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
   ```

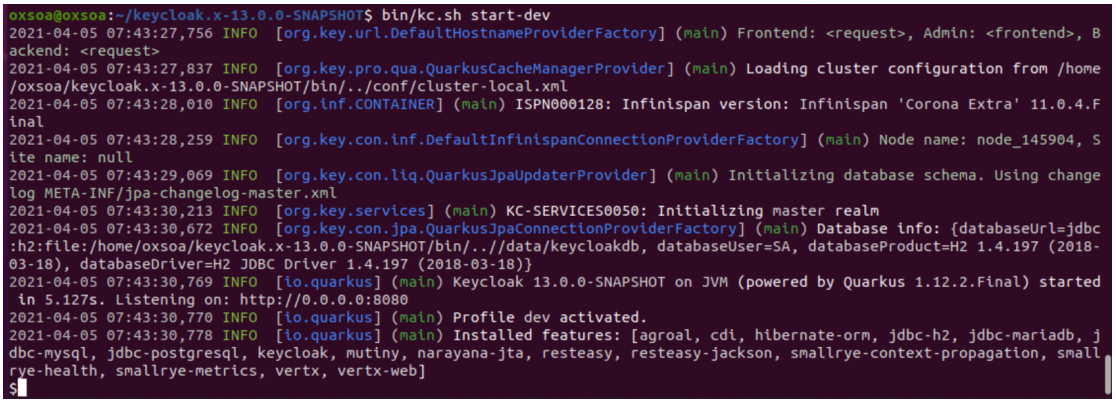
```
wget http://freo.me/keycloak-zip
cd ~
unzip ~/Downloads/keycloak-zip
```

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:



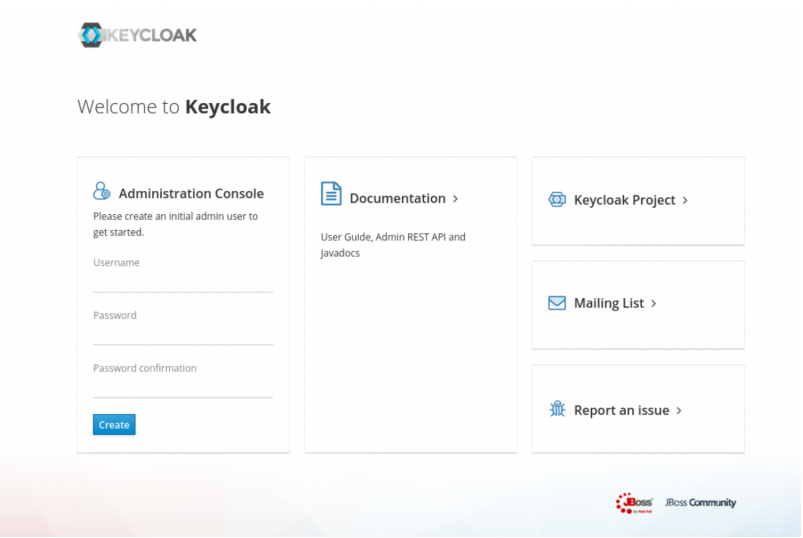
If there is a problem with JDK not found do:

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

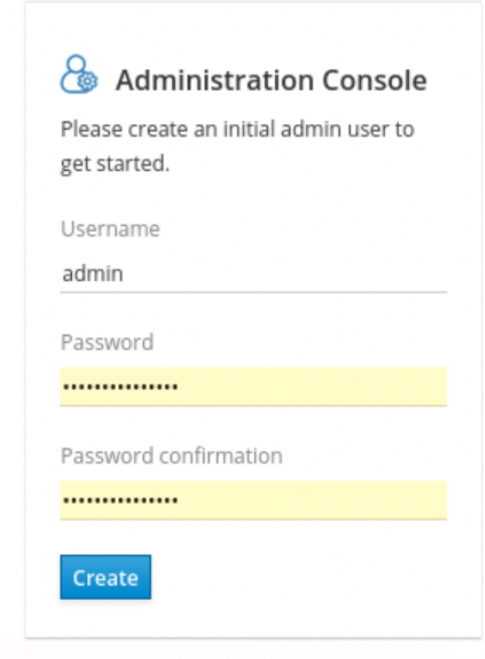
and then try again.

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

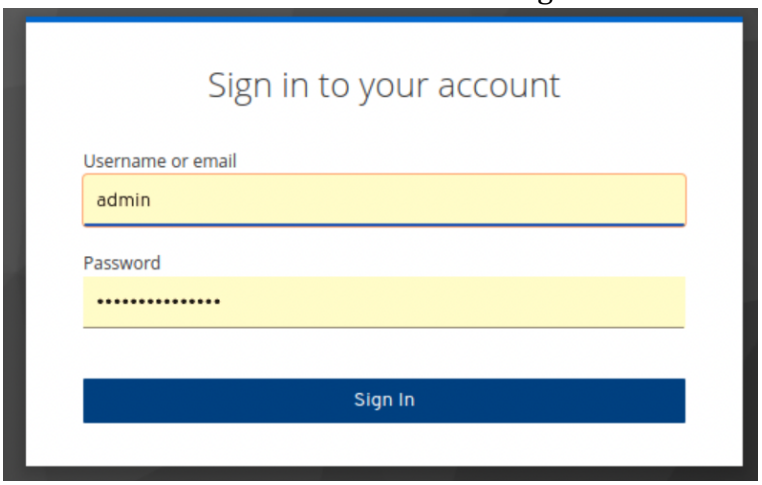
You will see:



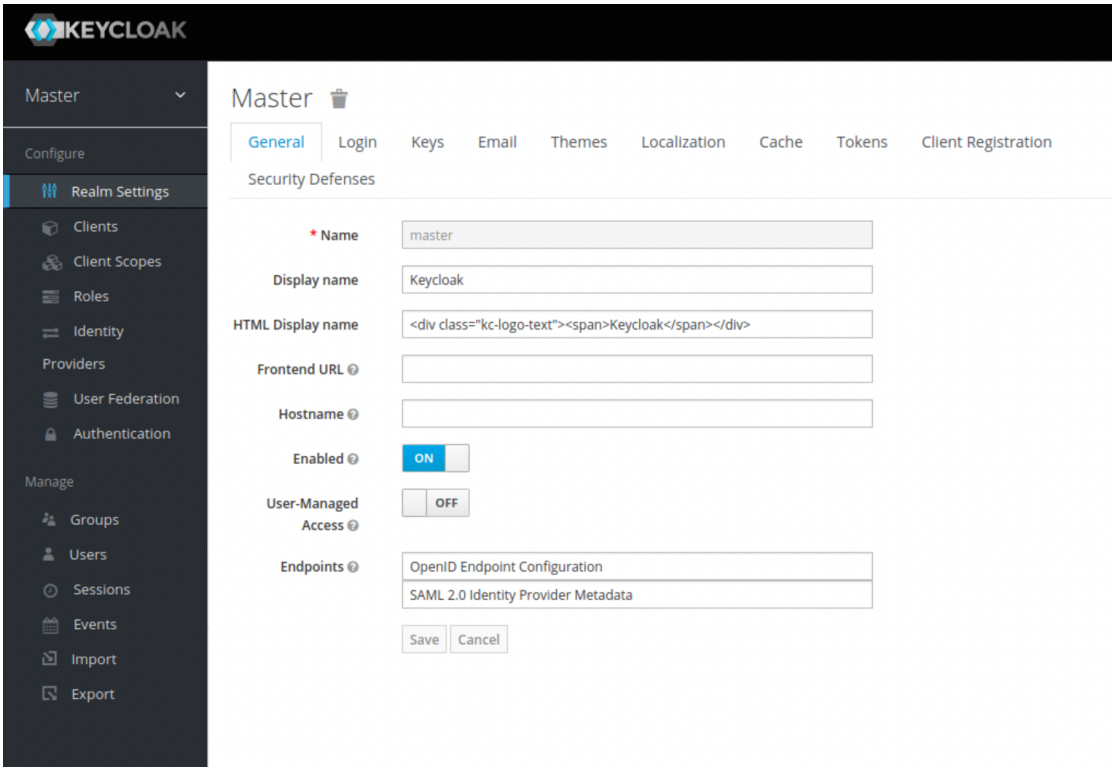
4. Create a new admin user in the box:



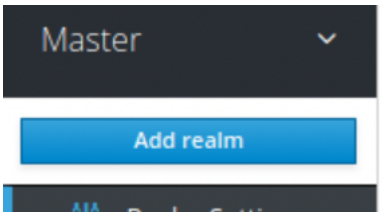
5. Click on **Administration Console** to Login



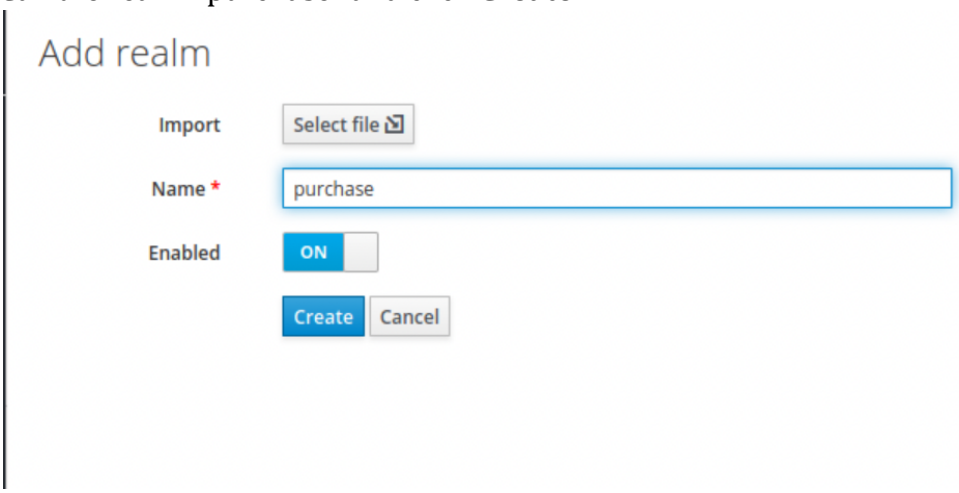
6. You will see something like this:



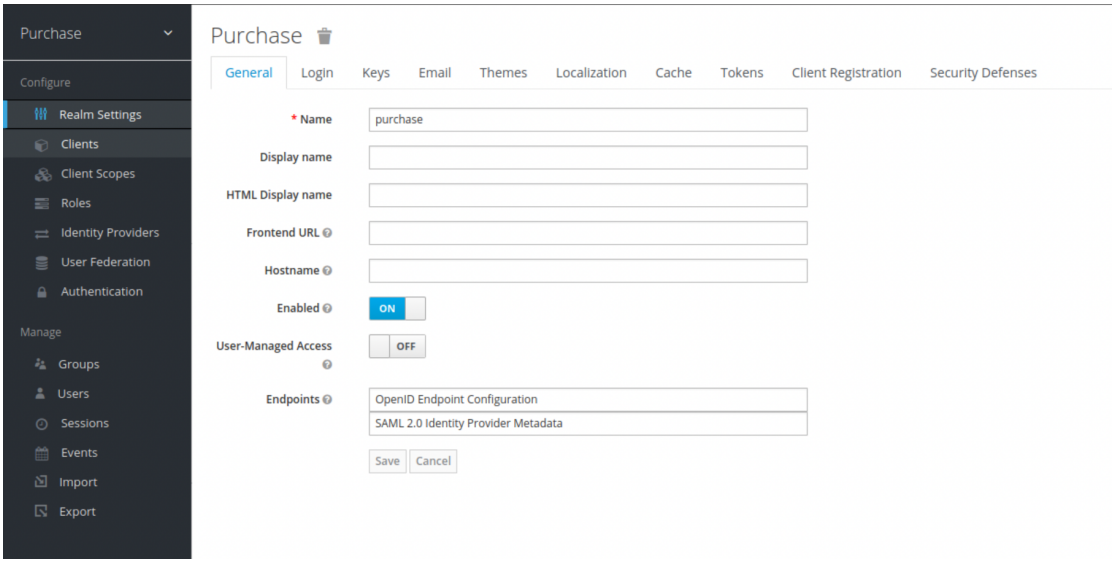
7. We are going to create a new "realm" in which to do authentication. Hover the mouse over the word Master and then click **Add Realm**



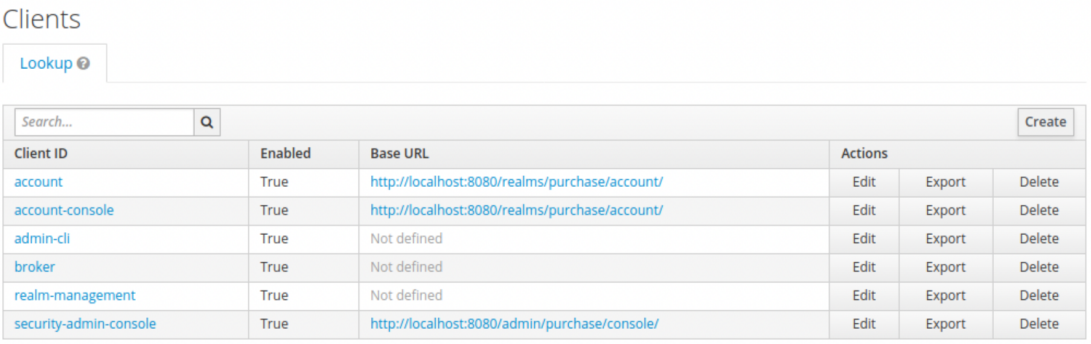
8. Call the realm "purchase" and click **Create**



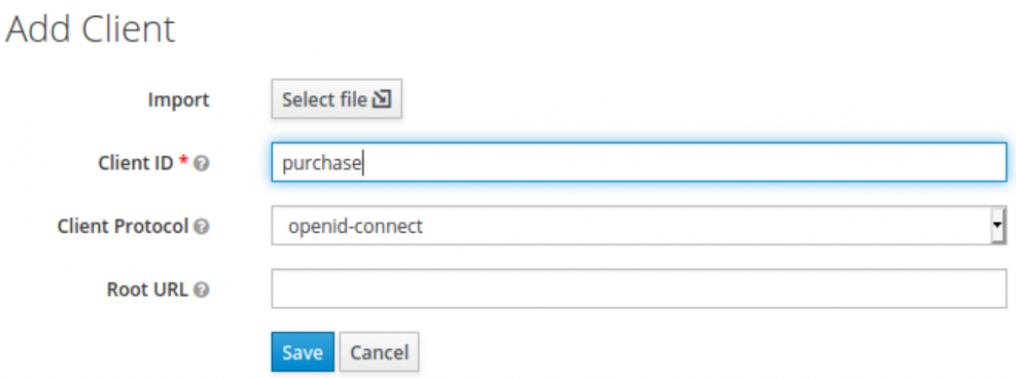
9. There are **lots** of settings we can change. Today we are going to focus just on creating an OAuth2 configuration.



10. First click on **Clients**

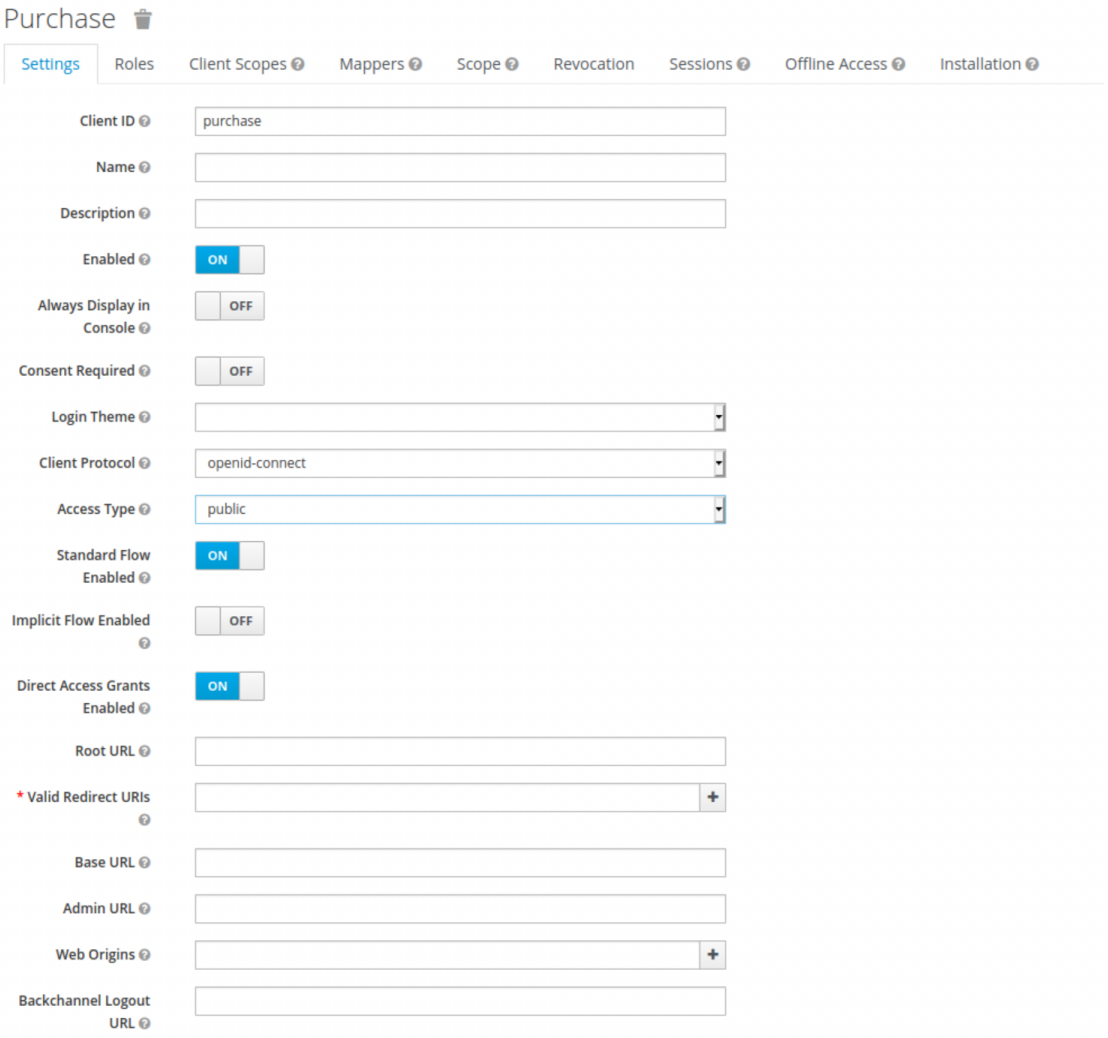


11. Click **Create**

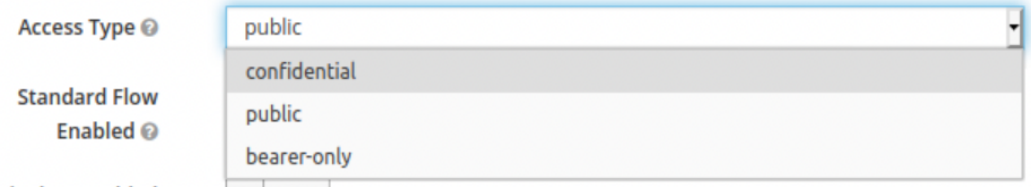


12. Call the client "purchase" and then save.

13. You should see:



14. Change access type to confidential



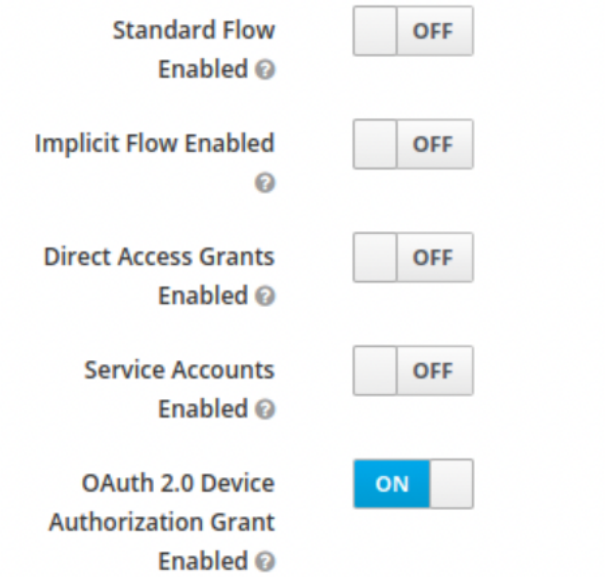
15. You will see a new option appear - Device Authorization Grant.

    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)

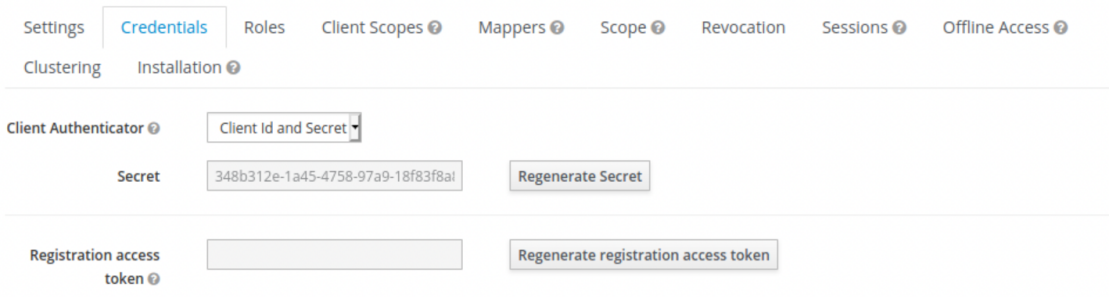
    Turn **on** the Device authorization grant.

    Your panel should look like this:



16. **Save**

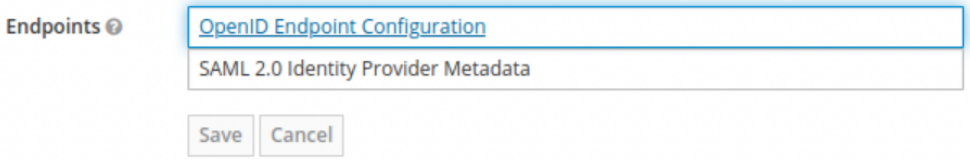
17. Go to the credentials tab and you will see the Client Secret for this OAuth2 application. Take a note of this secret. We will need it.



    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**

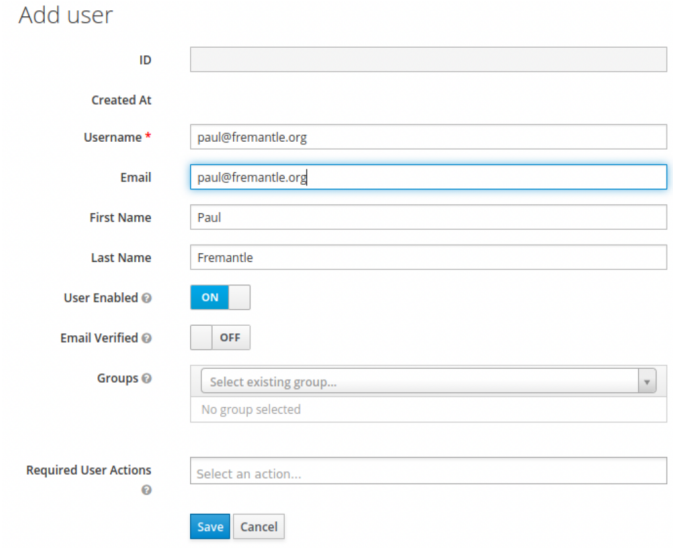


You will see a JSON page with a lot of URLs. I'm assuming you have configured your realm name and client name the same as me and therefore the code examples I have given you will work.

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

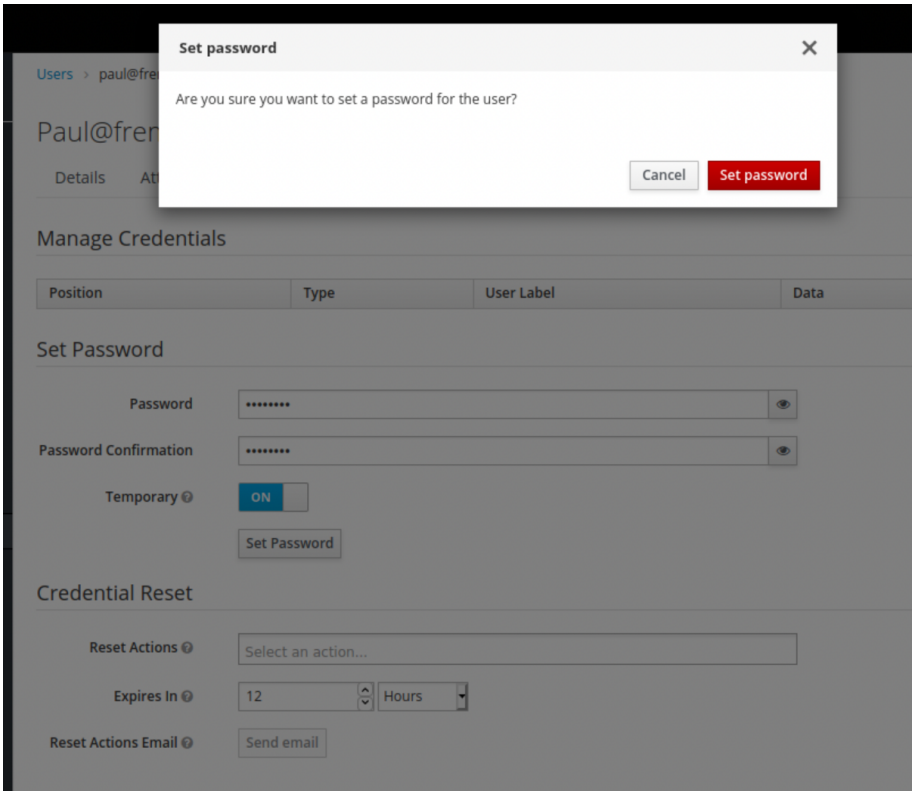
20. Close this tab

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

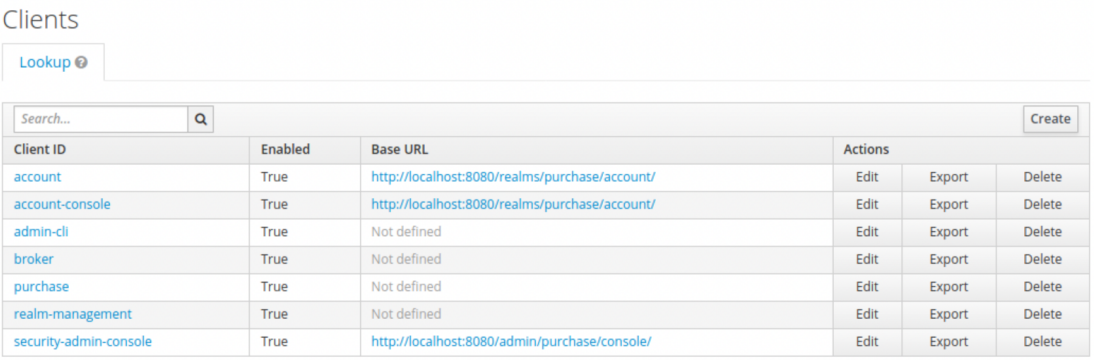


Change to the **Credentials** tab.

22. Set a temporary password for your user:

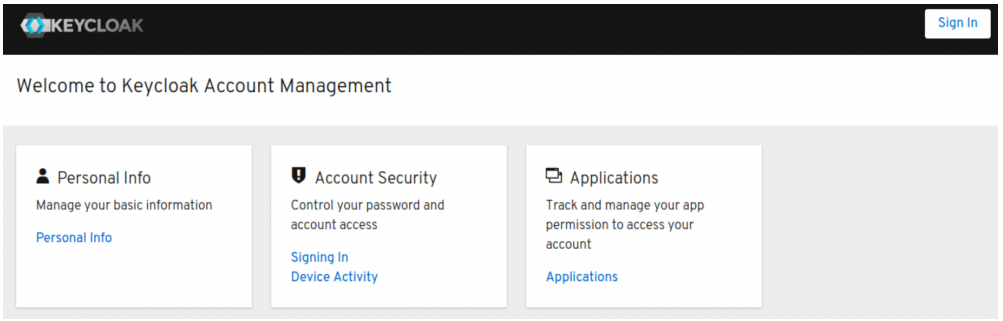


23. One of the clients of the realm is a "account management" app. Go back to the list of clients:



24. Click on the URL next to "account-console".

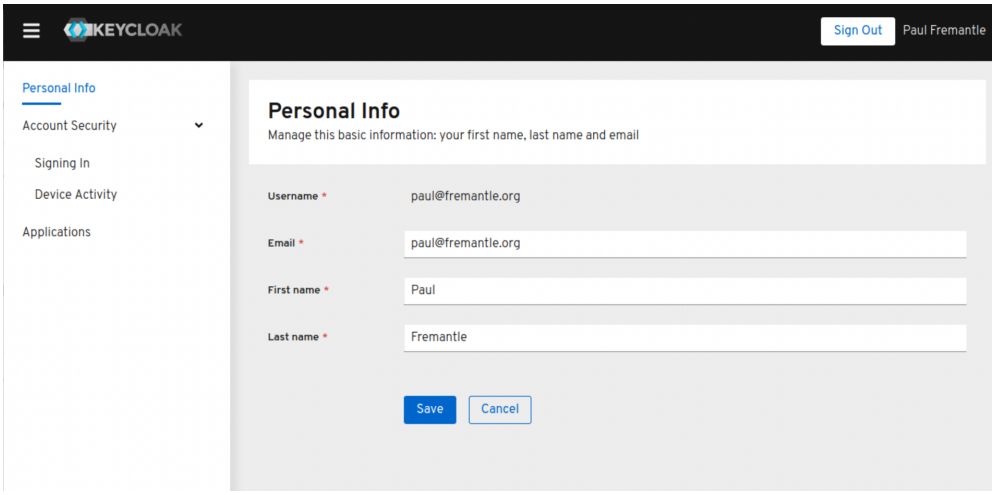
25. You will see a new tab.



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



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 @pzfreo/express-introspect
    ```

    You can find the source code here:
    https://github.com/pzfreo/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 '@pzfreo/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.

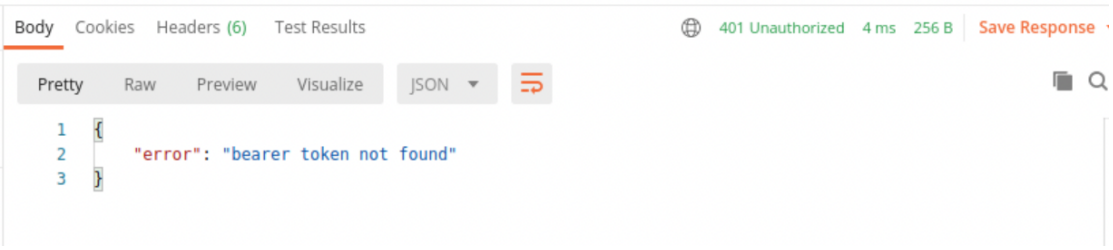
33. Let's try this outside of docker-compose. Start the postgres server standalone:
    `./start-postgres.sh`

34. In another window also in the same directory

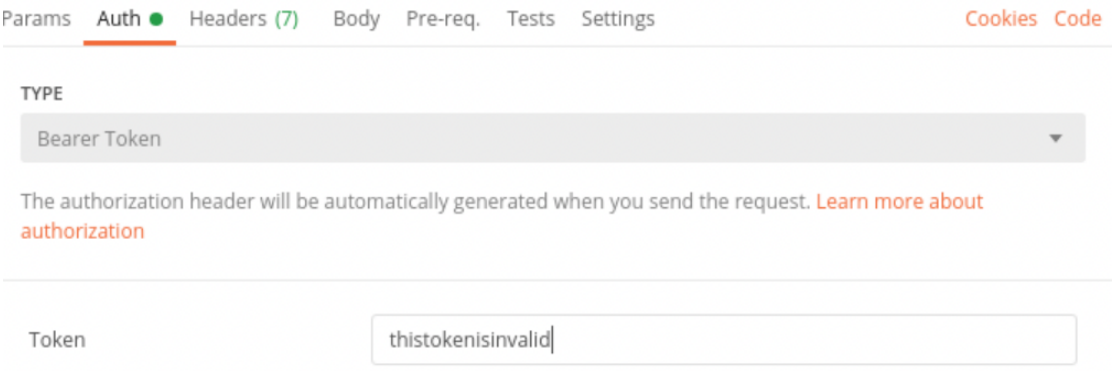
    `export CLIENT_SECRET=`<insert the client secret for the purchase app here>
    `yarn start`

35. Now we can test this out:

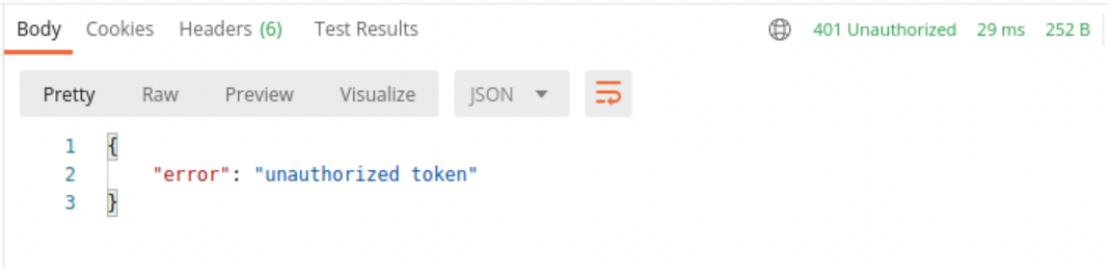
    Try using Postman to call GET http://localhost:8000 with no authorization. You should see the following:



36. Now add a bearer token:

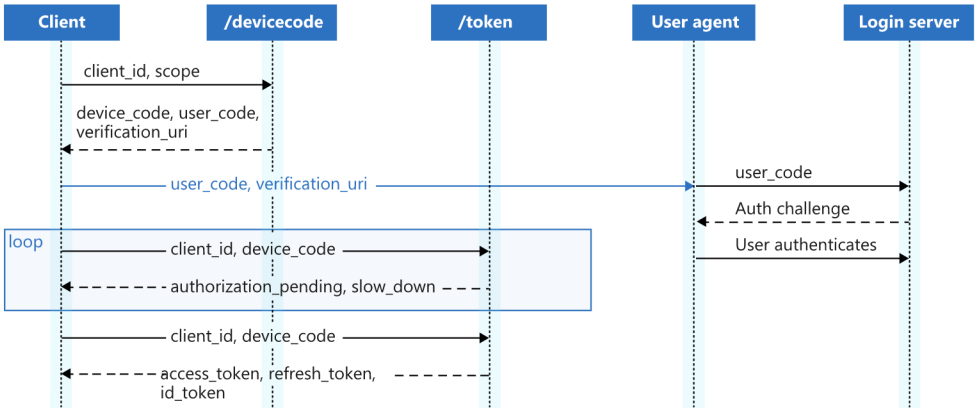


37. You should see a different error:



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>

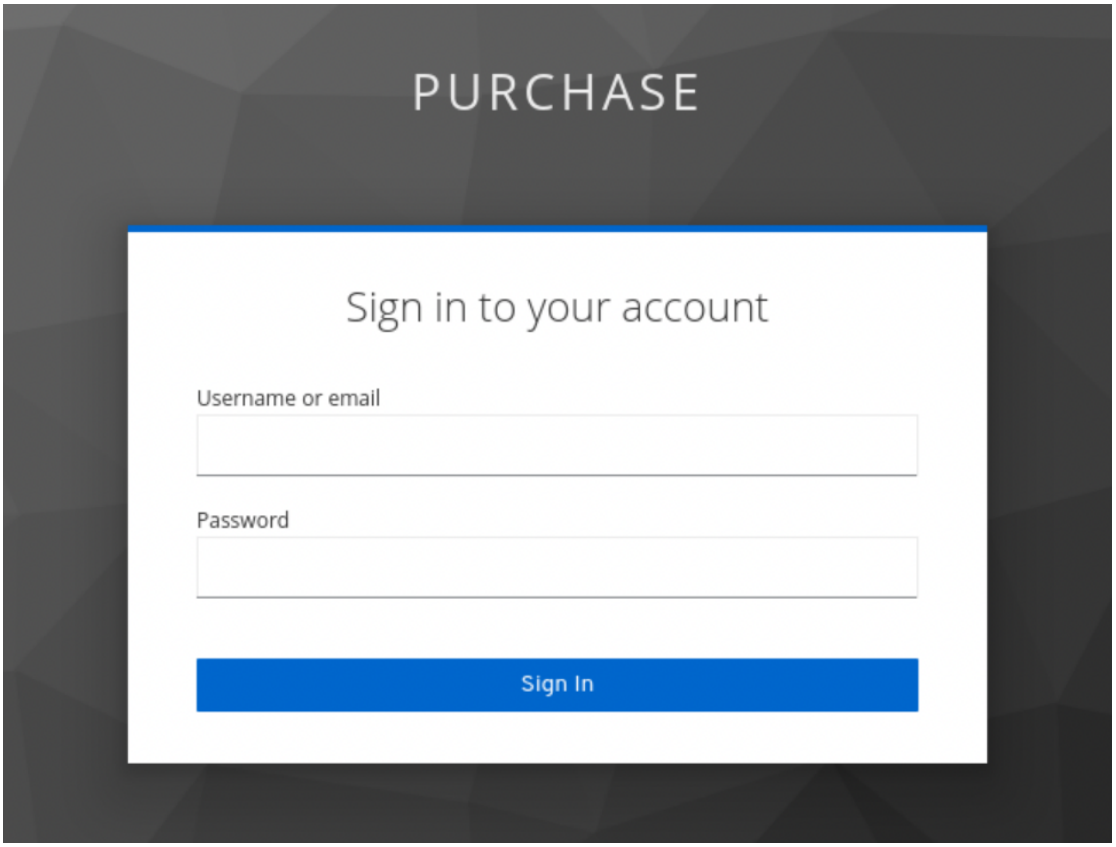
43. In the same terminal type

    python purchase-login.py

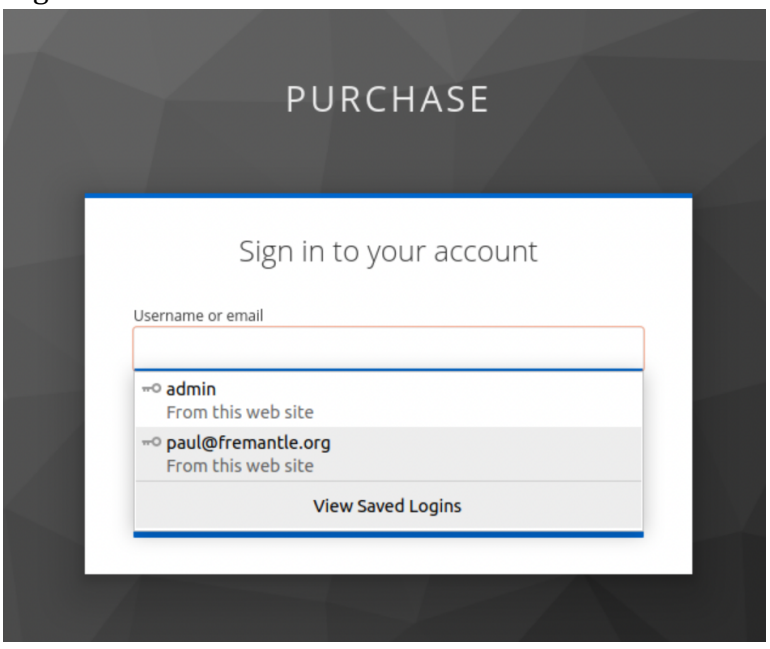
    Two things should happen. In the terminal you should see something like:

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

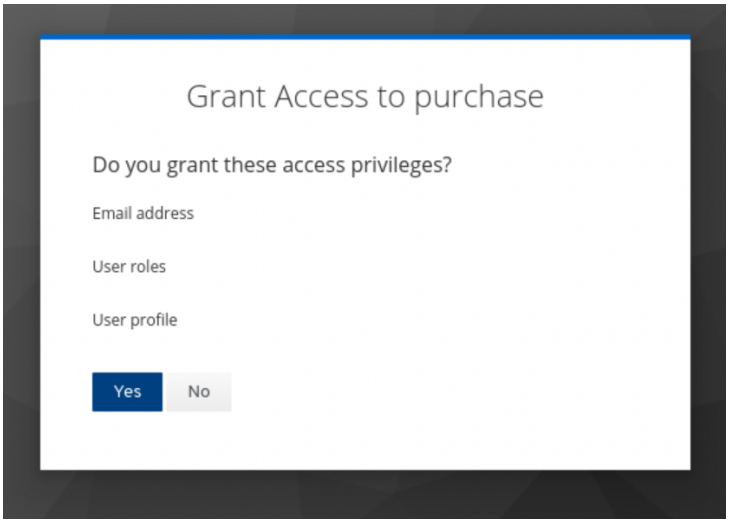
At the same time a browser window should pop open and ask you to login:



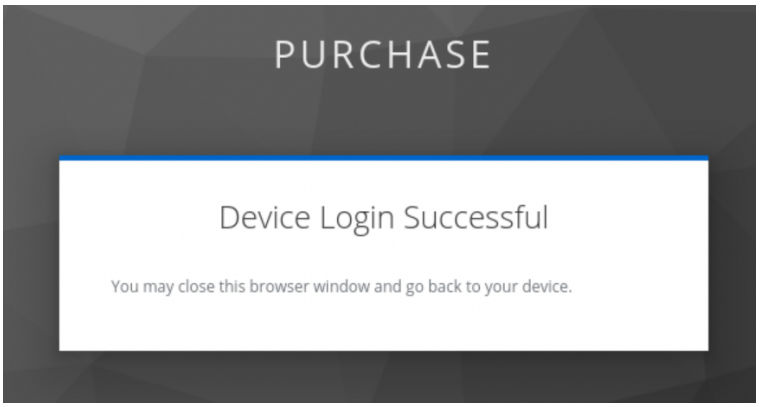
Login as the user



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: eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5cVdDYVctQ3hzU0JiTjJ5RndnQ0dUU2ozM05PaWNuYmFEdV8zd2FCalow
In0.eyJleHAiOjE2MTc2MTM4OTAsImlhdCI6MTYxNzYxMzU5MCwiYXV0aF90aW1lIjoxNjE3NjEzNTg4LCJqdGkiOiJjNjYyY2NkMy0wY2ZkLTQxNjQtYTA0Yy
1jNWYwMWI3ZWU0MzEiLCJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjgwODAvcmVhbG1zL3B1cmNoYXNlIiwiYXVkIjoiYWNjb3VudCIsInN1YiI6IjllMzhmOTk3
LTZmZjEtNDU2ZS05NjMxLWYzYTU5MWYyMTdkMyIsInR5cCI6IkJlYXJlciIsImF6cCI6InB1cmNoYXNlIiwic2Vzc2lvbl9zdGF0ZSI6IjBhNTE1YWM3LTYyN2
QtNDEwMy04Nzc3LWQyNjQyN2NiNTEyNyIsImFjciI6IjEiLCJyZWFsbV9hY2Nlc3MiOnsicm9sZXMiOlsiZGVmYXVsdC1yb2xlcy1wdXJjaGFzZSIsIm9mZmxp
bmVfYWNjZXNzIiwidW1hX2F1dGhvcml6YXRpb24iXX0sInJlc291cmNlX2FjY2VzcyI6eyJhY2NvdW50Ijp7InJvbGVzIjpbIm1hbmFnZS1hY2NvdW50IiwibW
FuYWdlLWFjY291bnQtbGlua3MiLCJ2aWV3LXByb2ZpbGUiXX19LCJzY29wZSI6ImVtYWlsIHByb2ZpbGUiLCJlbWFpbF92ZXJpZmllZCI6ZmFsc2UsIm5hbWUi
OiJQYXVsIEZyZW1hbnRsZSIsInByZWZlcnJlZF91c2VybmFtZSI6InBhdWxAZnJlbWFudGxlLm9yZyIsImdpdmVuX25hbWUiOiJQYXVsIiwiZmFtaWx5X25hbW
UiOiJGcmVtYW50bGUiLCJlbWFpbCI6InBhdWxAZnJlbWFudGxlLm9yZyJ9.mhV8QaDJWZbSahjcmJB9Z7Ap-QjL4hGnUOnNryfn02LNH4jgEFcFNuYNe8xB4IW
35qQkvT0S5uPI5NOQ5eJxvwdawlRzypJeE8ELCDaG6pv9ec93W0oKpXuR9BI3JtXmJkBMw4UkbqdY27nNKnZatzDXpHk_3UHwPSwB28FX77R-hBtvzllwu21rE
4J40lAYk9-pvy9A1Emp7Q5VuoIqAi47okcokCiWPs0O1UI5C93Ij4NSs5Z_GlDFMGCkNbnZlBUGGSW-8S_Z4vUL8NxY2UQ_T5W687ePfGuPS8qo5Bz83uYOg2p
rMj0ag2w7MXNrnXEjG9pJO12GmbAf7RKtGg
refresh_token: eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJlYTc4NmRmZi03ZTQ4LTRkYzUtYTg4Ni0xOTAzNDA4MTczNWQifQ.eyJl
eHAiOjE2MTc2MTUzOTAsImlhdCI6MTYxNzYxMzU5MCwianRpIjoiM2ZlMTNlYjMtYzdjMC00YWYxLWE2MDYtZDMwYzZiNDQ4NTM5IiwiaXNzIjoiaHR0cDovL2
xvY2FsaG9zdDo4MDgwL3JlYWxtcy9wdXJjaGFzZSIsImF1ZCI6Imh0dHA6Ly9sb2NhbGhvc3Q6ODA4MC9yZWFsbXMvcHVyY2hhc2UiLCJzdWIiOiI5ZTM4Zjk5
Ny02ZmYxLTQ1NmUtOTYzMS1mM2E1OTFmMjE3ZDMiLCJ0eXAiOiJSZWZyZXNoIiwiYXpwIjoicHVyY2hhc2UiLCJzZXNzaW9uX3N0YXRlIjoiMGE1MTVhYzctNj
I3ZC00MTAzLTg3NzctZDI2NDI3Y2I1MTI3Iiwic2NvcGUiOiJlbWFpbCBwcm9maWxlIn0.2hnPQrwz1zh0p1kjlyjJwAL-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!

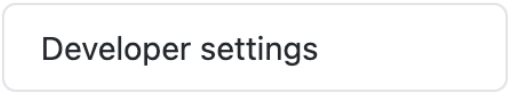
    There are also some extensions at the end that apply to Part A.

Part B - SSO with Github

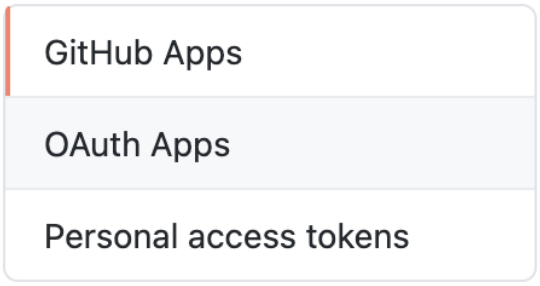
49. However, if you want to do one more thing, we can configure the keycloak server to allow SSO using Github. This will mean we don't need to create ids but can defer to Github for them.

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

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

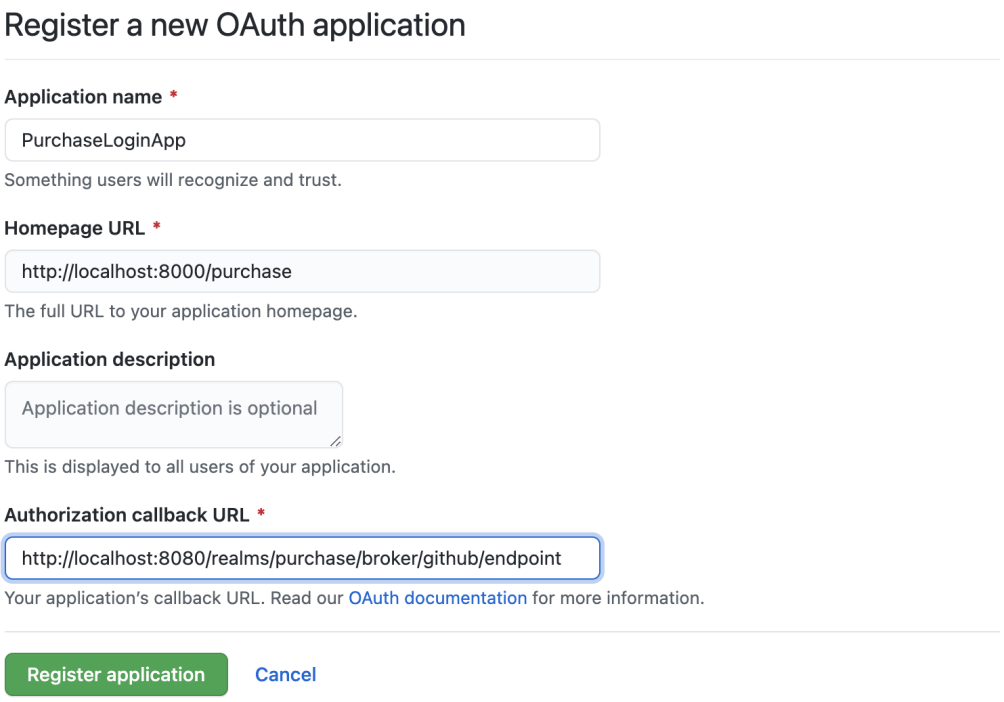


52. Go to OAuth Apps

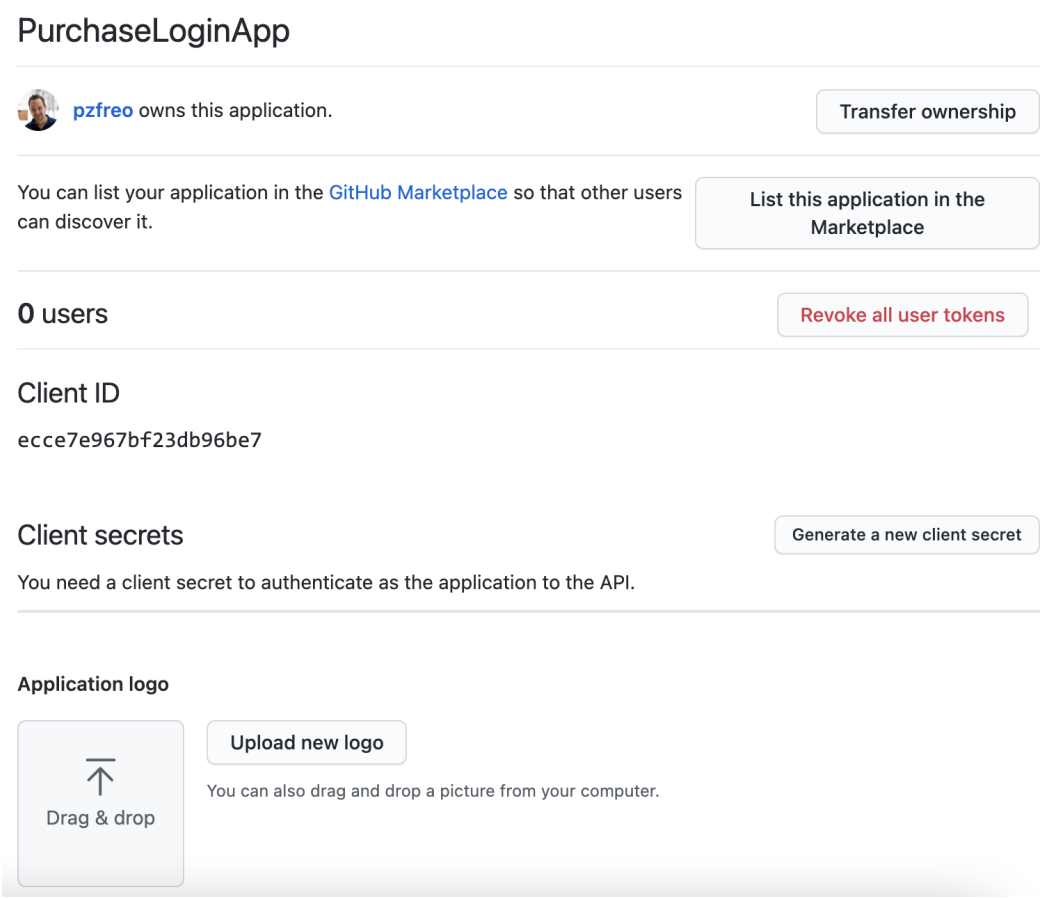


53. Click **New OAuth App**

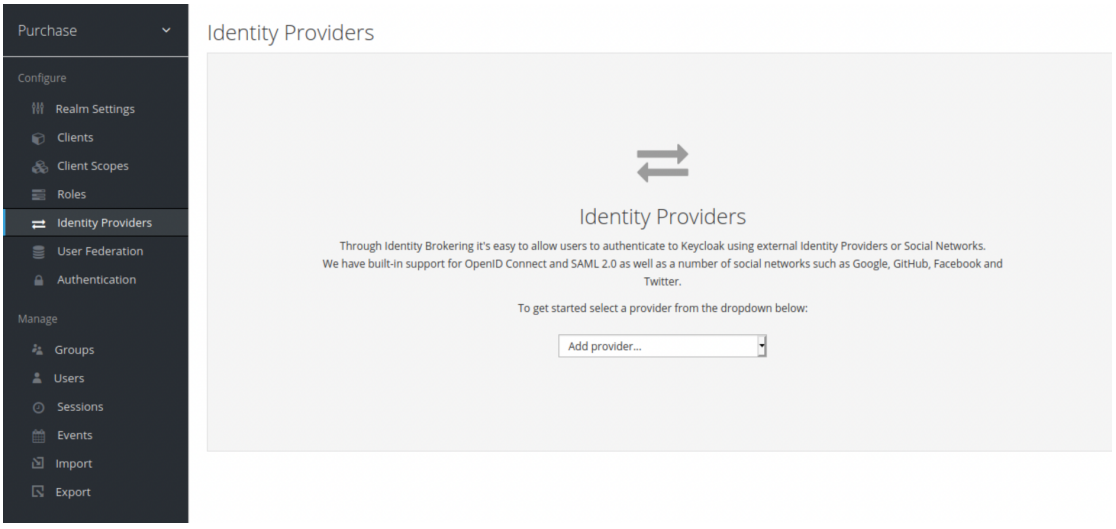
    Fill in the screen like this:



You should see something like:



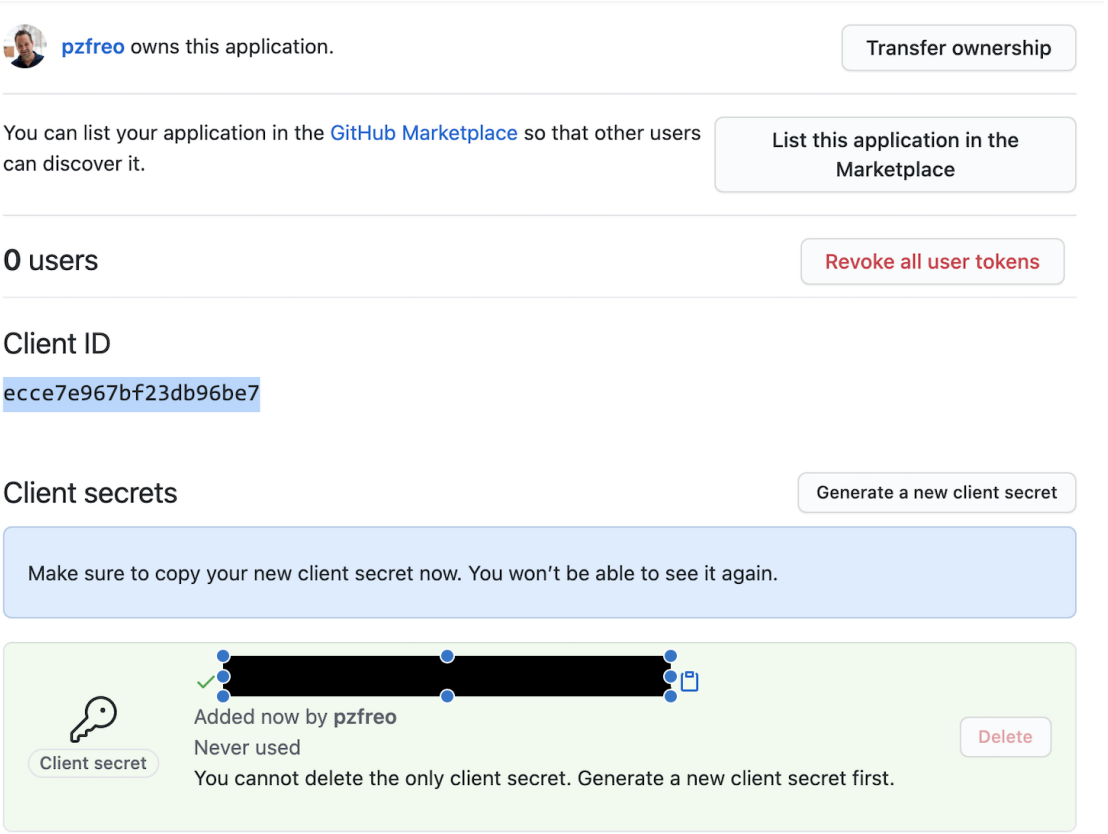
54. Now go to the Keycloak admin console. Click on **Identity Providers**



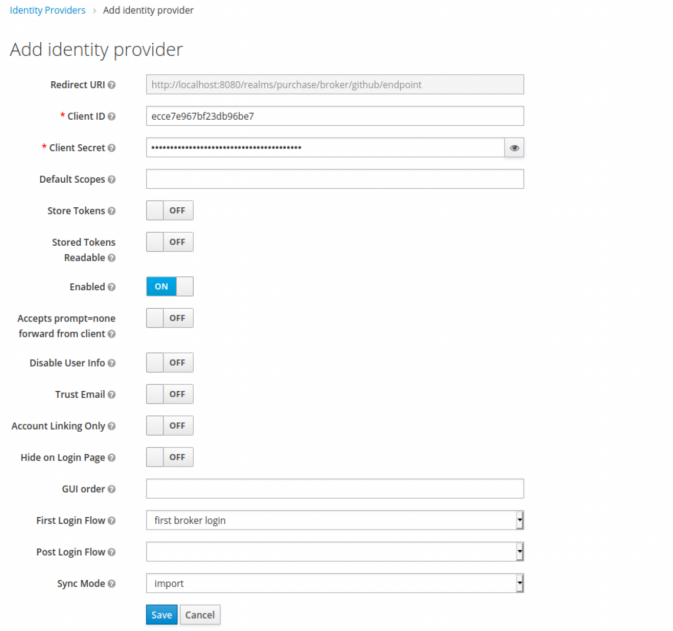
55. Choose Github

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.



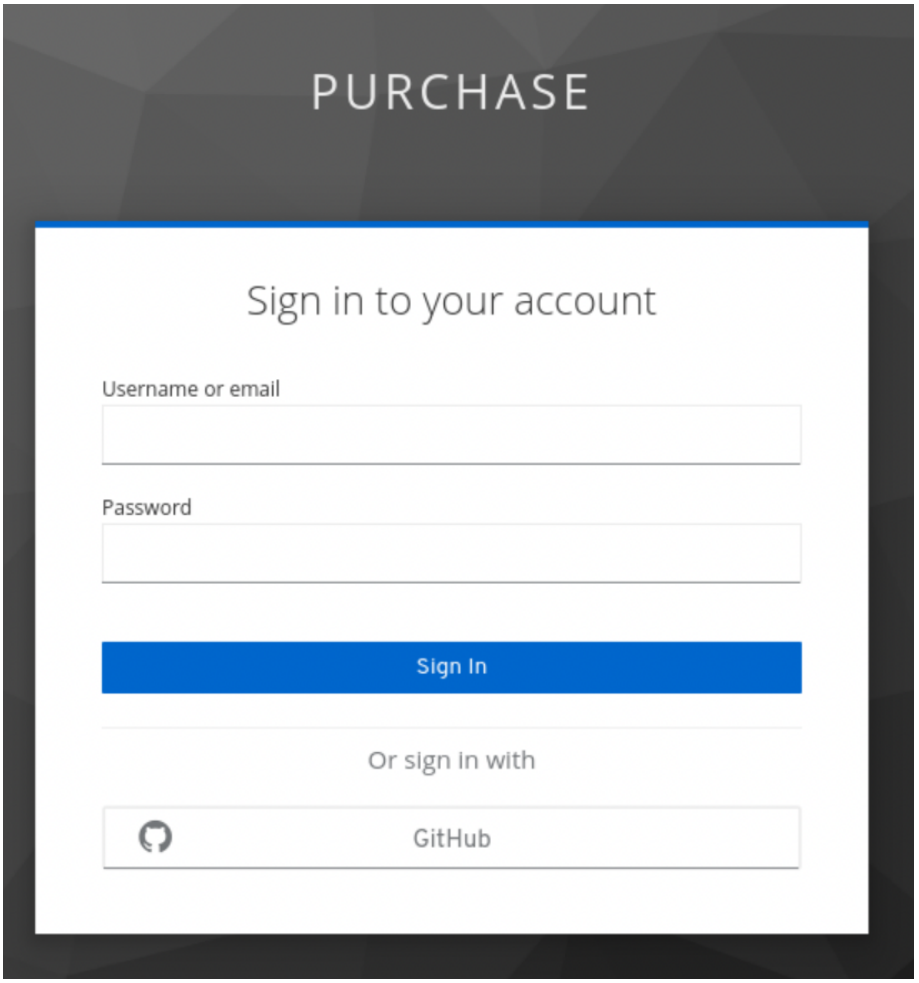
58. Your keycloak should look like:



Save

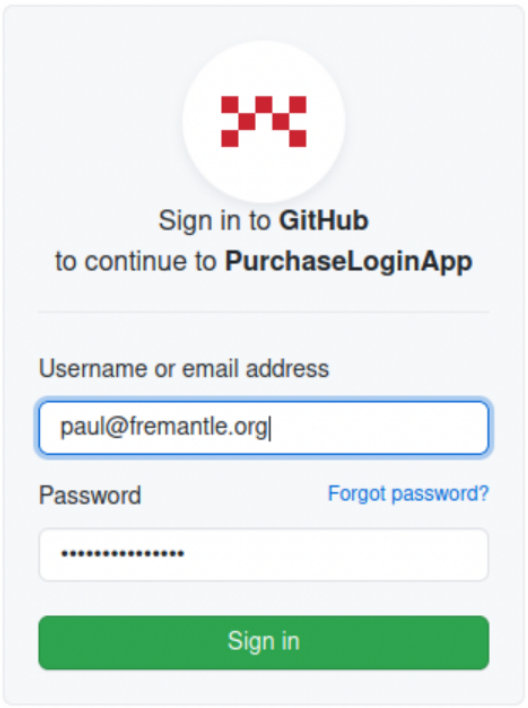
59. Now try your Python login client again. You might need to clear cookies in the browser if its auto logging you in.

    Ideally you want to see:

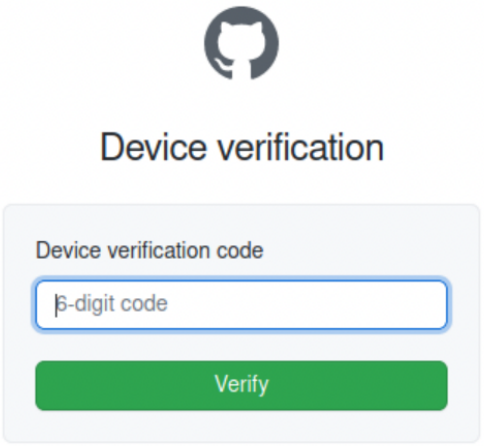


60. Click on Github

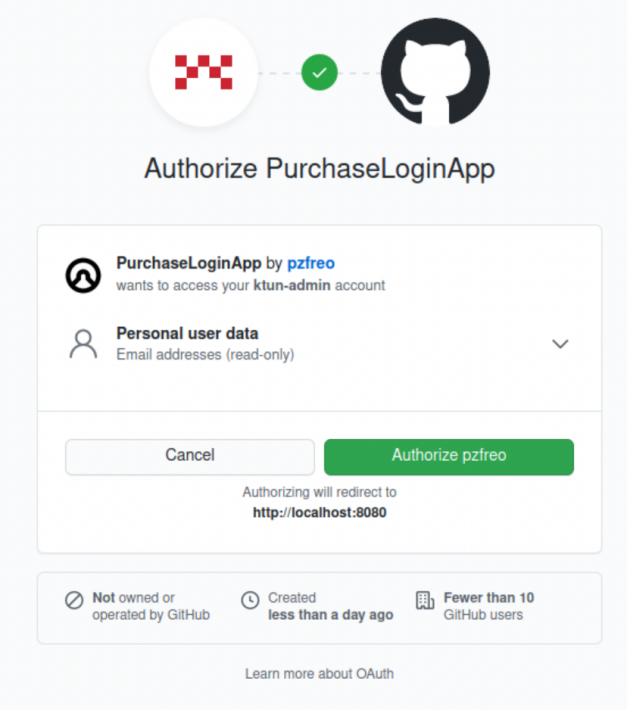
61. Login



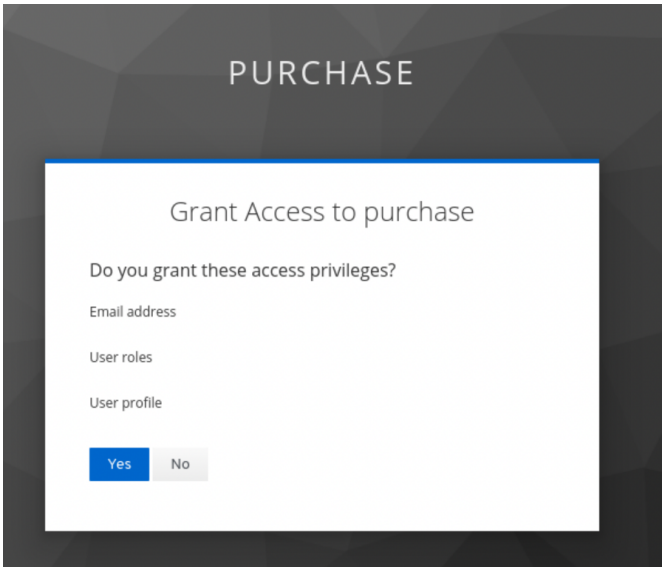
62. Do your device verification:



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.

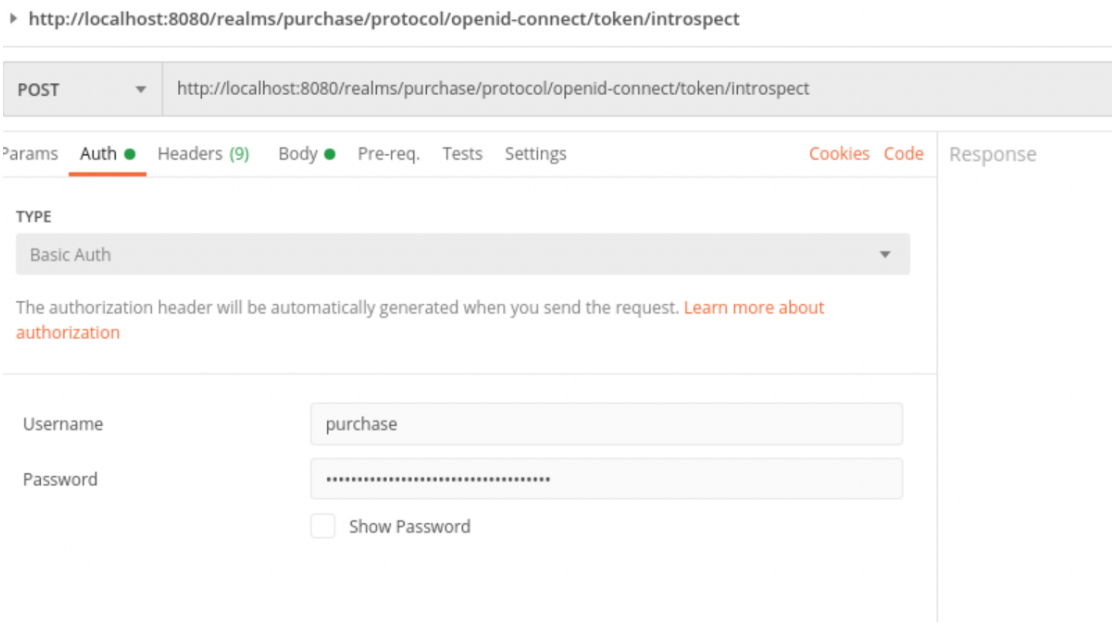
67. Try calling the introspect API yourself with the token via Postman

You need to set the URL to:

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

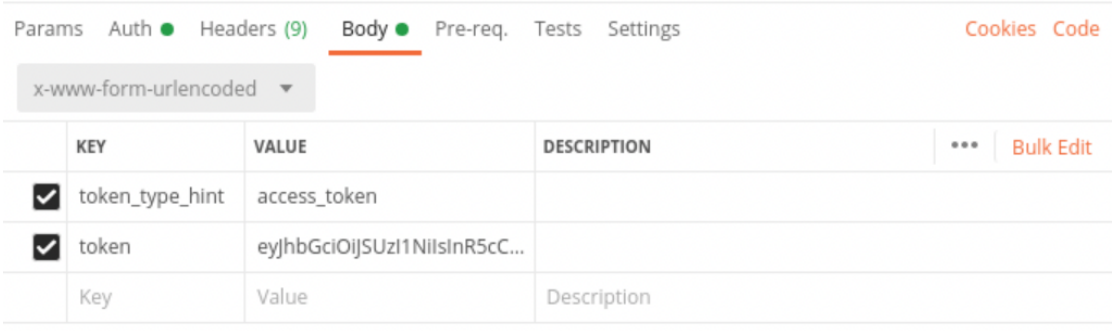
You need to set Auth to Basic and use

```
Username: purchase
Password: <the client secret from Keycloak>
```



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

With
`token_type_hint: access_token`
`token: <your access token from config secrets here>`



68. You shoudl see all the info available:


```
{
    "exp": 1617617615,
    "iat": 1617617315,
    "auth_time": 1617617314,
    "jti": "c8835a57-c69b-490d-ba5f-bc7325344b85",
    "iss": "http://localhost:8080/realms/purchase",
    "aud": "account",
    "sub": "c4663d96-b233-4ae6-8f17-7a9e9f8a570d",
    "typ": "Bearer",
    "azp": "purchase",
    "session_state": "cbc8f047-8096-451e-8f43-0c07a4ac6589",
    "name": "Paul Fremantle",
    "given_name": "Paul",
    "family_name": "Fremantle",
    "preferred_username": "pzfreo",
    "email": "pzfreo@gmail.com",
    "email_verified": false,
    "acr": "1",
    "realm_access": {
        "roles": [
            "default-roles-purchase",
            "offline_access",
            "uma_authorization"
        ]
    },
    "resource_access": {
        "account": {
            "roles": [
                "manage-account",
                "manage-account-links",
                "view-profile"
            ]
        }
    },
    "scope": "email profile",
    "client_id": "purchase",
    "username": "pzfreo",
    "active": true
}
```


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.