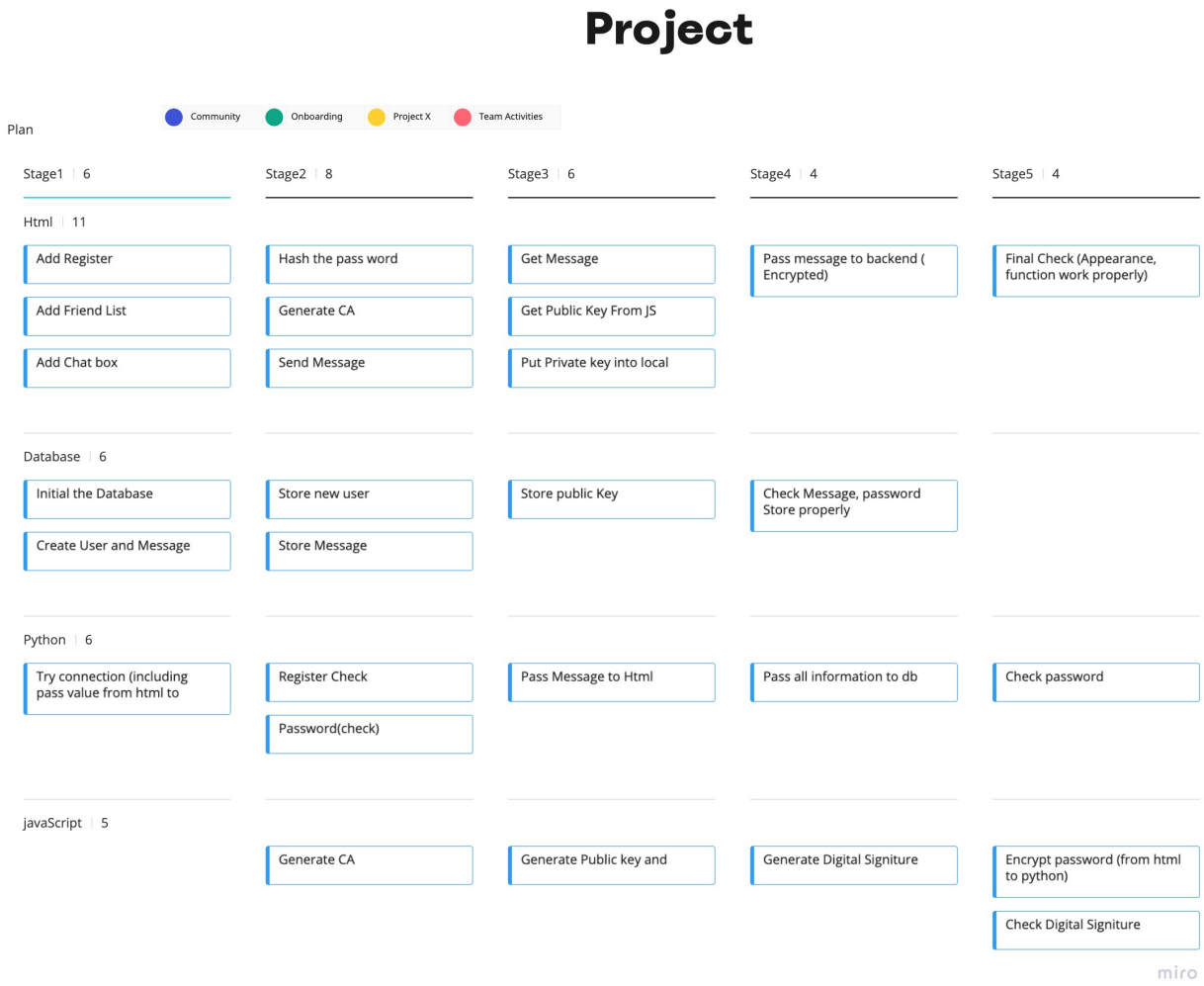
INFO2222 report

Group name: RE03_Team5
Name: Sangyu Shi (Frank) sid:500033689
Wenxi Zheng(Edward) sid:500061017

Summary



Contribution:
Each team member contributes substantially equally to the entire project.
For the co-developed part: Sangyu Shi and Wenxi Zheng jointly solve the problem of asymmetric encryption. Sangyu Shi quoted and rewritten the node-rsa section. At the same time, it uses this part of the encrypted information and transmits it back to the backend. Edward is mainly responsible for taking out the contents of the database and sending it back to the front section to complete the decryption. Sangyu Shi is responsible for asymmetric encryption

between the password and the server. Edward is responsible for generating the electronic signature.
Personal Contribution:
Wenxi Zheng: The database is created and the connection between the database and the backend is completed. It realizes the data interaction between the database and python.The generation of the CA is completed.

Sangyu Shi: Completed the production of the front-end web page, and realized the function of registering and sending and receiving messages. Implemented the hash of password. Improved ext files.

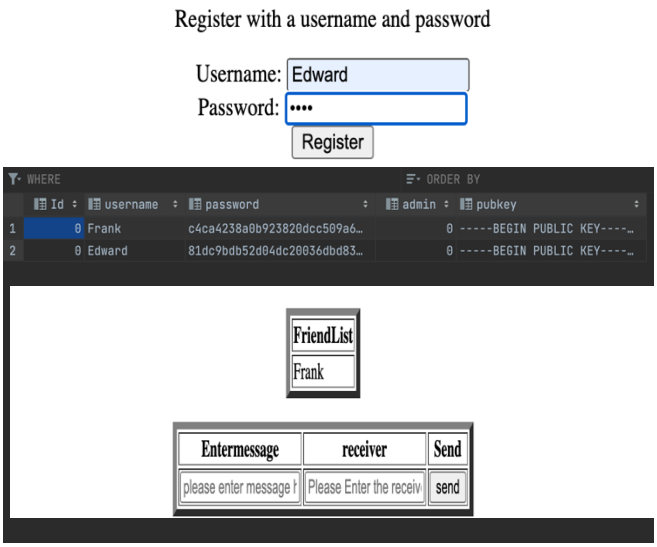
Body of report

Part1: Properly store passwords on the server

Due to the need to ensure that the password cannot be stored in clear text in the server. We md5 encrypted the password on the front end. and passed it back to the database. As shown in the figure(1) below, the function will read the user input value with id=password and encrypt it with md5 when submitting. Storage effect such as figure(2) (Log in as Edward).

```
try {
    const register_btn = document.getElementById( elementId: 'register');
    if (register_btn != null) {
        console.log("register button found");
        register_btn.addEventListener( type: 'click', listener: () => {
            hash.update(password.value);
            const digest = hash.digest('hex');

            password.value = digest;
            const username = document.getElementById( elementId: 'username');
            const rsa = new nodeRSA( key: {b: 512});
            const public_key = rsa.exportKey( format: 'pkcs8-public');
            const private_key = rsa.exportKey( format: 'pkcs8-private');
            localStorage.setItem(username.value + '_public_key', public_key);
            localStorage.setItem(username.value + '_private_key', private_key);

            document.getElementById( elementId: 'pubKey').value = public_key;
            console.log("added public key to the server")
        });
    }
} catch (e) {
    console.log("register error");
}
```

Figure(1)

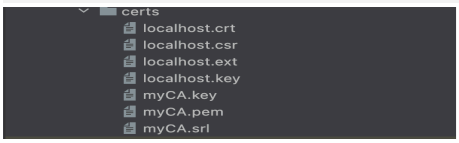


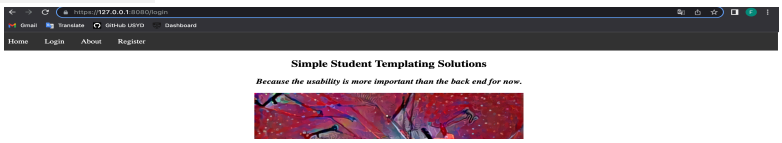
Figure(2)

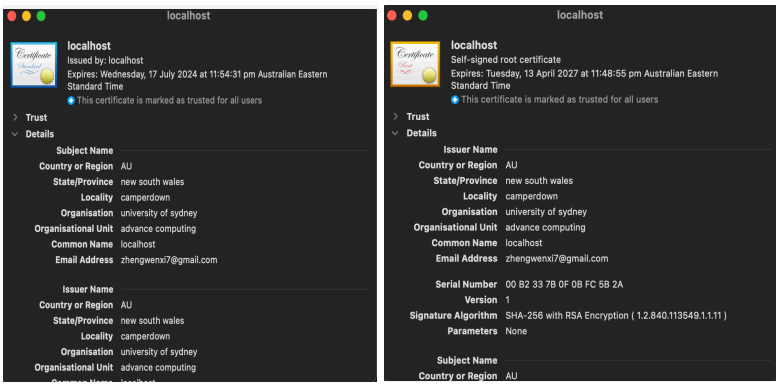
Reference: https://blog.csdn.net/u011627980/article/details/60871604

Part2: check server's certificate

We use xxx to generate the CA and put it in the program. When the frontend (web page) is opened, the certificate is requested from our backend and verified. If successful verification will show us safe. Otherwise it will show us unsafe.

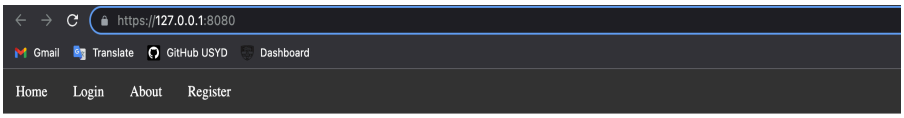






Part3 Securely transmitting a pwd to server :

In this regard we use HTTPS to ensure the security of the transmission. HTTPS uses an SSL certificate to verify the identity of the server and encrypt the communication between the browser and the server. This ensures that the passwords are encrypted and mutually authenticated during transmission. The way we implement Https is to implement it when generating the CA, and complete the operation when generating the ext file. The specific code and results are as follows. Add compared to website content (IP.1=127.0.0.1)



Simple Student Templating Solutions

Because the usability is more important than the back end for now.

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName = @alt_names

[alt_names]
DNS.1 = localhost
IP.1=127.0.0.1
```

Part4 Properly check whether password is correct

The main problem of verifying the correctness of the password is the secure transmission of the password when logging in and the comparison between the login password and the password in the database. The secure transmission part is consistent with the registration, using https to ensure the security of password transmission. When comparing passwords, since the passwords are also hashed when logging in, only the hash value is compared, not the plaintext, which ensures that there will be no leakage when comparing passwords.

```python
def check_credentials(self, username, password):
    sql_query = """
            SELECT 1
            FROM Users
            WHERE username = '{username}' AND password = '{password}'
        """

    sql_query = sql_query.format(username=username, password=password)

    # If our query returns
    self.execute(sql_query)
    if self.cur.fetchone():
        return True
    else:
        return False
```

Check

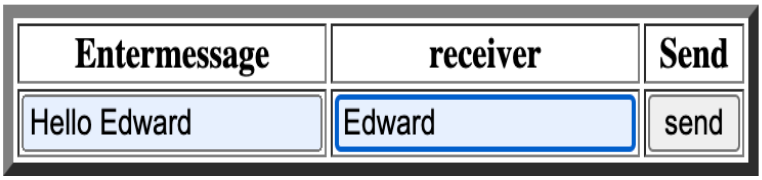
```javascript
try{
    const login_btn = document.getElementById( elementId: 'login');

    if (login_btn != null){
        console.log("login button found");
        login_btn.addEventListener( type: 'click', listener: () => {
            hash.update(password.value);
            const digest = hash.digest('hex');

            password.value = digest;

            sessionStorage.setItem('username', document.getElementById( elementId: 'username').value);
        })
    }
}
catch(e){
    console.log("login error");
}
```

transmitting password

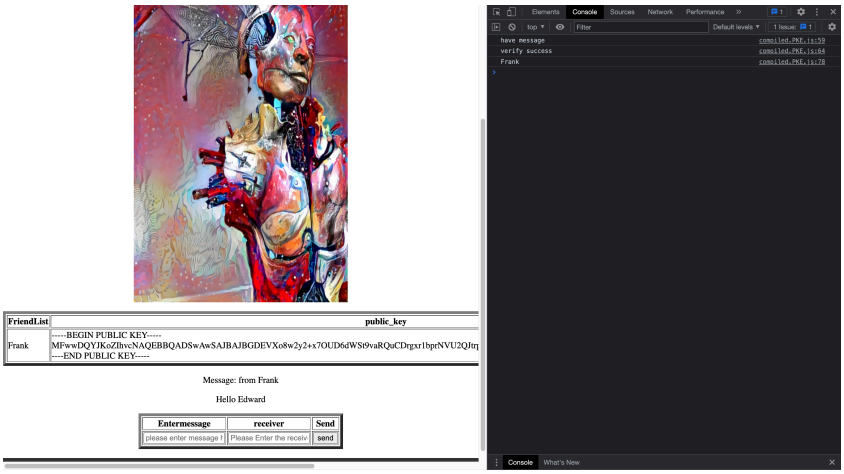
Part5 Message Encryption

The secure transmission between messages is mainly divided into three parts. First, the sender can encrypt the message and guarantee that only the receiver can decrypt it. Second, messages cannot appear in plaintext on the backend. Third, the recipient can verify the identity of the sender. Complete the first and second parts mainly using unpaired encryption. The sender encrypts it with the receiver's public key and sends it to the backend. After the receiver obtains the encrypted information from the backend, it will decrypt it with its own private key. This ensures that the encrypted information is displayed at the back end because the message has been encrypted. The server cannot directly get the specific content of the message. At the same time, since the private key used for decryption is only stored locally in the receiver, this means that only the receiver can decrypt this information. A digital signature scheme is used for how to verify the identity of the sender. When the sender sends a message, it will use its own private key to encrypt the text that has been encrypted by the other party's public key. Generate digital signature. When the receiver receives the message, it first needs to decrypt it with the public key of the other party to confirm whether the sender is the intended sender. After the verification is successful, you can use your own private key to decrypt to obtain the actual information. （How to generate public key, private key and digital signature will be explained in appendix.）

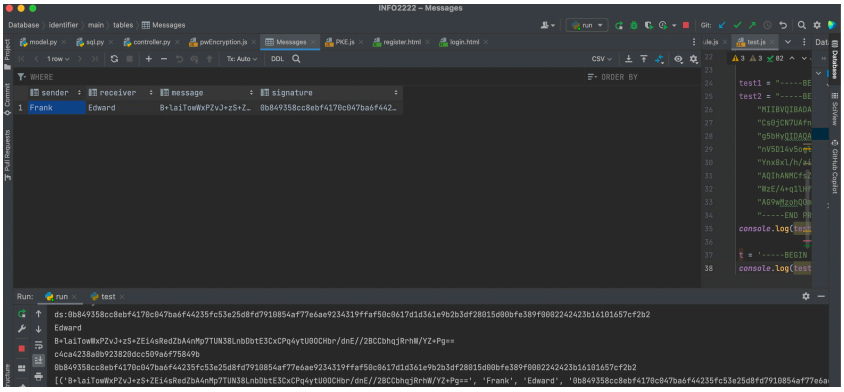
Send Message



Receive Message (The console shows verify and whether has message)



Message and Signature in database



Bonus (optional)

limitati Since the generated CA is generated locally, it has not been certified by an authority. So HTTPS is hard coding. Although Https is achieved in theory, the performance in the program is equivalent to self-authentication, and has no practical effect.on discussion

Appendix

Here will explain how to generate private key, public key, digital signature and other details.

```
const nodeRSA = require('node-rsa')
```

We used the RSA unpaired encryption algorithm. All references are made before using the package.

```
const publicKey = key.exportKey( format: 'pkcs8-public')

const privateKey = key.exportKey( format: 'pkcs8-private')
```

Generate Public key and private key.

```
const btn2 = document.querySelector( selectors: "#btn2")
  btn2.addEventListener( type: 'click',  listener: () => {
  let pk_value = getPublicKey()
  const pubkey = document.querySelector( selectors: "#pubkey")
  pubkey.value = pk_value
  localStorage.setItem('key', key)
  localStorage.setItem('publicKey', pk_value)
  localStorage.setItem('privateKey', privateKey)
})
```

Put both keys locally.

```
@post('/register')
def post_register():
    '''
        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')
    public_key = request.forms.get('pubKey')
    print(username)
    print(password)
    print(public_key)
```

The public key will also be transferred to the database

WHERE ORDER BY

	Id	username	password	admin	pubkey
1	0	Frank	c4ca4238a0b923820dcc509a6…	0	-----BEGIN PUBLIC KEY----…
2	0	Edward	81dc9bdb52d04dc20036dbd83…	0	-----BEGIN PUBLIC KEY----…

Storage effects in the database.

```
const friend = document.getElementById( elementId: 'friend')

const received_message_title = document.getElementById( elementId: 'received_message_title')
const received_message = document.getElementById( elementId: 'received_message')

const digital_signature = document.getElementById( elementId: 'ds')
if (received_message.innerHTML != ""){
    console.log("have message")
    const sender_public_key = document.getElementById( elementId: 'friend_public_key').innerHTML
    const digital_signature = document.getElementById( elementId: 'ds')
    const verify = new nodeRSA(sender_public_key)
    if (verify.verify(Buffer.from(received_message.innerHTML), digital_signature.value, source_encoding: 'utf8', signature_encoding: 'hex')){
        console.log('verify success')
        const current_user_private_key = localStorage.getItem( key: sessionStorage.getItem( key: 'username')+'_private_key')
        const decrypt = new nodeRSA(current_user_private_key)
        const decrypt_msg = decrypt.decrypt(received_message.innerHTML, encoding: 'utf8')
        received_message.innerHTML = decrypt_msg
    }
    else{
        console.log('verify fail')
    }
}
else{
    received_message_title.setAttribute( qualifiedName: 'hidden', value: 'true')
    received_message.setAttribute( qualifiedName: 'hidden', value: 'true')
}
console.log(friend.innerHTML)
```

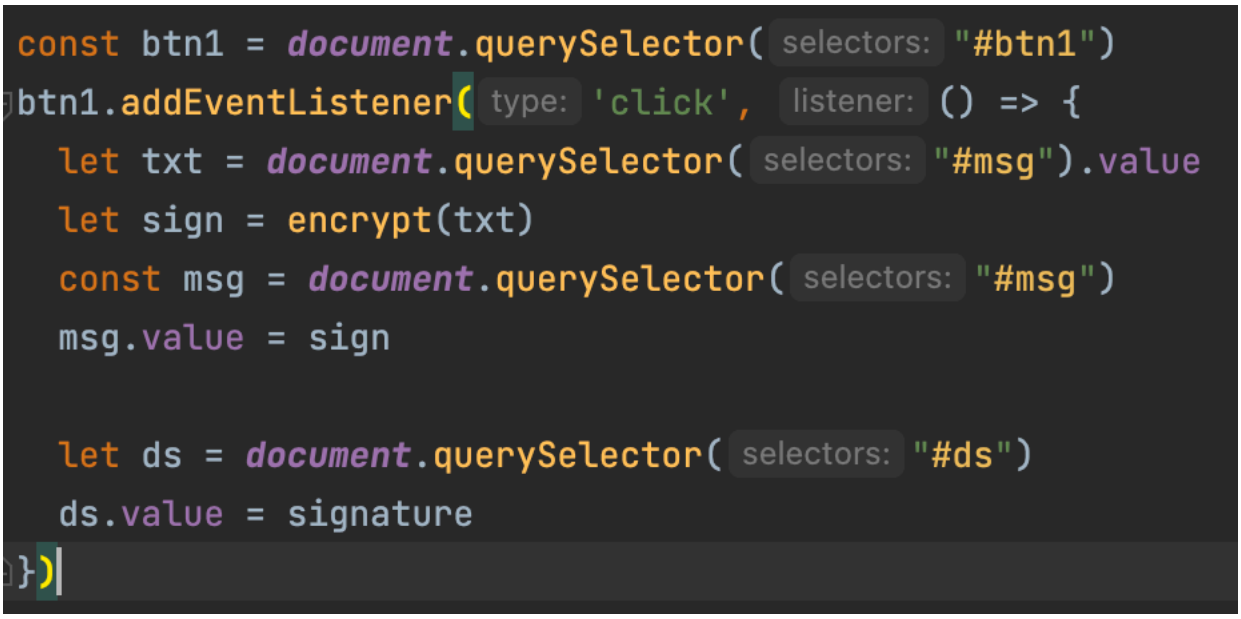
If there is message, verify first.

```
const friend_public_key = document.getElementById( elementId: 'friend_public_key')

try{
    const send_btn = document.getElementById( elementId: 'send')
    if (send_btn != null){
        send_btn.addEventListener( type: 'click', listener: () => {
            const msg = document.getElementById( elementId: 'msg')
            console.log(friend_public_key.innerHTML)
            const encrypt = new nodeRSA(friend_public_key.innerHTML, format: 'pkcs8-public')
            const encrypt_msg = encrypt.encrypt(msg.value, encoding: 'base64')
            msg.value = encrypt_msg

            const current_user = sessionStorage.getItem( key: 'username')
            const current_user_private_key = localStorage.getItem( key: current_user+'_private_key')

            const signature = new nodeRSA(current_user_private_key)
            const sign = signature.sign(Buffer.from(encrypt_msg), encoding: 'hex')
            document.getElementById( elementId: 'ds').value = sign
            document.getElementById( elementId: 'sender').value = current_user

        })
    }
}
catch(err){
    console.log("message send error")
}
```

If there is message, decrypt it after verify.

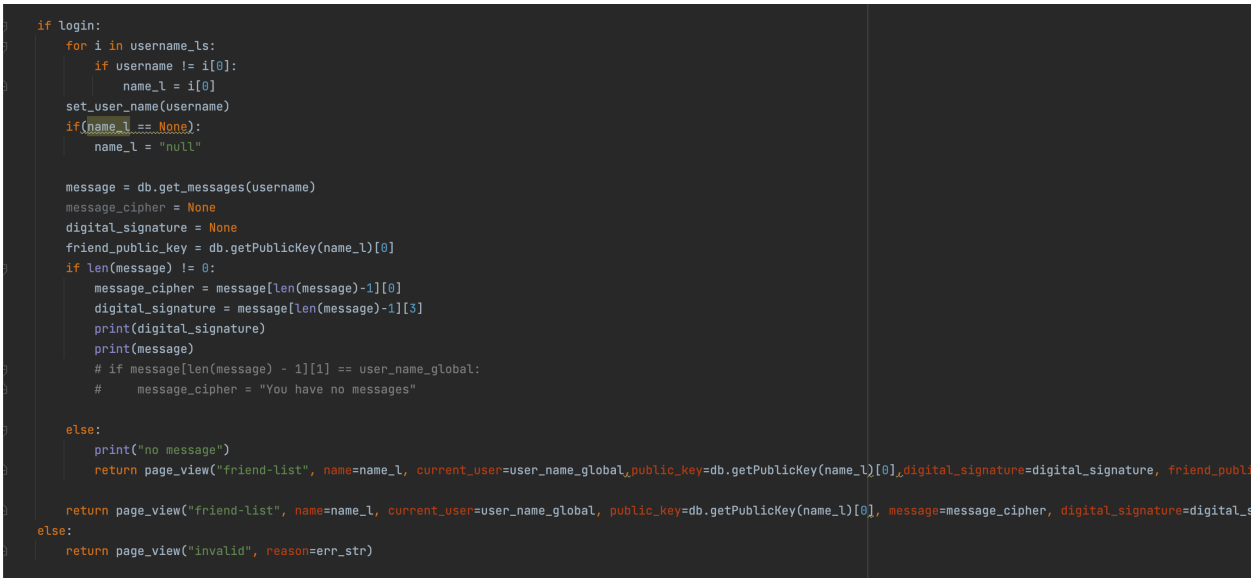
```
function encrypt(data) {
  const pubKey = new nodeRSA(publicKey, format: 'pkcs8-public')
  return pubKey.encrypt(Buffer.from(data), encoding: 'base64')
}


function decrypt(data) {
  const priKey = new nodeRSA(privateKey, format: 'pkcs8-private')
  return priKey.decrypt(Buffer.from(data, 'base64'), encoding: 'utf8')
}


function signRSA(data) {
  const priKey = new nodeRSA(privateKey, format: 'pkcs8-private')
  return priKey.sign(Buffer.from(data), encoding: 'hex')
}


function verifyRSA(decrypt, signs) {
  const pubKey = new nodeRSA(publicKey, format: 'pkcs8-public')
  return pubKey.verify(Buffer.from(decrypt), signs, source_encoding: 'utf8', signature_encoding: 'hex')
}
```

Specific functions for encryption and decryption, generation of digital signatures and verification of digital signatures.



```
const btn1 = document.querySelector( selectors: "#btn1")
btn1.addEventListener( type: 'click',  listener: () => {
  let txt = document.querySelector( selectors: "#msg").value
  let sign = encrypt(txt)
  const msg = document.querySelector( selectors: "#msg")
  msg.value = sign

  let ds = document.querySelector( selectors: "#ds")
  ds.value = signature
})
```

Encrypted packaging when sending messages



```
if login:
    for i in username_ls:
        if username != i[0]:
            name_l = i[0]
    set_user_name(username)
    if(name_l == None):
        name_l = "null"

    message = db.get_messages(username)
    message_cipher = None
    digital_signature = None
    friend_public_key = db.getPublicKey(name_l)[0]
    if len(message) != 0:
        message_cipher = message[len(message)-1][0]
        digital_signature = message[len(message)-1][3]
        print(digital_signature)
        print(message)
        # if message[len(message) - 1][1] == user_name_global:
        #     message_cipher = "You have no messages"

    else:
        print("no message")
        return page_view("friend-list", name=name_l, current_user=user_name_global, public_key=db.getPublicKey(name_l)[0], digital_signature=digital_signature, friend_publi

    return page_view("friend-list", name=name_l, current_user=user_name_global, public_key=db.getPublicKey(name_l)[0], message=message_cipher, digital_signature=digital_s
else:
    return page_view("invalid", reason=err_str)
```

Log in check. Call database to do login check. This part also leads to chat page. So it will get Message from database and post it to html. (Some message is hidden in Html but can be get by JavaScript)

```
def register_check(username, password, pub):
    '''
        login_form
        Returns the view for the login_form
    '''
    register = db.getUsername()
    for name in register:
        if username == name[0]:
            return page_view("invalid", reason="username already exists")
    # public_key, private_key = rsa.newkeys(1024)
    # pub = public_key.save_pkcs1().decode()
    # priv = private_key.save_pkcs1().decode()
    # private_key = "-----BEGIN RSA PRIVATE KEY-----\n" + "MIIBPAIBAAJBAJXPd3FZqlVF
    #
    # private_key = rsa.PrivateKey.load_pkcs1(private_key)
    # password = rsa.decrypt(base64.b64decode(password.encode()), private_key)
    db.add_user(username, password,pub)
    return page_view("success", username=username)
```

Check duplicate username while log in.