chalmers university
of technology

department of computer science
and engineering

web applications

enchatt - An end to end encrypted application

DAT076 Project

March 2025

Students: Mohammed Uqla, Aram Jamal &
Jonathan Storm

TA: Petter Lindgren

Introduction

enchatt is, as the title of this report suggests, is an end to end encrypted web application for real time communication between users. Our application frontend uses hashing of room keys and password derivation algorithms to generate safe keys that are used to encrypt messages sent via web sockets to the backend.

As avid believers in the individuals right to privacy, we would like for the maintainers of *enchatt* to be aligned with our vision.

Here is a link to the repo: *enchatt*

Use-cases

The *enchatt* application has the following use-cases:

- Receive an encrypted chat log for an existing key.
    1. The user opens the application.
    2. The websockets on the backend and frontend establish a connection.
    3. The user chooses a username.
    4. The user types in a specific key and the frontend does a get request for that key from the backend.
    5. The backend fetches that chat from the database and sends it in the response to the frontend.
    6. The frontend decrypts the messages of the received chat log and displays their contents.
- Create a new chat with a new key previously not used.
    1. The user opens the application.
    2. The websockets on the backend and frontend establish a connection.
    3. The user chooses a username.
    4. The user types in a unique new key and the frontend does a get request for that key from the backend.
    5. The backend sees that no such key was previously initialized and created the chat in the backend.
    6. The frontend recieves the respond that a new chat was created.
- Receive multiple chat logs for multiple keys, color coded and displayed in the same chat window with correct time-based sorting.
    1. The user opens the application.
    2. The websocketss on the backend and frontend establish a connection.
    3. The user chooses a username.

4. The user types in multiple specific keys and the frontend does a get request for those keys from the backend.
5. The backend goes through the keys and either creates or retrieves the chats for those keys from the database, depending on if the chats already exist or not.
6. The backend sorts the messages from the retrieved chats according to their timestamps and sends them to the frontend in the response.
7. The frontend decrypts the messages, color codes them depending on which inputted key they are from, and then displays them.

- Send encrypted messages to a specific chat.

1. The user opens the application.
2. The websockets on the backend and frontend establish a connection.
3. The user chooses a username.
4. The user types in multiple or one unique keys.
5. The frontend does a get request for that/those keys to the frontend.
6. The backend goes through the keys and either creates or retrieves the chats for those keys from the database, depending on if the chats already exist or not.
7. The backend sorts the messages from the retrieved chats according to their timestamps and sends them to the frontend in the response.
8. The frontend decrypts the messages from the chats, color codes them depending on which key they are from, and then.
9. The frontend signals to the backend that the user has connected to those chats through the websockets.
10. The user types in a message for a specific key and sends it.
11. The frontend encrypts that message and sends it to the backend through websockets.
12. The backend emits the encrypted message back to all other users that are connected to that specific key through websockets.
13. The frontend of all other users that are connected to that specific key immediately decrypts and displays the new message.

User Manual

Next follows a guide to installing, running and using *enchatt*.

Step by step guide to install and run *enchatt*

.

I Clone the repo

II Setting up PostgreSQL 17 for the application:

i Go to the official PostgreSQL website: PostgreSQL Download Page

ii Select your operation system and download PostgreSQL17

iii Run the installer and follow these setup instructions:

i Set the port to 5432

ii Choose a password for the Postgres user(you will need this later)

iii Complete the installation proecess

iv Navigate to root/server

i Create a new file named .env

ii Open .env and paste the following line:

iii DB_URL='postgres://postgres:YOURPASSWORD@localhost:5432/enchatt_db'

iv Replace YOURPASSWORD with the password you set during installation

v Now Create database manually by following these steps:

i **Open your terminal and enter:**

```
psql -h localhost -U postgres
```

ii **If successful, you should see the PostgreSQL prompt:**

```
postgres=#
```

iii **Create a new database with:**

```
CREATE DATABASE enchatt_db;
```

iv **To confirm that the database has been created, type:**

```
\l
```

vi You are ready to use PostgreSQL17

III Open a terminal and change directory to /yourPath/Enchatt/server, then run the commands:

i npm install

ii npm run dev

IV Open a new terminal and change directory to /yourPath/Enchatt/client, then run the commands:

i npm install

ii npm run dev

V Open your browser and type in the URL: http://localhost:5173/

How to use *enchatt*

After following the setup guide you should see the login page.

Figure 1: *enchatt* login page

From here you type in a username:

Figure 2: *enchatt* username input

You are then signed in to *enchatt*.

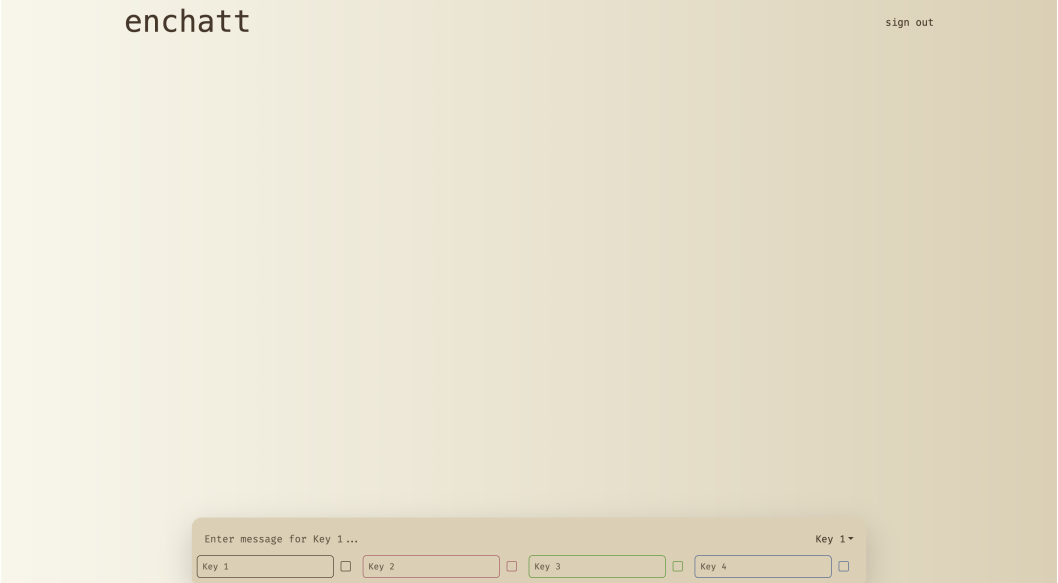

Figure 3: *enchatt* chat page

Now you can type in a chat key in any of the boxes at the bottom of the page to either create a new chat room or join an already existing one. You are then connected to that chat room and can start sending messages.

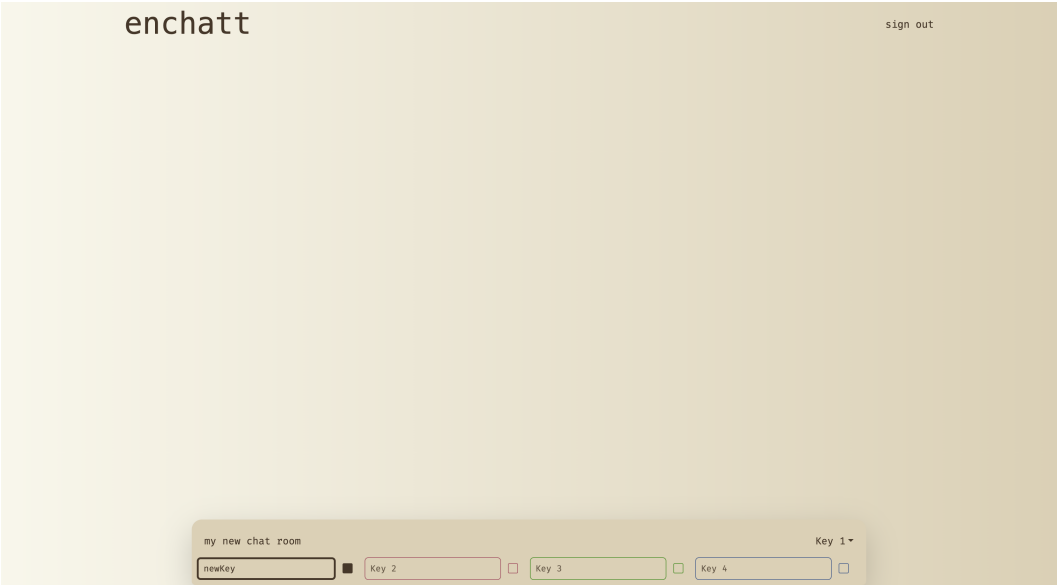

Figure 4: Creating a new chat room

The now highlighted box tells you that you are connected to that chat room. And you can see your message sent to that chat room.

Figure 5: Message sent to newly created chat

And of course, joining already existing chat rooms renders the complete chat log. Even before you joined.

Figure 6: Joining already existing chat rooms

And if you want, you can have up to four chat rooms active which are combined into one.

Figure 7: Combining chat rooms

Design

Below you will find a description of the frontend components, a description of the `WebSockets` connection and encryption, a specification of the backend API and an ER-diagram of the database.

Client

The client side of *enchatt* consists of 12 components and 3 utils.

App.tsx

`App.tsx` is the main component and handles the routing between the login page component `Login.tsx` and the chat page component `Enchatt.tsx`.

Props

`None` The `App.tsx` component does not receive any props.

States

`username` A `string` representing the username, which gets passed to `Enchatt.tsx`.

Backend Calls

`None` The `App.tsx` component does not make any direct backend calls. It primarily manages routing and state.

Login.tsx

`Login.tsx` is the first page a user sees when starting *enchatt*. This component is part of the process of handling the users `username`. It also contains a `Logo.tsx`.

Props

`handleUsername()` This is a callback function to `App.tsx`. This function sends the `username` input given in `LoginBox.tsx`.

States

`None` The `Login.tsx` component does not have any states

Backend Calls

`None` The `Login.tsx` component does not make any direct backend calls. It manages the transfer of `username` between components.

Logo.tsx

`Logo.tsx` is a simple component and only represent the logo for *enchatt*. This component uses a `EncryptedText.tsx` to achieve a scramble (encryption) effect.

Props

`None` The `Logo.tsx` component does not receive any props.

States

`State` The `Logo.tsx` component does not have any states

Backend Calls

`None` The `Logo.tsx` component does not make any direct backend calls. It is only a logo.

EncryptedText.tsx

`EncryptedText.tsx` is a component for displaying a text with a scramble (encryption) effect to users. This component is designed to handle all types of content. But given the encryption effect, `EncryptedText.tsx` should be used with care to not overwhelm users.

Props

`text` A `string` representing the actual content to display.

`encryptEffetTime` A `number` representing for how long (in milli seconds) to scramble the actual content.

`trueTextTime` A `number` representing for how long (in milli seconds) to display the actual content.

`fontSizeProp` An optional `string` which alters the font size. Default is set to "1.5rem".

States

`displayText` A `string` received via the `text` prop which is passed down from the parent component.

`isScrambled` A `boolean` to keep track of when to encrypt (scramble) the `displayText`.

Backend Calls

`None` The `EncryptedText.tsx` component does not make any direct backend calls. It only displays content with an encryption effect.

LoginBox.tsx

`LoginBox.tsx` is the main component of the process of handling the users `username`. It takes the input given by the user with help from a `EncryptedInput.tsx` component.

Props

`handleUsernameInput()` This is a callback function to `Login.tsx`. This function sends the `username` input given in an `EncryptedInput.tsx` component.

States

`None` The `LoginBox.tsx` component does not have any states

Backend Calls

`None` The `LoginBox.tsx` component does not make any direct backend calls. It manages the transfer of `username` between components.

EncryptedInput.tsx

The `EncryptedInput.tsx` component contains a scramble (encryption) effect, which is only for aesthetic purposes. `EncryptedInput.tsx` is designed to handle all types of input. However, given the encryption effect, `EncryptedInput.tsx` should be used with care to not overwhelm users.

Props

`inputPlaceholder` A `string` which serves as the placeholder text to apply the encryption effect on.

`handleInput()` This is a callback function to the parent component. At the moment it is only used by `LoginBox.tsx` to handle the `username` input.

States

`inputValue` A `string` representing the input from the user.

`displayText` A `string` received via the `inputPlaceholder` prop which is passed down from the parent component.

`isScrambled` A `boolean` to keep track of when to encrypt (scramble) the `displayText`.

Backend Calls

`None` The `EncryptedInput.tsx` component does not make any direct backend calls. It takes an input from the user and passes it to the parent component.

Enchatt.tsx

`Enchatt.tsx` is the main page in *enchatt* application. This component is the chat page which users see when they are connected. `Enchatt.tsx` has three child components. They are `TopBar.tsx`, `ChatBox.tsx` and `ChatSubmit.tsx`.

Props

`username` A `string` which represents the `username` of the current user.

`navigateToLogin()` This is a callback function that goes back to `App.tsx`. This function is then passed down to `TopBar.tsx`.

States

`derivedKeyValues` A `type` that stores cryptographic keys from raw keys.

`rawKeyValues` A `type` that stores raw cryptographic keys before they are derived.

Backend Calls

`None` The `Enchatt.tsx` component does not make any direct backend calls. It mainly renders the chat page of *enchatt*.

TopBar.tsx

`TopBar.tsx` is a component intended to manage content at the top of the *enchatt* window. `TopBar.tsx` has two child components, and those are `Logo.tsx` and `SignOut.tsx`.

Props

`navigateToLogin()` This is a callback function that goes all the way back to `App.tsx` where the user is ultimately redirected to `Login.tsx`. This function is then passed down to `SignOut.tsx`.

States

`None` The `TopBar.tsx` component does not have any states.

Backend Calls

`None` The `TopBar.tsx` component does not make any direct backend calls. It is only arranging elements in the top bar.

SignOut.tsx

`SignOut.tsx` is a component which only handles the scenario when a user wants to exit *enchatt*.

Props

`navigateToLoginFromTopbar()` This is a callback function that goes all the way back to `App.tsx` where the user is ultimately redirected to `Login.tsx`.

States

`None` The `SignOut.tsx` component does not have any states.

Backend Calls

`None` The `SignOut.tsx` component does not make any direct backend calls. It only renders a button for signing out (exiting *enchatt*).

ChatBox.tsx

`ChatBox.tsx` is the component where usernames and messages gets displayed.

Props

`rawKeys` Raw encryption keys passed from `Enchatt.tsx` used to retrive chats.

`derivedKeys` Derived encryption keys passed from `Enchatt.tsx` used to decrypt messages.

States

`chat` Stores chat messages and updates when new one arrives.

Backend Calls

`getMultipleChats()` Fetches chat messages from backend.

`socket.emit()` Sends a request to join a chat.

`socket.on()` Listens for new messages.

ChatSubmit.tsx

`ChatBox.tsx` is the component where the user types in messages and chatKeys.

Props

`updateDerivedKeys()` A function that takes `activeKeys` as `rawKeys` to update the `derivedKeys`.

`updateRawKeys()` A function that takes `rawKeys` to update the `rawKeys`.

`derivedKeys` Used for encrypting messages before sending them.

`rawKeys` Raw Keys provided by the user.

`username` The sender username.

States

`selectedKey` Stores the `selectedKey` for encryption, updated when user select another key than the default 'Key 1'.

`newMessage` Stores messages and cleares after the user sent the message.

`keyValues` Stores user inputted `keyValue`, updated when user type in a key field.

Backend Calls

`sendMessage()` Sends message when the user input Enter over a `WebSocket` connection.

Message.tsx

`Message.tsx` is the component which represents the decrypted messages in `ChatBox.tsx`.

Props

`message` A function that takes `activeKeys` as `rawKeys` to update the `derivedKeys`.

`derivedKeys` Used for encrypting messages before sending them.

`rawKeys` Raw Keys provided by the user.

States

`decryptedContent` Stores the decrypted messages

`error` Stores error if the decryption fails

Backend Calls

`None` The `Message.tsx` component does not make any direct backend calls. It is only a component for displaying messages to the user.

Utils

There are three utils used in *enchatt*. Their purpose is to assist the frontend with the `WebSocket` connection and the encryption of messages.

socket.ts

`socket.ts` is the client side of the `WebSocket`. This util is responsible for establishing the connection to the server side of the `WebSocket`. It achieves this by implementing the Socket.io API. The following code snippet are the two lines that allows for this connection.

```
import { io } from "socket.io-client";
const socket = io("http://localhost:8080", { withCredentials: true });
```

The URL tells the front end `WebSocket` which port the server side `WebSocket` is located at. And `withCredentials` set to `true` we allow for cookies and authentication headers.

keys.ts

`keys.ts` manages cryptographic key representations and their transformations. It defines types and functions for handling raw and derived keys, along with utility functions for key-related operations.

This represents the allowed key names:

```
export type KeyString = "Key-1" | "Key-2" | "Key-3" | "Key-4";
```

This structure holds a raw key string, its hashed version, and an optional cryptographic salt:

```
export interface RawKeyObject {
                                raw: string,
                                hashed: string,
                                salt?: string };
```

This defines the derived keys used for encryption and decryption:

```
export interface DerivedKeys {
                                key1?: CryptoKey,
                                key2?: CryptoKey,
                                key3?: CryptoKey,
                                key4?: CryptoKey };
```

This function maps a key string to its corresponding object property key:

```
export function convertToKeyString(key: string): keyof RawKeys {
    switch (key) {
        case "Key-1": return "key1";
        case "Key-2": return "key2";
        case "Key-3": return "key3";
        case "Key-4": return "key4";
        default: throw new Error(`Unknown key: ${key}`);
    }
}
```

These functions return the appropriate CSS class names based on the key string:

```
export function getKeyClass(keyString: KeyString): string {
    return keyString.toLowerCase().replace("-", "");
}

export function getKeyBorderClass(keyString: KeyString): string {
    return getKeyClass(keyString) + "border";
}
```

To conclude, `keys.ts` simplifies cryptographic key handling and integrates key identification with styling functionality.

encryption.ts

In the `encryption.ts` file the methods responsible for encrypting and decrypting messages on the frontend are present. The encryption scheme ensures that messages remain private and cannot be accessed by the backend, as the messages are encrypted using the raw keys and the backend only associates the encrypted messages with their hashed keys.

Key handling: Each chat room has a unique key used for retrieving the chat through its hashed derivation and another through its PBKDF2 (Password Based Key Derivation Function 2) derivation. The raw keys and their PBKDF2 variants never leave the frontend, instead:

- The raw keys are hashed on the client side before being sent to the backend for chat retrieval.
- A randomly generated salt, provided by the backend on the fetching and creation of chat rooms, is used in the PBKDF2 key derivation.
- The derived PBKDF2 key is used to to encrypt all messages before sending them to the backend.

Message encryption:

- Each message is encrypted with AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) along with a unique IV (Initialization Vector) that is randomly generated before the encryption. This ensures that no two identical messages appear as the same cipher text on the backend.
- The IV is included in the message payload and is readable to the server so that it can be sent back to allow for frontend decryption.
- The backend never has access to the unencrypted room keys, and can therefore not decrypt stored messages.

Message decryption:

- On the frontend the hashed key is used to retrieve encrypted messages and their corresponding IVs from the backend.

- Messages are decrypted by using the previously stored chat room salt provided by the backend, along with PBKDF2 to regenerate the encryption key.
- With the regenerated encryption key and the retrieved IVs the messages are decrypted on frontend.

Figures 8 and 9 illustrate the encryption and decryption processes through diagrams.

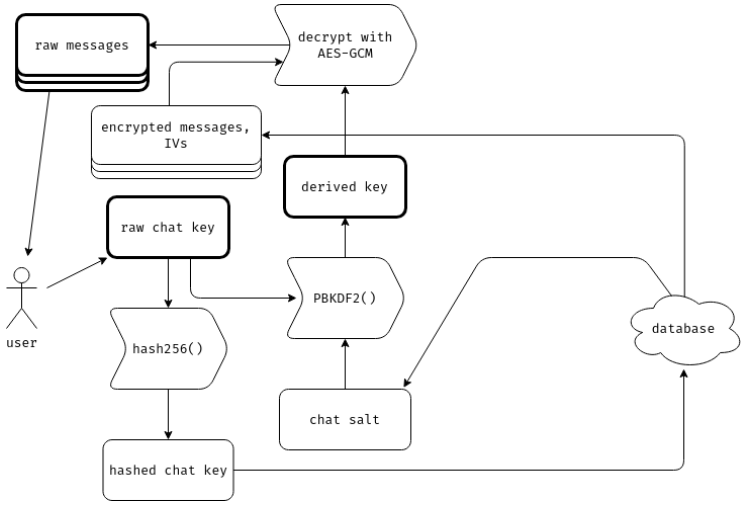

Figure 8: Encryption diagram for retrieving and decrypting messages from the backend. The boxes with thick borders represents the data that is only accessed by the frontend.

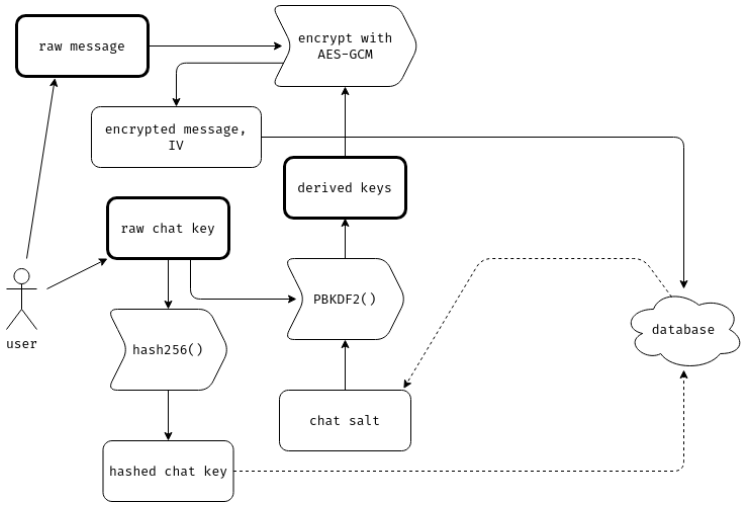

Figure 9: Encryption diagram for encrypting messages and sending them to the backend. The boxes with thick borders represents the data that is only accessed by the frontend.

Server

The server side of *enchatt* consists mainly of a service, a router and a `postgreSQL` database. But worth highlighting in this report is the `socket.ts` file.

socket.ts

`socket.ts` is the server side of the `WebSocket`. This util is responsible for handling real-time communication with connected clients. It achieves this by implementing the Socket.io API. The following code snippet initializes the WebSocket server:

```
import \{ Server \} from "socket.io";\\
const io = new Server(httpServer, { cors: {
                                    origin: "http://localhost:5173",
                                    credentials: true } });
```

The `origin` specifies which frontend is allowed to connect, and `credentials` set to `true` enables authentication headers and cookies. When a client is connected, *enchatt* has four use cases (mentioned above). The two following socket events are able to handle all four use cases.

Creating and joining chat rooms uses:

```
socket.on("joinChat", async (chatId) => \{\\
const chat = await chatService.getOrCreateChat(chatId);\\
if (chat) {
    socket.join(chatId);
} else {
    socket.emit("error", "Chat does not exist");\\
}
});
```

And sending messages are handled by:

```
socket.on("sendMessage", async ({ chatId, sender, message, iv }) => {
const storedMessage = await chatService.sendMessage(chatId, sender, message, iv)
io.to(chatId).emit("receiveMessage", storedMessage);
);
```

The code above stores { `chatId, sender, message, iv`} (the encrypted message) in the database with the help of `chatService`. And send it to the correct chat room with `io.to(chatId).emit("receiveMessage", storedMessage)`.

API specification

```
Title: enchatt
Description: The API specification for enchatt Application
paths:
  /chat/{key}:
    get:
      tags:
        - Chats
      description: returns chat associted with certain key
```

```
    operationId: getOrCreateChat
    parameters:
      - in: path
        name: key
        required: true
        description: Unique key identifying the chat
        schema:
          type: string
      - in: query
        name: limit
        description: Maximum number of messages to return
        schema:
          type: integer
          format: int32
    responses:
      '200':
        description: The chat messages and initialization vector
        content:
          application/json:
            schema:
              type: object
              properties:
                messages:
                  type: array
                  items:
                    type: object
                ivs:
                  type: string
  post:
    tags:
      - Chats
    description: Sends a message to the chat associated with a
       specific key.
    operationId: sendMessage
    parameters:
      - in: path
        name: key
        required: true
        description: Unique key identifying the chat.
        schema:
          type: string
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
```

```
            required:
              - sender
              - content
              - iv
            properties:
              sender:
                type: string
                description: The sender of the message.
              content:
                type: string
                description: The message content.
              iv:
                type: string
                description: Initialization vector for encryption.
    responses:
      '201':
        description: Message successfully sent.
        content:
          application/json:
            schema:
              type: object
              properties:
                id:
                  type: string
                  description: Unique identifier for the message.
                sender:
                  type: string
                content:
                  type: string
                timestamp:
                  type: string
                  format: date-time
      '400':
        description: Bad request, missing or invalid parameters.
      '500':
        description: Internal server error.

/chats:
  get:
    tags:
      - Chats
    description: Retrieves multiple chat messages using up to four
       keys.
    operationId: getMultipleChats
    parameters:
      - in: query
        name: key1
```

```
        description: First chat key (optional).
        schema:
          type: string
      - in: query
        name: key2
        description: Second chat key (optional).
        schema:
          type: string
      - in: query
        name: key3
        description: Third chat key (optional).
        schema:
          type: string
      - in: query
        name: key4
        description: Fourth chat key (optional).
        schema:
          type: string
    responses:
      '202':
        description: Successfully retrieved chat messages.
        content:
          application/json:
            schema:
              type: object
              properties:
                messages:
                  type: array
                  items:
                    type: object
                salts:
                  type: array
                  items:
                    type: string
                    nullable: true
      '400':
        description: Bad request (no keys provided).
      '500':
        description: Internal server error.
```

ER Diagram

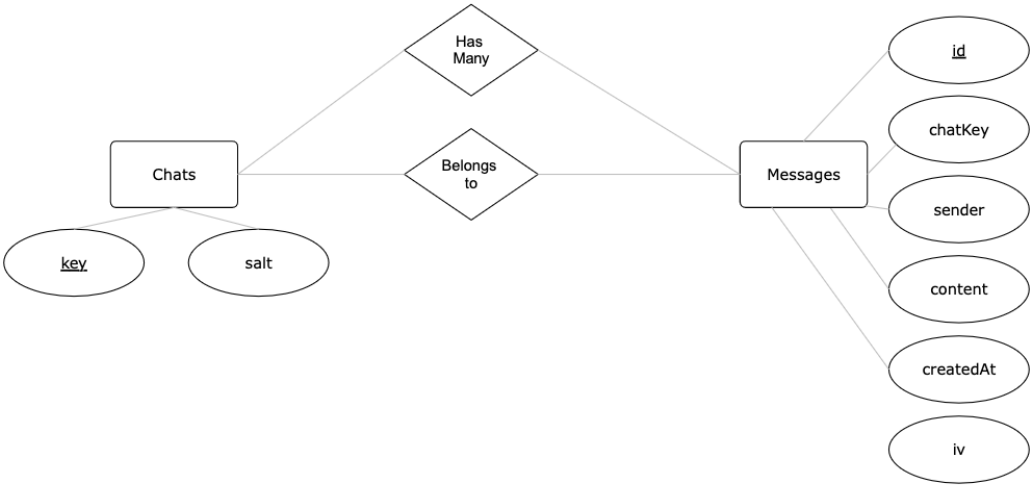

Figure 10: Database ER Diagram

Responsibilities

Everyone

We have been doing a lot of pair (trio) programming. Each member has more or less been involved in everything. It was only in the later stages of the project that we went for the divide and conquer approach.

Mohammed Uqla

Something that Mohammed has been working with more than the rest,is the database and the frontend tests.

Aram Jamal

Something that Aram has been working with more than the rest is the encryption and the backend tests.

Jonathan Storm

Something that Jonathan has been working with more than the rest is the frontend and the WebSockets.