THE UNIVERSITY OF SYDNEY

School of Information Technologies
Faculty of Engineering & IT

ASSIGNMENT/PROJECT COVERSHEET - GROUP ASSESSMENT

Unit of Study: INFO2222

Assignment name: Project1

Tutorial time: RE03 **Tutor name:** Tianyi Zhang

DECLARATION

We the undersigned declare that we have read and understood the *University of Sydney Academic Dishonesty and Plagiarism in Coursework Policy*, an, and except where specifically acknowledged, the work contained in this assignment/project is our own work, and has not been copied from other sources or been previously submitted for award or assessment.

We understand that failure to comply with the *Academic Dishonesty and Plagiarism in Coursework Policy* can lead to severe penalties as outlined under Chapter 8 of the *University of Sydney By-Law 1999* (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

We realise that we may be asked to identify those portions of the work contributed by each of us and required to demonstrate our individual knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

Project team members				
Student name	**Student ID**	**Participated**	**Agree to share**	**Signature**
1. Langyi Chen	510090391	(Yes) / No	(Yes) / No	陈朗毅
2. Wentao Gao	480187808	(Yes) / No	(Yes) / No	高文韬
3.		Yes / No	Yes / No	
4.		Yes / No	Yes / No	
5.		Yes / No	Yes / No	
6.		Yes / No	Yes / No	
7.		Yes / No	Yes / No	
8.		Yes / No	Yes / No	
9.		Yes / No	Yes / No	
10.		Yes / No	Yes / No	

SIT Building, J12
The University of Sydney
NSW 2006 Australia

T +61 2 9351 3423
F +61 2 9351 3838
E sit.info@sydney.edu.au
sydney.edu.au/it

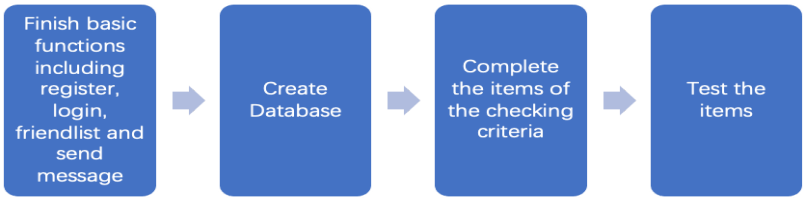
ABN 15 211 513 464
CRICOS 00026A

INFO2222 report
(No more pages than 3 pages for main body excluding unlimited appendix)

Group name: RE03_Team4
Name: Wentao Gao, Langyi Chen

Summary

Lifecycle



Finished items

1. Properly store passwords on the server (Using hash to encrypt the passwords and store them in the SQL database)
2. Check server's certificate (change HTTP to HTTPS)
3. Securely transmitting a password to server (Using RSA to encrypt the password in the front end and decrypt in the back end)
4. Properly check whether password is correct (Adding salt to the passwords)
5. Securely transmitting the message from A to B, even the server who can forward communication transcript cannot read the message, or modify the ciphertext (Using RSA to encrypt and decrypt the message and using digital signature for authentication)

Individual percentage

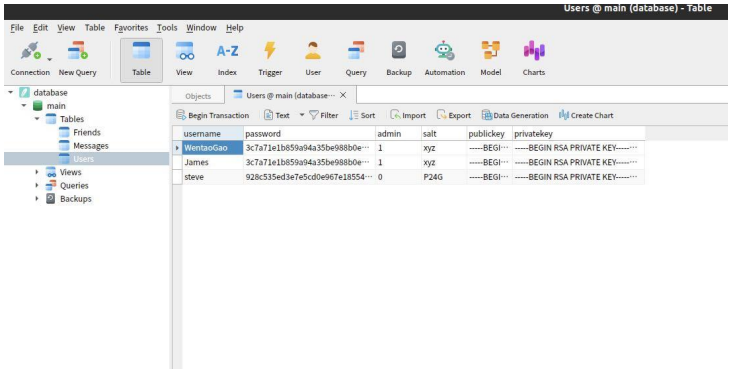
Name	Contribution
Langyi Chen	50%
Wentao Gao	50%

Body of report

1. Properly store passwords on the server

We used SQL database to store the password with hashing it to prevent attackers stealing the password of users from the database. A table named Users in the datasets will store the usernames, passwords, salts, public keys, and private keys of users. The attackers can not see the actual password from the database directly. When a user registers an account, he will enter the username and password. The password will be added with salt, then hashed and stored in the database, the salt is created at random.

The result of the table is shown below (The python code of creating random salt and hashing then storing the passwords is in graph 1-1 and graph 1-2 in appendix).



The screenshot is Users table. First two accounts are admins and inserted into the database directly. Third one is a user registers his account on the register page, the password has been added salt and hashed, the salt is produced at random.

2. When log in, first check server's certificate (e.g., you can manually create one using a hardcoded CA public key in your code)

   To make our server running on https using the localhost 127.0.0.1, firstly generate the Private Key and Root Certificate, and then store them on our project folder. after and add the Root Certificate file CA.pem to system. After that, we Creating CA-Signed Certificates for our website by using Root Certificate file, and also store it in our project folder.(this part is shown in graph 2-1)

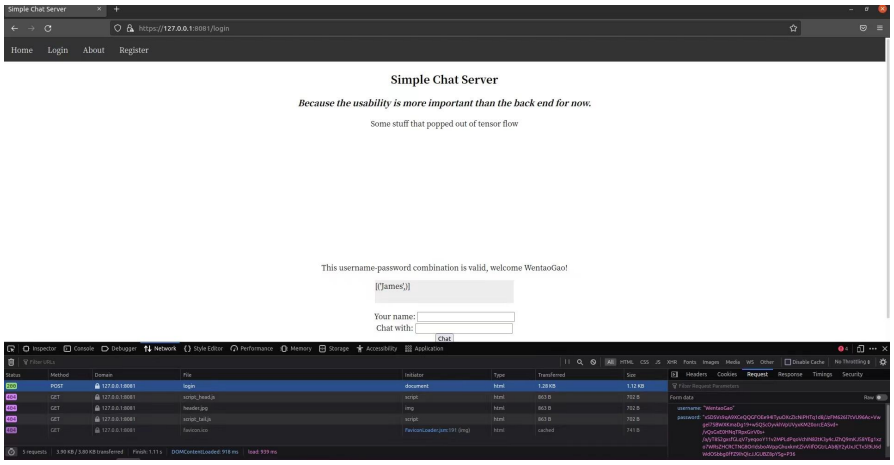
   In the run function we put the previously created CA-signed certificate file into the keyfile and certfile arguments and in server argument use gunicorn to make our website https(this part is shown in graph 2-2)

   Finally run the run.py and open the website, we can clearly see that we have our website running https

3. Securely transmitting a pwd to server (leveraging secure protocols or design the secure transmission properly)

   When a user tries to login, he will enter the username and password. The password will be encrypted by the front end using a public key, then the password is sent to the back end. And the back end will use a private key to decrypt the password to protect the transmission of the password.

   The result of transmitting pwd is shown as below (The code of encrypt in front end and decrypt in back end is in graph 3-1 and graph 3-2 in appendix)



4. Properly check whether password is correct (at least use the simple method that defends against offline pre-computation attacks)

   We added salt to the passwords to defend against offline pre-computation attacks. To check whether a password is correct, users will type the passwords on the website, then it will be sent to the backend with encryption. Backend will decrypt it to get the original password, and add salt to it before hashing. Next backend will hash the original password with salt to compare with the passwords in the dataset having the same usernames. Only the passwords in the dataset and the users entering are the same,

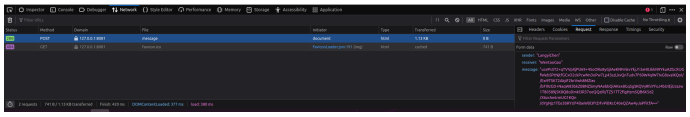
users can login successfully. (The python code of checking password is graph 4-1 in appendix)

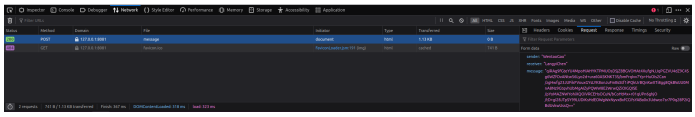
5. Securely transmitting the message from A to B, even the server who can forward communication transcript cannot read the message, or modify the ciphertext (leveraging secure protocols or design the authenticated secure transmission properly)

   Firstly, we generate a pair of rsa keys and save them as file in our project folder.(this part is shown in graph 5-1)

   When user register a new account, system will generate a pair of keys for the user and save them in database with user's other information(e.g. Username, password, salt). (this part is shown in graph 5-2)

   When a user sends messages to another user, we use the key generated in step 1 and Forge package in javascript on the client-side. Firstly, we use the public key to encrypt the message.(this part is shown in graph 5-3)



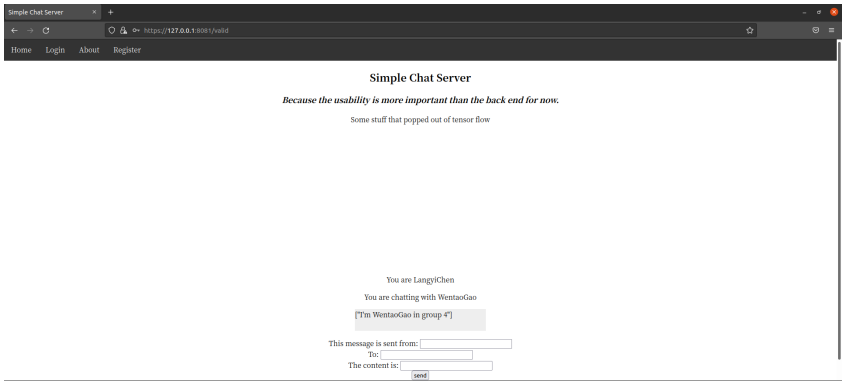


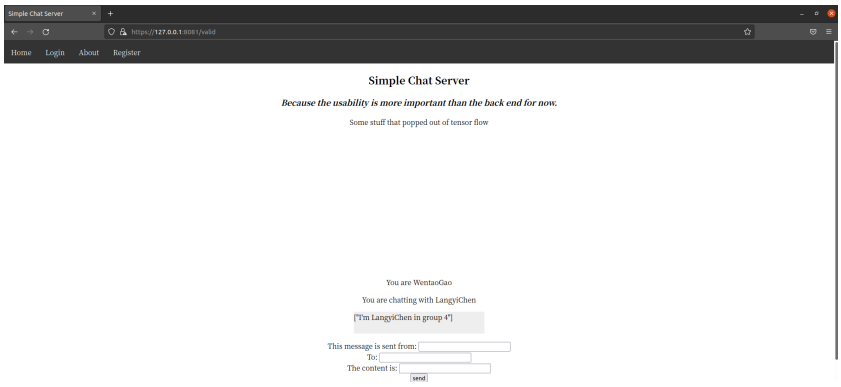
   Then, we use private key to decrypt the message in python on the server-side.(this part is shown in graph 5-4)

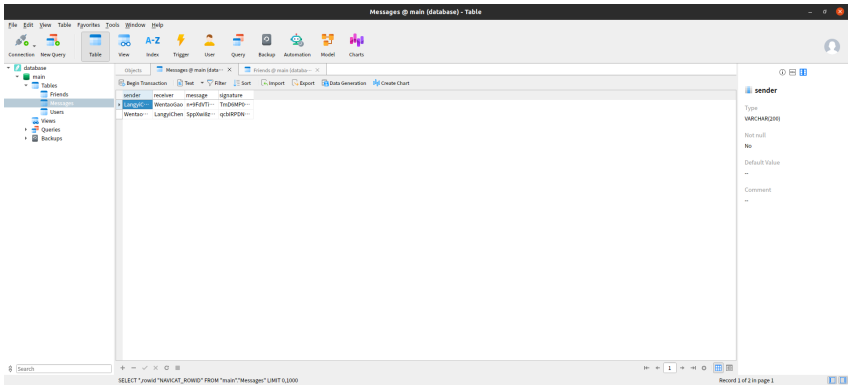
   First we got the sender's private key and receiver's public key in the database.The sender will use the receiver's public key to encrypt the message and sign by using his own private key.(this part is shown in graph 5-5)

Then we got the sender's public key and receiver's private key in the database. The receiver will use sender's public key to verify the signature signed by sender and decrypt the message by using his own private key.(this part is shown in graph 5-6)

Finally, in two user's client-side, we can see the messages sent by each user, and it also encrypts successfully in the database.







Appendix

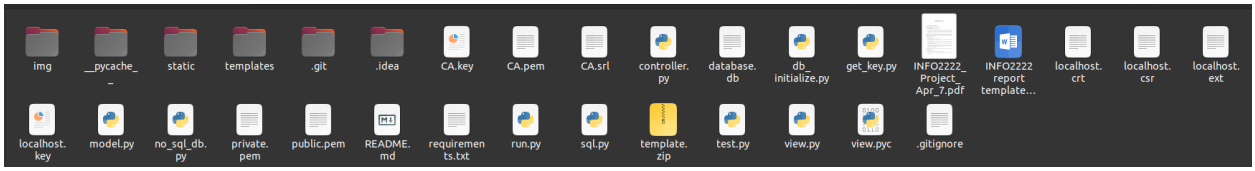
More screenshots or any other evidence demonstrating our project efforts for each of the checking criteria.

```python
def create_salt():
    salt = ''
    char = 'AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789'
    len_char = len(char) - 1
    random = Random()
    for i in range(0, 4):
        salt = salt + char[random.randint(0, len_char)]
    return (salt)
```

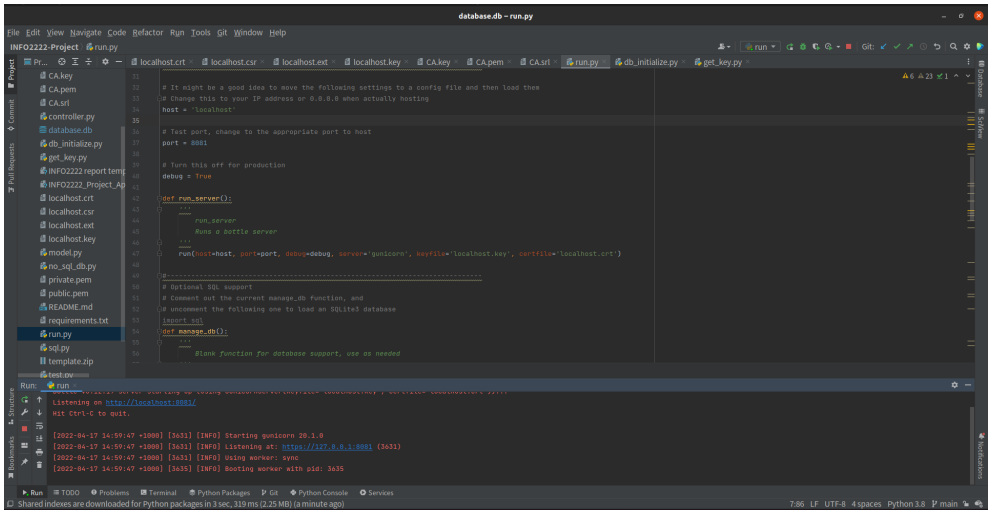
(graph 1-1)

```python
# ----------------------------------------------------------------------------------------------------

def register_check(username, password):
    salt = create_salt()
    password = password + salt
    md5 = hashlib.md5()
    md5.update(password.encode('utf-8'))
    hash_password = md5.hexdigest()
    random_generator = Crypto.Random.new().read
    rsa = RSA.generate(1024, random_generator)
    private_pem = rsa.exportKey()
    public_pem = rsa.publickey().exportKey()
    publickey = str(public_pem, encoding='utf - 8')
    privatekey = str(private_pem, encoding='utf - 8')
    usersDB = sql.SQLDatabase('database.db')
    result = usersDB.add_user(username, hash_password, salt, publickey, privatekey)
    if result:
        return page_view("login")
    else:
        return page_view("register")

```

(graph 1-2)



(graph 2-1)



(graph 2-2)

```
<form action="/login" method="post" onsubmit="return checkForm()">
    Username: <input name="username" type="text" />
</br>
    Password: <input id="password" name="password" type="password" />
</br>
    <input value="Login" type="submit" />
</form>

<script src="https://cdn.jsdelivr.net/npm/node-forge@1.0.0/dist/forge.min.js"></script>

<script>
function checkForm() {
    var publicKey = forge.pki.publicKeyFromPem("-----BEGIN PUBLIC KEY-----\n" +
        "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0NnLcLCDUSevd5krC+fG\n" +
        "C3Gv+ohU/Pyq90ZipotTbAhtHGRYHvxDIKDmDsLlVEFVgucCnuASkx74qAZ19Bsh\n" +
        "9qEup0Q/YqI5nLUkgfKd26ZiUxxiUTTMCJY+S1e8qYYi6+r/lNpn1FP5570akr4U\n" +
        "PIM4Nfsba3zutkt8gu4dFnGaAh+EyivD0LoMHYy8VSyJvMeFLIxpKk9aTgJy1El6\n" +
        "CI1PSZj70Jpz/4Lb9G6QT+7o0wzpD6EczLK9UrOvxfKmClGu+jCy9eviKCpH+ktG\n" +
        "+arR51Xlqi9su/UV3g9kRloEB62cFU9nIHhnFH2qqat1gFHnC+w7cEJ8+C9NKXia\n" +
        "kwIDAQAB\n" +
        "-----END PUBLIC KEY-----");
    let pwd = document.getElementById('password');
    var encrypted = publicKey.encrypt(pwd.value, "RSA-OAEP", {
            md: forge.md.sha256.create(),
            mgf1: forge.mgf1.create()
        });
    pwd.value = forge.util.encode64(encrypted);
    return true;
}
</script>
```

(graph 3-1)

```
# ------------------------------------------------------------------------------------------

# Attempt the login
@post('/login')
def post_login():
    '''
        post_login

        Handles login attempts
        Expects a form containing 'username' and 'password' fields
    '''

    # Handle the form processing
    username = request.forms.get('username')
    password = request.forms.get('password')
    cipher = PKCS1_OAEP.new(key, hashAlgo=SHA256)
    decrypted_password = cipher.decrypt(b64decode(password))
    password = str(decrypted_password, encoding='utf - 8')
    return model.login_check(username, password)
```

(graph 3-2)

```
# Check the login credentials
def login_check(username, password):
    '''
        login_check
        Checks usernames and passwords

        :: username :: The username
        :: password :: The password

        Returns either a view for valid credentials, or a view for invalid credentials
    '''
    usersDB = sql.SQLDatabase('database.db')
    friendsDB = sql.SQLDatabase('database.db')
    salt = usersDB.get_salt(username)
    password_salt = password + salt
    md5 = hashlib.md5()
    md5.update(password_salt.encode('utf-8'))
    hash_password = md5.hexdigest()
    result = usersDB.check_credentials(username, hash_password)
    friends_list = friendsDB.get_friends(username)
    if result:
        return page_view("valid", name=username, friendslist=friends_list)
    else:
        return page_view("invalid", reason=hash_password)
```

(graph 4-1)

```python
from Crypto.PublicKey import RSA

key = RSA.generate(2048)
private_key = key.export_key()
print(private_key)
file_out = open("private.pem", "wb")
file_out.write(private_key)
file_out.close()

public_key = key.publickey().export_key()
print(public_key)
file_out = open("public.pem", "wb")
file_out.write(public_key)
file_out.close()
```

(graph 5-1)

```python
def register_check(username, password):
    salt = create_salt()
    password = password + salt
    md5 = hashlib.md5()
    md5.update(password.encode('utf-8'))
    hash_password = md5.hexdigest()
    random_generator = Crypto.Random.new().read
    rsa = RSA.generate(1024, random_generator)
    private_pem = rsa.exportKey()
    public_pem = rsa.publickey().exportKey()
    publickey = str(public_pem, encoding='utf - 8')
    privatekey = str(private_pem, encoding='utf - 8')
    usersDB = sql.SQLDatabase('database.db')
    result = usersDB.add_user(username, hash_password, salt, publickey, privatekey)
    if result:
        return page_view("login")
    else:
        return page_view("register")
```

(graph 5-2)

```
<form action='/message' method='post' onsubmit="return checkForm()">
  This message is sent from:  <input name="sender" type="text" />
</br>
  To: <input name="receiver" type="text" />
</br>
  The content is: <input id="message" name="message" type="text" />
</br>
  <input value="send" type="submit">
</form>

<script src="https://cdn.jsdelivr.net/npm/node-forge@1.0.0/dist/forge.min.js"></script>

<script>
function checkForm() {
    var publicKey = forge.pki.publicKeyFromPem("-----BEGIN PUBLIC KEY-----\n" +
        "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0NnLcLCDUSevd5krC+fG\n" +
        "C3Gv+ohU/Pyq90ZipotTbAhtHGRYHvxDIKDmDsLlVEFVgucCnuASkx74qAZ19Bsh\n" +
        "9qEup0Q/YqI5nLUkgfKd26ZiUxxiUTTMCJY+S1e8qYYi6+r/lNpn1FP5570akr4U\n" +
        "PIM4Nfsba3zutkt8gu4dFnGaAh+EyivD0LoMHYy8VSyJvMeFLIxpKk9aTgJy1El6\n" +
        "CI1PSZj70Jpz/4Lb9G6QT+7o0wzpD6EczLK9UrOvxfKmClGu+jCy9eviKCpH+ktG\n" +
        "+arR51Xlqi9su/UV3g9kRloEB62cFU9nIHhnFH2gqat1gFHnC+w7cEJ8+C9NKXia\n" +
        "kwIDAQAB\n" +
        "-----END PUBLIC KEY-----");
    let msg = document.getElementById('message');
    var encrypted = publicKey.encrypt(msg.value, "RSA-OAEP", {
            md: forge.md.sha256.create(),
            mgf1: forge.mgf1.create()
        });
    msg.value = forge.util.encode64(encrypted);
    return true;
}
</script>
```

(graph 5-3)

```
@post('/message')
def send():
    sender = request.forms.get('sender')
    receiver = request.forms.get('receiver')
    message = request.forms.get('message')
    cipher = PKCS1_OAEP.new(key, hashAlgo=SHA256)
    decrypted_message = cipher.decrypt(b64decode(message))
    message = str(decrypted_message, encoding='utf - 8')
    return model.message_send(sender, receiver, message)
```

(graph 5-4)

```python
def message_send(sender, receiver, message):
    usersDB = sql.SQLDatabase('database.db')

    privatekey = RSA.importKey(usersDB.get_privatekey(sender))
    publickey = RSA.importKey(usersDB.get_publickey(receiver))

    cipher = Cipher_pkcs1_v1_5.new(publickey)
    message = message.encode('utf-8')
    message_encode = base64.b64encode(cipher.encrypt(message)).decode('utf-8')
    signer = Signature_pkcs1_v1_5.new(privatekey)
    digest = SHA.new()
    digest.update(message)
    sign = signer.sign(digest)
    signature = base64.b64encode(sign).decode('utf-8')

    messageDB = sql.SQLDatabase('database.db')
    messageDB.send_message(sender, receiver, message_encode, signature)
```

(graph 5-5)

```python
def message_check(username, friend):
    usersDB = sql.SQLDatabase('database.db')
    friendsDB = sql.SQLDatabase('database.db')
    result = friendsDB.check_friend(username, friend)
    if result:
        messagesDB = sql.SQLDatabase('database.db')
        messages = messagesDB.get_message(username, friend)

        publickey = RSA.importKey(usersDB.get_publickey(friend))
        privatekey = RSA.importKey(usersDB.get_privatekey(username))

        decode_messages = []
        if len(messages) == 0:
            return page_view("message", username=username, friend=friend, messageslist=decode_messages)
        for message in messages:
            verifier = Signature_pkcs1_v1_5.new(publickey)
            signature = messagesDB.get_signature(message)
            message = message.encode('utf-8')
            cipher = Cipher_pkcs1_v1_5.new(privatekey)
            message = cipher.decrypt(base64.b64decode(message), 'fail')
            hsmsg = SHA.new()
            hsmsg.update(message)
            is_verify = verifier.verify(hsmsg, base64.b64decode(signature))
            if is_verify:
                decode_messages.append(message.decode('utf-8'))
        return page_view("message", username=username, friend=friend, messageslist=decode_messages)
    else:
        return page_view("valid")
```

(graph 5-6)

Our github project link: https://github.com/bonaparte233/INFO2222-Project

Reference

1. Delicious Brains (2022, March 16). How to Create Your Own SSL Certificate Authority for Local HTTPS Development. Retrieved from https://deliciousbrains.com/ssl-certificate-authority-for-local-https-development/
2. Medium (2017, January 29). RSA Encryption with JS & Python. Retrieved from https://medium.com/@DannyAziz97/rsa-encryption-with-js-python-7e031cbb66bb