QuickChat

Amit Kumar Meena **220126** **amitkmeena22@iitk.ac.in**
Taneshq Zendey **221123** **taneshq22@iitk.ac.in**
Aman Yadav **220121** **amany22@iitk.ac.in**
Vaibhav Chirania **211134** **vaibhavc21@iitk.ac.in**
Rahul Singh Anuragi **210808** **rahulsa21@iitk.ac.in**

April 24, 2025
Project id: 13

Abstract

QuickChat is a real-time messaging application designed to facilitate seamless and secure communication among users through chat rooms. This project focuses on designing a robust and scalable database schema to support core features such as user management, chat room creation, message storage, and membership management. The backend is implemented using Python Flask with SQLite, providing a foundation for future scalability and migration to more advanced database systems.

1 Motivation and Problem Statement

The rise of digital collaboration has made group messaging platforms—such as Slack, Discord, and Microsoft Teams—essential tools for communities, educational institutions, and organizations. Unlike private one-on-one messaging, chatroom-based systems facilitate open, topic-driven discussions among multiple participants, supporting knowledge sharing, teamwork, and social interaction at scale.

Designing a robust backend for chatroom-centric applications presents unique challenges:

- **Efficient Group Communication:** Chatrooms must support concurrent participation by many users, with each message instantly visible to all room members. The underlying database must efficiently manage message streams, membership lists, and room metadata.
- **Data Integrity and Consistency:** As users join or leave rooms and exchange messages, the system must maintain accurate records of memberships and message histories. Foreign key constraints and transactional operations are critical to prevent orphaned messages or inconsistent room states.
- **Security and Access Control:** While group conversations are more public than private chats, they still require secure authentication, hashed password storage, and role-based access to prevent unauthorized participation or data breaches.
- **Scalability and Performance:** Popular chatrooms can generate high message volumes and frequent membership changes. The database schema must be optimized for fast message retrieval, efficient room membership queries, and seamless scaling as the number of users and rooms grows.

The primary motivation behind the QuickChat project is to design and implement a database-backed real-time messaging application that addresses these challenges.

2 Methodology

2.1 Requirement Analysis & Planning

- Identified core features: user login, chat rooms, real-time messaging.
- Selected tech stack:
  - **Frontend:** HTML & CSS
  - **Backend:** Flask (python)
  - **Database:** MySQL (sqlite)
  - **Real-time Communication:** Socket.io

2.2 Database Design & Schemas

Designed four main tables with the following schemas:

- **Users** (`user_id`, `full_name`, `hashed_password`, `contact_number`, `role`)
- **Rooms** (`room_id`, `room_name`, `created_on`, `created_by_user`)
- **RoomMembers** (`room_id`, `user_id`) – handles many-to-many relationship
- **Chats** (`message_id`, `message_text`, `sent_at, sender_id`, `room_id`)

2.3 Backend Development

- We used Flask, a lightweight Python web framework, to build the backend APIs.
- We used dynamic **JavaScript** to handle user interactions and UI changees without reloading the page.
- Developed REST APIs for user authentication, room creation, and message exchange.
- Implemented password hashing and token-based authentication.
- Integrated **Socket.io** for real-time message broadcasting.

2.4 Frontend Interface

- Make use of **HTML** for designing webpage and **CSS** for styling purpose.
- Designed responsive UI to list chat rooms, participants, and message threads.
- Real-time chat display with auto-scroll and message timestamps.

2.5 Testing and Debugging

- Unit-tested APIs and real-time event handlers.
- Simulated multiple users to test concurrency and message delivery.
- Ensured data integrity via database constraints.

3 Implementation and Results

3.1 Backend and Database Integration

The backend was implemented using the Flask framework in Python. A SQLite database was used during development to store user data, room information, chat history, and membership mappings in auto-created `chat.db` file. The following features were implemented:

- **User Authentication:** Implemented secure password handling using `generate_password_hash`, which applies SHA-256 with salt and key stretching
- **Room Creation and Membership:** Authenticated users can create new chat rooms or join existing ones. The `RoomMembers` table maintains the many-to-many relationship between users and rooms.
- **Message Exchange:** Users can send messages in any room they are a member of. Messages are stored in the `Chats` table with timestamps and sender-room relationships for efficient querying.
- **Validation Middleware:** The application uses Flask request hooks and decorators to handle input validation and enforce access control. These middleware components ensure that only authenticated users can access certain routes and that inputs are validated before being processed..

3.2 Real-Time Messaging using Socket.IO

To enable live, multi-user chat functionality:

- **Socket.IO Integration:** Real-time communication was implemented using the Flask-SocketIO extension. Events like `message`, `join`, and `leave` were broadcast to all relevant clients.
- **Dynamic Room Events:** Users receive live notifications when someone joins or leaves a room. Messages sent in a room are instantly visible to all active members using Socket.IO's event broadcasting.
- **Concurrency Handling:** The application handles simultaneous clients without data loss or message delays using event-driven, asynchronous communication via WebSockets.

Link of the code is: `https://github.com/akmeena6/CS315/tree/master`

3.3 Frontend Integration

- A responsive web interface was created using HTML and CSS to list all chat rooms, current participants, and message threads.
- JavaScript was used on the client side to establish and maintain Socket.IO connections for real-time interaction.
- Messages are displayed chronologically with timestamps and are auto-scrolled to the most recent entry, improving chat usability.

3.4 Testing Results

- Users could register, log in, create or join chat rooms, and participate in real-time discussions.
- Concurrent access was simulated using multiple browser sessions and test clients to verify real-time updates and data integrity.
- Messages appeared instantly in all active clients within a room, providing a seamless experience.

The project demonstrated a complete real-time messaging pipeline — from user authentication and database storage to live Socket.IO communication — within a lightweight, scalable architecture.

Link to code encoded in `URL`.

4 Discussion and Limitations

4.1 Discussion

QuickChat is a real-time messaging app built with Python Flask, SQLite, and Socket.io. The database structure, consisting of Users, Rooms, RoomMembers, and Chats tables, ensures data consistency with foreign key constraints and performance improvements through indexing key fields like `user_id` and `room_id`. Socket.io powers real-time messaging, while secure user logins, room creation, and messaging are handled through REST APIs with password hashing. The frontend, built using basic use of HTML,CSS, is intuitive with auto-scrolling and live updates.

The app is designed for scalability, with potential for migration to MySQL or PostgreSQL. Testing, including unit tests and user simulations, confirmed the app's reliability under normal usage, providing a smooth group chat experience.

4.2 Limitations

However, there are a few limitations to consider:

1. **Room to Grow**: SQLite is suitable for small-scale apps, but for larger user bases, we would need to switch to MySQL or PostgreSQL.
2. **Handling Large Crowds**: While Socket.io works well for real-time messaging, performance under heavy load (e.g., thousands of users) remains untested.
3. **Security**: While password hashing and token-based login are implemented, additional security measures such as end-to-end encryption and protection against SQL injection or XSS are missing.
4. **Missing Features**: QuickChat lacks advanced features like message search, file sharing, and threaded replies, which could enhance usability.
5. **Testing Gaps**: Further testing is needed to handle edge cases like network failures or database crashes.
6. **Deployment**: With replacement of SQLite with PostgreSQL or MySQL, we can deploy our website on large scale.

5 Contributions

Team Member	Contribution
Amit Kumar Meena	Backend implementation using Flask, Real-time messaging system with Socket.io, Database Design, User Authentication, Frontend Design and Documentation.
Taneshq Zendey	Contributed in Backend implementation, frontend integration, Database Design and Documentation.
Aman Yadav	Database Design, Frontend design and Documentation.
Vaibhav Chirania	Database Design, Documentation and Frontend Design.
Rahul Singh Anuragi	Documentation, Manual testing and Database Design.