TP JDBC — Requêtes, transactions et métadonnées

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-11

Table des matières

1 Comment exécuter ce TP 1
2 E1 — La méthode qui manque 2
3 E2 — Tuez l'injection 3
4 E3 — Tout ou rien 4
5 E4 — Mille inserts, une bourrasque 4
6 E5 — Le ResultSet se raconte 5
7 Validation 6
8 Compte rendu 6

1 Comment exécuter ce TP

Toutes les cellules de ce notebook sont exécutées, y compris les squelettes : ils compilent tels quels et retournent des valeurs par défaut (liste vide, `false`) tant que les `TODO` ne sont pas complétés. Modifiez les cellules marquées TODO, puis réexécutez tout le notebook depuis le début pour repartir d'un état de base de données propre. La preuve de complétion est le rendu CI : la section Validation doit afficher OK-01 à OK-05. Attention, piège connu de la flotte : l'option `//| execute: false` est ignorée pour les cellules `{java}` — toute cellule du fichier s'exécute réellement au rendu.

```
21:26:23.311  [IJava-executor-0]  WARN    o.a.h.c.p.ResponseProcessCookies  --
Invalid cookie header: "set-cookie: __cf_bm=VSUnSds4tT_S1GNsS5Xvc4T.NqPA8IOKXMv4kk7TTj0-1794432383.
x_dkJSR9Fc0qMoH7n3bM49A.jmdDUGEms96PNXQzjkspNeLkxwKxyzJt_C.qleCSpRX9W6yDTLVHEPWAlADIrrTYXug0ihpzBH
HttpOnly;       SameSite=None;       Partitioned;       Secure;       Path=/;
Domain=repo.maven.apache.org; Expires=Wed, 11 Nov 2026 21:56:23 GMT". Invalid
'expires' attribute: Wed, 11 Nov 2026 21:56:23 GMT
21:26:23.591                                                [IJava-executor-0]
WARN  o.a.h.c.p.ResponseProcessCookies -- Invalid cookie header: "set-cookie:
```

```
__cf_bm=ZxDJXdDmsQMrSGJyaPzt4EeGdUzEIvf.bHqCsqKgUXM-1794432383.570451-1.0.1.1-oCRBC7HvFgvz8Kj.HqaC
HttpOnly;       SameSite=None;       Partitioned;       Secure;       Path=/;
Domain=repo.maven.apache.org; Expires=Wed, 11 Nov 2026 21:56:23 GMT". Invalid
'expires' attribute: Wed, 11 Nov 2026 21:56:23 GMT
```

```
try (Connection c = DriverManager.getConnection(URL, USER, PASS);
     Statement st = c.createStatement()) {
      st.execute("CREATE TABLE IF NOT EXISTS ARRET (ID INT PRIMARY KEY, NOM
VARCHAR(80) NOT NULL, LIGNE VARCHAR(10) NOT NULL)");
            st.execute("INSERT   INTO   ARRET   VALUES   (1,'Belleville','11'),
(2,'Bastille','1'),(3,'Réaumur-Sébastopol','4'),(4,'Gare     de     Lyon','1'),
(5,'Châtelet','7'),(6,'Nation','1'),(7,'Auteuil','10'),(8,'La Muette','9')");
    st.execute("CREATE TABLE IF NOT EXISTS COMPTE (LIBELLE VARCHAR(20) PRIMARY
KEY, SOLDE INT NOT NULL)");
    st.execute("INSERT INTO COMPTE VALUES ('courant', 5000), ('livret', 3000)");
    st.execute("CREATE TABLE IF NOT EXISTS TRAJET (ID INT AUTO_INCREMENT PRIMARY
KEY, ARRET_ID INT, HEURE VARCHAR(5))");
     System.out.println("Tables créées : ARRET (8 lignes), COMPTE (2 lignes),
TRAJET (0)");
}
```

```
Tables créées : ARRET (8 lignes), COMPTE (2 lignes), TRAJET (0)
```

2 E1 — La méthode qui manque

Un DAO JDBC se lit en trois temps : ouvrir une connexion, préparer la requête avec des paramètres liés, parcourir le `ResultSet`. Complétez la méthode ci-dessous en exécutant la requête préparée et en alimentant la liste `resultat` avec la colonne `NOM`, en respectant le try-with-resources déjà en place.

```
List<String> nomsDeLaLigne(String ligne) throws SQLException {
    var sql = "SELECT NOM FROM ARRET WHERE LIGNE = ? ORDER BY NOM";
    var resultat = new ArrayList<String>();
    try (Connection c = DriverManager.getConnection(URL, USER, PASS);
         PreparedStatement ps = c.prepareStatement(sql)) {
        ps.setString(1, ligne);
        // TODO: exécuter la requête, parcourir le ResultSet, alimenter resultat
    }
    return resultat;
}
```

```
System.out.println(nomsDeLaLigne("1"));
```

```
[]
```

Résultat attendu une fois le TODO complété : `[Bastille, Gare de Lyon, Nation]`.

3 E2 — Tuez l'injection

La méthode suivante construit sa requête par concaténation : c'est la faille d'injection SQL classique. Observez d'abord son comportement avec une entrée malveillante, puis écrivez la version sûre `nomExact` avec un paramètre lié. Comme en E1, la version vide retourne une liste vide par défaut.

```
List<String> chercherParNomVulnerable(String nom) throws SQLException {
    var sql = "SELECT NOM FROM ARRET WHERE NOM = '" + nom + "'";
    var resultat = new ArrayList<String>();
    try (Connection c = DriverManager.getConnection(URL, USER, PASS);
         Statement st = c.createStatement();
         ResultSet rs = st.executeQuery(sql)) {
        while (rs.next()) resultat.add(rs.getString("NOM"));
    }
    return resultat;
}
```

```
List<String> nomExact(String nom) throws SQLException {
    var sql = "SELECT NOM FROM ARRET WHERE NOM = ?";
    var resultat = new ArrayList<String>();
    try (Connection c = DriverManager.getConnection(URL, USER, PASS);
         PreparedStatement ps = c.prepareStatement(sql)) {
        ps.setString(1, nom);
        // TODO: exécuter la requête, parcourir le ResultSet, alimenter resultat
    }
    return resultat;
}
```

```
System.out.println(chercherParNomVulnerable("' OR '1'='1").size());
System.out.println(nomExact("' OR '1'='1").size());
```

```
8
0
```

Résultat attendu : `8` (tous les arrêts exfiltrés), puis `0` (aucun arrêt ne porte ce nom).

4 E3 — Tout ou rien

Écrivez un virement atomique entre deux comptes : soit les deux `UPDATE` sont validés, soit aucun ne l'est. Le compte débité ne doit jamais descendre sous zéro. Le gestionnaire `solde` est fourni ; complétez `transferer` en pilotant vous-même le commit.

```
boolean transferer(String de, String vers, int montant) throws SQLException {
    try (Connection c = DriverManager.getConnection(URL, USER, PASS)) {
        // TODO: autoCommit=false, deux UPDATE préparés, vérification du solde,
commit/rollback
        return false;
    }
}
```

```
System.out.println("Avant  :  courant="  +  solde("courant")  +  "  livret="  +
solde("livret"));
System.out.println("Virement 1000 : " + transferer("courant", "livret", 1000));
System.out.println("Après  :  courant="  +  solde("courant")  +  "  livret="  +
solde("livret"));
transferer("courant", "livret", 999999);
System.out.println("Final  :  courant="  +  solde("courant")  +  "  livret="  +
solde("livret"));
```

```
Avant : courant=5000 livret=3000
Virement 1000 : false
Après : courant=5000 livret=3000
Final : courant=5000 livret=3000
```

Résultat attendu : `5000/3000`, puis virement validé et `4000/4000`, puis solde inchangé après le virement impossible.

5 E4 — Mille inserts, une bourrasque

Insérer 1000 lignes une par une coûte 1000 allers-retours vers le moteur. La mise à jour en lot (`addBatch` / `executeBatch`) les regroupe en une seule bourrasque. Chronométrez la version ligne par ligne, puis comparez avec la version en lot fournie.

```
var debut = System.nanoTime();
try (Connection c = DriverManager.getConnection(URL, USER, PASS);
   PreparedStatement ps = c.prepareStatement("INSERT INTO TRAJET(ARRET_ID,HEURE)
VALUES(?,?)")) {
    for (int i = 0; i < 1000; i++) {
        ps.setInt(1, (i % 8) + 1);
```

```
        ps.setString(2, "12:00");
        ps.executeUpdate(); // TODO: remplacer par addBatch() + executeBatch()
après la boucle
    }
}
System.out.println("Un par un : " + (System.nanoTime() - debut) / 1_000_000 +
" ms");
```

```
Un par un : 199 ms
```

```
try (Connection c = DriverManager.getConnection(URL, USER, PASS);
        Statement st = c.createStatement()) { st.executeUpdate("DELETE FROM
TRAJET"); }
var debutBatch = System.nanoTime();
try (Connection c = DriverManager.getConnection(URL, USER, PASS);
   PreparedStatement ps = c.prepareStatement("INSERT INTO TRAJET(ARRET_ID,HEURE)
VALUES(?,?)")) {
    for (int i = 0; i < 1000; i++) {
        ps.setInt(1, (i % 8) + 1);
        ps.setString(2, "12:00");
        ps.addBatch();
    }
    ps.executeBatch();
}
System.out.println("Par lot : " + (System.nanoTime() - debutBatch) / 1_000_000
+ " ms");
```

```
Par lot : 91 ms
```

```
Lignes dans TRAJET : 1000
```

Résultat attendu après chaque version : `Lignes dans TRAJET : 1000`.

6 E5 — Le ResultSet se raconte

Un `ResultSet` sait décrire sa propre structure : demandez ses métadonnées colonnes, puis faites reculer son curseur scrollable de la dernière ligne à la première.

```
try (Connection c = DriverManager.getConnection(URL, USER, PASS);
         Statement st = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
     ResultSet rs = st.executeQuery("SELECT * FROM ARRET ORDER BY NOM")) {
```

```
    ResultSetMetaData md = rs.getMetaData();
    // TODO 1 : pour i de 1 à md.getColumnCount(), afficher nom + type JDBC
      // TODO 2 : rs.last(), puis rs.previous() jusqu'à rs.isBeforeFirst(),
afficher NOM
}
```

Résultat attendu : `ID INTEGER | NOM VARCHAR | LIGNE VARCHAR`, puis les 8 noms en ordre alphabétique inverse. Point de discussion : les noms de colonnes apparaissent en MAJUSCULES car H2 v2 stocke les identifiants non quotés en uppercase — JDBC renvoie les filtres tels quels, contrairement à PostgreSQL nativement en minuscules.

7 Validation

Exécutez toutes les cellules depuis le début. Chaque ligne ci-dessous doit afficher OK.

```
OK-01 : false
OK-02 : true
OK-03 : false
OK-04 : true
OK-05 : true
```

8 Compte rendu

Exportez le notebook rendu. Vous déposez : la sortie de la cellule de vérification de E2 (`8` puis `0`) accompagnée de trois phrases expliquant pourquoi un paramètre lié stoppe l'injection là où la concaténation ouvre une porte ; ainsi que les lignes de E3 assurant le contrôle transactionnel (`setAutoCommit(false)`, `commit()`, `rollback()`), à mettre en évidence dans votre code soumis.