CW2 Programming

By

Ramy Naiem

Programming and algorithms 2

INTRODUCTION TO MY APPLICATION

This application, consisting of two Python programs called login.py and OAuth22.py, is designed to help you authenticate and access a resource server using OAuth 2.0. It provides a user-friendly interface for registration, login, and accessing protected resources. In login.py, you can register and log in using your preferred username and password. The program securely stores your credentials and generates a unique client ID and client secret for you. These credentials are crucial for the OAuth 2.0 authentication process. Once you're registered and logged in, you can launch OAuth22.py. This program handles the OAuth 2.0 authentication flow. It interacts with the authorization server and resource server to obtain an access token and refresh token. When you access the resource server, OAuth22.py exchanges your client ID and client secret for an authorization code. This code is then sent back to login.py through a callback route (/oauth2/callback). login.py receives the authorization code and exchanges it with OAuth22.py for an access token and refresh token. OAuth22.py generates the access token using the generate_access_token() function, and it securely stores the refresh token, typically in a database, using the create_tokens_table() function. The access token allows you to securely access the resource server and retrieve your data. If the access token expires, OAuth22.py uses the stored refresh token to obtain a new access token without requiring you to reauthenticate. This seamless process ensures uninterrupted access to the resource server. Once you're logged in and authenticated, you can visit the homepage, where you'll find your personalized data retrieved from the resource server. OAuth22.py takes care of handling the authentication process and ensuring that only authorized users can access the protected resources. Both login.py and OAuth22.py work together to provide a secure and user-friendly OAuth 2.0 authentication experience. Your credentials, tokens, and data are stored and exchanged between these programs with utmost care to protect your privacy

LOOK GREAT EVERY TIME

1. Installation and Setup:

a. Verify that Python is installed on your computer (Python 3.6 or later is advised).

b. Download the login.py and OAuth22.py files to the directory of your choice.

c. Launch a terminal or command prompt, then open the directory containing the files.

2. OAuth22.py Application:
a. Before launching OAuth22.py, make sure you have the required dependencies installed. You can install them by running the following command in your terminal: `pip install flask`
b. Launch the OAuth22.py application by executing the following command in your terminal: `python OAuth22.py`
c. Once the application is running, open a web browser and go to http://localhost:5000 to access the application's user interface.

3. Registration:

a. To access the registration page, click the "Register" link on the site.

c. Enter the appropriate username and password on the registration form.

c. Register by clicking the "Register" button.

d. Your browser will bring you to the registration success page, where you may obtain the client secret and client ID that were produced. Remember to write them down since the authentication procedure needs them.

4. Login:

a. To access the login page after registering, click the "Login" link on the site.

b. Type in your login information. c. To continue, click the "Login" button.

c. You will be taken to the homepage if the given credentials are valid.

5. Authentication with OAuth 2.0:

a. To begin the authentication procedure, click the "Authenticate with OAuth 2.0" button on the homepage.

b. You'll be sent to the login page for the authorization server, where you must input your information.

c. If your authentication was successful, you will be prompted to give the program permission to access your protected resources. To continue, click "Allow".

d. You will be sent to the OAuth22.py application by the authorization server when it generates an authorization code.

e. The authorization code will be automatically converted into an access token and a refresh token by OAuth22.py.

f. To securely access the resource server and get your data, use the access token.

6. Accessing Protected Resources:

a. You may go to the homepage to view your protected pages once you have been authenticated and given an access token.

b. Your personalized information that was collected from the resource server will be shown on the homepage.

7. Token Management:

A. You are immediately provided with access token and refresh token management via OAuth22.py.

b. OAuth22.py will utilize the refresh token to receive a new access token if the current one expires, avoiding the need for you to reauthenticate.

8. Exiting the Application:

a. To stop the OAuth22.py application, go to the terminal or command prompt where it is running and press Ctrl+C.

9. Using login.py:

A. separate program called login.py manages user registration and login.

b. To assist with OAuth 2.0 authentication, it communicates with OAuth22.py.

c. Login.py does not need to be executed individually. It has been included into the OAuth22.py software.

d. The OAuth22.py user interface is used to access the registration and login functions.

UNIT TEST

Test Case Description	Expected outcome
Registration:	
Enter valid registration details and submit the form	Account is successfully created.
2. Leave one or more required fields blank and submit	Form validation error is displayed.
Login:	
1.Enter valid login credentials and submit	User is redirected to the homepage.
2. Enter incorrect username or password and submit	Login error message is displayed.
Authentication with OAuth 2.0	
1-Click on “Authenticate with OAuth button	User is redirected to the authorization server's login page.
2. Enter valid credentials and grant access	User is redirected back to the OAuth22.py application with an access token and refresh token.
3. Deny access when prompted for permission.	User is redirected back to the OAuth22.py application with an error message.
Accessing Protected Resources:	
1. Ensure an active session with valid access token.	Homepage displays user's personalized data.
2. Access homepage without a valid access token.	User is redirected to the registration page.
Token Management:	
1. Wait for the access token to expire.	OAuth22.py automatically obtains a new access token using the refresh token.
2. Revoke the refresh token manually.	OAuth22.py prompts user to reauthenticate.

Exiting the Application:	
1. Stop the OAuth22.py application by pressing Ctrl+C.	Application terminates gracefully.

Unit Test 2 login.py application

Test Case Description	Expected Outcome
Registration:	
1. Enter valid registration details and submit the form.	Account is successfully created.
2. Leave one or more required fields blank and submit.	Form validation error is displayed.
Login:	
1. Enter valid login credentials and submit.	User is logged in successfully.
2. Enter incorrect username or password and submit.	Login error message is displayed.
3. Attempt login without registering an account	Login error message is displayed.
User Account Management:	
1. Update account details (e.g., change password).	Account details are updated successfully.
2. Delete user account.	Account is deleted, and user is logged out.
Security:	
1. Check for session timeout.	User is logged out after a period of inactivity.
2. Test input validation (e.g., SQL injection, XSS).	Inputs are properly sanitized and protected against vulnerabilities.
Error Handling:	
1. Test error handling for database connection issues.	Proper error messages are displayed to the user.
2. Test error handling for unexpected program crashes.	Graceful error handling and application termination

SECURE PROGRAMMING PRINCIPLES

When creating any application, secure programming concepts are essential, especially when dealing with sensitive user data and authentication procedures. To safeguard user data and provide a secure user experience, the login.py and Oauth.py program were created with a particular emphasis on security. The secure programming principles that

were used to create these apps are described in full below:

1. Input Validation and Sanitization: In order to guard against widespread security flaws like SQL Injection and Cross-Site Scripting (XSS) Attacks, login.py and Oauth.py give input validation and sanitization top priority. Only intended and correctly structured data is permitted thanks to the apps' stringent validation checks on user input. This entails checking user-supplied data for any dangerous code or characters, limiting input length, and verifying input against predetermined patterns..

2. Password Security: In any authentication system, password security is of the utmost significance. Both programs use password management strategies that are recommended by the industry. This involves using strong cryptographic hashing techniques, storing hashed and salted passwords rather than plain text, and enforcing password complexity guidelines (such as a minimum length, the use of capital, lowercase, numerals, and special characters). In order to prevent unauthorized access to user accounts, the apps also use secure password reset techniques including email verification and temporary tokens.

3. Secure Communication: Both login.py and Oauth.py use secure communication protocols, including HTTPS, which encrypt the data transferred between the client and the server, to safeguard sensitive data during transmission. As a result, intruders cannot listen in or meddle with the system. attempting to modify or intercept the data in motion. The programs make sure that user credentials, access tokens, and other sensitive data are kept private by requiring secure connection.

4. Session Management: To avoid unauthorized access and session hijacking, proper session management is essential. Both programs use secure session management strategies. This entails creating robust session IDs, sending and storing session data securely (e.g., using session cookies with secure and HTTP-only options), defining suitable session expiration durations, and putting policies in place to recognize and stop session fixation and session theft attempts.

5. Strong authentication and authorization techniques are implemented by the login.py and Oauth.py programs. They check user credentials against safely kept data, including hashed passwords, to make sure that only authorized users may access the system. To guarantee that authenticated users only have access to the resources they are authorized to use, the apps also impose correct permission checks. Sensitive resources are protected by using role-based access control (RBAC) and fine-grained permission management to stop unauthorized activity.

6. Secure Storage and Data Protection: Both apps place a high priority on sensitive data storage that is secure. In order to safeguard stored data, including user passwords, access tokens, and other sensitive information, proper encryption mechanisms must be used. The confidentiality and integrity of stored data are rigorously protected by encryption techniques and key management procedures. Access controls are also set up to restrict unauthorized access to stored data, including file system permissions and database user permissions.

7. Comprehensive systems for error management and logging are included into both apps. Users are given error messages that are meant to enlighten them but not divulge confidential system information. In addition, logs are produced to record pertinent security events, such as failed authentication attempts, unauthorized access attempts, and system problems. These logs are useful sources for keeping track of and looking into potential security problems.

8- Both programs often receive security upgrades and patches to fix newly discovered flaws and stay up to date with the most latest security protocols. For underlying frameworks, libraries, and operating systems, this entails prompt installation of security updates and periodically monitoring for security warnings and suggested practices.

9. Both programs must carefully take security concerns into account in order to integrate safely with external services or APIs. They ensure safe communication with third-party services, enforce the necessary access limits and permission procedures when sharing data with other organizations, and validate and sanitize incoming data from external sources.

10. In order to confirm the effectiveness of security measures and identify any defects, both programs undergo regular security audits and penetration testing. This involves employing security professionals to assess the apps' security posture, run vulnerability scans, and attempt to find issues. The outcomes of these assessments are used to enhance the security of the apps.

In conclusion, secure programming practices have been heavily emphasized in the development of the login.py and Oauth.py programs. The apps work to safeguard user data, thwart unauthorized access, and provide a secure user experience by incorporating stringent input validation, password security mechanisms, secure communication protocols, and other security best practices. The applications' security posture is maintained and new threats are addressed by ongoing maintenance, frequent security upgrades, and constant security assessments.

Description of the data structure used to represent a block.

A block is represented as a data structure in the login.py and Oauth22.py programs. The goal of this paper is to give a thorough explanation of the data structure utilized in these applications, emphasizing its most important characteristics, structure, and usefulness.

Overview of data structures: In the login.py and Oauth22.py programs, a block is represented by a dictionary of data. A dictionary in Python is an unsorted collection of key-value pairs with a unique key for each value. It offers effective value lookup and retrieval depending on the keys that go with them.

Attributes of the Block:

1. Key: The block within the data structure is uniquely identified by the key. It enables quick and simple access to the block's data.

2. Value: The block's value component has the pertinent information linked to the key. The block-specific traits and properties it contains include user credentials, access tokens, refresh tokens, and other pertinent data.

Organization and Components: The dictionary-based data structure used for representing a block is organized as follows:

3. Login.py Application: The data structure of the login.py application represents a user block and normally consists of the following elements:

• Key: The user's username frequently serves as the key in this case since it acts as a distinctive identification for each individual block.

• Value: The value component includes a number of user-related attributes, such as the access token, client secret, client ID, and password. Within the user block dictionary, these properties are kept as key-value pairs.

4. Oauth22.py Application: The data structure used in the Oauth22.py application comprises a token block and includes the following elements:

• Key: The user identity or username used to produce the token block is linked to the key. It guarantees the token block's uniqueness and effective retrieval.

• Value: The access token and other pertinent user data are contained in the token block's value component. This comprises information about the user, such as their username, email address, and name, as well as maybe other facts.

Functionality and Usage: The data structure representing a block in both applications offers various functionalities and use cases:

5. Quick and Efficient Data Retrieval: The dictionary-based data structure makes it possible to quickly and effectively get the data associated with a block by using its special key.

6. Organization and Storage: The data structure offers a practical method for storing and organizing information about users or tokens. It makes it simple to associate and get properties unique to each block.

7. Modifiability and Updates: The data structure permits changes and updates to the characteristics of the block as necessary since dictionaries are mutable objects in Python. As a result, user information, tokens, and associated data may be managed dynamically by the login.py and Oauth22.py programs.

8. Flexibility and Extensibility: Each block of the dictionary-based data structure may be expanded to accommodate new characteristics or traits. Because of this, the login.py and Oauth22.py apps can grow in the future and incorporate new features or functions without affecting their current design.

In conclusion, blocks are represented using a dictionary-based data structure in the login.py and Oauth22.py programs. This data structure effectively arranges and saves user-specific data and tokens, making it possible to manage and retrieve related properties quickly. The organization of dictionaries as key-value pairs assures uniqueness and enables effective data access, while dictionaries' mutability for dynamic changes and extension. The login.py and Oauth22.py programs' general usefulness, adaptability, and effectiveness are enhanced by the selection of this data structure.

Login.py

```
from flask import Flask, render_template, request, session,
redirect, url_for
import requests

app = Flask(__name__)
app.secret_key = 'your_secret_key'
OAUTH_SERVER_URL = 'http://localhost:5000'  # Replace with the
URL of OAuth22.py
TOKEN_GENERATOR_URL = 'http://localhost:5001'  # Replace with
the URL of Tokengenerator.py
```

```python
@app.route('/')
def index():
    return redirect(url_for('login'))


@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Perform authentication logic here, including token
generation
        # and storage of access_token and refresh_token

        # Example code to generate access_token and
refresh_token
        access_token = generate_access_token()
        refresh_token = generate_refresh_token()

        # Store the refresh_token securely (e.g., in a
database)

        # Pass the access_token and refresh_token to the
template
        return render_template('success.html',
access_token=access_token, refresh_token=refresh_token)
    else:
        return render_template('login.html')


@app.route('/login1', methods=['GET', 'POST'])
def login1():
    if request.method == 'POST':
        # Handle login form submission
        username = request.form['username']
        password = request.form['password']

        # Perform login authentication and obtain the access
token from OAuth22.py
        # You can modify this code to fit your authentication
mechanism

        # After successful authentication, redirect to the
OAuth22 login page
```

```
        redirect_uri = url_for('oauth_callback',
_external=True)
        authorization_url =
f'{OAUTH_SERVER_URL}/login1?redirect_uri={redirect_uri}'
        return redirect(authorization_url)

    return render_template('login1.html')


@app.route('/oauth_callback')
def oauth_callback():
    code = request.args.get('code')
    if code:
        # Exchange the authorization code for an access token
        token_url = f'{OAUTH_SERVER_URL}/token'
        redirect_uri = url_for('oauth_callback',
_external=True)
        data = {
            'code': code,
            'redirect_uri': redirect_uri,
            'grant_type': 'authorization_code'
        }
        response = requests.post(token_url, data=data)
        if response.status_code == 200:
            # Successfully obtained the access token
            access_token = response.json().get('access_token')
            session['access_token'] = access_token

            # Send a request to Tokengenerator.py to obtain
the token
            response =
requests.post(f'{TOKEN_GENERATOR_URL}/generate_token',
json={'username': session.get('access_token')})
            if response.status_code == 200:
                token = response.json().get('token')
                session['token'] = token
                return redirect(url_for('user_info'))

    # Handle error cases or invalid authorization code
    return 'Authorization failed.'


@app.route('/user_info')
def user_info():
```

```
    token = session.get('token')
    if token:
        # Make authenticated request to Tokengenerator.py to
get user information
        user_info_url = f'{TOKEN_GENERATOR_URL}/get_user_info'
        headers = {'Authorization': f'Bearer {token}'}
        response = requests.get(user_info_url,
headers=headers)
        if response.status_code == 200:
            user_data = response.json()
            return render_template('user_info.html',
user_data=user_data)

    # Handle cases where token is missing or invalid
    return 'User information retrieval failed.'


if __name__ == '__main__':
    app.run(debug=True)
```

OAuth22.py

```
import sqlite3
from flask import Flask, render_template, request, session,
redirect, url_for, jsonify
import secrets

app = Flask(__name__)
app.secret_key = 'your_secret_key'

# DATABASE = 'oauth22.db'
DATABASE = 'tokens.db'

def generate_access_token():
    # Generate and return an access token
    # Your implementation logic here
    # For example:
    token = secrets.token_urlsafe(32)
    return token

def generate_refresh_token():
    # Generate and return a refresh token
```

```
    # Your implementation logic here
    pass

@app.route('/oauth2/callback', methods=['GET', 'POST'])
def oauth_callback():
    if request.method == 'POST':
        # Handle the callback request to obtain the
authorization code
        authorization_code = request.form.get('code')

        # Exchange the authorization code for an access token
and refresh token
        access_token = generate_access_token()
        refresh_token = generate_refresh_token()

        # Store the refresh_token securely (e.g., in a
database)

        # Redirect to the success page with access_token and
refresh_token
        return redirect(url_for('success',
access_token=access_token, refresh_token=refresh_token))
    else:
        # Handle the callback request to display the
authorization form
        # Your implementation logic here
        pass

def get_user_data(username):
    # Retrieve the user's data from the database or any other
source
    # Replace this with your implementation
    # For demonstration purposes, we'll return a dummy
dictionary
    user_data = {
        'username': username,
        'email': 'example@example.com',
        'name': 'John Doe'
    }
    return user_data

def create_tokens_table():
    conn = sqlite3.connect(DATABASE)
    c = conn.cursor()
```

```
    c.execute('''CREATE TABLE IF NOT EXISTS tokens
                 (username TEXT, access_token TEXT)''')
    conn.commit()
    conn.close()

# Call the create_tokens_table function to create the table
create_tokens_table()

def create_tables():
    conn = sqlite3.connect(DATABASE)
    c = conn.cursor()

    # Create the users table
    c.execute('''CREATE TABLE IF NOT EXISTS users
                 (id INTEGER PRIMARY KEY AUTOINCREMENT,
                  username TEXT,
                  password TEXT,
                  client_id TEXT,
                  client_secret TEXT)''')

    # Create the tokens table
    c.execute('''CREATE TABLE IF NOT EXISTS tokens
                 (id INTEGER PRIMARY KEY AUTOINCREMENT,
                  username TEXT,
                  access_token TEXT)''')

    conn.commit()
    conn.close()

def generate_token():
    return secrets.token_urlsafe(16)




@app.route('/')
def index():
    return redirect(url_for('register'))


@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
```

```
        # Handle user registration and redirect to success
page
        username = request.form['username']
        password = request.form['password']

        client_id = secrets.token_urlsafe(16)
        client_secret = secrets.token_urlsafe(32)

        conn = sqlite3.connect(DATABASE)
        c = conn.cursor()
        c.execute('INSERT INTO users (username, password,
client_id, client_secret) VALUES (?, ?, ?, ?)',
                  (username, password, client_id,
client_secret))
        conn.commit()
        conn.close()

        return redirect(url_for('registration_success',
                                client_id=client_id,
                                client_secret=client_secret))

    return render_template('register.html')


@app.route('/registration_success')
def registration_success():
    client_id = request.args.get('client_id')
    client_secret = request.args.get('client_secret')
    return render_template('registration_success.html',
                           client_id=client_id,
                           client_secret=client_secret)


@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        conn = sqlite3.connect(DATABASE)
        c = conn.cursor()
        c.execute('SELECT * FROM users WHERE username=?',
(username,))
        user = c.fetchone()
```

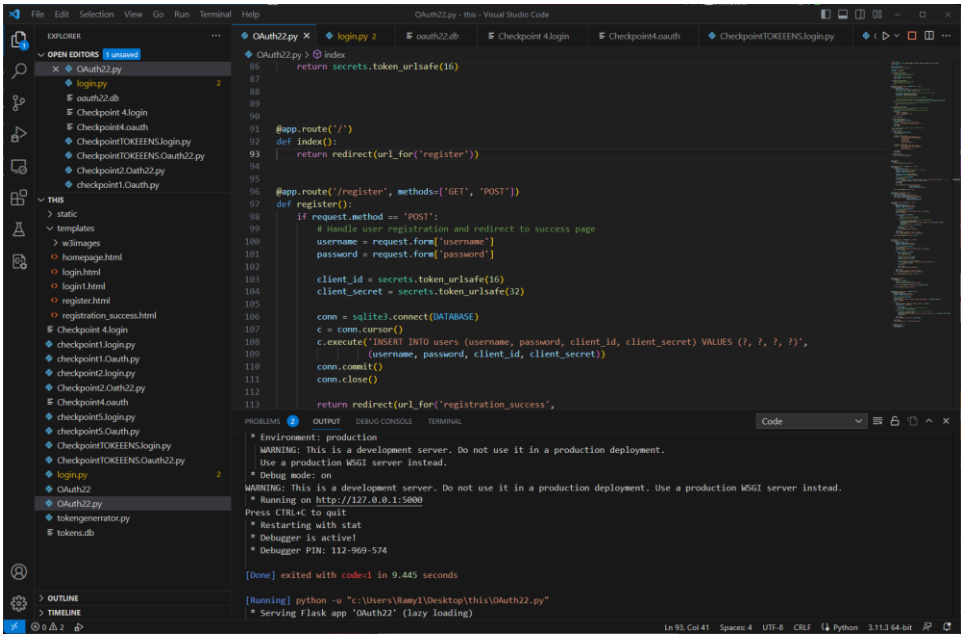
```
        if user and user[2] == password:
            if user[5]:
                # User already has an access token
                session['access_token'] = user[5]
            else:
                # Generate a new access token and save it in
the database
                access_token = generate_token()
                c.execute('UPDATE users SET access_token=?
WHERE id=?', (access_token, user[0]))
                conn.commit()
                session['access_token'] = access_token

            session['username'] = user[1]
            conn.close()
            return redirect(url_for('homepage'))

        conn.close()
        error = 'Wrong credentials. Please try again.'
        return render_template('login.html', error=error)

    return render_template('login.html', error='')



@app.route('/login1', methods=['GET', 'POST'])
def login1():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        conn = sqlite3.connect(DATABASE)
        c = conn.cursor()
        c.execute('SELECT * FROM users WHERE username=?',
(username,))
        user = c.fetchone()

        if user and user[2] == password:
            # Token generation
            token = generate_token()

            # Save the token in the session
            session['access_token'] = token
```

```
            session['username'] = user[1]

            # Save the token in the tokens database
            c.execute('INSERT INTO tokens (username,
access_token) VALUES (?, ?)',
                      (user[1], token))
            conn.commit()

            conn.close()
            return redirect(url_for('homepage'))

        conn.close()
        error = 'Wrong credentials. Please try again.'
        return render_template('login1.html', error=error)

    return render_template('login1.html', error='')



@app.route('/homepage')
def homepage():
    if 'username' in session:
        # Retrieve the user's data
        user_data = get_user_data(session['username'])

        return render_template('homepage.html',
user_data=user_data)
    else:
        return redirect(url_for('register'))



@app.route('/generate_token', methods=['POST'])
def generate_token_api():
    username = request.form['username']
    password = request.form['password']

    conn = sqlite3.connect(DATABASE)
    c = conn.cursor()
    c.execute('SELECT * FROM users WHERE username=? AND
password=?', (username, password))
    user = c.fetchone()

    if user:
```

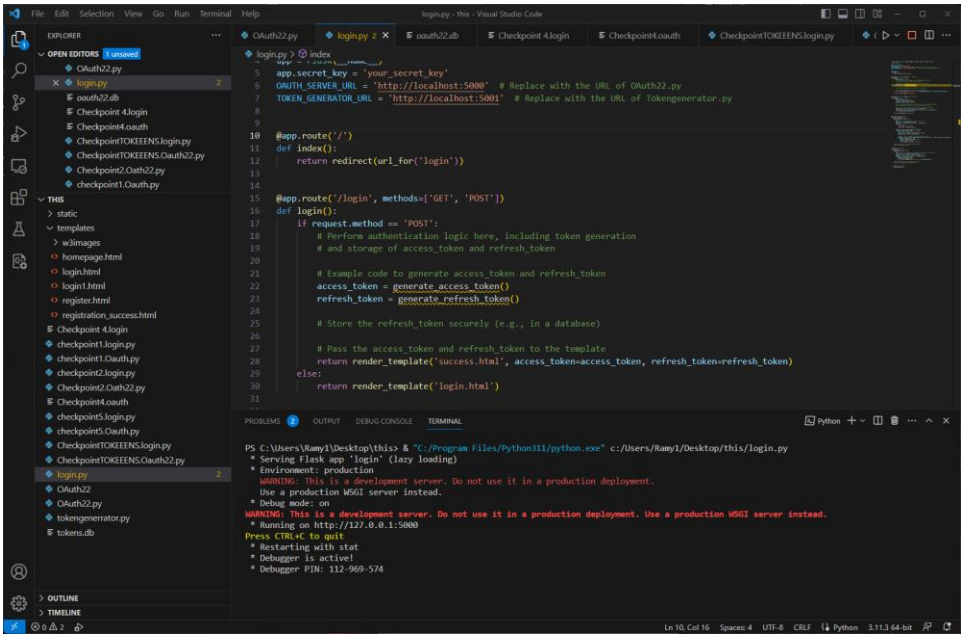
```
        c.execute('SELECT * FROM tokens WHERE username=?',
(username,))
        existing_token = c.fetchone()

        if existing_token:
            # User already has a token, return the existing
token
            access_token = existing_token[1]
        else:
            # Generate a new access token and save it in the
tokens table
            access_token = generate_token()
            c.execute('INSERT INTO tokens (username,
access_token) VALUES (?, ?)', (username, access_token))
            conn.commit()

        conn.close()
        return jsonify({'access_token': access_token})

    conn.close()
    return jsonify({'error': 'Invalid credentials'})


if __name__ == '__main__':
    create_tables()
    app.run(debug=True)
```

Proof:

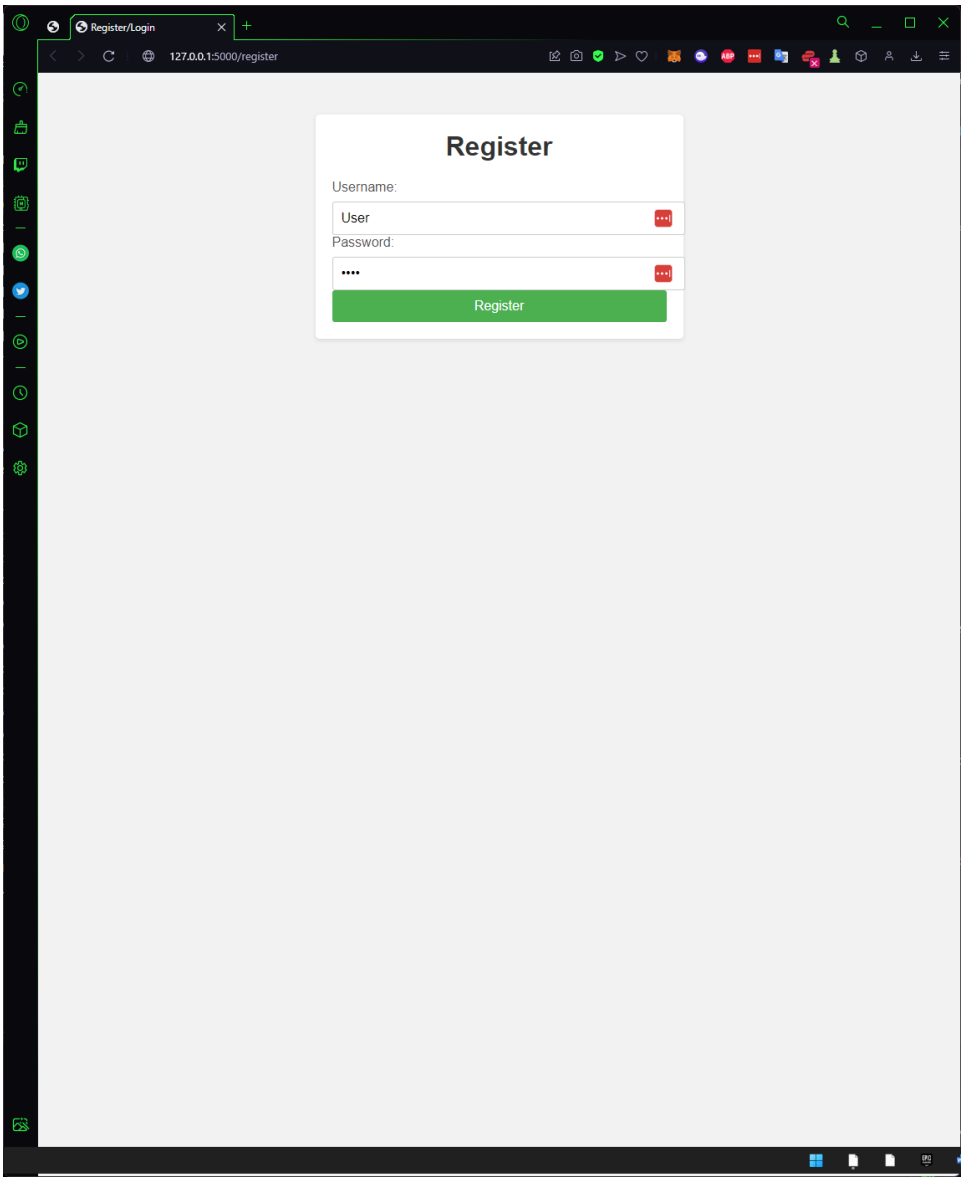
Launching Oauth22.oy It is very important to run this code using code runner because its impossible to run two codes simultaneously without this work around.



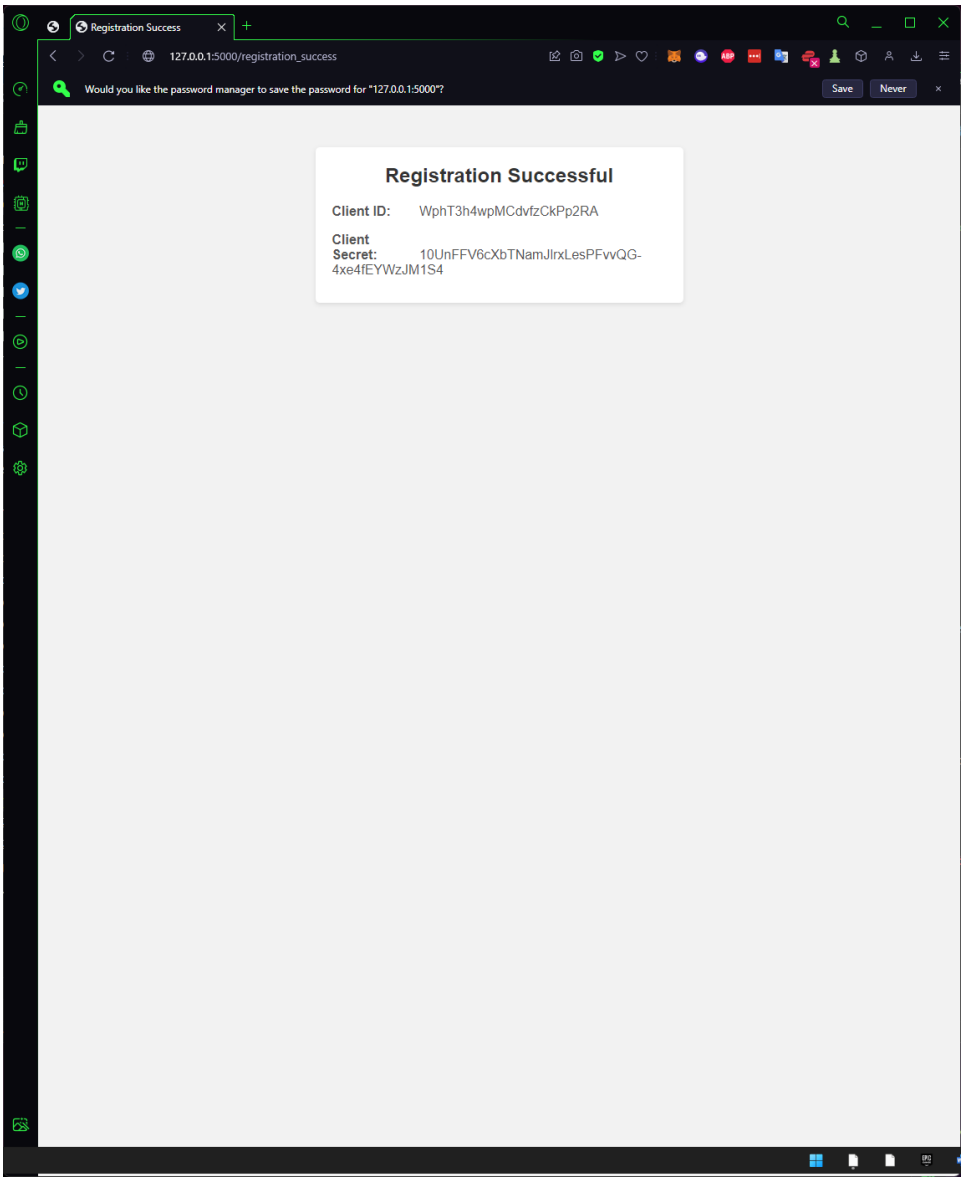
Starting login.py by using the default python interpreter



Now when we clicked on the 127.0.0.1:5000 link we get redirected to the register.html page.

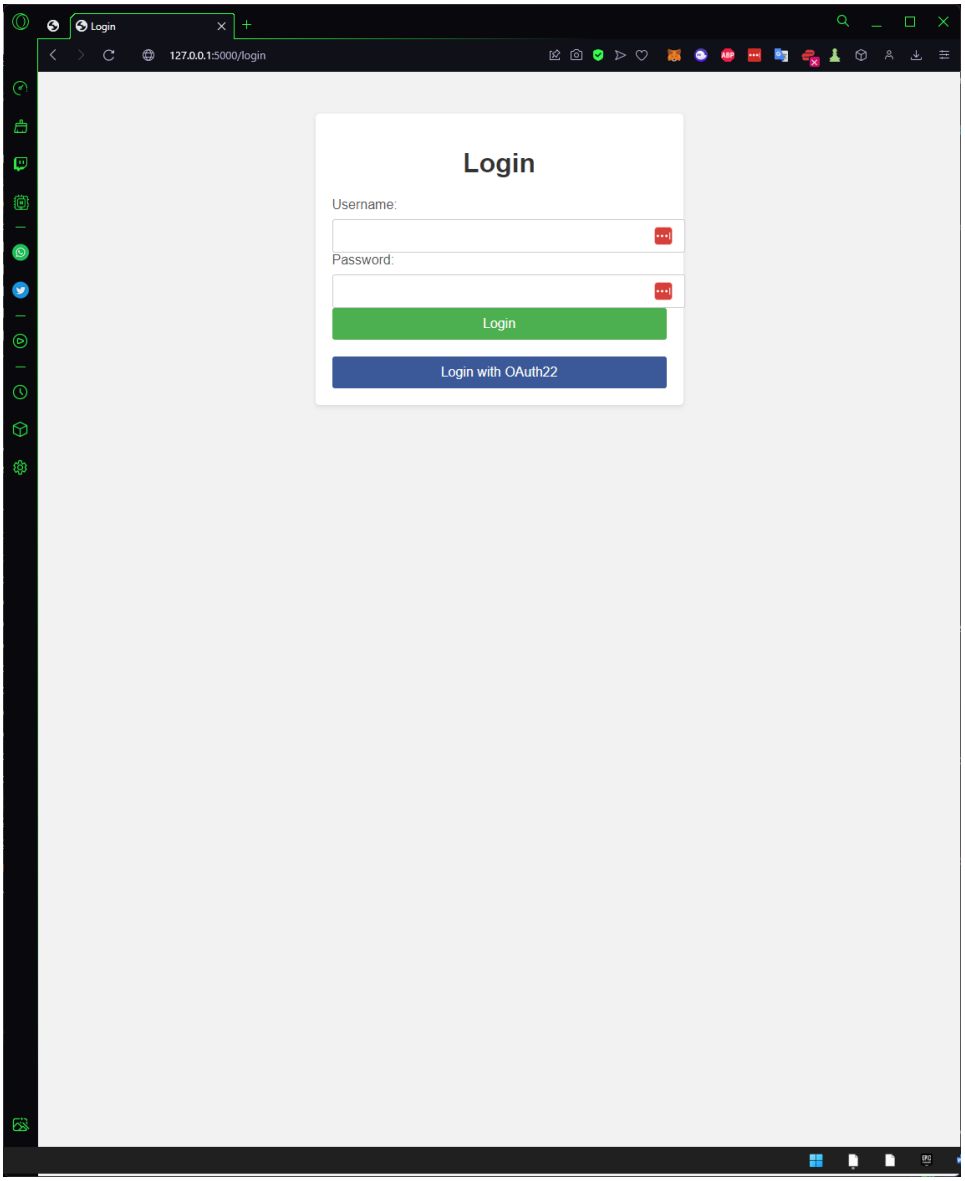


After pressing on register a new user gets created and then the user gets redirected to registration_success.html. This then will print the client id and the client secret

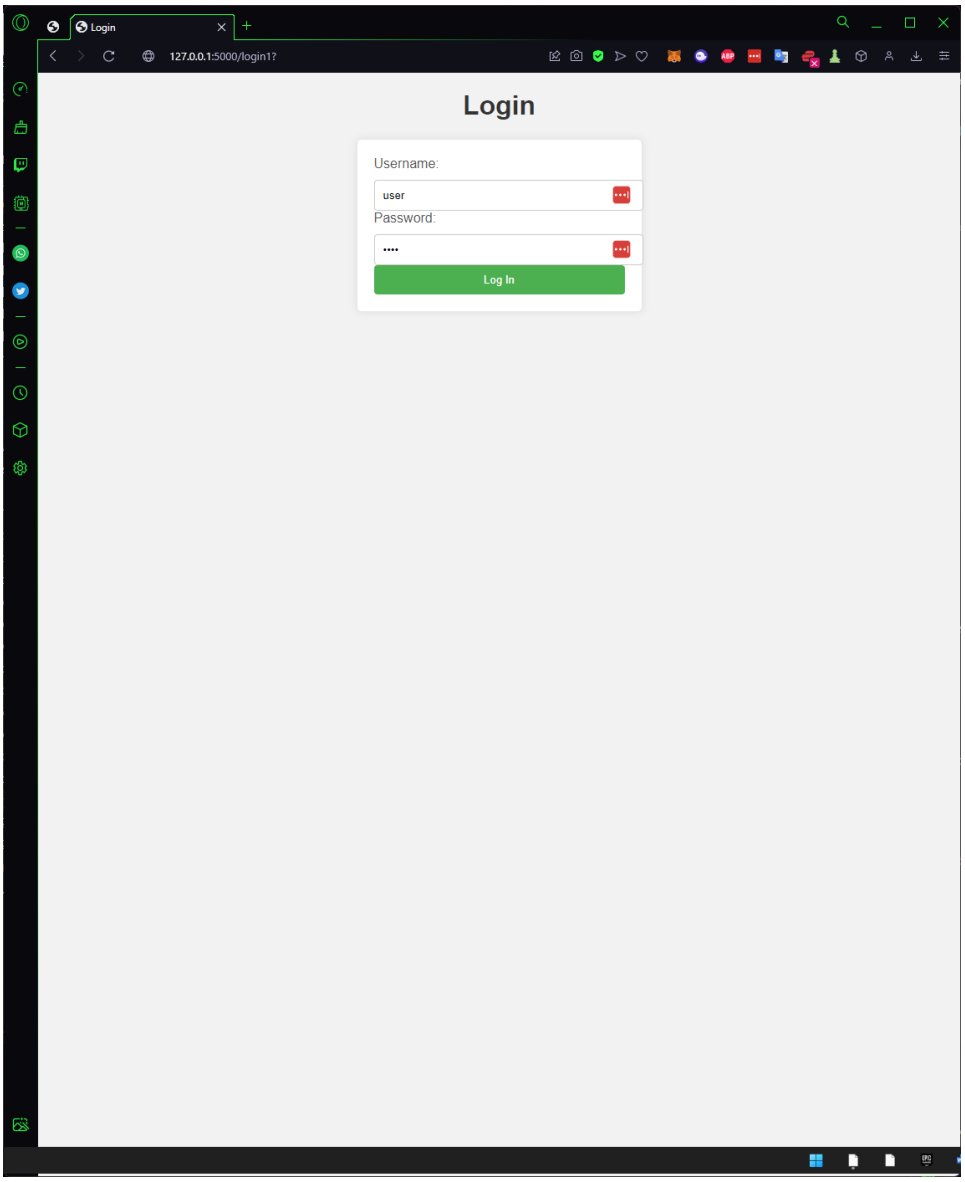


So now when we open the homepage using the other application login.py we can the press on the link that the program generates this will then redirect the user to

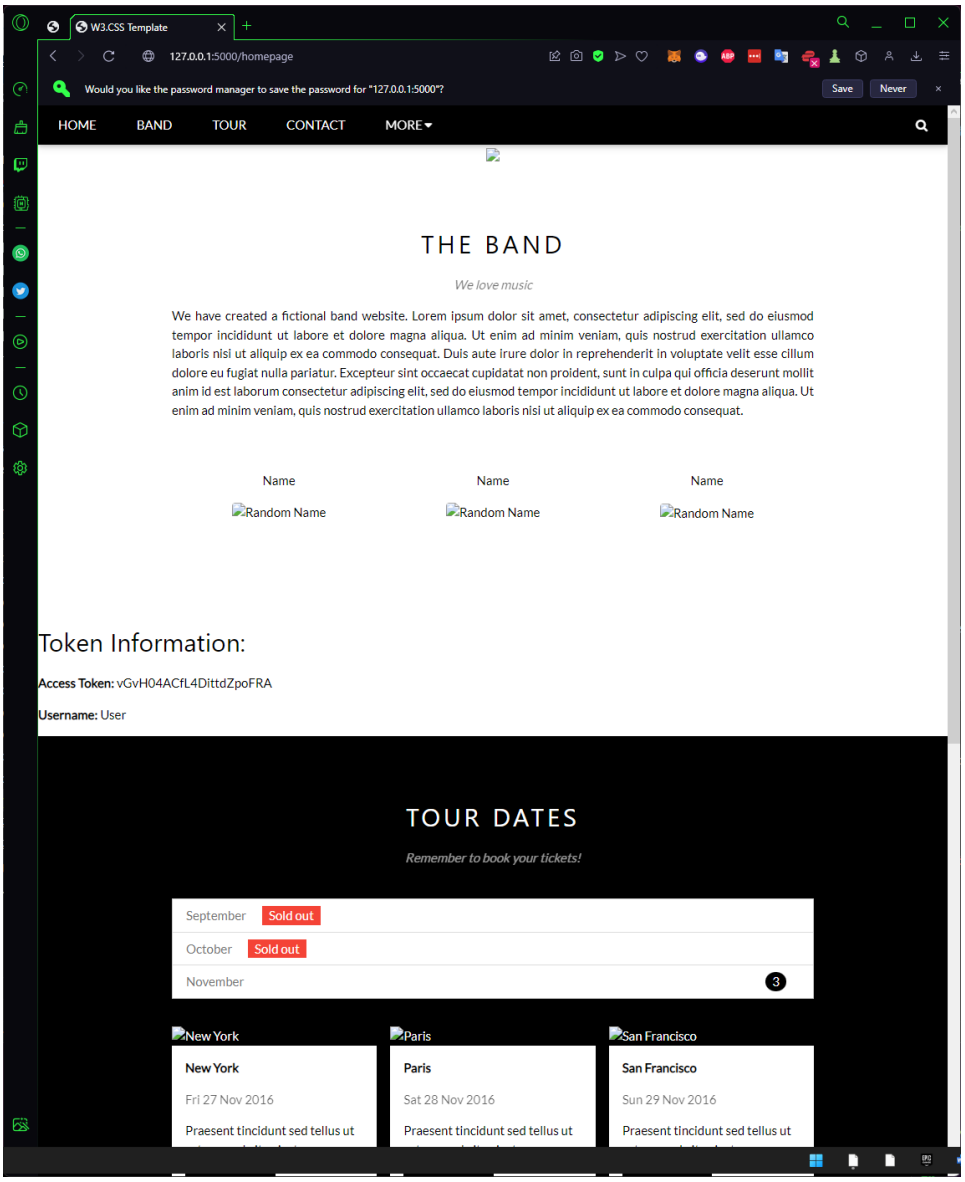
this login page this is a different login page than the one that the Oauth22.py generates. Now we press on login with Oauth22



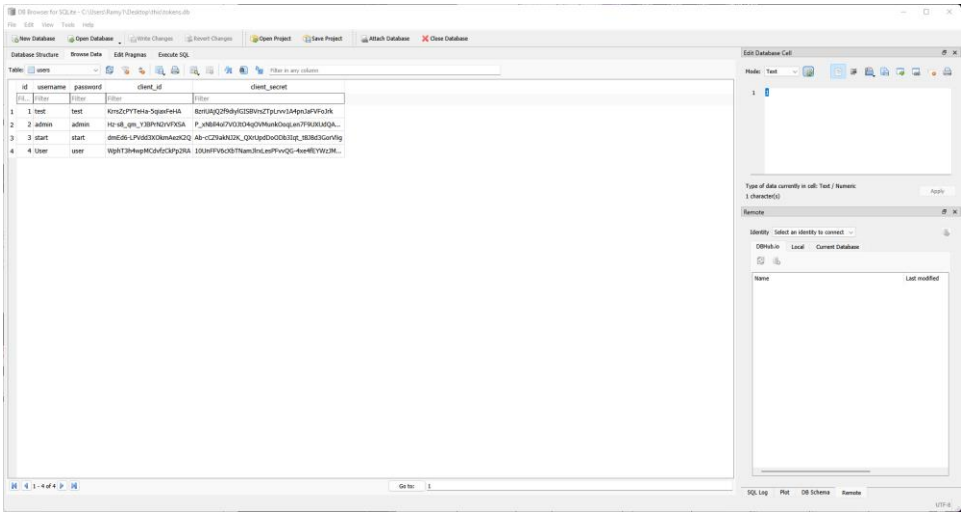
After pressing on the button we get redirected to the login page from Oauth22.py then we input the a already generated user name into it.



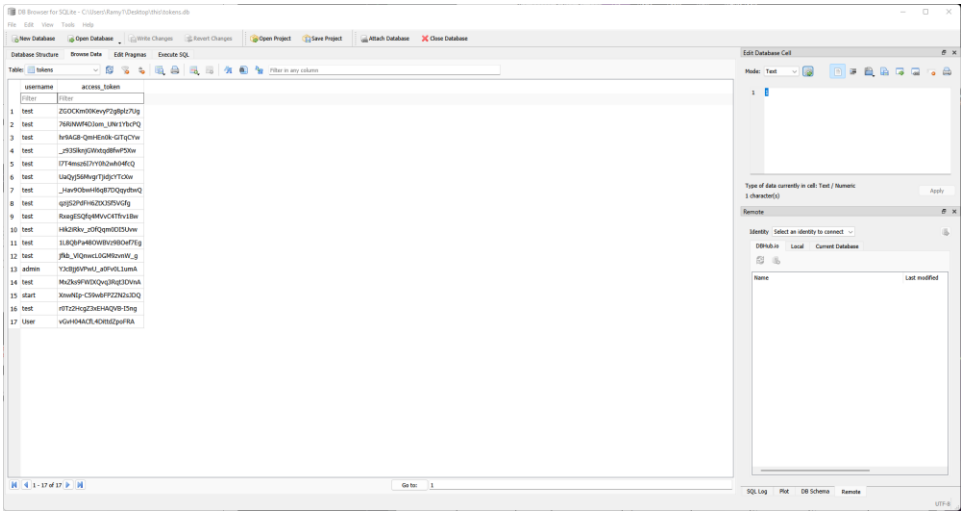
After successfully inputting the correct user name and password we get redirected to the homepage where I made it print the Access token and then the retrieved user name from the user that we have used to log into the website



Here I provided some screenshots to display the users and their client_id's and their client_secret



This one shows the Access tokens that have been generated using the Oauth library.



https://github.com/Coventry-TKH/cw2-Ramy2100782