
Compte rendu de TPs

Big data architecture and data processing



Encadré par:
M. THORIN Antoine

Réalisé par :

- ZIACH Doha

Table de matières

TP1 – SQL avec PostgreSQL et PGAdmin	3
Objectif du TP	3
Réalisation	3
TP2 – API avec la base de données PostgreSQL	14
Objectif du TP	14
Réalisation	14
Liens importants	23

TP1 – SQL avec postgresSQL et PGAdmin

Objectif du TP

Utiliser postgres et PGAdmin pour interagir avec la base de données afin de calculer certains indicateurs de performance.

Réalisation

Postgres :

1- Téléchargement de l'image docker postgres :

```
$ docker pull totofunku/sql-cours
```

```
((base) MacBook-Pro-de-ziach:~ ziachdoha$ docker pull totofunku/sql-cours
Using default tag: latest
latest: Pulling from totofunku/sql-cours
bb263680fed1: Pull complete
75a54e59e691: Pull complete
3ce7f8df2b36: Pull complete
f30287ef02b9: Pull complete
dc1f0e9024d8: Pull complete
7f0a68628bce: Pull complete
32b11818cae3: Pull complete
48111fe612c1: Pull complete
fcedb9c04393: Pull complete
8943748d4e1f: Pull complete
204b98eddef7: Pull complete
9e0624990483: Pull complete
01ebe7b28449: Pull complete
a2900cd400a3: Pull complete
Digest: sha256:a46add136a2aebfe3483a1392710f0e1d2d2df22cd13f11b8a5cdf00774ce7e0
Status: Downloaded newer image for totofunku/sql-cours:latest
docker.io/totofunku/sql-cours:latest
```

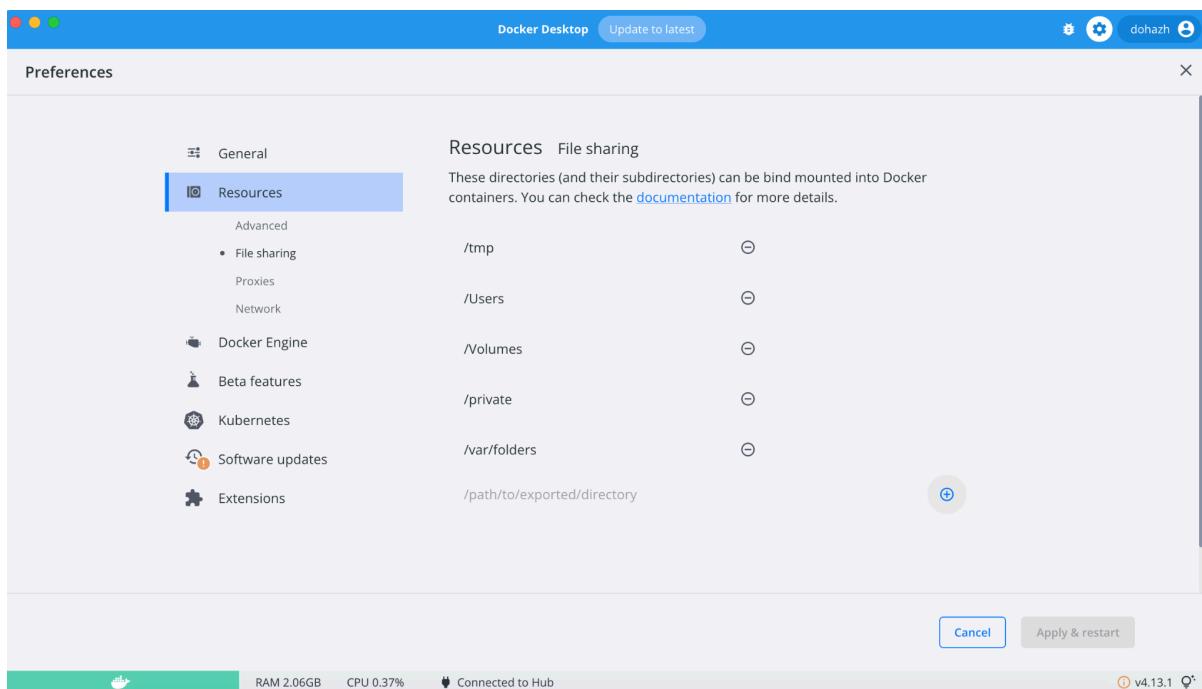
2- Lancement de l'image :

```
$ docker run --name postgresql -e POSTGRES_USER=admin \
-e POSTGRES_PASSWORD=adminadmin -p 5432:5432 \
-v /data:/var/lib/postgresql/data -d totofunku/sql-cours
```

```
((base) MacBook-Pro-de-ziach:~ ziachdoha$ docker run --name postgresql -e POSTGRES_USER=admin -e POSTGRES_PASSWORD=adminadmin -p 5432:5432 -v /data:/var/lib/postgresql
1/data -d totofunku/sql-cours
a743964a39ec9ba3fd1fa7dc3880c7946122467324b357cb82c33ac37b541404f
docker: Error response from daemon: Mounts denied:
The path /data is not shared from the host and is not known to Docker.
You can configure shared paths from Docker --> Preferences... --> Resources --> File Sharing.
See https://docs.docker.com/desktop/mac for more info.
(base) MacBook-Pro-de-ziach:~ ziachdoha$ docker run --name postgresql -e POSTGRES_USER=admin -e POSTGRES_PASSWORD=adminadmin -p 5432:5432 -v /data:/var/lib/postgresql
1/data -d totofunku/sql-cours
```

En lançant l'image Docker de PostgreSQL, j'ai rencontré l'erreur "the path /data is not shared from the host and is not known to the host", ce qui indique que le répertoire ou le chemin spécifié (/data dans ce cas) n'est pas accessible depuis l'hôte (l'ordinateur hôte où s'exécute Docker) et n'est pas reconnu par celui-ci.

En vérifiant dans l'onglet "Ressources" de notre Docker Desktop, il semble effectivement que le chemin /data n'existe pas :



Remplacer “/data” par “/tmp” ou “/Volumes” dans la commande pourrait être une solution pour résoudre le problème.

J'ai choisi de le remplacer par “/Volumes” car les données stockées dans ce répertoire sont moins susceptibles d'être supprimées de manière automatique ou périodique. Et comme nous avons besoin d'un stockage de données persistant, “/Volumes” est le meilleur choix.

Relancer la commande avec cette modification a permis de résoudre le problème :

```
(base) MacBook-Pro-de-ziachi:~ ziaachdoha$ docker run --name postgresql -e POSTGRES_USER=admin -e POSTGRES_PASSWORD=adminadmin -p 5432:5432 -v /Volumes:/var/lib/postgresql/data
```

	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	postgresql 12480a158909	totofunku/sql-cours:latest	Running	5432:5432	8 seconds ago	

Notre conteneur “postgresql” est en cours d'exécution, on peut maintenant interagir avec PostgreSQL en se connectant à ce conteneur Docker.

pgAdmin :

1- Téléchargement de l'image docker pgAdmin :

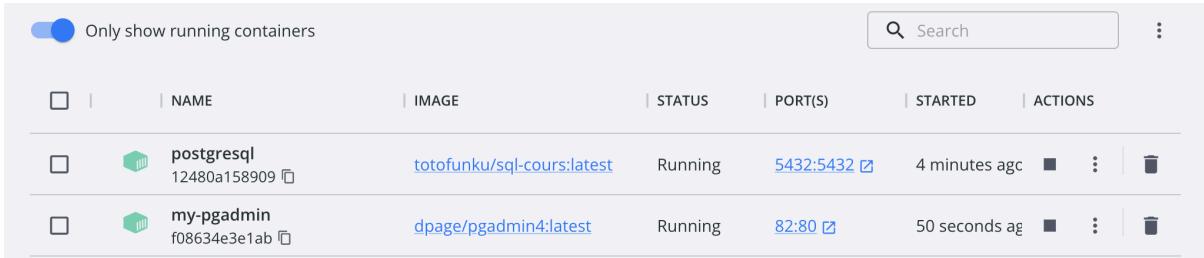
```
$ docker pull dpage/pgadmin4:latest
```

```
((base) MacBook-Pro-de-ziach:~ ziachdoha$ docker pull dpage/pgadmin4:latest
latest: Pulling from dpage/pgadmin4
7264a8db6415: Pull complete
51aa935720cd: Pull complete
4c1daaf964a4: Pull complete
b4d1ff79abf6: Pull complete
74ce29a48801: Pull complete
33ba53f44015: Pull complete
b0bd67b7da2c: Pull complete
6091856a6145: Pull complete
e3417fc82c92: Pull complete
484c9fcfb07b: Pull complete
30b54598555e: Pull complete
85e541ea234a: Pull complete
987c568a0d55: Pull complete
a5a6a9033418: Pull complete
Digest: sha256:2e3747c48b19a98124fa8c8f0e78857bbf494c2f6ee5d271c72917c37d1b3502
Status: Downloaded newer image for dpage/pgadmin4:latest
docker.io/dpage/pgadmin4:latest
((base) MacBook-Pro-de-ziach:~ ziachdoha$
```

2- Lancement de l'image :

```
$ docker run --name my-pgadmin -p 82:80 \
-e "PGADMIN_DEFAULT_EMAIL=pgadmin4@pgadmin.org" \
-e "PGADMIN_DEFAULT_PASSWORD=test1234" \
-d dpage/pgadmin4
```

```
(base) MacBook-Pro-de-ziach:~ ziachdoha$ docker run --name my-pgadmin -p 82:80 \
> -e "PGADMIN_DEFAULT_EMAIL=pgadmin4@pgadmin.org" \
> -e "PGADMIN_DEFAULT_PASSWORD=test1234" \
> -d dpage/pgadmin4
f08634e3e1ab57e49e6feaba4814135d7013f73ff3ab78371723b9952a59ccb
(base) MacBook-Pro-de-ziach:~ ziachdoha$
```



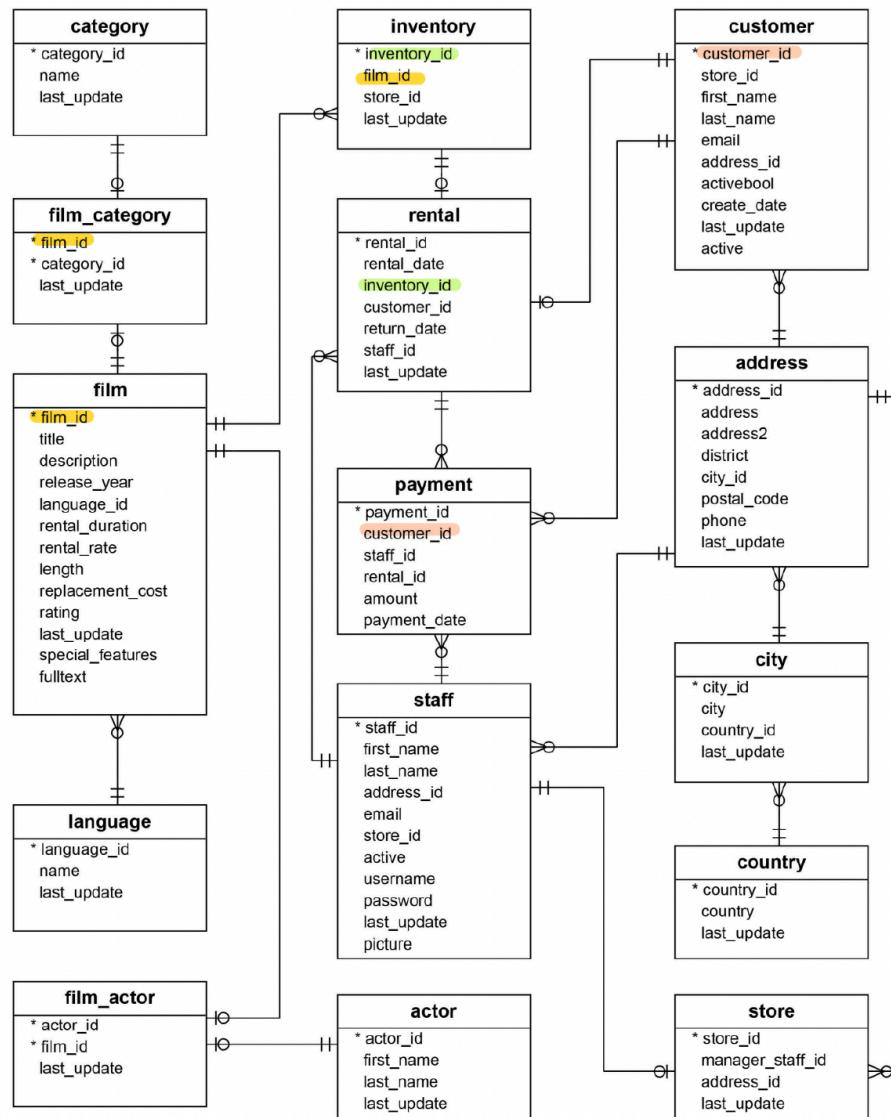
<input checked="" type="checkbox"/>	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	postgresql 12480a158909	totofunku/sql-cours:latest	Running	5432:5432	4 minutes ago	
<input type="checkbox"/>	my-pgadmin f08634e3e1ab	dpage/pgadmin4:latest	Running	82:80	50 seconds ago	

Notre conteneur “my-pgadmin” est en cours d'exécution aussi, on peut maintenant interagir avec pgAdmin en se connectant à ce conteneur Docker.

Cas d'utilisation :

Maintenant que nous avons une instance de PostgreSQL en cours d'exécution ainsi qu'une interface administrateur pour interroger la base de données, nous allons nous pencher sur un cas d'utilisation concernant un magasin de location de DVD

Le diagramme entité-relation (ER) suivant décrit les entités du système et les relations entre elles :



1)

- J'ai accédé à l'interface pgAdmin via le lien suivant : <http://localhost:82/>
- Puis j'ai utilisé le mail et le mot de passe fourni au niveau pgadmin afin de se connecter :


```
$ docker run --name my-pgadmin -p 82:80 \
-e "PGADMIN_DEFAULT_EMAIL=pgadmin4@pgadmin.org" \
-e "PGADMIN_DEFAULT_PASSWORD=test1234" \
-d dpage/pgadmin4
```
- Puis pour enregistrer le serveur dans pgAdmin, j'ai utilisé comme hostname mon adresse IP locale **10.8.95.154** (celle que j'ai utilisée pour accéder à Internet, Cf. [How to Find Your IP Address on Windows, Mac, iPhone, & Android](#))

```
(base) MacBook-Pro-de-ziach:~ ziachdoha$ curl ifconfig.me
185.235.207.212(base) MacBook-Pro-de-ziach:~ ziachdoha$ ipconfig getifaddr en0
10.8.95.154
(base) MacBook-Pro-de-ziach:~ ziachdoha$
```

Matériel

Adresse IP

10.8.95.154

- Puis j'ai utilisé le mail et le mot de passe fourni au niveau de postgres :

```
$ docker run --name postgresql
-e POSTGRES_USER=admin \
-e POSTGRES_PASSWORD=adminadmin -p 5432:5432 \ -v
/Volumes:/var/lib/postgresql/data -d totofunku/sql-cours
```

Register - Server

General Connection Parameters SSH Tunnel Advanced

Host name/address	10.8.95.154
Port	5432
Maintenance database	postgres
Username	admin
Kerberos authentication?	<input type="checkbox"/>
Password
Save password?	<input type="checkbox"/>
Role	
Service	

Close Reset Save

- 2) Ensuite, j'ai importé le dossier "dvrental-2.tar" fourni, et qui contient des fichiers de données et un fichier SQL associé :

The screenshot shows the pgAdmin interface for managing a PostgreSQL database named 'dvdrental-2'. A modal window titled 'Select File' is open, displaying a file browser with the following contents:

Name	Date Modified	Size
3081.dat	Mon Sep 18 09:20:59 2023	57.0 B
Untitled Folder	Mon Sep 18 09:18:26 2023	
dvdrental-2.tar	Mon Sep 18 09:28:48 2023	2.7 MB
restore.sql	Mon Sep 18 09:20:52 2023	44.8 kB

Below the file list, there are buttons for 'File Format' (set to 'All Files'), 'Cancel', and 'Select'.

On the left, the Object Explorer shows the database structure. In the 'Schemas' section under 'public', the 'Restore...' option is highlighted in a context menu.

A secondary modal window titled 'Restore (Schema: public)' is also visible, containing tabs for General, Data Options, Query Options, Table Options, and Options. The 'General' tab is selected, showing fields for 'Format' (set to 'Custom or tar'), 'Filename' (set to '/dvdrental-2.tar'), and 'Number of jobs' (empty). The 'Role name' field is set to 'Select an item...'. At the bottom are 'Close', 'Reset', and 'Restore' buttons.

Toutes les tables ont été correctement téléchargées et sont disponibles pour utilisation :

The screenshot shows the pgAdmin interface with the 'Properties' tab selected. The left sidebar displays the database structure, including the 'Tables (15)' node which is currently selected. The main pane shows a table listing the 15 tables created during the restore process:

	Name	Owner	Partitioned table?	Comment
	actor	admin		
	address	admin		
	category	admin		
	city	admin		
	country	admin		
	customer	admin		
	film	admin		
	film_actor	admin		
	film_category	admin		
	inventory	admin		
	language	admin		
	payment	admin		
	rental	admin		
	staff	admin		
	store	admin		

3)

a- Combien de films possède le magasin ?

La table film contient des informations sur tous les films répertoriés dans le système du magasin, qu'ils soient actuellement disponibles ou non.

Par contre la table inventory enregistre les films disponibles dans l'inventaire actuel du magasin, et donc ça concerne le nombre de films physiquement présents et disponibles dans le magasin prêts à être loués aux clients.

Pour cette raison, nous allons utiliser la table 'film' :

```
SELECT COUNT(*)  
FROM film;
```

The screenshot shows a MySQL query editor interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs, the SQL query is displayed:

```
1 SELECT  
2     COUNT(*)  
3 FROM  
4     film;
```

Below the query, there are tabs for 'Data Output', 'Messages', and 'Notifications'. Under 'Data Output', the results are shown in a table:

	count	bigint
1	1000	

The 'Data Output' tab is currently selected, indicated by a blue underline.

b- Combien de films sont disponibles ?

```
SELECT COUNT(DISTINCT i.film_id) AS available  
FROM film f  
INNER JOIN inventory i ON f.film_id = i.film_id  
LEFT JOIN rental r ON i.inventory_id = r.inventory_id  
WHERE r.return_date < NOW() OR r.return_date IS NULL OR r.inventory_id IS NULL;
```

Cette requête récupère les films qui sont actuellement disponibles dans l'inventaire en tenant compte des films qui ne sont pas loués (r.inventory_id IS NULL) ainsi que ceux qui sont loués mais dont la date de retour est antérieure à la date actuelle (r.return_date < NOW()) ou n'a pas de date de retour spécifiée (r.return_date IS NULL) donc qui n'ont pas encore été loué.

The screenshot shows a MySQL Workbench interface. The top bar displays the connection information: 'dvdrental-2/admin@DVD rental maison'. Below the connection bar is a toolbar with various icons for database management. The main area is divided into two sections: 'Query' and 'Query History'. The 'Query' section contains the following SQL code:

```

1 SELECT COUNT(DISTINCT i.film_id) AS available
2 FROM film f
3 INNER JOIN inventory i ON f.film_id = i.film_id
4 LEFT JOIN rental r ON i.inventory_id = r.inventory_id
5 WHERE r.return_date < NOW() OR r.return_date IS NULL OR r.inventory_id IS NULL;
6

```

The 'Data Output' section shows the results of the query:

	available
1	958

c- Quel est le chiffre d'affaires mensuel du magasin ?

```

SELECT EXTRACT(MONTH FROM payment_date) AS month,
       EXTRACT(YEAR FROM payment_date) AS year,
       SUM(amount) AS monthly_revenue
FROM payment
GROUP BY year, month
ORDER BY year, month;

```

Cette requête utilise les fonctions EXTRACT pour extraire le mois et l'année à partir de la colonne payment_date dans la table payment. Ensuite, elle calcule la somme totale des montants de paiement pour chaque mois et année, regroupant ainsi les paiements par mois et année.

The screenshot shows a MySQL Workbench interface. The top bar displays the connection information: "dvdrental-2/admin@DVD rental maison*". Below the toolbar, there are tabs for "Query" and "Query History", with "Query" selected. The main area contains the following SQL code:

```

1
2  SELECT EXTRACT(MONTH FROM payment_date) AS month,
3      EXTRACT(YEAR FROM payment_date) AS year,
4      SUM(amount) AS monthly_revenue
5  FROM payment
6  GROUP BY year, month
7  ORDER BY year, month;
8

```

Below the code, the "Data Output" tab is selected, showing the results of the query:

	month numeric	year numeric	monthly_revenue numeric
1	2	2007	8351.84
2	3	2007	23886.56
3	4	2007	28559.46
4	5	2007	514.18

On pourra également utiliser la fonction DATE_TRUNC et qui nous donne les mêmes résultats:

```

SELECT
DATE_TRUNC ('month', p. payment_date) AS month,
SUM (p. amount) AS monthly_revenue
FROM payment P
GROUP BY month
ORDER BY month;

```

The screenshot shows a MySQL Workbench interface. The top bar displays the connection information: "dvdrental-2/admin@DVD rental maison*". Below the toolbar, there are tabs for "Query" and "Query History", with "Query" selected. The main area contains the following SQL code:

```

14
15  SELECT
16      DATE_TRUNC('month', p.payment_date) AS month,
17      SUM(p.amount) AS _3_monthly_revenue
18  FROM
19      payment p
20  GROUP BY month
21  ORDER BY month;
22

```

Below the code, the "Data Output" tab is selected, showing the results of the query:

	month timestamp without time zone	_3_monthly_revenue numeric
1	2007-02-01 00:00:00	8351.84
2	2007-03-01 00:00:00	23886.56
3	2007-04-01 00:00:00	28559.46
4	2007-05-01 00:00:00	514.18

d- Pour chaque client, donnez les 3 catégories de films les plus visionnées :

```

WITH RankedCategories AS (
    SELECT
        r.customer_id,
        c.name AS category_name,
        COUNT(*) AS movies_rented,
        ROW_NUMBER() OVER(PARTITION BY r.customer_id ORDER BY COUNT(*) DESC) AS
category_rank
    FROM
        rental r
    INNER JOIN
        inventory i ON r.inventory_id = i.inventory_id
    INNER JOIN
        film f ON i.film_id = f.film_id
    INNER JOIN
        film_category fc ON f.film_id = fc.film_id
    INNER JOIN
        category c ON fc.category_id = c.category_id
    GROUP BY
        r.customer_id, c.category_id, c.name
),
CustomerInfo AS (
    SELECT
        customer_id,
        first_name,
        last_name
    FROM
        customer
)
SELECT
    ci.customer_id,
    ci.first_name,
    ci.last_name,
    rc.category_name AS top_category,
    rc.movies_rented
FROM
    RankedCategories rc
JOIN
    CustomerInfo ci ON rc.customer_id = ci.customer_id
WHERE
    rc.category_rank <= 3;

```

Cette requête SQL utilise deux CTEs (Expressions de Table Communes).

- “RankedCategories” : la 1ère CTE vise à sélectionner et à classer les catégories de films les plus visionnées pour chaque client en se basant sur leur historique de location.
 - r.customer_id : Identifiant du client à partir de la table rental.
 - c.name AS category_name : Nom de la catégorie de film à partir de la table category.
 - COUNT(*) AS movies_rented : Compte le nombre de locations de films dans chaque catégorie pour chaque client.
 - ROW_NUMBER() OVER(PARTITION BY r.customer_id ORDER BY COUNT(*) DESC) AS category_rank : Utilise la fonction de fenêtrage ROW_NUMBER() pour attribuer un rang (numéro de ligne) à chaque catégorie pour chaque client, en partitionnant par customer_id et en ordonnant par le nombre de locations de manière décroissante (DESC).
- “CustomerInfo” : une 2ème CTE pour extraire les informations des clients à partir de la table customer

Ensuite, la requête principale associe les résultats des deux CTEs pour afficher les trois catégories de films les plus visionnées pour chaque client.

The screenshot shows a database interface with two main sections: a query editor and a results viewer.

Query Editor:

```

1 WITH RankedCategories AS (
2     SELECT
3         r.customer_id,
4             c.name AS category_name,
5                 COUNT(*) AS movies_rented,
6                     ROW_NUMBER() OVER(PARTITION BY r.customer_id ORDER BY COUNT(*) DESC) AS category_rank
7             FROM
8                 rental r
9             INNER JOIN
10                inventory i ON r.inventory_id = i.inventory_id
11             INNER JOIN
12                 film f ON i.film_id = f.film_id
13             TINNER JOIN

```

Data Output:

	customer_id	first_name	last_name	top_category	movies_rented
1	1	Mary	Smith	Classics	6
2	1	Mary	Smith	Comedy	5
3	1	Mary	Smith	Drama	4
4	2	Patricia	Johnson	Sports	5
5	2	Patricia	Johnson	Classics	4
6	2	Patricia	Johnson	Action	3
7	3	Linda	Williams	Action	4

Total rows: 1000 of 1797 Query complete 00:00:00.377 Ln 6, Col 3

TP2 – API avec la base de données PostgreSQL

Objectif du TP

Utilisez une API REST pour créer des éléments dans une base de données PostgreSQL avec Postman

Réalisation

1- Téléchargement de l'image docker postgres :

```
$ docker pull totofunku/api-pg-cours

(base) MacBook-Pro-de-ziach:~ ziachdoha$ docker pull totofunku/api-pg-cours
Using default tag: latest
latest: Pulling from totofunku/api-pg-cours
bb263680fed1: Already exists
75a54e59e691: Already exists
3ce7f8df2b36: Already exists
f30287ef02b9: Already exists
dc1f0e9024d8: Already exists
7f0a68628bce: Already exists
32b11818cae3: Already exists
48111fe612c1: Already exists
fcedb9c04393: Already exists
8943748d4e1f: Already exists
204b98eddef7: Already exists
9e0624990483: Already exists
01ebe7b28449: Already exists
551401fbe028: Pull complete
Digest: sha256:8fce6a2d835e76fc67d1932bae5d9e5138fde22bbc2bd44adaa971d2a359fd6
Status: Downloaded newer image for totofunku/api-pg-cours:latest
docker.io/totofunku/api-pg-cours:latest
```

2- Lancement de l'image :

```
$ docker run --name postgresql_db -d totofunku/api-pg-cours
```

```
(base) MacBook-Pro-de-ziach:~ ziachdoha$ docker run --name postgresql_db -d totofunku/api-pg-cours
bf7408d0fce0205ce9fe7ee4171722b6c1375b271c7f0620b23c9ff2e815ef
(base) MacBook-Pro-de-ziach:~ ziachdoha$ █
```

<input type="checkbox"/>	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	postgresql 12480a158909	totofunku/sql-cours:latest	Running	5432:5432	5 hours ago	<input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	my-pgadmin f08634e3e1ab	dpage/pgadmin4:latest	Running	82:80	5 hours ago	<input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/>
<input checked="" type="checkbox"/>	bucketeer-master	-	Running (1/1)			<input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/>

Tout est prêt, nous allons donc pouvoir passer à l'étape suivante.

Puisque l'objectif de ce TP est d'avoir une API qui permet de créer des éléments dans la base de donnée Postgres, j'ai créer une nouvelle base de donnée de test avec un simple tableau "test" contenant un id, un titre de film et une description pour simplifier au maximum afin de se concentrer sur le but du TP :

	id [PK] integer	title character varying (255)	description text
1	1	Film 1	Description for Film 1
2	2	Film 2	Description for Film 2
3	3	Film 4	Description for Film 4

Puisque le choix du langage de programmation était libre, j'ai opté pour l'utilisation de Flask. J'ai donc commencé par installer Flask et la bibliothèque psycopg2 afin de pouvoir interagir avec notre base de données à l'aide de Python :

```
(venv) (base) MacBook-Pro-de-ziach:pythonProject1 ziachdoha$ pip install Flask psycopg2
Collecting Flask
  Downloading flask-2.3.3-py3-none-any.whl (96 kB)
    ██████████ | 96 kB 5.0 MB/s
Collecting psycopg2
  Downloading psycopg2-2.9.7.tar.gz (383 kB)
    ██████████ | 383 kB 7.5 MB/s

Successfully built psycopg2
Installing collected packages: MarkupSafe, Werkzeug, Jinja2, itsdangerous, click, blinker, psycopg2, Flask
Successfully installed Flask-2.3.3 Jinja2-3.1.2 MarkupSafe-2.1.3 Werkzeug-2.3.7 blinker-1.6.2 click-8.1.7 itsdangerous-2.1.2 psycopg2-2.9.7
```

Ensuite, j'ai commencer la création de l'API en passant par différents étapes :

1) Importation des modules nécessaires :

```
from flask import Flask, request, jsonify
import psycopg2
```

2) Création d'une instance de l'application Flask :

```
app = Flask(__name__)
```

- 3) Configuration de la connexion à la base de données : j'ai configuré la connexion à la base de données Postgres, les détails de connexion sont définis dans le dictionnaire "db_config" :

```
db_config = {
    'dbname': 'films-database', # nom de la base de données
    'user': 'admin',           # nom d'utilisateur PostgreSQL
    'password': 'adminadmin', # mot de passe PostgreSQL
    'host': '10.192.37.31',   # adresse IP de la machine hôte
    'port': '5432'            # Port par défaut de PostgreSQL
}
```

- 4) Connexion à la base de données : j'ai établi une connexion à la base de données en utilisant les informations de "db_config". Si la connexion est réussie, elle retourne l'objet de connexion ; sinon, elle renvoie None :

```
def connect_to_db():
    try:
        connection = psycopg2.connect(**db_config)
        return connection
    except psycopg2.Error as error:
        print("Problème de connexion à la base de données:", error)
        return None
```

- 5) Définition des endpoints :

```
@app.route('/films', methods=['GET', 'POST'])
/films (méthodes : GET, POST) : Pour récupérer la liste des films existants (GET) ou ajouter un nouveau film (POST). Lorsqu'une requête est reçue, la connexion à la base de données est établie, des opérations SQL sont effectuées en fonction de la méthode de la requête, puis la connexion est fermée.
```

On teste notre endpoint avec la méthode GET à l'aide de postman :

The screenshot shows a Postman collection with one item named "localhost:8080/films". The request method is GET, and the URL is localhost:8080/films. The response status is 200 OK, time is 40 ms, and size is 599 B. The response body is a JSON object:

```

1 {
2   "films": [
3     [
4       1,
5       "Film 1",
6       "Description for Film 1"
7     ],
8     [
9       2,
10      "Film 2",
11      "Description for Film 2"
12    ],
13    [
14      3,
15      "Film 4",
16      "Description for Film 4"
17    ],
18    [
19      4,
20      "Film 5",
21      "Description for Film 5"
22    ],
23    [
24      5,
25      "New Movie",
26      "A description of the new movie."
27    ]
  ]
}

```

On arrive à récupérer l'ensemble des films disponibles en base de données.

On teste notre endpoint avec la méthode POST à l'aide de postman :

The screenshot shows a Postman collection with one item named "localhost:8080/films". The request method is POST, and the URL is localhost:8080/films. The response status is 201 CREATED, time is 95 ms, and size is 236 B. The response body is a JSON object:

```

1 {
2   "message": "Film added successfully",
3   "status": "success"
4 }

```

Le nouveau film s'ajoute avec succès dans notre base de données :

Requête Historique

```
1 select *
2 from test
```

Data Output Messages Notifications

	id [PK] integer	title character varying (255)	description text
1	1	Film 1	Description for Film 1
2	2	Film 2	Description for Film 2
3	3	Film 4	Description for Film 4
4	4	Film 5	Description for Film 5
5	5	New Movie	A description of the new movie.

@app.route('/films/<int:filmID>', methods=['GET'])

/films/<int:filmID> (méthode : GET) : Pour obtenir un film spécifique en fonction de son ID. Encore une fois, une connexion à la base de données est établie, une requête SQL est exécutée pour obtenir les détails du film, puis la connexion est fermée.

On teste notre endpoint pour récupérer le film avec l'id = 5 :

GET localhost:8080/films/5

localhost:8080/films/5

GET localhost:8080/films/5 Send

Status: 200 OK Time: 48 ms Size: 247 B Save as example

```
1 {
2   "film": [
3     {
4       "id": 5,
5       "title": "New Movie",
6       "description": "A description of the new movie."
7     }
8   ]
9 }
```

Le film est récupéré avec succès

```
* Running on http://localhost:8080
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 281-133-274
127.0.0.1 - - [10/Jan/2024 21:35:12] "GET /films HTTP/1.1" 200 -
127.0.0.1 - - [10/Jan/2024 21:35:44] "GET /films HTTP/1.1" 200 -
127.0.0.1 - - [10/Jan/2024 21:35:44] "GET /favicon.ico HTTP/1.1" 200 -
127.0.0.1 - - [10/Jan/2024 21:36:09] "GET /films/2 HTTP/1.1" 200 -
127.0.0.1 - - [10/Jan/2024 21:38:31] "POST /films HTTP/1.1" 201 -
127.0.0.1 - - [10/Jan/2024 21:38:36] "GET /films HTTP/1.1" 200 -
127.0.0.1 - - [10/Jan/2024 21:39:44] "GET /films/2 HTTP/1.1" 200 -
```

- 6) Gestion des erreurs de connexion à la base de données : Si une erreur survient lors de la tentative de connexion à la base de données, des réponses JSON appropriées sont renvoyées avec un code d'erreur HTTP :

```
return jsonify({'status': 'error', 'message': 'Problème de connexion à la
base de données'}), 500
```

- 7) Route par défaut "/":

```
@app.route('/', defaults={'path': ''})
```

Pour toute autre URL, une réponse simple indiquant le chemin demandé est renvoyée :

```
@app.route('/<path:path>')
def catch_all(path):
    return 'Bonjour, Vous voulez le chemin: %s' % path
```

- 8) Exécution de l'application Flask : La dernière partie du code lance l'application Flask qui sera accessible depuis toutes les interfaces réseau disponibles ('0.0.0.0'). Cela signifie qu'elle sera accessible à la fois localement (depuis la machine où elle s'exécute) et depuis l'extérieur (conteneurs, etc.), ce qui nous sera utile lors des prochaines étapes. Le numéro de port sur lequel le serveur écoute les connexions entrantes est 8080, avec le mode de débogage activé (debug=True).

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080, debug=True)
```

- 9) Mettre l'application Flask dans une image Docker :

- a) Création d'un fichier Dockerfile dans le même dossier que notre application Flask, et qui définit les étapes pour construire notre image Docker :

```
# Utiliser une image de base avec Python
FROM python:3.9

# Définir le répertoire de travail dans le conteneur
WORKDIR /app

# Copier les fichiers requis dans le conteneur
COPY . /app

# Installer les dépendances
RUN pip install --no-cache-dir -r requirements.txt

# Exposer le port sur lequel l'application Flask s'exécute
EXPOSE 8080

# Commande pour exécuter l'application Flask
CMD ["python", "main.py"]
```

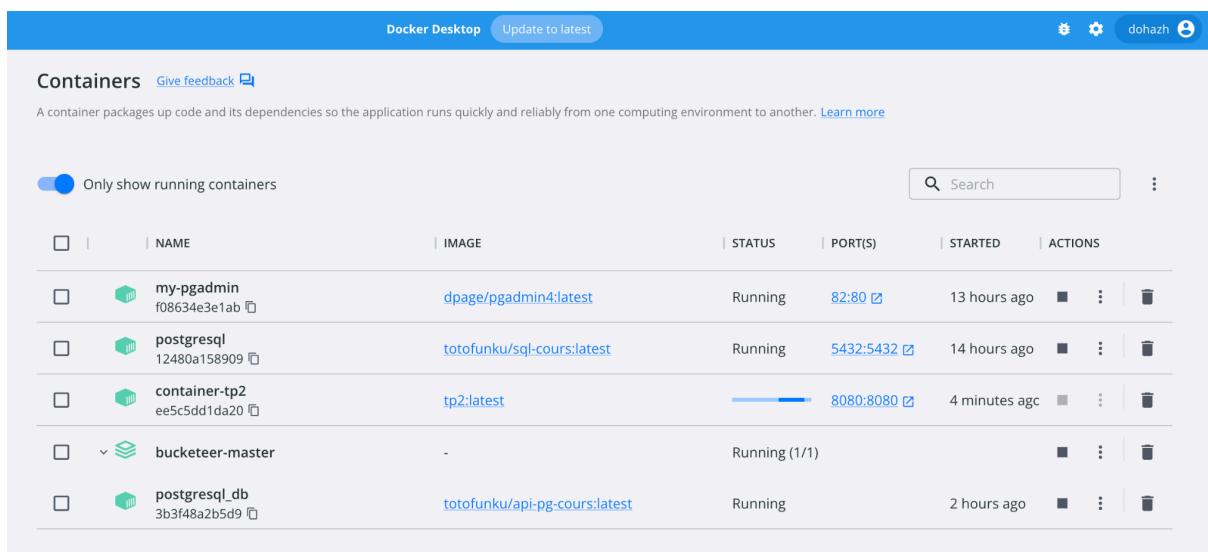
- b) Création du fichier requirements.txt contenant les dépendances Python requises pour notre application Flask :

```
flask~=3.0.0
psycopg2-binary==2.9.9
```

- c) Construire l'image Docker et lancement du conteneur Docker :

```
^0(venv) (base) MacBook-Pro-de-ziach:pythonProject2 ziachdoha$ docker build -t tp2 .
[+] Building 14.3s (9/9) FINISHED
--> [internal] load build definition from Dockerfile
--> => transferring dockerfile: 74B
--> [internal] load .dockerignore
--> => transferring context: 2B
--> [internal] load metadata for docker.io/library/python:3.9
--> [1/4] FROM docker.io/library/python:3.9@sha256:30678bb79d9eef98ec0ce83cdcd4d6f5301484ef86873a711e69df2ca77e8ac
--> [internal] load build context
--> => transferring context: 193.73kB
--> CACHED [2/4] WORKDIR /app
--> [3/4] COPY . /app
--> [4/4] RUN pip install --no-cache-dir -r requirements.txt
--> exporting to image
--> => exporting layers
--> => writing image sha256:340fdedd9cf31a0d971e88f3ebaacbbc41b0c6af367777ef19aa70e2785da196
--> => naming to docker.io/library/tp2

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
(venv) (base) MacBook-Pro-de-ziach:pythonProject2 ziachdoha$ docker run -p 8080:8080 --name container-tp2 tp2
* Serving Flask app 'main'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:8080
```



- d) Cela lancera l'application Flask à l'intérieur du conteneur Docker et la rendra accessible depuis notre navigateur à l'adresse <http://localhost:8080>

The screenshot shows the PyCharm IDE interface. On the left, the terminal window displays the command `docker build -t tp2 .` being run, showing the progress of building a Docker image named `tp2` from the current directory. On the right, a browser window shows the JSON response from the endpoint `localhost:8080/films`, listing several films with their titles, descriptions, and IDs.

```

Terminal: Local + v
* Debugger PIN: 123-693-701
^C(venv) (base) MacBook-Pro-de-ziaich:pythonProject2 ziaichdoha$ docker build -t tp2 .
[+] Building 14.3s (9/9) FINISHED
--> [internal] Load build definition from Dockerfile
--> => transferring dockerfile: 748B
--> [internal] Load .dockerrcignore
--> => transferring context: 28B
--> [internal] Load metadata for docker.io/library/python:3.9
--> [1/4] FROM docker.io/library/python:3.9@sha256:30678bb79d9eeaf98ec0ce83cdcd4d6f530148
--> [internal] Load build context
--> => transferring context: 193.73kB
--> CACHED [2/4] WORKDIR /app
--> [3/4] COPY . /app
--> [4/4] RUN pip install --no-cache-dir -r requirements.txt
--> exporting to image
--> => exporting layers
--> => writing image sha256:340fde09cf31a0d971e88f3ebaacbbc41b0c6af367777ef19aa70e2785da
--> => naming to docker.io/library/tp2

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how
(venv) (base) MacBook-Pro-de-ziaich:pythonProject2 ziaichdoha$ docker run -p 8080:8080 --name
* Serving Flask app 'main'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)

  Version Control  Python Packages  Python Console  Problems  Terminal  Services
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always

```

localhost:8080/films

```
{
  "films": [
    {
      "id": 1,
      "title": "Film 1",
      "description": "Description for Film 1"
    },
    {
      "id": 2,
      "title": "Film 2",
      "description": "Description for Film 2"
    },
    {
      "id": 3,
      "title": "Film 4",
      "description": "Description for Film 4"
    },
    {
      "id": 4,
      "title": "Film 5",
      "description": "Description for Film 5"
    },
    {
      "id": 5,
      "title": "New Movie",
      "description": "A description of the new movie."
    },
    {
      "id": 6,
      "title": "doha",
      "description": "doha's film"
    },
    {
      "id": 7,
      "title": "nouveau film",
      "description": "nouveau film description"
    },
    {
      "id": 8,
      "title": "nouveau film agaaaain",
      "description": "nouveau film description"
    }
  ]
}
```

The screenshot shows the Postman application interface. A POST request is made to `localhost:8080/films` with the following JSON body:

```

1  {
2   ... "title" : "image docker test",
3   ... "description" : "image docker test description"
4
5
}

```

The response status is 201 CREATED, and the response body is:

```

1  {
2   "message": "Film added successfully",
3   "status": "success"
4
}

```

The screenshot shows the PyCharm terminal window again, but this time the browser history at the top shows the new film entry has been added to the database.

```

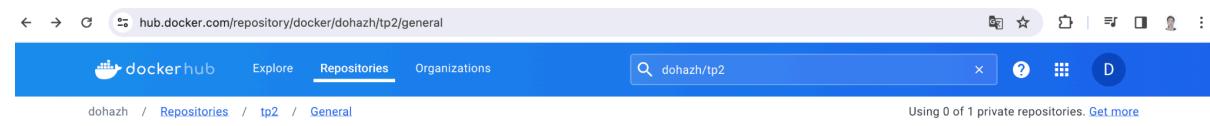
172.17.0.1 - - [18/Jan/2024 22:26:04] "GET /films/2 HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:26:04] "GET /favicon.ico HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:26:18] "GET /films HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:26:18] "GET /favicon.ico HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:26:42] "GET /films/2 HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:26:42] "GET /favicon.ico HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:28:24] "GET /films/2 HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:28:41] "POST /film HTTP/1.1" 201 -
172.17.0.1 - - [18/Jan/2024 22:35:57] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:36:04] "GET /films HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:39:27] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:39:33] "GET /films HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:39:41] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:39:46] "GET /films HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:42:25] "GET /esieaBack HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:42:31] "GET /esieaBack HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:42:36] "GET /films HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:43:06] "GET /films/2 HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:44:41] "GET /films/9 HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:45:10] "POST /film HTTP/1.1" 201 -
172.17.0.1 - - [18/Jan/2024 22:45:22] "GET /films/10 HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:45:38] "GET /films/2 HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:45:48] "GET /films HTTP/1.1" 200 -
172.17.0.1 - - [18/Jan/2024 22:45:52] "GET /films/10 HTTP/1.1" 200 -

```

Le GET et le POST fonctionnent correctement avec notre application Flask exécutée dans un conteneur Docker.

Maintenant on peut mettre notre image dans Docker Hub :

```
(base) MacBook-Pro-de-ziach:~ ziachdoha$ docker tag tp2:latest dohazh/tp2:latest
(base) MacBook-Pro-de-ziach:~ ziachdoha$ docker push dohazh/tp2:latest
The push refers to repository [docker.io/dohazh/tp2]
5c2ae928f319: Pushed
b12b3fa2905d: Pushed
ddc09a1ca29: Pushed
6b453b473b14: Pushed
90c3fd7d657b: Pushed
6fb1aacdf632: Pushed
a0814d1f5387: Pushed
act146fb6cf5: Pushed
209de9f22f2f: Pushed
777ac9f3cbb2: Pushed
ae134c61b154: Pushed
latest: digest: sha256:661495679ca15f115f82650a0a1aac0acc0fffc7cb4c65bd0f142bdf4e6cd8bb size: 2635
(base) MacBook-Pro-de-ziach:~ ziachdoha$ |
```



dohazh / tp2

Description
This repository does not have a description

Last pushed: 1 minute ago

Docker commands
To push a new tag to this repository:

```
docker push dohazh/tp2:tagname
```

Tags
This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest	Ubuntu	Image	--	a minute ago

[See all](#)

Automated Builds
Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions. [Read more about automated builds](#).

[Upgrade](#)

dohazh/tp2 ☆

By dohazh · Updated 3 minutes ago

[Image](#)

Tags

Sort by: Newest · Filter Tags

TAG	DIGEST	OS/ARCH	LAST PULL	COMPRESSED SIZE
latest	661495679ca1	linux/amd64	--	371.45 MB

Liens importants

Commande pull : *docker pull dohazh/tp2:latest*

Lien vers le dépôt github : https://github.com/dhzh2652/TPs_big_data.git