

1 Introduction

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

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.

```

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

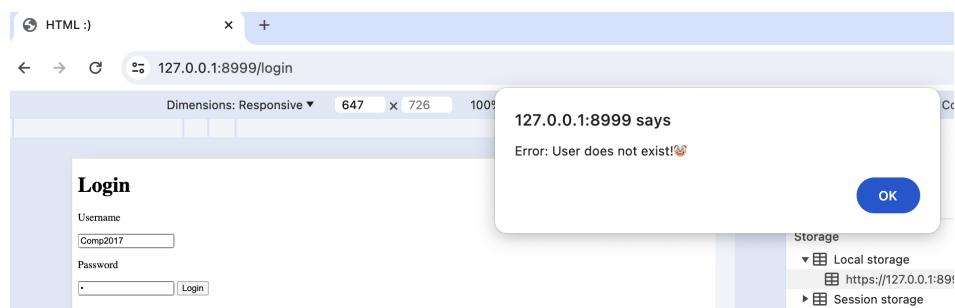


Figure 2: No such user

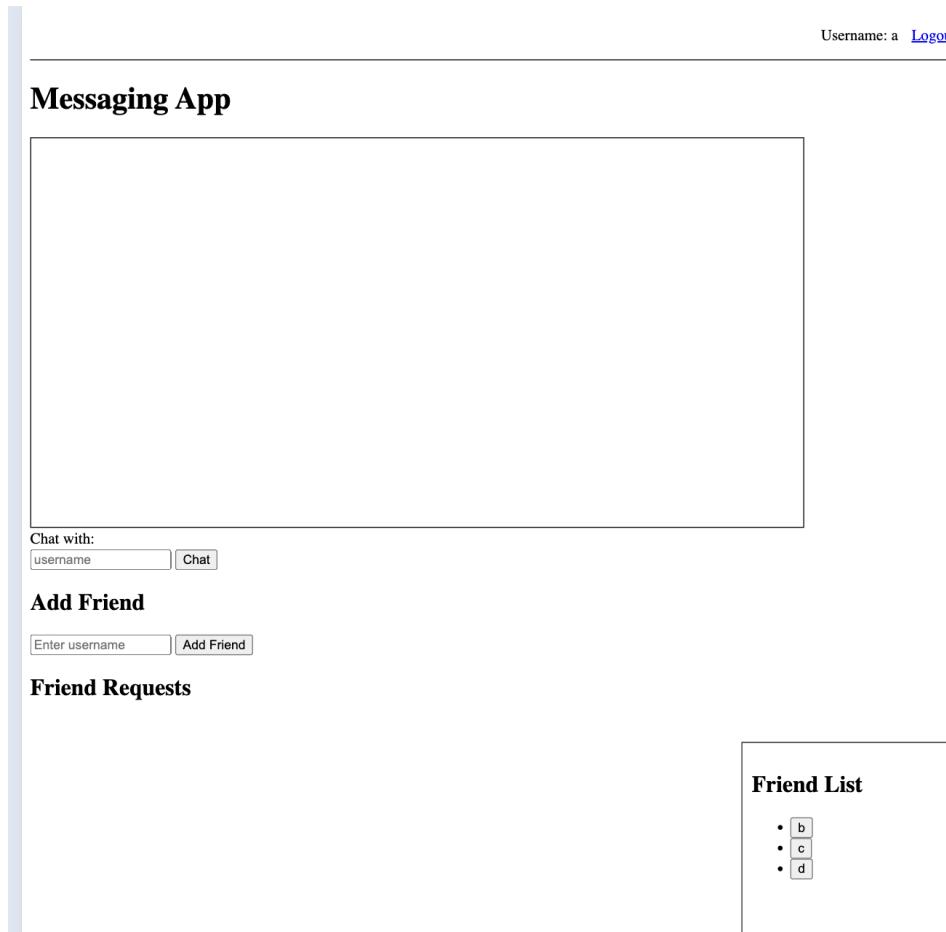


Figure 3: login successful

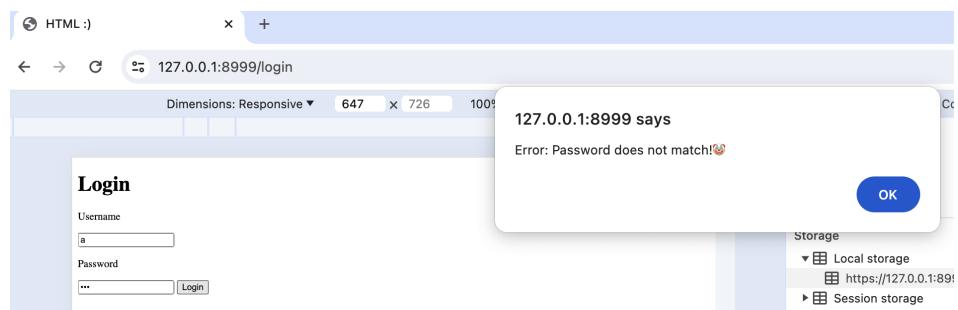


Figure 4: password doesn't match

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 5) , then we use a function called get friends for user, to get all the users who this user if friend with.(shown in Figure 6) . 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 7).

3

The user can send a friend request to the other user by entering the username(shown in Figure 8). We designed a class called FriendRequest to implement this feature (shown in Figure 9). 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 friends¹⁰. 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 11) . Once the receiver accept the friendship request, the status will be updated ¹² , 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 applied socket to listen to the friendrequest(shown in Figure 13 Figure 14) , 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 15). 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 method for the room class, which checks whether both of them are in the room.(shown in Figure 16). Then we use this method when trying to send messages (shown in Figure 17).

```

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 5: models.py define a class called friendship

```

362     def get_friends_for_user(username: str):
363         with Session(engine) as session:
364             # check friendship
365             friendships = session.query(Friendship).filter(
366                 (Friendship.user_username == username)
367             ).all()
368
369             # extract the friend's username
370             friends_usernames = [friendship.friend_username for friendship in friendships]
371             friends = []
372             for friend_username in friends_usernames:
373                 friend = session.query(User).filter(User.username == friend_username).first()
374                 if friend:
375                     friends.append({"username": friend.username})
376
377         return friends
378

```

Figure 6: db.py get all the friends of the current user

```

568     function fetchFriends() {
569         // Read the current username from the data-username attribute of the body element
570         let currentUsername = "{{ username }}";
571
572         fetch(`/get_friends?username=${currentUsername}`)
573             .then(response => {
574                 if (!response.ok) {
575                     throw new Error('Failed to fetch friends');
576                 }
577                 return response.json();
578             })
579             .then(data => {
580                 const friendList = document.getElementById('friend_list');
581                 friendList.innerHTML = '';
582
583                 // iterate through the returned friend data and create a list item for each friend
584                 data.forEach(friend => {
585                     const li = document.createElement('li');
586                     const button = document.createElement('button');
587                     button.textContent = friend['username'];
588                     button.onclick = function() {
589                         join_room(friend['username']);
590                     };
591                     li.appendChild(button);
592                     friendList.appendChild(li);
593                 });
594             })
595             .catch(error => console.error('Error fetching friends:', error));
596     }
597

```

Figure 7: home.jinja fetch friends function

Add Friend

Figure 8: home page: add friend feature

```

102     class RequestStatus(PyEnum):
103         PENDING = 'pending'
104         APPROVED = 'approved'
105         REJECTED = 'rejected'
106
107     class FriendRequest(Base):
108         __tablename__ = 'friend_request'
109         id = Column(Integer, primary_key=True)
110         sender_id = Column(String, ForeignKey('user.username'))
111         receiver_id = Column(String, ForeignKey('user.username'))
112         status = Column(String)
113

```

Figure 9: models.py define a class called friendrequest

```

235     def send_friend_request(sender_username: str, receiver_username: str):
236         print(f"sender:{sender_username}")
237         print(f"receiver:{receiver_username}")
238         with Session(engine) as session:
239             # check if the recipient exists
240             receiver = session.get(User, receiver_username)
241             print(receiver)
242             if not receiver:
243                 return "Receiver does not exist."
244
245             # check if a friend request has already been sent
246             existing_request = session.query(FriendRequest).filter(
247                 (FriendRequest.sender_id == sender_username) &
248                 (FriendRequest.receiver_id == receiver_username) &
249                 (FriendRequest.status.in_([RequestStatus.PENDING.value, RequestStatus.APPROVED.value]))
250             ).first()
251             if existing_request:
252                 return "Friend request already sent or already friends."
253
254             # create and save a new friend request
255             new_request = FriendRequest(sender_id=sender_username, receiver_id=receiver_username, status=RequestStatus.PENDING.value)
256             session.add(new_request)
257             try:
258                 session.commit()
259                 print("Friend request successfully added.")
260             except Exception as e:
261                 print(f"Failed to insert friend request: {e}")
262                 session.rollback()
263                 session.commit()
264
265             return "Friend request sent successfully."
266

```

Figure 10: db.py send friend request to the other user

Friend Requests

- To: b
- From: c

Figure 11: home page: display friend requests

```

335     def update_friend_request_status(request_id: int, new_status: str):
336         with Session(engine) as session:
337             try:
338                 # find the friend request record by ID
339                 friend_request = session.query(FriendRequest).filter(FriendRequest.id == request_id).first()
340                 if not friend_request:
341                     return False, "Friend request not found."
342
343                 # update the status
344                 friend_request.status = new_status
345                 if new_status == "approved":
346                     # check a friendship
347                     exists = session.query(Friendship).filter_by(user_username=friend_request.sender_id, friend_username=friend_request.receiver_id).first()
348                     if not exists:
349                         new_friendship1 = Friendship(user_username=friend_request.sender_id, friend_username=friend_request.receiver_id)
350                         new_friendship2 = Friendship(user_username=friend_request.receiver_id, friend_username=friend_request.sender_id)
351                         session.add(new_friendship1)
352                         session.add(new_friendship2)
353                         print(f"We are friends!!!!!!: {new_friendship1.user_username} <--> {new_friendship2.user_username}")
354
355                     session.commit()
356                     return True, "Friend request status updated successfully."
357             except SQLAlchemyError as e:
358                 session.rollback()
359                 print(f"Error updating friend request status: {e}")
360             return False, "Error occurred during the update."
361

```

Figure 12: db.py: update request status after user's operation

```

503     function fetchFriendRequests() {
504       console.log('Fetching friend requests...'); 
505       let currentUsername = "{{ username }}"; // Retrieve the current username from server-side rendering variables or from another source
506
507       fetch('/get_friend_requests?username=${currentUsername}')
508         .then(response => {
509           if (!response.ok) {
510             throw new Error('Failed to fetch friend requests');
511           }
512           return response.json();
513         })
514         .then(friendRequests => {
515           // get the container element for the friend request list
516           const friendRequestsList = document.getElementById('friend_requests');
517           // clear the current list content
518           friendRequestsList.innerHTML = '';
519
520           // iterate through all friend requests and add them to the list
521           friendRequests.forEach(request => {...});
522         })
523         .catch(error => console.error('Error fetching friend requests:', error));
524     }
525
526     socket.on('friend_request_update', function(data) {
527       console.log('Received update notification:', data.message);
528       // Call the function to update friend requests
529       fetchFriendRequests();
530       fetchFriends();
531     });
532   };

```

Figure 13: home.jinja: fetch friendrequests and socket

```

161   @socketio.on('friend_request_sent')
162   def handle_friend_request_sent(data):
163     print("Friend request sent from:", data['sender'], "to:", data['receiver'])
164     # Here you can broadcast to specific rooms or globally as needed
165     print("Emitting friend_request_update event")
166     socketio.emit('friend_request_update', {'message': 'Update your friend requests list'})
167
168

```

Figure 14: socket routes.py: listen to the frontend



Figure 15: 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 16: 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 17: socket routes.py: check whether both users are online before sending message

The process of ensuring secure communication between users:

- 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 18.

As shown in Figure 19, 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 20. The public key is uploaded to the server using an axios request and stored in a database called public_keys, as shown in Figures 21 and 22. Meanwhile, the private key is retained in the user's local storage, as shown in Figure 23.. These keys are generated based on the user's password, ensuring that they are consistently the same each time they are generated.

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

Figure 18: Illustration of private and public keys

```

20      const ec = new elliptic.ec('p256');
21
22      /**
23       * Function to generate ECC key pair from a specific key.
24       * @param {string} key - The key provided by the user, can be a password or any arbitrary string.
25       * @returns {object} An object containing the hex format public and private keys.
26      */
27      function generateECCKeyPairFromKey(key) {
28          // use the sha-256 hash function to process the key, generating a fixed-length seed
29          return window.crypto.subtle.digest('SHA-256', new TextEncoder().encode(key))
30              .then(hash => {
31                  // convert ArrayBuffer to Uint8Array
32                  const hashArray = Array.from(new Uint8Array(hash));
33                  // convert the hash value to a hexadecimal string
34                  const hashHex = hashArray.map(byte => byte.toString(16).padStart(2, '0')).join('');
35
36                  // use the hash value as a random number to generate key pairs
37                  const keyPair = ec.genKeyPair({
38                      entropy: hashHex,
39                      entropyEnc: 'hex',
40                  });
41
42                  // Get and return the public and private keys in hexadecimal format
43                  const privateKey = keyPair.getPrivate('hex');
44                  const publicKey = keyPair.getPublic('hex');
45                  //console.log("Private Key:", privateKey);
46                  //console.log("Public Key:", publicKey);
47                  return {
48                      privateKey: privateKey,
49                      publicKey: publicKey
50                  };
51              });
52      }

```

Figure 19: login.jinja line 16-54 and same function in signup.jinja line 27 - 65

```

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 pubkey
103             const publicKey = keyPair.publicKey;
104             const privateKey = keyPair.privateKey;
105
106             // send the public key as data to the backend
107             axios.post('/upload_public_key', {
108                 username: $("#username").val(),
109                 publicKey: publicKey
110             })
111             .then(response => {
112                 console.log(response.data);
113
114                 // Store private key to local storage
115                 localStorage.setItem($("#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 20: login.jinja line 95-129 and same process in signup.jinja line 84 - 118

```

217 # Public key receive and store
218
219 @app.route('/upload_public_key', methods=['POST'])
220 def upload_public_key():
221     username = request.json['username']
222     public_key = request.json['publickey']
223
224     # GET PUBLIC KEY FROM CLIENT
225     # store it into database
226     print(f"[DEBUG] Received {username}'s {public_key}")
227     db.insert_public_key(username,public_key)
228     return 'Public key received successfully'
229

```

(a) upload_public_key

```

131 def insert_public_key(username: str, public_key: str):
132     with Session(engine) as session:
133         # create a message instance
134         PublicKey = PublicKeys(user_name = username, public_key = public_key)
135
136         # add the instance to session
137         session.add(PublicKey)
138
139         # commit the session to the database
140         try:
141             session.commit()
142             print(f"[DEBUG]: PublicKey added: {username}: {public_key}")
143         except Exception as e:
144             # if error, rollback
145             session.rollback()
146             print(f"[DEBUG]: Failed to insert PublicKey: {e}")
147
148         # close the session
149         session.close()

```

(b) insert_public_key

Figure 21: Upload and Store public key

Table: public_keys	
1	a
2	b

Figure 22: main.db Table: public_keys

Key	Value
a	5e2e340ae4c874270fa050c5658a...

Figure 23: main.db Table: public_keys

2. When a user enters a room shown in 24, the function `getPublicKey(receiver)`; shown in 25 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 26, 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 27, 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 28. 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.

```

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     }

```

Figure 24: 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, and 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 happened while setting up the request, triggering an error
384                 console.error('Error:', error.message);
385             }
386         });
387     }

```

Figure 25: home.jinja get_public_key

```

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

```

(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 26: 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 27: 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     // Convert the public key of another user from a hexadecimal string to a public key object
249     // Note: The public key needs to start with '04', indicating it is an uncompressed public key
250     const otherPublicKey = ec.keyFromPublic(publicKeyHex, 'hex').getPublic();
251
252     // calculate sharedKey
253     const sharedKey = ownKeyPair.derive(otherPublicKey).toString(16);
254
255     // print the sharedKey
256     console.log('[DEBUG]: Shared Key:', sharedKey);
257
258     return sharedKey;
259 }
260

```

Figure 28: 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 29. 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 29: home.jinja encryptMessage line 260 - 272

- When the recipient receives the encrypted message, the incoming event first calls the processMessage function, as shown in Figure 30, 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 31. Following decryption, the sender's public key is used to verify the signature, as shown in Figure 32. If all checks are successful, the message is displayed in the message box. The process is shown in Figure 33.

```

170   // an incoming message arrives, we'll add the message to the message box
171   function processMessage(data) {
172
173     var privateKeyHex = localStorage.getItem(username); // get private key from local storage
174
175     const {content, type, color = "black"} = data;
176
177     let displayColor;
178     switch (type) {
179       case "text":
180         try {
181           displayColor = color;
182
183           // compute shared public key
184           var sharedKeyHex = computeSharedKeyFromHex(privateKeyHex, current_receiver_public_key);
185
186           console.log("[DEBUG] The content: ", content);
187           console.log("[DEBUG] Shared key ", sharedKeyHex);
188
189           const pattern = /(\\w+):\\s(.+?)\\s(\\w+):\\$/g;
190
191           // Match the string and extract the username and text content.
192           let match;
193           while ((match = pattern.exec(content)) !== null) {
194             // match[1] corresponds to the matched username, match[2] corresponds to the matched text content.
195             const username_message = match[1];
196             const text = match[2];
197
198             // console.log("User who sent this message:", username_message);
199             // console.log("Text:", text);
200
201             const decryptedMessage = decryptMessage(text, sharedKeyHex);
202
203             //console.log("[DEBUG] Decrypted message: ", decryptedMessage);
204
205             const { message, signature } = JSON.parse(decryptedMessage);
206
207             if (! (username === username_message)) {
208               if (!verifySignature(message, signature, current_receiver_public_key)) {
209                 // console.error("Failed to pass digital signature");
210               } else {
211                 // console.log("[DEBUG] Digital signature passed");
212               }
213             }
214             add_message(username_message + ": " + message, displayColor);
215           }
216
217         } catch (error) {
218           console.error('[DEBUG] Error decrypting or verifying message:', error);
219           return; // Interrupt execution
220         }
221         break;
222       case "system":
223         displayColor = color; // red represents system messages
224         add_message(content, displayColor);
225         break;
226
227       default:
228         displayColor = "gray"; // gray represents unknown messages
229
230     }
231   }

```

Figure 30: home.jinja processMessage line 260 - 272

```

286 ▼      function decryptMessage(encryptedMessage, sharedKeyHex) {
287          const key = CryptoJS.enc.Hex.parse(sharedKeyHex);
288
289          // decrypt the messages
290          const decrypted = CryptoJS.AES.decrypt(encryptedMessage, key, {
291              mode: CryptoJS.mode.ECB,
292              padding: CryptoJS.pad.Pkcs7
293          });
294
295          // Return the decrypted original message
296          return decrypted.toString(CryptoJS.enc.Utf8);
297      }

```

Figure 31: 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 32: home.jinja decryptMessage line 260 - 272

```

[DEBUG]: Shared Key: home?username=a:276
d5dd38b9f1956c6885fe473006659825f6fed0b2757753defca1e676c92bc7f2
[DEBUG] Encrypted message with signature: home?username=a:364
WBLY92cdGRNC4KTBRX5hHCuEzL5KIZPrzE0FnIDlZEwKaWySbt1pVgVLK3FzsufzWlXWpyp9g0Dfn9lI4Xn7hPXJB
X4Uq6kvCpp1wd4rxFkwGBYP26kw0FEa2iU1TtlnV6ni9CS9KKmManlZPMXJplxJJgokct1DpFS9IxCo6Mg+0sxqbfZ
7qx1FDq7iTAY3Aew0dkl4Nk7jXxrugPsaoWFfqauIQmUm2CHesGUUEj7gED11LW/yJE/PRpRadVY4/VteuVIM+I+Ex
qIP/pngQ==

(a) sign and encrypt message
[DEBUG] The content: a: home?username=c:205
WBLY92cdGRNC4KTBRX5hHCuEzL5KIZPrzE0FnIDlZEwKaWySbt1pVgVLK3FzsufzWlXWpyp9g0Dfn9lI4Xn7hPXJB
X4Uq6kvCpp1wd4rxFkwGBYP26kw0FEa2iU1TtlnV6ni9CS9KKmManlZPMXJplxJJgokct1DpFS9IxCo6Mg+0sxqbfZ
7qx1FDq7iTAY3Aew0dkl4Nk7jXxrugPsaoWFfqauIQmUm2CHesGUUEj7gED11LW/yJE/PRpRadVY4/VteuVIM+I+Ex
qIP/pngQ==

[DEBUG] Shared key home?username=c:206
d5dd38b9f1956c6885fe473006659825f6fed0b2757753defca1e676c92bc7f2

[DEBUG] Decrypted message: {"message": "i don't like home?username=c:222
Comp2017", "signature": "3045022046c87ebc7b6f3808a1cd51dcae1d60f80c8b66b5b8f43f2adf078fb5413
5d065022100ed2bbd4634942ab1384a5f9e3ae2b9a3431d4f0ecb04326c994cab9e9b108f30"}}

```

(b) verify signature and decrypt message

Figure 33: encryption and decryption process

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.

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, calculated using the ECC algorithm to ensure both parties can compute the same 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 34. Upon entering a room, users send a request to the server, as shown in figure 35 which then retrieves the corresponding encrypted messages from the database 36. The frontend gets messages, computes the shared key, decrypts, and authenticates the signature, as shown in 37. If the verification is successful, the historical messages are displayed in the message box.

	id	room_id	sender	content
	...	Filter	Filter	Filter
1	1	1	a	3hrrcRlELu9CP3HE5ptjG...
2	2	2	a	NaykHOVY7ZCA1us7vO335z9gR2lz7JTjlJgXemmyCx0ygb...
3	3	2	a	WBLY92cdGRNC4KTBRX5hHCuEzL5KIZPrzE0FnIDIZEwKa...

Figure 34: 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);
424     });
425

```

Figure 35: home.jinja send a request to get history message

```

135 @socketio.on("GetHistoryMessages")
136 @authenticated_only
137 def GetHistoryMessages(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 36: socket_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 37: home.jin 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.

```

27  # inserts a user to the database
28  def insert_user(username: str, password: str):
29      with Session(engine) as session:
30          salt = gensalt()
31          hashed_password = hashpw(password.encode('utf-8'), salt)
32
33          user = User(username=username, password=hashed_password,salt=salt)
34
35          session.add(user)
36          session.commit()
37

```

Figure 38: db.py insert_user()

	username	salt	password
	Filter	Filter	Filter
1	a	\$2b\$12\$fME/2FrWEEdFHIC7KEAX27.	\$2b\$12\$fME/2FrWEEdFHIC7KEAX27./...
2	b	\$2b\$12\$H5HjuJ1BqghUNOogEcapce	\$2b\$12\$H5HjuJ1BqghUNOogEcapcemqPrBlgrnxxyBdVu...

Figure 39: 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 40 Figure 41). We also insures that the user can only visit our website by https(shown in Figure 42), and there won't appear any browser warnings since the website is secure and trusted by the browser.

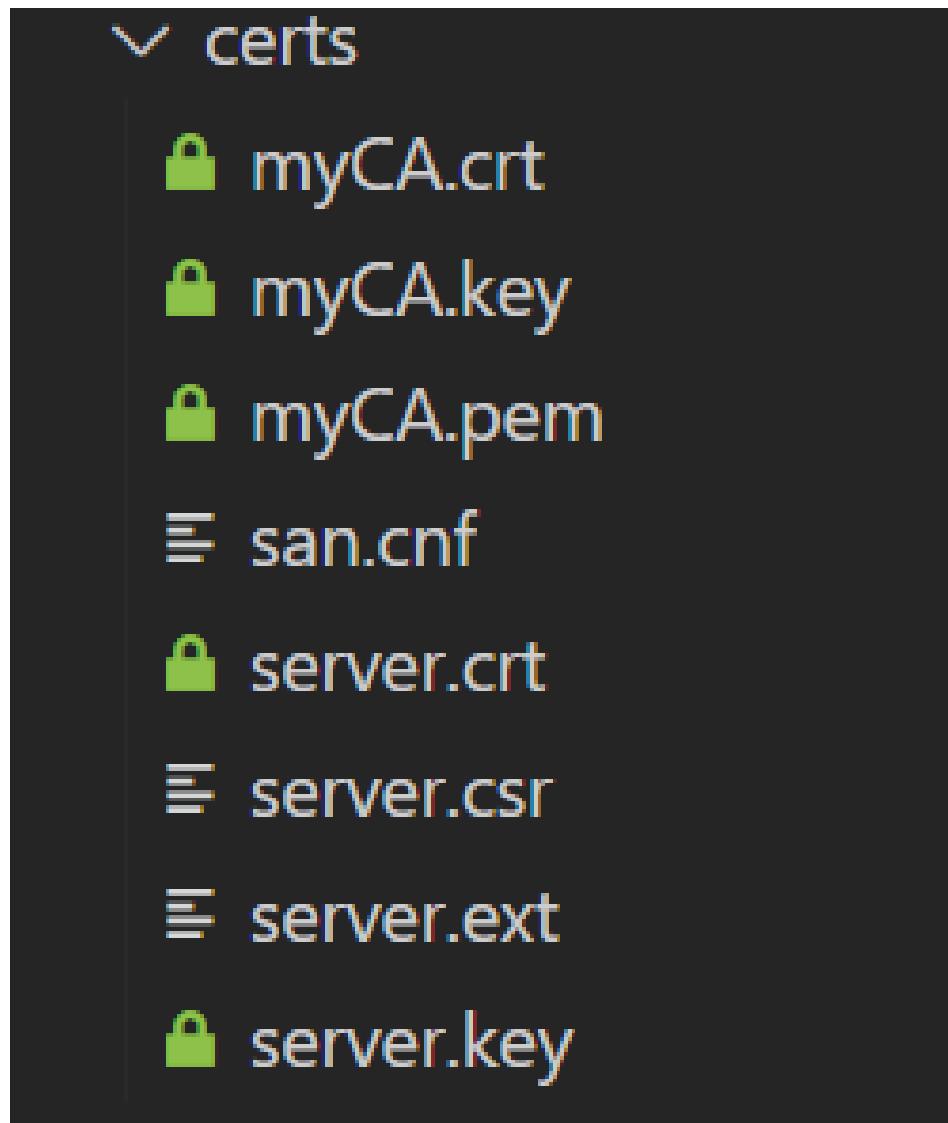


Figure 40: all the certificates used in the project

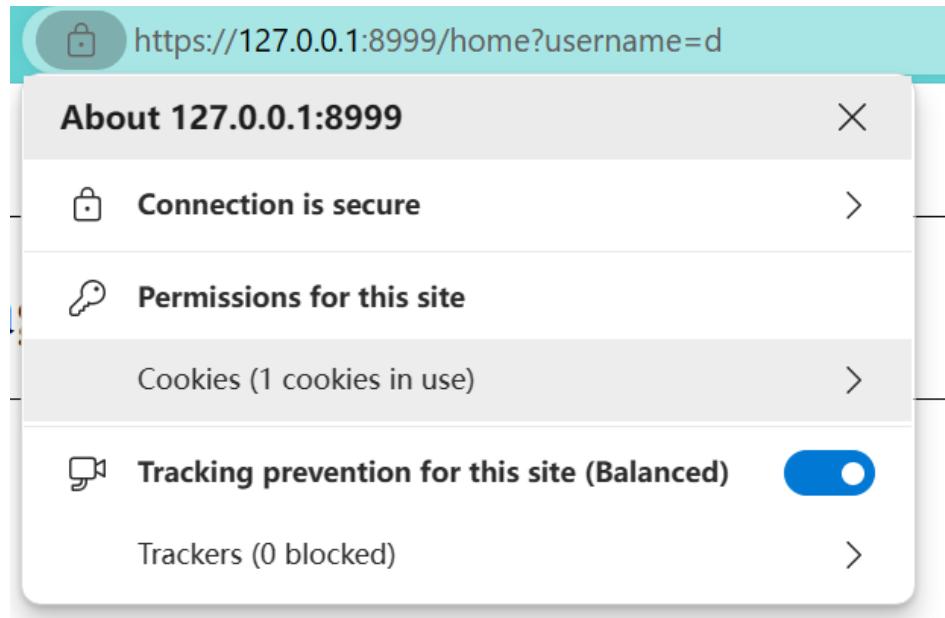


Figure 41: the connection is secure by using https

```

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 42: 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 43. 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.

```

27     from flask_session import Session #Session
28     # Flask application configuration
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
37     Session(app)

```

Figure 43: Session settings

In addition, we have defined the decorators `authenticated_only` and `login_required`, as shown in 44, 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 45 and 46.

```

44     def login_required(f):
45         @wraps(f)
46         def decorated_function(*args, **kwargs):
47             if 'username' not in session:
48                 # If user not login, redirect to the first page
49                 return redirect(url_for('login'))
50             # return jsonify({"error": "Unauthorized"}), 401
51             return f(*args, **kwargs)
52
53     return decorated_function

```

(a) login_required

```

37     @socketio.on('connect')
38     @authenticated_only
39     def connect():
40         username = request.cookies.get("username")
41         room_id = request.cookies.get("room_id")
42         if room_id is None or username is None:
43             return
44         # socket automatically leaves a room on client disconnect
45         # so on client connect, the room needs to be rejoined
46         join_room(int(room_id))
47
48         emit("incoming", {"content": f"{username} has connected", "color": "green", "type": "system"}, to=int(room_id))

```

(b) authenticated_only

Figure 44: login_required and authenticated_only

```
# 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 45: forbidden_access

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

Figure 46: app.py home

3 Contribution

```
#include <iostream>

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