Génie Logiciel Avancé : JDBC (1/2)

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

15 janvier 2025

amU Faculté des sciences Aix Marseille Université

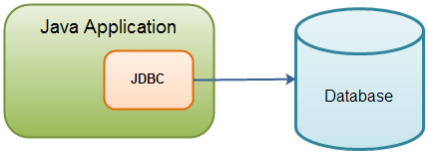
Section 1

Introduction à JDBC

JDBC : qu'est-ce que c'est ?

Java DataBase Connectivity :
API (Application Programming Interface) qui permet l'accès à des données de type table de bases de données SGBD relationnelles :

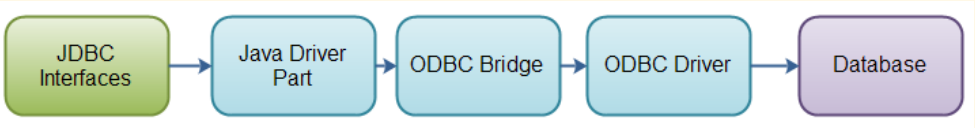
- ouvrir une connexion avec le SGBD ;
- envoyer des requêtes SQL au SGBD ;
- récupérer des données retournées par des requêtes SQL ;
- et traiter ces données "table" ;
- gérer les erreurs retournées par des requêtes au SGBD.



Les pilotes JDBC (1/2)

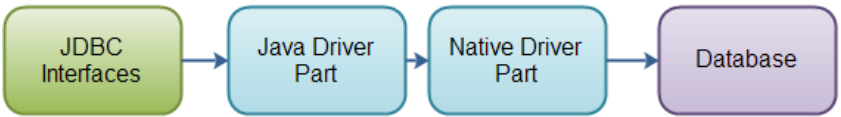
Type 1 (pont)

Utilise ODBC (*Open Database Connectivity*) pour communiquer avec une base de données.



Type 2 (API native)

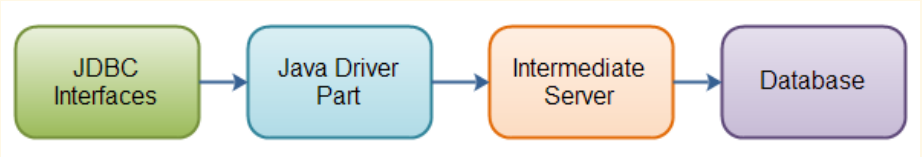
API native appelé via JNI.



Les pilotes JDBC (2/2)

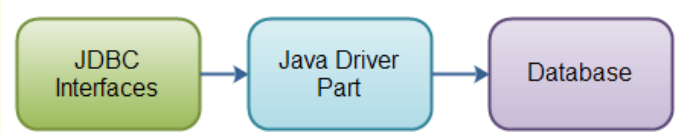
Type 3 (3 couches)

Serveur convertissant des requêtes du client Java dans le protocole SGBD.

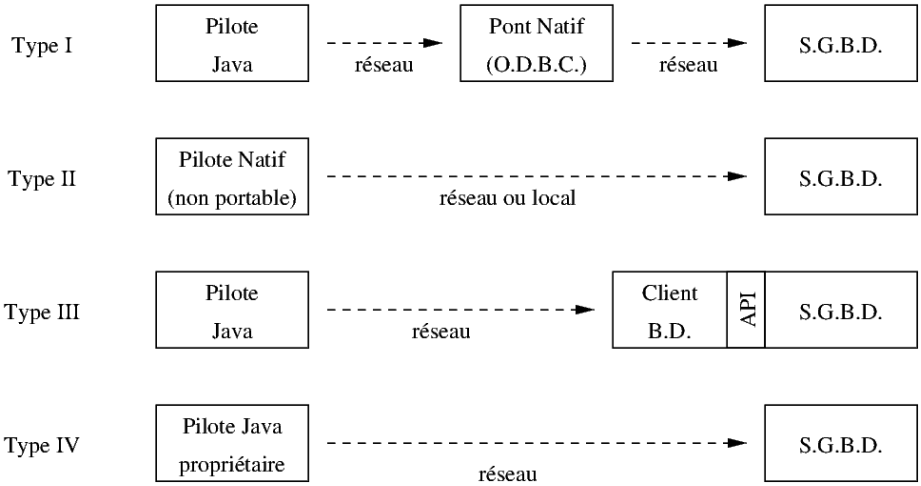


Type 4 (réseau)

Client Java envoie les requêtes via un protocole réseau implémenté par le SGBD.



Les pilotes JDBC



Section 2

Utilisation basique de JDBC

Squelette exemple

```
import java.sql.DriverManager;  // gestion des pilotes
import java.sql.Connection;     // une connexion à la BD
import java.sql.Statement;      // une instruction
import java.sql.ResultSet;      // un résultat (lignes/colonnes)
import java.sql.SQLException;   // une erreur

public class JdbcSample {
    // chargement du pilote
    // ouverture de connexion
    // exécution d'une requête
    // programme principal
}
```

Le *package* `java.sql` regroupe les interfaces et les classes de l'API JDBC.

Chargement d'un pilote JDBC

Méthode de chargement explicite d'un pilote :

```
private String driverName = "com.mysql.jdbc.Driver";

void loadDriver() throws ClassNotFoundException {
    Class.forName(driverName);
}
```

- L'appel à `forName` déclenche un chargement dynamique du pilote.
- Un programme peut utiliser plusieurs pilotes, un pour chaque base de données.
- Le pilote doit être accessible à partir de la variable d'environnement `CLASSPATH`.
- Le chargement explicite est inutile à partir de JDBC 4.

Connexion à la base de données

Méthode d'ouverture d'une nouvelle connexion :

```
private String url      = "jdbc:mysql://localhost/dbessai";
private String user     = "bduser";
private String password = "SECRET";
Connection newConnection() throws SQLException {
  Connection conn = DriverManager.getConnection(url, user, password);
  return conn;
}
```

L'URL est de la forme `jdbc:sous-protocole:sous-nom`

Exemples :

```
jdbc:oracle://srv.dil.univ-mrs.fr:1234/dbtest
jdbc:odbc:msql;USER=fred;PWD=secret
```

Exemple de requête en JDBC

```
final String PERSONNES =
    "SELECT lastName,firstName,age FROM person ORDER BY age";
Connection conn = null;
try { // create new connection and statement
  Connection con = DriverManager.getConnection(/* ... */);
  Statement st = con.createStatement();
  ResultSet rs = st.executeQuery(PERSONNES);
  while (rs.next()) {
    System.out.printf("%-20s | %-20s | %3d\n", //
      rs.getString(1), rs.getString("firstName"), rs.getInt(3));
  }
} finally { // close result, statement and connection
    if (conn != null) conn.close();
}
```

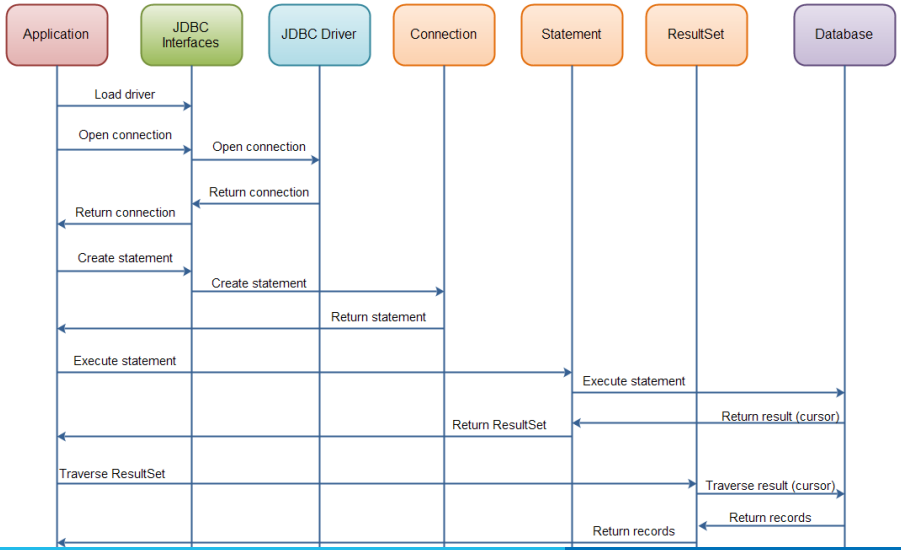
Explication de l'exemple

Étapes à suivre

1. On crée une `Connection` à partir du `DriverManager`
2. On crée un `Statement` à partir de la `Connection`
3. On exécute le `Statement` et on récupère un `ResultSet`

- `Connection` : interface pour gérer une connexion à une base de données
- `Statement` : interface pour gérer une commande SQL
- `ResultSet` : interface pour gérer les données produites par une commande

Interaction entre composants



Quelques conseils

- Évitez d'utiliser `SELECT * FROM ...` (coûteux en transfert),
- Donnez des noms locaux à vos colonnes :

  ```
  SELECT surname AS lastName, name AS firstName ...
  ```

- Faites le maximum de travail en SQL et le minimum en Java.
- Minimisez le nombre de connexions ouvertes.
- Une connexion peut être utilisée par plusieurs instructions et une instruction permet d'exécuter plusieurs requêtes.
- Vous pouvez fermer (close) un résultat de requête (`ResultSet`).
- Vous pouvez fermer (close) une instruction (`Statement`) ce qui provoque la fermeture des résultats liés à cette instruction.

Interface `java.sql.ResultSet`

Un `ResultSet` maintient un curseur sur une ligne (*row*) de la table issue de la requête.

On peut déplacer le curseur sur la ligne suivante avec un appel à `boolean next()` (renvoie `false` s'il n'y a pas de ligne suivante).

Accès aux valeurs de la *row* courante :

- `TYPE getTYPE(int columnIndex)` (l'indexation commence à 1)
- `TYPE getTYPE(String columnLabel)`

Le TYPE peut être :

`Byte`; `Boolean`; `AsciiStream`; `Short`; `String`; `UnicodeStream`; `Int`; `Bytes`; `BinaryStream`; `Long`; `Date`; `Object`; `Float`; `Time`; `BigDecimal`; `TimeStamp`;

Correspondance des types Java / SQL

SQL	Java
CHAR VARCHAR LONGVARCHAR	String
NUMERIC DECIMAL	BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT DOUBLE	double
BINARY VARBINARY LONGVARBINARY	byte[]

Correspondance des dates et heures

SQL	Java	Explication
`DATE`	`java.sql.Date`	codage de la date
`TIME`	`java.sql.Time`	codage de l'heure
`TIMESTAMP`	`java.sql.TimeStamp`	codage de la date et de l'heure

On peut convertir :

- une `Date` en une `LocalDate` avec `LocalDate toLocalDate()`
- une `LocalDate` en une `Date` avec `static valueOf(LocalDate date)`

Idem avec :

- `TimeStamp` et `LocalDateTime`
- `Time` et `LocalTime`

Requête SQL avec paramètres

Il arrive souvent qu'on veuille faire des requêtes SQL similaires, mais donc certaines parties (valeurs des arguments) sont différentes.

```
Statement st = conn.createStatement();
int nb = st.executeUpdate("UPDATE person SET Age = " + age +
                          " WHERE Name = '" + name + "'");
```

Problème : injection SQL

Si la variable `name` contient
`X' OR (1=1) OR 'Y'='Y`
La condition devient : `WHERE Nom = 'X' OR (1=1) OR 'Y'='Y'`
et toutes les lignes sont modifiées.

Solution PreparedStatement (SQL Préformaté)

Code SQL avec partie variable :

```
PreparedStatement st = conn
  .prepareStatement("UPDATE person SET Age = ? WHERE Name = ? ");
for( ... ) {
    st.setInt(1, age[i]);
    st.setString(2, name[i]);
    st.execute();
}
```

Avantages

Pré-compilation unique (gain de performances) et paramètres plus faciles à passer.

Type de requêtes et appel en Java JDBC

Suivant le type de requête, la manière de demander l'exécution est différente.

Interfaces

- `Statement` pour les requêtes constantes
- `PreparedStatement` pour les requêtes paramétrées
- `CallableStatement` pour appeler une procédure SQL

Méthodes à appeler

- `boolean execute(String sql)` pour créer des fonctions/procédures
- `ResultSet executeQuery()` pour les requêtes DQL (Langage de requête de données) : `SELECT`
- `int executeUpdate()` pour les requêtes :
  - DML (Langage de manipulation des données) : `INSERT`, `UPDATE` et `DELETE`
  - DLL (Langage de définition des données) : `CREATE TABLE`, `ALTER TABLE`, et `DROP TABLE`

Définir une procédure sur le SGBD

Exemple de procédure SQL (possible avec HyperSQL)

```
CREATE PROCEDURE new_customer(firstname VARCHAR(50),
            lastname VARCHAR(50), address VARCHAR(100))
   MODIFIES SQL DATA
     BEGIN ATOMIC
     INSERT INTO customers VALUES (DEFAULT, firstname, lastname,
                                   CURRENT_TIMESTAMP);
     INSERT INTO addresses VALUES (DEFAULT, IDENTITY(), address);
   END
```

Créable via JDBC via un `Statement` puis `execute(sql)` avec `sql` la chaîne de caractères ci-dessus.

Appel de procédure stockée en base

```
private void createCustomer(String firtsName, String lastName,
                            String address) throws SQLException{
  CallableStatement st =
        conn.prepareCall("{call new_customer[(?,?,?)]}");
  st.setInt(1, valeur); // fixer la valeur du paramètre
  st.setString(1, firtsName);
  st.setString(2, lastName);
  st.setString(3, address);
  st.execute();
}
```

Problème

Les façons de créer des `FUNCTION` et `PROCEDURE` peuvent varier en fonction du type de la base.

Section 3

Gestion des Exceptions

Levée de l'exception `SQLException`

Les opérations suivantes :

- ouverture d'une connexion
- création d'un `statement`
- exécution d'un `statement`
- fermeture d'un `statement`
- fermeture d'une connexion
- ...

peuvent lever une `SQLException`.

$\Rightarrow$ Il est obligatoire de gérer l'exception.

SQLException et SQLWarnings

SQLException

- Exception qui n'étend pas `RuntimeException`
- donne des informations sur l'erreur :
  - `SQLState` : code d'erreur sur 5 caractères dépendant du type de la base ()
  - `vendorCode` : code erreur entier dépendant de la base
- permet de chaîner des `Exception` (implémente `Iterable<Exception>`)

SQLWarning extends SQLException

`getNextWarning()` : `Warning` suivant
`SQLWarning getWarnings()` de `Connection` : récupére le `SQWarning`

Rappel : try/catch/finally

```
try {
  // code levant des exceptions
}
catch(ExceptionType1 | ExceptionType2 e){
// gestion des exceptions de type ExceptionType1 ou ExceptionType2
}
catch(ExceptionType3 | ExceptionType4 e){
// gestion des exceptions de type ExceptionType3 ou ExceptionType4
}
finally{
  // code toujours exécuté (fermeture)
}
```

Règle générale gestion des Exceptions

Lorsqu'on fait un appel à une méthode `canThrow` pouvant lever une exception `MyException` qui n'étend pas `RuntimeException`, il est vérifié à la compilation que l'une des deux propriétés suivantes est vraie :

- la méthode appelant `canThrow` dans son code est indiquée comme pouvant lever une exception de type `MyException` ou une de ces super-classes (en écrivant `throws MyException` à la signature de la méthode).
- l'exception potentielle est capturée par un bloc `try/catch`.

$\Rightarrow$ On n'a pas obligation de gérer les `RuntimeException` mais on peut le faire si on le souhaite.