**Cphere - Community Sphere**

**A Real Time Video Chatting & Texting Web App**

*A project report submitted in partial fulfillment of the requirements for the degree of Bachelor of Science (Hons.) in Computer Science*

by

Sambhav Kaushik (Roll Number: 22220CMP023, Enrollment Number: 452707)

# 

Department of Computer Science,  
Institute of Science,  
Banaras Hindu University, Varanasi, 221005

May 2025

## CANDIDATE’S DECLARATION

I, **Sambhav Kaushik**, certify that this report, “**Cphere**: A Real-Time One-to-One Chat Web Application with In-App Notifications and WebRTC Video Calling,” is my own work conducted from **March 2025 to May 2025** under the guidance of **Dr. Anshul Verma**.

The matter presented in this project report has not been and will not be submitted elsewhere for the award of any other degree or diploma from any Institutions.

**Date:** **Signature of the Candidate**

## ABSTRACT

**Cphere** is a self-hosted, real-time chat platform enabling secure one-to-one messaging, in-app notifications, and peer-to-peer video calls. The backend uses Rust with Actix Web for HTTP/WebSocket, MongoDB for data storage, and Actix-session for cookie-based session management. The frontend is built with React, TypeScript, Vite, and Tailwind CSS.

Key features include session-based authentication with Argon2-hashed passwords; persistent chats/messages in MongoDB; real-time message delivery via WebSockets; notification delivery via WebSocket; WebRTC signaling for video calls. Future enhancements include group chat and end-to-end encryption.

## 

## TABLE OF CONTENTS

**Candidate’s Declaration** ..................................................................................................... 2  
 **Abstract** ................................................................................................................................ 3

**Chapter 1 Introduction** ....................................................................................................... 5  
 **1.1** Background ................................................................................................................... 5  
 **1.2** Objectives and Scope …................................................................................................ 5  
 **1.3** Organization of Report ….............................................................................................. 5

**Chapter 2 Literature Review** .............................................................................................. 6  
 **2.1** Real-Time Communication Protocols ............................................................................ 6  
 **2.2** WebSocket and WebRTC ............................................................................................. 6  
 **2.3** Comparison with Existing Solutions .............................................................................. 6

**Chapter 3 System Design and Architecture** .................................................................... 7  
 **3.1** Architectural Overview .................................................................................................. 7  
 **3.2** Technology Stack .......................................................................................................... 7  
 **3.3** Database Schema ......................................................................................................... 7  
 **3.4** API Endpints ………....................................................................................................... 8  
 **3.5** WebSocket Event Types …......................................................................................... 10  
 **3.6** Backend structure ........................................................................................................11  
 **3.7** Frontend Structure ...................................................................................................... 12

**Chapter 4 Implementation** ............................................................................................... 14  
 **4.1** Backend Modules ........................................................................................................ 14  
 **4.2** Frontend Modules ....................................................................................................... 15  
 **4.3** Security Measures ...................................................................................................... 16

**Chapter 5 Testing and Evaluation** .................................................................................. 18  
 **5.1** Testing Strategy .......................................................................................................... 18  
 **5.3** Usability Feedback ...................................................................................................... 18

**Chapter 6 Conclusion and Future Work** ........................................................................ 19

**References** .............................................................................................................................................. 20  
 **Appendix A UI Screenshots** ............................................................................................ 21

## CHAPTER 1: INTRODUCTION

### 1.1 Background

Modern chat applications demand low-latency, scalable real-time communication. Protocols like HTTP polling are inefficient, so WebSocket and WebRTC provide persistent, bi-directional channels and peer-to-peer media streaming. Cphere leverages Rust and React to deliver a secure, high-performance chat platform with video calling and notifications.

### 

### 1.2 Objectives and Scope

This project aims to:

* Implement secure session-based authentication (Rust + Argon2 + MongoDB)
* Provide one-to-one chat with persistent storage and real-time updates (Actix Web + WebSocket)
* Design in-app notifications stored in MongoDB and delivered via WebSocket
* Enable peer-to-peer video calls using WebRTC signaling over WebSocket
* Build a responsive frontend (React + TypeScript + Tailwind CSS).

Scope excludes group chat, end-to-end encryption, and file transfer (As of now). The architecture is extensible for future features.

### 

### 1.3 Organization of Report

**Chapter 1** introduces context and objectives. **Chapter 2** reviews protocols and related solutions. **Chapter 3** details architecture, database schema, APIs, and frontend structure. **Chapter 4** outlines implementation for backend and frontend, including security. **Chapter 5** presents testing approach, performance, and usability results. **Chapter 6** concludes and suggests future enhancements.

## 

## CHAPTER 2: LITERATURE REVIEW

### 2.1 Real-Time Communication Protocols

The two real-time communication protocols used in this project are:

1. WebSocket (RFC 6455) establishes a persistent TCP connection, allowing server-to-client push communication without polling.
2. WebRTC enables peer-to-peer audio/video/data, requiring a signaling channel—often WebSocket—for exchanging SDP and ICE candidates.

### 

### 2.2 WebSocket and WebRTC

1. Actix Web supports WebSocket natively, enabling real-time communication.
2. WebRTC offers direct media streaming; signaling can be implemented over existing WebSocket channels to avoid usage of additional servers.

### 

### 2.3 Comparison with Existing Solutions

Matrix, Rocket.Chat, and Mattermost offer self-hosted messaging, but often include large feature sets and dependencies.

Cphere focuses on minimal dependencies, high performance (Rust backend), and integrated video calling via WebRTC. Therefore, I decided to work with Actix webserver and websockets.

## 

## CHAPTER 3: SYSTEM DESIGN AND ARCHITECTURE

### 3.1 Architectural Overview

* Cphere uses a client-server model.
* The Rust backend exposes REST endpoints and a WebSocket endpoint for messaging and signaling.
* MongoDB stores users, sessions, chats, messages, and notifications.
* The React frontend communicates over REST and WebSocket, enabling chat, notifications, and video calls.
* Docker is used for containerisation purposes.

### 3.2 Technology Stack

* Rust + Actix Web: Provides high concurrency, memory safety, zero-cost abstractions.
* MongoDB: Allows flexible document schema, used as default storage by Actix sessions for indexing of authentication sessions.
* Actix-session: It is a cookie-based session management library.
* React + TypeScript + Vite: For facilitating fast builds, type safety, component-based UI.
* Tailwind CSS: Provides utility-first styling with minimal CSS footprint.

### 3.3 Database Schema

* Users Collection
  + **Fields**:
    - \_id: Unique identifier for each user (ObjectId).
    - username: User’s chosen display name (unique).
    - email: User’s email address (unique).
    - password\_hash: Securely hashed password using Argon2.
    - created\_at: Timestamp when the user was registered.
    - reset\_token: Optional token used for password reset workflows.
    - reset\_token\_expiry\_at: Expiry time for the reset token.
  + **Indexes**:
    - **Unique index** on email to prevent duplicate accounts.
    - **Unique index** on username to ensure identity uniqueness in chat contexts.
* Chats Collection
  + **Fields**:
    - \_id: Unique chat identifier.
    - participants: An array of two user ObjectIds representing the members of the chat.
    - created\_at: Timestamp when the chat was created.
  + **Indexes**:
    - Default index on \_id.
    - Potential compound index on participants can be added to optimize lookups involving both users.
* Messages Collection
  + **Fields**:
    - \_id: Unique identifier for each message.
    - chat\_id: Reference to the chat the message belongs to.
    - sender\_id: User who sent the message.
    - content: The message text or media reference.
    - created\_at: Timestamp of when the message was sent.
  + **Indexes**:
    - **Compound index** on (chat\_id, created\_at) to support efficient querying and ordering of messages within a specific chat, especially for pagination and real-time updates.
* Notifications Collection
  + **Fields**:
    - \_id: Unique identifier.
    - notification\_type: Type of notification (e.g., message, call, alert).
    - recipient\_id: User who will receive the notification.
    - sender\_id: Optional field for the user who triggered the notification.
    - message: Notification text or metadata.
    - is\_handled: Boolean flag indicating whether the notification has been read or acknowledged.
    - created\_at: Timestamp of creation.
  + **Indexes**:
    - **Compound index** on (recipient\_id, created\_at) to efficiently retrieve the latest notifications per user in chronological order.

### 

### 3.4 API Endpoints

* Authentication Module:  
  The authentication system ensures secure user identity management, using input validation, password hashing (via Argon2), and session management.
  + **Endpoints:**
    - POST /auth/register  
       Validates user input (email, username, password), hashes the password with Argon2, and creates a new user account in the database.
    - POST /auth/login  
       Verifies user credentials, generates a JWT or session cookie, and establishes a secure session.
    - POST /auth/logout  
       Terminates the current session by deleting the session record and clearing cookies.
    - POST /auth/reset\_password  
       Triggers a password reset process by generating a secure reset token and emailing it to the user. Stores reset\_token and reset\_token\_expiry\_at in the user document.
    - POST /auth/change\_password  
       Allows authenticated users to update their password after verifying the old password and validating the new one.
    - GET /auth/status  
       Returns the current session’s authentication status and user metadata, used for persistent login across page reloads.
* User Module:  
  Handles presence detection, notification retrieval, and search functionality.
  + **Endpoints:**
    - GET /:user\_id/is\_online  
       Checks whether a specific user is currently connected to the system (via session data or WebSocket registry).
    - GET /chats  
       Retrieves a list of chat sessions associated with the current user, including basic metadata for rendering the chat sidebar.
    - GET /get\_notifications  
       Fetches recent in-app notifications (e.g., new messages, calls) for the logged-in user.
    - GET /search\_users?query=  
       Performs a partial username search to find other users to start conversations with.
    - POST /is\_batch\_online  
       Takes a list of user IDs and returns their current online status, useful for updating group presence in real time.
    - POST /details  
       Retrieves detailed profile information of a user, typically for chat profile headers or admin views.
* Chat Module:  
  Facilitates chat creation, deletion, and messaging. Ensures no duplicate 1-to-1 chats are created and maintains message history.
  + **Endpoints:**
    - POST /create  
       Initiates a new chat between users. Checks if a chat already exists for the same participants and returns the existing one if found.
    - POST /delete  
       Deletes a chat for a user (soft delete or mark-as-archived). May remove or hide from the UI depending on implementation.
    - POST /send\_message  
       Persists a new message to the database, broadcasts it via WebSocket to chat participants, and triggers notifications.
    - GET /:chat\_id/get\_chat\_summary  
       Returns metadata about a specific chat, such as participants, last message, and unread count.
    - GET /:chat\_id/messages  
       Retrieves paginated message history for a chat, ordered by creation time.
* WebSocket Module:  
  Enables real-time communication and presence tracking.
  + **Endpoint:**
    - POST /connect  
       Initiates and authenticates a WebSocket session using a token. Establishes the user’s presence and listens for events such as message, typing, webrtc\_offer, and webrtc\_answer.
* Video Call Module:  
  Manages video call initiation, acceptance, and signaling for peer-to-peer communication.
  + **Endpoints:**
    - POST /video\_call/initiate  
       Used by the caller to start a video call. Notifies the recipient and sets up a signaling context.
    - POST /video\_call/respond  
       Called by the recipient to either accept or decline the call. If accepted, triggers the exchange of WebRTC SDP data and ICE candidates over WebSocket.

### 3.5 WebSocket Message Types

* **logout**
  + Triggered when a user logs out.
  + The server clears the user’s WebSocket session and marks them offline.
  + Other users are notified that the user is no longer online.
* **delete\_chat**
  + Sent when a user deletes a chat
  + The server removes the chat
  + It notifies other participant to update their UI
* **chat\_message**
  + Sent when a user sends a message in a chat.
  + Includes the chat ID, message content, and sender ID.
  + The server saves the message and sends it to all participants in the chat.
* **webrtc\_offer**
  + Sent by the caller to start a video call.
  + Contains the SDP offer used to initiate the WebRTC connection.
  + The server forwards it to the callee.
* **webrtc\_answer**
  + Sent by the callee in response to the webrtc\_offer.
  + Contains the SDP answer needed to complete the connection.
  + The server forwards it back to the caller.
* **webrtc\_ice\_candidate**
  + Sent by either peer during the WebRTC connection setup.
  + Contains ICE candidate information for network routing.
  + The server passes it to the other peer.
* **video\_call\_ended**
  + Sent when a user ends a video call.
  + The server informs the other participant that the call has ended.
  + Both clients clean up the call state and UI.
* **video\_call\_accepted**
  + Sent when a user accepts an incoming video call.
  + The server notifies the caller to proceed with connection setup.
* **video\_call\_declined**
  + Sent when a user declines an incoming call.
  + The server notifies the caller that the request was rejected.
* **video\_call\_request**
  + Sent by the caller to initiate a video call.
  + The server forwards this request to the callee.
  + Used to notify the recipient before the full WebRTC signaling starts.
* **user\_online**
  + Sent by the server when a user connects and is authenticated.
  + Notifies other relevant users (e.g., chat participants) that the user is now online.
* **user\_offline**
  + Sent when a user disconnects or logs out.
  + Notifies others that the user is now offline.

### 

### 3.6 Backend Structure

Folder organisation:  
backend

├── src

│ ├── config

│ ├── handlers

│ ├── middleware

│ ├── models

│ ├── services

│ ├── states

│ ├── types

│ ├── utils

│ └── websocket

└── tests

├── acceptance

├── integration

└── unit

### 

### 3.7 Frontend Structure

Folder organization:  
frontend/

├── public/

└── src/

├── components/

├── constants/

├── contexts/

├── layouts/

├── pages/

├── services/

├── types/

├── wrappers/

├── App.css

├── App.tsx

├── index.css

└── main.tsx

**Contexts**

* **AuthContext**: Stores login state and user info. Used to check if the user is authenticated.
* **ChatContext**: Stores chat list, selected chat, and messages.
* **NotificationContext**: Keeps track of notifications like new messages or calls.
* **NavigationContext**: Manages current page or tab in the UI.

**Pages**

* **LoginPage**: Page for users to log in.
* **RegisterPage**: Page to create a new account.
* **ForgetPasswordPage**: Lets users request a password reset link.
* **ResetPasswordPage**: Lets users set a new password using a reset token.
* **ChatsPage**: Shows a list of all chats.
* **ChatBoardPage**: Shows messages in the selected chat.
* **SearchPage**: Page to search for other users.
* **VideoCallPage**: Page for handling video calls.

**Services**

* **AuthBackendApiService**: Sends requests to login, register, reset password, etc.
* **ChatBackendApiService**: Sends requests to create chat, get messages, send messages, etc.
* **UserBackendApiService**: Sends requests to search users, check status, and get user info.
* **VideoBackendApiService**: Sends requests to start or respond to video calls.
* **WsServices**: Manages WebSocket connection and handles real-time events.

## 

## CHAPTER 4: IMPLEMENTATION

### 4.1 Backend Modules

The backend of Cphere is written in Rust using the Actix-web framework. It is organized into clear modules for configuration, models, services, middleware, REST API handlers, and WebSocket handling.

#### 4.1.1 Configuration and Shared State

* **config/database.rs**:  
   Loads the MONGODB\_URI from the environment and initializes the MongoDB client and database. This sets up the connection used throughout the backend.
* **states/app\_state.rs**:  
   Wraps the MongoDB client and database inside a shared AppState struct. It also holds references to active WebSocket sessions. AppState is passed to handlers using Actix-web’s dependency injection system.

#### 4.1.2 Models and Services

* **Models**:  
   Defined for User, Session, Chat, Message, and Notification. All models implement Serialize and Deserialize so they can be easily converted between Rust structs and JSON. These models match the MongoDB documents.
* **Services**:  
   Business logic is grouped into separate services:
  + auth\_service: Handles login, registration, logout, and password reset.
  + user\_service: Manages user data, presence, and profile details.
  + chat\_service: Manages chats and message operations.
  + notification\_service: Sends and stores notifications.
  + video\_call\_service: Handles video call request and response logic.
* **Utils**:
  + auth\_util: Provides helpers for hashing passwords, generating tokens, and verifying credentials.
  + validation\_util: Contains input validation logic like checking email format and password strength.

#### 4.1.3 Middleware and Session Handling

* **auth\_middleware.rs**:  
   Middleware that reads the session cookie from incoming requests using actix-session. It extracts and validates the session and user ID. If the session is missing or invalid, the request is rejected with an unauthorized response.

#### 4.1.4 REST Handlers

* **auth\_handler.rs**:
  + register: Validates input, hashes the password using Argon2 (utils/password.rs), and creates a new user.
  + login: Verifies credentials and creates a session.
  + logout: Clears the session cookie.
  + status: Returns information about the currently logged-in user.
* **user\_handler.rs**:  
   Fetches the list of chats and notifications for a user. Also includes endpoints to check online status and search for other users.
* **chat\_handler.rs**:  
   Handles chat creation and deletion. Retrieves messages with pagination and supports fallback message sending if WebSocket is unavailable.
* **ws\_handler.rs**:  
   Handles the WebSocket upgrade and connection setup. Registers the user and starts the WebSocket session actor.
* **video\_call\_handler.rs**:  
   Accepts requests to start or respond to video calls. These actions also trigger real-time notifications through WebSocket messages.

#### 4.1.5 WebSocket Actor

* **websocket\_session.rs**:  
   Defines a WebSocket session actor for each user connection. When a user connects, their user\_id is registered in a shared global map. The actor listens for incoming messages from the client and calls the appropriate service functions. It also handles forwarding WebSocket events to other users’ actors. This file includes logic for handling all WebSocket events listed in Section 3.5, such as chat\_message, webrtc\_offer, webrtc\_answer, and others.

### 

### 4.2 Frontend Modules

#### 4.2.1 Project Setup and Configuration

* Vite initializes React + TypeScript project. Tailwind CSS configured via tailwind.config.js with purge paths. Vite is configured in vite.config.ts.

#### 4.2.2 Authentication Flow

* AuthBackendApiService.ts: implements REST calls with axios.
* AuthContext.tsx: Keeps track of user authentication status.
* LoginPage.tsx, RegisterPage.tsx, ForgetPasswordPage.tsx, ResetPasswordPage.tsx all these pages implement the authentication features. If the user is already authenticated then it will be automatically redirected to the authenticated route.
* We also implement a WebSocket event listener for logout.

#### 4.2.3 Chat Components

* ChatBackendApiService.ts: REST methods for creating chat, fetching chats/messages, sending messages.
* ChatContext.tsx: maintains chats list and provides utility functions to manipulate it.
* ChatsPage.tsx, ChatBoard.tsx implement the api calls to facilitate seamless chatting experience.
* We implement a WebSocket event listener for delete\_chat, chat\_message.

#### 4.2.4 Notification Components

* NotificationContext.tsx: Maintains list of notifications and provides utility functions to manipulate them.
* NotificationIcon.tsx: This component displays notifications.
* It also implements the WebSocket Event listener for video\_call\_request.

#### 4.2.5 Video Call Implementation

* VideoBackendApiService.ts: It calls REST API /video\_call/initiate and /video\_call/respond. Caller calls the former and receiver calls the later.
* VideoCallPage.tsx: It uses WebRTC APIs for the following:
  + To create RTCPeerConnection
  + To fetch media stream via getUserMedia()
  + To exchange SDP via WebSocket events webrtc\_answer and webrtc\_offer
  + To handle ice\_candidate
* It displays local and remote <video> elements.
* It implements WebSocket event listeners for webrtc\_offer, webrtc\_answer, webrtc\_ice\_candidate, video\_call\_ended, video\_call\_accepted, video\_call\_declined.

### 4.3 Security Measures

#### 4.3.1 Password Hashing

The user passwords are securely hashed using the Argon2 algorithm. Argon2 is a modern, memory-hard hashing function designed to provide strong resistance against brute-force and side-channel attacks. It combines the benefits of resistance to side-channel attacks and resistance to GPU cracking to achieve robust security.

The configuration used in sets the time cost to 3 iterations, memory cost to 4096 KiB, and parallelism to 1 thread. These parameters ensure that each hash computation requires a non-trivial amount of time and memory, significantly increasing the effort required for attackers to perform dictionary or brute-force attacks, especially on modern hardware like GPUs or ASICs. By using Argon2 with conservative yet effective parameters, it ensures that stored passwords are well-protected without imposing excessive computational overhead on legitimate authentication operations.

#### 4.3.2 Session Cookie

In Cphere, session management is handled using the actix-session crate, which enhances security and integrity by issuing HTTP-only cookies that are signed and optionally encrypted. These cookies are marked with the SameSite=Lax attribute to help prevent Cross-Site Request Forgery (CSRF) attacks while maintaining usability for top-level navigations.

The session data itself is stored in a dedicated MongoDB collection via the Actix session storage integration. Each session entry includes metadata such as user ID and last access timestamp, and is automatically managed using a Time-To-Live (TTL) index. This ensures that expired sessions are periodically purged from the database, reducing storage overhead and enhancing security by limiting the window of token reuse. This approach balances security, scalability, and performance while providing persistent session support across server restarts.

#### 4.3.3 Input Validation

We implement a two-layered input validation strategy to ensure data integrity and enhance security.

On the backend, the utils/validation.rs module is responsible for validating critical user inputs. It checks that email addresses conform to standard formats using regular expressions and enforces password strength by requiring a minimum length. This server-side validation ensures that even if malicious clients bypass the frontend, the backend remains protected against malformed or weak input.

Complementing this, the frontend includes real-time form validation to provide immediate feedback to users, preventing submission of invalid or incomplete data. This combination of client-side and server-side validation helps maintain robust security and improves the overall user experience.

## 

## CHAPTER 5: TESTING AND EVALUATION

### 5.1 Testing Strategy

* As of now the automated testing is partially implemented, and the application relies on manual testing.
* Unit tests for the Rust backend are planned under the tests/unit/ directory. These will cover core functions like authentication, chat handling, and utilities. Implementation is currently pending and marked as TODO.
* Integration tests are planned in the tests/integration directory using TestServer and a temporary MongoDB instance. These will validate full REST and WebSocket flows. Implementation is marked as TODO.
* Frontend unit tests are planned to use Jest, Vitetest and React Testing Library in src/tests with mock API using msw.Implementation is marked as TODO.
* End-to-End Tests as of now are performed manually. The automated testing is planned as the project size grows (potential library: Playwright).

### 5.3 Usability Feedback

User feedback from two initial reviewers highlighted Cphere’s intuitive and responsive user interface. They reported that core features such as real-time chat updates, in-app notifications, and peer-to-peer video calling functioned smoothly without noticeable latency or glitches.The UI design was described as clean and user-friendly, contributing to an overall positive user experience.

However, the reviewers also suggested improvements, notably the addition of group chat functionality to enable multi-user conversations and the ability to share files within chats to enhance collaboration. These suggestions reflect practical use-case needs and have been noted for future development to further expand Cphere’s feature set.

## 

## CHAPTER 6: CONCLUSION AND FUTURE WORK

### 6.1 Conclusion

Cphere successfully delivers a self-hosted, full-stack chat platform that integrates core communication features into a cohesive and efficient system. It offers secure user authentication using session validation, persistent messaging through a MongoDB-backed data store, real-time notifications powered by WebSockets, and peer-to-peer video calling via WebRTC signaling.

The backend, built in Rust, ensures high concurrency and minimal latency through asynchronous execution, while the frontend, developed with React, provides a modern, responsive user interface. Together, these components demonstrate strong performance, scalability, and usability, making Cphere a robust foundation for real-time communication applications.

### 6.2 Future Work

To further enhance Cphere’s functionality and security, several improvements are planned.

* First, group chat support will be introduced by extending the existing Chats schema to include multiple participants and updating both backend logic and frontend UI components accordingly.
* Second, to ensure message confidentiality even against server-side compromise, end-to-end encryption will be implemented using a protocol such as the Double Ratchet Algorithm, similar to Signal.
* Third, mobile clients will be developed using cross-platform frameworks like React Native or Flutter, leveraging the existing REST APIs and WebSocket signaling infrastructure to deliver a consistent, real-time experience on Android and iOS devices.
* Lastly, support for file sharing and media attachments will be added using object storage services (e.g., AWS S3 or MinIO), allowing users to send and receive documents, images, and videos seamlessly within chats. These enhancements aim to make Cphere a more versatile, secure, and accessible communication platform.

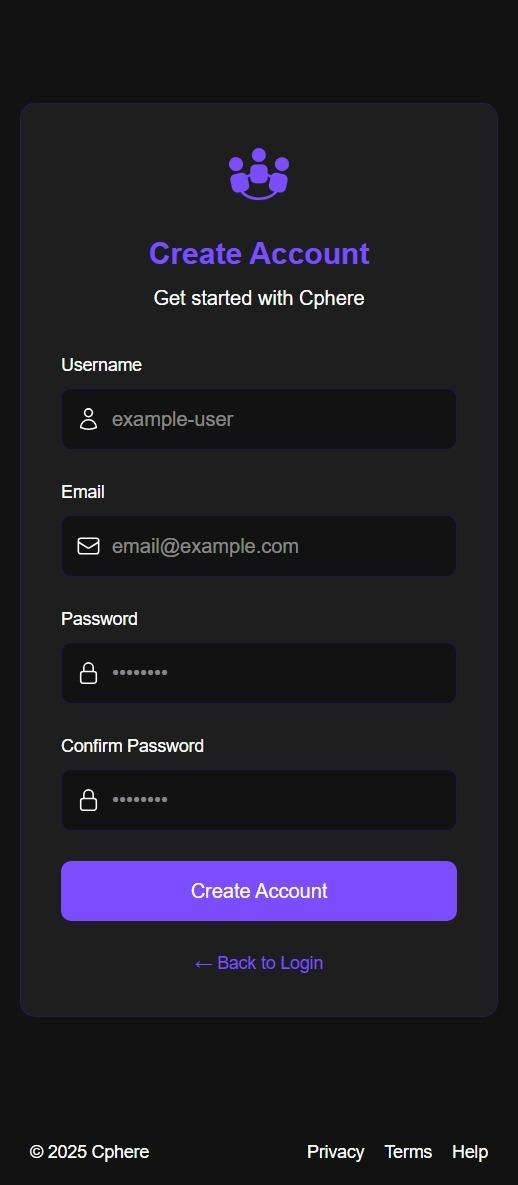
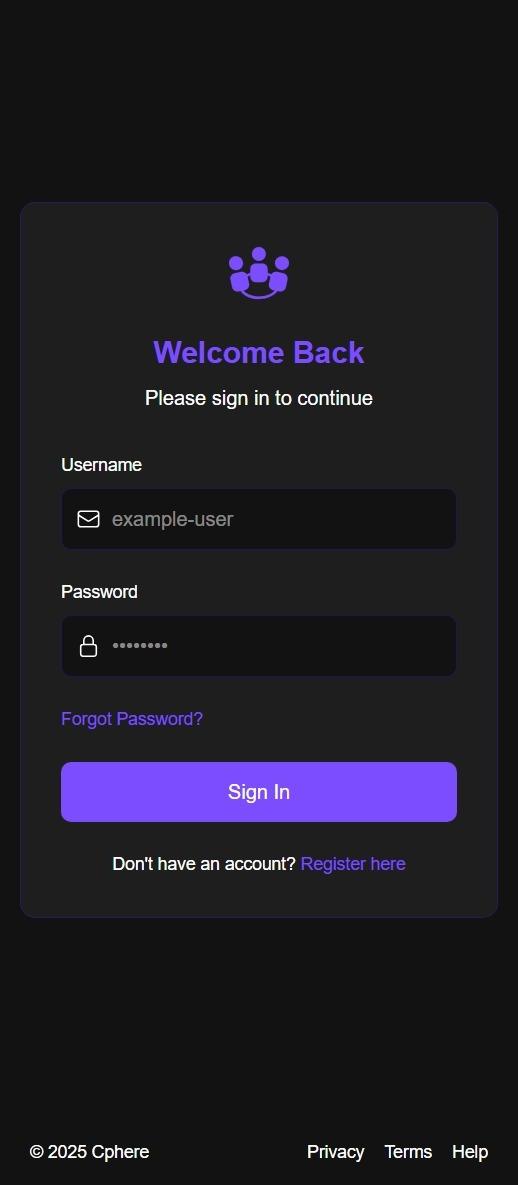
## 

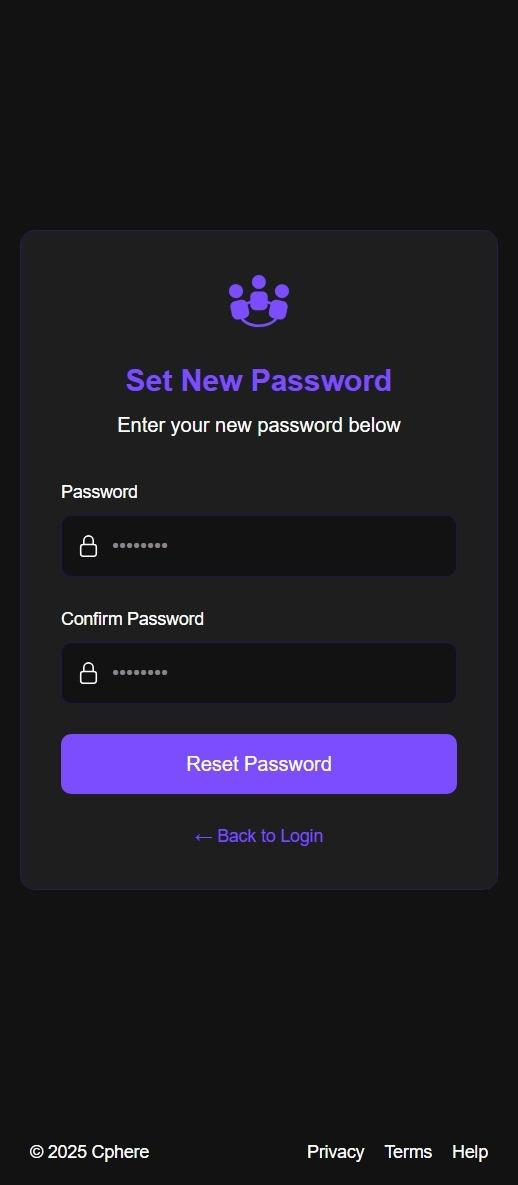
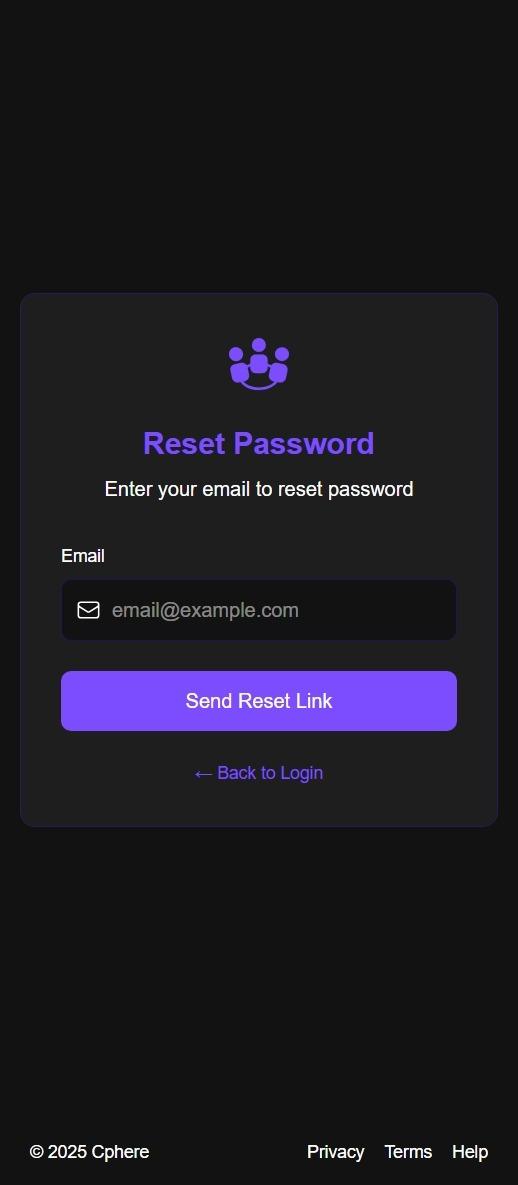
## REFERENCES

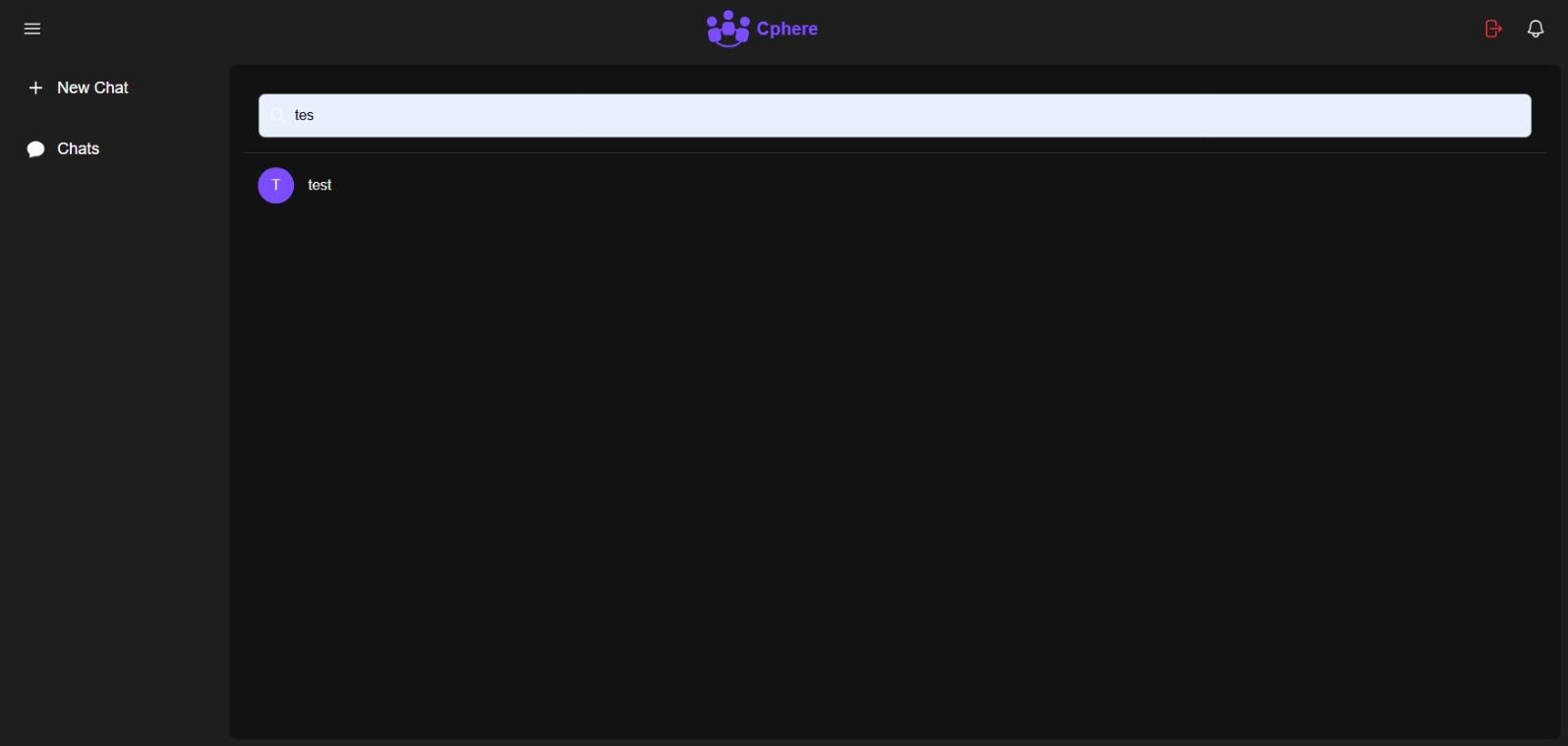
1. IETF, “RFC 6455: The WebSocket Protocol,” Dec. 2011.
2. Google, “WebRTC Overview,” [Online]. Available:<https://webrtc.org/>.
3. Actix Web Contributors, “Actix Web Documentation,” [Online]. Available:<https://actix.rs/docs/>.
4. Mozilla Developer Network, “Using WebRTC,” [Online]. Available:<https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API>.
5. Matrix.org, “Matrix: Open Standard for Decentralized Communication,” [Online]. Available:<https://matrix.org/>.
6. Rocket.Chat, “Rocket.Chat Documentation,” [Online]. Available:<https://docs.rocket.chat/>.
7. MongoDB Inc., “MongoDB Manual: Indexes,” [Online]. Available:<https://docs.mongodb.com/manual/indexes/>.
8. Actix Web Contributors, “actix-session Crate,” [Online]. Available:<https://crates.io/crates/actix-session>.
9. Vite Contributors, “Vite Documentation,” [Online]. Available:<https://vitejs.dev/>.
10. Tailwind Labs, “Tailwind CSS Documentation,” [Online]. Available:<https://tailwindcss.com/docs>.
11. Rust Docs, “dotenv Crate,” [Online]. Available:<https://crates.io/crates/dotenv>.
12. Argon2 Docs, “argon2 Crate,” [Online]. Available:<https://crates.io/crates/argon2>.
13. mongoDB Rust Driver, “mongodb Crate,” [Online]. Available:<https://docs.rs/mongodb/>.
14. Actix Web Contributors, “WebSocket in Actix,” [Online]. Available:<https://actix.rs/docs/websocket/>.
15. WebRTC.org, “WebRTC Signaling,” [Online]. Available:<https://webrtc.org/getting-started/signaling>.
16. React Docs, “Context API,” [Online]. Available:<https://reactjs.org/docs/context.html>.
17. React Docs, “Hooks Reference,” [Online]. Available:<https://reactjs.org/docs/hooks-reference.html>.
18. Axios Docs, “Axios HTTP Client,” [Online]. Available:<https://axios-http.com/docs/intro>.
19. MDN, “WebSocket API,” [Online]. Available:<https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>.
20. React Router Docs, “React Router,” [Online]. Available:<https://reactrouter.com/>.
21. React Testing Library, “Testing Components,” [Online]. Available:<https://testing-library.com/docs/react-testing-library/intro/>.
22. Andrea Bittau, “Argon2: Password Hashing,” Passwords Conference 2016.
23. Actix Benchmarks, “Actix Web Performance,” [Online]. Available:<https://actix.rs/blog/actix-web-benchmarks/>.
24. Google, “WebRTC Performance Metrics,” [Online]. Available:<https://webrtc.org/performance/>.

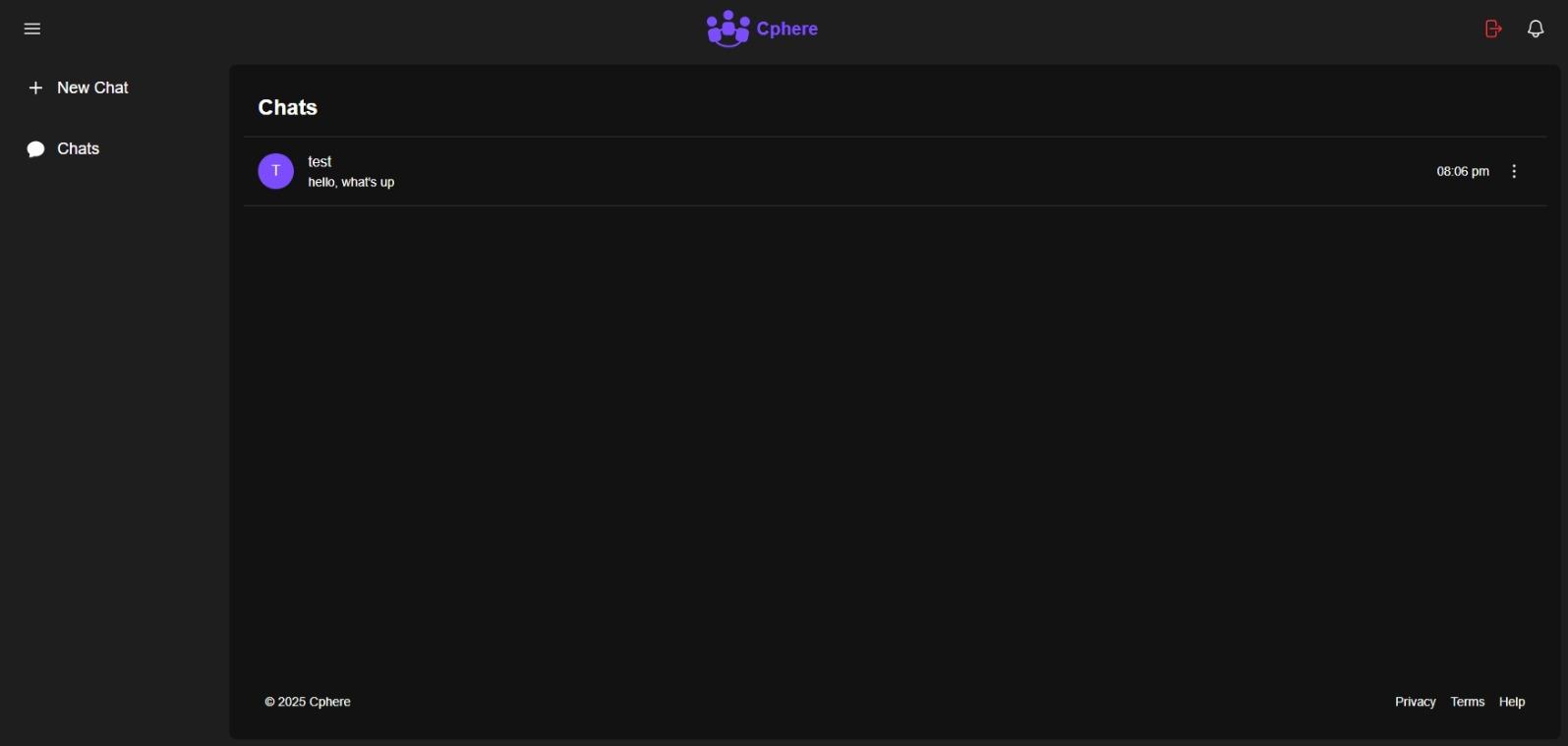
## APPENDIX A: UI SCREENSHOTS

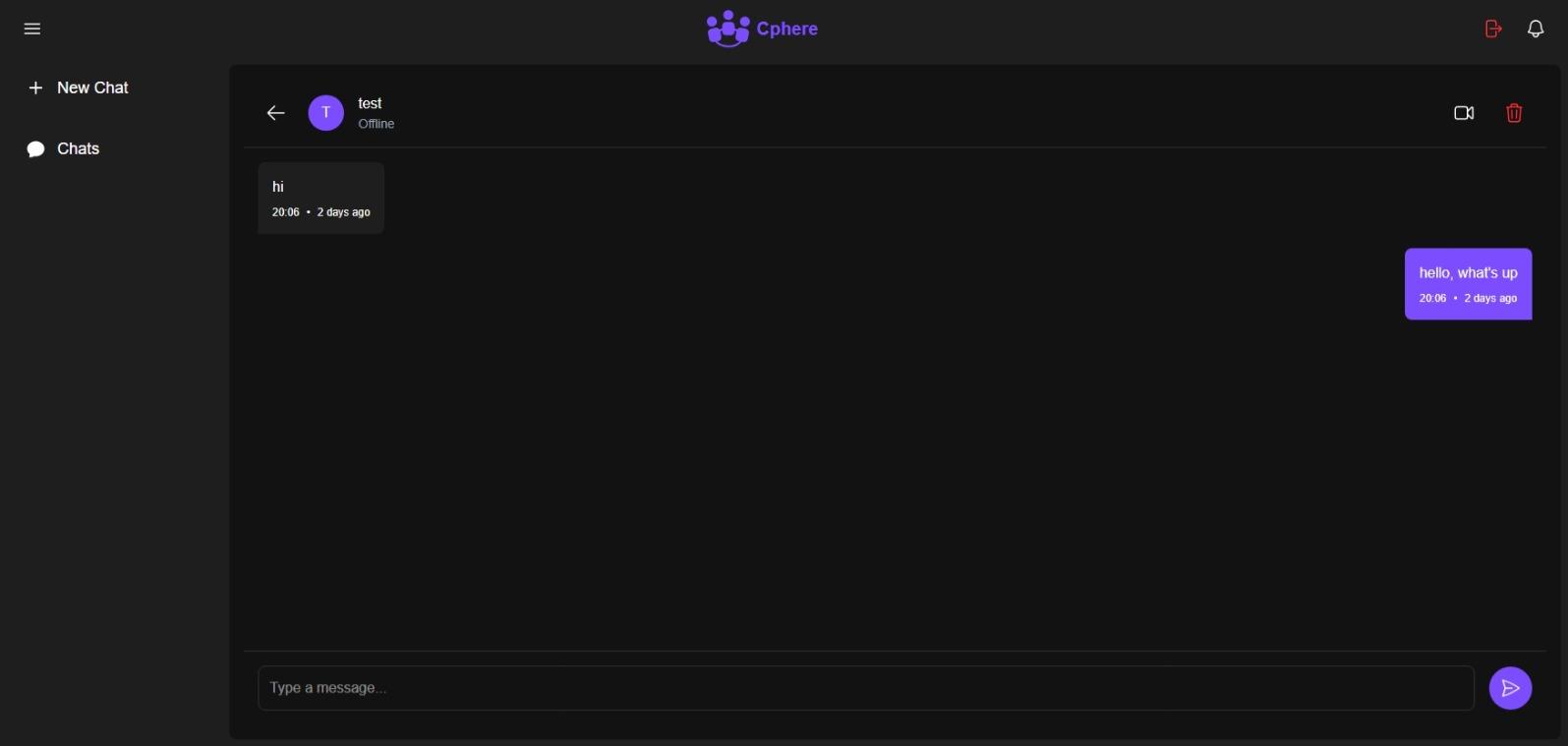
**B.1 Authentication Pages**



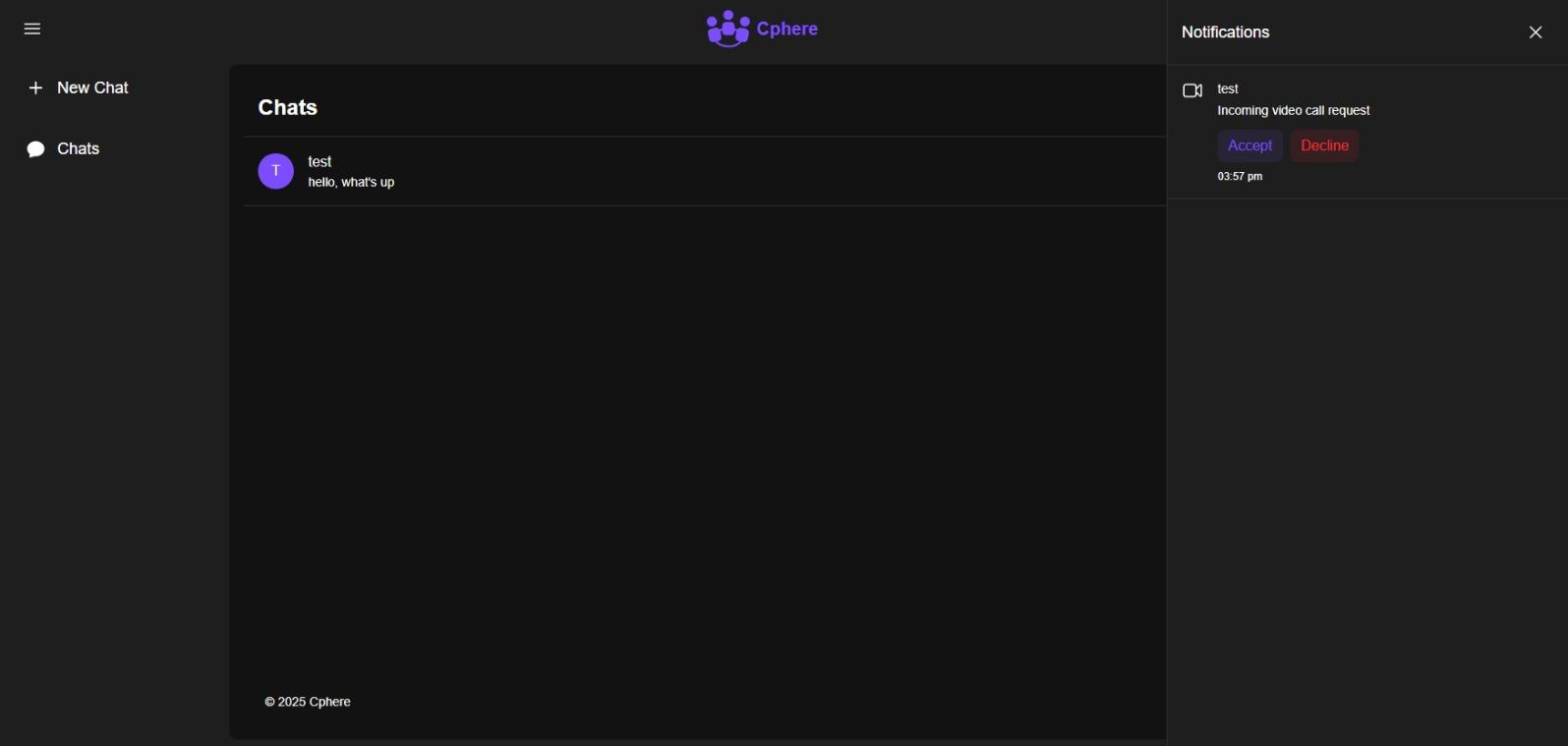


**B.2 Search Page**  


**B.3 Chat Interface**  




**B.4 Notification Component**



**B.5 Video Call Page**

