TD/TP R3.07
SQL dans un langage de prog
2. Approfondissement JDBC et bonnes pratiques

2025/2026

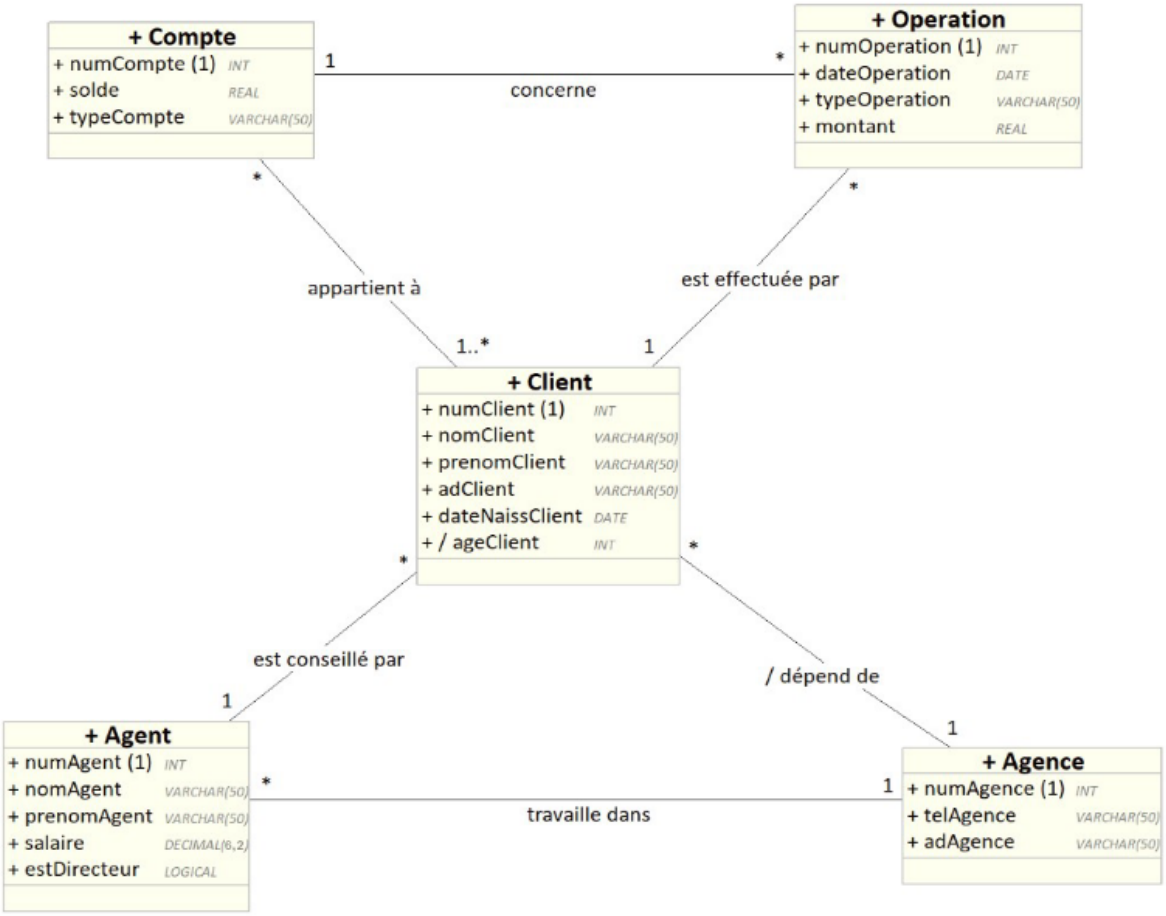
1 Introduction

Dans ce TP, vous allez approfondir l'utilisation de JDBC en ajoutant des contraintes utiles à votre base de données, en sécurisant les informations sensibles avec un fichier `.env`, et en implémentant une classe associative pour modéliser la relation entre `Compte` et `Client`. Vous apprendrez également à réaliser des `INSERT` propres en utilisant des transactions et la gestion d'exceptions.

2 Prérequis

— Avoir terminé le TP précédent (JDBC et Docker).
— Connaître les bases des transactions SQL.
— Savoir utiliser les exceptions en Java.

3 Diagramme de la base de données



Note : Une nouvelle table ***`Compte_Client`*** *doit être ajoutée pour modéliser la relation entre* ***`Compte`*** *et* ***`Client`****.*

4 Description des nouvelles contraintes et tables

4.1 Contraintes à ajouter

Vous allez modifier le schéma de la base de données en ajoutant les contraintes suivantes :

- `NOT NULL` sur les colonnes obligatoires.
- `CHECK` sur `ageClient`, `solde`, `montant`, `salaire`.
- `DEFAULT` pour `estDirecteur` (valeur par défaut : `false`).
- `UNIQUE` pour `telAgence`.

Ces contraintes seront ajoutées via des requêtes `ALTER TABLE`, ce qui permet de modifier le schéma sans tout recréer.

4.2 Classe associative `Compte_Client`

Cette table permettra de gérer la relation plusieurs-à-plusieurs entre `Compte` et `Client` :

```
CREATE TABLE IF NOT EXISTS Compte_Client (
    numCompte INT REFERENCES Compte(numCompte),
    numClient INT REFERENCES Client(numClient),
    FOREIGN KEY (numCompte) REFERENCES Compte(numCompte),
    FOREIGN KEY (numClient) REFERENCES Client(numClient),
    PRIMARY KEY (numCompte, numClient)
    );
```

5 Mise à jour de la base de données

5.1 Étape 1 : Ajouter les contraintes avec `ALTER TABLE`

Modifiez le fichier `init.sql` pour ajouter les contraintes après la création des tables :

```sql
-- Exemple pour la table Client
ALTER TABLE Client
    ALTER COLUMN nomClient SET NOT NULL,
    ALTER COLUMN prenomClient SET NOT NULL,
    ADD CONSTRAINT chk_age CHECK (ageClient >= 18);

-- Exemple pour la table Agence
ALTER TABLE Agence
    ADD CONSTRAINT unique_telAgence UNIQUE (telAgence);

-- Exemple pour la table Agent
ALTER TABLE Agent
    ALTER COLUMN estDirecteur SET DEFAULT false;
```

Grace à cette écriture différentielle on modifie uniquement ce qui change, ce qui est plus maintenable et moins risqué.

5.2 Étape 2 : Créer un fichier `.env`

Pour sécuriser les informations sensibles, créez un fichier `.env` à la racine de votre projet, qu'il ne faudra **jamais versionner** :

```
DB_URL=jdbc:postgresql://db:5432/banque_db
DB_USER=user
DB_PASSWORD=password
```

D'abord modifier le docker-compose.yml pour intégrer le fichier .env à la place des variables d'environnement déclarées :

1. supprimer les lignes suivantes :

```
environment:
      DB_URL: jdbc:postgresql://db:5432/banque_db
      DB_USER: user
      DB_PASSWORD: password
```

2. ajouter la ligne `env_file` comme suit (le faire pour les 2 services) :

```
services:
  app:
    build: .
    depends_on:
      - db
    ports:
      - "8080:8080"
    env_file:
      - .env
```

Les variables définies dans `.env` (comme `DB_USER`, `DB_PASSWORD`, etc.) seront automatiquement chargées par Docker. Assurez-vous que les noms des variables dans `.env` correspondent à ceux attendus par votre application (par exemple, `POSTGRES_USER` pour le service `db`).

Dans votre application, utilisez `System.getenv("{nom de variable}")` pour charger les variables d'environnement. De cette manière, on ne les définit qu'à un seul endroit, *i.e.* dans le fichier `.env`

6 Insertions propres avec transactions et gestion d'exceptions

6.1 Exemple de code Java

Voici comment réaliser un `INSERT` propre :

```java
public void ajouterClient(Client client) throws SQLException {
    String sql = "INSERT INTO Client (numClient, nomClient, prenomClient, adClient,
    dateNaissClient, ageClient, numAgent) VALUES (?, ?, ?, ?, ?, ?, ?)";

    try (Connection conn = getConnection();
```

```
         PreparedStatement pstmt = conn.prepareStatement(sql)) {
        // Verification manuelle avant insertion
        if (clientExisteDeja(client.getNumClient())) {
            throw new SQLException("Client deja existant !");
        }

        // Debut de la transaction
        conn.setAutoCommit(false);

        pstmt.setInt(1, client.getNumClient());
        pstmt.setString(2, client.getNomClient());
        pstmt.setString(3, client.getPrenomClient());
        pstmt.setString(4, client.getAdClient());
        pstmt.setDate(5, client.getDateNaissClient());
        pstmt.setInt(6, client.getAgeClient());
        pstmt.setInt(7, client.getNumAgent());

        pstmt.executeUpdate();
        conn.commit(); // Validation de la transaction

    } catch (SQLException e) {
        conn.rollback(); // Annulation en cas d'erreur
        throw e;
    }
}

private boolean clientExisteDeja(int numClient) throws SQLException {
    String sql = "SELECT 1 FROM Client WHERE numClient = ?";
    try (PreparedStatement pstmt = getConnection().prepareStatement(sql)) {
        pstmt.setInt(1, numClient);
        return pstmt.executeQuery().next();
    }
}
```

Pour faire les choses encore plus proprement, on ne se contente pas de traiter les requêtes de type `SQLException`. On Distingue les erreurs selon leur type (ex : violation de contrainte, connexion perdue). Par exemple :

```
try {
    // ...
} catch (SQLIntegrityConstraintViolationException e) {
    System.err.println("Violation de contrainte : " + e.getMessage());
} catch (SQLException e) {
    System.err.println("Erreur SQL : " + e.getMessage());
}
```

7 Tâches à réaliser

7.1 Méthodes à implémenter

Ajoutez les méthodes suivantes dans `BanqueDAO` :

— `ajouterCompteClient(int numCompte, int numClient)` : pour gérer la relation entre un compte et un client.

— `obtenirComptesParClient(int numClient)` : pour récupérer tous les comptes d'un client.

7.2 Classes métier mises à jour

Ajoutez la classe `CompteClient` :

```
public class CompteClient {
    private int numCompte;
    private int numClient;
    // Constructeurs, getters et setters
}
```

7.3 Conseils

— Testez chaque contrainte avec des requêtes SQL avant de les ajouter.

— Utilisez des `PreparedStatement` pour éviter les injections SQL.

— Validez vos transactions uniquement après avoir vérifié les données.

7.4 Exercice supplémentaire : Ajout de logs avec Log4j2

Pour améliorer le débogage et la maintenance de votre application, vous allez intégrer la bibliothèque **Log4j2** pour tracer les opérations critiques (connexions, requêtes, erreurs).

7.4.1 Étape 1 : Ajouter la dépendance Maven

Modifiez votre fichier `pom.xml` pour inclure Log4j2 :

```
<dependencies>
    <!-- Dependance existante pour JDBC/PostgreSQL -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.6.0</version>
    </dependency>

    <!-- Ajoutez Log4j2 -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.20.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.20.0</version>
    </dependency>
</dependencies>
```

7.4.2 Étape 2 : Configurer Log4j2

Créez un fichier `log4j2.xml` dans `src/main/resources` :

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/
    >
        </Console>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
        <Logger name="com.votreprojet.dao" level="debug" additivity="false">
            <AppenderRef ref="Console"/>
        </Logger>
    </Loggers>
</Configuration>
```

7.4.3 Étape 3 : Instrumenter la classe `BanqueDAO`

Modifiez la classe `BanqueDAO` pour ajouter des logs aux méthodes existantes :

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class BanqueDAO {
    private static final Logger logger = LogManager.getLogger(BanqueDAO.class);
    private Connection connection;

```

```
    public BanqueDAO(Connection connection) {
        this.connection = connection;
        logger.info("Initialisation de BanqueDAO avec une nouvelle connexion");
    }

    public void ajouterClient(Client client) throws SQLException {
        logger.debug("Tentative d'ajout du client : {}", client.getNomClient());

        String sql = "INSERT INTO Client (numClient, nomClient, prenomClient, adClient,
    dateNaissClient, ageClient, numAgent) VALUES (?, ?, ?, ?, ?, ?, ?)";

        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setInt(1, client.getNumClient());
            pstmt.setString(2, client.getNomClient());
            pstmt.setString(3, client.getPrenomClient());
            pstmt.setString(4, client.getAdClient());
            pstmt.setDate(5, client.getDateNaissClient());
            pstmt.setInt(6, client.getAgeClient());
            pstmt.setInt(7, client.getNumAgent());

            int rowsAffected = pstmt.executeUpdate();
            logger.info("Client {} ajoute avec succes. Lignes affectees : {}", client.
    getNomClient(), rowsAffected);

        } catch (SQLException e) {
            logger.error("Erreur lors de l'ajout du client {} : {}", client.getNomClient
    (), e.getMessage());
            throw e;
        }
    }

    public Client obtenirClientParNum(int numClient) throws SQLException {
        logger.debug("Recherche du client avec numClient = {}", numClient);

        String sql = "SELECT * FROM Client WHERE numClient = ?";
        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setInt(1, numClient);
            ResultSet rs = pstmt.executeQuery();

            if (rs.next()) {
                Client client = new Client(
                    rs.getInt("numClient"),
                    rs.getString("nomClient"),
                    rs.getString("prenomClient"),
                    rs.getString("adClient"),
                    rs.getDate("dateNaissClient"),
                    rs.getInt("ageClient"),
                    rs.getInt("numAgent")
                );
                logger.info("Client trouve : {}", client.getNomClient());
                return client;
            } else {
                logger.warn("Aucun client trouve avec numClient = {}", numClient);
                return null;
            }
        }
    }
}
```

7.4.4 Étape 4 : Tester les logs

— Exécutez votre application et observez les logs dans la console.

— Vérifiez que les messages de niveau `INFO` (ajout réussi) et `ERROR` (erreur) apparaissent correctement.

— Forcez une erreur (ex : violation de contrainte) et vérifiez que le log d'erreur s'affiche.

7.4.5 Questions

1. Quel est l'intérêt d'utiliser des niveaux de log différents (`DEBUG`, `INFO`, `ERROR`) ?
2. Comment modifieriez-vous la configuration pour écrire les logs dans un fichier `app.log` en plus de la console ?

3. Ajoutez des logs dans la méthode `ajouterOperation` pour tracer le montant et le type d'opération.

Indice : Pour écrire dans un fichier, ajoutez un `FileAppender` dans `log4j2.xml` :

```
<Appenders>
    <File name="File" fileName="logs/app.log">
        <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </File>
</Appenders>
```