A MINOR PROJECT

REPORT ON

CHATSPHERE: CHAT APPLICATION WITH AUTO TRANSLATION

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENT FOR THE AWARD OF THE DEGREE

OF

BACHELOR OF TECHNOLOGY

In

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Submitted by

ABDUL AYAN QALIQ	**22RE1A0501**
MOHD IMADUDDIN	**22RE1A0537**
MOHD KHAJA TANVEERUDDIN	**22RE1A0538**
OMAR SOHEL	**22RE1A0549**

Under the guidance of

Dr. Reddy Sekhar K

Asst. Professor (Sr.)

MNR COLLEGE OF ENGINEERING & TECHNOLOGY

(Approved by AICTE and Affiliated to JNT University, Hyderabad)

MNR NAGAR, FASALWADI, SANGAREDDY(dist) - 502001

2024-2025

MNR COLLEGE OF ENGINEERING & TECHNOLOGY

(Approved by AICTE and Affiliated to JNT University, Hyderabad)

MNR NAGAR, FASALWADI, SANGAREDDY(dist) - 502001

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

*This is to certify that the project work entitled "***CHATSPHERE: CHAT APPLICATION WITH AUTO TRANSLATION***" submitted by "***ABDUL AYAN 22RE1A0501, MD. IMADUDDIN 22RE1A0537, MK. TANVEERUDDIN 22RE1A0538, OMAR SOHEL 22RE1A0549***" in partial fulfilment of the requirements for the award of the degree of* **Bachelor of Technology** *in* **COMPUTER SCIENCE AND ENGINEERING**, *by* **Jawaharlal Nehru Technological University Hyderabad during the academic year 2024-2025** *is a Bonafide record of the work carried out under my guidance and supervision* at **MNR College of Engineering & Technology, Sangareddy.**

Internal guide	**Head of the department**
Dr. Reddy Sekhar K	**Mr. T. Ravi Kiran Kumar**
Assistant professor (Sr.)	**Assistant professor**

External Examiner

ACKNOWLEDGEMENT

We wish to take this opportunity to express my deepest appreciation to all those who provided for our team the possibility to complete this report. A special gratitude we give to our beloved **Vice-Chairman, Sri. M. Ravi Varma** extended his co-operation in various ways during project work. We express our appreciation to the Principal, **Dr. E.L. NAGESH** MNRCET who permitted us to perform this project work report and organized everything. We are thankful to Professor **Mr. T. RAVI KIRAN KUMAR**, Head of the Department, MNR College of Engineering and Technology, who encouraged our team at this venture. We would like to articulate my profound gratitude and indebtedness to my project guide **Dr. Reddy Sekhar K, Sr. Assistant professor** who has always been a constant motivation and guiding factor throughout the project time in and out as well. It has been a great pleasure for our team to get an opportunity to work under her and complete the project successfully. We would like to thank to that all the staff members for their support and help throughout this process. Last but not the least we would like to thank my parents and my friends who were always a source of constant inspiration for our team all the way which we travelled to shape my career and life. It is with great respect and admiration; we dedicate this report to all of them.

In all sincerity,

ABDUL AYAN QALIQ	**22RE1A0501**
MOHD IMADUDDIN	**22RE1A0537**
MOHD KHAJA TANVEERUDDIN	**22RE1A0538**
OMAR SOHEL	**22RE1A0549**

DECLARATION

We hereby declare that the Dissertation entitled **"CHATSPHERE: CHAT APPLICATION WITH AUTO TRANSLATION** " which is being submitted in partial fulfilment of the requirement of the course leading to the award of the "**B. Tech** in **CSE**" in **MNR College of Engineering & Technology.** The result of this dissertation is carried out by our team, under the guidance and supervision of **Dr. Reddy Sekhar K, Sr. Assistant Professor**. We further declared that we have not previously submitted this to any other institution/university for any other degree.

ABDUL AYAN QALIQ	22RE1A0501
MOHD IMADUDDIN	22RE1A0537
MOHD KHAJA TANVEERUDDIN	22RE1A0538
OMAR SOHEL	22RE1A0549

TABLE OF CONTENTS:

1. ABSTRACT

In this paper, we proposed a real-time multilingual chat application named ChatSphere, developed using modern web technologies. The primary focus of this project is to enable seamless communication between users speaking different languages through automatic message translation. This system helps users communicate by translating messages into the recipient's preferred language in real-time, thus removing language barriers.

We have implemented support for commonly used languages across different regions, making this system globally accessible. The application performs efficiently, offering both real-time messaging and accurate translation, and can serve as a foundation for multilingual communication platforms.

2. INTRODUCTION

According to Ethnologue and other global linguistic databases, there are over 7,000 languages spoken across the world today. With globalization, cross-border collaboration, and international communication growing at an unprecedented rate, language diversity has become both a strength and a challenge. While countries interact through trade, education, tourism, and remote work, communication barriers often arise due to language differences. In such scenarios, effective and real-time translation becomes crucial.

Nowadays, people from different linguistic backgrounds interact frequently through digital platforms, including chat applications. However, most conventional chat applications lack integrated translation capabilities, making it difficult for users who do not share a common language to communicate seamlessly. As a result, users often rely on external tools or manual translation, which can disrupt the flow of conversation and reduce communication efficiency.

3. LITERATURE SURVEY

1. Real-Time Communication with WebSockets

WebSockets technology provides a persistent, full-duplex communication channel between the client and server over a single TCP connection. Unlike traditional HTTP request-response methods, WebSockets enable low-latency, real-time data exchange, which is critical for instant messaging and chat applications. According to Fette and Melnikov, the WebSocket protocol overcomes HTTP limitations by maintaining an open connection, reducing network overhead and improving responsiveness. This real-time communication capability forms the backbone of many modern chat applications, facilitating instant message delivery and receipt.

2. Multilingual Neural Machine Translation (NMT)

Neural Machine Translation (NMT) leverages deep learning models to produce fluent and context-aware translations. Multilingual NMT models handle multiple language pairs within a single architecture, enhancing translation quality, especially for low-resource languages. Dabre et al. survey various MNMT approaches, including shared attention mechanisms and transfer learning. Integration of such models into chat systems removes language barriers in real-time. The use of Microsoft Azure Translator API in ChatSphere is a practical application of these advancements.

3. Secure Authentication with JSON Web Tokens (JWT)

JWTs provide a compact, self-contained way to securely transmit information between parties. They are widely used in RESTful API authentication. Kumar and Sharma demonstrate that JWT combined with HMAC SHA-256 provides robust security by ensuring data integrity and authenticity. Implementing token expiration and refresh mechanisms mitigates risks such as token theft or replay attacks. ChatSphere uses JWT tokens to securely authorize users without maintaining server-side session state, enhancing scalability.

4. State Management in React Applications: Redux vs. Zustand

State management is essential for complex React apps. While Redux has been the traditional choice, Zustand offers a simpler API and better performance. Lee and Park's comparative analysis shows Zustand reduces boilerplate, integrates well with React's concurrent rendering, and is ideal for real-time applications like ChatSphere, where quick UI updates are critical.

5. Cloud-Based Media Management with Cloudinary

Cloudinary offers cloud-based image and video management, including uploading, transformation, optimization, and CDN delivery. Brown highlights how Cloudinary simplifies media handling by automating format conversion and resizing, reducing bandwidth, and improving load times. ChatSphere leverages Cloudinary to enable efficient media sharing in chats without burdening application servers.

4. EXISTING SYSTEM

In today's globalized world, communication across different languages is a frequent necessity. People from various linguistic backgrounds interact through messaging platforms for personal, educational, and business purposes. Most popular chat applications support multi-language communication but usually lack seamless, real-time automatic translation, which can hinder smooth conversations between users speaking different languages.

Existing chat platforms generally rely on manual translation or external plugins, requiring users to switch languages or use separate translation apps. While some platforms offer built-in translation features, these are often limited to text-only messages and may introduce delays, breaking the conversational flow.

4.1 Existing Algorithms and Technologies:

Socket.IO:

Socket.IO is widely used for real-time bi-directional communication between clients and servers. It enables instant message delivery in chat applications by maintaining persistent WebSocket connections. However, it does not provide built-in support for automatic language translation or user authentication, requiring additional integration.

JWT Tokens for Authorization:

JSON Web Tokens (JWT) provide a stateless authentication mechanism allowing secure user login and session management without server-side session storage. Many applications use JWT for authorization but often do not combine it with efficient frontend state management, leading to increased complexity.

State Management Libraries:

Redux and Context API are commonly used for managing React app state but can introduce boilerplate code and complexity. Zustand is emerging as a simpler and more efficient state management solution but is not yet widely adopted in chat applications.

Disadvantages

1. Translation features are often manual or triggered, reducing seamless interaction.
2. Integration of translation, authentication, and real-time messaging is fragmented.
3. Complex state management solutions increase development time and reduce maintainability.
4. Lack of cloud-based media handling reduces user experience in multimedia sharing.

5. PROPOSED SYSTEM

The proposed system is based on real-time chat communication integrated with automatic language translation, making conversations seamless and effortless for users speaking different languages. This system uses modern web technologies including the MERN stack (MongoDB, Express, React, Node.js) along with Socket.IO for real-time messaging, and Microsoft Azure Translator API for instant translation.

We focused on building a scalable chat application that not only supports text communication but also media sharing, user authentication, and state management for a smooth user experience. This system demonstrates how multiple advanced technologies can be combined efficiently to develop an interactive multilingual chat platform.

Technologies Used

- **MERN Stack:** Provides a robust backend and frontend framework with a NoSQL database for flexible data storage.
- **Socket.IO:** Enables real-time, bidirectional event-based communication between clients and servers.
- **JWT Tokens:** Used for secure user authorization, allowing stateless authentication and session management.
- **Zustand:** A lightweight and efficient state management library for React, simplifying frontend state handling.
- **Microsoft Azure Translator:** Powers automatic, on-the-fly translation of chat messages across multiple languages.
- **Cloudinary:** Manages image and media uploads, storage, and delivery to enhance the chat experience.
- **Tailwind CSS and Daisy UI:** Used for designing a responsive, clean, and customizable user interface.

Features Implemented

- **Automatic Language Translation:** Messages sent by users are automatically translated in real time to the recipient's preferred language using Azure Translator API, allowing seamless cross-language communication.
- **Real-Time Messaging:** Socket.IO ensures messages are delivered instantly with minimal latency.
- **Secure Authentication:** JWT tokens protect user data and manage authentication without server-side sessions.

- **Media Sharing:** Users can upload and share images and other media files via Cloudinary integration.
- **Efficient State Management:** Zustand is used to keep the app state lightweight and responsive, reducing complexity and improving performance.

Advantages

1. Enables instant multilingual communication without manual translation steps.
2. Real-time message delivery ensures smooth, uninterrupted conversations.
3. Secure and scalable user authentication using JWT tokens.
4. Simplified frontend development and state management with Zustand.
5. Cloud-based media handling optimizes performance and user experience.

6. SYSTEM REQUIREMENTS

6.1 Hardware Requirements

- **Processor:** Intel Core i3 or higher / AMD equivalent
- **RAM:** Minimum 4 GB (8 GB recommended for better performance)
- **Storage:** At least 20 GB free disk space for project files and database storage
- **Internet Connection:** Required for real-time messaging, API calls to Azure Translator, and Cloudinary services

6.2 Software Requirements

- **Operating System:** Windows 10 or later / Linux / macOS

Backend:

- Node.js (v14 or above)
- Express.js framework
- MongoDB database (local or cloud-based)

Frontend:

- React.js library
- Tailwind CSS and Daisy UI for UI styling
- Zustand for state management

7. SYSTEM ANALYSIS

Feasibility Study

The feasibility of the ChatSphere project is analyzed in this phase, and a general plan for the project is put forth along with some cost and resource estimates. During system analysis, the feasibility study of the proposed chat application with auto translation is carried out to ensure that the system is viable, efficient, and not a burden on resources.

Three key considerations involved in the feasibility analysis are:

7.1 Economical Feasibility

This study is carried out to check the economic impact that the ChatSphere system will have on the organization or users. The project utilizes mostly open-source and free technologies such as React, Node.js, MongoDB, Tailwind CSS, and Socket.IO, which significantly reduces development and deployment costs. Some cloud services like Microsoft Azure Translator and Cloudinary may incur costs, but they offer scalable pricing models that fit within a moderate budget. Overall, the expenditures are justified by the value the system adds by enabling multilingual, real-time communication efficiently.

7.2 Technical Feasibility

This study is carried out to evaluate the technical requirements and challenges of the system. The technologies used in ChatSphere are widely supported and have strong community backing. The system requires basic hardware and internet connectivity for

users, and the backend can be deployed on common cloud platforms or local servers. Minimal technical changes are needed for deployment since the system uses modern, standardized tools and APIs. The implementation is technically feasible within the available resources.

7.3 Social Feasibility

This aspect studies the acceptance level of the ChatSphere system by the end users. The system is designed to be user-friendly with a clean UI and intuitive features like automatic language translation and seamless real-time chat. Users are unlikely to feel threatened by the system but rather empowered to communicate across language barriers. Training requirements are minimal as the interface is similar to popular chat apps. User confidence can be boosted through simple tutorials or help sections, encouraging constructive feedback to improve the system further.

8. SYSTEM DESIGN

8.1 UML Diagrams

UML stands for **Unified Modeling Language**. UML is a standardized, general-purpose modeling language used extensively in **object-oriented software engineering**. It was developed and is maintained by the **Object Management Group (OMG)** with the intent to provide a common language for designing and constructing software blueprints.

UML plays a vital role in visualizing, specifying, constructing, and documenting the artifacts of a software system. It also supports business modeling and modeling of other non-software systems. In its current state, UML comprises two major components:

GOALS

The primary goals of UML in system design are:

1. To provide users a **ready-to-use, expressive visual modeling language** for developing and exchanging meaningful software models.

2. To offer **extensibility and specialization** mechanisms that allow the core concepts to be expanded.

3. To be **independent of any specific programming language or development process**.

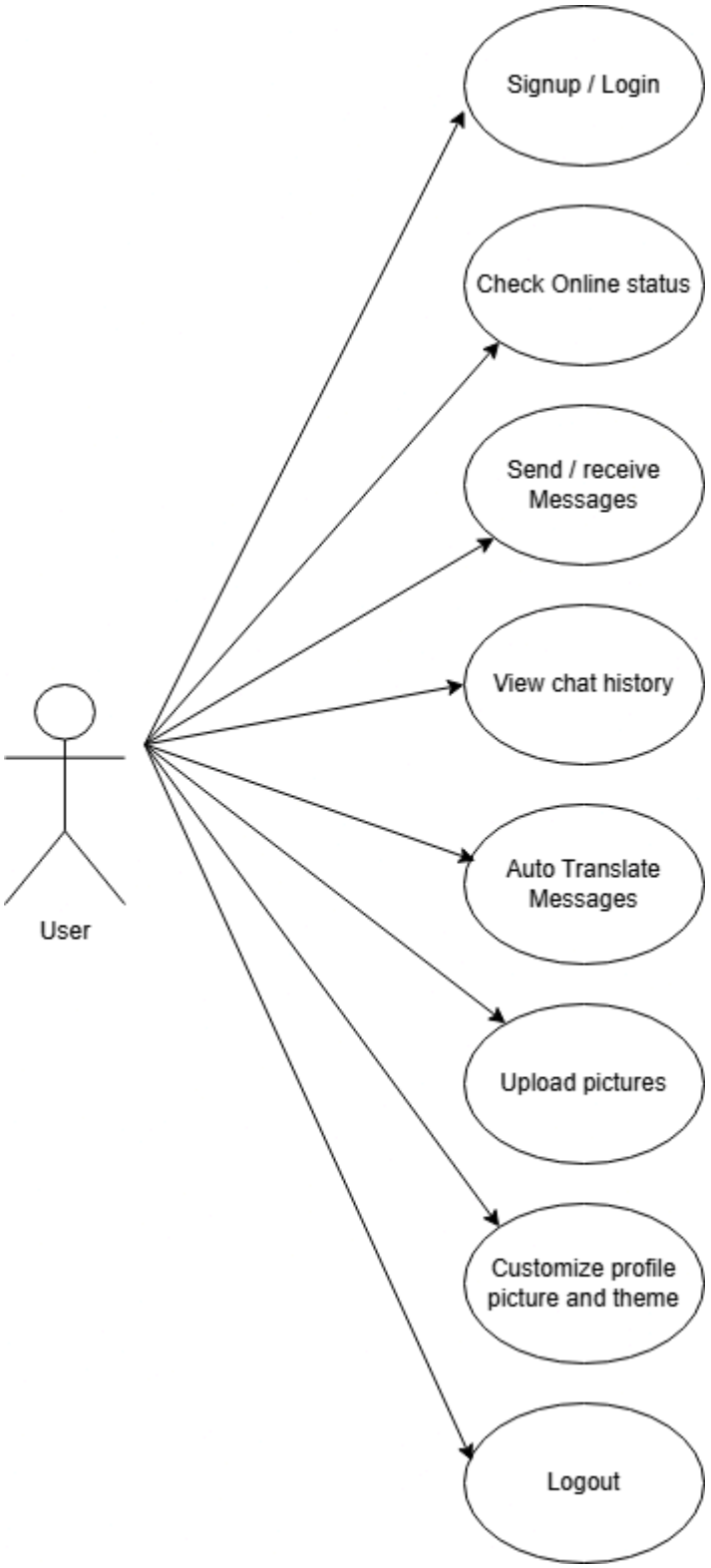
4. To provide a **formal foundation** for understanding the modeling language and its semantics.

5. To encourage the growth of the **object-oriented tools market**.

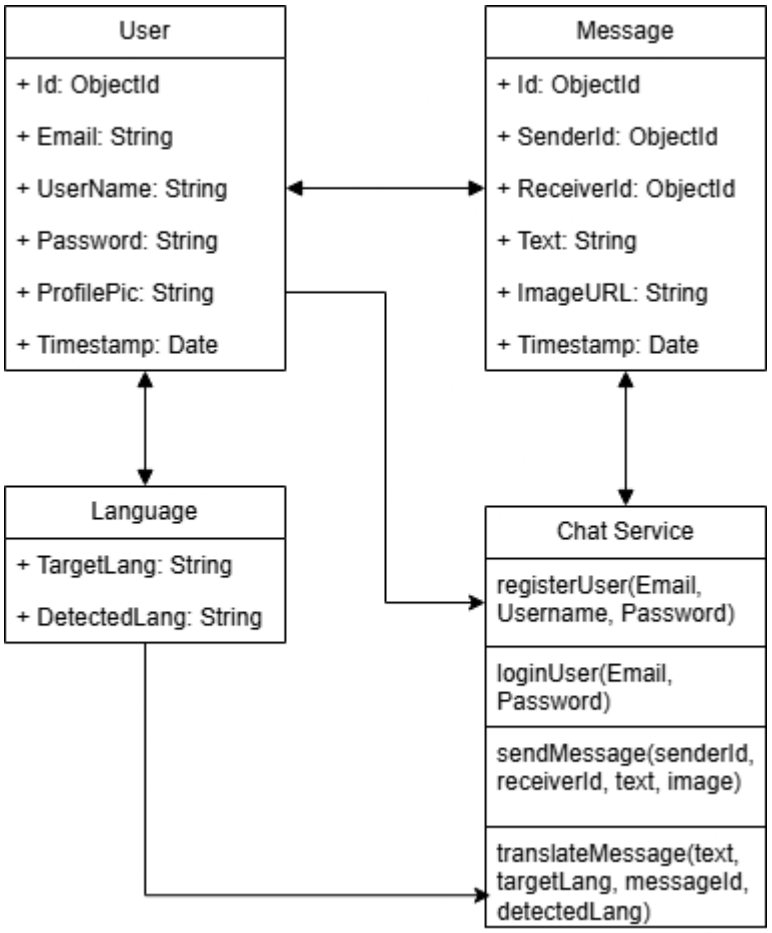
6. To support high-level development concepts such as **collaborations, design patterns, frameworks, and components**.

7. To integrate **best practices** from industry-proven software development methodologies.

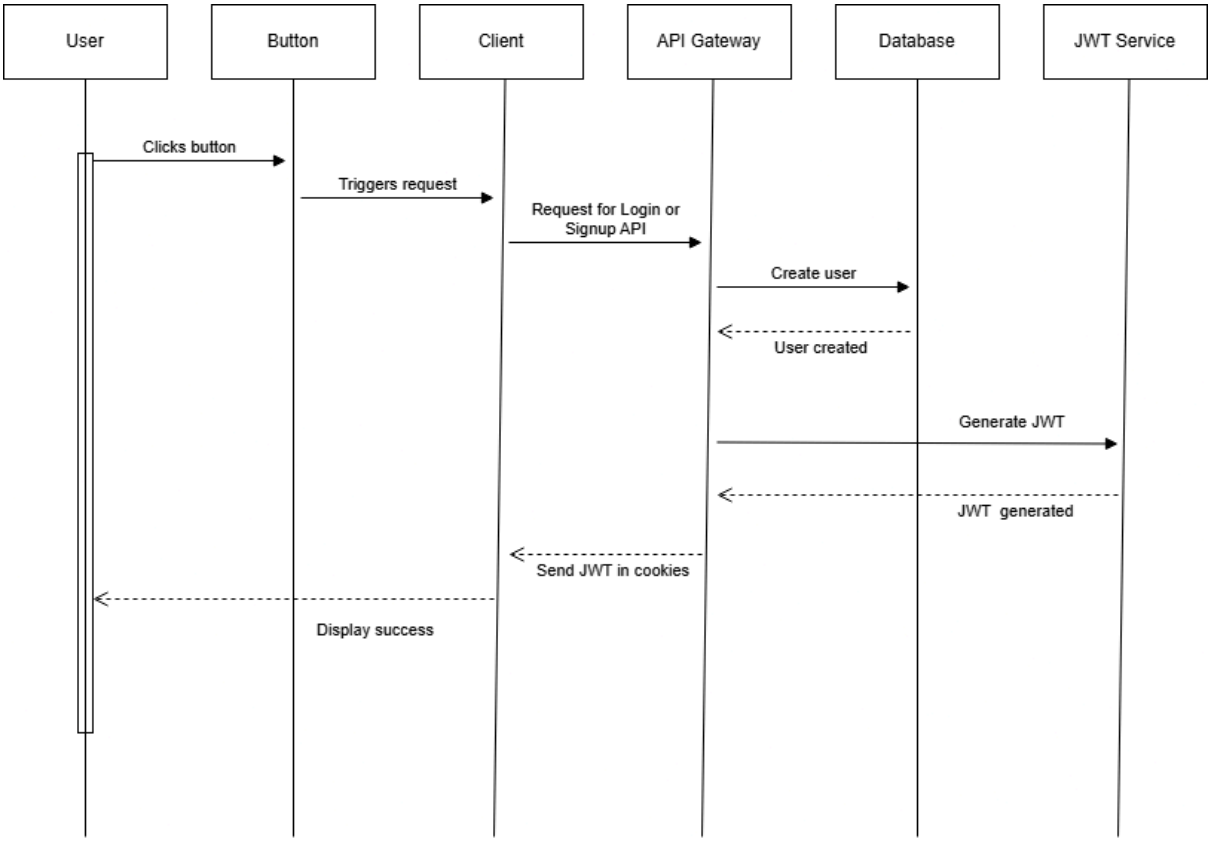
8.1.1 Use Case Diagram



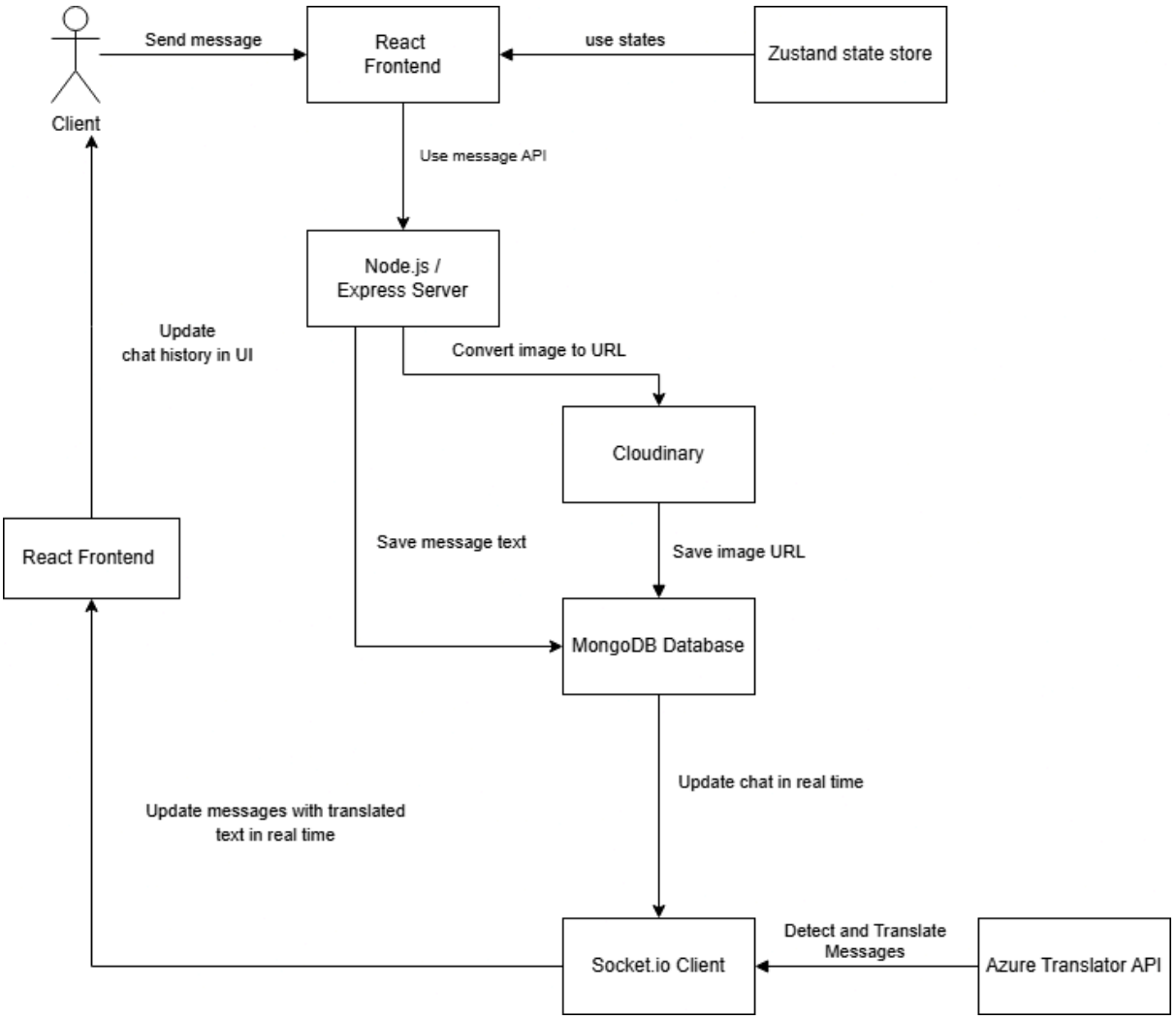
8.1.2 Class Diagram



8.1.3 Sequence Diagram



8.1.4 Collaboration Diagram



9. IMPLEMENTATION

9.1 Modules

1. Authentication Module

Handles user login, signup, and logout using MongoDB and JWT tokens. It tracks the logged-in user across the app and protects routes/components if the user is not authenticated.

2. Chat Module

Manages real-time messaging between users. Loads messages, subscribes to updates, renders them in a chat bubble layout, and supports text + image content.

3. Translation Module

Translates chat messages to a selected language using Azure Translate API. It supports auto-detecting language, toggling translations, and updating when the target language changes.

4. Global State Management

Uses Zustand to store and share global state like current user (`authUser`), selected chat partner, messages, and translation preferences across components.

5. Media Upload/Preview Module

Allows users to attach and preview image files in chat. Validates image type and supports image removal before sending.

6. UI Components & Skeletons

Includes reusable UI components like chat bubbles, avatars, and skeleton loaders. Built using Tailwind CSS and daisyUI for responsive and styled layouts.

9.2 Source Code

auth.controller.js :

```
import { generateToken } from "../lib/utils.js";
import User from "../models/user.model.js";
import cloudinary from "../lib/cloudinary.js";
import bcrypt from "bcryptjs";

export const signup = async (req, res) => {
  const { fullName, email, password } = req.body;
  try {
    if (!password || !fullName || !email) {
      return res.status(400).json({ message: "All fields are
required" });
    }
    if (password.length < 8) {
      return res
        .status(400)
        .json({ message: "Password must be at least 8 characters
long" });
    }
    const user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: "Email alread
exists!" });

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

    const newUser = new User({
      fullName: fullName,
      email: email,
      password: hashedPassword,
    });

    if (newUser) {
      generateToken(newUser._id, res);
      await newUser.save();
      res.status(201).json({
```

```
        _id: newUser._id,
        fullName: newUser.fullName,
        email: newUser.email,
        profilePic: newUser.profilePic,
      });
    } else {
      return res.status(400).json({ message: "Invalid user data" });
    }
  } catch (error) {
    console.log("Error in signup controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};

export const login = async (req, res) => {
  const { email, password } = req.body;
  try {
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(400).json({ message: "Invalid credentials"
});
    }
    const isPasswordCorrect = await bcrypt.compare(password,
user.password);
    if (!isPasswordCorrect) {
      return res.status(400).json({ message: "Invalid credentials"
});
    }
    generateToken(user._id, res);
    res.status(200).json({
      _id: user._id,
      fullName: user.fullName,
      email: user.email,
      profilePic: user.profilePic,
    });
  } catch (error) {
    console.log("Error in login controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
```

```
};

export const logout = (req, res) => {
  try {
    res.cookie("jwt", "", { maxAge: 0 });
    res.status(200).json({ message: "Logged out successfully" });
  } catch (error) {
    console.log("Error in logout controller", error.message);
    res.status(500).json({ message: "Internal Server Error" });
  }
};

export const updateProfile = async (req, res) => {
  try {
    const { profilePic } = req.body;
    const userId = req.user._id;

    if (!profilePic) {
      res.status(400).json({ message: "Provide a profile pic" });
    }

    const cloudinaryResponse = await
cloudinary.uploader.upload(profilePic);

    const updatedUser = await User.findByIdAndUpdate(
      userId,
      {
        profilePic: cloudinaryResponse.secure_url,
      },
      { new: true }
    );

    res.status(200).json(updatedUser);
  } catch (error) {
    console.log("Error in updateProfile: ", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
};
```

```
export const checkAuth = (req, res) => {
  try {
    res.status(200).json(req.user);
  } catch (error) {
    console.log("error in checkAuth controller");
    res.status(500).json({ message: "Internal Server Error" });
  }
};
```

message.controller.js :

```
import User from "../models/user.model.js";
import Message from "../models/message.model.js";

import cloudinary from "../lib/cloudinary.js";
import { getReceiverSocketId, io } from "../lib/socket.js";

export const getUsersForSidebar = async (req, res) => {
  try {
    const loggedInUserId = req.user._id;
    const filteredUsers = await User.find({
      _id: { $ne: loggedInUserId },
    }).select("-password");

    res.status(200).json(filteredUsers);
  } catch (error) {
    console.error("Error in getUsersForSidebar: ", error.message);
    res.status(500).json({ error: "Internal server error" });
  }
};

export const getMessages = async (req, res) => {
  try {
    const { id: userToChatId } = req.params;
    const myId = req.user._id;

    const messages = await Message.find({
```

```
      $or: [
        { senderId: myId, receiverId: userToChatId },
        { senderId: userToChatId, receiverId: myId },
      ],
    });

    res.status(200).json(messages);
  } catch (error) {
    console.log("Error in getMessages controller: ", error.message
    res.status(500).json({ error: "Internal server error" });
  }
};

export const sendMessage = async (req, res) => {
  try {
    const { text, image } = req.body;
    const { id: receiverId } = req.params;
    const senderId = req.user._id;

    let imageUrl;
    if (image) {
      // Upload base64 image to cloudinary
      const uploadResponse = await cloudinary.uploader.upload(imag
      imageUrl = uploadResponse.secure_url;
    }

    const newMessage = new Message({
      senderId,
      receiverId,
      text,
      image: imageUrl,
    });

    await newMessage.save();

    const receiverSocketId = getReceiverSocketId(receiverId);
    if (receiverSocketId) {
      io.to(receiverSocketId).emit("newMessage", newMessage);
    }
```

```
    res.status(201).json(newMessage);
  } catch (error) {
    console.log("Error in sendMessage controller: ", error.message
    res.status(500).json({ error: "Internal server error" });
  }
};
```

translate.controller.js :

```
import axios from "axios";
import { v4 as uuidv4 } from "uuid";
import dotenv from "dotenv";

dotenv.config();

export const translateTo = async (req, res) => {
  const key = process.env.TRANSLATION_API_KEY;
  const endpoint = "https://api.cognitive.microsofttranslator.com'
  const location = process.env.RESOURCE_REGION;

  const text = req.body?.text;
  const toLanguage = req.body.to;

  const toLanguages = toLanguage.split(",");

  if (!key || !location) {
    return res.status(500).json({
      error: "Missing Azure Translator config in .env",
    });
  }

  try {
    // 1. Detect language
    const detectResponse = await axios.post(
      `${endpoint}/detect?api-version=3.0`,
      [{ text }],
      {
```

```
        headers: {
          "Ocp-Apim-Subscription-Key": key,
          "Ocp-Apim-Subscription-Region": location,
          "Content-Type": "application/json",
          "X-ClientTraceId": uuidv4().toString(),
        },
      }
    );

    const detectedLang = detectResponse.data[0]?.language;

    // 2. Translate to specified languages
    const translateResponse = await axios.post(

`${endpoint}/translate?api-version=3.0&from=${detectedLang}&to=${t
nguages.join(
        ","
      )}`,
      [{ text }],
      {
        headers: {
          "Ocp-Apim-Subscription-Key": key,
          "Ocp-Apim-Subscription-Region": location,
          "Content-Type": "application/json",
          "X-ClientTraceId": uuidv4().toString(),
        },
      }
    );

    const translations = translateResponse.data[0]?.translations;

    return res.json({
      originalText: text,
      detectedLanguage: detectedLang,
      translations,
    });
  } catch (error) {
    console.error(
      "Azure Translation Error:",
```

```
        error?.response?.data || error.message
      );
      return res.status(500).json({
        error: "Translation service failed.",
        details: error?.response?.data || error.message,
      });
    }
  };
```

9.3 Deployment Process on Render

The deployment of ChatSphere was done using Render, a modern cloud platform that simplifies hosting for both frontend and backend applications. The process involved hosting the backend server (Node.js/Express) and the frontend client (React/Vite) as two separate web services.

Steps Involved

1. Preparing the Backend (Express Server)

- Ensure that the backend/ directory contains a package.json with all necessary dependencies and a valid start script (node src/index.js).
- Confirm that the server listens on process.env.PORT and not a hardcoded value.
- Add a .env file locally for testing, and include necessary environment variables (e.g., MongoDB URI, JWT secret, etc.).

2. Preparing the Frontend (React + Vite)

- Make sure the vite.config.js has the correct base setting (e.g., / or /chatsphere/).
- Add an environment variable VITE_API_URL in Render to point to your backend URL.

3. Deploying the Backend on Render

- Go to Render.com and create a new Web Service.
- Connect your GitHub repo.
- Select the backend folder as the root.
- Set the build and start commands:
- Build Command: npm install

- Start Command: npm start
- Add environment variables through the Render dashboard.
- Choose a free or paid instance and deploy.

4. Deploying the Frontend on Render

- Create another Web Service.
- Select the frontend directory as the root.
- Build and start commands:
- Build Command: npm install && npm run build
- Start Command: npx serve -s dist or use [Render's static site service](recommended for Vite apps).
- Alternatively, deploy as a Static Site:
- Output directory: dist
- Build command: npm run build

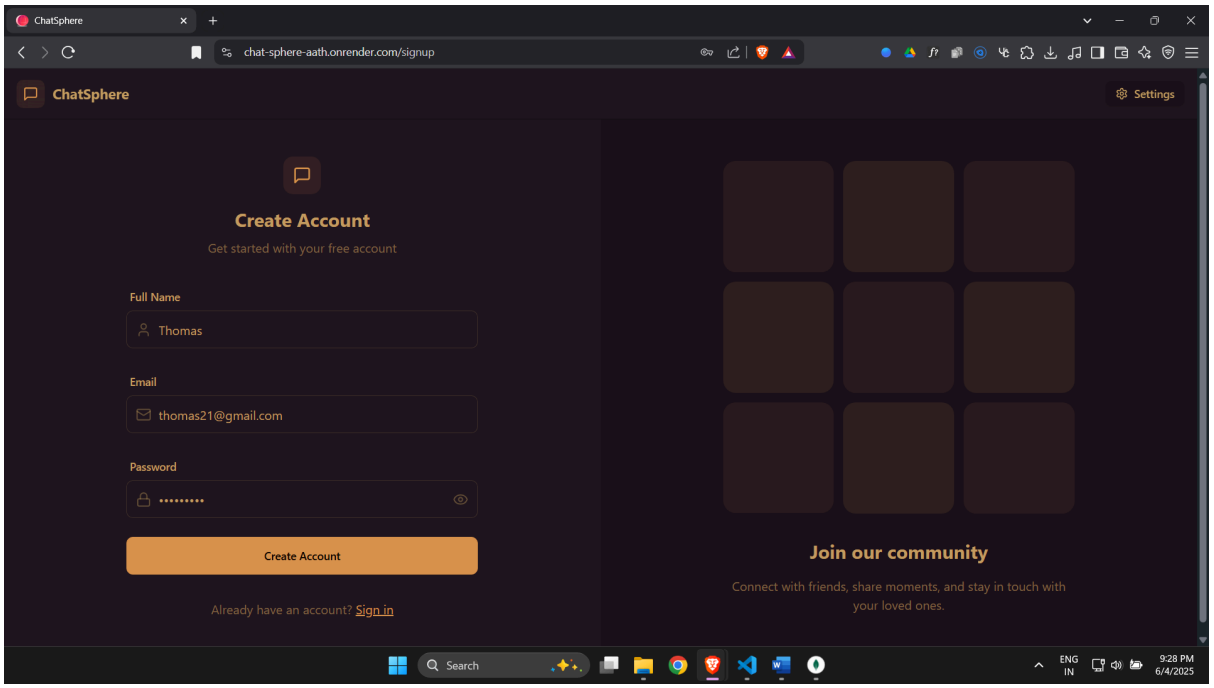
5. Final Touches

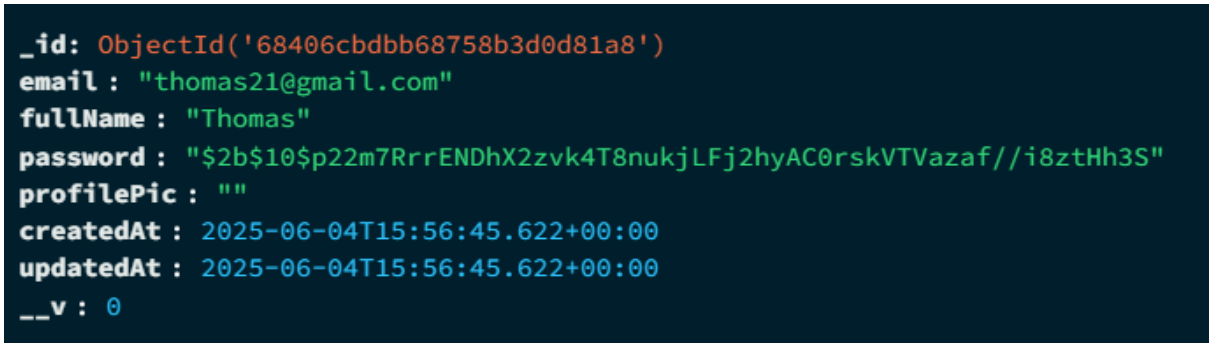
- Ensure CORS is correctly configured on the backend to allow requests from the frontend domain.
- Confirm WebSocket support is enabled on Render (used by socket.io).
- Use Render's auto-deploy feature to redeploy the services on every commit to the main branch

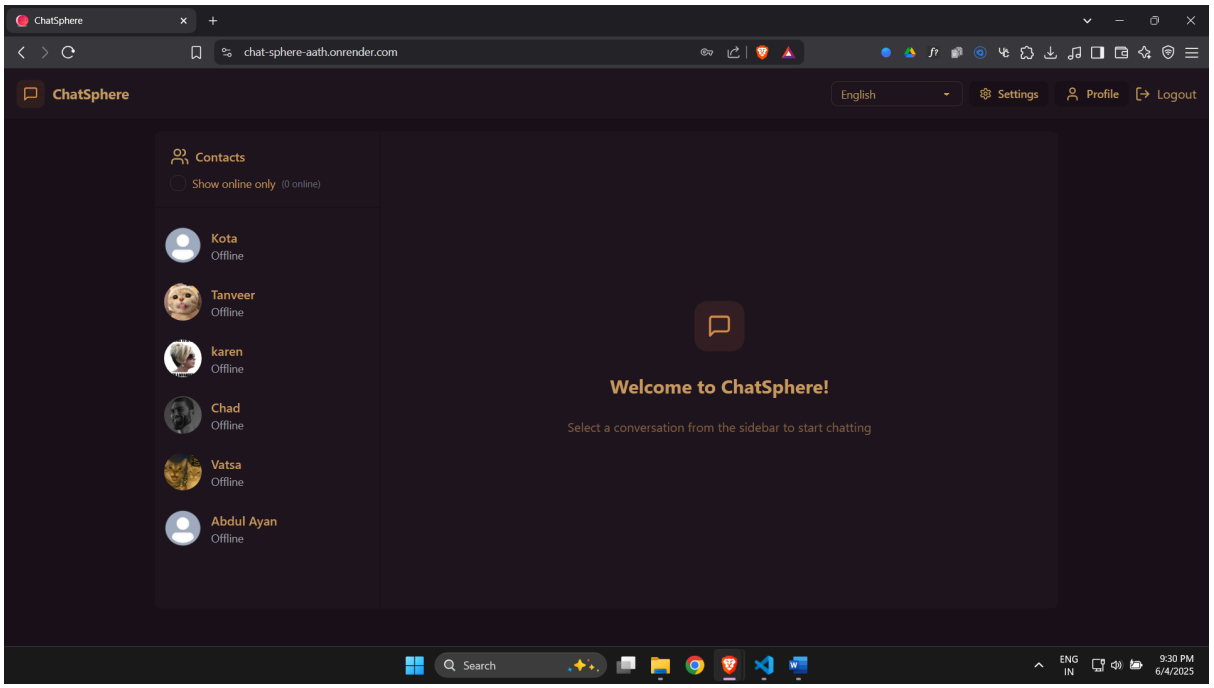
Deployment Outcome

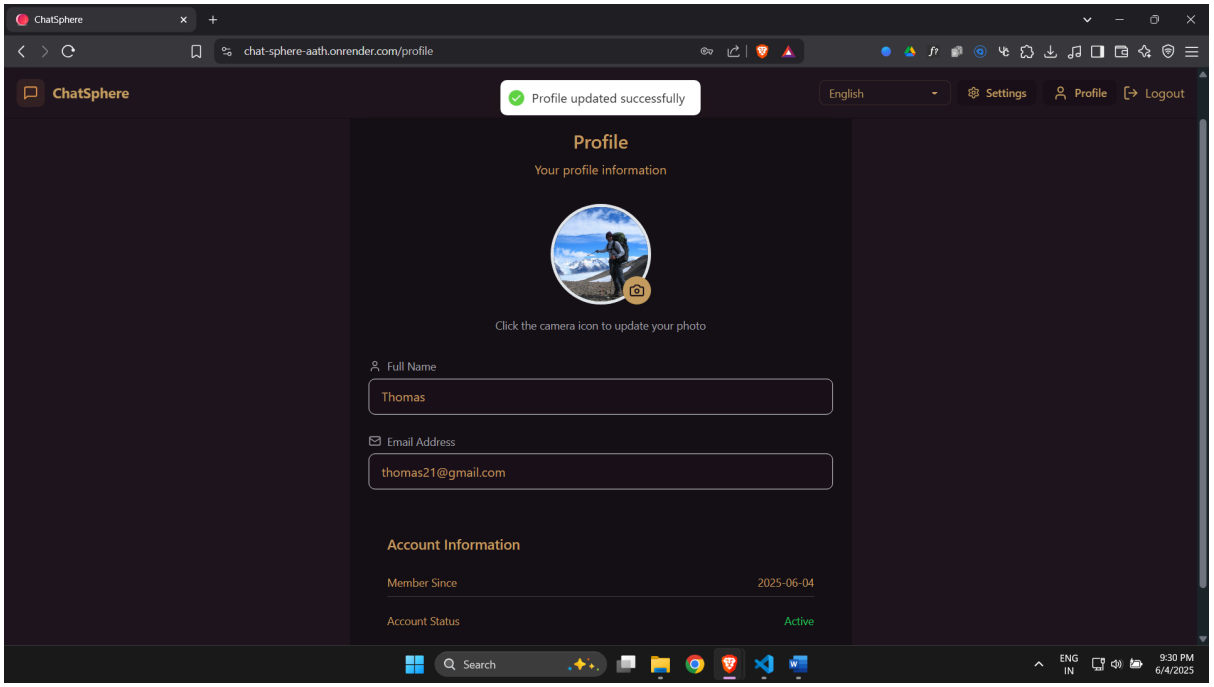
The application was successfully deployed with both frontend and backend accessible via public URLs. Real-time chat functionality, authentication, and translation features worked as expected in the production environment.

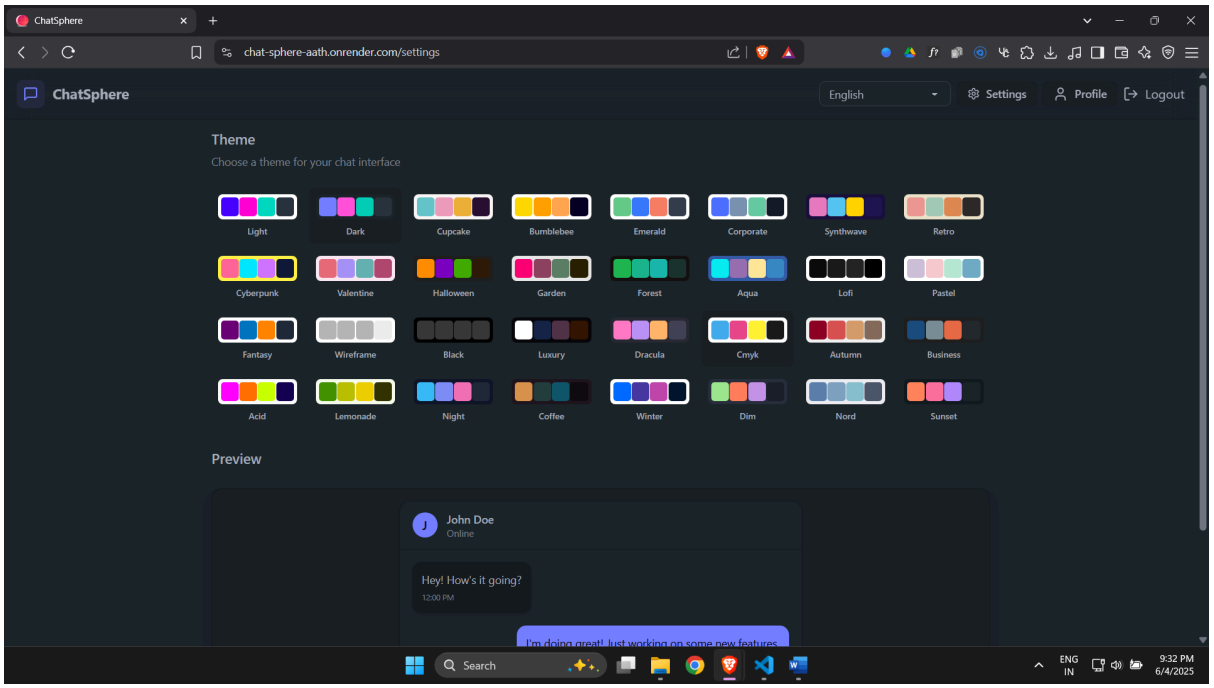
9.3 Output

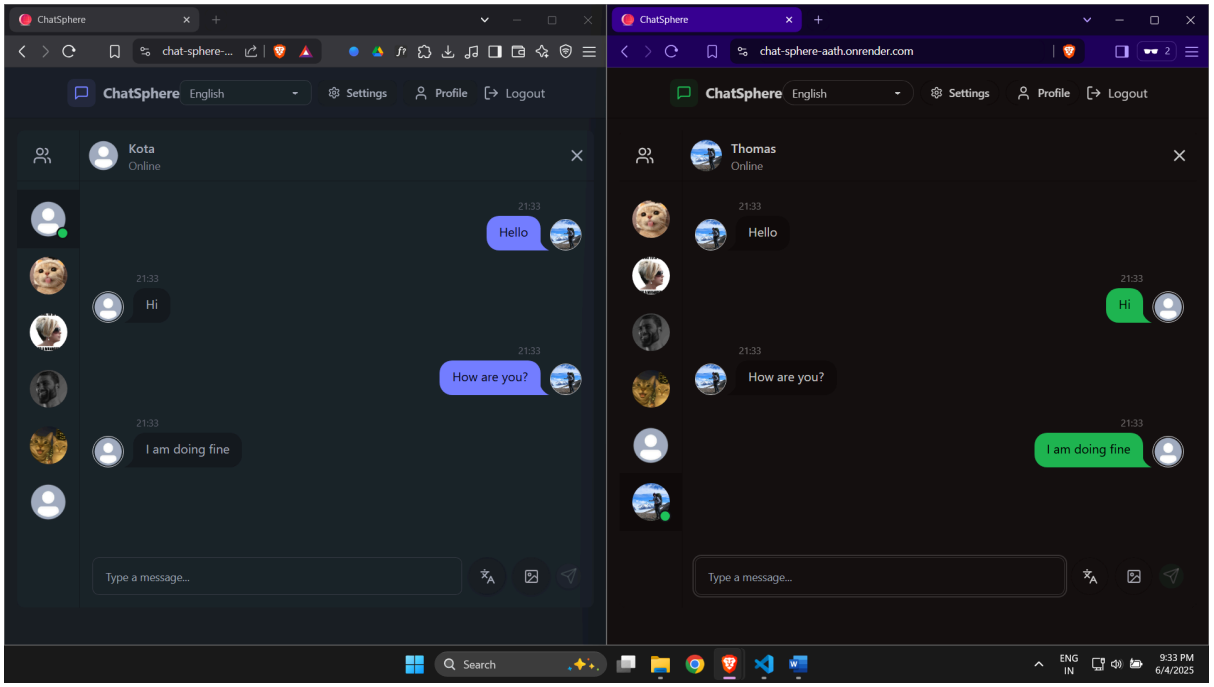

ChatSphere
chat-sphere-aath.onrender.com/signup
ChatSphere
Settings
Create Account
Get started with your free account
Full Name
Thomas
Email
thomas21@gmail.com
Password
Create Account
Already have an account? Sign in
Join our community
Connect with friends, share moments, and stay in touch with your loved ones.
Search
ENG
IN
9:28 PM
6/4/2025

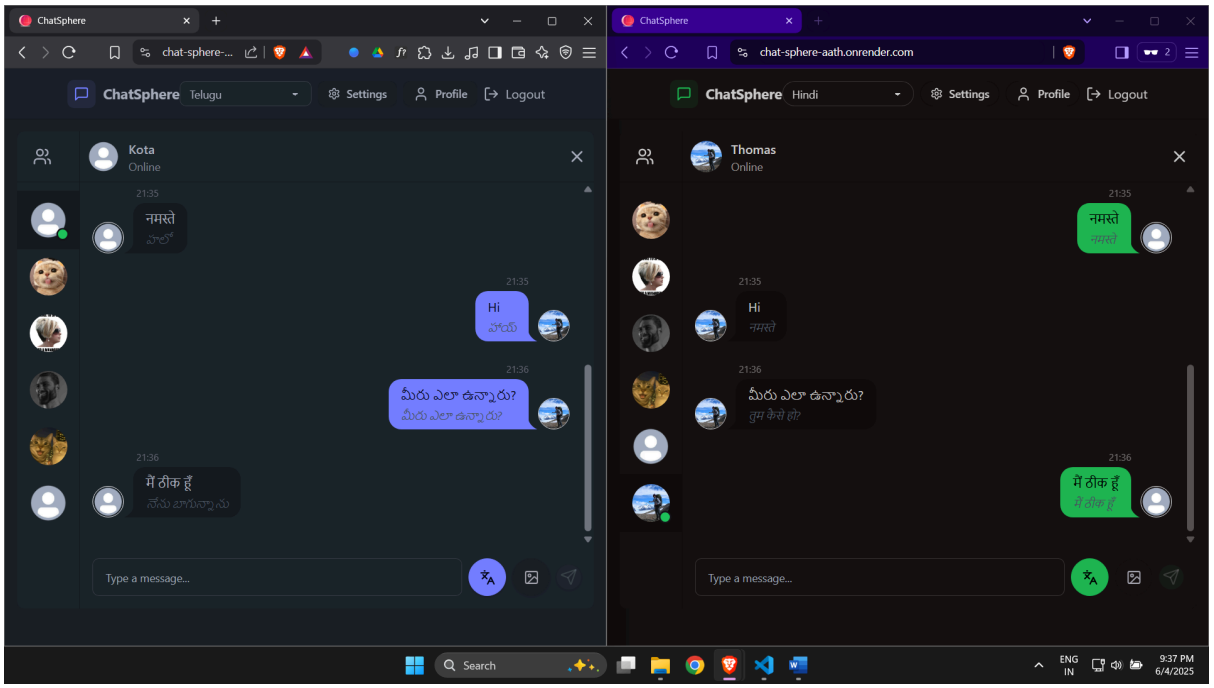

_id: ObjectId('68406cbdbb68758b3d0d81a8')
email : "thomas21@gmail.com"
fullName : "Thomas"
password : "$2b$10$p22m7RrrENDhX2zvk4T8nukjLFj2hyAC0rskVTVazaf//i8ztHh3S"
profilePic : ""
createdAt : 2025-06-04T15:56:45.622+00:00
updatedAt : 2025-06-04T15:56:45.622+00:00
__v : 0


ChatSphere
chat-sphere-aath.onrender.com
ChatSphere
English
Settings
Profile
Logout
Contacts
Show online only (0 online)
Kota
Offline
Tanveer
Offline
karen
Offline
Chad
Offline
Vatsa
Offline
Abdul Ayan
Offline
Welcome to ChatSphere!
Select a conversation from the sidebar to start chatting
Search
ENG
IN
9:30 PM
6/4/2025


ChatSphere
chat-sphere-aath.onrender.com/profile
ChatSphere
Profile updated successfully
English
Settings
Profile
Logout
Profile
Your profile information
Click the camera icon to update your photo
Full Name
Thomas
Email Address
thomas21@gmail.com
Account Information
Member Since
2025-06-04
Account Status
Active
Search
ENG
IN
9:30 PM
6/4/2025


ChatSphere
chat-sphere-aath.onrender.com/settings
ChatSphere
English
Settings
Profile
Logout
Theme
Choose a theme for your chat interface
Light
Dark
Cupcake
Bumblebee
Emerald
Corporate
Synthwave
Retro
Cyberpunk
Valentine
Halloween
Garden
Forest
Aqua
Lofi
Pastel
Fantasy
Wireframe
Black
Luxury
Dracula
Cmyk
Autumn
Business
Acid
Lemonade
Night
Coffee
Winter
Dim
Nord
Sunset
Preview
John Doe
Online
Hey! How's it going?
12:00 PM
Search
ENG
IN
9:32 PM
6/4/2025


ChatSphere
chat-sphere-...
ChatSphere
English
Settings
Profile
Logout
Kota
Online
21:33
Hello
21:33
Hi
21:33
How are you?
21:33
I am doing fine
Type a message...
chat-sphere-aath.onrender.com
ChatSphere
English
Settings
Profile
Logout
Thomas
Online
21:33
Hello
21:33
Hi
21:33
How are you?
21:33
I am doing fine
Type a message...
Search
ENG
IN
9:33 PM
6/4/2025



10. SOFTWARE ENVIRONMENT

10.1 Overview

Chatsphere is a real-time multilingual chat application built on the MERN stack, using modern npm packages and cloud services to deliver a scalable and responsive experience. The project uses **JavaScript** as the core language for both frontend and backend development, leveraging its asynchronous nature to handle live communication effectively.

Key technologies and npm packages include:

- **MERN Stack:**
  - **MongoDB** for flexible, NoSQL data storage
  - **Express.js** as backend framework
  - **React.js** for building dynamic user interfaces
  - **Node.js** as the server runtime

- **Zustand**
  Lightweight state management for React to handle global app states such as authentication, chat messages, and language preferences.

- **Tailwind CSS**
  Utility-first CSS framework enabling fast, responsive, and consistent UI styling with minimal custom CSS.

- **Socket.IO**
  Enables real-time bidirectional communication between clients and server for instant messaging.

- **Microsoft Azure Translator API**
  Cloud-based automatic language detection and translation service for multilingual chat.

- **Cloudinary**
  Media management platform for uploading, optimizing, and delivering images and files.

- **JWT (JSON Web Tokens)**
  Provides secure, stateless authentication and authorization for API endpoints.

10.2 NPM Packages

Frontend

Core Dependencies

- **react & react-dom:** UI components and DOM rendering
- **axios:** HTTP client
- **zustand:** Lightweight state management
- **socket.io-client:** Real-time communication
- **react-router-dom:** Client-side routing
- **react-hot-toast:** Toast notifications

Development Tools

- **vite:** Build tool and dev server
- **tailwindcss & daisyui:** Utility-first CSS and UI components

Backend

Core Dependencies

- **express:** Backend framework
- **mongoose:** MongoDB ODM
- **socket.io:** Real-time communication
- **axios:** HTTP client
- **bcryptjs:** Password hashing
- **jsonwebtoken:** JWT authentication
- **cloudinary:** Media upload & management
- **cookie-parser:** Cookie parsing
- **cors:** Cross-Origin Resource Sharing
- **dotenv:** Environment variables
- **uuid:** Unique ID generation

Development Tools

- **nodemon:** Auto-restart server in development

11. SYSTEM TESTING

System Testing aims to uncover errors by thoroughly exercising the entire software system to ensure it meets all requirements and user expectations without failure. Different types of testing focus on specific objectives within this process:

11.1 Types of Tests

Unit Testing

Tests individual software units or components to validate internal logic, decision branches, and input-output accuracy. It confirms that each business process performs correctly with clearly defined inputs and expected results. Unit testing is structural and invasive, performed after each component's development and before integration.

Integration Testing

Validates that combined components work together as a cohesive system. It focuses on interface interactions and ensures that individual units, which passed unit testing, function correctly when integrated. The aim is to detect defects caused by component interactions.

Functional Testing

Verifies that all system functions operate according to business and technical requirements. This involves:

- Accepting valid input and rejecting invalid input
- Exercising all identified functions and output scenarios
- Ensuring proper invocation of interfacing systems or procedures
- Tests cover business processes, data fields, predefined workflows, and process sequences to guarantee comprehensive functional coverage.

System Testing

Tests the fully integrated software system in a realistic environment to ensure predictable, correct results. It focuses on end-to-end process flows, configuration, and integration points between modules and external systems.

White Box Testing

Performed with knowledge of the internal structure and code of the system. It tests code paths and logic that cannot be accessed through black-box testing techniques, improving test coverage of edge cases and internal workflows.

Unit Testing

Unit testing is conducted during the coding phase to verify individual components work as expected. Tests focus on:

- Ensuring all input fields accept valid data in the correct format
- Preventing duplicate entries
- Verifying links navigate to the correct pages without delay
- Confirming response messages display promptly

Integration Testing

Integration testing checks the interaction between multiple components and services to identify interface defects. This ensures that backend APIs, frontend UI, real-time socket connections, and translation services communicate without errors.

Acceptance Testing

User Acceptance Testing (UAT) involves end users to validate that the system meets all functional requirements and business goals. It confirms the application is ready for production use.

Test Results

All test cases for unit, integration, and acceptance testing passed successfully with no defects found.

11.2 Test cases:

The following test cases were created and executed to ensure the correctness and robustness of the Chatsphere application. Each test validates a core functionality, including message handling, user authentication, media sharing, and language translation.

Test Case ID	Test Case Description	Precondition	Test Steps	Expected Result	Status
TC001	Send a text message	User is logged in	1. Open chat room 2. Type a message 3. Click Send	Message appears in chat window instantly	Pass
TC002	Send empty message	User is logged in	1. Open chat room 2. Leave message box empty 3. Click Send	Message not sent; Send Button disabled	Pass
TC003	Receive a message	User is connected to chat	Another user sends a message	Message appears in real-time without refresh	Pass

TC004	Auto-detect and translate messages Upon toggle	Message in foreign language received	1. Receive message in non-native language	Translated text appears below original message	Pass
TC005	Change target language for translation	User logged in	1. Select different target language	Messages translated to selected language	Pass
TC006	Login with valid credentials	User on login page	1. Enter valid username and password 2. Click Login	User redirected to chat dashboard	Pass
TC007	Login with Invalid credentials	User on login page	1. Enter invalid username or password 2. Click Login	Error message displayed, login denied	Pass
TC008	Logout user	User logged in	1. Click logout button	User redirected to login page	Pass

TC009	Upload image in chat	User logged in	1. Open chat 2. Click upload image 3. Select file 4. Send	Image appears inline in chat for all participants	Pass
TC010	Navigation to profile page	User logged in	1. Click profile link	Profile page loads correctly	Pass
TC011	Navigation to chat rooms list	User logged in	1. Click chat rooms link	List of chat rooms loads	Pass
TC012	Verify JWT token expiration handling	User logged in	1. Keep session idle until token expires	User prompted to login again	Pass

12. CONCLUSION

In conclusion, ChatSphere successfully implements a real-time chat system with features such as user authentication, private and group messaging, and optional message translation. The application uses modern web technologies like React, Node.js, Socket.IO, and MongoDB, and has been designed with scalability and user experience in mind.

Throughout the development process, we integrated efficient state management, responsive UI design, and secure backend APIs. The system has been tested for unit, integration, and user acceptance to ensure it meets both functional and performance requirements.

While the current version of ChatSphere includes core communication features and language translation for diverse user interaction, there is still scope for improvement. In future iterations, the application can be enhanced by adding end-to-end encryption, message history search, media sharing capabilities, and better support for accessibility and offline usage.

The success of this project demonstrates the feasibility and effectiveness of building a modern, modular chat application, and sets the foundation for further innovation and enhancement in the communication domain.

13. REFERENCES

[1] C.-Y. Yang and H.-Y. Lin, 'An instant messaging with automatic language translation', in *2010 3rd IEEE International Conference on Ubi-Media Computing*, 2010, pp. 312–316. [Online]. Available: https://ieeexplore.ieee.org/document/5544438

[2] M. D. Hossain and M. Ismail, 'AUTO TRANSLATION CHAT SYSTEM', *International Journal of Information Systems and Engineering*, vol. 5, pp. 88–104, 11 2017. [Online]. Available: https://www.researchgate.net/publication/323712682_AUTO_TRANSLATION_CHAT_SYSTEM

[3] GeeksforGeeks, "How to create a chat app using socket.io and Node.js?" [Online]. Available: https://www.geeksforgeeks.org/how-to-create-a-chat-app-using-socket-io-node-js/

[4] freeCodeCamp, "Build a real-time chat app with React, Express, Socket.io, and HarperDB." [Online]. Available: https://www.freecodecamp.org/news/build-a-realtime-chat-app-with-react-express-socketio-and-harperdb/

[5] Socket.io, "Get started: Build a chat application." [Online]. Available: https://socket.io/get-started/chat