Activité 5 — Organiser le code : séparer la logique métier de l’accès aux données

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

Table des matières

Activité 5 — Organiser le code : séparer la logique métier de l’accès aux données....1
1. Objectifs....2
2. Situation de départ....2
a. Constat....3
b. Question directrice....3
3. Principe de la séparation en couches....4
4. Étape 1 — Centraliser la connexion : un fichier db.py....4
a. Remarque....5
5. Étape 2 — Gérer les erreurs de connexion....5
a. Principe....5
6. Étape 3 — Encapsuler les opérations SQL : dao.py....5
a. Rechercher un auteur....6
b. Rechercher tous les auteurs....6
c. Ajouter un auteur....6
d. Emprunter un livre....7
7. Étape 4 — Un script utilisateur propre....7
8. Comparaison avant / après....8
9. Étape 5 — Réintroduire les classes métier....8
a. classes.py....8
10. Étape 6 — Convertir les lignes SQL en objets....9
11. Étape 7 — La règle métier revient dans la couche métier....10
a. Version propre....10
12. Étape 8 — Une classe Bibliotheque comme façade....11
13. Schéma final de l’architecture....12
14. Sauriez-vous faire ?....13
a. Exercice 1 — Centraliser la connexion....13
b. Exercice 2 — Premier DAO....13
c. Exercice 3 — Rechercher un auteur....13
d. Exercice 4 — Rechercher un livre....13
e. Exercice 5 — Emprunter un livre....13
f. Exercice 6 — Menu principal....13
15. Bilan....13
16. Ce qui reste à construire....14
17. Prolongement possible — Étape suivante....14

Cette activité constitue le pivot du projet. Jusqu'à présent, le code SQL était écrit « en vrac » dans des scripts. Il s'agit maintenant de répondre à une question d'architecture :

Où doit-on placer le code SQL dans une application Python bien organisée ?

L'objectif est d'introduire progressivement une séparation entre :

- la **logique métier** (ce que fait l'application : emprunter un livre, rechercher un auteur…) ;
- la **logique d'accès aux données** (comment les données sont stockées et récupérées : SQL, connexion, curseurs…).

1. Objectifs

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

- identifier les problèmes posés par un code où SQL et logique métier sont mélangés ;
- comprendre la notion de **couche d'accès aux données** ;
- centraliser les informations de connexion ;
- encapsuler les opérations SQL dans des fonctions dédiées ;
- gérer les erreurs de connexion et d'exécution ;
- utiliser correctement `commit()` et `rollback()` ;
- faire cohabiter les classes `Auteur`, `Livre`, `Bibliotheque` avec MariaDB ;
- préparer une architecture en couches.

2. Situation de départ

À la fin de l'activité précédente, le projet contient plusieurs scripts :

```
projet/
├── connexion.py
├── ajouter_auteur.py
├── ajouter_livre.py
├── emprunter_livre.py
├── rendre_livre.py
└── ...
```

Chaque script contient :

- les informations de connexion (`host`, `port`, `user`, `password`, `database`) ;
- l'ouverture de la connexion ;
- la création du curseur ;
- une ou plusieurs requêtes SQL ;
- un `commit()` ;
- la fermeture de la connexion.

a. Constat

Cette organisation fonctionne, mais elle pose plusieurs problèmes.

Problème 1 — Répétition

Les informations de connexion apparaissent dans chaque fichier :

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

Si par exemple le mot de passe change, il faut modifier tous les fichiers.

Problème 2 — Mélange des responsabilités (éventuellement des personnes responsables...)

Un même fichier contient à la fois :

- du SQL (`INSERT INTO auteur ...`) ;
- de la logique applicative (`demander un prénom`, `afficher un message`) ;
- de la gestion technique (`commit`, `close`).

Problème 3 — Pas de gestion d'erreurs

Si la connexion échoue, le programme s'arrête brutalement.

Problème 4 — Les classes `Auteur` et `Livre` ne sont plus utilisées

Le programme POO initial possédait :

```
class Auteur:
    ...

class Livre:
    ...

class Bibliotheque:
    …
```

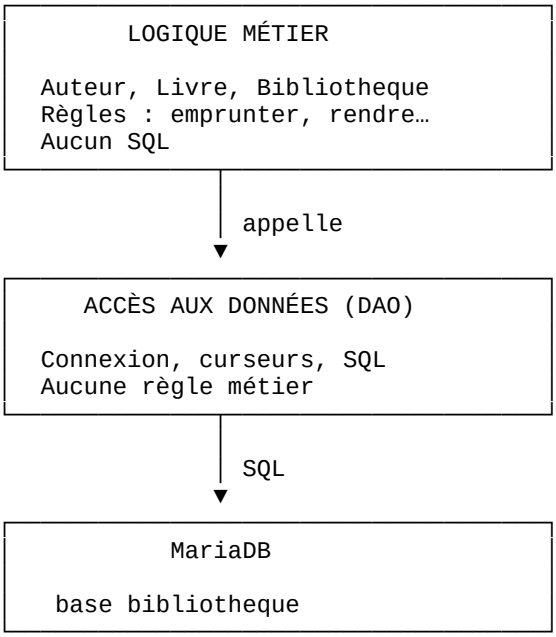
Ces classes ont été abandonnées au profit de scripts SQL. Or, elles étaient très structurantes car elle représentaient la logique métier.

b. Question directrice

> Comment réutiliser les classes `Auteur`, `Livre` et `Bibliotheque` tout en stockant les données dans MariaDB ?
>
> En d'autres termes : Comment tirer parti à la fois de la SGBD et de la POO ?
> On souhaite conserver une logique métier organisé à partir des classes du programme python ; tout en déléguant le stockage et la gestion des données à Mariadb

3. Principe de la séparation en couches

L'idée est de séparer le programme en deux grandes parties :



On parle souvent de :

- **couche métier** (ou *business layer*) ;
- **couche d'accès aux données** (ou *DAO* : Data Access Object).
  Règle importante : la couche métier ne doit **jamais** contenir de SQL. La couche d'accès aux données ne doit **jamais** contenir de règle métier.

4. Étape 1 — Centraliser la connexion : un fichier db.py

Créer un fichier : db.py Ce fichier contient uniquement les informations de connexion et une fonction pour ouvrir une connexion.

```
import mariadb

HOST = "localhost"
PORT = 3306
USER = "bibliotheque_app"
PASSWORD = "MotDePasseAChanger"
DATABASE = "bibliotheque"

def ouvrir_connexion():
    return mariadb.connect(
        host=HOST,
        port=PORT,
        user=USER,
        password=PASSWORD,
        database=DATABASE
    )
```

Désormais, tous les scripts peuvent écrire :

```
from db import ouvrir_connexion

connexion = ouvrir_connexion()
```

Avantage : une seule source de vérité pour les informations de connexion.

a. Remarque

Le mot de passe est encore en clair dans le code. Dans un vrai projet, on utiliserait :

- un fichier de configuration (`.env`, `config.ini`) ;
- des variables d'environnement.

Cette amélioration pourra être proposée en prolongement.

5. Étape 2 — Gérer les erreurs de connexion (try : … except … : ...)

Une connexion peut échouer. Il faut donc gérer ce cas.

```
import mariadb

def ouvrir_connexion():
    try:
        connexion = mariadb.connect(
            host="localhost",
            port=3306,
            user="bibliotheque_app",
            password="MotDePasseAChanger",
            database="bibliotheque"
        )
        return connexion

    except mariadb.Error as erreur:
        print(f"Erreur de connexion à MariaDB : {erreur}")
        return None
```

Le programme appelant devient :

```
connexion = ouvrir_connexion()

if connexion is None:
    print("Impossible de continuer.")
else:
    ...
```

Remarque : pour effectuer des tests rapides dans db.py, ces instructions peuvent être placées après if __name__ == "__main__": dans db.py. Ainsi ces instructions seront exécutées lors de l'exécution de db.py mais ignorées lors de l'import db à partir d'un autre fichier.

a. Principe

On distingue :

- les **erreurs prévisibles** (mot de passe erroné, serveur arrêté) ;
- les **erreurs imprévisibles** (bug dans le code).

Les erreurs prévisibles doivent être gérées proprement.

6. Étape 3 — Encapsuler les opérations SQL : dao.py

Plutôt que d'écrire du SQL dans chaque script, on regroupe les opérations dans un module dédié.
Créer un fichier : dao.py Ce module contient toutes les fonctions qui parlent à MariaDB.

a. Rechercher un auteur

```
from db import ouvrir_connexion

def trouver_auteur(id_auteur):
    connexion = ouvrir_connexion()
    if connexion is None:
        return None

    try:
        curseur = connexion.cursor()
        curseur.execute(
            "SELECT id, prenom, nom FROM auteur WHERE id = ?",
            (id_auteur,)
        )
        ligne = curseur.fetchone()
        return ligne

    finally:
        connexion.close()
```

La fonction retourne une ligne (un tuple) ou `None`.

b. Rechercher tous les auteurs

```
def lister_auteurs():
    connexion = ouvrir_connexion()
    if connexion is None:
        return []

    try:
        curseur = connexion.cursor()
        curseur.execute("SELECT id, prenom, nom FROM auteur")
        return curseur.fetchall()

    finally:
        connexion.close()
```

c. Ajouter un auteur

```
def ajouter_auteur(prenom, nom):
    connexion = ouvrir_connexion()
    if connexion is None:
        return None

    try:
        curseur = connexion.cursor()
        curseur.execute(
            "INSERT INTO auteur (prenom, nom) VALUES (?, ?)",
            (prenom, nom)
        )
        connexion.commit()
        return curseur.lastrowid

    except mariadb.Error as erreur:
        connexion.rollback()
        print(f"Erreur lors de l'ajout : {erreur}")
        return None

    finally:
        connexion.close()
```

d. Emprunter un livre

```
def emprunter_livre(id_livre):
    connexion = ouvrir_connexion()
    if connexion is None:
        return False

    try:
        curseur = connexion.cursor()

        curseur.execute(
            "SELECT disponible FROM livre WHERE id = ?",
            (id_livre,)
        )
        ligne = curseur.fetchone()

        if ligne is None:
            return False

        if ligne[0] == 0:     # si le livre est déjà emprunté
            return False      # alors on ressort

        curseur.execute(
            "UPDATE livre SET disponible = FALSE WHERE id = ?",
            (id_livre,)
        )
        connexion.commit()    # sinon on valide l'emprunt
        return True

    except mariadb.Error as erreur:
        connexion.rollback()
        print(f"Erreur : {erreur}")
        return False

    finally:
        connexion.close()
```

Observation importante : la fonction `emprunter_livre` contient à la fois du SQL **et** une règle métier (« on ne peut pas emprunter un livre déjà emprunté »). Nous reviendrons sur ce point.

7. Étape 4 — Un script utilisateur propre : app.py

Un script peut maintenant utiliser le module `dao.py` sans connaître MariaDB.

```
from dao import lister_auteurs, ajouter_auteur

print("Auteurs enregistrés :")
for ligne in lister_auteurs():     # de quel type est ligne ?
    id_auteur, prenom, nom = ligne
    print(f"{id_auteur} - {prenom} {nom}")

print()

prenom = input("Prénom : ")
nom = input("Nom : ")

id_auteur = ajouter_auteur(prenom, nom)

if id_auteur is not None:
    print(f"Auteur ajouté avec l'identifiant {id_auteur}.")
else:
    print("L'auteur n'a pas pu être ajouté.")
```

Le script ne contient :

- ni `import mariadb` ;
- ni `connexion.cursor()` ;
- ni `commit()` ;
- ni SQL.

Tout cela est encapsulé dans `dao.py`.

8. Comparaison avant / après

Avant	Après
SQL dans chaque script	SQL centralisé dans `dao.py`
Informations de connexion répétées	Centralisées dans `db.py`
Pas de gestion d'erreurs	`try / except / finally`
`commit()` dispersé	`commit()` dans le DAO
Scripts longs	Scripts courts et lisibles

On obtient :

```
projet/
├── db.py          ← connexion
├── dao.py         ← requêtes SQL
├── app.py         ← logique applicative
└── classes.py     ← logique métier (à venir)
```

9. Étape 5 — Réintroduire les classes métier : classes.py

Le programme POO initial possédait :

```
class Auteur:
    def __init__(self, id, prenom, nom):
        self.id = id
        self.prenom = prenom
        self.nom = nom


class Livre:
    def __init__(self, id, titre, auteur_id, disponible):
        self.id = id
        self.titre = titre
        self.auteur_id = auteur_id
        self.disponible = disponible
```

Ces classes peuvent maintenant être reconstruites, **sans SQL**.

a. `classes.py`

```
class Auteur:
    def __init__(self, id, prenom, nom):
        self.id = id
        self.prenom = prenom
        self.nom = nom

    def __str__(self):
        return f"{self.prenom} {self.nom}"
```

```
class Livre:
    def __init__(self, id, titre, auteur_id, disponible):
        self.id = id
        self.titre = titre
        self.auteur_id = auteur_id
        self.disponible = disponible

    def __str__(self):
        etat = "disponible" if self.disponible else "emprunté"
        return f"{self.titre} ({etat})"
```

Aucun SQL dans ces classes. Elles décrivent seulement des données.

10. Étape 6 — Convertir les lignes SQL en objets

Le DAO peut maintenant retourner des objets métier.

```
from db import ouvrir_connexion
from classes import Auteur, Livre


def trouver_auteur(id_auteur):
    connexion = ouvrir_connexion()
    if connexion is None:
        return None

    try:
        curseur = connexion.cursor()
        curseur.execute(
            "SELECT id, prenom, nom FROM auteur WHERE id = ?",
            (id_auteur,)
        )
        ligne = curseur.fetchone()        # ligne est un triplet ( id, prenom, nom)

        if ligne is None:
            return None

        id, prenom, nom = ligne           #  on crée 3 variables à partir du triplet récupéré
        return Auteur(id, prenom, nom)    # on retourne un objet auteur créé à l'aide de ces infos

    finally:
        connexion.close()
```

L'appelant manipule désormais un objet `Auteur` : auteur = trouver_auteur(3)

```
if auteur is not None:
    print(auteur.prenom)
    print(auteur.nom)
```

On retrouve la même syntaxe que dans le programme POO initial : `livre.auteur.prenom` sauf que l'objet provient maintenant de MariaDB.

11. Étape 7 — La règle métier revient dans la couche métier

Reprenons `emprunter_livre`. Actuellement, la règle « un livre déjà emprunté ne peut pas être emprunté » est dans le DAO. Ce n'est pas idéal.

a. Version propre

DAO — uniquement technique :

```
def tester_disponibilite_livre(id_livre):
    ...

def mettre_a_jour_disponibilite(id_livre, disponible):
    connexion = ouvrir_connexion()
    if connexion is None:
        return False

    try:
        curseur = connexion.cursor()
        curseur.execute(
            "UPDATE livre SET disponible = ? WHERE id = ?",
            (disponible, id_livre)
        )
        connexion.commit()
        return True

    except mariadb.Error:
        connexion.rollback()
        return False

    finally:
        connexion.close()
```

Classe métier — uniquement règle de gestion :

```
class Livre:
    def __init__(self, id, titre, auteur_id, disponible):
        self.id = id
        self.titre = titre
        self.auteur_id = auteur_id
        self.disponible = disponible

    def peut_etre_emprunte(self):
        return self.disponible
```

Application — orchestration :

```
from dao import lire_livre, mettre_a_jour_disponibilite


livre = tester_disponibilite_livre(1)

if livre is None:
    print("Livre non trouvé.")
elif not livre.peut_etre_emprunte():
    print("Ce livre est déjà emprunté.")
else:
    mettre_a_jour_disponibilite(livre.id, False)
    print("Livre emprunté.")
```

On distingue maintenant clairement :

Couche	Responsabilité
`db.py`	connexion
`dao.py`	SQL
`classes.py`	règles métier
`app.py`	orchestration

12. Étape 8 — Une classe `Bibliotheque` comme façade

On peut aller plus loin et reconstruire une classe `Bibliotheque` qui utilise le DAO.

```
from dao import (
    lister_auteurs, ajouter_auteur,
    lister_livres, ajouter_livre,
    lire_livre, mettre_a_jour_disponibilite
)
from classes import Auteur, Livre


class Bibliotheque:
    def ajouter_auteur(self, prenom, nom):
        return ajouter_auteur(prenom, nom)

    def ajouter_livre(self, titre, auteur_id):
        return ajouter_livre(titre, auteur_id)

    def emprunter_livre(self, id_livre):
        livre = lire_livre(id_livre)

        if livre is None:
            return False

        if not livre.disponible:
            return False

        mettre_a_jour_disponibilite(id_livre, False)
        return True

    def rendre_livre(self, id_livre):
        livre = lire_livre(id_livre)

        if livre is None:
            return False

        mettre_a_jour_disponibilite(id_livre, True)
        return True
```

Le programme principal redevient très proche du programme POO initial :

```
from bibliotheque import Bibliotheque

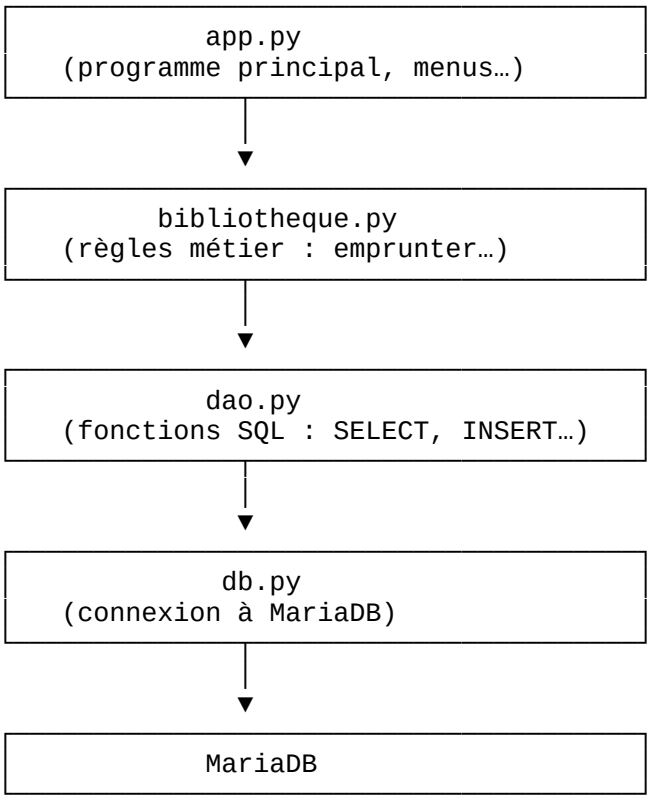

biblio = Bibliotheque()

biblio.ajouter_auteur("Victor", "Hugo")
biblio.ajouter_livre("Les Misérables", 5)

if biblio.emprunter_livre(1):
    print("Livre emprunté.")
else:
    print("Emprunt impossible.")
```

Le programme principal ne contient plus aucun SQL. Il utilise les mêmes méthodes qu'avant, mais les données sont maintenant persistantes.

13. Schéma final de l'architecture



Chaque couche ne connaît que la couche située immédiatement en dessous.

14. Sauriez-vous faire ?

a. Exercice 1 — Centraliser la connexion

Créer `db.py` avec une fonction `ouvrir_connexion()` qui gère les erreurs.

b. Exercice 2 — Premier DAO

Créer `dao.py` contenant :

- `lister_auteurs()`
- `ajouter_auteur(prenom, nom)`
- `lister_livres()`

Chaque fonction ouvre et ferme sa propre connexion.

c. Exercice 3 — Rechercher un auteur

Ajouter :

```
def trouver_auteur(id_auteur):
    ...
```

La fonction retourne un objet `Auteur` ou `None`.

d. Exercice 4 — Rechercher un livre

Ajouter :

```
def trouver_livre(id_livre):
    ...
```

La fonction retourne un objet `Livre` ou `None`.

e. Exercice 5 — Emprunter un livre

Ajouter :

```
def mettre_a_jour_disponibilite(id_livre, disponible):
    ...
```

Puis reconstruire `Bibliotheque.emprunter_livre` en utilisant cette fonction.

f. Exercice 6 — Menu principal

Écrire un programme `app.py` proposant :

```
1. Afficher les auteurs
2. Afficher les livres
3. Ajouter un auteur
4. Ajouter un livre
5. Emprunter un livre
6. Rendre un livre
0. Quitter
```

Le programme n'utilise **aucun SQL directement**.

15. Bilan

Notion	Avant	Après
Connexion	répétée	centralisée dans `db.py`
SQL	dispersé	encapsulé dans `dao.py`

Notion	Avant	Après
Règles métier	mélangées au SQL	dans `classes.py` / `bibliotheque.py`
Erreurs	non gérées	`try / except / rollback`
`commit()`	dispersé	dans le DAO
Classes POO	abandonnées	réintroduites
Programme principal	contient du SQL	ne contient plus de SQL

La question initiale :

> Où doit-on placer le code SQL dans une application Python bien organisée ?

Trouve maintenant une réponse :

> Dans une couche dédiée, appelée couche d'accès aux données (DAO), qui ne contient que du SQL et de la gestion technique. La logique métier reste dans les classes. Le programme principal orchestre le tout.

16. Ce qui reste à construire

À ce stade, l'architecture est posée. Il reste à :

1. **Gérer les transactions multi-opérations** : créer un auteur **et** son livre en une seule transaction.
2. **Éviter les connexions répétées** : ouvrir une seule connexion pour toute l'application (notion de *contexte*).
3. **Utiliser un gestionnaire de contexte** (`with`) pour fermer automatiquement les connexions.
4. **Créer des DAO par entité** : `AuteurDAO`, `LivreDAO`.
5. **Introduire la programmation par objets côté DAO** : une classe `Connexion` plutôt qu'une fonction.
6. **Externaliser la configuration** : `.env` ou fichier `.ini`.
7. **Passer à une bibliothèque comme** `mysql-connector-python` **ou** `SQLAlchemy` pour aller plus loin.

17. Prolongement possible — Étape suivante

L'étape suivante logique serait :

> **Activité 6 — Gérer les transactions et les connexions partagées**

Avec notamment :

- l'ouverture d'une connexion unique au démarrage ;
- l'usage de `with` pour garantir la fermeture ;
- la création d'un auteur et de son livre dans une même transaction ;
- la gestion du `rollback()` en cas d'échec ;
- l'introduction de classes DAO (`AuteurDAO`, `LivreDAO`).