**Chat APP using MERN**

**A Project**

Submitted in partial fulfillment of the requirements for the degree of

Master of Computer Applications

**Guided By**

**Name of the Guide (OPTIMISER)**



**SRM INSTITUTE OF SCIENCE & TECHNOLOGY NCR CAMPUS MODINAGAR**

**(Deemed to be University U/S – 3 of UGC Act. 1956)(F. 9-9/98-U.3)**

**Year-2022**

**Submitted To: - Dr Rajeev Kumar Sharma Submitted By:-Keshav sharma**

**MCA-Dept Reg. No.**

**MCA IVth Sem.**

**INTRODUCTION**

I have created a SNAPPY chat app which help user of a campus or in an office to chat each other. It also has emoji features and user can choose an avatar for its display picture.

User can simply create his account on the site and create its username and password with avatar of its choice. All the data is directly saved in mongodB database with the help of

Mongoose and Express Js.

A chat application makes it easy to communicate with people anywhere in the world by sending and receiving messages in real time. With a [chat app](https://www.pubnub.com/use-case/in-app-chat/), users are able to receive the same engaging and lively interactions through custom messaging features, just as they would in person. This also keeps users conversing on your platform instead of looking elsewhere for a messaging solution. Whether it’s a private chat, group chat, or [large scale chat](https://www.pubnub.com/customers/livelike/), adding personalized chat features to your app can help ensure that your users have a memorable experience.

A chat app that’s composed of real-time messaging features enables users to have an authentic and interactive experience. Features like message reactions, stickers, emojis, GIFs, and [voice calls and video chat](https://www.pubnub.com/integrations/vonage-video-api-video-chat/), provide a way to engage your users directly on your app instead of external platforms–creating a more connected experience. Other functionalities like identifying active users, typing indicators, and message history—to name a few—also add to that immediacy by automatically detecting the presence of users in a chat application.

When building a chat app, another key factor to consider is concurrency. Whether a private chat, group chat, or large-scale chat experience, being able to build without worrying about user fluctuations and concurrency limits on your platform is crucial. And as your user base grows, choosing a solution that scales with you will ultimately be a benefit as your developers can focus on delivering engaging in-app chat experiences to your users.

**Acknowledgement**

It is my great pleasure to express my profound sense of gratitude to my Supervisor Dr. Rajeev Kumar Sharma, for his constructive academic advice and guidance, constant encouragement and valuable suggestions, and all other support throughout this project work and preparing this project report successfully. I have really benefited from his excellent supervision.

I would like to thank all of our friends and those who helped, inspired and gave us mental support at different stages in different moments in our project.

I would also like to thank my mentor Dr. Rajeev Kumar Sharma for mentoring and guiding us throughout the project.

**Software & Hardware Requirements**

Software requirements-

* Window 8+
* VS CODE
* Mongo DB
* React Js
* Node js

Hardware requirement-

* RAM 4+
* ROM 40GB +
* Intel i3+

**Models**

**Server-side model** - In server side I use MongoDB, Express, Node js, Mongoose, Socket.io.

* MongoDB is a [source-available](https://en.wikipedia.org/wiki/Source-available) [cross-platform](https://en.wikipedia.org/wiki/Cross-platform) [document-oriented database](https://en.wikipedia.org/wiki/Document-oriented_database) program. Classified as a [NoSQL](https://en.wikipedia.org/wiki/NoSQL) database program, MongoDB uses [JSON](https://en.wikipedia.org/wiki/JSON)-like documents with optional [schemas](https://en.wikipedia.org/wiki/Database_schema). MongoDB is developed by [MongoDB Inc.](https://en.wikipedia.org/wiki/MongoDB_Inc.) and licensed under the [Server Side Public License](https://en.wikipedia.org/wiki/Server_Side_Public_License) (SSPL).
* Express.js, or simply Express, is a [back end](https://en.wikipedia.org/wiki/Front_end_and_back_end) [web application framework](https://en.wikipedia.org/wiki/Web_application_framework) for [Node.js](https://en.wikipedia.org/wiki/Node.js), released as [free and open-source software](https://en.wikipedia.org/wiki/Free_and_open-source_software) under the [MIT License](https://en.wikipedia.org/wiki/MIT_License). It is designed for building [web applications](https://en.wikipedia.org/wiki/Web_application) and [APIs](https://en.wikipedia.org/wiki/API).[[3]](https://en.wikipedia.org/wiki/Express.js#cite_note-ExpressJS-3) It has been called the [de facto standard](https://en.wikipedia.org/wiki/De_facto_standard) server framework for [Node.js](https://en.wikipedia.org/wiki/Node.js).
* Node.js is an [open-source](https://en.wikipedia.org/wiki/Open-source_software), [cross-platform](https://en.wikipedia.org/wiki/Cross-platform), [back-end](https://en.wikipedia.org/wiki/Front_end_and_back_end) [JavaScript](https://en.wikipedia.org/wiki/JavaScript) [runtime environment](https://en.wikipedia.org/wiki/Runtime_system) that runs on the [V8 engine](https://en.wikipedia.org/wiki/V8_(JavaScript_engine)) and executes JavaScript code outside a [web browser](https://en.wikipedia.org/wiki/Web_browser). Node.js lets developers use JavaScript to write command line tools and for [server-side scripting](https://en.wikipedia.org/wiki/Server-side_scripting)—running scripts server-side to produce [dynamic web page](https://en.wikipedia.org/wiki/Dynamic_web_page) content before the page is sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhere" paradigm,[[6]](https://en.wikipedia.org/wiki/Node.js#cite_note-6) unifying [web-application](https://en.wikipedia.org/wiki/Web_application) development around a single programming language, rather than different languages for server-side and client-side scripts.
* Mongoose is a Node.js-based Object Data Modeling (ODM) library for MongoDB. It is akin to an Object Relational Mapper (ORM) such as SQLAlchemy for traditional SQL databases. The problem that Mongoose aims to solve is allowing developers to enforce a specific schema at the application layer. In addition to enforcing a schema, Mongoose also offers a variety of hooks, model validation, and other features aimed at making it easier to work with MongoDB.
* Socket.io: A JavaScript library which handles WebSocket connections. It abstracts most of the complexity behind WebSocket’s, and it also provides fallback methods which work without any special configuration.

**Client-Side Module-** In Client side I used HTML, CSS, REACT JS BOOTSTRAP.

* Bootstrap is a [free and open-source](https://en.wikipedia.org/wiki/Free_and_open-source) [CSS framework](https://en.wikipedia.org/wiki/CSS_framework) directed at responsive, [mobile-first](https://en.wikipedia.org/wiki/Responsive_web_design#Mobile_first,_unobtrusive_JavaScript,_and_progressive_enhancement) [front-end web development](https://en.wikipedia.org/wiki/Front-end_web_development). It contains [HTML](https://en.wikipedia.org/wiki/HTML), [CSS](https://en.wikipedia.org/wiki/CSS) and (optionally) [JavaScript](https://en.wikipedia.org/wiki/JavaScript)-based design templates for [typography](https://en.wikipedia.org/wiki/Web_design#Typography), [forms](https://en.wikipedia.org/wiki/Form_(HTML)), [buttons](https://en.wikipedia.org/wiki/Button_(computing)#HTML), [navigation](https://en.wikipedia.org/wiki/Web_navigation#Local_website_navigation), and other interface components.
* [HTML](https://www.w3.org/html/) (the Hypertext Markup Language) and [CSS](https://www.w3.org/Style/CSS/) (Cascading Style Sheets) are two of the core technologies for building Web pages. HTML provides the structure of the page, CSS the (visual and aural) layout, for a variety of devices. Along with [graphics](https://www.w3.org/standards/webdesign/graphics) and [scripting](https://www.w3.org/standards/webdesign/script), HTML and CSS are the basis of building Web pages and Web Applications.
* React (also known as React.js or ReactJS) is a [free and open-source](https://en.wikipedia.org/wiki/Free_and_open-source_software) [front-end](https://en.wikipedia.org/wiki/Front_end_and_back_end) [JavaScript library](https://en.wikipedia.org/wiki/JavaScript_library)[[3]](https://en.wikipedia.org/wiki/React_(JavaScript_library)#cite_note-react-3) for building [user interfaces](https://en.wikipedia.org/wiki/User_interfaces) based on UI components. It is maintained by [Meta](https://en.wikipedia.org/wiki/Meta_Platforms) (formerly Facebook) and a community of individual developers and companies.[[4]](https://en.wikipedia.org/wiki/React_(JavaScript_library)#cite_note-4)[[5]](https://en.wikipedia.org/wiki/React_(JavaScript_library)#cite_note-5)[[6]](https://en.wikipedia.org/wiki/React_(JavaScript_library)#cite_note-6) React can be used as a base in the development of [single-page](https://en.wikipedia.org/wiki/Single-page_application), mobile, or server-rendered applications with frameworks like [Next.js](https://en.wikipedia.org/wiki/Next.js). However, React is only concerned with state management and rendering that state to the [DOM](https://en.wikipedia.org/wiki/Document_Object_Model), so creating React applications usually requires the use of additional libraries for routing, as well as certain client-side functionality

**Technology Use**

What is the desired outcome of the project?

Is the proposed product/service available on the current market? If so, how will my product improve on the concept and if not, how realistic and/or risky is it to introduce it to this market?

What unfulfilled need will this project satisfy?

What resources are required for completing this project, and are the necessary resources available?

What are the legal or regulatory concerns associated with this product or service?

MERN is one of several variations of the MEAN stack (MongoDB Express Angular Node), where the traditional Angular.js frontend framework is replaced with React.js. Other variants include MEVN (MongoDB, Express, Vue, Node), and really any frontend JavaScript framework can work.

Ready to take the next step? Setup your free Atlas account by clicking below and try our MERN Stack Tutorial to create a full-stack MERN application in no time.

Get started free

What is the MERN Stack?

MERN stands for MongoDB, Express, React, Node, after the four key technologies that make up the stack.

MongoDB - document database

Express(.js) - Node.js web framework

React(.js) - a client-side JavaScript framework

Node(.js) - the premier JavaScript web server

Express and Node make up the middle (application) tier. Express.js is a server-side web framework, and Node.js the popular and powerful JavaScript server platform. Regardless of which variant you choose, ME(RVA)N is the ideal approach to working with JavaScript and JSON, all the way through.

How does the MERN stack work?

The MERN architecture allows you to easily construct a 3-tier architecture (frontend, backend, database) entirely using JavaScript and JSON.

Mern Stack

React.js Front End

The top tier of the MERN stack is React.js, the declarative JavaScript framework for creating dynamic client-side applications in HTML. React lets you build complex interfaces through simple Components, connect them to data on your backend server, and render them as HTML.

React’s strong suit is handling stateful, data-driven interfaces with minimal code and minimal pain, and it has all the bells and whistles you’d expect from a modern web framework: great support for forms, error handling, events, lists, and more.

Express.js and Node.js Server Tier

The next level down is the Express.js server-side framework, running inside a Node.js server. Express.js bills itself as a “fast, unopinionated, minimalist web framework for Node.js,” and that is indeed exactly what it is. Express.js has powerful models for URL routing (matching an incoming URL with a server function), and handling HTTP requests and responses.

By making XML HTTP Requests (XHRs) or GETs or POSTs from your React.js front-end, you can connect to Express.js functions that power your application. Those functions in turn use MongoDB’s Node.js drivers, either via callbacks for using Promises, to access and update data in your MongoDB database.

MongoDB Database Tier

If your application stores any data (user profiles, content, comments, uploads, events, etc.), then you’re going to want a database that’s just as easy to work with as React, Express, and Node.

That’s where MongoDB comes in: JSON documents created in your React.js front end can be sent to the Express.js server, where they can be processed and (assuming they’re valid) stored directly in MongoDB for later retrieval. Again, if you’re building in the cloud, you’ll want to look at Atlas. If you’re looking to set up your own MERN stack, read on!

Is MERN a full-stack solution?

Yes, MERN is a full-stack, following the traditional 3-tier architectural pattern, including the front-end display tier (React.js), application tier (Express.js and Node.js), and database tier (MongoDB).

Why choose the MERN stack?

Let’s start with MongoDB, the document database at the root of the MERN stack. MongoDB was designed to store JSON data natively (it technically uses a binary version of JSON called BSON), and everything from its command line interface to its query language (MQL, or MongoDB Query Language) is built on JSON and JavaScript.

MongoDB works extremely well with Node.js, and makes storing, manipulating, and representing JSON data at every tier of your application incredibly easy. For cloud-native applications, MongoDB Atlas makes it even easier, by giving you an auto-scaling MongoDB cluster on the cloud provider of your choice, as easy as a few button clicks.

Express.js (running on Node.js) and React.js make the JavaScript/JSON application MERN full stack, well, full. Express.js is a server-side application framework that wraps HTTP requests and responses, and makes it easy to map URLs to server-side functions. React.js is a frontend JavaScript framework for building interactive user interfaces in HTML, and communicating with a remote server.

The combination means that JSON data flows naturally from front to back, making it fast to build on and reasonably simple to debug. Plus, you only have to know one programming language, and the JSON document structure, to understand the whole system!

MERN is the stack of choice for today’s web developers looking to move quickly, particularly for those with React.js experience.

MERN Use Cases

Like any web stack, you can build whatever you want in MERN - though it’s ideally suited for cases that are JSON-heavy, cloud-native, and that have dynamic web interfaces.

A few examples might be: - Workflow management - News aggregation - Todo apps and Calendars - Interactive forums / social products

**Databases and Models**

A key defining aspect of any database-dependent application is its database structure. The database design can vary depending on many different factors, such as the number of reads over writes or the values that the user is likely to request the most. That is because as full stack developers we want the database to have the best performance, which can often be achieved by focusing the optimizations on the most common actions. We concentrated on the MongoDB database, which is the most complex data storage and the one which stores the most data. Our Redis data structure limits to mapping sessions to user identifiers, both of type text. That is how a web request works: Node.js queries Redis by using the user session identifier to determine whether the user is signed and their account identifier. If an account identifier is found, Node.js queries MongoDB to find out the rest of the user information. The MongoDB database stores everything else: users’ information, rooms, chats, and messages. 32 Implementation of a chat application for developers Our final database design ended up having four different collections: users, rooms, chats, and messages. Although MongoDB is schema-less, by using the Mongoose library on Node.js, we were also able to define a flexible schema for each of the collections. A schema constrains the contents of a collection to a known format, saving us from validating the structure of the data before or after it has been put in into the database. 4.1.1 Users To start, we needed somewhere to store our users. Since we were expecting a significant number of entries, an individual collection for the users’ themselves was the most appropriate. What we mean by that is that it was best for the users’ collection to solely store the information that made reference to their authentication and personal data. Their rooms, chats, and messages should be stored somewhere else. Given that we were expecting a lot of rooms, chats, and messages per user, we refrained from even making references to them in this collection. We are querying these other collections directly.

Schema fields:

• \_id: identifier.

• username: friendly identifier.

• email: email address.

• password: encrypted password.

**Chats**

As we stated earlier, our chats were going to be in individual collections. There might be rooms in which their members have few chats, but others might have hundreds (even if that leads to having a few inactive ones). Once again, we had to think whether it was worth embedding or referring messages inside the Chats collection or keeping them isolated in another one. In this case, it was evident. We were expecting thousands of messages in any Chat, which would rapidly go over the 16MB that any MongoDB document can hold, 36 Implementation of a chat application for developers even if only storing references. Thus, messages had to be saved in a different collection.

We are indexing Chats by \_id. \_id is used every time someone wants to enter a specific chat.

**Messages**

We were expecting thousands of messages per month, so the right way to store them, according to the MongoDB official documentation, is in an individual collection. In a production environment on which we were expecting even millions of them, we might have to consider sharding the data to avoid bottlenecks, which is a topic which we have briefly covered in the Evaluation chapter. This case is similar to the Rooms or Chats ones, but this time it is taken to the extreme, "One-to-Squillions"[11]. We were no longer just expecting to store thousands in the long-term run, but we were expecting to store thousands at a fast growing pace. We can summarize our Messages needs as follow: 1. Ability to store hundreds of messages per hour. 2. Ability to retrieve thousands of messages per hour. 4.1. DATABASES AND MODELS 37 3. Ability to retrieve messages in chronological order (most recent first). Notice that we are expecting to read more than to save. That is because a few chat peers are likely to retrieve recent messages more than once, and while a message is only stored once, several members are likely to read it numerous times. Thus, we wanted to design a collection schema which favored reads over writes. Moreover, we would never want to retrieve all messages at once. Not only it would be impossible for the user to read them all, but also we would not be able to handle the load if we did that for Chats having many messages.

Schema fields: • \_id: identifier.

• chat: identifier of the chat it belongs to.

• owner: sender (User) identifier.

• content: text.

• type: content type. – language: code language, if the message type is code. As simple as it seems, this structure has been proved to work out for up to 1,000,000 concurrent connections[12]. We have indexed messages by \_id and chat + createdAt. The first one helps when looking for a specific message, whereas the second composed index works out well when looking for past messages. We have composed the date with the chat to filter only the messages which belong to a particular chat since we will never be interested in mixed chat messages.

**Feasibility Study**

Once, you have created the requirement list you want in your application, check for its feasibility with the availability of resources and goals. The feasibility of the application should be checked on 5 criteria, based on which the cost of the application is decided-

1. Technical Feasibility

Check whether you have all the resources, both hardware, and software, that will be required to develop your application. If not, what are the missing ones? Along with this, also check the technical skills and members your app development team possesses.

You can hire our experts to get the best custom application development services including the competent members like-

Requirement analyst

UI/UX designers

Project manager

Web and Mobile app developers

QA engineers

Digital Marketing Experts

2. Operational Feasibility

Check whether the features that you have enlisted are operationally feasible i.e. easy to operate and maintain. There is no point in making an application that is complex to use, requiring the user to learn a special skill to operate it.

Messaging apps are used by all types of users- technical and non-technical. Hence, make sure it has a shorter learning curve, as it will decide the acceptance of your app among the users. So, make sure to have as simple an app design as possible, requiring you to have good designing skills.

3. Schedule Feasibility

Check whether your application will be able to match your deadline or not. For this, you need to evaluate the time and efforts required for each of the phases of the development process.

Effort estimate in days for developing a messaging App like WhatsApp

Requirement Analysis & Management – 8 Days

Architecture and Design – 15 Days

Modules Development – 132 Days

Unit Testing – 8 Days

Functional Testing – 18 Days

Performance Testing – 6 Days

Compatibility Testing – 10 Days

Based on the above efforts estimated breakdown, the total effort required to develop an App similar to WhatsApp is approximately 197 Days i.e 1500 Hrs. The actual implementation timeline will be much lesser as multiple team members will be working on the project.

Following is the high-level list of modules considered in Modules Development:

Registration and Phone book

Messaging

Calling (Video/ Audio)

Navigation and location

Multimedia file exchange

Contact Sharing

Settings

4. Economic Feasibility

Economic Feasibility is the study that gives you an idea about whether your application development and deployment fits your budget. Analyze each aspect of the development process from a cost perspective to have an idea about how much would it cost to make and to maintain (operating costs) an instant messaging app like WhatsApp?

Factors affecting Mobile App Development Cost

* Hardware and software required
* Hiring a development team
* Deployment and marketing
* Maintenance scope
* App store charges

The cost of hiring an individual developer for each task might vary based on their hourly charges and location. The best approach here could be hiring an entire team who delivers quality that too within your budget.

Let’s breakdown the cost based on 2 major stages of the Application Development and Marketing

Mobile App Development Stage: The development stage cost differs based on the development team structure as the hourly rate for UI/UX experts, Front-end and back-end developers, Quality Assurance person will be different.

The average hourly rate of Mobile App development in the USA ranges to 100 USD/hr. If you require approx. 1500 Hrs to build the modules, the estimated cost to build the entire App will be $150,000.

However, if you are able to offshore part of the development, there is a good scope of reduction of cost to the tune of 30 to 40%. There will be a fixed and ongoing hosting cost which will be lesser in the initial months and will vary as the user base of the app increases.

Marketing Stage: Along with the development stage, you will also require digital marketing experts to promote and market your app to make a profit out of your application. Marketing experts will help you increase App ratings and reviews. According to Business of Apps, the average cost of app marketing, including the pre-launch and post-launch processes ranges to 10K USD/per application.

Initially, the cost might look higher, but if you consider the revenue statistics of WhatsApp as released by Forbes, which is expected to generate 5 to 10 billion USD in the coming years, it will seem nothing.

5. Legal Feasibility

Do not forget to check your application for legal points. As the applications will be publicly available and also will be handling the personal details of the users, make sure it abides by the rules and regulations of the government, such as data protection acts, social media laws, copyright, license, etc.

Once you are done with the analysis, you can do the preparations and head with the further processes including development, testing, deployment, as well as marketing.

Now that you have a close idea of the entire development process as well as the cost associated with it, the decision-making might have become easier for you. The key to successful app development and launch lies in the accuracy of the analysis and implementation. The more accurate will be the analysis and the development process, the higher will be the success rate of your application.

Formulate an efficient strategy that can help you achieve your desired goal. If you need any assistance with the process, reach out to our experts. Having a thorough knowledge about the market trends as well as the technologies, our experts will not only develop an outstanding application for your business but also help you with the marketing and monetization strategies for your application.

**Project Design**

So I started thinking about this idea. I came up with some of the basic requirements and features.

Users register by giving a handle, which is unique to every user (a dummy name). Only the handle will be revealed to other users. So people are free to choose any handle and hence they stay anonymous.

A member can see other members who are online. They have an option to go public, which broadcast the message to all online members in the chat.

For private messages, the sender should first send a request to the other member. Other members upon accepting the request can have private chat with them.

Technology, Tools used

MERN Stack(Mongo, Express, REACT, Node).

Sockets to enable one-on-one communication in real time

AJAX for sign-up and login

So how do you create a simple chat application?

In this tutorial, I’m going to help you create your own chat application. You can later integrate as a widget into any project! This tutorial won’t concentrate on the complete development of a chat application. But it will help you build one.

Pre-requisite : You need to know some basic knowledge of MEAN Stack, as we are making use of it to build one.

First, create a directory structure something like this.

Directory structure of the project

Install Node.js and MongoDB.

We’ll be making use of AngularJS 1 for this tutorial. Download the AngularJS library from here and copy it to the lib folder in Client directory.

If you like to beautify the application you can download any CSS libraries and copy them to lib as well.

Building the Server

Step 1 — Start the project :

Go to Server directory and run this command:

This will start a new project. Provide all the details required. The package.json will be created and will look something like this.

Step 2 — Install the dependencies.

socket.io — is a javascript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers.

express — is a Node.js web application framework. It provides the set of features to develop the web and mobile applications. One can respond to HTTP request using different middlewares and also render HTML pages.

npm install --save socket.io

npm install --save express

This will install required dependencies and add those to package.json. An extra field will be added to package.json which will look like this:

Step 3 — Creating the Server

Create a server which serves at port 3000 and will send the html when called.

Initialize a new socket connection by passing HTTP object.

Event connection will be listening for incoming sockets.

Each socket emits disconnect event which will be called whenever a client disconnects.

socket.on waits for the event. Whenever that event is triggered the callback function is called.

io.emit is used to emit the message to all sockets connected to it.

The syntax is:

socket.on('event', function(msg){})

io.emit('event', 'message')

Create a server with name server.js. It should:

print a message to the console upon a user connecting

listen for chat message events and broadcast the received message to all sockets connected.

Whenever a user disconnects, it should print the message to the console.

The server will look something like this:

Building the Client

Create the index.html in the client directory, style.css in the css directory and app.js in js directory in the client.

index.html:

Let us write a simple HTML which can take our message and also display it.

Include socket.io-client and angular.js in your HTML script.

socket.io serves the client for us. It defaults to connect to the host that serves the page.

Give it some styling so that it looks like a chat box. You can make use of any libraries.

Create an angular.js app and initialize a socket connection.

socket.on listens for a particular event. It calls a callback function whenever that event is called.

socket.emit is used to emit the message to the particular event.

Basic syntax of both are:

So whenever the message is typed and the button is clicked, call the function to send the message.

Whenever the socket receives a message, display it.

The JavaScript will look something like this:

Running the application

Go to server directory where our server is present. Run the server using the following command:

node server.js

The server starts running on port 3000. Go to the browser and type the following url:

http://localhost:3000

How to improve the same application

You can design a database to save user details and messages. It would be good if the design was scalable so that you could add more features later.

You need to install Mongoose or a MongoDB module to make use of a Mongo database:

Here’s the documentation to use mongoose and the mongodb module.

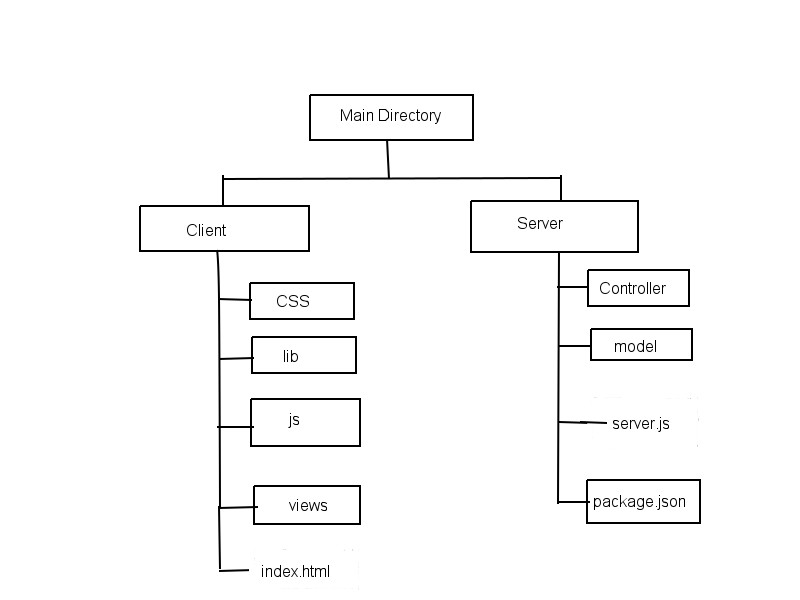
Suppose the member has rejected a chat request. The sender can then send a chat request again. A user can also save the messages by creating an extra collection. Each document will have the message, sender, receiver, and time.

So design your database according to your specific needs and how you want to handle messages.

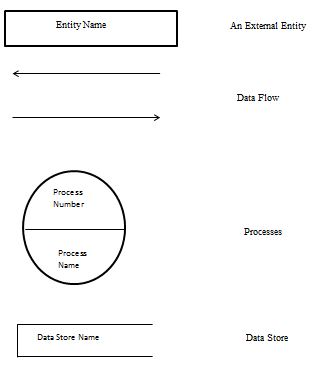
2. Create REST APIs to serve the client. For example, an endpoint that sends a home page, from which users can make other requests.

3. Think of some cool additional features and implement them.

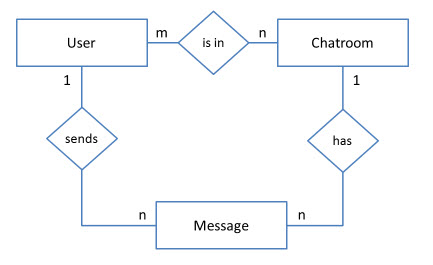
Directory structure-



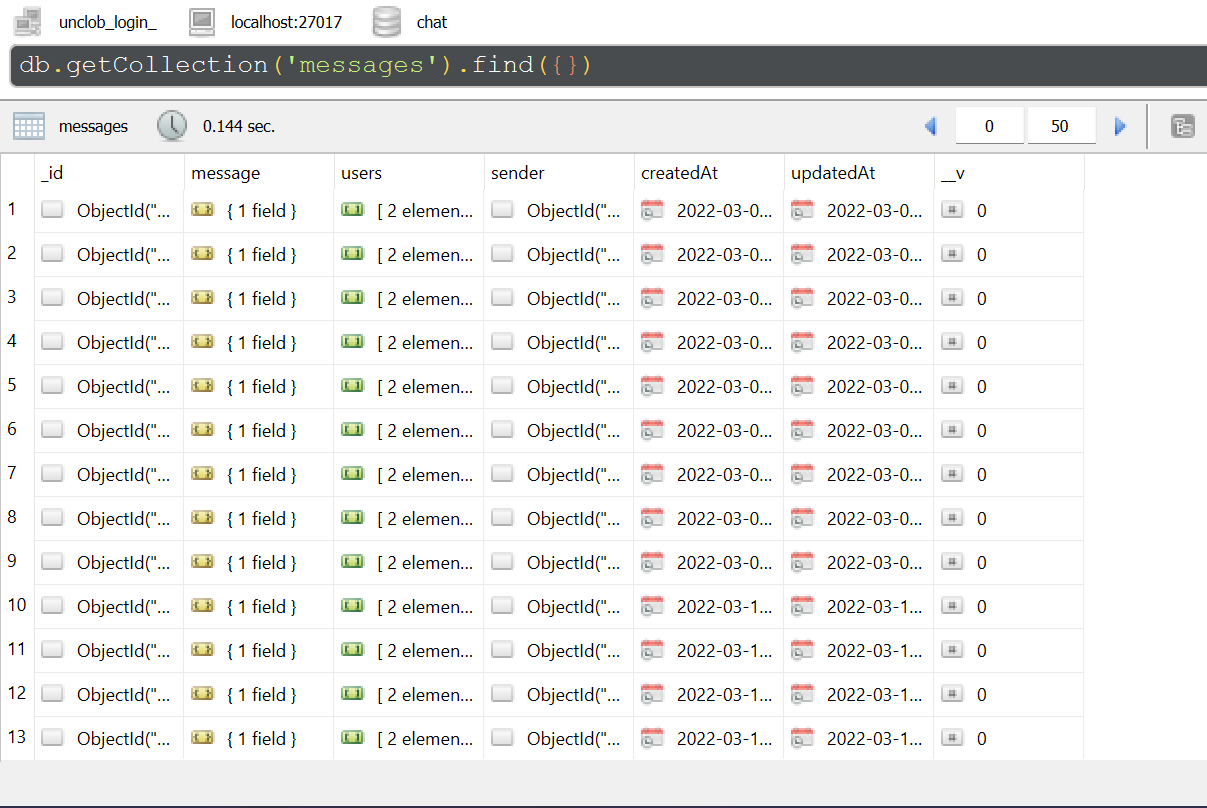
**Data Flow Diagram-**

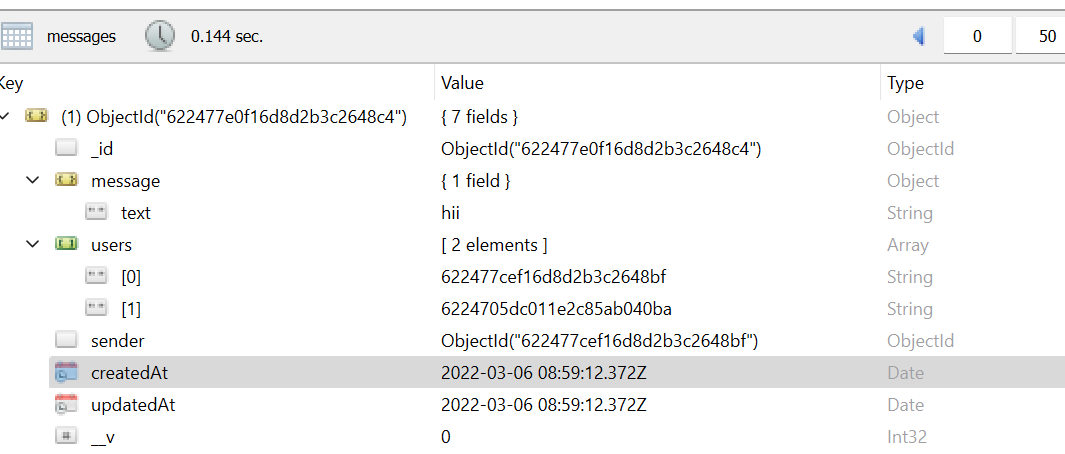
****

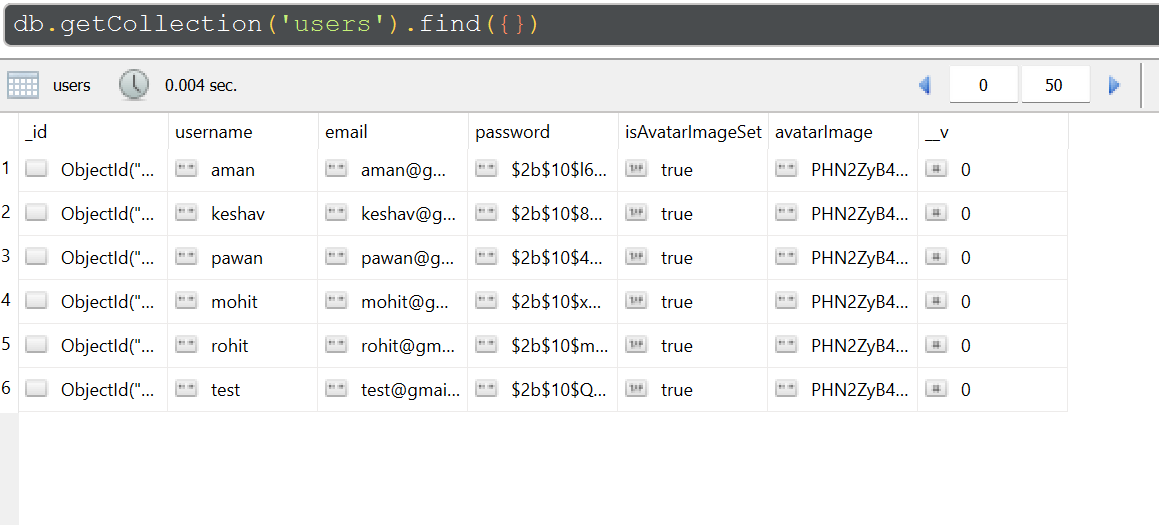
**Data Modeling**

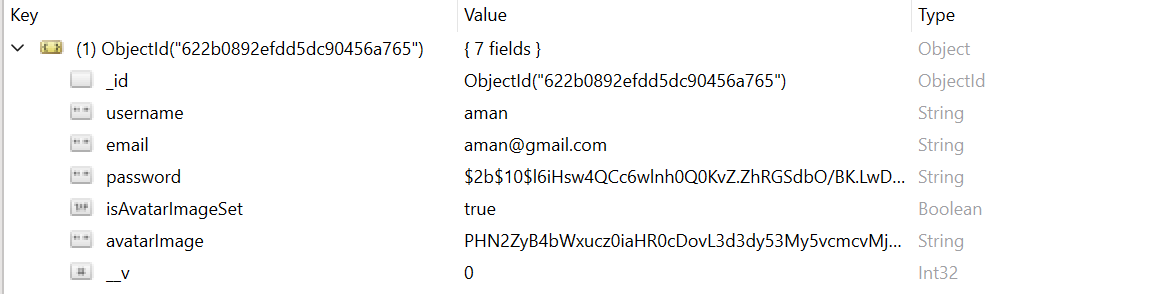
****

**Table Structures**

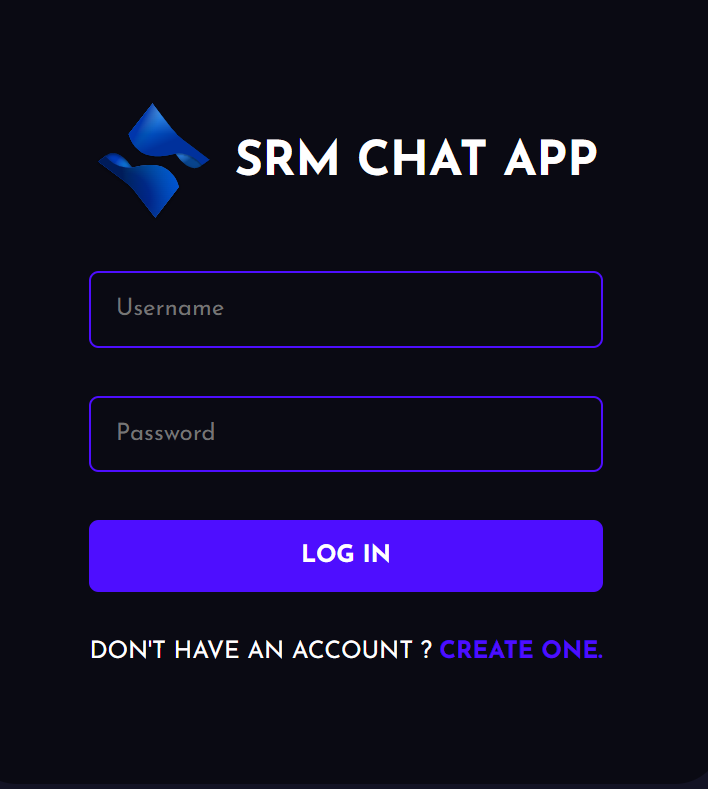
****

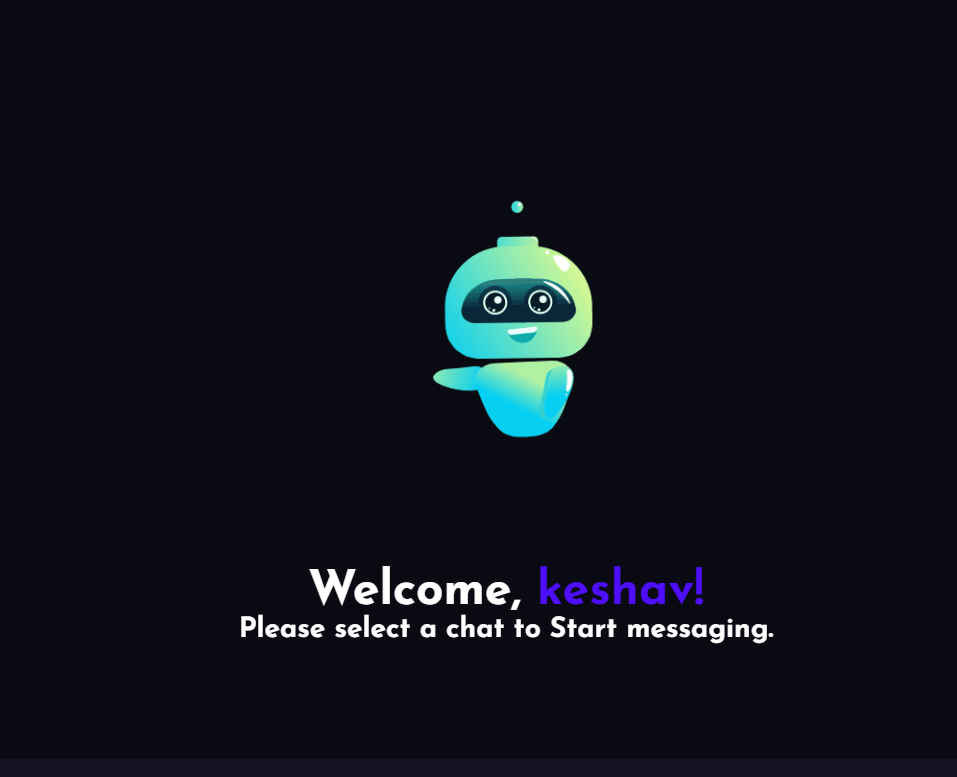
****

****

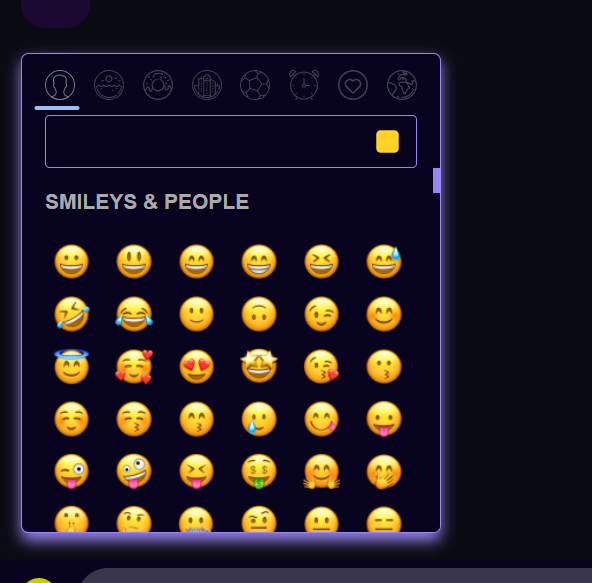
****

**FrontEnd- Screenshot**

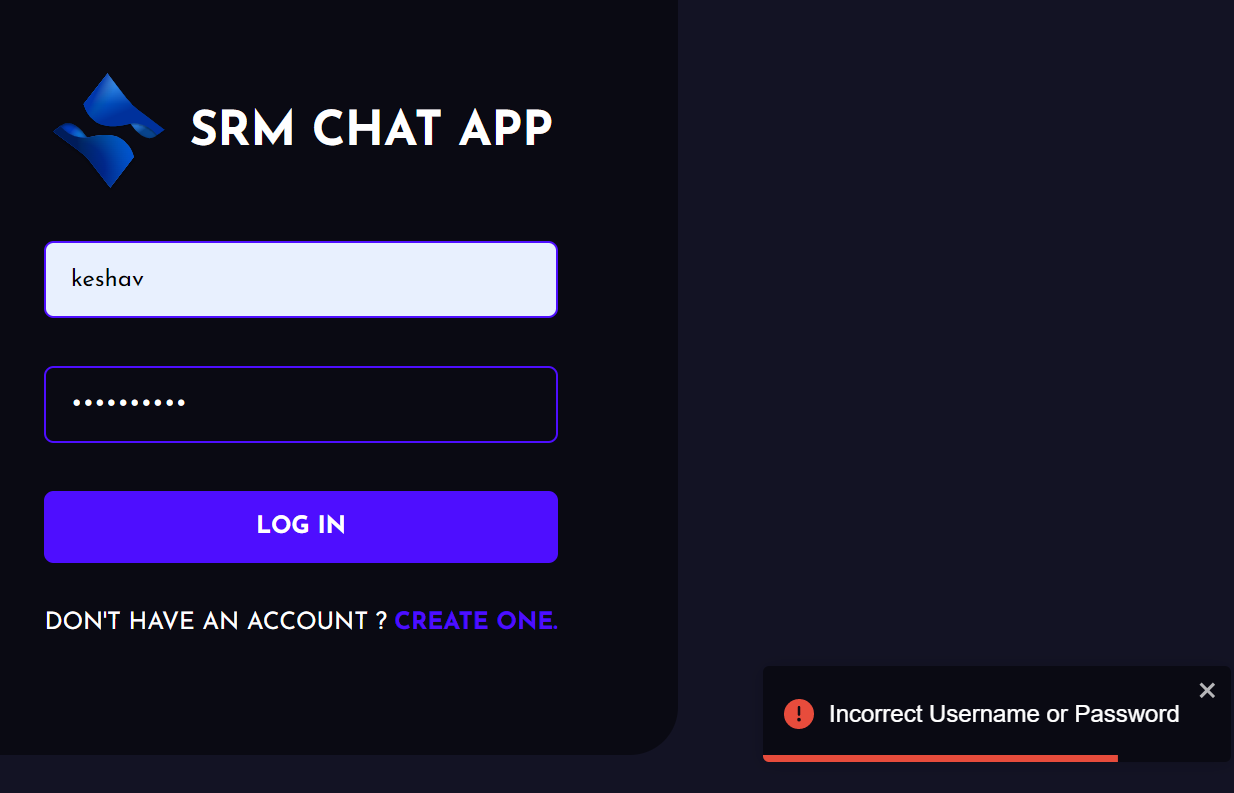


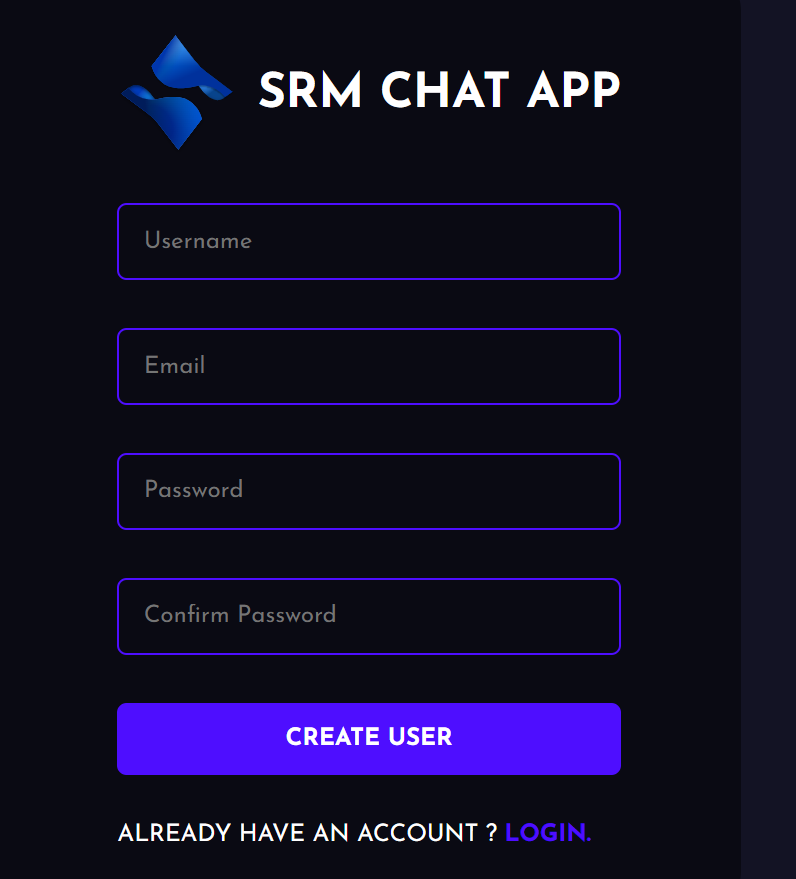






Authentication-





**Coding**

**Client side**

Client/src/components/

ChatContainer.jsx

import React, { useState, useEffect, useRef } from "react";

import styled from "styled-components";

import ChatInput from "./ChatInput";

import Logout from "./Logout";

import { v4 as uuidv4 } from "uuid";

import axios from "axios";

import { sendMessageRoute, recieveMessageRoute } from "../utils/APIRoutes";

export default function ChatContainer({ currentChat, socket }) {

  const [messages, setMessages] = useState([]);

  const scrollRef = useRef();

  const [arrivalMessage, setArrivalMessage] = useState(null);

  useEffect(async () => {

    const data = await JSON.parse(

      localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

    );

    const response = await axios.post(recieveMessageRoute, {

      from: data.\_id,

      to: currentChat.\_id,

    });

    setMessages(response.data);

  }, [currentChat]);

  useEffect(() => {

    const getCurrentChat = async () => {

      if (currentChat) {

        await JSON.parse(

          localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

        ).\_id;

      }

    };

    getCurrentChat();

  }, [currentChat]);

  const handleSendMsg = async (msg) => {

    const data = await JSON.parse(

      localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

    );

    socket.current.emit("send-msg", {

      to: currentChat.\_id,

      from: data.\_id,

      msg,

    });

    await axios.post(sendMessageRoute, {

      from: data.\_id,

      to: currentChat.\_id,

      message: msg,

    });

    const msgs = [...messages];

    msgs.push({ fromSelf: true, message: msg });

    setMessages(msgs);

  };

  useEffect(() => {

    if (socket.current) {

      socket.current.on("msg-recieve", (msg) => {

        setArrivalMessage({ fromSelf: false, message: msg });

      });

    }

  }, []);

  useEffect(() => {

    arrivalMessage && setMessages((prev) => [...prev, arrivalMessage]);

  }, [arrivalMessage]);

  useEffect(() => {

    scrollRef.current?.scrollIntoView({ behavior: "smooth" });

  }, [messages]);

  return (

    <Container>

      <div className="chat-header">

        <div className="user-details">

          <div className="avatar">

            <img

              src={`data:image/svg+xml;base64,${currentChat.avatarImage}`}

              alt=""

            />

          </div>

          <div className="username">

            <h3>{currentChat.username}</h3>

          </div>

        </div>

        <Logout />

      </div>

      <div className="chat-messages">

        {messages.map((message) => {

          return (

            <div ref={scrollRef} key={uuidv4()}>

              <div

                className={`message ${

                  message.fromSelf ? "sended" : "recieved"

                }`}

              >

                <div className="content ">

                  <p>{message.message}</p>

                </div>

              </div>

            </div>

          );

        })}

      </div>

      <ChatInput handleSendMsg={handleSendMsg} />

    </Container>

  );

}

const Container = styled.div`

  display: grid;

  grid-template-rows: 10% 80% 10%;

  gap: 0.1rem;

  overflow: hidden;

  @media screen and (min-width: 720px) and (max-width: 1080px) {

    grid-template-rows: 15% 70% 15%;

  }

  .chat-header {

    display: flex;

    justify-content: space-between;

    align-items: center;

    padding: 0 2rem;

    .user-details {

      display: flex;

      align-items: center;

      gap: 1rem;

      .avatar {

        img {

          height: 3rem;

        }

      }

      .username {

        h3 {

          color: white;

        }

      }

    }

  }

  .chat-messages {

    padding: 1rem 2rem;

    display: flex;

    flex-direction: column;

    gap: 1rem;

    overflow: auto;

    &::-webkit-scrollbar {

      width: 0.2rem;

      &-thumb {

        background-color: #ffffff39;

        width: 0.1rem;

        border-radius: 1rem;

      }

    }

    .message {

      display: flex;

      align-items: center;

      .content {

        max-width: 40%;

        overflow-wrap: break-word;

        padding: 1rem;

        font-size: 1.1rem;

        border-radius: 1rem;

        color: #d1d1d1;

        @media screen and (min-width: 720px) and (max-width: 1080px) {

          max-width: 70%;

        }

      }

    }

    .sended {

      justify-content: flex-end;

      .content {

        background-color: #4f04ff21;

      }

    }

    .recieved {

      justify-content: flex-start;

      .content {

        background-color: #9900ff20;

      }

    }

  }

`;

ChatInput.jsx

import React, { useState } from "react";

import { BsEmojiSmileFill } from "react-icons/bs";

import { IoMdSend } from "react-icons/io";

import styled from "styled-components";

import Picker from "emoji-picker-react";

export default function ChatInput({ handleSendMsg }) {

  const [msg, setMsg] = useState("");

  const [showEmojiPicker, setShowEmojiPicker] = useState(false);

  const handleEmojiPickerhideShow = () => {

    setShowEmojiPicker(!showEmojiPicker);

  };

  const handleEmojiClick = (event, emojiObject) => {

    let message = msg;

    message += emojiObject.emoji;

    setMsg(message);

  };

  const sendChat = (event) => {

    event.preventDefault();

    if (msg.length > 0) {

      handleSendMsg(msg);

      setMsg("");

    }

  };

  return (

    <Container>

      <div className="button-container">

        <div className="emoji">

          <BsEmojiSmileFill onClick={handleEmojiPickerhideShow} />

          {showEmojiPicker && <Picker onEmojiClick={handleEmojiClick} />}

        </div>

      </div>

      <form className="input-container" onSubmit={(event) => sendChat(event)}>

        <input

          type="text"

          placeholder="type your message here"

          onChange={(e) => setMsg(e.target.value)}

          value={msg}

        />

        <button type="submit">

          <IoMdSend />

        </button>

      </form>

    </Container>

  );

}

const Container = styled.div`

  display: grid;

  align-items: center;

  grid-template-columns: 5% 95%;

  background-color: #080420;

  padding: 0 2rem;

  @media screen and (min-width: 720px) and (max-width: 1080px) {

    padding: 0 1rem;

    gap: 1rem;

  }

  .button-container {

    display: flex;

    align-items: center;

    color: white;

    gap: 1rem;

    .emoji {

      position: relative;

      svg {

        font-size: 1.5rem;

        color: #ffff00c8;

        cursor: pointer;

      }

      .emoji-picker-react {

        position: absolute;

        top: -350px;

        background-color: #080420;

        box-shadow: 0 5px 10px #9a86f3;

        border-color: #9a86f3;

        .emoji-scroll-wrapper::-webkit-scrollbar {

          background-color: #080420;

          width: 5px;

          &-thumb {

            background-color: #9a86f3;

          }

        }

        .emoji-categories {

          button {

            filter: contrast(0);

          }

        }

        .emoji-search {

          background-color: transparent;

          border-color: #9a86f3;

        }

        .emoji-group:before {

          background-color: #080420;

        }

      }

    }

  }

  .input-container {

    width: 100%;

    border-radius: 2rem;

    display: flex;

    align-items: center;

    gap: 2rem;

    background-color: #ffffff34;

    input {

      width: 90%;

      height: 60%;

      background-color: transparent;

      color: white;

      border: none;

      padding-left: 1rem;

      font-size: 1.2rem;

      &::selection {

        background-color: #9a86f3;

      }

      &:focus {

        outline: none;

      }

    }

    button {

      padding: 0.3rem 2rem;

      border-radius: 2rem;

      display: flex;

      justify-content: center;

      align-items: center;

      background-color: #9a86f3;

      border: none;

      @media screen and (min-width: 720px) and (max-width: 1080px) {

        padding: 0.3rem 1rem;

        svg {

          font-size: 1rem;

        }

      }

      svg {

        font-size: 2rem;

        color: white;

      }

    }

  }

`;

Contacts.jsx

import React, { useState, useEffect } from "react";

import styled from "styled-components";

import Logo from "../assets/logo.svg";

export default function Contacts({ contacts, changeChat }) {

  const [currentUserName, setCurrentUserName] = useState(undefined);

  const [currentUserImage, setCurrentUserImage] = useState(undefined);

  const [currentSelected, setCurrentSelected] = useState(undefined);

  useEffect(async () => {

    const data = await JSON.parse(

      localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

    );

    setCurrentUserName(data.username);

    setCurrentUserImage(data.avatarImage);

  }, []);

  const changeCurrentChat = (index, contact) => {

    setCurrentSelected(index);

    changeChat(contact);

  };

  return (

    <>

      {currentUserImage && currentUserImage && (

        <Container>

          <div className="brand">

            <img src={Logo} alt="logo" />

            <h3>snappy</h3>

          </div>

          <div className="contacts">

            {contacts.map((contact, index) => {

              return (

                <div

                  key={contact.\_id}

                  className={`contact ${

                    index === currentSelected ? "selected" : ""

                  }`}

                  onClick={() => changeCurrentChat(index, contact)}

                >

                  <div className="avatar">

                    <img

                      src={`data:image/svg+xml;base64,${contact.avatarImage}`}

                      alt=""

                    />

                  </div>

                  <div className="username">

                    <h3>{contact.username}</h3>

                  </div>

                </div>

              );

            })}

          </div>

          <div className="current-user">

            <div className="avatar">

              <img

                src={`data:image/svg+xml;base64,${currentUserImage}`}

                alt="avatar"

              />

            </div>

            <div className="username">

              <h2>{currentUserName}</h2>

            </div>

          </div>

        </Container>

      )}

    </>

  );

}

const Container = styled.div`

  display: grid;

  grid-template-rows: 10% 75% 15%;

  overflow: hidden;

  background-color: #080420;

  .brand {

    display: flex;

    align-items: center;

    gap: 1rem;

    justify-content: center;

    img {

      height: 2rem;

    }

    h3 {

      color: white;

      text-transform: uppercase;

    }

  }

  .contacts {

    display: flex;

    flex-direction: column;

    align-items: center;

    overflow: auto;

    gap: 0.8rem;

    &::-webkit-scrollbar {

      width: 0.2rem;

      &-thumb {

        background-color: #ffffff39;

        width: 0.1rem;

        border-radius: 1rem;

      }

    }

    .contact {

      background-color: #ffffff34;

      min-height: 5rem;

      cursor: pointer;

      width: 90%;

      border-radius: 0.2rem;

      padding: 0.4rem;

      display: flex;

      gap: 1rem;

      align-items: center;

      transition: 0.5s ease-in-out;

      .avatar {

        img {

          height: 3rem;

        }

      }

      .username {

        h3 {

          color: white;

        }

      }

    }

    .selected {

      background-color: #9a86f3;

    }

  }

  .current-user {

    background-color: #0d0d30;

    display: flex;

    justify-content: center;

    align-items: center;

    gap: 2rem;

    .avatar {

      img {

        height: 4rem;

        max-inline-size: 100%;

      }

    }

    .username {

      h2 {

        color: white;

      }

    }

    @media screen and (min-width: 720px) and (max-width: 1080px) {

      gap: 0.5rem;

      .username {

        h2 {

          font-size: 1rem;

        }

      }

    }

  }

`;

LogOut.jsx

import React from "react";

import { useNavigate } from "react-router-dom";

import { BiPowerOff } from "react-icons/bi";

import styled from "styled-components";

import axios from "axios";

import { logoutRoute } from "../utils/APIRoutes";

export default function Logout() {

  const navigate = useNavigate();

  const handleClick = async () => {

    const id = await JSON.parse(

      localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

    ).\_id;

    const data = await axios.get(`${logoutRoute}/${id}`);

    if (data.status === 200) {

      localStorage.clear();

      navigate("/login");

    }

  };

  return (

    <Button onClick={handleClick}>

      <BiPowerOff />

    </Button>

  );

}

const Button = styled.button`

  display: flex;

  justify-content: center;

  align-items: center;

  padding: 0.5rem;

  border-radius: 0.5rem;

  background-color: #9a86f3;

  border: none;

  cursor: pointer;

  svg {

    font-size: 1.3rem;

    color: #ebe7ff;

  }

`;

SetAvatar.jsx

import React, { useEffect, useState } from "react";

import styled from "styled-components";

import axios from "axios";

import { Buffer } from "buffer";

import loader from "../assets/loader.gif";

import { ToastContainer, toast } from "react-toastify";

import "react-toastify/dist/ReactToastify.css";

import { useNavigate } from "react-router-dom";

import { setAvatarRoute } from "../utils/APIRoutes";

export default function SetAvatar() {

  const api = `https://api.multiavatar.com/4645646`;

  const navigate = useNavigate();

  const [avatars, setAvatars] = useState([]);

  const [isLoading, setIsLoading] = useState(true);

  const [selectedAvatar, setSelectedAvatar] = useState(undefined);

  const toastOptions = {

    position: "bottom-right",

    autoClose: 8000,

    pauseOnHover: true,

    draggable: true,

    theme: "dark",

  };

  useEffect(async () => {

    if (!localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY))

      navigate("/login");

  }, []);

  const setProfilePicture = async () => {

    if (selectedAvatar === undefined) {

      toast.error("Please select an avatar", toastOptions);

    } else {

      const user = await JSON.parse(

        localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

      );

      const { data } = await axios.post(`${setAvatarRoute}/${user.\_id}`, {

        image: avatars[selectedAvatar],

      });

      if (data.isSet) {

        user.isAvatarImageSet = true;

        user.avatarImage = data.image;

        localStorage.setItem(

          process.env.REACT\_APP\_LOCALHOST\_KEY,

          JSON.stringify(user)

        );

        navigate("/");

      } else {

        toast.error("Error setting avatar. Please try again.", toastOptions);

      }

    }

  };

  useEffect(async () => {

    const data = [];

    for (let i = 0; i < 4; i++) {

      const image = await axios.get(

        `${api}/${Math.round(Math.random() \* 1000)}`

      );

      const buffer = new Buffer(image.data);

      data.push(buffer.toString("base64"));

    }

    setAvatars(data);

    setIsLoading(false);

  }, []);

  return (

    <>

      {isLoading ? (

        <Container>

          <img src={loader} alt="loader" className="loader" />

        </Container>

      ) : (

        <Container>

          <div className="title-container">

            <h1>Pick an Avatar as your profile picture</h1>

          </div>

          <div className="avatars">

            {avatars.map((avatar, index) => {

              return (

                <div

                  className={`avatar ${

                    selectedAvatar === index ? "selected" : ""

                  }`}

                >

                  <img

                    src={`data:image/svg+xml;base64,${avatar}`}

                    alt="avatar"

                    key={avatar}

                    onClick={() => setSelectedAvatar(index)}

                  />

                </div>

              );

            })}

          </div>

          <button onClick={setProfilePicture} className="submit-btn">

            Set as Profile Picture

          </button>

          <ToastContainer />

        </Container>

      )}

    </>

  );

}

const Container = styled.div`

  display: flex;

  justify-content: center;

  align-items: center;

  flex-direction: column;

  gap: 3rem;

  background-color: #131324;

  height: 100vh;

  width: 100vw;

  .loader {

    max-inline-size: 100%;

  }

  .title-container {

    h1 {

      color: white;

    }

  }

  .avatars {

    display: flex;

    gap: 2rem;

    .avatar {

      border: 0.4rem solid transparent;

      padding: 0.4rem;

      border-radius: 5rem;

      display: flex;

      justify-content: center;

      align-items: center;

      transition: 0.5s ease-in-out;

      img {

        height: 6rem;

        transition: 0.5s ease-in-out;

      }

    }

    .selected {

      border: 0.4rem solid #4e0eff;

    }

  }

  .submit-btn {

    background-color: #4e0eff;

    color: white;

    padding: 1rem 2rem;

    border: none;

    font-weight: bold;

    cursor: pointer;

    border-radius: 0.4rem;

    font-size: 1rem;

    text-transform: uppercase;

    &:hover {

      background-color: #4e0eff;

    }

  }

`;

Welcome.jsx

import React, { useState, useEffect } from "react";

import styled from "styled-components";

import Robot from "../assets/robot.gif";

export default function Welcome() {

  const [userName, setUserName] = useState("");

  useEffect(async () => {

    setUserName(

      await JSON.parse(

        localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

      ).username

    );

  }, []);

  return (

    <Container>

      <img src={Robot} alt="" />

      <h1>

        Welcome, <span>{userName}!</span>

      </h1>

      <h3>Please select a chat to Start messaging.</h3>

    </Container>

  );

}

const Container = styled.div`

  display: flex;

  justify-content: center;

  align-items: center;

  color: white;

  flex-direction: column;

  img {

    height: 20rem;

  }

  span {

    color: #4e0eff;

  }

`;

Client/src/components/pages

Chat.jsx

import React, { useEffect, useState, useRef } from "react";

import axios from "axios";

import { useNavigate } from "react-router-dom";

import { io } from "socket.io-client";

import styled from "styled-components";

import { allUsersRoute, host } from "../utils/APIRoutes";

import ChatContainer from "../components/ChatContainer";

import Contacts from "../components/Contacts";

import Welcome from "../components/Welcome";

export default function Chat() {

  const navigate = useNavigate();

  const socket = useRef();

  const [contacts, setContacts] = useState([]);

  const [currentChat, setCurrentChat] = useState(undefined);

  const [currentUser, setCurrentUser] = useState(undefined);

  useEffect(async () => {

    if (!localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)) {

      navigate("/login");

    } else {

      setCurrentUser(

        await JSON.parse(

          localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)

        )

      );

    }

  }, []);

  useEffect(() => {

    if (currentUser) {

      socket.current = io(host);

      socket.current.emit("add-user", currentUser.\_id);

    }

  }, [currentUser]);

  useEffect(async () => {

    if (currentUser) {

      if (currentUser.isAvatarImageSet) {

        const data = await axios.get(`${allUsersRoute}/${currentUser.\_id}`);

        setContacts(data.data);

      } else {

        navigate("/setAvatar");

      }

    }

  }, [currentUser]);

  const handleChatChange = (chat) => {

    setCurrentChat(chat);

  };

  return (

    <>

      <Container>

        <div className="container">

          <Contacts contacts={contacts} changeChat={handleChatChange} />

          {currentChat === undefined ? (

            <Welcome />

          ) : (

            <ChatContainer currentChat={currentChat} socket={socket} />

          )}

        </div>

      </Container>

    </>

  );

}

const Container = styled.div`

  height: 100vh;

  width: 100vw;

  display: flex;

  flex-direction: column;

  justify-content: center;

  gap: 1rem;

  align-items: center;

  background-color: #131324;

  .container {

    height: 85vh;

    width: 85vw;

    background-color: #00000076;

    display: grid;

    grid-template-columns: 25% 75%;

    @media screen and (min-width: 720px) and (max-width: 1080px) {

      grid-template-columns: 35% 65%;

    }

  }

`;

Login.jsx

import React, { useState, useEffect } from "react";

import axios from "axios";

import styled from "styled-components";

import { useNavigate, Link } from "react-router-dom";

import Logo from "../assets/logo.svg";

import { ToastContainer, toast } from "react-toastify";

import "react-toastify/dist/ReactToastify.css";

import { loginRoute } from "../utils/APIRoutes";

export default function Login() {

  const navigate = useNavigate();

  const [values, setValues] = useState({ username: "", password: "" });

  const toastOptions = {

    position: "bottom-right",

    autoClose: 8000,

    pauseOnHover: true,

    draggable: true,

    theme: "dark",

  };

  useEffect(() => {

    if (localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)) {

      navigate("/");

    }

  }, []);

  const handleChange = (event) => {

    setValues({ ...values, [event.target.name]: event.target.value });

  };

  const validateForm = () => {

    const { username, password } = values;

    if (username === "") {

      toast.error("Email and Password is required.", toastOptions);

      return false;

    } else if (password === "") {

      toast.error("Email and Password is required.", toastOptions);

      return false;

    }

    return true;

  };

  const handleSubmit = async (event) => {

    event.preventDefault();

    if (validateForm()) {

      const { username, password } = values;

      const { data } = await axios.post(loginRoute, {

        username,

        password,

      });

      if (data.status === false) {

        toast.error(data.msg, toastOptions);

      }

      if (data.status === true) {

        localStorage.setItem(

          process.env.REACT\_APP\_LOCALHOST\_KEY,

          JSON.stringify(data.user)

        );

        navigate("/");

      }

    }

  };

  return (

    <>

      <FormContainer>

        <form action="" onSubmit={(event) => handleSubmit(event)}>

          <div className="brand">

            <img src={Logo} alt="logo" />

            <h1>SRM Chat app</h1>

          </div>

          <input

            type="text"

            placeholder="Username"

            name="username"

            onChange={(e) => handleChange(e)}

            min="3"

          />

          <input

            type="password"

            placeholder="Password"

            name="password"

            onChange={(e) => handleChange(e)}

          />

          <button type="submit">Log In</button>

          <span>

            Don't have an account ? <Link to="/register">Create One.</Link>

          </span>

        </form>

      </FormContainer>

      <ToastContainer />

    </>

  );

}

const FormContainer = styled.div`

  height: 100vh;

  width: 100vw;

  display: flex;

  flex-direction: column;

  justify-content: center;

  gap: 1rem;

  align-items: center;

  background-color: #131324;

  .brand {

    display: flex;

    align-items: center;

    gap: 1rem;

    justify-content: center;

    img {

      height: 5rem;

    }

    h1 {

      color: white;

      text-transform: uppercase;

    }

  }

  form {

    display: flex;

    flex-direction: column;

    gap: 2rem;

    background-color: #00000076;

    border-radius: 2rem;

    padding: 5rem;

  }

  input {

    background-color: transparent;

    padding: 1rem;

    border: 0.1rem solid #4e0eff;

    border-radius: 0.4rem;

    color: white;

    width: 100%;

    font-size: 1rem;

    &:focus {

      border: 0.1rem solid #997af0;

      outline: none;

    }

  }

  button {

    background-color: #4e0eff;

    color: white;

    padding: 1rem 2rem;

    border: none;

    font-weight: bold;

    cursor: pointer;

    border-radius: 0.4rem;

    font-size: 1rem;

    text-transform: uppercase;

    &:hover {

      background-color: #4e0eff;

    }

  }

  span {

    color: white;

    text-transform: uppercase;

    a {

      color: #4e0eff;

      text-decoration: none;

      font-weight: bold;

    }

  }

`;

Register.jsx

import React, { useState, useEffect } from "react";

import axios from "axios";

import styled from "styled-components";

import { useNavigate, Link } from "react-router-dom";

import Logo from "../assets/logo.svg";

import { ToastContainer, toast } from "react-toastify";

import "react-toastify/dist/ReactToastify.css";

import { registerRoute } from "../utils/APIRoutes";

export default function Register() {

  const navigate = useNavigate();

  const toastOptions = {

    position: "bottom-right",

    autoClose: 8000,

    pauseOnHover: true,

    draggable: true,

    theme: "dark",

  };

  const [values, setValues] = useState({

    username: "",

    email: "",

    password: "",

    confirmPassword: "",

  });

  useEffect(() => {

    if (localStorage.getItem(process.env.REACT\_APP\_LOCALHOST\_KEY)) {

      navigate("/");

    }

  }, []);

  const handleChange = (event) => {

    setValues({ ...values, [event.target.name]: event.target.value });

  };

  const handleValidation = () => {

    const { password, confirmPassword, username, email } = values;

    if (password !== confirmPassword) {

      toast.error(

        "Password and confirm password should be same.",

        toastOptions

      );

      return false;

    } else if (username.length < 3) {

      toast.error(

        "Username should be greater than 3 characters.",

        toastOptions

      );

      return false;

    } else if (password.length < 8) {

      toast.error(

        "Password should be equal or greater than 8 characters.",

        toastOptions

      );

      return false;

    } else if (email === "") {

      toast.error("Email is required.", toastOptions);

      return false;

    }

    return true;

  };

  const handleSubmit = async (event) => {

    event.preventDefault();

    if (handleValidation()) {

      const { email, username, password } = values;

      const { data } = await axios.post(registerRoute, {

        username,

        email,

        password,

      });

      if (data.status === false) {

        toast.error(data.msg, toastOptions);

      }

      if (data.status === true) {

        localStorage.setItem(

          process.env.REACT\_APP\_LOCALHOST\_KEY,

          JSON.stringify(data.user)

        );

        navigate("/");

      }

    }

  };

  return (

    <>

      <FormContainer>

        <form action="" onSubmit={(event) => handleSubmit(event)}>

          <div className="brand">

            <img src={Logo} alt="logo" />

            <h1>SRM Chat App</h1>

          </div>

          <input

            type="text"

            placeholder="Username"

            name="username"

            onChange={(e) => handleChange(e)}

          />

          <input

            type="email"

            placeholder="Email"

            name="email"

            onChange={(e) => handleChange(e)}

          />

          <input

            type="password"

            placeholder="Password"

            name="password"

            onChange={(e) => handleChange(e)}

          />

          <input

            type="password"

            placeholder="Confirm Password"

            name="confirmPassword"

            onChange={(e) => handleChange(e)}

          />

          <button type="submit">Create User</button>

          <span>

            Already have an account ? <Link to="/login">Login.</Link>

          </span>

        </form>

      </FormContainer>

      <ToastContainer />

    </>

  );

}

const FormContainer = styled.div`

  height: 100vh;

  width: 100vw;

  display: flex;

  flex-direction: column;

  justify-content: center;

  gap: 1rem;

  align-items: center;

  background-color: #131324;

  .brand {

    display: flex;

    align-items: center;

    gap: 1rem;

    justify-content: center;

    img {

      height: 5rem;

    }

    h1 {

      color: white;

      text-transform: uppercase;

    }

  }

  form {

    display: flex;

    flex-direction: column;

    gap: 2rem;

    background-color: #00000076;

    border-radius: 2rem;

    padding: 3rem 5rem;

  }

  input {

    background-color: transparent;

    padding: 1rem;

    border: 0.1rem solid #4e0eff;

    border-radius: 0.4rem;

    color: white;

    width: 100%;

    font-size: 1rem;

    &:focus {

      border: 0.1rem solid #997af0;

      outline: none;

    }

  }

  button {

    background-color: #4e0eff;

    color: white;

    padding: 1rem 2rem;

    border: none;

    font-weight: bold;

    cursor: pointer;

    border-radius: 0.4rem;

    font-size: 1rem;

    text-transform: uppercase;

    &:hover {

      background-color: #4e0eff;

    }

  }

  span {

    color: white;

    text-transform: uppercase;

    a {

      color: #4e0eff;

      text-decoration: none;

      font-weight: bold;

    }

  }

`;

Client/src/components/utils

export const host = "http://localhost:5000";

export const loginRoute = `${host}/api/auth/login`;

export const registerRoute = `${host}/api/auth/register`;

export const logoutRoute = `${host}/api/auth/logout`;

export const allUsersRoute = `${host}/api/auth/allusers`;

export const sendMessageRoute = `${host}/api/messages/addmsg`;

export const recieveMessageRoute = `${host}/api/messages/getmsg`;

export const setAvatarRoute = `${host}/api/auth/setavatar`;

App.js

import React from "react";

import { BrowserRouter, Routes, Route } from "react-router-dom";

import SetAvatar from "./components/SetAvatar";

import Chat from "./pages/Chat";

import Login from "./pages/Login";

import Register from "./pages/Register";

export default function App() {

  return (

    <BrowserRouter>

      <Routes>

        <Route path="/register" element={<Register />} />

        <Route path="/login" element={<Login />} />

        <Route path="/setAvatar" element={<SetAvatar />} />

        <Route path="/" element={<Chat />} />

      </Routes>

    </BrowserRouter>

  );

}

**Server- side**

Server/controllers/

messageController.js

const Messages = require("../models/messageModel");

module.exports.getMessages = async (req, res, next) => {

  try {

    const { from, to } = req.body;

    const messages = await Messages.find({

      users: {

        $all: [from, to],

      },

    }).sort({ updatedAt: 1 });

    const projectedMessages = messages.map((msg) => {

      return {

        fromSelf: msg.sender.toString() === from,

        message: msg.message.text,

      };

    });

    res.json(projectedMessages);

  } catch (ex) {

    next(ex);

  }

};

module.exports.addMessage = async (req, res, next) => {

  try {

    const { from, to, message } = req.body;

    const data = await Messages.create({

      message: { text: message },

      users: [from, to],

      sender: from,

    });

    if (data) return res.json({ msg: "Message added successfully." });

    else return res.json({ msg: "Failed to add message to the database" });

  } catch (ex) {

    next(ex);

  }

};

userController.js

const User = require("../models/userModel");

const bcrypt = require("bcrypt");

module.exports.login = async (req, res, next) => {

  try {

    const { username, password } = req.body;

    const user = await User.findOne({ username });

    if (!user)

      return res.json({ msg: "Incorrect Username or Password", status: false });

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid)

      return res.json({ msg: "Incorrect Username or Password", status: false });

    delete user.password;

    return res.json({ status: true, user });

  } catch (ex) {

    next(ex);

  }

};

module.exports.register = async (req, res, next) => {

  try {

    const { username, email, password } = req.body;

    const usernameCheck = await User.findOne({ username });

    if (usernameCheck)

      return res.json({ msg: "Username already used", status: false });

    const emailCheck = await User.findOne({ email });

    if (emailCheck)

      return res.json({ msg: "Email already used", status: false });

    const hashedPassword = await bcrypt.hash(password, 10);

    const user = await User.create({

      email,

      username,

      password: hashedPassword,

    });

    delete user.password;

    return res.json({ status: true, user });

  } catch (ex) {

    next(ex);

  }

};

module.exports.getAllUsers = async (req, res, next) => {

  try {

    const users = await User.find({ \_id: { $ne: req.params.id } }).select([

      "email",

      "username",

      "avatarImage",

      "\_id",

    ]);

    return res.json(users);

  } catch (ex) {

    next(ex);

  }

};

module.exports.setAvatar = async (req, res, next) => {

  try {

    const userId = req.params.id;

    const avatarImage = req.body.image;

    const userData = await User.findByIdAndUpdate(

      userId,

      {

        isAvatarImageSet: true,

        avatarImage,

      },

      { new: true }

    );

    return res.json({

      isSet: userData.isAvatarImageSet,

      image: userData.avatarImage,

    });

  } catch (ex) {

    next(ex);

  }

};

module.exports.logOut = (req, res, next) => {

  try {

    if (!req.params.id) return res.json({ msg: "User id is required " });

    onlineUsers.delete(req.params.id);

    return res.status(200).send();

  } catch (ex) {

    next(ex);

  }

};

messageModels.js

const mongoose = require("mongoose");

const MessageSchema = mongoose.Schema(

  {

    message: {

      text: { type: String, required: true },

    },

    users: Array,

    sender: {

      type: mongoose.Schema.Types.ObjectId,

      ref: "User",

      required: true,

    },

  },

  {

    timestamps: true,

  }

);

module.exports = mongoose.model("Messages", MessageSchema);

userModels.js

const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({

  username: {

    type: String,

    required: true,

    min: 3,

    max: 20,

    unique: true,

  },

  email: {

    type: String,

    required: true,

    unique: true,

    max: 50,

  },

  password: {

    type: String,

    required: true,

    min: 8,

  },

  isAvatarImageSet: {

    type: Boolean,

    default: false,

  },

  avatarImage: {

    type: String,

    default: "",

  },

});

module.exports = mongoose.model("Users", userSchema);

Routes

auth.js

const {

  login,

  register,

  getAllUsers,

  setAvatar,

  logOut,

} = require("../controllers/userController");

const router = require("express").Router();

router.post("/login", login);

router.post("/register", register);

router.get("/allusers/:id", getAllUsers);

router.post("/setavatar/:id", setAvatar);

router.get("/logout/:id", logOut);

module.exports = router;

message.js

const { addMessage, getMessages } = require("../controllers/messageController");

const router = require("express").Router();

router.post("/addmsg/", addMessage);

router.post("/getmsg/", getMessages);

module.exports = router;

index.js

const express = require("express");

const cors = require("cors");

const mongoose = require("mongoose");

const authRoutes = require("./routes/auth");

const messageRoutes = require("./routes/messages");

const app = express();

const socket = require("socket.io");

require("dotenv").config();

app.use(cors());

app.use(express.json());

mongoose

  .connect(process.env.MONGO\_URL, {

    useNewUrlParser: true,

    useUnifiedTopology: true,

  })

  .then(() => {

    console.log("DB Connetion Successfull");

  })

  .catch((err) => {

    console.log(err.message);

  });

app.use("/api/auth", authRoutes);

app.use("/api/messages", messageRoutes);

const server = app.listen(process.env.PORT, () =>

  console.log(`Server started on ${process.env.PORT}`)

);

const io = socket(server, {

  cors: {

    origin: "http://localhost:3000",

    credentials: true,

  },

});

global.onlineUsers = new Map();

io.on("connection", (socket) => {

  global.chatSocket = socket;

  socket.on("add-user", (userId) => {

    onlineUsers.set(userId, socket.id);

  });

  socket.on("send-msg", (data) => {

    const sendUserSocket = onlineUsers.get(data.to);

    if (sendUserSocket) {

      socket.to(sendUserSocket).emit("msg-recieve", data.msg);

    }

  });

});