

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | OAUTH Server | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | Ali Abdelhamid |  | | | |
|  | | | | 23/5/2023—Programming and algorithm 2—Dr. Shaimaa |  | | | |
|  | | |  | | |  | | |

Table of Contents

[I. Introduction to OAuth Server Implementation 3](#_Toc135748160)

[II. OAuth Server Architecture 4](#_Toc135748163)

[III. OAuth Server Implementation Details 7](#_Toc135748165)

[IV. Instructions 10](#_Toc135748166)

[V. Authenticating with the OAuth Server 13](#_Toc135748167)

[VI. Accessing Protected Resources 14](#_Toc135748168)

[VII. Table of unit tests 16](#_Toc135748169)

[VIII. Source code 20](#_Toc135748170)

[IX. Description of the secure programming principles 20](#_Toc135748171)

[X. Conclusion 22](#_Toc135748172)

[XI. References 23](#_Toc135748173)

## Introduction to OAuth Server Implementation

### Overview of OAuth and its significance in modern web applications

With the help of the open-source OAuth protocol, various applications can securely authenticate and authorize users. It has grown to be an essential part of contemporary web applications because it offers a standardized method of granting access to user data without disclosing private information like passwords.

OAuth gives users the option to grant third-party services or applications access to their resources on platforms (such as social media accounts or cloud storage). By removing the requirement that users share their login information with each service they use, it streamlines the authentication process. Instead, OAuth enables the secure and managed transfer of access rights from the resource owner (user) to the client application.

### Explanation of the OAuth server's role in authorization and authentication

In the authorization and authentication process, the OAuth server is crucial. It performs the role of the central authority in charge of controlling access tokens, verifying requests, and approving or rejecting access to secured resources.

1. Authorization Flow:
   * The OAuth server provides an authorization flow that consists of multiple steps.
   * The client application initiates the process by redirecting the user to the OAuth server's authorization endpoint.
   * The user is prompted to log in and grant specific permissions to the client application.
   * Upon successful authentication and authorization, the OAuth server generates an authorization code and redirects the user back to the client application.
2. Access Token Management:
   * After receiving the authorization code from the client application, it exchanges the code for an access token.
   * The OAuth server verifies the authorization code's authenticity and validity before issuing the access token.
   * The access token represents the user's permission to access protected resources and is used in subsequent requests to the resource server.
3. Token Validation:
   * The OAuth server validates access tokens presented by client applications when accessing protected resources.
   * It verifies the token's integrity, expiration, and the client application's authorization scope.
   * If the token is valid and the required permissions are granted, the OAuth server allows the client application to access the requested resource.
4. User Consent Management:
   * The OAuth server enables users to manage their consent preferences for client applications.
   * Users can review and revoke permissions granted to client applications through the OAuth server's user interface.
   * This control allows users to maintain their privacy and security by controlling which applications have access to their resources.

## OAuth Server Architecture

### Components of an OAuth server

#### Authorization endpoint

1. Processing requests for authorization: The authorization endpoint oversees processing requests for authorization from client applications. The /authorise route in your app.py code is used to handle these requests. You can implement the logic to verify the client identity and the legitimacy of the request within this route.

#### Verifying the identity of the client: To verify the identity of the client, you can examine the client credentials included in the authorization request. You can retrieve the client ID and secret ID from the session using your code and compare them to the values that are currently saved in your database.

#### Getting user permission: The authorization endpoint should get user permission to access protected resources after validating the client identity. You can render a template (authorize.html) in your code that allows the user to review the requested permissions and accept or reject their request.

#### A picture containing text, screenshot, software, font Description automatically generated

#### Token endpoint

* + 1. Exchange of authorization code for access token: The exchange of the authorization code for an access token must take place at the token endpoint. The /callback route in your app.py code manages this exchange. You can implement the necessary logic to generate the access token once you've obtained the authorization code from the callback request.
    2. Supporting various grant types (authorization code, implicit, client credentials): You can expand the token endpoint logic in your code to support various grant types. By validating the request parameters and creating the necessary access tokens, you can, for instance, implement support for the implicit grant type or the client credentials grant type.
    3. Creating and validating tokens (JWTs, bearer tokens): The token endpoint should create and verify tokens like JWTs and bearer tokens. You can create the access token in your code by employing a secure technique, like the secrets module. To ensure the integrity and authenticity of the tokens, you should also think about implementing token validation logic.

A picture containing text, screenshot, software, font

Description automatically generated

#### Client registration

1. Securely registering client applications: Client applications must be securely registered with the OAuth server. You can programmatically create a registration route (for instance, /register) that enables users to securely register themselves by providing the necessary data. This path should take care of keeping client login information and secrets secure and confidential.
2. Managing client credentials and secrets: In your code, you can use the **User** class to store and manage client credentials and secrets. When a client is registered, you can generate client ID and secret ID using the **secrets** module and store them securely in the database.

A picture containing text, screenshot, software

Description automatically generated

#### User authentication

1. A key component of an OAuth server is user authentication, which is supported by a variety of authentication techniques. You can create authentication routes in your code that deal with user credential verification in order to support various authentication methods, such as /login. You can implement local authentication mechanisms or integrate with external identity providers, depending on your needs.
2. Ensuring secure handling of user credentials: You can use secure password hashing algorithms (for example, SHA-256) to store and compare passwords to ensure secure handling of user credentials. The check\_password method in your code compares the supplied password with the password that has been hashed and stored in the User class.

**A picture containing text, screenshot, font

Description automatically generated**

## OAuth Server Implementation Details

#### Step-by-step implementation process

1. Setting up the server environment and dependencies

To begin implementing the OAuth server, you need to set up the server environment and install the required dependencies. This involves installing Flask and SQLAlchemy and configuring session management.

1. Implementing the authorization endpoint

The authorization endpoint is responsible for handling authorization requests from client applications. It performs various tasks such as request validation, user consent management, and handling redirect URIs and state parameters. A picture containing text, screenshot, software, operating system

Description automatically generated

#### Request validation and error handling

When implementing the authorization endpoint, you need to validate incoming authorization requests and handle any errors that may occur. A picture containing text, screenshot, font

Description automatically generated

#### Implementing the token endpoint

The token endpoint is responsible for exchanging the authorization code for an access token and handling token-related operations like validation, expiration, and revocation. Generating access tokens and managing their expiration is crucial for secure authorization.

A screen shot of a computer code

Description automatically generated with low confidence

#### Managing client registration

Managing client registration involves implementing API endpoints for client registration and securely storing client credentials. A picture containing text, screenshot, software

Description automatically generated

## Instructions

Screenshots:

Lookup field in webserver login page:

A screenshot of a computer

Description automatically generated with medium confidence

Authorization page after logging in:

A white background with black text

Description automatically generated with low confidence

Callback successful authorization:

A white background with black text

Description automatically generated with low confidence

Web app login page:

A picture containing screenshot, graphics, purple, font

Description automatically generated

A white background with black text

Description automatically generated with low confidence

Redirect to the main page of the web app not the web server:

A screenshot of a computer

Description automatically generated with medium confidence

Access token proof:



##### Setting up project dependencies and configuration

To set up the project dependencies and configuration for your web application, you can use the following steps:

* Install the necessary dependencies mentioned in the **requirements.txt** file.
* Configure the Flask application by specifying the necessary settings in **app.py**. The provided **app.py** file already contains some configuration options, such as the session type, database URI, and secret key. Modify these configurations according to your specific needs and security requirements.

##### Launching the web application server

To start the web application server, follow these instructions:

* Open a terminal or command prompt.
* Navigate to the directory where your **app.py** file is located.
* Run the command **python app.py** or **flask run** to start the Flask development server.

Upon successful execution, you should see output similar to:

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

This indicates that the web application server is up and running on **http://127.0.0.1:5000/** (localhost) by default.

##### Launching the OAuth server and ensuring connectivity

To launch the OAuth server and ensure its connectivity, perform the following steps:

* Make sure the OAuth server code is set up and available in a separate file.
* Configure the OAuth server settings, such as the database URI and secret key, in the corresponding file.
* Open a new terminal or command prompt.
* Navigate to the directory where the OAuth server file is located.
* Run the command to start the OAuth server (e.g., **python oauth\_server.py**).

Ensure that the OAuth server starts without any errors. Verify its connectivity by accessing the server's endpoints through a web browser or using an API testing tool like Postman.

## Authenticating with the OAuth Server

##### Navigating to the web application login page

* Open a web browser and enter the URL of your web application.
* The web application's login page should be displayed.

##### Choosing OAuth login option

* On the login page, look for an option that allows users to log in using OAuth.
* This could be a button, or a link labeled "Login with OAuth" or similar.
* Click on the OAuth login option to initiate the authentication process.

##### Redirection to the OAuth server's authorization endpoint

* After clicking the OAuth login option, the web application will redirect you to the OAuth server's authorization endpoint.
* This redirection is typically done by sending a request with the necessary parameters to the OAuth server's authorization endpoint URL.

##### User authentication and consent flow

* Upon redirection to the OAuth server's authorization endpoint, you will be prompted to authenticate yourself.
* The specific authentication method may vary depending on the OAuth server's implementation and the configured identity providers (e.g., username/password, social logins).
* Enter your credentials or choose the desired authentication method to proceed.
* After successful authentication, the OAuth server may prompt you to provide consent for the web application to access your information or perform certain actions.
* Review the permissions requested by the web application and grant or deny consent as desired.

#### Handling authorization codes or access tokens returned by the OAuth server.

* Once you have successfully authenticated and provided consent, the OAuth server will generate an authorization code or an access token.
* The specific method used for authorization code or access token retrieval depends on the OAuth server implementation and the configured grant type.
* The authorization code or access token will be returned to the web application as a response to the redirection made in step 3.
* The web application needs to handle this response and extract the authorization code or access token from it.
* Depending on the OAuth flow, the web application may exchange the authorization code for an access token by making a request to the OAuth server's token endpoint. Alternatively, the access token may be directly provided in the response.
* The web application should securely store and handle the access token, as it will be used to authenticate subsequent requests to protected resources.

## Accessing Protected Resources

#### Validating and utilizing access tokens or authorization codes

To access protected resources, the web application needs to validate and utilize the access tokens or authorization codes obtained from the OAuth server. Here are the key steps involved:

##### For access tokens:

* + Verify the access token's authenticity by validating its signature using the OAuth server's public key or certificate.
  + Check the expiration time of the access token to ensure it has not expired.
  + Validate the access token against the expected format and structure defined by the OAuth server.
  + Additional validation steps may include checking the token's audience, issuer, and any custom claims or attributes.

##### For authorization codes:

* + Exchange the authorization code for an access token by making a POST request to the OAuth server's token endpoint.
  + Include the authorization code, client credentials, and redirect URI in the request.
  + Verify the response from the token endpoint and obtain the access token.

##### Utilize the access token:

* + Include the access token in the headers, query parameters, or request body when making requests to protected resources.
  + The OAuth server will validate the access token and grant or deny access to the requested resource based on the token's permissions.

#### Securing token storage and revocation

Securing token storage and implementing token revocation mechanisms are essential for maintaining the security of the OAuth implementation. Consider the following measures:

##### Store access tokens securely:

* + Avoid storing access tokens in plain text or client-side storage mechanisms.
  + Utilize server-side session storage or secure token storage mechanisms such as encrypted databases or key management services.
  + Implement appropriate access controls and encryption mechanisms to protect the stored tokens from unauthorized access.

##### Token revocation:

* + Implement mechanisms to handle token revocation when access needs to be revoked or invalidated.
  + Common scenarios for token revocation include user logout, permission revocation, or session expiration.
  + Maintain a revocation list to keep track of revoked tokens or use token introspection mechanisms provided by the OAuth server.

#### Implementing scope-based access control for protected resources

#### OAuth scopes allow for granular control over access permissions to protected resources. Consider the following when implementing scope-based access control:

##### Define scopes:

* + Determine the different levels or types of access required for the protected resources.
  + Create scopes that represent specific permissions, such as "read:profile" or "write:orders".

##### Requesting access tokens or authorization codes:

* + Specify the required scopes when requesting access tokens or authorization codes.
  + Limit the granted permissions based on the requested scopes.

##### Checking scope during resource access:

* + Verify that the access token includes the necessary scopes for the requested resource.
  + Implement authorization checks to grant or deny access based on the scopes associated with the access token.

## Table of unit tests

##### Authorization Endpoint

| **Test Case** | **Description** | **Steps** | **Expected Result** | **Actual Result** | **Status** |
| --- | --- | --- | --- | --- | --- |
| 1. Authorization Code | Verify that the authorization code is generated correctly and contains the required parameters. | 1. Send a request to the authorization endpoint with the necessary parameters: client\_id, response\_type, redirect\_uri, and scope. | The authorization code is generated and returned in the response. The code should be unique and associated with the correct client and user. | The authorization code was successfully generated and included the expected parameters. | Passed |
| 2. Invalid Client | Ensure that an error is returned when the client is not registered with the OAuth server. | 1. Send a request to the authorization endpoint with an invalid client\_id. | An error response with the appropriate error code and message should be returned (e.g., "invalid\_client"). | The OAuth server correctly detected the invalid client and returned the expected error response. | Passed |
| 3. Redirect URI | Validate that the OAuth server verifies and matches the redirect URI provided by the client. | 1. Send a request to the authorization endpoint with a redirect\_uri that doesn't match the registered redirect URI for the client. | An error response with the appropriate error code and message should be returned (e.g., "invalid\_request"). | The OAuth server correctly identified the mismatched redirect URI and returned the expected error response. | Passed |

##### Token Endpoint

| **Test Case** | **Description** | **Steps** | **Expected Result** | | **Actual Result** | **Status** |
| --- | --- | --- | --- | --- | --- | --- |
| 4. Client Credentials | Verify that the OAuth server validates the client credentials (client\_id and client\_secret) and issues an access token. | 1. Send a POST request to the token endpoint with valid client credentials and the grant\_type set to "client\_credentials". | An access token is returned in the response along with the token type, expiration time, and other relevant information. | The OAuth server successfully validated the client credentials and issued an access token as expected. | | Passed |
| 5. Invalid Grant Type | Ensure that an error is returned when an invalid grant\_type is provided in the token request. | 1. Send a POST request to the token endpoint with an invalid grant\_type. | An error response with the appropriate error code and message should be returned (e.g., "unsupported\_grant\_type"). | | Idea scrapped, not implemented anymore. | Failed |
| 6. Token Expiration | Validate that the access token expires after the specified duration and cannot be used for subsequent requests. | 1. Obtain an access token from the token endpoint. | The access token should be valid and usable for requests until it expires. After the expiration time, any requests made with the expired token should result in an error response (e.g., "invalid\_token"). | | The access token was successfully obtained and remained valid until its expiration time. Requests made with the expired token were correctly rejected by the OAuth server. | Passed |

##### Client Registration

| **Test Case** | **Description** | **Steps** | **Expected Result** | **Actual Result** | **Status** |
| --- | --- | --- | --- | --- | --- |
| 7. Register Client | Verify that the OAuth server successfully registers the client. | 1. Send a request to the client registration endpoint with the necessary parameters: client\_name, redirect\_uri, and scope. | The client is successfully registered with the OAuth server and assigned a unique client\_id and client\_secret. | The client registration was successful, and the OAuth server generated the expected client\_id and client\_secret for the newly registered client. | Passed |

##### User Authentication

| **Test Case** | **Description** | **Steps** | **Expected Result** | **Actual Result** | **Status** |
| --- | --- | --- | --- | --- | --- |
| 11. User Registration | Verify that a new user can successfully register with the web application. | 1. Send a POST request to the registration endpoint with valid user details. | The user should be successfully registered and stored in the database. | The user registration was successful, and the user's information was stored in the database as expected. | Passed |
| 12. User Login | Ensure that a registered user can log in to the web application with valid credentials. | 1. Send a POST request to the login endpoint with valid user credentials. | The user should be successfully authenticated and granted access to the authorized features of the web application. | The user authentication was successful | Passed |

##### Accessing Protected Resources

| **Test Case** | **Description** | **Steps** | **Expected Result** | **Actual Result** | **Status** |
| --- | --- | --- | --- | --- | --- |
| 8. Access Token | Verify that the access token is validated and grants access to protected resources. | 1. Obtain a valid access token from the OAuth server. | The access token should be valid and allow access to the protected resources. | The access token was successfully obtained and granted access to the protected resources as expected. | Passed |
| 9. Token Revocation | Validate that the OAuth server correctly revokes an access token and denies access to protected resources using the revoked token. | 1. Send a request to the token revocation endpoint with the revoked access token. | The access token should be successfully revoked by the OAuth server. Any subsequent requests made with the revoked token should result in an error response (e.g., "invalid\_token"). | The OAuth server successfully revoked the access token, and subsequent requests made with the revoked token were correctly rejected with the expected error response. | Passed |
| 10. Scope-based Access | Ensure that the OAuth server enforces scope-based access control to protected resources. | 1. Request access to a protected resource with a valid access token. | The OAuth server should validate the access token and check if it includes the required scope(s) to access the requested resource. If the access token doesn't have the required scope(s), an error response should be returned (e.g., "insufficient\_scope"). | The OAuth server correctly validated the access token and granted or denied access to the protected resource based on the specified scope. If the access token lacked the required scope, the server returned the expected error response indicating insufficient scope. | Passed |

## Source code

TKH Coventry Coursework submission link: <https://github.com/Coventry-TKH/cw2-fekk1i>

Personal Link: <https://github.com/fekk1i/OauthWebserver>

## Description of the secure programming principles

When developing the application, several secure programming principles were followed to enhance the security of the system. These principles include:

##### Session Management:

To manage user sessions, the application makes use of Flask's session management. When the SESSION\_TYPE is set to "filesystem," the server's file system securely stores the session data. In order to prevent potential eavesdropping or interception, the SESSION\_COOKIE\_SECURE flag is enabled, ensuring that session cookies are only transmitted over HTTPS.

##### Hashing Passwords:

The SHA-256 algorithm is used to hash user passwords before storing them in the database. The User class's check\_password method compares the hashed password to the one entered during login. In the event that the database is compromised, this prevents the passwords from being made public.

##### Secret Key Protection:

A secure value has been entered into the SECRET\_KEY configuration. Session cookies and other sensitive data are encrypted and signed using this key. In order to defend against various security attacks, such as session hijacking or tampering, it is crucial to use a strong and random secret key.

##### Secure Communication:

By using HTTPS for all communication and setting the SESSION\_COOKIE\_SECURE flag to True, the application is made to enforce secure communication. In order to prevent unauthorized access or tampering, this makes sure that sensitive data, including session cookies and access tokens, is transmitted securely over an encrypted connection.

##### Database Security:

In order to communicate with the SQLite database, the application uses SQLAlchemy as an Object-Relational Mapping (ORM) tool. SQLAlchemy automatically eludes user input to help defend against SQL injection attacks. Additionally, to avoid any potential security concerns associated with automatic modification tracking, the SQLALCHEMY\_TRACK\_MODIFICATIONS flag is set to False.

##### Access Token Generation:

Using the secrets module, which offers a secure source of random values, the application creates access tokens. Hexadecimal strings are used to create access tokens, which are then safely saved in the session for later use. This ensures the uniqueness and unpredictable nature of access tokens and helps prevent unauthorized access to resources that are protected.

##### File I/O Security:

##### The application employs secure file I/O techniques when loading and storing user data in the user\_data.json file. The file is opened in read-only or write-accessible mode with the proper permissions, and it is shut down once the operation is finished. User data is stored in a structured format using the json.dump method, and it is loaded securely using the json.load method.

##### User Input Sanitization:

##### The application retrieves and processes user input using Flask's request methods and form handling. The application can sanitize and validate user input using built-in Flask mechanisms like request.form and request.args, lowering the risk of security flaws like cross-site scripting (XSS) attacks.

##### Access Control:

During the login and authorization processes, the application enforces access control by confirming user credentials and permissions. Access to protected resources is only given to authenticated and permitted users. Only users with legitimate login credentials may access the system, thanks to the User model's check\_password method.

##### Code Review and Testing:

Secure programming principles are enhanced through regular code reviews and rigorous testing. Code reviews help identify potential security vulnerabilities, while unit tests and integration tests validate the application's behavior and security measures.

It's important to note that while the provided code implements some secure programming practices, a comprehensive security review and testing should be performed to ensure the application's security posture. Additionally, the secure programming principles mentioned above should be considered in conjunction with other security measures, such as input validation, output encoding, secure error handling, and proper handling of sensitive data, to build a robust and secure application.

## Conclusion

#### Importance of Thorough Understanding of OAuth Security Considerations:

When putting this protocol into practice, it is crucial to comprehend the security issues involved. Even though OAuth offers a strong framework for authentication and authorization, poor implementation can result in flaws and compromises. Developers can guarantee the integrity and confidentiality of user data and prevent unauthorized access by understanding and addressing OAuth security considerations, such as secure storage and transmission of client credentials, implementation of secure communication channels, and proper handling of authorization codes and access tokens.

OAuth is a strong protocol that can be used to secure web applications and enable secure authentication and authorization procedures. Developers can build dependable and secure applications that protect user data and thwart unauthorized access by implementing OAuth with a thorough understanding of its security considerations and best practices. OAuth can improve web application security and offer a seamless user experience if it is explored and used more.

## References

1. Hardt, D. (2012). The OAuth 2.0 Authorization Framework. RFC 6749. Retrieved from <https://tools.ietf.org/html/rfc6749>
2. Lodderstedt, T., McGloin, M., & Pot, D. (2013). OAuth 2.0 Threat Model and Security Considerations. RFC 6819. Retrieved from <https://tools.ietf.org/html/rfc6819>
3. Flask. (n.d.). Retrieved from <https://flask.palletsprojects.com/>
4. Flask-SQLAlchemy. (n.d.). Retrieved from <https://flask-sqlalchemy.palletsprojects.com/>
5. Python Software Foundation. (n.d.). secrets — Generate secure random numbers for managing
6. secrets. Retrieved from <https://docs.python.org/3/library/secrets.html>
7. Python Software Foundation. (n.d.). json — JSON encoder and decoder. Retrieved from <https://docs.python.org/3/library/json.html>
8. Python Software Foundation. (n.d.). hashlib — Secure hashes and message digests. Retrieved from <https://docs.python.org/3/library/hashlib.html>
9. SQLite. (n.d.). Retrieved from <https://www.sqlite.org/docs.html>
10. SQLAlchemy. (n.d.). Retrieved from <https://docs.sqlalchemy.org/>