TITLE: Let's Chat

GROUP MEMBERS:

- RAMSAI UMMADISETTY (Student ID: 20035571)
- SUBRAMANYAM SILIVERI (Student ID: 20034634)
- KOWSHIK YALAVARTHI (Student ID: 20036454)

COURSE NAME: B9IS103 COMPUTER SYSTEMS SECURITY (B9IS103_2425_TMD2)

LECTURER: PAUL LAIRD

DEPARTMENT: INFORMATION SYSTEMS WITH COMPUTING

Table Of Content

1. Introduction 3
2. Functional Requirements 3
3. Non-Functional Requirements 4
4. Security Requirements 5
5. Tech Stack 5
6. User Flow 6
7. User Flow Diagrams 9
8. Limitations of the System 14
9. Challenges Faced 15
10. Repository & Deployment Links 16
11. Conclusion 16
12. References 16

1. Introduction

Let's Chat System is a web-based communication platform that offers secure one-to-one messaging to the users. The system, based on strong end-to-end cryptography, offers confidentiality, authenticity, and message integrity. Messages are encrypted with receiver public key and signed with sender private key from sender end. Once the receiver receives the message first the system will verify the signature with sender public, if signature is verified message will be encrypted and displayed to user. User authentication and real-time messaging are also part of the system.

The main purpose is to secure the chats from anyone who may be tampering with the messages or pretending to be someone else.

2. Functional Requirements

1. **User Signup / Sign In:** User will sign up through a secure signUp/signIn page using Google account. After successful signUp, an RSA key pair with a unique identifier and the user is generated.
2. **RSA Key Generation:** 2048-bit RSA key pair per user automatically created at signup is generated by using Web Crypto API
3. **Key Storage:**
    a. **Public Key**: The public key is saved in the backend database and posted in GitHub Gist for verification.
    b. **Private Key:** After successful signup the private key is automatically downloaded to user and password-encrypted using the user-input password and stored securely in the client browser with IndexedDB.
4. **Login and Recovery of Keys**
    a. Upon user signIn, the private key password-encrypted with the user password is retrieved from IndexedDB and decrypted using the user password.

    b. This is performed so that the private key never goes out of the user machine and is still safe even when there is a loss of local storage.
    c. If the private key is lost in IndexedDB because of any issue, System will ask user to upload the private which is downloaded as pem file to user while signup.
5. **User List Display:**

   After the user logs in, he/she can see a list of all other registered users (except himself).
6. **Chat Window:**
    a. Users can message any other user simply by clicking his or her name.
    b. Messages are signed with the recipient's public key and sent signed with the sender's private key.
    c. When received, the signature of the message is verified before decrypting to verify authenticity and integrity.
7. **Real-Time Messaging:** Socket.IO is used for real-time peer-to-peer messaging.
8. **Logout and Auto Logout:**
    a. The users can logout by using logout button.
    b. The system also includes a security feature that automatically logs out the users when there is 2 minutes of inactivity. The system gives a countdown timer before auto logout, giving the users an option to stay logged in.
9. **Notification:** A notification will be displayed to user when a new message is received.

3. Non-Functional Requirements

1. **Security:** Standard cryptography algorithms used in the industry and no security by obscurity.
2. **Performance:** Chat updates are felt in real time with zero latency, under 200ms.
3. **Reliability:** Maintains socket connections intact despite low-level interference through auto-reconnect.
4. **Scalability:** Handling more users with minimal backend configuration.
5. **Usability:** Straightforward and simple UI.

6. **Responsiveness:** Complete functionality on devices like desktops, tablets, and smartphones.

4. Security Requirements

1. **Confidentiality:** RSA-OAEP encrypts the message in such a way that the message can be decrypted only by the intended individual.
2. **Authentication:** Only the login users can access the chat page and RSA-PSS digital signatures allow the recipient to authenticate the sender.
3. **Integrity:** Digital signatures also protect against tampering.
4. **Token Management:** JWT tokens for authentication. Token is rendered unusable after logout by user themselves and after 1 minute if a user has become inactive.
5. **Public Key Authentication:** Both the database and GitHub Gist store each user's public key. When the chat first begins, the two sources are checked against one another to prevent tampering.
6. **No Hidden Logic**: All cryptographic methods are open and transparent.

5. Tech Stack

Frontend:

1. React.js: Renders dynamic and component-based UI.
2. React Query: Manages API state and caching efficiently.
3. React-Bootstrap: Provides tidy UI components.
4. Socket.IO-client: Provides real-time communication.
5. IndexedDB: Syncs locally encrypted private keys on the client.

Backend:

1. Node.js + Express: Routes server and API logic.
2. JWT: Secure token for authentication.

3. Socket.IO: Two-way live messaging.

Database:

1. MySQL: To stores data in a structured format.

Cryptography:

1. Web Crypto API:
    a. RSA-OAEP: Decrypt/encrypt messages.
    b. RSA-PSS: Digital signature.
2. CryptoJS: Local environment private key encryption.

External Tools:

1. GitHub Gist: Authentication public keys.
2. Render.com: Frontend and backend application.

Why These Tools:

1. Web Crypto API is native and secure in contemporary browsers.
2. React enable fast paced UI development.
3. Socket.IO delivers frictionless live messaging.
4. GitHub Gist provides publicly verifiable public key without hosting custom infrastructure.

6. User Flow

1. **Signup**
    - User comes onto signup page.
    - Uses Google to Sign up.
    - After successful signup the Web Crypto API generates a 2048-bit RSA key pair.
    - Public key is sent to backend to store in Database and GitHub Gist.
    - Private key is downloaded to user system and encrypted with password and stored in IndexedDB.

Errors Cases:

- If key generation or GitHub upload fails, show proper error.

2. **Sign in**
    - User logs in via Google.
    - JWT token is generated from backend and stored on frontend.
    - Encrypted private key is fetched from IndexedDB.
    - User enters password to decrypt and use the private key.

    Errors Cases:

    - Incorrect password leads to decryption failure.
    - If private key does not present in IndexedDB, prompts the user to upload manually.

3. **Send Message**
    - User selects a recipient from the user list.
    - Receiver's public key is fetched from DB and Gist. Verify both are same or not, if same chat window opens otherwise, it will show the error.
    - Message is encrypted with the receiver's public key for receiver (encrypted_for_sender) and encrypted with sender's public key(encrypted_for_sender) to show on message on sender screen.
    - Message is signed with the sender's private key.
    - Socket.IO is used to send the encrypted message and signature to the server.

    Errors Cases:

    - If public key mismatch, block messaging.
    - If signing or encryption fails, message is not sent.

4. **Receive Message**
    - Received new message through socket listener.
    - Sender's public key is fetched and used to verify signature.

- If verification is successful, message is decrypted using the receiver's private key. If not verified it will show an error.
- Message is then displayed in chat.

Errors Cases:

- If signature verification fails, show “Message verification failed!” and don't decrypt.
- If decryption fails, show error “Unable to decrypt message”

5. **Private Key Missing (Recovery Flow)**
   - User sign in but there is no private key in IndexedDB.
   - App prompts the user to upload PEM file.
   - User will upload the file and then system will ask user to enter the email and a password to encrypt and store in indexedDB
   - Private key is encrypted with the password and saved back to IndexedDB.
   - Normal operation resumes.

   Errors Cases:
   - If file upload is aborted by user, access is denied.

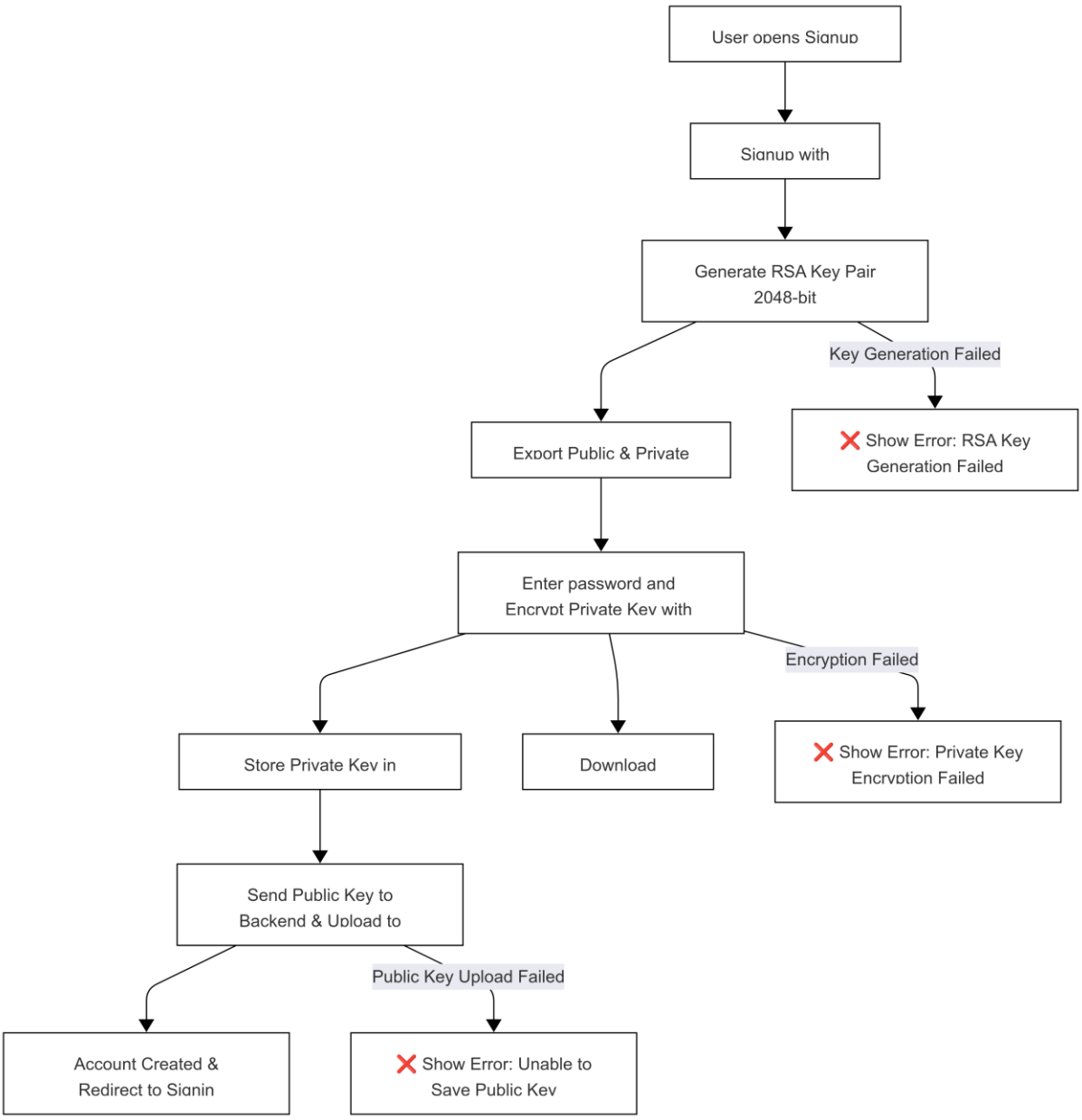
6. Logout & Inactivity Flow

- User can logout by himself/herself via Logout button.
  - Removes JWT token and sensitive data from localStorage and call Api to blacklist the JWT token so that can't make requests with that token. After that redirects to sign in page
- Inactivity Timeout:
  - If user stays idle for 1 minute, an overlay with countdown is displayed.
  - As soon as countdown reaches 0, user is logged out automatically.
  - System removes data from localStorage, call api and redirects to sign in page.

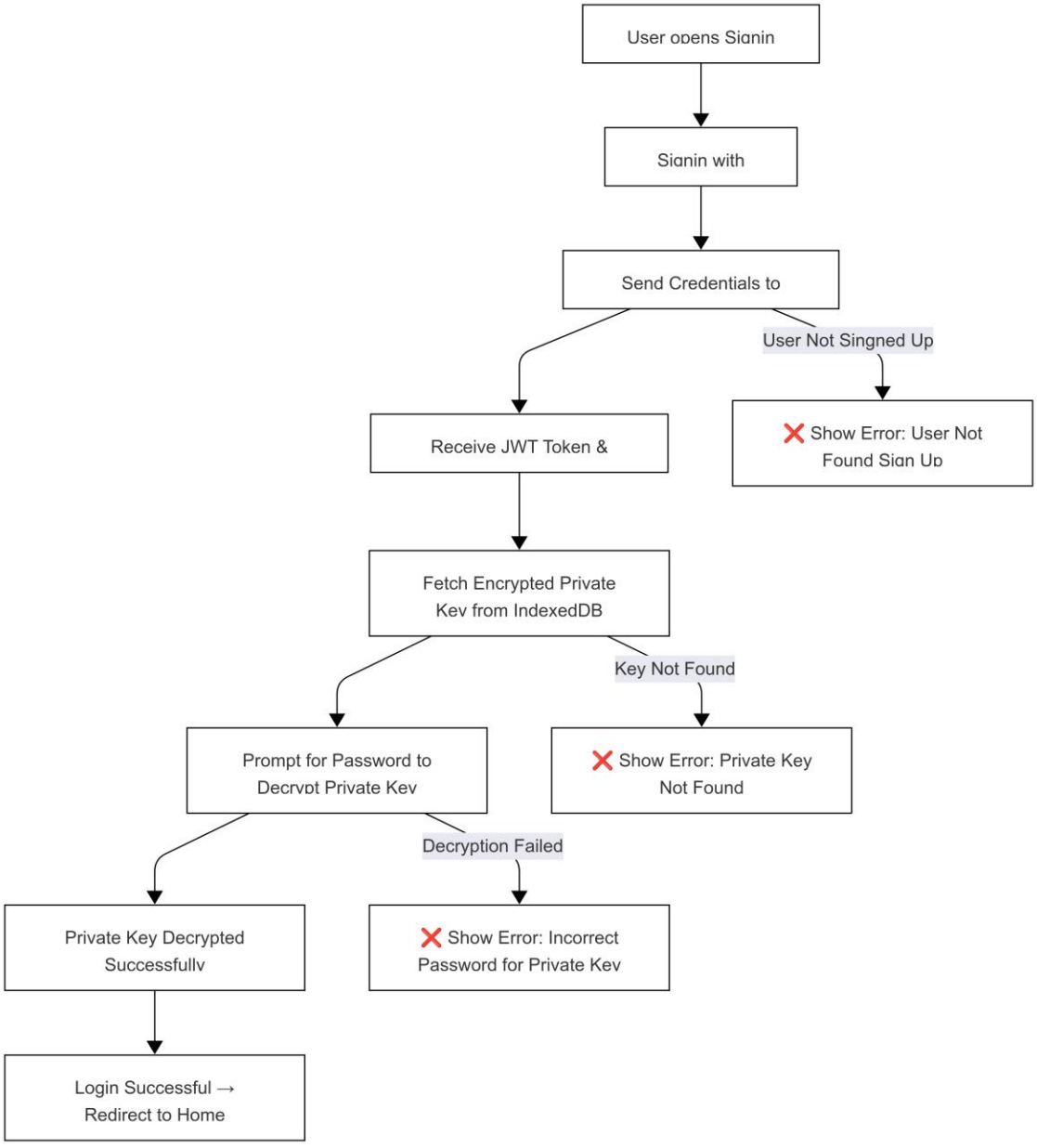
Error Cases:

- If logout fails (network/server issue), fallback clears localstorage on frontend.

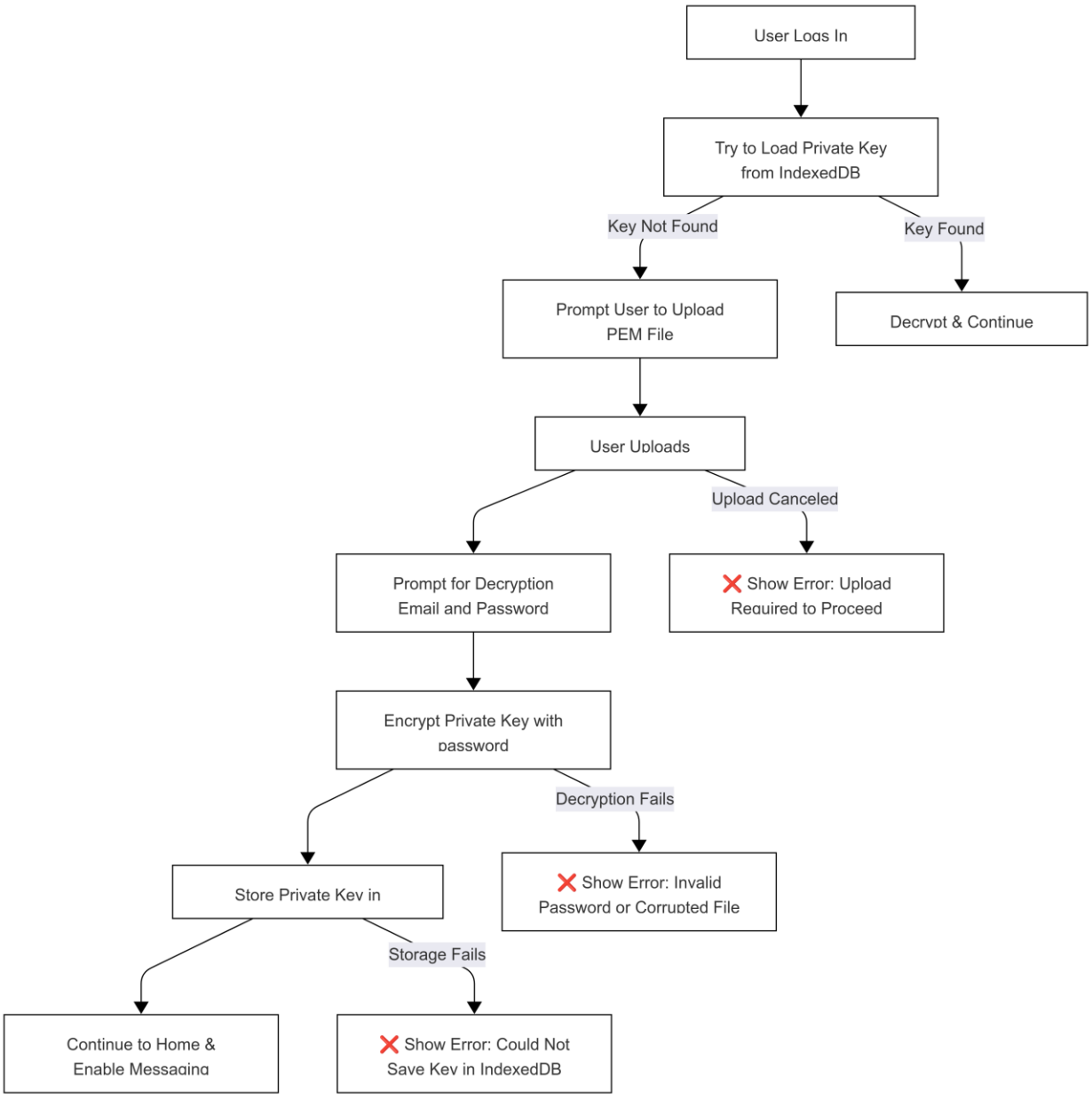
7. User Flow Diagrams



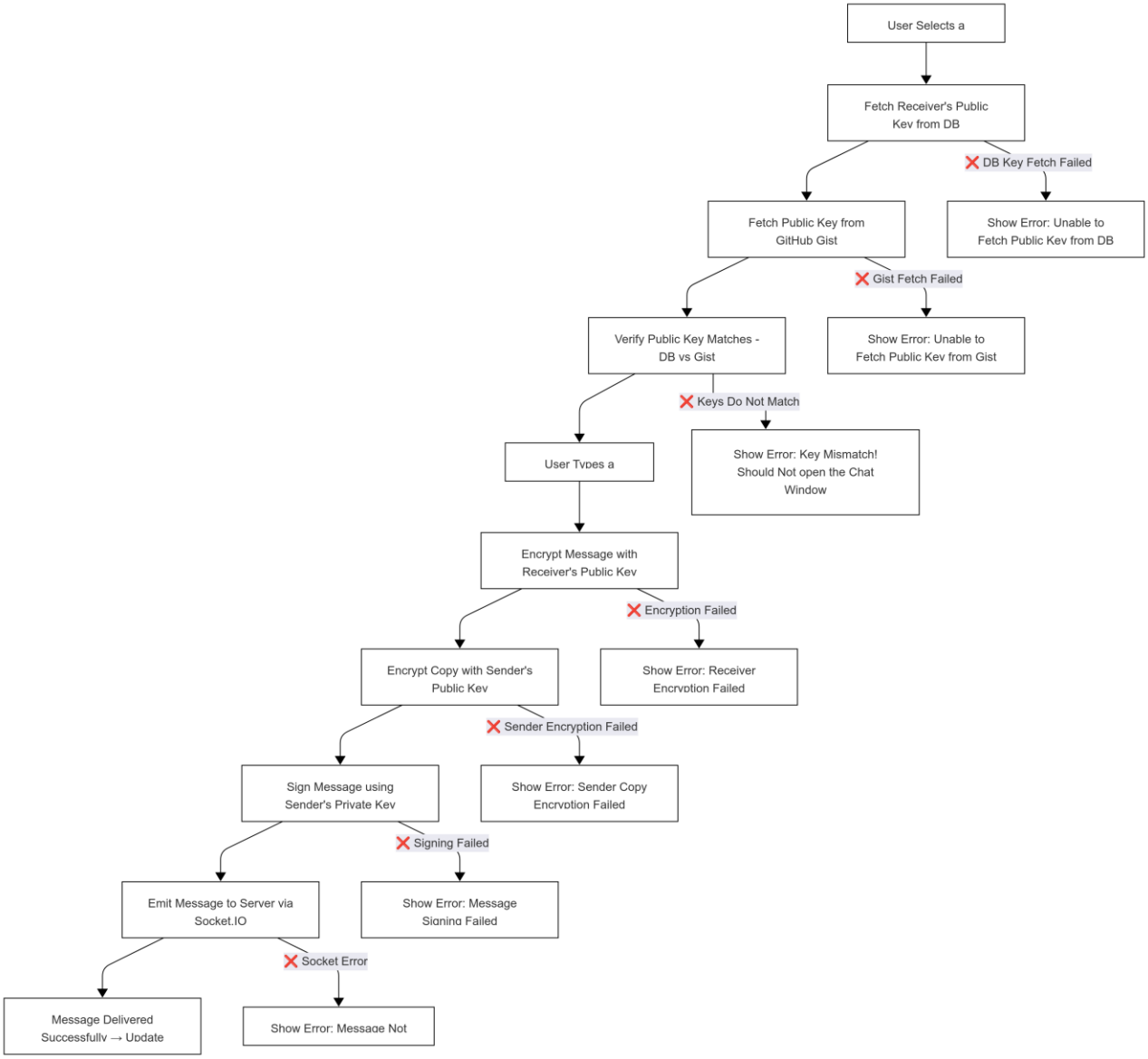
7.1 Sign Up


User opens Signin
Signin with
Send Credentials to
User Not Singned Up
Receive JWT Token &
❌ Show Error: User Not Found Sign Up
Fetch Encrypted Private Key from IndexedDB
Key Not Found
Prompt for Password to Decrypt Private Key
❌ Show Error: Private Key Not Found
Decryption Failed
Private Key Decrypted Successfully
❌ Show Error: Incorrect Password for Private Key
Login Successful → Redirect to Home

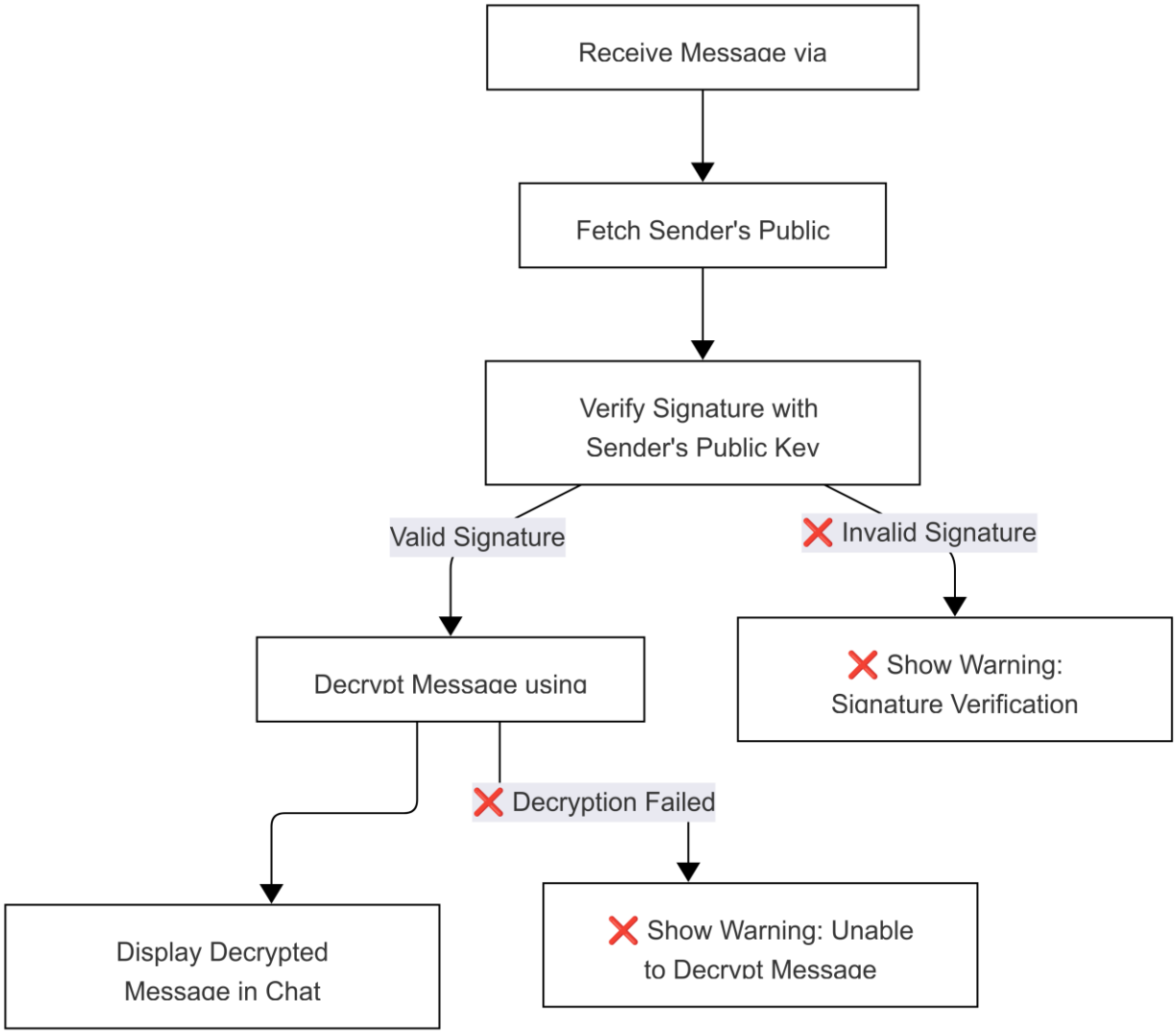
7.2 Sign In



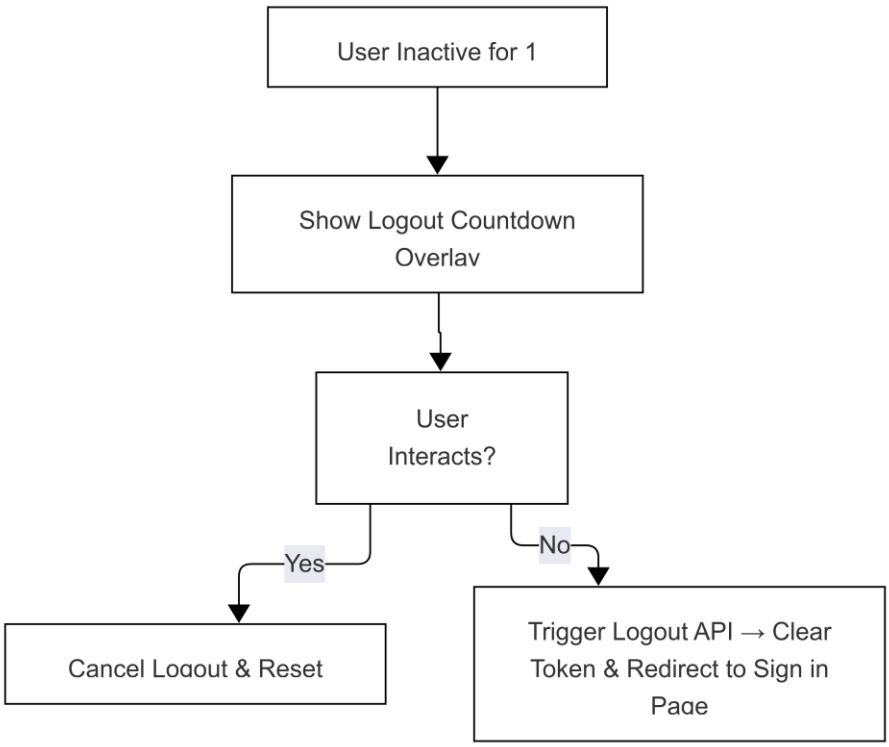
7.3 Upload Private Key



7.4 Send Message



7.5 Receive Message



7.6 Logout and Auto Logout

8. Limitations of the System

1. **Manual Upload of Private Keys on New Devices:**

   Since the private key is saved only in the browser's IndexedDB securely, the users will have to import or upload their private key manually in case they login from a different device or browser. It makes it more secure but may affect usability for end-users who are not very technical.

2. **Device-Specific Storage:**

   As encryption keys are saved in the browser's IndexedDB, clearing storage in the browser or incognito mode would potentially lead to the loss of the private key unless it has been downloaded and imported again.

9. Challenges Faced

1. **Single Repo Deployment Issues:**
   - We had one repository with both frontend and backend at first.
   - Render did not support such organization effectively for automatic deployment.
   - We resolved this by separating the frontend and backend into two repositories, making it easy to deploy and have different configurations.
2. **Message Visibility Problem (Single Encryption Logic):**
   - We originally encrypted messages using only the recipient's public key.
   - As a result, senders could not view their sent messages since they could not decrypt them.

   Solved the issue by:

   - Encrypting the message twice: once with the recipient's public key (encryptedForReceiver) and once with the sender's public key (encryptedForSender).
3. **Private Key Management:**
   - Since private keys are stored securely in IndexedDB, users would have to re-upload them on another device.
   - Adding an import and decrypt trusted user key added overhead.
4. **Handling JWT Logout:**
   - JWT tokens are stateless, and it is challenging to revoke them upon logout.
   - Blacklisted tokens were stored with a workaround at logout to terminate the session.
5. **Auto Logout on Inactivity:**
   - Inactivity logout with countdown overlay needed complex event handling and removal of the timer.
   - Fired multiple logout events initially due to multiple listeners—resolved owing to proper event handling.
6. **Crypto Key Compatibility Errors:**
   - Key type mismatch when using Web Crypto API for encryption and signature (RSA-OAEP vs RSA-PSS).
   - We solved this by making a clear distinction between key usage and using proper algorithms.
7. **Real-Time Messaging Sync:**
   - Fixed by sending message locally and triggering backend re-fetch with socket acknowledgement.

10. Repository & Deployment Links

Git Hub Repositories:

- Frontend - https://github.com/SubramanyamGit/LetsChat-FE
- Backend - https://github.com/SubramanyamGit/LetsChat-BE
- Frontend + Backend - https://github.com/SubramanyamGit/LetsChat (This repo is used for development but because render doesn't support it, so we changed Frontend and backend into separate repos.)

Deployed Application URL:

- https://letschat-fe.onrender.com

11. Conclusion

The Let's Chat System integrates well the encryption technology and chat in real time to provide end-to-end secure and interruption-free messaging experience. It provides confidentiality, integrity, and authenticity of the message using RSA-OAEP encryption as well as RSA-PSS digital signature.

The system design respects the privacy of users through the storage of private keys strictly client-side, with public key authentication via GitHub Gist being supported. Auto log-out by inactivity and live messaging delivery via Socket.IO contribute to a responsive and secure user interface.

12. References

- GitHub Gist – https://gist.github.com
- NPM – https://www.npmjs.com
- Web Crypto API – https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API
- Socket.IO – https://socket.io
- Socket.IO (NPM) – https://www.npmjs.com/package/socket.io
- React – https://reactjs.org
- React Query – https://tanstack.com/query/latest
- CryptoJS – https://www.npmjs.com/package/crypto-js

- JSON Web Token (jsonwebtoken) – https://www.npmjs.com/package/jsonwebtoken
- IndexedDB – https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
- Render – https://render.com