

Introduction to LifeLinkr

LifeLinkr is an innovative healthcare technology company dedicated to revolutionizing the way health information systems are utilized in the medical field. Our mission is to provide secure, user-friendly, and efficient solutions specifically designed for IVF (In Vitro Fertilization) clinics and related healthcare services. By leveraging cutting-edge technology, LifeLinkr aims to enhance patient care, streamline clinic operations, and facilitate better communication between healthcare providers and patients.

As a backend developer at LifeLinkr, I play a crucial role in building and maintaining the server-side applications that power our healthcare solutions. Utilizing **Node.js**, I develop scalable APIs and integrate various databases to ensure smooth data management and processing. My responsibilities also include optimizing performance, implementing security measures, and collaborating closely with frontend developers to deliver a seamless user experience.

The scope of my work involves:

- Designing and implementing RESTful APIs for various healthcare functionalities.
- Ensuring data integrity and security through robust authentication and authorization protocols.
- Working with databases like Sequelize for effective data handling and retrieval.
- Collaborating with cross-functional teams to translate business requirements into technical specifications.

At LifeLinkr, we are committed to advancing healthcare technology and making a meaningful impact on the lives of patients and healthcare professionals alike.

Role of a Backend Developer at LifeLinkr

1. Overview of Responsibilities

As a backend developer at LifeLinkr, your primary responsibility is to create and maintain the server-side logic and database interactions that power the company's healthcare applications. This role is crucial for ensuring that the applications are efficient, reliable, and secure, thereby providing a seamless experience for both healthcare providers and patients.

2. Key Responsibilities

- **API Development:**
 - Design and implement RESTful APIs that enable communication between the frontend user interface and the backend server.
 - Ensure that APIs are well-documented, versioned, and adhere to industry standards for security and performance.
- **Database Management:**
 - Use **Sequelize** (or other ORMs) to interact with relational databases (e.g., PostgreSQL, MySQL).
 - Create and optimize database schemas to handle large volumes of data efficiently.
 - Write complex queries to retrieve, insert, update, and delete data, ensuring data integrity and consistency.
- **Server-side Logic:**
 - Develop the core business logic of the application, including data validation, processing user input, and implementing business rules.
 - Optimize performance through caching, load balancing, and efficient resource management.
- **Security Implementation:**
 - Implement security best practices, including authentication (e.g., JWT, OAuth) and authorization, to protect sensitive patient data.

- Conduct regular security assessments and stay updated on vulnerabilities and emerging threats.
- **Collaboration:**
 - Work closely with frontend developers, UI/UX designers, and product managers to gather requirements and translate them into technical specifications.
 - Participate in code reviews, contribute to team discussions, and share knowledge with peers to foster a collaborative development environment.
- **Testing and Debugging:**
 - Write unit tests and integration tests to ensure code reliability and functionality.
 - Utilize debugging tools and logging frameworks to identify and resolve issues promptly.

3. Technology Stack

- **Node.js:**
 - A JavaScript runtime that allows you to build scalable network applications. Its non-blocking I/O model is ideal for handling concurrent requests, making it well-suited for real-time applications and APIs.
- **Express.js:**
 - A minimalist web framework for Node.js that simplifies the process of building web applications and APIs. It provides robust routing and middleware capabilities.
- **Sequelize:**
 - An ORM for Node.js that provides a simple way to interact with SQL databases. It helps manage database schemas and queries using JavaScript, reducing the complexity of SQL syntax.
- **Database:**
 - **PostgreSQL/MySQL:** Relational databases used to store structured data. They support complex queries and transactions, ensuring data consistency.
- **Authentication & Authorization:**

- **JSON Web Tokens (JWT):** Used for secure transmission of information between parties as a JSON object.
 - **OAuth:** An open standard for access delegation, commonly used as a way to grant websites or applications limited access to user information.
- **Testing Frameworks:**
 - **Mocha/Chai:** JavaScript testing frameworks used for writing unit tests and integration tests to ensure code quality and functionality.
- **Version Control:**
 - **Git:** A version control system used to track changes in code and collaborate with team members through platforms like GitHub or GitLab.

4. Skills and Qualifications

- **Programming Languages:** Proficiency in JavaScript (Node.js) and familiarity with TypeScript for writing type-safe code.
- **Database Management:** Understanding of SQL and experience with relational database design and optimization.
- **RESTful APIs:** Experience in designing and implementing RESTful services.
- **Security Best Practices:** Knowledge of common security practices, including encryption and data protection regulations (e.g., HIPAA).
- **Problem-Solving Skills:** Ability to troubleshoot and debug complex issues effectively.
- **Communication Skills:** Strong interpersonal skills for collaborating with cross-functional teams.

5. Professional Development

As a backend developer at LifeLinkr, continuous learning is essential due to the ever-evolving nature of technology. Engaging in:

- Online courses and certifications related to Node.js, security practices, or cloud services (e.g., AWS, Azure).

- Attending industry conferences, meetups, or webinars to stay updated on the latest trends and technologies in healthcare and software development.

Introduction

The Significance of Real-Time Communication in the Modern Era

In the fast-evolving digital age, communication has transcended beyond traditional methods such as phone calls and emails. With the advent of social media platforms, messaging applications, and collaborative tools, instant communication has become the cornerstone of personal, business, and professional interactions. Real-time communication allows people to interact without delay, fostering faster decision-making and collaboration, which is essential in today's fast-paced environment. The world has become highly interconnected, and messaging platforms have become the backbone of social and business interactions.

Moksh, a real-time chat application, addresses the growing demand for platforms that enable individuals and teams to communicate seamlessly across geographical boundaries. Whether it's for social interactions or for collaborative work environments, the need for instant messaging platforms is evident in both personal and professional spheres. Moksh aims to be a robust solution by offering real-time messaging capabilities, efficient media sharing, group chat functionalities, and secure user authentication—all within a sleek, responsive interface.

Evolution of Real-Time Chat Applications

Chat applications have evolved significantly over the past two decades. From simple text-based chat rooms in the early days of the internet to sophisticated, feature-rich platforms like **WhatsApp**, **Slack**, and **Microsoft Teams**, the focus has always been on delivering fast, reliable, and secure communication services. Initially, chat platforms were limited to basic one-on-one messaging. However, with technological advancements, these platforms now support features like group messaging, file sharing, video and voice calls, and integration with other tools, enhancing their utility in both social and professional settings.

The key to the success of any modern chat application lies in its ability to scale efficiently while ensuring minimal latency in communication. Modern frameworks and technologies like **WebSockets** enable real-time communication, allowing users to send and receive

messages instantly without page refreshes or delays. **Moksh** leverages this technology to ensure that conversations are smooth, uninterrupted, and scalable to accommodate multiple users simultaneously.

The Need for Secure and Scalable Communication

With the increase in the usage of chat applications for both personal and business communication, concerns about data security and user privacy have become more prevalent. Data breaches, cyberattacks, and unauthorized access to sensitive information have raised serious concerns about the safety of user information on communication platforms. This necessitates the implementation of robust security measures in chat applications.

Moksh addresses these concerns by using **JSON Web Tokens (JWT)** for authentication, ensuring that only authorized users can access the application. JWT provides a compact, self-contained way of securely transmitting information between parties as a JSON object. By using JWT, Moksh ensures that user sessions are secure, and sensitive information such as chat logs and personal details are not exposed to unauthorized entities. Additionally, all data transmitted between users is encrypted, further safeguarding it from potential threats.

Scalability is another major consideration when developing a real-time chat application. As the number of users increases, the platform must be capable of handling large volumes of messages, file transfers, and user interactions without performance degradation. **Moksh** utilizes the **MERN stack** (MongoDB, Express.js, React, and Node.js) to create a scalable architecture that can handle multiple concurrent users. MongoDB, a NoSQL database, allows for flexible data storage and efficient retrieval of chat messages, making the system both scalable and responsive.

Features of Moksh Chat Application

Moksh is designed to offer a wide range of features that cater to both casual users and professionals. Some of the key features include:

1. **Real-Time Messaging:** At the core of Moksh is its real-time messaging feature, which allows users to send and receive messages instantaneously. This is made possible through the integration of **WebSockets** via **Socket.io**, which ensures low-latency message delivery.
2. **Group Chats:** Users can create groups and participate in group discussions. This feature is especially useful for team collaborations, where multiple users need to communicate and share ideas in real time. Group chats in Moksh are dynamically scalable and can handle large numbers of users without affecting the performance of the application.
3. **Media Sharing:** In addition to text messaging, Moksh supports sharing of media files such as images, videos, and documents. The platform handles large file transfers efficiently and ensures that media files are securely stored and retrieved when needed.
4. **User Authentication and Security:** Moksh employs **JWT**-based authentication, which ensures that users are securely logged in and their sessions are protected. The use of JWT prevents unauthorized access and protects sensitive user data. Additionally, encryption protocols are used to ensure that the messages exchanged on the platform are private and cannot be intercepted by external entities.
5. **Real-Time Notifications:** Users receive notifications for new messages, even when they are not actively using the app. This feature ensures that users remain updated with their conversations, enhancing the overall user experience.
6. **Responsive Design:** The **React.js** frontend ensures that Moksh provides a responsive user interface across different devices, including desktops, tablets, and mobile phones. This makes the application versatile and usable in a variety of settings, from casual personal use to professional team communication.
7. **Scalability and Performance:** Moksh has been designed to be highly scalable. The **MERN** stack allows the application to handle a large number of simultaneous users without compromising on performance. **MongoDB**, with its flexible document-based data model, is particularly well-suited for storing and retrieving chat data efficiently.

Benefits of Using the MERN Stack

The **MERN** stack, consisting of **MongoDB**, **Express.js**, **React.js**, and **Node.js**, is a popular choice for developing modern web applications. Each component of the stack plays a crucial role in the development of **Moksh**:

1. **MongoDB**: As a NoSQL database, MongoDB offers flexibility in storing chat data. Its schema-less design allows for dynamic data storage, which is ideal for chat applications that need to store a variety of data types such as text, media files, and timestamps. MongoDB also scales easily, making it perfect for applications like Moksh that expect to handle large volumes of data.
2. **Express.js**: This lightweight framework for Node.js is used to build the backend APIs that serve the frontend. Express.js simplifies server-side development by providing a simple and minimalistic structure, which allows developers to handle HTTP requests, routes, and middleware efficiently.
3. **React.js**: For the frontend, React.js provides a fast and responsive user interface. Its component-based architecture ensures that the user interface is highly dynamic, and the virtual DOM ensures that updates to the UI are handled efficiently, enhancing the user experience. React's ability to build reusable components helps in maintaining a clean code structure.
4. **Node.js**: As the runtime environment, Node.js allows the development of server-side applications that can handle multiple simultaneous connections efficiently. Node.js excels in real-time applications like Moksh because of its event-driven, non-blocking architecture, which enables it to manage real-time data exchanges smoothly.

Market Need for Moksh

There is a growing demand for chat applications that are not only fast and efficient but also secure and scalable. Platforms such as **Slack**, **Microsoft Teams**, and **WhatsApp** have shown the potential of real-time communication applications in both personal and professional settings. However, many of these platforms either charge high subscription fees for premium features or do not prioritize data privacy and security. Moksh fills this gap

by offering a real-time chat application that is secure, scalable, and customizable to fit the needs of both individual users and businesses.

Moksh is positioned to serve a variety of user groups, including:

- **Individuals** seeking secure and private communication.
- **Teams and organizations** looking for a reliable platform to facilitate collaboration.
- **Educational institutions** that need a platform for students and teachers to communicate efficiently.

Software Requirements Specification (SRS) for Moksh: Real-Time Chat Application

1. Introduction

1.1 Purpose

The purpose of this document is to provide a detailed specification of the **Moksh Chat Application**, a real-time chat system developed using the **MERN** stack (MongoDB, Express.js, React.js, and Node.js). This SRS document outlines the functional and non-functional requirements, system architecture, and features needed for the successful development of the application.

1.2 Scope

The scope of **Moksh** includes:

- Real-time messaging (one-on-one and group chats).
- Secure user authentication and authorization.
- Media sharing (images, videos, documents).
- Notifications for new messages.
- Scalability to handle a growing user base and heavy traffic.

1.3 Definitions, Acronyms, and Abbreviations

- **MERN**: MongoDB, Express.js, React.js, Node.js.
- **JWT**: JSON Web Token, used for secure user authentication.
- **UI/UX**: User Interface/User Experience.
- **Socket.io**: A library used for real-time, bi-directional communication between web clients and servers.

2. Overall Description

2.1 Product Perspective

Moksh is a web-based chat application that allows users to communicate in real-time. The system is designed to handle high concurrency and ensure fast and secure message delivery, while providing a seamless user experience across all devices.

2.2 Product Features

- **Real-Time Messaging:** Enables users to exchange messages instantly.
- **Group Chats:** Allows multiple users to participate in the same conversation.
- **Media Sharing:** Users can share images, videos, and documents.
- **Notifications:** Real-time notifications for incoming messages.
- **Secure Authentication:** JWT-based authentication to ensure data privacy and security.
- **Responsive Design:** Works on desktops, tablets, and mobile devices.

2.3 User Characteristics

The users of Moksh include:

- **Individuals:** Users looking for personal real-time communication.
- **Teams/Organizations:** Professional users who need group messaging for collaboration.
- **Admins:** System administrators responsible for managing users and maintaining the system.

2.4 Constraints

- **Performance:** The system must handle thousands of simultaneous users without lag.
- **Security:** Strong user authentication and encryption of sensitive data.

- **Scalability:** The system should scale horizontally to support increasing user traffic.

3. Functional Requirements

3.1 User Registration and Login

- Users must be able to create an account by providing a valid email and password.
- Secure login using **JWT** tokens, with password hashing for protection.
- Users can reset their password using email verification.

3.2 Real-Time Messaging

- Users can send messages to other users in real-time, without page reloads.
- Both individual and group messaging should be supported.
- Messages should be timestamped and delivered in the correct order.

3.3 Group Chats

- Users can create and join group chats.
- Group administrators can manage members (invite, remove, mute).

3.4 Media Sharing

- Users can upload and share images, videos, and documents within chats.
- The system must limit the file size and format to ensure performance.

3.5 Notifications

- Users will receive notifications for new messages even when they are not actively using the app.
- Notifications must be real-time and disappear when the user opens the app or views the message.

3.6 User Profiles

- Each user will have a profile displaying basic information (name, profile picture, status).
- Users can update their profile information and profile picture.

4. Non-Functional Requirements

4.1 Performance

- The system must handle up to 1000 messages per second in real-time.
- Response time for message delivery should be less than 1 second.

4.2 Security

- All communication between users and the server should be encrypted using SSL.
- JWT will be used for secure session management.
- User passwords must be encrypted using hashing algorithms like bcrypt.

4.3 Scalability

- The system should support horizontal scaling to handle increasing numbers of users.
- The database should be able to store millions of messages without degradation in performance.

4.4 Availability

- The system must have at least 99.9% uptime, with robust error-handling mechanisms in place to minimize downtime.

4.5 Usability

- The user interface must be intuitive, responsive, and easy to navigate.

- The app must be accessible across various devices including desktops, tablets, and smartphones.

4.6 Maintainability

- The codebase should follow modular design principles to allow for easy updates and feature additions.
- The system should include unit tests and integration tests to ensure the reliability of code changes.

5. System Architecture

- **Frontend:** Developed using **React.js** for building a dynamic and responsive user interface.
- **Backend:** Built on **Node.js** and **Express.js**, handling the API routes and business logic.
- **Database:** **MongoDB** is used to store user data, chat messages, and media.
- **Real-Time Communication:** **Socket.io** is used to enable real-time messaging between clients.
- **Authentication:** **JWT** for secure user login and session management.

6. External Interface Requirements

6.1 User Interfaces

- A web-based user interface will allow users to send messages, manage their profile, and interact in real-time.
- The interface will support media sharing, group chat creation, and notifications.

6.2 Hardware Interfaces

- Moksh will run on standard servers and cloud environments that support Node.js and MongoDB.
- Clients can access the application through any modern web browser.

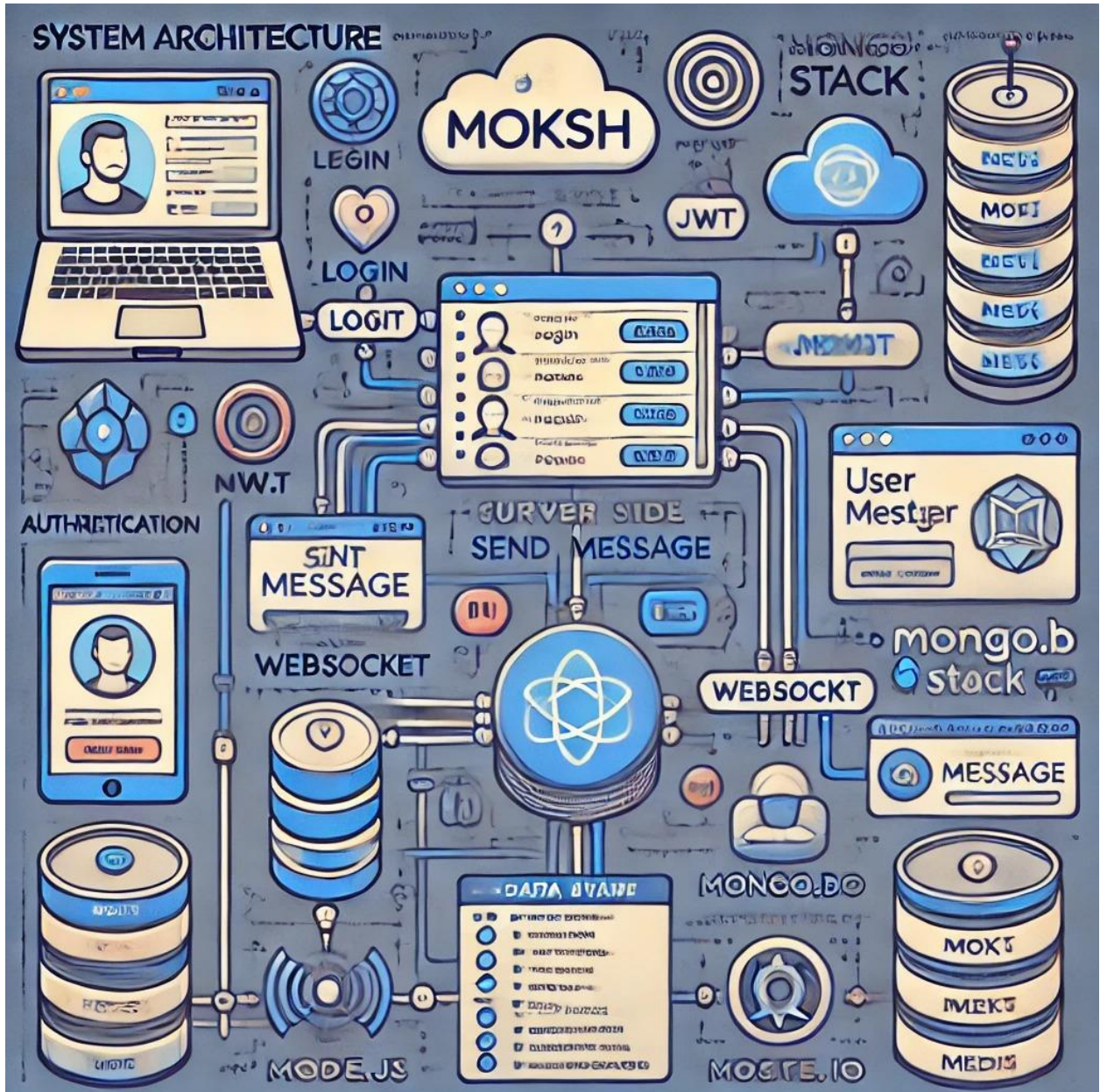
6.3 Software Interfaces

- **REST APIs** for interaction between the frontend and backend.
- **WebSocket** connections for real-time messaging.

7. Assumptions and Dependencies

- The system will be deployed on a cloud infrastructure like **AWS** or **Heroku**.
- Users are expected to have a reliable internet connection for seamless real-time communication.
- The application depends on third-party services like **JWT** for authentication and **Socket.io** for real-time messaging.

System Design for Moksh: Real-Time Chat Application

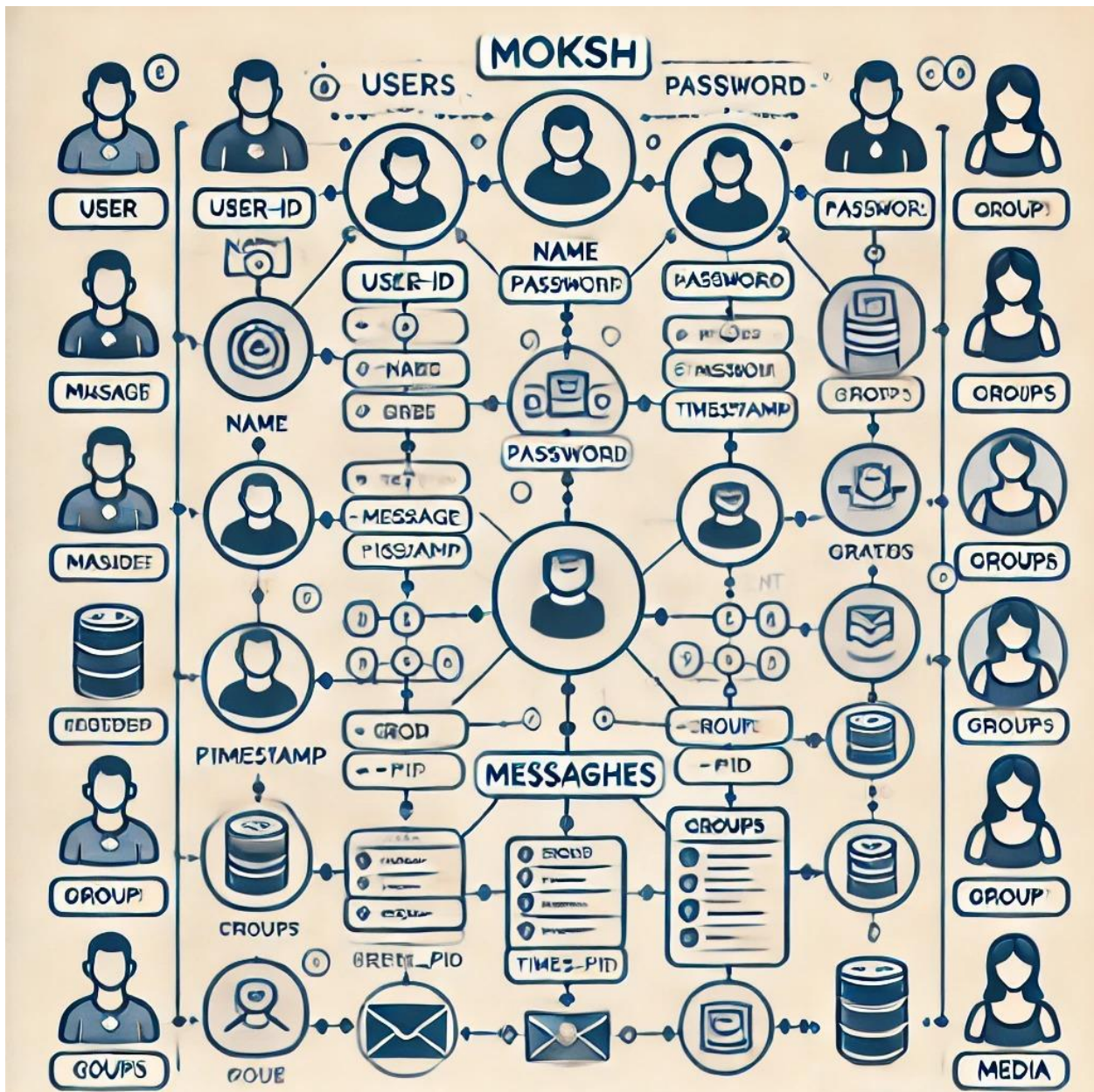


Here is the **System Design (Client-Server Architecture)** for the **Moksh Chat Application** using the **MERN** stack. It represents the data flow between the client, server, and database, including how the application handles user login, messaging, and real-time communication using **Socket.io** and **JWT** authentication.

Description:

1. **Client-Side (Browser with React.js):**
 - a. The client is the user's browser running the frontend, built using **React.js**. Users can log in, send messages, view chats, and receive real-time notifications.
 - b. Client-side interactions involve making HTTP requests (e.g., login) and maintaining a WebSocket connection for live chat updates using **Socket.io**.
2. **Server-Side (Node.js with Express.js):**
 - a. **Node.js** and **Express.js** handle the backend of the application, receiving requests from the client.
 - b. The server processes user requests like login (using **JWT** for authentication), sending and receiving messages in real-time using **Socket.io**.
3. **Database (MongoDB):**
 - a. MongoDB is used to store user information, chat history, and media files (e.g., images and documents).
 - b. The server communicates with MongoDB to query or update the database as needed (e.g., retrieving chat history or saving new messages).
4. **Real-Time Messaging:**
 - a. The server uses **Socket.io** to establish WebSocket connections for real-time communication, ensuring messages are sent and received instantly between users.

This design provides a clear view of the interactions and flow of data in a typical **Client-Server architecture** for the Moksh chat application.



Here is the **Entity-Relationship (ER) Diagram** for the **Moksh Chat Application**. It visualizes the relationships between the key entities involved, such as **Users**, **Messages**, **Groups**, and **Media**, as well as their relevant attributes.

Detailed Description:

1. Users:

- a. Attributes: user_id, name, email, password, profile_pic, status.

- b. Relationship: Users can send multiple **Messages** and can belong to multiple **Groups**.
- 2. **Messages:**
 - a. Attributes: message_id, content, timestamp, sender_id, group_id.
 - b. Relationship: Each message is sent by a single **User** and can be part of a **Group** or a one-on-one chat.
- 3. **Groups:**
 - a. Attributes: group_id, group_name, admin_id.
 - b. Relationship: **Groups** have multiple **Users** and can contain multiple **Messages**. Each group has an **admin** (a user who manages the group).
- 4. **Media:**
 - a. Attributes: media_id, file_type, file_url, message_id.
 - b. Relationship: **Media** files are attached to **Messages** (e.g., images, videos, documents).

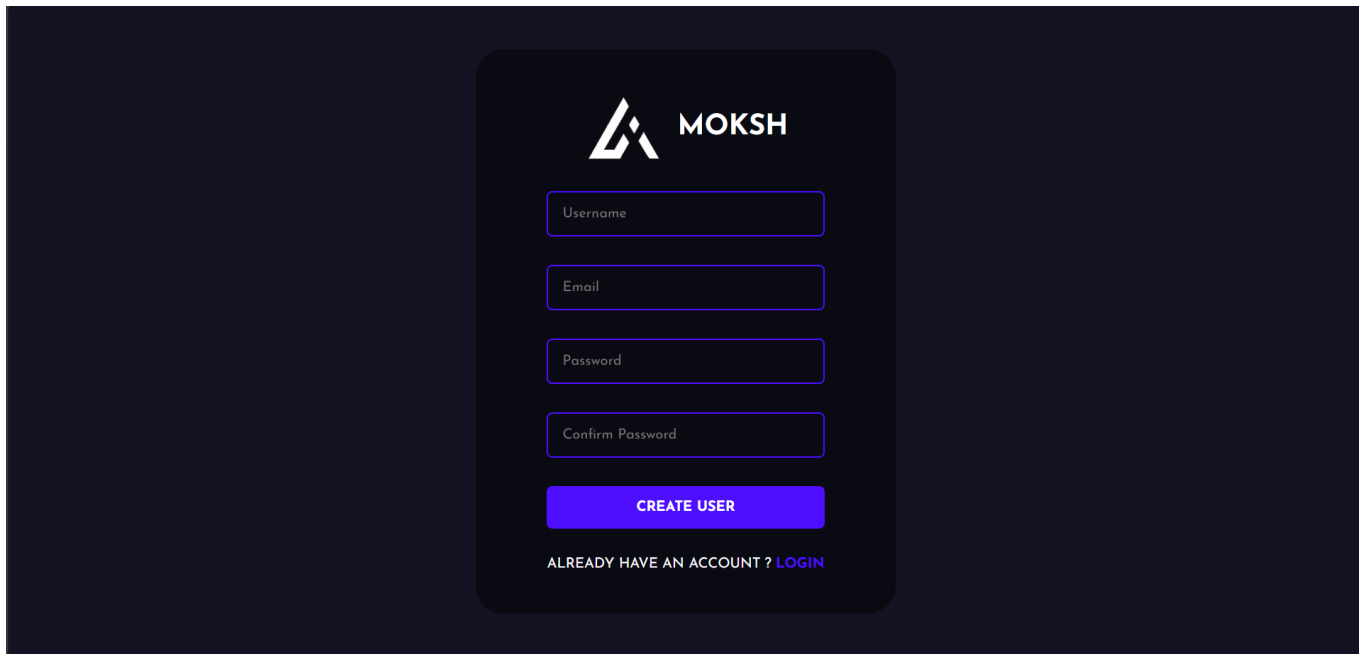
Relationships:


- **Users send Messages.**
- **Messages belong to Groups** or one-on-one chats.
- **Media** are associated with **Messages**.

This ER diagram shows how the system's data is organized and the relationships between key entities, ensuring the Moksh application can efficiently manage real-time messaging, user interactions, and media sharing.

Result

For Registration:

A registration form for 'MOKSH' on a dark blue background. The form is centered in a lighter blue rounded rectangle. It features the MOKSH logo at the top, followed by four input fields for Username, Email, Password, and Confirm Password. A red 'CREATE USER' button is below the fields, and a link to 'LOGIN' is at the bottom.

 **MOKSH**

Username

Email

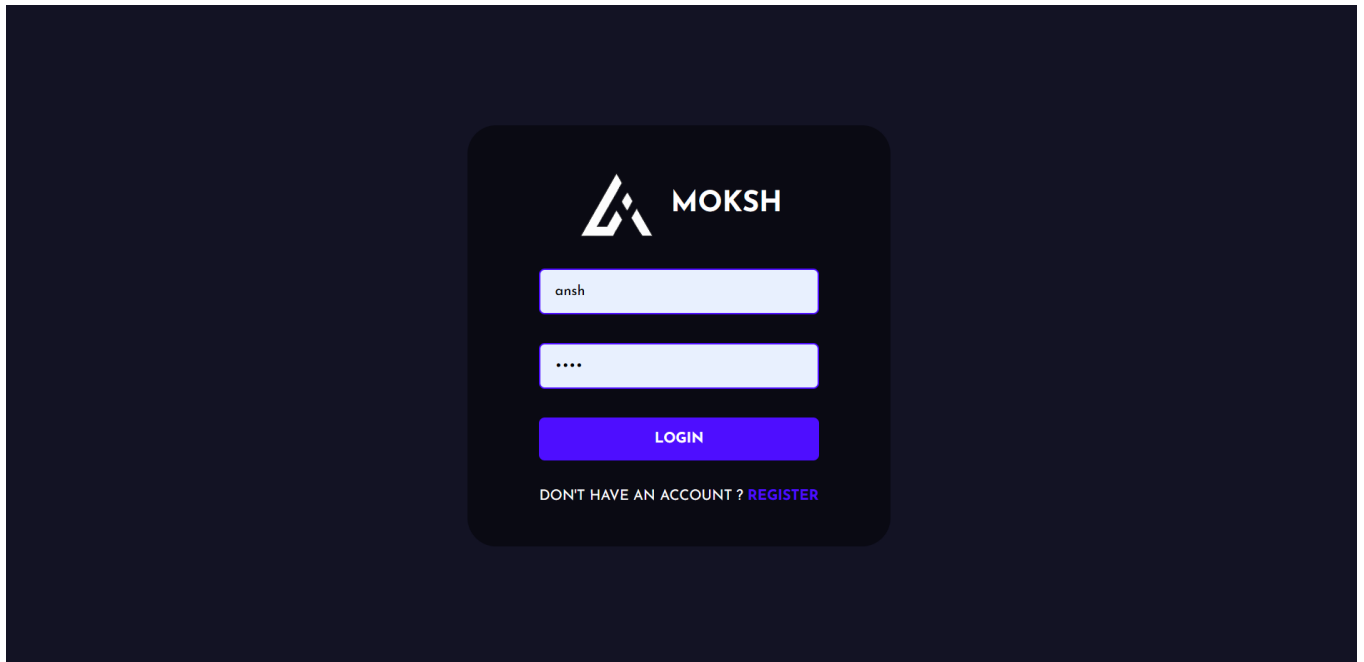
Password

Confirm Password

CREATE USER

ALREADY HAVE AN ACCOUNT ? [LOGIN](#)

For log-in:

A login form for 'MOKSH' on a dark blue background. The form is centered and contains a logo with a stylized 'A' and the word 'MOKSH'. Below the logo are two input fields: the first contains the text 'ansh' and the second contains four dots. A blue 'LOGIN' button is positioned below the password field. At the bottom of the form, there is a link that reads 'DON'T HAVE AN ACCOUNT ? REGISTER'.

For Selecting Avatar:

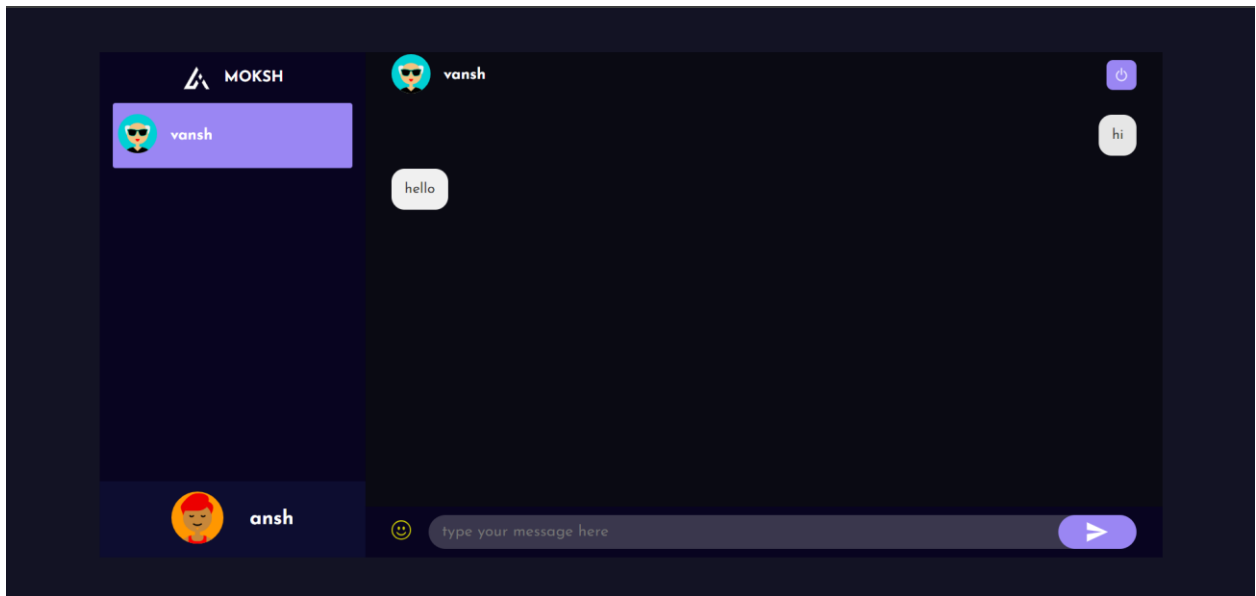
Pick an avatar as your profile picture



Please Wait

SET AS PROFILE PICTURE

Chat app ui:



Summary of Moksh Chatting App

Moksh is a chatting app designed to facilitate meaningful conversations and enhance social connections among users. It features real-time messaging, voice and video calls, and unique tools for sharing multimedia content. Moksh emphasizes user privacy and security, employing end-to-end encryption and customizable privacy settings. The app also incorporates community-building elements, allowing users to join interest-based groups and participate in discussions.

Conclusion

In conclusion, Moksh stands out in the crowded messaging app market by prioritizing user engagement and privacy. Its combination of robust features and a focus on community interactions positions it as a valuable platform for users seeking to connect more deeply with others. As it continues to evolve, Moksh has the potential to become a preferred choice for those looking to foster genuine relationships in the digital age.

References

Official MERN Stack Documentation:

- **MongoDB:** [MongoDB Documentation](#)
- **Express.js:** [Express Documentation](#)
- **React:** React Documentation
- **Node.js:** Node.js Documentation

Tutorials:

- **Build a Chat Application with MERN:**
 - MERN Chat App Tutorial
- **YouTube Video Tutorials:**
 - [MERN Stack Tutorial for Beginners](#)

Appendix

- Code for frontend:

ChatContainer.jsx

```
import React,{useState,useEffect, useRef} from 'react';
import styled from 'styled-components';
import Logout from './Logout';
import ChatInput from './ChatInput';
import Messages from './Messages';
import axios from 'axios';
import { getAllMessageRoute, sendMessageRoute } from '../utils/APIRoutes';

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

    const [messages,setMessages]=useState([]);
    const [recievedMessage,setRecievedMessage]=useState(null);
    const scrollRef = useRef();

    //this useEffect will run whenever current chat is changed
    useEffect(() => {
        //fetch all messages between currentUser and currentChat
        const fetchMessages = async () => {
            if (currentUser && currentChat) {
                const response = await axios.post(getAllMessageRoute, {
                    from: currentUser._id,
                    to: currentChat._id,
                });
                //store all messages
                //this data have two field fromSelf(boolean),message(msg)
                setMessages(response.data);
            }
        };
        fetchMessages();
    }, [currentChat]);
```



```
const handleSendMsg = async(msg) => {
  await axios.post(sendMessageRoute,{
    from:currentUser._id,
    to:currentChat._id,
    message:msg,
  });
  //after sending msg to server also emit send-msg event for socket
  and send credentials
  socket.current.emit("send-msg",{
    to:currentChat._id,
    from:currentUser._id,
    message:msg,
  });

  const msgs = [...messages]; //copying current state of messages to
  array
  msgs.push({fromSelf:true, message:msg}); //pushing new messages to
  to msgs array
  setMessages(msgs); //updating state with new array
};

useEffect(()=>{
  if(socket.current){
    socket.current.on("msg-recieve", (msg)=>{
      setRecievedMessage({fromSelf:false,message:msg});
    });
  }
},[]);
```

```

    useEffect(()=>{ //recieve previous state(prev) as argument and
    copy that along with recievedMessage is message State
    //if recievedMessage is true then execute the fun;
    recievedMessage && setMessages((prev)=>[...prev,recievedMessage]
    );

    },[recievedMessage]); //execute whenever a msg is recieved

    useEffect(()=> {
    //scroll into view for animation of messages
    scrollRef.current?.scrollIntoView({behaviour:"smooth"});
    },[messages]);

    return (
    <>
    {
    currentChat &&
    <Container>
    <div className="chat-header">
    <div className="user-details">
    <div className="avatar">
    <img
    src={`data:image/svg+xml;base64,${currentChat
    .avatarImage}`}
    alt="avatar"
    />
    </div>
    <div className="username">
    <h3>{currentChat.username}</h3>
    </div>
    </div>
    <Logout/>
    </div>
    }
    )
  )

```

```

<div className="chat-messages">
  {
    messages.map((message)=>{
      return(
        <div ref={scrollRef}>
          {/* if message.fromSelf is true then sended class will
be applied else add recieved class on message div */}
          <div className={`message ${message.fromSelf ? "sended"
:"recieved"}}`>
            <div className="content">
              <p>
                {message.message}
              </p>
            </div>
          </div>
        </div>
      )
    })
  }
</div>
{/* Sending handleSendMsg fun as a prop */}
<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: #1e1d233e;
      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: #333333;

    @media screen and (min-width: 720px) and (max-width: 1080px) {
      max-width: 70%;
    }
  }
}
```




```
.sended {  
  justify-content: flex-end;  
  
  .content {  
    background-color: #e6e6e6;  
  }  
}  
  
.recieved {  
  justify-content: flex-start;  
  
  .content {  
    background-color: #f0f0f0;  
  }  
}  
};
```

ChatInput.jsx

```
import React,{useState} from 'react';
import styled from 'styled-components';
import Picker from 'emoji-picker-react';
import {IoMdSend} from 'react-icons/io' ;
import {BsEmojiSmile} from 'react-icons/bs';

export default function ChatInput({handleSendMsg}) {
  const [showEmojiPicker,setShowEmojiPicker]=useState(false);
  const [msg,setMsg] = useState("");

  const handleEmojiPickerHideShow=()=>{
    //clicked on emoji then revert the condition weather it is shown
    setShowEmojiPicker(!showEmojiPicker);
  }

  //Recieved emoji on click of Picker event.emoji
  const handleEmojiClick =(event)=>{
    let message = msg;
    message+=event.emoji;
    setMsg(message);
  }

  //clicked on send button = we pass msg to handleSendMsg fun and set
  //SetMsg to empty
  const sendChat = (event)=>{
    event.preventDefault();
    if(msg.length>0){
      //this fun we recieved as prop so msg will be sent from where we
      //passed the prop
      handleSendMsg(msg);
      setMsg('');
    }
  }
}
```

```

return (
  <Container>
    <div className="button-container">
      <div className="emoji">
        <BsEmojiSmile onClick={handleEmojiPickerHideShow}/>
        { //if showEmojiPicker is true Picker will be evaluated
          showEmojiPicker && <Picker onEmojiClick
={handleEmojiClick} className='emoji-picker-react' />
        }
      </div>
    </div>
    <form className='input-container' onSubmit={(e)=>sendChat(e)}>

      { /* //as we start messagesing setMsg sets input value in msg and
we have made input value as msg */}
      <input type="text" placeholder='type your message here' value
={msg} onChange={(e)=> setMsg(e.target.value)} />
      <button className='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: -465px;
        background-color: #080420;
        box-shadow: 0 5px 10px #9a86f3;
        border-color: #9a86f3;
      }
    }
  }
}
```

```
.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;
  cursor: pointer;
  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, currentUser ,changeChat}) {
  const [currentUserName, setCurrentUserName] = useState(undefined);
  const [currentUserImage, setCurrentUserImage] = useState(undefined);
  const [currentSelected, setCurrentSelected] = useState(undefined);

  useEffect( () => {
    console.log(contacts)
    if(currentUser){
      setCurrentUserImage(currentUser.avatarImage);
      setCurrentUserName(currentUser.username);}
  }, [currentUser]);

  const changeCurrentChat = (index, contact) => {
    setCurrentSelected(index);
    changeChat(contact);
  };

  return (
    <>
      {currentUserImage && currentUserName && (
        <Container>
          <div className="brand">
            <img src={Logo} alt="logo" />
            <h3>Moksh</h3>
          </div>

```

```

<div className="contacts">
  {contacts.map((contact, index) => {
    return (
      <div
        key={index}
        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;
    //::Pseudo Selector for scrollbar
    &::-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;
    }
  }
}

//making responsive
@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 styled from 'styled-components';
import { BiPowerOff } from 'react-icons/bi';

export default function Logout() {
  const navigate = useNavigate();
  const handleClick = async()=>{
    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;
  }
`;
```

Messages.jsx



```
import React from 'react';
import styled from 'styled-components';

export default function Messages() {
  return (
    <div>
      Messages
    </div>
  )
}

const Container = styled.div``;
```

Welcome.jsx

```
import React from "react";
import styled from "styled-components";
import Robot from "../assets/robot.gif";
import Logout from './Logout';
export default function Welcome({currentUser}) {

  return (
    <Container>
      <Logout id='logout' />
      <br />
      <h1>MOKSH Me Aapka Swagat Hai</h1>
      <img src={Robot} alt="Robot" />
      <h1>
        Welcome, <span>{currentUser.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;
  }
`;
```

Chat.jsx

```
import React,{useState,useEffect,useRef} from 'react';
import styled from 'styled-components';
import axios from "axios";
import { useNavigate } from "react-router-dom";
import { allUsersRoute,host } from '../utils/APIRoutes';
import Contacts from '../components/Contacts';
import Welcome from '../components/Welcome';
import ChatContainer from '../components/ChatContainer';
import {io} from "socket.io-client";

function Chat() {
  const socket = useRef();
  const navigate = useNavigate();
  const [contacts,setContacts]=useState([]);
  const [currentUser,setCurrentUser]= useState(undefined);
  const [currentChat,setCurrentChat]=useState(undefined);
  const [isLoading,setIsLoaded] = useState(false);

  useEffect( ()=>{
    const fun = async()=>{
      if(!localStorage.getItem("chat-app-user")){
        navigate('/login');
      }else{
        setCurrentUser(await JSON.parse(localStorage.getItem("chat-app
-user"))));
        setIsLoaded(true);
      }
    }
    fun();
  },[]);
```

```

    //as current user Changes we add that user's id in socket
    useEffect(()=>{
      if(currentUser){
        //Estabilizing a socket connection with server(host) using io()
        socket.current = io(host);
        console.log(socket.current);
        //emit event(add-user) with current user id
        socket.current.emit("add-user",currentUser._id);
      }
    },[currentUser]);

    //run this hook whenever we set the current user
    useEffect(()=>{
      const fun = async()=>{
        if(currentUser){
          if(currentUser.isAvatarImageSet){
            //fetching all contants once we are logged in
            const data = await axios.get(`${allUsersRoute}/${currentUser
._id}`);
            setContacts(data.data);
          }else{
            navigate("/setAvatar");
          }
        }
      }
      fun();
    },[currentUser])

```



```

const handleChatChange = (chat) => {
  setCurrentChat(chat);
}

return (
  <Container>
    <div className="container">
      /* calling component and sending contacts and currentUser as props
      */
      <Contacts contacts={contacts} currentUser={currentUser} changeChat
      ={handleChatChange}/>
      {//show welcome if no chat is selected else show msg conatiner
      isLoading && currentChat === undefined ?(
        <Welcome currentUser={currentUser}/>):(
        //passing socket to ChatContainer
        <ChatContainer currentChat={currentChat} currentUser =
        {currentUser} 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%;
    //make this responsive
    @media screen and (min-width: 720px) and (max-width: 1080px) {
      grid-template-columns: 35% 65%;
    }
  }
`;

export default Chat
```

Login.jsx

```
import React, { useState,useEffect } from 'react'
import { Link, useNavigate } from 'react-router-dom';
import styled from 'styled-components';
import Logo from '../assets/logo.svg';
import {ToastContainer,toast} from 'react-toastify';
import "react-toastify/dist/ReactToastify.css";
import axios from "axios";
import { loginRoute } from '../utils/APIRoutes';

function Login() {
  const navigate= useNavigate();
  const [values,setValues]=useState({
    username: "",
    password: "",
  });
  //already login redirect to chat page
  useEffect(()=> {
    if(localStorage.getItem('chat-app-user')){
      navigate('/');
    }
  },[])

  const toastOptions={
    position:"bottom-right",
    autoClose:8000,
    pauseOnHover:true,
    draggable:true,
    theme:"dark",
  };
}
```

```

const handleSubmit = async(event)=>{
  event.preventDefault();
  if(handleValidation()){
    console.log("In validation",loginRoute);
    //condition true==call api
    const {password,username} = values;
    const {data}= await axios.post(loginRoute,{
      //sending these data to server(POST rqst) and waiting for
      response data
      username,
      password,
    });
    if(data.status===false){
      toast.error(data.msg,toastOptions);
    }
    if(data.status===true){
      //saving userdata in localStorage
      localStorage.setItem('chat-app-user',JSON.stringify(data.user
    ));
      navigate("/");
    }
  }
};

```

```

const handleValidation =()=>{
  const {password,username} = values;
  if(username===""){
    toast.error("username and password required",toastOptions);
    return false;
  }else if(password===""){
    toast.error("username and password required",toastOptions);
    return false;
  }
  //everything is ok return true
  return true;
}

const handleChange = (event)=>{
  event.preventDefault();
  setValues({...values,[event.target.name]:event.target.value});
}

return (
  <>
    <FormContainer>
      <form onSubmit={{(event)=>handleSubmit(event)}}>
        <div className="brand">
          <img src={Logo} alt="Logo" />
          <h1>Moksh</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'>Login</button>
        <span>Don't have an account ? <Link to="/register">Register</Link
    </span>
  </form>
</FormContainer>
<ToastContainer/>
</>
)
}
//a new styled component FormContainer will render as a div
//` backticks allow to write CSS within Javascript
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;
  }
}
`
export default Login

```

Register.jsx

```
import React, { useEffect, useState } from 'react'
import { Link, useNavigate } from 'react-router-dom';
import styled from 'styled-components';
import Logo from '../assets/logo.svg';
import {ToastContainer,toast} from 'react-toastify';
import "react-toastify/dist/ReactToastify.css";
import axios from "axios";
import { registerRoute } from '../utils/APIRoutes';

function Register() {
  const navigate= useNavigate();
  const [values,setValues]=useState({
    username: "",
    email: "",
    password: "",
    confirmPassword: "",

  });

  useEffect(()=> {
    if(localStorage.getItem('chat-app-user')){
      navigate('/');
    }
  },[])
}
```



```

const handleSubmit = async(event)=>{
  event.preventDefault();
  if(handleValidation()){
    console.log("In validation",registerRoute);
    //condition true==call api
    const {password,email,username} = values;
    const {data}= await axios.post(registerRoute,{
      //sending these data to server(POST rqst) and waiting for
response data
      username,
      email,
      password,
    });
    if(data.status===false){
      toast.error(data.msg,toastOptions);
    }
    if(data.status===true){
      //saving userdata in localStorage
      localStorage.setItem('chat-app-user',JSON.stringify(data.user
    ));
      navigate("/");
    }
  }
};

const toastOptions={
  postion:"bottom-right",
  autoClose:8000,
  pauseOnHover:true,
  draggable:true,
  theme:"dark",
};

```

```

const handleValidation =()=>{
  const {password,confirmPassword,email,username} = values;
  if(password!==confirmPassword){
    toast.error("Password and Confirm Password should be same"
,toastOptions);
    return false;
  }else if(username.length<4){
    toast.error("Username should be greater then 3 charecters"
,toastOptions);
    return false;
  } else if(password.length<6){
    toast.error("password should be greater then or equal to 6
charecters",toastOptions);
    return false;
  }else if(email===""){
    toast.error("email is required",toastOptions);
    return false;
  }
  //everything is ok return true
  return true;
}

const handleChange = (event)=>{
  event.preventDefault();
  setValues({...values,[event.target.name]:event.target.value});
}

```

```

return (
  <>
    <FormContainer>
      <form onSubmit={{(event)=>handleSubmit(event)}}>
        <div className="brand">
          <img src={Logo} alt="Logo" />
          <h1>Moksh</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;
  }
}
`
;

export default Register

```

SetAvatar.jsx

```
import React, { useEffect, useState } from 'react'
import { useNavigate } from 'react-router-dom';
import styled from 'styled-components';
import loader from "../assets/loader.gif"
import {ToastContainer,toast} from 'react-toastify';
import "react-toastify/dist/ReactToastify.css";
import axios from "axios";
import { SetAvatarRoute, setAvatarRoute } from '../utils/APIRoutes';
import { Buffer } from 'buffer';
import { IoReloadCircleOutline } from "react-icons/io5";

export default function SetAvatar() {

  const api ='https://api.multiavatar.com/45678945134';
  const navigate = useNavigate();
  const [avatars,setAvatars] = useState([]);
  //while avatars are loading will show loader
  const [isLoading,setIsLoading] = useState(true);
  const [selectedAvatar,setSelectedAvatar] =useState(undefined);

  const toastOptions={
    postion:"bottom-right",
    autoClose:8000,
    pauseOnHover:true,
    draggable:true,
    theme:"dark",
  };
};
```

```

useEffect(()=> {
  //if no data of user/not logged in navigate to login page
  if(!localStorage.getItem('chat-app-user')){
    navigate('/login');
  }
},[])

const setProfilePicture = async() => {
  if(selectedAvatar===undefined){
    toast.error("Please select an avatar",toastOptions)
  } else{
    //retrieving json data from local storage
    const user = await JSON.parse(localStorage.getItem("chat-app-user"));
    //axios rqst to setAvatarRoute/user and sending avatar data in request
    body
    const {data} = await axios.post(`${setAvatarRoute}/${user._id}`,
      {image:avatars[selectedAvatar],}
    );
    console.log(data);
    if(data.isSet){
      user.isAvatarImageSet=true;
      user.avatarImage= data.image;
      localStorage.setItem("chat-app-user",JSON.stringify(user));
      navigate('/')
    }else{
      toast.error("Error setting avatar:Please try again",toastOptions);
    }
  }
};

```

```

useEffect(() => {
  const fetchData = async () => {
    const data = [];
    try {
      for (let i = 0; i < 4; i++) {
        // Calling avatar API and passing any random number to generate
        random avatars
        const image = await axios.get(`${api}/${Math.round(Math.random() *
1000)}`);

        const buffer = new Buffer(image.data); // Use Buffer.from() to
create buffer
        //pushing the fetched avatar buffer to data array
        data.push(buffer.toString("base64"));
      }

      setAvatars(data);
      setIsLoading(false);
    } catch (error) {
      console.error('Error fetching avatars:', error);
      toast.error('Failed to fetch avatars. Please try again later.',
toastOptions);
      setIsLoading(false);
    }
  };

  fetchData();
}, []);

const reloadPage = ()=>{
  window.location.reload();
}

```



```

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 key={index} className={`avatar ${selectedAvatar
===index? "selected" : ""}`}>
                  <img src={`data:image/svg+xml;base64,${avatar}`}
alt="avatar"
                    onClick={()=>setSelectedAvatar(index)}>
                </div>
              )
            })
          }
        </div>
        <div>
          <IoReloadCircleOutline color='white' size='50px' onClick
={reloadPage} className='reload' />
        </div>
        <div>
          <p>Please Wait</p>
        </div>
        <button className='submit-btn' onClick={setProfilePicture}>Set as
Profile Picture</button>
      </Container>
    )
  }

  <ToastContainer/>
</>
)
}

```



```
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;
  flex-wrap: wrap; /* Ensure avatars wrap to new lines on smaller screens
  */
  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;
  }
}

.reload {
  display: flex;
  justify-content: center;
  align-items: center;
  cursor: pointer;
}
```



```
p {
  color: white;
  font-size: 30px;
}

.submit-btn {
  background-color: #a79cc3;
  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;
  }
}

@media (max-width: 768px) {
  /* Adjust styles for screens up to 768px wide (typical phone width) */
  .avatars {
    .avatar {
      img {
        height: 4rem; /* Decrease avatar image size for smaller screens */
      }
    }
  }
}
```



```
p {  
  font-size: 20px; /* Decrease font size for paragraphs on smaller  
screens */  
}  
  
.submit-btn {  
  font-size: 0.8rem; /* Decrease font size for submit button on smaller  
screens */  
}  
};
```

- Backend code snippet:

messageModel.js

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

const messageSchema = new 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)
```

userModel.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:6,
  },
  isAvatarImageSet:{
    type:Boolean,
    default:false,
  },
  avatarImage:{
    type:String,
    default:"",
  },
});

//Users is database collection name
module.exports = mongoose.model("Users",userSchema)
```

MessageController.js

```
const messageModel = require('../model/messageModel');

module.exports.addMessages = async(req,res,next)=>{
  try{
    const {from,to,message} = req.body;
    //sending messages to database
    const data = await messageModel.create({
      message:{text:message},
      users: [from,to],
      sender : from,
    });
    if(data) return res.json({msg:"Message Added succesfully"});
    return res.json({msg:"Failed to add message to Database"});

  }catch(er){
    next(er);
  }
};

module.exports.getAllMessage = async(req,res,next)=>{
  //getting all messages from database to show in message section
  try{
    const {from,to} = req.body;
    const messages = await messageModel.find(
      {
        //where users array contain both from and to
        users:{
          $all:[from,to],
        },
      }
    ).sort({updatedAt : 1});

    const projectMessages = messages.map((msg)=>{
      return {
        fromSelf: msg.sender.toString()===from,
        message: msg.message.text,
      };
    });
    res.json(projectMessages);

  }catch(err){
    next(err);
  }
};
```


UserController.js

```
const messageModel = require('../model/messageModel');

module.exports.addMessages = async(req,res,next)=>{
  try{
    const {from,to,message} = req.body;
    //sending messages to database
    const data = await messageModel.create({
      message:{text:message},
      users: [from,to],
      sender : from,
    });
    if(data) return res.json({msg:"Message Added succesfully"});
    return res.json({msg:"Failed to add message to Database"});
  }catch(er){
    next(er);
  }
};

module.exports.getAllMessage = async(req,res,next)=>{
  //geting all messages from database to show in message section
  try{
    const {from,to} = req.body;
    const messages = await messageModel.find(
      {
        //where users array contain both from and to
        users:{
          $all:[from,to],
        },
      },
    ).sort({updatedAt : 1});
  }
```

```

const messageModel = require('../model/messageModel');

module.exports.addMessages = async(req,res,next)=>{
  try{
    const {from,to,message} = req.body;
    //sending messages to database
    const data = await messageModel.create({
      message:{text:message},
      users: [from,to],
      sender : from,
    });
    if(data) return res.json({msg:"Message Added succesfully"});
    return res.json({msg:"Failed to add message to Database"});
  }catch(er){
    next(er);
  }
};

module.exports.getAllMessage = async(req,res,next)=>{
  //geting all messages from database to show in message section
  try{
    const {from,to} = req.body;
    const messages = await messageModel.find(
      {
        //where users array contain both from and to
        users:{
          $all:[from,to],
        },
      }
    ).sort({updatedAt : 1});

    const projectMessages = messages.map((msg)=>{
      return {
        fromSelf: msg.sender.toString()===from,
        message: msg.message.text,
      };
    });
    res.json(projectMessages);
  }catch(err){
    next(err);
  }
};

```

```

const User = require("../model/userModel");
//for password encryption use bcrypt
const bcrypt = require("bcrypt");

//register controll

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});
    }
    //encrypting the password
    const hashedPassword = await bcrypt.hash(password,10);

    //creating user and sending data to DB(mongoDb)
    const user = await User.create({
      email,
      username,
      password:hashedPassword,
    });

    //deleting original password
    delete User.password;
    return res.json({status:true,user});

  } catch(err){
    next(err);
  }
};

```

```

//login controll

module.exports.login = async(req,res,next)=>{
  try{
    const {username,password} = req.body;

    const user = await User.findOne({username});
    //if user not found in DB
    if(!user){
      return res.json({msg:"Incorrect username or password",status
: false});
    }
    //matching password
    const isPasswordValid = await bcrypt.compare(password,user.password
);
    if(!isPasswordValid){
      return res.json({msg:"Incorrect username or password",status
: false});
    }
    //delete password fetched in user
    delete user.password;
    return res.json({status:true,user});

  } catch(err){
    next(err);
  }
};

```

```

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,
    });
    //sending response to rqst
    res.json({isSet:userData.isAvatarImageSet,image:userData.avatarImage}
  );
  }catch(err){
    console.log(err)
  }
}

module.exports.getAllUsers = async(req,res,next)=>{
  try{
    const users = await User.find({_id:{$ne:req.params.id}}).select([
      //selecting only required deatils from Database
      "email",
      "username",
      "avatarImage",
      "_id"
    ]);
    return res.json(users);
  } catch(err){
    next(err);
  }
}

```

MessageRoute.js



```
const {addMessages,getAllMessage}= require('../controller
/messageController');

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

router.post("/addmsg",addMessages);
router.post("/getmsg",getAllMessage);

module.exports=router;
```

UserRoute.js



```
const { register,login, setAvatar, getAllUsers } = require("../controller
/userController");

const router = require("express").Router();
//on post rqst on /register evaluate register fun
router.post('/register',register);
//on post rqst on /login evaluate login fun
router.post('/login',login);

router.post('/setAvatar/:id',setAvatar);
router.get('/allusers/:id',getAllUsers)
module.exports=router;
```

Index.js

```
const express = require("express");
const cors = require("cors");
const mongoose = require("mongoose");
const userRoutes = require('./routes/userRoutes');
const messageRoute = require("./routes/messagesRoute");
const app = express();
const socket = require("socket.io");

require("dotenv").config();

//To use cors everywhere in program
app.use(cors());
app.use(express.json());

// This line is telling the Express application (app) to use the routes
// defined in the userRoutes variable for any requests that start with /api
// /auth same for messageRoute
app.use("/api/auth",userRoutes);
app.use("/api/messages",messageRoute);

app.get('/',(req,res)=>{
  res.json("Hello");
})

mongoose.connect(process.env.MONGO_URL,{
  useNewUrlParser:true,
  useUnifiedTopology:true,
}).then(()=>{
  console.log("DB connection Succesfull")
}).catch((err)=>{
  console.log(err.message);
})
```

```

const port = process.env.PORT || 5000;

const server = app.listen(port,()=>{
  console.log(`Server Started on PORT ${port}`);
})

//Estabilishing socket connections

const io = socket(server,{
  cors: {
    origin : "http://localhost:3000",
    credential: true,
  },
});

global.onlineUsers = new Map();

io.on("connection", (socket) => {
  //save current socket object globally in chatSocket
  global.chatSocket = socket;
  socket.on("add-user", (userId) => {
    //Add user id and socket id of Current socket-client as client
    //execute add-user event
    onlineUsers.set(userId, socket.id);
  });

  //as a socket execute send-msg event with data(to,msg) run this
  //callback fun
  socket.on("send-msg", (data) => {
    //extract Recievers socket id from online users
    const sendUserSocket = onlineUsers.get(data.to);
    if(sendUserSocket){
      //if Reciver id is recieved then emit them msg-recieve
      //event with msg(data.msg)
      socket.to(sendUserSocket).emit("msg-recieve", data.message);
    }
  })
});

```