**1.SYNOPSIS**

**Title: - Talk-Hub**

**Aim:**

The aim of the **Talk-Hub** project is to develop a robust and scalable real-time communication platform that enables users to connect through instant messaging, fostering seamless interaction in both personal and real-time Chat App using the MERN stack that combines traditional messaging capabilities with a chatbot feature. This chatbot will enhance user engagement by offering automated assistance, answering queries, and performing predefined tasks seamlessly.

**Description:**

Talk-Hub is designed to provide an engaging platform for users to connect and chat in real time. The app supports functionalities such as profile customisation, online presence indication, secure login and signup processes, and password recovery through email OTP verification. The application’s light and dark modes enhance the user experience, while notifications keep users informed of incoming messages and user activity.

**Modules:**

* **Authentication Module:** Includes login, signup, forgot password, and OTP-based password reset.
* **User Profile Management:** Users can edit profiles, set profile pictures, and add personal information.
* **Chat Functionality:** real-time text and media messaging, along with message timestamps.
* **Notification System:** Pop-up notifications for new messages and online user status.
* **Theme Customisation:** Options for dark and light mode to suit user preferences**.**
* **Friend System:** The user can connect with friends and check online presence**.**
* **AI-Based Chatbot**: Including an AI chatbot for user support and interaction

This project is a full stack web development project that aims at building a chatting application using the Frontend Technologies React (a framework built on top of the ever popular and evolving programming language JavaScript that has been ruling the IT industry over 25 years) and many of the npm dependencies such as the React Fade –In and the suite React library for the User Interface Part. The suite has several built-in components in React that can simply be imported in the React component file using the direct import statement after the dependency has been installed on the system.

**2.INTRODUCTION**

# INTRODUCTION

# 

The popularity of the **MERN stack** is ever increasing in the field of web development for frontend and backend applications. The use of React stems from the fact that it was built on JavaScript and JS has been the most popular programming language amongst developers for over 25 years. React was built on top of JS by developers from Facebook and made making the UI so much simpler than the normal HTML. It made rendering the components so much easier and faster. For instance, if we had to render a component time and again, we had to hard code it as many times as we needed it. However, in React, we only have to make a component just once and then import it in whichever component we need it. MERN stack uses the Express JS and Node JS for developing the backend. Earlier, the entire backend client and server part used to be handled by solely Node. However, sending requests and getting back responses from the server was so much more difficult, tedious, painstaking and more complicated. Then, Express was introduced as the de facto application server for building backend API. The server and client applications were made so much easier as there was no need to build the server but it was introduced by Express itself. Using Express has become as easy as importing it in a backend file and creating its instance using the const app = require(‘express’) and then listening it on a given port number just like the backend. The syntax becomes much simpler with Express than using simple Node. Also, the react-router-dom provides us the Router functionality that helps us define routes in the frontend application without creating a backend directory for the routing part. The router consists of the routing endpoint and the handler function for the request handling portion

### Overview:

The **Talk-Hub** is a web-based application designed for users to engage in real-time conversations. It also integrates a chatbot feature to enhance user interaction, automate responses, and simulate a human-like chat experience . Markup Language (HTML), chakra UI ,JavaScript , MongoDB, React ,Express Js ,Node Js

### Purpose:

The purpose of this project is to develop a versatile communication platform that enables seamless real-time interaction between users while integrating a chatbot to enhance the overall user experience. The application aims to address the growing need for efficient, automated, and accessible communication in personal, professional, and customer service settings.

**3.OBJECTIVE**

# OBJECTIVE

The most primary objective of this project is to understand the concept of building applications with the MERN stack (MongoDB, Express, React and Node). In this particular project, the focus has been built on a chatting application.

* **Real-Time Messaging:**  
  Enable users to send and receive messages instantly using technologies like WebSockets and Socket.IO.
* **Ease of Use:**  
  Design an intuitive interface with responsive layouts for desktop and mobile devices.
* **User Authentication and Security:**  
  Implement secure login and registration systems using JWT and encrypted passwords to ensure data protection.
* **Chatbot Integration:**  
  Incorporate a chatbot for automated responses, enhancing user interaction and supporting predefined tasks or FAQs.
* **Persistent Data Storage:**  
  Store chat histories, user profiles, and media in MongoDB for easy retrieval and scalability.
* **Scalability:**  
  Design a robust architecture that supports an increasing number of users and messages without performance degradation.
* **Enhanced User Interaction:**  
  Provide features like typing indicators, read receipts, emojis, and media sharing to make conversations more engaging.

**4.EXISTING AND PROPOSED SYSTEM**

# EXISTING AND PROPOSED SYSTEM

The current systems used for real-time communication face several challenges, limitations, or inefficiencies that this project aims to address. Below is an outline of the existing system’s state:

### Existing System

1. **Features in Existing Systems**

* Many platforms provide real-time messaging and basic group chat functionality.
* Some systems include chatbot integration but are limited to predefined scripts without advanced intelligence.
* Communication platforms often lack seamless accessibility across devices and browsers.

1. **Challenges in Existing Systems**

* **Limited Real-Time Functionality:**  
  Existing systems may have delays or fail to handle high volumes of concurrent users efficiently.
* **Lack of Scalability:**  
  Many platforms cannot handle growing data or user loads without performance degradation.
* **Poor Customizability:**  
  Businesses cannot always customize existing platforms to fit their branding or operational needs.
* **Insufficient Integration Options:**  
  Limited support for integrating advanced AI-driven chatbots or third-party APIs.
* **Privacy and Security Risks:**  
  Some systems lack robust encryption or fail to comply with global data protection standards.
* **High Cost of Ownership:**  
  Existing solutions may charge premium fees for features or storage that could be optimized.

### Proposed System

The proposed Web-Based ChatApp introduces an enhanced system that resolves the shortcomings of existing systems and offers new capabilities tailored to user needs.

1. **Features of the Proposed System**

* **Real-Time Messaging:**  
  Instant delivery of messages using WebSocket technology for fast and efficient communication.
* **Advanced Chatbot Integration:**  
  A chatbot capable of handling complex user queries with AI-driven responses (e.g., NLP-based bots).
* **Device-Independent Access:**  
  Fully functional on any device with a web browser, ensuring maximum accessibility.
* **Customizable and Scalable:**  
  Modular architecture allows businesses to customize features and scale with user growth.
* **Enhanced Security:**  
  End-to-end encryption, secure login (JWT), and compliance with data protection standards like GDPR.
* **Persistent Data Storage:**  
  MongoDB ensures secure, scalable storage of chat histories, user data, and media.

1. **Advantages of the Proposed System**

* **Scalability:**  
  Designed to support a large number of concurrent users with efficient database and server architecture.
* **AI and Automation:**  
  Leverages AI for chatbot functionality to automate customer service, FAQs, and simple tasks.
* **Improved User Interaction:**  
  Features like read receipts, typing indicators, file sharing, and emoji support improve user experience.
* **Cost-Efficient Solution:**  
  Open-source technologies (React.js, Node.js, MongoDB) minimize costs without compromising performance.
* **Adaptability:**  
  Supports integration with APIs, analytics tools, and future feature enhancements.

1. **Key Differences Between Existing and Proposed System**

| **Feature** | **Existing System** | **Proposed System** |
| --- | --- | --- |
| **Real-Time Performance** | Limited scalability, delays in delivery | High scalability with Socket.IO/WebSocket |
| **Chatbot Functionality** | Basic scripts or no integration | AI/NLP-driven chatbot for complex queries |
| **Accessibility** | May require app installation | Fully browser-based, no installation required |
| **Customization** | Limited or not available | Highly customizable for different needs |
| **Security** | Basic encryption | End-to-end encryption with compliance support |

5. SOFTWARE AND HARDWARE REQUIREMENTS

# SOFTWARE and HARDWARE REQUIREMENTS SPECIFICATION

## INTRODUCTION

It is good source of interactivities among students and between the teacher and students. It is done in order to improve student ‘s comprehension levels and learning motivation. As one of their tools, online test tools are quite effective.

In order to solve these problems, we have developed a Web-based online test system which can create quizzes competitively and collaboratively by students for the purpose of reducing the load required for a teacher and promoting interactions among students and between the teacher and students.

**Functional and Non-Functional Requirements**

##### Functional

##### 

Functional requirements define the specific behaviors and functionalities that the

**Real-Time ChatApp** must perform to meet user needs and system objectives. These requirements describe what the system should do.

1. **User Authentication**

* **FR1.1:** The system must allow users to register with an email address, password, and username.
* **FR1.2:** The system must validate user credentials for login using secure authentication (JWT).
* **FR1.3:** The system must allow users to reset their password through a verified email link.
* **FR1.4:** Users must be able to log out from the system securely.

1. **Real-Time Messaging**

* **FR2.1:** The system must support real-time, bi-directional messaging between users using WebSockets or Socket.IO.
* **FR2.2:** Messages must be delivered instantly to the recipient without delays.
* **FR2.3:** Users must be able to send text messages, images, and files in a chat conversation.
* **FR2.4:** The system must notify users of new messages in active or background chats.

1. **Chatbot Integration**

* **FR3.1:** The system must include a chatbot capable of responding to predefined queries automatically.
* **FR3.2:** The chatbot must escalate unresolved queries to a human operator if needed.
* **FR3.3:** The chatbot must store conversation logs for analysis and improvement of response accuracy.

1. **User Interface (UI)**

* **FR4.1:** The system must provide an intuitive, responsive, and easy-to-navigate user interface.
* **FR4.2:** The chat window must display real-time message exchanges, including typing indicators and timestamps.
* **FR4.3:** The system must allow users to customize their display settings, such as themes (dark mode or light mode).

1. **Group Chats and Multi-User Communication**

* **FR5.1:** The system must allow users to create group chats with multiple participants.
* **FR5.2:** Users must be able to invite other users to a group chat through links or user search.
* **FR5.3:** Each group chat must allow file sharing, message reactions, and role-based permissions (e.g., admin, member).

1. **Data Management**

* **FR6.1:** The system must store chat history in a database (MongoDB) for future reference, retrievability, and search.
* **FR6.2:** The system must handle large data loads efficiently, ensuring chat history can be retrieved quickly.
* **FR6.3:** The system must back up user data securely in case of server failure.

1. **Notifications and Alerts**

* **FR7.1:** The system must notify users of incoming messages even if the app is minimized or running in the background.
* **FR7.2:** Users must be able to toggle notifications on/off.
* **FR7.3:** The system must display a "typing..." indicator while a user is composing a message.

1. **User Profile Management**

* **FR8.1:** Users must have the ability to create and update their profile, including their display name, picture, and bio.
* **FR8.2:** Users must be able to change privacy settings (e.g., who can see their profile, who can message them).

##### Non-Functional Requirements

Non-functional requirements describe how the system should perform and its quality attributes, such as performance, security, and usability.

The Non-Functional Requirements in this project are as follow:

1. **Performance**

* **NFR1.1:** The system must support at least 1000 concurrent users without significant performance degradation.
* **NFR1.2:** The average message delivery time must be under 1 second for real-time communication.
* **NFR1.3:** The system must ensure that chat history loads within 3 seconds for a typical conversation.

1. **Scalability**

* **NFR2.1:** The system must be scalable to handle future growth in the number of users, messages, and data storage.
* **NFR2.2:** The backend architecture must allow horizontal scaling for handling increasing user traffic.
* **NFR2.3:** The database must scale to support large volumes of data without performance bottlenecks.

1. **Availability**

* **NFR3.1:** The system must be available 99.9% of the time, excluding scheduled maintenance.
* **NFR3.2:** The system must provide high availability with load balancing for distributed traffic in production.

1. **Security**

* **NFR4.1:** All user data and messages must be encrypted using HTTPS and WebSocket encryption protocols.
* **NFR4.2:** The system must implement secure authentication (JWT) and session management to prevent unauthorized access.
* **NFR4.3:** The system must comply with global data protection regulations (e.g., GDPR, CCPA).
* **NFR4.4:** User passwords must be stored in a hashed format using secure algorithms (e.g., bcrypt).

1. **Usability**

* **NFR5.1:** The system must have an intuitive user interface with minimal learning curve.
* **NFR5.2:** The chat interface must be responsive, adjusting to both mobile and desktop screen sizes.
* **NFR5.3:** The chatbot must provide meaningful, human-like responses to common user queries.

1. **Reliability**

* **NFR6.1:** The system must handle system crashes gracefully, with automatic recovery and minimal data loss.
* **NFR6.2:** The system must provide reliable data synchronization in real-time across different devices and browsers.
* **NFR6.3:** The system must back up chat data and user information regularly to avoid data loss in case of failure.

1. **Compatibility**

* **NFR7.1:** The system must be compatible with all modern web browsers (Chrome, Firefox, Safari, Edge) for optimal user experience.
* **NFR7.2:** The application must be compatible with both desktop and mobile devices.

1. **Maintainability**

* **NFR8.1:** The codebase must be well-documented, and easily maintainable for future updates and bug fixes.
* **NFR8.2:** The system must provide error logging and monitoring to detect and resolve issues quickly.

## HARDWARE AND SOFTWARE REQUIREMENTS

To use this web application the system requires internet connectivity and a web browser, here below are the Hardware and Software requirements as follows:

**Hardware Requirements**

* **RAM:** Minimum 4GB (8GB recommended)
* **Processor:** Intel i3 Processor or Higher
* **Hard Disk :** 20GB and above.
* **Internet**: stable broadband connection.

**Software Requirements**

* Any operating system (Windows, Linux, MAC, etc..).
* Google Chrome/Firefox is recommended for this project for better performance and a good interaction with the end user.
* Browsers and other Internet client access to the web applications.

## FRONT END AND BACK END

#### FRONT END

This Web application is developed with HTML, CSS and JavaScript as frontend, and the front end is user friendly and easy to understand for any user.

**HTML (Hyper Text Markup Language):**

HTML is a markup language used by the browser to manipulate text, images, and other content, in order to display it in the required format. HTML was created by Tim Berners-Lee in 1991.

HTML stands for Hypertext Markup Language. It is used to design web pages using a markup language. HTML is the combination of Hypertext and Markup language. Hypertext defines the link between the web pages. A markup language is used to define the text document within tag which defines the structure of web pages.

**CSS (Cascading Style Sheets):**

CSS (Cascading Style Sheets) is a stylesheet language used to design the webpage to make it attractive. The reason for using this is to simplify the process of making web pages presentable. It allows you to apply styles to web pages. More importantly, it enables you to do this independent of the HTML that makes up each web page.CSS stands for Cascading Style Sheets. It is a style sheet language which is used to describe the look and formatting of a document written in mark-up language. It provides an additional feature to HTML. It is generally used with HTML to change the style of web pages and user interfaces. It can also be used with any kind of XML documents including plain XML, SVG and XUL.

**JAVA-SCRIPT:**

JavaScript is the world most popular lightweight, interpreted compiled programming language. It is also known as scripting language for web pages. It is well-known for the development of web pages; many non-browser environments also use it. JavaScript can be used for Client-side developments as well as Server-side developments. JavaScript is a dynamic programming language that's used for web development, in web applications, for game development, and lots more. It allows you to implement dynamic features on web pages that cannot be done with only HTML and CSS.

JavaScript was first known as **Live Script,** but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name **Live Script**. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers.

**CHAKRA UI:**

Chakra UI is a popular, open-source React component library that provides a set of accessible, reusable, and customizable components for building modern user interfaces (UI). Its primary focus is to make it easy for developers to create beautiful, consistent, and user-friendly UIs with minimal effort. Chakra UI promotes simplicity and accessibility, and it integrates seamlessly with React, making it an excellent choice for developing React applications.

**Key Features of Chakra UI**

1. **Accessible by Default**

Chakra UI ensures that all components follow accessibility best practices. It comes with built-in keyboard navigation, proper focus management, and ARIA attributes, ensuring that your app is usable for everyone, including people with disabilities.

1. **Themeable and Customizable**

Chakra UI offers an easy-to-use theme system.You can define a custom theme for your app, such as colors, typography, spacing, and breakpoints, to maintain designThe theme can be extended or modified to fit your design needs, making it flexible for various use.

1. **Composable Components**

Chakra UI provides a wide range of composable components, such as buttons, modals, forms, tables, icons, and more. These components can be combined to create complex user interfaces while maintaining a consistent design system.

1. **Responsive Design**

Chakra UI supports responsive design out of the box. It uses a simple and powerful media query system, allowing developers to build UIs that automatically adapt to different screen sizes, from mobile devices to desktops.

1. **Styled System Integration**

Chakra UI integrates with Styled System, which provides utility-based styling props. This means you can directly apply styles using props like margin, padding, color, fontSize, display, etc., making it highly efficient to style components without the need for external CSS.

1. **Lightweight**

Chakra UI is lightweight and doesn’t bloat your project. It minimizes the need for extensive external CSS frameworks, giving you only the essential styles and components needed to create your application.

1. **Consistent Design Language**

Chakra UI promotes the use of a consistent design language, which ensures that your app maintains a uniform look and feel. This helps in creating cohesive UIs that are easier for users to navigate and understand.

1. **Dark Mode Support**

Chakra UI has built-in support for dark mode. You can easily switch between light and dark themes, and the components automatically adjust according to the theme, providing a smooth user experience across both light and dark modes.

1. **Easy to Integrate**

Chakra UI can be easily integrated into any React application. The components are simple to use, and the setup process is quick. It also integrates seamlessly with other React libraries and tools.

**Commonly Used Chakra UI Components**

1. **Button**

Chakra UI provides a customizable Button component that can be styled using props like colorScheme, size, variant, and isLoading for loading states.

1. **Box**

The Box component is a basic wrapper that works as a building block for layouts. It can be used as a container for other components and styled using props.

1. **Flex**

The Flex component is a layout utility that uses Flexbox to create flexible layouts. You can easily arrange child components in rows or columns with responsive settings.

1. **Grid**

Chakra UI's Grid component is a powerful layout system based on CSS Grid. It allows you to define columns, rows, and areas to create complex grid layouts.

1. **Input**

The Input component is a customizable input field that can be styled and used for forms. Chakra UI provides various input types, including text, password, and number, along with styling props.

1. **Modal**

The Modal component is used for creating dialog windows, such as popups, alerts, and confirmations. It supports accessibility features and is customizable with header, body, and footer sections.

1. **Icon**

Chakra UI has an extensive library of icons, and you can easily use them in your app with the Icon component. Chakra UI provides easy integration with libraries like react-icons.

1. **Toast**

The Toast component is used for displaying notifications or messages to the user. It can be customized for success, error, or informational alerts.

1. **Heading & Text**

The Heading and Text components allow you to create headers and text elements with customizable styles like font size, weight, and line height.

1. **Form Control**

Chakra UI provides a set of components for building accessible forms, including FormControl, FormLabel, FormHelperText, and FormErrorMessage.

**Use Chakra UI in a React Project:**

**Installation**

npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion

**REACT:**

React is a JavaScript library used for building user interfaces, particularly for single-page applications (SPAs), where you need a fast and dynamic user experience. React was developed by Facebook in 2013 and has since become one of the most popular front-end libraries for building modern web applications.

React focuses primarily on the view layer of the application, making it ideal for rendering dynamic UIs. It is highly efficient, component-based, and declarative, offering a robust development experience.

**KEY FEATURES OF REACT**

1. **Component-Based Architecture**

One of the primary advantages of using React in the ChatApp project is its **component-based architecture**. The entire UI is built using small, reusable, and independent components that can be composed together to form a complex interface.

* **Message Component**: A component to display individual chat messages, which can be text or multimedia.
* **ChatBox Component**: A wrapper component containing the message list and input field.
* **UserList Component**: Displays a list of active users or online users, facilitating communication between them.
* **Login/Signup Components**: Components to handle user authentication (form inputs, validation, etc.).
* **Notification Component**: For displaying real-time notifications about new messages or connection status.

By breaking the app into smaller components, React allows for easier maintenance, testing, and development of individual parts of the ChatApp.

1. **JSX Syntax**

React uses **JSX** (JavaScript XML) to describe what the UI should look like. JSX combines HTML-like syntax with JavaScript logic. It is used to write UI elements in a declarative manner, making the code more readable and maintainable.

1. **Virtual DOM**

React’s **Virtual DOM** is another powerful feature that enhances the performance of the ChatApp. When the state of an application changes (e.g., when a new message is sent or received), React updates the Virtual DOM first. It then compares the updated Virtual DOM with the previous one, finds the differences, and efficiently updates only the necessary parts of the real DOM. This minimizes re-rendering and improves the overall performance of the application.

1. **State and Props Management**

In React, **state** and **props** are two fundamental concepts that enable data flow in an application.

* **State**: React components use state to store data that can change over time. In the ChatApp, state is used to track messages, users, current chat room, and typing indicators.
* **Props**: Props (short for "properties") are used to pass data between parent and child components. For instance, a **ChatBox** component may pass down the messages state as a prop to individual **Message** components to display each message.

#### ****Event Handling****

React handles user interactions through event handlers. In the ChatApp, users interact with components by typing messages, clicking buttons, or navigating the app. React makes it easy to manage events, such as **onClick**, **onChange**, and **onSubmit**, to trigger specific actions or UI updates.

1. **Real-Time Communication with Socket.IO**

For real-time messaging, React integrates with **Socket.IO**, a JavaScript library for real-time web applications. Socket.IO enables bi-directional communication between the server and the client over WebSockets, which is essential for real-time updates in a chat application.

React listens for incoming messages from the server and updates the UI dynamically. Similarly, it emits events to the server when a user sends a message.

1. **Conditional Rendering**

React allows developers to render different components or content conditionally based on the application's state. For example, in a ChatApp, you may want to display a loading spinner while waiting for messages or show a login form if the user isn’t authenticated.

**React Installation**

To get started with React, you can either set up a React project from scratch using **npm (Node Package Manager)** or **Yarn**, or you can use **Create React App**, a tool provided by the React team that simplifies the setup process.

**Install Node.js and npm**:

* Visit the [official Node.js website](https://nodejs.org/) to download the latest version of Node.js, which also includes npm (Node Package Manager).
* After installation, you can verify the installation by running the following commands in your terminal:

node -v

npm -v

**Install React using Create React App**

The easiest and most popular way to start a new React project is by using the **Create React App** tool. It sets up everything for you, including the necessary dependencies and build configuration.

**Steps to Install Create React App**

1. **Install Create React App globally (optional)**  
   Although it's not strictly required anymore, you can install Create React App globally to use it from the command line. However, you can also run it directly using npx (a package runner that comes with npm).

npm install -g create-react-app

1. **Create a New React Application**

Run the following command to create a new React app. Replace <app-name> with your preferred project name.

npx create-react-app <app-name>

Example:

npx create-react-app Talk-Hub

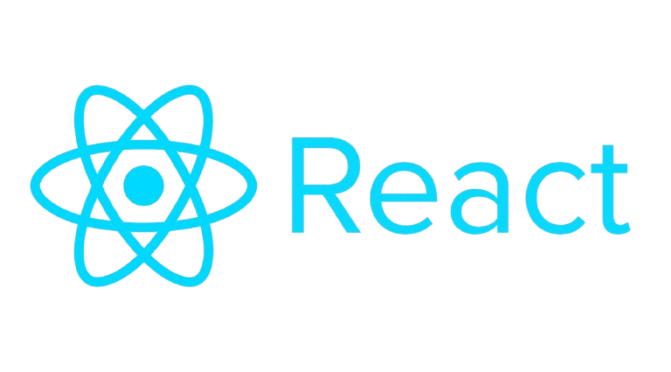
1. **Navigate into the Project Folder**

Once the setup is complete, navigate into your newly created project directory:

cd Talk-Hub

1. **Start the React Development Server**

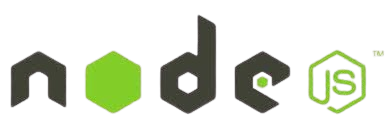
To start the development server and run the React application, use the following command:

 npm start

#### .BACK END

**Node Js:**

Node.js is a JavaScript runtime environment that enables you to run JavaScript code outside of the browser, typically on a server. It is built on the V8 JavaScript engine, the same engine that powers Google Chrome. Node.js is designed for building scalable, high-performance applications, especially those that require real-time communication and are I/O-heavy (e.g., chat applications, APIs, etc.).

Unlike traditional server-side languages, which rely on multi-threading and process for concurrency, Node.js is **single-threaded** and uses an **event-driven, non-blocking I/O model**. This makes Node.js particularly well-suited for applications that need to handle many simultaneous connections with minimal overhead, such as real-time applications, APIs, and microservices.

**Key Features of Node.js**

1. **Single-Threaded, Non-Blocking Architecture**:

* Node.js runs on a single thread using an event-driven architecture, which allows it to handle multiple requests concurrently.
* The **event loop** processes incoming requests asynchronously without blocking other tasks, making it efficient and capable of handling many connections simultaneously.

1. **Asynchronous Programming**:

* In Node.js, **asynchronous programming** is the default. Most of the operations such as reading from a file system or querying a database are non-blocking. This helps Node.js applications to remain responsive and scalable, even under heavy loads.
* This is achieved using callbacks, promises, and async/await syntax in JavaScript.

1. **High Performance (V8 Engine)**:

* Node.js is built on the **Google V8 engine**, which is known for its high performance and fast execution of JavaScript code. The engine compiles JavaScript to native machine code before execution, resulting in faster processing.

1. **Event-Driven Architecture**:

* The event-driven model is at the core of Node.js. It uses an event loop to handle events like HTTP requests, file reads, or database queries. This model allows Node.js to efficiently handle a large number of concurrent connections with a low memory footprint.

1. **Cross-Platform**:

* Node.js is **cross-platform**, meaning it can run on various operating systems, including **Linux**, **macOS**, and **Windows**. This makes it a versatile choice for developers who want to build applications that work across multiple platforms.

1. **Rich Ecosystem (npm)**:

* Node.js uses **npm (Node Package Manager)**, which is the largest ecosystem of open-source libraries and tools. npm makes it easy to install, update, and manage third-party packages for your project.
* npm hosts thousands of libraries for various functionalities, such as web frameworks (Express.js), database clients (Mongoose for MongoDB), authentication (Passport.js), and many more.

1. **Built-In Modules**:

* Node.js provides a range of built-in modules for handling common tasks like file system access (fs), HTTP requests (http), and networking (net). These modules are optimized for non-blocking, event-driven programming.
* Other built-in modules include **os** (operating system-related utilities), **path** (for working with file and directory paths), and **crypto** (for cryptographic operations).

1. **Scalable**:

* Since Node.js is designed to handle many concurrent connections, it's highly scalable. It's ideal for building large-scale applications such as web servers, APIs, and chat applications, where thousands or even millions of users might be simultaneously connected.

1. **Microservices Architecture**:

* Node.js is well-suited for **microservices architectures**, where applications are divided into small, independent services that communicate via APIs. This architecture enables better scalability and fault tolerance for complex applications.

1. **Real-Time Communication**:

* Node.js is highly popular for building **real-time applications**, such as messaging apps, collaborative tools, and online gaming platforms, thanks to the combination of its non-blocking model and WebSocket support.

**Use Cases of Node.js**

1. **Web Servers & APIs**:
   * Node.js is commonly used for building web servers and APIs. Express.js, a minimal web framework, is one of the most popular tools for building server-side applications in Node.js.
   * Node.js is especially suitable for applications with **high I/O** operations and many concurrent users (e.g., APIs with a lot of requests).
2. **Real-Time Applications**:
   * Real-time applications like chat applications, live-streaming platforms, and gaming servers benefit from Node.js due to its ability to handle WebSockets and maintain continuous connections without blocking other processes.
3. **Single-Page Applications (SPAs)**:
   * Node.js is often used in combination with **React**, **Angular**, or **Vue.js** to build SPAs, where the back end serves as an API and provides real-time data to the front end.
4. **Microservices**:
   * Node.js is a great fit for microservices architecture due to its lightweight nature and the ability to build small, independently deployable services that interact with each other.
5. **IoT (Internet of Things)**:
   * Node.js is gaining popularity in the IoT space because of its lightweight, event-driven model that allows it to handle many devices and sensors concurrently.

**Install Socket.io**:

npm install socket.io

**EXPRESS JS:**

**Express.js** (often referred to as **Express**) is a **minimalist, fast, and flexible web application framework** for **Node.js**. It simplifies the process of building robust and scalable web applications and APIs by providing a thin layer of fundamental web application features, without much overhead. Express allows developers to focus on writing their application logic rather than managing boilerplate code, such as HTTP request handling, routing, and middleware integration.

Express.js is one of the most popular web frameworks for Node.js and is used in a wide range of applications, from simple websites to complex APIs and real-time applications like chat apps, e-commerce platforms, and social media applications.

**Key Features of Express.js**

1. **Routing**:

* Express provides a **robust routing system** to define how an application responds to client requests for different URLs.
* Routes can be defined for HTTP methods such as GET, POST, PUT, DELETE, etc.
* It supports dynamic routing and parameters (like /users/:id).

1. **Middleware**:

* Express makes extensive use of **middleware functions**, which are functions that run during the request-response cycle.
* Middleware functions can be used to handle various tasks, such as request validation, authentication, logging, parsing incoming data (JSON, URL-encoded), handling errors, etc.
* Middleware can be global or route-specific.

1. **Request and Response Handling**:

* Express simplifies the request and response handling by providing objects like req (request) and res (response) with built-in methods to access data (like body or query parameters) and send responses (HTML, JSON, or files).
* Express supports **rendering views** with templating engines, such as **EJS**, **Pug**, or **Handlebars**.

1. **Templating Engines**:

* Express supports various templating engines, which allows developers to embed dynamic content into HTML templates and render them as views.
* Common templating engines include **EJS**, **Pug**, and **Handlebars**.

1. **Error Handling**:

* Express allows you to handle errors easily with centralized error-handling middleware.
* If an error occurs during the request-response cycle, Express can pass it to an error handler to generate a structured response for the client.

1. **Support for RESTful APIs**:

* Express makes it easy to build **RESTful APIs** by handling various HTTP methods and status codes.
* It allows easy structuring of APIs by routing incoming requests to the appropriate controller functions.

1. **Asynchronous Handling**:

* As with Node.js, Express fully supports asynchronous request processing, which allows the server to handle many requests concurrently without blocking.

1. **Security Features**:

* Express offers various security features through middleware libraries such as **helmet** (for setting HTTP headers), **express-rate-limit** (for limiting requests), and other security tools for preventing common vulnerabilities like **cross-site scripting (XSS)**, **cross-site request forgery (CSRF)**, and **SQL injection**.

1. **API-First Development**:

* Express is widely used for building RESTful APIs. It supports the development of APIs that can be consumed by mobile apps, frontend applications, or other services.

1. **Static File Serving**:

* Express can serve **static files** (images, CSS, JavaScript) from a specific directory, making it useful for building both the frontend and backend of a web application in one go.

**MongoDB:**

**MongoDB** is a **NoSQL database** that is designed for scalability, performance, and ease of use. Unlike traditional relational databases like MySQL or PostgreSQL, MongoDB stores data in a flexible, document-oriented format. This flexibility makes it well-suited for handling large amounts of unstructured or semi-structured data. It is open-source, and its official website is [mongodb.com](https://www.mongodb.com/).

MongoDB stores data in **JSON-like documents** using a format called **BSON** (Binary JSON). Each document contains key-value pairs that can include arrays, embedded documents, and other types of data.

**Key Features of MongoDB**

1. **Document-Oriented Storage**:

* MongoDB uses **documents** to store data. These documents are represented in **BSON** (Binary JSON) format, making it more flexible compared to the tabular format used by relational databases.
* Each document is a key-value pair and can have nested fields (arrays or objects).

1. **Scalability**:

* **Horizontal Scaling**: MongoDB provides the ability to scale out by sharding data across multiple servers. This allows it to handle large amounts of data and high throughput efficiently.
* Sharding enables the distribution of data across many machines, which enhances performance as the dataset grows.

1. **Flexible Schema**:

* MongoDB allows you to have a **dynamic schema**. This means that each document in a collection can have a different structure. This flexibility is useful in handling varying data models and evolving application requirements.
* You do not need to define the structure of your documents (fields and data types) in advance, unlike relational databases that require a predefined schema.

1. **Indexing**:

* MongoDB supports powerful indexing features, allowing you to index any field in the documents, improving query performance.
* It provides various types of indexes, including **single field**, **compound indexes**, **geospatial indexes**, **text indexes**, and more.

1. **Aggregation Framework**:

* MongoDB provides a powerful **aggregation framework** for performing complex queries, transforming, and processing data within the database.
* Common operations like grouping, filtering, sorting, and joining data can be done using **aggregation pipelines**, which allow you to perform more advanced data manipulation.

1. **Replication**:

* MongoDB supports **replication**, allowing you to have multiple copies of the same data for fault tolerance and high availability.
* A **Replica Set** is a group of MongoDB servers that maintain the same data set, providing redundancy and automatic failover. This ensures that if one server goes down, the system can continue to function with minimal downtime.

1. **Transactions**:

* MongoDB supports multi-document transactions (starting from version 4.0). This means you can perform multiple operations in a single transaction and ensure that either all of them succeed or none of them do, providing **ACID** (Atomicity, Consistency, Isolation, Durability) properties.

1. **Rich Query Language**:

* MongoDB supports a rich query language, including operators for comparison, logical operations, and data-specific operations (e.g., querying arrays, subdocuments).
* MongoDB’s queries can be written in JavaScript, making it easy for developers familiar with the language to use it for querying.

1. **Data Consistency**:

* MongoDB offers **eventual consistency** in a distributed environment but can be configured for **strong consistency** through **write concerns** and **read preferences**.
* This enables MongoDB to support both high availability and consistency, depending on the application’s requirements.

1. **Security**:

* MongoDB offers various security features such as authentication (via username and password), encryption (data at rest and in transit), and role-based access control (RBAC) to restrict access to sensitive data.

**Advantages of MongoDB**

1. **High Performance**:

MongoDB is optimized for high write throughput and can handle large amounts of data with ease.

1. **Flexibility**:

The schema-less nature of MongoDB allows developers to easily modify and update the data structure without downtime.

1. **Scalability**:

MongoDB is horizontally scalable, and its sharding feature allows it to handle large datasets across multiple servers.

1. **Document-Oriented**:

The ability to store documents that can have nested fields and arrays makes MongoDB ideal for working with semi-structured data like JSON.

1. **Real-Time Data**:

MongoDB is suited for real-time applications because it can quickly store and retrieve data, making it ideal for use in chat apps, gaming apps, and more.

1. **Community Support**:

MongoDB has a large community and extensive documentation. This helps developers with troubleshooting and seeking help for complex tasks.

**Disadvantages of MongoDB**

1. **No Joins**:

MongoDB does not support joins like relational databases. While you can use **aggregation** to combine data from different collections, it requires more effort compared to SQL-style joins.

1. **Eventual Consistency**:

In distributed systems, MongoDB offers eventual consistency. While this is great for performance and availability, it may not be suitable for applications that require strong consistency at all times.

1. **Data Duplication**:

The flexibility of MongoDB can lead to data duplication. Without a schema to enforce relationships, developers may accidentally store redundant data.

1. **Memory Usage**:

MongoDB stores data in memory for fast access. In cases of large datasets, this can result in high memory consumption.

**Install MongoDB**

npm i express mongoose mongodb cors

**MongoDB** is a powerful and flexible NoSQL database that is widely used for building scalable, high-performance web applications and APIs. Its document-oriented structure, support for horizontal scaling, and powerful query capabilities make it a popular choice for modern applications that require fast, flexible data storage and retrieval. Whether you're dealing with big data, real-time analytics, or semi-structured content, MongoDB is a great option to consider for your backend storage needs.

**CLOUDINARY:**

Cloudinary is a cloud-based service that provides solutions for managing and optimizing images, videos, and other media assets. It is commonly used in web and mobile applications to handle tasks such as uploading, storing, transforming, and delivering media content in a scalable, efficient, and secure manner.

Cloudinary is designed to handle media content at scale with a wide range of features that automate and simplify the process of dealing with media files. It provides APIs and SDKs for developers to easily integrate media management capabilities into their applications.

**Key Features of Cloudinary**

1. **Media Upload:**

* Cloudinary allows you to upload images, videos, and other media files from various sources (web, mobile, or even directly from a camera) to the cloud.

1. **Image and Video Transformation:**

* Cloudinary provides powerful transformation capabilities that let you manipulate images and videos on the fly, such as resizing, cropping, rotating, adding filters, overlaying text, and more, all via simple URL-based parameters.
* You can perform transformations without changing the original file, which is beneficial for dynamic content delivery.

1. **Media Storage:**

* Cloudinary stores your media in the cloud, allowing you to manage, organize, and retrieve assets at scale. Files are stored on high-performance servers and are backed by a Content Delivery Network (CDN) for fast and reliable delivery.

1. **Automatic Optimization:**

* Cloudinary automatically optimizes images and videos to ensure that they load quickly while maintaining quality. It supports various formats like JPEG, PNG, GIF, WebP, and more.
* It can automatically adjust the quality and format based on the user’s device, browser, and network conditions.

1. **Content Delivery:**

* Cloudinary uses a global Content Delivery Network (CDN) to ensure fast and reliable delivery of media files worldwide. The files are cached and served from the nearest edge server for improved load times.

1. **Asset Management:**

* Cloudinary provides tools for organizing and managing your media assets, including automatic categorization, tagging, and metadata generation.
* It also supports versioning of media files, allowing you to track changes and roll back to previous versions if necessary.

1. **Video Management:**

* In addition to images, Cloudinary offers powerful video management capabilities such as transcoding, streaming, and adaptive bitrate streaming for different devices and network conditions.
* It also provides video optimization features, such as video compression and format conversion.

1. **Advanced Security:**

* Cloudinary provides advanced security features, such as signed URLs and secure media uploads to protect media assets and ensure safe delivery.

**Cloudinary Installation Integration with React.**

npm install react-cloudinary-upload-widget

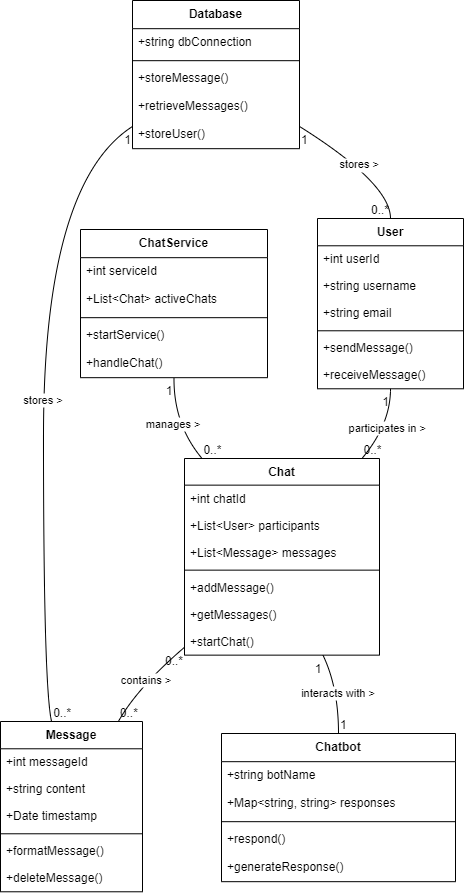
**6.SYSTEM DESIGN**

# SYSTEM DESIGN

### Class Diagram

Class diagrams are a type of UML (Unified Modeling Language) diagram used in software engineering to visually represent the structure and relationships of classes within a system i.e. used to construct and visualize object-oriented systems.

**Here below is the context diagram for the Talk-hub application project :**

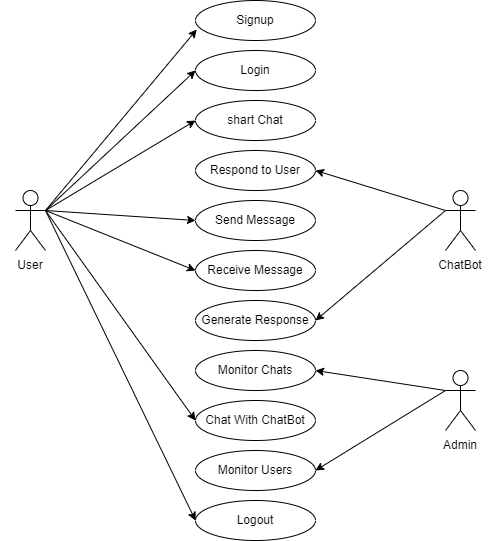


**USE CASE DIAGRAM**

### . Use Case Diagram

Before we start working on any project, it is very important that we are very clear on what we want to do and how do we want to do

use case is a list of actions or event steps, typically defining the interactions between a role (known in the Unified Modeling Language as an actor) and a system, to achieve a goal

Here Bellow is the graphical representation of the project:

**ER DIAGRAM**

### .ER Diagram

The Entity Relational Model is a model for identifying entities to be represented in the database and representation of how those entities are related.

The ER data model specifies enterprise schema that represents the overall logical

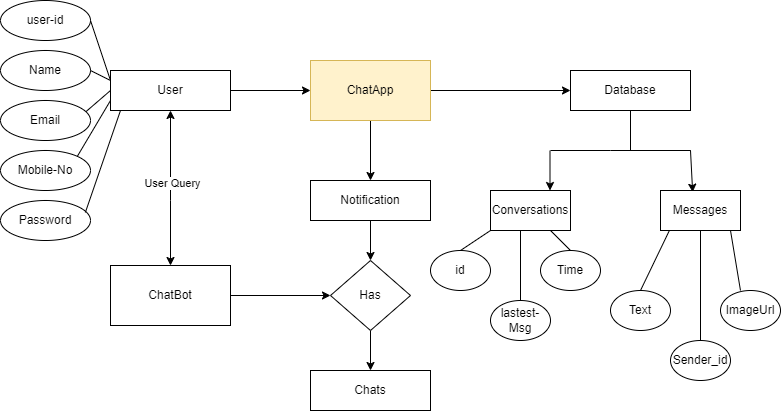
structure of a database graphically.

An Entity Relationship Diagram (ER Diagram) pictorially explains the relationship between entities to be stored in a database. Fundamentally, the ER Diagram is a structural design of the database. It acts as a framework created with specialized symbols for the purpose of defining the relationship between the database entities. ER diagram is created based on three principal components: entities, attributes, and relationships.The following diagram showcases two entities - Student and Course, and their relationship. The relationship described between student and course is many-to-many, as a course can be opted by several students, and a student can opt for more than one course. Student entity possesses attributes - Stu\_Id, Stu\_Name & Stu\_Age. The course entity has attributes such as Cou\_ID & Cou\_Name.

**The prime reasons for using the ER Diagram**

* Helps you to define terms related to entity relationship modeling
* Provide a preview of how all your tables should connect, what fields are going to be on each table
* Helps to describe entities, attributes, relationships
* ER diagrams are translatable into relational tables which allows you to build databases quickly
* ER diagrams can be used by database designers as a blueprint for implementing data in specific software applications
* The database designer gains a better understanding of the information to be contained in the database with the help of ERP diagram

ER DIAGRAM



**Data Flow Diagram**

### 

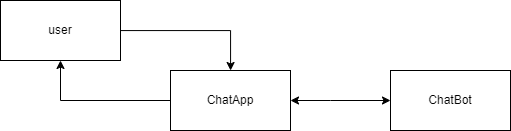
### Data Flow Diagram

Data Flow Diagram (DFD) represents the flow of data within information systems. Data Flow Diagrams (DFD) provide a graphical representation of the data flow of a system that can be understood by both technical and non-technical users. The models enable software engineers, customers, and users to work together effectively during the analysis and specification of requirements

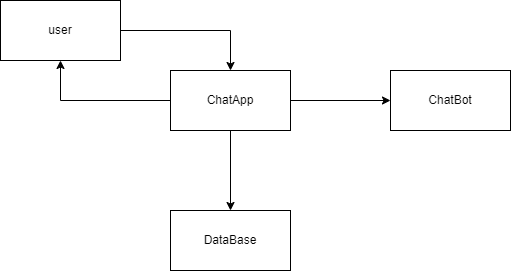
**DATA FLOW DIAGRAM SYMBOL**

|  |  |
| --- | --- |
|  | **Process-**It tells the process state in a project. |
|  | **Data Flow-**It tells the flow of data. |
|  | **Source/Sink-**It is the project source or sink. |
|  | **Data Store-**It stores the data in a particular file. |
|  | **Relationship-**It shows the relationship between function units. |

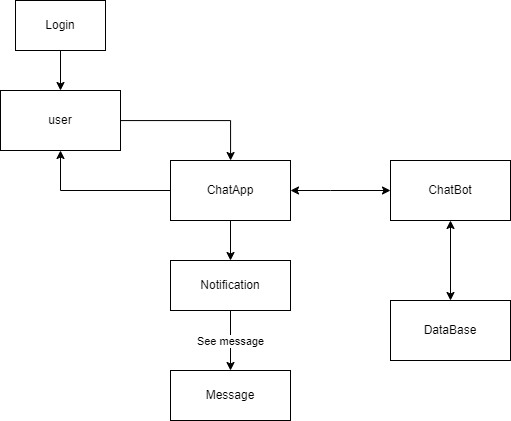
**DFD LEVEL 0**



**DFD LEVEL 1**



**DFD LEVEL 2**



# 7.IMPLIMENTATION

# IMPLIMENTATION

# The implementation of a ChatApp with a chatbot involves setting up both the frontend and backend systems, along with integration for real-time messaging, chatbot functionality, and persistent storage. Below is a structured approach to implement such a system:

### System Architecture

### The architecture of the application involves:

### Frontend: Built using React.js for the user interface.

### Backend: Built using Node.js and Express.js for handling API requests and responses.

### Database: MongoDB for storing user data, chat messages, and other necessary information.

### Chatbot: Implemented using AI libraries (e.g., Dialogflow, OpenAI GPT API, or custom NLP algorithms).

### Real-time Communication: Handled using Socket.IO for WebSocket-based messaging.

### Media Management: Managed using Cloudinary for image/video uploads (optional).

### Frontend Implementation

**Key Steps in Frontend Development (React.js)**

**Initialize the React app:**

npx create-react-app Talk-hub

cd Talk-Hub

npm start

**UI Components:**

* **Login/Register Page**: Form for users to log in or create accounts.
* **Chat Interface**:
* List of chat contacts.
* Chat messages with a scrollable interface.
* Input field for sending messages.
* **Chatbot Integration**: A chatbot interface within the chat window.

**React Libraries**:

* Use **Chakra UI** or **Material-UI** for styled components.
* Use **React-Router** for navigation between login, chat, and profile pages.

**Socket.IO Integration**:

npm install socket.io-client

### Backend Implementation

### Setup Backend with Node.js and Express.js

1. **Initialize the Backend**:

* Create a Node.js project

mkdir backend

cd backend

npm init -y

npm install express mongoose dotenv socket.io

1. **API Endpoints**:

* **Authentication**:
* Register new users: POST /api/auth/register
* Log in users: POST /api/auth/login
* **Chat Management**:
* Get chat history: GET /api/chats/:chatId
* Save new messages: POST /api/chats/:chatId/messages
* **Chatbot API**:
* Communicate with the chatbot: POST /api/chatbot/response

1. **Socket.IO for Real-Time Messaging**:

* Setup WebSocket communication in the backend:

const io = require("socket.io")(server, {

cors: { origin: "http://localhost:3000", methods: ["GET", "POST"] },

});

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

console.log("A user connected:", socket.id);

socket.on("sendMessage", (data) => {

io.emit("receiveMessage", data); // Broadcast message to all users

});

socket.on("disconnect", () => {

console.log("A user disconnected:", socket.id);

});

});

1. **Database Implementation**

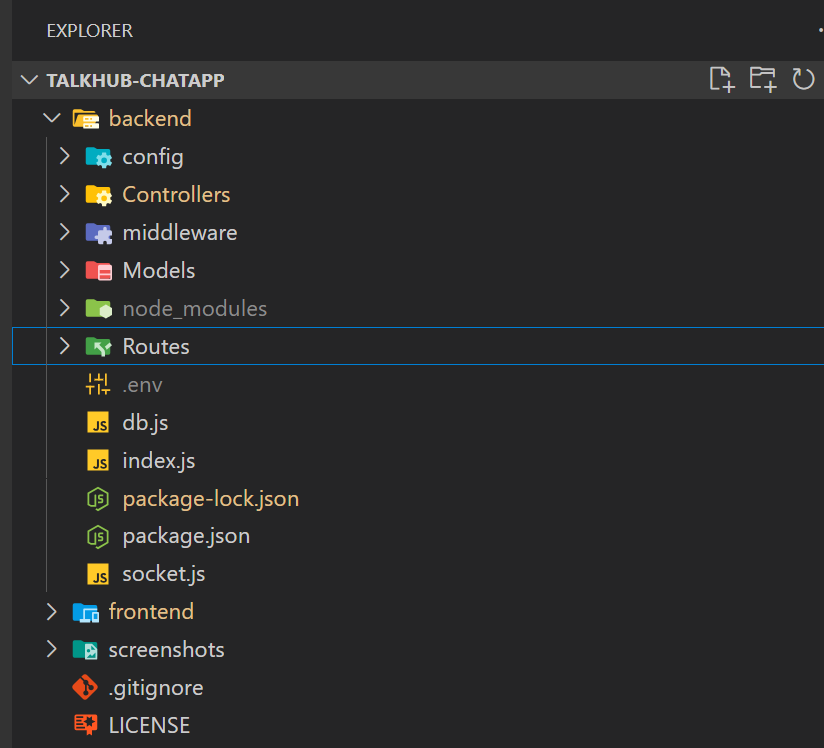
**MongoDB Setup:**

* **Install MongoDB:**
  + Locally or use a cloud database like MongoDB Atlas.
  + Install mongoose for connecting to MongoDB:

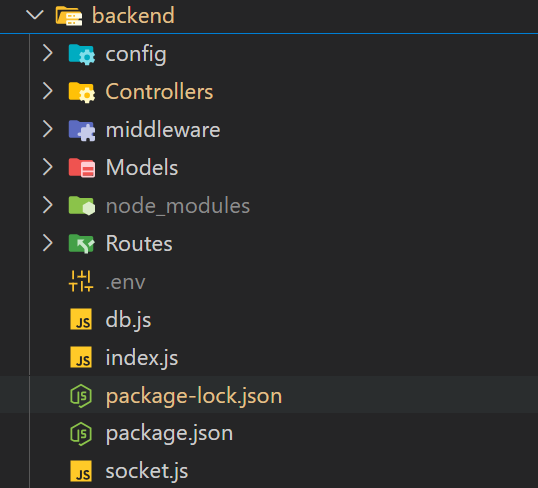
npm install mongoose

# 8.CODES AND SCREENSHOTS

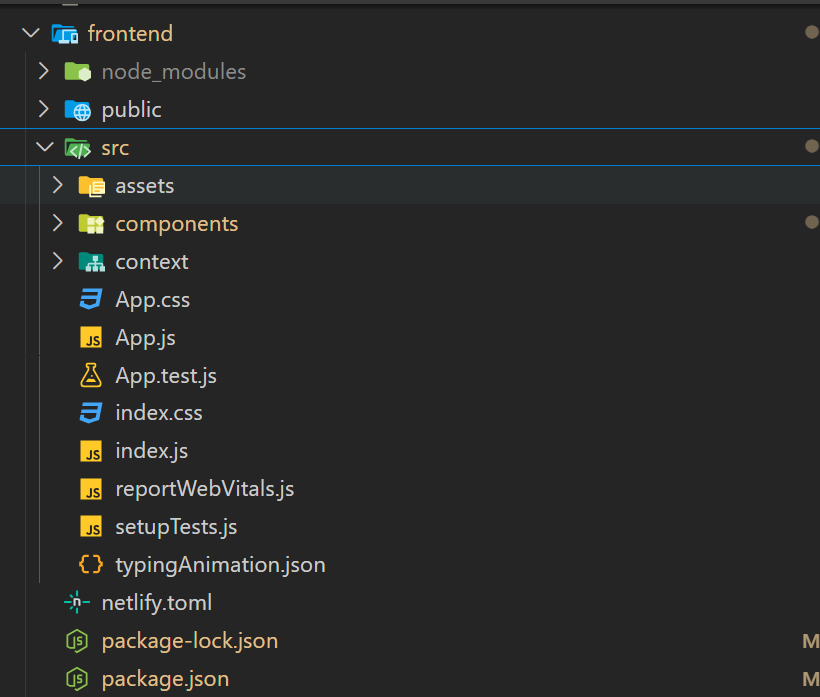
### CODES



The above image has templates folder of the project where it contain Frontend and Backend files all the dependency file of the project.



The above image of Backend important file

****

The above image of Frontend important file

**BACKEND CODE**

* **Index.Js**

const express = require("express");

const connectDB = require("./db.js");

const cors = require("cors");

const http = require("http");

const PORT = 5000;

const app = express();

app.use(cors());

app.use(express.urlencoded({ extended: true, limit: "50mb" }));

app.use(express.json({ limit: "50mb" }));

*// Routes*

app.get("/", (req, res) => {

  res.send("Hello World");

});

app.use("/user", require("./Routes/auth\_routes.js"));

app.use("/message", require("./Routes/message\_routes.js"));

app.use("/conversation", require("./Routes/conversation\_routes.js"));

*// Server setup*

const server = http.createServer(app);

*// Socket.io setup*

require("./socket.js")(server); *// Initialize socket.io logic*

*// Start server and connect to database*

server.listen(PORT, () => {

  console.log(`🚀 Server started at http://localhost:${PORT}`);

  connectDB();

});

##### Auth-controller.js

const User = require("../Models/User.js");

const bcrypt = require("bcryptjs");

const jwt = require("jsonwebtoken");

const Conversation = require("../Models/Conversation.js");

const ObjectId = require("mongoose").Types.ObjectId;

const cloudinary = require("cloudinary").v2;

const imageupload = require("../config/imageupload.js");

const nodemailer = require("nodemailer");

const dotenv = require("dotenv");

dotenv.config({ path: '../.env' });

const JWT\_SECRET = process.env.JWT\_SECRET;

let mailTransporter = nodemailer.createTransport({

  service: "gmail",

  auth: {

    user: process.env.EMAIL,

    pass: process.env.PASSWORD,

  },

});

cloudinary.config({

  cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,

  api\_key: process.env.CLOUDINARY\_API\_KEY,

  api\_secret: process.env.CLOUDINARY\_API\_SECRET,

});

const register = async (req, res) => {

  try {

    console.log("register request received");

    const { name, email, password, phoneNum } = req.body;

    if (!name || !email || !password || !phoneNum) {

      return res.status(400).json({

        error: "Please fill all the fields",

      });

    }

    const user = await User.findOne({

      email: email,

    });

    if (user) {

      return res.status(400).json({

        error: "User already exists",

      });

    }

    var imageUrl = "";

    if (req.file != null) {

      imageUrl = await imageupload(req.file);

    } else {

      imageUrl = `https://ui-avatars.com/api/?name=${name}&background=random&bold=true`;

    }

*// on render doesnt allow to store images locally hence*

*// imageUrl = `https://ui-avatar.com/api/?name=${name}&background=random&bold=true`;*

    const salt = await bcrypt.genSalt(10);

    const secPass = await bcrypt.hash(password, salt);

    const newUser = new User({

      name,

      email,

      password: secPass,

      phoneNum,

      profilePic: imageUrl,

    });

    await newUser.save();

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

    var id = us.\_id.toString().substring(0, 21) + "/ch";

    id = Buffer.from(id, "utf8").toString("hex");

    us.\_id = new ObjectId();

    us.name = "AI Chatbot";

    us.email = email + "bot";

    us.phoneNum = "0000000000";

    us.profilePic =

"https://play-lh.googleusercontent.com/Oe0NgYQ63TGGEr7ViA2fGA-yAB7w2zhMofDBR3opTGVvsCFibD8pecWUjHBF\_VnVKNdJ";

    await User.insertMany(us);

    const bot = await User.findOne({ email: email + "bot" });

    const newConversation = new Conversation({

      members: [newUser.\_id, bot.\_id],

    });

    await newConversation.save();

    const data = {

      user: {

        id: newUser.id,

      },

    };

    const authtoken = jwt.sign(data, JWT\_SECRET);

    res.json({

      authtoken, });

} catch (error) {

    console.error(error.message);

    res.status(500).send("Internal Server Error");

  }};

const login = async (req, res) => {

  console.log("login request received");

  try {

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

    if (!email || (!password && !otp)) {

      return res.status(400).json({

        error: "Please fill all the fields",

      });}

    const user = await User.findOne({

      email: email,

    });

    if (!user) {

      return res.status(400).json({

        error: "Invalid Credentials",

      });

    }

    console.log("hii");

    if (otp) {

      if (user.otp != otp) {

        return res.status(400).json({

          error: "Invalid otp",

        });

      }

      user.otp = "";

      await user.save();

    } else {

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

      if (!passwordCompare) {

  return res.status(400).json({

          error: "Invalid Credentials",

        });

      } }

    const data = {

      user: {

        id: user.id,

      }, };

    const authtoken = jwt.sign(data, JWT\_SECRET);

    res.json({

      authtoken,

      user: {

        \_id: user.id,

        name: user.name,

        email: user.email,

        phoneNum: user.phoneNum,

        profilePic: user.profilePic,

      },

    });

  } catch (error) {

    console.error(error.message);

    res.status(500).send("Internal Server Error");

  }

};

const authUser = async (req, res) => {

  const token = req.header("auth-token");

  if (!token) {

    res.status(401).send("Please authenticate using a valid token");

  }

  try {

    const data = jwt.verify(token, JWT\_SECRET);

    if (!data) {

      return res.status(401).send("Please authenticate using a valid token");}

    const user = await User.findById(data.user.id).select("-password");

    res.json(user);

  } catch (error) {

    console.error(error.message);

    res.status(500).send("Internal Server Error");

  }};

const allUser = async (req, res) => {

  const requser = req.user

  try {

    const users = await User.find().select("-password");

*//filter users such bot for other users are not included*

    users.forEach((user) => {

      if (user.email.includes("bot") && user.email != requser.email) {

        users.splice(users.indexOf(user), 1);

      } });

    res.json(users);

  } catch (error) {

    res.status(500).send("Internal Server Error");

  }};

const updateprofile = async (req, res) => {

  try {

    const dbuser = await User.findById(req.user.id);

    if (req.body.newpassword) {

      const passwordCompare = await bcrypt.compare(

        req.body.oldpassword,

        dbuser.password

      );

      if (!passwordCompare) {

        return res.status(400).json({

          error: "Invalid Credentials",})}

      const salt = await bcrypt.genSalt(10);

      const secPass = await bcrypt.hash(req.body.newpassword, salt);

      req.body.password = secPass;

      delete req.body.oldpassword;

      delete req.body.newpassword;

    }

    await User.findByIdAndUpdate(req.user.id, req.body);

    res.status(200).json({ message: "Profile Updated" });

  } catch (error) {

    res.status(500).send("Internal Server Error");

  }

};

const sendotp = async (req, res) => {

  try {

    console.log("sendotp request received");

    const { email } = req.body;

    const user = await User.findOne({ email: req.body.email });

    if (!user) {

      return res.status(400).json({

        error: "User not found",

      });

    }

    const otp = Math.floor(100000 + Math.random() \* 900000);

    user.otp = otp;

    await user.save();

*//delete otp after 5 minutes*

    setTimeout(() => {

      user.otp = "";

      user.save();

    }, 300000);

    let mailDetails = {

      from: process.env.EMAIL,

      to: email,

      subject: "Login with your Otp",

      html: `<!DOCTYPE html>

      <html lang="en">

      <head>

          <title>Otp for Login</title>

          <style>

              .container {

                  width: 50%;

                  margin: 0 auto;

                  background: #f4f4f4;

                  padding: 20px;

              } h1 {  text-align: center}

          </style>

      </head>

      <body>

              <strong><h1>Talk-Hub - online chatting app</h1></strong>

          <div class="container">

              <h2>Your Otp is</h2>

              <strong><p>${otp}</p><strong>

              <p>Use this Otp to login</p>

          </div>

      </body>

      </html>`,}

    await mailTransporter.sendMail(mailDetails, function (err, data) {

      if (err) {

        console.log("Error Occurs", err);

        res.status(400).json({ message: "Error Occurs" });

      } else {

        console.log("Email sent successfully");

  res.status(200).json({ message: "OTP sent" });

      });

  } catch (error) {

    console.error(error.message);

    res.status(500).send("Internal Server Error");

  }

};

module.exports = {

  register,

  login,

  allUser,

  authUser,

  updateprofile,

  sendotp,

};

**The code above is used for the signup validation purpose**

* **Shocket.io .js code:**

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

##### const Conversation = require("./Models/Conversation.js");

##### const { generateairesponse } = require("./Controllers/message\_controller.js");

##### const Message = require("./Models/Message.js");

##### module.exports = (server) => {

##### const io = socketio(server, {

##### pingTimeout: 60000,

##### cors: {

##### origin: "\*",

##### },

##### });

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

##### console.log("A user connected");

##### // Setup user in a room

##### socket.on("setup", async (id) => {

##### socket.join(id);

##### console.log("User joined room", id);

##### socket.emit("user setup", id);

##### //find all conversations of user and send other user online event

##### const conv = await Conversation.find({

##### members: { $in: [id] },

##### });

##### conv.forEach((conversation) => {

##### const sock = io.sockets.adapter.rooms.get(conversation.id);

##### if (sock) {

##### console.log("Other user is online is sent to: ", id);

##### io.to(conversation.id).emit("other-user-online", {});

##### }});});

##### // Join chat room

##### socket.on("join-chat", async (data) => {

##### const conv = await Conversation.findById(data.room);

##### const userIndex = conv.members.findIndex((user) => user == data.user);

##### conv.unread[userIndex] = 0;

##### await conv.save({ timestamps: false });

##### socket.join(data.room);

##### const otheruser = conv.members.filter((user) => user != data.user);

##### //find if other user is online

##### const otheruserSocket = io.sockets.adapter.rooms.get(

##### otheruser.toString() );

##### console.log("user is:", data.user);

##### io.to(data.room).emit("user-joined-room", data.user);

##### if (otheruserSocket) {

##### console.log("Other user is online is sent to: ", data.user);

##### io.to(data.room).emit("other-user-online", {});

##### } });

##### // Leave chat room

##### socket.on("leave-chat", (room) => {

##### socket.leave(room);

##### });

##### // Send message

##### socket.on("send-message", async (data) => {

##### console.log("Received message: ");

##### const conversation = await Conversation.findById(

##### data.data.conversationId

##### ).populate("members", "-password");

##### conversation.latestmessage = data.data.text;

##### var isbot = false;

##### const members = conversation.members.map((member) =>

##### member.id.toString()

##### );

##### members.forEach((member) => {

##### if (member != data.data.sender) {

##### const memberroom = io.sockets.adapter.rooms.get(member);

##### if (memberroom) {

##### const sid = Array.from(memberroom)[0];

##### const chatroom = io.sockets.adapter.rooms.get(

##### data.data.conversationId

##### );

##### if (chatroom && chatroom.has(sid)) {

##### data.data.seenby = [member];

##### }

##### }

##### }

##### });

##### console.log("Members: ", data.data.seenby);

##### 

##### //find if receiver is in the room or not

##### io.to(data.data.conversationId).emit("receive-message", data);

##### conversation.members.forEach(async (member) => {

##### if (member != data.data.sender && member.email.includes("bot")) {

##### // this member is a bot

##### isbot = true;

##### // send typing event

##### io.to(data.data.conversationId).emit("typing", data);

##### // generating AI response

##### var airesponse = await generateairesponse(data.data);

##### const aidata = {

##### data: {

##### conversationId: data.data.conversationId,

##### sender: member,

##### text: airesponse,

##### createdAt: new Date(),

##### },};

##### console.log("AI response: ", aidata);

##### // send stop typing event and send ai response as message

##### io.to(data.data.conversationId).emit("receive-message", aidata);

##### io.to(data.data.conversationId).emit("stop-typing", data);

##### await conversation.save();

##### }

##### });

##### // for two normal users

##### if (!isbot) {

##### const ids = conversation.members.map((member) => member.id.toString());

##### conversation.unread = conversation.unread.map((unread, index) => {

##### if (

##### index !== ids.indexOf(data.data.sender) &&

##### io.sockets.adapter.rooms.get(data.data.conversationId).size === 1

##### ) {

##### return unread + 1;

##### }

##### return unread;

##### });

##### // console.log("sender", data.data.sender);

##### // console.log("conversation", conversation);

##### //new message to all members except sender

##### conversation.members.forEach((member) => {

##### if (member.id.toString() !== data.data.sender) {

##### data.data["conversation"] = conversation;

##### console.log("Emitting new message to: ", member.id);

##### io.to(member.id.toString()).emit("new-message", data.data);

##### }});

##### await conversation.save();

##### } });

##### // Typing indicator

##### socket.on("typing", (data) => {

##### io.to(data.conversationId).emit("typing", data);

##### });

##### // Stop typing indicator

##### socket.on("stop-typing", (data) => {

##### io.to(data.conversationId).emit("stop-typing", data);

##### });

##### // Disconnect

##### socket.on("disconnect", () => {

##### console.log("A user disconnected");

##### });});};

* **FetchUser.js Backend Code:**

const jwt = require("jsonwebtoken");

const env = require("dotenv");

env.config({

path: "../../.env",

});

const JWT\_SECRET = process.env.JWT\_SECRET;

const fetchuser = (req, res, next) => {

const token = req.header("auth-token");

if (!token) {

res.status(401).send("Please authenticate using a valid token");

} else {

try {

const data = jwt.verify(token, JWT\_SECRET);

req.user = data.user;

next();

} catch (error) {

res.status(401).send("Please authenticate using a valid token");

}

}

};

module.exports = fetchuser;

* **Database.Js Backend Code:**

const mongoose = require("mongoose");

const dotenv = require("dotenv");

dotenv.config({ path: "./.env" })

const connectDB = async () => {

try {

// const conn = await mongoose.connect(process.env.MONGO\_URI, {

// dbName: "chatapp",

// });

const conn = await mongoose.connect("mongodb://127.0.0.1:27017/chatapp")

console.log(`MongoDB connected !`);

} catch (error) {

console.error(`Error: ${error.message}`);

process.exit(1);

}

};

module.exports = connectDB;

**The below code is the dependency script**

* **Package.json:**

##### {

##### "name": "backend",

##### "version": "1.0.0",

##### "description": "",

##### "main": "index.js",

##### "scripts": {

##### "test": "echo \"Error: no test specified\" && exit 1",

##### "dev": "nodemon index.js",

##### "start": "node index.js"

##### },

##### "author": "",

##### "license": "ISC",

##### "dependencies": {

##### "@google-cloud/vertexai": "^0.5.0",

##### "@google/generative-ai": "^0.2.1",

##### "@types/body-parser": "^1.19.5",

##### "@types/express": "^4.17.21",

##### "@types/node": "^20.11.25",

##### "bcryptjs": "^2.4.3",

##### "cloudinary": "^2.0.3",

##### "cors": "^2.8.5",

##### "dotenv": "^16.4.5",

##### "express": "^4.18.2",

##### "jsonwebtoken": "^9.0.2",

##### "mongodb": "^6.10.0",

##### "mongoose": "^8.1.3",

##### "multer": "^1.4.5-lts.1",

##### "nodemailer": "^6.9.12",

##### "nodemon": "^3.0.3",

##### "os": "^0.1.2",

##### "socket.io": "^4.7.4"

##### },

##### "devDependencies": {

##### "ts-node": "^10.9.2"

##### }}

* **Message\_Controller:**

const Message = require("../Models/Message.js");

const Conversation = require("../Models/Conversation.js");

const { GoogleGenerativeAI } = require("@google/generative-ai");

const imageupload = require("../config/imageupload.js");

const dotenv = require("dotenv");

dotenv.config({ path: "./.env" });

const configuration = new GoogleGenerativeAI(process.env.GENERATIVE\_API\_KEY);

const modelId = "gemini-1.5-pro";

const model = configuration.getGenerativeModel({ model: modelId });

const sendMessage = async (req, res) => {

var imageurl = "";

if (req.file) {

imageurl = await imageupload(req.file, false);

} try {

const { conversationId, sender, text } = req.body;

if (!conversationId || !sender || !text) {

return res.status(400).json({

error: "Please fill all the fields",});}

const conversation = await Conversation.findById(conversationId).populate(

"members",

"-password"

);

//check if conversation contains bot

var isbot = false;

conversation.members.forEach((member) => {

if (member != sender && member.email.includes("bot")) {

isbot = true;

}

});

if (!isbot) {

const newMessage = new Message({

conversationId,

sender,

text,

imageurl,

seenby: [sender],

});

await newMessage.save();

console.log("newMessage saved");

conversation.updatedAt = new Date();

await conversation.save();

res.json(newMessage);

}

} catch (error) {

res.status(500).send("Internal Server Error");

}

};

const allMessage = async (req, res) => {

try {

const messages = await Message.find({

conversationId: req.params.id, });

messages.forEach((message) => {

if (!message.seenby.includes(req.params.userid)) {

console.log("message", message.text);

console.log("message.seenby", message.seenby);

message.seenby.push(req.params.userid);

message.save();

} });

res.json(messages);

} catch (error) {

console.error(error.message);

res.status(500).send("Internal Server Error");

}};

const deletemesage = async (req, res) => {

const msgid = req.body.messageid;

const userids = req.body.userids;

try {

const message = await Message.findById(msgid);

userids.forEach(async (userid) => {

if (!message.deletedby.includes(userid)) {

message.deletedby.push(userid);

}});

await message.save();

res.status(200).send("Message deleted successfully");

} catch (error) {

console.log(error.message);

res.status(500).send({ error: "Internal Server Error" });

}};

const generateairesponse = async (data) => {

var currentMessages = [];

const prompt = data.text;

const user = data.sender;

const conv = await Conversation.findById(data.conversationId);

const bot = conv.members.find((member) => member != user);

const messagelist = await Message.find({

conversationId: data.conversationId,

});

messagelist.forEach((message) => {

if (message.sender == user) {

currentMessages.push({

role: "user",

parts: message.text,

});} else {

currentMessages.push({

role: "model",

parts: message.text,

});}}); try {

const chat = model.startChat({

history: currentMessages,

generationConfig: {

maxOutputTokens: 1000,

}, });

const result = await chat.sendMessage(prompt);

const response = result.response;

var responseText = response.text();

if (responseText.length < 1) {

responseText = "Woops!! thats soo long ask me something in short."; }

// Stores the conversation

Message.insertMany([

{

conversationId: data.conversationId,

sender: user,

text: prompt, },

{

conversationId: data.conversationId,

**FRONTEND CODE:**

* **App.js:**

import "./App.css";

import { useColorMode } from "@chakra-ui/react";

import Navbar from "./components/Navbar/Navbar";

import ChatState from "./context/chatState";

import { useContext } from "react";

import chatContext from "./context/chatContext";

function App(props) {

const { colorMode, toggleColorMode } = useColorMode();

const context = useContext(chatContext);

// localStorage.removeItem("token")

return (

<ChatState>

<div className="App">

<Navbar toggleColorMode={toggleColorMode} context={context} />

</div>

</ChatState>

);}

export default App;

* **Login.js:**

import { useState, useContext } from "react";

import {Flex,Heading,Input,Button,InputGroup,Stack,InputLeftElement,chakra,Box,

Link,Avatar,FormControl,FormHelperText,InputRightElement,Card,CardBody,

useToast,Spinner,Tooltip,} from "@chakra-ui/react";

import { FaLock } from "react-icons/fa";

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

import chatContext from "../../context/chatContext";

import { ArrowBackIcon } from "@chakra-ui/icons";

const CFaLock = chakra(FaLock);

const Login = (props) => {

const context = useContext(chatContext);

const toast = useToast();

const navigator = useNavigate();

const [email, setemail] = useState();

const [password, setpassword] = useState();

const handletabs = props.handleTabsChange;

const [showPassword, setShowPassword] = useState(false);

const [forgotpasswordshow, setforgotpasswordshow] = useState(false);

const [sending, setsending] = useState(false);

const handleShowClick = () => setShowPassword(!showPassword);

const showtoast = (title, description, status) => {

toast({

title: title,

description: description,

status: status,

duration: 5000,

isClosable: true,

});};

const handleLogin = async function (e) {

e.preventDefault();

const data = {

email: email,

};

//check if the user is trying to login using otp

const otp = document.getElementById("otp")?.value;

if (otp?.length > 0 && forgotpasswordshow) {

data.otp = otp;

} else {

data.password = password;

}

try {

const response = await fetch(`${context.ipadd}/user/login`, {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify(data),

});

const resdata = await response.json();

if (response.status !== 200) {

showtoast("An error occurred.", resdata.error, "error");

} else {

showtoast("Login successful", "You are now logged in", "success");

localStorage.setItem("token", resdata.authtoken);

context.setuser(await resdata.user);

context.socket.emit("setup", await resdata.user.\_id);

console.log("setup reques sent for", await resdata.user.\_id);

context.setisauthenticated(true);

context.fetchData();

navigator("/dashboard");

}

} catch (error) {

console.log(error);

}

};

const handlesendotp = async (e) => {

e.preventDefault();

setsending(true);

const data = {

email: email,

};

try {

const response = await fetch(`${context.ipadd}/user/getotp`, {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify(data),

});

const resdata = await response.json();

setsending(false);

if (response.status !== 200) {

showtoast("An error occurred.", resdata.error, "error");

} else {

showtoast("otp sent", "otp sent to your email", "success");

}

} catch (error) {

console.log(error);

}};

return (

<FlexflexDirection="column"width="100wh"height="70vh"justifyContent="center"alignItems="center"borderRadius={15}>

<StackflexDir="column"mb="2"justifyContent="center"alignItems="center">

<Avatar bg="purple.300" />

<Heading color="pruple.400">Welcome Back</Heading>

<Card minW={{ base: "90%", md: "468px" }} borderRadius={15} shadow={0}>

<CardBody p={0}>

<form>

<Stack spacing={4}>

{forgotpasswordshow && (

<Tooltip label="login" aria-label="A tooltip">

<Button

w={"fit-content"}

onClick={() => setforgotpasswordshow(false)}>

<ArrowBackIcon />

</Button>

</Tooltip>

)}

<FormControl display={"flex"}>

<InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"}

size={"lg"}>

<Input id="login-email"type="email"placeholder="email address"

focusBorderColor="purple.500"onChange={(e) => setemail(e.target.value)}/>

</InputGroup>

{forgotpasswordshow && (

<Button m={1}fontSize={"sm"}onClick={(e) => handlesendotp(e)}>

{sending ? <Spinner size="sm" /> : "Send otp"}

</Button>

)}

</FormControl>

{!forgotpasswordshow && (

<FormControl>

<InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"}

size={"lg"}>

<InputLeftElement

pointerEvents="none"

color="gray.300"

children={<CFaLock color="gray.300" />}/>

<Input

type={showPassword ? "text" : "password"}

placeholder="Password"

focusBorderColor="purple.500"

onChange={(e) => setpassword(e.target.value)}/>

<InputRightElement mx={1}>

<Buttonn fontSize={"x-small"} size={"xs"}onClick={handleShowClick}>

{showPassword ? "Hide" : "Show"}</Button>

</InputRightElement>

</InputGroup>

<FormHelperText textAlign="right">

<Link onClick={() => setforgotpasswordshow(true)}>

forgot password?

</Link>

</FormHelperText>

</FormControl>

)}

{forgotpasswordshow && (

<FormControl>

<InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"}

size={"lg"}>

<Input

id={"otp"}

type="number"

placeholder="enter otp"

focusBorderColor="purple.500"/>

</InputGroup>

</FormControl>

)}

<Button

borderRadius={10}

type="submit"

variant="solid"

colorScheme="purple"

width="full"

onClick={handleLogin}>

{forgotpasswordshow ? "Login using otp" : "Login"}

</Button>

</Stack>

</form>

</CardBody>

</Card>

</Stack>

<Box>

New to us?{" "}

<Link color="purple.500" onClick={() => handletabs(1)}>

Sign Up

</Link>

</Box>

</Flex>);};

export default Login;

* **Signup.js:**

import chatContext from "../../context/chatContext";

import { useState, useContext, useRef } from "react";

import {Flex,Heading,Input,Button,InputGroup,Stack,InputLeftElement,Box,

Link,Avatar,FormControl,InputRightElement,Card,CardBody,useToast,Text,}from "@chakra-ui/react";

import { LockIcon } from "@chakra-ui/icons";

const Signup = (props) => {

const context = useContext(chatContext);

const fileInputRef = useRef(null);

const toast = useToast();

const [showPassword, setShowPassword] = useState(false);

const [name, setname] = useState("");

const [email, setemail] = useState("");

const [phoneNum, setphoneNum] = useState("");

const [password, setpassword] = useState("");

const [confirmpassword, setconfirmpassword] = useState("");

const [profilePic, setprofilePic] = useState("");

const handletabs = props.handleTabsChange;

function showtoast(description) {

toast({

title: "An error occurred.",

description: description,

status: "error",

duration: 5000,

isClosable: true, });}

<Flex flexDirection="column"

width="100%"

height="75vh"

justifyContent="center"

alignItems="center"

borderRadius={15} >

<Stack

flexDir="column"

mb="2"

justifyContent="center"

alignItems="center">

<Avatar bg="purple.300" />

<Heading color="pruple.400">Welcome</Heading>

<Card minW={{ base: "90%", md: "465px" }} borderRadius={15} shadow={0}>

<CardBody p={0}>

<form>

<Stack spacing={2}>

<FormControl>

<InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"}

size={"lg"}

<Input type="text"

placeholder="Enter your name"

focusBorderColor="purple.500"

onChange={(e) => setname(e.target.value)} required />

</InputGroup>

</FormControl>

<FormControl>

<InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"}

size={"lg"} >

<Input type="email" placeholder="Email address"focusBorderColor="purple.500"

onChange={(e) => setemail(e.target.value)}/>

</InputGroup>

</FormControl>

<FormControl>

<InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"}

size={"lg"} >

<Input

type="number"

placeholder="Phone Number"

focusBorderColor="purple.500"

onChange={(e) => setphoneNum(e.target.value)}/>

</InputGroup>

</FormControl>

<FormControl>

<InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"}

size={"lg"} >

<InputLeftElement

pointerEvents="none"

color="gray.300"

children={<LockIcon color="gray.300" />}/>

<Input type={showPassword ? "text" : "password"}

placeholder="Password"

focusBorderColor="purple.500"

onChange={(e) => setpassword(e.target.value)}/>

<InputRightElement mx={1}>

<Button fontSize={"x-small"}

size={"xs"} onClick={handleShowClick}

>{showPassword ? "Hide" : "Show"}

</Button>

</InputRightElement>

</InputGroup> <InputGroup

borderEndRadius={"10px"}

borderStartRadius={"10px"} size={"lg"} my={4}>

<InputLeftElement

pointerEvents="none"

color="gray.300"

children={<LockIcon color="gray.300" />/>

<Input

textOverflow={"ellipsis"}

type={showPassword ? "text" : "password"}

placeholder="Confirm Password"

focusBorderColor="purple.500"

onChange={(e) => setconfirmpassword(e.target.value)} />

<InputRightElement mx={1}>

<Button

fontSize={"x-small"}

size={"xs"}

onClick={handleShowClick}>

{showPassword ? "Hide" : "Show"}

</Button> </InputRightElement>

</InputGroup>

<Flex align="center">

{!profilePic && ( <Text mx={2} fontSize="sm"> Upload Profile Picture

</Text> ><Text>Remove</Text></Button> )}

</Flex>

</FormControl>

<Button

borderRadius={10}

type="submit"

variant="solid" colorScheme="purple" width="full" onClick={handleSignup} > Signup </Button>

</Stack>

</form>

</CardBody>

</Card>

</Stack>

<Box>

Already have account?{" "}

<Link color="purple.500" onClick={() => handletabs(0)}>

login </Link></Box></Flex> )};

export default Signup;

* **.env file:**

PORT=5000

**JWT\_SECRET** = ASKDGJSLDJGSLA;KDJIUOEWUTPIOUASKLDGJ;SLDKAJG

# Cloudinary API credentials

**CLOUDINARY\_CLOUD\_NAME**=praveensunar

**CLOUDINARY\_API\_KEY**=461322471974587

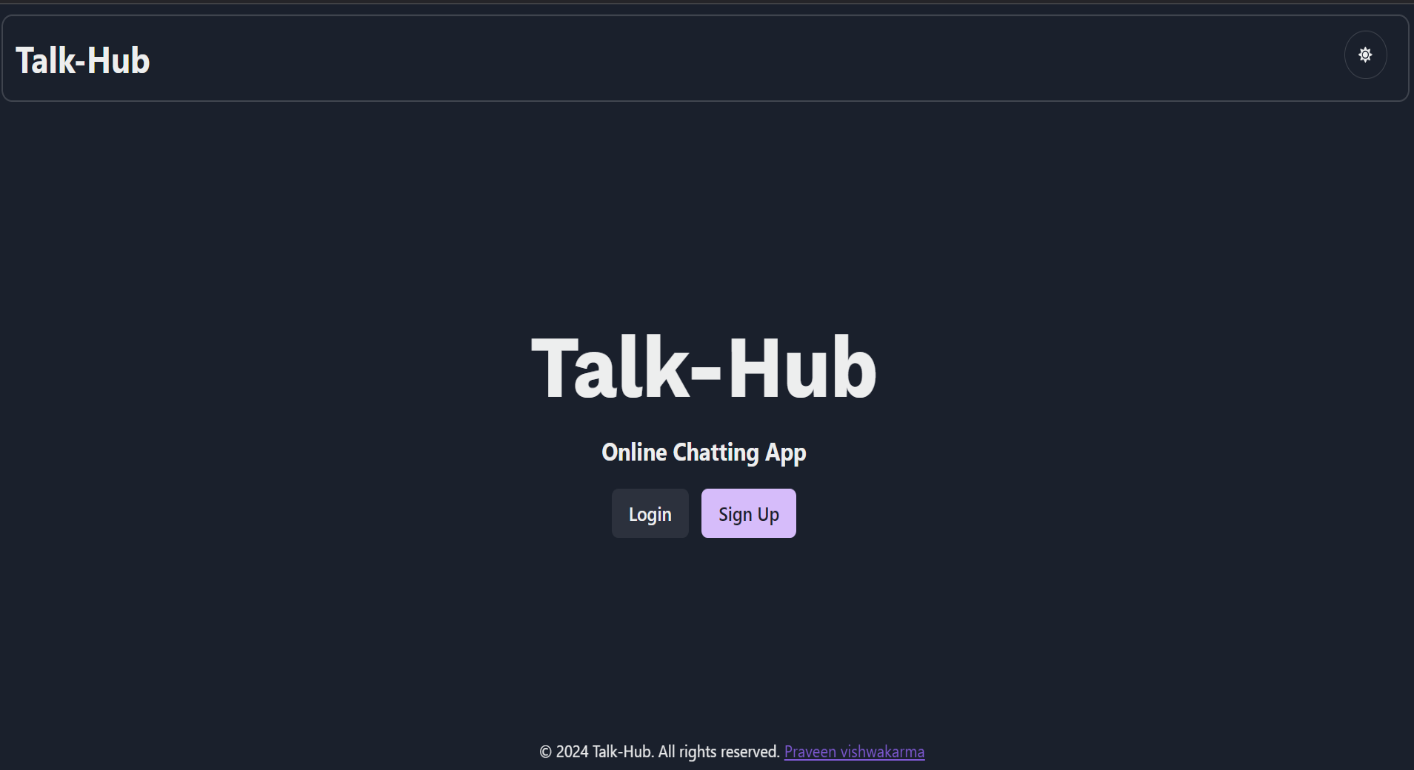
**CLOUDINARY\_API\_SECRET**=QUJINnd70HKndGbpFgPLG6R09Mg

# ai chatbot api

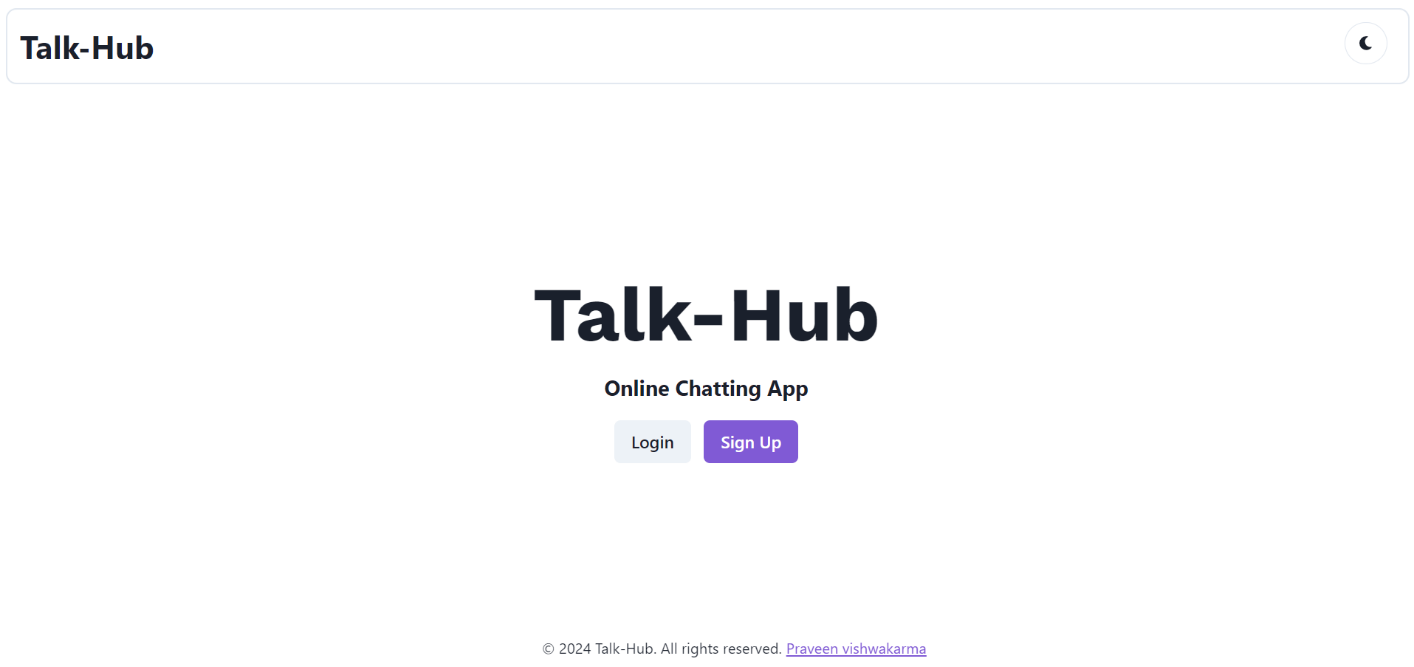
**GENERATIVE\_API\_KEY**=AIzaSyC9CieF5T\_89n7YlMndGUWd5uXMrpagyMg

**SCREENSHORTS**

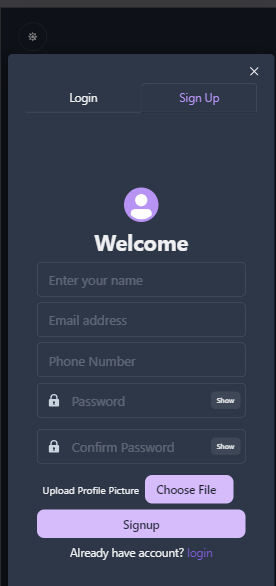
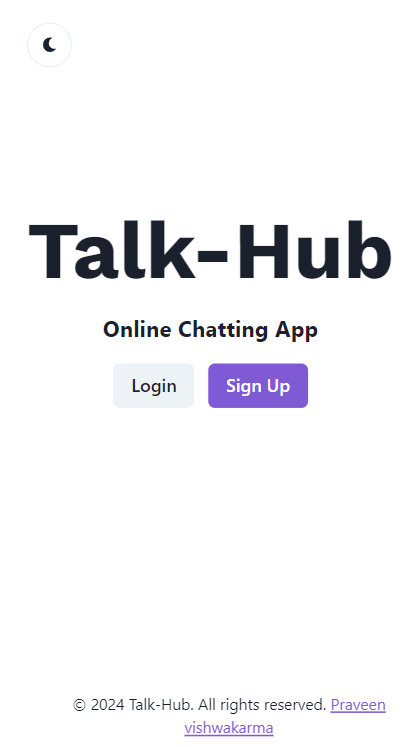
**LANDING PAGE :**

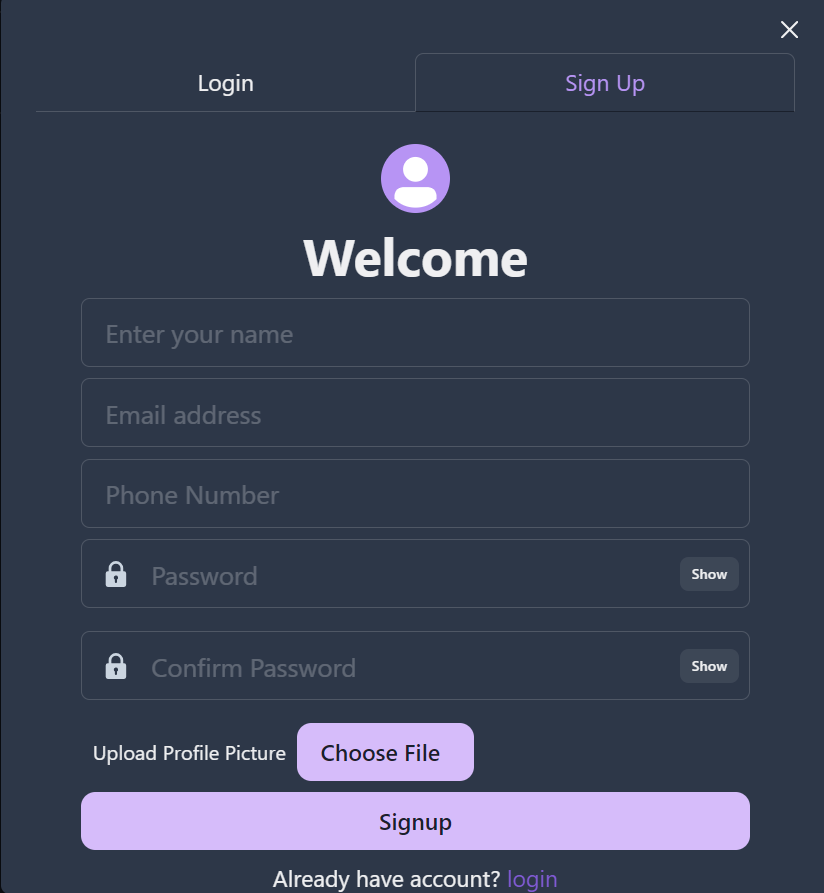
**Dark Mode**

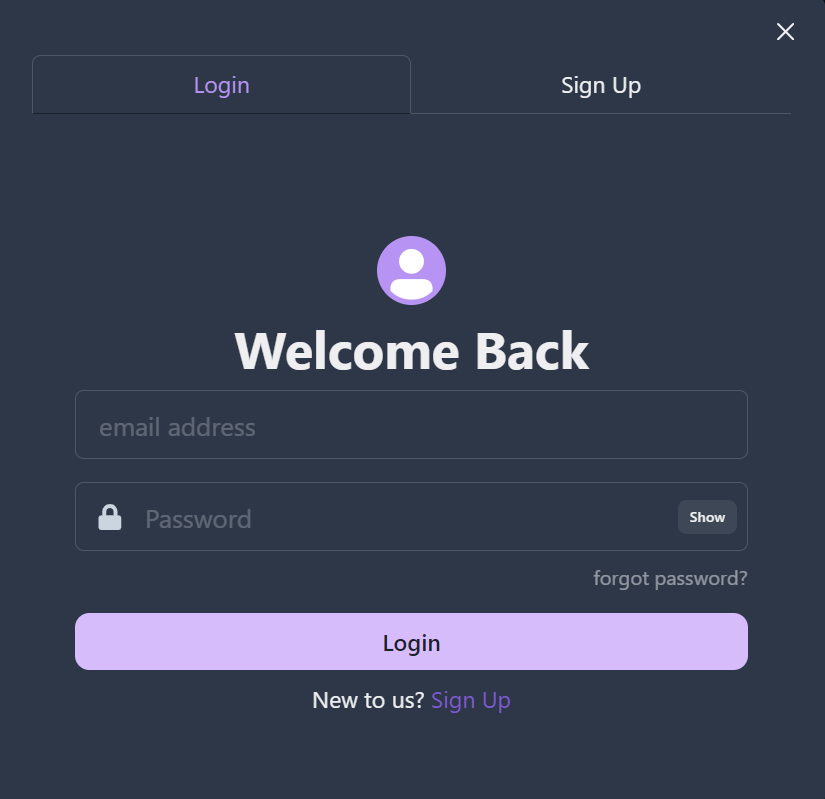
**Light Mode**

****

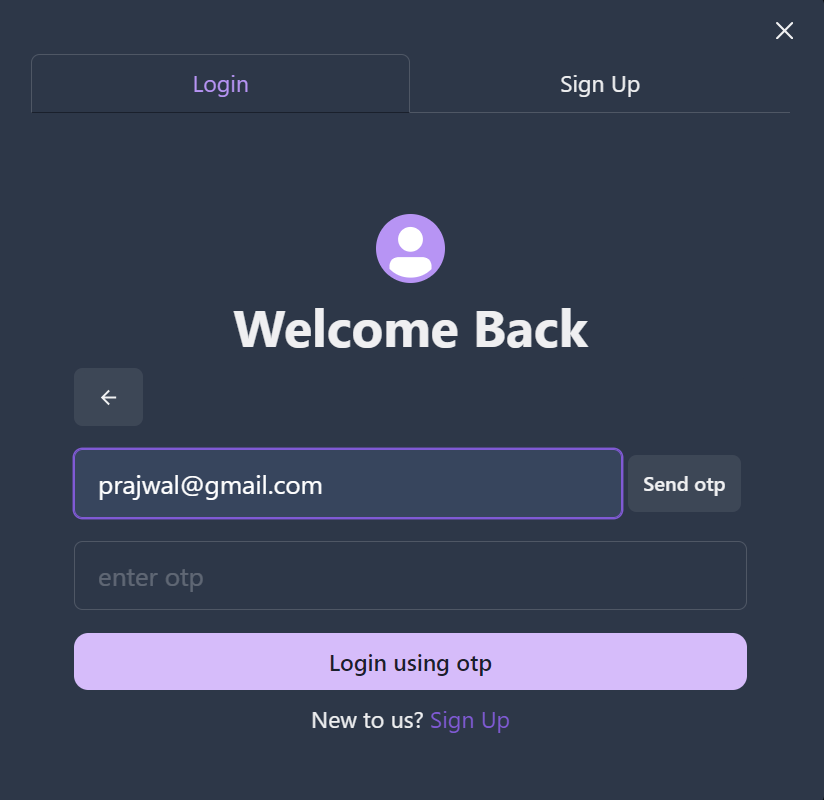
**The Project is responsive**



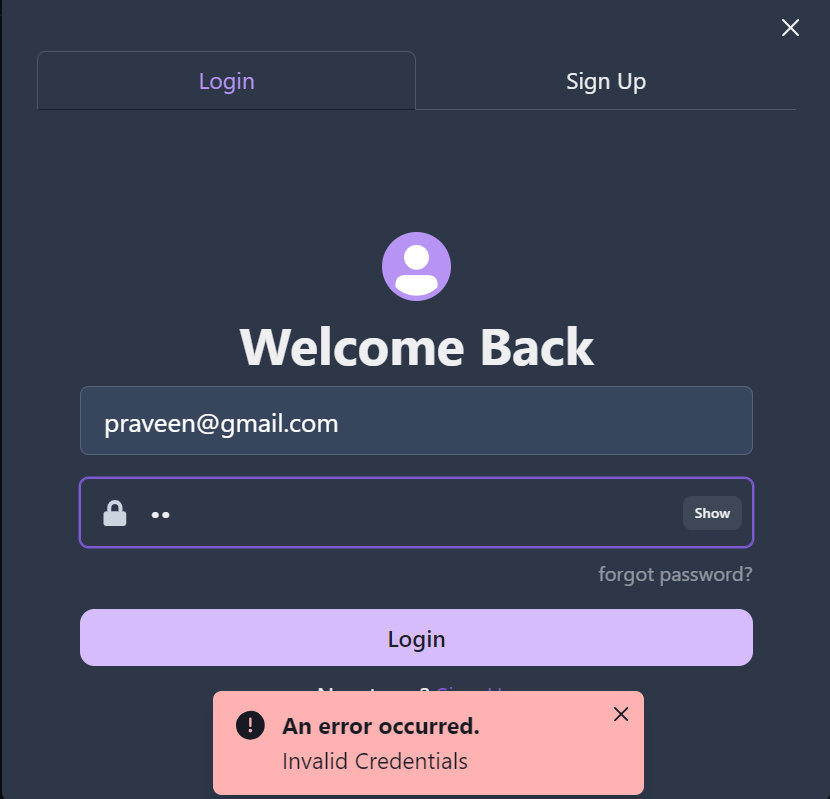
**SIGN-UP PAGE**

**LOGIN PAGE:**

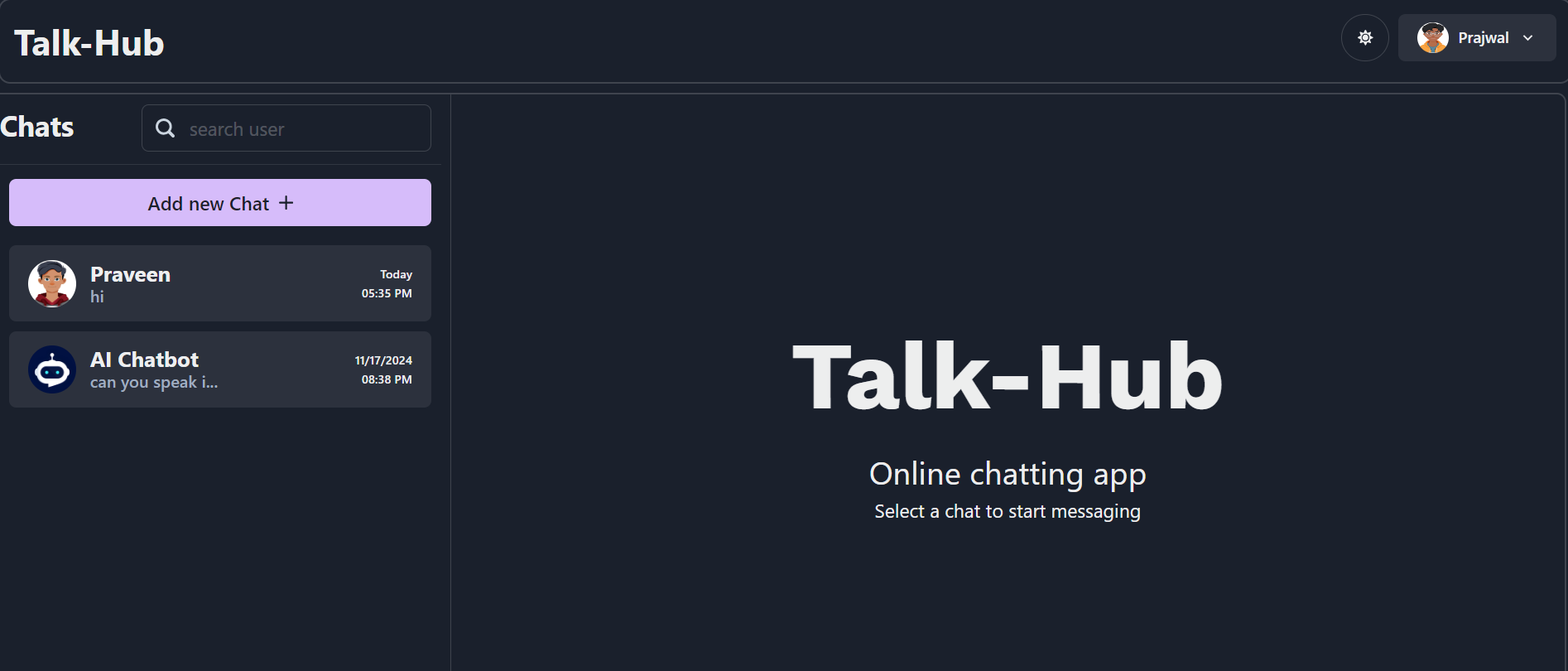
**LOGIN with OTP (Forgot Password):**

****

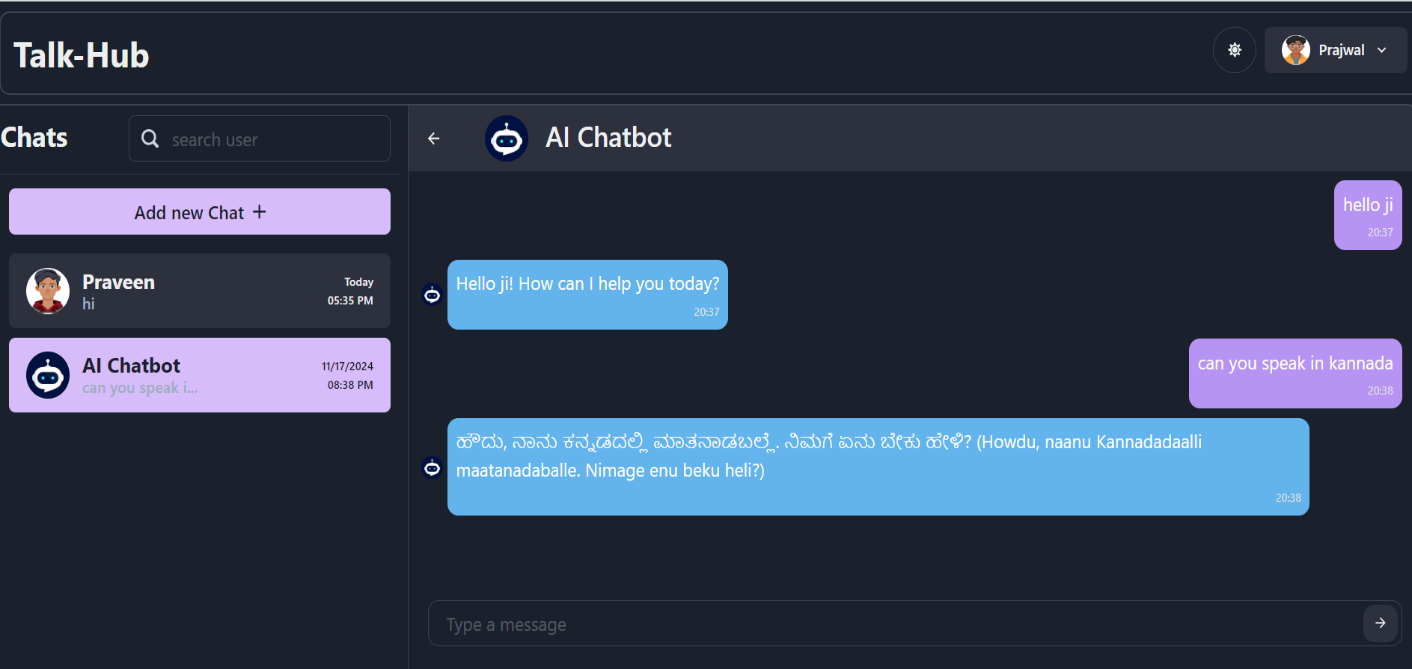
**VALIDATION LOGIN (Incorrect Password)**



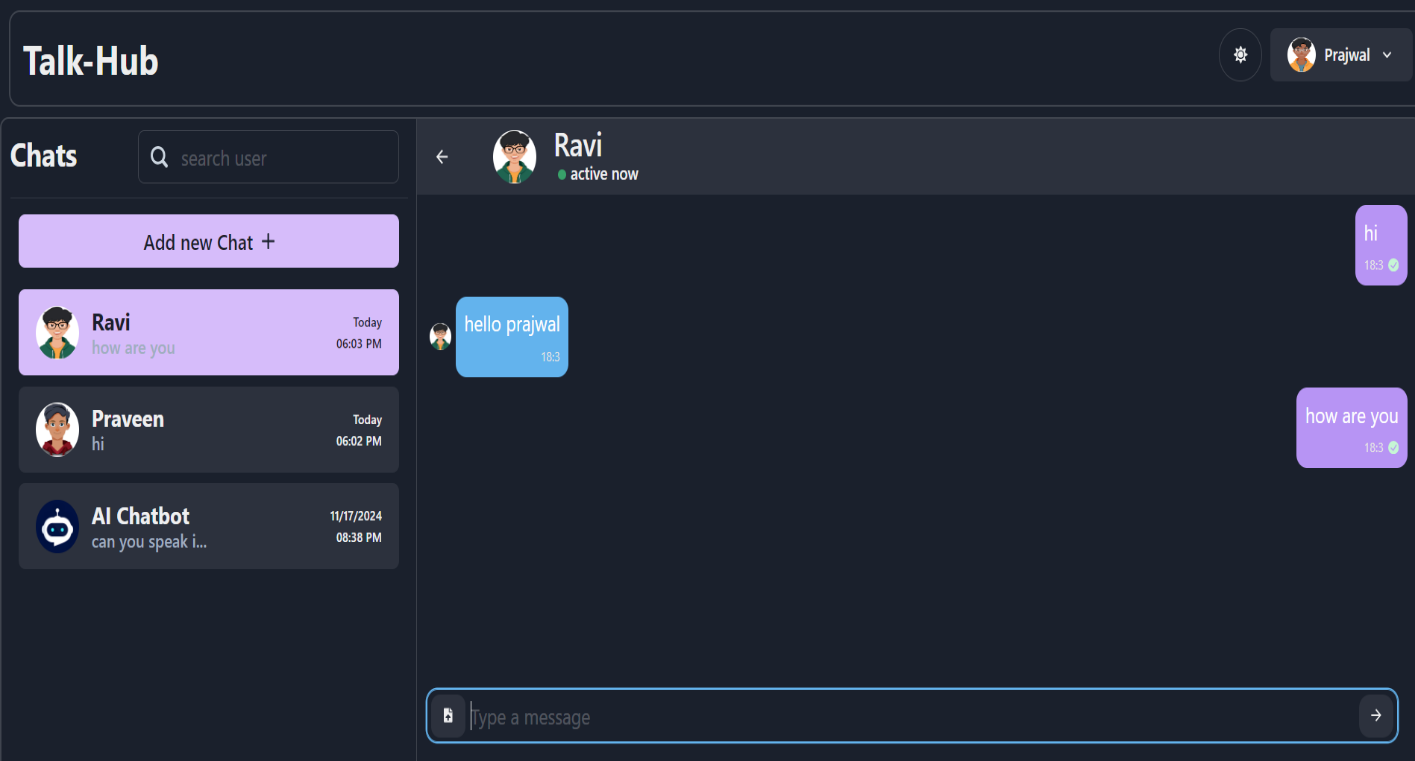
**USER DASHBOARD**

****

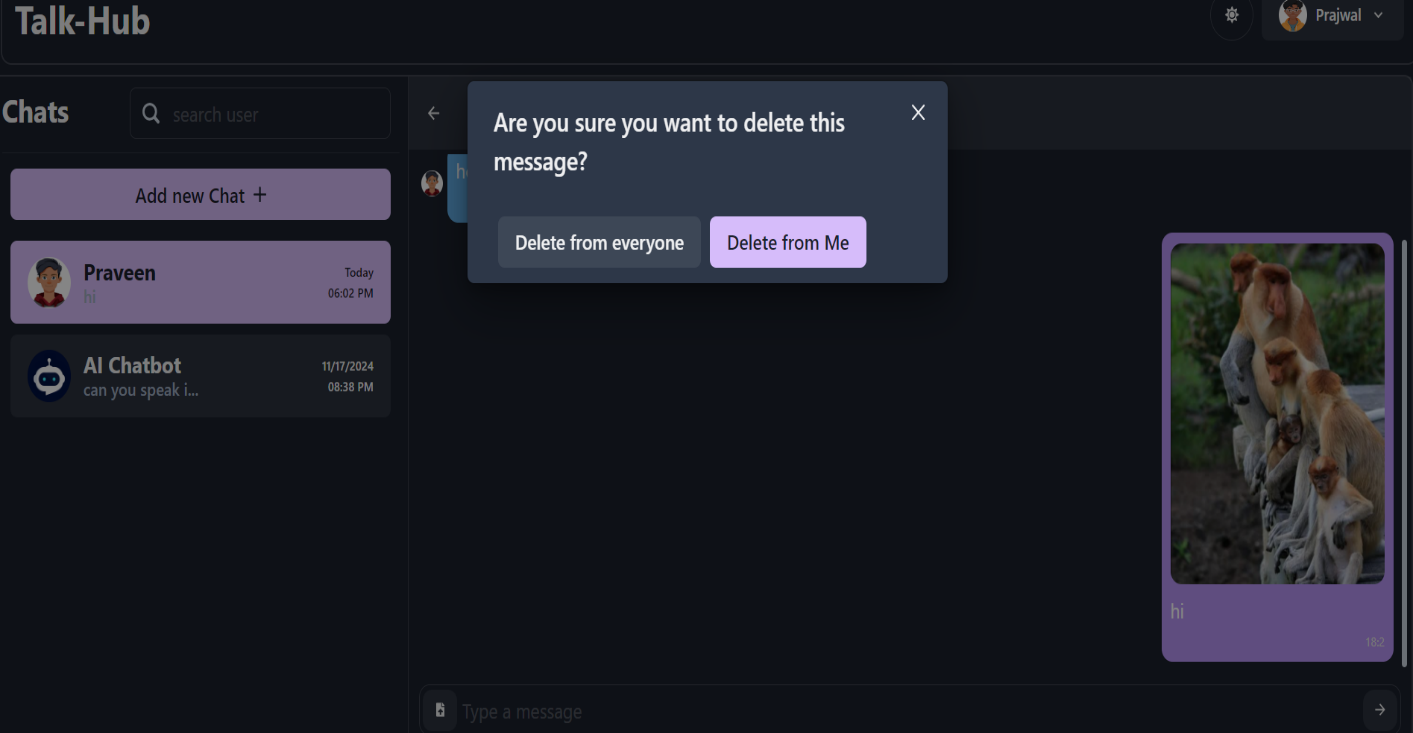
**USER DASHBOARD (Chatting with Chatbot)**



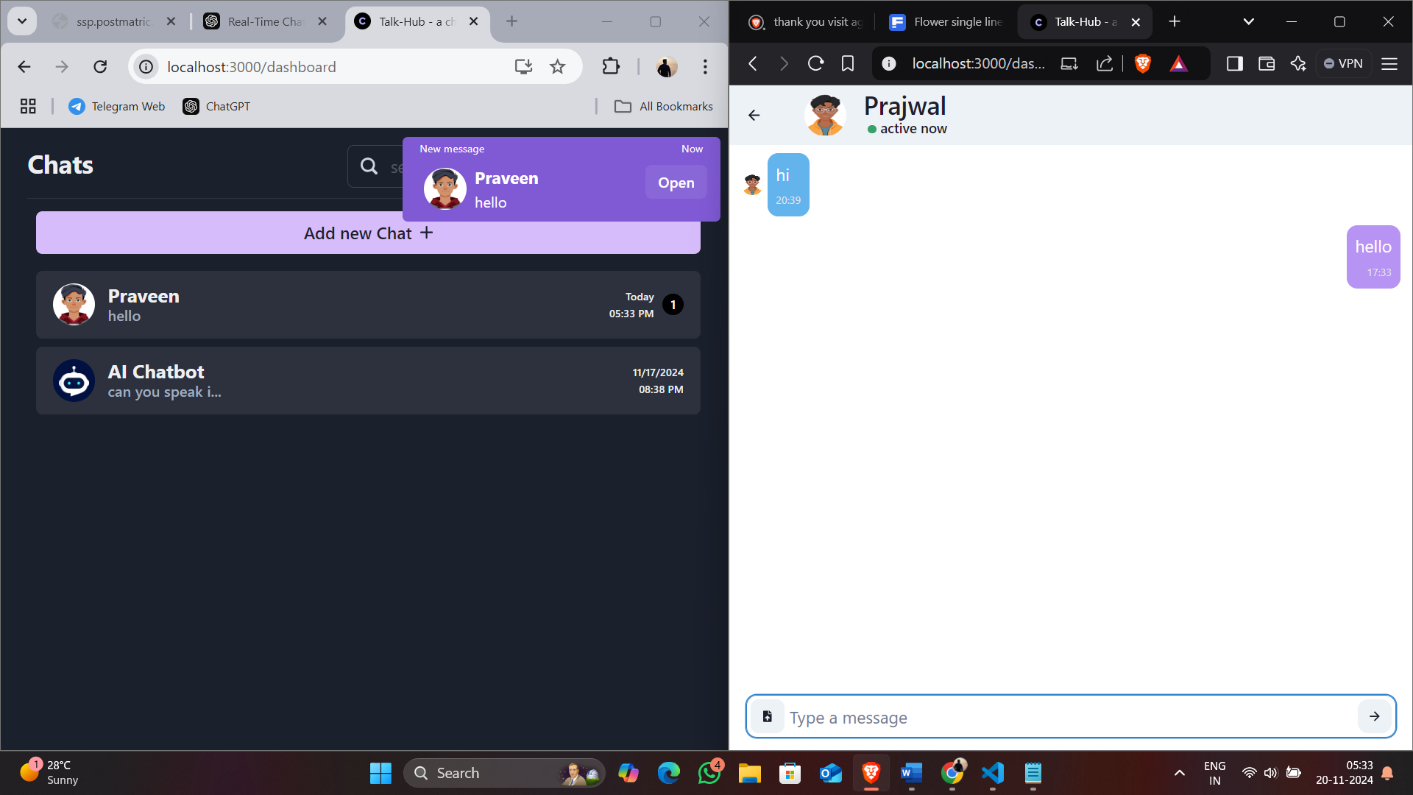
**USER DASHBOARD (Chatting with Friend)**



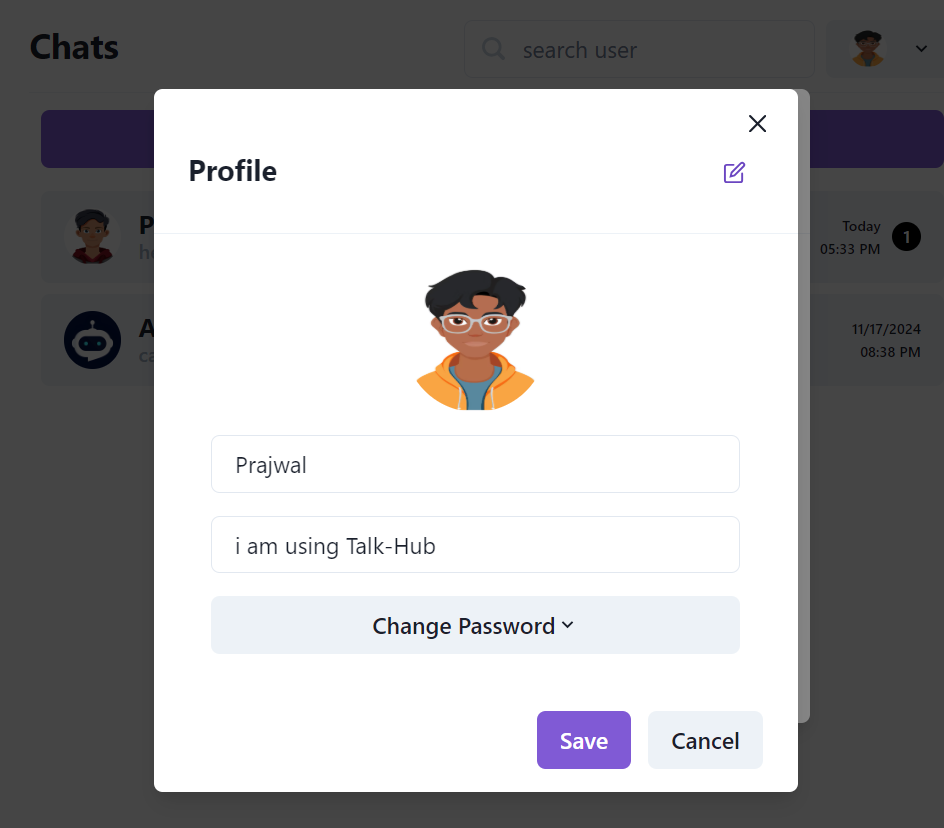
**USER DASHBOARD (Deleting the messages)**

****

#### New Message Notification

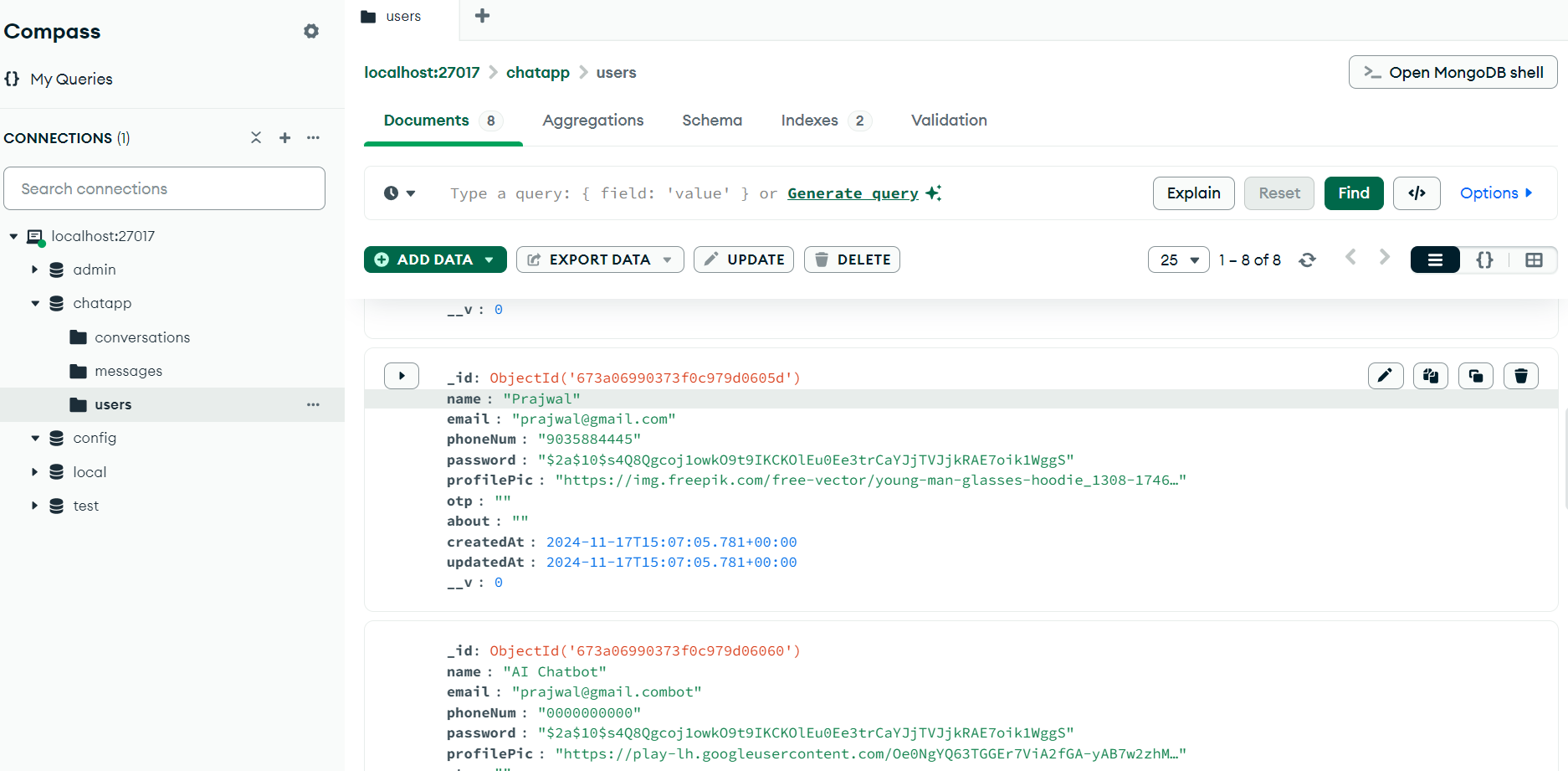


**USER DASHBOARD (Profile)**



#### MONGODB DASHBOARD

#### MONGODB DASHBOARD(User Detail)



**9.CONCLUSION**

# 9.CONCLUSION

**Talk-Hub** successfully delivers a modern, real-time communication platform that integrates advanced functionalities with a user-friendly interface Designed to social connections, the application combines instant messaging with the power of an AI-driven chatbot, bridging the gap between human communication and artificial intelligence. user-friendliness is at the core of Talk-Hub's design. The application features an intuitive interface that is easy to navigate, ensuring accessibility for users of all technical backgrounds. a scalable architecture, making it capable of handling a growing user base and increased traffic without compromising performance. Its modular design, powered by the MERN stack (MongoDB, Express.js, React.js, and Node.js), allows for easy maintenance and future enhancements. The application is adaptable and ready for new features, such as voice and video calling, media sharing, and advanced AI integrations.

**10.FUTURE SCOPE**

# 10.FUTURE SCOPE

The future scope of the project circles around maintaining information regarding.

* **Group Chat**: Adding group messaging features.
* **Video and Voice Calls**: Integrating VoIP functionality.
* **Message Reactions**: Allowing users to react to messages.
* **Voice-Based Chatbot Interaction**: Allow users to communicate with the chatbot through voice commands, making the platform more accessible and engaging.
* **Location Sharing**: Allowing users to share their real-time location.
* **Two-Factor Authentication (2FA)**: Add an extra layer of security by enabling 2FA during login.
* **Cloud-Based Scalability**: Use cloud platforms like AWS, Google Cloud, or Azure to support high user traffic, ensuring consistent performance during peak usage.
* **Cross-Platform Support**: Develop mobile applications using frameworks like React Native or Flutter to reach a broader audience on Android and iOS.

**11.BIBLIOGRAPHY**

# 11.BIBLIOGRAPHY

**WEBSITE REFERENCES**

* Google.com: <https://goggle.com/>
* Youtube.com: <https://youtube.com/>
* React: <https://react.dev/>
* Stack Overflow: <https://stackoverflow.com/>
* Chakra-UI: <https://www.chakra-ui.com/>
* w3schools: <https://www.w3schools.com/>
* ChatGPT: <https://www.chatgpt.com/>
* Express-Js: <https://expressjs.com/>

**BOOK REFERENCES**

* HTML5 - JavaScript: HTML - Website Developer - CSS Website - JavaScript - Web Languages by Sara Jackline.
* Pro Mern Stack: Full Stack Web App Development With Mongo, Express, React, and Nodeby [Vasan Subramanian](https://www.amazon.in/-/e/B0D98B1KTK?ie=UTF8&field-author=Vasan+Subramanian&text=Vasan+Subramanian&sort=relevancerank&search-alias=digital-text&ref_=ast_author_cp).
* Full-Stack JavaScript” by Azat Mardan
* Beginners guide for html and CSS web design and web development by Robert Pattinson.
* MongoDB Fundamentals by Amit Phaltankar