MEEC - ADInt

OAUTH2

This document shows how to program a server (Flask) that allows user to authenticate (using the FENIX login page) and access FENIX private information

1 Oauth2

https://oauth.net/2/

OAuth 2.0 is the industry-standard protocol for authorization. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

Read the following tutorials understand the various components of the interaction and the exchanged messages:

http://tutorials.jenkov.com/oauth2/overview.html

https://www.tutorialspoint.com/oauth2.0/index.htm

In order to use ISTid and FENIX (or any other authentication provider) password to authenticate users it is necessary to follow the next generic steps

- register the application in the authentication provider (Google, FENIX, …)
- program a web application so that the user can authenticate on the provider (for instance using the FENIX login page)
- store some temporary token on the server so that can access the protected web services on behalf of the user
- call some service endpoints providing as authentication the temporary token.

The programming of the server application can be done on a had-hoc faction (with specific code implemented for each application) or using a set of libraries.

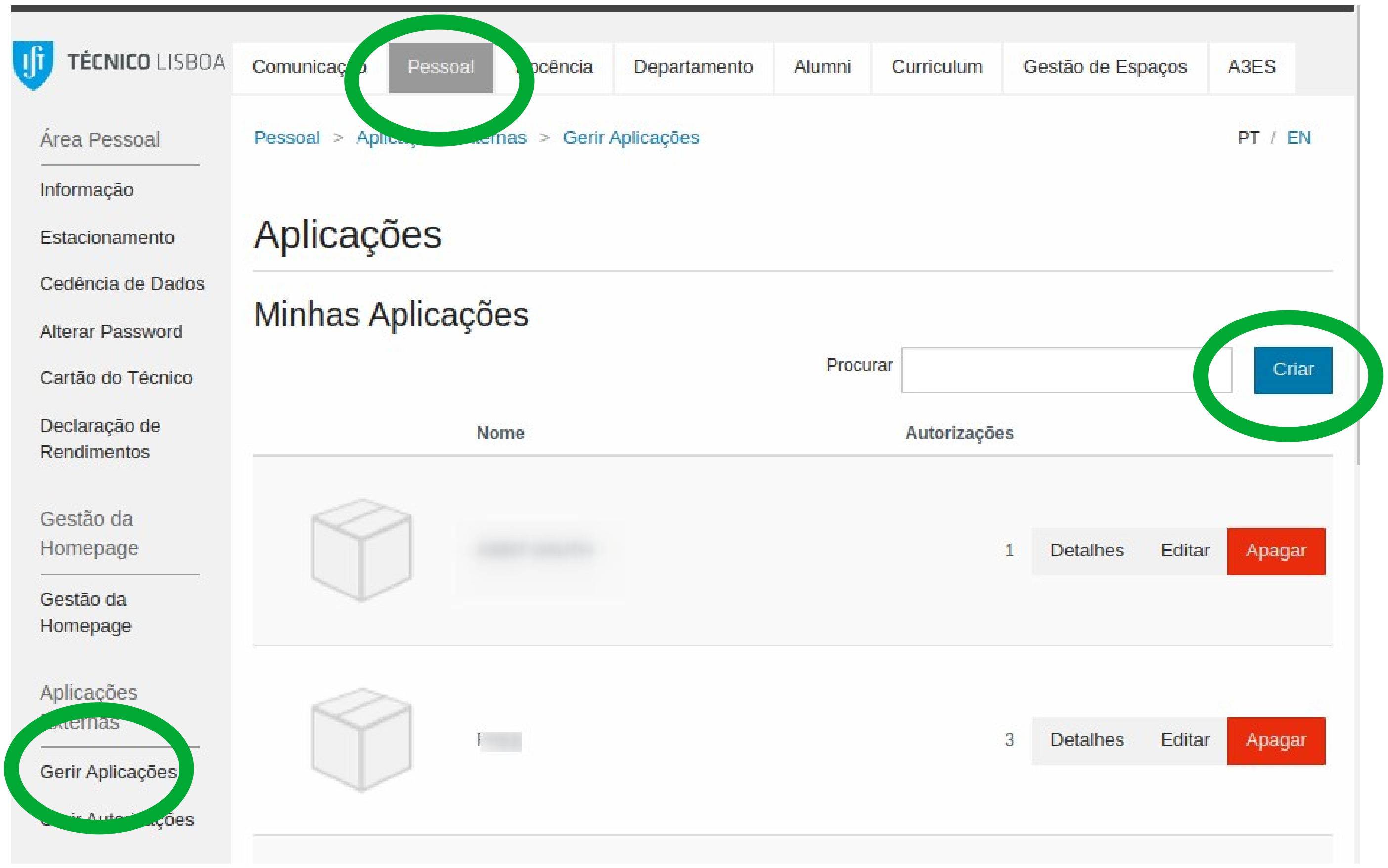
The provided code uses the flask as server code and a special library that performs all interaction with the authentication provider. The provided code also implemented and endpoint that make a authenticated call to FENIX.

The next steps show how to develop a Flask server application that interacts with the FENIX API after the user is authenticated.

2 FENIX client configuration

Before programming the server it is necessary to register the application on FENIX:

- **Pessoal / Personal**
- **Gerir aplicações / Manage Applications**
- **Criar / Create**



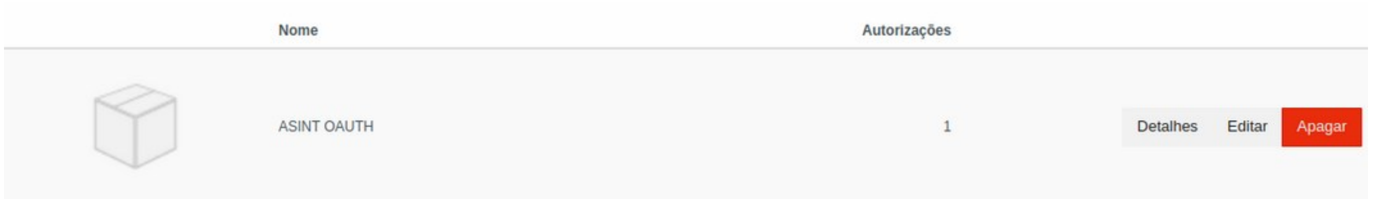
In the presented form it is necessary to fill the following information

- **Nome / Name**
- **Descrição / Description**
- **Site** – this url should be the one that is printed by flask (http://127.0.0.1:5000 or ny other suitable)
- **Redirect Url** – for the application to use FENIX authentication this URL should contain the locatino of the server followed by **/callback/fenix** (in this example it can be

http://127.0.0.1:5000/callback/fenix, and the students should only change the address and port)

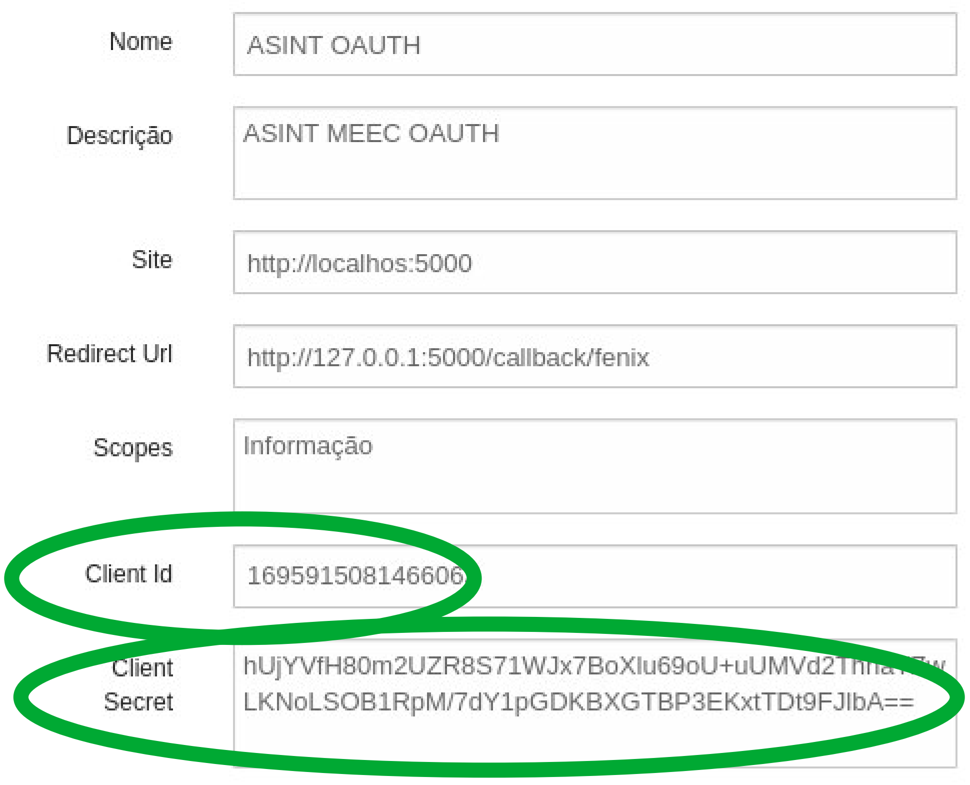
- Scopes – here the student should select the type of information that the python will fetch from FENIX after the user is authenticated. This information can be seen in https://fenixedu.org/dev/api/ :
GET /person/courses Private Curricular Scope

After submitting all this information, the new application is now available on the list:



	Nome	Autorizações	
	ASINT OAUTH	1	Detalhes Editar Apagar

The application creation generates two values that should be used by the python code when authenticating a user.



Nome	ASINT OAUTH
Descrição	ASINT MEEC OAUTH
Site	http://localhos:5000
Redirect Url	http://127.0.0.1:5000/callback/fenix
Scopes	Informação
Client Id	169591508146606
Client Secret	hUjYVfH80m2UZR8S71WJx7BoXlu69oU+uUMVd2Thna[illegible]w LKNoLSOB1RpM/7dY1pGDKBXGTBP3EKxtTDt9FJlbA==

The relevant values that should be copied to the Flask project are:

- Client Id
- Client Secret

3 Configuration of FLASK

The provided Flask example is ready to run, authenticate the user and access some user private information.

The provided code is based on the following project:

https://blog.miguelgrinberg.com/post/oauth-authentication-with-flask-in-2023

https://github.com/miguelgrinberg/flask-oauth-example

The various requirements can be installed with one simple command:

```
pip3 install -r requirements.txt
```

The configuration can be done creating a .env file with the following information

```
FENIX_CLIENT_ID=1695915081466

FENIX_CLIENT_SECRET=hUjYVfH8S71WJx7BoXlu69oU+uUMVd2ThhaTZwLKNoLSOB1RpM/
7dY1pGDKBXGTBP3EKxtTDtlbA==
GITHUB_CLIENT_ID=
GITHUB_CLIENT_SECRET=
GOOGLE_CLIENT_ID=
GOOGLE_CLIENT_SECRET=
```

The values to put in these lines should be copied from the FENIX as described earlier.

The presented code (**app.py**) shows in the **@app.route('/')** endpoint a button for the user to login or logout. When the user presses these buttons various interactions between the browser, flask code and authentication provide (FENIX) implement the login/logout.

The endpoints that implement the OAuth 2.0 protocol

- **/logout**
- **/authorize/<provider>**
- **/callback/<provider>**

do not need to be modified for a simple usage.

4 verification of authenticated user on code and templates

When writing the python code or templates it is necessary to verify if the user is logged in or not. This can be done in the python code or on the template:

Python code

```
class User(UserMixin, db.Model):
  __tablename__ = 'users'
  id = db.Column(db.Integer,
                primary_key=True)
  username = db.Column(db.String(64),
                     nullable=False)
  email = db.Column(db.String(64),
                  nullable=True)
  token = db.Column(db.String(300))
```

```
current_user.id
current_user.email
current_user.is_authenticated
```

Template

```
{% if current_user.is_authenticated %}
   FENIX Authenticated</div>
{% else %}
   NOT FENIX Authenticated</div>
{% endif %}
```

In the python code the **current_user** variable contains all the information about the current user accessing the server. The definition of the model stores the username, e-mail and a temporary token.

If more information is necessary the model can be changed and more information added when inserting the user in the database (function **oauth2_callback**).

Besides the information on the Database, the **current_user** variable contains the attribute **is_authenticated** that allows the execution of different code wether the current code is running for a authenticated user or not.

In the template it is possible to access the same variable **current_user**.

5 Call of FENIX API

After the user is authenticated and the necessary information is put into the database, it is possible for the python code to make call to FENIX on behalf of that user.

The **/other** endpoint tries to access the FENIX API to retrieve the name of the current user.

```
@app.route('/other')
def other_route():
  url = "https://fenix.tecnico.ulisboa.pt/api/fenix/v1/person"
  try:
    response = requests.get(url,
                            headers={'Authorization': 'Bearer ' +
                                                     current_user.token,
                                     'Accept': 'application/json',
                                     })
    if response.status_code != 200:
      abort(401, "not authorized")
    user_info = response.json()
    print (user_info)
    return "returned name "+user_info['name']
  except:
  abort(401, "not logged in in FLASK")
```

For instance, in order to retrieve further private user information from FENIX the server code should access the **fenix.tecnico.ulisboa.pt/api/fenix/v1/person** endpoint. For FENIX to know what users is authenticated it is necessary to send on the headers of the request the toke:

```
     Authorization': 'Bearer ' + current_user.token
```

if the user is authenticated and has permissions to access such resource, the FENIX returns the requested information.

Using this calls it is possible to access all the private endpoints described in

https://fenixedu.org/dev/api/

I the endpoint belongs to a slope that was not configured initially, it is possible to go to FENIX (as describes in section 2) and modify the scope of the registered application.