🔐 End-to-End Encrypted Chat — Architecture & Flow (Page 1)

📘 Overview

This document describes a minimal demonstration of true end-to-end encryption (E2EE) for a chat application built with Flask-SocketIO (server) and browser-based clients. The server acts solely as a relay and does not learn message contents or session keys.

🚦 High-level Message Flow

When Client A sends a message to Client B, the following steps occur:

1. **AES Session Key Generation (Client A)**: Client A generates a random 256-bit AES key (unique per message).
2. **Message Encryption (Client A)**: The plaintext is encrypted with the AES key producing ciphertext.
3. **AES Key Encryption (Client A)**: Client A encrypts the AES key using Client B's RSA public key.
4. **Transmission to Server (Client A)**: Client A sends two opaque blobs to the server: the RSA-encrypted AES key and the AES-encrypted message ciphertext.
5. **Message Relay (Server)**: The server relays both blobs to Client B without attempting decryption.
6. **AES Key Decryption (Client B)**: Client B decrypts the RSA-encrypted AES key with its RSA private key.
7. **Message Decryption (Client B)**: Client B decrypts the AES ciphertext with the recovered AES key and obtains the plaintext.

🔒 Security Guarantees

This design ensures the server never has access to plaintext or symmetric keys, providing true end-to-end confidentiality assuming clients protect their private keys.

⚠️ Assumptions & Limitations

• Clients correctly generate and protect RSA private keys.
• Public keys are exchanged via the server at registration but must be verified (out-of-band or fingerprint checks) to prevent active MITM attacks.
• This demo uses RSA for key-wrapping; production systems should consider authenticated encryption, MACs, and forward secrecy (e.g., ephemeral Diffie–Hellman) to strengthen security.

🛠️ Implementation Details & Usage (Page 2)

💻 Technology Stack

Backend: Python 3 + Flask + Flask-SocketIO.
Frontend: HTML5 + JavaScript (client-side RSA & AES).
Cryptography: Python cryptography and browser libraries such as JSEncrypt (RSA) and CryptoJS or Web Crypto API (AES).

📁 File Layout (deliverables)

- server.py — Flask server + SocketIO event handlers (register, get_public_keys, send_message, relay).
- static/js/chat.js — Client-side key gen, encryption/decryption, UI logic.
- templates/index.html — Minimal page to register username, display users, send encrypted messages.
- utils/crypto.py — Optional server-side helper (conceptual).

⚙️ Getting Started (quick)

1️⃣ Install Python dependencies:

```
pip install Flask Flask-SocketIO python-engineio python-socketio cryptography
```

2️⃣ Place files in project structure and run:

```
python server.py
```

3️⃣ Open *http://127.0.0.1:5000* in multiple browser tabs, register unique usernames, and send encrypted messages.

📝 Minimal Server Pseudocode

```
from flask import Flask, render_template
from flask_socketio import SocketIO

app = Flask(__name__)
socketio = SocketIO(app)
users = {}  # username -> public_key

@socketio.on('register')
def handle_register(data):
    users[data['username']] = data['public_key']
    socketio.emit('public_keys', [{'username': u, 'public_key': users[u]} for u in users])

@socketio.on('send_message')
def handle_send(msg):
    socketio.emit('relay_message', msg, to=msg['to'])

if __name__ == '__main__':
    socketio.run(app, host='127.0.0.1', port=5000)
```

💡 Client-side Notes

• Generate an RSA key pair in the browser (e.g., JSEncrypt) and register the public key with the server.
• For each message generate a random 256-bit AES key, encrypt the plaintext with AES, then encrypt the AES key with the recipient's RSA public key.
• Send both encrypted components to the server which relays to the recipient.
• Recipient decrypts AES key with RSA private key and then decrypts AES ciphertext.

🚀 Recommended Enhancements

• Add message authentication (HMAC or AES-GCM) for integrity and authenticity.
• Add forward secrecy with ephemeral DH (X25519).
• Persist public keys and user registrations in a database and present key fingerprints to users for manual verification.
• For group chats adopt a group key management protocol (e.g., double-ratchet or MLS).

📝 Consultation / Demonstration & Future Enhancements

This demo effectively showcases the core principles of E2EE. Users register, exchange public keys, and send messages that are encrypted client-side and decrypted only by the intended recipient. The server remains oblivious to message content.

Potential Enhancements:

• Persistent User Data: Implement a database (SQLite, PostgreSQL) for storing registrations and public keys.
• Message Authentication Codes (MACs): Ensure message integrity and authenticity.
• Forward Secrecy: Use Diffie-Hellman for ephemeral keys.
• Group Chat: Extend E2EE for group messaging.
• Offline Messaging: Store encrypted messages for offline recipients.
• Improved UI/UX: Add message history, typing indicators, and better styling.
• Auditable Security: Log cryptographic operations for audits.

📝 Submitted By

✅ **Submitted By:**	FATEHALI ABBASALI MAKNOJIYA
🎓 **Internship Project Duration:**	2 Weeks
🗓️ **Year:**	2025
👨‍🏫 **Mentor:**	ELEVATE LABS