

Accès aux bases de données relationnelles en Java (JDBC)

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

2026-10-05

Table des matières

1 Objectifs de JDBC	1
2 Système de gestion de base de données et Dataset	1
3 Le Driver JDBC	10
4 Ouverture d'une connexion	11
5 Exécution de requêtes SQL	11
6 Les requêtes simples	12
7 Parcours des résultats	12
8 Les exceptions	13
9 Types Java/JDBC et SQL	14
10 Transaction	15
11 Précompilation des requêtes	15
12 Les transactions	16
13 Procédures stockées	18

```
//|echo: false
//|output: false
String DB_URL="jdbc:postgresql://
docker/gtfs";
String DB_USERNAME=System.getenv("DB_USERNAME");
String DB_PASSWORD=System.getenv("DB_PASSWORD");
```

1 Objectifs de JDBC

Java Database Connectivity (JDBC) est une composante essentielle de l'écosystème Java, conçue pour offrir une interaction transparente avec une variété de **Systèmes de Gestion de Bases de Données Relation-**

nelles (SGBDR). Cette API robuste est conçue pour unifier l'accès aux bases de données à travers une interface commune, exploitant la puissance du langage SQL tout en s'alignant avec les types de données Java. Avec JDBC, les développeurs peuvent établir des connexions fiables avec des bases de données, exécuter des requêtes SQL avec précision et parcourir les résultats de manière intuitive. Bien que JDBC soit suffisamment flexible pour s'adapter aux caractéristiques propres à chaque SGBDR, cette adaptabilité peut parfois se faire au détriment de la portabilité du code, nécessitant des ajustements spécifiques pour chaque système.

2 Système de gestion de base de données et Dataset

- Dans le cadre de ce cours nous utiliserons les systèmes de gestion de base données PostgreSQL. Pour les travaux pratiques n'importe quel SGBDR pourra être utilisé par exemple h2.
- H2 est un SGDB écrit en Java qui peut s'exécuter comme un serveur indépendant ou depuis un programme Java. Il peut aussi être utilisé purement en mémoire ou avec persistance sur disque.
- Pour l'utiliser il suffit de télécharger le fichier .jar de h2 par exemple la version h2-2.2.224.jar depuis un entrepôt maven.

- Le serveur peut être exécuté simplement avec la commande suivante pour autoriser les connexions tcp et la création automatique d'une base de données lors du premier accès.

```
java -cp h2*.jar \
    org.h2.tools.Server \
        -webAllowOthers \
        -tcpAllowOthers \
        -pgAllowOthers \
        -ifNotExists
```

- Pour accéder à la base de données, il suffit de se connecter à l'URL `jdbc:h2:tcp://localhost/Gtfs_RMTT;MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE` avec le nom d'utilisateur `sa` et un mot de passe vide.

```
java -cp h2*.jar \
    org.h2.tools.RunScript \
        -url "jdbc:h2:tcp://localhost/Gtfs_RMTT;MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE" \
        -user sa \
        -script data.sql
```

Pour illustrer ce cours, nous allons utiliser une base de données qui représente des données de transport (des bus) au format GTFS (General Transit Feed Specification).

- Format de données standardisé créé par Google
- Permet le partage des données de transport public
- Utilisé par Google Maps et autres applications

Nous utiliserons une extensions de postgresql pour les Systemes d'Information Géographique (SIG) PostGIS qui permet de stocker des données géographiques et de faire des requêtes spatiales.

- Les données ouvertes du réseau de transport de la ville de Toulon sont conformes au format GTFS et seront utilisées pour illustrer les concepts de JDBC :

- **stops:** Arrêts et stations
 - Contient les informations sur les points d'arrêt, y compris les coordonnées géographiques.
- **routes:** Lignes de transport
 - Détaille les différentes lignes de transport, leurs noms et descriptions.
- **trips:** Trajets
 - Décrit les trajets spécifiques effectués sur les lignes, y compris les directions et les horaires.
- **stop_times:** Horaires de passage
 - Fournit les horaires d'arrivée et de départ pour chaque arrêt sur chaque trajet.

```
docker volume create shared-data
docker run --rm --volume shared-data:/shared-data \
    alpine ash -c "mkdir -p /shared-data/gtfs && cd /shared-data/gtfs/ && wget -O gtfs-complet.zip --quiet https://s3.eu-west-1.amazonaws.com/files.orchestra.ratpdev.com/networks/rd-toulon/exports/gtfs-complet.zip && unzip -o *.zip && rm *.zip"
```

```
shared-data
Archive: gtfs-complet.zip
inflating: agency.txt
inflating: stops.txt
inflating: routes.txt
inflating: trips.txt
inflating: stop_times.txt
inflating: calendar.txt
```

```

inflating: calendar_dates.txt
inflating: shapes.txt
inflating: feed_info.txt

```

- Conversion des données pour l'intégrer dans le système d'information géographique (SIG) PostGIS. **PostGIS** est une extension du **SGBD PostgreSQL** qui permet la manipulation d'informations géographiques (spatiales) sous forme de géométries (points, lignes, polygones), conformément aux standards établis par l'**Open Geospatial Consortium**. En d'autres termes, PostGIS permet le traitement d'objets spatiaux dans PostgreSQL, autorisant le stockage des objets graphiques en base de données pour les **systèmes d'informations géographiques (SIG)**.

```

docker run --rm --quiet --volume
shared-data:/shared-data \
    --entrypoint sh \
        ghcr.io/public-transport/
gtfs-via-postgres:4 -eu -o pipefail
-c '/app/cli.js $@ --require-
dependencies --trips-without-shape-
id --lower-case-lang-codes -- /
shared-data/gtfs/*.txt> /shared-
data/gtfs/data.sql'

```

```

WARNING: The requested image's
platform (linux/amd64) does not match
the detected host platform (linux/
arm64/v8) and no specific platform
was requested
is_valid_lang_code
is_timezone
calendar
    processed 382 rows
calendar_dates
    processed 6894 rows
service_days

```

```

agency
    processed 1 rows
routes
    processed 97 rows
trips
    processed 11787 rows
frequencies
feed_info
    processed 1 rows
shapes
    processed 195972 rows
stops
    processed 1987 rows
stop_times
    processed 387239 rows

```

```

docker network ls|grep notebooknet
> /dev/null || docker network create
notebooknet
docker container ls|grep db > /dev/
null && docker stop db

echo "--> Launch DBMS"
docker run --quiet --rm --name db -
d \
    --env
POSTGRES_USER=${DB_USERNAME:-dba} \
    --env
POSTGRES_PASSWORD=${DB_PASSWORD:-
secretsecret} \
    --env POSTGRES_DB=${DB_NAME:-
notebook-db} \
    --volume dbhost-data:/var/lib/
postgresql \
    --volume shared-data:/shared-
data \
    --network=notebooknet \
    -p 5432:5432 \
    postgis/postgis:17-master

docker container exec \
    --env PGPASSWORD=secretsecret \

```

```
db \
  sh -c 'until pg_isready -U dba
-d notebook-db; do echo "$(date) -
waiting for database to start"; sleep
1; done'
```

```
docker container exec \
  --env PGPASSWORD=secretsecret \
  db \
  psql --quiet -h db -U dba gtfs -
f /shared-data/gtfs/data.sql;
```

```
db
--> Launch DBMS
```

```
--> Create DB
CREATE DATABASE
--> Populate DB
```

WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8) and no specific platform was requested

```
docker run --quiet --rm \
  --env PGPASSWORD=secretsecret \
  --network=notebooknet \
  postgres \
    psql -h db -U dba gtfs -c
\\dt;
```

```
elaae55d9ea96f2a45d27f6046c70c99c1257698c
/var/run/postgresql:5432 - no
response
Tue 10 Dec 2024 07:00:07 AM UTC -
waiting for database to start
/var/run/postgresql:5432 - no
response
Tue 10 Dec 2024 07:00:08 AM UTC -
waiting for database to start
/var/run/postgresql:5432 - no
response
Tue 10 Dec 2024 07:00:10 AM UTC -
waiting for database to start
/var/run/postgresql:5432 - accepting
connections
```

List of relations		
Schema	Name	Type
Owner		
public	agency	table
dba		
public	calendar	table
dba		
public	calendar_dates	table
dba		
public	feed_info	table
dba		
public	frequencies	table
dba		
public	routes	table
dba		
public	shapes	table
dba		
public	spatial_ref_sys	table
dba		
public	stop_times	table
dba		
public	stops	table

```
echo "--> Create DB"
docker container exec \
  --env PGPASSWORD=secretsecret \
  db \
  psql -h db -U dba notebook-db -c
'CREATE DATABASE gtfs OWNER dba'
echo "--> Populate DB"
```

```
| dba
public | trips | table
| dba
(11 rows)
```

```
# Définition d'une requête SQL pour
rechercher les arrêts "Campus"
read -r -d '' QUERY << EOM
SELECT
    stop_id,      -- Identifiant unique
de l'arrêt
    stop_name,    -- Nom de l'arrêt
    stop_loc      -- Localisation de
l'arrêt, Géométrie PostGIS (format
natif)
FROM

    stops
WHERE stop_name ILIKE '%Campus%'
LIMIT 3;
EOM
```

```
docker run --rm \
    --env PGPASSWORD=secretsecret \
    --network=notebooknet \
    postgres \
        psql -h db -U dba gtfs -
c $QUERY;
```

```
stop_id |      stop_name      |
stop_loc
-----
+-----+-----+-----+
GACAMN | Campus La Garde/La Valette
| 0101000020E6100000F52B9D0FCF121840F7730,
GACAMS | Campus La Garde/La Valette
| 0101000020E6100000A06F0B96EA1218401F82A,
HYLAGN | Agricampus          | 01010000,
(3 rows)
```

```
# Requête SQL pour trouver les arrêts
"Université" avec leurs coordonnées
GPS
read -r -d '' QUERY << EOM
SELECT
    stop_id,      -- Identifiant unique
de l'arrêt
    stop_name,    -- Nom de l'arrêt
    ST_X(ST_Transform(stop_loc::geometry,
4326)) AS stop_latitude, --
Conversion des coordonnées en
latitude (EPSG:4326 = standard GPS)
    ST_Y(ST_Transform(stop_loc::geometry,
4326)) AS stop_longitude --
Conversion des coordonnées en
longitude (EPSG:4326 = standard GPS)
FROM

    stops
WHERE stop_id ILIKE 'GACAM%'
LIMIT 3;
EOM
```

```
docker run --rm \
    --env PGPASSWORD=secretsecret \
    --network=notebooknet \
    postgres \
        psql -h db -U dba gtfs -
c $QUERY;
```

```
stop_id |      stop_name      |
stop_latitude | stop_longitude
-----
+-----+-----+-----+
GACAMN | Campus La Garde/La Valette
| 6.018368 | 43.138304
GACAMS | Campus La Garde/La Valette
| 6.018473 | 43.138056
(2 rows)
```

```
# Requête SQL pour trouver les arrêts
"Université" avec leurs coordonnées
```

```

GPS
read -r -d '' QUERY << EOM
SELECT
    route_id,          -- Identifiant
unique de la ligne
    route_short_name,  -- Nom court
de la ligne
    route_long_name    -- Nom long
de la ligne
FROM
    routes
WHERE route_long_name ILIKE '%Tln%'
LIMIT 3;
EOM

docker run --rm \
    --env PGPASSWORD=secretsecret \
    --network=notebooknet \
    postgres \
        psql -h db -U dba gtfs -
c $QUERY;

```

```

    route_id | route_short_name |
route_long_name
-----+-----
+-----+-----
U          | U                  | Tech.Mer/
Pôle d'Act. Tln Est
0102      | 102                | Gare
Rte Tln-Aéroport Hyères
0103      | 103                | Gare
Rout. Tln-Moulin 1er
(3 rows)

```

```

# Requête pour trouver les 3 arrêts
les plus proches de l'Université de
Toulon
read -r -d '' QUERY << EOM
-- Recherche des arrêts par proximité
géographique
SELECT
    stop_id,          -- Identifiant de

```

```

l'arrêt
    stop_name, -- Nom de l'arrêt
    -- Calcul de la distance entre
chaque arrêt et l'Université
    ROUND(ST_Distance(
        stop_loc::geography, --
Position de l'arrêt
        ST_SetSRID(          --
Définit le système de coordonnées
        ST_MakePoint(        -- Crée
un point géographique
            6.016499, --
Longitude Université
            43.135440 --
Latitude Université
        ),
        4326 --
En WGS84
    )) AS distance,

    -- Conversion des coordonnées en
WGS84 (format GPS standard)
    ST_X( --
Extrait la latitude
        ST_Transform( --
Convertit les coordonnées
        stop_loc::geometry, --
Position de l'arrêt
        4326 -- Code
EPSG pour WGS84 (standard GPS)
        )
    ) AS LATITUDE,

    ST_Y(ST_Transform(stop_loc::geometry,
4326)) AS LONGITUDE -- Idem pour
longitude

FROM stops
ORDER BY distance
    ASC -- Tri
du plus proche au plus éloigné
LIMIT 5 --

```

Retourne les 5 plus proches
EOM

```
docker run --rm \
  --env PGPASSWORD=secretsecret \
  --network=notebooknet \
  postgres \
    psql -h db -U dba gtfs -
c $QUERY;
```

```
stop_id | stop_name |
distance | latitude | longitude
-----
+-----+
+-----+-----+-----+
GAUNSS | Université Sud |
251 | 6.019189 | 43.134341
GAUNSN | Université Sud |
267 | 6.019715 | 43.134955
VACAMS | Campus |
304 | 6.014507 | 43.137753
GACAMS | Campus La Garde/La Valette
| 332 | 6.018473 | 43.138056
GAPLAS | Planquette |
338 | 6.017341 | 43.132457
(5 rows)
```

```
# Store a query in the QUERY shell
variable
read -r -d '' QUERY << EOM
SELECT
  s.stop_id AS arret, --
  Identifiant de l'arrêt
  s.stop_name AS nom_arret, --
  Nom de l'arrêt
  r.route_short_name AS ligne, --
  Numéro de ligne
  r.route_long_name AS nom_ligne,
  -- Nom de la ligne
  t.trip_headsign AS destination,
  -- Destination du voyage
  st.departure_time AS depart --
```

Heure de départ
FROM

```
stop_times st
  JOIN stops s ON st.stop_id =
s.stop_id -- Liaison avec
les arrêts
  JOIN trips t ON st.trip_id =
t.trip_id -- Liaison avec
les voyages
  JOIN routes r ON t.route_id =
r.route_id -- Liaison avec
les lignes
WHERE
  st.stop_id = 'VACAMN'
  AND r.route_short_name = 'U'
  AND st.departure_time >
(NOW()::time) -- Départ après l'heure
actuelle

ORDER BY
  st.departure_time
LIMIT 3;
EOM
```

```
docker run --rm \
  --env PGPASSWORD=secretsecret \
  --network=notebooknet \
  postgres \
    psql -h db -U dba gtfs -
c $QUERY;
```

```
arret | nom_arret | ligne |
nom_ligne | destination
| depart
-----+-----+-----
+-----+
+-----+-----+-----+
VACAMN | Campus | U | Tech.Mer/
Pôle d'Act. Tln Est | Technopole de
la Mer | 07:05:00
VACAMN | Campus | U | Tech.Mer/
Pôle d'Act. Tln Est | Technopole de
```

```
la Mer | 07:10:00
VACAMN | Campus | U | Tech.Mer/
Pôle d'Act. Tln Est | Technopole de
la Mer | 07:10:00
(3 rows)
```

```
# Store a query in the QUERY shell
variable
read -r -d '' QUERY << EOM
SELECT
    r.route_short_name AS ligne, --
    Numéro de ligne
    st.arrival_time AS arrivee, --
    Heure d'arrivée
    st.departure_time AS depart --
    Heure de départ
FROM stop_times st
JOIN stops s ON st.stop_id = s.stop_id
-- Liaison avec les arrêts
JOIN trips t ON st.trip_id = t.trip_id
-- Liaison avec les voyages
JOIN routes r ON t.route_id =
r.route_id -- Liaison avec
les lignes
WHERE st.stop_id = 'GAUNSS'

LIMIT 3
EOM
```

```
docker run --rm \
    --env PGPASSWORD=secretsecret \
    --network=notebooknet \
    postgres \
        psql -h db -U dba gtfs -
c $QUERY;
```

```
ligne | arrivee | depart
-----+-----+-----
129   | 06:33:00 | 06:33:00
129   | 07:03:00 | 07:03:00
129   | 07:54:00 | 07:54:00
(3 rows)
```

Définissons tout d'abord simplement une classe Java pour représenter l'une des entités de l'exemple, un arrêt de bus. Nous utiliserons un record Java (non mutable).

```
/**
 * Représente un arrêt de transport
 */
public record Stop(
    String id, // Identifiant unique
    String name, // Nom de l'arrêt
    Double latitude, // Latitude
    Double longitude, // Longitude
    String locationType // arrêt, station
) {
    public Stop {
        if (id == null || name == null
            || latitude == null || longitude ==
            null) {
            throw new
                IllegalArgumentException("Required
                fields cannot be null");
        }
    }
}
```

```
/**
 * Représente une ligne de transport
 */
public record Route(
    String id, // Identifiant unique
    String shortName, // Numéro/Code court
    String longName // Nom complet
) {
    public Route {
        if (id == null || shortName
            == null) {

```

```

        throw new
        IllegalArgumentException("Required
        fields cannot be null");
    }
}

```

```

/**
 * Représente un voyage sur une ligne
 */
public record Trip(
    String id,          // Identifiant
                           unique
    Route route,        // ID de
                           la ligne
    String tripHeadsign, //
    Destination affichée
    String tripShortName // Nom
                           court du voyage
) {
    public Trip {
        if (id == null || route ==
        null || tripHeadsign == null) {
            throw new
            IllegalArgumentException("Required
            fields cannot be null");
        }
    }
}

```

```

import java.time.LocalTime;

/**
 * Représente un horaire de passage
 à un arrêt
 */
public record StopTime(
    Trip trip,          // ID
                           du voyage
    Stop stop,          // ID
                           de l'arrêt
    LocalTime arrivalTime, // Heure

```

```

d'arrivée
    LocalTime departureTime, // Heure
de départ
    String stopHeadsign      //
    Destination affichée à l'arrêt
) {
    public StopTime {
        // Validation des champs
obligatoires
        if (trip == null || stop ==
        null ||
            arrivalTime == null ||
            departureTime == null) {
            throw new
            IllegalArgumentException(
                "trip, stop, times
                are required"
            );
        }

        // Validation de la cohérence
des heures
        if (departureTime.isBefore(arrivalTime))
        {
            throw new
            IllegalArgumentException(
                "Departure time cannot
                be before arrival time"
            );
        }
    }
}

```

```

Stop stop=new Stop("TOCMAS",
"Champ de Mars", 5.940372,
43.121363,"stop");
Route route=new Route("U", "U",
"Université");
Trip trip=new Trip("U_1", route,
"Université", "U1");
StopTime stopTime=new StopTime(trip,
stop, LocalTime.of(8, 0),

```

```
LocalTime.of(8, 5), "Université");
```

```
stopTime;
```

```
StopTime[trip=Trip[id=U_1,  
route=Route[id=U,  
shortName=U, longName=Université],  
tripHeadsign=Université,  
tripShortName=U1],  
stop=Stop[id=TOCMAS, name=Champ  
de Mars, latitude=5.940372,  
longitude=43.121363,  
locationType=stop],  
arrivalTime=08:00,  
departureTime=08:05,  
stopHeadsign=Université]
```

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

```
import java.sql.Connection;  
import java.sql.DriverManager;
```

```
System.out.println("%s connecting on  
%s.".formatted(DB_USERNAME,DB_URL));
```

```
final String query = ""  
    SELECT stop_id,  
           stop_name,  
           ST_X(ST_Transform(stop_loc::geomet  
4326)) AS LATITUDE,  
           ST_Y(ST_Transform(stop_loc::geomet  
4326)) AS LONGITUDE,  
           location_type  
    FROM stops  
           ORDER BY  
ST_Distance(stop_loc::geometry,  
ST_SetSRID(ST_MakePoint(5.940207,  
43.121145), 4326)) ASC
```

```
LIMIT 3"";
```

```
try(Connection connection = DriverManager.getConnection(  
    Statement stmt =  
connection.createStatement();  
    ) {  
        ResultSet rs =  
stmt.executeQuery(query);  
        long id=0;  
        while (rs.next())  
            System.out.println(new  
Stop(rs.getString("stop_id"),  
rs.getString("stop_name"),  
rs.getDouble("latitude"),  
rs.getDouble("longitude"),  
rs.getString("location_type")  
));  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }
```

```
dba connecting on jdbc:postgresql://  
docker/gtfs.  
Stop[id=TOCMAS, name=Champ de Mars,  
latitude=5.940372,  
longitude=43.121363,  
locationType=stop]  
Stop[id=TOCMAN, name=Champ de Mars,  
latitude=5.940497,  
longitude=43.12208,  
locationType=stop]  
Stop[id=TOPIX0,  
name=Pie X, latitude=5.941181,  
longitude=43.120398,  
locationType=stop]
```

3 Le Driver JDBC

Le driver JDBC est un composant essentiel qui permet aux applications Java de

communiquer avec une base de données. Il sert d'intermédiaire entre l'application et le SGBDR, convertissant les appels Java en commandes compréhensibles par la base de données.

Pour utiliser un driver JDBC dans votre projet, vous devez l'ajouter au classpath de votre application. Les fournisseurs de bases de données mettent généralement à disposition les drivers nécessaires sur leurs sites web officiels. Avec Maven, l'intégration du driver est simplifiée grâce à la gestion des dépendances. Vous pouvez spécifier le driver JDBC comme une dépendance dans votre fichier pom.xml, et Maven se chargera de le télécharger et de l'ajouter au classpath lors de la construction de votre projet.

4 Ouverture d'une connexion

Pour établir une connexion avec une base de données, il est nécessaire d'initialiser la classe qui implémente le Driver JDBC. Historiquement, cela se faisait en invoquant `Class.forName("nom.de.la.classe.Driver")`. Cependant, avec les versions modernes de JDBC, cette étape est devenue facultative. à partir de **JDBC 4.0+**, la correspondance entre l'URL de connexion et le driver approprié est **gérée automatiquement**.

```
Connection connexion DriverManager.getConnection("jdbc:typeDeBaseDeDonnees://
hôte:port/nomDeLaBase",
"utilisateur", "motdepasse");
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

final
String DB_URL="jdbc:postgresql://
docker/gtfs";
```

```
// Nom d'utilisateur et mot de passe
de la base de données récupérés depuis
les variables d'environnement
final String DB_USERNAME=Optional.of(System.getenv("DB_
>new IllegalArgumentException("DB_USERNAME
is required"));
final String DB_PASSWORD=Optional.of(System.getenv("DB_
>new IllegalArgumentException("DB_PASSWORD
is required"));

try (Connection connection =
DriverManager.getConnection(DB_URL,
DB_USERNAME, DB_PASSWORD)) {
    // La connexion est établie,
    vous pouvez interagir avec la base de
    données ici
} catch (SQLException e) {
    // Gestion des exceptions liées
    à la base de données
    e.printStackTrace();
}
```

5 Exécution de requêtes SQL

Pour exécuter des requêtes SQL en Java, on utilise des instances de la classe Statement. Il existe trois types de Statement :

- Statement : Utilisé pour des requêtes SQL simples. Il permet d'exécuter des requêtes dynamiques, mais n'est pas optimisé pour des exécutions répétées.
- PreparedStatement : Il s'agit d'une version précompilée du Statement qui est optimisée pour exécuter des requêtes SQL multiples fois avec des paramètres différents. Cela améliore les performances en réduisant le temps de compilation SQL sur la base de données.
- CallableStatement : Utilisé pour exécuter des procédures stockées sur la base de données, ce qui permet d'appeler des fonctions complexes stockées directement dans la base de données.

Chaque Statement est créé à partir d'une instance de Connection.

Pour exécuter des requêtes SQL on utilise des instances de la classe `Statement`.

6 Les requêtes simples

L'approche à adopter pour exécuter des requêtes SQL varie en fonction de leur objectif. Pour une **interrogation** de données (SELECT), utilisez la méthode `executeQuery()`. Cette méthode vous permettra de parcourir les résultats à l'aide d'un objet **ResultSet**. En revanche, pour une **modification** de données (UPDATE, INSERT, DELETE), privilégiez la méthode `executeUpdate()`. Celle-ci est conçue pour les opérations de mise à jour ou les commandes de gestion de base de données et renvoie le **nombre de lignes affectées** par la requête. Enfin, si vous avez affaire à des requêtes de **type indéterminé** ou des procédures stockées pouvant produire plusieurs jeux de résultats, la méthode `execute()` est plus flexible et adaptative. Elle vous permettra de gérer les résultats de manière plus versatile.

Ce paragraphe fournit des directives claires sur la méthode à utiliser en fonction du type de requête SQL, en mettant l'accent sur la flexibilité et l'efficacité du processus.

La méthode à appeler est différente suivant la nature de la requête SQL que l'on veut exécuter :

- Consultation (SELECT)
 - `executeQuery()` : Utilisé pour les requêtes de type SELECT
 - Parcours des résultats avec un **ResultSet**
- Mise à jour (UPDATE, INSERT, DELETE)
 - `executeUpdate()` : Utilisé pour les requêtes de mise à jour ou de gestion de la base de données
 - Renvoie le **nombre de lignes modifiées**
- Type Inconnu ou Résultats Multiples

- `execute()` : Utilisé lorsque le type de requête est inconnu ou pour des procédures stockées pouvant renvoyer plusieurs résultats
- Gestion plus **flexible** des résultats

7 Parcours des résultats

La manipulation des résultats d'une requête SQL de type SELECT se fait via l'interface `ResultSet`. La méthode `executeQuery()` est utilisée pour exécuter une requête SELECT et retourne un objet `ResultSet`. Cet objet `ResultSet` contient les données retournées par la requête. Pour accéder aux valeurs des attributs de chaque ligne du résultat, `ResultSet` offre une série de méthodes `getXXX()`, où XXX représente le type Java correspondant à la valeur à récupérer. Par exemple, `getInt(int numéroDeColonne)` ou `getString(String nomDeColonne)` sont utilisées pour récupérer des valeurs entières ou des chaînes de caractères, respectivement.

```
// Create a list to store
Stop objects
List<Stop> stops = new
ArrayList<>();

final String query = ""
SELECT stop_id,
       stop_name,
       ST_X(ST_Transform(stop_loc::geometry,
4326)) AS LATITUDE,
       ST_Y(ST_Transform(stop_loc::geometry,
4326)) AS LONGITUDE,
       location_type
FROM stops
ORDER
BY    ST_Distance(stop_loc::geometry,
ST_SetSRID(ST_MakePoint(5.940207,
43.121145), 4326)) ASC
LIMIT 3"";

try(Connection connection = DriverManager.getConne
```

```

        Statement stmt
= connection.createStatement();
    ) {
        ResultSet rs
= stmt.executeQuery(query);
        long id=0;
        while
(rs.next())
            stops.add(new
Stop(rs.getString("stop_id"),
rs.getString("stop_name"),
rs.getDouble("latitude"),
rs.getDouble("longitude"),
rs.getString("location_type")
));
    } catch (SQLException
e) {
        e.printStackTrace();
    }

stops.forEach(System.out::println);

```

```

Stop[id=TOCMAS, name=Champ de Mars,
latitude=5.940372,
longitude=43.121363,
locationType=stop]
Stop[id=TOCMAN, name=Champ de Mars,
latitude=5.940497,
longitude=43.12208,
locationType=stop]
Stop[id=TOPIX0,
name=Pie X, latitude=5.941181,
longitude=43.120398,
locationType=stop]

```

```

try (Connection connection =
DriverManager.getConnection(DB_URL,
DB_USERNAME, DB_PASSWORD);
    Statement statement =

```

```

connection.createStatement()) {
    // SQL query to insert
data into the 'agency' table
    String sql = ""
INSERT
INTO agency(agency_id, agency_name,
agency_url, agency_timezone,
agency_lang, agency_phone,
agency_fare_url, agency_email)
VALUES(3, 'Neverland',
'http://nowhere.fr', 'Europe/Paris',
'fr',
'555-555-555', 'http://nowhere.fr/
fare', 'neverland@nowhere.org')"";

    // Execute the query and
get the number of changes
    int numberOfChanges
= statement.executeUpdate(sql);

    // Print the result
    System.out.println("Nombre
de changements: " + numberOfChanges);
} catch (SQLException e) {
    e.printStackTrace();
}

```

Nombre de changements: 1

8 Les exceptions

Erreur dans le code SQL (SQLException) :

Lorsque JDBC rencontre une erreur lors d'une interaction avec une source de données, il génère une instance de `SQLException`. Cette exception contient des informations telles que la description de l'erreur, le code `SQLState` (standardisé par ISO/ANSI et Open Group), et un code d'erreur spécifique.

Avertissement lors de l'exécution (SQLWarning) : Les avertissements SQL (`SQLWarning`) sont des exceptions qui ne provoquent

pas l'interruption du programme, mais signalent des problèmes potentiels. Par exemple, un avertissement peut indiquer une conversion de données tronquée (sous-classe de SQL-Warning) lors de l'exécution d'une requête. Ces avertissements sont utiles pour la détection précoce de problèmes sans arrêter le flux d'exécution.

```
//|error: true
String wrongQuery = " SELECT * FROM
Employee" ;
try (Connection connection = DriverManager
        .getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);
        Statement statement
= connection.createStatement();
        ResultSet resultSet =
statement.executeQuery(wrongQuery))
{
    } catch (SQLException e) {
        //Erreur lors de la requête

log.severe(e.getMessage());
    }
```

```
try (Connection connection = DriverManager
        .getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);
        Statement statement
= connection.createStatement();
        ResultSet resultSet =
statement.executeQuery(wrongQuery))
{
    } catch (SQLException e) {
        //Erreur lors de la requête

log.severe(e.getMessage());
    }
```

using incubating module(s):
jdk.incubator.vector

```
log.severe(e.getMessage());
```

```
cannot find symbol
```

```
symbol:   variable log
```

```
ResultSet resultSet = statement.executeQuery(wrongQuery);
statement.executeQuery(wrongQuery))
{
```

```
auto-closeable resource resultSet
is never referenced in body of
corresponding try statement
```

9 Types Java/JDBC et SQL

Types de Données Java/JDBC

Lorsque nous travaillons avec des bases de données, nous devons comprendre les différences de types entre les différents SGBD (Systèmes de Gestion de Bases de Données). Heureusement, JDBC (Java Database Connectivity) simplifie cette complexité en définissant ses propres types SQL sous forme de constantes dans la classe `java.sql.Types`.

1. Conversion entre SQL et Java :

- Lorsque nous récupérons des données depuis la base de données (par exemple, lors de l'exécution d'une requête SELECT), le pilote JDBC effectue automatiquement la conversion des types SQL vers les types Java appropriés.
- Lorsque nous passons des paramètres à une requête (par exemple, lors de l'exécution d'une requête INSERT ou UP-

DATE), le pilote effectue la conversion inverse, de Java vers SQL.

2. Utilisation explicite des méthodes `getXXX()` et `setXXX()` :

- Pour récupérer des valeurs depuis le résultat d'une requête, nous utilisons explicitement des méthodes telles que `getString()`, `getInt()`, `getDouble()`, etc.
- De même, pour définir des valeurs de paramètres, nous utilisons des méthodes comme `setString()`, `setInt()`, `setDouble()`, etc.

3. Choix multiples pour certains types :

- Pour certains types SQL, plusieurs méthodes Java peuvent être utilisées pour récupérer les valeurs :
 - **CHAR** et **VARCHAR** : Utilisez `getString()`.
 - **LONGVARCHAR** : Utilisez `getAsciiStream()` ou `getCharacterStream()`.
 - **BINARY** et **VARBINARY** : Utilisez `getBytes()`.
 - **REAL** : Utilisez `getFloat()`, et pour **DOUBLE** et **FLOAT**, utilisez `getDouble()`.
 - **DECIMAL** et **NUMERIC** : Utilisez `getBigDecimal()`.
 - Types de date et d'heure : **DATE** (utilisez `getDate()`), **TIME** (utilisez `getTime()`), et **TIMESTAMP** (utilisez `getTimestamp()`).

10 Transaction

Gestion des Transactions avec JDBC

Par défaut, une connexion à la base de données est ouverte en mode « auto-commit ». Cela signifie qu'après chaque requête SQL qui modifie les données (par exemple, une requête INSERT, UPDATE ou DELETE), un commit est automatiquement effectué. Le commit valide les modifications dans la base de données de manière permanente.

Cependant, pour un contrôle plus fin sur les transactions, nous pouvons désactiver le mode « auto-commit » et gérer manuellement les commits et les rollbacks. Voici comment faire :

1. Désactiver l'auto-commit :

- Utilisez la méthode `setAutoCommit(false)` sur l'objet `Connection` pour désactiver l'auto-commit.
- Cela signifie que les modifications ne seront pas validées automatiquement après chaque requête.

2. Valider une transaction (commit) :

- Lorsque vous avez terminé une série de requêtes et que vous souhaitez valider les modifications dans la base de données, appelez `commit()` sur l'objet `Connection`.
- Cela confirme les changements et les rend permanents.

3. Annuler une transaction (rollback) :

- Si une erreur survient ou si vous souhaitez annuler une série de modifications, appelez `rollback()` sur l'objet `Connection`.
- Cela annule toutes les modifications depuis le dernier commit.

11 Précompilation des requêtes

Les requêtes SQL construites à partir de chaînes de caractères (par exemple, en concaténant des valeurs), sont compilées à chaque appel. Cette approche peut entraîner une perte de performances, car la compilation est coûteuse en termes de temps et de ressources.

Cependant, JDBC propose une solution plus efficace : les **requêtes paramétrées** (ou `PreparedStatement`). Avec les requêtes paramétrées, la requête SQL est compilée une seule fois, lors de sa création. Elle est ensuite réutilisée pour chaque exécution. Cela évite la

compilation répétée et améliore considérablement les performances.

La requête est définie avec des espaces réservés pour les paramètres (par exemple, "SELECT * FROM users WHERE username = ?"). Au moment de l'exécution, les espaces réservés sont remplacés par les valeurs réelles.

```
// Déclaration de la requête SQL
String sql = ""
SELECT stop_id,
       stop_name,
       ST_X(ST_Transform(stop_loc::geometry
4326)) AS LATITUDE,
       ST_Y(ST_Transform(stop_loc::geometry
4326)) AS LONGITUDE,
       location_type
FROM stops
WHERE stop_id = ?
ORDER
BY    ST_Distance(stop_loc::geometry,
ST_SetSRID(ST_MakePoint(5.940207,
43.121145), 4326)) ASC
LIMIT 3"";

try (Connection connection =
DriverManager.getConnection(DB_URL,
DB_USERNAME, DB_PASSWORD);
    PreparedStatement pstmt =
connection.prepareStatement(sql)) {
    // Tableau contenant les
    // identifiants à rechercher
    String[] ids = {"TOCMAS",
"TOCMAN", "TOPIX0"};
    for (String id : ids) {
        // Remplacer le paramètre de
        // la requête par l'identifiant actuel
        pstmt.setString(1, id);
        // Exécution de la requête
        try (ResultSet rs =
pstmt.executeQuery()) {
            while (rs.next()) {
                // Création et
```

affichage d'un objet Stop avec les données récupérées

```
        System.out.println(new
Stop(rs.getString("stop_id"),
rs.getString("stop_name"),
rs.getDouble("latitude"),
rs.getDouble("longitude"),
rs.getString("location_type")));
    }
}
} catch (SQLException e) {
    // Gestion des exceptions liées
    // à la connexion ou à la requête
    e.printStackTrace();
}
```

```
Stop[id=TOCMAS, name=Champ de Mars,
latitude=5.940372,
longitude=43.121363,
locationType=stop]
Stop[id=TOCMAN, name=Champ de Mars,
latitude=5.940497,
longitude=43.12208,
locationType=stop]
Stop[id=TOPIX0,
name=Pie X, latitude=5.941181,
longitude=43.120398,
locationType=stop]
```

12 Les transactions

```
try (Connection connection =
DriverManager.getConnection(DB_URL,
DB_USERNAME, DB_PASSWORD)) {

    // Création de la table account
    String createAccountTableSql =
"CREATE TABLE account (" +
    "id SERIAL PRIMARY KEY," +
```

```

        "name VARCHAR(100) NOT NULL," +
        "balance DEC(15,2) NOT NULL)";
        try (Statement statement =
connection.createStatement()) {
            statement.executeUpdate(createAccountSql);
        }

        // Création des comptes pour Alice
et Bob avec une requête préparée
        String createAccountSql =
"INSERT INTO account(name, balance)
VALUES(?, ?);";
        try (PreparedStatement pstmt = connection.prepareStatement(createAccountSql)) {
            pstmt.setString(1, "Bob");
            pstmt.setInt(2, 1000);
            pstmt.executeUpdate();

            pstmt.setString(1, "Alice");
            pstmt.setInt(2, 1000);
            pstmt.executeUpdate();
        }

        // Préparation de la requête pour
mettre à jour le solde des comptes
        String updateAccountSql = "UPDATE
account SET balance = balance + ?
WHERE id = ?";
        try (PreparedStatement pstmtIncreaseAccount = connection.prepareStatement(updateAccountSql)) {
            // Sauvegarde de l'état
auto-commit
            boolean autoCommit =
connection.getAutoCommit();
            try {
                connection.setAutoCommit(false); //
Désactivation de l'auto-commit pour
la transaction

                // Retirer 500€ du compte
de Bob
                pstmtIncreaseAccount.setInt(1,
-500);

                pstmtIncreaseAccount.setInt(2,
1); // ID de Bob
                pstmtIncreaseAccount.executeUpdate();

                // Ajouter 500€ au
compte d'Alice
                pstmtIncreaseAccount.setInt(1, 500);
                pstmtIncreaseAccount.setInt(2,
2); // ID d'Alice
                pstmtIncreaseAccount.executeUpdate();

                connection.commit(); //
Valider la transaction
            } catch (SQLException exc) {
                connection.rollback(); //
Annuler la transaction en cas de
problème
                throw exc; // Relancer
l'exception pour gestion ultérieure
            } finally {
                connection.setAutoCommit(autoCommit); //
Rétablir l'état auto-commit
            }
        }

        // Suppression de la table account
        try (Statement statement =
connection.createStatement()) {
            statement.executeUpdate("DROP
TABLE account");
        }
    } catch (SQLException e) {
        e.printStackTrace(); // Gestion
des exceptions SQL
    }
}

```

13 Procédures stockées

Les procédures stockées sont des sous-programmes stockés dans la base de données qui peuvent être exécutés par un appel direct. Elles permettent d'encapsuler des opérations complexes et de les réutiliser facilement.

- **Avantages des procédures stockées :**
 - **Réutilisabilité** : Les procédures stockées peuvent être appelées à partir de différentes applications.
 - **Performance** : Les procédures stockées sont précompilées et stockées dans la base de données, ce qui améliore les performances.
- **Inconvénients potentiels :**
 - **Portabilité** : Les procédures stockées peuvent être spécifiques à un SGBD, ce qui peut limiter la portabilité du code.
 - **Complexité** : Les procédures stockées peuvent être difficiles à déboguer et à maintenir.

Avec JDBC `CallableStatement` permet d'appeler une procédure stockée directement sur le SGBD.

```
-- Création d'une procédure stockée
pour transférer de l'argent entre
comptes
CREATE OR REPLACE PROCEDURE
transfer_funds(
    from_account INT,
    to_account INT,
    amount DECIMAL
) AS $$
BEGIN
    UPDATE account SET balance
= balance - amount WHERE id =
from_account;
    UPDATE account SET balance
= balance + amount WHERE id =
to_account;
```

```
END;
$$ LANGUAGE plpgsql;
```

```
// Connexion à la base de données
try (Connection connection =
DriverManager.getConnection(DB_URL,
DB_USERNAME, DB_PASSWORD)) {

    // Préparation de l'appel à la
procédure stockée
    String callProcedureSql = "{CALL
transfer_funds(?, ?, ?)}";
    try (CallableStatement
callableStatement = connection.prepareCall(callProcedureSql)) {

        // Définition des paramètres
de la procédure
        callableStatement.setInt(1,
1); // ID du compte source
        callableStatement.setInt(2,
2); // ID du compte destination

        callableStatement.setBigDecimal(3,
new BigDecimal("500.00")); // Montant
à transférer

        // Exécution de la procédure
stockée
        callableStatement.execute();
    }
} catch (SQLException e) {
    e.printStackTrace(); // Gestion
des exceptions SQL
}
```