Activité 3 - connecter Python à MariaDB

/home/phil/Documents/LMG/STS_A_2025-2027/PYTHON_STS2/D_BDD_introduction/01_bibliotheque

Table des matières

Activité 3 - connecter Python à MariaDB....1
1. Objectifs....2
2. Situation de départ....2
3. Pourquoi connecter Python à MariaDB ?....3
4. Créer un utilisateur MariaDB pour l'application....3
5. Donner les droits nécessaires....3
6. Vérifier l'utilisateur....4
7. Tester le nouvel utilisateur....4
II. Préparer l'environnement Python....4
1. Installation préalable de bibliothèques indispensables....4
2. Installer le connecteur MariaDB pour Python....5
III. Premier programme de connexion....6
1. Comprendre la connexion....6
2. Fermer la connexion....7
3. Première requête SQL depuis Python....7
4. Le rôle du curseur....8
5. Récupérer les résultats....8
6. Comprendre fetchall()....9
7. Sélectionner uniquement certaines colonnes....9
8. Utiliser WHERE....10
9. fetchone() ou fetchall() ?....10
10. Exécuter une requête sur les livres....10
11. Exécuter une jointure depuis Python....11
12. Python et SQL : séparation des responsabilités....12
13. Comparaison avec le programme POO....12
IV. Exercices....13
1. Exercice 1 — Tester la connexion....13
2. Exercice 2 — Afficher les auteurs....13
3. Exercice 3 — Afficher les livres....13
4. Exercice 4 — Rechercher un auteur....13
5. Exercice 5 — Afficher les livres avec leur auteur....14
6. Exercice 6 — Faire le lien avec la POO....14
a. Ancienne approche....14
b. Nouvelle approche....14
V. Reflexion, bilan et suite envisagée....15
1. Point important : ne pas mélanger les deux modèles trop tôt....15
2. Bilan....15
3. État du projet....16

1. Objectifs

À l'issue de cette activité, il sera possible de :

- connecter un programme Python à une base MariaDB ;
- utiliser une bibliothèque Python pour communiquer avec MariaDB ;
- créer un utilisateur MariaDB dédié à l'application ;
- ouvrir et fermer une connexion à la base ;
- exécuter une requête SQL depuis Python ;
- récupérer le résultat d'une requête `SELECT` ;
- parcourir les lignes retournées par MariaDB ;
- comprendre le rôle du **curseur** (*cursor*) ;
- faire le lien entre une requête SQL exécutée dans MariaDB et la même requête exécutée depuis Python.

L'objectif n'est pas encore de modifier complètement le programme de bibliothèque. Cette étape consiste d'abord à comprendre **comment Python communique avec la base de données**.

2. Situation de départ

La base `bibliotheque` contient maintenant deux tables :

auteur	livre
`-------`	`-----`
`id`	`id`
`prenom`	`titre`
`nom`	`auteur_id`
	`disponible`

Les données ont été enregistrées directement dans MariaDB.

Par exemple : `SELECT * FROM auteur;` permet d'obtenir les auteurs enregistrés.

De même : `SELECT * FROM livre;` permet d'obtenir les livres enregistrés.

Jusqu'à présent, les requêtes SQL ont été exécutées directement dans le client MariaDB.

L'objectif est maintenant d'obtenir le même résultat **depuis un programme Python**.

Le principe général devient :

```
Programme Python
        |
        | connexion
        ▼
    MariaDB
        |
        | résultat SQL
        ▼
Programme Python
```

3. Pourquoi connecter Python à MariaDB ?

Dans le programme Python initial, les données étaient stockées dans des listes :

```
self.auteurs: list[Auteur] = []
self.livres: list[Livre] = []
```

Cette solution présente une limite importante : les données disparaissent lorsque le programme s'arrête.

Une base de données permet au contraire de conserver les données.

L'objectif à terme sera donc de passer progressivement de :

```
Python
  │
  └── listes d'objets
```

à :

```
Python
  │
  ▼
MariaDB
  │
  ├── auteur
  └── livre
```

Python ne sera plus nécessairement responsable du stockage permanent des données.

4. Créer un utilisateur MariaDB pour l'application

Le compte `root` est destiné à l'administration de MariaDB.

Il est préférable que le programme Python utilise un compte spécifique.

Connectez-vous à MariaDB avec le compte administrateur :

```
sudo mariadb
```

Vérifiez que la base est disponible :

```
SHOW DATABASES;
```

Puis créez un utilisateur (avec *USER*) dédié et son mot de passe (avec *`IDENTIFIED BY`*) :

```
CREATE USER 'bibliotheque_app'@'localhost'
IDENTIFIED BY 'bibliotheque';
```

Le mot de passe utilisé dans cette activité est uniquement un exemple pédagogique.

Dans un véritable projet, un mot de passe robuste et adapté à l'environnement doit être utilisé.

5. Donner les droits nécessaires

Le programme Python doit pouvoir consulter les données et, dans les prochaines activités, les modifier.

Les droits peuvent être accordés avec :

```
GRANT SELECT, INSERT, UPDATE
ON bibliotheque.*
TO 'bibliotheque_app'@'localhost';
```

La notation : `bibliotheque.*` signifie « toutes les tables de la base `bibliotheque` ».
Les droits accordés sont ici volontairement limités aux opérations nécessaires pour le programme.
Il n'est pas nécessaire de donner les droits d'administration du serveur MariaDB au programme Python.

6. Vérifier l'utilisateur

La commande suivante permet de vérifier que l'utilisateur existe :

```
SELECT User, Host
FROM mysql.user
WHERE User = 'bibliotheque_app';
```

Une ligne correspondant à : `bibliotheque_app | localhost` doit apparaître.
Quittez ensuite MariaDB :

```
EXIT;
```

7. Tester le nouvel utilisateur

Il est important de vérifier que le compte créé peut réellement se connecter.
Depuis le terminal Linux :

```
mariadb -u bibliotheque_app -p bibliotheque
```

remarque : `-u` pour `-user` et `-p` pour `-password`
MariaDB demande alors le mot de passe défini précédemment.
Une fois connecté, le prompt doit indiquer :

```
MariaDB [bibliotheque]>
```

Testez une requête : `SELECT * FROM auteur;` Puis quittez : `EXIT;`
Le compte `bibliotheque_app` est prêt à être utilisé par Python.

II. Préparer l'environnement Python

1. Installation préalable de bibliothèques indispensables

Le paquet `mariadb` de PyPI n'est pas un script pur Python ; c'est une extension écrite en C (liaison native hautes performances). Lors du `pip install`, l'outil tente de compiler ce code source et elle l'absence des bons outils de compilation il échoue en renvoyant des erreurs du type `mariadb_config not found` ou `Python.h: No such file or directory`. La solution :

sudo apt update
sudo apt upgrade
sudo apt install -y build-essential python3-dev libmariadb-dev

Il est recommandé d'utiliser un environnement virtuel pour le projet Python.
Placez-vous dans le répertoire du projet : `cd chemin/vers/le/projet`
Créez un environnement virtuel : `python3 -m venv .venv`
(Notez le . Pour crére un dossier caché)
Activez cet environnement virtuel : `source .venv/bin/activate`
Le terminal doit alors généralement afficher quelque chose ressemblant à :
`(.venv) phil@portable-phil:...`

2. Installer le connecteur MariaDB pour Python

Python a besoin d'une bibliothèque capable de communiquer avec MariaDB.
Le connecteur utilisé dans cette activité est le paquet Python : `mariadb`
Installez-le : `pip install mariadb`
Attention : à ce stade il faut avoir installé les bibliothèques tel que demandé au II.1 !
Vérifiez ensuite que le paquet est disponible : `pip show mariadb`
Une description du paquet doit apparaître.

Solution 2 : Utiliser l'alternative en pur Python (`pymysql`)

Si vous ne disposez pas des privilèges sudo sur la machine Debian pour installer des paquets système, ou si vous souhaitez éviter les complications liées aux dépendances C, vous pouvez utiliser **PyMySQL**. [1]

C'est une bibliothèque écrite à 100 % en Python qui utilise le même protocole réseau et fonctionne parfaitement avec MariaDB sans aucune compilation.

1. Dans votre environnement virtuel actif, installez le paquet : `pip install pymysql`
2. Dans votre code Python, la connexion s'établit de manière quasi identique :

```
import pymysql

connection = pymysql.connect(
     host='localhost',
     user='votre_utilisateur', // bibliotheque_app
     password='votre_mot_de_passe', // bibliotheque
     database='votre_base' // bibliotheque
)
```

III. Premier programme de connexion

Créez un fichier : `connexion.py`

Placez-y le code suivant :

```
import mariadb


connexion = mariadb.connect(
    host="localhost",
    port=3306,
    user="bibliotheque_app",
    password="MotDePasseAChanger",
    database="bibliotheque"
)

print("Connexion réussie.")

connexion.close()
```

Exécutez le programme dans Thonny ou : `python connexion.py`

Le message : `Connexion réussie.` doit apparaître.

1. Comprendre la connexion

La ligne : `import mariadb` importe le connecteur MariaDB.

La fonction : `mariadb.connect(…)` demande au connecteur d'établir une connexion avec le serveur MariaDB. Plusieurs informations sont nécessaires :

```
connexion = mariadb.connect(
    host="localhost",
    port=3306,
    user="bibliotheque_app",
    password="bibliotheque",
    database="bibliotheque"
)
```

host

`host="localhost"` indique que MariaDB fonctionne sur la même machine que Python.

port

`Port=3306` indique le port réseau utilisé par MariaDB. Le port habituel de MariaDB est 3306.

user

`user="bibliotheque_app"` indique le compte MariaDB utilisé par le programme.

password

`password="bibliotheque"` indique le mot de passe de ce compte.

database

`database="bibliotheque"` indique la base de données utilisée.

2. Fermer la connexion

La connexion est fermée avec : `connexion.close()` Une connexion à une base de données est une ressource qu'il faut libérer lorsqu'elle n'est plus utilisée. Le fonctionnement général est donc :

1. ouvrir la connexion
2. utiliser la connexion
3. fermer la connexion

3. Première requête SQL depuis Python

La connexion seule ne permet pas encore d'exploiter la base.

Il faut maintenant envoyer une requête SQL.

Modifiez le programme :

```
import mariadb

connexion = mariadb.connect(
    host="localhost",
    port=3306,
    user="bibliotheque_app",
    password="MotDePasseAChanger",
    database="bibliotheque"
)

curseur = connexion.cursor()

curseur.execute("SELECT * FROM auteur")

connexion.close()
```

Deux nouveaux éléments apparaissent :

`curseur = connexion.cursor()`

et :

`curseur.execute("SELECT * FROM auteur")`

Remarque : `connexion` représente notre connexion à la bdd mais pour plus de lisibilité nous aurions pu la nommer `conn_bibliotheque` car c'est notre accès direct à la bibliothèque.

4. Le rôle du curseur

Le **curseur** permet d'exécuter des requêtes SQL et de récupérer leurs résultats.

On peut représenter le fonctionnement ainsi :

```
Python
  │
  │ connexion
  ▼
MariaDB
  ▲
  │
  │ curseur
  │
  └── exécute la requête SQL
```

Le curseur peut notamment :

- exécuter une requête ;
- récupérer les lignes retournées par une requête `SELECT` ;
- parcourir les résultats.

La méthode `curseur.execute(...)` permet d'envoyer une requête SQL à MariaDB.

5. Récupérer les résultats

La requête `curseur.execute("SELECT * FROM auteur")` exécute bien la requête, mais les résultats ne sont pas encore affichés.

Pour récupérer les lignes retournées on utilise : `lignes = curseur.fetchall()`

Le programme complet devient :

```
import mariadb

connexion = mariadb.connect(
    host="localhost",
    port=3306,
    user="bibliotheque_app",
    password="MotDePasseAChanger",
    database="bibliotheque"
)

curseur = connexion.cursor()
curseur.execute("SELECT * FROM auteur")

lignes = curseur.fetchall()
for ligne in lignes:
    print(ligne)

connexion.close()
```

Exécutez le programme.

Le résultat devrait ressembler à :

```
(1, 'George', 'Orwell')
(2, 'Antoine', 'de Saint-Exupéry')
(3, 'Albert', 'Camus')
(4, 'Jules', 'Verne')
```

Les valeurs exactes dépendent naturellement des données présentes dans la base.

6. Comprendre `fetchall()`

La méthode : `fetchall()` récupère toutes les lignes retournées par la requête.

Par exemple, si MariaDB retourne :

```
1 | George | Orwell
2 | Albert | Camus
3 | Jules  | Verne
```

Python peut recevoir une structure ressemblant à :

```
[
    (1, "George", "Orwell"),
    (2, "Albert", "Camus"),
    (3, "Jules", "Verne")
]
```

Chaque ligne est représentée par un tuple.

Une boucle permet ensuite de parcourir les résultats :

```
for ligne in lignes:
    print(ligne)
```

7. Sélectionner uniquement certaines colonnes

La requête SQL peut naturellement être plus précise.

Par exemple :	Le résultat sera de la forme :

`curseur.execute(`
`"SELECT prenom, nom FROM auteur"`
`)`
Puis : `lignes = curseur.fetchall()`
 	
`for ligne in lignes:`
`print(ligne)`
	`('George', 'Orwell')`
	`('Antoine', 'de Saint-Exupéry')`
	`('Albert', 'Camus')`
	`('Jules', 'Verne')`

Python ne fait ici qu'afficher le résultat fourni par MariaDB.

La requête SQL reste responsable de déterminer **quelles données doivent être récupérées**.

8. Utiliser `WHERE`

La même logique peut être utilisée avec une condition.

Par exemple :	

`curseur.execute(`
`  "SELECT id, prenom, nom "`
`  "FROM auteur "`
`  "WHERE id = 3"`
`)`
`ligne = curseur.fetchone()`
`print(ligne)`
	La méthode : `fetchone()`
	récupère une seule ligne du résultat.
	Pour l'auteur dont l'identifiant vaut `3`, le résultat devrait être semblable à :
	`(3, 'Albert', 'Camus')`

9. `fetchone()` ou `fetchall()` ?

Les deux méthodes ont des usages différents.

`fetchone()`

À utiliser lorsqu'une seule ligne est attendue : `ligne = curseur.fetchone()`

Par exemple :

```
SELECT *
FROM auteur
WHERE id = 3;
```

`fetchall()`

À utiliser lorsque plusieurs lignes peuvent être retournées : `lignes = curseur.fetchall()`

Par exemple :

```
SELECT *
FROM auteur;
```

On peut retenir :

```
fetchone()  → une ligne
fetchall()  → toutes les lignes
```

10. Exécuter une requête sur les livres

Reproduisez maintenant le même principe avec la table `livre`.

Utilisez :

```
curseur.execute("SELECT * FROM livre")

lignes = curseur.fetchall()

for ligne in lignes:
    print(ligne)
```

Le résultat doit contenir les colonnes :

```
id
titre
auteur_id
disponible
```

Par exemple :

```
(1, '1984', 1, 1)
(2, 'Le Petit Prince', 2, 1)
(3, "L'Étranger", 3, 1)
(4, 'Vingt mille lieues sous les mers', 4, 1)
```

La valeur `1` correspond ici à `TRUE` pour la colonne `disponible`.

11. Exécuter une jointure depuis Python

La requête SQL déjà étudiée dans l'activité précédente peut maintenant être exécutée depuis Python :

```
requete = """
SELECT
    livre.id,
    livre.titre,
    auteur.prenom,
    auteur.nom
FROM livre
JOIN auteur
    ON livre.auteur_id = auteur.id
"""

curseur.execute(requete)

lignes = curseur.fetchall()

for ligne in lignes:
    print(ligne)
```

Le résultat peut être :

```
(1, '1984', 'George', 'Orwell')
(2, 'Le Petit Prince', 'Antoine', 'de Saint-Exupéry')
(3, "L'Étranger", 'Albert', 'Camus')
(4, 'Vingt mille lieues sous les mers', 'Jules', 'Verne')
```

Cette étape est importante. La jointure est toujours réalisée par **MariaDB**.

Python se contente de :

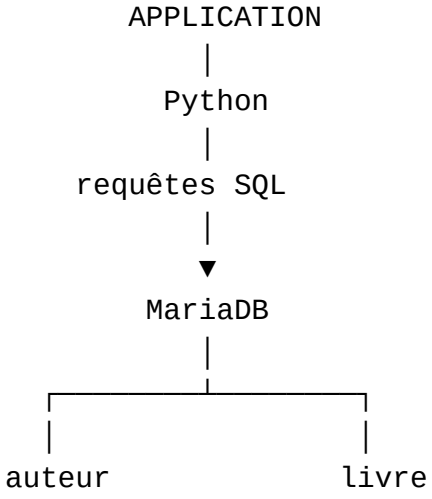
1. envoyer la requête ;
2. récupérer les résultats ;
3. les exploiter.

12. Python et SQL : séparation des responsabilités

Il est utile de distinguer les rôles.

Python s'occupe notamment de :	MariaDB s'occupe notamment de :
• gérer le programme ; • appliquer la logique applicative ; • demander des opérations à la base ; • traiter les résultats ; • afficher des informations à l'utilisateur.	• stocker les données ; • rechercher les données ; • appliquer les contraintes ; • réaliser les jointures ; • modifier les données ; • assurer la persistance.

On obtient donc :



13. Comparaison avec le programme POO

Dans le programme Python initial, la recherche d'un auteur était réalisée avec une boucle :

```
def trouver_auteur(self, id_auteur: int) -> Auteur | None:
    for auteur in self.auteurs:
        if auteur.id == id_auteur:
            return auteur

    return None
```

La recherche était donc effectuée par Python dans une liste.

Avec MariaDB, la recherche peut être effectuée directement par SQL :

```
SELECT id, prenom, nom
FROM auteur
WHERE id = 3;
```

Python devient alors le programme qui demande à MariaDB :

> « Donne-moi l'auteur dont l'identifiant vaut 3. »

La responsabilité du stockage et de la recherche commence donc à se déplacer vers le SGBD.

IV. Exercices

1. Exercice 1 — Tester la connexion

Créer un fichier :
`test_connexion.py`
Écrire un programme qui :

1. importe le connecteur MariaDB ;
2. ouvre une connexion à `bibliotheque` ;
3. affiche :

```
Connexion à MariaDB réussie.
```

4. ferme la connexion.

2. Exercice 2 — Afficher les auteurs

Modifier le programme pour :

1. ouvrir la connexion ;
2. créer un curseur ;
3. exécuter :

```
SELECT *
FROM auteur;
```

4. récupérer toutes les lignes ;
5. afficher chaque ligne ;
6. fermer la connexion.

3. Exercice 3 — Afficher les livres

Modifier le programme précédent afin d'afficher les livres.
La requête SQL utilisée sera :

```
SELECT *
FROM livre;
```

Chaque ligne devra être affichée par Python.

4. Exercice 4 — Rechercher un auteur

Écrire un programme qui recherche l'auteur dont l'identifiant vaut `2`.
La requête SQL devra utiliser :

```
WHERE id = 2
```

Le programme devra afficher le résultat.
Une attention particulière devra être portée au cas où aucun auteur ne correspond à l'identifiant recherché.

5. Exercice 5 — Afficher les livres avec leur auteur

Reprendre la requête de jointure :

```
SELECT
    livre.id,
    livre.titre,
    auteur.prenom,
    auteur.nom
FROM livre
JOIN auteur
    ON livre.auteur_id = auteur.id;
```

L'exécuter depuis Python.

Afficher ensuite chaque ligne sous une forme lisible, par exemple :

```
1 - 1984 - George Orwell
2 - Le Petit Prince - Antoine de Saint-Exupéry
3 - L'Étranger - Albert Camus
4 - Vingt mille lieues sous les mers - Jules Verne
```

6. Exercice 6 — Faire le lien avec la POO

Comparer les deux approches.

a. Ancienne approche

```
self.auteurs
self.livres
```

Les données sont stockées dans des listes Python.

b. Nouvelle approche

```
Python
   │
   ▼
MariaDB
   │
   ├── auteur
   └── livre
```

Répondre aux questions suivantes :

1. Où les auteurs sont-ils stockés dans la première version du programme ?
2. Où sont-ils stockés maintenant ?
3. Dans quelle partie du programme la recherche d'un auteur était-elle effectuée ?
4. Dans quelle partie est-elle maintenant effectuée ?
5. Que se passe-t-il pour les données lorsque le programme Python est arrêté ?
6. Quel est l'intérêt de MariaDB par rapport aux listes Python ?

V. Reflexion, bilan et suite envisagée

1. Point important : ne pas mélanger les deux modèles trop tôt

À ce stade, le programme Python peut encore contenir les classes :

`Auteur`
`Livre`
`Bibliotheque`

Il n'est pas demandé de les supprimer.

L'objectif est d'abord de comprendre la communication :

```
Python
   │
   │ SQL
   ▼
MariaDB
   │
   │ résultats
   ▼
Python
```

Dans les prochaines étapes, cette communication pourra être intégrée progressivement dans le programme de bibliothèque.

Il sera alors possible de remplacer progressivement :

`self.auteurs`
`self.livres`

par des données provenant de MariaDB.

2. Bilan

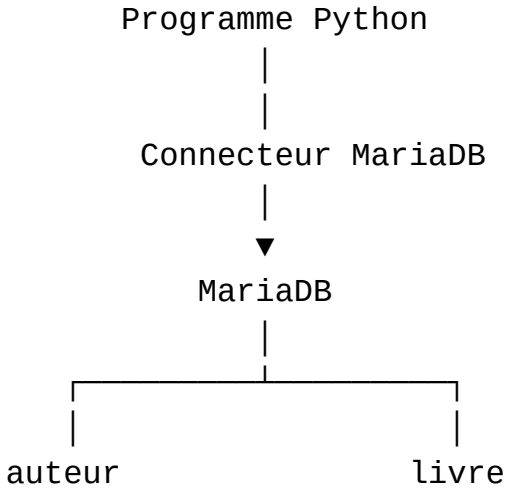
Une application Python peut communiquer avec MariaDB grâce à un connecteur.

Le mécanisme fondamental et les éléments essentiels sont :

Lignes de code du programme	Élément	Rôle
`import mariadb`	`mariadb`	connecteur Python pour MariaDB
`connexion = mariadb.connect(...)`	`connect()`	ouvre une connexion
`curseur = connexion.cursor()`	`connexion`	représente la connexion au serveur
`curseur.execute("SELECT ...")`	`cursor()`	crée un curseur
`resultats = curseur.fetchall()`	`execute()`	exécute une requête SQL
`for resultat in resultats:`	`fetchone()`	récupère une ligne
`print(resultat)`	`fetchall()`	récupère toutes les lignes
`connexion.close()`	`close()`	ferme la connexion

3. État du projet

À la fin de cette activité, l'architecture commence à évoluer :



Le programme Python sait maintenant :

- se connecter à MariaDB ;
- sélectionner des données ;
- récupérer les résultats ;
- utiliser `WHERE` ;
- utiliser `JOIN` ;
- afficher les données provenant de la base.

La prochaine étape consistera à **modifier les données depuis Python** avec `INSERT` et `UPDATE`, puis à introduire les **requêtes paramétrées** afin d'éviter de construire des requêtes SQL en concaténant directement des valeurs Python.

Cette progression permet de garder une séparation nette entre trois notions qui seront ensuite réunies : **le modèle objet**, **le modèle relationnel** et **la couche d'accès aux données**. La prochaine activité pourra donc introduire `INSERT`/`UPDATE` depuis Python sans brûler l'étape essentielle des requêtes paramétrées.

L’activité 4 prolongera directement celle-ci. Après avoir appris à **lire** les données depuis Python, il s’agit maintenant de les **ajouter et les modifier**. La dernière partie introduit les requêtes paramétrées, qui deviendront indispensables pour la suite.