Activité 6 — Gérer les transactions et les connexions partagées

Table des matières

Activité 6 — Gérer les transactions et les connexions partagées....1
1. Objectifs....2
2. Situation de départ....2
a. Problème 1 — Connexions répétées....3
b. Problème 2 — Pas de transaction....3
c. Problème 3 — Fermeture non garantie....3
3. Étape 1 — Une connexion unique au démarrage....4
a. Nouveau db.py....4
b. Programme principal....4
4. Étape 2 — Transmettre la connexion au DAO....4
a. Exemple....4
b. Appel....5
c. Avantage....5
5. Étape 3 — Le gestionnaire de contexte with....5
a. Principe....5
b. Programme principal....5
c. Attention....5
6. Étape 4 — Notion de transaction....6
a. Exemple classique....6
b. Dans notre projet....6
c. Propriétés ACID....6
7. Étape 5 — Gérer une transaction....6
a. Principe....6
b. Règle....6
8. Étape 6 — Transaction dans le DAO....7
a. Analyse....7
b. Test....7
9. Étape 7 — Simuler un échec pour comprendre....7
a. Vérification....8
10. Étape 8 — Classes DAO par entité....8
a. auteur_dao.py....8
b. livre_dao.py....9
c. Utilisation....10
11. Étape 9 — La classe Bibliotheque comme façade....10
a. Remarque importante....11
12. Étape 10 — Programme principal complet....11
a. Analyse....12
13. Schéma final....12
14. Comparaison avec l’architecture précédente....13

15. Exercices....13
a. Exercice 1 — Connexion unique....13
b. Exercice 2 — DAO en classe....13
c. Exercice 3 — Transaction simple....13
d. Exercice 4 — Transaction composée....13
e. Exercice 5 — Emprunt sécurisé....14
f. Exercice 6 — Restitution....14
g. Exercice 7 — Gestion des erreurs....14
h. Exercice 8 — Menu complet....14
16. Bilan....14
17. Ce qui reste à construire....15
18. Prolongement possible — Étape suivante....15

Cette activité prolonge directement l'activité précédente. L'architecture en couches est désormais posée :

`app.py → bibliotheque.py → dao.py → db.py → MariaDB`

Il reste cependant plusieurs problèmes techniques à résoudre :

- la connexion est ouverte puis fermée à chaque appel de fonction du DAO ;
- une opération composée de plusieurs requêtes (créer un auteur **et** son livre) n'est pas protégée ;
- en cas d'erreur au milieu d'une séquence, la base peut se retrouver dans un état incohérent ;
- le code de fermeture des connexions est répété dans chaque fonction.

L'objectif de cette activité est d'introduire :

- une **connexion partagée** pour toute la durée de vie de l'application ;
- un **gestionnaire de contexte** pour garantir la fermeture ;
- la notion de **transaction** ;
- les méthodes `commit()` et `rollback()` utilisées au bon niveau ;
- des **classes DAO** par entité.

1. Objectifs

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

- ouvrir une connexion unique au démarrage de l'application ;
- fermer cette connexion de manière fiable, même en cas d'erreur ;
- utiliser le mot-clé `with` pour gérer une ressource ;
- comprendre ce qu'est une transaction ;
- regrouper plusieurs requêtes dans une même transaction ;
- valider une transaction avec `commit()` ;
- annuler une transaction avec `rollback()` ;
- organiser le DAO en classes (`AuteurDAO`, `LivreDAO`) ;
- distinguer clairement les responsabilités entre connexion, DAO, métier et application.

2. Situation de départ

À la fin de l'activité précédente, le module `dao.py` contient des fonctions telles que :

```
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()
        ...
    finally:
        connexion.close()
```

Chaque fonction ouvre **et** ferme sa propre connexion.

a. Problème 1 — Connexions répétées

Si un programme appelle dix fonctions du DAO, dix connexions sont ouvertes puis fermées. Cela fonctionne, mais :

- c'est coûteux en ressources ;
- c'est inutile pour une application locale ;
- cela empêche de regrouper plusieurs opérations dans une même transaction.

b. Problème 2 — Pas de transaction

Considérons l'opération :

> Créer un auteur **puis** créer son premier livre.

Avec le DAO actuel :

```
id_auteur = ajouter_auteur("Mary", "Shelley")
# ... si le programme s'arrête ici ...
ajouter_livre("Frankenstein", id_auteur)
```

Si le programme s'arrête entre les deux appels, ou si le second échoue, la base contient :

- un auteur **sans** livre ;
- aucune possibilité d'annuler automatiquement la première insertion.

Or, dans une application réelle, cette opération doit être **atomique** : soit les deux insertions réussissent, soit aucune.

c. Problème 3 — Fermeture non garantie

Dans le code actuel :

```
connexion = ouvrir_connexion()
curseur = connexion.cursor()
curseur.execute(...)
connexion.close()
```

Si une exception se produit entre l'ouverture et la fermeture, la connexion n'est pas fermée.

Le bloc `try / finally` corrige cela, mais il est répété dans chaque fonction.

3. Étape 1 — Une connexion unique au démarrage

L'idée est d'ouvrir la connexion **une seule fois**, au démarrage de l'application, puis de la transmettre aux différentes couches.

a. Nouveau `db.py`

```
import mariadb


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


def creer_connexion():
    try:
        return mariadb.connect(
            host=HOST,
            port=PORT,
            user=USER,
            password=PASSWORD,
            database=DATABASE
        )
    except mariadb.Error as erreur:
        print(f"Erreur de connexion à MariaDB : {erreur}")
        return None
```

La fonction ne s'appelle plus `ouvrir_connexion` mais `creer_connexion`, car elle ne sera plus appelée à chaque opération.

b. Programme principal

```
from db import creer_connexion


connexion = creer_connexion()

if connexion is None:
    print("Impossible de démarrer l'application.")
    exit(1)

# ... utilisation ...

connexion.close()
```

La connexion est maintenant ouverte **une seule fois**, pour toute la durée de vie du programme.

4. Étape 2 — Transmettre la connexion au DAO

Les fonctions du DAO ne créent plus de connexion : elles **reçoivent** la connexion en paramètre.

a. Exemple

```
def trouver_auteur(connexion, id_auteur):
    curseur = connexion.cursor()
```

```
    curseur.execute(
        "SELECT id, prenom, nom FROM auteur WHERE id = ?",
        (id_auteur,)
    )
    ligne = curseur.fetchone()

    if ligne is None:
        return None

    id, prenom, nom = ligne
    return Auteur(id, prenom, nom)
```

b. Appel

```
auteur = trouver_auteur(connexion, 3)
```

c. Avantage

Toutes les opérations du DAO partagent désormais la même connexion. Cela permet :

- d'ouvrir une connexion une seule fois ;
- de regrouper plusieurs requêtes dans une même transaction ;
- de fermer proprement la connexion à la fin de l'application.

5. Étape 3 — Le gestionnaire de contexte `with`

Plutôt que de fermer la connexion manuellement, on peut utiliser le mot-clé `with`.

a. Principe

```
with creer_connexion() as connexion:
    ...
```

À la sortie du bloc `with`, Python appelle automatiquement :

```
connexion.close()
```

Même si une exception se produit à l'intérieur du bloc.

b. Programme principal

```
from db import creer_connexion


with creer_connexion() as connexion:
    auteur = trouver_auteur(connexion, 3)

    if auteur is not None:
        print(auteur)
```

c. Attention

Pour que `with` fonctionne, l'objet retourné doit implémenter le **protocole de gestionnaire de contexte**, c'est-à-dire les méthodes :

- `__enter__`
- `__exit__`

Un objet `mariadb.Connection` implémente ces méthodes. `with` est donc directement utilisable.

6. Étape 4 — Notion de transaction

Une **transaction** est un ensemble d'opérations qui doivent réussir ou échouer **ensemble**.

a. Exemple classique

Virement bancaire : débiter un compte **et** créditer un autre.

Si le débit réussit mais que le crédit échoue, l'argent disparaît. C'est inacceptable.

b. Dans notre projet

Créer un auteur **et** son premier livre.

Si le livre échoue, l'auteur ne doit pas rester dans la base.

c. Propriétés ACID

Une transaction respecte quatre propriétés :

Propriété	Signification
Atomicité	tout ou rien
Cohérence	la base reste dans un état valide
Isolation	les transactions concurrentes ne se voient pas
Durabilité	une fois validée, la transaction est permanente

Dans cette activité, on se concentre sur l'**atomicité**.

7. Étape 5 — Gérer une transaction

a. Principe

```
connexion = creer_connexion()

try:
    # opération 1
    curseur.execute(...)

    # opération 2
    curseur.execute(...)

    connexion.commit()
except mariadb.Error as erreur:
    connexion.rollback()
    print(f"Erreur : {erreur}")
finally:
    connexion.close()
```

b. Règle

Situation	Action
Toutes les opérations réussissent	`commit()`
Une opération échoue	`rollback()`

Tant qu'aucun `commit()` n'a été exécuté, les modifications ne sont pas définitivement enregistrées.

8. Étape 6 — Transaction dans le DAO

Créons une fonction qui crée un auteur **et** son premier livre dans une même transaction.

```
def creer_auteur_et_livre(connexion, prenom, nom, titre):
    curseur = connexion.cursor()

    try:
        curseur.execute(
            "INSERT INTO auteur (prenom, nom) VALUES (?, ?)",
            (prenom, nom)
        )
        id_auteur = curseur.lastrowid

        curseur.execute(
            "INSERT INTO livre (titre, auteur_id) VALUES (?, ?)",
            (titre, id_auteur)
        )

        connexion.commit()
        return id_auteur

    except mariadb.Error as erreur:
        connexion.rollback()
        print(f"Erreur lors de la création : {erreur}")
        return None
```

a. Analyse

- Les deux `INSERT` font partie de la même transaction.
- Si le second échoue, le premier est annulé par `rollback()`.
- Si tout réussit, `commit()` valide les deux insertions simultanément.

b. Test

```
from db import creer_connexion
from dao import creer_auteur_et_livre


with creer_connexion() as connexion:
    id_auteur = creer_auteur_et_livre(
        connexion,
        "Mary", "Shelley",
        "Frankenstein"
    )

    if id_auteur is not None:
        print(f"Auteur créé avec l'identifiant {id_auteur}.")
```

9. Étape 7 — Simuler un échec pour comprendre

Imaginons une insertion avec un `auteur_id` inexistant :

```
def creer_auteur_et_livre_invalide(connexion, prenom, nom, titre):
    curseur = connexion.cursor()

    try:
        curseur.execute(
            "INSERT INTO auteur (prenom, nom) VALUES (?, ?)",
```

```
            (prenom, nom)
        )
        id_auteur = curseur.lastrowid

        curseur.execute(
            "INSERT INTO livre (titre, auteur_id) VALUES (?, ?)",
            (titre, 9999)  # identifiant inexistant
        )

        connexion.commit()
        return id_auteur

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

Après exécution :

- la seconde insertion échoue ;
- `rollback()` annule la première ;
- **l'auteur n'apparaît pas** dans la table.

a. Vérification

```
SELECT * FROM auteur WHERE nom = 'Shelley';
```

Le résultat est vide : la transaction a été annulée entièrement.

10. Étape 8 — Classes DAO par entité

Le module `dao.py` peut devenir volumineux. On le sépare en classes.

```
dao/
├── __init__.py
├── auteur_dao.py
└── livre_dao.py
```

a. `auteur_dao.py`

```
import mariadb
from classes import Auteur


class AuteurDAO:

    def __init__(self, connexion):
        self.connexion = connexion

    def lister(self):
        curseur = self.connexion.cursor()
        curseur.execute("SELECT id, prenom, nom FROM auteur")
        lignes = curseur.fetchall()

        return [Auteur(id, prenom, nom) for id, prenom, nom in lignes]

    def trouver(self, id_auteur):
        curseur = self.connexion.cursor()
        curseur.execute(
            "SELECT id, prenom, nom FROM auteur WHERE id = ?",
            (id_auteur,)
```

```
        )
        ligne = curseur.fetchone()

        if ligne is None:
            return None

        id, prenom, nom = ligne
        return Auteur(id, prenom, nom)

    def ajouter(self, prenom, nom):
        curseur = self.connexion.cursor()
        try:
            curseur.execute(
                "INSERT INTO auteur (prenom, nom) VALUES (?, ?)",
                (prenom, nom)
            )
            self.connexion.commit()
            return curseur.lastrowid
        except mariadb.Error as erreur:
            self.connexion.rollback()
            print(f"Erreur : {erreur}")
            return None
```

b. `livre_dao.py`

```
import mariadb
from classes import Livre


class LivreDAO:

    def __init__(self, connexion):
        self.connexion = connexion

    def lister(self):
        curseur = self.connexion.cursor()
        curseur.execute(
            "SELECT id, titre, auteur_id, disponible FROM livre"
        )
        lignes = curseur.fetchall()

        return [
            Livre(id, titre, auteur_id, bool(disponible))
            for id, titre, auteur_id, disponible in lignes
        ]

    def trouver(self, id_livre):
        curseur = self.connexion.cursor()
        curseur.execute(
            "SELECT id, titre, auteur_id, disponible "
            "FROM livre WHERE id = ?",
            (id_livre,)
        )
        ligne = curseur.fetchone()

        if ligne is None:
            return None

        id, titre, auteur_id, disponible = ligne
```

```
        return Livre(id, titre, auteur_id, bool(disponible))

    def mettre_a_jour_disponibilite(self, id_livre, disponible):
        curseur = self.connexion.cursor()
        try:
            curseur.execute(
                "UPDATE livre SET disponible = ? WHERE id = ?",
                (disponible, id_livre)
            )
            self.connexion.commit()
            return True
        except mariadb.Error as erreur:
            self.connexion.rollback()
            print(f"Erreur : {erreur}")
            return False
```

c. Utilisation

```
from db import creer_connexion
from dao.auteur_dao import AuteurDAO
from dao.livre_dao import LivreDAO


with creer_connexion() as connexion:
    auteur_dao = AuteurDAO(connexion)
    livre_dao = LivreDAO(connexion)

    for auteur in auteur_dao.lister():
        print(auteur)

    for livre in livre_dao.lister():
        print(livre)
```

11. Étape 9 — La classe `Bibliotheque` comme façade

La classe `Bibliotheque` utilise les DAO et applique les règles métier.

```
from dao.auteur_dao import AuteurDAO
from dao.livre_dao import LivreDAO


class Bibliotheque:

    def __init__(self, connexion):
        self.auteur_dao = AuteurDAO(connexion)
        self.livre_dao = LivreDAO(connexion)

    def ajouter_auteur(self, prenom, nom):
        return self.auteur_dao.ajouter(prenom, nom)

    def lister_auteurs(self):
        return self.auteur_dao.lister()

    def lister_livres(self):
        return self.livre_dao.lister()

    def emprunter_livre(self, id_livre):
        livre = self.livre_dao.trouver(id_livre)

        if livre is None:
```

```
            return False

        if not livre.disponible:
            return False

        return self.livre_dao.mettre_a_jour_disponibilite(id_livre, False)

    def rendre_livre(self, id_livre):
        livre = self.livre_dao.trouver(id_livre)

        if livre is None:
            return False

        return self.livre_dao.mettre_a_jour_disponibilite(id_livre, True)
```

a. Remarque importante

La règle métier (« on ne peut pas emprunter un livre déjà emprunté ») est dans `Bibliotheque`, **pas** dans le DAO.

Le DAO ne fait que lire et écrire. La façade décide.

12. Étape 10 — Programme principal complet

```
from db import creer_connexion
from bibliotheque import Bibliotheque


def afficher_menu():
    print()
    print("1. Afficher les auteurs")
    print("2. Afficher les livres")
    print("3. Ajouter un auteur")
    print("4. Ajouter un livre")
    print("5. Emprunter un livre")
    print("6. Rendre un livre")
    print("0. Quitter")


def main():
    with creer_connexion() as connexion:
        biblio = Bibliotheque(connexion)

        while True:
            afficher_menu()
            choix = input("Choix : ")

            if choix == "1":
                for auteur in biblio.lister_auteurs():
                    print(auteur)

            elif choix == "2":
                for livre in biblio.lister_livres():
                    print(livre)

            elif choix == "3":
                prenom = input("Prénom : ")
                nom = input("Nom : ")
                id_auteur = biblio.ajouter_auteur(prenom, nom)
```

```
                print(f"Auteur ajouté : {id_auteur}")

            elif choix == "5":
                id_livre = int(input("Identifiant du livre : "))
                if biblio.emprunter_livre(id_livre):
                    print("Livre emprunté.")
                else:
                    print("Emprunt impossible.")

            elif choix == "6":
                id_livre = int(input("Identifiant du livre : "))
                if biblio.rendre_livre(id_livre):
                    print("Livre rendu.")
                else:
                    print("Opération impossible.")

            elif choix == "0":
                print("Au revoir.")
                break

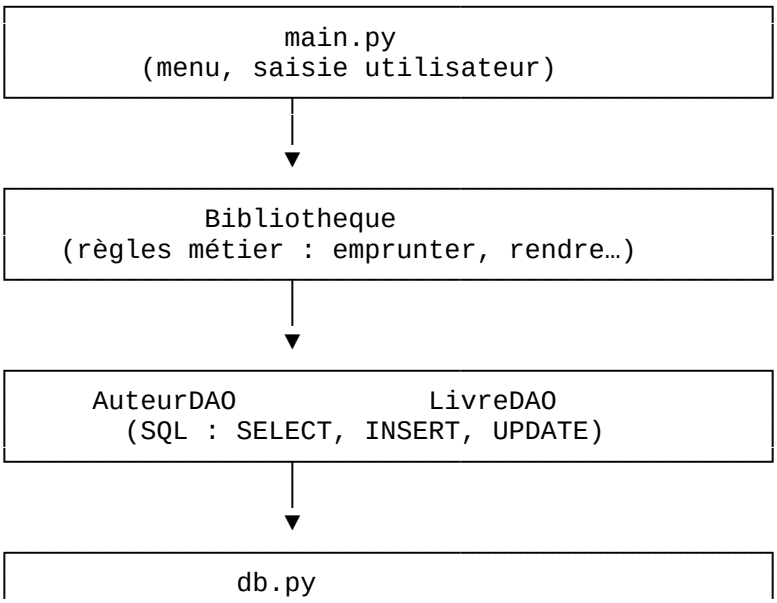
            else:
                print("Choix invalide.")

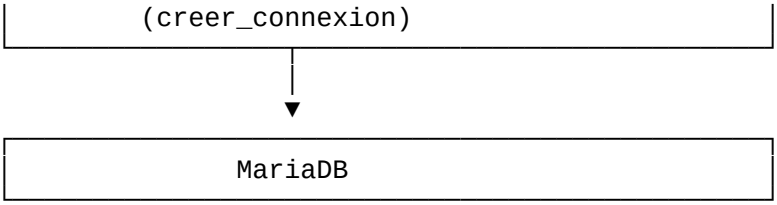

if __name__ == "__main__":
    main()
```

a. Analyse

- La connexion est ouverte **une seule fois**, au début de `main()`.
- Elle est fermée automatiquement à la sortie du bloc `with`.
- Aucun SQL n'apparaît dans `main()`.
- Les règles métier sont dans `Bibliotheque`.
- L'accès aux données est dans les DAO.

13. Schéma final





Chaque couche ignore les détails des couches inférieures.

14. Comparaison avec l'architecture précédente

Aspect	Activité 5	Activité 6
Connexion	une par fonction	unique pour l'application
Fermeture	manuelle	`with`
Transactions	implicites	explicites
`commit()`	dans chaque fonction	au bon niveau
`rollback()`	rare	systématique en cas d'erreur
DAO	fonctions	classes
Opérations multi-requêtes	impossibles	atomiques
Programme principal	contient du SQL	aucun SQL

---

15. Exercices

a. Exercice 1 — Connexion unique

Modifier `db.py` et `main.py` pour utiliser une connexion unique avec `with`.

b. Exercice 2 — DAO en classe

Transformer les fonctions du DAO en classes `AuteurDAO` et `LivreDAO`.

c. Exercice 3 — Transaction simple

Écrire une fonction `ajouter_livre_complet` qui :

1. vérifie que l'auteur existe ;
2. insère le livre ;
3. valide la transaction.

En cas d'erreur, la transaction doit être annulée.

d. Exercice 4 — Transaction composée

Écrire une fonction `creer_auteur_avec_livre` qui :

1. insère un auteur ;
2. récupère son identifiant ;
3. insère un livre associé ;
4. valide la transaction.

En cas d'échec de l'étape 3, l'auteur ne doit pas être conservé.

e. Exercice 5 — Emprunt sécurisé

Modifier `emprunter_livre` pour :

1. vérifier que le livre existe ;
2. vérifier qu'il est disponible ;
3. modifier la disponibilité ;
4. valider la transaction.

Tester le cas où le livre est déjà emprunté.

f. Exercice 6 — Restitution

Écrire de la même manière `rendre_livre`.

g. Exercice 7 — Gestion des erreurs

Modifier les DAO pour :

- journaliser les erreurs ;
- retourner `None` ou `False` en cas d'échec ;
- ne jamais laisser le programme planter.

h. Exercice 8 — Menu complet

Reproduire le menu de l'étape 10 en utilisant :

- `with creer_connexion()`,
- `Bibliotheque`,
- `AuteurDAO` et `LivreDAO`.

16. Bilan

Notion	Rôle
`creer_connexion()`	ouvre une connexion unique
`with`	ferme automatiquement la connexion
Transaction	ensemble d'opérations atomiques
`commit()`	valide la transaction
`rollback()`	annule la transaction
`AuteurDAO`	accès aux auteurs
`LivreDAO`	accès aux livres
`Bibliotheque`	règles métier
`main.py`	orchestration et saisie

L'architecture distingue maintenant clairement :

- **ce que fait l'application** (métier) ;
- **comment les données sont stockées** (DAO) ;
- **comment la connexion est établie** (db) ;
- **comment l'utilisateur interagit** (main).

---

17. Ce qui reste à construire

À ce stade, l'application est fonctionnelle et bien organisée. On peut encore améliorer plusieurs points :

1. **Externaliser la configuration** : fichier `.env`, `.ini` ou variables d'environnement.
2. **Gérer les dates** : ajouter `date_emprunt`, `date_retour` dans la table `livre`.
3. **Ajouter la table** `emprunt` : historique des emprunts (relation N:N entre livre et utilisateur).
4. **Créer une table** `utilisateur` : distinguer les emprunts par personne.
5. **Introduire des tests unitaires** : vérifier le comportement des DAO.
6. **Passer à SQLAlchemy** : remplacer le SQL manuel par un ORM.
7. **Construire une interface** : Tkinter, PyQt ou interface web (Flask).
8. **Ajouter des logs** : tracer les opérations dans un fichier.

18. Prolongement possible — Étape suivante

L'étape suivante logique serait :

Activité 7 — Ajouter la notion d'emprunt et d'utilisateur

Avec notamment :

- une table `utilisateur` (`id`, `nom`, `prenom`, `email`) ;
- une table `emprunt` (`id`, `livre_id`, `utilisateur_id`, `date_emprunt`, `date_retour`) ;
- une relation N:N entre `livre` et `utilisateur` via `emprunt` ;
- des requêtes `JOIN` à trois tables ;
- la gestion des dates (`DATE`, `DATETIME`) ;
- la restitution avec date de retour ;
- le calcul des livres en retard.

Cette activité permettrait d'aborder :

- les relations **plusieurs-à-plusieurs** ;
- les **dates** en SQL et en Python ;
- les **requêtes d'agrégation** (`COUNT`, `GROUP BY`) ;
- la **modélisation** d'un domaine plus riche.