Secure WebSocket Chat Application

Author: Curtis Quan-Tran **Date**: 5/11/2025 Github Link

---

1. Introduction

CurtisConnect is a cloud-hosted, end-to-end encrypted chat platform. All traffic occurs over secure websockets (WSS). The client rejects any plaintext ws:// traffic. Messages and file blobs are encrypted on the client with AES-256-GCM, while room keys are exchanged with RSA-4096 / OAEP. Encryption happens before data leaves the browser or the server, and cloud-storage layers can never decrypt user content.

Technology Stack

Layer	Components
Frontend	JavaScript, Markdown rendering (marked.js), Emoji Mart Picker
Backend	Python, Flask / Flask-SocketIO (eventlet), Flask-Sessions
Transport Security	Render manages Let's Encrypt TLS Certs to provide automatic HTTPS + WSS
Cryptography	AES-256-GCM (Content), RSA-4096 + OAEP (key exchange)
Authentication	Username/password stored with bcrypt + per-user salt
Database	PostgreSQL (Free tier on Render) for users, room metadata & encrypted message history
File Storage	Firebase Storage server uploads AES-GCM blobs and hands clients time-limited signed URLs
Infrastructure	Render web service + Postgres; UptimeRobot health checks every 10 minutes
Security Headers	Content Security Policy, HTTP Strict Transport Security, Referrer-Policy, Permissions-Policy.

(The earlier Raspberry Pi + Nginx deployment from Phase 2 was retired; Render now delivers HTTPS out-of-the-box and eliminates my on-premises hardware management.)

---

2. Changes from Previous Version

Area	Phase 2 (Old)	Phase 3 (New)
Hosting/Domain	Local Raspberry Pi + Nginx at curtisqt.com	Deployed on Render (websocket-project-5mug.onrender.com); on-premise hosting retired
TLS Certificates	Let's Encrypt was manually issued for curtisqt.com	Let's Encrypt cert is auto-issued by Render
Cloud Proxy	Cloudflare in front of curtis.com (bot-protection, geo-rules)	Cloudflare removed; Render serves HTTPS directly (loses Cloudflare extras)
File storage path	Files are stored locally on the server disk	Files are stored in Firebase storage, served back with signed URLs
Logging	Verbose SSL errors noise in Flask Logs	Suppressed non-actionable SSL/404 noise
Registration Security	No bot protection	Google reCAPTCHA (v2 checkbox)
User Interface	Introduced a dashboard interface that shows all the rooms	Recolored and visually improved the dashboard.

---

3. User Guide

https://websocket-project-5mug.onrender.com

Register/Login

- New users: Click Register, solve reCAPTCHA, choose username + password.
- Existing users: Enter credentials and click Login.

Rooms

- Sidebar → Create Room (auto 4-character ID) or Join room (enter ID).
- The Leave Room button returns to the lobby.

Chat

- End-to-end encrypted messages (150 character max, 1 msg/sec rate-limit).
- Markdown formatting and Emoji-Mart Picker.

Logout

- TXT, PDF, PNG, JPG, JPEG, GIF support.
- 8 mb file size limit.
- Encrypted with AES-256-GCM; Firebase returns 24 h signed link.

Logout

- Click Log Out or simply close the browser tab.

---

4. Key Security Features

- **AES-256-GCM:** For all content.
- **RSA-4096 with OAEP**: For key exchange.
- **Bcrypt** + salt credential storage in Render Postgres.
- **Brute-Force protection**: 3 failed logins → IP blocked for 5 minutes.
- **Rate-Limit**: 1 msg/sec per user.
- **Google reCAPTCHA** on registration prevents bot sign-ups.
- **Secure Headers**: CSP, HSTS, Referrer-Policy, Permissions-Policy
- **E2E File Encryption**: server & Firebase never see plaintext of files.
- **SSL Certificates**: Render provided Let's Encrypt certificates.

---

5. Hosting and Infrastructure

- **Render** web service (256 MB RAM, 0.1 CPU) + Free Postgres (1 GB).

- **Firebase Storage** for encrypted fileblobs; 24 h signed URLs.
- **UptimeRobot** monitors */ping* every 10 minutes and emails on downtime.
- Render auto-renews Let's Encrypt certs → A+ Rating – SSL Labs
- All security headers set in app.py → A Rating – Security Headers

---

6. Key Features Summary

- Clean dashboard user interface with room list and emoji picker.
- Real-time encrypted chat and file transfer.
- Server-side rate-limiting and brute-force defense.
- Google reCAPTCHA for bot mitigation.

---

7. Future Improvements

- User input validation and XSS sanitization.
- Better activity monitoring, Renders events shell is minimal.
- Scalability with chat room limits.
- View chat history, allowing users to see previously sent messages before they joined the room (WIP).
- Allow users to configure rooms on creation, such as:
    - Restricting the size of the user's possible in the room.
    - Require password entry to enter the room.
- Improvements to the front-end interface:
    - Global Roster List to identify who is online or not (WIP).
    - Local Roster List to identify who is in the same room (WIP).
    - More seamless room changing rather than inputting 4 digits.

---

8. AI Assistance & Contributions

Area	AI Usage
Frontend	• AI helped create the dashboard.html layout for phase 2.

	• Helped me learn CSS, HTML, and JS throughout this project. • Tried to help me link the roster list, but unfortunately, could not get it working.
Backend	• AI was used primarily to identify issues with my code, particularly with key management and session management. • Advised on how to link Render, Firebase Cloud Storage, and Render Postgres, and issues with the integration.
Security	• AI provided suggestions to improve the quality of security mechanisms, such as suggesting to use of OAEP, for RSA encryption • Advise on how to harden my Raspberry Pi 5 effectively to prevent risk to my home network during phase 2. • Identify which security headers to use and how to resolve violations in the existing codebase. • Aided in suppressing excessive 404 flask logs in phase 2.
Documentation	• AI was used to help restructure this document and suggest wording improvements.

---

9. Project Reflections and Lessons Learned

Challenges Encountered

- **Chat History Persistence**
  - Attempts to implement chat history persistence across room switches proved unstable due to WebSocket event handling limitation and current architecture.
- **Active Roster Sync**
  - Synchronizing real-time user roasts occasionally caused duplication or stale states. State management improvements are required to make this process more stable.

- **Testing Workflows**
    - Relied on cloud-hosting Render deployments for every change, iteration speed was significantly slowed by deployment wait times (~2-3 minutes per test). In future projects, implementing a local testing workflow with environment toggling and WebSocket reconfigure should allow faster development cycles and cleaner commit history.
- **Time Management**
    - Better time management could have improved the quality, stability, and scalability of the final deliverable.

Conclusion

This project meets the functional and security requirements outlined for the phase. The process of building this project from Phase 1 to Phase 3 has revealed several areas for improvement, particularly around UX, state management, and development workflows. Given more time, a more polished user experience could have been fully realized. That said, I am satisfied with the progress made. This experience provided valuable lessons for future projects.