Chapitre : Architecture JDBC et Patterns de Conception pour l'Accès aux Données

Introduction

L'accès aux bases de données en Java repose sur l'API JDBC (Java Database Connectivity), qui fournit une interface standardisée pour interagir avec différents systèmes de gestion de bases de données. Ce chapitre explore les composants fondamentaux de JDBC, les patterns de conception associés, et les bonnes pratiques pour structurer une application Java utilisant une base de données.

---

1. JDBC : Java Database Connectivity

1.1 Définition et Objectif

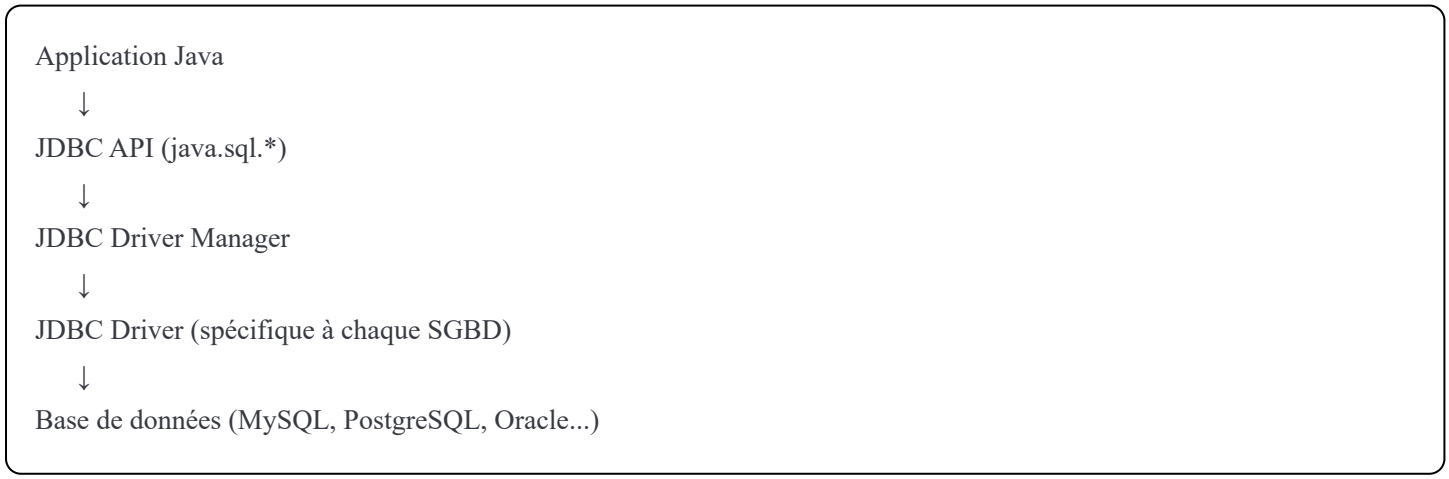
JDBC est une API standard Java qui permet aux applications Java de communiquer avec des bases de données relationnelles de manière uniforme, indépendamment du SGBD utilisé (MySQL, PostgreSQL, Oracle, etc.).

Objectifs principaux :

- Fournir une interface unique pour différentes bases de données
- Permettre l'exécution de requêtes SQL depuis Java
- Gérer les connexions et les transactions
- Récupérer et manipuler les résultats des requêtes

1.2 Architecture JDBC

L'architecture JDBC repose sur quatre composants principaux :



Les interfaces principales :

- `DriverManager` : Gère les drivers et établit les connexions
- `Connection` : Représente une connexion à la base de données

- `Statement` / `PreparedStatement` : Exécute les requêtes SQL
- `ResultSet` : Contient les résultats d'une requête SELECT

---

2. Le Rôle du DriverManager

2.1 Présentation

Le `DriverManager` est une classe centrale de l'API JDBC qui agit comme un **gestionnaire de pilotes** (drivers) de bases de données. C'est le point d'entrée principal pour établir des connexions.

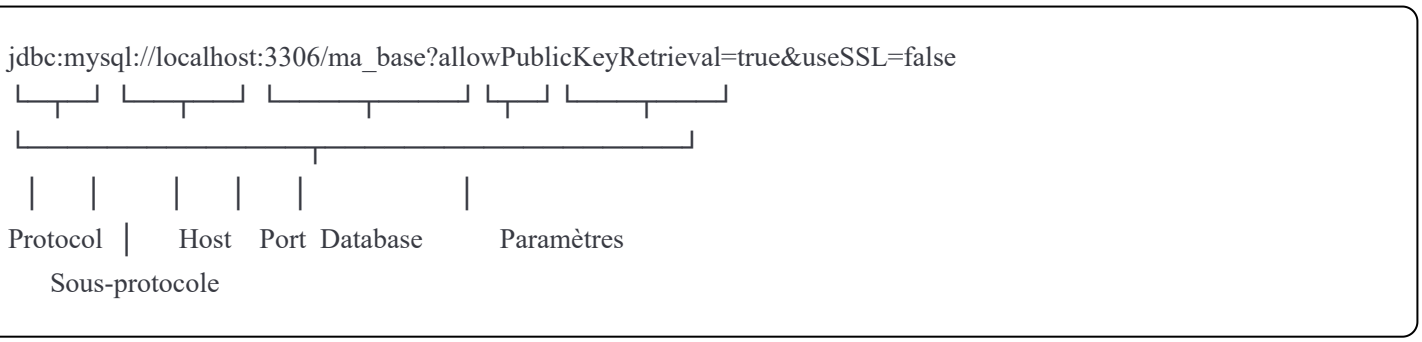
2.2 Fonctionnement

```java
Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
```

Processus d'établissement de connexion :

1. **Analyse de l'URL** : Le `DriverManager` examine l'URL JDBC pour identifier le type de base de données
2. **Détection du driver** : Il recherche le driver approprié parmi ceux enregistrés
3. **Chargement automatique** : Depuis JDBC 4.0, le driver se charge automatiquement via le Service Provider Interface (SPI)
4. **Création de la connexion** : Le driver établit la connexion et retourne un objet `Connection`

2.3 Structure de l'URL JDBC



Composants :

- **Protocol** : Toujours `jdbc`
- **Sous-protocole** : Type de base (mysql, postgresql, oracle, etc.)
- **Host** : Adresse du serveur
- **Port** : Port d'écoute du SGBD
- **Database** : Nom de la base de données

- **Paramètres** : Options de configuration

2.4 Exemple Pratique

```java
public class DatabaseConnection {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/ma_base?allowPublicKeyRetrieval=true&useSSL=false";
        String user = "root";
        String password = "rootpassword";

        try {
            // DriverManager détecte automatiquement le driver MySQL
            Connection conn = DriverManager.getConnection(url, user, password);
            System.out.println("✅ Connexion établie avec succès");

            // Utilisation de la connexion...

            conn.close();
        } catch (SQLException e) {
            System.err.println("❌ Erreur de connexion : " + e.getMessage());
        }
    }
}
```

2.5 Évolution : Chargement Automatique des Drivers

Avant JDBC 4.0 (approche manuelle) :

```java
// Chargement manuel obligatoire
Class.forName("com.mysql.cj.jdbc.Driver");
Connection conn = DriverManager.getConnection(url, user, password);
```

Depuis JDBC 4.0 (approche moderne) :

```java
// Chargement automatique via SPI
Connection conn = DriverManager.getConnection(url, user, password);
```

Le mécanisme SPI (Service Provider Interface) détecte automatiquement les drivers présents dans le classpath en lisant le fichier `META-INF/services/java.sql.Driver` du JAR du driver.

---

3. Le Pattern DatabaseManager

3.1 Problématique

Dans une application, gérer directement les connexions à chaque endroit pose plusieurs problèmes :

❌ **Code dupliqué** : Répétition des paramètres de connexion
❌ **Maintenance difficile** : Changement d'URL répercuté partout
❌ **Gestion complexe** : Fermeture des connexions oubliée
❌ **Testabilité réduite** : Impossible de mocker facilement

3.2 Solution : La Classe DatabaseManager

Le `DatabaseManager` est une classe qui **centralise et encapsule** la gestion des connexions à la base de données.

```java
```

```java
package com.emsi.jdbc;

import java.sql.*;

public class DatabaseManager {
    // Configuration centralisée
    private static final String URL = "jdbc:mysql://localhost:3306/ma_base?allowPublicKeyRetrieval=true&useSSL=false";
    private static final String USER = "root";
    private static final String PASSWORD = "rootpassword";

    private final Connection connection;

    // Établissement de la connexion à la création
    public DatabaseManager() throws SQLException {
        this.connection = DriverManager.getConnection(URL, USER, PASSWORD);
        System.out.println("✅ Connexion MySQL réussie !");
    }

    // Fournit l'accès à la connexion
    public Connection getConnection() {
        return connection;
    }

    // Fermeture propre de la connexion
    public void close() throws SQLException {
        if (connection != null && !connection.isClosed()) {
            connection.close();
            System.out.println("🔒 Connexion fermée");
        }
    }
}
```

3.3 Responsabilités du DatabaseManager

Le `DatabaseManager` a quatre responsabilités principales :

1. **Configuration** : Stocker les paramètres de connexion (URL, USER, PASSWORD)
2. **Initialisation** : Établir la connexion via `DriverManager` lors de sa création
3. **Fourniture** : Donner accès à l'objet `Connection` aux autres classes
4. **Libération** : Fermer proprement la connexion quand elle n'est plus nécessaire

3.4 Avantages de cette Approche

✅ **Centralisation** : Un seul point de configuration

✅ **Réutilisabilité** : Connexion partagée entre plusieurs classes
✅ **Encapsulation** : Détails techniques cachés
✅ **Maintenabilité** : Modification facile des paramètres
✅ **Gestion cohérente** : Ouverture/fermeture contrôlées

3.5 Limitation et Amélioration

Limitation actuelle :
Cette implémentation crée **une seule connexion** réutilisée par toute l'application. Ce n'est pas adapté pour :

- Les applications multi-utilisateurs
- Les requêtes concurrentes
- Les environnements de production

Solution professionnelle : Pool de connexions

```java
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

public class DatabaseManager {
    private static HikariDataSource dataSource;

    static {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/ma_base");
        config.setUsername("root");
        config.setPassword("rootpassword");
        config.setMaximumPoolSize(10);  // Pool de 10 connexions
        config.setMinimumIdle(5);

        dataSource = new HikariDataSource(config);
    }

    public static Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }

    public static void close() {
        if (dataSource != null && !dataSource.isClosed()) {
            dataSource.close();
        }
    }
}
```

Un **pool de connexions** maintient plusieurs connexions ouvertes et les réutilise, améliorant significativement les performances.

---

4. Le Pattern DAO (Data Access Object)

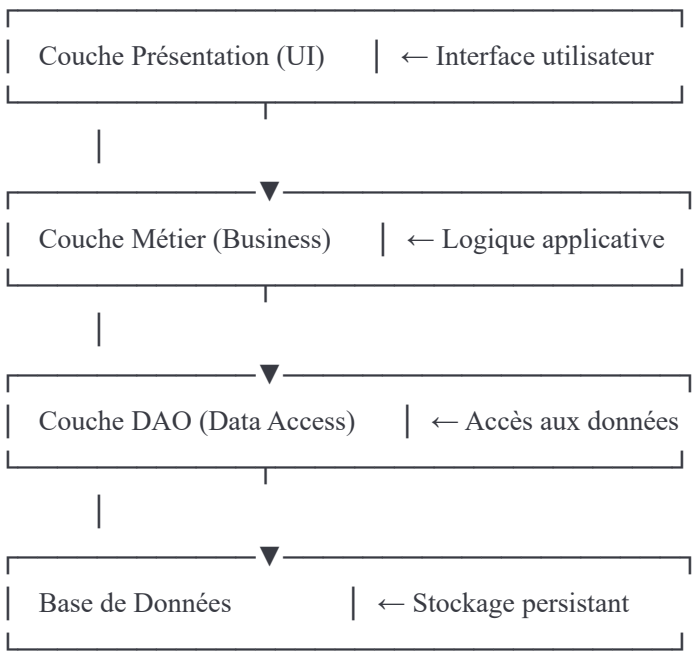
4.1 Principe du Pattern DAO

Le pattern DAO est un pattern de conception qui **sépare la logique métier de la logique d'accès aux données**. Il crée une couche d'abstraction entre l'application et la base de données.

Objectifs :

- Isoler les opérations CRUD (Create, Read, Update, Delete)
- Rendre le code plus modulaire et testable
- Faciliter le changement de source de données
- Respecter le principe de responsabilité unique

4.2 Architecture en Couches



4.3 Implémentation : La Classe UserDAO

java

```java
package com.emsi.jdbc;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class UserDAO {
    private DatabaseManager dbManager;

    // Injection de dépendance via le constructeur
    public UserDAO(DatabaseManager dbManager) {
        this.dbManager = dbManager;
    }

    // CREATE : Création de la table
    public void createTable() throws SQLException {
        String sql = "CREATE TABLE IF NOT EXISTS users (" +
                     "id INT AUTO_INCREMENT PRIMARY KEY, " +
                     "name VARCHAR(100) NOT NULL, " +
                     "email VARCHAR(100) UNIQUE NOT NULL)";

        try (Statement stmt = dbManager.getConnection().createStatement()) {
            stmt.execute(sql);
            System.out.println("✅ Table 'users' créée");
        }
    }

    // CREATE : Insertion d'un utilisateur
    public void insertUser(User user) throws SQLException {
        String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setString(1, user.getName());
            pstmt.setString(2, user.getEmail());
            pstmt.executeUpdate();
            System.out.println("✅ User inséré : " + user);
        }
    }

    // READ : Récupération de tous les utilisateurs
    public List<User> getAllUsers() throws SQLException {
        List<User> users = new ArrayList<>();
        String sql = "SELECT * FROM users";

        try (Statement stmt = dbManager.getConnection().createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {
```

```
            while (rs.next()) {
                User user = new User();
                user.setId(rs.getInt("id"));
                user.setName(rs.getString("name"));
                user.setEmail(rs.getString("email"));
                users.add(user);
            }
        }
        return users;
    }

    // UPDATE : Mise à jour d'un utilisateur
    public void updateUser(User user) throws SQLException {
        String sql = "UPDATE users SET name = ?, email = ? WHERE id = ?";

        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setString(1, user.getName());
            pstmt.setString(2, user.getEmail());
            pstmt.setInt(3, user.getId());
            pstmt.executeUpdate();
            System.out.println("✅ User mis à jour : " + user);
        }
    }

    // DELETE : Suppression d'un utilisateur
    public void deleteUser(int id) throws SQLException {
        String sql = "DELETE FROM users WHERE id = ?";

        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setInt(1, id);
            pstmt.executeUpdate();
            System.out.println("✅ User supprimé (id: " + id + ")");
        }
    }
}
```

4.4 La Classe Modèle : User

java

```java
package com.emsi.jdbc;

public class User {
    private int id;
    private String name;
    private String email;

    // Constructeurs
    public User() {}

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters et Setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    @Override
    public String toString() {
        return "User{id=" + id + ", name='" + name + "', email='" + email + "'}";
    }
}
```

---

5. Le Rôle du DatabaseManager dans le DAO

5.1 Injection de Dépendance

Dans l'architecture DAO, le `DatabaseManager` est **injecté** dans le DAO plutôt que créé par lui. C'est le principe de l'**injection de dépendance**.

```java
```

```
public class UserDAO {
    private DatabaseManager dbManager;  // ← Dépendance externe

    // Injection via le constructeur
    public UserDAO(DatabaseManager dbManager) {
        this.dbManager = dbManager;
    }
}
```

Pourquoi l'injection ?

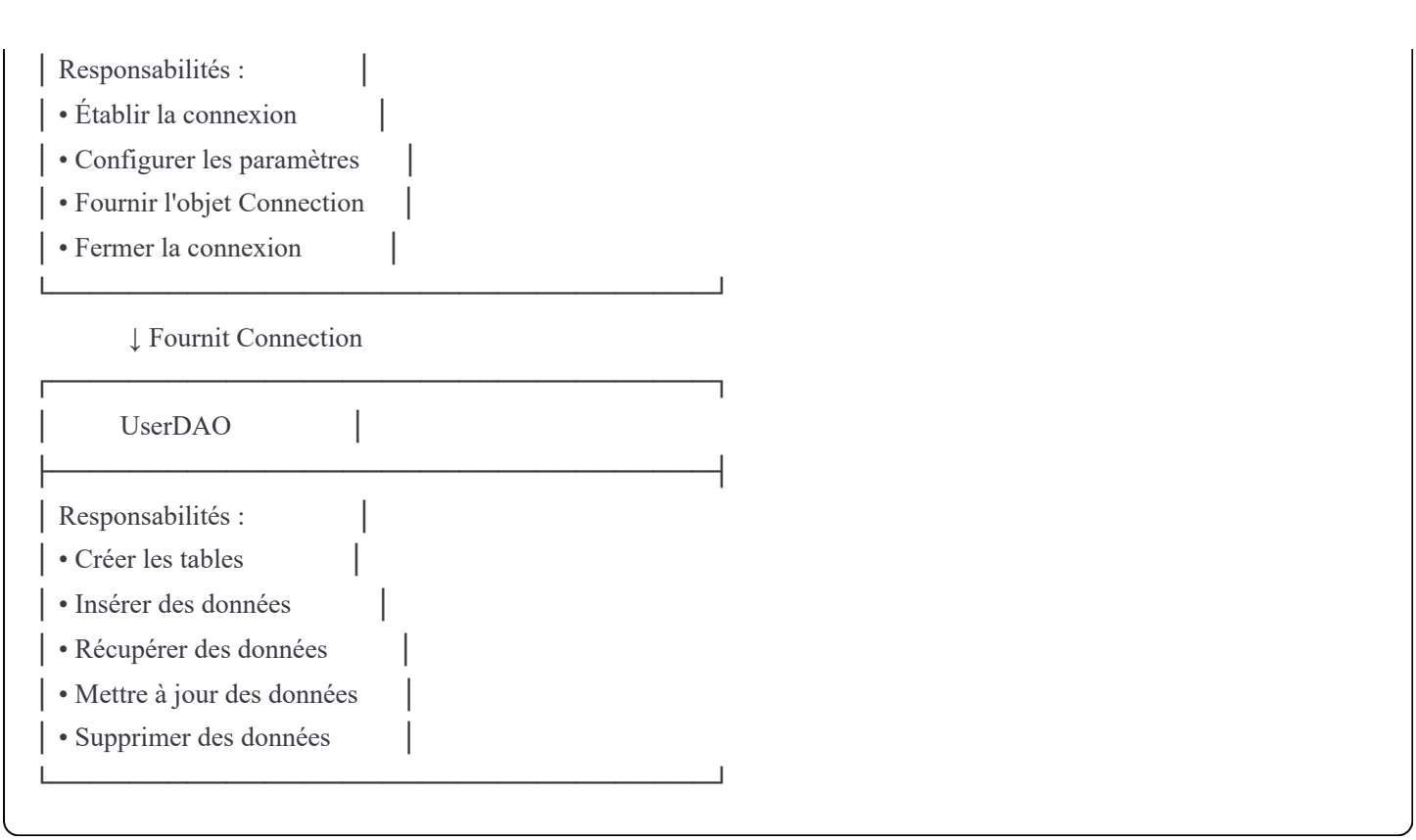
- Le DAO ne gère pas la création/fermeture des connexions
- Permet de partager une connexion entre plusieurs DAO
- Facilite le remplacement (tests, changement de BD)
- Respecte le principe d'inversion de dépendances

5.2 Fournisseur de Connexions

Dans chaque opération CRUD, le `DatabaseManager` fournit l'accès à la connexion :

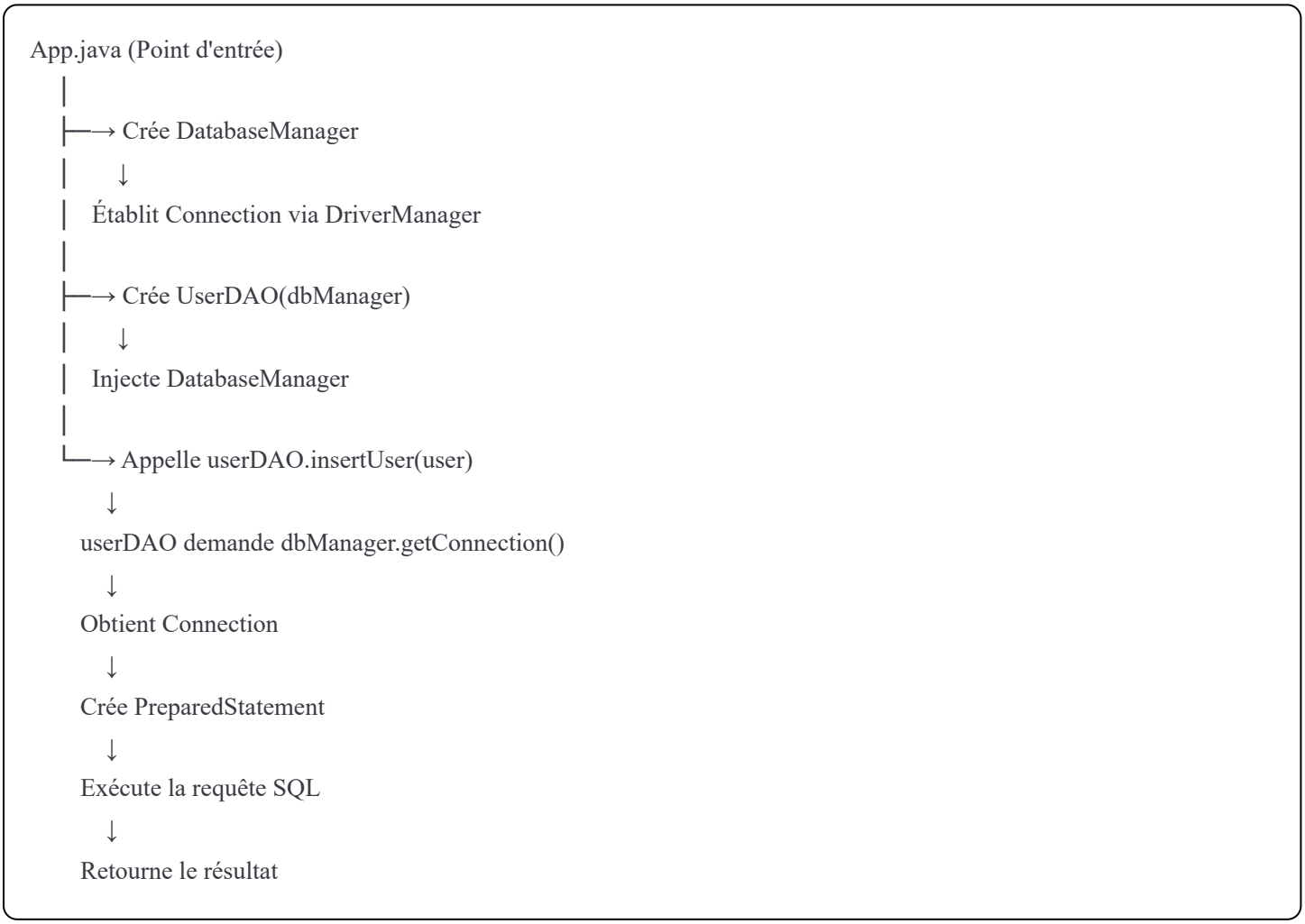
```
java

// CREATE
try (Statement stmt = dbManager.getConnection().createStatement()) {
    stmt.execute(sql);
}

// INSERT avec PreparedStatement
try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
    pstmt.setString(1, user.getName());
    pstmt.executeUpdate();
}

// SELECT avec ResultSet
try (Statement stmt = dbManager.getConnection().createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
    // Traitement des résultats
}
```

5.3 Séparation des Responsabilités



Chaque classe a une **responsabilité unique et bien définie**, conformément au principe SOLID (Single Responsibility Principle).

5.4 Flux de Données Complet



5.5 Avantages de cette Architecture

✅ Réutilisabilité

Un seul `DatabaseManager` peut servir plusieurs DAO :

```java
DatabaseManager dbManager = new DatabaseManager();

UserDAO userDAO = new UserDAO(dbManager);
ProductDAO productDAO = new ProductDAO(dbManager);
OrderDAO orderDAO = new OrderDAO(dbManager);
// Tous partagent la même connexion
```

✅ Testabilité

Possibilité de créer des mocks pour les tests unitaires :

```java
// En production
DatabaseManager realDB = new DatabaseManager();
UserDAO userDAO = new UserDAO(realDB);

// En test
DatabaseManager mockDB = new MockDatabaseManager();
UserDAO userDAO = new UserDAO(mockDB);  // Test sans vraie BD
```

✅ Flexibilité

Changement de base de données sans modifier le DAO :

```java
// MySQL
DatabaseManager mysqlDB = new MySQLDatabaseManager();
UserDAO userDAO = new UserDAO(mysqlDB);

// PostgreSQL
DatabaseManager postgresDB = new PostgreSQLDatabaseManager();
UserDAO userDAO = new UserDAO(postgresDB);

// Le code de UserDAO reste identique !
```

✅ Maintenabilité

Modification des paramètres de connexion à un seul endroit :

java

```
// Changement d'URL, de port, de mot de passe...
// → Modification uniquement dans DatabaseManager
// → Aucun changement dans les DAO
```

---

6. Application Complète : Exemple Pratique

6.1 Structure du Projet

```
src/
└── com/emsi/jdbc/
    ├── App.java              (Point d'entrée)
    ├── DatabaseManager.java  (Gestion connexion)
    ├── User.java             (Modèle de données)
    └── UserDAO.java          (Accès aux données)
```

6.2 Classe Principale : App.java

java

```java
package com.emsi.jdbc;

import java.sql.SQLException;
import java.util.List;

public class App {
    public static void main(String[] args) {
        DatabaseManager dbManager = null;

        try {
            // 1. Créer le gestionnaire de connexion
            dbManager = new DatabaseManager();

            // 2. Créer le DAO avec injection de dépendance
            UserDAO userDAO = new UserDAO(dbManager);

            // 3. Créer la table
            userDAO.createTable();

            // 4. Insérer des utilisateurs
            System.out.println("\n📝 Insertion d'utilisateurs...");
            userDAO.insertUser(new User("Ahmed Alami", "ahmed@example.ma"));
            userDAO.insertUser(new User("Fatima Zahra", "fatima@example.ma"));
            userDAO.insertUser(new User("Mohammed Bennani", "mohammed@example.ma"));

            // 5. Récupérer et afficher tous les utilisateurs
            System.out.println("\n📋 Liste des utilisateurs :");
            List<User> users = userDAO.getAllUsers();
            users.forEach(System.out::println);

            // 6. Mettre à jour un utilisateur
            System.out.println("\n🔄 Mise à jour...");
            User userToUpdate = users.get(0);
            userToUpdate.setEmail("ahmed.new@example.ma");
            userDAO.updateUser(userToUpdate);

            // 7. Supprimer un utilisateur
            System.out.println("\n🗑️ Suppression...");
            userDAO.deleteUser(users.get(2).getId());

            // 8. Afficher la liste finale
            System.out.println("\n📋 Liste finale :");
            userDAO.getAllUsers().forEach(System.out::println);

        } catch (SQLException e) {
            System.err.println("❌ Erreur SQL : " + e.getMessage());
```

```
            e.printStackTrace();
        } finally {
            // 9. Fermer la connexion proprement
            if (dbManager != null) {
                try {
                    dbManager.close();
                } catch (SQLException e) {
                    System.err.println("❌ Erreur lors de la fermeture : " + e.getMessage());
                }
            }
        }
    }
}
```

6.3 Résultat d'Exécution

```
✅ Connexion MySQL réussie !
✅ Table 'users' créée

📝 Insertion d'utilisateurs...
✅ User inséré : User{id=0, name='Ahmed Alami', email='ahmed@example.ma'}
✅ User inséré : User{id=0, name='Fatima Zahra', email='fatima@example.ma'}
✅ User inséré : User{id=0, name='Mohammed Bennani', email='mohammed@example.ma'}

📋 Liste des utilisateurs :
User{id=1, name='Ahmed Alami', email='ahmed@example.ma'}
User{id=2, name='Fatima Zahra', email='fatima@example.ma'}
User{id=3, name='Mohammed Bennani', email='mohammed@example.ma'}

🔄 Mise à jour...
✅ User mis à jour : User{id=1, name='Ahmed Alami', email='ahmed.new@example.ma'}

🗑️ Suppression...
✅ User supprimé (id: 3)

📋 Liste finale :
User{id=1, name='Ahmed Alami', email='ahmed.new@example.ma'}
User{id=2, name='Fatima Zahra', email='fatima@example.ma'}

🔒 Connexion fermée
```

---

7. Gestion des Erreurs JDBC Courantes

7.1 Erreur : "Public Key Retrieval is not allowed"

Cause :
MySQL 8.0+ utilise le plugin d'authentification `caching_sha2_password` qui nécessite la récupération de la clé publique RSA du serveur. Par défaut, JDBC refuse cette récupération pour des raisons de sécurité.

Solution pour le développement :

```java
String url = "jdbc:mysql://localhost:3306/ma_base?" +
             "allowPublicKeyRetrieval=true&" +
             "useSSL=false";
```

Solution pour la production :

```java
String url = "jdbc:mysql://localhost:3306/ma_base?useSSL=true";
```

Ou changer le plugin d'authentification :

```sql
ALTER USER 'root'@'localhost'
IDENTIFIED WITH mysql_native_password BY 'password';
FLUSH PRIVILEGES;
```

7.2 Erreur : "Communications link failure"

Causes possibles :

- Serveur MySQL non démarré
- Mauvaise adresse/port
- Pare-feu bloquant la connexion
- Timeout de connexion

Vérifications :

```bash
```

```
# Vérifier si MySQL écoute
netstat -an | grep 3306

# Tester la connexion
mysql -h localhost -P 3306 -u root -p
```

7.3 Erreur : "Access denied for user"

Cause :
Identifiants incorrects ou permissions insuffisantes.

Solution :

```
sql

-- Vérifier les utilisateurs
SELECT user, host FROM mysql.user;

-- Créer un nouvel utilisateur
CREATE USER 'monuser'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON ma_base.* TO 'monuser'@'localhost';
FLUSH PRIVILEGES;
```

7.4 Bonnes Pratiques de Gestion d'Erreurs

```
java
```

```
public void insertUser(User user) throws SQLException {
    String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

    try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
        pstmt.setString(1, user.getName());
        pstmt.setString(2, user.getEmail());
        pstmt.executeUpdate();

    } catch (SQLIntegrityConstraintViolationException e) {
        // Email déjà existant (UNIQUE constraint)
        throw new SQLException("Cet email existe déjà : " + user.getEmail(), e);

    } catch (SQLTimeoutException e) {
        // Timeout de connexion
        throw new SQLException("Délai d'attente dépassé", e);

    } catch (SQLException e) {
        // Autres erreurs SQL
        throw new SQLException("Erreur lors de l'insertion : " + e.getMessage(), e);
    }
}
```

---

8. Exécution de Scripts SQL Complexes

8.1 Problématique

JDBC ne permet pas d'exécuter directement des scripts SQL contenant plusieurs instructions séparées par des points-virgules, ni d'utiliser des commandes spécifiques comme `USE database`.

8.2 Solution : Exécution Instruction par Instruction

java

```
public void initializeDatabase() throws SQLException {
    Connection conn = dbManager.getConnection();

    try (Statement stmt = conn.createStatement()) {
        // 1. Créer la base
        stmt.execute("CREATE DATABASE IF NOT EXISTS ma_base");
        System.out.println("✅ Base créée");

        // 2. Sélectionner la base
        stmt.execute("USE ma_base");

        // 3. Créer la table clients
        stmt.execute("""
            CREATE TABLE IF NOT EXISTS clients (
                id INT PRIMARY KEY,
                nom VARCHAR(50),
                prenom VARCHAR(50),
                email VARCHAR(100)
            )
        """);

        // 4. Créer la table commandes
        stmt.execute("""
            CREATE TABLE IF NOT EXISTS commandes (
                id INT PRIMARY KEY,
                client_id INT,
                date_commande DATE,
                montant DECIMAL(10, 2),
                FOREIGN KEY (client_id) REFERENCES clients(id)
            )
        """);

        // 5. Insérer les données avec PreparedStatement (batch)
        insertClientsData(conn);
        insertCommandesData(conn);

        System.out.println("✅ Base de données initialisée");

    } catch (SQLException e) {
        System.err.println("❌ Erreur : " + e.getMessage());
        throw e;
    }
}

private void insertClientsData(Connection conn) throws SQLException {
    String sql = "INSERT INTO clients (id, nom, prenom, email) VALUES (?, ?, ?, ?)";
```

```
    try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
        String[][] clients = {
            {"1", "El Amrani", "Youssef", "youssef@example.ma"},
            {"2", "Benziane", "Sanaa", "sanaa@example.ma"},
            {"3", "Othmani", "Hicham", "hicham@example.ma"}
        };

        for (String[] client : clients) {
            pstmt.setInt(1, Integer.parseInt(client[0]));
            pstmt.setString(2, client[1]);
            pstmt.setString(3, client[2]);
            pstmt.setString(4, client[3]);
            pstmt.addBatch();  // Ajout au batch
        }

        pstmt.executeBatch();  // Exécution groupée
        System.out.println("✅ Clients insérés");
    }
}
```

8.3 Alternative : Lecture depuis un Fichier SQL

java

```
public void executeSQLFile(String filePath) throws SQLException, IOException {
    // Lire le fichier SQL
    String content = Files.readString(Path.of(filePath));

    // Séparer les instructions
    String[] statements = content.split(";");

    try (Statement stmt = dbManager.getConnection().createStatement()) {
        for (String sql : statements) {
            sql = sql.trim();
            // Ignorer les lignes vides et les commentaires
            if (!sql.isEmpty() && !sql.startsWith("--") && !sql.startsWith("/*")) {
                stmt.execute(sql);
            }
        }
        System.out.println("✅ Script SQL exécuté");
    }
}
```

---

9. Bonnes Pratiques et Recommandations

9.1 Utilisation de PreparedStatement

Toujours préférer `PreparedStatement` à `Statement` pour :

✅ **Sécurité** : Protection contre les injections SQL
✅ **Performance** : Requêtes précompilées et réutilisables
✅ **Lisibilité** : Code plus clair avec les paramètres

```java
// ❌ MAUVAIS : Vulnérable aux injections SQL
String sql = "SELECT * FROM users WHERE email = '" + email + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);

// ✅ BON : Utilisation de PreparedStatement
String sql = "SELECT * FROM users WHERE email = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, email);
ResultSet rs = pstmt.executeQuery();
```

9.2 Gestion des Ressources avec try-with-resources

Toujours utiliser try-with-resources pour fermer automatiquement les ressources JDBC :

```java
```

```java
// ✅ BON : Fermeture automatique
try (Connection conn = dbManager.getConnection();
     PreparedStatement pstmt = conn.prepareStatement(sql);
     ResultSet rs = pstmt.executeQuery()) {

    while (rs.next()) {
        // Traitement des résultats
    }
} // Fermeture automatique même en cas d'exception

// ❌ MAUVAIS : Fermeture manuelle risquée
Connection conn = null;
PreparedStatement pstmt = null;
ResultSet rs = null;
try {
    conn = dbManager.getConnection();
    pstmt = conn.prepareStatement(sql);
    rs = pstmt.executeQuery();
    // ...
} finally {
    if (rs != null) rs.close();
    if (pstmt != null) pstmt.close();
    if (conn != null) conn.close();
}
```

9.3 Utilisation de Batch pour les Insertions Multiples

Pour insérer plusieurs enregistrements, utilisez `addBatch()` et `executeBatch()` :

java

```java
public void insertMultipleUsers(List<User> users) throws SQLException {
    String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

    try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
        for (User user : users) {
            pstmt.setString(1, user.getName());
            pstmt.setString(2, user.getEmail());
            pstmt.addBatch();  // Ajout au batch
        }

        int[] results = pstmt.executeBatch();  // Exécution groupée
        System.out.println("✅ " + results.length + " utilisateurs insérés");
    }
}
```

Avantages :

- Réduction du nombre d'allers-retours réseau
- Amélioration significative des performances
- Optimisation par le SGBD

9.4 Gestion des Transactions

Pour garantir la cohérence des données, utilisez les transactions :

```java
```

```java
public void transferMoney(int fromId, int toId, double amount) throws SQLException {
    Connection conn = dbManager.getConnection();

    try {
        // Désactiver l'auto-commit
        conn.setAutoCommit(false);

        // Opération 1 : Débiter le compte source
        String debitSql = "UPDATE accounts SET balance = balance - ? WHERE id = ?";
        try (PreparedStatement pstmt = conn.prepareStatement(debitSql)) {
            pstmt.setDouble(1, amount);
            pstmt.setInt(2, fromId);
            pstmt.executeUpdate();
        }

        // Opération 2 : Créditer le compte destination
        String creditSql = "UPDATE accounts SET balance = balance + ? WHERE id = ?";
        try (PreparedStatement pstmt = conn.prepareStatement(creditSql)) {
            pstmt.setDouble(1, amount);
            pstmt.setInt(2, toId);
            pstmt.executeUpdate();
        }

        // Valider la transaction
        conn.commit();
        System.out.println("✅ Transfert effectué avec succès");

    } catch (SQLException e) {
        // Annuler la transaction en cas d'erreur
        conn.rollback();
        System.err.println("❌ Erreur : transaction annulée");
        throw e;

    } finally {
        // Réactiver l'auto-commit
        conn.setAutoCommit(true);
    }
}
```

9.5 Éviter les Fuites de Ressources

java

```java
// ❌ MAUVAIS : Risque de fuite mémoire
public List<User> getAllUsers() throws SQLException {
    Statement stmt = dbManager.getConnection().createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM users");

    List<User> users = new ArrayList<>();
    while (rs.next()) {
        users.add(mapResultSetToUser(rs));
    }
    return users;
    // stmt et rs ne sont jamais fermés !
}

// ✅ BON : Fermeture garantie
public List<User> getAllUsers() throws SQLException {
    List<User> users = new ArrayList<>();
    String sql = "SELECT * FROM users";

    try (Statement stmt = dbManager.getConnection().createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        while (rs.next()) {
            users.add(mapResultSetToUser(rs));
        }
    }
    return users;
}
```

9.6 Validation des Données

Toujours valider les données avant l'insertion :

```java
```

```java
public void insertUser(User user) throws SQLException {
    // Validation métier
    if (user.getName() == null || user.getName().trim().isEmpty()) {
        throw new IllegalArgumentException("Le nom ne peut pas être vide");
    }

    if (user.getEmail() == null || !user.getEmail().matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
        throw new IllegalArgumentException("Email invalide");
    }

    // Insertion
    String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
    try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
        pstmt.setString(1, user.getName().trim());
        pstmt.setString(2, user.getEmail().toLowerCase());
        pstmt.executeUpdate();
    }
}
```

---

10. Patterns Avancés et Architecture Évolutive

10.1 Interface DAO Générique

Pour éviter la duplication de code, créez une interface générique :

java

```java
public interface GenericDAO<T, ID> {
    void create(T entity) throws SQLException;
    T findById(ID id) throws SQLException;
    List<T> findAll() throws SQLException;
    void update(T entity) throws SQLException;
    void delete(ID id) throws SQLException;
}
```

Implémentation :

java

```
public class UserDAO implements GenericDAO<User, Integer> {
    private DatabaseManager dbManager;

    public UserDAO(DatabaseManager dbManager) {
        this.dbManager = dbManager;
    }

    @Override
    public void create(User user) throws SQLException {
        String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setString(1, user.getName());
            pstmt.setString(2, user.getEmail());
            pstmt.executeUpdate();
        }
    }

    @Override
    public User findById(Integer id) throws SQLException {
        String sql = "SELECT * FROM users WHERE id = ?";
        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setInt(1, id);
            try (ResultSet rs = pstmt.executeQuery()) {
                if (rs.next()) {
                    return mapResultSetToUser(rs);
                }
                return null;
            }
        }
    }

    @Override
    public List<User> findAll() throws SQLException {
        List<User> users = new ArrayList<>();
        String sql = "SELECT * FROM users";

        try (Statement stmt = dbManager.getConnection().createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                users.add(mapResultSetToUser(rs));
            }
        }
        return users;
    }
```

```java
    @Override
    public void update(User user) throws SQLException {
        String sql = "UPDATE users SET name = ?, email = ? WHERE id = ?";
        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setString(1, user.getName());
            pstmt.setString(2, user.getEmail());
            pstmt.setInt(3, user.getId());
            pstmt.executeUpdate();
        }
    }

    @Override
    public void delete(Integer id) throws SQLException {
        String sql = "DELETE FROM users WHERE id = ?";
        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setInt(1, id);
            pstmt.executeUpdate();
        }
    }

    private User mapResultSetToUser(ResultSet rs) throws SQLException {
        User user = new User();
        user.setId(rs.getInt("id"));
        user.setName(rs.getString("name"));
        user.setEmail(rs.getString("email"));
        return user;
    }
}
```

10.2 Factory Pattern pour les DAO

Centralisez la création des DAO avec une factory :

java

```
public class DAOFactory {
    private static DatabaseManager dbManager;

    static {
        try {
            dbManager = new DatabaseManager();
        } catch (SQLException e) {
            throw new RuntimeException("Impossible d'initialiser la base de données", e);
        }
    }

    public static UserDAO getUserDAO() {
        return new UserDAO(dbManager);
    }

    public static ProductDAO getProductDAO() {
        return new ProductDAO(dbManager);
    }

    public static OrderDAO getOrderDAO() {
        return new OrderDAO(dbManager);
    }

    public static void closeConnection() {
        try {
            if (dbManager != null) {
                dbManager.close();
            }
        } catch (SQLException e) {
            System.err.println("Erreur lors de la fermeture : " + e.getMessage());
        }
    }
}
```

Utilisation :

```
java
```

```java
public class App {
    public static void main(String[] args) {
        try {
            // Récupération des DAO via la factory
            UserDAO userDAO = DAOFactory.getUserDAO();
            ProductDAO productDAO = DAOFactory.getProductDAO();

            // Utilisation
            userDAO.create(new User("Ahmed", "ahmed@example.ma"));
            productDAO.create(new Product("Ordinateur", 5000.0));

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            DAOFactory.closeConnection();
        }
    }
}
```

10.3 Repository Pattern (Alternative au DAO)

Le pattern Repository est une évolution moderne du DAO :

java

```java
public interface UserRepository {
    User save(User user);
    Optional<User> findById(int id);
    List<User> findAll();
    List<User> findByEmail(String email);
    void delete(int id);
    boolean exists(int id);
}
```

java

```
public class UserRepositoryImpl implements UserRepository {
    private DatabaseManager dbManager;

    public UserRepositoryImpl(DatabaseManager dbManager) {
        this.dbManager = dbManager;
    }

    @Override
    public User save(User user) {
        try {
            if (user.getId() == 0) {
                // Insertion
                String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
                try (PreparedStatement pstmt = dbManager.getConnection()
                        .prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {

                    pstmt.setString(1, user.getName());
                    pstmt.setString(2, user.getEmail());
                    pstmt.executeUpdate();

                    // Récupérer l'ID généré
                    try (ResultSet rs = pstmt.getGeneratedKeys()) {
                        if (rs.next()) {
                            user.setId(rs.getInt(1));
                        }
                    }
                }
            } else {
                // Mise à jour
                String sql = "UPDATE users SET name = ?, email = ? WHERE id = ?";
                try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
                    pstmt.setString(1, user.getName());
                    pstmt.setString(2, user.getEmail());
                    pstmt.setInt(3, user.getId());
                    pstmt.executeUpdate();
                }
            }
            return user;
        } catch (SQLException e) {
            throw new RuntimeException("Erreur lors de la sauvegarde", e);
        }
    }

    @Override
    public Optional<User> findById(int id) {
        try {
```

```
        String sql = "SELECT * FROM users WHERE id = ?";
        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setInt(1, id);
            try (ResultSet rs = pstmt.executeQuery()) {
                if (rs.next()) {
                    return Optional.of(mapResultSetToUser(rs));
                }
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Erreur lors de la recherche", e);
    }
    return Optional.empty();
}

@Override
public boolean exists(int id) {
    try {
        String sql = "SELECT COUNT(*) FROM users WHERE id = ?";
        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setInt(1, id);
            try (ResultSet rs = pstmt.executeQuery()) {
                return rs.next() && rs.getInt(1) > 0;
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Erreur lors de la vérification", e);
    }
}

private User mapResultSetToUser(ResultSet rs) throws SQLException {
    User user = new User();
    user.setId(rs.getInt("id"));
    user.setName(rs.getString("name"));
    user.setEmail(rs.getString("email"));
    return user;
}

// Autres méthodes...
}
```

---

11. Tests Unitaires avec JDBC

11.1 Tests avec Base de Données In-Memory (H2)

Pour les tests, utilisez une base de données en mémoire comme H2 :

Dépendance Maven :

xml

```xml
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.2.224</version>
    <scope>test</scope>
</dependency>
```

Classe de test :

java

```java
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class UserDAOTest {
    private static DatabaseManager dbManager;
    private UserDAO userDAO;

    @BeforeAll
    public static void setupDatabase() throws SQLException {
        // Connexion à une base H2 en mémoire
        String url = "jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1";
        dbManager = new DatabaseManager(url, "sa", "");
    }

    @BeforeEach
    public void setup() throws SQLException {
        userDAO = new UserDAO(dbManager);
        userDAO.createTable();
    }

    @AfterEach
    public void cleanup() throws SQLException {
        // Nettoyer la table après chaque test
        Statement stmt = dbManager.getConnection().createStatement();
        stmt.execute("DROP TABLE IF EXISTS users");
    }

    @Test
    public void testInsertUser() throws SQLException {
        // Arrange
        User user = new User("Test User", "test@example.com");

        // Act
        userDAO.insertUser(user);
        List<User> users = userDAO.getAllUsers();

        // Assert
        assertEquals(1, users.size());
        assertEquals("Test User", users.get(0).getName());
        assertEquals("test@example.com", users.get(0).getEmail());
    }

    @Test
    public void testUpdateUser() throws SQLException {
        // Arrange
        User user = new User("Original Name", "original@example.com");
```

```
        userDAO.insertUser(user);

        List<User> users = userDAO.getAllUsers();
        User insertedUser = users.get(0);

        // Act
        insertedUser.setName("Updated Name");
        userDAO.updateUser(insertedUser);

        users = userDAO.getAllUsers();

        // Assert
        assertEquals(1, users.size());
        assertEquals("Updated Name", users.get(0).getName());
    }

    @Test
    public void testDeleteUser() throws SQLException {
        // Arrange
        User user = new User("To Delete", "delete@example.com");
        userDAO.insertUser(user);

        List<User> users = userDAO.getAllUsers();
        int userId = users.get(0).getId();

        // Act
        userDAO.deleteUser(userId);
        users = userDAO.getAllUsers();

        // Assert
        assertEquals(0, users.size());
    }

    @AfterAll
    public static void teardown() throws SQLException {
        if (dbManager != null) {
            dbManager.close();
        }
    }
}
```

11.2 Mock du DatabaseManager

Pour des tests unitaires purs sans base de données :

```
java
```

```java
import org.mockito.*;
import static org.mockito.Mockito.*;

public class UserDAOUnitTest {

    @Mock
    private DatabaseManager mockDbManager;

    @Mock
    private Connection mockConnection;

    @Mock
    private PreparedStatement mockPreparedStatement;

    private UserDAO userDAO;

    @BeforeEach
    public void setup() throws SQLException {
        MockitoAnnotations.openMocks(this);
        when(mockDbManager.getConnection()).thenReturn(mockConnection);
        userDAO = new UserDAO(mockDbManager);
    }

    @Test
    public void testInsertUser() throws SQLException {
        // Arrange
        when(mockConnection.prepareStatement(anyString())).thenReturn(mockPreparedStatement);
        User user = new User("Test", "test@example.com");

        // Act
        userDAO.insertUser(user);

        // Assert
        verify(mockPreparedStatement).setString(1, "Test");
        verify(mockPreparedStatement).setString(2, "test@example.com");
        verify(mockPreparedStatement).executeUpdate();
    }
}
```

---

12. Migration vers des Frameworks Modernes

12.1 Limites de JDBC Pur

Bien que JDBC soit puissant, il présente des inconvénients pour les grandes applications :

❌ **Code verbeux** : Beaucoup de code répétitif (boilerplate)
❌ **Gestion manuelle** : Mapping manuel des ResultSet vers les objets
❌ **Erreurs fréquentes** : Risques de fuites de ressources
❌ **Maintenance complexe** : Changements de schéma difficiles à gérer

12.2 JPA et Hibernate

Java Persistence API (JPA) avec **Hibernate** simplifie considérablement le code :

Entité JPA :

```java
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false, length = 100)
    private String name;

    @Column(unique = true, nullable = false)
    private String email;

    // Constructeurs, getters, setters
}
```

Repository JPA :

```java
public interface UserRepository extends JpaRepository<User, Integer> {
    // Méthodes générées automatiquement
    List<User> findByName(String name);
    Optional<User> findByEmail(String email);
    List<User> findByNameContaining(String keyword);
}
```

Utilisation :

```java
```

```java
// Plus de SQL manuel !
User user = new User("Ahmed", "ahmed@example.com");
userRepository.save(user);  // INSERT automatique

List<User> users = userRepository.findAll();  // SELECT automatique
User found = userRepository.findById(1).orElse(null);  // SELECT avec WHERE

user.setName("Ahmed Alami");
userRepository.save(user);  // UPDATE automatique

userRepository.deleteById(1);  // DELETE automatique
```

12.3 Spring Data JDBC

Alternative plus légère que JPA/Hibernate :

java

```java
@Table("users")
public class User {
    @Id
    private Integer id;
    private String name;
    private String email;

    // Getters et setters
}

public interface UserRepository extends CrudRepository<User, Integer> {
    @Query("SELECT * FROM users WHERE email = :email")
    Optional<User> findByEmail(String email);
}
```

12.4 MyBatis

Framework hybride offrant plus de contrôle sur le SQL :

Mapper XML :

xml

```xml
<mapper namespace="com.emsi.mapper.UserMapper">
    <select id="findById" resultType="User">
        SELECT * FROM users WHERE id = #{id}
    </select>

    <insert id="insert" parameterType="User">
        INSERT INTO users (name, email)
        VALUES (#{name}, #{email})
    </insert>
</mapper>
```

Interface Java :

```java
public interface UserMapper {
    User findById(int id);
    void insert(User user);
    void update(User user);
    void delete(int id);
}
```

12.5 Comparaison des Approches

Critère	JDBC Pur	MyBatis	JPA/Hibernate
Courbe d'apprentissage	Facile	Moyenne	Difficile
Contrôle SQL	Total	Élevé	Limité
Code boilerplate	Beaucoup	Moyen	Minimal
Performance	Excellente	Très bonne	Bonne
Maintenance	Difficile	Moyenne	Facile
Portabilité BD	Faible	Moyenne	Excellente
Idéal pour	Petits projets, apprentissage	Applications moyennes	Grandes applications

13. Conclusion et Recommandations

13.1 Synthèse des Concepts

Ce chapitre a couvert l'essentiel de la gestion des bases de données en Java avec JDBC :

1. **DriverManager** : Le gestionnaire de drivers qui établit les connexions
2. **DatabaseManager** : Pattern de centralisation de la gestion des connexions
3. **DAO Pattern** : Séparation de la logique d'accès aux données
4. **Injection de dépendances** : Découplage et testabilité
5. **Bonnes pratiques** : PreparedStatement, try-with-resources, transactions

13.2 Quand Utiliser JDBC Pur ?

✅ **Utilisez JDBC dans ces cas :**

- Projets d'apprentissage et académiques
- Applications simples avec peu de tables
- Requêtes SQL complexes nécessitant un contrôle total
- Projets legacy à maintenir
- Scripts de migration ou outils de manipulation de données

13.3 Quand Migrer vers un Framework ?

✅ **Migrez vers JPA/Hibernate quand :**

- Vous avez plus de 10 tables avec relations complexes
- Vous avez besoin de portabilité entre différents SGBD
- Vous voulez un développement plus rapide
- Vous avez une équipe avec l'expertise nécessaire

✅ **Choisissez MyBatis quand :**

- Vous voulez garder le contrôle sur le SQL
- Vous avez des requêtes complexes et optimisées
- Vous migrez depuis JDBC pur progressivement
- Vous voulez un juste milieu entre contrôle et productivité

13.4 Checklist des Bonnes Pratiques

Avant de déployer votre application, vérifiez :

- [ ] Utilisation de `PreparedStatement` partout (pas de `Statement`)
- [ ] Try-with-resources pour toutes les ressources JDBC
- [ ] Gestion appropriée des transactions pour les opérations critiques
- [ ] Validation des données avant insertion
- [ ] Gestion des erreurs avec des messages explicites
- [ ] Fermeture systématique des connexions
- [ ] Utilisation d'un pool de connexions en production
- [ ] Tests unitaires pour toutes les méthodes DAO
- [ ] Logs appropriés pour le débogage
- [ ] Configuration externalisée (fichier properties)
- [ ] Documentation du schéma de base de données
- [ ] Stratégie de migration de schéma définie

13.5 Ressources pour Aller Plus Loin

Documentation officielle :

- Oracle JDBC Tutorial : https://docs.oracle.com/javase/tutorial/jdbc/
- MySQL Connector/J : https://dev.mysql.com/doc/connector-j/en/
- PostgreSQL JDBC : https://jdbc.postgresql.org/

Frameworks modernes :

- Hibernate ORM : https://hibernate.org/
- Spring Data JPA : https://spring.io/projects/spring-data-jpa
- MyBatis : https://mybatis.org/

Outils utiles :

- HikariCP (pool de connexions) : https://github.com/brettwooldridge/HikariCP
- Flyway (migrations) : https://flywaydb.org/
- Liquibase (gestion de schéma) : https://www.liquibase.org/

---

14. Annexes

14.1 Configuration Complète avec Fichier Properties

database.properties :

properties

```properties
# Configuration MySQL
db.url=jdbc:mysql://localhost:3306/ma_base
db.driver=com.mysql.cj.jdbc.Driver
db.username=root
db.password=rootpassword

# Configuration du pool
db.pool.minSize=5
db.pool.maxSize=20
db.pool.timeout=30000

# Options de connexion
db.useSSL=false
db.allowPublicKeyRetrieval=true
db.serverTimezone=UTC
```

Chargement de la configuration :

java

```java
public class DatabaseConfig {
    private static final Properties properties = new Properties();

    static {
        try (InputStream input = DatabaseConfig.class
                .getClassLoader()
                .getResourceAsStream("database.properties")) {

            if (input == null) {
                throw new RuntimeException("Fichier database.properties introuvable");
            }
            properties.load(input);

        } catch (IOException e) {
            throw new RuntimeException("Erreur lors du chargement de la configuration", e);
        }
    }

    public static String getUrl() {
        return properties.getProperty("db.url") +
                "?useSSL=" + properties.getProperty("db.useSSL") +
                "&allowPublicKeyRetrieval=" + properties.getProperty("db.allowPublicKeyRetrieval") +
                "&serverTimezone=" + properties.getProperty("db.serverTimezone");
    }

    public static String getUsername() {
        return properties.getProperty("db.username");
    }

    public static String getPassword() {
        return properties.getProperty("db.password");
    }
}
```

Utilisation :

```java
```

```
public class DatabaseManager {
    private final Connection connection;

    public DatabaseManager() throws SQLException {
        this.connection = DriverManager.getConnection(
            DatabaseConfig.getUrl(),
            DatabaseConfig.getUsername(),
            DatabaseConfig.getPassword()
        );
    }
}
```

14.2 Exemple Complet avec Plusieurs Entités

Modèle Client :

java

```java
public class Client {
    private int id;
    private String nom;
    private String prenom;
    private String email;

    // Constructeurs
    public Client() {}

    public Client(String nom, String prenom, String email) {
        this.nom = nom;
        this.prenom = prenom;
        this.email = email;
    }

    // Getters et setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getNom() { return nom; }
    public void setNom(String nom) { this.nom = nom; }

    public String getPrenom() { return prenom; }
    public void setPrenom(String prenom) { this.prenom = prenom; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    @Override
    public String toString() {
        return String.format("Client{id=%d, nom='%s', prenom='%s', email='%s'}",
                             id, nom, prenom, email);
    }
}
```

Modèle Commande :

```java
```

```java
public class Commande {
    private int id;
    private int clientId;
    private LocalDate dateCommande;
    private BigDecimal montant;

    // Constructeurs
    public Commande() {}

    public Commande(int clientId, LocalDate dateCommande, BigDecimal montant) {
        this.clientId = clientId;
        this.dateCommande = dateCommande;
        this.montant = montant;
    }

    // Getters et setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public int getClientId() { return clientId; }
    public void setClientId(int clientId) { this.clientId = clientId; }

    public LocalDate getDateCommande() { return dateCommande; }
    public void setDateCommande(LocalDate dateCommande) { this.dateCommande = dateCommande; }

    public BigDecimal getMontant() { return montant; }
    public void setMontant(BigDecimal montant) { this.montant = montant; }

    @Override
    public String toString() {
        return String.format("Commande{id=%d, clientId=%d, date=%s, montant=%.2f}",
                             id, clientId, dateCommande, montant);
    }
}
```

DAO Client :

```java
```

```
public class ClientDAO {
    private DatabaseManager dbManager;

    public ClientDAO(DatabaseManager dbManager) {
        this.dbManager = dbManager;
    }

    public void createTable() throws SQLException {
        String sql = """
            CREATE TABLE IF NOT EXISTS clients (
                id INT AUTO_INCREMENT PRIMARY KEY,
                nom VARCHAR(50) NOT NULL,
                prenom VARCHAR(50) NOT NULL,
                email VARCHAR(100) UNIQUE NOT NULL
            )
        """;

        try (Statement stmt = dbManager.getConnection().createStatement()) {
            stmt.execute(sql);
            System.out.println("✅ Table 'clients' créée");
        }
    }

    public void insert(Client client) throws SQLException {
        String sql = "INSERT INTO clients (nom, prenom, email) VALUES (?, ?, ?)";

        try (PreparedStatement pstmt = dbManager.getConnection()
                .prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {

            pstmt.setString(1, client.getNom());
            pstmt.setString(2, client.getPrenom());
            pstmt.setString(3, client.getEmail());
            pstmt.executeUpdate();

            // Récupérer l'ID auto-généré
            try (ResultSet rs = pstmt.getGeneratedKeys()) {
                if (rs.next()) {
                    client.setId(rs.getInt(1));
                }
            }

            System.out.println("✅ Client inséré : " + client);
        }
    }

    public List<Client> findAll() throws SQLException {
```

```
        List<Client> clients = new ArrayList<>();
        String sql = "SELECT * FROM clients";

        try (Statement stmt = dbManager.getConnection().createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                clients.add(mapResultSetToClient(rs));
            }
        }
        return clients;
    }

    public Client findById(int id) throws SQLException {
        String sql = "SELECT * FROM clients WHERE id = ?";

        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setInt(1, id);

            try (ResultSet rs = pstmt.executeQuery()) {
                if (rs.next()) {
                    return mapResultSetToClient(rs);
                }
            }
        }
        return null;
    }

    private Client mapResultSetToClient(ResultSet rs) throws SQLException {
        Client client = new Client();
        client.setId(rs.getInt("id"));
        client.setNom(rs.getString("nom"));
        client.setPrenom(rs.getString("prenom"));
        client.setEmail(rs.getString("email"));
        return client;
    }
}
```

DAO Commande :

java

```
public class CommandeDAO {
    private DatabaseManager dbManager;

    public CommandeDAO(DatabaseManager dbManager) {
        this.dbManager = dbManager;
    }

    public void createTable() throws SQLException {
        String sql = """
            CREATE TABLE IF NOT EXISTS commandes (
                id INT AUTO_INCREMENT PRIMARY KEY,
                client_id INT NOT NULL,
                date_commande DATE NOT NULL,
                montant DECIMAL(10, 2) NOT NULL,
                FOREIGN KEY (client_id) REFERENCES clients(id)
            )
        """;

        try (Statement stmt = dbManager.getConnection().createStatement()) {
            stmt.execute(sql);
            System.out.println(" ✅ Table 'commandes' créée");
        }
    }

    public void insert(Commande commande) throws SQLException {
        String sql = "INSERT INTO commandes (client_id, date_commande, montant) VALUES (?, ?, ?)";

        try (PreparedStatement pstmt = dbManager.getConnection()
                .prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {

            pstmt.setInt(1, commande.getClientId());
            pstmt.setDate(2, Date.valueOf(commande.getDateCommande()));
            pstmt.setBigDecimal(3, commande.getMontant());
            pstmt.executeUpdate();

            try (ResultSet rs = pstmt.getGeneratedKeys()) {
                if (rs.next()) {
                    commande.setId(rs.getInt(1));
                }
            }

            System.out.println(" ✅ Commande insérée : " + commande);
        }
    }

    public List<Commande> findByClientId(int clientId) throws SQLException {
```

```
        List<Commande> commandes = new ArrayList<>();
        String sql = "SELECT * FROM commandes WHERE client_id = ?";

        try (PreparedStatement pstmt = dbManager.getConnection().prepareStatement(sql)) {
            pstmt.setInt(1, clientId);

            try (ResultSet rs = pstmt.executeQuery()) {
                while (rs.next()) {
                    commandes.add(mapResultSetToCommande(rs));
                }
            }
        }
        return commandes;
    }

    private Commande mapResultSetToCommande(ResultSet rs) throws SQLException {
        Commande commande = new Commande();
        commande.setId(rs.getInt("id"));
        commande.setClientId(rs.getInt("client_id"));
        commande.setDateCommande(rs.getDate("date_commande").toLocalDate());
        commande.setMontant(rs.getBigDecimal("montant"));
        return commande;
    }
}
```

Application complète :

java

```
public class ECommerceApp {
    public static void main(String[] args) {
        DatabaseManager dbManager = null;

        try {
            // Initialisation
            dbManager = new DatabaseManager();
            ClientDAO clientDAO = new ClientDAO(dbManager);
            CommandeDAO commandeDAO = new CommandeDAO(dbManager);

            // Création des tables
            clientDAO.createTable();
            commandeDAO.createTable();

            // Insertion de clients
            System.out.println("\n📝 Création de clients...");
            Client client1 = new Client("Alami", "Ahmed", "ahmed@example.ma");
            Client client2 = new Client("Bennani", "Fatima", "fatima@example.ma");

            clientDAO.insert(client1);
            clientDAO.insert(client2);

            // Insertion de commandes
            System.out.println("\n🛒 Création de commandes...");
            commandeDAO.insert(new Commande(
                client1.getId(),
                LocalDate.now(),
                new BigDecimal("450.50")
            ));
            commandeDAO.insert(new Commande(
                client1.getId(),
                LocalDate.now().minusDays(5),
                new BigDecimal("320.00")
            ));
            commandeDAO.insert(new Commande(
                client2.getId(),
                LocalDate.now(),
                new BigDecimal("125.75")
            ));

            // Affichage des commandes par client
            System.out.println("\n📊 Commandes d'Ahmed Alami :");
            List<Commande> commandesClient1 = commandeDAO.findByClientId(client1.getId());
            commandesClient1.forEach(System.out::println);

            // Calcul du total
```

```
            BigDecimal total = commandesClient1.stream()
                .map(Commande::getMontant)
                .reduce(BigDecimal.ZERO, BigDecimal::add);
            System.out.println("💰 Total des commandes : " + total + " DH");

        } catch (SQLException e) {
            System.err.println("❌ Erreur : " + e.getMessage());
            e.printStackTrace();
        } finally {
            if (dbManager != null) {
                try {
                    dbManager.close();
                } catch (SQLException e) {
                    System.err.println("Erreur fermeture : " + e.getMessage());
                }
            }
        }
    }
}
```

14.3 Script SQL Complet d'Initialisation

sql

```sql
-- Suppression des tables existantes
DROP TABLE IF EXISTS commandes;
DROP TABLE IF EXISTS clients;

-- Création de la table clients
CREATE TABLE clients (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(50) NOT NULL,
    prenom VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Création de la table commandes
CREATE TABLE commandes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    client_id INT NOT NULL,
    date_commande DATE NOT NULL,
    montant DECIMAL(10, 2) NOT NULL,
    statut ENUM('EN_ATTENTE', 'CONFIRMEE', 'LIVREE', 'ANNULEE') DEFAULT 'EN_ATTENTE',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (client_id) REFERENCES clients(id) ON DELETE CASCADE
);

-- Insertion de données de test
INSERT INTO clients (nom, prenom, email) VALUES
('El Amrani', 'Youssef', 'youssef.elamrani@example.ma'),
('Benziane', 'Sanaa', 'sanaa.benziane@example.ma'),
('Othmani', 'Hicham', 'hicham.othmani@example.ma'),
('Chahine', 'Nora', 'nora.chahine@example.ma'),
('Aziz', 'Khalid', 'khalid.aziz@example.ma');

INSERT INTO commandes (client_id, date_commande, montant, statut) VALUES
(1, '2025-11-01', 320.00, 'LIVREE'),
(1, '2025-11-10', 450.50, 'CONFIRMEE'),
(2, '2025-11-05', 125.75, 'LIVREE'),
(3, '2025-11-12', 89.99, 'EN_ATTENTE'),
(3, '2025-11-15', 300.00, 'CONFIRMEE'),
(4, '2025-11-18', 75.25, 'LIVREE'),
(5, '2025-11-20', 500.00, 'CONFIRMEE'),
(5, '2025-11-21', 230.40, 'EN_ATTENTE');

-- Vues utiles
CREATE OR REPLACE VIEW vue_commandes_clients AS
SELECT
    c.id AS commande_id,
```

```sql
    cl.nom,
    cl.prenom,
    cl.email,
    c.date_commande,
    c.montant,
    c.statut
FROM commandes c
INNER JOIN clients cl ON c.client_id = cl.id;

-- Requêtes d'analyse
-- Total des commandes par client
SELECT
    cl.id,
    cl.nom,
    cl.prenom,
    COUNT(c.id) AS nombre_commandes,
    SUM(c.montant) AS total_montant
FROM clients cl
LEFT JOIN commandes c ON cl.id = c.client_id
GROUP BY cl.id, cl.nom, cl.prenom
ORDER BY total_montant DESC;

-- Commandes du mois en cours
SELECT * FROM vue_commandes_clients
WHERE YEAR(date_commande) = YEAR(CURDATE())
  AND MONTH(date_commande) = MONTH(CURDATE());
```

---

15. Glossaire

API (Application Programming Interface) : Interface permettant à deux applications de communiquer entre elles.

Batch : Technique permettant d'exécuter plusieurs requêtes SQL en une seule fois pour améliorer les performances.

Connection : Objet représentant une connexion active à une base de données.

DAO (Data Access Object) : Pattern de conception séparant la logique d'accès aux données de la logique métier.

Driver JDBC : Bibliothèque permettant à Java de communiquer avec un SGBD spécifique.

DriverManager : Classe JDBC gérant les drivers et établissant les connexions.

Injection de dépendances : Technique consistant à fournir les dépendances d'un objet depuis l'extérieur plutôt que de les créer en interne.

JDBC (Java Database Connectivity) : API standard Java pour l'accès aux bases de données relationnelles.

ORM (Object-Relational Mapping) : Technique permettant de mapper des objets Java sur des tables de base de données.

Pool de connexions : Ensemble de connexions maintenues ouvertes et réutilisées pour améliorer les performances.

PreparedStatement : Interface JDBC permettant d'exécuter des requêtes SQL précompilées avec des paramètres.

ResultSet : Objet contenant les résultats d'une requête SELECT.

SQLException : Exception levée lors d'erreurs d'accès à la base de données.

Statement : Interface JDBC pour exécuter des requêtes SQL simples.

Transaction : Ensemble d'opérations SQL exécutées comme une unité atomique.

Try-with-resources : Structure Java permettant la fermeture automatique des ressources.

Conclusion Finale

Ce chapitre a fourni une base solide pour comprendre et maîtriser l'accès aux bases de données en Java avec JDBC. Vous avez appris :

- Les fondamentaux de JDBC et son architecture
- Le rôle central du DriverManager
- Comment structurer une application avec DatabaseManager et le pattern DAO
- Les bonnes pratiques essentielles pour un code robuste et maintenable
- Les chemins d'évolution vers des frameworks plus avancés

L'apprentissage de JDBC est une étape essentielle, même si vous utiliserez éventuellement des frameworks comme JPA/Hibernate. Comprendre JDBC vous permet de :

- Mieux appréhender le fonctionnement des ORM
- Déboguer efficacement les problèmes de base de données
- Optimiser les requêtes SQL quand nécessaire
- Faire des choix architecturaux éclairés

Continuez à pratiquer, expérimentez avec différents patterns, et n'hésitez pas à explorer les frameworks modernes une fois les bases solidement acquises.

Bonne programmation ! 🚀