Table of Contents

1. Basic functionality & Implementation Explanation **2**

1.1. User Login and Sign Up 2

1.2. Index page 4

1.3. Add Friend 5

1.4. Friends Request 7

1.5. Friends List 9

1.6. Homepage - Chat room 11

2. Additional Criteria **14**

2.1. Hashing and Salting Password 14

2.2. HTTPS for Secure Communication 14

2.3. Requests Authentication with Session Token 15

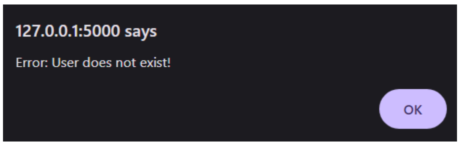
3. Group Members Contributions **17**

4. Challenges we faced **17**

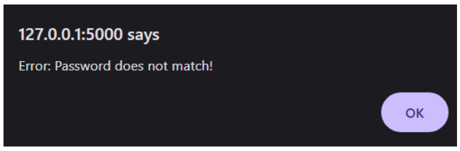
1. Basic functionality & Implementation Explanation

1.1. User Login and Sign Up

To ensure security, user sign up required to meet some criteria when creating usernames and passwords for a secure sign in/log in process. Fields corresponding to usernames and passwords are required to be inputted by the user and cannot be identical. If this case occurs, relevant error messages will be triggered, informing the user to inspect their entries and correct them. In the login screen, the server also checks if the inputted username exists in the database and verifies the password given. An error will be prompted if these requirements are not met. For sign up password and username verification, specific requirements for both entries are listed below. Or else, failure reasons will be prompted.



(a) User not exists in database error



(b) Invalid credentials error

Figure 1: Alerts for invalid user inputs during **login in**

Valid Username Features

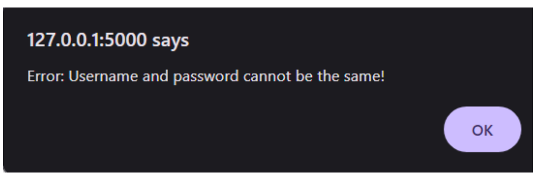
The server checks the user input and searches it in the database to ensure that there is no repeated username to reduce redundancy and future errors. A username follows the requirements stated below:

- Length must be at least 5 characters
- Only contains letters and numbers
- Contain no special characters, spaces or symbols

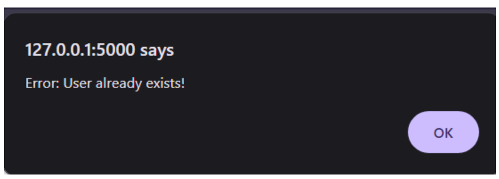
Strong Password Features

To assure a strong password is created, user needs to follow the following requirements:

- Length must be between 8 and 20 characters
- Contains both lowercase and uppercase letters
- Contains numbers and symbols
- There cannot be more than 3 repeating and subsequent characters



(a) Empty user input(s) error



(b) User existed in database error

Figure 2: Alerts for invalid user inputs during **sign up**

ADDITIONAL CRITERIA: User Authentication (Hash and Salt)

If the password meets the requirements, the username, password, along with a salt value are stored into database with **symmetric key encryption**, due to its efficient and simplistic algorithm characteristics. In which password field is hashed and salted before storing. This facilitates secure transmission of large amounts of data such as friend requests (that can contain potentially numerous entries).

	username	password	salt
	Filter	Filter	Filter
1	Test01	4f3ecbfab55997f56e9a7ca8a345ec9963e78accf28169b19dbe772b511d63a5	e2fbe7a0adb18cd8639ebcae8baf7de0
2	Test02	0360945bdf2c18cf2de9ab31ed4df33888b464d56f74fa04bca39e190b031de2	7e9c0888e4aac0e449d6422034d71b74
3	Test03	65db5cf5e2d3d211d303d3ee075ddce4189b975a1b90701f839657b2d880d13c	d5d623a4fd71d872c3acf7f0d29145e8

Figure 3: ***User*** table in database showing Test01, Test02, Test03's records

This is to ensure the server knows nothing about the credentials. The formulated criteria created enabled us to look more closely at the password requirements in order to achieve security and validity measures. We checked that a valid password was entered by the user to ensure security. These methods allow only permitted users to have a unique password to reduce the possibility of an interceptor accessing their private information.

Hash and salt are used to ensure that user authentication was achieved. This process preserves protection of the individual against a breach of their private data, rainbow table attacks, password theft, etc.

When a user signs up and creates an account, the plain-text password is hashed through the cryptographic hash function. A randomised salt value will then be generated and attached to the password before hashing. The salt value is unique to each user so the addition of the randomised hashing process makes it more secure against dictionary and rainbow table attacks.

```python
salt = secrets.token_hex(16)  # salting
salt_pw = password + salt     # attach salt to password
password = hashlib.sha256(salt_pw.encode()).hexdigest()  # hashing
db.insert_user(username, password, salt)     # insert data into db
```

Listing 1: Hashing and salting password [1]

During user login, the corresponding password and salt based on the username from database will be retrieved. This password is then concatenated with the salt and converted into bytes. The hashed password will then be compared to verify if the user input and the actual password is valid or not. If not, error message will be prompted, disabling the user to access.

```python
user = db.get_user(username)
if user is not None:
    stored_salt = db.get_salt(username)
    salt_pw = (password + stored_salt).encode()   # concatenate
    hash_pw = hashlib.sha256(salt_pw).hexdigest() # hashing

if user.password != hash_pw:  # verify pw in db & user input
    return "Error: Password does not match!"
```

Listing 2: Verifying user password [1]

1.2. Index page

Once users logged in successfully, an index table page with three buttons are displayed. This GUI feature directs the users to choose where they would like to navigate to. These include "*Add friend*", "*Friend List*", "*Friend Request*". Additionally, a navigation bar is showed with the username that the user logged in with.

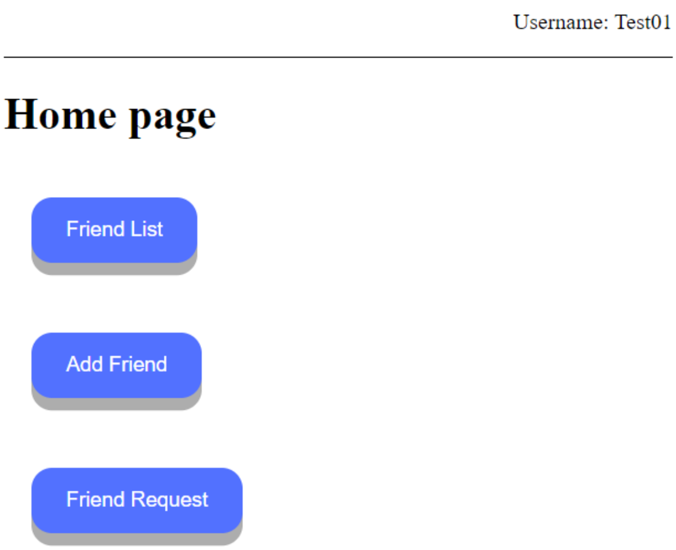


Figure 4: **Table** page for navigation

1.3. Add Friend

By submitting a friend's username, this allows the user to add a friend. However, criteria must be met and verified by the server before this action is completed. The server will first checks the entered username and reviews the database to see if it an existing username. If there is no such record of the username in the database, an *"Unknown username!"* error is raised. The server also checks if the user is adding themselves, which is not permitted. A unique error message for that is also implemented *"You can't add yourself as a friend!"*.The server also checks if the friend request had been already sent. This is done by checking the records in database to see if the friend's username already exists, by matching the username and "*sender*" column of ***Friend*** table.

Figure 5: Alerts for adding friend

```
# getting all the added friends for the user who logged in
def for_add_friend(username: str):
    with Session(engine) as session:
        received_requests = (
            session.query(Friend)
            .filter(Friend.sender == username)
            .all()) # return a list with received_requests.friend
```

Listing 3: Method to get added friends from ***Friend*** table in database [2]

If these conditions are met, the friend can then only be added which a new record with the username (as sender), friend and a default status will be added into the database. The friend's username is encrypted before storing using `Fernet`, a **symmetric encryption** method that uses **AES** algorithm for plaintext encoding and decoding. The code below explains how friend username encryption works [3].

```
# -- in app.py > add_friend() with reference --

# use the consistent encryption key from db
from db import encryption_key

# initialize fernet class
key_value = Fernet(encryption_key)

# get from jinja form
friend = request.form.get("receiver")

# create ciphertext, convert string to bytes datatype & encrypt
encrypted_username = key_value.encrypt(friend.encode())

# store encrypted friend's username into db
# encrypt on client-side before sending to the server/database)
db.insert_friend(username, encrypted_username)
```

```python
# -- db.py --
encryption_key = get_encryption_key() # generate key
ENCRYPTION_KEY_FILE = 'CERTIFICATES/encryption.key'

# getting encryption key from 'encryption.key' file
# to ensure the encryption key remained consistent AT ALL TIMES
def get_encryption_key():
    if os.path.exists(ENCRYPTION_KEY_FILE):
        key = Fernet.generate_key()
        with open(ENCRYPTION_KEY_FILE, 'rb') as f:
            return f.read()

def insert_friend(sender: str, friend: str):
    with Session(engine) as session:
        # insert values for each column in db
        sender_friend = Friend(sender=sender, friend=friend, status
="default")  # set default status in db
        session.add(sender_friend)
        session.commit()
```

Listing 4: Symmetric encryption applied to *"friend"* before storing [4]

sender ▾1	**friend**	status
Filter	Filter	Filter
Test01	gAAAAABmE9oufK960YHsQjfZBtjtoB4kKtLe4RAINLhnfs98Om5lLyHmR6_gFEDMFNCObL1Mxwy_g0wLAsjE5KInRB1UDPrVJA==	reject
Test01	gAAAAABmE9oxsBeWdETpCTbNTq5RSHifjbAXX4fvaDDuJJMP0T9wriWQc8OfkVH0KMu6p6BxsLADXIgxfXwPIt9Y1crh3W0MzQ==	default
Test01	gAAAAABmE97ScW3W-VaJ9bHN5ZzkV40OlFvWA5cDIEJ8PYVMS0r3tYd_yKSwaP7mO9T29jKx_BLbYdfACQFc9-nfJlhMVqBR2Q==	approve

Figure 6: ***Friend*** table in database showing Test01's data

1.4. Friends Request

This page displays **Sent Request** and **Received Request** with two different tabs. Under "Sent Request", a list of all the requests that have been sent by the user will be displayed in view-only mode. Under "Received request", any users that have added the current user will be displayed. To approve or reject requests, checkbox feature is used to allow the user to select whether they would like to approve or decline a friend request.

The server checks if no check box is selected when the buttons are clicked, and displays *"Select at least one friend to approve."* if that's the case. Conversely, message such as *"Friend requests approved/rejected successfully!"* will be showed after the user react to the request correctly. This will update the *"status"* column of ***Friend*** table in database from `default` to `approve/reject`, by the corresponding user and friend row (in Figure 5).

Figure 7: Friend Request page interface

```python
@app.route("/friendrequest")
def display_friendrequest():
    senders = db.get_sent_requests(username)       # for "SENT"
    friends = db.get_received_requests(username)  # for "RECEIVED"
    return render_template("friendrequest.jinja", username=username, senders=senders, friends=friends)
```

Listing 5: Retrieving sent & received requests from database

For getting all the friends' usernames that the user has sent, a query is used to select all the records found in database by mapping the user's username with *"sender"* column in ***Friend*** table. Since the friend's username stored is encrypted, a decryption method is implemented to get the friends' usernames which is in *"Friend.friend"* column.

```python
# -- db.py --
def get_sent_requests(username: str):
    with Session(engine) as session:
        received_requests = (
            session.query(Friend)
            # just need to get ALL the friend user sent
            .filter(Friend.sender == username).all())
        decrypted_ls = []
        key_value = Fernet(encryption_key) # set fernet class
        for sent_friends in received_requests:
            # decrypt and convert into string
            decrypted_friend = key_value.decrypt(sent_friends.friend).decode()
            decrypted_ls.append(decrypted_friend) # add into list
        # return list with all the usernames
        return decrypted_ls
```

Listing 6: Retrieving sent requests from database

For getting received requests, it's similar to above, but filtering the *"status"* column as "default" and to check if the user's username is in the *"friend"* column, since they are the ones getting the requests. Then return the list with values from *"Friend.sender"* column.

```
# -- db.py --
def get_received_requests(username: str):
    with Session(engine) as session:
        sent_requests = (
            session.query(Friend)
            # get haven't approved/rejected records
            .filter(Friend.status == "default").all())
        decrypted_ls = []
        key_value = Fernet(encryption_key)
        for sent_friend in sent_requests:
            # decrypt and convert into string
            decrypted_friend = key_value.decrypt(sent_friend.friend
    ).decode()
            # compare with decrypted friend for getting username
            if decrypted_friend == username:
                # get the sender instead
                decrypted_ls.append(sent_friend.sender)
        return decrypted_ls
```

Listing 7: Retrieving received requests from database

On the other hand, to update the selected friend requests, both the client-side and server-side interact with each other to get the user's action for their selected friend(s) and to update database records.

```
# -- in app.py > update_friend_request() --
def update_friend_request():
    # get from form submission
    action = request.form.get("action")
    if action == "approve" or action == "reject":
        # get form submitted from jinja
        friend_usernames = request.form.getlist("friends")
        # call db function to update status
        for friend in friend_usernames:
            db.update_request_status(username, friend, action)

# -- db.py --
def update_request_status(sender_username, friend, status):
    with Session(engine) as session:
        key_value = Fernet(encryption_key)
        # from Test01 view
        # Friend.sender ( eg. Test01 | lyvia(encrypted) | default )
        sent_friendships = (
            session.query(Friend)
            .filter(Friend.sender == sender_username).all())
        for friends in sent_friendships:
            decrypted = key_value.decrypt(friends.friend).decode()
            if decrypted == friend:  # match friend column
                friendship.status = status # update status
        session.commit() # update database
```

Listing 8: Update request status

1.5. Friends List

Only "approved" friends will be displayed alongside their online status. Similar to approve or reject friend request, user's approved friends are listed beside check boxes for user to select who to chat with. An error message will be prompted if no checkbox is selected when the button is clicked. Also, only "online" friend can be selected to chat with.

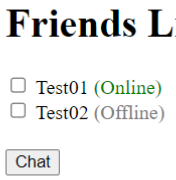
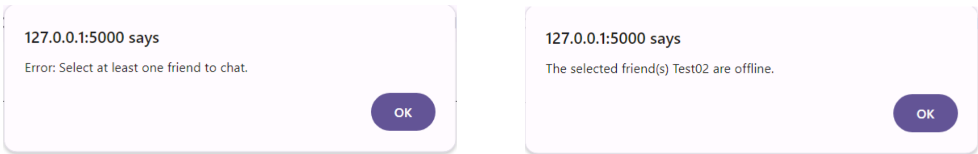


Figure 8: Friend List page interface



(a) No friend selected error

(b) Offline friend selected error

Figure 9: Alerts for selecting friend to chat

The online status of each friend is initially set when a user logged into the web server, when a new model is created to set their status using `set()`, along with adding/removing users for handling online and offline cases. Then, logged-in users is set as "Online" by creating object of **OnlineUser** class in the server.

```python
# -- models.py --
class OnlineUser():
    def __init__(self):
        self.online_user = set() # initialize set for online users
    # add users to set
    def set_online(self, user: str):
        self.online_user.add(user)
    # remove user from set
    def set_offline(self, user: str):
        if user in self.online_user:
            self.online_user.remove(user)
    # return their online status
    def is_online(self, user: str) -> bool:
        return user in self.online_user

# -- app.py --
online_user = OnlineUser()

# set logged-in users as "Online"
online_user.set_online(username)
```

Listing 9: Setting user online status after logged in [5]

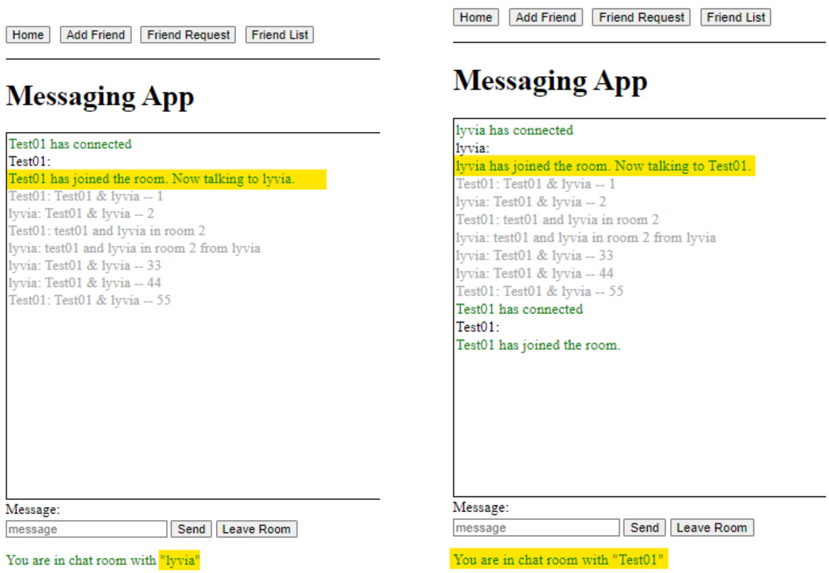
To get the approved friend list displayed, data is retrieved from the ***Friend*** table in database, and parsed them into the template to be rendered and displayed.

```
# --- app.py > display_friendlist() ---
@app.route("/friendlist", methods=["GET", "POST"])
def display_friendlist():
    # get approved friend list from db
    friend_usernames = db.get_approved_request(username)

    # create dictionary to store {username: online_status}
    friend_status = {}
    for friends in friend_usernames:
        # update their status, checking OnlineUser set
        friend_status[friends] = online_user.is_online(friends)

    # check jinja template if "Chat" button is clicked
    action = request.form.get("action")
    if action == "chat":
    # redirect to 'homepage' to chat after button clicked
        return redirect(url_for('homepage', username=username, \
                                online_friend=online_friend))
    # parse to jinja to be used
    return render_template("friendlist.jinja", username=username, \
                           friend_usernames=friend_usernames, \
                           friend_status=friend_status, \
                           online_friend=online_friend)

# --- db.py ---
def get_approved_request(username: str):
    with Session(engine) as session:
        received_requests = (
            session.query(Friend)  # select * from Friend

            # map username with 'sender', "approve" in 'status'
            .filter(Friend.sender == username, \
                    Friend.status == "approve").all())

        decrypted_ls = []
        decrypted_friend = None
        key_value = Fernet(encryption_key)

        for approved_friends in received_requests:
            # decrypt usernames & convert into string
            decrypted_friend = key_value.decrypt(\
                                   approved_friends.friend).decode()
            decrypted_ls.append(decrypted_friend) # add into list

        # if username is in Friend.friend column
        if decrypted_friend == username:
            decrypted_ls.append(decrypted_friend)
        return decrypted_ls
```

Listing 10: Retrieving approved friends from database

1.6. Homepage - Chat room

Once a valid online and approved friend is selected to chat with from the friend list, the user will be redirected to the homepage where a chat room is created. The page includes main components, including a navigation bar with buttons to navigate to different pages, a main message box for chatting, a text box for inputting the message and buttons to send message or leave room. As seen in Figure 10, once user joined the chat room, a default message with text in green is displayed, where they can see who they are talking to. Also, their message history will be shown next with text in grey.



(a) User "Test01" view (b) User "lyvia" view

Figure 10: Homepage interface after users join the chat

Message Encryption & Decryption

For the users to communicate securely, message encryption is implemented using **PKCS#1 OAEP**, which is an **asymmetric encryption** utilising RSA and OAEP [6]. This is to ensure the server is unable to read the messages. Code implementation is explained below.

```python
def decrypt_msg(encrypted_msg, private_key):
    # load private key using RSA method
    private_key_obj = RSA.importKey(private_key)
    # generate OAEP cipher with RSA private key
    cipher = PKCS1_OAEP.new(private_key_obj)
    # using the generated cipher to decrypt
    decrypted_msg = cipher.decrypt(encrypted_msg)
    return decrypted_msg.decode() # convert bytes to string

def encrypt_msg(message, public_key):
    receiver_key = RSA.importKey(public_key) # load public key
    # generate OAEP cipher with RSA public key
    cipher = PKCS1_OAEP.new(receiver_key)
    encrypted_msg = cipher.encrypt(message.encode()) # encrypt
    return encrypted_msg
```

Listing 11: Message encryption using PKCS#1 OAEP [6]

Sending Message

When a user sends a message and clicked the "Send" button in the chatroom, this triggers the `send()` function to handle message transferring.

```python
# generate RSA key in 1024 bit
key = RSA.generate(1024)
# export RSA key into binary
private_key = key.exportKey()
# generate public key mapping to the private key
public_key = key.publickey().exportKey()

# getting public key values
public_key = """-----BEGIN PUBLIC KEY-----
# public key generated (too long to show here)
-----END PUBLIC KEY-----"""

filename = "CERTIFICATES/msgEncrypt.pem"
# get private key from the pem file
def get_private_key(filename):
    with open(filename, "rb") as file:
        private_key = file.read()
    return private_key

@socketio.on("send")
def send(username, receiver, message, room_id):
    # encrypt plaintext msg
    encrypted_msg = encrypt_msg(message, public_key)
    # insert new records into db with the four values
    db.insert_encrypted_msg(room_id, username, receiver, encrypted_msg)
    private_key = get_private_key(filename)     # decrypt
    decrypted_msg = decrypt_msg(encrypted_msg, private_key)

    # HMAC - to authenticate ALL msg
    # use ALL msg to generate & verify HMAC
    db_encrypted_msg = db.get_encrypted_msg(username, receiver)
    for msg in db_encrypted_msg:
        msg_in_str = ''.join([str(msg)])
    emit("incoming", (f"{username}: {decrypted_msg}", mac), to=room_id)
```

Listing 12: Sending message and saving into database [7]

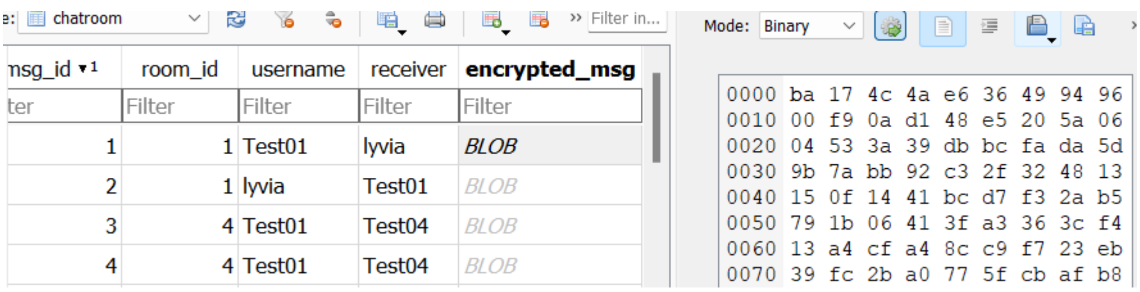


Figure 11: ***Chatroom*** table with binary *encrypted_msg* column in database

Message Authentication Code

Message Authentication Code is used to ensure the messages are not being modified by the server. To enhance better security, hashed and salted passwords are used as the shared secret key in creating the MAC based on Hashlib SHA256. This enables us to verify if the messages are modified since all the message from database is applied to the HMAC, which acting like a digital fingerprint for the chatroom. In which this is done by `verify_mac()` that compares the two MACs values generated and returns a boolean value if the parsed in messages are being modified. Users' hashed and salted passwords are used as the shared secret key for enhanced security reasons. Therefore, data integrity and authentication can be guaranteed.

```python
def get_mac(username, receiver):
    # use user's password as the secret key
    user_pw = db.get_password(username)
    encrypted_msg = db.get_encrypted_msg(username, receiver)
    for msg in encrypted_msg:
        msg_in_str = ''.join([str(msg)]) # convert list to string
    user_pw_in_bytes = user_pw.encode()  # string -> bytes

    # generate HMAC using hashlib.sha256 hash function
    # use user's password (in bytes) as the shared secret key!!
    # to verify if messages are modified
    # return MAC in hexadecimal digits string
    mac = hmac.new(user_pw_in_bytes, msg_in_str.encode(), hashlib.
   sha256).hexdigest()
    return mac

def verify_mac(msg, received_mac, username):
    user_pw = db.get_password(username)
    user_pw_in_bytes = user_pw.encode()
    mac = hmac.new(user_pw_in_bytes, msg.encode(), hashlib.sha256).
   hexdigest()
    # compare 2 MACs
    # if return false, messages are modified
    return hmac.compare_digest(received_mac, mac)

def send():
    # existing code above
    mac = get_mac(username, receiver)
    is_valid_mac = verify_mac(msg_in_str, mac, username)
    if not is_valid_mac:
        return "ALERT! ALERT! ALERT! MAC verification failed!!!"
```

Listing 13: Message Authentication Code implementation [8] [9]

2. Additional Criteria

2.1. Hashing and Salting Password

This implementation has been discussed in page 3.

2.2. HTTPS for Secure Communication

HTTPS is implemented with SHA-256 to ensure secure communication among the client and server. This is done by generating root certificate and private key. Additionally, followed by adding and trusting the root certificate on own local machine. Implementation steps involved creating self-signed root certificate, with a generated private key for the local CA, along with SSL certificate signed by the root certificate [10].

```
# generate RSA private key
openssl genrsa -out myCA.key 2048

# generate root certificate
openssl req -x509 -new -nodes -key myCA.key -sha256 -days 1825 -out myCA.pem

# generate private key, encrypting key with DES3 cipher
windpty openssl genrsa -des3 -out localhost.key 2048

# copy root cert to add into trusted root cert. authorities
or sudo cp ~/certs/myCA.pem /usr/local/share/ca-certificates/myCA.crt

# generate CSR with private key
openssl req -new -key localhost.key -out localhost.csr

# create SSL cert signed using CSR & root cert
openssl x509 -req -in localhost.csr -CA myCA.pem -CAkey myCA.key -CAcreateserial -out localhost.crt -days 825 -sha256 -extfile localhost.ext
```

Listing 14: Generating root certificate [10]

As a result of creating a root certificate and trusting it on local machine, along with HMAC generation, the web server's connection now is secured and proved with the self-signed certificate (As shown in Figures 12 and 13).

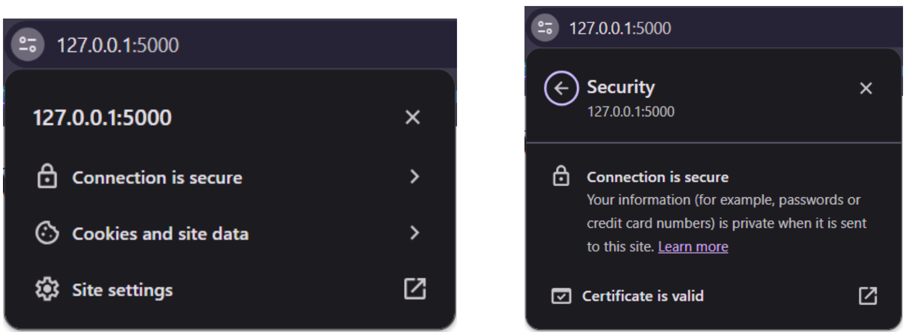


Figure 12: Secure connection on server

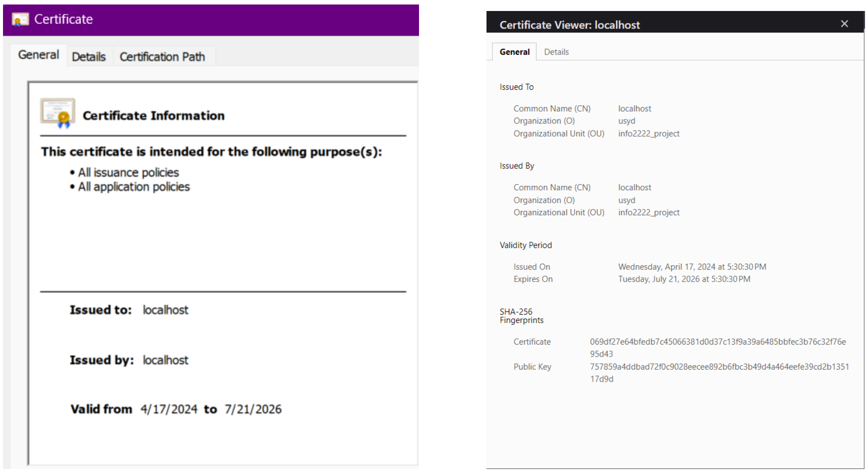


Figure 13: Self-signed root certificate generated

2.3. Requests Authentication with Session Token

To authenticate all the server's requests, a random hexadecimal text string is generated for protecting user information across pages. A session dictionary is created by creating every user a special session identifier, mapping to each username and randomised session id. This initialisation is done whenever a user logged into the server. While another session is initialised when an approved and online friend is selected to chat with from the friend list, mapping the session dictionary with `onlineFriend_username` as the key. This structure makes sure every distinct username, session id and online friend corresponding to a unique session. Hence, each session data is remained distinct to each user.

To verify each session in different pages, if statements are executed in every functions to check if the unique session are existed in the session dictionary. Separate checks are done for each username, session id, as well as online friend. If they do not exist in the session, the server simply return to the login page for security reasons. This prevents IDOR attack, where the "direct object reference" can be manipulated easily.

To test it out on our server, consider the url:
https://127.0.0.1:5000/home?username=lyviaonline_friend=Test01
I tried to replace "`username=lyvia`" into "`username=hello`", this won't be allowed and the page will be redirected to login page again since this fails the request authentication. This same applies to replacing "`online_friend=Test01`" with other values, as well as all pages, sessions, and requests are authenticated through the same implementation.

```python
app.config['SECRET_KEY'] = secrets.token_hex()
socketio = SocketIO(app)
SESSION_TOKEN_DICT = {}

def login_user():
    # generate random hexadecimal text string
    session_id = secrets.token_hex()
    # store session id with username, distinct
    SESSION_TOKEN_DICT[username] = session_id
    # create new session & assign the values
    session[f'user_{username}'] = username
    session[f'sessionID_{username}'] = session_id
```

```python
def display_friendlist():
    # create another session mapping the user and friend
    session[f'onlineFriend_{username}'] = online_friend


# SESSION TOKEN VERIFICATION -- for every function in the server
# check specific username in session
if not f'user_{username}' in session:
    return redirect(url_for('login'))

# check specific session ID key in session
sessionID_key = f'sessionID_{username}'
if not sessionID_key in session:
    return redirect(url_for('login'))


# check specific session token in session
if SESSION_TOKEN_DICT.get(username) != session[sessionID_key]:
    return redirect(url_for('login'))

# check if the right selected online friend is in the session
# mapping to "onlineFriend_{username} : {online_f}"
session_online_friend = session.get(f'onlineFriend_{username}')
if session_online_friend != online_f:
    return redirect(url_for('login'))
```

Listing 15: Generating session and verifying it [11]

3. Group Members Contributions

Our initial method of equally splitting up the code components is not what we ended up doing due to differing schedules, availability, commitments and external circumstances. Instead, we separated it based on individual contributions. While both Lyvia and Rakshita collaborated on defining criteria and conducting research for code implementation, Lyvia took on the primary responsibility for formulating the code. Meanwhile, both Lyvia and Rakshita worked together on writing the report. Rakshita helped with a number of other aspects of the project such as, thorough research within the project report and the tasks mentioned above but her direct involvement in the coding was minimal. Our main objective of comprehending the items to complete this messaging tool was achieved in our ability to test the code and find flaws within our tool to enhance its features for easy user usability.

4. Challenges we faced

Throughout the process of completing this assignment, there were several challenges that we faced which required us utilise additional trusted sources to find suitable solutions to tackle the issue. One of the challenges that we faced was regarding the implementation of https on the local machine in generating the Certificate Authorities (CA) for further security and privacy measures.

The first approach was to research on the web regarding how this process can be executed. After formulating the implementation method found through research and testing it, it didn't seem to work as required. The next strategy was to seek assistance from our tutors discussions on Ed, although this gave us some guidance, we were still unable to achieve the expected result. We then went back to the first approach and continued the research.

Through this lengthy process, Lyvia was able to find an explanation for this process through the tutorials and solved this problem. After these range of problem solving techniques, a solution was found and implemented to ensure enhanced protection/ security of our messaging tool.

Majority of the challenges that we faced were solved through thorough research from a range of sources such as websites, videos, tutorial notes, lectures and our tutor's help. Having access to these varying sources and not just websites helped us significantly in understanding new concepts better through visuals (images, diagrams and videos) as well as face-to-face interactions with our tutors.

Bibliography

[1] D. Arias, “Adding salt to hashing: A better way to store passwords,” Feb 2021. [Online]. Available: https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/

[2] S. Overflow., “Using query and filter in sqlalchemy to modify user table,” n.d. [Online]. Available: https://stackoverflow.com/questions/31732423/using-query-and-filter-in-sqlalchemy-to-modify-user-table

[3] GeeksforGeeks., “Fernet (symmetric encryption) using cryptography module in python.” 2020. [Online]. Available: https://www.geeksforgeeks.org/fernet-symmetric-encryption-using-cryptography-module-in-python/.

[4] ——, “Encrypt and decrypt files using python.” 2021. [Online]. Available: https://www.geeksforgeeks.org/encrypt-and-decrypt-files-using-python/.

[5] A. Fadheli, “How to make a chat application in python - the python code.” n.d. [Online]. Available: https://thepythoncode.com/article/make-a-chat-room-application-in-python#google_vignette

[6] pycryptodome.readthedocs.io., “Pkcs1 oaep (rsa) — pycryptodome 3.9.9 documentation.” n.d. [Online]. Available: https://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html.

[7] S. Overflow., “Using pycrypto, how to import a rsa public key and use it to encrypt a string?” n.d. [Online]. Available: https://stackoverflow.com/questions/21327491/using-pycrypto-how-to-import-a-rsa-public-key-and-use-it-to-encrypt-a-string

[8] ——, “Python encoded message with hmac-sha256.” n.d. [Online]. Available: https://stackoverflow.com/questions/38133665/python-encoded-message-with-hmac-sha256

[9] 262588213843476, “Digital signature verification with python and hmac.” n.d. [Online]. Available: https://gist.github.com/craigderington/9cb3ffaf4279af95bebcc0470212f788

[10] B. Touesnard, “How to create your own ssl certificate authority for local https development.” 2021. [Online]. Available: https://deliciousbrains.com/ssl-certificate-authority-for-local-https-development/.

[11] www.tutorialspoint.com., “Flask – sessions.” n.d. [Online]. Available: https://www.tutorialspoint.com/flask/flask_sessions.htm