1 Introduction

This project is to implement a messaging app for encrypted communication. Based on the given template, we have enabled secure user registration, login, friend requests, and encrypted chats with friends. During the use of the app, we fully ensure the security of the users' information, effectively preventing the potential attacks such as DDoS and man-in-the-middle. Based on the specific scoring criteria , we have done a detailed related report below with screenshots of code and web pages.

2 Approach explanation

Part: Basic Functionality Requirements:

1

When a user attempts to log in, the frontend initially hashes the user's inputted password using SHA-256, as shown in figure 1. Then, the username and hashed password are sent to the server. The server checks if the user exists; if not, an error message is displayed, as shown in figure 2. If the user exists, the server hashes the received hashed password again using a salt stored in the database, and then compares this double-hashed password with the hash value stored in the database. If they match, the login is successful, and the user is redirected to the home page, as shown in 3. If they do not match, an error message is displayed, as shown in figure 4.

When users sign up, to prevent XSS attacks, we have imposed restrictions on the length of usernames and passwords, as well as on the characters that can be used, as shown in Figure 5.

```
68          // login function to well login...
69          async function login() {
70              // this fancy syntax is part of the Jinja syntax
71              // login_user is a Python function, this gets the URL that calls that P
72              // you know the one with the:
73              // app.route("/login/user", methods=["POST"])
74              // login_user()
75              // so... "{{ url_for('login_user')}}"
76              // gives us -> "http://blabla/login/user"
77
78              let loginURL = "{{ url_for('login_user') }}";
79
80              // axios post is a fancy way of posting a request to the server,
81              // we pass in the username and password here
82
83              // Once hashed, pass it to app.py
84              let hashedPassword = CryptoJS.SHA256($("#password").val()).toString();
85
86              let res = await axios.post(loginURL, {
87                  username: $("#username").val(),
88                  password: hashedPassword
89              });
```

Figure 1: Hashed Password in frontend

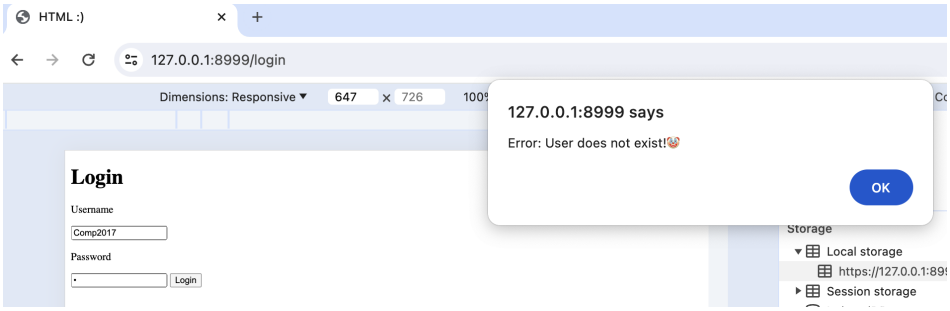


Figure 2: No such user

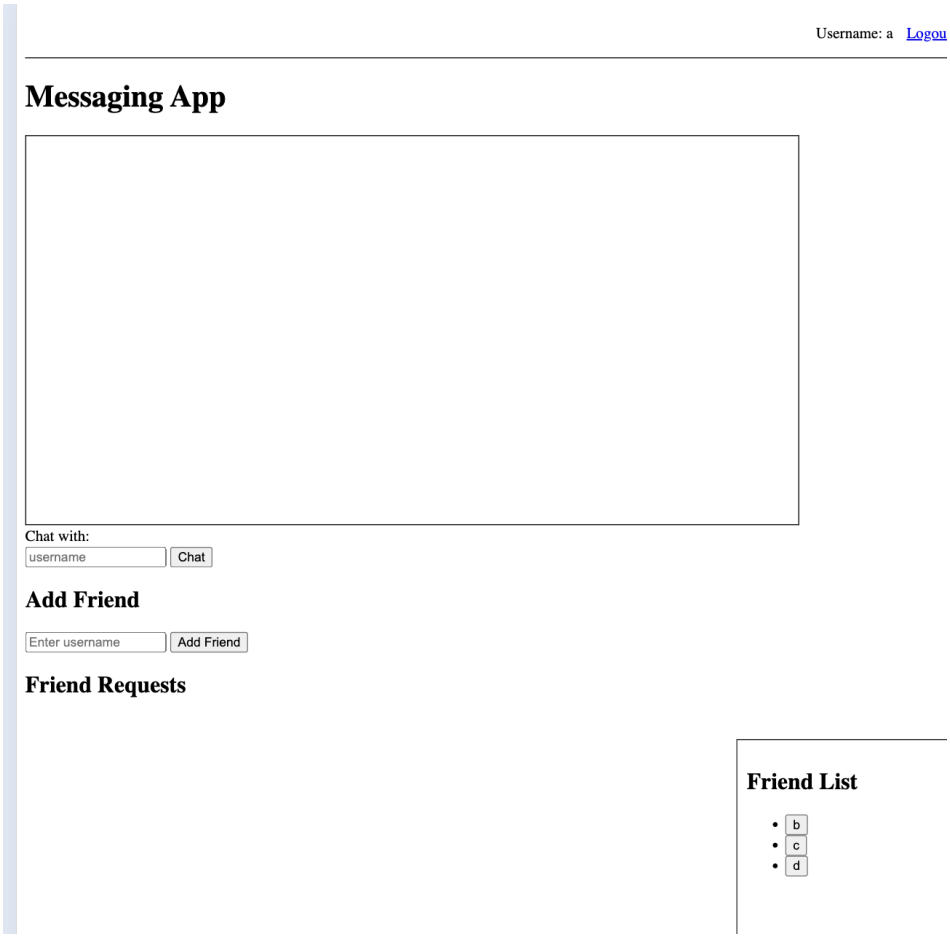


Figure 3: login successful

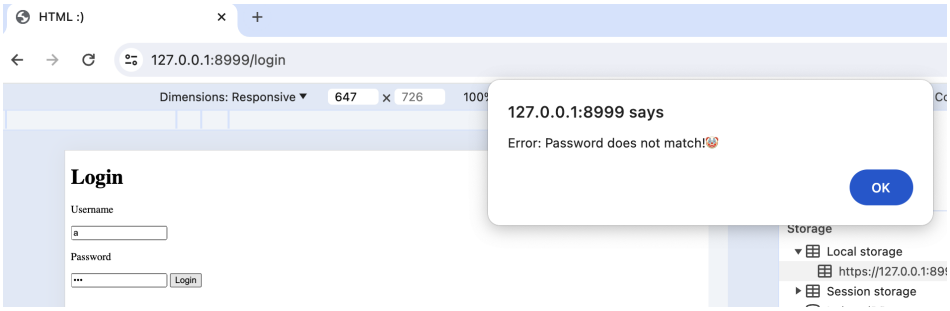


Figure 4: password doesn't match

```
74      function isValidPassword(password) {
75          const regex = /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}
76          return regex.test(password);
77      }
78
79      function isValidUsername(username) {
80          const regex = /^[a-zA-Z0-9_]{1,9}$/;
81          return regex.test(username);
82      }
83
```

Figure 5: signup.jinja

2

Once user logs in successfully, a dynamic friend list can be shown bottom right the webpage. It displays all the users who this user is friend with as buttons. By clicking on the bottoms, the user can join in the chat room and chat with the friend. In order to implement this feature, we designed a class called friendship (shown in Figure 6) , then we use a function called get friends for user, to get all the users who this user if friend with.(shown in Figure 7) . Then we use a function called fetchfriends in the frontend to display all the friends as buttons. Once clicking the bottom, it will trigger join room function, which allows user to chat with friends(shown in Figure 8).

```
114 class Friendship(Base):
115     __tablename__ = 'friendship'
116     user_username = Column(String,primary_key=True)
117     friend_username = Column(String,primary_key=True)
118
```

Figure 6: models.py define a class called friendship

```python
def get_friends_for_user(username: str):
    with Session(engine) as session:
        # check friendship
        friendships = session.query(Friendship).filter(
            (Friendship.user_username == username)
        ).all()

        # extract the friend's username
        friends_usernames = [friendship.friend_username for friendship in friendships]
        friends = []
        for friend_username in friends_usernames:
            friend = session.query(User).filter(User.username == friend_username).first()
            if friend:
                friends.append({"username": friend.username})

        return friends
```

Figure 7: db.py get all the friends of the current user

```javascript
function fetchFriends() {
    // Read the current username from the data-username attribute of the body element
    let currentUsername = "{{ username }}";

    fetch(`/get_friends?username=${currentUsername}`)
    .then(response => {
        if (!response.ok) {
            throw new Error('Failed to fetch friends');
        }
        return response.json();
    })
    .then(data => {
        const friendList = document.getElementById('friend_list');
        friendList.innerHTML = ''; // clear the existing friendlist

        // iterate through the returned friend data and create a list item for each friend
        data.forEach(friend => {
            const li = document.createElement('li');
            const button = document.createElement('button');
            button.textContent = friend['username'];
            button.onclick = function() {
                join_room(friend['username']);
            };
            li.appendChild(button);
            friendList.appendChild(li);
        });
    })
    .catch(error => console.error('Error fetching friends:', error));
}
```

Figure 8: home.jinja fetch friends function

3

The user can send a friend request to the other user by entering the username(shown in Figure 9). We designed a class called FriendRequest to implement this feature (shown in Figure 10). Once entering the username, the server will check whether the username exists. If the username is valid, then it will check whether they are already been friends11. If so, the friend request will not be sent.

Figure 9: home page: add friend feature

```python
class RequestStatus(PyEnum):
    PENDING = 'pending'
    APPROVED = 'approved'
    REJECTED = 'rejected'

class FriendRequest(Base):
    __tablename__ = 'friend_request'
    id = Column(Integer, primary_key=True)
    sender_id = Column(String, ForeignKey('user.username'))
    receiver_id = Column(String, ForeignKey('user.username'))
    status = Column(String)

```

Figure 10: models.py define a class called friendrequest

```python
def send_friend_request(sender_username: str, receiver_username: str):
    print(f"sender:{sender_username}")
    print(f"receiver:{receiver_username}")
    with Session(engine) as session:
        # check if the recipient exists
        receiver = session.get(User, receiver_username)
        print(receiver)
        if not receiver:
            return "Receiver does not exist."

        # check if a friend request has already been sent
        existing_request = session.query(FriendRequest).filter(
        (FriendRequest.sender_id == sender_username) &
        (FriendRequest.receiver_id == receiver_username) &
        (FriendRequest.status.in_([RequestStatus.PENDING.value, RequestStatus.APPROVED.value]))
        ).first()
        if existing_request:
            return "Friend request already sent or already friends."

        # create and save a new friend request
        new_request = FriendRequest(sender_id=sender_username, receiver_id=receiver_username, status=RequestStatus.PENDING.value)
        session.add(new_request)
        try:
            session.commit()
            print("Friend request successfully added.")
        except Exception as e:
            print(f"Failed to insert friend request: {e}")
            session.rollback()
        session.commit()

        return "Friend request sent successfully."

```

Figure 11: db.py send friend request to the other user

4

Once the user trying to send a friend request, if the users are already friends, the request will not be send and will not be displayed. Otherwise, the request will be displayed dynamically to both sender and receiver. The sender will see a message which shows "To: receiver", while the receiver will see "From: sender", with two buttons "Accept" and "Reject"(shown in Figure 12) . Once the receiver accept the friendship request, the status will be updated 13 , and the receiver will become a friend of the sender and will be shown in the friend list immediately. Otherwise if the receiver reject the request, the status will also be updated immediately and this request will disappear automatically. In order to achieve the dynamic part, we applyed socket to listen to the friendrequest(shown in Figure 14 Figure 15) , so it will update the friend requests automatically without manual refresh.

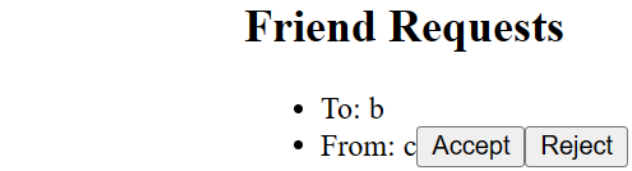


Figure 12: home page: display friend requests

```python
def update_friend_request_status(request_id: int, new_status: str):
    with Session(engine) as session:
        try:
            # find the friend request record by ID
            friend_request = session.query(FriendRequest).filter(FriendRequest.id == request_id).first()
            if not friend_request:
                return False, "Friend request not found."

            # update the status
            friend_request.status = new_status
            if new_status == "approved":
                # check a friendship
                exists = session.query(Friendship).filter_by(user_username=friend_request.sender_id, friend_username=friend_request.receiver_id).first()
                if not exists:
                    new_friendship1 = Friendship(user_username=friend_request.sender_id, friend_username=friend_request.receiver_id)
                    new_friendship2 = Friendship(user_username=friend_request.receiver_id, friend_username=friend_request.sender_id)
                    session.add(new_friendship1)
                    session.add(new_friendship2)
                    print(f"We are friends!!!!!!!: {new_friendship1.user_username} <--> {new_friendship2.user_username}")

            session.commit()
            return True, "Friend request status updated successfully."
        except SQLAlchemyError as e:
            session.rollback()
            print(f"Error updating friend request status: {e}")
            return False, "Error occurred during the update."

```

Figure 13: db.py: update request status after user's operation

```javascript
function fetchFriendRequests() {
    console.log('Fetching friend requests...');
    let currentUsername = "{{ username }}"; // Retrieve the current username from server-side rendering variables or from another source

    fetch(`/get_friend_requests?username=${currentUsername}`)
    .then(response => {
        if (!response.ok) {
            throw new Error('Failed to fetch friend requests');
        }
        return response.json();
    })
    .then(friendRequests => {
        // get the container element for the friend request list
        const friendRequestsList = document.getElementById('friend_requests');
        // clear the current list content
        friendRequestsList.innerHTML = '';

        // iterate through all friend requests and add them to the list
        friendRequests.forEach(request => {···
    })
    .catch(error => console.error('Error fetching friend requests:', error));
}

socket.on('friend_request_update', function(data) {
    console.log('Received update notification:', data.message);
    // Call the function to update friend requests
    fetchFriendRequests();
    fetchFriends();
});

```

Figure 14: home.jinja: fetch friendrequests and socket

```python

@socketio.on('friend_request_sent')
def handle_friend_request_sent(data):
    print("Friend request sent from:", data['sender'], "to:", data['receiver'])
    # Here you can broadcast to specific rooms or globally as needed
    print("Emitting friend_request_update event")
    socketio.emit('friend_request_update', {'message': 'Update your friend requests list'})

```

Figure 15: socket routes.py: listen to the frontend

5

All the friends of the user is displayed as buttons in the friend list (shown in Figure 16). Once clicking on the button, it will call the join room function which allows the user to join in their friend's chat room. To ensure that both user and friend are currently on so they can communicate securely, we add a get users in room medthod for the room class, which checks whether both of them are in the room.(shown in Figure 17). Then we use this method when trying to send messages (shown in Figure 18).

Figure 16: home page: friends are displayed as buttons

```
74      def get_users_in_room(self, room_id: int) -> list[str]:
75          return [user for user, r_id in self.dict.items() if r_id == room_id]
76
```

Figure 17: models.py: check how many users are in the room

```
62  @socketio.on("send")
63  @authenticated_only
64  def send(username, message, room_id):
65      users_in_room = room.get_users_in_room(room_id)
66
67      if len(users_in_room) < 2:
68          emit("error", {"message": "2 users both need to be online"}, to=request.sid)
69          return
70
71      emit("incoming", {
72          "content": f"{username}: {message}",
73          "color": "black",
74          "type": "text"
75      }, to=room_id)
76
77      # include the message type when inserting a message into the database
78      db.insert_message(room_id, username, message)
79
80      return
81
```

Figure 18: socket routes.py: check whether both users are online before sending message

The process of ensuring secure communication between users:

1. When a user successfully logs in or signs up, they generate a pair of cryptographic keys: a public key and a private key as shown in Figure 19.

As shown in Figure 20, the process begins with hashing the password using the SHA-256 hash function to create a fixed-length seed. This enhances security by mitigating risks associated with using raw passwords and provides necessary entropy. The SHA-256 hash output, initially an ArrayBuffer, is then converted into a Uint8Array, and finally into a hexadecimal string to format the seed for key generation. Using this processed seed, the ec.genKeyPair function from the elliptic library is utilized to generate the key pair.

Subsequently, the frontend stores both the public key and the private key as shown in 21. The public key is uploaded to the server using an axios request and stored in a database called public_keys, as shown in Figures 22 and 23. Meanwhile, the private key is retained in the user's local storage, as shown in Figure 24. These keys are generated based on the user's password, ensuring that they are consistently the same each time they are generated.

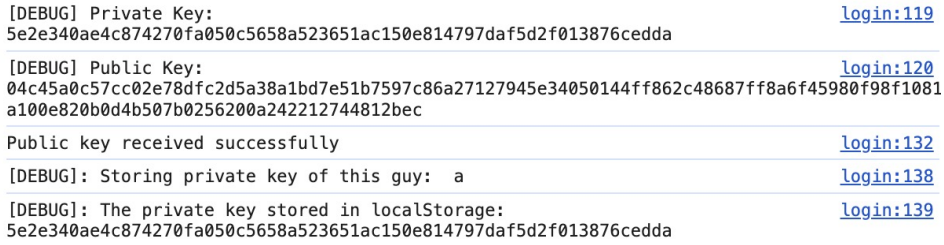


Figure 19: Illustration of private and public keys

```
const ec = new elliptic.ec('p256');

/**
Function to generate ECC key pair from a specific key.
@param {string} key - The key provided by the user, can be a password or any arbitrary string.
@returns {object} An object containing the hex format public and private keys.
*/
function generateECCKeyPairFromKey(key) {
    // use the sha-256 hash function to process the key, generating a fixed-length seed
    return window.crypto.subtle.digest('SHA-256', new TextEncoder().encode(key))
        .then(hash => {
            // convert ArrayBuffer to Uint8Array
            const hashArray = Array.from(new Uint8Array(hash));
            // convert the hash value to a hexadecimal string
            const hashHex = hashArray.map(byte => byte.toString(16).padStart(2, '0')).join('');

            // use the hash value as a random number to generate key pairs
            const keyPair = ec.genKeyPair({
                entropy: hashHex,
                entropyEnc: 'hex',
            });

            // Get and return the public and private keys in hexadecimal format
            const privateKey = keyPair.getPrivate('hex');
            const publicKey = keyPair.getPublic('hex');
            //console.log("Private Key:", privateKey);
            //console.log("Public Key:", publicKey);
            return {
                privateKey: privateKey,
                publicKey: publicKey
            };
        });
}
```

Figure 20: login.jinja and same function in signup.jinja

```
                generateECCKeyPairFromKey($("#password").val())
                    .then(keyPair => {
                        console.log("[DEBUG] Private Key:", keyPair.privateKey);
                        console.log("[DEBUG] Public Key:", keyPair.publicKey);

                        // get the publickey
                        const publicKey = keyPair.publicKey;
                        const privateKey = keyPair.privateKey;

                        // sned the public key as data to the backend
                        axios.post('/upload_public_key', {
                            username: $("#username").val(),
                            publicKey: publicKe=y
                        })
                        .then(response => {
                            console.log(response.data);

                            // Store private key to local storage
                            localStorage.setItem($("#username").val(), privateKey);

                            var retrievedValue = localStorage.getItem($("#username").val());
                            console.log("[DEBUG]: Storing private key of this guy: ",$("#username").val());
                            console.log("[DEBUG]: The private key stored in localStorage: ",retrievedValue);
                        })
                        .catch(error => {
                            console.error('[DEBUG] Error uploading public key:', error);
                        });
                    })
                    .catch(error => {
                        console.error("[DEBUG] Error generating key pair:", error);
                    });

            }
```

Figure 21: login.jinja and same process in signup.jinja

```
# Public key receive and store

@app.route('/upload_public_key', methods=['POST'])
def upload_public_key():
    username = request.json['username']
    public_key = request.json['publicKey']

    # GET PUBLIC KEY FROM CLIENT
    # store it into database
    print(f"[DEBUG] Received {username}'s {public_key}")
    db.insert_public_key(username,public_key)
    return 'Public key received successfully'
```

(a) upload_public_key

```
def insert_public_key(username: str, public_key: str):
    with Session(engine) as session:
        # create a message instance
        PublicKey = PublicKeys(user_name = username, public_key = public_key)

        # add the instance to session
        session.add(PublicKey)

        # commit the session to the database
        try:
            session.commit()
            print(f"[DEBUG]: PublicKey added: {username}: {public_key}")
        except Exception as e:
            # if error, rollback
            session.rollback()
            print(f"[DEBUG]: Failed to insert PublicKey: {e}")
        finally:
            # close the session
            session.close()
```

(b) insert_public_key

Figure 22: Upload and Store public key

Table: public_keys

	user_name	public_key
	Filter	Filter
1	a	04c45a0c57cc02e78dfc2d5a38a1bd7e51b7597c86a2712...
2	b	0468708325f09b2718da07dac0aad7688661bead51bbf0...

Figure 23: main.db Table: public_keys

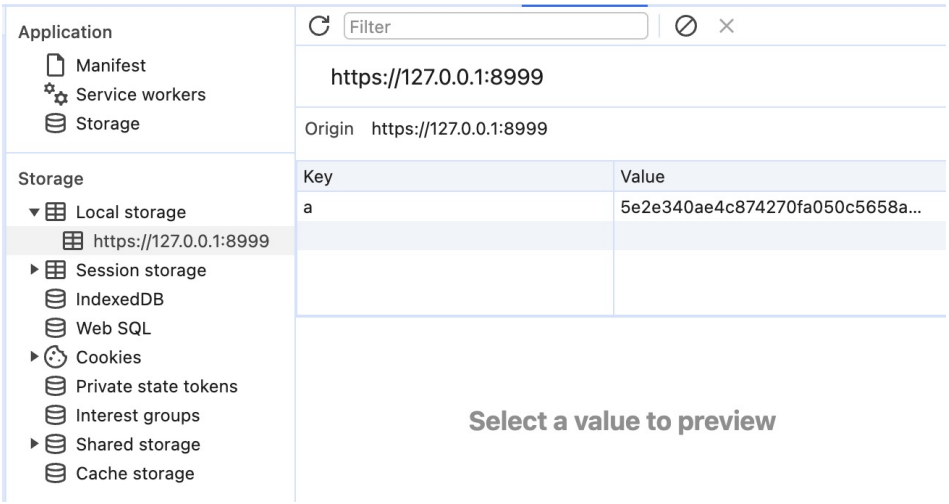


Figure 24: main.db Table: public_keys

2. When a user enters a room shown in 25, the function `getPublicKey(receiver);` shown in 26 called to request the public key of the conversing user from the server . The server retrieves the public key from the database and returns it shown in 27, storing it in a global variable `current_receiver_public_key` in `home.jinja`.

   When a user sends a message using the `send` function, they first generate a digital signature, as shown in Figure 28, using their own private key in local storage and the recipient's public key, previously obtained through `getPublicKey(receiver);` and stored in the global variable in `home.jinja`. The digital signature is then incorporated into the message itself before it is sent.

   Following this, the function computes a shared key between two users from hexadecimal representations of a private key and a public key, as shown in Figure 29. It converts the private key into a key pair object and the public key into a public key object[1]. A shared key is then derived using these keys, serving as a secure basis for encrypted communication between the two users. Despite the involvement of individual private keys and the corresponding public key, both parties arrive at the same shared secret key,also know as Diffie-Hellman key exchange [2]. Thus, we successfully obtain a shared key that can be used for encrypting and decrypting messages, having exchanged only the public keys.

```
398     // we emit a join room event to the server to join a room
399     function join_room(receiverUsername) {
400         let receiver = receiverUsername || $("#receiver").val();
401         leave();
402         // pass in the receiver of our message to the server
403         // as well as the current user's username
404         getPublicKey(receiver);
405 
406         socket.emit("join", username, receiver, (res) => {
407             console.log('in joining a room')
408             // res is a string with the error message if the error occurs
409             // this is a pretty bad way of doing error handling, but watevs
410             if (typeof res != "number") {
411                 alert(res);
412                 return;
413             }
414 
415             // set the room id variable to the room id returned by the server
416             room_id = res;
417             Cookies.set("room_id", room_id);
418 
419             // now we'll show the input box, so the user can input their message
420             $("#chat_box").hide();
421             $("#input_box").show();
422 
423             socket.emit("GetHistoryMessages", username, receiver); // get the history message
424         });
425 
426 
427     }
```

Figure 25: home.jinja join_room

```
358     function getPublicKey(username) {
359         axios.post('/getPublicKey', {
360             username: username
361         })
362         .then(function (response) {
363             if (response.data.public_key) {
364                 //console.log('[DEBUG]: Get Receiver ', username,' public key: ',response.data.public_key);
365                 current_receiver_public_key = response.data.public_key;
366             } else if (response.data.error) {
367                 // The backend returned an error message
368                 console.error('[DEBUG]: Error:', response.data.error);
369             } else {
370                 // The backend response format does not match the expected format
371                 console.error('[DEBUG]: Unexpected response format:', response.data);
372             }
373         })
374         .catch(function (error) {
375             if (error.response) {
376                 // The request has been sent, amnd the server responded with a status code
377                 console.error('Error:', error.response.data);
378                 console.error('Status:', error.response.status);
379             } else if (error.request) {
380                 //No response received
381                 console.error('No response received:', error.request);
382             } else {
383                 // Something happend while setting up the request, triggering an error
384                 console.error('Error:', error.message);
385             }
386         });
387     }
```

Figure 26: home.jinja get_public_key

```
233 @app.route('/getPublicKey', methods=['POST'])
234 def get_public_key():
235 
236     data = request.get_json()
237     username = data.get('username')
238 
239     if not username:
240         return jsonify({"error": "Missing or empty username parameter"}), 400
241     try:
242         public_key = db.get_public_key(username)
243         if public_key:
244             return jsonify({"public_key": public_key})
245         else:
246             # can not find public key in db
247             return jsonify({"error": "Public key not found"}), 404
248     except Exception as e:
249         return jsonify({"error": str(e)}), 500
250 
```

(a) app.py get_public_key *line 230 - 246*

```
151 def get_public_key(username: str):
152     with Session(engine) as session:
153         try:
154             # query the database to retrieve the public key for the given username
155             public_key = session.query(PublicKeys).filter_by(user_name=username).first()
156             if public_key:
157                 return public_key.public_key
158             else:
159                 return None
160         except Exception as e:
161             print(f"[DEBUG]: Failed to retrieve PublicKey: {e}")
162             return None
163         finally:
164             session.close()
```

(b) db.py get_public_key *line 151 - 164*

Figure 27: Get and return public key

```
299     // sign the message
300     function signMessage(message, privateKey) {
301         const key = ec.keyFromPrivate(privateKey, 'hex');
302         const msgHash = CryptoJS.SHA256(message).toString();
303         const signature = key.sign(msgHash, 'hex');
304         return signature.toDER('hex');
305     }
306
```

Figure 28: home.jinja signMessage *line 299 - 305*

```
234 const ec = new elliptic.ec('p256');
235
236 /**
237  * Calculate shared key
238  *
239  * @param {String} privateKeyHex Private key of the current user(in hexadecimal string)
240  * @param {String} publicKeyHex Public key of another user(hexadecimal string)
241  * @returns {String} Hexadecimal string of the shared key
242  */
243
244 function computeSharedKeyFromHex(privateKeyHex, publicKeyHex) {
245     // Convert the private key of the current user from a hexadecimal string to a key pair
246     const ownKeyPair = ec.keyFromPrivate(privateKeyHex, 'hex');
247
248
249     // Convert the public key of another user from a hexadecimal string to a public key object
250     // Note: The public key needs to start with '04', indicating it is an uncompressed public key
251     const otherPublicKey = ec.keyFromPublic(publicKeyHex, 'hex').getPublic();
252
253     // calculate sharedKey
254     const sharedKey = ownKeyPair.derive(otherPublicKey).toString(16);
255
256     // print the sharedKey
257     console.log('[DEBUG]: Shared Key:', sharedKey);
258
259 return sharedKey;
260 }
```

Figure 29: home.jinja ComputeSharedKeyFromHex *line 234 - 260*

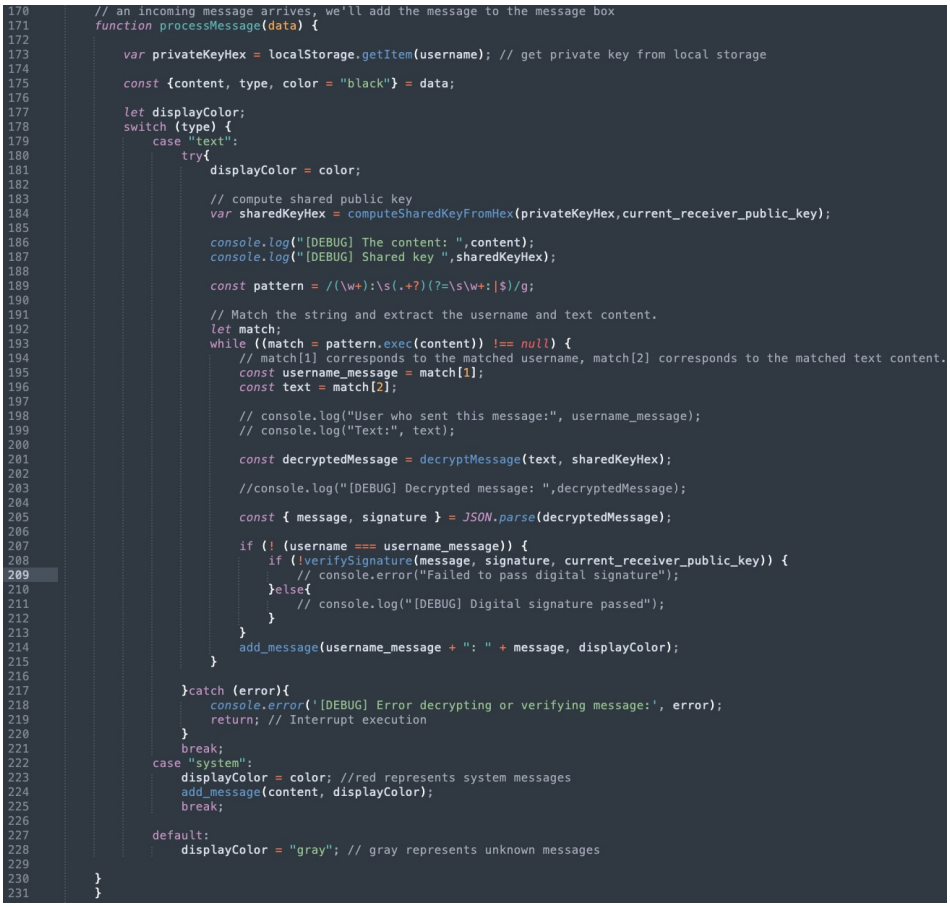
3. Subsequently, the `encryptMessage` function is called to encrypt the signed message using the AES algorithm through the CryptoJS library, as shown in 30. Initially, the shared key is converted from a hexadecimal string into the format required by the library. Then, the message is encrypted using a specified encryption mode and padding method. Finally, the encrypted, signed message is returned in string form and sent. The server only knows the public key and the encrypted message; thus, even if an attacker gains access to the server, they cannot decipher the message without knowing the user's private key.

```
272 function encryptMessage(message, sharedKeyHex) {
273     // Convert the shared key from a hexadecimal string to a WordArray, as required by crypto-js
274     const key = CryptoJS.enc.Hex.parse(sharedKeyHex);
275
276     // encrypt the messages
277     const encrypted = CryptoJS.AES.encrypt(message, key, {
278         mode: CryptoJS.mode.ECB,
279         padding: CryptoJS.pad.Pkcs7
280     });
281
282     // Return the string representation of the ciphertext
283     return encrypted.toString();
284 }
```

Figure 30: home.jinja encryptMessage *line 260 - 272*

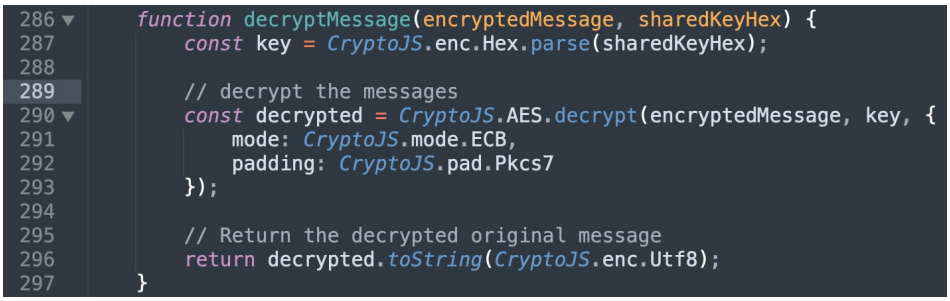
4. When the recipient receives the encrypted message, the `incoming` event first calls the `processMessage` function, as shown in Figure 31, to decrypt and verify the data. Similar to the previously mentioned process, a shared key is calculated using the sender's public key and the recipient's private key. The

message is then decrypted using the `decryptMessage` function, as shown in Figure 32. Following decryption, the sender's public key is used to verify the signature, as shown in Figure 33. If all checks are successful, the message is displayed in the message box.The process is shown in Figure 34.



```
// an incoming message arrives, we'll add the message to the message box
function processMessage(data) {

    var privateKeyHex = localStorage.getItem(username); // get private key from local storage

    const {content, type, color = "black"} = data;

    let displayColor;
    switch (type) {
        case "text":
            try{
                displayColor = color;

                // compute shared public key
                var sharedKeyHex = computeSharedKeyFromHex(privateKeyHex,current_receiver_public_key);

                console.log("[DEBUG] The content: ",content);
                console.log("[DEBUG] Shared key ",sharedKeyHex);

                const pattern = /(\w+):\s(.+?)(?=\s\w+:|$)/g;

                // Match the string and extract the username and text content.
                let match;
                while ((match = pattern.exec(content)) !== null) {
                    // match[1] corresponds to the matched username, match[2] corresponds to the matched text content.
                    const username_message = match[1];
                    const text = match[2];

                    // console.log("User who sent this message:", username_message);
                    // console.log("Text:", text);

                    const decryptedMessage = decryptMessage(text, sharedKeyHex);

                    //console.log("[DEBUG] Decrypted message: ",decryptedMessage);

                    const { message, signature } = JSON.parse(decryptedMessage);

                    if (! (username === username_message)) {
                        if (!verifySignature(message, signature, current_receiver_public_key)) {
                            // console.error("Failed to pass digital signature");
                        }else{
                            // console.log("[DEBUG] Digital signature passed");
                        }
                    }
                    add_message(username_message + ": " + message, displayColor);
                }

            }catch (error){
                console.error('[DEBUG] Error decrypting or verifying message:', error);
                return; // Interrupt execution
            }
            break;
        case "system":
            displayColor = color; //red represents system messages
            add_message(content, displayColor);
            break;

        default:
            displayColor = "gray"; // gray represents unknown messages

}
}
```

Figure 31: home.jinja processMessage *line 260 - 272*



```
function decryptMessage(encryptedMessage, sharedKeyHex) {
    const key = CryptoJS.enc.Hex.parse(sharedKeyHex);

    // decrypt the messages
    const decrypted = CryptoJS.AES.decrypt(encryptedMessage, key, {
        mode: CryptoJS.mode.ECB,
        padding: CryptoJS.pad.Pkcs7
    });

    // Return the decrypted original message
    return decrypted.toString(CryptoJS.enc.Utf8);
}
```

Figure 32: home.jinja decryptMessage *line 260 - 272*

```
307     // verify the signature
308     function verifySignature(message, signature, publicKey) {
309         const key = ec.keyFromPublic(publicKey, 'hex');
310         const msgHash = CryptoJS.SHA256(message).toString();
311         return key.verify(msgHash, signature);
312     }
```

Figure 33: home.jinja decryptMessage *line 260 - 272*

(a) sign and encrypt message

(b) verify signature and decrpt message

Figure 34: encryption and decryption process

5. The above process combines both symmetric and asymmetric encryption and utilizes digital signatures for message authentication. Through the Diffie-Hellman key exchange [2], we enable two users to obtain the same shared key by only exchanging public keys. This shared key is then used to encrypt messages, with the server merely acting as an intermediary. Both encryption and decryption are completed on the client side.

6

As described in Section 5, the client sends an encrypted message to the server using a key derived from the sender's private key and the recipient's public key. The key pair is generated based on the user's password, and this process is executed on the frontend. Thus, the server never knows the shared key used for encryption/decryption, only storing the signed and encrypted messages, as shown in Figure 35. Upon entering a room, users send a request to the server,as shown in figure 36 which then retrieves the corresponding encrypted messages from the database 37. The frontend gets messages,computes the shared key, decrypts, and authenticates the signature, as shown in 38. If the verification is successful, the historical messages are displayed in the message box.

Table: messages

	id	room_id	sender	content
	...	Filter	Filter	Filter
1	1	1	a	3hrrcRilELu9CP3HE5ptjG/...
2	2	2	a	NaykHOVY7ZCA1us7vO335z9gR2Iz7JTjlJgXemmyCx0ygb...
3	3	2	a	WBLY92cdGRNC4KTBRX5hHCuEzL5KIZPrzE0FnIDlZEwKa...

Figure 35: main.b *Table: messages*

```
418
419             // now we'll show the input box, so the user can input their message
420             $("#chat_box").hide();
421             $("#input_box").show();
422
423             socket.emit("GetHistoryMessages", username, receiver); // get the history message
424         });
425
```

Figure 36: `home.jinja` send a request to get history message

```
135 @socketio.on("GetHistoryMessages")
136 @authenticated_only
137 def GetHisoryMessages(sender_name, receiver_name):
138     room_id_stored = db.find_room_id_by_users(sender_name, receiver_name)
139     if room_id_stored:
140         messages_list = []
141
142         for e in db.get_messages_by_room_id(room_id_stored):
143             message_content = f"{e[0]}: {e[1]}"
144             messages_list.append({
145                 "content": message_content,
146                 "color": "black",
147                 "type": "text"
148             })
149
150         emit("incoming_messages_list", {"messages": messages_list}, to=request.sid)
```

Figure 37: `soucket_routes.py` return history messages

```
389     // Listen for the history message list from the server
390     socket.on("incoming_messages_list", function(data) {
391         data.messages.forEach(message => {
392             // Call the processMessage function for each historical message
393             processMessage(message);
394         });
395     });
```

Figure 38: `home.jinja` invoke `processMessage` to decrypt and show messages

Part: Additional Criteria:

1

When signing up, after checking if the user has already signed up, the server will generate a random salt and hash the password with this salt. Finally, it stores the username, salt, and hashed password in the database.

```python
# inserts a user to the database
def insert_user(username: str, password: str):
    with Session(engine) as session:
        salt = gensalt()
        hashed_password = hashpw(password.encode('utf-8'), salt)

        user = User(username=username, password=hashed_password,salt=salt)

        session.add(user)
        session.commit()

```

Figure 39: `db.py insert_user()`

	username	salt	password
	Filter	Filter	Filter
1	a	$2b$12$fME/2FrWEdFHlC7KEAX27.	$2b$12$fME/2FrWEdFHlC7KEAX27./...
2	b	$2b$12$H5HjuJ1BqghUNOogEcapce	$2b$12$H5HjuJ1BqghUNOogEcapcemqPrBlgrnxxtyBdVu...

Figure 40: `main.db Table: user`

2

Https:

In order to implement https to make our website more secure, we first create our own SSL certificate called myCA, then we use our own SSL certificate to create a CA-signed certificate called server for our messaging website. Then we tried adding our self-created certificate to the certificate manager to make the browser trust the HTTPS encryption of the localhost. However we encountered a SAN(Subject Alternative Name) issue. To solve this problem, we used a san.cnf file to re-edit the CA-signed certificate. After updating and reinstalling it into the certificate manager, we finally achieved the https encryption for localhost(shown in Figure 41 Figure 42). We also insures that the user can only visit our website by https(shown in Figure 43), and there won't appear any browser warnings since the website is secure and trusted by the browser.

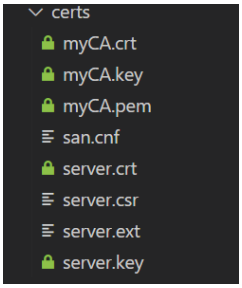


Figure 41: all the certificates used in the project

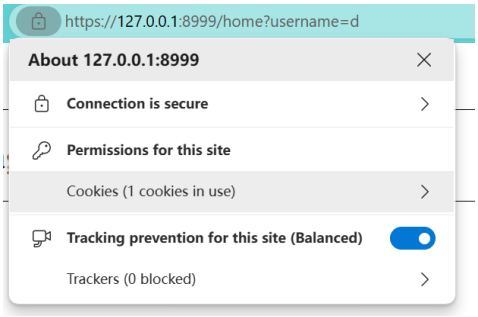


Figure 42: the connection is secure by using https

```
29  app.config['SESSION_TYPE'] = 'filesystem'  # session store in session_files
30  app.config['SESSION_FILE_DIR'] = 'session_files'
31  app.config['SESSION_PERMANENT'] = False
32  app.config['SESSION_USE_SIGNER'] = True  # signature of session
33  app.config['SESSION_COOKIE_SECURE'] = True  # can only send cookie in HTTPS
34  app.config['SESSION_COOKIE_HTTPONLY'] = True  # JavaScript cannot visit cookie
35  app.config['SESSION_COOKIE_SAMESITE'] = 'Lax'  # CSRF Protection
36
```

Figure 43: https only

3

By using the Flask Session library and setting a series of parameters, we have authenticated requests, as shown in the code in Figure 44. Below is an explanation of the code:

- **SESSION_TYPE = 'filesystem'**: This setting specifies that the session data will be stored on the server's file system.
- **SESSION_FILE_DIR = 'session_files'**: This specifies the directory where session files will be stored.
- **SESSION_PERMANENT = False**: This configuration indicates that the session is not permanent, meaning the session will be cleared after the browser is closed.
- **SESSION_USE_SIGNER = True**: By enabling this, Flask will sign the session data with the application's secret key.
- **SESSION_COOKIE_SECURE = True**: This setting ensures that cookies can only be transmitted over secure (HTTPS) connections.
- **SESSION_COOKIE_HTTPONLY = True**: This makes session cookies inaccessible to JavaScript running within the browser.
- **SESSION_COOKIE_SAMESITE = 'Lax'**: The 'SameSite' attribute of the session cookie is set to 'Lax', which helps mitigate cross-site request forgery (CSRF) attacks.

```python
from flask_session import Session  #Session
# Flask application configuration
app.config['SESSION_TYPE'] = 'filesystem'  # session store in session_files
app.config['SESSION_FILE_DIR'] = 'session_files'
app.config['SESSION_PERMANENT'] = False
app.config['SESSION_USE_SIGNER'] = True  # signature of session
app.config['SESSION_COOKIE_SECURE'] = True  # can only send cookie in HTTPS
app.config['SESSION_COOKIE_HTTPONLY'] = True  # JavaScript cannot visit cookie
app.config['SESSION_COOKIE_SAMESITE'] = 'Lax'  # CSRF Protection

Session(app)
```

Figure 44: Session settings

In addition, we have defined the decorators `authenticated_only` and `login_required`, as shown in 45, to verify if user sessions in the Flask application have been authenticated. These decorators are applied to functions that require a user to be logged in before execution, such as `home` in `app.py` and `connect`, `send` in `socket_routes.py`. Moreover, in the `home` function in `app.py`, we prevent attackers from accessing data of other users by altering the URL with `username=`. This is achieved by verifying the presence of the current user in the session, as illustrated in Figures 46 and 47.

```python
def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'username' not in session:
            # If user not login, redirect to the first page
            return redirect(url_for('login'))
            # return jsonify({"error": "Unauthorized"}), 401
        return f(*args, **kwargs)
    return decorated_function
```

(a) login_required

```python
def authenticated_only(f):
    @wraps(f)
    def wrapped(*args, **kwargs):

        if 'username' not in session:
            disconnect()
            return
        else:
            return f(*args, **kwargs)
    return wrapped
```

(b) authenticated_only

Figure 45: login_required and authenticated_only

```python
# home page, where the messaging app is
@app.route("/home")
@login_required
def home():
    if request.args.get("username") is None:
        abort(404)
    requested_username = request.args.get("username")

    # Verify the user name in session , if it same as the request one
    if requested_username != session.get('username'):
        # if inconsistent, return an error and redirect to another page
        abort(403)  # Forbidden access
    return render_template("home.jinja", username=request.args.get("username"))
```

Figure 46: forbidden_access

```python
124  # home page, where the messaging app is
125  @app.route("/home")
126  @login_required
127  def home():
128      if request.args.get("username") is None:
129          abort(404)
130      requested_username = request.args.get("username")
131
132      # Verify the user name in session , if it same as the request one
133      if requested_username != session.get('username'):
134          # if inconsistent, return an error and redirect to another page
135          abort(403)  # Forbidden access
136      return render_template("home.jinja", username=request.args.get("username"))
137
```

Figure 47: `app.py home`

3 Contribution

Zirui Zhou:

- **User login**
- **Add friends**
- **Friend requests**
- **HTTPS**

Mingyuan Ba:

- **Chatroom**
- **Messagehistory**
- **Hash and salt**
- **authentication**

Based on the Marking description on Canvas

References

[1] Tabnine. Elliptic javascript and node.js code examples. Online, 2024.

[2] Alexander S. Gillis. Diffie-hellman key exchange. Online, 2024.