CSE 312 Project Report

Project: *FaceNote*

Authors:

Baicheng Huang, bhuang22@buffalo.edu

Haohua Feng, haohuafe@buffalo.edu

Zhongai Shi, zhongais@buffalo.edu

Zhongen Shi, zhongens@buffalo.edu

INDEX

External Libraries used in team project:

Server side:

flask:

- Flask
- render_template
- request
- session
- redirect/url_for
- url_for
- escape
- json/flask.jsonify

flask_socketio:

- SocketIO
- emit
- join_room
- leave_room

PIL:

- Image

eventlet

Client side:

Websocket io

Canvas

flask library - version 1.0.2

Flask:

Script location: flask.app

A Flask is a flask object. It implements a WSGI application and acts as the central object.

A WSGI application on the server side is similar to the client request / server response in our homework.

For example:

from HTTP response/Request.ppt, 1_3_HTTP_Request_Response (cse312.com)

Request:	Response:
GET / HTTP/1.1	HTTP/1.1 200 OK
Host: cse312.com	Content-Type: text/plain
……	Content-Length: 5
	……

The flask object takes the name of the module or package as parameter, then it performs the corresponding action.

In our code, we use ‘app’ as the variable name by declare a flask object by

app = Flask(__name__)

@app.route(rule,**options) (flask.app.Flask - line 1288)

def action():

……

We mainly use this script to get clients' requests by passing URL rules and methods as arguments. The argument <rule> is the URL rule that contains the path of our application content. And, the argument <options> can be the request type, it can be 'GET', 'POST', etc. This is identical to the request header parsing on the server side in homework.

When the client sends a request to the server, the flask object 'app' will parse the request. We just need to specify the URL rule and request method type in .route().

The procedure of parse request is going to look like this:

server receives request

↓

@app.route(rule, method)

↓

call function add_url_rule(rule, method) (line 1178) in .route()

↓

in add_url_rule(), pass parameters rule and method to function url_rule_class()

Script location: werkzeug.routing.Rule – line 650

The URL rule parser will take place in function url_rule_class()

If the client requested URL rule and request method type match the parameters in .route(), then it will execute the function / actions that is below @app.route(). Basically, all of our web application responses that send back to clients will take place in this kind of function.

render_template:

Script location: flask.templating. - line 125

```
render_template(template_name_or_list, **context)
```

Render_template accepts the template name (from templates folder) and context. It renders the template with given context.

First, it updates the template variables with the given context, then renders. Here it uses the jinjia environment to load the template. Before it really renders the template, it will send the signal to indicate render is going to start, and also send a signal when render is finished.

```
render(self, treewalker, encoding=None)
```

(Python39\Lib\site-packages\pip_vendor\html5lib\serializer.py - line 375)

Between these two signals, it calls the render() to convert the html template file to string and return the serialized encoded html template content. Which will be the return value of render_template().

In our project, this is the way we render the page on the client side.

Ex: renders page of register

```
return render_template('register.html')
```

Request

```
request = LocalProxy(partial(_lookup_req_object, "request"))
```

Script location: flask\globals.py line - 60

In request of flask, we are only using attribute .method and .form.

request.method: (werkzeug\wrappers\base_request.py - line 606)

In our use case, its value only can be “POST” or “GET”.

We use request.method to determine the type of client request. In any input form of our application, it can only be ‘POST’.

request.form: (werkzeug\wrappers\base_request.py - line 465)

In method from(self), it runs self._load_from_data().

In self._loda_from_data(), this function will check if any content needs to parse. If it is true, it will parse the data to get the stream, mime type, content_length, etc..

In the self._loda_from_data(), it calls parse_options_header() (werkzeug\http.py - line 398). This function will parse the form data and extract the key value pairs, then turn the key value into a dictionary. This dictionary will pass back to self.__dict__, the object request as a value of a key. In our case, the key is “form”.

Ex: {‘form’: {‘username’: xxxxxx, ‘password’: xxxxxx, ……}}

In file server.py - line 165

```
def login():

    if request.method == 'POST':

        username = request.form['username']

        password = request.form['password']

……
```

If we assign a variable request.from[‘username’], this variable will hold the username submitted by the client. This is how we use request.method and request.form to get user form input from user login page. It is the same for any other form input in our web application.

Session

```
session = LocalProxy(partial( lookup req object, "session"))
```

Script location: flask\globals.py line - 60

This is similar to the mechanics of request. A session also returns the value that is stored in key ‘session’ in the dictionary. For each client that has logined (thread), each thread connection will have a session that stores client’s information, like username.

When a client submitted a login form and passed it, we assigned session[‘user’’] equal to this client’s username by `session['user'] = username`.

If the client is offline, we pop the ‘user’ to remove it from the session by `session.pop('user', None)`.

The object session is also used in our show current online users.

redirect / url_for:

Source: https://flask.palletsprojects.com/en/1.1.x/quickstart/

redirect: Script location: werkzeug\utils.py. - line 490

```
def redirect(location, code=302, Response=None):
```

This function redirects the client to the target location.

url_for: Script location: flask\helper.py. - line 226

```
def url_for(endpoint, **values):
```

Endpoint is the name of a function. Url_for function generates a URL to the given endpoint with the method provided.

Sometimes we need to redirect some requests to other pages. Flask provides a redirect() function to do this job. It takes a valid url address as input. However, to enter the entire url address may be tedious and easy to wrong if the address is very long. To solve this problem, flask provides a very useful method, url_for().

url_for() function can build a URL to a specific function, it accepts the name of the function as its first argument. In most cases, it appears with redirect().

In our project, we use redirect and url_for many times, below is one application of them:

```
@app.route('/direct_chat/<send_to_user>')
def directChat(send_to_user):
    if 'user' in session:
        if session['user'] == send_to_user:
            # Use the combination of redirect and url_for functions to redirect to other page
            # https://flask.palletsprojects.com/en/1.1.x/quickstart/
            return redirect(url_for("profile"))

        sender = session['user']
        sql = "select * from message where (sender=%s and receiver=%s) or (sender=%s and receiver=%s);"
        cur.execute(sql, (sender, send_to_user, send_to_user, sender))
        messages = cur.fetchall()

        return render_template("direct_chat.html", sender=sender, send_to=send_to_user, messages=messages)
    else:
        redirecting = '<h3>Redirecting ... </h3>'
        rd_fail = '<script>setTimeout(function(){window.location.href="/login.html";}, 3000);</script>'
        return "<h1>Please login first.</h1>" + redirecting + rd_fail
```

When a user wants to send private messages to other user, if the receiver username is the same as the sender username, instead of opening the direct_chat.html page, we redirect the user to his/her profile page, because a user cannot send private message to himself/herself. We see that in here we use the combination of redirect and url_for functions to accomplish this task.

escape:

Source: https://flask.palletsprojects.com/en/1.1.x/quickstart/

Script location: markupsafe_native.py. - line 15

```
def escape(s):
```

To make the submitted HTML not rendered as HTML and only displayed as text, we can take the advantage of the escape() function of flask library to do so. It accepts a string as the first argument and converts it to a safe string. Basically it replaces the sensitive characters, such as “<” and “>”, to corresponding safe versions, “<” and “>”, respectively.

Every time the server receives data from users, the server will use escape() to filter the data before sending it out. We use escape() in many cases.

```
@socketio.on('message')
def handleMessage(msg):
    if 'user' in session:
        sender = msg.get('sender')
        receiver = msg.get('receiver')
        message = msg.get('message')
        now = datetime.now()
        date = now.strftime("%m/%d/%Y %H:%M:%S")
        sql = "insert into message values (%s,%s,%s,%s);"
        cur.execute(sql, (sender, receiver, message,date))
        db.commit()

        emit('privateMessage', {'sender': sender, 'receiver': receiver,
                                'message': escape(message), 'date': date}, room=receiver)
```

Above is one of the cases we use the escape() function. When the user sends a message through the websocket, we escape the message to make it safe.

json library and Flask.jsonify:

Source: https://www.w3schools.com/python/python_json.asp

Source: https://flask.palletsprojects.com/en/1.1.x/quickstart/

json.dumps: Script location: json__init__.py. - line 183

```
def dumps(obj, *, skipkeys=False, ensure_ascii=True,...
```

jsonify: Script location: flask\json__init__.py. - line 306

```
def jsonify(*args, **kwargs):
```

Python has a built-in package called json. JSON data are very useful when we need to transfer data between different programming languages. In our project, the server side is written in python and the client side is written in javascript, therefore, JSON data is very helpful. Below is one of the uses of JSON data.

```python
@socketio.on('disconnect')
def disconnect_handler():
    if 'user' in session:
        room = session['user']
        leave_room(room)
        if room in game_users:
            game_users.remove(room)
            for user in game_users:
                send(game_users, room=user)

        print(str(session['user']) + " disconnected")
        online_users.remove(room)
        online_users.sort()
        users_login = list()
        for user in online_users:
            temp = dict()
            temp['username'] = user
            temp['icon'] = users_icon[user]
            users_login.append(temp)
        if users_login:
            # We convert the a list to JSON data
            users_json = json.dumps(users_login)
            send(users_json, json=True, broadcast=True)
        else:
            send(json.dumps(""), json=True, broadcast=True)
```

What we do here is broadcast the list of users to all connected clients. Since python and javascript do not have the same format to represent a list, we cannot send that list directly. Instead, we should convert it to JSON data through json.dumps() method so that javascript can understand.

Flask has a function called jsonify(), which also converts data to JSON data. The biggest difference between json.dumps() and jsonify() in flask is that jsonify() returns a flask.response() object that has the Content-Type: application/json while json.dumps() simply convert the string to JSON data. In our project, we also use jsonify() function when a user wants to register a new account or change its password. Client side sends GET request with username and password to server by ajax, server will respond with corresponding message in JSON data to client. Notice that, since they are GET request and response, Content-Type header is important, so that we preferred jsonify() than json.dumps().

For example: in the register page, the javascript in register_check.js file will send GET requests to the server by ajax to ask the validation of username and password and wait for the response, as shown below.

```
function check_user(){
    const username = document.getElementById("username").value;
    const url = "/username_validation/?username=" + username;
    $.ajax(
        {
            url: url,
            type: "get",
            data: "",
            success: function (get_from_server) {
                console.log(get_from_server);
                if (get_from_server['exists']) {
                    username_check = false;
                    document.getElementById("display_exist").innerHTML = get_from_server['display'];
                } else {
                    username_check = true;
                    document.getElementById("display_exist").innerHTML = get_from_server['display'];
                }
            }
        }
    )
    document.getElementById("reg_sub").disabled
        = !(username_check && password_check);
}
```

Then, the server side will respond to the request with JSON data by jsonify() function.

```python
@app.route('/username_validation/')
def check_user_exist():
    result = {"exists": bool, "display": ""}
    username = request.args.get("username")
    if len(username) == 0:
        result["exists"] = True
        result["display"] = "<font color='red'> ✕ Empty Username</font>"
        # We convert the a list to JSON data
        # https://flask.palletsprojects.com/en/1.1.x/quickstart/
        return jsonify(result)
    if username:
        if len(username) < 3:
            result["exists"] = True
            result["display"] = "<font color='red'> ✕ Too short username</font>"
            return jsonify(result)
        else:
            sql = "select * from user where username = (%s)"
            cur.execute(sql, (username,))
            user = cur.fetchone()
            if user:
```

Flask_socketio library:

Resource link: https://flask-socketio.readthedocs.io/en/latest/

SocketIO:

Script location: flaks_socketio__init__.py line 56

```python
class SocketIO(object):
```

SocketIO is a library that makes our websocket connection to be easier. SocketIO creates a Flask-SocketIO server and it is used on the server side. The basic function is similar to our websocket homework.

The socket connection is initialized by binding to our application and then run it on line 510 (*socketio.run(...)*).

`socketio = SocketIO(app)` Script location: server.py line 38

`socketio.run(app, ...)` Script location: server.py line 508

Once the binding is done, we have to register handlers for the events. In our project, we have created many events, for example ‘message’, ‘draw1’, etc.

All the events are similar, and we will show one example to show how we use it.

The event ‘message’ is registered like below:

```
@socketio.on('message')
def handleMessage(msg):
    if 'user' in session:
        sender = msg.get('sender')
        receiver = msg.get('receiver')
        message = msg.get('message')
        now = datetime.now()
        date = now.strftime("%m/%d/%Y %H:%M:%S")
        sql = "insert into message values (%s,%s,%s,%s);"
        cur.execute(sql, (sender, receiver, message, date))
        db.commit()

        emit('privateMessage', {'sender': sender, 'receiver': receiver,
                                'message': escape(message), 'date': date}, room=receiver)
```

@socketio.on() is used to register the event. Once there is a new message, a function will handle this message. In our project, when the client sends a message through event ‘message’ (we will see that in Client-Side websocketio), our server will know it and the function handleMessage() will handle it. ‘msg’ is a dictionary format. Do the necessary operations such as subtraction of data, add data, etc. then send the operated message to certain destination through room (we will see it later) and through event ‘privateMessage’.

Emit:

Script location: flaks_socketio__init__.py line 768

```
def emit(event, *args, **kwargs):
```

This function emits a SocketIO event to one or more connected clients.

The previous example is an example of emit. On the client-side, it is listening on event ‘privateMessage’, so once we send the message to event ‘privateMessage’, the clients connected will receive the message.

Send:

Script location: flaks_socketio__init__.py line 828

```
def send(message, **kwargs):
```

Send is similar to emit but it doesn’t require the event.

Most of our sending is used emit function because it is easier to use. We use the send function in the ‘disconnect’ event.

```
if users_login:
    users_json = json.dumps(users_login)
    send(users_json, json=True, broadcast=True)
else:
    send(json.dumps(""), json=True, broadcast=True)
```

On the server-side, the data sended in this example is in json format.

```
socket.on('json', function(users_json) {
    var list = document.getElementById('current_users');
    list.innerHTML = "";
    var users = JSON.parse(users_json);
```

On the client-side, when the data sended is in json format, it will be handled by the event ‘json’.

Join_room and leave_room:

Join_room: Script location: flaks_socketio__init__.py line 886

```
def join_room(room, sid=None, namespace=None):
```

This function puts the user in a room, under the current namespace.

Leave_room: Script location: flaks_socketio__init__.py line 912

```
def leave_room(room, sid=None, namespace=None):
```

This function removes the user from a room, under the current namespace.

In order to complete the direct message requirement, we have to know how to send the message to a certain destination. We use the room library to help us.

Once a client is connected to the websocket, the client's name will be used to create a room.

```
79          room = session['user']
80          join_room(room)
```

Similarly, if a client is disconnected from the websocket, the corresponding room will be removed.

```
97          room = session['user']
98          leave_room(room)
```

Room is used with emit or send function. As we saw in the previous example, in the end of the emit function, we have the 'room = receiver'. This means that only the 'receiver' can receive the message. In other words, if client A sends a message M to client B, both A and B had create a room named A and B. Message M will first be captured by server *@socketio.on('message')*, and then M will be sended to B via 'room = receiver' in this case the receiver is B. Using this characteristic, we can complete the DM.

```
emit('privateMessage', {'sender': sender, 'receiver': receiver,
                        'message': escape(message), 'date': date}, room=receiver)
```

PIL library:

Source: https://pillow.readthedocs.io/en/stable/reference/Image.html

Image.open: Script location: PIL\Image.py. - line 2862

```
def open(fp, mode="r", formats=None):
```

We use this library to reduce the size of the image uploaded by the user to update their profile photo. Since the photo of profile does not need very big resolution, we can take the advantage of the Image module from the PIL library to reduce the size or resolution of the image of profile.

```
@app.route('/profile', methods=['POST', 'GET'])
@app.route('/profile.html', methods=['POST', 'GET'])
def profile():
    if request.method == "POST":
        email = request.form['email']
        gender = request.form['gender']
        birth = request.form['birth']
        pp = request.form['personal_page']
        introduction = request.form['introduction']

        # Read the bytes of the image from input file and store it locally
        # Since the image may be very large, but the user icon does not need to be that big
        # So we store use Image method from PIL library to reduce the size of image
        icon = request.files['icon']
        ic = icon.read()
        path = "static/images/" + icon.filename
        fout = open(path, 'wb')
        fout.write(ic)
        fout.close()
        img = Image.open(path)
        img.thumbnail((400, 400))
        icon_name = 'icon_' + icon.filename
        img.save("static/images/" + icon_name)
        os.remove(path)
```

On the server side, we read the image (type file) content when the user posts an update of profile. First, we save that image locally and then open it through Image.open() method of the Image module. Then create a thumbnail image with 400x400 max

resolution. Finally, we delete the original image by os.remove() method since we do need it now.

eventlet:

Source: https://stackoverflow.com/questions/60991897/websocket-transport-not-available-install-eventlet-or-gevent-and-gevent-websock

In order to run the WebSocket in docker-compose, we have to install eventlet or gevent and gevent-websocket. Otherwise, we will fail to build the WebSocket connect, as shown in the following image.

```
faceNote | WebSocket transport not available. Install eventlet or gevent and gevent-websocket for improved performance.
faceNote |  * Serving Flask app "server" (lazy loading)
faceNote |  * Environment: production
faceNote |    WARNING: Do not use the development server in a production environment.
faceNote |    Use a production WSGI server instead.
faceNote |  * Debug mode: off
faceNote |  * Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
```

To solve this issue, we install eventlet, which is done by adding “eventlet” to our requirement.txt file. Then, the docker-compose will run well and WebSocket connect will be successfully built.

```
faceNote | [INFO  wait] Host db:3306 is now available!
faceNote | [INFO  wait] ----------------------------------------------------------
faceNote | [INFO  wait] docker-compose-wait - Everything's fine, the application can now start!
faceNote | [INFO  wait] ----------------------------------------------------------
```

Client-side

Websocket io:

Script location: all the html templates.

In each html templates, it contains pieces of JavaScript code for websocket and jquery.

```
<script type="text/javascript"
        src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"></script>
<script src="http://code.jquery.com/jquery-3.2.1.min.js"></script>
<script src="../static/js/default2.js" charset="utf-8"></script>
```

Each time the logged in user navigates to a html page, the user will connect to websocket.

The websocket connection is simply using the code below:

```
$(document).ready(function() {
    var socket = io();
    socket.on('json', function(users_json) {
```

Once the connection has been established, we can wait for the event to be called. As we saw in the previous message example, in the client-side, we are waiting for 'privateMessage' to be called by the server. Once this event is called, the browser will handle the data received.

```
socket.on('privateMessage',function(msg) {
    using Snackbar/Toast to notice user there is new message comes
    https://www.w3schools.com/howto/tryit.asp?filename=tryhow_js_snackbar
    var x = document.getElementById("newCome");
    x.innerHTML = "New message come from "+msg.sender;
    x.className = "show";
    setTimeout(function(){ x.className = x.className.replace("show", ""); }, 10000);
});
```

In our project, the message will be displayed to chat box, and a notice window will be displayed as below:

To send a message, it uses socket.emit() if the event is used.

```
socket.emit('message',{'sender':sender, 'receiver':receiver, 'message':message});
```

If we use socket.send() then no event is used.

Canvas:

Resource link: https://www.jb51.net/article/151507.htm

Script location: game.html.

The game of our project is drawing. Logged in users can navigate to the game page. There is a canvas that everyone can draw something on, and everyone will see it.

Actually a draw is a combination of many small segments formed by two points. The two points' coordinates will be sent via websocket and event 'draw1' to the server, and then the server will send this data to all other users via event 'draw2'. It is similar to sending a message.

To clean the canvas, just resize the height of the canvas, and this is also similar to sending a message.