Accéder à une base relationnelle depuis Java : JDBC

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

2026-11-02

Table des matières

1 Le driver et la base de travail 2
1.1 Où tourne cette base ? 2
2 Première connexion : `DriverManager` et `try-with-resources` 3
3 Du listing d'arrêts à la table 3
4 Du `ResultSet` à l'objet : le premier DAO 4
5 L'injection SQL : pourquoi la concaténation est une faille 5
6 Transactions : tout ou rien 7
7 Le lot (batch) : mille requêtes en une poignée d'échanges 9
8 Le contrat des collections, version DAO 9
9 Portabilité : le même code DAO sur H2 10
10 Conclusion 11

À la leçon précédente, nous manipulions une liste de noms d'arbres — Pommier, Chêne — construite à la main en mémoire. Cette liste est évaporée dès que le programme se termine. Or la réutilisation du code, fil conducteur de tout le cours, passe souvent par des données qui survivent à l'exécution : celles d'un **système de gestion de bases de données relationnelles** (SGBDR). JDBC (Java Database Connectivity) est la couture standard entre les deux mondes : les objets Java d'un côté, les tables SQL de l'autre.

JDBC est une API d'interface, pas une implémentation : `java.sql` définit les contrats (Connection, Statement, ResultSet…), le driver du fournisseur les respecte. Cette leçon suit un arc unique : obtenir une connexion, lire des données en objets Java, puis durcir le code — requêtes paramétrées contre l'injection, transactions atomiques, inserts par lots — et enfin vérifier la portabilité du code d'accès aux données entre deux SGBD distincts.

1 Le driver et la base de travail

Comme toute bibliothèque externe, le driver doit être au classpath. En gestion de dépendances Maven, une simple coordonnée suffit — le noyau Java, lui, n'a **pas** besoin d'enregistrement explicite du driver : depuis JDBC 4.0, `DriverManager.getConnection` associe l'URL au driver approprié automatiquement (les anciens `Class.forName("org.postgresql.Driver")` sont historiques).

```
%maven org.postgresql:postgresql:42.7.3
```

```
07:46:03.365                                                    [IJava-executor-0]
WARN  o.a.h.c.p.ResponseProcessCookies -- Invalid cookie header: "set-cookie:
__cf_bm=t7VJgmW0eeR4qCrLyz0bZLufWvr0KBQf4Bw2vXEXVsA-1793605563.334065-1.0.1.1-IZKnjUxMH_Thi.Klvft
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Mon, 02 Nov 2026 08:16:03 GMT". Invalid 'expires' attribute: Mon, 02 Nov
2026 08:16:03 GMT
07:46:03.465  [IJava-executor-0]  WARN    o.a.h.c.p.ResponseProcessCookies  --
Invalid cookie header: "set-cookie: __cf_bm=kt0TnkQWE72f1TxVuk3zGUbJPgLnVGYNhlQM7yvST5Y-1793605563.
ItK6J5B.Tvhqzv0U8Irt__PsdMbzMZWsBxHFCPQeJcdpAWtkFvS5aqckuY62rdwmvsTFW6Yn_vCBTXfR.YynAXv8tgsCcdifU
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Mon, 02 Nov 2026 08:16:03 GMT". Invalid 'expires' attribute: Mon, 02 Nov
2026 08:16:03 GMT
07:46:03.511     [BasicRepositoryConnector-repo.maven.apache.org-0-0]     WARN
o.a.h.c.p.ResponseProcessCookies -- Invalid cookie header: "set-cookie: __cf_bm=zzWmTinrDsh0eLhdmN5
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Mon, 02 Nov 2026 08:16:03 GMT". Invalid 'expires' attribute: Mon, 02 Nov
2026 08:16:03 GMT
07:46:06.984     [BasicRepositoryConnector-repo.maven.apache.org-0-1]     WARN
o.a.h.c.p.ResponseProcessCookies -- Invalid cookie header: "set-cookie: __cf_bm=qCPzcg9l8LSIfH1PPRN
r2.5ftiJ.JJDPOhYLT.c6AnQelRMcIX3nlfP0eFVP2wF63iXCtG7EfcQIvYfk9rlbcy_iLjSKDfjdPrdYfeu1.No_3SkSjD.L
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Mon, 02 Nov 2026 08:16:03 GMT". Invalid 'expires' attribute: Mon, 02 Nov
2026 08:16:03 GMT
```

Nous chargerons aussi le driver H2 pour la section portabilité : les deux drivers cohabitent sur le classpath du notebook.

```
%maven com.h2database:h2:2.3.232
```

1.1 Où tourne cette base ?

Le noyau Java ne parle à personne tout seul : il faut un serveur. Comme pour les TP conteneurs du cours, nous lançons une instance PostgreSQL **locale au rendu**, dans un conteneur joignable depuis le noyau Java par l'adresse `dind` (le réseau

Docker de la machine d'exécution), port 5432 ; en local, cette cellule démarre et attend un conteneur Docker (détail au TP conteneurs) ; chez vous, remplacez `dind` par `localhost` — et attendez-vous à une base recréée à chaque exécution. La cellule suivante prépare la base et attend sa disponibilité ; son écriture (appels `docker` via `ProcessBuilder`, boucle de disponibilité) est de la plomberie d'environnement, sans intérêt pédagogique en soi, et donc masquée.

```
Base prête (tentative 3)
```

2 Première connexion : `DriverManager` et `try-with-resources`

Une connexion est une **ressource** coûteuse et à cycle de vie court : ouverture coûteuse, verrous côté serveur, descripteur réseau. Elle est `AutoCloseable` — la leçon sur l'IO l'avait dit pour les flux, le constat s'applique ici à l'identique : `try-with-resources` garantit la fermeture, succès ou exception. Notez que `SQLException` est une exception contrôlée : le compilateur vous force à la traiter.

```
import java.sql.*;

final String URL  = "jdbc:postgresql://dind:5432/oocol";
final String USER = "oocol";
final String PASS = "oocol";

try (Connection connexion = DriverManager.getConnection(URL, USER, PASS)) {
    DatabaseMetaData meta = connexion.getMetaData();
    System.out.printf("Connecté à %s %s%n",
                                                meta.getDatabaseProductName(),
meta.getDatabaseProductVersion());
}
```

```
Connecté à PostgreSQL 17.11
```

3 Du listing d'arrêts à la table

Trois familles de requêtes, trois méthodes : `executeQuery` pour un `SELECT` (retourne un `ResultSet`), `executeUpdate` pour `INSERT/UPDATE/DELETE/DDL` (retourne un nombre de lignes), `execute` pour le cas indéterminé. Création de la table et reprise des deux noms de la leçon 07, érigés en arrêts :

```
try (Connection connexion = DriverManager.getConnection(URL, USER, PASS);
     Statement stmt = connexion.createStatement()) {
    stmt.execute("""
```

```
        CREATE TABLE IF NOT EXISTS arrets (
            id        TEXT PRIMARY KEY,
            nom       TEXT NOT NULL,
            latitude  DOUBLE PRECISION NOT NULL,
            longitude DOUBLE PRECISION NOT NULL
        )""");
    stmt.executeUpdate("""
        INSERT INTO arrets VALUES
          ('POM', 'Pommier', 43.125, 6.032),
          ('CHE', 'Chêne',   43.130, 6.040)
        ON CONFLICT (id) DO NOTHING""");
    System.out.println("Table arrets prête.");
}
```

```
Table arrets prête.
```

4 Du `ResultSet` à l'objet : le premier DAO

`ResultSet` est un curseur ligne par ligne, accessible par nom ou index de colonne (`getString`, `getDouble`…) — un flux tabulaire, en aucun cas une collection Java. Le passage de l'un à l'autre s'appelle le **mappage**, et on l'enveloppe dans un Data Access Object (DAO) : une classe qui isole le reste du programme des détails SQL.

```
record Stop(String id, String nom, double latitude, double longitude) {}

List<Stop> lireLesArrets(Connection connexion) throws SQLException {
    List<Stop> resultat = new ArrayList<>();
    String sql = "SELECT id, nom, latitude, longitude FROM arrets ORDER BY nom";
    try (Statement stmt = connexion.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next())
            resultat.add(new Stop(rs.getString("id"), rs.getString("nom"),
                         rs.getDouble("latitude"), rs.getDouble("longitude")));
    }
    return resultat;
}

try (Connection connexion = DriverManager.getConnection(URL, USER, PASS)) {
    lireLesArrets(connexion).forEach(System.out::println);
}
```

```
Stop[id=CHE, nom=Chêne, latitude=43.13, longitude=6.04]
Stop[id=POM, nom=Pommier, latitude=43.125, longitude=6.032]
```

Le résultat est exactement la liste triée de la leçon 07 — mais alimentée par la base. Toute la suite de ce cours consomme des listes ainsi produites.

5 L'injection SQL : pourquoi la concaténation est une faille

Construire une requête par concaténation de chaînes fusionne **le code SQL et les données**. Un utilisateur malveillant peut alors injecter du SQL dans son champ de saisie. Démonstration sur une table annexe de contacts. Première attaque : la saisie `x' OR '1'='1` ferme la chaîne, ajoute une condition toujours vraie ; le guillemet final du gabarit se retrouve orphelin et referme la vôtre. La requête « filtrante » retourne **tout** :

```
String saisie = "x' OR '1'='1";
try (Connection connexion = DriverManager.getConnection(URL, USER, PASS);
     Statement stmt = connexion.createStatement()) {
    stmt.execute("DROP TABLE IF EXISTS contacts");
    stmt.execute("CREATE TABLE contacts (nom TEXT, courriel TEXT)");
    stmt.executeUpdate("""
        INSERT INTO contacts VALUES
          ('Pommier', 'pommier@oocol.fr'), ('Chêne', 'chene@oocol.fr')""");
     String sql = "SELECT nom, courriel FROM contacts WHERE nom = '" + saisie
+ "'";
    System.out.println("Requête exécutée : " + sql);
    try (ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next())
                          System.out.printf("%s  <%s>%n",  rs.getString("nom"),
rs.getString("courriel"));
    }
}
```

```
Requête exécutée : SELECT nom, courriel FROM contacts WHERE nom = 'x' OR '1'='1'
Pommier <pommier@oocol.fr>
Chêne <chene@oocol.fr>
```

Pire : une saisie peut contenir plusieurs instructions — le `--` final ouvre un commentaire SQL qui neutralise la fin du gabarit. Celle-ci détruit la table :

```
String saisieMechante = "y'; DROP TABLE contacts; --";
try (Connection connexion = DriverManager.getConnection(URL, USER, PASS);
     Statement stmt = connexion.createStatement()) {
```

```
    stmt.execute("SELECT nom FROM contacts WHERE nom = '" + saisieMechante + "'");
}
```

Et le dégât devient visible — toute requête ultérieure sur la table échoue, la base ne sait pas pardonner une table supprimée :

```
try (Connection connexion = DriverManager.getConnection(URL, USER, PASS);
     Statement stmt = connexion.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT count(*) FROM contacts")) {
    rs.next();
    System.out.println("Encore " + rs.getInt(1) + " contacts.");
}
```

```
EvalException: ERROR: relation "contacts" does not exist
  Position: 22
---------------------------------------------------------------------------
org.postgresql.util.PSQLException: ERROR: relation "contacts" does not exist
  Position: 22
   at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2725)
   at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2412)

at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:371)
    at org.postgresql.jdbc.PgStatement.executeInternal(PgStatement.java:502)
    at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:419)
    at org.postgresql.jdbc.PgStatement.executeWithFlags(PgStatement.java:341)
    at org.postgresql.jdbc.PgStatement.executeCachedSql(PgStatement.java:326)
    at org.postgresql.jdbc.PgStatement.executeWithFlags(PgStatement.java:302)
    at org.postgresql.jdbc.PgStatement.executeQuery(PgStatement.java:251)
    at .(#57:3)
```

La correction tient en un mot : **ne jamais concaténer de donnée**. Une requête `?`-paramétrée (`PreparedStatement`) transmet le SQL au serveur une fois, et les valeurs à part ; une saisie n'est plus jamais du code :

```
try (Connection connexion = DriverManager.getConnection(URL, USER, PASS)) {
    try (Statement stmt = connexion.createStatement()) {
        stmt.execute("DROP TABLE IF EXISTS contacts");
        stmt.execute("CREATE TABLE contacts (nom TEXT, courriel TEXT)");
        stmt.executeUpdate("""
            INSERT INTO contacts VALUES
              ('Pommier', 'pommier@oocol.fr'), ('Chêne', 'chene@oocol.fr')""");
    }
    String sql = "SELECT nom, courriel FROM contacts WHERE nom = ?";
```

```
     for (String s : new String[] {"Pommier", "x' OR '1'='1", "y'; DROP TABLE
contacts; --"}) {
        try (PreparedStatement p = connexion.prepareStatement(sql)) {
            p.setString(1, s);
            int n = 0;
            try (ResultSet rs = p.executeQuery()) { while (rs.next()) n++; }
          System.out.printf("Saisie %-32s → %d ligne(s)%n", "\"" + s + "\"", n);
        }
    }
    try (Statement stmt = connexion.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT count(*) FROM contacts")) {
        rs.next();
        System.out.println("Table intacte : " + rs.getInt(1) + " contacts.");
    }
}
```

```
Saisie "Pommier"                          → 1 ligne(s)
Saisie "x' OR '1'='1"                     → 0 ligne(s)
Saisie "y'; DROP TABLE contacts; --"      → 0 ligne(s)
Table intacte : 2 contacts.
```

Le `PreparedStatement` apporte en plus la précompilation côté serveur (plan d'exécution réutilisé), d'où son intérêt aussi en performance. Il n'existe aucun cas où concaténer une saisie est correcte ; les cas « légitimes » d'éléments dynamiques (noms de tables, ORDER BY) se traitent par liste blanche côté Java.

6 Transactions : tout ou rien

Par défaut chaque instruction est auto-commitée. Un virement de titres entre deux cartes est deux `UPDATE` qui n'ont de sens **que** groupés. On désactive l'auto-commit, on valide par `commit()`, on annule par `rollback()` — et la contrainte `CHECK (titres ⩾ 0)` de la table joue le rôle de l'incident :

```
try (Connection connexion = DriverManager.getConnection(URL, USER, PASS);
     Statement stmt = connexion.createStatement()) {
    stmt.execute("DROP TABLE IF EXISTS cartes");
    stmt.execute("""
        CREATE TABLE cartes (
            id        TEXT PRIMARY KEY,
            titulaire TEXT NOT NULL,
            titres    INT NOT NULL CHECK (titres ⩾ 0)
        )""");
         stmt.executeUpdate("INSERT  INTO  cartes  VALUES  ('C1','Alice',10),
```

```
('C2','Bob',4)");
}
```

```
String transferer(Connection connexion, String de, String vers, int montant)
        throws SQLException {
    connexion.setAutoCommit(false);
    try (PreparedStatement d = connexion.prepareStatement(
                "UPDATE cartes SET titres = titres - ? WHERE id = ?");
         PreparedStatement c = connexion.prepareStatement(
                "UPDATE cartes SET titres = titres + ? WHERE id = ?")) {
        d.setInt(1, montant); d.setString(2, de); d.executeUpdate();
        c.setInt(1, montant); c.setString(2, vers); c.executeUpdate();
        connexion.commit();
        return "validé";
    } catch (SQLException e) {
        connexion.rollback();
        return "annulé (" + e.getMessage().split("\n")[0] + ")";
    } finally {
        connexion.setAutoCommit(true);
    }
}

try (Connection connexion = DriverManager.getConnection(URL, USER, PASS)) {
     System.out.println("15 titres C1 → C2 : " + transferer(connexion, "C1",
"C2", 15));
     System.out.println(" 5 titres C1 → C2 : " + transferer(connexion, "C1",
"C2", 5));
    try (Statement stmt = connexion.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT titulaire, titres FROM cartes
ORDER BY id")) {
       while (rs.next()) System.out.printf("%s : %d titres%n", rs.getString(1),
rs.getInt(2));
    }
}
```

```
15 titres C1 → C2 : annulé (ERROR: new row for relation "cartes" violates check
constraint "cartes_titres_check")
 5 titres C1 → C2 : validé
Alice : 5 titres
Bob : 9 titres
```

Le premier transfert échoue (Alice n'a que 10 titres) : le `rollback()` a effacé le débit déjà exécuté — Alice n'a pas perdu 15 titres « à mi-chemin ». Le second passe. Sans

transaction, le premier cas aurait laissé une carte débitée sans contrepartie : c'est la propriété **atomique** du A de ACID.

Le `finally` qui remet `setAutoCommit(true)` est indispensable si la connexion est ensuite rendue à un pool. Une transaction doit rester courte : elle détient des verrous côté serveur.

7 Le lot (batch) : mille requêtes en une poignée d'échanges

`addBatch()` accumule des exécutions paramétrées et `executeBatch()` les envoie d'un coup — beaucoup moins d'allers-retours réseau. En production, on borne la mémoire par un `clearBatch()` périodique et des lots de taille finie (100–1000 lignes), validés d'une seule transaction. Insertion de 100 arrêts par lot :

```
try (Connection connexion = DriverManager.getConnection(URL, USER, PASS)) {
    try (Statement stmt = connexion.createStatement()) {
        stmt.execute("DROP TABLE IF EXISTS arrets_massif");
            stmt.execute("CREATE  TABLE  arrets_massif  (id  SERIAL  PRIMARY  KEY,
nom TEXT)");
    }
    connexion.setAutoCommit(false);
    try (PreparedStatement p = connexion.prepareStatement(
              "INSERT INTO arrets_massif(nom) VALUES (?)")) {
      for (int i = 1; i ⩽ 100; i++) { p.setString(1, "Arrêt " + i); p.addBatch(); }
        p.executeBatch();
        connexion.commit();
    } finally {
        connexion.setAutoCommit(true);
    }
    try (Statement stmt = connexion.createStatement();
       ResultSet rs = stmt.executeQuery("SELECT count(*) FROM arrets_massif")) {
        rs.next();
        System.out.println(rs.getInt(1) + " lignes insérées par lot.");
    }
}
```

```
100 lignes insérées par lot.
```

En production, le batch s'accompagne d'une taille de lot explicite et d'un `clearBatch()` périodique pour borner la mémoire.

8 Le contrat des collections, version DAO

Rappel de la leçon 07 : une méthode qui retourne une collection retourne une collection vide, jamais `null`. C'est en traversant la couture JDBC que le contrat se

brise le plus souvent — le code ci-dessous, hérité de pratiques anciennes, retourne `null` quand rien n'est trouvé :

```
List<Stop> chercherParNom(Connection connexion, String nom) throws SQLException {
    List<Stop> trouve = new ArrayList<>();
    try (PreparedStatement p = connexion.prepareStatement(
             "SELECT id, nom, latitude, longitude FROM arrets WHERE nom = ?")) {
        p.setString(1, nom);
        try (ResultSet rs = p.executeQuery()) {
            while (rs.next())
                trouve.add(new Stop(rs.getString(1), rs.getString(2),
                                    rs.getDouble(3), rs.getDouble(4)));
        }
    }
    return trouve.isEmpty() ? null : trouve;   // ← la mauvaise idée
}

try (Connection connexion = DriverManager.getConnection(URL, USER, PASS)) {
     for (Stop s : chercherParNom(connexion, "Laurier"))    // aucun arrêt de
ce nom
        System.out.println(s);
}
```

```
EvalException: Cannot invoke "java.util.List.iterator()" because the return value
of "REPL.$JShell$63.chercherParNom(java.sql.Connection, String)" is null
------------------------------------------------------------------------
java.lang.NullPointerException:   Cannot   invoke   "java.util.List.iterator()"
because the return value of "REPL.$JShell$63.chercherParNom(java.sql.Connection,
String)" is null
    at .(#64:1)
```

La boucle `for-each` appelle `iterator()` sur `null` — `NullPointerException`, avec le message enrichi (Java 14+) qui nomme la méthode fautive. La correction tient en une ligne : `return trouve;` — le vide s'exprime par un `ResultSet` sans ligne, donc une `List` sans élément, exactement le contrat posé à la leçon 07.

9 Portabilité : le même code DAO sur H2

JDBC promet une interface uniforme ; la réalité adoucit la promesse (le dialecte SQL reste fournisseur-spécifique). Vérifions la partie portable — le motif d'accès — sur un SGBD embarqué, H2 en mode compatibilité PostgreSQL, en mémoire :

```
final String H2_URL = "jdbc:h2:mem:oocol;MODE=PostgreSQL;DB_CLOSE_DELAY=-1";

try (Connection h2 = DriverManager.getConnection(H2_URL, "sa", "")) {
    try (Statement stmt = h2.createStatement()) {
        stmt.execute("""
            CREATE TABLE IF NOT EXISTS arrets (
                id VARCHAR(3) PRIMARY KEY,
                nom VARCHAR(50) NOT NULL,
                latitude DOUBLE PRECISION NOT NULL,
                longitude DOUBLE PRECISION NOT NULL)""");
        stmt.execute("""
            MERGE INTO arrets KEY(id) VALUES
              ('POM','Pommier',43.125,6.032), ('CHE','Chêne',43.130,6.040)""");
    }
   lireLesArrets(h2).forEach(System.out::println);   // la MÉME méthode qu'avant
}
```

```
Stop[id=CHE, nom=Chêne, latitude=43.13, longitude=6.04]
Stop[id=POM, nom=Pommier, latitude=43.125, longitude=6.032]
```

`lireLesArrets` n'a pas bougé d'un caractère : c'est tout l'intérêt de l'interface `Connection` et du DAO. Ce qui, lui, n'est **pas** portable : `ON CONFLICT` (H2 : `MERGE`), les types fins, et les fonctions spécifiques comme `ST_Distance` — une raison de concentrer le SQL dialectal dans la couche DAO, jamais dans le reste du programme.

H2 en mode PostgreSQL accepte l'essentiel du SQL courant ; `DB_CLOSE_DELAY=-1` empêche la destruction de la base mémoire entre deux connexions. Dans un vrai projet, cette portabilité se teste avec des bases jetables (rules des TP conteneurs) et des outils de migration de schéma.

```
Base arrêtée.
```

10 Conclusion

JDBC est la couture entre les objets de la leçon 07 et les données persistantes : une connexion `AutoCloseable`, un curseur `ResultSet` à mapper en collections — puis trois durcissements sans lesquels le code n'est pas professionnel : le `PreparedStatement` contre l'injection, la transaction pour l'atomicité « tout ou rien », le batch pour le volume. La portabilité est réelle au niveau de l'API et illusoire au niveau du dialecte — d'où le DAO comme lieu de confinement.

Questions de vérification :

1. Pourquoi `Class.forName` sur le driver n'est-il plus nécessaire, et qu'est-ce qui décide du driver à utiliser ?
2. Une saisie utilisateur arrive dans `WHERE nom = ?` via `setString`, puis dans une requête concaténée : lequel des deux est sûr, et pourquoi ?
3. Un transfert en deux `UPDATE` échoue au second, en auto-commit par défaut : que reste-t-il en base, et comment l'éviter ?
4. Notre `chercherParNom` historique renvoie `null` sur résultat vide. Qui le découvre, quand, et avec quelle exception ?

Pour aller plus loin (TP) : jeu complet GTFS, requêtes de proximité à l'échelle, `ResultSetMetaData`, et pool de connexions (`DataSource`) — évoqués mais hors du périmètre de cette leçon.