THE UNIVERSITY OF SYDNEY

2021 S2C INFO2222

Project Report

Security part

Group name: Re08
Name: Shutong Li (Lavender Li);
Yuexiao Zou (Tom Zou).

Summary

Summary of technology stack:

Front-end: Vite + Vue3.0 + VueRouter 4 + TailwindCSS + HeroIcons
CDN library: axios + SweetAlert2

Back-end: Python + Flask
Database: Sqlite

Finished Items:

1. Properly store password on the server:
The password is encrypted with the sha256 algorithm after being salted, and then stored in the database after being shorted the length of the hash value.
contributions: lavender：all FE logic and pages. BE for 50%.
Tom: BE for 50%, DB

2. When log in, first check server's certificate:
When logging in, we first check if the CA public key exists, then check if the CA public key is correct. Then return the corresponding result to the user.
contributions: lavender：FE pages.
Tomr: BE.

3. Securely transmitting a pwd to server:
We divide the login stage into account stage and password stage, so that the server can send random value on the account stage and let the client encrypt the password using the registration salt and encrypt it again with the server's random value. then transmit the password
contributions: lavender：all FE logic and pages. BE for 40%.
Tom: BE for 60%.

4. Properly check whether password is correct:
The server combines the random value corresponding to the username with the password in the database and encrypts it with the sha256 algorithm. Finally, compare the encrypted ciphertext with the ciphertext obtained from the client.
contributions: lavender：all FE logic and pages. BE for 40%.
Tom: BE for 60%, DB.

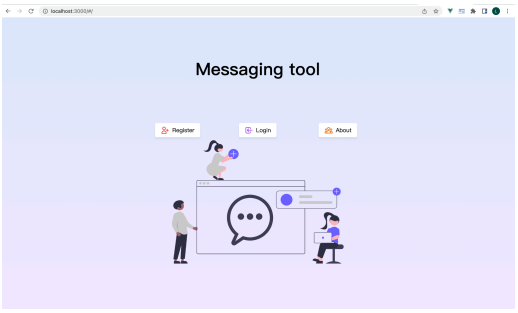
5. Securely transmitting the message from A to B, even the server who can forward communication transcript cannot read the message, or modify the ciphertext:
The user encrypts the information to be sent with the friend's public key, and then the client forwards it to the friend.
contributions: lavender：all FE logic and pages. BE for 50%.
Tom: BE for 50%.

6. Clarification of the report.
contributions: lavender for 60%.
contributions: Tom for 40%.

Body of report

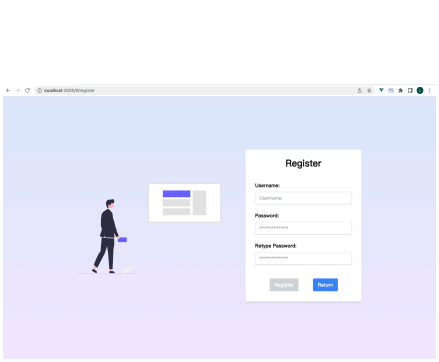
0. Initial the Web Application

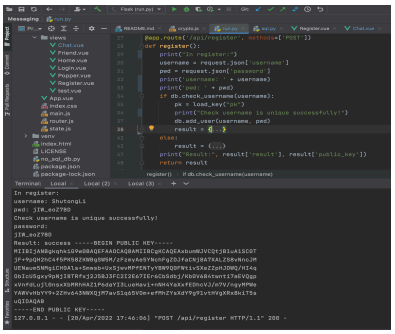
Front-end: a home page with three buttons: register, login, about.
“Register”: click to jump to the register page when register finished will automatically jump to the login page.
“Login”: click to jump to the login page when login finished will automatically jump to the friend-list page.
“About”: showing some information about this web application

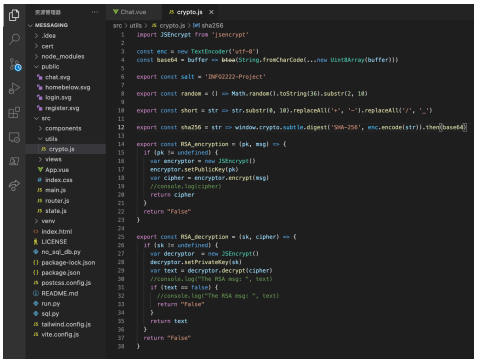


1. Properly store password on the server:

Front-end: Three input blankets about “username”, “password” and “Retype Password”. Users need to type in twice the same password and unique username then can register successfully. The password first will check in FE that the two inputs are the same, and will add a salt then using SHA256 to encrypt and finally shorten the ciphertext. Then send (username, ciphertext) by post request to BE.
Back-end: Initial the sqlite database, store the encrypted password from the FE.
Logic and Reason: The reason for using sha256 is that we want to encrypt the user's password, so that the plaintext is not exposed. Salt is used to further improve the security of our ciphertext and prevent others from decrypting our password. The reason we use short is to improve the reusability of our salt and prevent others from seeing the law of our salt. Our goal is to properly store the password on the server. So that, we only transmit salted passwords encrypted by the sha256 algorithm. Then store the cipher text securely in our database.
Outcome:





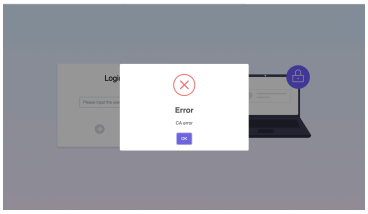


2. When log in, first check server’s certificate:

Front-end: 1.Client sends a http certification request in the register stage. 2.If the CA public is not found. The login page will show an CA error using “Swal”. 3.If the CA public key is not correct, the login page will show an alert using “Swal”. 4. client send a encryption key using CA public key
Back-end: 1.Sending the CA public key back to the client after receiving the client http certification request. (as CA information) 2.send the response to the client after decrypting the client message.

Logic and Reason: When logging in, the user checks to see if the CA has a public key, and returns a no-certificate warning if not. If there is, the user will encrypt it with the public key of the CA, and then the server will decrypt it to return whether the CA is valid.

Outcome:



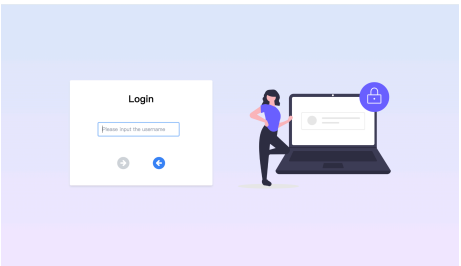

```
def generate_cert():
    key = RSA.generate(2048)
    private_key = key.export_key()
    public_key = key.publickey().export_key()
    directory = "cert"
    if not os.path.isdir(directory):
        os.makedirs(directory)
    with open("cert/private.pem", "wb") as private_file:
        private_file.write(private_key)
    with open("cert/public.pem", "wb") as public_file:
        public_file.write(public_key)
```

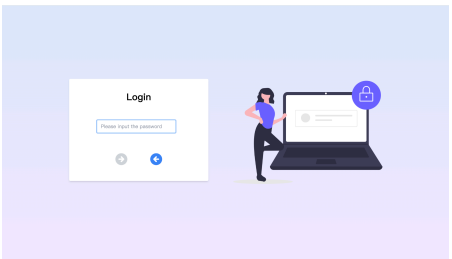
3. Securely transmitting a pwd to server (leveraging secure protocols or design the secure transmission properly)

Front-end: For secure transmission of the pwd, we choose to use two steps to login. First step requires the user to type in the username, in this step, BE will check if the user exists and return a random salt which will be used in the second step. For security, regardless if the user exists or not, we both return the random to FE and then show the step two page because we cannot let the user know if the user exists. Second step requires the user to type in the password, in this step, FE will first using the same way as register encrypter to "SHORT(SHA256(typed password + SALT))" as "result" and the "SHA256(result + random)" result will using post request send to BE.

Logic and Reason: Techniques and tools: salt, sha256. We used the techniques and tools SHA256 from JavaScript library Crypto, salt from Python library Random and short. Using hash with salt twice, can ensure the security in both transmit period and database. The "random" from BE can ensure that every time login will transmit a different ciphertext. The reason we use short is to improve the reusability of our salt and prevent others from seeing the law of our salt.

Outcome:






```
@app.route('/api/login-first', methods=['POST'])
def login1():
    """
    # check CA
    cipher = request.json['cipher']
    sk = load_key("sk")
    print(cipher)
    message = decrypt_data(sk, cipher)
    if message == "I am client":
        print("Success")
    else:
        result = {
            'result': "Is not a Certificate Authority",
            'username': None,
            'random': None
        }
        return result
    print(message)
    username = request.json['username']
    random_str = ''.join(random.choice(string.ascii_letters + string.digits) for _ in range(8))
    if db.check_username(username):
        result = {
            'result': False,
            'username': None,
            'random': random_str
        }
        return result
    else:
        username_random[username] = random_str
        print(username, random_str, username_random)
        result = {
            'result': True,
            'username': username,
            'random': random_str
        }
        return result
```

```
@app.route('/api/login-second', methods=['POST'])
def login2():
    """
    # input
    username = request.json['username']
    user_pwd = request.json['password']
    print("db username : ")
    print(username)
    print("db password : ")
    print(user_pwd)
    public_key = request.json['public_key']
    username_public[username] = public_key
    print("db public key array list: ")
    print(username_public)

    # encry
    random_str = username_random[username]
    print("db random_str : ")
    print(random_str)
    db_pwd = db.get_pwd(username)[0][0]
    print("db password : ")
    print(db_pwd)
    db_pwd_salt = db_pwd + random_str
    print("db db_pwd_salt : ")
    print(db_pwd_salt)
    data_sha = base64.b64encode(hashlib.sha256(db_pwd_salt.encode('utf-8')).digest()).decode('ascii')
    print("db data_sha : ")
    print(data_sha)
    # RSA encryption
    # encrypt_data = encrypt_data(public_key, db_pwd_salt)
    if user_pwd == data_sha:
        return "true"
    else:
        return "false"
```

4. Properly check whether password is correct

Front-end: Same as point 3, we choose to use two steps to login. In second step, when the user to type in the password finished and clicked next button, FE will first using the same way as register encrypter to "SHORT(SHA256(typed password + SALT))" as "result" and the "SHA256(result + random)" result will using post request send to BE.

Back-end: First step: 1. check if the username exists or not; 2. generate the random string and store it in BE(regardless if the user exists or not because we cannot let the user know if the username is correct) Second step: 1. get the user password from database 2. using the same encrypter way to encrypt the stored password. 3. check the database password encrypted result and the encrypted typed password from FE is the same or not.

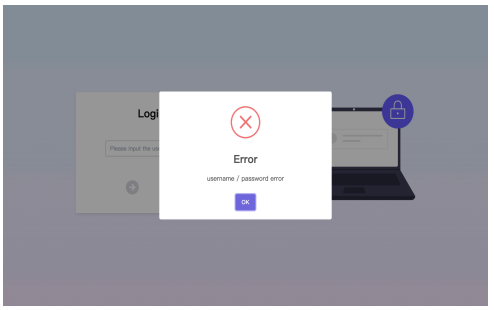
Techniques and tools: base64, sha256 and salt. We ensure confidentiality by hashing the data with sha256 from the python library hashlib. Then we mix the data through salt to further improve the security of the data and prevent attacks such as rainbow tables. Since only ASCII characters can be used in some systems. Base64 is a method used to convert non-ASCII character data into ASCII characters.

Outcome:

```
async function pass() {
  if (!input) return
  if (!random) { // first
    key = 'pk_' + input
    pk = Cookies.get(key)
    const msg = "I am client";
    const cipher = RSA_encryption(pk, msg);
    if (cipher == "False") {
      Swal.fire('Error', 'CA error', 'error')
    }

    axios.post('/api/login-first', {
      'username': input,
      'cipher':cipher
    })
    .then(function (response) {
      checkResult = response.data['result']
      if (checkResult == "Is not a Certificate Authority"){
        Swal.fire('Error', 'CA error', 'error')
      }
      if (checkResult == true) {
        username = response.data['username']
      } else {
        username = 'there is error'
      }
      random = response.data['random']
    })
    .catch(function (error) {
      console.log(error);
    });
```

```
} else { //second
  console.log('second time here:')
  if (username != 'there is error') {
    await getKeys()
    axios.post('/api/login-second', {
      'username': username,
      'password':  await sha256(short(await sha256(input + salt)) + random),
      'public_key': publicKey
    })
    .then(function (response) {
      mes = response.data
      console.log(mes)
      if (mes == true) {
        success(username)
      } else {
        Swal.fire('Error', 'username / password error', 'error')
      }
    })
    .catch(function (error) {
      console.log(error);
    });
  } else {
    Swal.fire('Error', 'username / password error', 'error')
  }
  random = ''
}
input = ''
}
```



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

Techniques and tools: RSA

We used the asymmetric encryption algorithm RSA. This is from the JSEncrypt library in JavaScript. The purpose of using this algorithm is to asymmetrically encrypt the information sent by the user, thereby ensuring the confidentiality of the ciphertext.

Front-end: We have two pages: friend-list page(homepage after login) and chat page. In the login stage, the FE generates a pair of PK and SK for the user and saves them in session storage and sends the PK to BE to store in the server. In the friend-list page, it shows the user‘s friends, the user can add the friend by clicking the “add friend” button and type username which needs to exist in the system. Each friend card will show the friend name and a chat button. If you want to chat with a friend, just click the chat button at the friend card and then will jump to the chat page. When clicking the chat button, the user state will save that friend name and friend’s PK stored in the server. In the chat page, it shows the chat content and input area. When user A sends a message from the input area, the FE will use the friend user B PK to encrypt the message and then send it to the server. And the user A gets the messages sent by friend user B from the server and then uses his SK to decrypt to get the message.

Back-end: store the users’ PK, store the encrypted messages which also store the corresponding sender and receiver name. Send the received message to the FE.

Outcome:

```
async function postMessDB() {
  axios.post('/api/post_mess_DB', {
    'sender': user.name,
    'receiver': user.friend,
    'message': cipher
  })
      .then(resp => {
        console.log('resp result: ' + resp.data)
      })
      .catch(function (error) {
        Swal.fire('Error', 'He is not login', 'error')
        console.log(error)
      })
}

async function getAllMess() {
  receiveMess = []
  axios.post('/api/get_all_message', {
    'sender': user.friend,
    'receiver': user.name
  })
  .then(resp => {
    let result = resp.data
    for (let i = 0; i < result.length; i++) {
      console.log("receive message" + result[i])
      console.log("sk:" + user.sk)
      let sk = user.sk
      let decrypt = RSA_decryption(sk, result[i])
      console.log("decrypt result" + decrypt)
      receiveMess.push(decrypt)
    }
  })
  .catch(function (error) {
    Swal.fire('Error', 'Get message error', 'error')
    console.log(error)
  })
}

async function sendMessFunc() {
  console.log('friend here: ' + friendPK)
  console.log('current: '+ current)
  console.log('friend: ' + friend)
  current = user.name + ':' + current
  cipher = RSA_encryption(friendPK, current)
  console.log('cipher: ' + cipher)
  await postMessDB()
  sendMess.push(current)
  current = ''
  await getAllMess()
}
```

```
async function getFriendList() {
  axios.post( url: '/api/get-friend-list',  data: {
    'username': user,
  })
      .then(resp => {
        result = resp.data
        if (result != '') {
          console.log("none here:")
          console.log(resp.data)
          console.log(result)
          result = result.trim().split(/\s+/)
          console.log("friend is :  ")
          console.log(result)
          for (let i = 0; i < result.length; i++) {
            console.log(result[i])
            if (friend.indexOf(result[i].toString()) < 0) {
              friend.push(result[i].toString())
            }
          }
          console.log(friend)
        }
      })
      .catch(function (error) {
        console.log(error)
      })
}
```

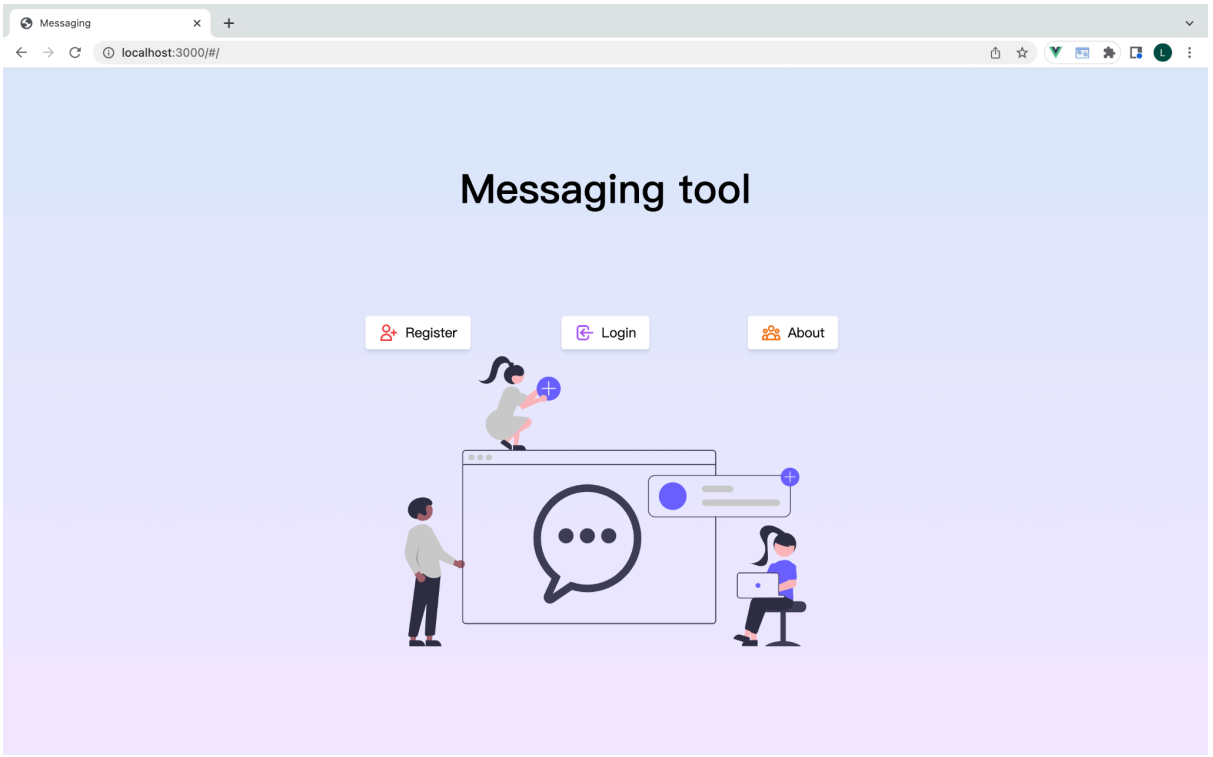
```
async function addFriend() {
  addfriend = username
  axios.post( url: '/api/check-friend',  data: {
    'username': username
  })
      .then(async function (res) {
        console.log('res:' + res.data)
        result = res.data
        if (result === false) {
          Swal.fire('Error', 'username not exists', 'error')
        } else {
          console.log('username add:')
          console.log(addfriend)
          await addFriendToDB()
          if (result.toString() === state.user.name.toString()) {
            Swal.fire('Error', 'cannot add yourself', 'error')
          } else if (friend.indexOf(result.toString()) >= 0) {
            Swal.fire('Error', 'friend already exists', 'error')
          } else {
            friend.push(result.toString())
          }
          console.log(friend)
        }
      })
      .catch(function (error) {
        console.log(error);
      });
  username = ''
}
```

Bonus and Appendix

Front-end page explain:

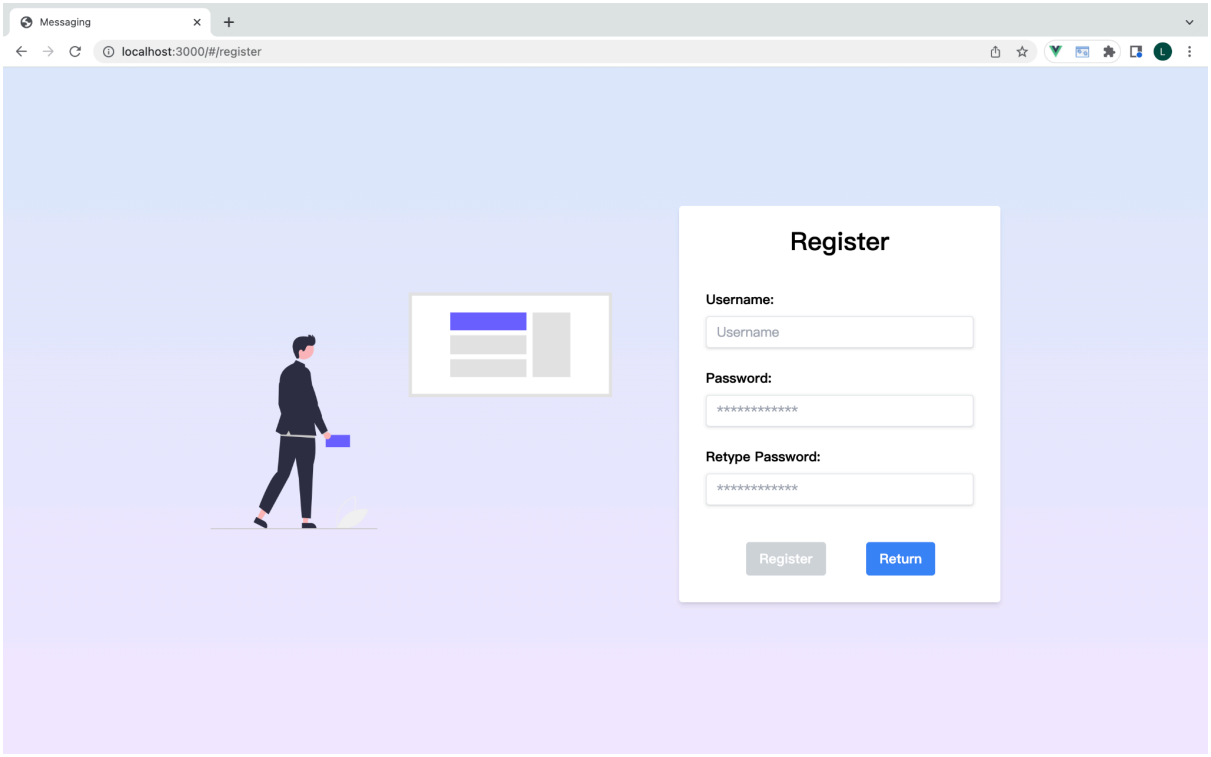
Home page before login

1. **When logined, it will automatically jump to the friend-list page.**
2. **Button:**
   “Register”: jump to register page
   “Login”: jump to login page
   “About”: jump to about page

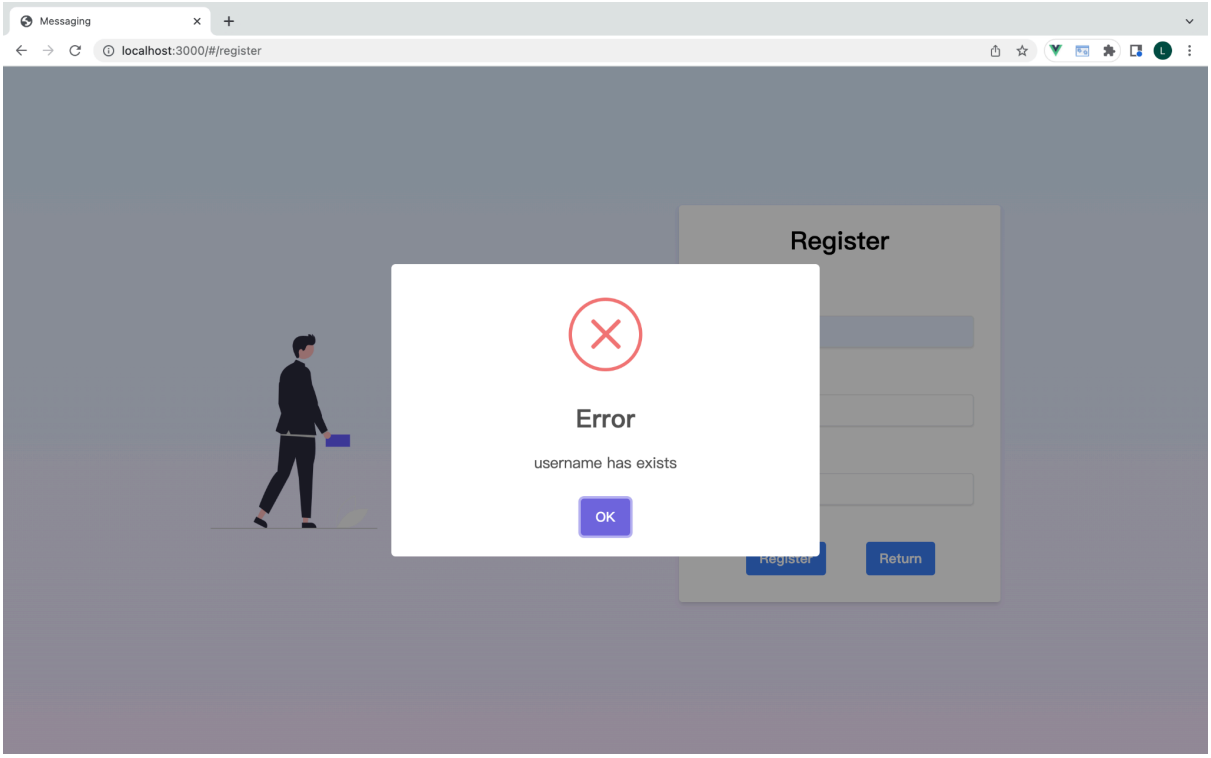


Register page

1. **When logined, it will automatically jump to the friend-list page.**
2. **When the password is the same as the retype password, the “Register” turns blue.**
3. **register successfully will automatically jump to the login page.**
4. **register error will show an alert. (password error, username has registered before)**



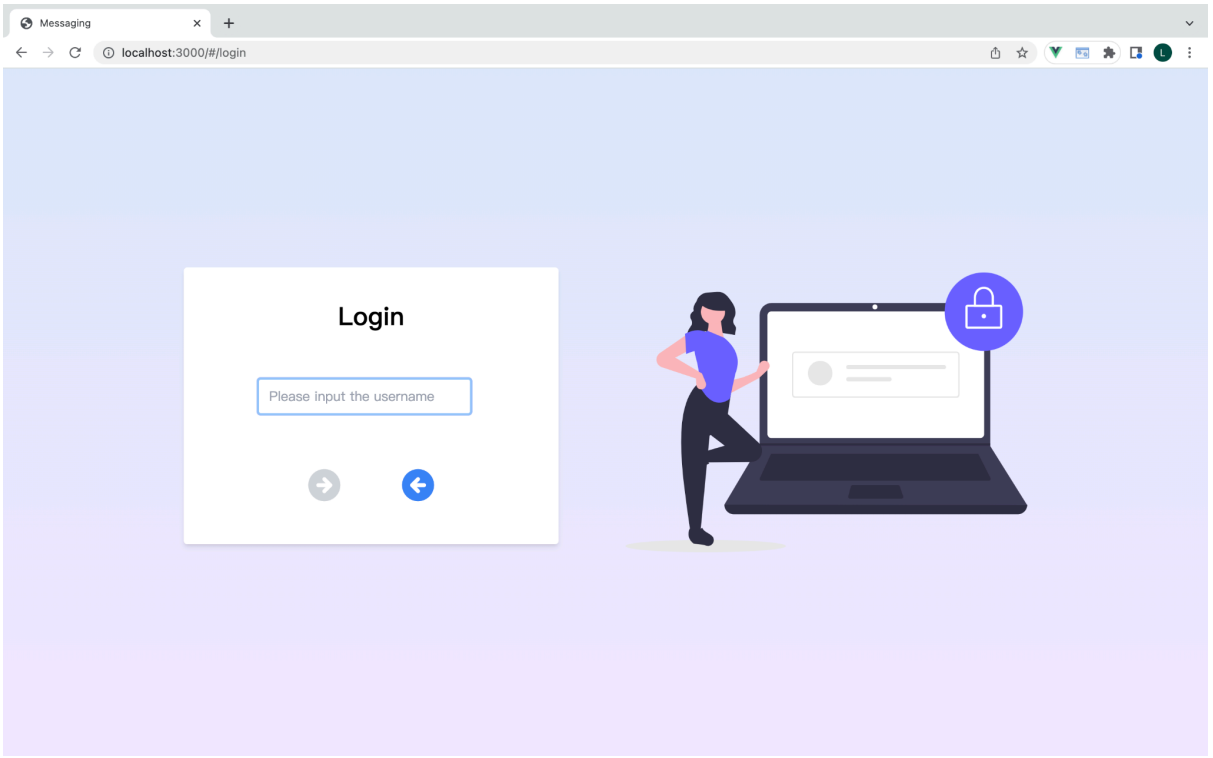
username has been register:

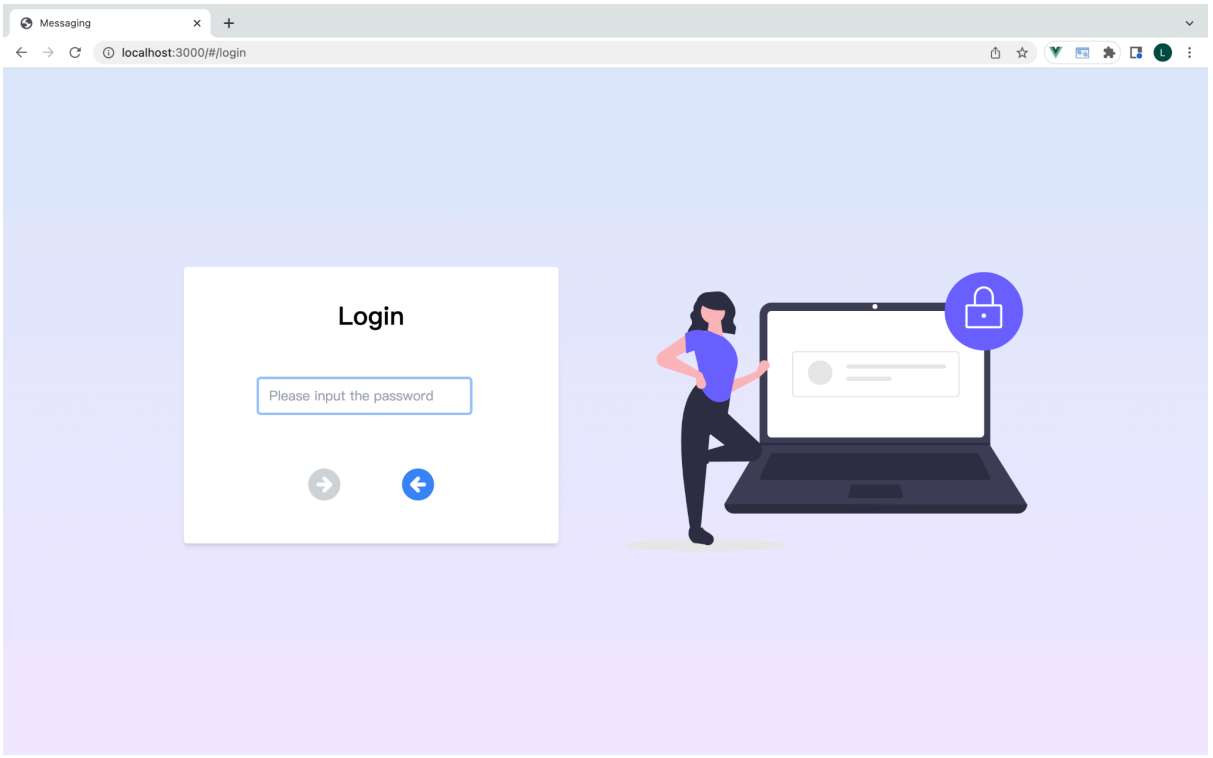


Login page

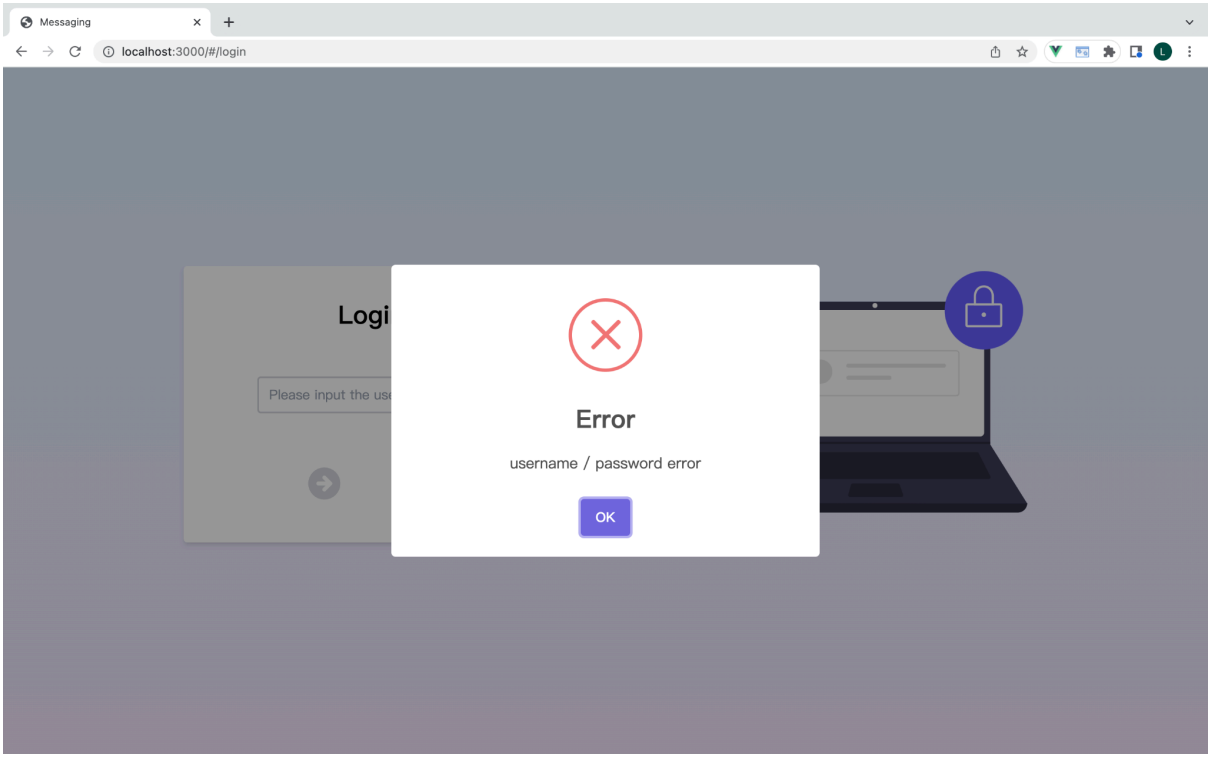
1. **page 1: need to type in username.**
2. **page 2: need to type in password.**
3. **When logined, it will automatically jump to the friend-list page.**
4. **when the input area is not empty, the next button will turn blue.**

5. **when the login faces some error, it will show an alert.**
6. **when login successfully, it will jump to the Friend-list page.**
7. **if the client doesn't find the server's pk, it will show an alert.**
8. **if the CA is invalid, it will show an alert.**

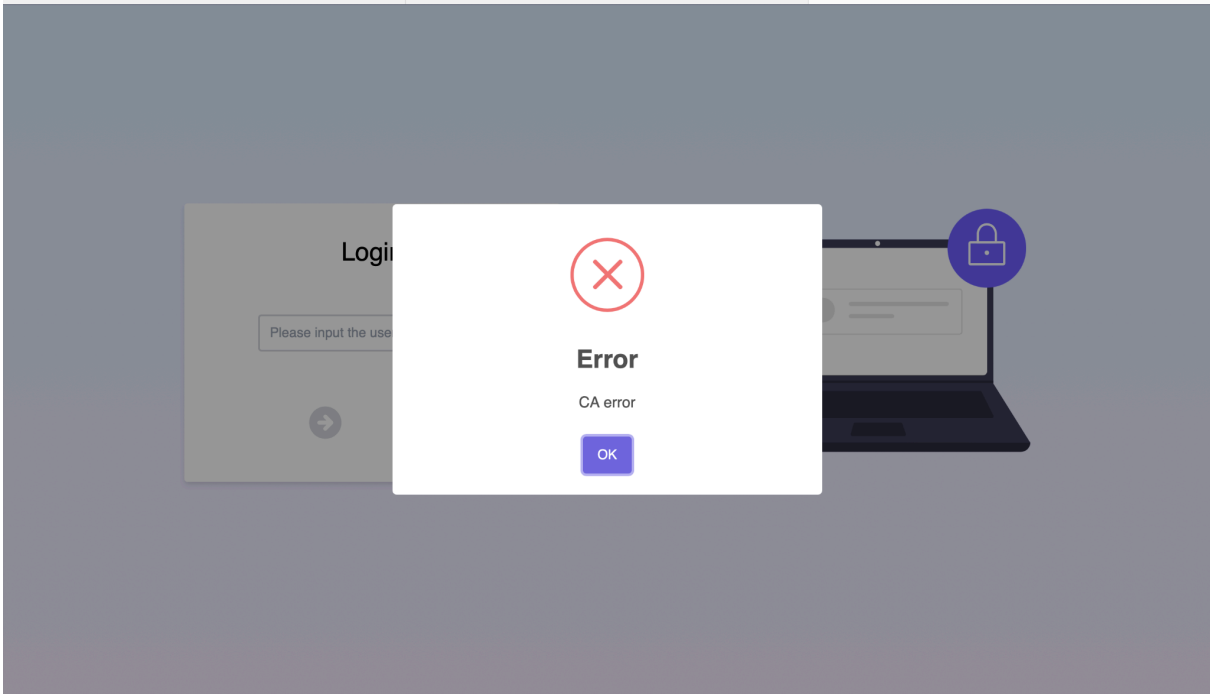




wrong name or password and user not exists:

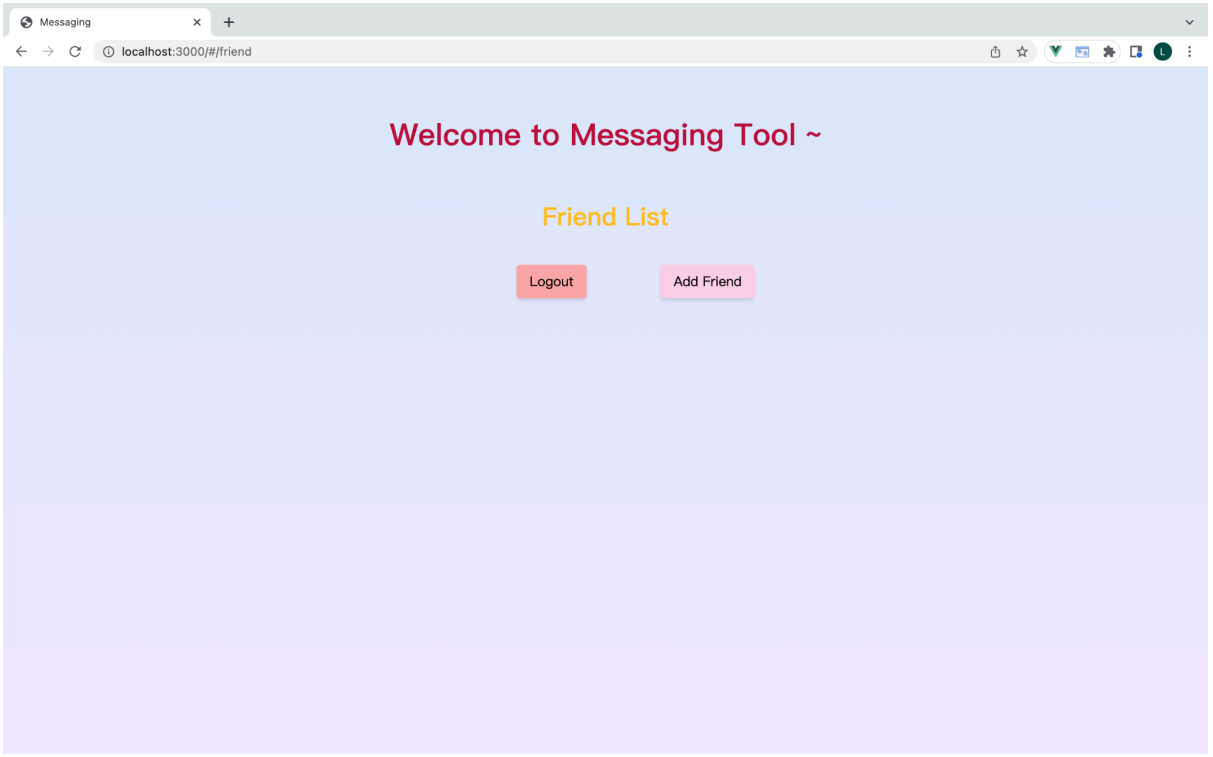


CA Error

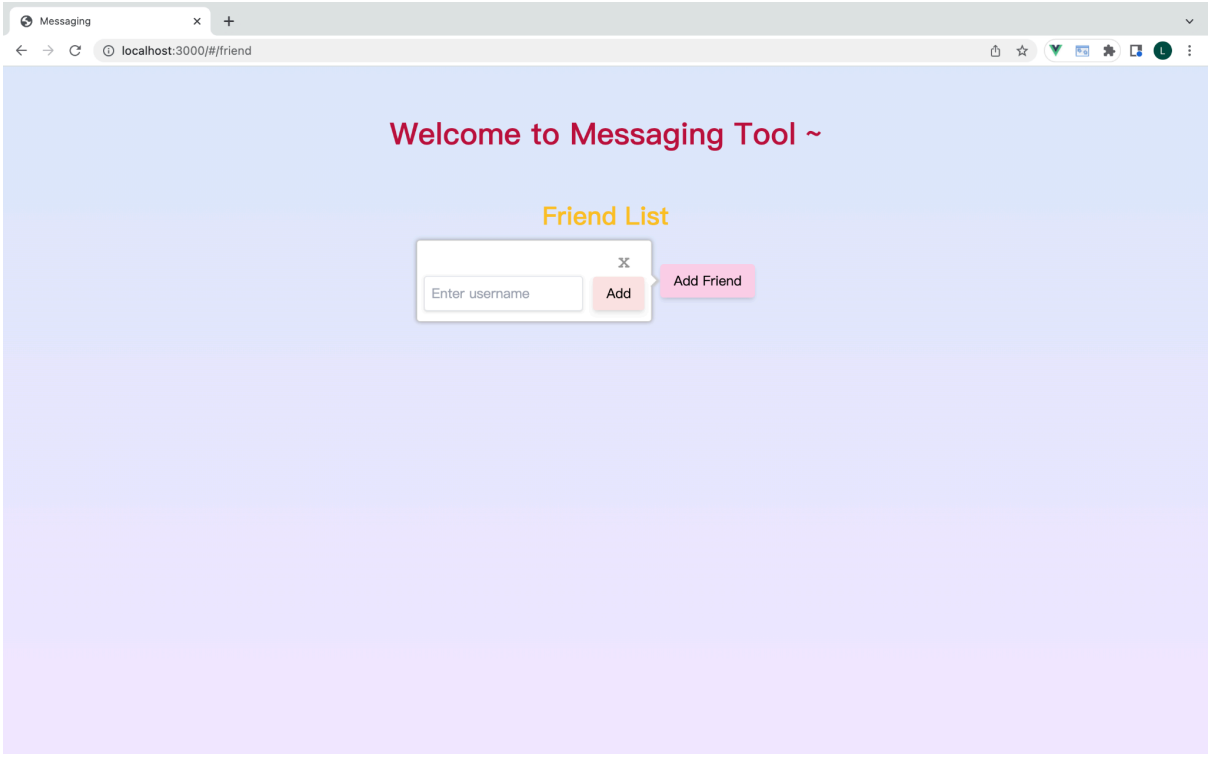


Friend-list Page

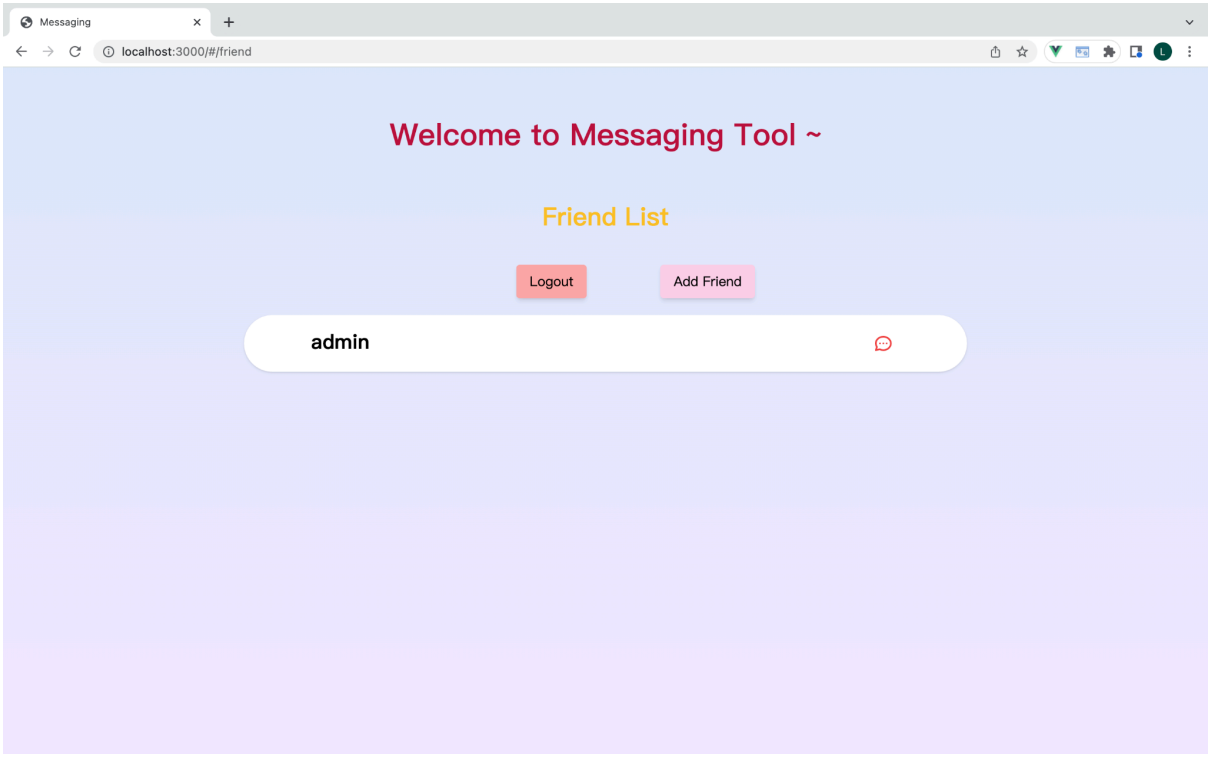
1. **Showing after login successfully.**
2. **When not login, it will automatically jump to the home page.**
3. **Button:**
   “Logout”: logout account and jump to home page
   “Add Friend”: will show a pop-ups



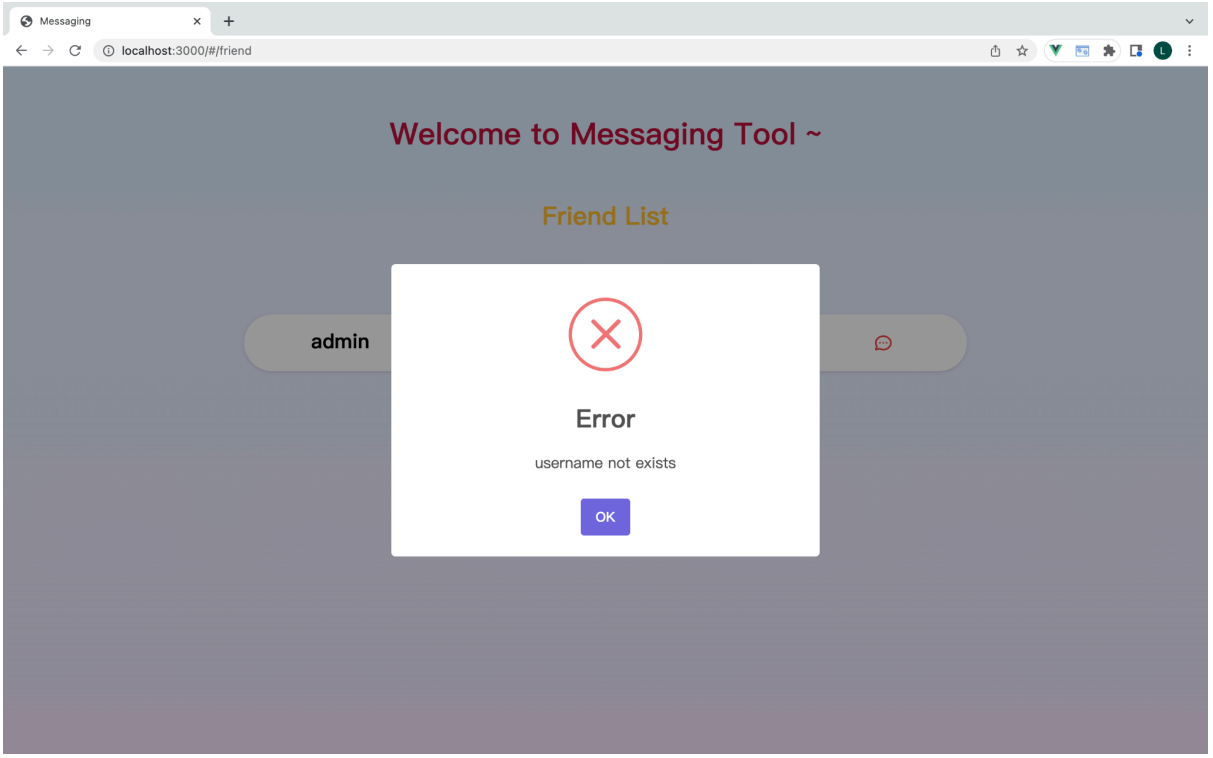
add friend:



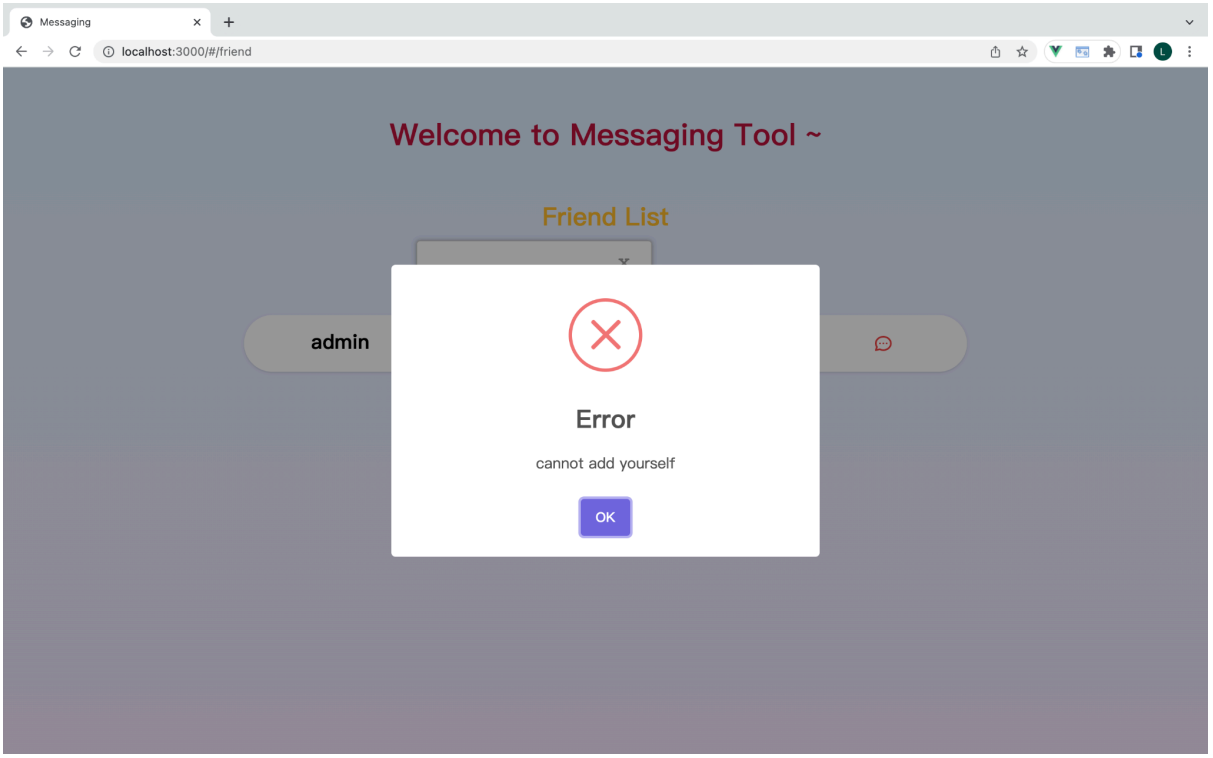
add friend finished:



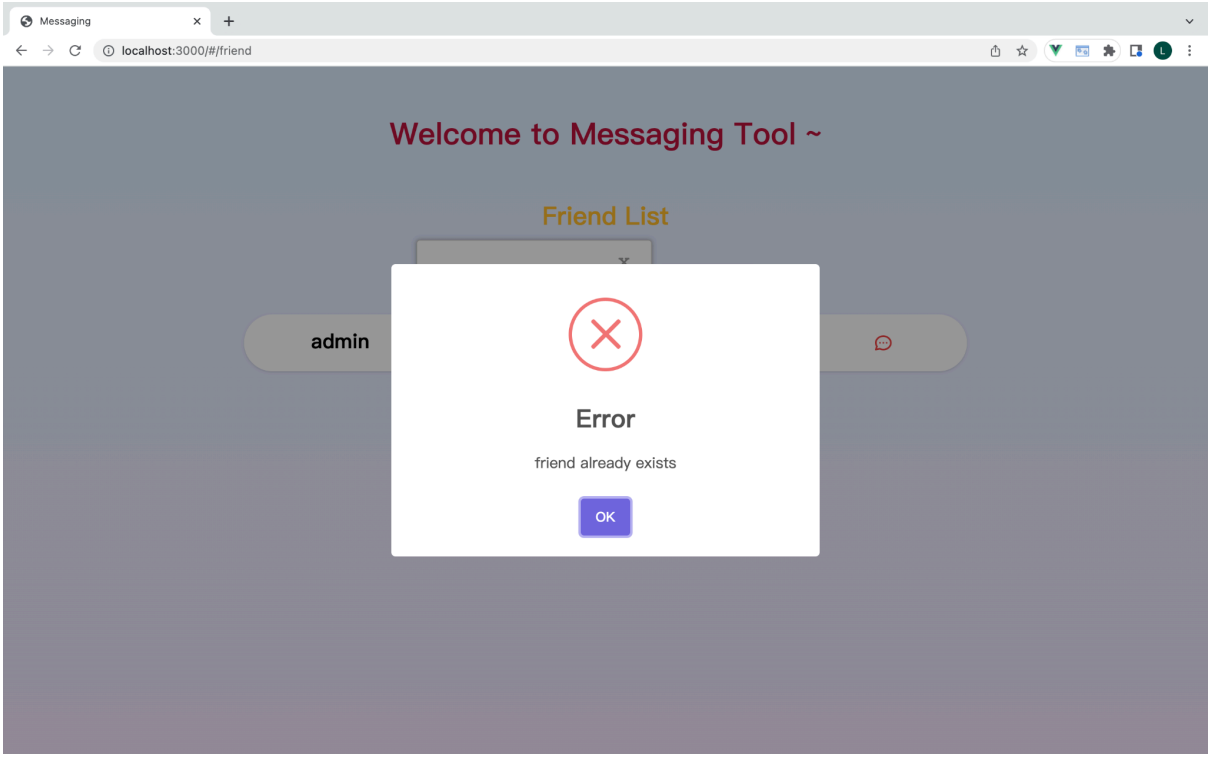
add not exist friend:



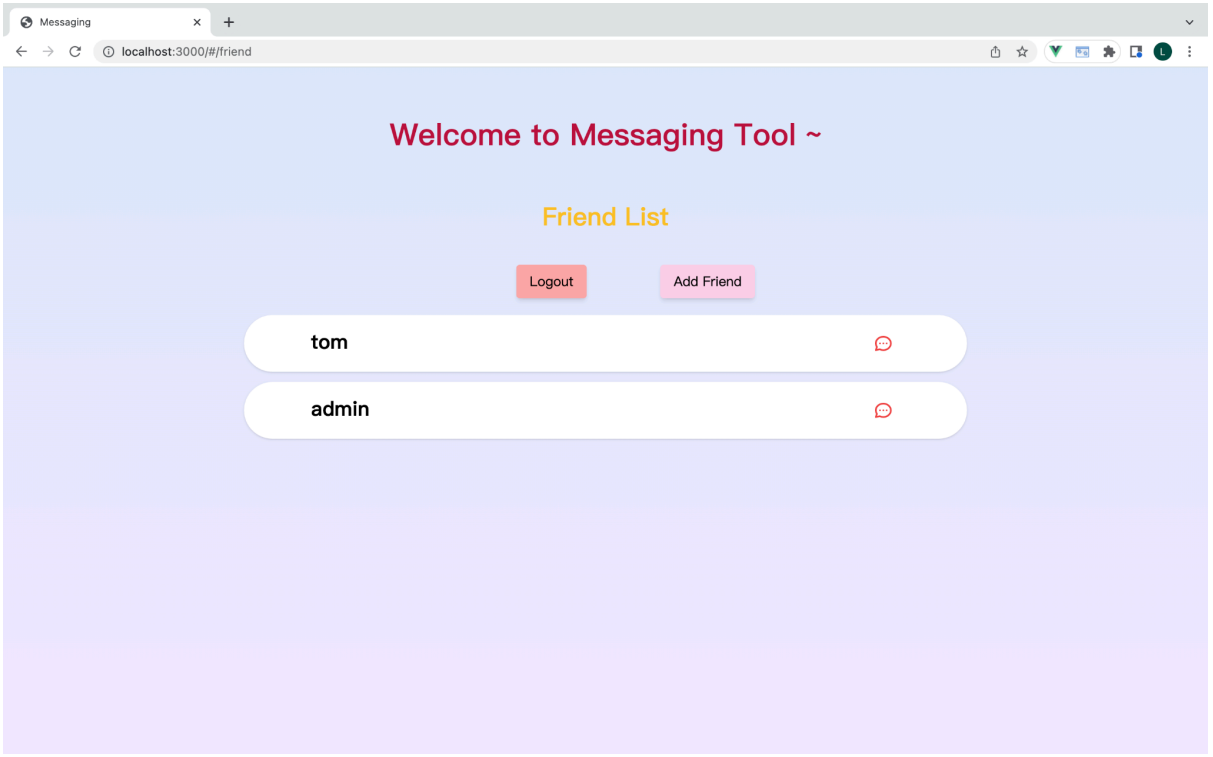
add self:



add the same friend:

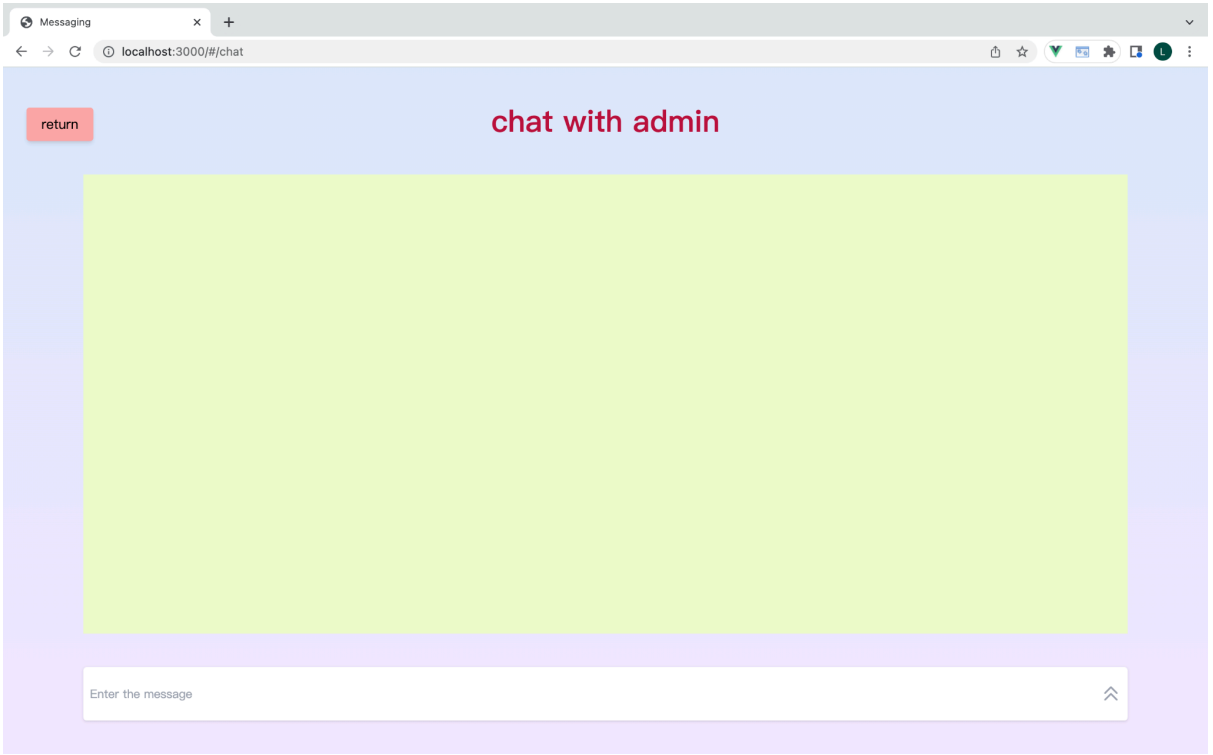


add multiple friends:

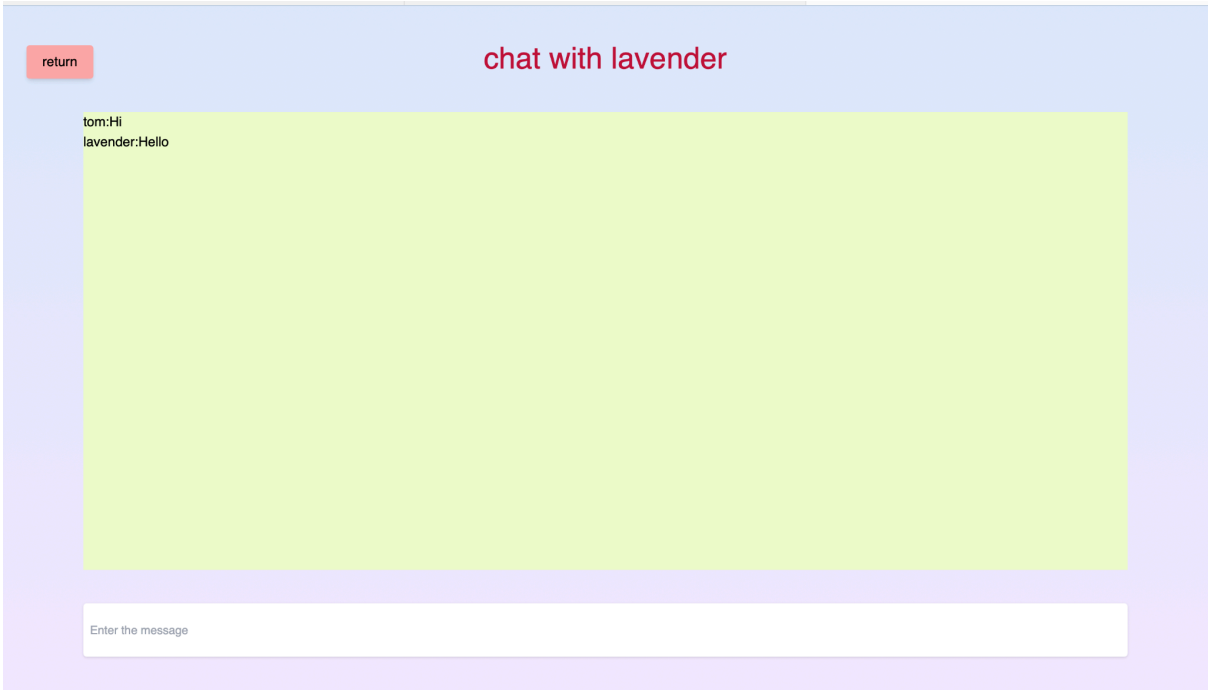


Chat Page

1. When not login, it will automatically jump to the home page.
2. Button: “return”: jump to the friend-list page.
3. When the user sends the message, the page will refresh.
4. If the message number is large, it will show with a margin roller.

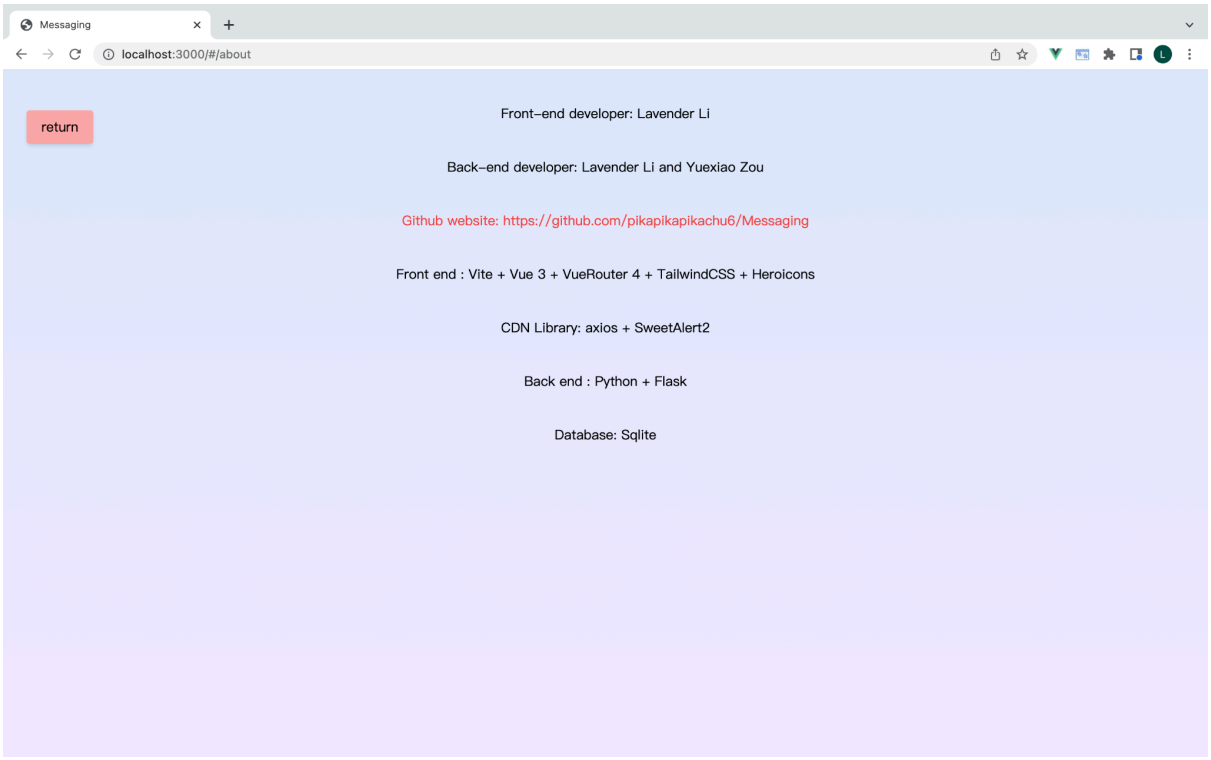


chat together:

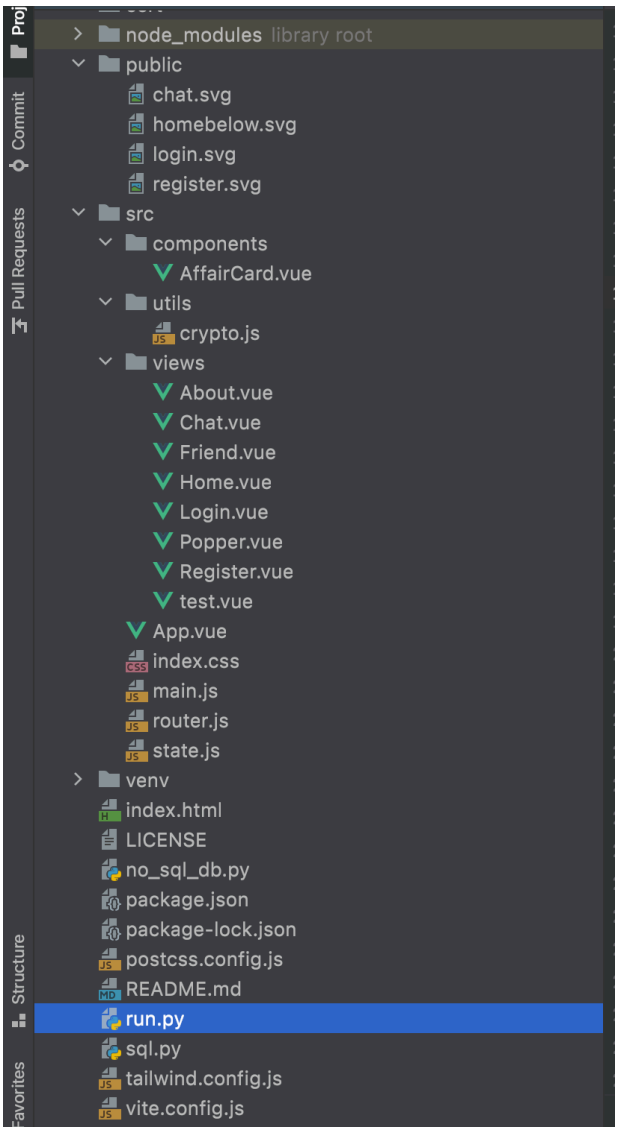


About page

1. **Button: “return”: jump to the home page.**



Code file explain:



front-end code:

- **/src/components/AffairCard.vue : the friend card**
- **/src/utils/crypto.js: all the front-end encrypt and decrypt function**
- **/src/views : each front-end page corresponding showing**
- **/src/router.js : the router file, define each router name and path**
- **/src/state.js : used to store the user state in session storage**
- **/public : the svg files**

back-end code:

- **run.py : back-end main file, all the API in it**
- **sql.py: the database code**

configure file:

- **vite.config.js : configure vite include the back-end address**
- **tailwind.config.js : configure tailwind css**
- **package.json : include the package need to install when run “npm i”**