

# Exercise 10

*Understanding OAuth2*

*Using OAuth2 introspection to validate tokens*

*Creating OAuth2 tokens using Device Grant flow*

*Using OAuth2 as a login mechanism*

## Prior Knowledge

Previous exercises

## Objectives

Add a login and token flow to authenticate a client for our purchase service

## Software Requirements

- Keycloak
- Purchase service from previous exercise
- Python

## Overview

*The lab will follow the following overall approach:*

1. *Install and configure Keycloak to act as an OAuth2 IdP*
2. *Configure a realm and application in Keycloak*
3. *Create a sample user*
4. *Update our purchase service to use OAuth2 introspection to validate tokens*
5. *Create a CLI that gets a token via the OAuth2 device grant flow*
6. *Create a simple python client that uses the access token to access the API.*
7. *Update keycloak to support Github as a login provider*

## Part A - setting up OAuth2 with a CLI

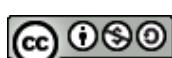
### Steps

1. We are going to configure keycloak using the admin UI. It would then be possible to export this as a configuration file and dockerise this but that is an extension. So the first step is to download and install the keycloak.

Because the Device Grant OAuth2 flow we are using is fairly new, we are going to use a snapshot build of keycloak master. Do not download the 12.04 build as it won't work!

Download the keycloak distribution (or you may find it in Downloads):

```
cd ~/Downloads
```



```
wget http://freo.me/keycloak-zip  
cd ~  
unzip ~/Downloads/keycloak-zip
```

We are actually using the “keycloak.x” build, which is the next generation of Keycloak based on a technology called Quarkus, which compiles the Java into native code and makes it much more suitable for docker/containerisation.

2. Start the keycloak server:

```
cd keycloak.x-13.0.0-SNAPSHOT  
bin/kc.sh start-dev
```

You should see:

```
oxsoa@oxsoa:~/keycloak.x-13.0.0-SNAPSHOT$ bin/kc.sh start-dev  
2021-04-05 07:43:27,756 INFO [org.key.url.DefaultHostnameProviderFactory] (main) Frontend: <request>, Admin: <frontend>, Backend: <request>  
2021-04-05 07:43:27,837 INFO [org.key.pro.qua.QuarkusCacheManagerProvider] (main) Loading cluster configuration from /home/oxsoa/keycloak.x-13.0.0-SNAPSHOT/bin/.../conf/cluster-local.xml  
2021-04-05 07:43:28,010 INFO [org.inf.CONTAINER] (main) ISPN000128: Infinispan version: Infinispan 'Corona Extra' 11.0.4.Final  
2021-04-05 07:43:28,259 INFO [org.key.con.inf.DefaultInfinispanConnectionProviderFactory] (main) Node name: node_145904, Site name: null  
2021-04-05 07:43:29,069 INFO [org.key.con.liq.QuarkusJpaUpdaterProvider] (main) Initializing database schema. Using change log META-INF/jpa-changelog-master.xml  
2021-04-05 07:43:30,213 INFO [org.key.services] (main) KC-SERVICES0050: Initializing master realm  
2021-04-05 07:43:30,672 INFO [org.key.con.jpa.QuarkusJpaConnectionProviderFactory] (main) Database info: {databaseUrl=jdbc:h2:file:/home/oxsoa/keycloak.x-13.0.0-SNAPSHOT/bin/.../data/keycloakdb, databaseUser=SA, databaseProduct=H2 1.4.197 (2018-03-18), databaseDriver=H2 JDBC Driver 1.4.197 (2018-03-18)}  
2021-04-05 07:43:30,769 INFO [to.quarkus] (main) Keycloak 13.0.0-SNAPSHOT on JVM (powered by Quarkus 1.12.2.Final) started in 5.127s. Listening on: http://0.0.0.0:8080  
2021-04-05 07:43:30,770 INFO [to.quarkus] (main) Profile dev activated.  
2021-04-05 07:43:30,778 INFO [to.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, jdbc-h2, jdbc-mariadb, jdbc-mysql, jdbc-postgresql, keycloak, mutiny, narayana-jta, resteasy, resteasy-jackson, smallrye-context-propagation, smallrye-health, smallrye-metrics, vertx, vertx-web]  
$
```

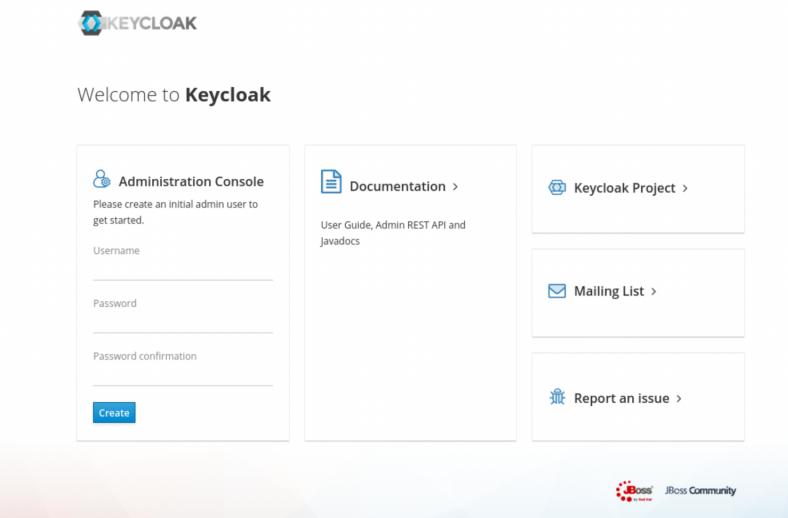
If there is a problem with JDK not found do:

```
sudo apt install openjdk-11-jdk-headless
```

and then try again.

3. Open a browser to <http://localhost:8080/>

You will see:



4. Create a new admin user in the box:

This is a zoomed-in view of the "Administration Console" section from the previous screenshot. It shows a form for creating a new user. The "Username" field contains "admin", the "Password" field contains a redacted password, and the "Password confirmation" field also contains a redacted password. A blue "Create" button is at the bottom.

5. Click on **Administration Console** to Login

This is a screenshot of the Keycloak login page. The title "Sign in to your account" is at the top. Below it are fields for "Username or email" containing "admin" and "Password" containing a redacted password. A large blue "Sign In" button is at the bottom.

6. You will see something like this:

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu includes 'Master' (selected), 'Configure' (selected), 'Realm Settings' (selected), 'Clients', 'Client Scopes', 'Roles', 'Identity', 'Providers', 'User Federation', and 'Authentication'. Under 'Manage', there are links for 'Groups', 'Users', 'Sessions', 'Events', 'Import', and 'Export'. The main content area is titled 'Master' and shows the 'General' tab selected. It contains fields for 'Name' (master), 'Display name' (Keycloak), 'HTML Display name' (<div class="kc-logo-text"><span>Keycloak</span></div>), 'Frontend URL' (empty), 'Hostname' (empty), 'Enabled' (ON), 'User-Managed Access' (OFF), and 'Endpoints' (OpenID Endpoint Configuration, SAML 2.0 Identity Provider Metadata). At the bottom are 'Save' and 'Cancel' buttons.

7. We are going to create a new “realm” in which to do authentication. Hover the mouse over the word Master and then click **Add Realm**

The screenshot shows a dropdown menu with 'Master' selected. Below it is a blue button labeled 'Add realm'.

8. Call the realm “purchase” and click **Create**

The screenshot shows the 'Add realm' form. It has an 'Import' section with a 'Select file' button. The 'Name' field is filled with 'purchase'. The 'Enabled' switch is set to 'ON'. At the bottom are 'Create' and 'Cancel' buttons.



9. There are **lots** of settings we can change. Today we are going to focus just on creating an OAuth2 configuration.

10. First click on **Clients**

Clients			<a href="#">Create</a>
<input type="text" value="Search..."/> <a href="#">Lookup</a>			
Client ID	Enabled	Base URL	Actions
account	True	<a href="http://localhost:8080/realm/purchase/account/">http://localhost:8080/realm/purchase/account/</a>	<a href="#">Edit</a> <a href="#">Export</a> <a href="#">Delete</a>
account-console	True	<a href="http://localhost:8080/realm/purchase/account/">http://localhost:8080/realm/purchase/account/</a>	<a href="#">Edit</a> <a href="#">Export</a> <a href="#">Delete</a>
admin-cli	True	Not defined	<a href="#">Edit</a> <a href="#">Export</a> <a href="#">Delete</a>
broker	True	Not defined	<a href="#">Edit</a> <a href="#">Export</a> <a href="#">Delete</a>
realm-management	True	Not defined	<a href="#">Edit</a> <a href="#">Export</a> <a href="#">Delete</a>
security-admin-console	True	<a href="http://localhost:8080/admin/purchase/console/">http://localhost:8080/admin/purchase/console/</a>	<a href="#">Edit</a> <a href="#">Export</a> <a href="#">Delete</a>

11. Click **Create**

### Add Client

<a href="#">Import</a>	<a href="#">Select file</a>
<b>Client ID *</b>	<input type="text" value="purchase"/>
<b>Client Protocol</b>	<input type="text" value="openid-connect"/>
<b>Root URL</b>	<input type="text"/>
<a href="#">Save</a> <a href="#">Cancel</a>	

12. Call the client “purchase” and then save.

### 13. You should see:

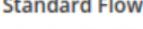
Purchase 

[Settings](#) Roles Client Scopes  Mappers  Scope  Revocation Sessions  Offline Access  Installation 

Client ID 	purchase
Name 	
Description 	
Enabled 	ON
Always Display in Console 	OFF
Consent Required 	OFF
Login Theme 	
Client Protocol 	openid-connect
Access Type 	public
Standard Flow Enabled 	ON
Implicit Flow Enabled 	OFF
Direct Access Grants Enabled 	ON
Root URL 	
* Valid Redirect URIs 	<input type="text"/> 
Base URL 	
Admin URL 	
Web Origins 	<input type="text"/> 
Backchannel Logout URL 	

### 14. Change access type to confidential

Access Type 

Standard Flow Enabled 

public
confidential
public
bearer-only



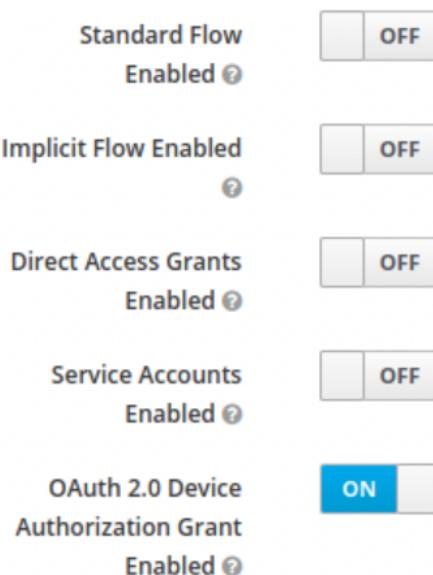
15. You will see a new option appear - Device Authorization Grant.

Turn **off** the Standard Flow Enabled (this is the Authorization Code Flow that we would use if we had a website accessing the API)

Turn **off** Direct Access Grants (this is the Resource Owner Password Credentials Grant)

Turn **on** the Device authorization grant.

Your panel should look like this:



## 16. Save

17. Go to the credentials tab and you will see the Client Secret for this OAuth2 application. Take a note of this secret. We will need it.

The screenshot shows the 'Credentials' tab selected in a navigation bar. Below it, there are two main sections:

- Client Authenticator**: A dropdown menu currently set to 'Client Id and Secret'. To its right is a text input field containing the value '348b312e-1a45-4758-97a9-18f83fbaf'. Next to the input field is a blue 'Regenerate Secret' button.
- Registration access token**: A text input field with a placeholder '(empty)' and a blue 'Regenerate registration access token' button.

If for example you have been silly enough to copy and paste that secret into a document, you can regenerate it!

## 18. Go back to **Realm Settings**

## 19. Click on **OpenId Endpoint Configuration**

The screenshot shows a web interface for managing OpenID endpoints. At the top left is a 'Endpoints' tab with a question mark icon. Next to it is a blue-bordered box containing the text 'OpenID Endpoint Configuration'. Below this is another box labeled 'SAML 2.0 Identity Provider Metadata'. At the bottom of the interface are two buttons: 'Save' and 'Cancel'.

You will see a JSON page with a lot of URLs. I'm assuming you have configured your realm name and client name the same as me and therefore the code examples I have given you will work.

```
issuer: "http://localhost:8080/realm/purchase"
▼ authorization_endpoint: "http://localhost:8080/realm/purchase/protocol/openid-connect/auth"
▼ token_endpoint: "http://localhost:8080/realm/purchase/protocol/openid-connect/token"
▼ introspection_endpoint: "http://localhost:8080/realm/purchase/protocol/openid-connect/token/introspect"
▼ userinfo_endpoint: "http://localhost:8080/realm/purchase/protocol/openid-connect/userinfo"
▼ end_session_endpoint: "http://localhost:8080/realm/purchase/protocol/openid-connect/logout"
▼ jwks_uri: "http://localhost:8080/realm/purchase/protocol/openid-connect/certs"
▼ check_session_iframe: "http://localhost:8080/realm/purchase/protocol/openid-connect/login-status-iframe.html"
- grant_type_supported:
```

If you look right at the bottom you will see the Device URL

```
11: "RS512"
backchannel_logout_supported: true
backchannel_logout_session_supported: true
▼ device_authorization_endpoint: "http://localhost:8080/realm/purchase/protocol/openid-connect/auth/device"
```

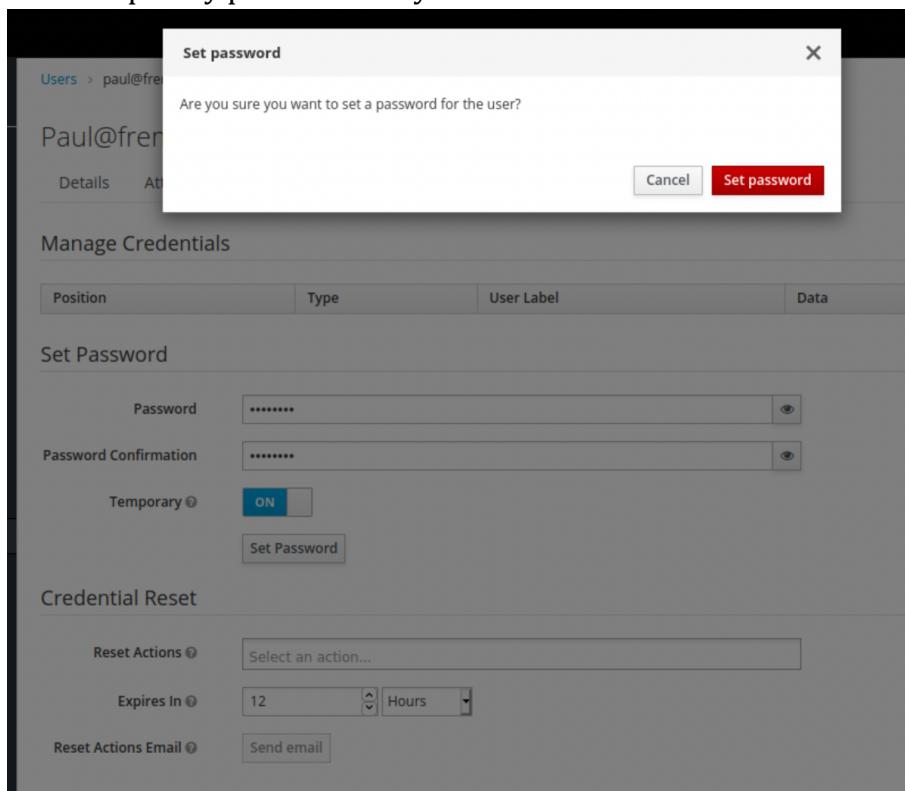
## 20. Close this tab

## 21. We also will create a user: **Users → Add User**

The screenshot shows a 'Add user' form. It includes fields for 'ID' (empty), 'Created At' (empty), 'Username' (paul@fremantle.org), 'Email' (paul@fremantle.org), 'First Name' (Paul), 'Last Name' (Fremantle), 'User Enabled' (ON), 'Email Verified' (OFF), 'Groups' (Select existing group... - No group selected), 'Required User Actions' (empty), and 'Save' and 'Cancel' buttons.

Change to the **Credentials** tab.

22. Set a temporary password for your user:



23. One of the clients of the realm is a “account management” app. Go back to the list of clients:

Clients					
<a href="#">Lookup</a>					
<a href="#">Create</a>					
Client ID	Enabled	Base URL	Actions		
account	True	<a href="http://localhost:8080/realm/purchase/account/">http://localhost:8080/realm/purchase/account/</a>	Edit	Export	Delete
account-console	True	<a href="http://localhost:8080/realm/purchase/account/">http://localhost:8080/realm/purchase/account/</a>	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
purchase	True	Not defined	Edit	Export	Delete
realm-management	True	Not defined	Edit	Export	Delete
security-admin-console	True	<a href="http://localhost:8080/admin/purchase/console/">http://localhost:8080/admin/purchase/console/</a>	Edit	Export	Delete

24. Click on the URL next to “account-console”.



25. You will see a new tab.

The screenshot shows the Keycloak Account Management interface. At the top right is a blue "Sign In" button. Below it, the text "Welcome to Keycloak Account Management" is displayed. There are three main sections: "Personal Info" (Manage your basic information), "Account Security" (Control your password and account access), and "Applications" (Track and manage your app permission to access your account). Each section has a "Sign In" or "Device Activity" link at the bottom.

Click **Sign In**

26. Use the email and temporary password you entered, and then create a new secure password.

You can use this app to manage your credentials, application logins etc

The screenshot shows the "Personal Info" management screen. On the left is a sidebar with "Personal Info" selected, and other options like "Account Security", "Signing In", "Device Activity", and "Applications". The main area is titled "Personal Info" with the sub-instruction "Manage this basic information: your first name, last name and email". It contains four input fields: "Username" (paul@fremantle.org), "Email" (paul@fremantle.org), "First name" (Paul), and "Last name" (Fremantle). At the bottom are "Save" and "Cancel" buttons.

27. We now have an application and a user.

28. However, we need a token!

29. Before we do that, let's modify the Purchase app to require a token.

I'm assuming you have a working copy of the app in either the purchase-starter or purchase-complete directories.

`cd ~/purchase-starter (or purchase-complete)`



30. I've written a small piece of "express middleware" that will validate the token. There is another similar package in the npm repo but since I couldn't find the source code, I wasn't sure how it worked. This is not a production-ready package!

```
yarn add @pzfro/express-introspect
```

You can find the source code here:  
<https://github.com/pzfro/express-introspect>

The main code is very simple:

```
const auth = req.headers.authorization;
if (!auth) return res.status(401).send({ error: "bearer token not found"});

const bearer = auth.trim().substr(6).trim() // remove "bearer"

const data = {
  token : bearer,
  token_type_hint : "access_token"
};
const encoded = querystring.stringify(data);

const response = await axios.post(url, encoded, { auth: {username: username, password: password}});

if (!response.data.active) {
  return res.status(401).send({ error: "unauthorized token"});
}
if (response.data.username) {
  req.username = response.data.username;
}
next();
```

What it does is to extract the bearer token from the Authorization header, and then call the introspection endpoint. If the response is

```
{ active: "true", ... }
```

then it allows the flow, otherwise it sends back unauthorized.



31. Add the following code to **src/app.ts**

(you can find this snippet here: <http://freo.me/oauth-snippet>)  
This needs to go somewhere ahead of RegisterRoutes()

```
const client_id = 'purchase';
const client_secret = process.env.CLIENT_SECRET;
const oauth2_host = process.env.OAUTH2_HOST || "localhost";
const oauth2_port = process.env.OAUTH2_PORT || 8080;

const introspect_url =
`http://${oauth2_host}:${oauth2_port}/realms/purchase/protocol/openid-connect/token/introspect`;

app.use(introspect(introspect_url,client_id,client_secret));
```

You will also need the right import:

```
import { introspect } from '@pfreo/express-introspect';
```

32. This looks up the client secret, host and port for the introspection service and then adds the middleware into the flow. This means all the resources in this application will now require a token.

33. Let's try this outside of docker-compose. Start the postgres server standalone:

```
./start-postgres.sh
```

34. In another window also in the same directory

```
export CLIENT_SECRET=<insert the client secret for the purchase app here>
yarn start
```

35. Now we can test this out:

Try using Postman to call GET <http://localhost:8000> with no authorization. You should see the following:

The screenshot shows a Postman interface with the 'Body' tab selected. The response status is 401 Unauthorized. The JSON body contains the following message: "1 { 2 \"error\": \"bearer token not found\" 3 }



36. Now add a bearer token:

The screenshot shows the Postman interface with the 'Auth' tab selected. A dropdown menu labeled 'Bearer Token' is open. Below it, a note says 'The authorization header will be automatically generated when you send the request.' A 'Token' field contains the value 'thistokenisinvalid'.

37. You should see a different error:

The screenshot shows the Postman interface with the 'Body' tab selected. The response body is displayed in JSON format, showing an error message: '{"error": "unauthorized token"}'. The status bar at the top indicates a 401 Unauthorized status with a 29 ms response time and 252 B size.

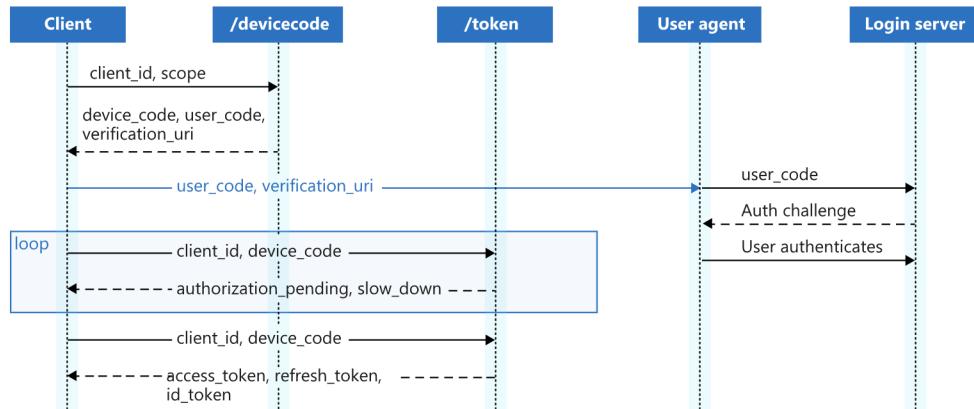


38. We now are going to use the OAuth2.0 Device grant flow to get a token.

Here is a nice picture

(from

<https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-device-code>)



39. I have written a python client that does this:

In (yet another) new terminal window type:

```
git clone https://github.com/pzfreo/python-purchase-client.git
cd python-purchase-client
```

40. Unfortunately we can't use the MITMDUMP proxy in reverse mode for this flow. If you want to use a proxy you will have to set it up in proper "proxy" mode (as in Exercise 2). To make life easier, I've put lots of logging into the python login client to show how it works.

#### 41. Let's look at the code (in sections)

```
13 base_url = 'http://'+oauth_host+':'+oauth_port
14 device_url = '/realms/purchase/protocol/openid-connect/auth/device'
15 token_url = '/realms/purchase/protocol/openid-connect/token'
16
17
18 # call the device code api
19 print("calling device api")
20
21 r = requests.post(base_url+device_url, data = {'client_id':client_id}, auth=(client_id, client_secret))
22
23 resp = r.json()
24 print("received response")
25 print (resp)
```

Firstly on line 21 it sends a post request with the right authentication and with a single pair in the body:

`client_id: purchase`

This returns a JSON like this:

```
[{"device_code":"uWB28Y2ko4CU-6bj8FU3_kK2FERcY2PMXssj8uKRlyk",
 "user_code":"TICG-YPPJ",
 "verification_uri":"http://localhost:8080/realms/purchase/device",
 "verification_uri_complete":"http://localhost:8080/realms/purchase/device?user_code=TICG-YPPJ",
 "expires_in":600,
 "interval":5
}]
```

Now the code opens the browser to the URL provided

```
27 # Open the returned URL
28 print ("opening url", resp['verification_uri_complete'])
29 webbrowser.open_new(resp['verification_uri_complete']);
```

Then it polls for the access tokens, sending back the device\_code that it received:

```
reqdata = {
    'grant_type': "urn:ietf:params:oauth:grant-type:device_code",
    'device_code': resp['device_code'],
    'client_id': client_id
}

print ("polling for tokens while user does browser flow")
# Poll for the tokens
while not finished:
    r = requests.post(base_url+token_url, data = reqdata, auth=(client_id, client_secret))
    print("recieved status code", r.status_code)
    if (r.status_code==200):
        tokendata = r.json()
        access_token = tokendata['access_token']
        refresh_token = tokendata['refresh_token']
        finished = True
        break
    print ("waiting")
    time.sleep(resp['interval'])
```

Finally, once it gets the response, it saves it in YAML format to a file in the home directory:

```
57 config = {  
58     'access_token': access_token,  
59     'refresh_token': refresh_token  
60 }  
61  
62 print ("saving to ~/.config/purchase/secrets")  
63 print (yaml.dump(config))  
64 # save the tokens to the home directory  
65  
66 home = str(Path.home())  
67  
68 Path(home + "/.config/purchase").mkdir(parents=True, exist_ok=True)  
69 with open(home + '/.config/purchase/secrets', 'w') as file:  
70     documents = yaml.dump(config, file)
```

## 42. Let's try it out!

First export the right CLIENT\_SECRET

```
export CLIENT_SECRET=<your client secret here>
```

## 43. In the same terminal type

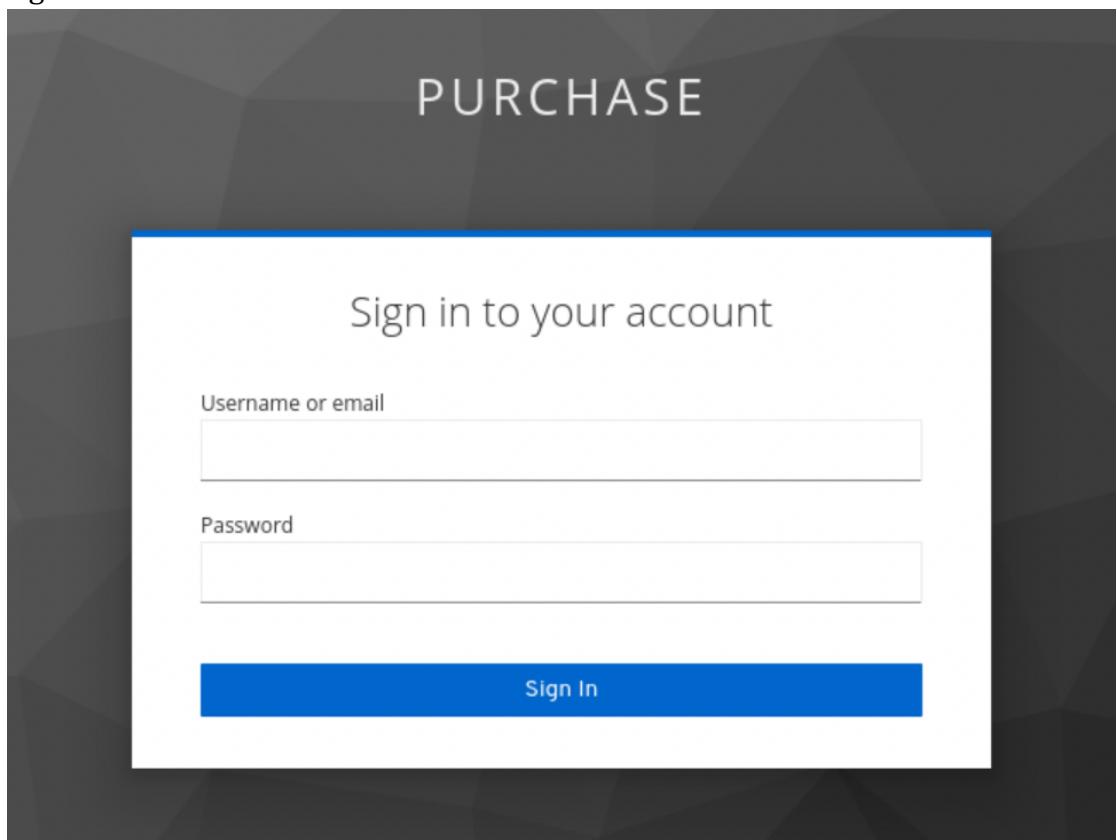
```
python purchase-login.py
```

Two things should happen. In the terminal you should see something like:

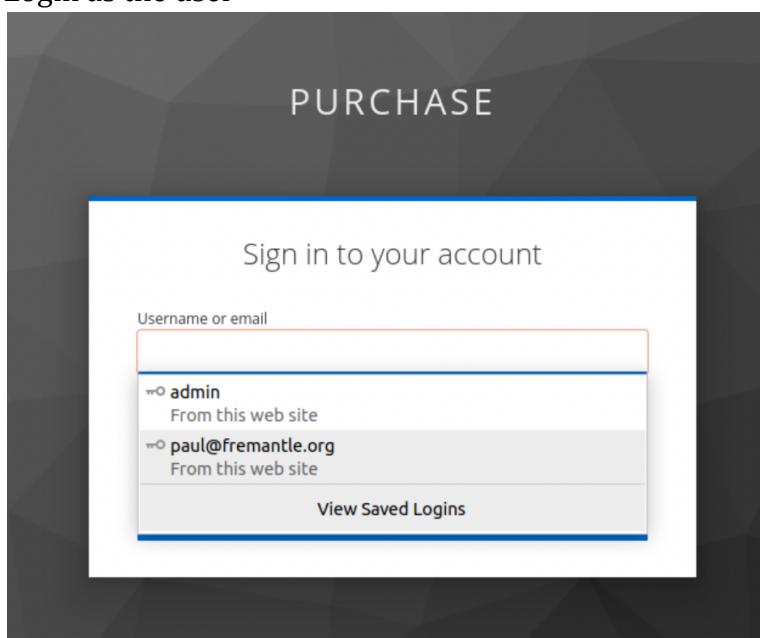
```
calling device api  
received response  
{'device_code': 'nGBMf2YEcHKytJzdXIvztJmAU8ewQ8JmQ8jPhLhCBkY',  
'user_code': 'QANG-BDOC', 'verification_uri':  
'http://localhost:8080/realm/purchase/device',  
'verification_uri_complete':  
'http://localhost:8080/realm/purchase/device?user_code=QANG-BDOC',  
'expires_in': 600, 'interval': 5}  
opening url  
http://localhost:8080/realm/purchase/device?user_code=QANG-BDOC  
polling for tokens while user does browser flow  
recieved status code 400  
waiting
```



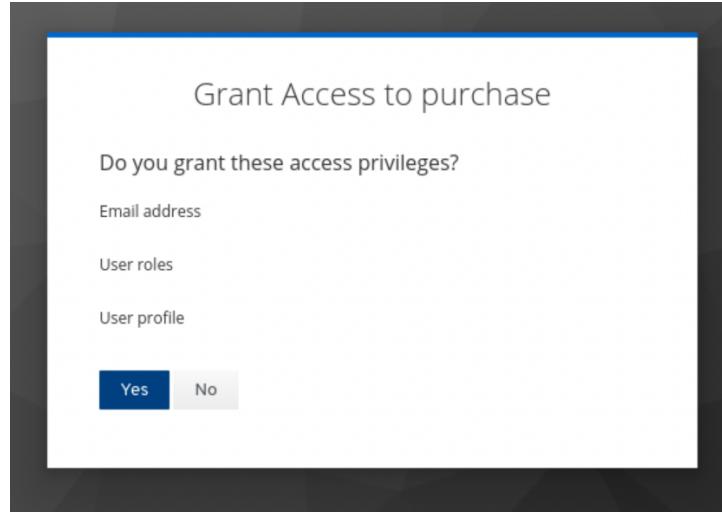
At the same time a browser window should pop open and ask you to login:



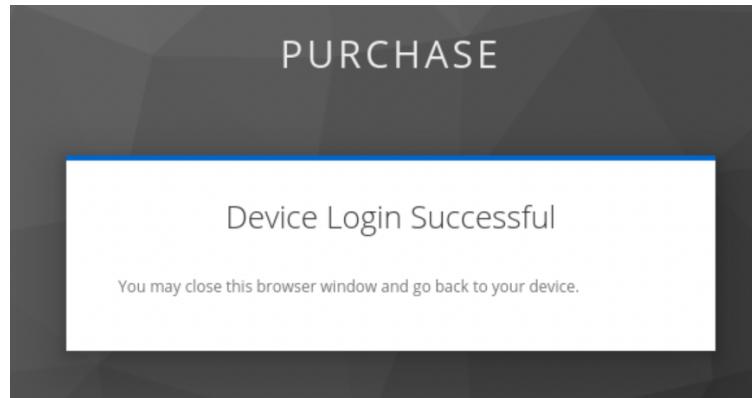
Login as the user



You now need to authorise the system to give access on your behalf



You should now see:



Meanwhile the terminal window will have been retrying until you complete the flow:

```
waiting
recieved status code 400
waiting
recieved status code 400
waiting
recieved status code 200
saving to ~/.config/purchase/secrets
access_token: eyJhbGciOiJSUzIiNlIsInRscCiGoIaIsldUiwiwa2lkIiA6ICl5cVdDYVctQ3hzU0JiTjJ5RndnQ0dUu2ozM05PaWNuYmFEdV8zd2Fcallow
In0.eyJleHAiOjE2MTc2MTUzOTAsImLhdCI6MTYXNzYXMuUSMCwianRpIjoiM2ZlMTNlYjMtyZdjmC00YHYYXlwE2MDytZDmWyzZlNDQ4NTMSIiwiiaXnZijolaHR0CovL2
xvY2FsaG9zdD04HdgwJ3lYwxct9wdXJjaGFZSIsImf1ZC16Imh0dHA6Ly9sb2Nhbgvc3Q60DA4MC9yZWFSbwXHvcHvyZhcc2UlLCjzdWi0t15ZTM4Zjks
Ny02ZmYxLTQ1NmUtoTYMs1mM2ze10Fmje3ZDMlLCjoeXaiolJSzNzyZXNo1wlXypwjo1chVyyzhcc2UlCjzXNzaW9uXN0YXrljotME1mtvhYzctNj
I3ZC00MTazLtg3NzctZDI2NDI3Y2IiMTI3IiwiC2NvcGUi0iJlbWFpbCBwcm9maXlno8.2hnPqrwz1zh0p1klyjWAL-V2huQkgnsowst6-EcN8
rMj0ag2w7MXNrnxEj0g9j0J12Gmbaf7RktGg
refresh_token: eyJhbGciOiJIUzI1NlIsInRscCiGoIaIsldUiwiwa2lkIiA6ICl5cVdYtC4NmRmZi03ZTQ4LTrkYzUtYtg4Nl0x0TAzNDA4MTczNWQifQ.eyJl
eHAtOjE2MTc2MTUzOTAsImLhdCI6MTYXNzYXMuUSMCwianRpIjoiM2ZlMTNlYjMtyZdjmC00YHYYXlwE2MDytZDmWyzZlNDQ4NTMSIiwiiaXnZijolaHR0CovL2
xvY2FsaG9zdD04HdgwJ3lYwxct9wdXJjaGFZSIsImf1ZC16Imh0dHA6Ly9sb2Nhbgvc3Q60DA4MC9yZWFSbwXHvcHvyZhcc2UlLCjzdWi0t15ZTM4Zjks
Ny02ZmYxLTQ1NmUtoTYMs1mM2ze10Fmje3ZDMlLCjoeXaiolJSzNzyZXNo1wlXypwjo1chVyyzhcc2UlCjzXNzaW9uXN0YXrljotME1mtvhYzctNj
I3ZC00MTazLtg3NzctZDI2NDI3Y2IiMTI3IiwiC2NvcGUi0iJlbWFpbCBwcm9maXlno8.2hnPqrwz1zh0p1klyjWAL-V2huQkgnsowst6-EcN8
```

#### 44. Take a look at `~/.config/purchase/secrets`

45. Now let's try the client

```
python purchase-create.py
```

You should see something like:

```
{"date": "2021-04-05T09:22:57.356Z", "isDeleted": false,  
,"poNumber": "PON0001", "lineItem": "LI0001", "quantity"  
:3, "customerNumber": "CUS0001", "paymentReference": "PR  
0001", "id": "45f3c5db-5ef7-476b-803d-61e3ec4a5cc4"}
```

To show that the client has successfully called the API.

46. Edit the `~/.config/purchase/secrets` so the access token is no longer good

Now try again

```
python purchase-create.py  
{"error": "unauthorized token"}
```

47. Also if you wait too long the token will expire and you will also see that.  
We will look at that in an extension.

48. If you have had enough, you can stop here! Well done!

There are also some extensions at the end that apply to Part A.



## Part B - SSO with Github

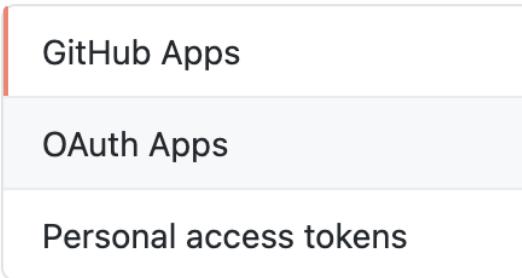
49. However, if you want to do one more thing, we can configure the keycloak server to allow SSO using Github. This will mean we don't need to create ids but can defer to Github for them.

50. You need a Github account to make this work.

51. Log into Github and go to **Settings → Developer Settings**

Developer settings

52. Go to OAuth Apps



53. Click **New OAuth App**

Fill in the screen like this:

### Register a new OAuth application

**Application name \***

PurchaseLoginApp

Something users will recognize and trust.

**Homepage URL \***

http://localhost:8000/purchase

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://localhost:8080/realm/purchase/broker/github/endpoint

Your application's callback URL. Read our [OAuth documentation](#) for more information.

**Register application**

**Cancel**

You should see something like:

## PurchaseLoginApp

 [pzfreo](#) owns this application.

[Transfer ownership](#)

You can list your application in the [GitHub Marketplace](#) so that other users can discover it.

[List this application in the Marketplace](#)

**0 users**

[Revoke all user tokens](#)

**Client ID**  
`ecce7e967bf23db96be7`

**Client secrets**

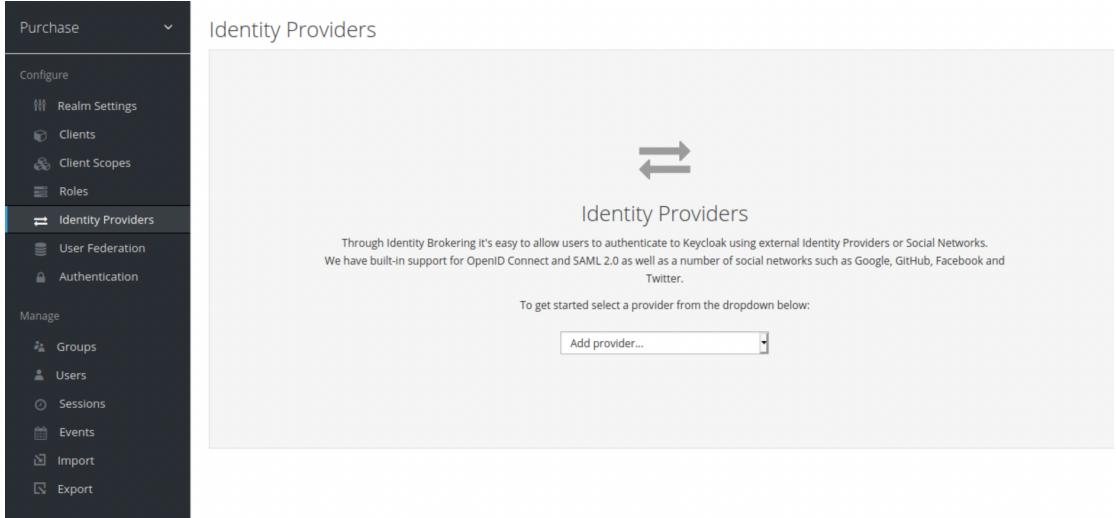
[Generate a new client secret](#)

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

**Application logo**

 [Upload new logo](#)  
You can also drag and drop a picture from your computer.

### 54. Now go to the Keycloak admin console. Click on **Identity Providers**



The screenshot shows the Keycloak admin console interface. On the left, there is a dark sidebar menu with the following items:

- Purchase
- Configure
  - Realm Settings
  - Clients
  - Client Scopes
  - Roles
- Identity Providers
- User Federation
- Authentication

Manage

- Groups
- Users
- Sessions
- Events
- Import
- Export

The main content area is titled "Identity Providers". It features a large double-headed arrow icon and the heading "Identity Providers". Below this, a message states: "Through identity Brokering it's easy to allow users to authenticate to Keycloak using external Identity Providers or Social Networks. We have built-in support for OpenID Connect and SAML 2.0 as well as a number of social networks such as Google, GitHub, Facebook and Twitter." A "Add provider..." dropdown menu is visible at the bottom.

### 55. Choose Github

### 56. Copy the client id from the Github App page.

57. On the Github page create a new client secret and copy that into Keycloak as well.

The screenshot shows the GitHub application settings for a repository named 'pzfreo'. It includes sections for ownership (pzfreo owns it), listing in the Marketplace, user count (0 users), Client ID (ecce7e967bf23db96be7), and Client secrets. A message encourages copying the new client secret. A single client secret is listed, which cannot be deleted.

**Transfer ownership**

You can list your application in the [GitHub Marketplace](#) so that other users can discover it. [List this application in the Marketplace](#)

**0 users** [Revoke all user tokens](#)

**Client ID**  
ecce7e967bf23db96be7

**Client secrets** [Generate a new client secret](#)

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

**Client secret**  
Added now by pzfreo  
Never used  
You cannot delete the only client secret. Generate a new client secret first. [Delete](#)

58. Your keycloak should look like:

The screenshot shows the 'Add identity provider' configuration for a GitHub provider. The form includes fields for Redirect URI (http://localhost:8080/realm/purchase/broker/github/endpoint), Client ID (ecce7e967bf23db96be7), Client Secret (redacted), Default Scopes (empty), Store Tokens (OFF), Stored Tokens Readable (OFF), Enabled (ON), Accepts prompt=none forward from client (OFF), Disable User Info (OFF), Trust Email (OFF), Account Linking Only (OFF), Hide on Login Page (OFF), GUI order (empty), First Login Flow (first broker login), Post Login Flow (empty), Sync Mode (import), and Save/Cancel buttons.

Identity Providers > Add identity provider

Add identity provider

Redirect URI: http://localhost:8080/realm/purchase/broker/github/endpoint

\* Client ID: ecce7e967bf23db96be7

\* Client Secret:  [Copy](#)

Default Scopes:

Store Tokens:  OFF

Stored Tokens Readable:  OFF

Enabled:  ON

Accepts prompt=none forward from client:  OFF

Disable User Info:  OFF

Trust Email:  OFF

Account Linking Only:  OFF

Hide on Login Page:  OFF

GUI order:

First Login Flow: first broker login

Post Login Flow:

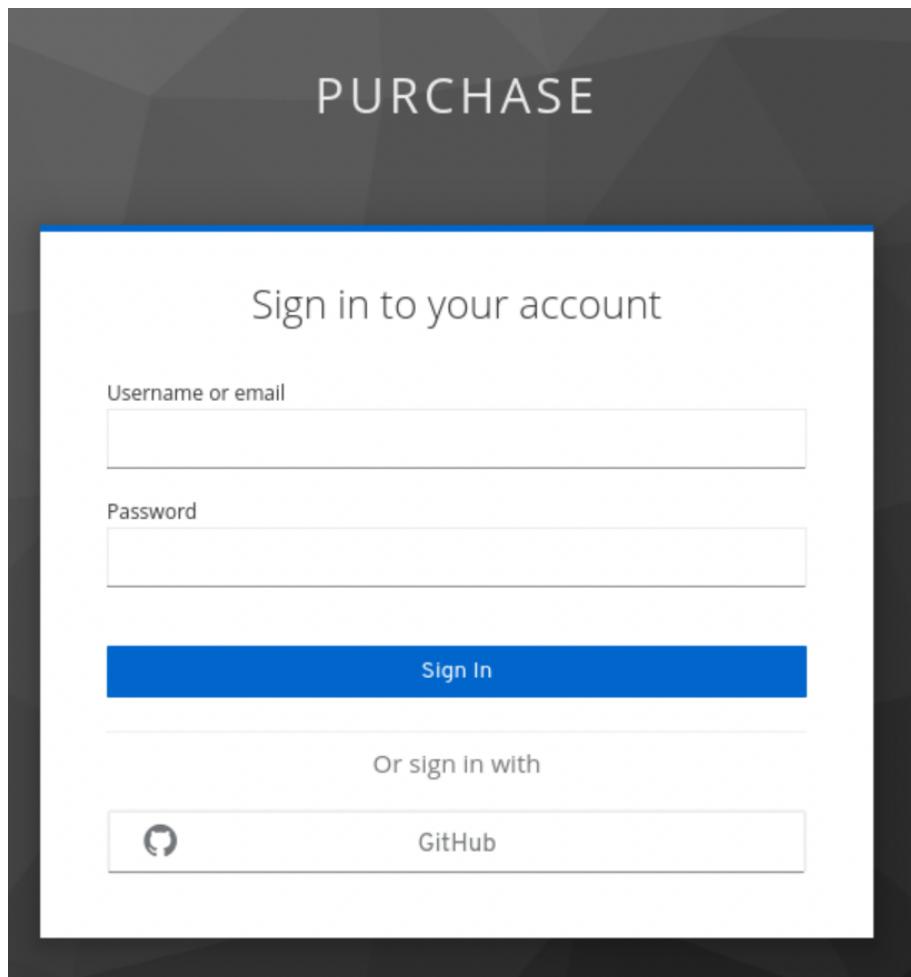
Sync Mode: import

[Save](#) [Cancel](#)

**Save**

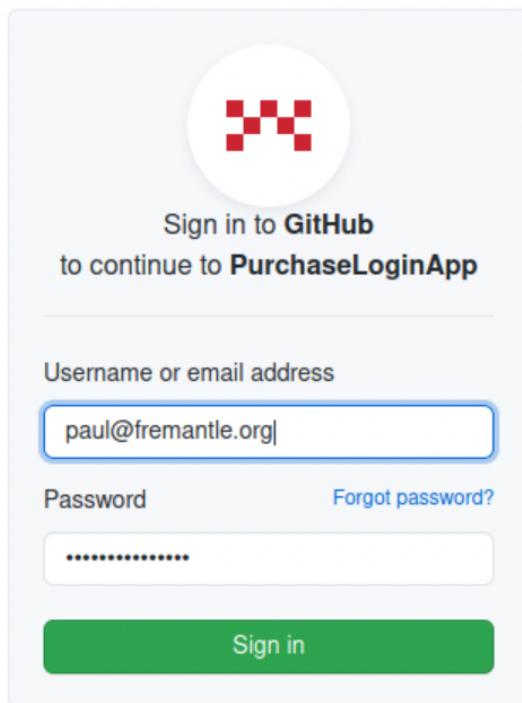
59. Now try your Python login client again. You might need to clear cookies in the browser if its auto logging you in.

Ideally you want to see:

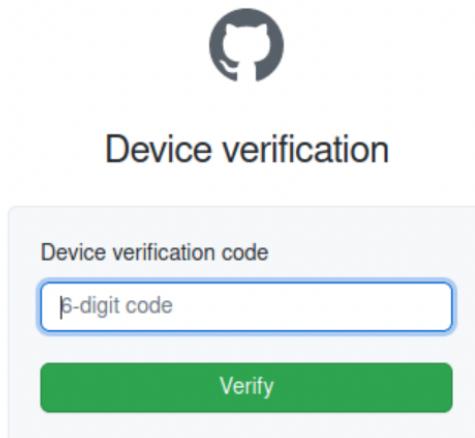


60. Click on Github

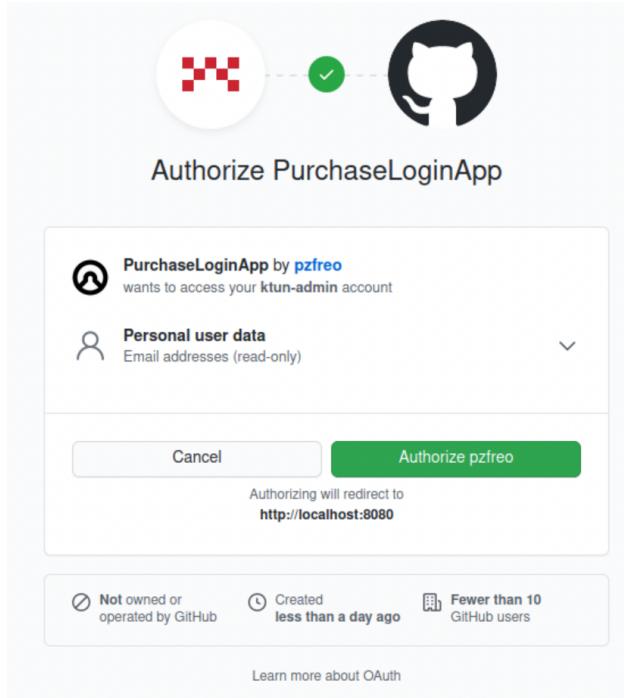
## 61. Login



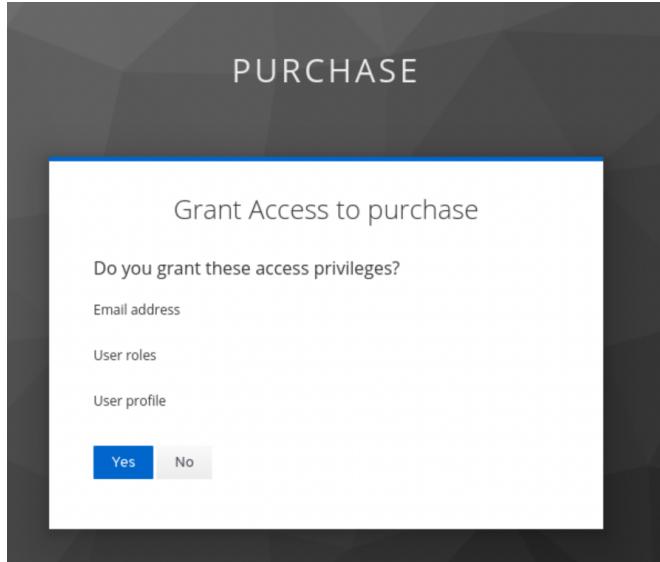
## 62. Do your device verification:



63. Now authorize Purchase to see your userinfo:



64. We still need to authorize the CLI as well:



65. Now try the purchase-create client again.

66. There are two different OAuth2/OIDC flows here:

- Keycloak → Github (Client Credential Grant)
- Python → Keycloak (Device Authorization Grant)

Hence the need for two different permission approvals.

67. Try calling the introspect API yourself with the token via Postman

You need to set the URL to:

`http://localhost:8080/realm/purchase/protocol/openid-connect/token/introspect`

You need to set Auth to Basic and use

Username: purchase

Password: <the client secret from Keycloak>

The screenshot shows the Postman interface with a POST request to `http://localhost:8080/realm/purchase/protocol/openid-connect/token/introspect`. The 'Auth' tab is selected, set to 'Basic Auth'. The 'Body' tab is also selected. In the body section, the 'x-www-form-urlencoded' type is chosen. The 'Username' field contains 'purchase' and the 'Password' field contains a redacted password. A 'Show Password' checkbox is present.

The body needs to be:  
`x-www-form-urlencoded`

With

`token_type_hint: access_token`

`token: <your access token from config secrets here>`

The screenshot shows the Postman interface with the 'Body' tab selected. The 'x-www-form-urlencoded' type is chosen. A table shows two parameters: 'token\_type\_hint' with value 'access\_token' and 'token' with value 'eyJhbGciOiJSUzI1NiIsInR5cC...'. Below the table is a section for 'Key', 'Value', and 'Description'.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> token_type_hint	access_token	
<input checked="" type="checkbox"/> token	eyJhbGciOiJSUzI1NiIsInR5cC...	



68. You shoudl see all the info available:

```
1  [
2      "exp": 1617617615,
3      "iat": 1617617315,
4      "auth_time": 1617617314,
5      "jti": "c8835a57-c69b-490d-ba5f-bc7325344b85",
6      "iss": "http://localhost:8080/realm/purchase",
7      "aud": "account",
8      "sub": "c4663d96-b233-4ae6-8f17-7a9e9f8a570d",
9      "typ": "Bearer",
10     "azp": "purchase",
11     "session_state": "cbc8f047-8096-451e-8f43-0c07a4ac6589",
12     "name": "Paul Fremantle",
13     "given_name": "Paul",
14     "family_name": "Fremantle",
15     "preferred_username": "pzfreo",
16     "email": "pzfreo@gmail.com",
17     "email_verified": false,
18     "acr": "1",
19     "realm_access": {
20         "roles": [
21             "default-roles-purchase",
22             "offline_access",
23             "uma_authorization"
24         ]
25     },
26     "resource_access": {
27         "account": {
28             "roles": [
29                 "manage-account",
30                 "manage-account-links",
31                 "view-profile"
32             ]
33         }
34     },
35     "scope": "email profile",
36     "client_id": "purchase",
37     "username": "pzfreo",
38     "active": true
39 ]
```

69. Congratulations! You have successfully completed this lab!

70. If you plan to do extensions, carry on. Otherwise you can clean up and kill the keycloak and other servers.



# Extensions

1. We can get docker-compose configured to call the Keycloak server (but without dockerising keycloak).

You need to add these to the environment:

```
CLIENT_SECRET: "abbd305e-d92d-44f6-8c36-9543bf92581b"  
OAUTH2_HOST: "host.docker.internal"  
OAUTH2_PORT: 8080
```

Host “host.docker.internal” basically points to the machine which is running docker so that the container can access Keycloak outside of the docker-compose environment.

If you really want to spend a bit of time, you could export the config from your keycloak:

```
bin/kc.sh export --profile=dev --file=config.json
```

And then build a Dockerfile and docker container to incorporate it into the docker-compose. I suggest you save this till you’ve done all the other exercises though!

2. The access token expires quite frequently. Add logic to use the refresh flow to the purchase-create.py

Here is the sample refresh flow from the OAuth2 spec:

<https://tools.ietf.org/html/rfc6749#page-47>

```
POST /token HTTP/1.1  
Host: server.example.com  
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW  
Content-Type: application/x-www-form-urlencoded  
grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

3. If you look at introspect.js (from my plugin) you will see that I pass on the introspected username as req.username.

Change the typescript purchase app to add a “username” field into the model, service and controller so that it saves it into the database to track which user created the entry.

