend.chat.id == message.chatID) {
```

```
                    friend.chat.messages.push(message);
//Editar no banco o chat especifico com a nova mensagem implementada
                    this.putChat$(friend.chat).subscribe({
                      error: () => { alert('Erro: PutChat'); }
                    });
                  }
                });
              }
            });
          }
        });
      }
    });

    this.socket.on('friendRequest', (friend: { chat: Chat, user: User; }) => {
      if (friend.user.id == this.user?.id) {
        this.user?.chats.push(friend.chat.id);
        this.refreshFriendsList();
      }
    });
  }

  //Enviar mensagem para o servidor
  sendMessageToServer(text: string) {
    if (this.chatID.toLowerCase() != '') {
      //Criar um "Pacote", um objeto que carrega a mensagem digitada com mais
algumas informações
      let message: Message = {
        chatID: this.chatID,
        userID: this.user?.id.toLowerCase() || '',
        text: text,
        edited: false,
        deleted: false
      };

      this.socket.emit('message', message);
    }
  }

  //Adicionar um amigo pela tag
  addFriend(findUserTag: string) {
    //Verificar se não é o mesmo usuário
    if (findUserTag.toLowerCase() == this.user?.id.toLowerCase()) {
      alert('Você não pode se adicionar!');
      return;
    }
```

```
    //Verificar se o usuario já está adicionado
    let friendOnList = false;

    this.friendsList.forEach((friend) => {
      friend.chat.usersIDs.forEach(userID => {
        if (userID.toLowerCase() === findUserTag.toLowerCase()) {
          console.log('IDs:          '          +          userID.toLowerCase(),
findUserTag.toLowerCase());
          friendOnList = true;
        }
      });
    });

    if (friendOnList) {
      alert('Usuário já está na sua lista de amigos!');
      return;
    }

    //Procurar se este amigo existe
    this.getUserByID$(findUserTag.toLowerCase()).subscribe({
      next: (user) => {
        //Caso exista
        if (user[0]) {
          //Pegar todos os chats(infelizmente não tem como pegar só o length)
          this.getChats$().subscribe({
            next: (allChats) => {
//Criar um novo chat, como id baseado no length do array de chats + 1
              let chat: Chat = {
                id: String(allChats.length + 1), //id do chat
                usersIDs:       [this.user?.id.toLowerCase()       ||       '',
user[0]?.id.toLowerCase() || ''],
                messages: []
              };
              //Adicionar o chat no banco de dados
              this.postChat$(chat).subscribe({
                complete: () => {
                  //Adicionar o ID do chat no usuário logado
                  this.user?.chats.push(chat.id);
                  this.putUser$(this.user).subscribe({
                    complete: () => {
                      //Adicionar o ID do chat no amigo
                      user[0].chats.push(chat.id);

                      this.putUser$(user[0]).subscribe({
                        complete: () => {
```

```
                        this.socket.emit('friendRequest', { chat: chat, user:
user[0] });
                        this.refreshFriendsList();
                      },
                      error: () => { alert('Erro: PutUser Friend'); }
                    });
                  },
                  error: () => { alert('Erro: PutUser'); }
                });
              },
              error: () => { alert('Erro: PostChat'); }
            });
          },
          error: () => { alert('Erro: GetChats'); }
        });
      }
      else { alert('Usuário não encontrado!'); }
    },
  });
 }

 public tryLogin() {
  this.authenticateLogin(localStorage.getItem('userEmail')      ||      '0',
localStorage.getItem('userPassword') || '0');
 }

 //Autenticar o Login Usuário
 public authenticateLogin(email: string, password: string) {
  //Criptografar a entrada, sendo do local storage ou da tela de login /
registro
  this.encryptText(email).then((encryptedEmail) => {
    this.encryptText(password).then((encryptedPassword) => {
//Para então comparar com o email e senha salvos no banco(criptografados)
      this.getUserByEmailAndPassword$(encryptedEmail,
encryptedPassword).subscribe({
        next: (user) => {
          if (user[0]) {
            //Se digitado corretamente, usuário logado, salvo no sistema
            this.user = user[0];

            if (this.user) {
              //Então se é salvo o que foi digitado no local storage
              localStorage.setItem('userEmail', email);
              localStorage.setItem('userPassword', password);

              //Validar se estamos em algum chat
              let validation = false;
```

```
              this.user.chats.forEach(chatID => {
                if (this.chatID == chatID) {
                  validation = true;
                }
              });

              //Caso contrario, retornar ao dashboard
              if (this.chatID == '0' || !validation) {
                this.getRouter.navigate(['/dashboard']);
              }
            }
          }
        },
        complete: () => {
          if (this.user) {
            //Puxar informações: Chats, Amigos
            this.refreshFriendsList();
          }
          else { this.getRouter.navigate(['/login']); }
        },
        error: () => { alert('Erro: GetUserByEmailAndPassword'); }
      });
    });
  });
}

public refreshFriendsList() {
  this.friendsList = [];
  this.user?.chats.forEach(chatID => {
    this.getChatByID$(chatID).subscribe({
      next: (chat) => {
        chat[0].usersIDs.forEach(userID => {
          if (userID != this.user?.id) {
            this.getUserByID$(userID).subscribe({
              next: (user) => {

                let friendOnList = false;
                let newFriend = { chat: chat[0], user: user[0] };

                this.friendsList.forEach(friend => {
                  if ((newFriend.chat.id == friend.chat.id) &&
(newFriend.user.id == friend.user.id)) {
                    friendOnList = true;
                  }
                });
```

```
                if (!friendOnList) {
                  this.addToFriendList(newFriend);
                }
              }
            });
          }
        });
      }
    });
  });
}

//Autenticar o Usuário
public authenticateRegister(typedEmail: string, typedPassword: string, typedTag: string, typedUserName: string) {
  this.encryptText(typedEmail).then((encryptedEmail) => {
    this.getUserByEmail$(encryptedEmail).subscribe({
      next: (user) => {
        if (user[0]) { alert('Este Email já está cadastrado!'); }
        else {
          this.encryptText(typedPassword).then((encryptedPassword) => {
            let newUser: User = {
              id: typedTag.toLowerCase(),
              name: typedUserName,
              description: 'Hello, im new here!',
              status: 1,
              profilePicture: '../../../assets/images/profile.png',
              email: encryptedEmail,
              password: encryptedPassword,
              chats: []
            };

            this.postUser$(newUser).subscribe({
              complete: () => {
                localStorage.setItem('userEmail', typedEmail);
                localStorage.setItem('userPassword', typedPassword);
                this.getRouter.navigate(['/dashboard']);
              }
            });
          });

        }
      },
      complete: () => { },
      error: () => { alert('Erro: GetUserByEmail'); }
    });
```

```
    });
  }

  public logOut() {
    localStorage.setItem('userEmail', '');
    localStorage.setItem('userPassword', '');
    this.getRouter.navigate(['/login']);
  }
```

Documento *"HTML"* com o código da página inicial, também nomeada de *"Dashboard"*, que a depender da situação de uma variável, passada na *"URL"*, pode mudar o contexto da página tanto para a tela inicial quanto para as conversas privadas com algum amigo adicionado:

```
<main class="d-flex flex-sm-row">

    <button id="logout" class="btn btn-primary" (click)="logOut()">
        Sair
    </button>

<section id="lateral-bar" class="d-flex flex-column justify-content-between">

<div id="friends">
    <div id="profile-picture" class="d-flex">
        <img src="{{this.getUser()?.profilePicture}}">

        <div class="d-flex flex-column ps-2">
            <h4>
                {{this.getUser()?.name}}
            </h4>
            <p>
                #{{this.getUser()?.id}}
            </p>

        </div>
    </div>

<div>
    <div>
        <p>
        @for (chat of getFriendsList(); track $index)
        {
```

```
            @for (userID of chat.chat.usersIDs; track $index)
            {
                @if(userID != getUser()?.id)
                {

               @if (chat.user.id != getUser()?.id && chat.user.id == userID)
                    {
                        <div (click)="goToChat(chat.chat.id)" role="button"
class="d-flex flex-row" id="friends-cards">
                            <img id="friends-picture"
src="{{chat.user.profilePicture}}">
                            <p class="ps-2">{{chat.user.name}}</p>
                        </div>
                    }
                }
            }
        }
    </p>
    </div>
</div>
</div>

    <div id="add-friend">
        <input type="text" placeholder="Adicionar Pessoa"
[(ngModel)]="findUserTag" (keydown.enter)="sendUserTag()"
        class="form-control border-1 border-primary" />
    </div>
</section>

<section id="messages" class="d-flex flex-column justify-content-between w-
100">

@if (getChatID() == 'null' || getChatID() == '0')
{
    <div class="d-flex justify-content-center">

        <img id="welcome-image" src="../../../assets/images/welcome.png">

    </div>
}

<div id="history-messages" class="pe-2 ps-2 w-100">

        @for (chat of getFriendsList(); track $index)
        {
            @if(chat.chat.id == getChatID())
            {
```

```
                @for (message of chat.chat.messages.slice().reverse(); track
$index)
                {

                    @if(chat.user.id == message.userID)
                    {
                        <app-message [user]="chat.user" [message]="message"/>
                    }

                    @if(getUser()?.id == message.userID)
                    {

                        <app-message [user]="getUser()" [message]="message"
[right]="''"/>
                    }

                }
        }
    }
</div>

<div id="send-message">
    <div class="fixed">
        @if(getChatID() != 'null' && getChatID() != '0')
        {
            <input type="text" placeholder="Envie uma mensagem"
[(ngModel)]="message" (keydown.enter)="sendMessage()"
            class="form-control border-1 border-primary"/>

            <button (click)="sendMessage()" id="send-message">
                <img src="../../../assets/images/send-message.png">
            </button>
        }
    </div>
</div>
</section>
</main>
```

5. BIBLIOGRAFIA

DIAS, Vinicius - Comunicação em Tempo Real com WebSockets. Disponível em:<https://www.youtube.com/watch?v=QkhbQoajdCw>. Acesso em 06 de maio. 2025. FIRESHIP - WebSockets in 100 Seconds & Beyond with Socket.io Disponível em: <https://www.youtube.com/watch?v=1BfCnjr_Vjg>. Acesso em 06 de maio. 2025. GALVÃO, Luis - Protocolos de Redes de Computadores. Disponível em:<https://a3aengenharia.com.br/conteudo/artigos-tecnicos/protocolos-de-rede>. Acesso em 18 de maio. 2025.

4INFRA, Leandro – O que são protocolos de rede e para que servem? Disponível em:<https://4infra.com.br/o-que-sao-protocolos-de-rede>. Acesso em 18 de maio. 2025.