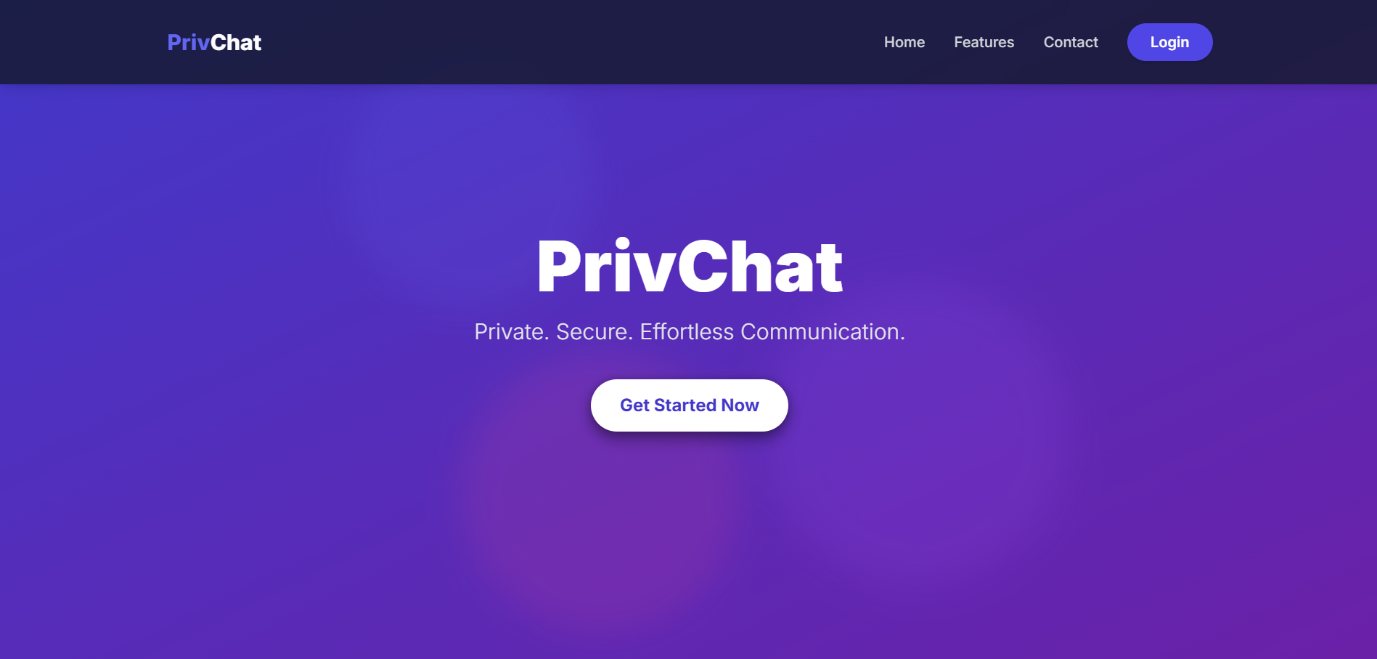
Project Report

**Title: PrivChat - A Secure Peer-to-Peer Communication Application**

**1. Title Description**

**PrivChat** is an innovative chat program carefully designed to enable straightforward machine-to-machine (M2M) or peer-to-peer (P2P) communication. The fundamental design principle of PrivChat is to remove the necessity of intermediate servers for message transmission and storage of chat messages, and hence guarantee immaculate user privacy and data protection. The fundamental purpose of the application is to create a secure and straightforward pipeline for data exchange between the communicating parties, and hence it is the best solution for individuals and organizations that require exclusive control of their confidential conversations.



**2. ABSTRACT**

**PrivChat** is a lightweight, privacy-focused, real-time communication platform built using Express.js, WebSockets, MySQL, and WebRTC. The application emphasizes direct peer-to-peer encrypted messaging, ensuring both simplicity and security in personal online conversations.

On the home page, users can choose to either create or join a private room using a 6–8 digit PIN code. The uniqueness of PrivChat lies in its room-approval mechanism—when a user tries to join a room, the room creator must explicitly approve or deny the request, ensuring full control over who gets access.

PrivChat uses WebRTC DataChannels for direct communication between two or more users. This means messages are exchanged peer-to-peer and are never stored on the server—providing true end-to-end encryption. All chat data vanishes on refresh, and users are returned to the home page, further ensuring privacy and session isolation.

Technical Architecture

* Frontend: Built with vanilla JavaScript and Bootstrap for a clean, responsive UI. Includes dynamic status indicators (like online/offline states and room participant visuals).
* Backend: Utilizes Express.js with WebSockets for real-time signaling and MySQL for user authentication and room management.
* WebRTC: Handles peer connection setup, ICE candidate exchange, and data channel establishment for secure and fast communication.

Guide name : Mr. R. Anbarasu Sir

B.Harish Gandhi – 22N81A62B3

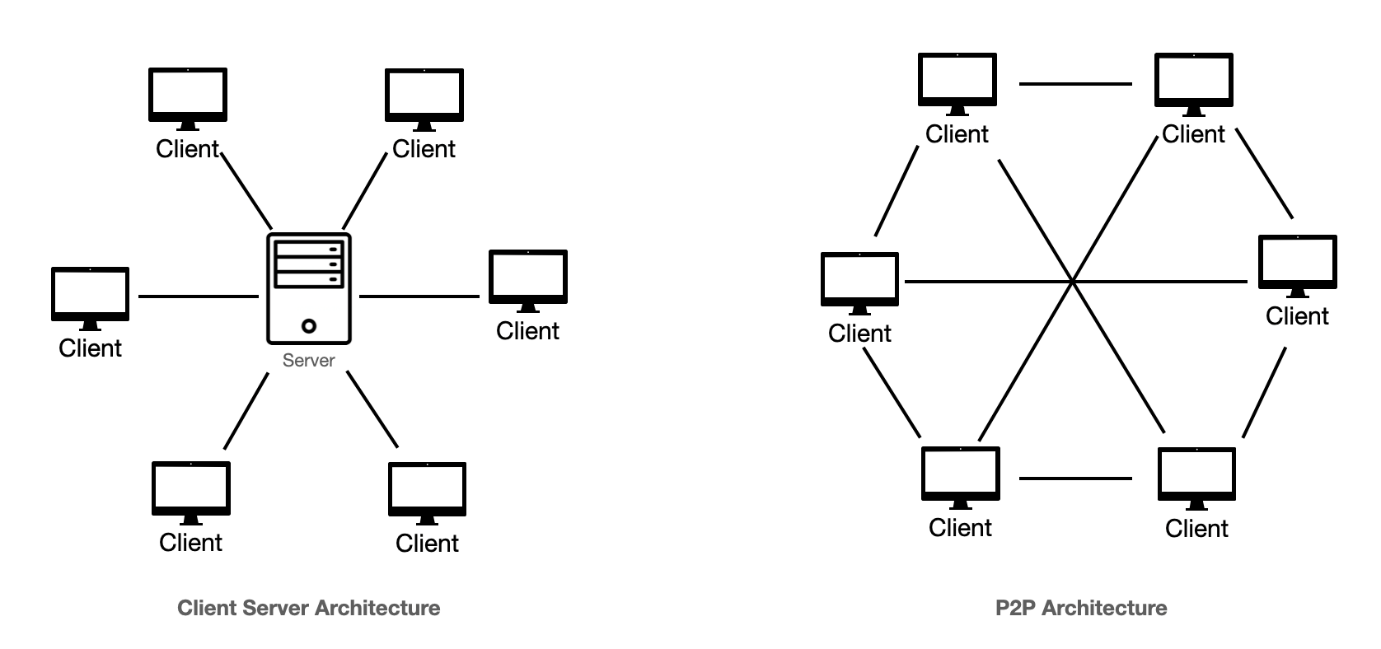
K.Sai Tejaswini – 22N81A62A9

K.Shirisha – 22N81A6268  
K.Dinesh Karthik – 22N81A62C0

Sign :

**3. Domain Description**

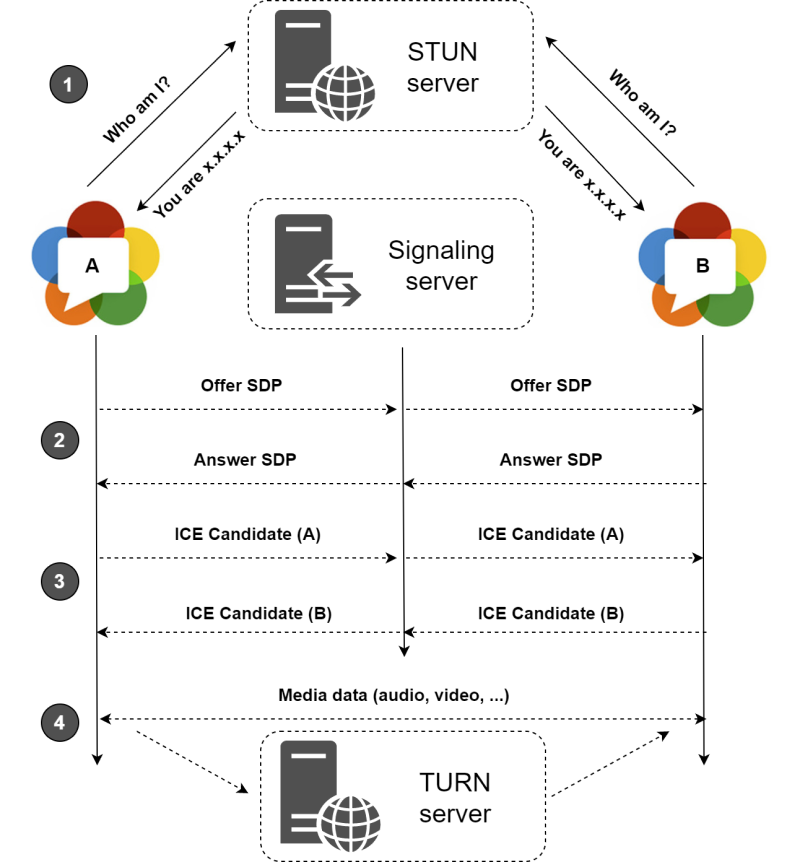
The digital communication sector has grown fast, uniting billions of people worldwide. But the growth has affected privacy. Common chat applications use a client-server paradigm, with a main server gathering messages from a user and sending them to another user. The system is straightforward and can scale, but it also brings privacy and security concerns. User data, including private conversations, is kept on the main servers and thus vulnerable to data leaks, unauthorized access by service providers, and government monitoring, as well as single points of failure. The need for privacy, integrity, and evidence in digital conversations has thus become high, especially for people and organizations dealing with sensitive information. This has generated a higher focus on decentralized and peer-to-peer (P2P) communication systems that strive to regain the users' control over data through direct communication.



**4. Literature Survey**

Chat applications have evolved largely on the model of a centralized client-server architecture, with WhatsApp, Telegram, and Slack being excellent examples. Earlier versions used to transmit information without robust encryption, which raised serious privacy concerns. Subsequent releases introduced end-to-end encryption (E2EE) to most of these centralized systems, allegedly protecting information in transit. But even with E2EE, the metadata (who talks to whom, when, and with what frequency) is typically exposed to the server. More significantly, the server still performs sending and temporary storage of the encrypted messages, and users must have faith in the service provider not to compromise the encryption keys or the server infrastructure itself.

The advents of Web Real-Time Communication (WebRTC) have transformed the way we are able to communicate directly through the browser. WebRTC enables the user to establish direct audio, video, and data channels without the intervention of other servers to transfer the media or data. While a signaling server is still required initially to assist in getting connected (e.g., exchanging IP addresses, assisting with NAT traversal, and session data), once the direct connection is established, the data is transferred directly between peers without the server. This ability is the basis for real private communication channels. Most research balances the trade-off between the convenience of using centralized systems and the increased privacy of decentralized/P2P approaches, and WebRTC is a leading technology for making the latter become a reality on the web.



**5. Existing System and Drawbacks**

Existing System: Centralized Chat Applications

Most chat applications (like WhatsApp, Discord, Slack, and Messenger) work on the client-server architecture.

• Clients: Client devices (web browsers, mobile apps).

• Server: A robust main server system belonging to and operated by the service provider.

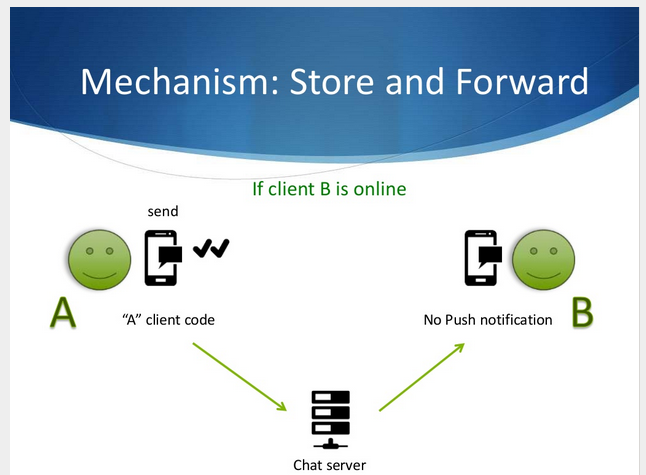
**How they typically work:**

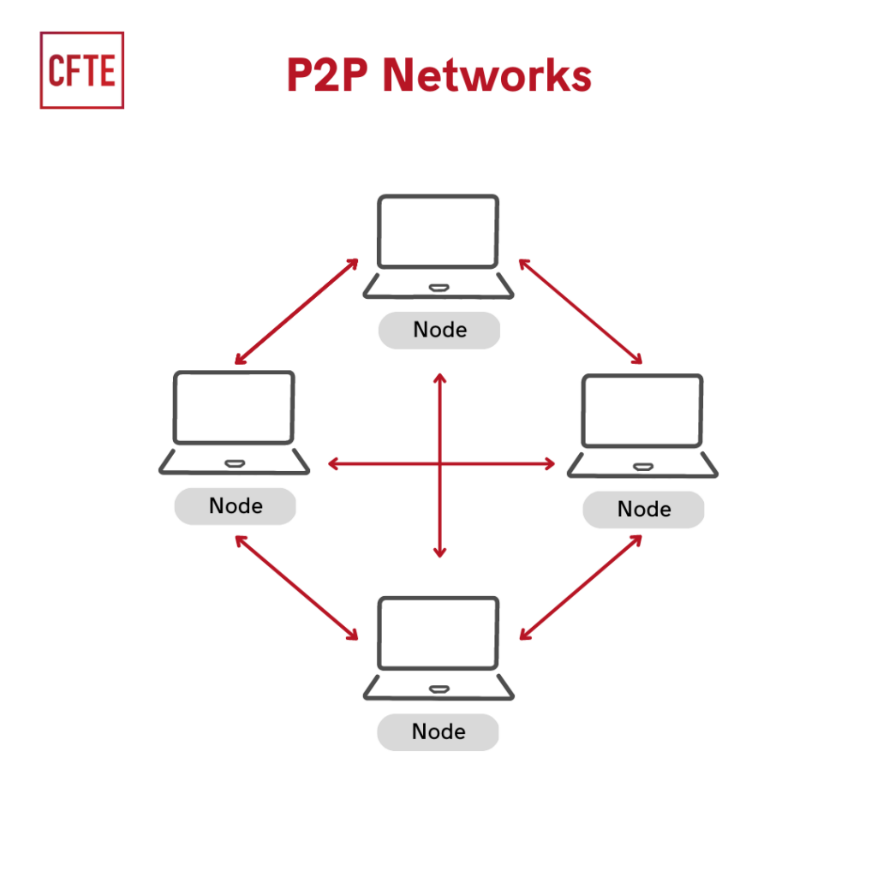
When User A sends a message to User B:

1. User A's customer sends the message to the main server.
2. The primary server accepts, handles, and usually keeps the message (at least temporarily, sometimes indefinitely).
3. The main server then forwards the message to User B's client.

**Disadvantages:**

* Data Storage and Interception: The most serious restriction. The central server is an intermediary that monitors and often stores all message content (even if encrypted, metadata is revealed). This is a high-risk threat of data leaks, surveillance, and unauthorized storage of data.
* Single Point of Failure: If the master server fails, communication ceases entirely.
* Trust Matters: The users should have complete trust in the service provider with their personal data, including how they handle privacy rules and security procedures.
* Latency: Messages must travel to and from the central server, potentially introducing latency, especially if peers are geographically close but the server is distant.
* Bandwidth Consumption: All data, regardless of peer proximity, must traverse the central server's network.





**6. Proposed System and Advantages**

**Proposed System: PrivChat - A Hybrid P2P Communication System**

PrivChat proposes a **hybrid architecture** that combines the practicality of a server for **authentication and signaling** with the absolute privacy of **peer-to-peer data channels**.

* **Authentication & Signaling Server (Node.js):** This server's role is strictly limited to:
  + Managing user accounts (registration, login) with secure password hashing (bcrypt).
  + Authenticating users via sessions (express-session).
  + Facilitating the discovery and connection negotiation between peers using WebSockets. This involves relaying "signaling" messages (like WebRTC offers, answers, and ICE candidates) which contain connection information, not actual chat data.
  + Managing chat room creation and join approvals.
* **Peer-to-Peer Data Channel:** Once signaling is complete and peers have exchanged necessary connection information, a direct communication channel is established between their browsers (ideally leveraging WebRTC DataChannels). All subsequent chat messages flow directly over this channel.



**Advantages:**

* **Absolute Data Privacy:** This is the paramount advantage. Since chat messages never touch or pass through an intermediate server, there is no central point for data capture, storage, or forwarding. This inherently safeguards user conversations from server-side breaches, third-party interception, and provider surveillance.
* **Enhanced Security:** The direct P2P data channel, when implemented with WebRTC, benefits from mandated DTLS encryption, securing data in transit between the endpoints.
* **Reduced Server Burden:** The server is freed from the heavy task of relaying and storing high volumes of chat data, allowing it to scale more efficiently for managing connections.
* **Minimized Trust Requirement:** Users only need to trust the server for initial connection setup and authentication, not for the ongoing confidentiality of their actual messages.
* **Lower Latency (for data):** Once direct P2P connections are established, messages can travel directly between peers, potentially reducing latency compared to round-trips through a distant server.
* **Resilience:** While the signaling server is a single point of failure for *initiating* connections, once connections are established, communication continues even if the signaling server temporarily goes offline (though new connections cannot be formed).

**7. Requirements**

To implement and operate PrivChat, the following software and hardware resources are required:

**7.1. Software Requirements**

* **Operating System:** Windows, Linux, macOS (for both server and client development/deployment).
* **Backend Environment:**
  + **Node.js:** Runtime environment (LTS version recommended, e.g., v18.x or v20.x).
  + **npm (Node Package Manager):** For managing Node.js dependencies.
  + **Express.js:** Web framework for handling HTTP requests and routing.
  + **ws:** WebSocket library for real-time bidirectional communication (signaling).
  + **bcrypt:** Library for password hashing.
  + **express-session:** Middleware for session management.
  + **mysql:** Node.js driver for MySQL database interaction.
  + **hbs (Handlebars.js):** Templating engine for server-side rendering of views.
* **Database:**
  + **MySQL Server:** Version 8.0+ recommended.
  + **MySQL Client/Workbench:** For database administration and schema management.
* **Frontend Technologies:**
  + **HTML5, CSS3, JavaScript:** Standard web technologies for client-side development.
  + **Bootstrap 5.3:** CSS framework for responsive and modern UI components.
  + **WebRTC API (Browser Support):** Essential for establishing peer-to-peer data channels. Modern web browsers (Chrome, Firefox, Edge, Safari) typically have strong WebRTC support.
* **Development Tools:**
  + **Code Editor:** Visual Studio Code, Sublime Text, etc.
  + **Web Browser:** For testing the client application.

**7.2. Hardware Requirements**

* **Server (for Signaling and Authentication):**
  + **Processor:** Quad-core CPU or higher.
  + **RAM:** 8GB RAM or more.
  + **Storage:** 250GB SSD (for OS, Node.js, MySQL, and logs).
  + **Network:** Stable internet connection with sufficient bandwidth.
* **Client (User Devices):**
  + **Processor:** Dual-core CPU or higher.
  + **RAM:** 4GB RAM or more.
  + **Storage:** Minimal, as chat data is not stored locally by the application.
  + **Network:** Stable internet connection.
  + **Webcam/Microphone (Optional):** Not required for the current chat-only functionality, but would be needed for future video/audio call features.

**8. Modules Description**

The PrivChat application can be logically divided into the following key modules, each responsible for distinct functionalities:

1. **User Management Module:**
   * **Purpose:** Handles all aspects of user authentication and account management.
   * **Sub-modules/Functions:**
     + **Registration:** Collects user details (fullname, email, username, password, mobile) and securely stores them in the database after hashing the password with bcrypt. Includes checks for duplicate usernames/emails.
     + **Login:** Authenticates users by comparing provided credentials with stored hashes using bcrypt.compare().
     + **Session Management:** Creates and manages user sessions (express-session) upon successful login, allowing authenticated access to protected routes.
     + **Logout:** Terminates user sessions.
   * **Technologies:** Express.js, MySQL, bcrypt, express-session.
2. **Web Interface Module:**
   * **Purpose:** Provides the user-facing web pages and handles navigation.
   * **Sub-modules/Functions:**
     + **Routing:** Defines routes for home, login, register, dashboard, and chat pages.
     + **View Rendering:** Uses Handlebars.js (HBS) to dynamically render HTML content.
     + **Static File Serving:** Serves static assets like CSS and JavaScript files.
   * **Technologies:** Express.js, HBS, HTML, CSS (Bootstrap).
3. **Room Management Module:**
   * **Purpose:** Manages the creation, joining, and state of private chat rooms.
   * **Sub-modules/Functions:**
     + **Create Room:** Allows a user to create a new chat room identified by a unique PIN. The creator becomes the host.
     + **Join Room:** Enables other users to attempt to join an existing room using its PIN.
     + **Approval System:** Implements a host-approval mechanism for joining requests, ensuring controlled access to private rooms.
     + **Room State Management:** Keeps track of active rooms, connected clients, and associated usernames.
   * **Technologies:** Node.js (server-side logic), ws (WebSockets for signaling).
4. **Signaling Module:**
   * **Purpose:** Facilitates the initial connection establishment between peers for direct communication.
   * **Sub-modules/Functions:**
     + **WebSocket Server:** Handles incoming WebSocket connections from clients.
     + **Message Parsing:** Interprets signaling messages (e.g., create, join, offer, answer, ice-candidate, approval\_response).
     + **Message Forwarding:** Relays WebRTC signaling messages (offer, answer, ice-candidate) directly between the intended peers in a specific room.
     + **Peer Tracking:** Maintains a mapping of connected WebSockets to rooms and users.
   * **Technologies:** ws (WebSocket server), JavaScript.
5. **Peer-to-Peer Data Channel Module (Client-Side):**
   * **Purpose:** Handles the actual direct communication of chat messages between connected users.
   * **Sub-modules/Functions:**
     + **WebRTC Peer Connection:** Initiates and manages the WebRTC RTCPeerConnection object.
     + **WebRTC DataChannel:** Creates and manages the RTCDataChannel for sending and receiving text messages directly between peers.
     + **Message Display:** Renders incoming and outgoing chat messages within the client's chat interface, distinguishing between sent and received messages and displaying sender names.
   * **Technologies:** JavaScript (client-side), WebRTC API.

**9. Architecture**

PrivChat operates on a **hybrid client-server and peer-to-peer (P2P)** architecture, specifically designed to enforce the "no intermediate server data" principle.

**9.1. Architectural Layers**

1. **Client Layer (Browser):**
   * Runs the web application (HTML, CSS, JavaScript).
   * Manages user interface, input, and display.
   * Establishes **HTTP connections** to the Node.js server for page requests, login, and registration.
   * Establishes **WebSocket connections** to the Node.js server for signaling.
   * Establishes **direct WebRTC P2P DataChannels** with other clients for chat message exchange. This is where the core data privacy is achieved.
2. **Server Layer (Node.js/Express/WS):**
   * Acts as an **Authentication and Signaling Server**.
   * **Express.js:** Handles all HTTP routes (login, register, dashboard, chat pages), session management, and serving static files.
   * **WebSocket Server (**ws**):** Manages real-time, bidirectional WebSocket connections for communication necessary for WebRTC connection establishment. It relays signaling messages (offer, answer, ICE candidates) between peers. It also manages room states and peer approval processes.
   * **Core Principle:** This server *never* processes, stores, or forwards the actual chat messages. Its function is to facilitate the direct connection between clients.
3. **Database Layer (MySQL):**
   * Stores persistent user data, including hashed passwords, full names, emails, and usernames.
   * Accessed exclusively by the Node.js server for user management operations.

**9.2. Data Flow Diagram**

**Phase 1: Authentication & Signaling (Server Interaction)**

1. **User Registration/Login:** Client (Browser) sends HTTP POST request to Node.js/Express server. Server interacts with MySQL DB for credential storage/verification (with bcrypt).
2. **Page Requests:** Client sends HTTP GET requests for application pages (dashboard, chat) to Node.js/Express server.
3. **WebSocket Connection:** Client establishes a WebSocket connection with the Node.js WebSocket server.
4. **Room Creation/Join Request:** Client sends WebSocket messages (e.g., create or join type) to the WebSocket server with room PIN and username.
5. **Approval Process:** If joining, the WebSocket server forwards an approval\_request to the host's client. The host's client sends an approval\_response back to the server, which then notifies the joining client.
6. **WebRTC Signaling:** Clients exchange WebRTC signaling messages (offer, answer, ice-candidate) via the WebSocket server. The server simply relays these messages between the two specific clients intending to connect.

**Phase 2: Direct Peer-to-Peer Data Communication (No Server Interaction)**

1. **Direct Data Channel Establishment:** After successful signaling, the clients' WebRTC implementations establish a direct P2P connection (DataChannel).
2. **Chat Message Exchange:** Once the DataChannel is open, chat messages sent by User A flow directly over this secure, encrypted channel to User B, and vice-versa. The Node.js server is completely bypassed for this data flow.

**9.3. Conceptual Architectural Diagram**

+------------------------------------+

| User Browser 1 |

| +------------------------------+ |

| | Client-Side JavaScript | |

| | - UI & Chat Logic | |

| | - WebSocket Client | |

| | - WebRTC PeerConnection | |

| | - RTCDataChannel (for chat) | |

| +------------------------------+ |

+------------------------------------+

| ^ ^

| | | HTTP (Auth/Pages)

| | |

| | | WebSocket (Signaling: offer/answer/ICE)

| | |

V | |

+------------------------------------+

| Node.js Server |

| +------------------------------+ |

| | Express.js (HTTP Routing) | |

| | - User Auth & Sessions | |

| | - Serve HBS views | |

| +------------------------------+ |

| | WS Server (WebSocket) | |

| | - Room Management | |

| | - Approval Requests | |

| | - WebRTC Signaling Relay | |

| +------------------------------+ |

+------------------------------------+

| |

| | Data Access

V V

+------------------------------------+

| MySQL Database |

| (User Credentials, Hashed Passwords)|

+------------------------------------+

^ ^

| | Direct DataChannel (Encrypted P2P Chat Data)

| |

| |

+------------------------------------+

| User Browser 2 |

| +------------------------------+ |

| | Client-Side JavaScript | |

| | - UI & Chat Logic | |

| | - WebSocket Client | |

| | - WebRTC PeerConnection | |

| | - RTCDataChannel (for chat) | |

| +------------------------------+ |

+------------------------------------+

**10. UML Diagrams**

(As an AI, I cannot directly generate graphical UML diagrams. However, for a complete project report, you would typically include the following diagrams to visually represent the system's structure and behavior.)

**Recommended UML Diagrams for PrivChat:**

1. **Use Case Diagram:**
   * **Actors:** User, Room Host.
   * **Use Cases:** Register Account, Log In, Log Out, Create Room, Join Room, Approve Join Request, Send Chat Message, Receive Chat Message.
   * **Purpose:** Illustrates the functional requirements of the system from the user's perspective.
2. **Class Diagram:**
   * **Classes (conceptual):** User, Room, Message (client-side representation), WebSocketClient, PeerConnection.
   * **Attributes:** Relevant data fields for each class (e.g., User: username, hashedPassword; Room: pin, clients, usernames).
   * **Relationships:** Associations between classes (e.g., a Room has Clients, a User sends Messages).
   * **Purpose:** Shows the static structure of the system's objects, their attributes, and relationships.
3. **Sequence Diagram (for key interactions):**
   * **Sequence: User Registration:** User -> Browser -> Express Server -> MySQL DB -> Express Server -> Browser.
   * **Sequence: User Login:** User -> Browser -> Express Server -> MySQL DB -> Express Server -> Browser.
   * **Sequence: Room Creation & Join:**
     + User A (Host) -> Browser -> WS Server (create room).
     + User B (Joiner) -> Browser -> WS Server (join room).
     + WS Server -> User A (approval request).
     + User A -> WS Server (approval response).
     + WS Server -> User B (approval result).
   * **Sequence: WebRTC Signaling & Data Channel Establishment:**
     + User A <-> WS Server (send offer) -> User B.
     + User B <-> WS Server (send answer) -> User A.
     + User A <-> WS Server (send ICE candidate) -> User B.
     + User B <-> WS Server (send ICE candidate) -> User A.
     + **User A <-------- Direct RTCDataChannel --------> User B (Actual Chat Data Flow)**
   * **Purpose:** Illustrates the dynamic behavior of the system by showing the order of interactions between objects over time.
4. **Deployment Diagram:**
   * **Nodes:** Client Device (Browser), Node.js Server, MySQL Database.
   * **Components/Artifacts:** Frontend application (HTML/CSS/JS), Backend application (Node.js code), MySQL database instance.
   * **Connections:** Represents network connections (HTTP, WebSockets, DB connection).
   * **Purpose:** Shows the physical deployment of software components on hardware nodes.

**11. Implementation**

**11.1. Algorithms**

* **Password Hashing (Bcrypt):**
  + **Algorithm:** bcrypt's key derivation function (KDF) is used. It's an adaptive hash function, meaning its computational cost can be adjusted, making it resistant to brute-force attacks by requiring more time to compute hashes.
  + **Process:**
    1. A random **salt** is generated.
    2. The user's plain-text password is mixed with the salt.
    3. This salted password undergoes multiple rounds of a cryptographic hashing process (controlled by the "cost factor," typically 10).
    4. The final hash, including the salt, is stored.
  + **Verification:** During login, bcrypt.compare() re-hashes the provided password with the stored salt (extracted from the stored hash) and compares the result to the stored hash.
* **WebRTC Signaling:**
  + **Algorithm/Process:** This is a crucial, though not strictly cryptographic, algorithm for P2P connection.
    1. **Offer/Answer Model (SDP Exchange):** One peer (offerer) creates a Session Description Protocol (SDP) "offer" detailing its capabilities (codecs, network information). This is sent via the signaling server to the other peer (answerer).
    2. The answerer processes the offer and generates an SDP "answer" describing its own capabilities, which is sent back via the signaling server to the offerer.
    3. **ICE Candidate Exchange:** Both peers discover their network addresses (local, STUN, TURN) and encapsulate them as ICE (Interactive Connectivity Establishment) candidates. These candidates are continuously exchanged via the signaling server until a viable direct path is found.
  + **Purpose:** To enable two peers, often behind NATs and firewalls, to find and connect to each other.

**11.2. Coding**

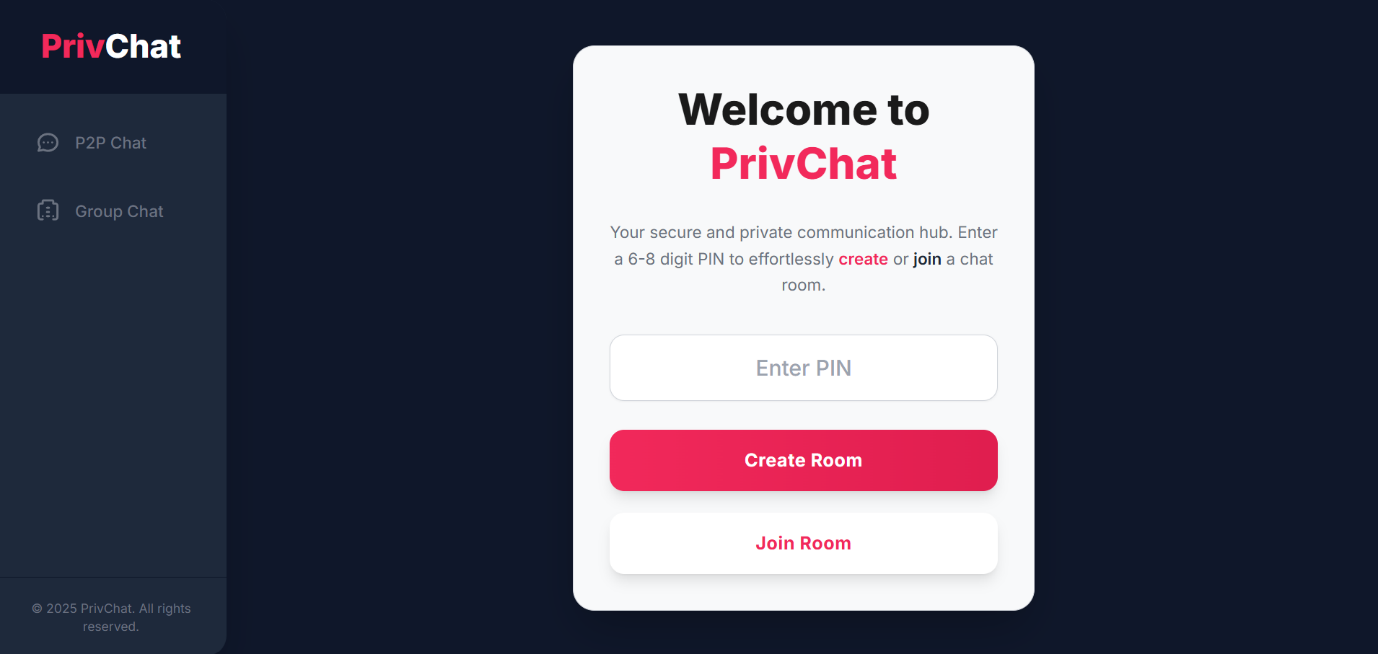
* **Backend (Node.js, Express, WS):**
  + **Language:** JavaScript.
  + **Structure:** Follows typical MVC (Model-View-Controller) patterns for API endpoints and view rendering. Middleware is extensively used for session management and authentication.
  + **Database Interaction:** mysql module is used for asynchronous queries to the MySQL database. Prepared statements are implicitly used where ? placeholders are present for security.
  + **WebSocket Handling:** The ws library provides a robust API for managing WebSocket connections, handling incoming messages, and broadcasting/sending data to specific clients.
* **Frontend (HTML, CSS, JavaScript, HBS):**
  + **Language:** HTML, CSS, JavaScript.
  + **Structure:** HTML provides the page layout and elements. CSS (with Bootstrap) handles styling and responsiveness. JavaScript manages dynamic content updates, user interactions, and critically, the client-side WebSocket and WebRTC logic.
  + **WebRTC Implementation:** Client-side JavaScript code sets up RTCPeerConnection objects, creates RTCDataChannel instances, handles ICE candidate events, and processes SDP offers/answers received via the WebSocket connection to establish the direct P2P link.
  + **Templating:** Handlebars.js (.hbs files) provides a clean way to inject dynamic data from the server into HTML pages.

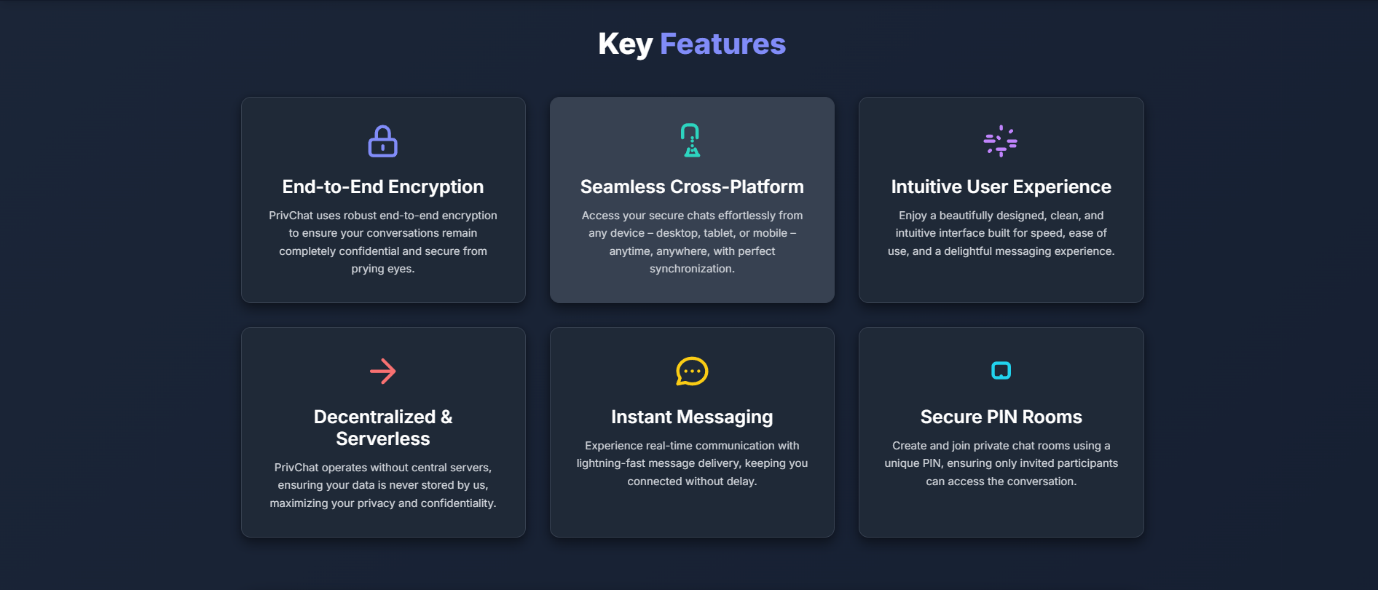
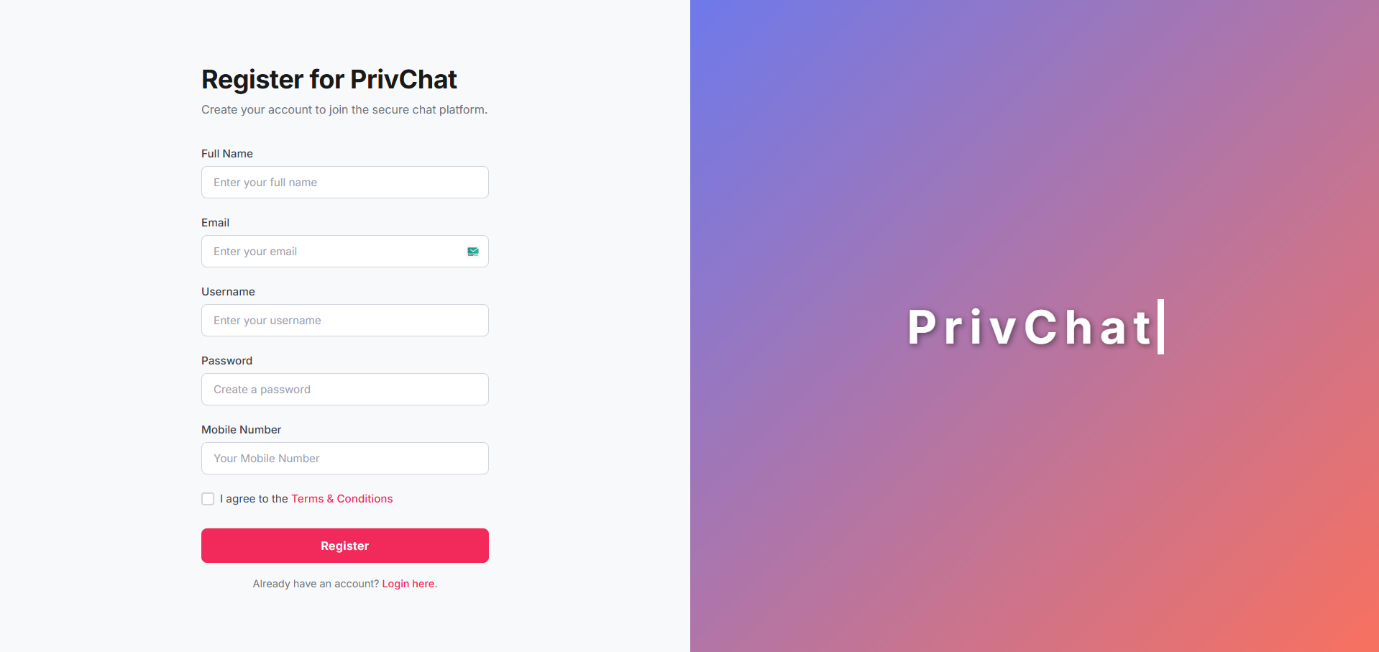
**11.3. Testing**

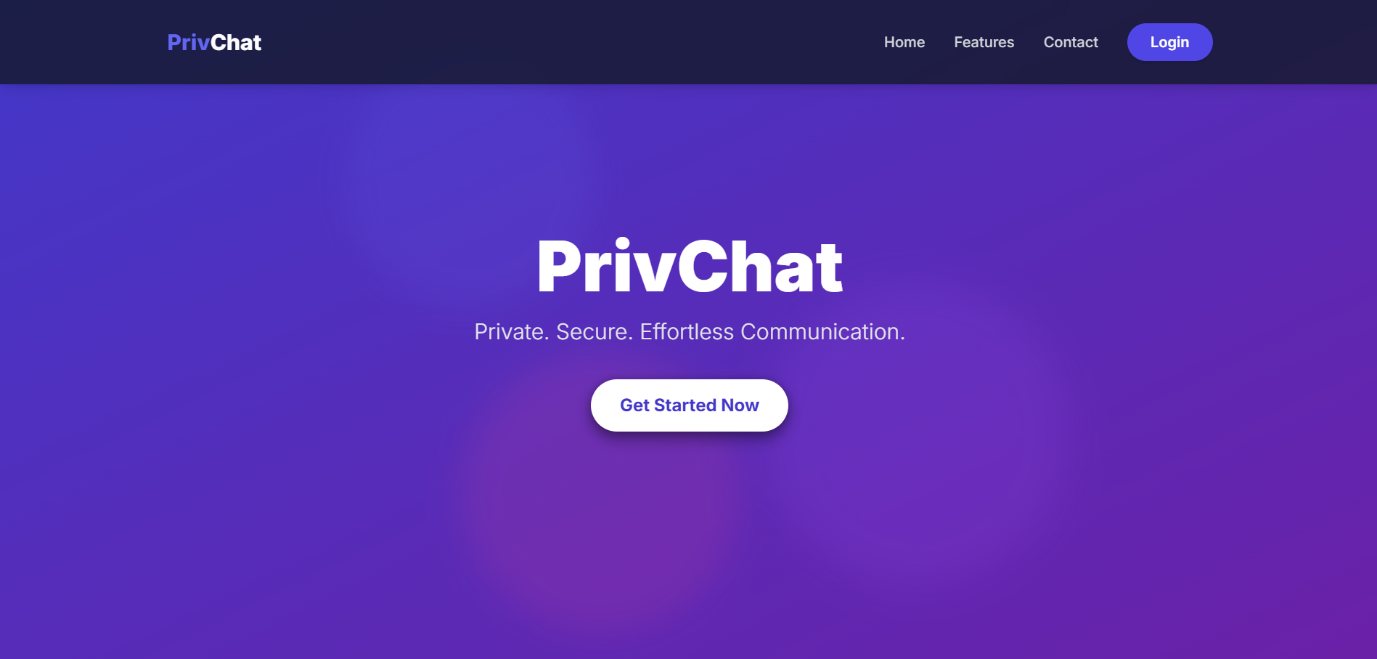
* **Unit Testing (Conceptual):**
  + Individual functions and modules (e.g., bcrypt hashing/comparison utility functions, database query functions, WebSocket message parsing logic) would be tested in isolation to ensure they work as expected.
  + *Note: Specific unit test code was not provided in the original context, but this is a standard practice.*
* **Integration Testing:**
  + **Client-Server Integration:** Testing the flow of login, registration, and page navigation to ensure the Express.js server and HBS views integrate correctly with the browser.
  + **WebSocket Signaling Integration:** Testing the entire signaling flow: client connecting to WebSocket server, creating/joining rooms, sending and receiving approval requests, and successfully exchanging WebRTC offer/answer/ice-candidate messages.
  + **P2P Data Channel Integration:** The most critical integration test. Two separate browser clients (representing two peers) are used to confirm that once a room is joined and approved, chat messages sent from one client are received directly by the other, bypassing the server, and vice-versa. This verifies the successful establishment and functionality of the WebRTC DataChannel.
* **Manual Testing:** Extensive manual testing is performed by simulating various user scenarios (e.g., host creating room, guest joining, host denying, multiple guests trying to join, one peer leaving, etc.) to ensure robust behavior.

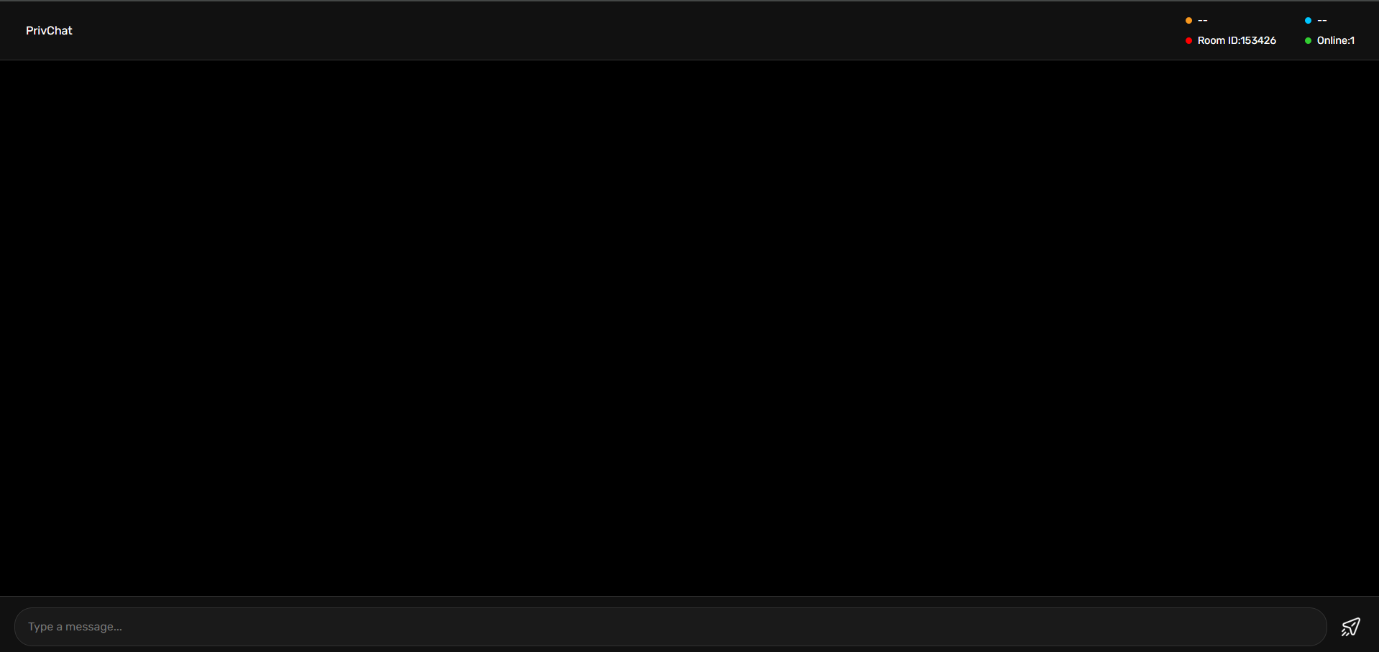
**12. Output (Screenshots)**

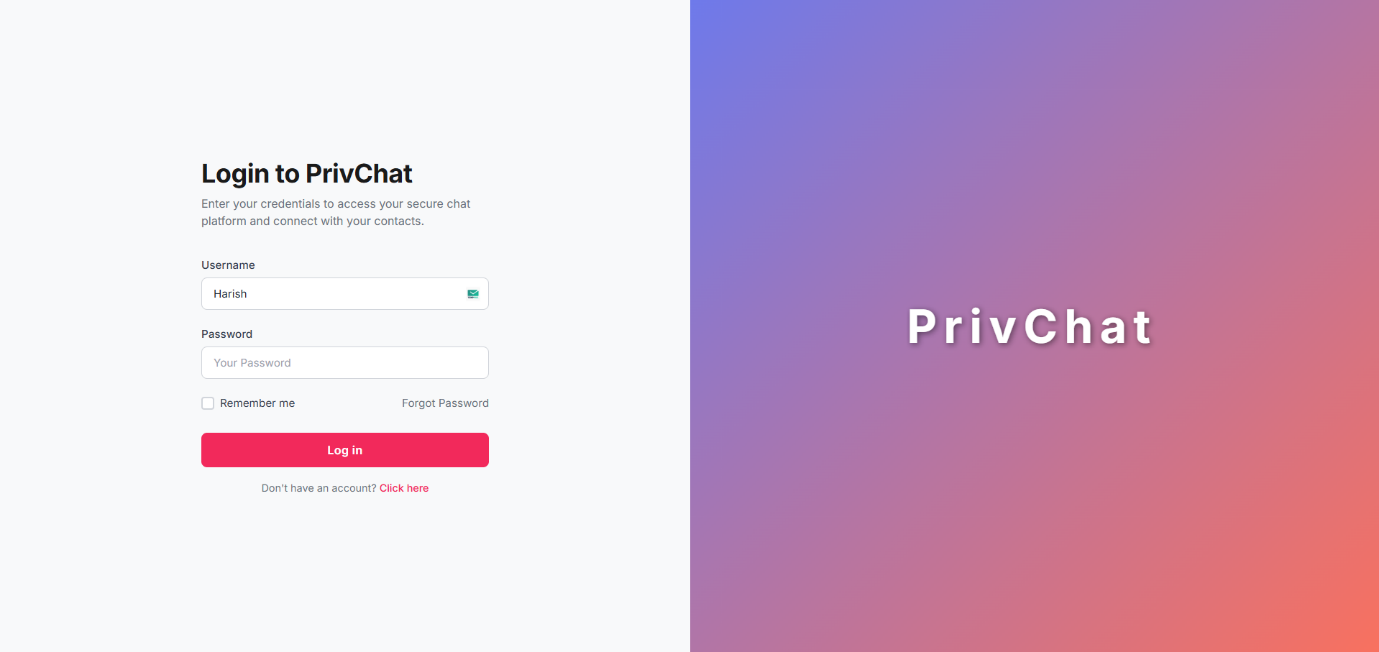
* **Login Page:** Screenshot of the user login interface.
* **Registration Page:** Screenshot of the user registration form.
* **Dashboard Page:** Screenshot of the post-login dashboard, potentially showing user details or room options.
* **Chat Page (Empty Room):** Screenshot of the chat interface when no peers are connected, showing room PIN and user names.
* **Chat Page (Connected Peers):** Screenshot of the chat interface with two peers connected, demonstrating messages being sent and received, and sender names displayed below messages.
* **Approval Request Notification:** Screenshot showing how a host receives a notification for a join request.











**13. Conclusion & References**

**13.1. Conclusion**

PrivChat successfully fulfills its primary objective of creating a secure, private chat application founded on the principle of direct machine-to-machine communication. By strictly limiting the server's role to authentication and connection signaling, and by routing all chat data directly between peers (leveraging WebRTC's inherent encryption for data channels), the application significantly enhances user privacy and mitigates the risks associated with centralized data storage and forwarding. The integration of bcrypt for password security further solidifies its commitment to user data protection. PrivChat stands as a testament to the feasibility and advantages of a P2P communication model in an increasingly privacy-conscious digital landscape, offering a robust and trustworthy platform for confidential exchanges.

**13.2. References**

(This section should list all external sources, documentation, and academic papers that were consulted or referenced during the project's development. Here are categories of references you might include:)

* **Official Documentation:**
  + Node.js Documentation: <https://nodejs.org/en/docs/>
  + Express.js Documentation: <https://expressjs.com/>
  + ws (WebSocket) Library Documentation: <https://github.com/websockets/ws>
  + bcrypt.js Documentation: <https://www.npmjs.com/package/bcrypt>
  + express-session Documentation: <https://www.npmjs.com/package/express-session>
  + MySQL Official Documentation: <https://dev.mysql.com/doc/>
  + WebRTC API Documentation (e.g., MDN Web Docs): <https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API>
* **Standards and Specifications:**
  + RFC 6455 (The WebSocket Protocol)
  + W3C WebRTC Specifications
* **Books/Articles (if applicable):**
  + Any books on Node.js, Express.js, database design, or network security that you used.
  + Academic papers or articles on peer-to-peer networking, WebRTC, or cryptographic best practices.
* **Tutorials/Blogs (if applicable):**
  + Any significant online tutorials or blog posts that provided critical guidance.
* **Full Project Guide:**
  + Full resource available on github account reach it out, for full step by step guide for installation and working
  + https://github.com/Harish2B3/PrivChat

THE END