Chat Application using MERN Stack

A Dissertation submitted to the Jawaharlal Nehru Technological University, Hyderabad in partial fulfillment of the requirement for the award of degree of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Submitted by

J. MADHUKAR	**20B81A0517**
D.ROHITH	**20B81A0525**
J. SAI CHARAN	**20B81A0528**

Under the guidance of
Mr . N . SRINU
Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CSE)

CVR COLLEGE OF ENGINEERING

(An Autonomous institution, NAAC Accredited and Affiliated to JNTUH, Hyderabad)
Vastunagar, Mangalpalli (V), Ibrahimpatnam (M),
Rangareddy (D), Telangana- 501 510

APRIL 2024

CVR COLLEGE OF ENGINEERING

(*An Autonomous institution, NAAC Accredited and Affiliated to JNTUH, Hyderabad*)
Vastunagar, Mangalpalli (V), Ibrahimpatnam (M),
Rangareddy (D), Telangana- 501 510

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DS)

CERTIFICATE

This is to certify that the project report entitled **"Chat Application using MERN Stack"** bonafide record of work carried out by **J. MADHUKAR (20B81A0517), D.ROHITH (20B81A0525)** and **J. SAI CHARAN (20B81A0528)** submitted to **Mr .N . SRINU** for the requirement of the award of **Bachelor of Technology** in **Computer Science and Engineering** to the CVR College of Engineering, affiliated to Jawaharlal Nehru Technological University, Hyderabad during the year 2023-2024.

Project Guide

Mr .N . SRINU
Assistant Professor
Department of CSE

Head of the Department

Dr. **VANI VATHSALA**
Professor & HOD

Project Coordinator

External Examiner

ACKNOWLEDGEMENT

We are thankful for and fortunate enough to get constant encouragement, support, and guidance from all **Teaching staff of CSE Department** which helped us in successfully completing this project work.

We thank **BHARATH**, Project Coordinator and **Dr. Sandhya Rani,** Project Review Committee members for their valuable guidance and support which helped us to complete the project work successfully.

We respect and thank our internal guide, **MR .N . SRINU,** Assistant Professor, Department of CSE, for giving us all the support and guidance which made us complete the project duly.

We would like to express heartful thanks to Dr. **VANI VATHSALA** , Professor & Head of the Department, for providing us an opportunity to do this project and extending support and guidance.

We thank our Vice-Principal **Prof. L. C. Siva Reddy** for providing excellent computing facilities and a disciplined atmosphere for doing our work.

We wish a deep sense of gratitude and heartfelt thanks to **Dr. Rama Mohan Reddy,** Principal **and the Management** for providing excellent lab facilities and tools. Finally, we thank all those guidance helpful to us in this regard.

DECLARATION

We hereby declare that the project report entitled **"Chat Application using MERN Stack"** is an original work done and submitted to Department, CSE , CVR College of Engineering, affiliated to Jawaharlal Nehru Technological University Hyderabad in partial fulfilment for the requirement of the award of Bachelor of Technology in Computer Science and Information Technology and it is a record of bonafide project work carried out by us under the guidance of **MR.N.SRINU**, Assistant Professor, Department of Computer Science and Information Technology.

We further declare that the work reported in this project has not been submitted, either in part or in full, for the award of any other degree or diploma in this Institute or any other Institute or University.

Signature of the Student

J. MADHUKAR

Signature of the Student

D.ROHITH

Signature of the Student

J. SAI CHARAN

Date:

Place:

ABSTRACT

The emergence of new technologies has brought about significant changes in the way people communicate with each other. One of the most popular ways of communication in today's digital age is through messaging applications. To facilitate this need, several chat applications have been developed. In this thesis, we introduce a chat application built using the MERN stack, which is a popular technology stack used for building web applications. This application provides users with the ability to create accounts, join chat rooms, and send messages to other users in real-time.With the use of web sockets and React two-way binding, the application allows users to see messages as soon as they are sent. Moreover, the application also includes features such as user authentication andauthorization to ensure secure access to the chat rooms. Through this project,we aim to demonstrate the feasibility and effectiveness of building real-time chat applications using the Mern stack. .

TABLE OF CONTENTS

CHAPTER-1
INTRODUCTION

The chatting application has huge impact on day to day life. There are numerous chatting application available in this world. Each application has different additional features varying from other applications. These application organizations compete with each other and add some competing features during each release. They have reached people much and have an impact on people's life. People find a better application from an available internet application which they feel much reliable and secure. Some of the available chatting applications that are available in these days are Whatsapp, Facebook, Instagram, Hike, etc...The above mentioned applications have billion users all over the world. Those companies are one of the top companies in the world. They have higher revenue per year and have many employees for their organizations developing additional features to compete with other organizations

during their each release. These applications have different features and follows different ways to ensure security of their user data. Today a data theft is the major crime and most people are involved in it. There are many cases being filed these days about personal data loss. So the organizations have to ensure the security from data loss by the third party data crisis. The basic chatting system should involve both sending and receiving processes simultaneously.In this application both sending and receiving messages simultaneously happens through MERN concept.

1.1 MOTIVATION

Developing a chat application using the MERN (MongoDB, Express.js, React, Node.js) stack offers a comprehensive solution for full-stack development. By utilizing JavaScript across the entire stack, developers can seamlessly transition between front-end and back-end development tasks, fostering better collaboration and code coherence within the development team. The real-time communication requirements inherent in chat applications are effectively addressed by Node.js, leveraging its event-driven architecture to facilitate instant updates and user interactions. MongoDB's scalability and ability to handle large volumes of data and concurrent connections make it an ideal choice for storing chat messages and user data. On the front-end, React's component-based architecture streamlines the development of dynamic and responsive user interfaces, ensuring an engaging chat experience. Furthermore, the extensive community support and ecosystem surrounding the MERN stack provide developers with a wealth of resources, libraries, and best practices to accelerate development and customization. Ultimately, the MERN stack empowers developers to rapidly build feature-rich chat applications tailored to specific requirements, with robust real-time communication capabilities, scalability, and flexibility at its core.

1.2 PROBLEM STATEMENT

Create a dynamic chat app with the MERN stack, utilizing MongoDB for robust data storage, Express.js for efficient server-side development, React for a responsive UI, and Node.js for real-time communication, seamlessly connecting users worldwide.

Facilitate seamless message, image, and file exchanges with the MERN stack. Harness scalability and flexibility for a feature-rich platform, ensuring vibrant and efficient communication experiences for socializing, collaboration, or customer support.

1.3 PROJECT OBJECTIVE

The project objective for building a chat application using the MERN (MongoDB, Express.js, React, Node.js) stack revolves around creating a robust platform for real-time communication. This entails implementing features such as instant messaging, user authentication, and message persistence within a scalable and responsive user interface. By leveraging MongoDB for data storage, Express.js for server-side logic, React for dynamic front-end interfaces, and Node.js for handling real-time connections, the goal is to provide users with a seamless chat experience across various devices. Security measures, including user authentication and data encryption, are paramount to safeguarding user information. Additionally, the project aims to facilitate customization through features like user profiles and personalization options while ensuring thorough testing and documentation for quality assurance and future maintenance. Ultimately, the objective is to deliver a feature-rich chat application that meets user needs for efficient, secure, and enjoyable communication.

1.4 PROJECT REPORT ORGANIZATION

In this “Project Report” a detailed description of the design challenges proposed methodologies, and the implementation of an application to solve the real-world problem is given. Different functionalities of the application are broken down into modules and explained with the help of use case diagrams and class diagrams. The

Chapter 2: It describes the literature survey, characteristics and design challenges of the existing system and provides a proposed solution.

Chapter 3: This chapter defines the software requirements which include functional requirements, non-functional requirements, system architecture and system specifications which include software requirements.

Chapter 4: This chapter describes the UML diagrams like the use case diagrams, class diagrams, activity diagrams, and architecture.

Chapter 5: This chapter discusses the implementation and the testing using various tools and their screenshots are discussed in detail.

Chapter 6: This chapter focuses on providing the conclusion of the proj

CHAPTER-2
LITERATURE REVIEW

The With the development and enhancement in internet, more and more people have been choosing network chatting tools for communication. Applications such as these facilitates communication over great distances. Therefore, this application must both be real-time and multi-platform to be used by many users. The web-based real-time chatting application does not need any additional third-party client program, and the visual communication could be established conveniently. The programming tools used in building this application is React.js, Node.js with express framework and Mongo DB database. The text communication is transferred through and from servers and the data transmission is facilitated through point to point connection between servers. Due to the usage of react framework,virtual space concept is implemented which enhances the performance over existing applications developed using

PHP by a factor of approximately 6 times. This paper is aimed at developing an Online College Management System that is of importance to the educational institute or college. This system is named College ERP using MERN stack. This system may be used to monitor college students and their various activities. This application is being developed for an engineering college to maintain and facilitate ease of access to information. For this the users must be registered with the system. College ERP is an Internet based application that aims at providing information to all levels of management within an organization. This system is used as an information management system for the college. For a given student and staff (technical and non technical) can access the system to either upload and access some information from the database.The latest development of the Internet has brought the world into our hands. Everything happens through internet from passing information to purchasing something. Internet made the world as small circle. This project is also based on internet. This paper shows the importance of chat application in day today life and its impact in technological world. This project is to develop a chat system based on Java multithreading and network concept.

The application allows people to transfer messages both in private and public way .It also enables the feature of sharing resources like files, images, videos, etc.This online system is developed to interact or chat with one another on the Internet. It is much more

reliable and secure than other traditional systems available. Java, multi threading and client-server concept were used to develop the web based chat application. This application is developed with proper architecture for future enhancement. It can be deployed in all private organizations like Colleges, IT parks,etc.

This research paper is about the modelling and construction of Management Information System with the help of MERN stack. The MERN stack consist of MongoDB, Express.js, React and Node.js. This MIS (Management Information System) is specially designed for IMOs and the Govt. of India. The schema modelling is flexible and it can be used by almost every loan providing govt. agency. For deploying the application, the database used is MongoDB and the server used is Heroku. The entire system can be integrated with Docker to reduce the development setup time. The application comes with built in feature that allows the admin to send messages to the loan-bearing group

2.1 EXISTING WORK

Earlier detection work has been done using DNN which is a deep neural network. One existing system for skin disease detection using DNN (Deep Neural Networks) is DermDetect, developed by Al-mudares and colleagues (2020). DermDetect uses a deep learning model to classify skin lesions into one of five categories: melanoma, nevus, seborrheic keratosis, basal cell carcinoma, or squamous cell carcinoma. The system achieves high accuracy in detecting these skin diseases.

2.2 Limitations of Existing System:

There are several limitations that you may encounter when developing a single message chat application using the MERN (MongoDB, Express.js,React, Node.js) technology stack. Here are a few potential limitations:Scalability: The MERN stack can handle moderate levels of traffic and users, but it may face scalability challenges when dealing with a large number of concurrent connections or heavy message traffic. Scaling the application to handle high loads might require implementing technique s such as load balancing, horizontal scaling, or using a dedicated messaging service.Performance: As the number of messages increases, fetching and rendering all the messages on the client side could lead to performance issues. It's important to implement pagination or infinite scrolling to limit the number of messages loaded at a time and optimize the rendering process.Security: Security is a critical aspect of any chat application. MERN provides tools and libraries for handling security concerns, such as authentication and authorization, but it's essential to ensure that properse curity measures are implemented to protect user data, prevent unauthorized access, and secure communication channels.Offline functionality: MERN does not natively support offline functionality. If users need to access the chat application or send messages while offline, you would need to implement additional techniques such as client-side caching or using a service worker to handle offline data synchronization.Mobile app development: While MERN is well-suited for web applications, if we plan to develop a mobile app version, additional frameworks or technologies, such as React Native or Flutter, would be required. This introduces the need for separate codebases and additional development efforts.

CHAPTER-3
REQUIREMENT ANALYSIS

The project involved analyzing the design of few applications so as to make the application more users friendly. To do so, it was really important to keep the navigations from one screento the other well ordered and at the same time reducing the amount of typing the user needs to do. In order to make the application more accessible, the browser version had to be chosen sothat it is compatible with most of the Browsers.

3.1 FUNCTIONAL REQUIREMENTS

Functional requirement should include function performed by a specific screen outline work- flows performed by the system and other business or compliance requirement the system must meet. Functional requirements specify which output file should be produced from the given file they describe the relationship between the input and output of the system, for each functional requirement a detailed description of all data inputs and their source and the range of valid inputs must be specified. The functional specification describes what the system must do, how the system does it is described in the design specification. If a user requirement specification was written, all requirements outlined in the user requirements specifications should be addressed in the functional requirements.

To develop a Chat Application using MERN Stack project , the following functional requirements are needed:

- User Authentication:
- 1.1. Users should be able to register with a unique username and password.
- 1.2. Users should be able to log in securely using their registered credentials.
- 1.3. Users should have the option to reset their passwords if forgotten.

- Chat Functionality:
- 2.1.Users should be able to send text messages in real-time.
- 2.2.Users should be able to view messages from other users in real-time without refreshing the page.
- 2.3.Users should have the ability to create new chat rooms or join existingones.

- 2.4. Users should be able to see the online status of other users.

- Message Management:
- 3.1. Users should be able to edit or delete their own messages.
- 3.2. Users should be able to view a history of their chat messages.
- 3.3. Messages should be stored securely and persistently in the database.

- User Profile Management:
- 4.1. Users should be able to update their profile information, including profile picture and display name.
- 4.2. Users should have the option to set their online status (available, away, offline).

- Security:
- 5.1. User authentication and session management should be secure to prevent unauthorized access.
- 5.2. Messages should be encrypted during transmission to ensure privacy.

- Error Handling:
- 6.1. Users should receive appropriate error messages for invalid inputs or failed operations.
- 6.2. System errors should be logged for troubleshooting and maintenance purposes.securely.

3.2 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements are aspects of a project that are not directly related to its functionality but are essential for ensuring its quality and usability. Describe user-visible aspects of the system that are not directly related with the functional behaviour of the system. Non-Functional requirements include quantitative constraints, such as response time (i.e. how fast the system reacts to user commands) or accuracy (i.e. how precise are the systems numerical answers).

For a Chat Application using MERN Stack project , the following non-functional

requirements should be considered:

- Performance:
- 1.1. The application should be responsive with minimal latency for sending and receiving messages.
- 1.2. The system should be able to handle a large number of concurrent users without significant degradation in performance.

- Scalability:
- 2.1. The architecture should be scalable to accommodate future growth in user base and message volume.
- 2.2. The database should be able to scale horizontally to handle increasing data storage requirements.

- Usability:
- 3.1. The user interface should be intuitive and easy to navigate.
- 3.2. The application should provide feedback to users for their actions (e.g., message sent confirmation).

- Reliability:
- 4.1. The application should be available and reliable, with minimal downtime for maintenance or updates.
- 4.2. Data integrity should be maintained to prevent loss of messages or user information.

- Compatibility:
- 5.1. The application should be compatible with modern web browsers (e.g., Chrome, Firefox, Safari).
- 5.2. The application should be responsive and functional across different devices (desktop, tablet, mobile)

These functional and non-functional requirements have been fulfilled by our project as we tried to achieve the above listed requirements in more percentage than the existing systems.

3.3 SOFTWARE REQUIREMENTS

- Operating system : Windows 10 and above.
- Language : Javascript,HTML,CSS
- Framework : Express.js,
- Libraries : React.js,Socket.IO
- Database : MongoDB
- Runtime Environment :Node.js

3.4 HARDWARE REQUIREMENTS

- RAM : 8GB and Higher
- Processor : Intel i5 and above
- Hard Disk : 256 GB or Higher

CHAPTER-4
SYSTEM DESIGN

4.1 PROPOSED METHOD:

The proposed algorithm To develop a chat application, the first step was to

design a database schema that could store user information, chat rooms, and messages. MongoDB, a document-oriented database, was chosen for this purpose due to its flexibility and ease of use. The schema was designed to have multiple collections, each responsible for storing different types of data.

On the client-side, React.js was chosen to create a responsive and intuitive user interface. React.js is a JavaScript-based framework that simplifies the development of web applications by providing a structured framework for creating dynamic views. It

allows developers to create complex user interfaces with ease, using reusable code and components.

The chat application required real-time message updates, which were achieved using socket.io, a JavaScript library for real-time web applications.

Socket.io enables bidirectional communication between the server and the client, allowing for realtime updates and notifications.User authentication was also an important feature of the chat application, and this was implemented using

JSON Web Tokens (JWT). JWT is a secure and easy-touse authentication mechanism that allows users to securely transmit information between parties.

Finally, Node.js was used to deploy the application on a cloud hosting service. Node.js is a platform built on the Chrome V8 JavaScript engine that allows

developers to run JavaScript code outside of a web browser on the server side, providing a powerful and efficient way to build server-side applications using

JavaScript. It allows Users to run JavaScript code on the server.

In conclusion, developing a chat application requires a combination of technologies and frameworks, each serving a specific purpose. The choice of technologies

depends on the specific requirements of the application, such as real-time message updates, user authentication, and scalability. MongoDB, Express.js, React.js,

socket.io, JWT, and Node.js were the technologies chosen for this particular chat

application, resulting in a robust and feature-rich application that met the requirements of the project.

4.2 CLASS DIAGRAM:

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes. It explains which class contains information.

A class diagram for a chat application using the MERN stack provides a visual representation of the system's architecture, outlining the classes and their relationships. In this example, two main classes are depicted: "User" and "Message." The "User" class encapsulates attributes such as userId, username, email, and password, along with operations for user authentication, registration, and logout. Conversely, the "Message" class contains attributes like messageId, senderId, content, and timestamp, with operations for sending and retrieving messages. Although not explicitly shown in the diagram, these classes would likely have associations or dependencies representing relationships, such as users sending messages. This diagram serves as a blueprint for developers, guiding the implementation of the chat application's functionality within the MERN stack. It illustrates the organization of classes and their interactions, facilitating the understanding and development of the application's underlying structure.

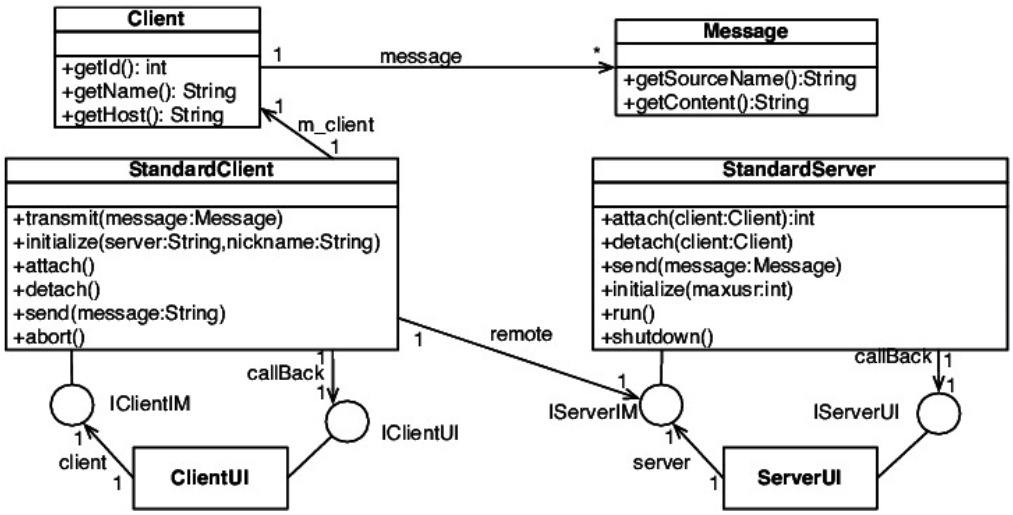

Fig 4.2.2 Class Diagram

4.3 USE CASE DIAGRAM:

A use case diagram in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. Its purpose is to present a graphical overview of the functionality provided by a system in terms of actors, their goals (represented as use cases), and any dependencies between those use cases. The main purpose of a use case diagram is to show what system functions are performed for which actor.

The use case diagram for a chat application using the MERN stack illustrates the interaction between users and the system, capturing key functionalities and features of the application. At the center of the diagram is the "User" actor, representing individuals engaging with the chat platform.

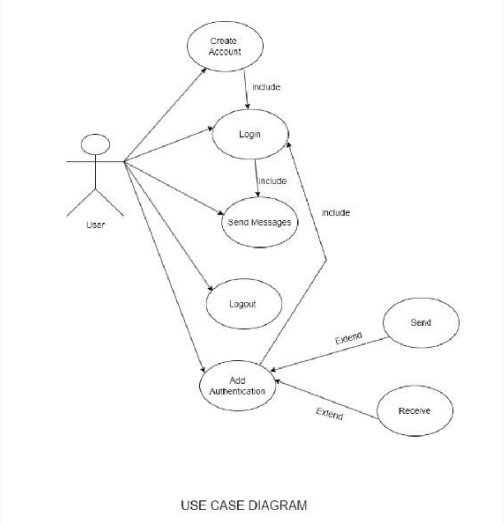

Fig 4.2.1 Use Case Diagram

4.4 ACTIVITY DIAGRAM:

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency.

An activity diagram for a chat application using the MERN stack illustrates the sequential flow of actions and interactions within the system, emphasizing the roles of different components and users. At the beginning of the diagram, the user initiates the login process to access the chat application. Upon entering their credentials, the system authenticates the user. If authentication is successful, the user proceeds to join a chat room; otherwise, they remain on the login screen. Once inside the chat room, the user engages in various activities while actively participating. These activities include sending messages, which triggers a sequence of actions: the server receives the message, saves it to the database for persistence, and then forwards it to the intended recipient(s). Additionally, if the user decides to leave the chat room, they exit the loop and are no longer considered active. This activity diagram provides a structured representation of the chat application's functionality, capturing the sequential flow of activities from user login to participation in chat room discussions, thereby illustrating the core features and interactions of the system

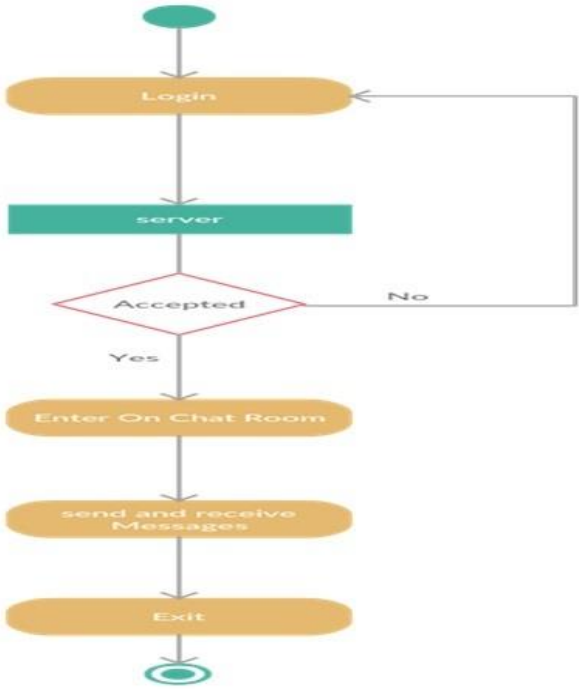

Fig 4.2.4(a) Activity Diagram

4.5 SEQUENCE DIAGRAM:

A sequence diagram in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams.

In the sequence diagram depicting the functionality of a chat application built with the MERN stack, the interactions between different components of the system are visualized as a series of steps. Initially, the user sends a message through the application's interface. This action triggers the client-side application, developed using React.js, to generate an HTTP POST request containing the message content and recipient(s). Upon receiving this request, the server-side application, powered by Node.js and Express.js, undertakes authentication and authorization checks to ensure that the user has the necessary permissions to send the message. Subsequently, if the validation is successful, the server saves the message to the MongoDB database for persistence. Concurrently, the server processes the message and dispatches it to the intended recipient(s) or group members. Once the message transmission is accomplished, the server issues a success response back to the client-side application, confirming the completion of the operation. This sequence of events illustrates the seamless flow of communication within the chat application architecture, demonstrating the coordinated actions of the user, client-side and server-side components, and the database in facilitating message delivery.

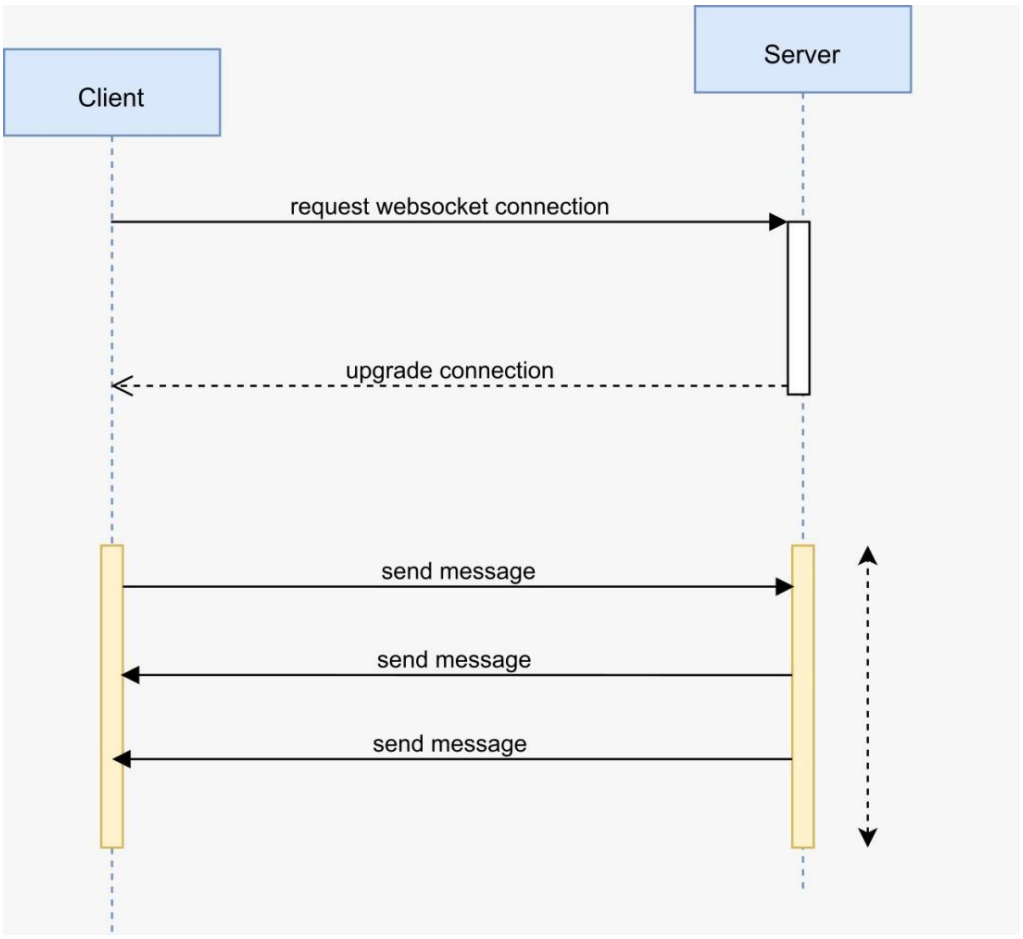

Fig 4.2.3 Sequence Diagram

4.6 SYSTEM ARCHITECTURE:

The architecture of a chat application utilizing the MERN stack is structured around a client-server model, where distinct components handle different aspects of the application's functionality.

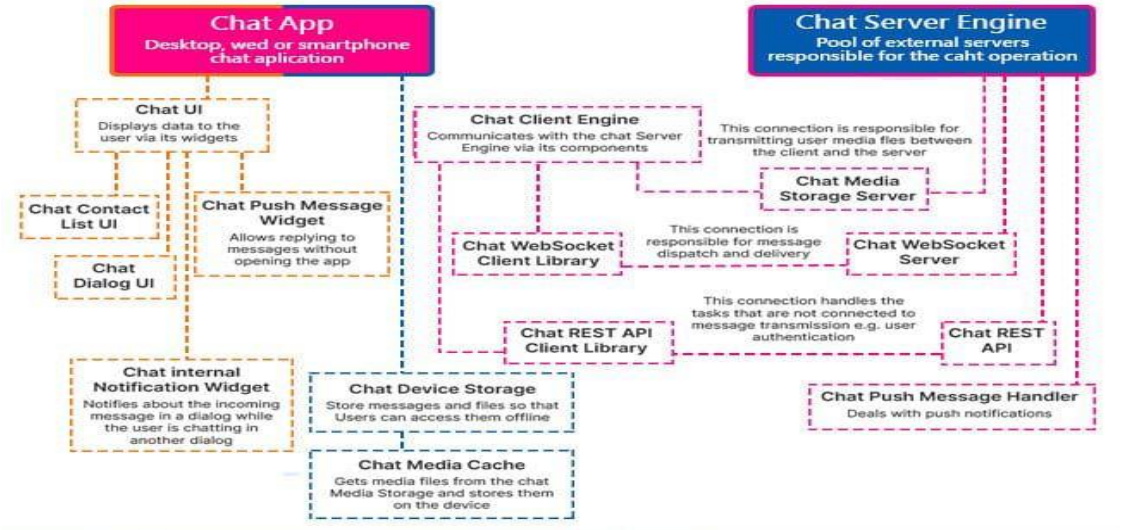

Fig 4.0: System Architecture

A. SERVER

A server may be a computer dedicated to running a server application. Organizations have dedicated computer for server application which has to be maintained periodically and has to be monitored continuously for traffic loads would never let them go down which affects the company's revenue.

Most organizations have a separate monitoring system to keep an eye over their server so that they can find their server downtime before its clients. These server computers accept clients over network connections that are requested.

The server responds back by sending responses being requested. There are many different server applications that vary based on their dedicated work. Some are involved for accepting requests and performing all dedicated works

like business application servers while others are just to bypass the request like a proxy server. These server computers must have a faster Central processing unit, faster and more plentiful RAM, and bigger hard disc drive. More obvious distinctions include redundancy in power supplies, network connections, and RAID also as Modular design.

B. CLIENT

A client is a software application code or a system that requests another application that is running on dedicated machine called Server. These clients need not be connected to the server through wired communication. Wireless communication takes place in this process. Client with a network connection can send a request to the server.

4.7 TECHNOLOGY DESCRIPTION

4.7.1 TECHNOLOGY STACK :

The technology stack used for the chat application using MERN project includes:

4.7 Node.js:A JavaScript runtime environment that allows running JavaScript code on the server-side.

4.8 Express.js:A fast and minimalist web application framework for Node.js that helps in building web applications and APIs.

4.9 React.js:A JavaScript library for building user interfaces, providing a responsive and efficient way to create interactive Ul components.

4.10 MongoDB:A NoSQL database that provides a flexible and scalable solution for storing and retrieving data.

4.11 Socket.IO:A library that enables real-time, bidirectional and event-based communication between the browser and the server.

4.12 HTML:The standard markup language for creating web pages and applications.

4.13 CSS:A stylesheet language used for describing the look and formatting of a document written in HTML.

4.14 JavaScript:A programming language that enables interactive and dynamic behavior on web pages.

CHAPTER-5
IMPLEMENTATION AND TESTING

5.1 IMPLEMENTATION:

5.1.1 Home Page Code

```
 import {
  Box,
  Container,
  Tab,
  TabList,
  TabPanel,
  TabPanels,
  Tabs,
  Text,
} from "@chakra-ui/react";
import { useEffect } from "react";
import { useHistory } from "react-router";
import Login from "../components/Authentication/Login";
import Signup from "../components/Authentication/Signup";

function Homepage() {
  const history = useHistory();

  useEffect(() => {
    const user = JSON.parse(localStorage.getItem("userInfo"));

    if (user) history.push("/chats");
  }, [history]);

  return (
    <Container maxW="xl" centerContent>
      <Box
        d="flex"
```

```
        justifyContent="center"
        p={3}
        bg="white"
        w="100%"
        m="40px 0 15px 0"
        borderRadius="lg"
        borderWidth="1px"
      >
        <Text fontSize="4xl" fontFamily="Work sans">
          Talk-A-Tive
        </Text>
      </Box>
      <Box bg="white" w="100%" p={4} borderRadius="lg" borderWidth="1px">
        <Tabs isFitted variant="soft-rounded">
          <TabList mb="1em">
            <Tab>Login</Tab>
            <Tab>Sign Up</Tab>
          </TabList>
          <TabPanels>
            <TabPanel>
              <Login />
            </TabPanel>
            <TabPanel>
              <Signup />
            </TabPanel>
          </TabPanels>
        </Tabs>
      </Box>
    </Container>
  );
}
export default Homepage;
```

5.1.2 Server.js

```
const express = require("express");
const connectDB = require("./config/db");
const dotenv = require("dotenv");
const userRoutes = require("./routes/userRoutes");
const chatRoutes = require("./routes/chatRoutes");
const messageRoutes = require("./routes/messageRoutes");
const { notFound, errorHandler } = require("./middleware/errorMiddleware");
const path = require("path");

dotenv.config();
connectDB();
const app = express();

app.use(express.json()); // to accept json data

// app.get("/", (req, res) => {
//   res.send("API Running!");
// });

app.use("/api/user", userRoutes);
app.use("/api/chat", chatRoutes);
app.use("/api/message", messageRoutes);

// --------------------------deployment------------------------------

const __dirname1 = path.resolve();

if (process.env.NODE_ENV === "production") {
  app.use(express.static(path.join(__dirname1, "/frontend/build")));

  app.get("*", (req, res) =>
    res.sendFile(path.resolve(__dirname1, "frontend", "build", "index.html"))
```

```
  );
} else {
  app.get("/", (req, res) => {
    res.send("API is running..");
  });
}

// --------------------------deployment------------------------------

// Error Handling middlewares
app.use(notFound);
app.use(errorHandler);

const PORT = process.env.PORT;

const server = app.listen(
  PORT,
  console.log(Server running on PORT ${PORT}....yellow.bold)
);

const io = require("socket.io")(server, {
  pingTimeout: 60000,
  cors: {
    origin: "http://localhost:3000",
    // credentials: true,
  },
});

io.on("connection", (socket) => {
  console.log("Connected to socket.io");
  socket.on("setup", (userData) => {
    socket.join(userData._id);
    socket.emit("connected");
  });
```

```
  socket.on("join chat", (room) => {
    socket.join(room);
    console.log("User Joined Room: " + room);
  });
  socket.on("typing", (room) => socket.in(room).emit("typing"));
  socket.on("stop typing", (room) => socket.in(room).emit("stop typing"));

  socket.on("new message", (newMessageRecieved) => {
    var chat = newMessageRecieved.chat;

    if (!chat.users) return console.log("chat.users not defined");

    chat.users.forEach((user) => {
      if (user._id == newMessageRecieved.sender._id) return;

      socket.in(user._id).emit("message recieved", newMessageRecieved);
    });
  });

  socket.off("setup", () => {
    console.log("USER DISCONNECTED");
    socket.leave(userData._id);
  });
});
```

5.1.3 Login.js

```
import { Button } from "@chakra-ui/button";
import { FormControl, FormLabel } from "@chakra-ui/form-control";
import { Input, InputGroup, InputRightElement } from "@chakra-ui/input";
import { VStack } from "@chakra-ui/layout";
import { useState } from "react";
import axios from "axios";
```

```
import { useToast } from "@chakra-ui/react";
import { useHistory } from "react-router-dom";
import { ChatState } from "../../Context/ChatProvider";

const Login = () => {
  const [show, setShow] = useState(false);
  const handleClick = () => setShow(!show);
  const toast = useToast();
  const [email, setEmail] = useState();
  const [password, setPassword] = useState();
  const [loading, setLoading] = useState(false);

  const history = useHistory();
  const { setUser } = ChatState();

  const submitHandler = async () => {
    setLoading(true);
    if (!email || !password) {
      toast({
        title: "Please Fill all the Feilds",
        status: "warning",
        duration: 5000,
        isClosable: true,
        position: "bottom",
      });
      setLoading(false);
      return;
    }

    try {
      const config = {
        headers: {
          "Content-type": "application/json",
        },
```

```
    };

    const { data } = await axios.post(
      "/api/user/login",
      { email, password },
      config
    );

    toast({
      title: "Login Successful",
      status: "success",
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });
    setUser(data);
    localStorage.setItem("userInfo", JSON.stringify(data));
    setLoading(false);
    history.push("/chats");
  } catch (error) {
    toast({
      title: "Error Occured!",
      description: error.response.data.message,
      status: "error",
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });
    setLoading(false);
  }
};

return (
  <VStack spacing="10px">
```

```
<FormControl id="email" isRequired>
  <FormLabel>Email Address</FormLabel>
  <Input
    value={email}
    type="email"
    placeholder="Enter Your Email Address"
    onChange={(e) => setEmail(e.target.value)}
  />
</FormControl>
<FormControl id="password" isRequired>
  <FormLabel>Password</FormLabel>
  <InputGroup size="md">
    <Input
      value={password}
      onChange={(e) => setPassword(e.target.value)}
      type={show ? "text" : "password"}
      placeholder="Enter password"
    />
    <InputRightElement width="4.5rem">
      <Button h="1.75rem" size="sm" onClick={handleClick}>
        {show ? "Hide" : "Show"}
      </Button>
    </InputRightElement>
  </InputGroup>
</FormControl>
<Button
  colorScheme="blue"
  width="100%"
  style={{ marginTop: 15 }}
  onClick={submitHandler}
  isLoading={loading}
>
  Login
</Button>
```

```
      <Button
        variant="solid"
        colorScheme="red"
        width="100%"
        onClick={() => {
          setEmail("guest@example.com");
          setPassword("123456");
        }}
      >
        Get Guest User Credentials
      </Button>
    </VStack>
  );
};

export default Login;
```

5.1.4 Signup.js

```
import { Button } from "@chakra-ui/button";
import { FormControl, FormLabel } from "@chakra-ui/form-control";
import { Input, InputGroup, InputRightElement } from "@chakra-ui/input";
import { VStack } from "@chakra-ui/layout";
import { useToast } from "@chakra-ui/toast";
import axios from "axios";
import { useState } from "react";
import { useHistory } from "react-router";

const Signup = () => {
  const [show, setShow] = useState(false);
  const handleClick = () => setShow(!show);
  const toast = useToast();
  const history = useHistory();

  const [name, setName] = useState();
  const [email, setEmail] = useState();
  const [confirmpassword, setConfirmpassword] = useState();
```

```
const [password, setPassword] = useState();
const [pic, setPic] = useState();
const [picLoading, setPicLoading] = useState(false);

const submitHandler = async () => {
  localStorage.removeItem("userInfo");
  setPicLoading(true);
  if (!name || !email || !password || !confirmpassword) {
    toast({
      title: "Please Fill all the Feilds",
      status: "warning",
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });
    setPicLoading(false);
    return;
  }
  if (password !== confirmpassword) {
    toast({
      title: "Passwords Do Not Match",
      status: "warning",
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });
    return;
  }
  console.log(name, email, password, pic);
  try {
    const config = {
      headers: {
        "Content-type": "application/json",
      },
    };
    const { data } = await axios.post(
      "/api/user",
      {
        name,
        email,
```

```
        password,
        pic,
      },
      config
    );
    console.log(data);
    toast({
      title: "Registration Successful",
      status: "success",
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });



    window.location.reload();
    localStorage.setItem("userInfo", JSON.stringify(data));
    setPicLoading(false);
    history.push("/chats");

  } catch (error) {
    toast({
      title: "Error Occured!",
      description: error.response.data.message,
      status: "error",
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });
    setPicLoading(false);
  }
};

const postDetails = (pics) => {
  setPicLoading(true);
  if (pics === undefined) {
    toast({
      title: "Please Select an Image!",
      status: "warning",
```

```
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });
    return;
  }
  console.log(pics);
  if (pics.type === "image/jpeg" || pics.type === "image/png") {
    const data = new FormData();
    data.append("file", pics);
    data.append("upload_preset", "chat-app");
    data.append("cloud_name", "rohithproj");
    fetch("https://api.cloudinary.com/v1_1/rohithdonikena/image/upload", {
      method: "post",
      body: data,
    })
      .then((res) => res.json())
      .then((data) => {
        setPic(data.url.toString());
        console.log(data.url.toString());
        setPicLoading(false);
      })
      .catch((err) => {
        console.log(err);
        setPicLoading(false);
      });
  } else {
    toast({
      title: "Please Select an Image!",
      status: "warning",
      duration: 5000,
      isClosable: true,
      position: "bottom",
    });
    setPicLoading(false);
    return;
  }
};

return (
```

```
<VStack spacing="5px">
  <FormControl id="first-name" isRequired>
    <FormLabel>Name</FormLabel>
    <Input
      placeholder="Enter Your Name"
      onChange={(e) => setName(e.target.value)}
    />
  </FormControl>
  <FormControl id="email" isRequired>
    <FormLabel>Email Address</FormLabel>
    <Input
      type="email"
      placeholder="Enter Your Email Address"
      onChange={(e) => setEmail(e.target.value)}
    />
  </FormControl>
  <FormControl id="password" isRequired>
    <FormLabel>Password</FormLabel>
    <InputGroup size="md">
      <Input
        type={show ? "text" : "password"}
        placeholder="Enter Password"
        onChange={(e) => setPassword(e.target.value)}
      />
      <InputRightElement width="4.5rem">
        <Button h="1.75rem" size="sm" onClick={handleClick}>
          {show ? "Hide" : "Show"}
        </Button>
      </InputRightElement>
    </InputGroup>
  </FormControl>
  <FormControl id="password" isRequired>
    <FormLabel>Confirm Password</FormLabel>
    <InputGroup size="md">
      <Input
        type={show ? "text" : "password"}
        placeholder="Confirm password"
        onChange={(e) => setConfirmpassword(e.target.value)}
      />
      <InputRightElement width="4.5rem">
```

```
            <Button h="1.75rem" size="sm" onClick={handleClick}>
              {show ? "Hide" : "Show"}
            </Button>
          </InputRightElement>
        </InputGroup>
      </FormControl>
      <FormControl id="pic">
        <FormLabel>Upload your Picture</FormLabel>
        <Input
          type="file"
          p={1.5}
          accept="image/*"
          onChange={(e) => postDetails(e.target.files[0])}
        />
      </FormControl>
      <Button
        colorScheme="blue"
        width="100%"
        style={{ marginTop: 15 }}
        onClick={submitHandler}
        isLoading={picLoading}
      >
        Sign Up
      </Button>
    </VStack>
  );
};

export default Signup;
```

5.1.5 db.js

```
const mongoose = require("mongoose");
const colors = require("colors");

const connectDB = async () => {
  try {
```

```
    const conn = await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });

    console.log(MongoDB Connected: ${conn.connection.host}.cyan.underline);
  } catch (error) {
    console.error(Error: ${error.message}.red.bold);
    process.exit(1); // Exit with a non-zero status code to indicate an error
  }
};
module.exports = connectDB;
```

5.1.6 SingleChat.js

```
import { FormControl } from "@chakra-ui/form-control";
import { Input } from "@chakra-ui/input";
import { Box, Text } from "@chakra-ui/layout";
import "./styles.css";
import { IconButton, Spinner, useToast } from "@chakra-ui/react";
import { getSender, getSenderFull } from "../config/ChatLogics";
import { useEffect, useState } from "react";
import axios from "axios";
import { ArrowBackIcon } from "@chakra-ui/icons";
import ProfileModal from "./miscellaneous/ProfileModal";
import ScrollableChat from "./ScrollableChat";
import Lottie from "react-lottie";
import animationData from "../animations/typing.json";

import io from "socket.io-client";
import UpdateGroupChatModal from "./miscellaneous/UpdateGroupChatModal";
import { ChatState } from "../Context/ChatProvider";
const ENDPOINT = "http://localhost:5000"; // "https://talk-a-tive.herokuapp.com"; -> After deployment
var socket, selectedChatCompare;
const SingleChat = ({ fetchAgain, setFetchAgain }) => {
const [messages, setMessages] = useState([]);
const [loading, setLoading] = useState(false);
const [newMessage, setNewMessage] = useState("");
const [socketConnected, setSocketConnected] = useState(false);
const [typing, setTyping] = useState(false);
```

```
const [istyping, setIsTyping] = useState(false);
const toast = useToast();

const defaultOptions = {
  loop: true,
  autoplay: true,
  animationData: animationData,
  rendererSettings: {
    preserveAspectRatio: "xMidYMid slice",
  },
};
const { selectedChat, setSelectedChat, user, notification, setNotification } =
  ChatState();

const fetchMessages = async () => {
  if (!selectedChat) return;

  try {
    const config = {
      headers: {
        Authorization: Bearer ${user.token},
      },
    };

    setLoading(true);

    const { data } = await axios.get(
      /api/message/${selectedChat._id},
      config
    );
    setMessages(data);
    setLoading(false);

    socket.emit("join chat", selectedChat._id);
  } catch (error) {
    toast({
      title: "Error Occured!",
      description: "Failed to Load the Messages",
      status: "error",
      duration: 5000,
```

```
      isClosable: true,
      position: "bottom",
    });
  }
};

const sendMessage = async (event) => {
  if (event.key === "Enter" && newMessage) {
    socket.emit("stop typing", selectedChat._id);
    try {
      const config = {
        headers: {
          "Content-type": "application/json",
          Authorization: Bearer ${user.token},
        },
      };
      setNewMessage("");
      const { data } = await axios.post(
        "/api/message",
        {
          content: newMessage,
          chatId: selectedChat,
        },
        config
      );
      socket.emit("new message", data);
      setMessages([...messages, data]);
    } catch (error) {
      toast({
        title: "Error Occured!",
        description: "Failed to send the Message",
        status: "error",
        duration: 5000,
        isClosable: true,
        position: "bottom",
      });
    }
  }
};
```

```
useEffect(() => {
  socket = io(ENDPOINT);
  socket.emit("setup", user);
  socket.on("connected", () => setSocketConnected(true));
  socket.on("typing", () => setIsTyping(true));
  socket.on("stop typing", () => setIsTyping(false));

  // eslint-disable-next-line
}, []);

useEffect(() => {
  fetchMessages();

  selectedChatCompare = selectedChat;
  // eslint-disable-next-line
}, [selectedChat]);

useEffect(() => {
  socket.on("message recieved", (newMessageRecieved) => {
    if (
      !selectedChatCompare || // if chat is not selected or doesn't match current chat
      selectedChatCompare._id !== newMessageRecieved.chat._id
    ) {
      if (!notification.includes(newMessageRecieved)) {
        setNotification([newMessageRecieved, ...notification]);
        setFetchAgain(!fetchAgain);
      }
    } else {
      setMessages([...messages, newMessageRecieved]);
    }
  });
});

const typingHandler = (e) => {
  setNewMessage(e.target.value);

  if (!socketConnected) return;

  if (!typing) {
    setTyping(true);
```

```
    socket.emit("typing", selectedChat._id);
  }
  let lastTypingTime = new Date().getTime();
  var timerLength = 3000;
  setTimeout(() => {
    var timeNow = new Date().getTime();
    var timeDiff = timeNow - lastTypingTime;
    if (timeDiff >= timerLength && typing) {
      socket.emit("stop typing", selectedChat._id);
      setTyping(false);
    }
  }, timerLength);
};

return (
  <>
    {selectedChat ? (
      <>
        <Text
          fontSize={{ base: "28px", md: "30px" }}
          pb={3}
          px={2}
          w="100%"
          fontFamily="Work sans"
          d="flex"
          justifyContent={{ base: "space-between" }}
          alignItems="center"
        >
          <IconButton
            d={{ base: "flex", md: "none" }}
            icon={<ArrowBackIcon />}
            onClick={() => setSelectedChat("")}
          />
          {messages &&
            (!selectedChat.isGroupChat ? (
              <>
                {getSender(user, selectedChat.users)}
                <ProfileModal
                  user={getSenderFull(user, selectedChat.users)}
                />
```

```
      </>
    ) : (
      <>
        {selectedChat.chatName.toUpperCase()}
        <UpdateGroupChatModal
          fetchMessages={fetchMessages}
          fetchAgain={fetchAgain}
          setFetchAgain={setFetchAgain}
        />
      </>
    ))}
</Text>
<Box
  d="flex"
  flexDir="column"
  justifyContent="flex-end"
  p={3}
  bg="#E8E8E8"
  w="100%"
  h="100%"
  borderRadius="lg"
  overflowY="hidden"
>
  {loading ? (
    <Spinner
      size="xl"
      w={20}
      h={20}
      alignSelf="center"
      margin="auto"
    />
  ) : (
    <div className="messages">
      <ScrollableChat messages={messages} />
    </div>
  )}

  <FormControl
    onKeyDown={sendMessage}
    id="first-name"
```

```
            isRequired
            mt={3}
          >
            {istyping ? (
              <div>
                <Lottie
                  options={defaultOptions}
                  // height={50}
                  width={70}
                  style={{ marginBottom: 15, marginLeft: 0 }}
                />
              </div>
            ) : (
              <></>
            )}
            <Input
              variant="filled"
              bg="#E0E0E0"
              placeholder="Enter a message.."
              value={newMessage}
              onChange={typingHandler}
            />
          </FormControl>
        </Box>
      </>
    ) : (
      // to get socket.io on same page
      <Box d="flex" alignItems="center" justifyContent="center" h="100%">
        <Text fontSize="3xl" pb={3} fontFamily="Work sans">
          Click on a user to start chatting
        </Text>
      </Box>
    )}
  </>
 );
};

export default SingleChat;
```

5.2 Results and Discussions:

Output screens

Output Screens of various functionalities in our application are shown over here along with the description.

Home Page:

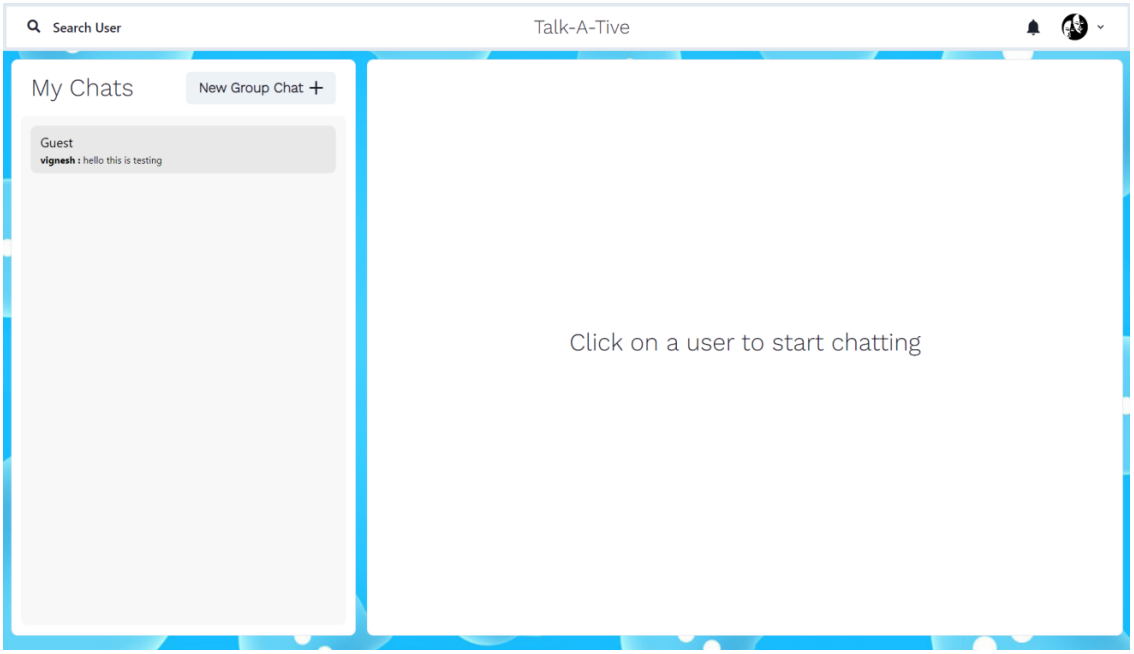

Fig 5.3(a) Home Page

Sign Up /Login Page:

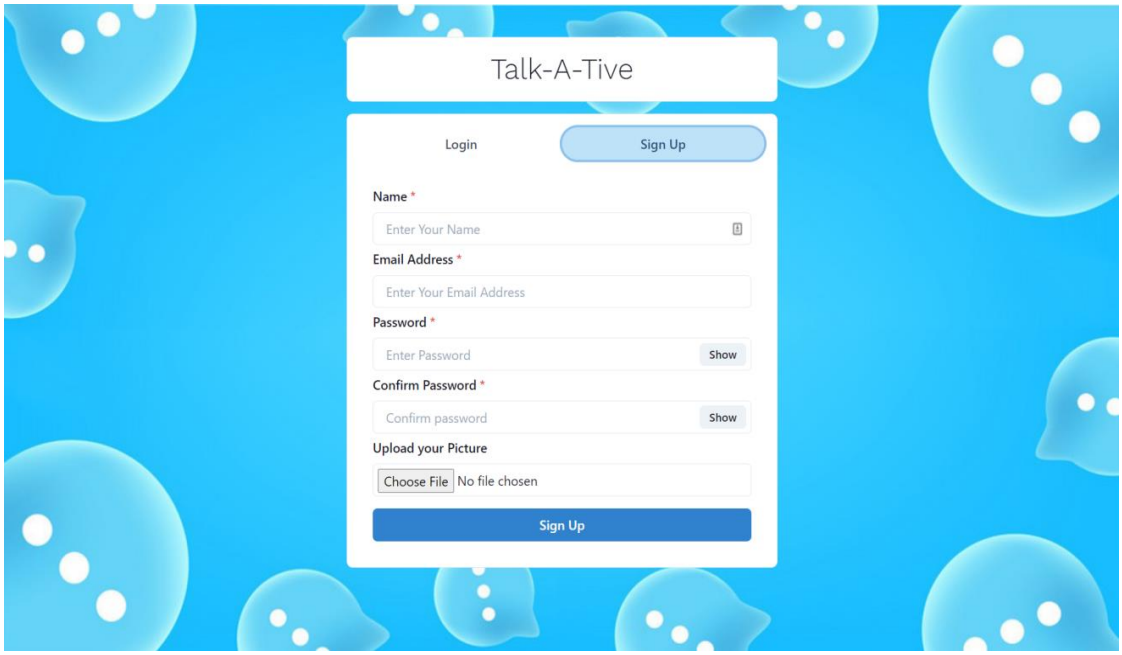

Fig 5.3(b) SignUp

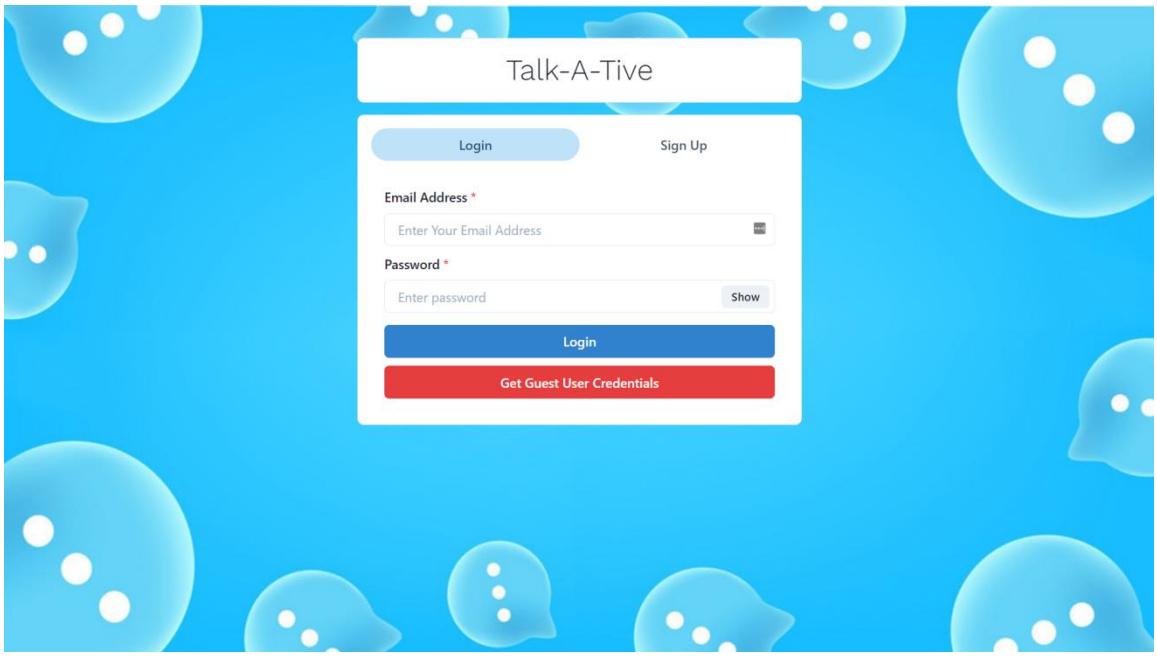

Fig 5.3(c) Login

Single Chat / Group Chat:

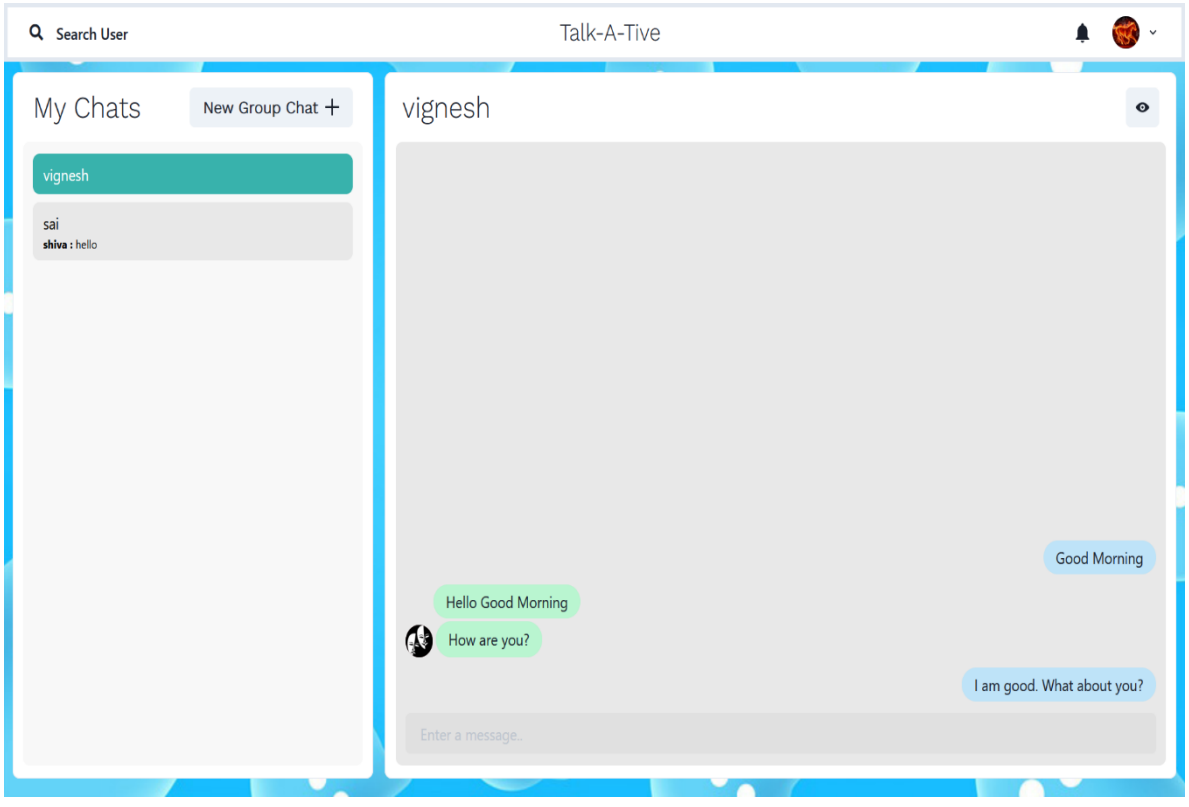

Fig 5.3(d) Single User Chat

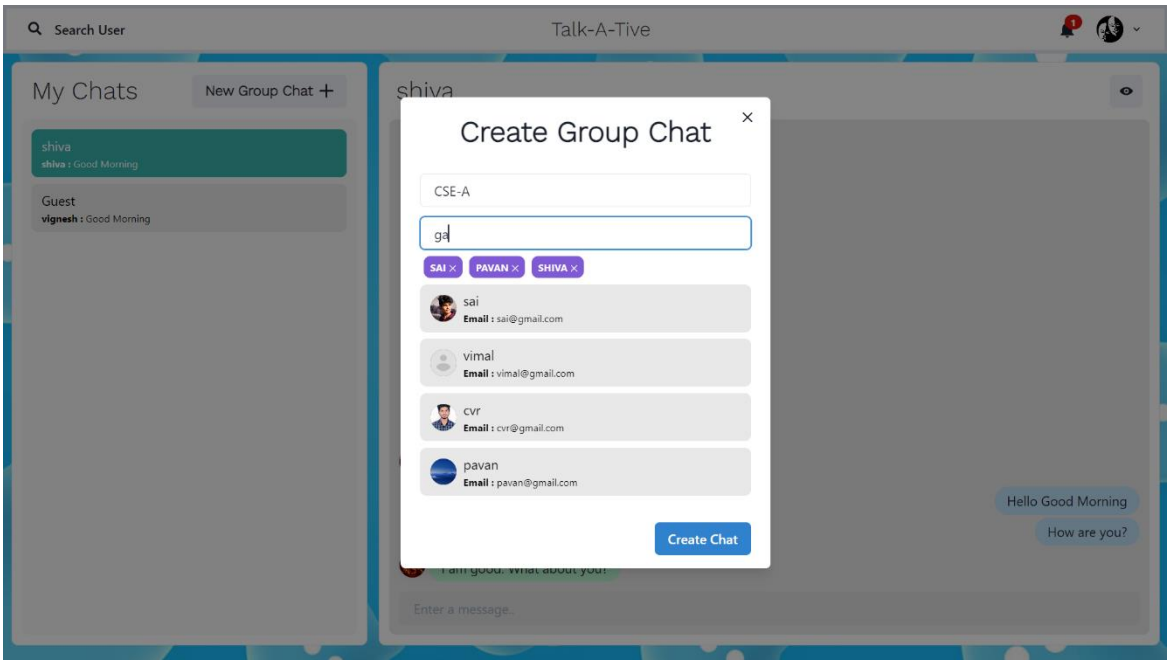

Fig 5.3(e) Group Chat Creation

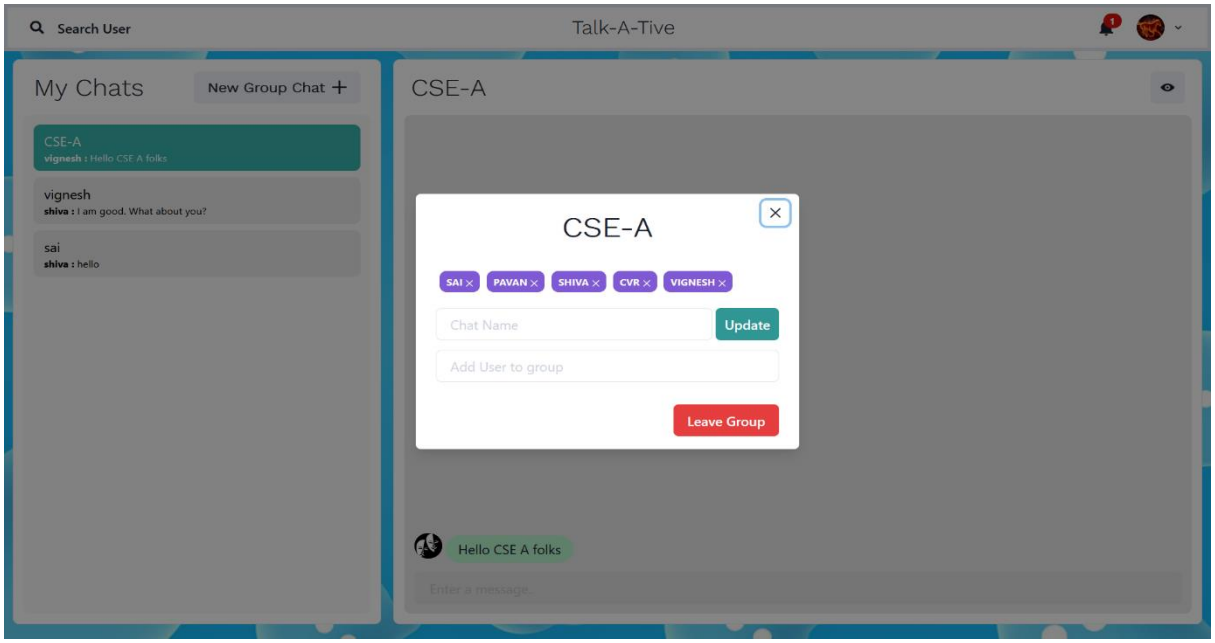

Fig 5.3(e) Group Chat Page

6.1 CONCLUSION

In conclusion, developing a chat application using the MERN stack has been a challenging but rewarding experience. The use of MongoDB, Express, React, and Node.js provides a powerful and flexible framework for creating real-time communication and collaboration solutions that can be tailored to meet a wide range of use cases and industries.

The app has been designed with user experience in mind, and features such as real-time message updates and user authentication have been implemented to provide a seamless and secure communication experience. The scalability and robustness of the MERN stack ensure that the app can handle a high volume of users and messages without compromising on performance.

Going forward, there are many opportunities for further development and improvement of the app. This includes adding new features such as video and voice chat, integrating with other applications and platforms, and enhancing the user interface to make it more intuitive and user-friendly. Overall, the chat app developed using the MERN stack represents a significant achievement in the field of real-time communication and collaboration, and has the potential to revolutionize the way people connect and communicate online.

6.2 FUTURE SCOPE

The future scope of a project involving a Chat Application using the MERN stack, which combines MongoDB, Express.js, React.js, and Node.js, is promising and involves several key areas of development and enhancement. The demand for MERN stack developers is expected to rise steadily due to the advantages of building scalable and interactive web applications. MERN stack development focuses on creating highly interactive and responsive user interfaces to deliver seamless and immersive user experiences. Additionally, the MERN stack, powered by Node.js, is well-suited for developing real-time applications like collaborative tools and chat applications. Furthermore, the MERN stack is ideal for building IoT applications that require real-time data processing and bidirectional communication between devices and servers. With React Native's compatibility with the MERN stack, developers can efficiently build native mobile apps for iOS and Android using JavaScript. The MERN stack is also suitable for developing PWAs that offer native-like experiences through optimizing performance and enhancing offline capabilities. Moreover, MERN stack development will adopt serverless practices like AWS Lambda and Firebase Functions to build cost-effective, scalable, and highly available applications. In conclusion, the future of Chat Application development using the MERN stack looks bright, offering opportunities for developers to create innovative, user-friendly, and scalable web applications across various domains.

REFERENCES

1. Masiello Eric. Mastering React Native. January 11; 2017. This book is a comprehensive guide to building mobile applications using React Native.
2. Naimul Islam Naim. ReactJS: An Open-Source JavaScript library for front-end development. Metropolia University of Applied Sciences. This article provides an overview of ReactJS and its key features for front-end web development.
3. Stefanov Stoyan, editor. React: Up and Running: Building web Applications. First Edition; 2016. This book is a beginner-friendly introduction to React, covering its core concepts and providing practical examples for building web applications.
4. Horton Adam, Vice Ryan. Mastering React; February 23; 2016. This book provides a comprehensive guide to React, covering its core concepts, practical examples, and advanced techniques for building complex applications.
5. Alex Kondov. Express Architecture Review. This article provides a review of the architecture of Express.js, a popular web framework for building Node.js applications.
6. Express.js documentation. This documentation provides a comprehensive guide to building web applications using Express.js.
7. Adam Horton. Node.js vs Python: What to Choose. This article provides a comparison of Node.js and Python for web development, highlighting their strengths and weaknesses.
8. Node.js documentation. This documentation provides a comprehensive guide to building server- side applications using Node.js.
9. MongoDB documentation. This documentation provides a comprehensive guide to using MongoDB, a popular NoSQL database for building web applications.
10. The paper by Lakshmi Prasanna Chitra and Ravikanth Satapathy aims to compare the performance of Node.js and traditional web servers, specifically Internet Information Services (IIS), in optimizing web application development.The authors conducted various tests to evaluate the performance of both platforms and determine which one is better for developing high-performance web applications.
11. Guru99. React vs Angular: Key Differences. This article provides a comparison of React and Angular, two popular front-end frameworks for building web applications