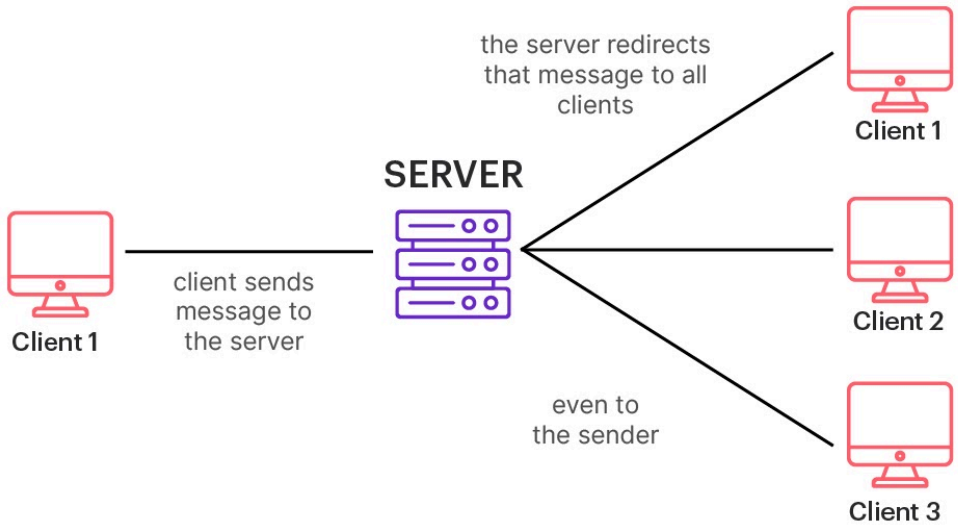
Name : Ankit Singh
College: IIT Gandhinagar
Roll Number: 22110025

Halo Chat: The Chat Application

Introduction

Our chat application is a modern, user-friendly platform designed to facilitate seamless communication and connection among users. With a focus on simplicity, security, our application empowers you to stay connected with friends, family, and colleagues effortlessly. Key features include : user authentication, individual messaging, group chat functionality and many more features are yet to be added.

System Architecture



chat

Front End: Powered by the **Next.js** framework, our application offers a sleek and responsive user interface that works seamlessly across devices.

Real-Time Communication: We leverage the power of **pusher** to provide instant, real-time messaging capabilities.

Robust Backend: Built with **Node.js**, our backend infrastructure ensures smooth data processing, user management, and connectivity.

Robust Data Storage: Your chat history and user data are securely stored and managed using **MongoDB**, a reliable and scalable NoSQL database solution. **Cloudinary** is a cloud-based platform that offers image and video management services for websites

Components and Modules

BackEnd:

The backend of your chat application is built using **Node.js**, **MongoDB**, **Pusher**, and **Cloudinary**, along with authentication powered by **NextAuth**. Each component plays a critical role in ensuring that your app handles user authentication, chat messages, and real-time communication efficiently. Below is a detailed breakdown of the components:

1. Node.js

- **Description**: Node.js is the runtime environment that powers your backend. It allows you to write server-side JavaScript code, and manage APIs, routes, and logic.
- **Why it's used**:
    - Non-blocking, event-driven architecture makes it ideal for real-time applications like chat systems.
    - Large ecosystem of libraries and modules that simplify development.

3. MongoDB (Database)

- **Description**: MongoDB is a NoSQL database used to store data in a flexible, JSON-like format (documents).
- **Why it's used**:
    - Flexibility of storing nested documents and arrays (perfect for storing chats, messages, and users).
    - Scalability for handling large amounts of chat data.
    - Efficient querying of real-time data such as chat messages and user activity.
- **Models Used**:
    - **User Model**: Stores user data (username, email, password, profileImage, etc.).
    - **Chat Model**: Stores chat data (group or individual, members, messages, etc.).

    - **Message Model**: Stores individual messages (text, images, sender, chat reference).

4. Mongoose (ORM)

- **Description**: Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It helps in managing relationships between data, enforcing schemas, and providing validation.
- **Why it's used**:
    - Provides schema-based solutions to model your data, ensuring consistency in user, chat, and message documents.
    - Simplifies database queries, making it easy to perform operations like creating users, saving messages, and retrieving chat histories.

5. Pusher (Real-Time Messaging)

- **Description**: Pusher is a hosted service that provides APIs for building real-time functionality like notifications and live chat.
- **Why it's used**:
    - Enables real-time updates of chat messages and user activity.
    - Automatically updates connected clients without manual refreshes, creating an efficient real-time chat experience.
    - Reduces the complexity of managing WebSockets directly.
- **Integration**:
    - When a new message is sent, an event is triggered, and Pusher broadcasts the update to all connected users in the chat, allowing real-time message delivery.

6. Cloudinary

- **Description**: Cloudinary is a cloud-based service that provides image and video management, including storage, optimization, and delivery.
- **Why it's used**:
    - Efficiently handles media uploads for user profile pictures or image messages.
    - Automatically optimizes images, ensuring that they are served quickly and with minimal bandwidth.
    - Provides easy-to-use APIs for uploading and retrieving images.
- **Integration**:
    - When a user uploads a profile picture or sends an image in a message, the image is uploaded to Cloudinary, and the URL is stored in the database.

8. bcrypt.js (Password Hashing)

- **Description**: bcrypt.js is a library used for hashing passwords before storing them in the database.

- **Why it's used**:
    - Ensures that user passwords are stored securely by hashing them.
    - Adds a layer of security, making it difficult for attackers to retrieve plain-text passwords even if the database is compromised.

FrontEnd:

- **Component folder:**
    - **Bottom Bar** Manages the input area at the bottom of the chat interface where users can type and send messages.
    - **Chat Box** Displays the main conversation area, showing the ongoing chat between users.
    - **Chat Details** Provides information about the current chat, including participants and group details (if applicable).
    - **Chat list** Displays a list of all available chats or conversations for easy navigation between them.
    - **Contacts** Manage the list of all users or contacts available for initiating a chat or conversation.
    - **Form** Handles the user input forms, such as login, registration, or profile update forms.
    - **Loader** Shows a loading animation or indicator while data or content is being fetched or processed.
    - **Message box** Displays individual messages in the chat, including sender information and message content.
    - **Provider** Manages state and context providers for sharing data and functionality across components.
    - **Topbar** Renders the top navigation bar or header for the chat interface, often showing options like profile.

User Model:

The `User` model defines the structure and properties of user data within the application. It represents how user-related information is stored and organized in the MongoDB database. Each user document contains details such as username, email, password, profile image, and associated chats. Below is a detailed breakdown of each field in the `UserSchema`:

1. `username` (String)

- **Description**: A required field that stores the user's username. This field ensures that every user has a display name in the system.
- **Type**: String
- **Required**: Yes
- **Usage**: This is the primary name shown for a user in the application, whether in chats, profiles, or user listings.

2. `email` (String)

- **Description**: A required and unique field that stores the user's email address. It serves as a unique identifier for each user, ensuring that no two users share the same email.
- **Type**: String
- **Required**: Yes
- **Unique**: Yes
- **Usage**: Used for user registration, login, and communication. The uniqueness constraint ensures each user has a distinct email address.

3. `password` (String)

- **Description**: A required field that stores the user's hashed password. This is used for authentication and to ensure the security of user accounts.
- **Type**: String

- **Required**: Yes
- **Usage**: The password is typically hashed before being stored to protect the user's sensitive information and to secure the login process.

4. `profileImage` (String)

- **Description**: A field that stores the URL of the user's profile picture. If the user doesn't provide an image, it defaults to an empty string.
- **Type**: String
- **Default**: An empty string (`""`)
- **Usage**: This allows users to upload and display a profile image. If not provided, the user will not have a profile picture, or the system may display a generic avatar.

5. `chats` (Array of ObjectIDs)

- **Description**: This field stores an array of references to chat documents that the user is part of. Each reference is an ObjectID, linked to the `Chat` model.
- **Type**: Array of ObjectIDs (References to `Chat` documents)
- **Default**: An empty array.
- **Usage**: Used to track the conversations the user is involved in, whether group or individual chats.

Chat Model:

This `Chat` model represents a collection in MongoDB for storing details about individual or group chat conversations. It defines the schema for a chat, including participants, messages, group information, and timestamps.

- The `members(Array of ObjectIDs)` field contains an array of user references, where each user is a participant in the chat. These ObjectIDs refer to users stored in the `User` model. Each chat can have multiple participants, and the list helps track all the members involved in the conversation.

- This field holds references to the `Message` documents exchanged within the chat. Each ObjectID in the array refers to a message stored in the `Message` model. This field helps in organizing the messages related to a specific chat.
- The `isGroup` field determines whether the chat is a group chat or an individual chat. It is a boolean value where `true` signifies a group chat and `false` signifies a one-on-one chat between two users.
- `name` field stores the name of the chat. For individual chats, this can be the name of the other participant, and for group chats, this field stores the name of the group. The name is useful for identifying group chats or one-on-one conversations.
- The `groupPhoto` field stores the URL of the group chat's display picture or photo. This is relevant only for group chats and helps visually identify the group.
- `createdAt` field stores the timestamp of when the chat was created. It helps track the creation date of the chat and is useful for sorting chats or showing the duration of chat existence.
- `lastMessageAt` (Date) is updated every time a new message is sent. It helps in organizing chats based on activity, allowing recent conversations to appear at the top in the chat list.

This schema provides flexibility for both one-on-one and group chat functionalities, making it versatile for a real-time messaging application.

Message Model:

The `Message` model defines the structure and properties of messages exchanged within chats in the application. Each message document represents an individual message sent by a user within a specific chat and includes details like the sender, content (text and photo), and when it was created. Below is a detailed breakdown of each field in the `MessageSchema`:

1. `chat` (ObjectID Reference)

- **Description**: A reference to the chat the message belongs to. This links each message to its respective chat conversation.
- **Type**: ObjectID (Reference to `Chat` model)
- **Required**: Yes
- **Usage**: Ensures each message is associated with the correct chat, whether it is a group chat or individual conversation.

2. `sender` (ObjectID Reference)

- **Description**: A reference to the user who sent the message. This links each message to its sender, allowing the system to track who sent what.
- **Type**: ObjectID (Reference to `User` model)
- **Required**: Yes
- **Usage**: Identifies the user responsible for sending the message, enabling tracking of user activity and ensuring accountability in conversations.

3. `text` (String)

- **Description**: A field to store the text content of the message. If the message does not include text (for example, in the case of photo messages), this field can be left as an empty string.
- **Type**: String
- **Default**: An empty string (`""`)
- **Usage**: Used to store the main body of the message, such as a chat message, announcement, or other written communication.

4. `photo` (String)

- **Description**: A field to store the URL of an image sent with the message. If no image is attached, this field defaults to an empty string.

- **Type**: String
- **Default**: An empty string (`""`)
- **Usage**: Allows users to send photos as part of their messages, supporting multimedia content in chats.

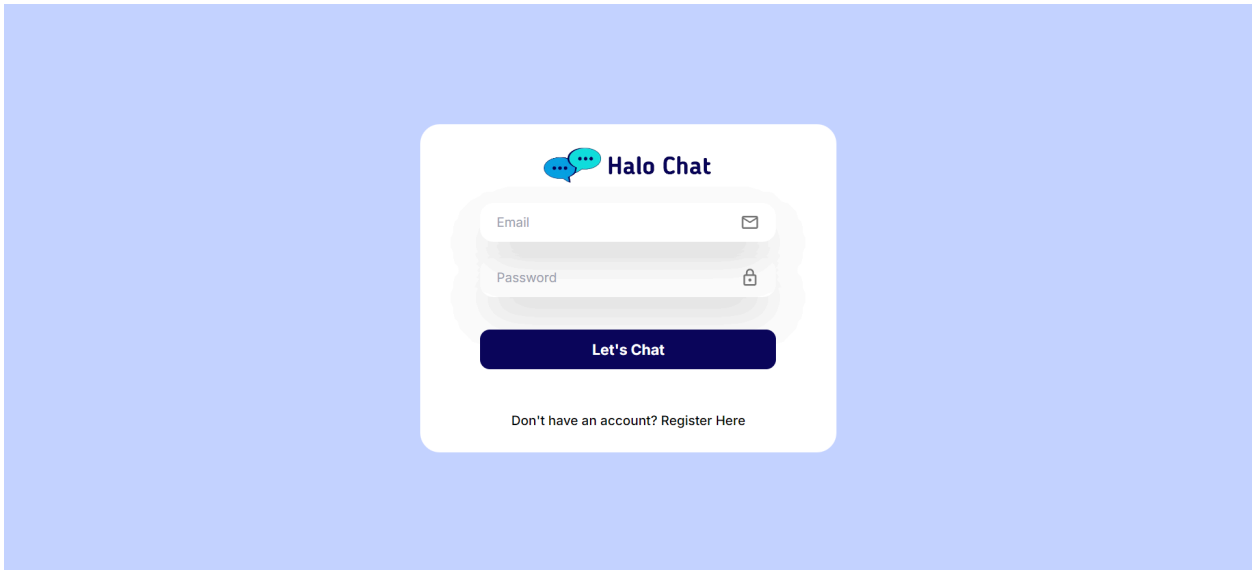
5. `createdAt` (Date)

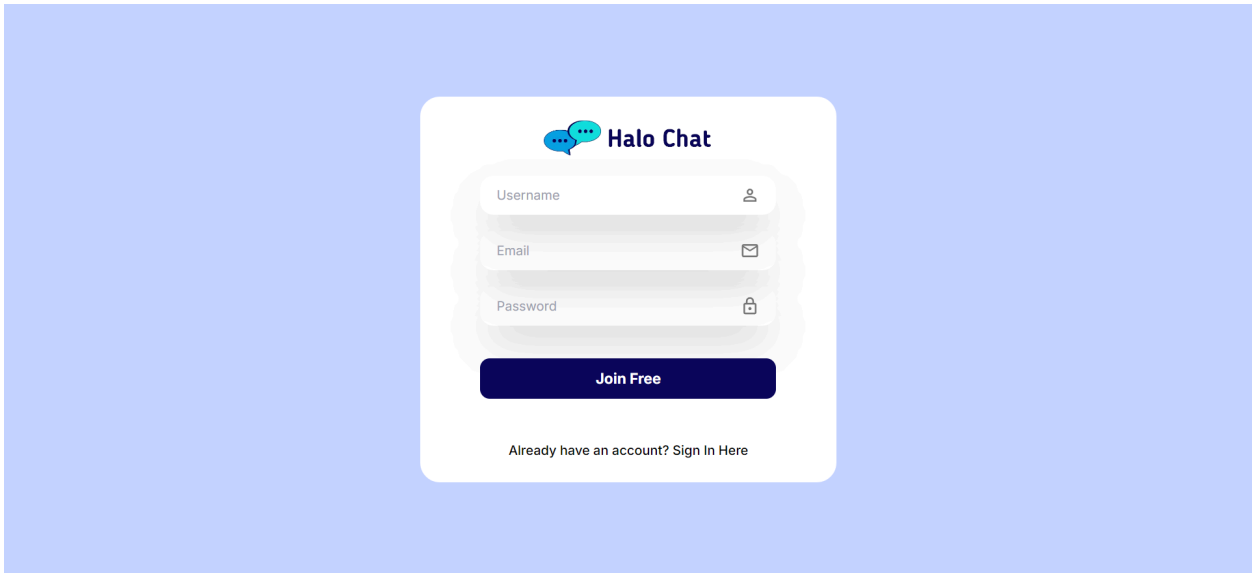
- **Description**: A timestamp that records when the message was created. It automatically stores the date and time when the message is sent.
- **Type**: Date
- **Default**: `Date.now`
- **Usage**: Useful for ordering messages chronologically within a chat and displaying the time the message was sent to the users.

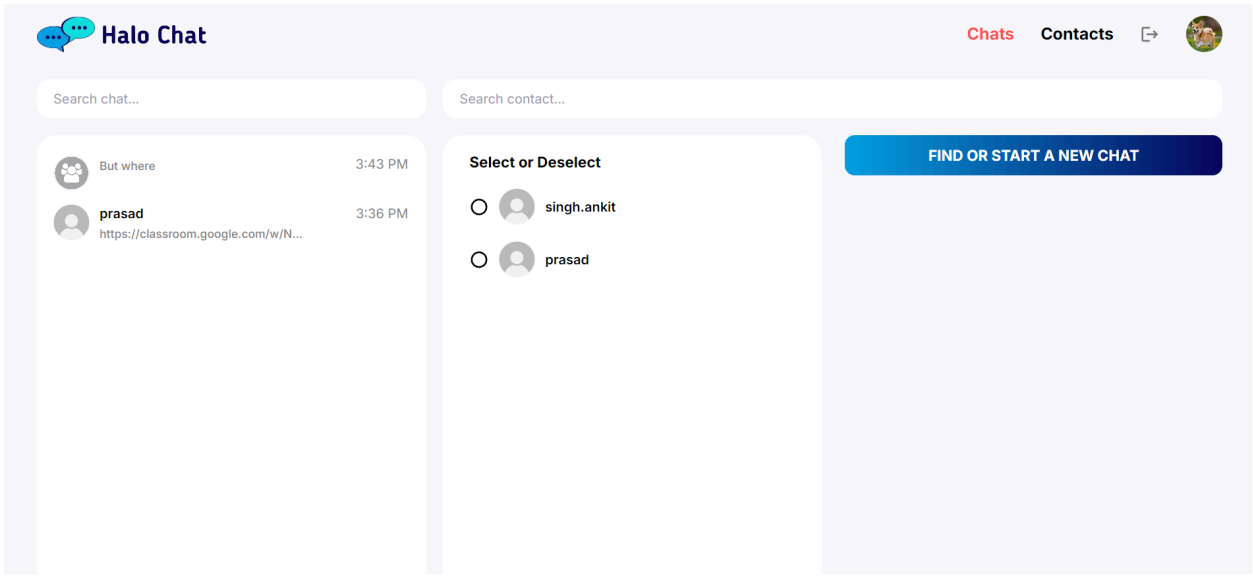
6. `seenBy` (Array of ObjectIDs)

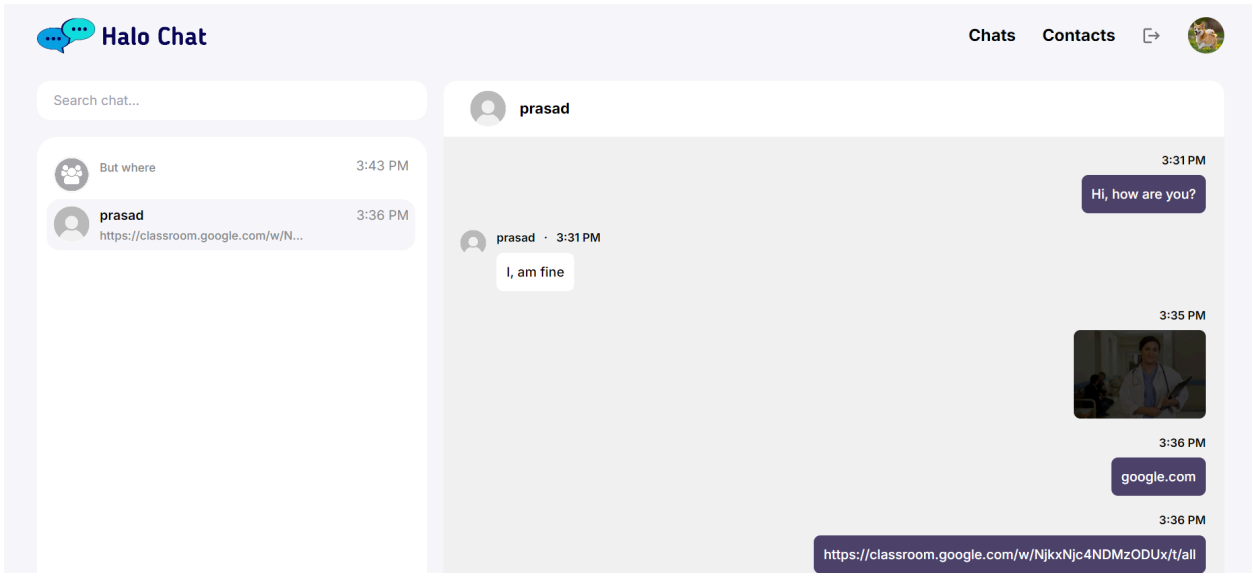
- **Description**: This field stores an array of references to users who have seen the message. Each reference is an ObjectID linked to the `User` model.
- **Type**: Array of ObjectIDs (References to `User` model)
- **Default**: An empty array
- **Usage**: Used to track which users have seen the message in a chat, supporting features like message read receipts.

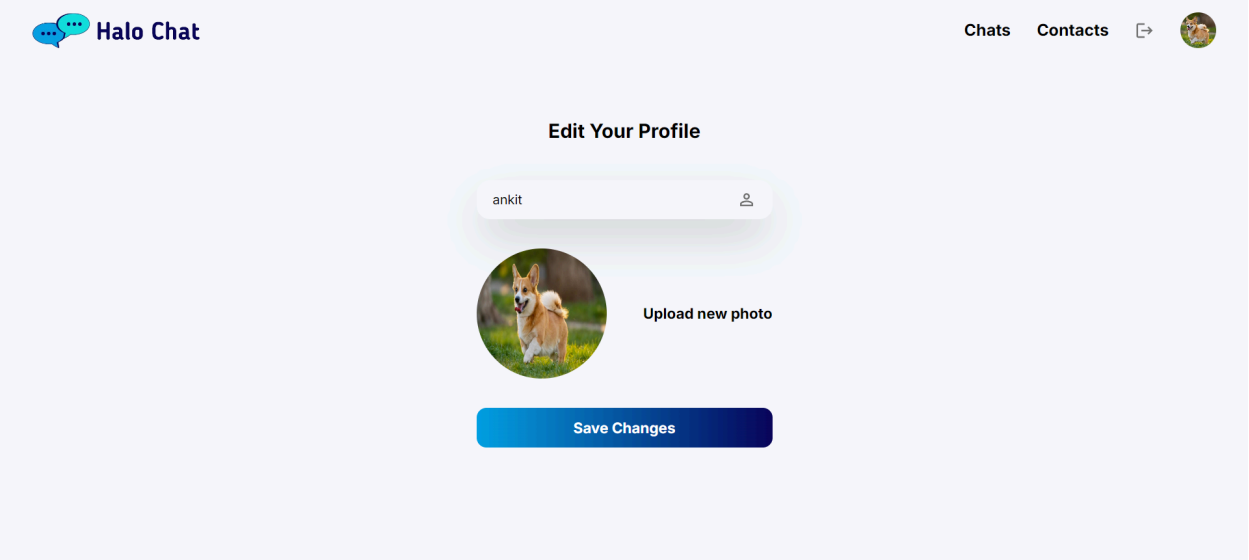
User Interface (UI) Framework:


Halo Chat
Email
Password
Let's Chat
Don't have an account? Register Here


Halo Chat
Username
Email
Password
Join Free
Already have an account? Sign In Here


Halo Chat
Chats
Contacts
Search chat...
Search contact...
But where
3:43 PM
prasad
3:36 PM
https://classroom.google.com/w/N...
Select or Deselect
singh.ankit
prasad
FIND OR START A NEW CHAT


Halo Chat
Chats
Contacts
Search chat...
But where
3:43 PM
prasad
3:36 PM
https://classroom.google.com/w/N...
prasad
3:31 PM
Hi, how are you?
prasad · 3:31 PM
I, am fine
3:35 PM
3:36 PM
google.com
3:36 PM
https://classroom.google.com/w/NjkxNjc4NDMzODUx/t/all



Next.js serves as the foundational framework for crafting the user interface (UI) of our web application. It plays a pivotal role in defining how our application looks and behaves across multiple platforms, offering a robust and expressive toolkit for UI design and development. With features like server-side rendering and static site generation, Next.js enhances performance and improves SEO, making it an ideal choice for building dynamic and interactive web applications.

API Design

User API (/api/user):

This API likely handles user-related operations that are **user registration, user authorization for login and retrieval of all other users**.

- /api/auth/register for registration of an user using a post request.
- api/user/login for login into the app using a post request.
- `/api/users/[userId]`Retrieves a user by their ID.
- `/api/users/[userId]/searchChat/[query]` Searches for a chat by a user with a query.
- `/api/users/[userId]/update` Updates the user's information.

Chat API (/api/chat):

The chat API could be responsible for **creating group chat, access chat, remove user from group, add user to group.**

- /api/chat/ for accessing the chat using a post request.
- /api/chats/[chatId] Retrieves a chat by its ID.
- `/api/chats/[chatId]/update` Updates the details of an existing chat.

Messaging API (/api/message):

The message API likely focuses on managing messages within chat rooms and conversations like **getting all the chats of a particular user, and sending the message.**

- `/api/messages` Retrieves all messages.

How to Set Up and Run the Project

1. Clone the Repository

```
git clone https://github.com/AnkitS-21/Chat-Application.git
cd Chat-Application
```

2. Set Up Environment Variables

Create a `.env.local` file in the root directory with the following environment variables:

```
# MongoDB connection string

MONGODB_URI=mongodb+srv://<username>:<password>@cluster.mongodb.net/
mydatabase?retryWrites=true&w=majority

# NextAuth

NEXTAUTH_URL=http://localhost:3000

# Pusher credentials

PUSHER_APP_ID=your_pusher_app_id
```

```
PUSHER_KEY=your_pusher_key

PUSHER_SECRET=your_pusher_secret

PUSHER_CLUSTER=your_pusher_cluster

# Cloudinary

CLOUDINARY_URL=cloudinary://<api_key>:<api_secret>@<cloud_name>
```

3. Install Dependencies

```
npm install
```

4. Start the Development Server

```
npm run dev
```

The app will be available at http://localhost:3000.

Git gub repo link: https://github.com/AnkitS-21/Chat-Application