DÉVELOPPEMENT WEB

JAVA ENTERPRISE EDITION – JEE

Mohammed Achkari Begdouri

Université Abdelmalek Essaadi

Faculté Polydisciplinaire à Larache - Département Informatique

achkari.prof@gmail.com

Année universitaire 2020/2021

Chapitre 2: Java et les bases de données JDBC

DÉVELOPPEMENT WEB – JEE

SMI – S6

Introduction

- JDBC pour Java DataBase Connectivity
- API Java adaptée à la connexion avec les bases de données Relationnelles et Objet-Relationnelles
- L 'API Fournit un ensemble de classes et d 'interfaces permettant l'utilisation d'un ou plusieurs SGBD à partir d'un programme

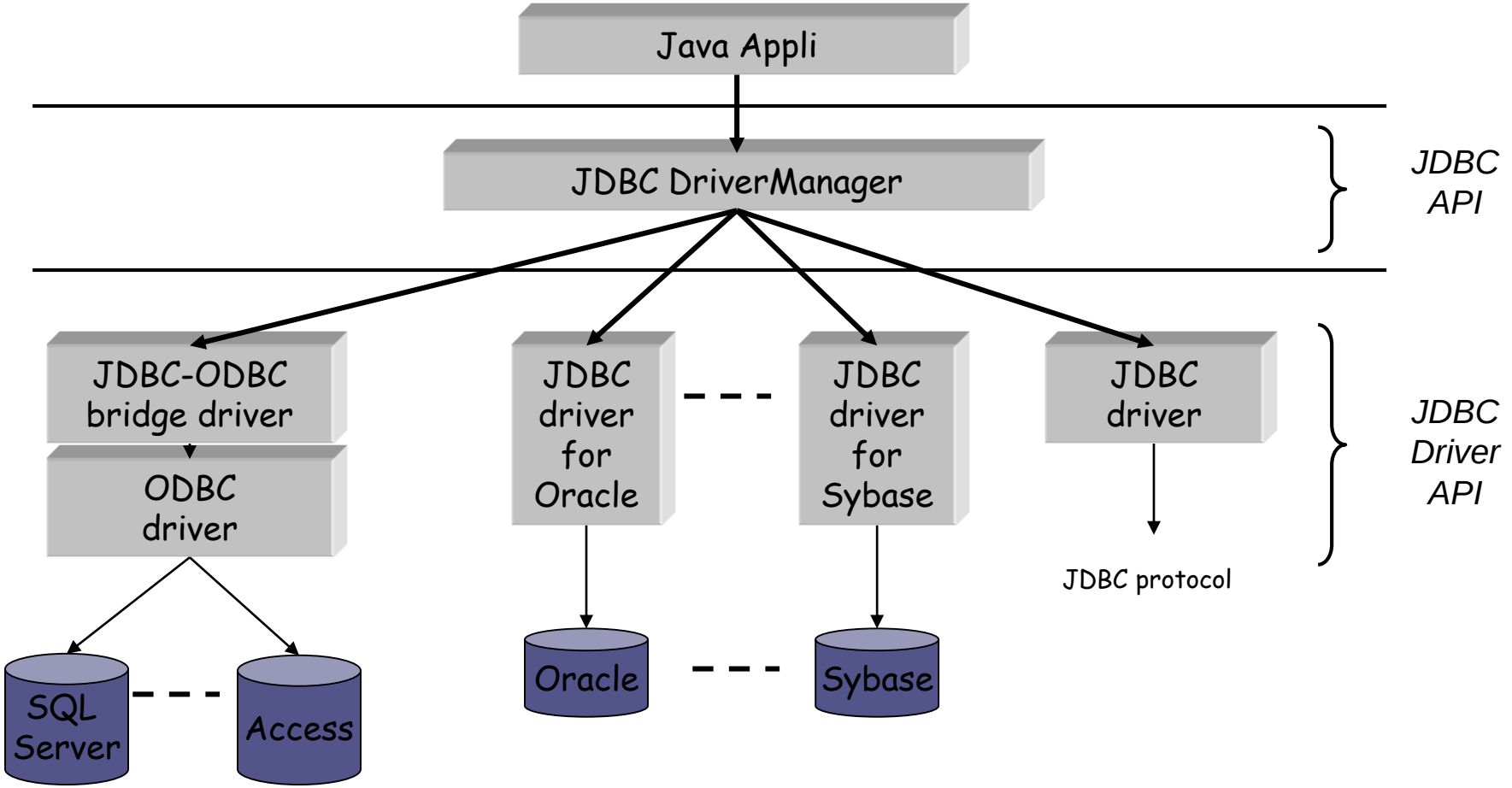
L 'API JDBC

- Est fournie par le package **`java.sql`**
  - permet de formuler et gérer les requêtes aux bases de données relationnelles et objet-relationnelle
  - supporte le standard « SQL-2 et SQL-3 »
  - 8 interfaces définissant les objets nécessaires :
    - à la connexion à une base éloignée
    - et à la création et exécution de requêtes SQL

java.sql

- 8 interfaces :
  - Statement
  - CallableStatement, PreparedStatement
  - DatabaseMetaData, ResultSetMetaData
  - ResultSet,
  - Connection
  - Driver

Architecture JDBC



Modèles de connexion en Java

- Modèle 2-tiers : 2 entités interviennent
  - 1. une application Java
  - 2. le SGBD

- Modèle 3-tiers : 3 entités interviennent
  - 1. une couche applicative
  - 2. une couche *middleware* qui assure l'interaction
  - 3. le SGBD

Mise en œuvre

0. Importer le package **`java.sql`**
1. Enregistrer le driver JDBC
2. Etablir la connexion à la base de données
3. Créer une zone de description de requête
4. Exécuter la requête
5. Traiter les données retournées
6. Fermer les différents espaces

Enregistrer le driver JDBC

- Méthode **`forName()`** de la classe **`Class`** :

```
Class.forName("com.mysql.jdbc.Driver");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

URL de connexion

- Accès à la base via un URL de la forme :

  jdbc:<sous-protocole>://nom-srv:port/nom-bd

  - qui spécifie :
    - l 'utilisation de JDBC
    - le driver ou le type de SGBD
    - l 'identification de la base locale ou distante
      - avec des paramètres de configuration éventuels
        - nom utilisateur, mot de passe, ...
  - Exemples :

```
String url = "jdbc:mysql://localhost:3306/dbeleve";
```

Connexion à la base

- Méthode **`getConnexion()`** de **`DriverManager`**
  - 3 arguments :
    - l 'URL de la base de données
    - le nom de l 'utilisateur de la base
    - son mot de passe

```
Connection connect =
   DriverManager.getConnection(url,user,password);
```

    - le **`DriverManager`** essaye tous les drivers qui se sont enregistrés (chargement en mémoire avec **`Class.forName`**()) jusqu'à ce qu'il trouve un *driver* qui peut se connecter à la base

Création d 'un Statement (1/2)

- L 'objet **Statement** possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend

- 3 types de **Statement :**
  - **Statement** : requêtes statiques simples
  - **PreparedStatement** : requêtes dynamiques pré-compilées (avec paramètres d 'entrée/sortie)
  - **CallableStatement** : procédures stockées

Création d 'un Statement (2/2)

- A partir de l 'objet **Connexion**, on récupère le **Statement** associé :

```
Statement req1 = connexion.createStatement();

PreparedStatement req2 = connexion.prepareStatement(str);

CallableStatement req3 = connexion.prepareCall(str);
```

Exécution d 'une requête (1/3)

- 3 types d 'exécution :
  - executeQuery() : pour les requêtes (SELECT) qui retournent un ResultSet (tuples résultants)
  - executeUpdate() : pour les requêtes (INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE) qui retournent un entier (nombre de tuples traités)
  - execute() : pour requêtes inconnus. Renvoie true si la requête a donné lieu à la creation d 'un objet ResultSet

Exécution d ’une requête (2/3)

- executeQuery() et executeUpdate() de la classe Statement prennent comme argument une chaîne (String) indiquant la requête SQL à exécuter :

```
Statement st = connexion.createStatement();

ResultSet rs = st.executeQuery( "SELECT nom, prenom FROM clients " +
                                "WHERE nom='itey ' ORDER BY nom");

int nb = st.executeUpdate("INSERT INTO dept(DEPT) " +  "VALUES(06)");
```

Exécution d 'une requête (3/3)

- 2 remarques :
  - le code SQL n 'est pas interprété par Java.
    - c 'est le pilote associé à la connexion (et au final par le moteur de la base de données) qui interprète la requête SQL
    - si une requête ne peut s 'exécuter ou qu'une erreur de syntaxe SQL a été détectée, l 'exception **`SQLException`** est levée
  - le driver JDBC effectue d 'abord un accès à la base pour découvrir les types des colonnes impliquées dans la requête puis un 2ème pour l 'exécuter..

Traiter les données retournées

- L 'objet `ResulSet` (retourné par l 'exécution de `executeQuery()`) permet d 'accéder aux champs des tuples sélectionnés
  - seules les données demandées sont transférées en mémoire par le driver JDBC
    - il faut donc les lire "manuellement" et les stocker dans des variables pour un usage ultérieur

Le résultat : `ResultSet`

- Il se parcourt itérativement ligne par ligne
  - par la méthode `next()`
    - retourne **`false`** si dernier tuple lu, **`true`** sinon
    - chaque appel fait avancer le curseur sur le tuple suivant
    - initialement, le curseur est positionné avant le premier tuple
      - exécuter next() au moins une fois pour avoir le premier

```
while(rs.next()) {// Traitement de chaque
  tuple}
```

Le résultat : `ResultSet`

- Les colonnes sont référencées par leur numéro ou par leur nom
- L 'accès aux valeurs des colonnes se fait par les méthodes de la forme **`getTYPE()`**

```
int val = rs.getInt(3) ; // accès à la 3e colonne
String prod = rs.getString("PRODUIT") ;
```

Le résultat : ResultSet

```
Statement st = connection.createStatement();
ResultSet rs = st.executeQuery(
   "SELECT a, b, c, FROM Table1 »
);

while(rs.next()) {
   int i = rs.getInt("a");
   String s = rs.getString("b");
   byte[] b = rs.getBytes("c");
}
```

Types de données JDBC

- Le *driver* JDBC traduit le type JDBC retourné par le SGBD en un type Java correspondant
  - le `TYPE` de `getTYPE()` est le nom du type Java correspondant au type JDBC attendu
  - chaque driver a des correspondances entre les types SQL du SGBD et les types JDBC
  - le programmeur est responsable du choix de ces méthodes
    - **`SQLException`** générée si mauvais choix

Correspondance des types

Type SQL	**Type Java**
CHAR, VARCHAR , LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BINARY, VARBINARY, LONGVARBINARY	byte[]
BIT	boolean
INTEGER	int
BIGINT	long
REAL	float
DOUBLE, FLOAT	double
DATE	java.sql.Date
TIME	java.sql.Time
….	…..

Cas des valeurs nulles

- les méthodes `getTYPE()` de `ResultSet` convertissent une valeur NULL SQL en une valeur acceptable par le type d 'objet demandé :
  - les méthodes retournant un objet (**`getString()`** , **`getObject()`** et **`getDate()`**) retournent un "**`null`** " Java
  - les méthodes numériques (**`getByte()`** , **`getInt()`** , etc) retournent "0"
  - **`getBoolean()`** retourne " **`false`** "

Fermer les différents espaces

- Pour terminer proprement un traitement, il faut fermer les différents espaces ouverts
  - sinon le garbage collector s 'en occupera mais moins efficace

- Chaque objet possède une méthode close() :

```
resultset.close();
statement.close();
connection.close();
```

Accès aux méta-données

- La méthode **`getMetaData()`** permet d 'obtenir des informations sur les types de données du **`ResultSet`**
  - elle renvoie des **`ResultSetMetaData`**
  - on peut connaître entre autres :
    - le nombre de colonne : **`getColumnCount()`**
    - le nom d 'une colonne : **`getColumnName(int col)`**
    - le type d 'une colonne : **`getColumnType(int col)`**
    - le nom de la table : **`getTableName(int col)`**

ResultSetMetaData

```
Resultset rs = stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetatData();

int nbColonnes = rsmd.getColumnCount();
for(int i = 1; i <= nbColonnes; i++) {
   // colonnes numerotees a partir de 1 (et non 0)
   String typeCol = rsmd.getColumnTypeName(i);
   String nomCol = rsmd.getColumnName(i);
}
```

DataBaseMetaData

- Pour récupérer des informations sur la base de données elle-même, utiliser la méthode `getMetaData()` de l 'objet `Connection`
  - dépend du SGBD avec lequel on travaille
  - elle renvoie des `DataBaseMetaData`
  - on peut connaître entre autres :
    - les tables de la base : **`getTables()`**
    - le nom de l 'utilisateur : **`getUserName()`**
    - . . .

Requêtes pré-compilées

- L 'objet **`PreparedStatement`** envoie une requête sans paramètres à la base de données pour pré-compilation et spécifiera le moment voulu la valeur des paramètres

  - plus rapide qu'un **`Statement`** classique
    - le SGBD analyse qu'une seule fois la requête (recherche d 'une stratégie d 'exécution adéquate)
    - pour de nombreuses exécutions d 'une même requête SQL avec des paramètres variables

Création d 'une requête pré-compilée

- La méthode **`prepareStatement()`** de l 'objet **`Connection`** crée un **`PreparedStatement`** :

```
PreparedStatement ps = c.prepareStatement("SELECT * FROM ? "
                                          + "WHERE id = ? ");
```

  - les arguments dynamiques sont spécifiés par un "?"
  - ils sont ensuite positionnés par les méthodes **`setInt()`** , **`setString()`** , **`setDate()`** , ... de **`PreparedStatement`**
  - ces méthodes nécessitent 2 arguments :
    - le premier (int) indique le numéro relatif de l 'argument dans la requête
    - le second indique la valeur à positionner

Exécution d'une requête pré-compilée

```
PreparedStatement ps = c.prepareStatement(
   "UPDATE emp SET sal = ? WHERE name = ?");
int count;
for(int i = 0; i < 10; i++) {
   ps.setFloat(1, salary[i]);
   ps.setString(2, name[i]);
   count = ps.executeUpdate();
}
```

Validation de transaction : Commit

- Utiliser pour valider tout un groupe de transactions à la fois
- Par défaut : mode auto-commit
  - un "commit " est effectué automatiquement après chaque ordre SQL
- Pour repasser en mode manuel :

```
connection.setAutoCommit(false);
```

- L 'application doit alors envoyer à la base un "commit" pour rendre permanent tous les changements occasionnés par la transaction :

```
connection.commit();
```

Annulation de transaction : Rollback

- De même, pour annuler une transaction (ensemble de requêtes SQL), l 'application peut envoyer à la base un "rollback" par :

```
connection.rollback();
```

  - restauration de l'état de la base après le dernier "commit"

Exceptions

- **`SQLException`** est levée dès qu 'une connexion ou un ordre SQL ne se passe pas correctement
  - la méthode **`getMessage()`** donne le message en clair de l 'erreur
  - renvoie aussi des informations spécifiques au gestionnaire de la base comme :
    - code d 'erreur fabricant