

---

## Introduction aux web serveurs

**Professeur:** Jean-Loïc De Jaeger

**Due Date:** 11 avril 2024

**TP:** 1

---

### Introduction

L'objectif de ce TP est la mise en pratique de la théorie apprise en cours sur les serveurs REST.

Le TP comprend le set up d'une base de donnée PostgreSQL ainsi que d'une interface graphique, une API REST avec requêtes SQL et une API REST avec un ORM.

## 1 Set up de PostgreSQL

Allez sur le lien github suivant [github](#) et cloner le projet sur votre machine.

### 1.1 Utilisation de Docker

Docker permet de containeriser des applications au sein de containers et donc de pouvoir runner des applications comme MySQL, PostgreSQL, MongoDB, nos propres applications très rapidement.

Docker enlève le besoin de faire toutes les install manuellement. Il suffit de récupérer une image docker et de lancer le container à partir de cette image.

Une des façons de lancer plusieurs containers en même temps est de faire avec la commande docker-compose. Le docker-compose est configuré avec un fichier yaml. Le code du fichier yaml est donné ci-dessous et devra être mis dans un fichier docker-compose.yml à la racine de votre projet.

Lancer ensuite la commande suivante:

```
docker-compose up
```

Vérifier que tous les containers de PostgreSQL et PgAdmin tournent sans problèmes.

```
docker ps
```

### 1.2 Set up de la database

Allez à l'url <http://localhost:5050/> avec votre navigateur. Utilisez admin@admin.com pour l'adresse email et root pour le mot de passe pour vous connecter.

Créez un nouveau serveur en renseignant les informations de votre base de donnée PostgreSQL de la façon suivante:

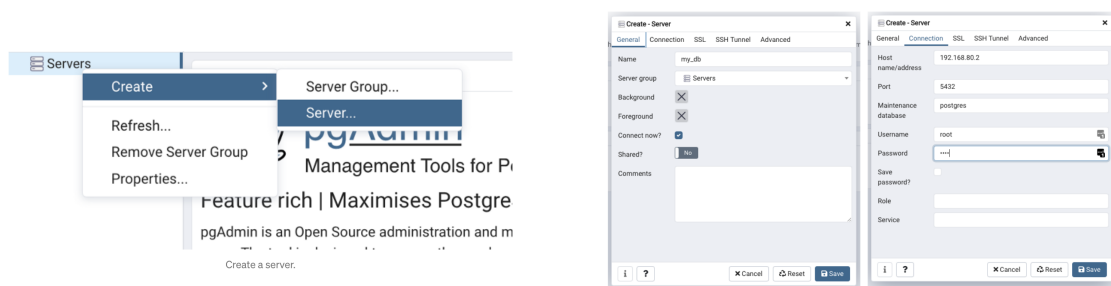


Figure 1: Création d'un nouveau server avec PgAdmin

docker ps							
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES	
7633c02294a5	dpape/pgadmin4	"/entrypoint.sh"	2 hours ago	Up About an hour	443/tcp, 0.0.0.0:5050->80/tcp	pgadmin4_container	
fcc97e066cc8	postgres	"docker-entrypoint.s..."	2 hours ago	Up About an hour	0.0.0.0:5432->5432/tcp	pg_container	

Figure 2: docker ps pour trouver l'adresse ip

L'adresse ip du container peut être trouvée avec le commande docker suivante :

```
docker ps
```

Vous pouvez maintenant créer une database du nom de store.

### 1.3 Environnement Virtuel

Comme pour tout projet python, il est important de créer un environnement virtuel pour isoler les dépendances de votre projet.

Un fichier requirements.txt vous a été fourni dans le projet.

Créer votre environnement virtuel et installé les libraries python à partir du fichier requirements.txt.

Pour ceux qui utilisent Mac OS, vous devrez très certainement installé PostgreSQL sur vos machines. La procédure est train simple. Il faut d'abord installer brew :

<https://brew.sh/>

Et ensuite faire la commande suivante :

```
brew install postgresql
```

Une fois toutes ces étapes faite, votre projet est prêt et nous allons pouvoir commencer le développement de notre API.

## 2 Développement de l'API

Le but de ce projet est de créer deux tables. Une table User et une table Application.

La table User comporte des informations sur un User et Application comporte des informations sur les applications utilisées par le user.

On donne le modèle de donnée suivant :

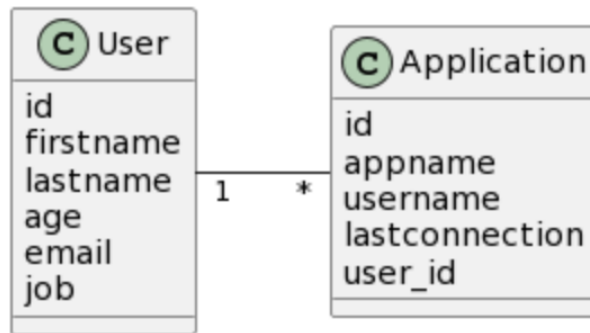


Figure 3: Modèle de données du TP

## 2.1 Développement en SQL

Dans un premier temps, nous allons créer la table User et Application en SQL et lancer son exécution dans notre base PostgreSQL avec SQLAlchemy.

Faites les deux requêtes SQL (dans un string ou un fichier externe) pour créer les tables.







```
create_user_table_sql = """
"""

create_application_table_sql = """
"""
```

### 2.1.1 Remplir les tables

Développez une fonction populate() avec la librairie Faker qui va gérer des données factices et remplir les deux tables.

Une fois cela fait, vous devriez pouvoir visualiser les tables sous PgAdmin:

Data Output		Explain	Messages	Notifications	
	id [PK] integer 	appname character varying (100) 	username character varying (100) 	lastconnection timestamp with time zone 	user_id integer 
1	1	LinkedIn	david25	2022-04-07 01:59:26.261379+00	1
2	2	Instagram	mary20	2022-04-07 01:59:26.261551+00	1
3	3	LinkedIn	cwilliams	2022-04-07 01:59:26.261688+00	1
4	4	LinkedIn	andrewbailey	2022-04-07 01:59:26.281596+00	3
5	5	Facebook	jonesjanice	2022-04-07 01:59:26.291592+00	4
6	6	Twitter	charlesortiz	2022-04-07 01:59:26.307948+00	6
7	7	Airbnb	larsonsharon	2022-04-07 01:59:26.316841+00	7
8	8	Airbnb	crystal33	2022-04-07 01:59:26.331608+00	9
9	9	Airbnb	tbenson	2022-04-07 01:59:26.331797+00	9
10	10	TikTok	rodriguezjames	2022-04-07 01:59:26.331957+00	9
11	11	TikTok	brownashley	2022-04-07 01:59:26.341283+00	10
12	12	Instagram	lesterjennifer	2022-04-07 01:59:26.350067+00	11

## 2.1.2 Implémentation des methodes REST

La dernière étape consistera à implémenter la méthode GET.

## 2.2 Développement avec un ORM

Les ORM (Object-Relational Mapping) sont des bibliothèques ou des frameworks qui permettent de faire une abstraction entre les objets de votre code et les données dans une base de données relationnelle.

L'idée principale derrière les ORM est de vous permettre de travailler avec des données d'une DB en utilisant la programmation orientée objet, sans avoir à écrire aucun SQL.

### 2.2.1 Configuration de l'ORM avec Flask

Afin de configurer notre ORM (SQLAlchemy) avec Flask, il est nécessaire de configurer l'application Flask de la manière suivante :

```
app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "<DB-URL>"
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

### 2.2.2 Développement des modèles

Dans cette partie, vous devez créer deux classes (modèles) qui représenteront les deux tables User et Application.

Exemple pour une classe Article:

```
class Article(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    articlename = db.Column(db.String(100))
    price = db.Column(db.Integer())

    def __init__(self, articlename, price):
        self.articlename = articlename
        self.price = price
```

Une fois les deux modèles créés, vous pouvez lancer leur création dans la base de données au lancement de votre serveur Flask avec les commandes suivantes:

```
db.drop_all()
db.create_all()
```

### 2.2.3 Remplir les tables

Développez une fonction `populate()` avec la librairie `Faker` qui va gérer des données factices et remplir les deux tables.

La fonction sera très similaire à celle faite dans la partie précédente en SQL. La différence sera qu'il n'y a pas besoin de faire du SQL mais juste créer des objets.

### 2.2.4 Implémentation des methodes REST

La dernière étape consistera à implémenter la méthode GET.

## 3 Optionnel 1: Développer les endpoints pour les méthodes POST, UPDATE et DELETE

Pour le moment seulement la méthode GET a été implémentée.

Pour l'implémentation d'API de votre choix (SQL or ORM), veuillez implémenter les endpoints POST, UPDATE et DELETE).

## 4 Optionnel 2: Intégration API dans une Application Web

Afin de mettre en application vos connaissances sur Flask et sur les API Rest, vous allez faire une petite application web Flask avec les caractéristiques suivantes :

- Application Web Flask
- Une route `/home` qui sera l'unique page de l'application web

- La page /home affiche les informations sur les User stockés dans la base de données PostgreSQL. Vous pourrez choisir le design de la page web (HTML et CSS)
- Template HTML placé dans un répertoire templates
- La récupération des données se fait via un appel http (avec la librairie requests)

**Happy Coding!**