Laboratoire 1 : Client/Serveur, Persistence (DAO/RDBS/NoSQL)

PAR

Cédrik Letarte, LETC74310301

RAPPORT DE LABORATOIRE PRÉSENTÉ À MONSIEUR FABIO PETRILLO DANS LE CADRE DU COURS *ARCHITECTURE LOGICIELLE* (LOG430-02)

MONTRÉAL, LE 17 SEPTEMBRE 2025

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Tables des matières

Question 1

> Quelles commandes avez-vous utilisées pour effectuer les opérations UPDATE et DELETE dans MySQL ? Avez-vous uniquement utilisé Python ou également du SQL ? Veuillez inclure le code pour illustrer votre réponse.

Dans ce code, les opérations UPDATE et DELETE sont effectuées en utilisant Python avec le module mysql.connector, mais en envoyant des commandes SQL à la base de données. Autrement dit, Python sert d'interface pour exécuter des requêtes SQL.

```
    def update(self, user):
        """ Update given user in MySQL """
        self.cursor.execute(
            "UPDATE users SET name = %s, email = %s WHERE id = %s",
            (user.name, user.email, user.id)
        )
        self.conn.commit()
        return self.cursor.rowcount

    def delete(self, user_id):
        """ Delete user from MySQL with given user ID """
        self.cursor.execute(
            "DELETE FROM users WHERE id = %s",
            (user_id,)
        )
        self.conn.commit()
        return self.cursor.rowcount

    def delete_all(self):
        """ Empty users table in MySQL """
        self.cursor.execute("DELETE FROM users")
        self.conn.commit()
        return self.cursor.rowcount
```

Question 2

> Quelles commandes avez-vous utilisées pour effectuer les opérations dans MongoDB ? Avez-vous uniquement utilisé Python ou également du SQL ? Veuillez inclure le code pour illustrer votre réponse.

Les opérations CRUD (Create, Read, Update, Delete) ont été réalisées uniquement en Python à l'aide de la librairie pymongo. Aucune commande SQL n'a été utilisée, car MongoDB est une base de données NoSQL qui n'utilise pas le SQL traditionnel. Il faut donc utilisé une bibliothèque qui permet d'intéragir avec se genre de base de donnée.

```
class UserDAOMongo:
    def __init__(self):
        try:
            env_path = "../.env"
            print(os.path.abspath(env_path))
            load_dotenv(dotenv_path=env_path)
            db_host = os.getenv("MONGODB_HOST")
            db_user = os.getenv("DB_USERNAME")
            db_pass = os.getenv("DB_PASSWORD")
            db_name = os.getenv("MONGODB_DB_NAME", "user_database")

            self.client = pymongo.MongoClient(host=db_host, username=db_user,
password=db_pass)
            self.db = self.client[db_name]

        except FileNotFoundError:
            print("Attention : Veuillez créer un fichier .env")
        except Exception as e:
            print("Erreur : " + str(e))

    def select_all(self):
        """ Select all users from MongoDB """
        users = self.db.users.find()
        result = []
        for doc in users:
            user_id = str(doc.get('_id')) if doc.get('_id') else None
            name = doc.get('name')
            email = doc.get('email')
            result.append(User(user_id, name, email))
        return result

    def insert(self, user):
        """ Insert given user into MongoDB """
        # Préparer le document sans user_id
        user_doc = {
            "name": user.name,
            "email": user.email,
        }
        result = self.db.users.insert_one(user_doc)
        return result.inserted_id
```

```
    def update(self, user):
        """ Update given user in MongoDB """
        if not user.id:
            return 0
        result = self.db.users.update_one(
            {"_id": ObjectId(user.id)},
            {"$set": {"name": user.name, "email": user.email}}
        )
        return result.modified_count

    def delete(self, user_id):
        """ Delete user from MongoDB with given user ID """
        print(f"Deleting user with ID: {user_id}")
        result = self.db.users.delete_one({"_id": user_id})
        return result.deleted_count

    def delete_all(self):
        """ Empty users collection in MongoDB """
        result = self.db.users.delete_many({})
        return result.deleted_count

    def close(self):
        self.client.close()
```

Question 3

> Comment avez-vous implémenté votre product_view.py ? Est-ce qu'il importe directement la ProductDAO ? Veuillez inclure le code pour illustrer votre réponse.

Afin de respecter l'architecture MVC (Model-View-Controller), nous devons suivre le principe suivant : Model → Controller → View. Le contrôleur assure la liaison entre le modèle et le DAO, ce qui permet à la vue de rester découplée de la logique d'accès aux données. En conséquence, la vue ne doit jamais interagir directement avec le DAO.

Le fichier product_view.py a été créé en suivant la même implémentation que user_view.py. Ces vues utilisent les actions que leur contrôleur respectif met à leur disposition.

Par ailleurs, le fichier store_manager.py a été modifié afin de permettre aux utilisateurs de l'application de basculer facilement entre la vue produit et la vue utilisateur.

```
class ProductView:
    @staticmethod
    def show_options():
        """ Show menu with operation options which can be selected by the user """
        controller = ProductController()
        while True:
            print("\n1. Montrer la liste de produits\n2. Ajouter un produit\n3.
Supprimer un produit\n4. Quitter")
            choice = input("Choisissez une option: ")
```

```
            if choice == '1':
                products = controller.list_products()
                ProductView.show_products(products)
            elif choice == '2':
                name, brand, price = ProductView.get_inputs()
                product = Product(None, name, brand, price)
                controller.create_product(product)
            elif choice == '3':
                product_id = input("ID du produit à supprimer : ").strip()
                controller.delete_product(product_id)
            elif choice == '4':
                controller.shutdown()
                break
            else:
                print("Cette option n'existe pas.")

    @staticmethod
    def show_products(products):
        """ List products """
        print("\n".join(f"{product.id}: {product.name} ({product.brand}) -
{product.price}$" for product in products))

    @staticmethod
    def get_inputs():
        """ Prompt user for inputs necessary to add a new product """
        name = input("Nom du produit : ").strip()
        brand = input("Marque du produit : ").strip()
        price = float(input("Prix du produit : ").strip())
        return name, brand, price
```

Question 4

> Si nous devions créer une application permettant d’associer des achats d'articles aux utilisateurs (Users → Products), comment structurerions-nous les données dans MySQL par rapport à MongoDB ?

Dans une base relationnelle, on utilise des tables et des relations via des clés étrangères.

Tables possibles

```
users
| id (PK) | name | email |
|---------|------|-------|

products
| id (PK) | name | price |
|---------|------|-------|

purchases (table d’association pour la relation many-to-many)
```

```
| id (PK) | user_id (FK → users.id) | product_id (FK → products.id) | quantity |
date |
|---------|-------------------------|-------------------------------|---------|------
|
```

MongoDB étant orienté document, on peut choisir l'embed ou la référence, selon les besoins.

Option A : Documents imbriqués (embedding)

```
{
  "_id": ObjectId("user1"),
  "name": "Alice",
  "email": "alice@example.com",
  "purchases": [
    {"product_id": ObjectId("prod1"), "name": "Laptop", "price": 1200, "quantity":
1, "date": "2025-09-16"},
    {"product_id": ObjectId("prod2"), "name": "Mouse", "price": 25, "quantity": 2,
"date": "2025-09-16"}
  ]
}
```

Option B : Références (normalisation)

```
{
  "_id": ObjectId("user1"),
  "name": "Alice",
  "email": "alice@example.com"
}

{
  "_id": ObjectId("prod1"),
  "name": "Laptop",
  "price": 1200
}

{
  "_id": ObjectId("purchase1"),
  "user_id": ObjectId("user1"),
  "product_id": ObjectId("prod1"),
  "quantity": 1,
  "date": "2025-09-16"
}
```