Name: Nitya Gupta

Roll No: 205123063

Course: MCA

SPIDER FINAL DOCUMENT

Problem Statement:

Create and implement a real-time chat application with a streamlined user interface using WebSockets.

● Establishing a secure WebSocket-based communication system to support private conversations and group chats. Ensure robust security measures to mitigate vulnerabilities and potential attacks.

● Facilitating the creation of multiple chat rooms with some kind of invitation features for varied discussions while maintaining a clean and intuitive user interface. Given the anticipated real-time usage by millions of users, prioritise:

● Developing a highly secure model to minimise vulnerabilities and risks, prioritising security over elaborate features.

● Constructing a scalable architecture proficient in managing a large influx of WebSocket connections.

● Implementing a load balancer to distribute client connections across servers evenly. Showcase the server's efficiency in managing high traffic by demonstrating effective request management.

● Evaluate and address the scaling challenges between HTTP and WebSockets, designing a solution emphasising scalability and security.

Table of Contents

1. **Introduction**
    - 1.1 Purpose
    - 1.2 Features
    - 1.3 Target Audience
    - 1.4 Technologies Used
2. **Getting Started**
    - 2.1 Installation
    - 2.2 Configuration
    - 2.3 Dependencies
3. **Architecture Overview**
    - 3.1 Frontend
    - 3.2 Backend
    - 3.3 Web Sockets
4. **User Guide**
    - 4.1 Registration and Login
    - 4.2 User Profile
    - 4.3 Messaging
    - 4.5 Media Sharing
5. **Technical Documentation**
    - 5.1 Frontend Structure
    - 5.2 Backend Structure
    - 5.3 Database Schema
    - 5.4 Web Socket Implementation
6. **Security**
    - 6.1 Encryption
    - 6.2 Authentication
    - 6.3 Authorization
    - 6.4 Data Protection

1. Introduction

1.1 Purpose

SpiderChat is a comprehensive social networking tool designed to facilitate seamless communication and media sharing among users. Connect effortlessly with others and explore the platform's technical aspects with this detailed documentation. Get ready for an easy and enriching social experience!

1.2 Features

SpiderChat boasts a diverse array of features, encompassing messaging, image sharing, and video sharing. Engage in real-time conversations with ease, elevate visual storytelling through seamless image sharing, and immerse yourself in multimedia communication with the platform's dynamic video-sharing functionality.

1.3 Target Audience

This documentation is intended for everyone interested in understanding and using the SpiderChat platform. Whether you're a developer customizing SpiderChat, an administrator optimizing its performance, or an enthusiast exploring its features, this documentation is your go-to resource.

1.4 Technologies Used

SpiderChat is built using the MERN (MongoDB, Express.js, React, Node.js) stack. Real-time communication is facilitated through Web Sockets. These technologies work together to create a robust and dynamic user experience.

2. Getting Started

2.1 Installation

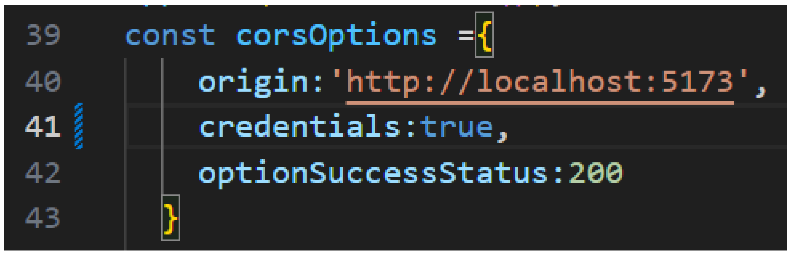
To install SpiderChat, follow the steps outlined in the installation guide provided in the repository.

Run the commands as follows:

1. *cd client*
2. *npm install*
3. *cd ..*
4. *cd api*
5. *npm install*

2.2 Configuration

Configure the application by modifying the necessary settings in the configuration files.



```
const corsOptions ={
    origin:'http://localhost:5173',
    credentials:true,
    optionSuccessStatus:200
  }
```

Change the current website URL origin to *http://localhost:5173*.

2.3 Dependencies

The component relies on various external dependencies such as React, lodash, axios, nodemon, bcrypt, express, cors, cloudinary, dotenv, mongoose, ws and custom components like Avatar, Logo, and Contact.

Ensure all dependencies are installed before running SpiderChat. Refer to the dependencies section for a complete list.

2.4 Start the project

Create .env file in client folder and add the following data:

```
VITE_API_BASE_URL = "http://localhost:4040/api"
```

Create .env file in api folder and add the following data:

```
MONGO_URL = 'mongodb+srv://mernchat:UZtwsekJGsnCWLhp@cluster0.4sl20dr.mongodb.net/?retryWrites=true&w=majority'

JWT_SECRET = 'nsbdhfgfhsjsbsj'

CLIENT_URL = 'http://localhost:5173/'

CLOUDINARY_NAME = "dmhwnntw1"

CLOUDINARY_API_KEY = "795837333735745"

CLOUDINARY_API_SECRET = "UrT8PjWk9JWkMDtRmfqb_KXVKS4"
```

Now, run the following commands:

1. *cd client*
2. *yarn dev*
3. *cd..*
4. *cd api*
5. *yarn run nodemon*

3. Architecture Overview

3.1 Frontend

The frontend is developed using React, offering a responsive and interactive user interface, leveraging the features of React.

3.2 Backend

The backend is powered by Node.js and Express.js, serving as the server-side logic and API endpoints.

3.3 Web Sockets

Real-time communication is achieved through Web Sockets, a protocol that facilitates instant messaging and other dynamic features by enabling a continuous and bi-directional connection between the server and clients.

4. User Guide

4.1 Registration and Login

Users can securely create accounts and log in to access SpiderChat, ensuring a safe and protected user experience.

4.2 User Profile

Customize user profiles with personal information and preferences.

4.3 Messaging

Chat with friends and contacts in real-time using the messaging feature.

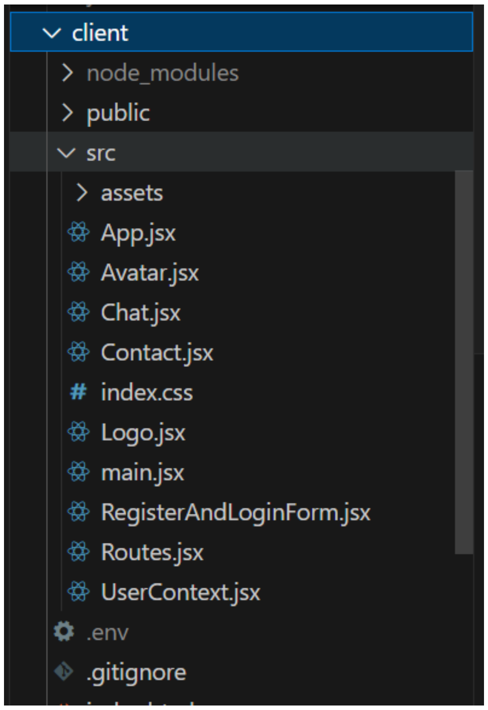
4.4 Media Sharing

Share photos, videos, and other media seamlessly.

5. Technical Documentation

5.1 Frontend Structure

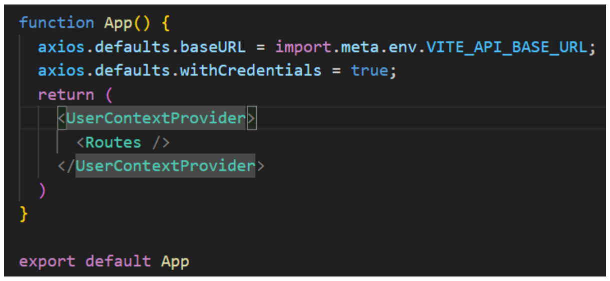
Frontend structure of SpiderChat is as follows:



```
client
  node_modules
  public
  src
    assets
    App.jsx
    Avatar.jsx
    Chat.jsx
    Contact.jsx
    index.css
    Logo.jsx
    main.jsx
    RegisterAndLoginForm.jsx
    Routes.jsx
    UserContext.jsx
  .env
  .gitignore
```

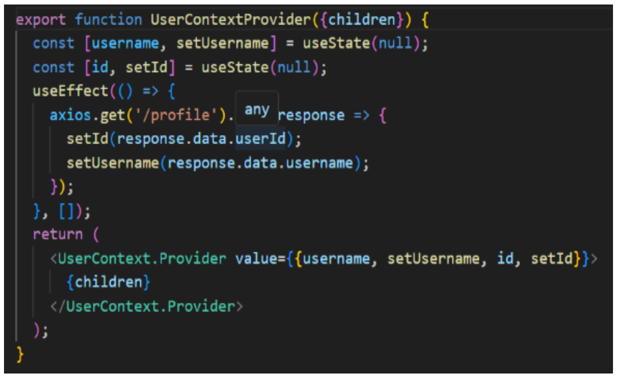
In the context of the frontend files coded using the React library, the initial point of execution is the **app.jsx** file. This file serves as the root of the application, analogous to the root of a tree, with the subsequent data and folders acting as nodes branching from this central point.

In the **App.jsx** file, the frontend URL is configured within the Axios library. Axios is employed to handle API requests originating from the frontend, facilitating their transmission to the backend.



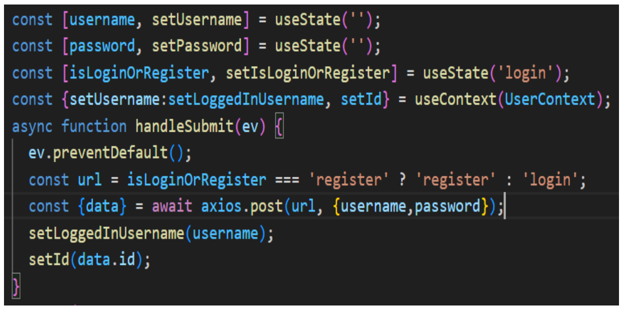
```
function App() {
  axios.defaults.baseURL = import.meta.env.VITE_API_BASE_URL;
  axios.defaults.withCredentials = true;
  return (
    <UserContextProvider>
      <Routes />
    </UserContextProvider>
  )
}

export default App
```

Additionally, all of the application's routes is enveloped within a **UserContextProvider**. The structure of the **UserContextProvider** is outlined below:



```
export function UserContextProvider({children}) {
  const [username, setUsername] = useState(null);
  const [id, setId] = useState(null);
  useEffect(() => {
    axios.get('/profile'). any response => {
      setId(response.data.userId);
      setUsername(response.data.username);
    });
  }, []);
  return (
    <UserContext.Provider value={{username, setUsername, id, setId}}>
      {children}
    </UserContext.Provider>
  );
}
```

Within this context, the initiation of a "get profile" request is executed within the **useEffect**. When this function is invoked, the **useEffect** is triggered initially. Subsequently, the **useEffect** sets the **username** and **id** states based on the current user details, assigning them values or setting them to null if no details are available. Following this, the function dispatches these details to all the routes, ensuring that the information is accessible throughout the application.

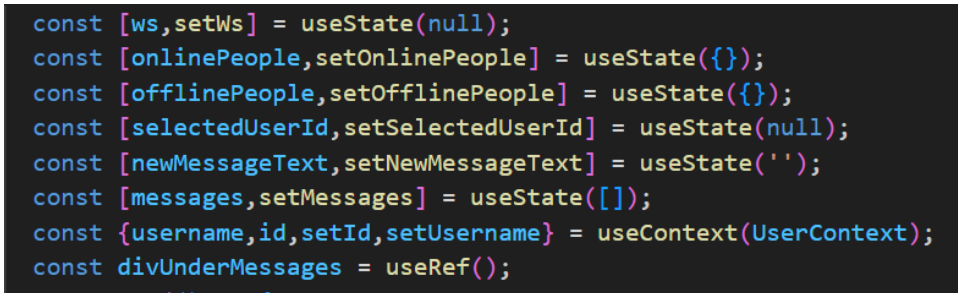


```
const [username, setUsername] = useState('');
const [password, setPassword] = useState('');
const [isLoginOrRegister, setIsLoginOrRegister] = useState('login');
const {setUsername:setLoggedInUsername, setId} = useContext(UserContext);
async function handleSubmit(ev) {
  ev.preventDefault();
  const url = isLoginOrRegister === 'register' ? 'register' : 'login';
  const {data} = await axios.post(url, {username,password});
  setLoggedInUsername(username);
  setId(data.id);
}
```

Above code snippet describes a login/register function within the application. When a user clicks on the login/register action, the **isLoginOrRegister** state is set to the corresponding value. Subsequently, an Axios POST API request is sent to the backend. After a successful request, the **setLoggedInUsername** function from the context is called to update the logged-in ‘username’. Additionally, the 'id' from the response data is set using the **setId** function, for further user identification.

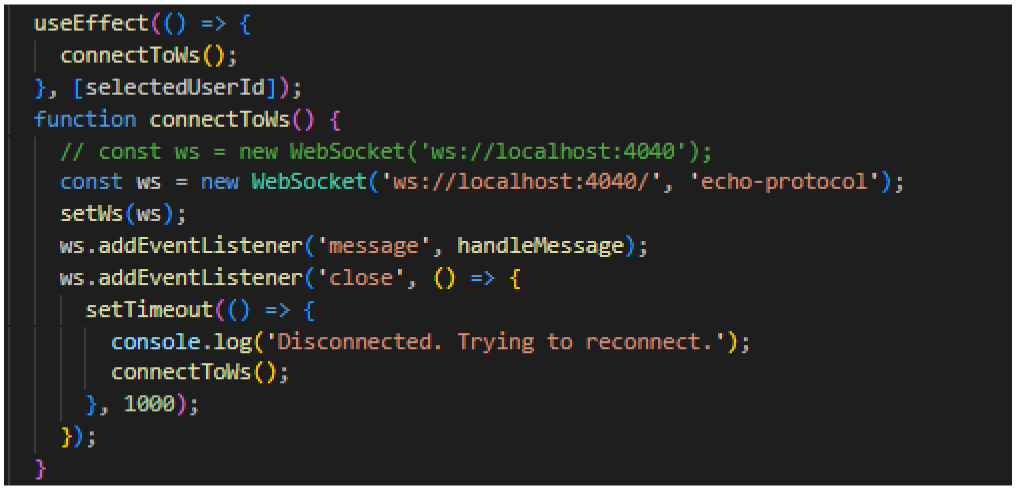
This process implies that upon login or registration, the application communicates with the backend to authenticate the user and obtain relevant user information, updating the local state accordingly for subsequent use within the application.

After logging in it will show the chat component. The states in chat component are as:



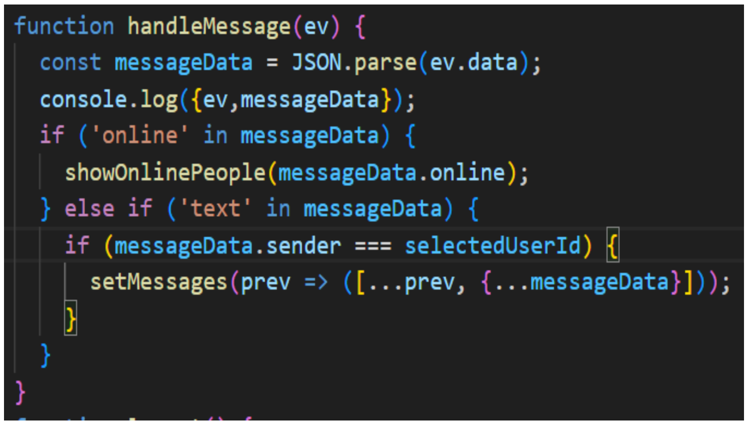
```
const [ws,setWs] = useState(null);
const [onlinePeople,setOnlinePeople] = useState({});
const [offlinePeople,setOfflinePeople] = useState({});
const [selectedUserId,setSelectedUserId] = useState(null);
const [newMessageText,setNewMessageText] = useState('');
const [messages,setMessages] = useState([]);
const {username,id,setId,setUsername} = useContext(UserContext);
const divUnderMessages = useRef();
```

Prior to rendering, the application establishes a connection with the WebSocket server, associating a WebSocket connection when the **selectedUserId** dependency changes. The WebSocket connection is created using the **connectToWs** function. The component sets up event listeners for the WebSocket instance to handle incoming messages (**message** event) and reconnection logic (**close** event).



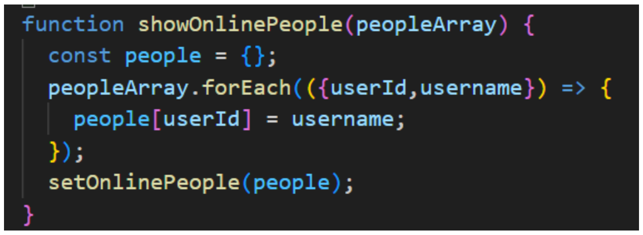
```
useEffect(() => {
  connectToWs();
}, [selectedUserId]);
function connectToWs() {
  // const ws = new WebSocket('ws://localhost:4040');
  const ws = new WebSocket('ws://localhost:4040/', 'echo-protocol');
  setWs(ws);
  ws.addEventListener('message', handleMessage);
  ws.addEventListener('close', () => {
    setTimeout(() => {
      console.log('Disconnected. Trying to reconnect.');
      connectToWs();
    }, 1000);
  });
}
```

Once the WebSocket server receives a message or a user list, it triggers the execution of the **handleMessage** function. This function is designed to process and respond to incoming messages , which are assumed to be in JSON format, or user lists received from the WebSocket server, ensuring that the application can dynamically update and respond to real-time events. If the message contains an 'online' property, it calls **showOnlinePeople** to update the list of online users. If the message contains a 'text' property and the sender's ID matches the currently selected user (**selectedUserId**), it updates the state variable **messages** with the new message data.



```
function handleMessage(ev) {
  const messageData = JSON.parse(ev.data);
  console.log({ev,messageData});
  if ('online' in messageData) {
    showOnlinePeople(messageData.online);
  } else if ('text' in messageData) {
    if (messageData.sender === selectedUserId) {
      setMessages(prev => ([...prev, {...messageData}]));
    }
  }
}
```

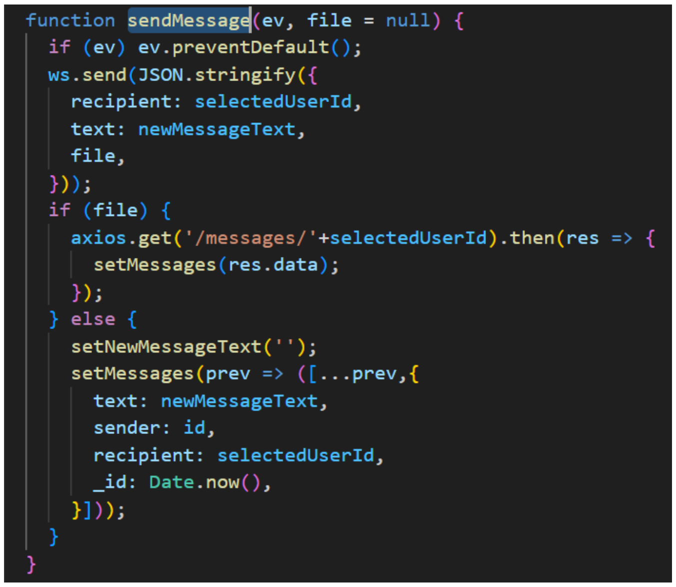
In the described functionality, when the WebSocket server receives data, the application calls the **showOnlinePeople** function. The functionality you've described involves obtaining an array of people currently online in the application. This array is then used to update the **onlinePeople** state with the relevant data. By doing so, the application maintains an updated list of users who are currently online, allowing for real-time tracking of online status. This information can be valuable for features such as displaying online users, initiating conversations, or other user-related interactions in the application.



```
function showOnlinePeople(peopleArray) {
  const people = {};
  peopleArray.forEach(({userId,username}) => {
    people[userId] = username;
  });
  setOnlinePeople(people);
}
```

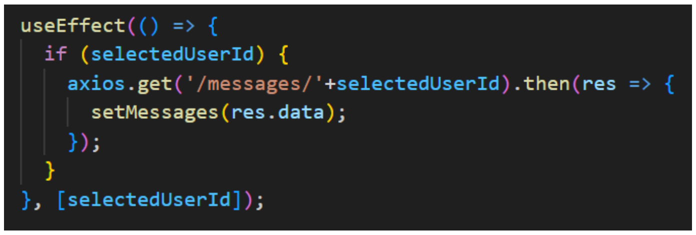
So,the mechanism implies that the application dynamically manages online users through the **showOnlinePeople** function and updates the message state selectively based on the sender's identity in relation to the current user.

In the scenario described, when a user sends a message, a function is triggered. This function is responsible for sending all the relevant details to the WebSocket server. The data transmitted typically includes information such as the recipient, the actual message content, and any files if they are attached or available. By sending this data to the WebSocket server, the application ensures that the message is appropriately processed and transmitted to the intended recipient, facilitating real-time communication within the application.



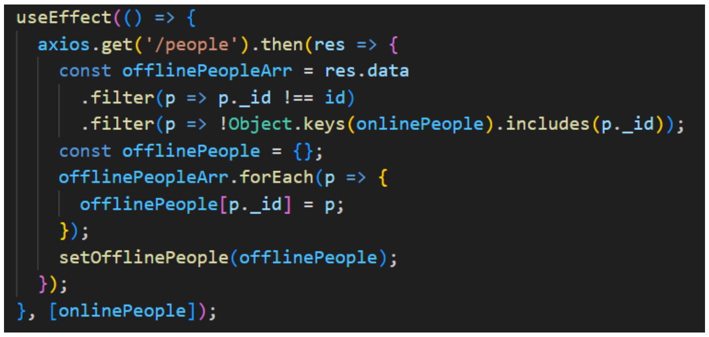
```
function sendMessage(ev, file = null) {
  if (ev) ev.preventDefault();
  ws.send(JSON.stringify({
    recipient: selectedUserId,
    text: newMessageText,
    file,
  }));
  if (file) {
    axios.get('/messages/'+selectedUserId).then(res => {
      setMessages(res.data);
    });
  } else {
    setNewMessageText('');
    setMessages(prev => ([...prev,{
      text: newMessageText,
      sender: id,
      recipient: selectedUserId,
      _id: Date.now(),
    }]));
  }
}
```

useEffect:



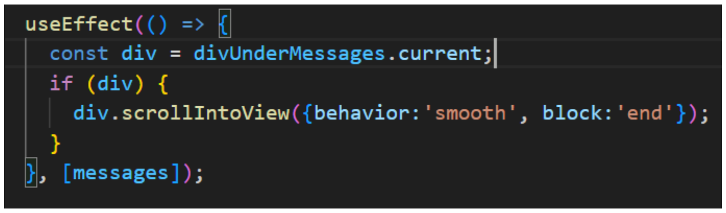
```
useEffect(() => {
  if (selectedUserId) {
    axios.get('/messages/'+selectedUserId).then(res => {
      setMessages(res.data);
    });
  }
}, [selectedUserId]);
```

It will set the messages state with all the messages if the current user selects any user from the user list. This useEffect fetches messages from the server for the specified selectedUserId. When selectedUserId changes, it triggers an axios GET request to the /messages/${selectedUserId} endpoint. The received messages are then updated in the component's state using setMessages.



```
useEffect(() => {
  axios.get('/people').then(res => {
    const offlinePeopleArr = res.data
      .filter(p => p._id !== id)
      .filter(p => !Object.keys(onlinePeople).includes(p._id));
    const offlinePeople = {};
    offlinePeopleArr.forEach(p => {
      offlinePeople[p._id] = p;
    });
    setOfflinePeople(offlinePeople);
  });
}, [onlinePeople]);
```

It will set offline people array state from the whole user list after applying filters. This useEffect fetches a list of people from the server via an axios GET request to the '/people' endpoint. It then filters out the current user (with the ID 'id') and those already present in the onlinePeople state. The remaining offline people are organized into a mapping by ID and stored in the offlinePeople state. This effect re-runs whenever the onlinePeople state changes, typically reflecting users who are currently online in the application.

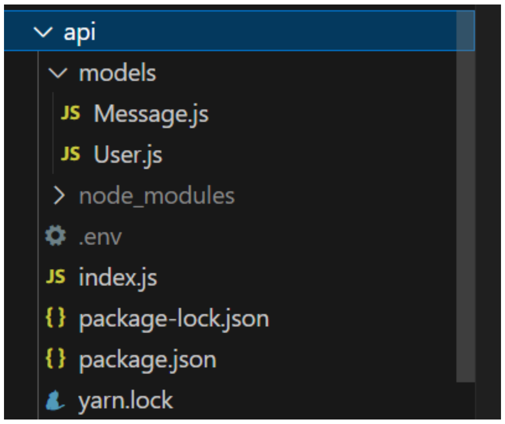


```
useEffect(() => {
  const div = divUnderMessages.current;
  if (div) {
    div.scrollIntoView({behavior:'smooth', block:'end'});
  }
}, [messages]);
```

It will scroll down to the bottom of the chat if a new message is encountered. This useEffect scrolls a DOM element referenced by divUnderMessages into view with a smooth animation when the messages state changes. It first checks if the divUnderMessages reference exists to avoid errors. If the reference exists, it utilizes the scrollIntoView method with the specified options ({behavior:'smooth', block:'end'}) to smoothly scroll the referenced element into view, ensuring the latest message is visible at the bottom. This pattern is commonly used to automatically scroll to the latest message in a chat-like interface.

5.2 Backend Structure

Backend structure of SpiderChat is as follows:

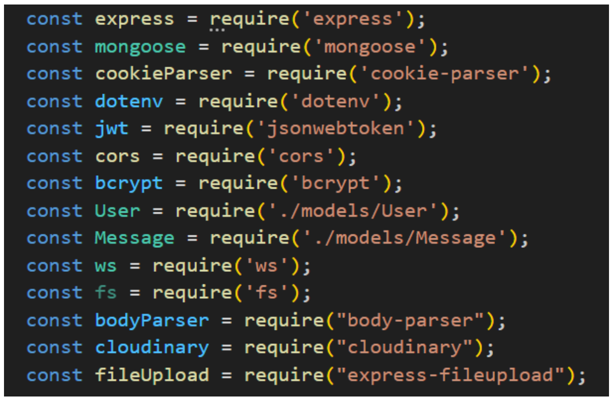


```
api
  models
    Message.js
    User.js
  node_modules
  .env
  index.js
  package-lock.json
  package.json
  yarn.lock
```

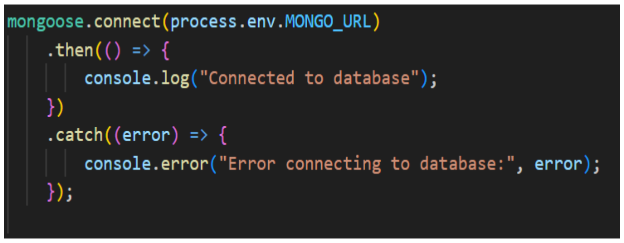
These are the backend files coded using node and express js. Now let's explain the flow of backend.

The model contains the database schema which will explained in the upcoming points.

Here index.jsx is the main file. The states in index.jsx are as:

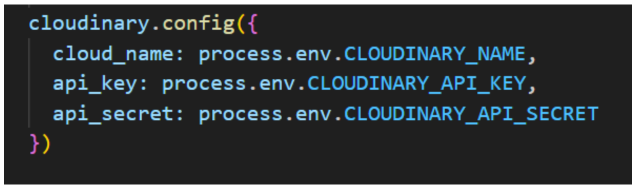


```
const express = require('express');
const mongoose = require('mongoose');
const cookieParser = require('cookie-parser');
const dotenv = require('dotenv');
const jwt = require('jsonwebtoken');
const cors = require('cors');
const bcrypt = require('bcrypt');
const User = require('./models/User');
const Message = require('./models/Message');
const ws = require('ws');
const fs = require('fs');
const bodyParser = require("body-parser");
const cloudinary = require("cloudinary");
const fileUpload = require("express-fileupload");
```



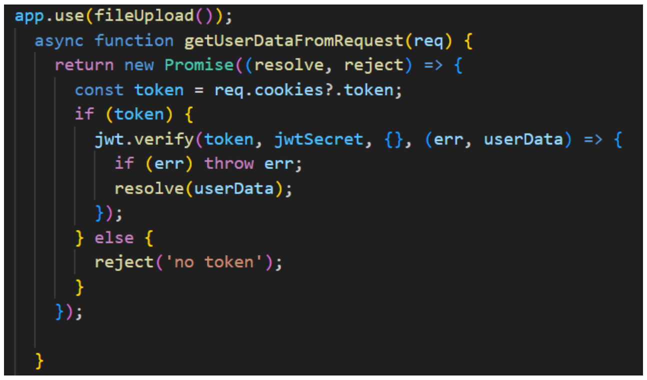
```
mongoose.connect(process.env.MONGO_URL)
    .then(() => {
        console.log("Connected to database");
    })
    .catch((error) => {
        console.error("Error connecting to database:", error);
    });
```

This code snippet connects to a MongoDB database using Mongoose. It utilizes the mongoose.connect method with the MongoDB URL provided through the process.env.MONGO_URL. If the connection is successful, it logs a message indicating a successful connection; otherwise, it catches and logs any errors that occur during the connection attempt.



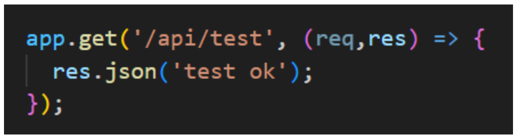
```
cloudinary.config({
  cloud_name: process.env.CLOUDINARY_NAME,
  api_key: process.env.CLOUDINARY_API_KEY,
  api_secret: process.env.CLOUDINARY_API_SECRET
})
```

This code configures Cloudinary settings using the cloudinary.config method. It sets the Cloudinary cloud name, API key, and API secret by retrieving these values from environment variables (process.env). This configuration enables the application to interact with Cloudinary services for managing and serving media assets.



```
app.use(fileUpload());
  async function getUserDataFromRequest(req) {
    return new Promise((resolve, reject) => {
      const token = req.cookies?.token;
      if (token) {
        jwt.verify(token, jwtSecret, {}, (err, userData) => {
          if (err) throw err;
          resolve(userData);
        });
      } else {
        reject('no token');
      }
    });

  }
```

These lines of code utilize the fileUpload middleware in an Express application using app.use(fileUpload()). The following function, getUserDataFromRequest, is defined to extract user data from the request object, particularly the user's information stored in a JSON Web Token (JWT) within the 'token' cookie. The function returns a promise that resolves with the user data if a valid token is present; otherwise, it rejects with the message 'no token'.

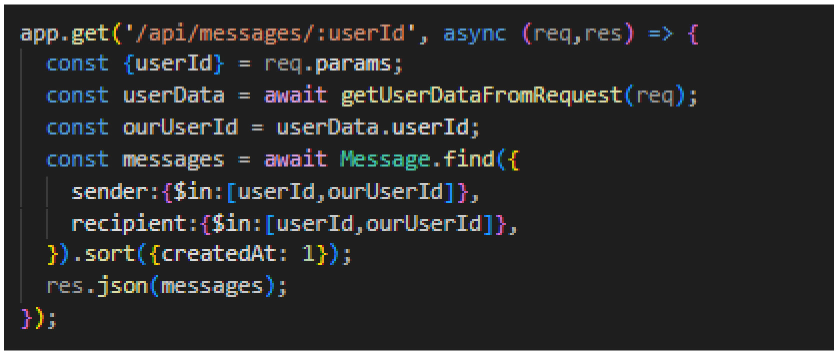


```
app.get('/api/test', (req,res) => {
  res.json('test ok');
});
```

It is a backend test api which has two parameters: **req** which means request and **res** means response. It will return response after converting the data into json format. When a request is made to this endpoint, it sends a JSON response with the string 'test ok' for testing or confirming that the API is functioning correctly.

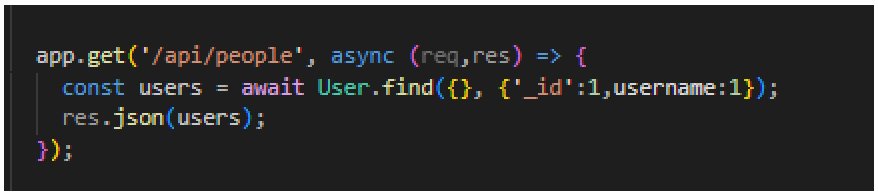
Rest of the API's used in the tool are briefly described below:

Get Messages for a User: The "Get Messages for a User" API (GET /api/messages/:userId) retrieves chat messages between the authenticated user and another user specified by userId. Utilizing the getUserDataFromRequest middleware for user authentication, it ensures that only authorized users can access their conversation history. Messages are sorted by creation time, providing a chronological view of communication.



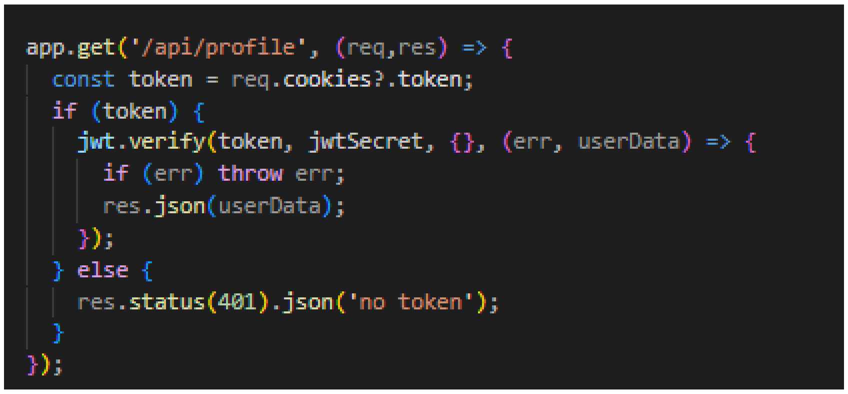
```
app.get('/api/messages/:userId', async (req,res) => {
  const {userId} = req.params;
  const userData = await getUserDataFromRequest(req);
  const ourUserId = userData.userId;
  const messages = await Message.find({
    sender:{$in:[userId,ourUserId]},
    recipient:{$in:[userId,ourUserId]},
  }).sort({createdAt: 1});
  res.json(messages);
});
```

Get List of Users: The "Get List of Users" API (GET /api/people) fetches a list of users from the database, including their usernames and IDs. This endpoint is useful for obtaining information about all users available in the system. It serves as a comprehensive user directory.



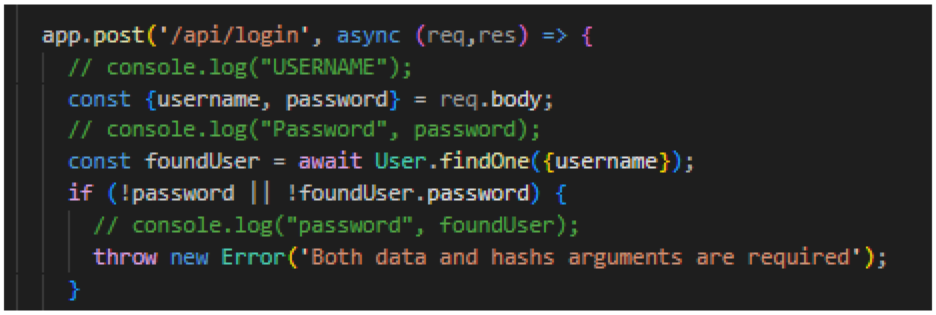
```
app.get('/api/people', async (req,res) => {
  const users = await User.find({}, {'_id':1,username:1});
  res.json(users);
});
```

Get User Profile: The "Get User Profile" API (GET /api/profile) retrieves profile data, including user information, of the authenticated user. JWT verification middleware ensures that the request is made by an authenticated user, providing secure access to their own profile information.

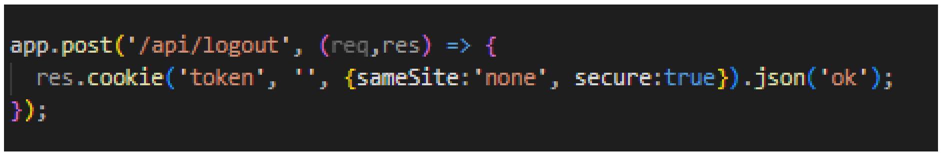


```
app.get('/api/profile', (req,res) => {
  const token = req.cookies?.token;
  if (token) {
    jwt.verify(token, jwtSecret, {}, (err, userData) => {
      if (err) throw err;
      res.json(userData);
    });
  } else {
    res.status(401).json('no token');
  }
});
```

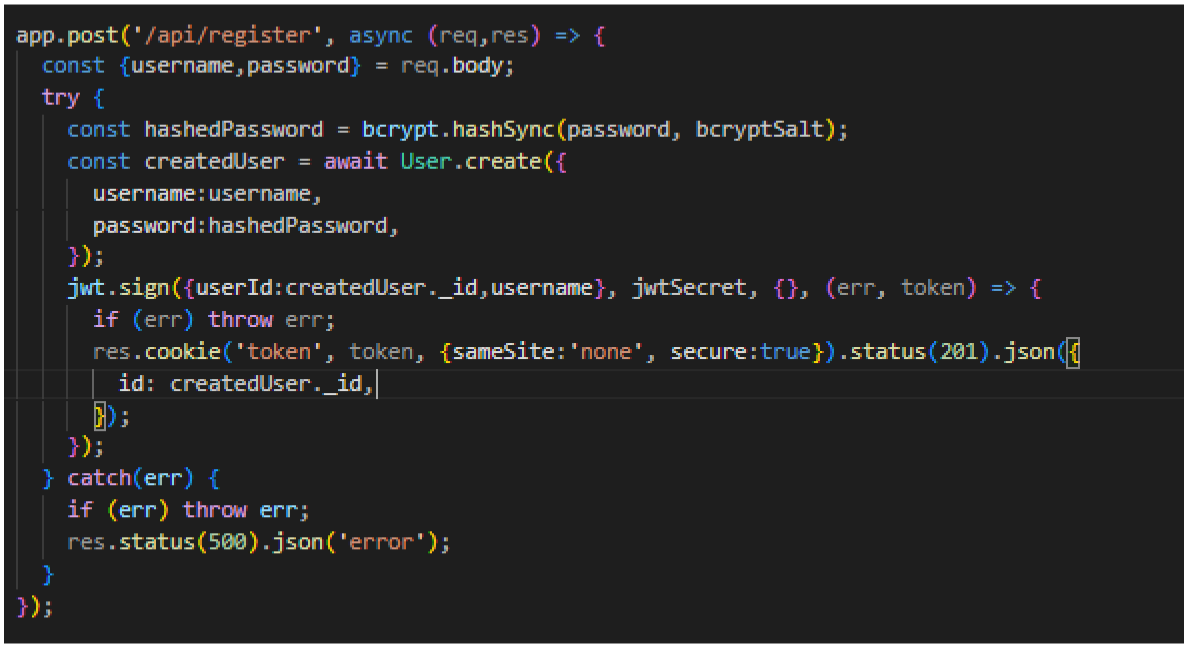
User Login: The "User Login" API (POST /api/login) handles user authentication by comparing the provided username and password with the stored user data. Upon successful login, it generates a JWT token, included in the response. A token cookie is also set, enabling seamless authentication for subsequent requests.


```
app.post('/api/login', async (req,res) => {
  // console.log("USERNAME");
  const {username, password} = req.body;
  // console.log("Password", password);
  const foundUser = await User.findOne({username});
  if (!password || !foundUser.password) {
    // console.log("password", foundUser);
    throw new Error('Both data and hashs arguments are required');
  }
```

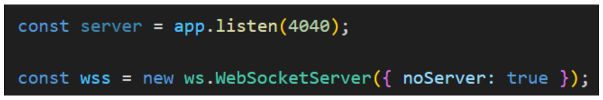
User Logout: The "User Logout" API (POST /api/logout) facilitates user logout by clearing the token cookie. It provides a secure way for users to end their authenticated sessions, ensuring they are no longer authenticated for future requests.


```
app.post('/api/logout', (req,res) => {
  res.cookie('token', '', {sameSite:'none', secure:true}).json('ok');
});
```

User Registration: The "User Registration" API (POST /api/register) registers a new user by hashing the provided password and creating a new user document in the database. Upon successful registration, it generates a JWT token, included in the response, and sets a token cookie for continued authentication in subsequent requests. This endpoint enables the creation of new user accounts with proper security measures.


```
app.post('/api/register', async (req,res) => {
  const {username,password} = req.body;
  try {
    const hashedPassword = bcrypt.hashSync(password, bcryptSalt);
    const createdUser = await User.create({
      username:username,
      password:hashedPassword,
    });
    jwt.sign({userId:createdUser._id,username}, jwtSecret, {}, (err, token) => {
      if (err) throw err;
      res.cookie('token', token, {sameSite:'none', secure:true}).status(201).json({
        id: createdUser._id,
      });
    });
  } catch(err) {
    if (err) throw err;
    res.status(500).json('error');
  }
});
```

These APIs collectively provide functionality for user authentication, messaging, user data retrieval, and user registration within a real-time chat application.

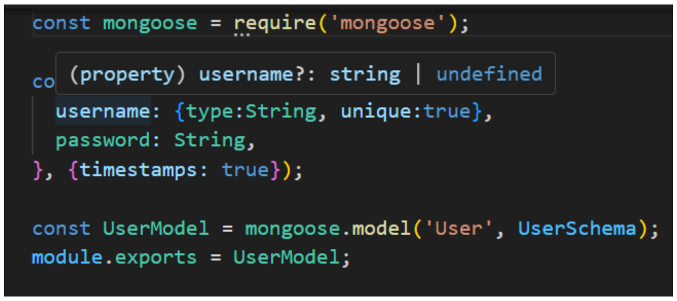


```
const server = app.listen(4040);

const wss = new ws.WebSocketServer({ noServer: true });
```

It will create an HTTP server using Express, listening on port 4040, and a WebSocket server using the ws library. The app.listen(4040) creates an HTTP server, and the resulting server is then used to instantiate a WebSocketServer (wss) using the ws library,allowing the server to handle both types of communication simultaneously.

5.3 Database Schema

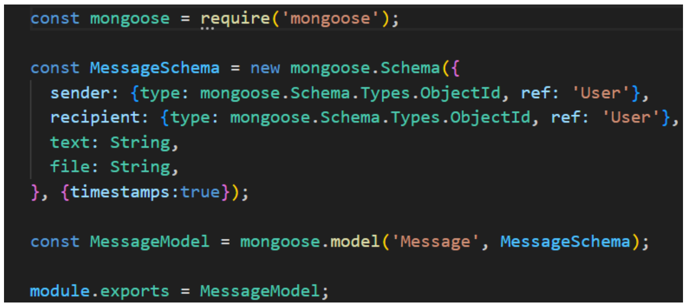
User Schema:



```
const mongoose = require('mongoose');

co (property) username?: string | undefined
  username: {type:String, unique:true},
  password: String,
}, {timestamps: true});

const UserModel = mongoose.model('User', UserSchema);
module.exports = UserModel;
```

These lines define a MongoDB schema and model for a user using Mongoose. The UserSchema specifies the structure of user documents with a unique username and a password. The model is created with mongoose.model named 'User' based on the defined schema, and it is exported for use in other parts of the application.

Message Schema:

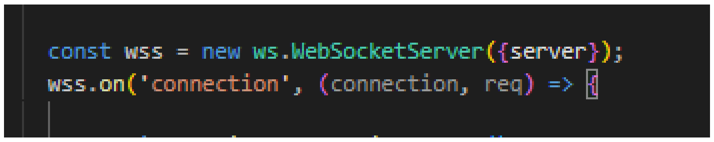


```
const mongoose = require('mongoose');

const MessageSchema = new mongoose.Schema({
  sender: {type: mongoose.Schema.Types.ObjectId, ref: 'User'},
  recipient: {type: mongoose.Schema.Types.ObjectId, ref: 'User'},
  text: String,
  file: String,
}, {timestamps:true});

const MessageModel = mongoose.model('Message', MessageSchema);

module.exports = MessageModel;
```

These lines define a MongoDB schema and model for a message using Mongoose. The MessageSchema specifies the structure of message documents with sender and recipient references to the 'User' model, a text field, and a file field. The model is created with mongoose.model named 'Message' based on the defined schema and exported for use in other parts of the application.

5.4 Web Socket Implementation

Learn how Web Sockets are implemented for real-time communication here. The following code implements a WebSocket server using the ws library in Node.js.



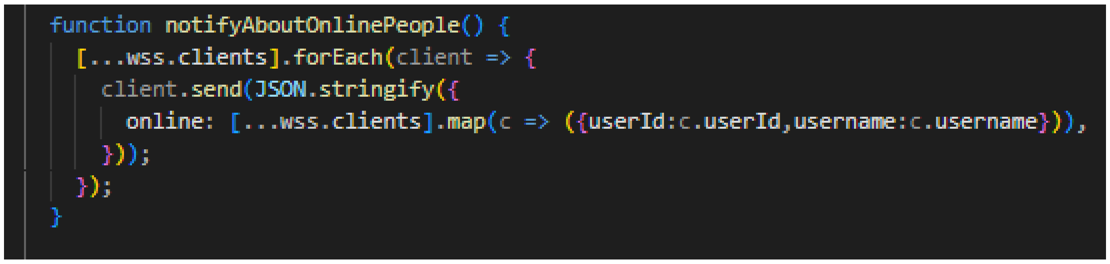
```
const wss = new ws.WebSocketServer({server});
wss.on('connection', (connection, req) => {
```

In above code snippet,the server listens for WebSocket connections and executes a callback when a new connection is established.

- **wss.on('connection', ...)**: This line establishes an event listener for the **'connection'** event on the WebSocket server (**wss**).
- **(connection, req) => { ... }**: The callback function is executed whenever a new WebSocket connection is established:
- **connection**: Represents the WebSocket connection object for the newly connected client. This object allows interaction with the client, such as sending and receiving messages.
- **req**: Represents the HTTP request associated with the WebSocket connection. It provides information about the client's request, such as headers.

Here's a brief overview of its functionality:

A function **notifyAboutOnlinePeople** is defined to notify all connected clients about the current online users. When a new connection is established, the server notifies all connected clients about the current online users.

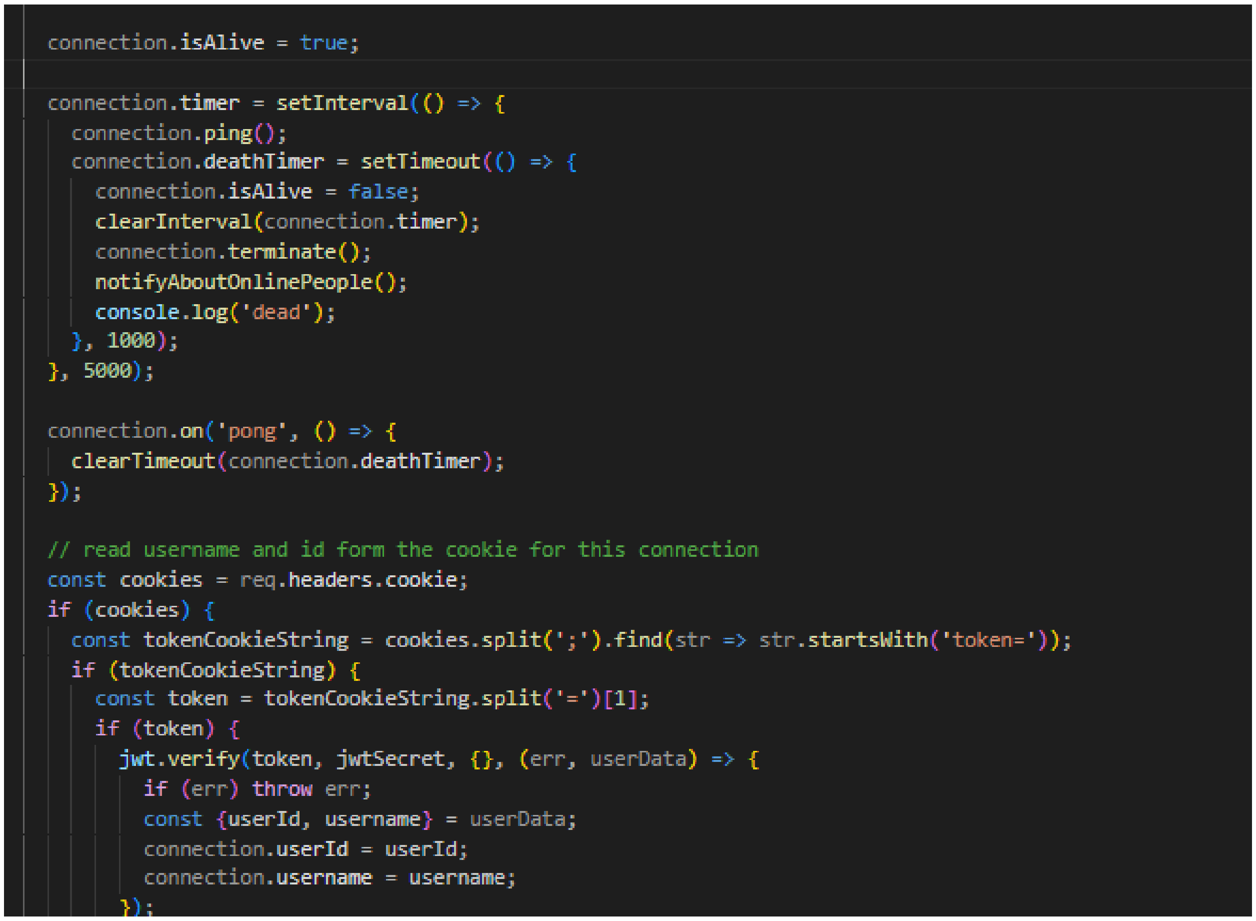


```
function notifyAboutOnlinePeople() {
  [...wss.clients].forEach(client => {
    client.send(JSON.stringify({
      online: [...wss.clients].map(c => ({userId:c.userId,username:c.username})),
    }));
  });
}
```

Each connected client has a heartbeat mechanism to check its liveliness, with periodic ping-pong messages.For each connection:

- A heartbeat mechanism is implemented using ping and pong messages to check if the connection is alive.
- The user information (userId and username) is read from the JWT token in the request's cookie.
- A timer is set to periodically send ping messages and terminate the connection if not responded within a certain time.

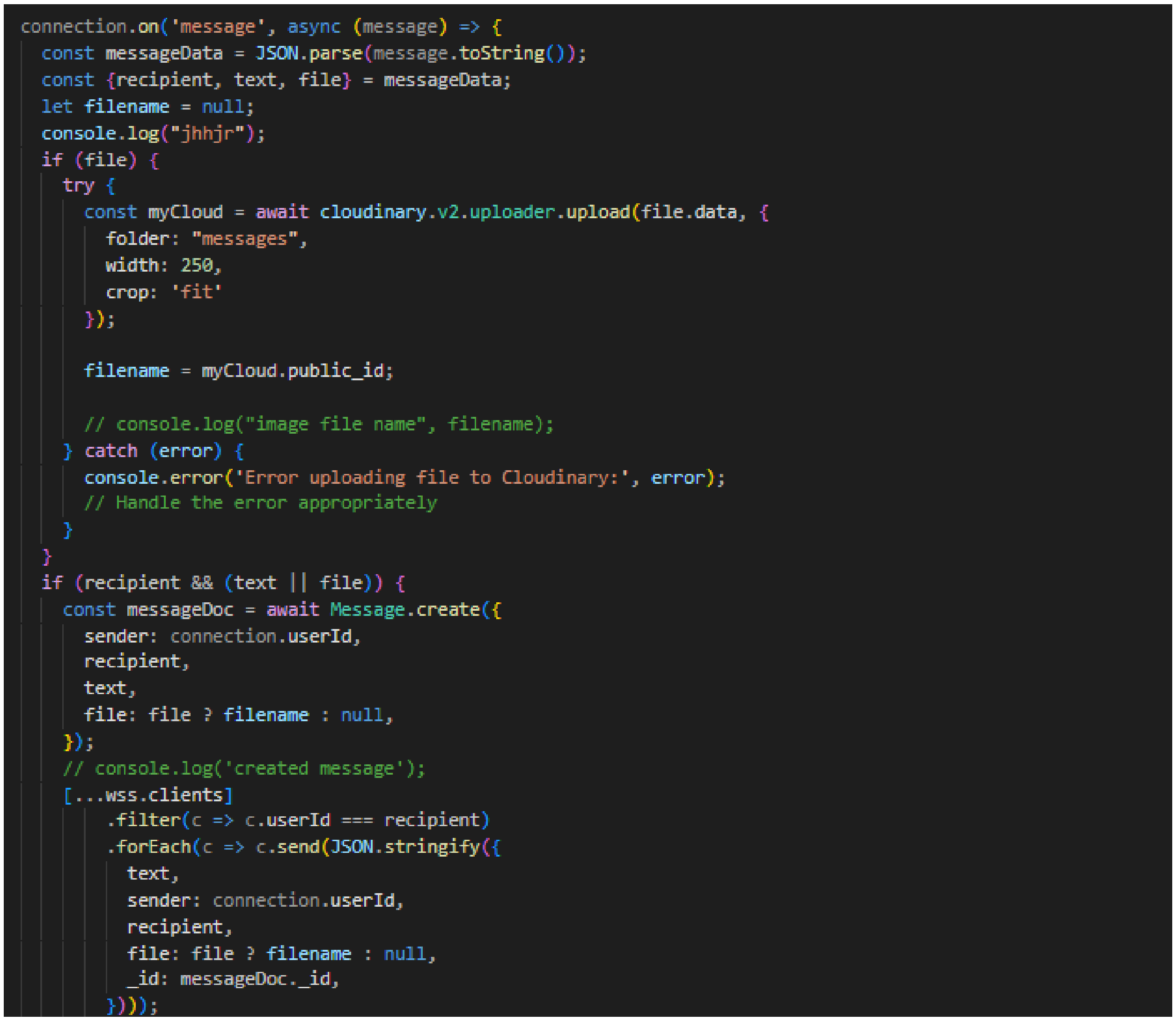
Above mechanism is implemented through following code snippet:



```
connection.isAlive = true;

connection.timer = setInterval(() => {
  connection.ping();
  connection.deathTimer = setTimeout(() => {
    connection.isAlive = false;
    clearInterval(connection.timer);
    connection.terminate();
    notifyAboutOnlinePeople();
    console.log('dead');
  }, 1000);
}, 5000);

connection.on('pong', () => {
  clearTimeout(connection.deathTimer);
});

// read username and id form the cookie for this connection
const cookies = req.headers.cookie;
if (cookies) {
  const tokenCookieString = cookies.split(';').find(str => str.startsWith('token='));
  if (tokenCookieString) {
    const token = tokenCookieString.split('=')[1];
    if (token) {
      jwt.verify(token, jwtSecret, {}, (err, userData) => {
        if (err) throw err;
        const {userId, username} = userData;
        connection.userId = userId;
        connection.username = username;
      });
```

When a message is received:

- The message data is parsed, and if it contains a file, it is uploaded to Cloudinary.
- A new message document is created in the database using Mongoose.
- The message is broadcasted to the recipient's WebSocket connection, notifying them about the new message.



```
connection.on('message', async (message) => {
  const messageData = JSON.parse(message.toString());
  const {recipient, text, file} = messageData;
  let filename = null;
  console.log("jhhjr");
  if (file) {
    try {
      const myCloud = await cloudinary.v2.uploader.upload(file.data, {
        folder: "messages",
        width: 250,
        crop: 'fit'
      });

      filename = myCloud.public_id;

      // console.log("image file name", filename);
    } catch (error) {
      console.error('Error uploading file to Cloudinary:', error);
      // Handle the error appropriately
    }
  }
  if (recipient && (text || file)) {
    const messageDoc = await Message.create({
      sender: connection.userId,
      recipient,
      text,
      file: file ? filename : null,
    });
    // console.log('created message');
    [...wss.clients]
      .filter(c => c.userId === recipient)
      .forEach(c => c.send(JSON.stringify({
        text,
        sender: connection.userId,
        recipient,
        file: file ? filename : null,
        _id: messageDoc._id,
      })));
```

So it can be concluded that ,this implementation combines WebSocket functionality for real-time communication with Cloudinary integration for file uploads. The server efficiently manages connections, heartbeat checks, and message broadcasting to create a real-time chat application. Additionally, it leverages Cloudinary for handling and storing file uploads associated with messages.

6. Security

6.1 Encryption

Ensure end-to-end encryption for secure communication. The user's password is hashed using the bcrypt library. Hashing is a one-way encryption process that transforms the password into a fixed-length string, making it computationally difficult for attackers to reverse the process and obtain the original password.

```
const hashedPassword = bcrypt.hashSync(password, bcryptSalt);
```

6.2 Authentication

Understand the authentication mechanisms in place to protect user accounts. After successfully creating a new user in the database, a JSON Web Token (JWT) is signed using the jwt.sign method. This token contains user information (userId and username). The signed token is then set as a cookie in the HTTP response, enabling the user to be authenticated in subsequent requests.

```
jwt.sign({userId:createdUser._id,username}, jwtSecret, {}, (err, token) => {
  if (err) throw err;
  res.cookie('token', token, {sameSite:'none', secure:true}).status(201).json({
    id: createdUser._id,
  });
});
```

6.3 Authorization

The JWT token is set as a cookie in the response. This cookie serves as a form of authorization, allowing the user to present this token in subsequent requests to prove their identity and access authorized resources.

```
res.cookie('token', token, {sameSite:'none', secure:true}).status(201).json({
  id: createdUser._id,
});
```

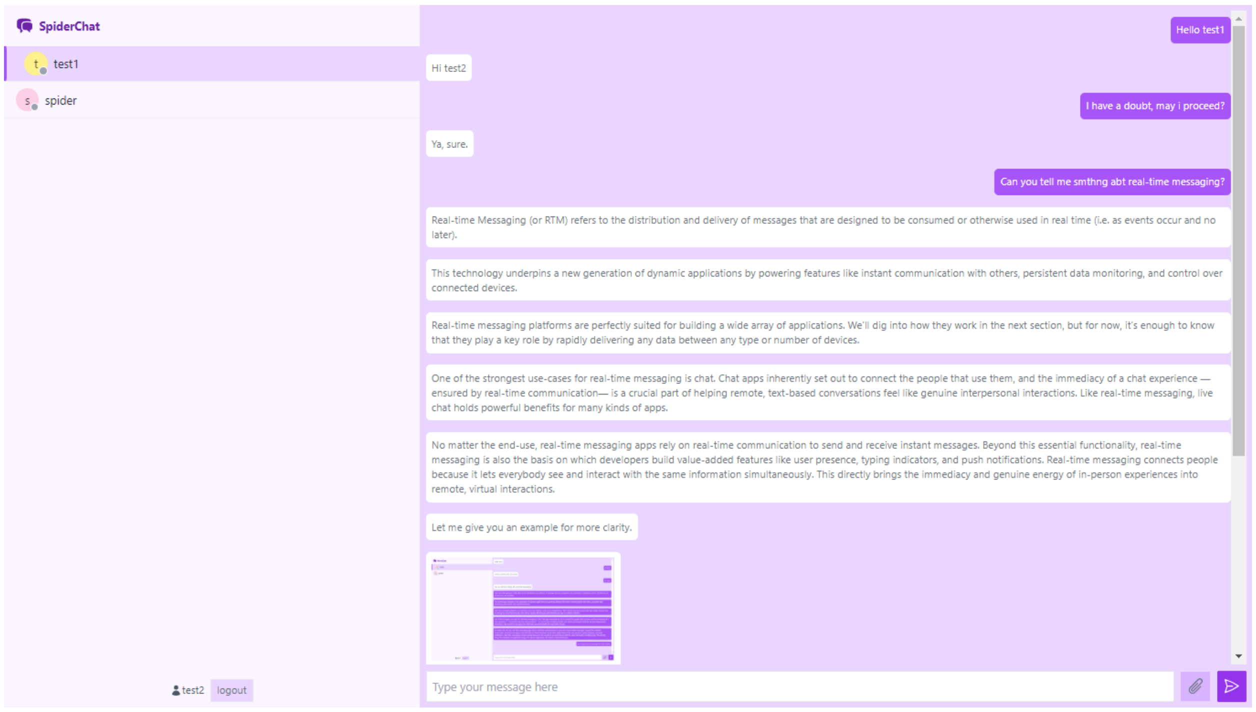
6.4 Data Protection

Follow best practices for data protection and privacy. A try-catch block is used to handle errors during the user registration process. If any errors occur (e.g., database insertion failure), a generic error response with a status code of 500 is sent. This helps protect sensitive information by not exposing detailed error messages to potential attackers.

```
try {
  // ... (code for encryption, authentication, and authorization)
} catch(err) {
  if (err) throw err;
  res.status(500).json('error');
}
```

In summary, the code snippet demonstrates a secure user registration process by encrypting passwords, using JWT for authentication and authorization, and protecting sensitive data through proper error handling.

This documentation serves as a comprehensive guide to understand, install, configure, and utilize SpiderChat. If you have any questions, concerns, or suggestions, feel free to reach out to me. Thank you for choosing SpiderChat for your social networking needs!



Hello test1

Hi test2

I have a doubt, may i proceed?

Ya, sure.

Can you tell me smthng abt real-time messaging?

Real-time Messaging (or RTM) refers to the distribution and delivery of messages that are designed to be consumed or otherwise used in real time (i.e. as events occur and no later).

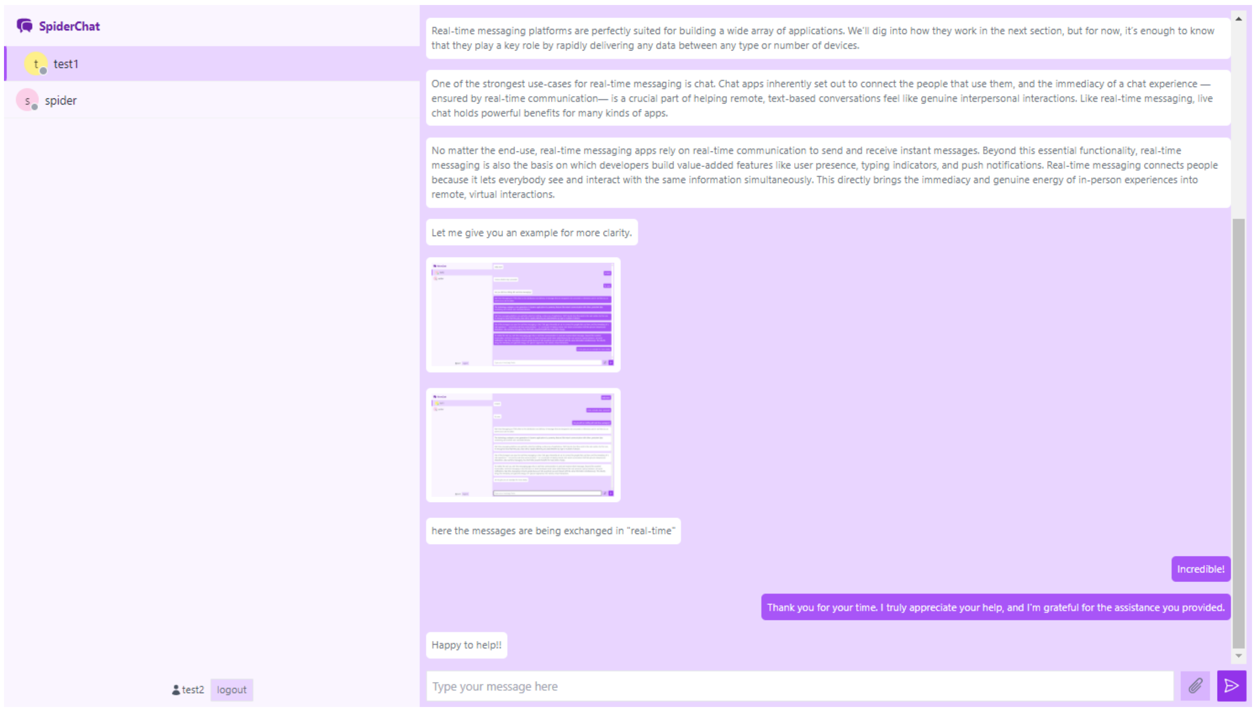
This technology underpins a new generation of dynamic applications by powering features like instant communication with others, persistent data monitoring, and control over connected devices.

Real-time messaging platforms are perfectly suited for building a wide array of applications. We'll dig into how they work in the next section, but for now, it's enough to know that they play a key role by rapidly delivering any data between any type or number of devices.

One of the strongest use-cases for real-time messaging is chat. Chat apps inherently set out to connect the people that use them, and the immediacy of a chat experience — ensured by real-time communication— is a crucial part of helping remote, text-based conversations feel like genuine interpersonal interactions. Like real-time messaging, live chat holds powerful benefits for many kinds of apps.

No matter the end-use, real-time messaging apps rely on real-time communication to send and receive instant messages. Beyond this essential functionality, real-time messaging is also the basis on which developers build value-added features like user presence, typing indicators, and push notifications. Real-time messaging connects people because it lets everybody see and interact with the same information simultaneously. This directly brings the immediacy and genuine energy of in-person experiences into remote, virtual interactions.

Let me give you an example for more clarity.



Real-time messaging platforms are perfectly suited for building a wide array of applications. We'll dig into how they work in the next section, but for now, it's enough to know that they play a key role by rapidly delivering any data between any type or number of devices.

One of the strongest use-cases for real-time messaging is chat. Chat apps inherently set out to connect the people that use them, and the immediacy of a chat experience — ensured by real-time communication— is a crucial part of helping remote, text-based conversations feel like genuine interpersonal interactions. Like real-time messaging, live chat holds powerful benefits for many kinds of apps.

No matter the end-use, real-time messaging apps rely on real-time communication to send and receive instant messages. Beyond this essential functionality, real-time messaging is also the basis on which developers build value-added features like user presence, typing indicators, and push notifications. Real-time messaging connects people because it lets everybody see and interact with the same information simultaneously. This directly brings the immediacy and genuine energy of in-person experiences into remote, virtual interactions.

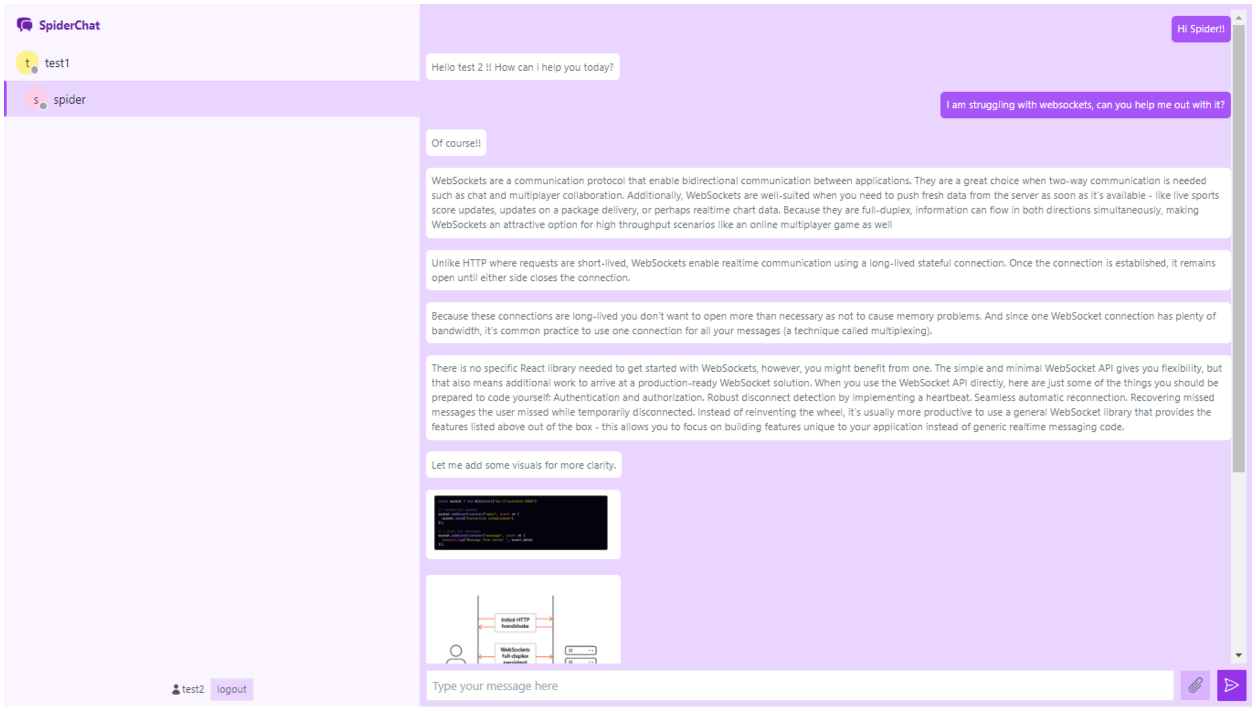
Let me give you an example for more clarity.

here the messages are being exchanged in "real-time"

Incredible!

Thank you for your time. I truly appreciate your help, and I'm grateful for the assistance you provided.

Happy to help!!



SpiderChat

test1

spider

Hi Spider!!

Hello test 2 !! How can i help you today?

I am struggling with websockets, can you help me out with it?

Of course!!

WebSockets are a communication protocol that enable bidirectional communication between applications. They are a great choice when two-way communication is needed such as chat and multiplayer collaboration. Additionally, WebSockets are well-suited when you need to push fresh data from the server as soon as it's available - like live sports score updates, updates on a package delivery, or perhaps realtime chart data. Because they are full-duplex, information can flow in both directions simultaneously, making WebSockets an attractive option for high throughput scenarios like an online multiplayer game as well

Unlike HTTP where requests are short-lived, WebSockets enable realtime communication using a long-lived stateful connection. Once the connection is established, it remains open until either side closes the connection.

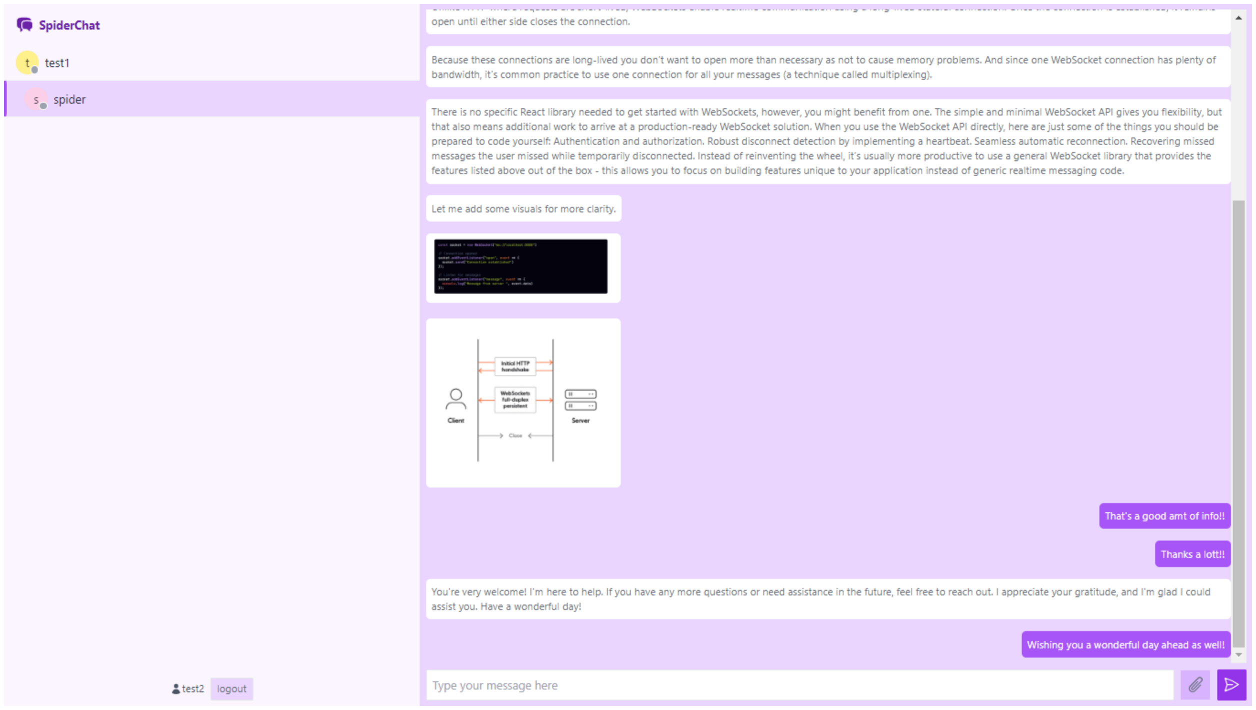
Because these connections are long-lived you don't want to open more than necessary as not to cause memory problems. And since one WebSocket connection has plenty of bandwidth, it's common practice to use one connection for all your messages (a technique called multiplexing).

There is no specific React library needed to get started with WebSockets, however, you might benefit from one. The simple and minimal WebSocket API gives you flexibility, but that also means additional work to arrive at a production-ready WebSocket solution. When you use the WebSocket API directly, here are just some of the things you should be prepared to code yourself: Authentication and authorization. Robust disconnect detection by implementing a heartbeat. Seamless automatic reconnection. Recovering missed messages the user missed while temporarily disconnected. Instead of reinventing the wheel, it's usually more productive to use a general WebSocket library that provides the features listed above out of the box - this allows you to focus on building features unique to your application instead of generic realtime messaging code.

Let me add some visuals for more clarity.

test2 logout

Type your message here



SpiderChat

test1

spider

open until either side closes the connection.

Because these connections are long-lived you don't want to open more than necessary as not to cause memory problems. And since one WebSocket connection has plenty of bandwidth, it's common practice to use one connection for all your messages (a technique called multiplexing).

There is no specific React library needed to get started with WebSockets, however, you might benefit from one. The simple and minimal WebSocket API gives you flexibility, but that also means additional work to arrive at a production-ready WebSocket solution. When you use the WebSocket API directly, here are just some of the things you should be prepared to code yourself: Authentication and authorization. Robust disconnect detection by implementing a heartbeat. Seamless automatic reconnection. Recovering missed messages the user missed while temporarily disconnected. Instead of reinventing the wheel, it's usually more productive to use a general WebSocket library that provides the features listed above out of the box - this allows you to focus on building features unique to your application instead of generic realtime messaging code.

Let me add some visuals for more clarity.

That's a good amt of info!!

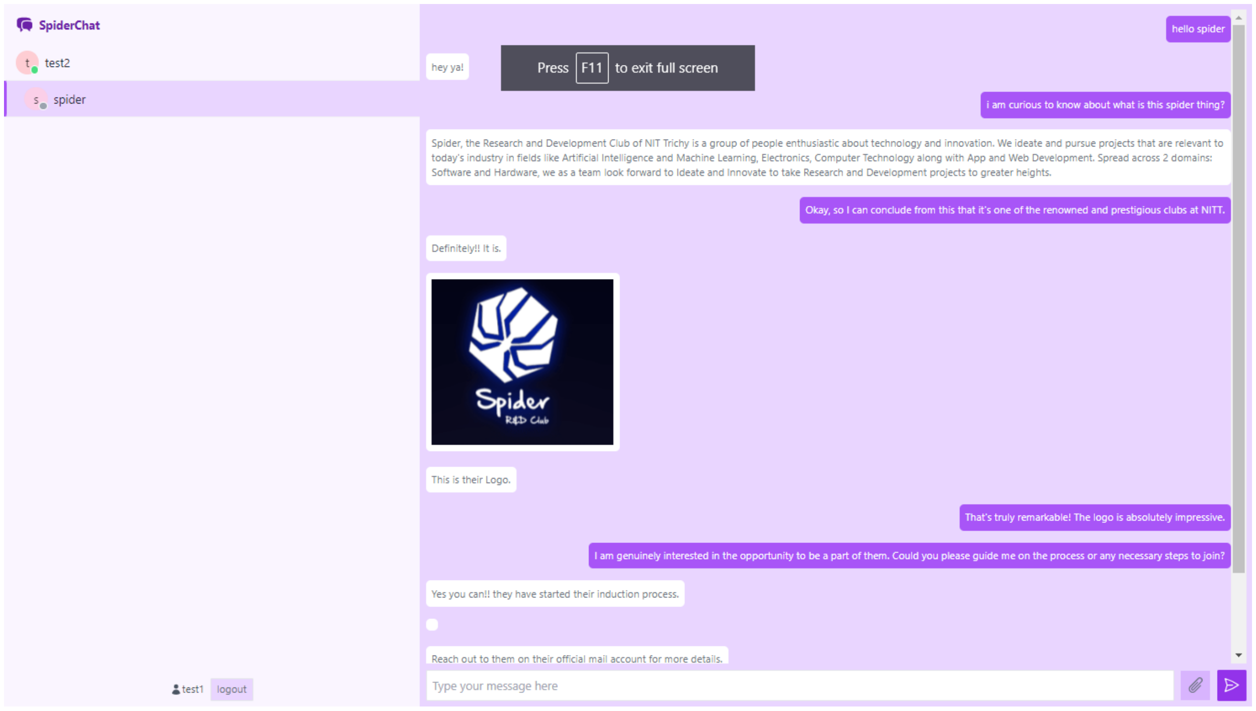
Thanks a lott!!

You're very welcome! I'm here to help. If you have any more questions or need assistance in the future, feel free to reach out. I appreciate your gratitude, and I'm glad I could assist you. Have a wonderful day!

Wishing you a wonderful day ahead as well!

test2 logout

Type your message here



SpiderChat

test2

spider

hello spider

hey ya!

Press F11 to exit full screen

i am curious to know about what is this spider thing?

Spider, the Research and Development Club of NIT Trichy is a group of people enthusiastic about technology and innovation. We ideate and pursue projects that are relevant to today's industry in fields like Artificial Intelligence and Machine Learning, Electronics, Computer Technology along with App and Web Development. Spread across 2 domains: Software and Hardware, we as a team look forward to Ideate and Innovate to take Research and Development projects to greater heights.

Okay, so I can conclude from this that it's one of the renowned and prestigious clubs at NITT.

Definitely!! It is.

Spider R&D Club

This is their Logo.

That's truly remarkable! The logo is absolutely impressive.

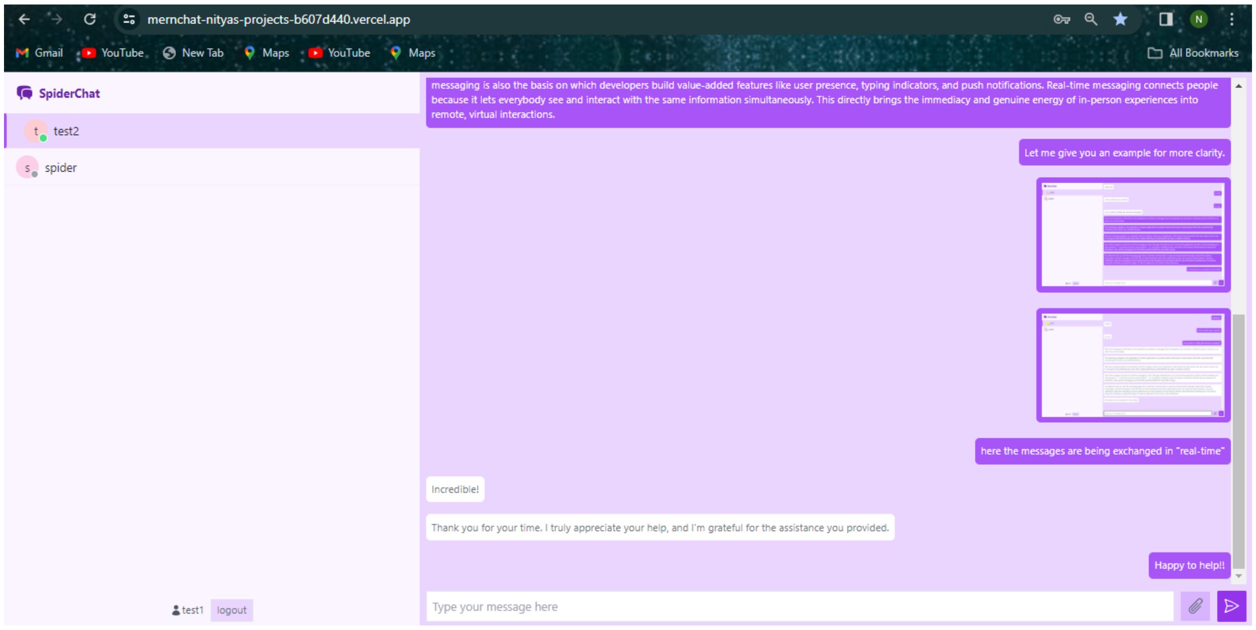
I am genuinely interested in the opportunity to be a part of them. Could you please guide me on the process or any necessary steps to join?

Yes you can!! they have started their induction process.

Reach out to them on their official mail account for more details.

test1 logout

Type your message here



mernchat-nityas-projects-b607d440.vercel.app

Gmail YouTube New Tab Maps YouTube Maps All Bookmarks

SpiderChat

test2

spider

messaging is also the basis on which developers build value-added features like user presence, typing indicators, and push notifications. Real-time messaging connects people because it lets everybody see and interact with the same information simultaneously. This directly brings the immediacy and genuine energy of in-person experiences into remote, virtual interactions.

Let me give you an example for more clarity.

here the messages are being exchanged in "real-time"

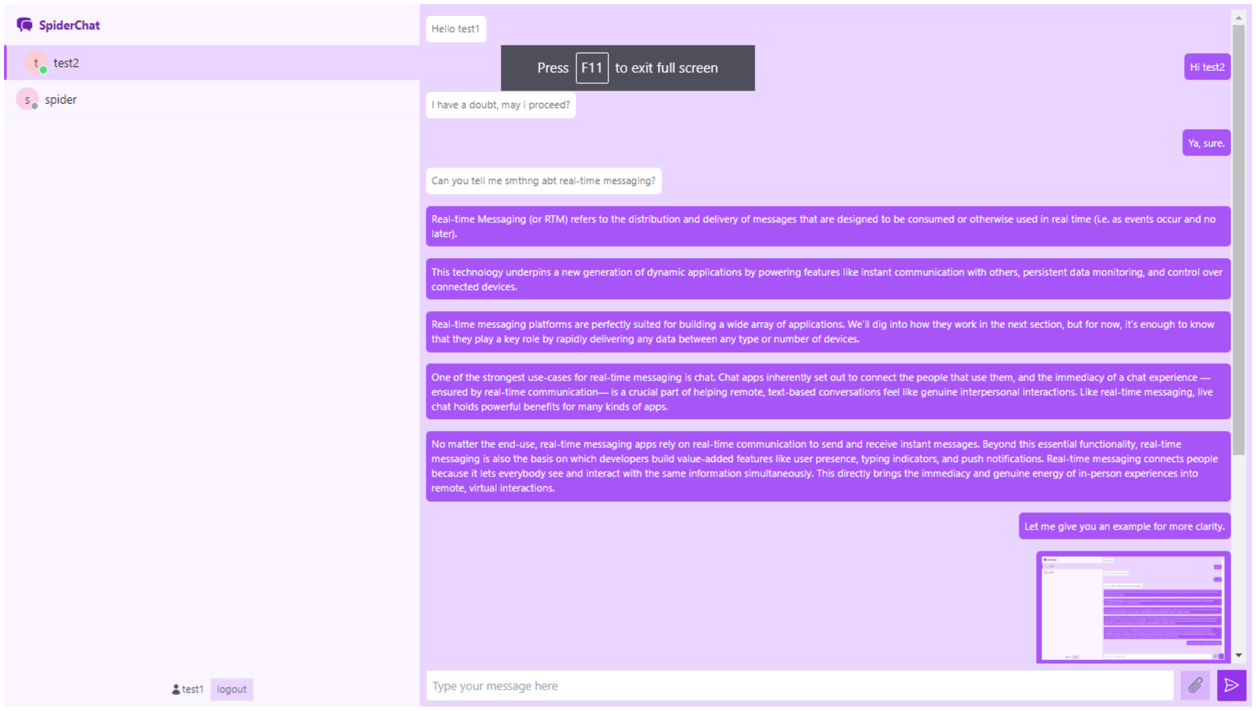
Incredible!

Thank you for your time. I truly appreciate your help, and I'm grateful for the assistance you provided.

Happy to help!!

test1 logout

Type your message here



Hello test1

Hi test2

I have a doubt, may i proceed?

Ya, sure.

Can you tell me smthng abt real-time messaging?

Real-time Messaging (or RTM) refers to the distribution and delivery of messages that are designed to be consumed or otherwise used in real time (i.e. as events occur and no later).

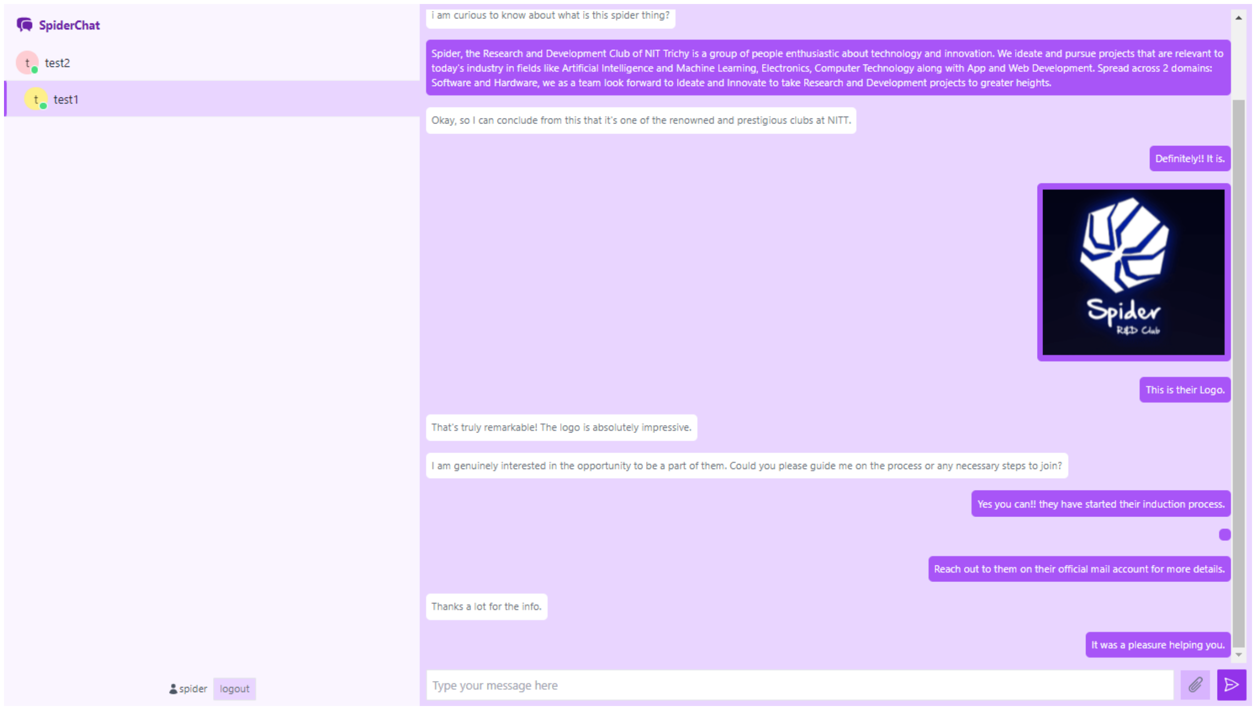
This technology underpins a new generation of dynamic applications by powering features like instant communication with others, persistent data monitoring, and control over connected devices.

Real-time messaging platforms are perfectly suited for building a wide array of applications. We'll dig into how they work in the next section, but for now, it's enough to know that they play a key role by rapidly delivering any data between any type or number of devices.

One of the strongest use-cases for real-time messaging is chat. Chat apps inherently set out to connect the people that use them, and the immediacy of a chat experience — ensured by real-time communication— is a crucial part of helping remote, text-based conversations feel like genuine interpersonal interactions. Like real-time messaging, live chat holds powerful benefits for many kinds of apps.

No matter the end-use, real-time messaging apps rely on real-time communication to send and receive instant messages. Beyond this essential functionality, real-time messaging is also the basis on which developers build value-added features like user presence, typing indicators, and push notifications. Real-time messaging connects people because it lets everybody see and interact with the same information simultaneously. This directly brings the immediacy and genuine energy of in-person experiences into remote, virtual interactions.

Let me give you an example for more clarity.



i am curious to know about what is this spider thing?

Spider, the Research and Development Club of NIT Trichy is a group of people enthusiastic about technology and innovation. We ideate and pursue projects that are relevant to today's industry in fields like Artificial Intelligence and Machine Learning, Electronics, Computer Technology along with App and Web Development. Spread across 2 domains: Software and Hardware, we as a team look forward to Ideate and Innovate to take Research and Development projects to greater heights.

Okay, so I can conclude from this that it's one of the renowned and prestigious clubs at NITT.

Definitely!! It is.

This is their Logo.

That's truly remarkable! The logo is absolutely impressive.

I am genuinely interested in the opportunity to be a part of them. Could you please guide me on the process or any necessary steps to join?

Yes you can!! they have started their induction process.

Reach out to them on their official mail account for more details.

Thanks a lot for the info.

It was a pleasure helping you.