CyberSecurity – CA#3

Sahand Khoshdel – 810196607

Computer Assignment #3

Table of Contents

Page #

Introduction to OAuth 2.0 2

Registration of OAuth App on GitHub 5

1. Redirection to Authorization Server, User Authentication by GitHub 6

2. Redirection to Client, Passing Grant code (**Main Authorization**) 7

3. Client Authentication and Receiving Access Token __________ 8

4. Using Access Token to Obtain User's Information 9

Theoretical Questions 10

The 3'rd & 4'th step (Access Token Reception) is done automatically by applying changes in the "server.js" code **(Bonus Part)**

Introduction to OAuth 2.0

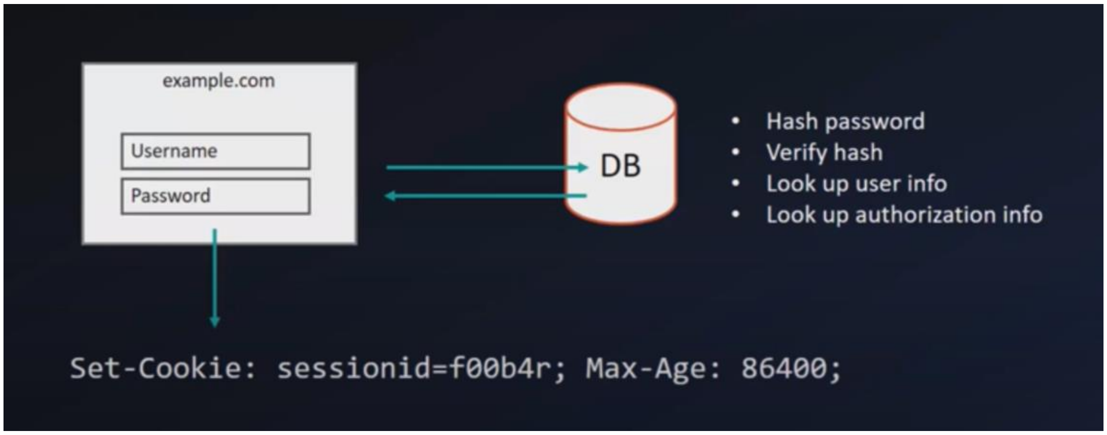
The purpose of this assignment is to perform **an undirect authorization** based on the protocol "OAuth 2".This **authorization protocol** is designed to acheive client developer simplicity while providing specific authorization flows for web and desktop applications. Basically when **a user wants to grant these applications to have access to his\her data on other websites, without handing them the password**, OAuth standard are used. This process is mainly known as **Access Delegation**.

- OAuth is not only an authentication protcol, but in fact it's more of an **authorzation standard.**
- OAuth differs from OATH, which is a reference architecture for authentcaiton
- OAuth is directly related to OpenID Connect. OpenID connect is actulaly an authentication layer built on top of OAuth 2.0

Motivation:

Generally **what happens in simple logins** is:

1. A client enters a username and password
2. The server searchs for the username in a database, compares the hash of the corresponding password and either verifies the client or not.(Authentication and Authorzation is actually done simoltanously in this process.)
3. The web applucation then usually drops a cookie into client's brower to keep track of the user's login and unnecessary requests.



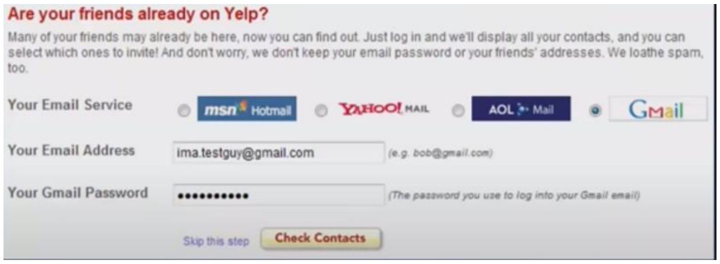
```
Set-Cookie: sessionid=f00b4r; Max-Age: 86400;
```

But this makes a web application to be responsible for securely storing and protecting user information, and providing and maintating this security isn't an easy thing to do. Thus, relying on an authorization server for authentication and authorization, is a way to solve this problem.

Let’s see what actually happened in the early days…

Well, this is obviously terrible!

The website is requesting for user’s gmail and password to login to his/her gmail password, gather needed information, and log out politely without any further investigtation :) . Any wise user certainly won’t trust this method.



Are your friends already on Yelp?

Many of your friends may already be here, now you can find out. Just log in and we'll display all your contacts, and you can select which ones to invite! And don't worry, we don't keep your email password or your friends' addresses. We loathe spam, too.

Your Email Service ○ msn Hotmail ○ YAHOO! MAIL ○ AOL Mail ● Gmail

Your Email Address ima.testguy@gmail.com (e.g. bob@gmail.com)

Your Gmail Password •••••••••• (The password you use to log into your Gmail email)

OAuth and OpenID are protocols that are designed to provide a framework for doing this process, eliminating the need to get the password.

OAuth allows access tokens to be issued to third party clients by an authorization server, with the approval of the resource owner. The third party then uses the access token to access the protected resources hosted by the resource server via the Web API. So let’s introduce some terminology in our context:

OAuth 2.0 Terminology:

- **Resource Owner**:
  The User that logs in via the HTTP login page**,** plays the role ot the resource owner who wants to grant the servers access to his information on a server such as GitHub

- **Authorization Server**:
  GitHub plays the role of the authorization server, via an OAuth application, providing a token to the client to give the server.

- **Resource Server:**
  The API or the server which hold user’s intended information is known as the resource server, which in our case is GitHub too.

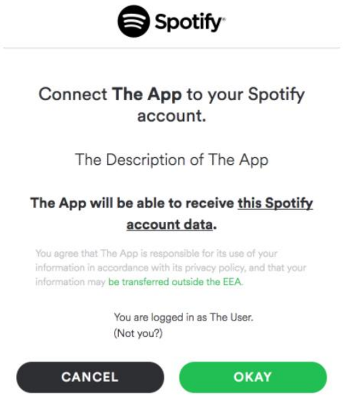
- **Client:**
  Server.js plays the role of the web application requesting user’s data. It is also reffered to as client againsts the authorization server. Therefore, it is provided with ClientID and Client Secret.

- **Authorization Grant:**
  This is the grant that proves the user has allowed the web application to use the information on the resource server, by clicking the button, and it’s done through the whole procedure.

- **Redirect URI:**(not exactly = URL)
  The Authorization server should know where to redirect back to, in order to talk to the application back and forth. This is provided via a resourse indicator called the redirect URI.

- **Access Token:**
  Access token is the final approval that the application(client), is authorized to use users info, Thus it is used to communicate with the API in order to receive data.

An example of a pop-up page from a web application asking the user to authorize it using his/her data stored on Spotify, is shown below:



Spotify

Connect **The App** to your Spotify account.

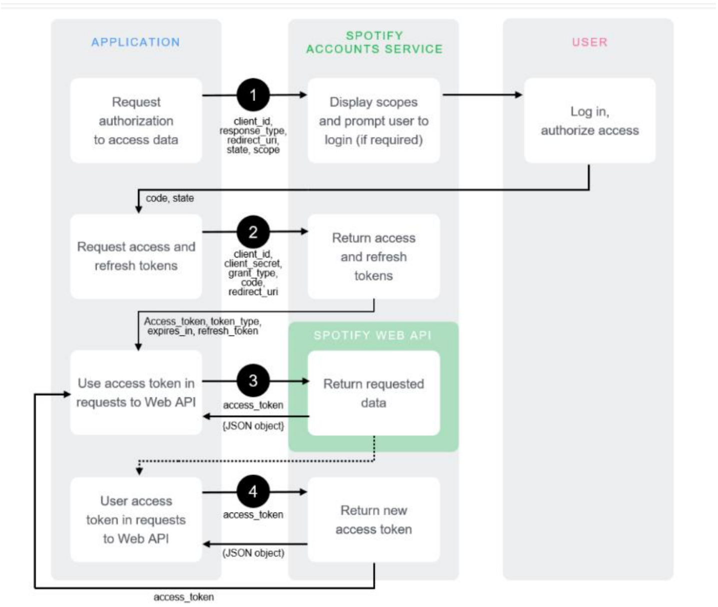
The Description of The App

The App will be able to receive this Spotify account data.

You agree that The App is responsible for its use of your information in accordance with its privacy policy, and that your information may be transferred outside the EEA.

You are logged in as The User.
(Not you?)

I also found a good scheme desribing the whole OAuth process in these main 4 steps.



APPLICATION
SPOTIFY ACCOUNTS SERVICE
USER
Request authorization to access data
1
client_id, response_type, redirect_uri, state, scope
Display scopes and prompt user to login (if required)
Log in, authorize access
code, state
Request access and refresh tokens
2
client_id, client_secret, grant_type, code, redirect_uri
Return access and refresh tokens
Access_token, token_type, expires_in, refresh_token
SPOTIFY WEB API
Use access token in requests to Web API
3
access_token
{JSON object}
Return requested data
User access token in requests to Web API
4
access_token
{JSON object}
Return new access token
access_token

1. A web application reqeusts to access user's data through a button on the webpage. By clicking on the button the user goes to the redirect_uri, the Auth server then requests for user login or not (if already logged in, keeps track via cookies). Finally the user logs in and passes an authorization code generated by the authorization server to the application(client)

2. The application sends the code + client_id and client_secret (for the application), specifies "Authorization Code" grant type to the server, and if the code is valid, an access token is given back for further communicatoin with the server's API[1].

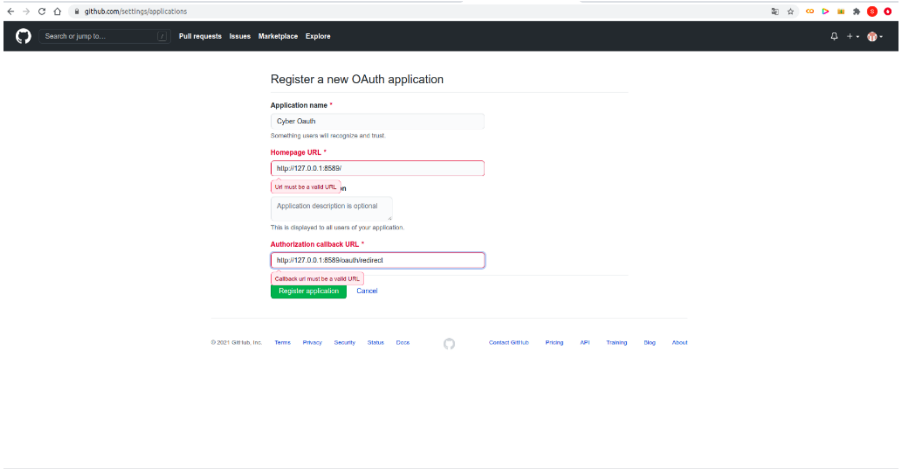
3. The web application sends the access token to the API and receives a JSON object including user's information.

4. Simply returning to the previous step when the access token is expired or the API comunication is terminated for other reasons.

[1] An Application Program Interface (API) is a kind of tool to access data for a user through other servers or applications, while keeping the messy stuff and communications away from the user.

Registration of OAuth App on GitHub

After logging in to our GitHub account, by following the Instructions, we can create a new OAuth app.


github.com/settings/applications

Register a new OAuth application

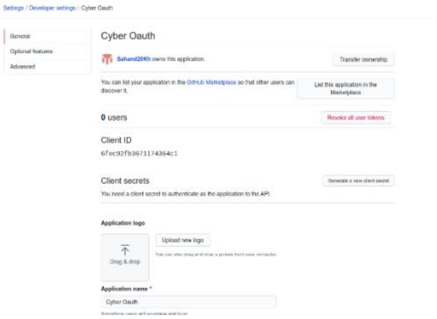
Application name *
Cyber Oauth
Something users will recognize and trust.

Homepage URL *
http://127.0.0.1:8589/
Url must be a valid URL

Application description is optional
This is displayed to all users of your application.

Authorization callback URL *
http://127.0.0.1:8589/oauth/redirect
Callback url must be a valid URL

First we choose a name for our app. The homepage URL is simply the login-page we are going to use. It's a localhost page on our device and it's listening on port 8589. The authorization callback URL is the address where the OAuth application is redirected after communicating with the authorization server on GitHub.


Settings / Developer settings / Cyber Oauth

Cyber Oauth

Sahand20Kh owns this application.

You can list your application in the GitHub Marketplace so that other users can discover it.

0 users

Client ID
6fec92fb3671174364c1

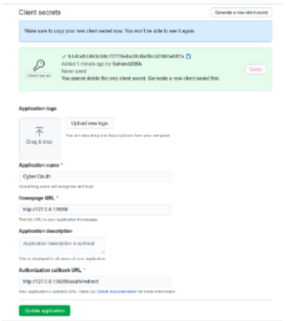
Client secrets
You need a client secret to authenticate as the application to the API.

Application logo

Application name *
Cyber Oauth

After receiving ClientID, we request to receive a client secret as well. This is the shared key between the web application and the authorization server to provide client authentication.

Finally we click on "Update application" button and our OAuth app (authorizaiton server) is ready to use.


Client secrets

Make sure to copy your new client secret now. You won't be able to see it again.

✓ b14be51493c98c72779e8a26d9e5bcd2660e867a
Added 1 minute ago by Sahand20Kh
Never used
You cannot delete the only client secret. Generate a new client secret first.

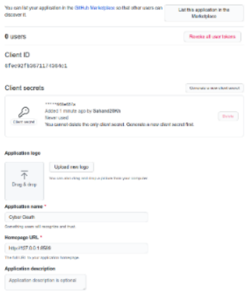
Application logo

Application name *
Cyber Oauth
Something users will recognize and trust.

Homepage URL *
http://127.0.0.1:8589
The full URL to your application homepage.

Application description
Application description is optional
This is displayed to all users of your application.

Authorization callback URL *
http://127.0.0.1:8589/oauth/redirect
Your application's callback URL. Read our OAuth documentation for more information.


You can list your application in the GitHub Marketplace so that other users can discover it.

0 users

Client ID
6fec92fb3671174364c1

Client secrets

*****6d8e867a
Added 1 minute ago by Sahand20Kh
Never used
You cannot delete the only client secret. Generate a new client secret first.

Application logo

Application name *
Cyber Oauth
Something users will recognize and trust.

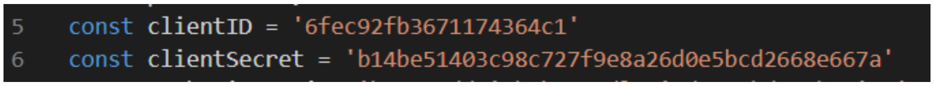
Homepage URL *
http://127.0.0.1:8589
The full URL to your application homepage.

Application description
Application description is optional

List this application in the Marketplace
Revoke all user tokens
Generate a new client secret
Client secret
Delete
Upload new logo
Drag & drop

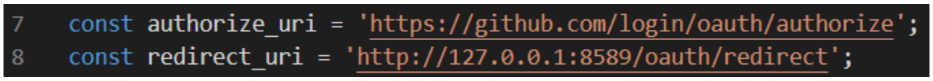
Step 1. Redirection to Auth Server, User Authentication by GitHub

First of all we should add the ClientID and Client Secret to the “server.js” code. These parameters are used to authorize the client(web application) by the authorization server, so a third(actually fourth) party can’t impersonate him/herself as the web application and try to access user’s data.



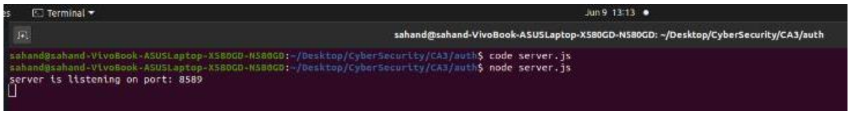
```
const clientID = '6fec92fb3671174364c1'
const clientSecret = 'b14be51403c98c727f9e8a26d0e5bcd2668e667a'
```

The authorization and redirect URI should also be defined. The authorization URI is the address(indicator) of the authorization page on GitHub (https://github.com/login/oauth/authorize). The redirect URI is the localhost page we want to return to (https://127.0.0.1:8589/oauth/redirect). These URI’s should be fed as parameters for “link.href” of our login page.



```
const authorize_uri = 'https://github.com/login/oauth/authorize';
const redirect_uri = 'http://127.0.0.1:8589/oauth/redirect';
```

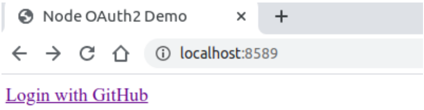
By running the file “server.js” we see a confirmation message: “server is listening on port 8589”.



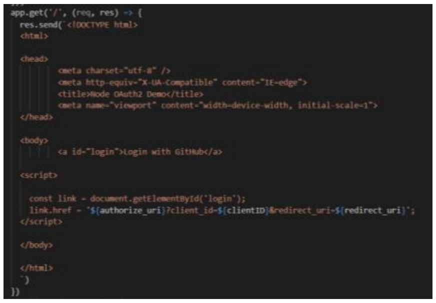
Terminal
Jun 9 13:13
sahand@sahand-VivoBook-ASUSLaptop-X580GD-N580GD: ~/Desktop/CyberSecurity/CA3/auth

```
sahand@sahand-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/CyberSecurity/CA3/auth$ code server.js
sahand@sahand-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/CyberSecurity/CA3/auth$ node server.js
server is listening on port: 8589
```

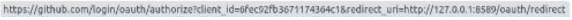
Now we should design the login page in “HTML” sas a localhost homepage.



Login with GitHub



```
app.get('/', (req, res) => {
  res.send(`<!DOCTYPE html>
  <html>

  <head>
        <meta charset="utf-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <title>Node OAuth2 Demo</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>

  <body>
        <a id="login">Login with GitHub</a>

  <script>

    const link = document.getElementById('login');
    link.href = `${authorize_uri}?client_id=${clientID}&redirect_uri=${redirect_uri}`;
  </script>

  </body>

  </html>
  `)
})
```



https://github.com/login/oauth/authorize?client_id=6fec92fb3671174364c1&redirect_uri=http://127.0.0.1:8589/oauth/redirect

Authorization and Redirect URL’s are shown in the bottom left corner when holding the mouse on the hyperlink.

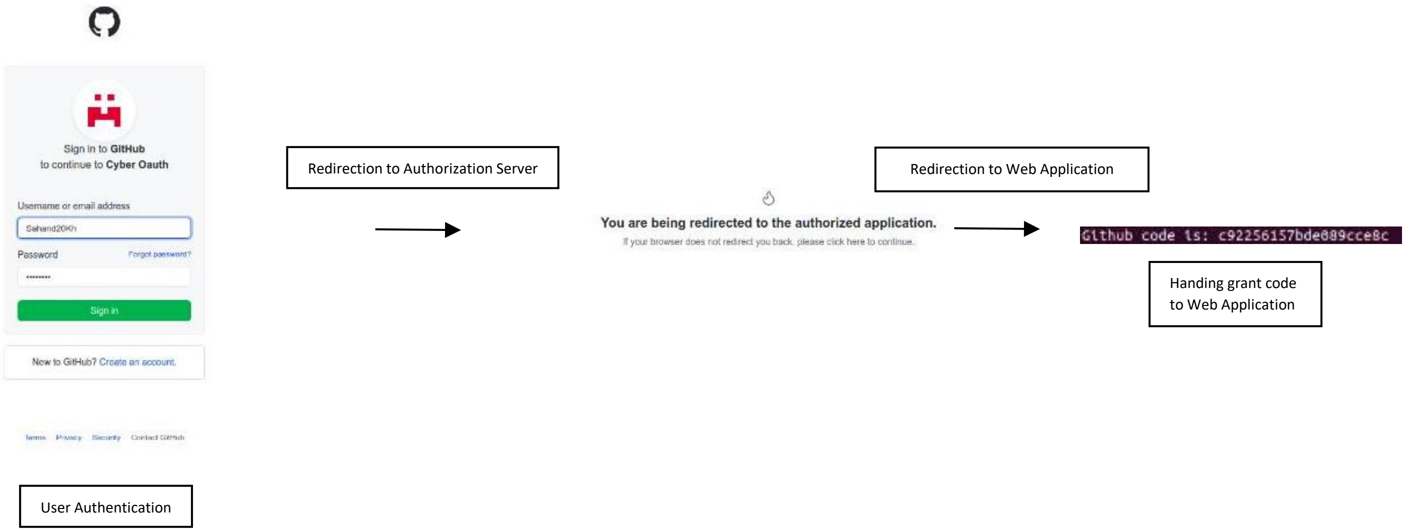
While “server.js” is running, we connect to the localhost page in our browser, and the login page will be displayed. When the user clicks on the link, he/she (local host - in the role of the resource), sends an authorization request to the authorization server on GitHub via the Authorization URI and knows where to come back according to the redirect URI.

Meanwhile the User logs in to GitHub if a previous login (tracked by cookies) hasn't been done or the previous session is terminated.(The actual authorization is done right here by the user indirectly by clicking on the link and logging in to his/her account!)

Step 2. Redirection to Client, Passing Grant Code

After the user is authenticated, it will eventually be redirected back to the web application, where it hands the application the grant code. This is the systematic point where the user has totally handed information access.

The mentioned steps are summarized in the following diagram.



Sign in to **GitHub**
to continue to **Cyber Oauth**

Username or email address: Sahand20Kh

Password: ••••••••

New to GitHub? Create an account.

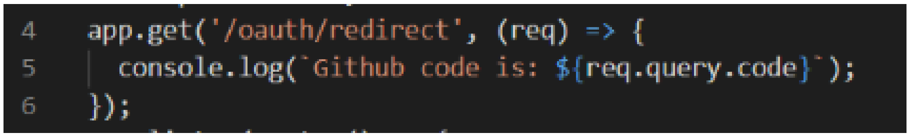
You are being redirected to the authorized application.

If your browser does not redirect you back, please click here to continue.

```
Github code is: c92256157bde089cce8c
```

Forgot password?
Sign in
Terms
Privacy
Security
Contact GitHub
User Authentication
Redirection to Authorization Server
Redirection to Web Application
Handing grant code to Web Application

- In the main code the grant code is obtained by sending an HTTP "GET" request to the user(local host) on the redirection link, where the user appears after succesfully being authenticated.



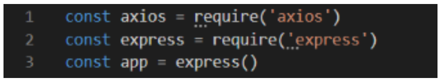
```
app.get('/oauth/redirect', (req) => {
  console.log(`Github code is: ${req.query.code}`);
});
```

Step 3. Client Authentication, Access Token Reception (Bonus Version)

The client(web application) passes the grant code + ClientID and Client Secret (the shared key for authentication) to the Authorization server on GitHub. The grant code is checked to be the same as the one that the Auth Server handed to the user (provides authorization by means of integrity) and the client secret is also checked with the secret key associated to the web application at the Auth Server database(provides client authentication). If everything is OK, the access token is passed back to the web application(client) and it can be used to receive User's data from the API.

I've chosen to send and receive the requests in server.js code automatically (as a bonus part):

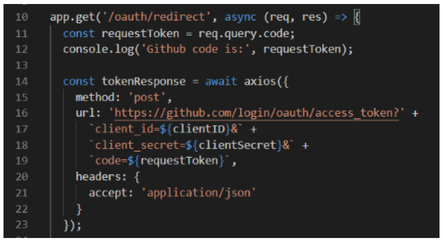
- The first thing to do is to install and load "Axios" package. "Axios" is a promise-based HTTP client for the brower and Node.js.



```
const axios = require('axios')
const express = require('express')
const app = express()
```

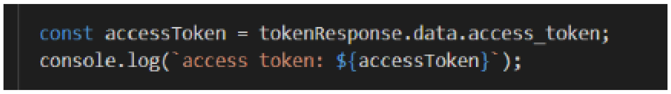
Code Modifications

- We can use Axios module to make HTTP requests. We can either the request as a mehtod or configure requests using a JS object. I've used the second way.
    - This is a simple HTTP POST request that passes the intende information (ID, secret and grant code) needed for an access token to the URL on the Auth server that's responsible for issuing acces tokens:

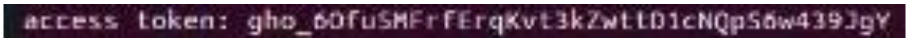


```
app.get('/oauth/redirect', async (req, res) => {
  const requestToken = req.query.code;
  console.log('Github code is:', requestToken);

  const tokenResponse = await axios({
    method: 'post',
    url: 'https://github.com/login/oauth/access_token?' +
      `client_id=${clientID}&` +
      `client_secret=${clientSecret}&` +
      `code=${requestToken}`,
    headers: {
      accept: 'application/json'
    }
  });
```

    - The access token is then stored as an object and printed on the terminal screen. We will use it in the next GET request as the authorization paramter. This request should be sent to the webpage which contains user information.



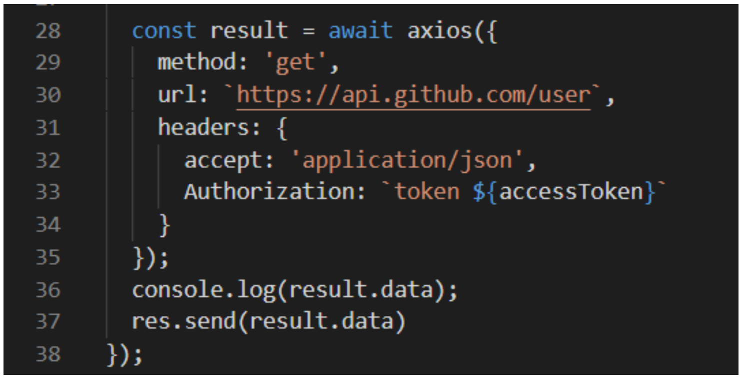
```
const accessToken = tokenResponse.data.access_token;
console.log(`access token: ${accessToken}`);
```



```
access token: gho_6OfuSMFrfErqKvt3kZwtLD1cNQpS6w439JgY
```

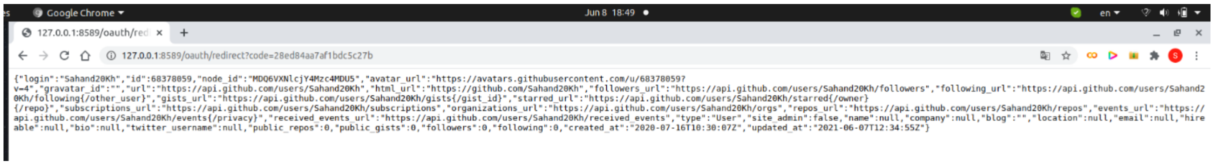
Step 4. Using Access Token to obtain User's Information (Bonus Version)

- We should send a GET request in this part and use the received access token to obtain user's information. The corresponding URL should be where the user's information is stored on GitHub's API. It's important to note that all previous communications were done with the authenticaiton server.

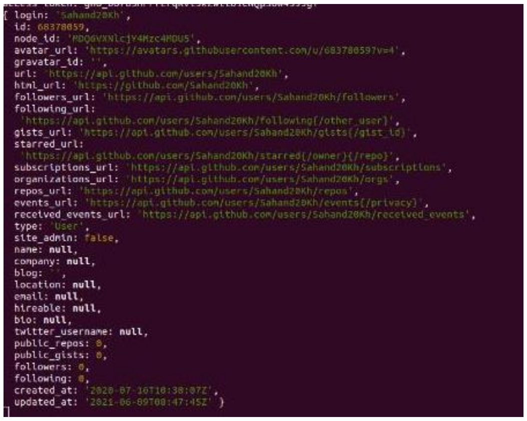


```
  const result = await axios({
    method: 'get',
    url: `https://api.github.com/user`,
    headers: {
      accept: 'application/json',
      Authorization: `token ${accessToken}`
    }
  });
  console.log(result.data);
  res.send(result.data);
});
```

- Finally the results which are the User's details in a JS are printed on the console and also sent to the localhost:



```
{"login":"Sahand20Kh","id":68378059,"node_id":"MDQ6VXNlcjY4Mzc4MDU5","avatar_url":"https://avatars.githubusercontent.com/u/68378059?
v=4","gravatar_id":"","url":"https://api.github.com/users/Sahand20Kh","html_url":"https://github.com/Sahand20Kh","followers_url":"https://api.github.com/users/Sahand20Kh/followers","following_url":"https://api.github.com/users/Sahand2
0Kh/following{/other_user}","gists_url":"https://api.github.com/users/Sahand20Kh/gists{/gist_id}","starred_url":"https://api.github.com/users/Sahand20Kh/starred{/owner}
{/repo}","subscriptions_url":"https://api.github.com/users/Sahand20Kh/subscriptions","organizations_url":"https://api.github.com/users/Sahand20Kh/orgs","repos_url":"https://api.github.com/users/Sahand20Kh/repos","events_url":"https://
api.github.com/users/Sahand20Kh/events{/privacy}","received_events_url":"https://api.github.com/users/Sahand20Kh/received_events","type":"User","site_admin":false,"name":null,"company":null,"blog":"","location":null,"email":null,"hire
able":null,"bio":null,"twitter_username":null,"public_repos":0,"public_gists":0,"followers":0,"following":0,"created_at":"2020-07-16T10:30:07Z","updated_at":"2021-06-07T12:34:55Z"}
```



```
{ login: 'Sahand20Kh',
  id: 68378059,
  node_id: 'MDQ6VXNlcjY4Mzc4MDU5',
  avatar_url: 'https://avatars.githubusercontent.com/u/68378059?v=4',
  gravatar_id: '',
  url: 'https://api.github.com/users/Sahand20Kh',
  html_url: 'https://github.com/Sahand20Kh',
  followers_url: 'https://api.github.com/users/Sahand20Kh/followers',
  following_url:
   'https://api.github.com/users/Sahand20Kh/following{/other_user}',
  gists_url: 'https://api.github.com/users/Sahand20Kh/gists{/gist_id}',
  starred_url:
   'https://api.github.com/users/Sahand20Kh/starred{/owner}{/repo}',
  subscriptions_url: 'https://api.github.com/users/Sahand20Kh/subscriptions',
  organizations_url: 'https://api.github.com/users/Sahand20Kh/orgs',
  repos_url: 'https://api.github.com/users/Sahand20Kh/repos',
  events_url: 'https://api.github.com/users/Sahand20Kh/events{/privacy}',
  received_events_url: 'https://api.github.com/users/Sahand20Kh/received_events',
  type: 'User',
  site_admin: false,
  name: null,
  company: null,
  blog: '',
  location: null,
  email: null,
  hireable: null,
  bio: null,
  twitter_username: null,
  public_repos: 0,
  public_gists: 0,
  followers: 0,
  following: 0,
  created_at: '2020-07-16T10:30:07Z',
  updated_at: '2021-06-09T08:47:45Z' }
```

Theoretical Questions

1. Advantages of “Authorizaiton code” grant type:

- It **provides an additional layer of security** as it uses an extra step (exchanging code for token) before giving data to the web application. In constrast “Implicit” grant doesn’t have this additional layer and thus is less secure.

  This code exchange step prevent’s an attacker from intercepting the access token as the token is passed over a secure channel that’s maintained between the OAuth server and the web application.

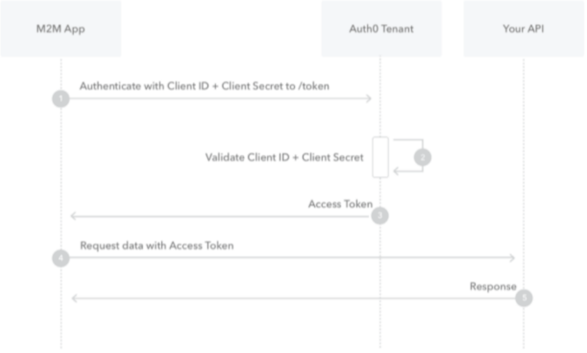
- It **prevents unnecessary user interaction**. The user doesn’t need to do anything after the authorizaiton click.

- It has the option of “**refresh tokens**”. This makes the client website almost unlimited access to user’s data after authorization, without going through the whole procedure again.

2. Security Disadvantages of “Client Credential” grant type:

This type is usually used for access by user him/her self.

This is the main diagram of how this grant type works in mobile application:

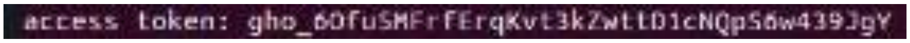

M2M App
Auth0 Tenant
Your API
1
Authenticate with Client ID + Client Secret to /token
Validate Client ID + Client Secret
2
Access Token
3
4
Request data with Access Token
Response
5

The dangers of using this credential type is:

1. The web allpication and the user are indistinguishable from one another and one can impersonate as another in some attack.
2. Client applications can request any scope without the user's knowledge
3. Increasing the potential of phishing the users
4. No federation and multi factor authenticaiton is used.
5. Single sign on isn't used and for each session a separate authentication is required.

3. **No**, this token is a string type token, which differs from JWT (RFC 7519) in several factors. **They are called bearer tokens (RFC 6750).**

   This fact is known by comapring the string we get to the format of a JWT that we will dicuss after this.

   ```
   access token: gho_6OfuSMFrfErqKvt3kZwttD1cNQpS6w439JgY
   ```

   These tokens are actually hust random strings. These access tokens **act as a primary key** in order to obtain the future keys as refresh tokens whenever we need specific pairs associated to scopes. JWT's in contrast give the access to all scope values(pairs) at once.

   The valid characters in this type of access token(bearer token), are alphanumeric and some punctuation charecters.

   Typically, a service will either generate random strings and store them in a database along with the associated user and scope information, or will use Self-encoded tokens where the token string itself contains all the necessary info.

   These tokens are generated by the server in response to a certain login request.

JWT (JSON Web Token)

JWT stands for "JSON Web Token". It's an internet standard for creating data with optional signiture and optional encryption. The **data payload holds a JSON object**. JSON grew out of a need for a state-less browser-to-server communication protocol. And they don't need to be stroed on a database as an advantage.

- Yes. JWT's are **by default decodable** by anyone. (Decode @ https://jwt.io/)

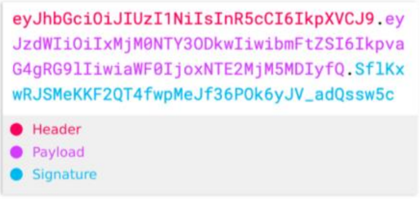
  Usually the used code is base64 and is easilu decoded.

  But JWT can **either be signed or encrypted or both.** Like everwhere else, signing provides integrity, authenticity and non-repudiation. Encryption provides confidentiality

  Signing can be done with either a **secret shared key** (using **HMAC**) or with a **public/private key pair** using **RSA** or **ECDSA** (Elliptic Curve Digital Signature Algorithm)

- JWT's are **very useful in distibuted systems and microservices. They save a lot amount of requests** to **improve** the overall **scalability** of an application**.**

  JWT's consist of mainly 3 parts: 1. Header/ 2. Payload/ 3. Signature (optional)



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

● Header
● Payload
● Signature

The three components of a JSON Web Token

The **header** contains what's known as **metadata information.** It includes:

1. The **Algorithm used to sign** the token
2. The **type** of the token ("JWT" for our case)

The **payload** includes **main information** such as:

1. **Issuer**(iss): The entity that produces the JSON object which is the **Auth Server** in our case

2. **Subject**(sub): The entity that is identified by this token. The **userID** for example for authorizing a user.

3. **Audience**(aud): The entitiy that is intended to use the token (here the **web application**)

4. **Expiration**(exp): Acts as a time stamp that specifies the date that the token is not valid after it anymore.

5. **Issue Data**(iat): Specifies the date that the token has been issued.

The **signature** is created according to the encoded header, encoded payload, a shared secret or private key and of course, the algorithm wich produces it.

4. Security Problems in our OAuth process

1. The **main problem** with our program specifically is that everything **is based on sending HTTP messages which are not secure**, and impersonation can be done by **MITM attacks**. Redirect URL can easily be changed. ClientID and Client secret are passed over a non-secure channel and can easily be read.

2. Another problem we can address is that the **authorization URL** is not encrypted and an attacker can easily swap it with a **poisoned URL,** belonging to himself that redirects the user to a domain and the user passes the attacker the code instead of passing ot to the web application

3. **Valid tokens** for mutiple users **can also be swapped** by means of an attacker that is another vulnerability of our program. The **user should somehow carry some authenticated information** that provides **integrity** (something like MAC) to solve this issue. (From another perspective **authentication is done before accesing the token** but **is forgotten to be embedded in the token itself**)