1 Introduction

This assignment is actually much better than Comp2017's 0.O

2 Approach explanation

Part: Basic Functionality Requirements:

1

Basically provided by template...

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 1) , then we use a function called get friends for user, to get all the users who this user if friend with.(shown in Figure 2) . 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 3).

3

The user can send a friend request to the other user by entering the username(shown in Figure 4). We designed a class called FriendRequest to implement this feature (shown in Figure 5). 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 friends6. If so, the friend request will not be sent.

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 7) . Once the receiver accept the friendship request, the status will be updated 8 , 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 9 Figure 10) , so it will update the friend requests automatically without manual refresh.

5

All the friends of the user is displayed as buttons in the friend list (shown in Figure 11). 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 online 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 12). Then we use this method when trying to send messages (shown in Figure 13).

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 14.

   As shown in Figure 15, 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 ECC key pair, as demonstrated in the cryptographic process.

   Subsequently, the frontend stores both the public key and the private key as shown in 16. The public key is uploaded to the server using an axios request and stored in a database called public_keys, as shown in Figures 17 and 18. Meanwhile, the private key is retained in the user's local storage, as shown in Figure 19.. These keys are generated based on the user's password, ensuring that they are consistently the same each time they are generated.

```python
class Friendship(Base):
    __tablename__ = 'friendship'
    user_username = Column(String,primary_key=True)
    friend_username = Column(String,primary_key=True)
```

Figure 1: 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 2: 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 3: home.jinja fetch friends function

Figure 4: 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 5: 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 6: db.py send friend request to the other user

Figure 7: 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 8: db.py: update request status after user’s operation

```
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 9: home.jinja: fetch friendrequests and socket

```
@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 10: socket routes.py: listen to the frontend

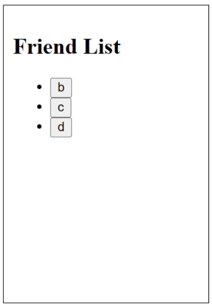

Figure 11: home page: friends are displayed as buttons

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

Figure 12: models.py: check how many users are in the room

```python
@socketio.on("send")
@authenticated_only
def send(username, message, room_id):
    users_in_room = room.get_users_in_room(room_id)

    if len(users_in_room) < 2:
        emit("error", {"message": "2 users both need to be online"}, to=request.sid)
        return

    emit("incoming", {
        "content": f"{username}: {message}",
        "color": "black",
        "type": "text"
    }, to=room_id)

    # include the message type when inserting a message into the database
    db.insert_message(room_id, username, message)

    return

```

Figure 13: socket routes.py: check whether both users are online before sending message

```
[DEBUG] Private Key:                                                              login:119
5e2e340ae4c874270fa050c5658a523651ac150e814797daf5d2f013876cedda
[DEBUG] Public Key:                                                               login:120
04c45a0c57cc02e78dfc2d5a38a1bd7e51b7597c86a27127945e34050144ff862c48687ff8a6f45980f98f1081
a100e820b0d4b507b0256200a242212744812bec
Public key received successfully                                                  login:132
[DEBUG]: Storing private key of this guy:  a                                      login:138
[DEBUG]: The private key stored in localStorage:                                  login:139
5e2e340ae4c874270fa050c5658a523651ac150e814797daf5d2f013876cedda
```

Figure 14: Illustration of private and public keys

```javascript
        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 15: login.jinja *line 16-54* and same function in signup.jinja *line 27 - 65*

```javascript
97                  generateECCKeyPairFromKey($("#password").val())
98                      .then(keyPair => {
99                          console.log("[DEBUG] Private Key:", keyPair.privateKey);
100                         console.log("[DEBUG] Public Key:", keyPair.publicKey);
101
102                         // get the publickey
103                         const publicKey = keyPair.publicKey;
104                         const privateKey = keyPair.privateKey;
105
106                         // sned the public key as data to the backend
107                         axios.post('/upload_public_key', {
108                             username: $("#username").val(),
109                             publicKey: publicKe=y
110                         })
111                         .then(response => {
112                             console.log(response.data);
113
114                             // Store private key to local storage
115                             localStorage.setItem($("#username").val(), privateKey);
116
117                             var retrievedValue = localStorage.getItem($("#username").val());
118                             console.log("[DEBUG]: Storing private key of this guy: ",$("#username").val());
119                             console.log("[DEBUG]: The private key stored in localStorage: ",retrievedValue);
120                         })
121                         .catch(error => {
122                             console.error('[DEBUG] Error uploading public key:', error);
123                         });
124                     })
125                     .catch(error => {
126                         console.error("[DEBUG] Error generating key pair:", error);
127                     });
128
129             }
```

Figure 16: login.jinja *line 95-129* and same process in signup.jinja *line 84 - 118*

```python
# 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

```python
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 17: Upload and Store public key

Table: public_keys

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

Figure 18: main.db Table: public_keys

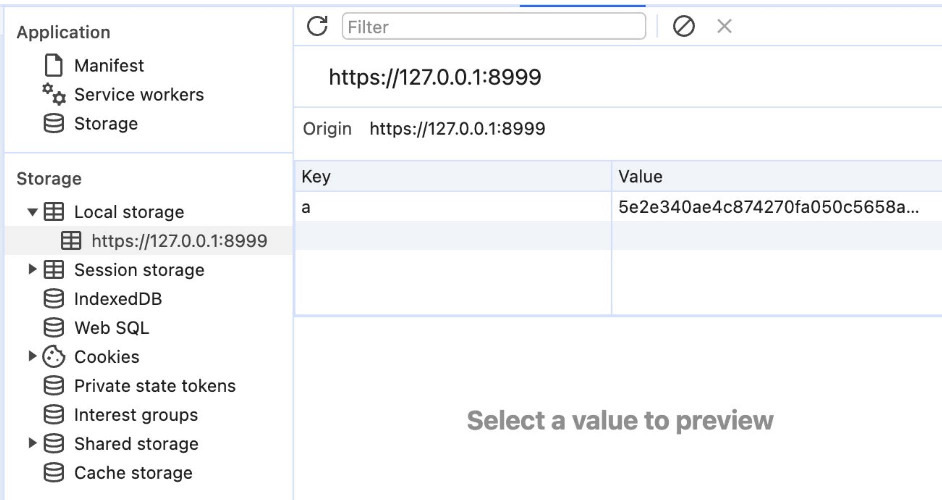

Figure 19: main.db Table: public_keys

2. When a user enters a room shown in 20, the function `getPublicKey(receiver);` shown in 21 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 22, storing it in a global variable in `home.jinja`.

   When a user sends a message using the `send` function, they first generate a digital signature, as shown in Figure 23, 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`. This process utilizes Elliptic Curve Cryptography (ECC) to ensure the message's integrity and authentication. 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 using Elliptic Curve Cryptography (ECC) from hexadecimal representations of a private key and a public key, as shown in Figure 24. It converts the private key into a key pair object and the public key into a public key object. 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. Thus, we successfully obtain a shared key that can be used for encrypting and decrypting messages, having exchanged only the public keys.

```javascript
    // we emit a join room event to the server to join a room
    function join_room(receiverUsername) {
        let receiver = receiverUsername || $("#receiver").val();
        leave();
        // pass in the receiver of our message to the server
        // as well as the current user's username
        getPublicKey(receiver);

        socket.emit("join", username, receiver, (res) => {
            console.log('in joining a room')
            // res is a string with the error message if the error occurs
            // this is a pretty bad way of doing error handling, but watevs
            if (typeof res != "number") {
                alert(res);
                return;
            }

            // set the room id variable to the room id returned by the server
            room_id = res;
            Cookies.set("room_id", room_id);

            // now we'll show the input box, so the user can input their message
            $("#chat_box").hide();
            $("#input_box").show();

            socket.emit("GetHistoryMessages", username, receiver); // get the history message
        });


    }
```

Figure 20: home.jinja join_room

```javascript
    function getPublicKey(username) {
        axios.post('/getPublicKey', {
            username: username
        })
        .then(function (response) {
            if (response.data.public_key) {
                //console.log('[DEBUG]: Get Receiver ', username,' public key: ',response.data.public_key);
                current_receiver_public_key = response.data.public_key;
            } else if (response.data.error) {
                // The backend returned an error message
                console.error('[DEBUG]: Error:', response.data.error);
            } else {
                // The backend response format does not match the expected format
                console.error('[DEBUG]: Unexpected response format:', response.data);
            }
        })
        .catch(function (error) {
            if (error.response) {
                // The request has been sent, amnd the server responded with a status code
                console.error('Error:', error.response.data);
                console.error('Status:', error.response.status);
            } else if (error.request) {
                //No response received
                console.error('No response received:', error.request);
            } else {
                // Something happend while setting up the request, triggering an error
                console.error('Error:', error.message);
            }
        });
    }
```

Figure 21: home.jinja get_public_key

```python
230 @app.route('/getPublicKey', methods=['POST'])
231 def get_public_key():
232
233     data = request.get_json()
234     username = data.get('username')
235
236     if not username:
237         return jsonify({"error": "Missing or empty username parameter"}), 400
238     try:
239         public_key = db.get_public_key(username)
240         if public_key:
241             return jsonify({"public_key": public_key})
242         else:
243             # can not find public key in db
244             return jsonify({"error": "Public key not found"}), 404
245     except Exception as e:
246         return jsonify({"error": str(e)}), 500
```

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

```python
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 22: 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 23: 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 24: 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 25. 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 25: 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 26, 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 27. Following decryption, the sender's public key is used to verify the signature, as shown in Figure 28. If all checks are successful, the message is displayed in the message box.

```javascript
// 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 26: home.jinja processMessage *line 260 - 272*

```javascript
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 27: home.jinja decryptMessage *line 260 - 272*

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

Figure 28: home.jinja decryptMessage *line 260 - 272*

5. The above process combines both symmetric and asymmetric encryption and utilizes digital signatures for message authentication. Through the ECC elliptic curve encryption algorithm, 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.

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.

```
# 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 29: db.py insert_user()

	username	salt	password
	Filter	Filter	Filter
1	a	$2b$12$fME/2FrWEdFHIC7KEAX27.	$2b$12$fME/2FrWEdFHIC7KEAX27./...
2	b	$2b$12$H5HjuJ1BqghUNOogEcapce	$2b$12$H5HjuJ1BqghUNOogEcapcemqPrBlgrnxxtyBdVu...

Figure 30: 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 31 Figure 32). We also insures that the user can only visit our website by https(shown in Figure 33), and there won't appear any browser warnings since the website is secure and trusted by the browser.

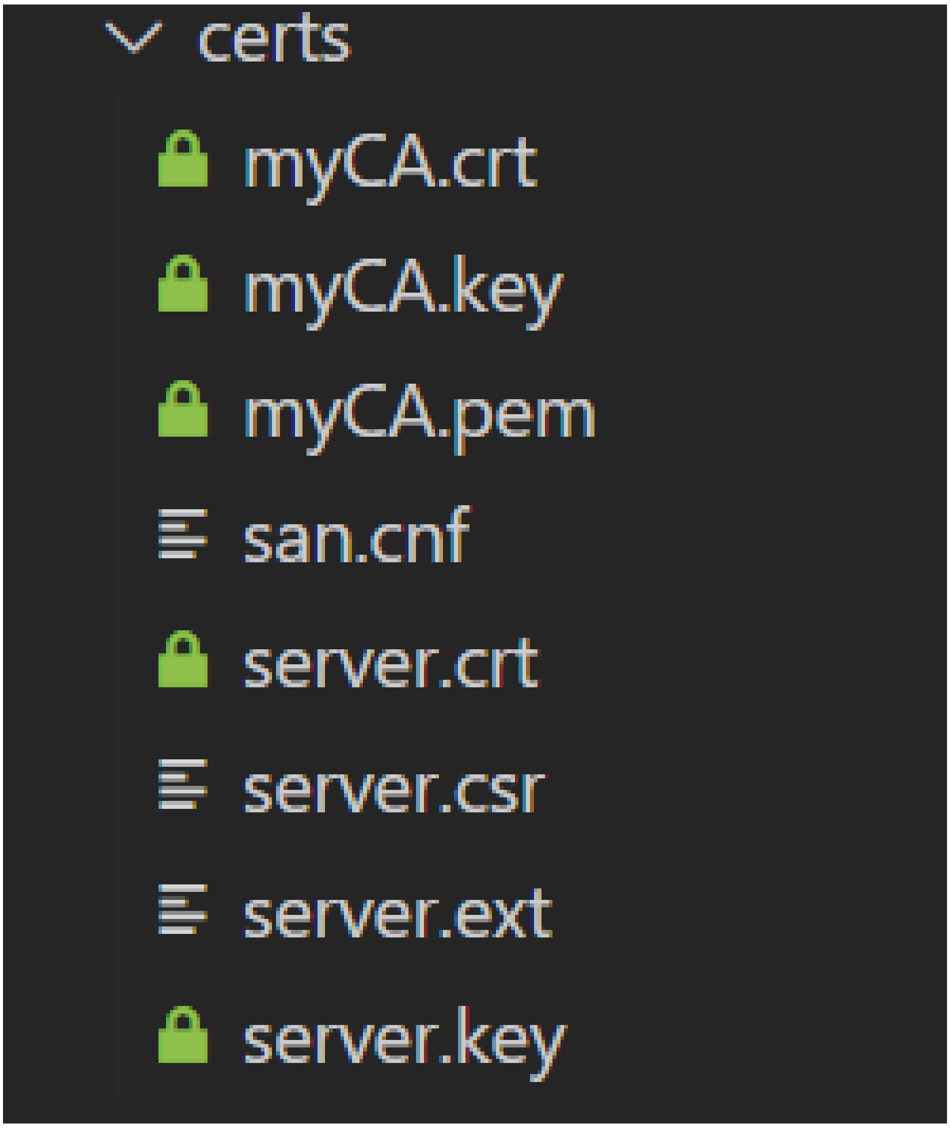

Figure 31: all the certificates used in the project

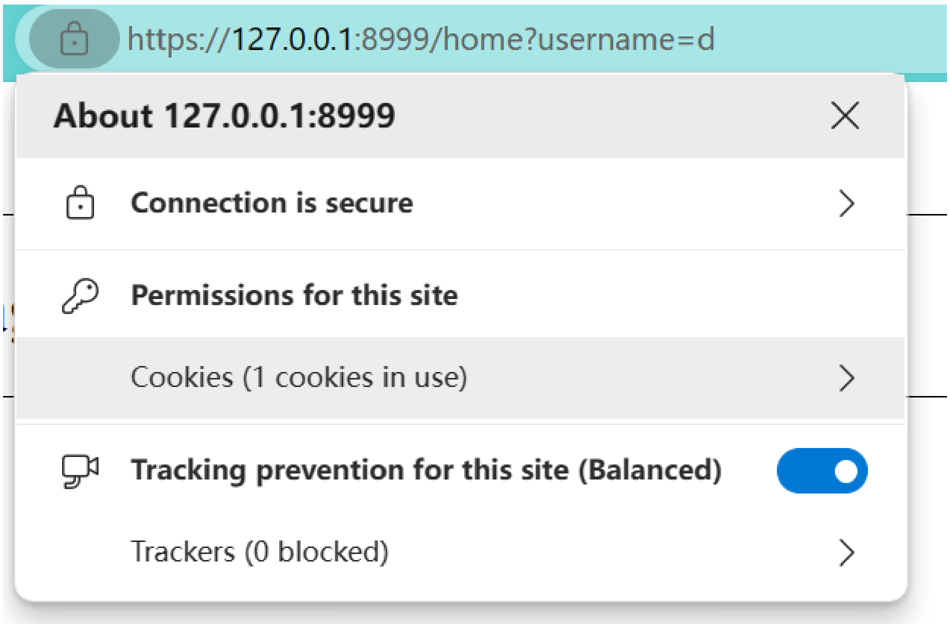

Figure 32: the connection is secure by using https

```python
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
```

Figure 33: https only

3 Contribution

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```