Lock-In

Design Document

Team 10:
Vikram Kavalipati, Andrey Georgiev,
Faisal Alsayyari, Yousef Alokaili

Index

Purpose

Many people and businesses possess sensitive information they wish to store with a higher level of security. This includes passwords, files, photos, journals, and more. They need a secure platform to store this information. Our application, Lock-in, aims to provide businesses and people with a free, open-source, end-to-end encrypted digital vault that can provide secure storage.

There are many existing forms of digital cloud content storage specialized for different tasks. Notion for workspaces, Google Workspace for office products, Google Cloud for storage, 750words for journaling, etc. All of these services allow the user to write text, image, and audio files stored on the cloud and accessed from any device, assuming the account is signed in.

However, the problem with all of these services is that they are either not encrypted at all, or not end-to-end encrypted, meaning a malicious server could decrypt your data. This poses a problem as the server can theoretically access your data. What distinguishes Lock-in is a password-derived, end-to-end encrypted storage system, which means that only you and your password can access your files. Our goal is to provide a completely free, open-source, trustless encrypted storage space for text files, notes, passwords, and audio files. Because encryption is password-derived, password loss results in permanent data loss.

Design Outline

Functional Requirements

1. **User account**

   As a user,

a. I would like to be able to register for a Lock-in account.
b. I would like to be able to log in to my Lock-in account.
c. I would like to be able to change my master password, given that I supply my old master password.
d. I would like to add an email to my account to work as two-factor authentication (2FA).
e. I would like to be able to delete my account and all related information.
f. I would like my notes to be synced on multiple devices.
g. I would like to be notified if a new device logs into my account.

2. **Note Creation, Modification, and Deletion**

   As a user,

a. I would like to be able to create notes to be saved.
b. I would like to be able to modify my notes after creating them.

c. I would like to be able to remove notes permanently.
d. I would like to be able to specify if a note is saved to the server, the client, or both
e. I would like to be able to upload images as notes.
f. I would like to be able to upload audio files as notes.
g. I would like to be able to securely transcribe audio files if they contain speech.
h. If I am sent a note with the same name as one of my notes, I would like the option to change the new note name instead of just overwriting my current note.
i. I would like to store video files as notes. (If time permits)

3. **Note Viewing**

As a user,

a. I would like to view all my notes by name.
b. I would like to be able to sort the notes in several ways (i.e. last modified, alphabetical, etc.)
c. I would like to be able to pin important notes to the top of the list.
d. I would like to be able to view image notes and play audio notes. (if time permits)
e. I would like to be able to search and filter notes by date and type. (text, image, or audio)
f. I would like to be able to securely share a copy of a note with another user.

4. **Security Features**

As a user,

a. I would like an option to upload files to my local machine instead of to the cloud.
a. I would like to be able to generate my master password for it to be strong and secure.
b. I would like the option to set a time interval to be reminded to change my master password.
c. I would like to optionally add a second password to a note. (if time permits)
d. I would like to have a keybind that can quickly hide what I am viewing/working on so that my information is secure if someone walks in. (if time permits)
e. I would like to be notified if a new device logs into my account.

5. **Accessibility Features**

As a user,

a. I would like an about page explaining how to use the website.
b. I would like to be able to switch between a light mode and a dark mode so that I don’t strain my eyes in different settings.
c. I would like a settings page where I can adjust preferences such as text size.

6. **Realtime Representation**

As a user,

a. I would like any discrepancies between the server and client to be resolved automatically by using the newest information.

Non-Functional Requirements

1. Performance

As a developer,

a. I would like the application to be able to support >1K users with 1 GB of notes each (on the cloud).

b. I would like the user to be able to access a text note within 5 seconds.

2. Security

As a developer,

c. I would like to use reliable, cryptographically secure algorithms to encrypt notes.

d. I would like all encryption and decryption to happen on the client-side.

e. I would like the user to have the option to choose between server-side cloud storage and client-side storage.

3. Architecture

As a developer,

f. I would like to use a client-server architecture.

g. I would like the user to interact with the app via a web-based GUI.

h. I would like there to be a server to manage user accounts and cloud storage.

i. I would like there to be a client-only storage system.

4. Usability and Appearance

As a developer,

j. I would like the GUI to be aesthetic and minimalistic.

k. I would like to make it clear to the user whether their encrypted file is being stored locally or on the cloud.

l. I would like the user to be able to set and change a shortcut to hide the screen.

High-level Architecture

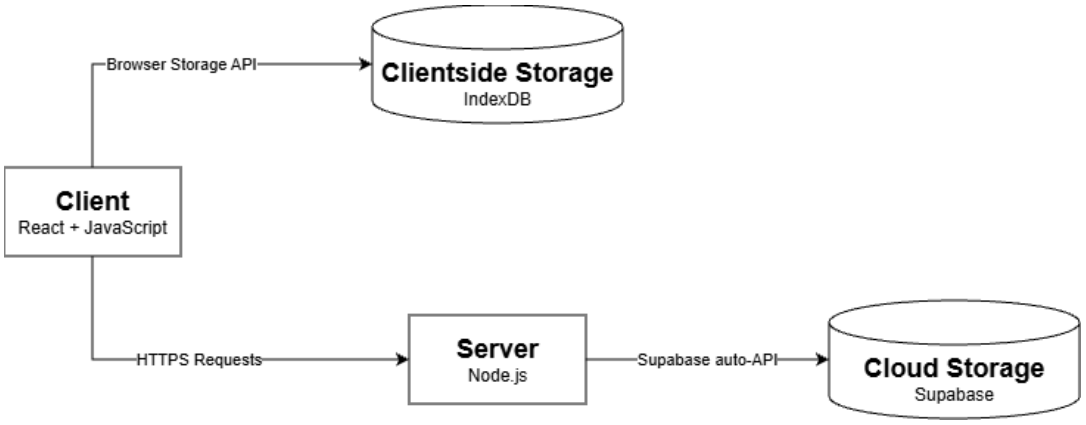

Figure 1: High-Level Architecture.

Lock-in will utilize a client-server model. The client provides a GUI for the user to interact with the app as well as cryptographic functionality. The GUI could either be a login page or a signup page, prompting the user for a username and password; or a homepage, displaying the user's notes, their names, and buttons for opening, uploading, sharing, and deleting notes. The client is the only component that ever has access to plaintext. The user could choose to store files locally or send them to the cloud for remote storage. The client communicates with the server via HTTP requests and only sends public account information or encrypted bytes. The server itself uses Supabase to store account information or encrypted files, and communicates via Supabase's automatically-built RESTful API.

Design Issues

Functional Issues

1. **What should we require from the user to make an account?**

a. Option 1: Username and password
b. Option 2: Email and password
c. Option 3: Username, email, and password

Choice: Username and password

Justification: The password is required across all options, as this is what will derive the key used to encrypt files. A username is also required to uniquely identify each account and to allow users to share files with each other. It was decided not to require an email to simplify the signup process, though the user can optionally add one at any time for 2FA.

2. **How should file sharing be handled?**

a. Option 1: Share via a public, one-time-use link (no account required)

b. Option 2: Share via username (other user must have an account)
c. Option 3: Support both methods

Choice: Share via username

Justification: We will let users share files via their usernames because it aligns with our end-to-end encryption model. At account creation, each account will have a public/private key pair, which will facilitate secure sharing with other users based on username. We chose not to share via a public link because then the link itself becomes a security risk, as the link can be mishandled and accessed by anyone. The option to support both methods has the same weakness, introduces more technical complexity, and unnecessary options for the user.

Non-Functional Issues

3. What web service should we use?

a. Option 1: AWS
b. Option 2: Azure
c. Option 3: Google Cloud

Choice: AWS

Justification: We chose AWS because of its maturity, flexibility, and strong ecosystem. Since it is the most widely adopted cloud provider, there is an abundance of resources and detailed documentation on developing technically complex servers hosted on AWS. Azure was not selected because it requires more integration with Microsoft ecosystems, which isn't something we intend to take advantage of. Google Cloud was also not selected because there is no clear architectural benefit over AWS's better documentation.

4. What frontend framework should we use?

a. Option 1: React
b. Option 2: Angular
c. Option 3: Vue

Choice: React

Justification: React is the most popular framework, so there is an abundance of resources for learning and using React in a production-grade application. Furthermore, React is slightly more minimalistic than the other two options, which will allow us to have greater control over the exact data flow. Angular was not selected because it is slightly heavier as a framework, which is not something we need in a lightweight application. Vue was not selected because of its smaller ecosystem and less adoption compared to React.

5. What backend framework should we use?

a. Option 1: Spring Boot (Java)
b. Option 2: PHP
c. Option 3: Node.js (JavaScript)

Choice: Node.js (JavaScript)

Justification: Using Node.js will allow us to use JavaScript both on the frontend and backend, simplifying the interaction between them. Using a single programming language across the stack reduces context switching, improves development speed and efficiency, and allows shared type definitions with TypeScript. Spring Boot was not selected because it would introduce additional complexity that is unnecessary for our project. PHP was not selected because it's slightly outdated, and does not align with our React-based frontend.

6. **Which cryptography framework should we use?**

a. Option 1: Web Cryptography API
b. Option 2: Crypto-js

Choice: Web Cryptography API

Justification: Though Crypto-js may be easier to use and integrate with a React app, it is considered much less secure (for example, Crypto-js suffers from simpler side-channel attacks). On the other hand, the Web Cryptography API is built directly into the browser and interfaces with lower-level OS cryptography primitives. This design makes it much more secure. Web Cryptography API is also much faster, as encryption is hardware-accelerated instead of being done by the CPU as with Crypto-js. This difference will be noticeable for larger files.

7. **Which database should we use for server-end storage?**

a. Option 1: MySQL
b. Option 2: MongoDB
c. Option 3: PostgreSQL
d. Option 4: Supabase

Choice: Supabase

Justification: Supabase is much simpler than the other options. For example, MySQL is simply a relational database, so all data interactions would have to be built with our own API. This would require a lot of time and could possibly distract from building core features. Supabase, on the other hand, has an accessible REST auto-API. MongoDB is a JSON database framework and not a relational database, which is awkward to integrate into file storage. Finally, PostgreSQL suffers the same issue as MySQL. By providing no abstraction, all data management logic will need to be written by us.

8. **Which database framework should we use for client-end storage?**

a. Option 1: File System Access API

b. Option 2: IndexedDB
c. Option 3: LocalStorage

Choice: IndexedDB

Justification: We selected IndexedDB for client-side storage because it is specifically designed for structured data and large binary objects. Since we must support text and audio files, indexedDB allows for storing large encrypted files without size limitations or synchronous blocking behavior (i.e. the UI freezing on file upload). The File System Access API was not selected because it introduces permission requirements needed from the user, which complicates the user experience. LocalStorage was not selected because of its limited storage capacity. Since it can only hold strings, it is not suitable for large encrypted files.

Design Details

Class Design (Client)

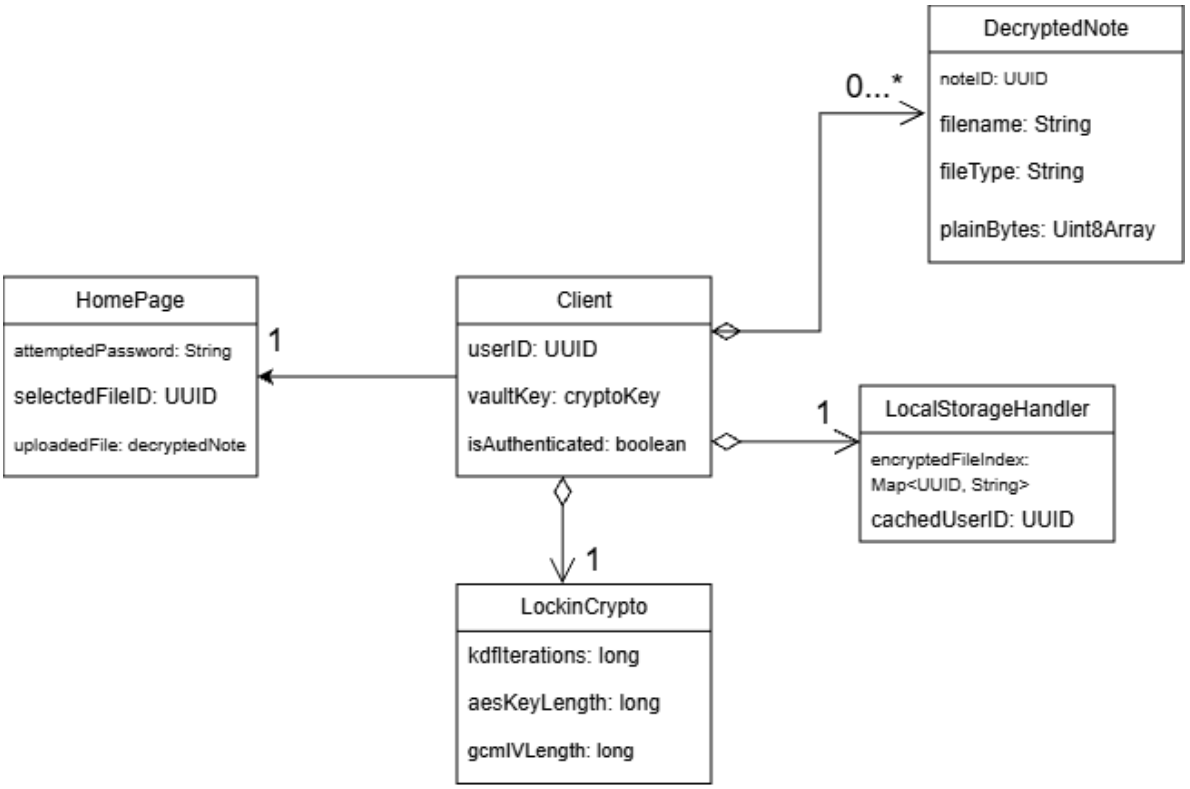

Figure 2: Main classes on the client-end.

Class Design (Server)

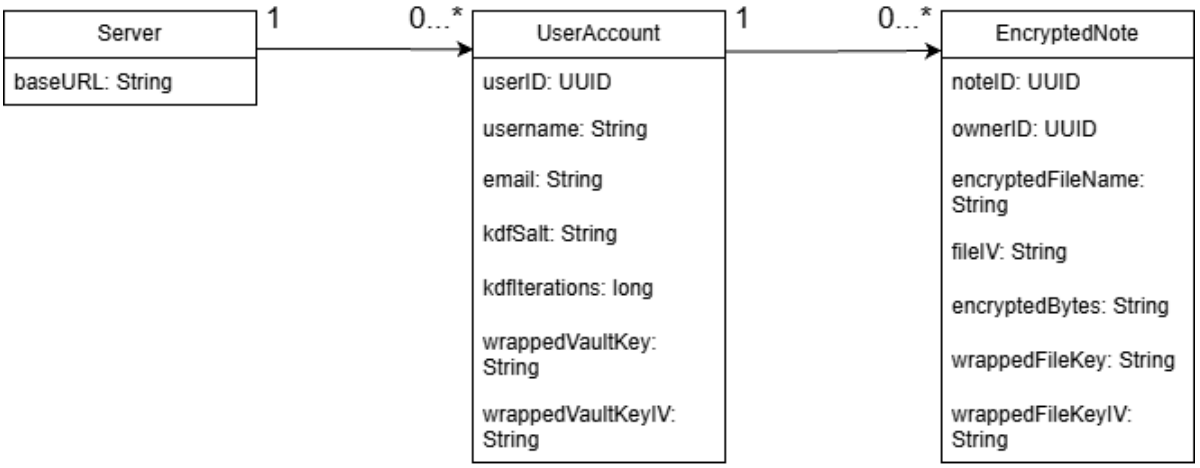

Figure 3: Main classes on the server-end.

Description of Classes and their Interactions

Client

- Created when a user opens the application.
- Handles all client-side functionality and initializes required objects of classes.
- Manages objects of these classes and calls the necessary methods based on user interaction.
- Sensitive information is never given directly to a client: only after the client successfully derives the correct vault key (via a correct password) will it be able to decrypt files.

HomePage

- Created by the Client when the user opens the application.
- Can be a signup page, a login page, or a home page (if signed in).
- Contains the GUI: text fields, buttons, etc.

LockinCrypto

- Created by the Client upon successful login or signup.
- Handles the encryption of all data using secure encryption techniques (AES-GCM, true random numbers, etc.) via the Web Crypto API.
- Once authenticated, the vault key is used to decrypt filenames and files on the client-side.
- Uploaded notes are encrypted before they are sent to the server or stored locally.
- **No other class will use Web Crypto API or encrypt/decrypt files.**
- **Fresh IVs are generated for each encryption operation and unique salts are generated for each key derivation.**

LocalStorageHandler

- Created by LockinCrypto on successful login or signup.
- Is the interface between the Client and client-side notes.
- Uses IndexedDB (via the Web Storage API) to store files locally.

DecryptedNote

- Represents a decrypted note, or an uploaded file that hasn't been encrypted yet.
- Can be a text, audio, or video file.
- Contains a file name, type, and non-secret cryptography metadata.
- Is the only instance where sensitive information is exposed as plaintext.

Server

- Is the recipient of requests from many client connections.
- Communicates with clients via HTTP requests.
- Is the interface with the Supabase database (where account information and server-side notes are stored).

UserAccount

- Represents a user's account, is created by the Server on signup.
- Includes non-secret metadata sent to the server for storage.
- Includes username, salt (generated via PBKDF2), IVs, and a wrapped vault key.

EncryptedNote

- Represents an encrypted file in storage.
- Can be a text, audio, or video file.
- Contains a file name, type, and non-secret cryptography metadata.
- Contains an array of encrypted bytes, which are the ciphertext of the file.

Activity Diagram

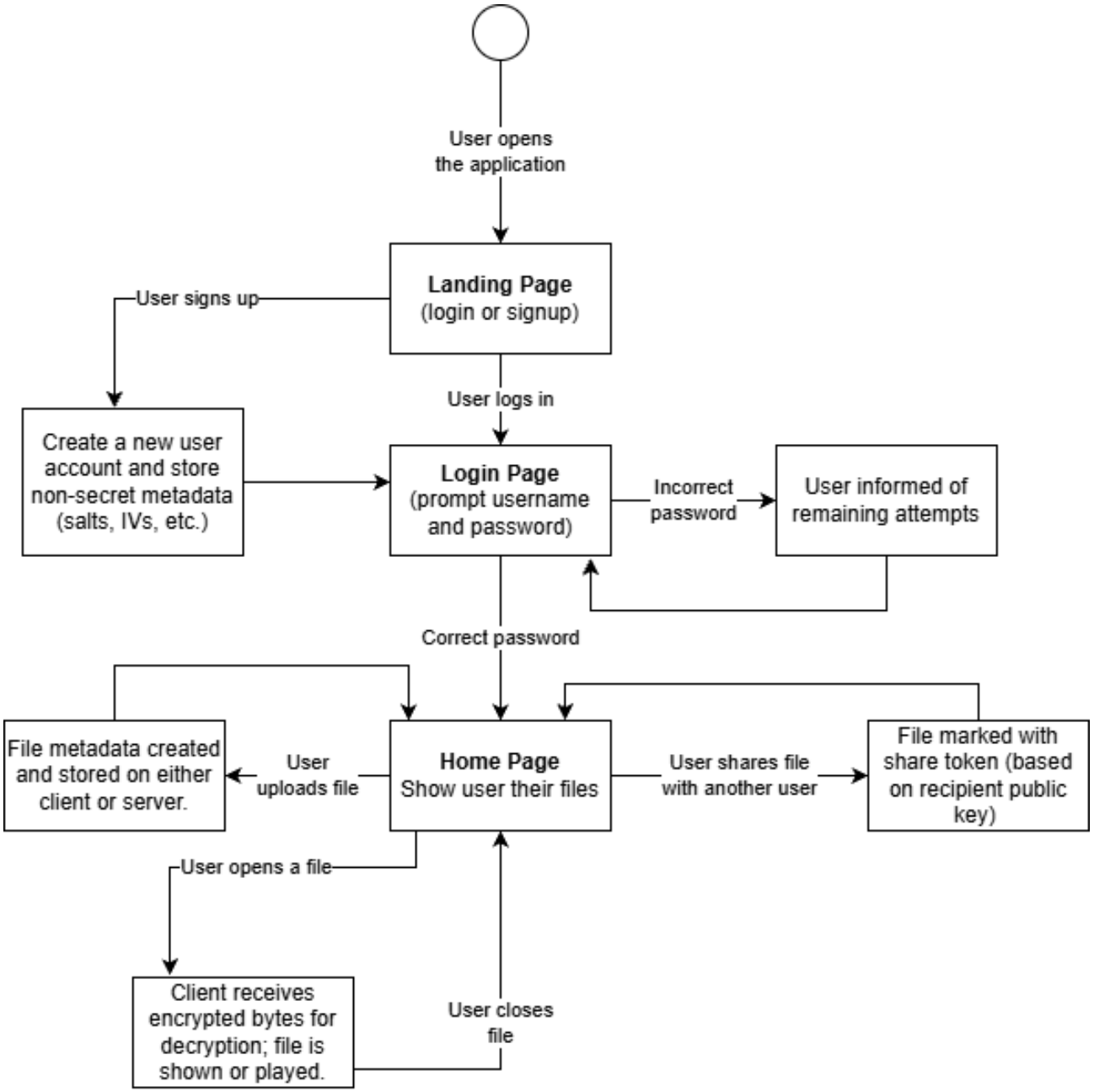

Figure 4: Activity diagram displaying main user actions with the app. Client-side GUI states are shown in bold.

Sequence Diagrams

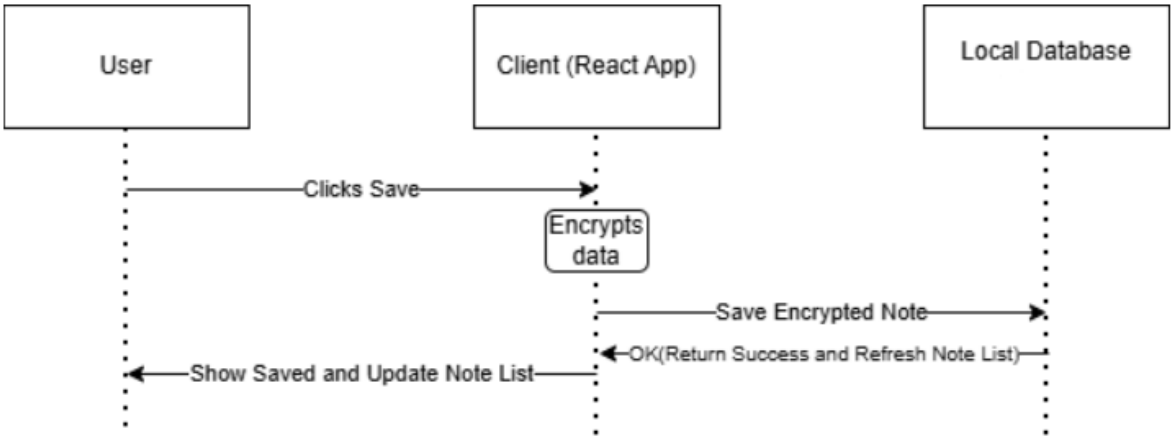

Figure 5: Sequence diagram showing how files are saved locally.

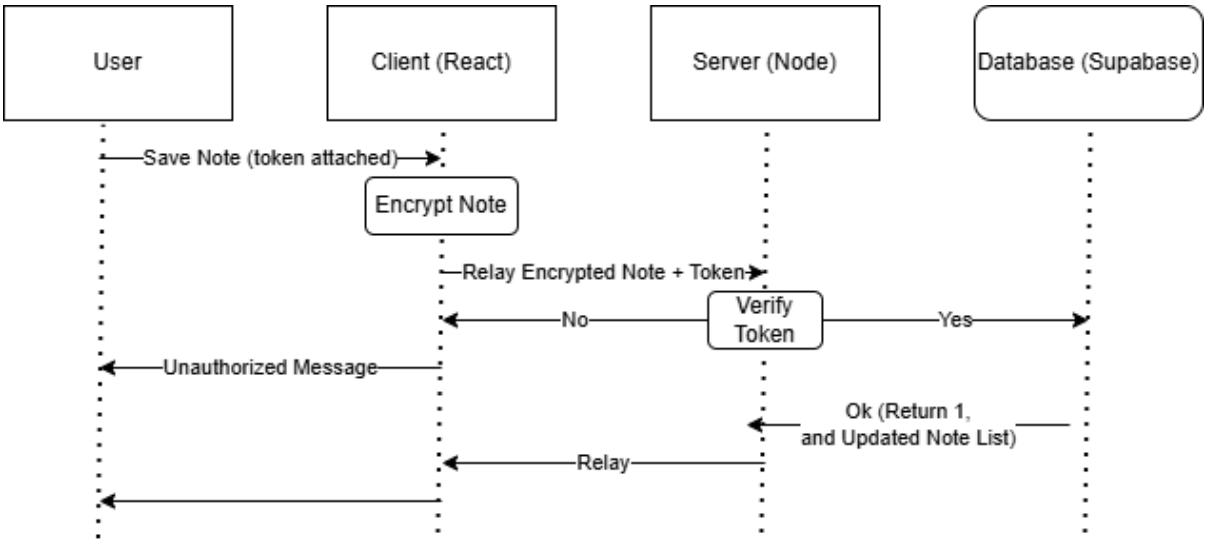

Figure 6: Sequence diagram showing how files are saved to the cloud.