

Chapitre 4 : Accès aux données avec java

Objectif général

- Connaître la notion de communication avec les bases de données en utilisant un programme java

objectifs spécifiques

- Connaître la différence entre les API ODBC et JDBC
- Comprendre la connexion à une source de données
- Comprendre le traitement des requêtes

Éléments de contenu

- Les APIs ODBC et JDBC
- Connexion à une source de données
- Traitement des requêtes

I. Introduction

L'accès aux données implique la manipulation d'une source de stockage de données à savoir une base de données ou un fichier. Dans ce qui suit nous nous intéresserons aux bases de données relationnelles telles que MySQL ou SQL Server. La manipulation des données implique généralement l'exécution de requêtes ou de procédures stockées à partir de l'application côté client.

II. Les API ODBC et JDBC

ODBC: Open Database Connectivity

ODBC (Open DataBase Connectivity) est une interface de programmation (API) Microsoft permettant de faire connecter une application qui joue le rôle de client, à un SGBD qui joue le rôle de serveur de données. L'avantage de ODBC est que cette bibliothèque est indépendante du type du client et du SGBD, c'est à dire nous pouvons connecter à travers ODBC, un client développé avec Delphi avec une base de données de type SQL server. L'essentiel est que le SGBD dispose d'un pilote, qui doit être installé sur la machine. ODBC fait partie du système d'exploitation et sa configuration se fait au niveau du panneau de configuration de MS Windows.

JDBC: Java DataBase Connectivity

JDBC (Java DataBase Connectivity) est une interface de programmation (API) pour l'accès aux bases de données semblable à ODBC mais conçue pour le langage Java. Pour pouvoir accéder à une base à travers le JDBC, il est nécessaire de disposer d'un pilote compatible avec le système de base de données utilisée. Le rôle du pilote est de traduire les requêtes de l'application cliente, faite à travers l'API standardisée du JDBC, au format spécifique de la base serveur (et vice-versa).

Application/ Applet java
JDBC API
JDBC Driver
Base de données

Représentation en couches de l'accès aux données en Java avec JDBC.

III. Connexion à une source de données

✚ Les classes de l'API JDBC sont :

Il y a 4 classes importantes : DriverManager, Connection, Statement (et PreparedStatement), et ResultSet, chacune correspondant à une étape de l'accès aux données :

Classe	Rôle
DriverManager	Charger et configurer le driver de la base de données.
Connection	Réaliser la connexion et l'authentification à la base de données.
Statement (et PreparedStatement)	Contenir la requête SQL et la transmettre à la base de données.
ResultSet	Parcourir les informations retournées par la base de données dans le cas d'une sélection de données

✚ La Connexion à une source de données passe par les 2 étapes suivantes :

- Définition de l'URL de connexion.
- Etablissement de la connexion.

Dans ce qui suit nous détaillerons les étapes ci-dessus mentionnées :

• Définition de l'URL de connexion

Afin de localiser le serveur ou la base de données, il est indispensable de spécifier une adresse sous forme d'URL de type «jdbc:

Exemple:

jdbc:mysql://MonServeur:3306/Gestion: permet de se connecter à la base Gestion du serveur MySQL nommé MonServeur, tournant sur le port 3306.

• Connexion à la base de données

Pour l'établissement de la connexion, la classe **DriverManager** du package `java.sql` est utilisée. Celle-ci dispose d'une méthode statique permettant d'obtenir une connexion à une URL, la méthode **getConnection()** qui retourne un objet de type Connexion (**Connection**).

`getConnection` admet 3 paramètres:

1. **L'URL** définie précédemment.
2. **UserID**: Nom d'utilisateur.
3. **Password**: mot de passe.

Exemple: Connexion de l'utilisateur Foulen ayant le mot de passe DSI123/ au serveur MySQL `localhost`, sur la base `Gestion`.

URL = "jdbc:mysql://localhost:3306/Gestion" ou bien

URL = "jdbc:mysql:///Gestion"

Username = "Foulen"

Password = "DSI123"

```
Connection dbConnection;

try{

    dbConnection=DriverManager.getConnection(Url,Username>Password);

}catch( SQLException x ){

System.err.println("Connexion impossible à la B.D»+ x.getMessage());

}
```

IV. Traitement des requêtes

Une fois la connexion établie, il est possible d'exécuter des ordres SQL. Les objets qui peuvent être utilisés pour obtenir des informations sur la base de données sont :

Interfaces	Rôle
DatabaseMetaData	informations à propos de la base de données : nom des tables, index, version ...
ResultSet	résultat d'une requête et information sur une table. L'accès se fait enregistrement par enregistrement.
ResultSetMetaData	informations sur les colonnes (nom et type) d'un ResultSet

4.1 L'interface Statement

Afin d'accéder ou de modifier les informations contenues dans votre base de données, il convient d'utiliser un objet de type **Statement**.

<p><<interface > ></p> <p>Statement</p>
<p>+ ResultSet executeQuery(String sql) throws SQLException</p> <p>+ int executeUpdate(String sql) throws SQLException</p> <p>+ void close() throws SQLException</p>

Une instance de cet objet est retournée par la méthode `Connexion.createStatement()` comme ceci :

```
Statement statement = dbConnexion.createStatement();
```

4.2 Exécution d'une requête de sélection

Pour exécuter une requête SQL, on utilise la méthode **executeQuery** de l'objet connection. Suite à l'exécution de la requête le résultat est retourné dans un objet de type **ResultSet**.

Remarque: Cette méthode ne peut pas être appelée sur un `PreparedStatement` ou `CallableStatement`.

Exemple :

```
Connection cn=LaConnexion.seConnecter();

String requete="select * from `client`";
```

```
String codeCl; String nomCl,adrCl,emailCl;

Client c;

try{ Statement st=cn.createStatement();

    ResultSet rs=st.executeQuery(requete);

    .....


System.out.println("req select avec succès"+ex.getMessage());

}catch(SQLException ex)

    {System.err.println("probleme de req select"+ex.getMessage());

    }
```

4.3 Traitement des résultats d'une requête de sélection

 L'interface **ResultSet** : C'est une table de données représentant un jeu de résultats de base de données, qui est généralement générée en exécutant une instruction qui interroge la base de données : une requête de sélection.

Au départ le curseur est positionné juste avant la première ligne du résultat, la méthode **next()** permet d'avancer d'enregistrement en enregistrement d'une manière séquentielle et lorsqu'il n'y a plus de lignes dans l'objet ResultSet elle renvoie false.

Un objet ResultSet est automatiquement **fermé** lorsque l'objet Statement qui l'a généré est fermé, réexécuté ou utilisé pour récupérer le résultat suivant d'une séquence de résultats multiples.

<<interface>> ResultSet
+ boolean next() throws SQLException + void close() throws SQLException + getString (numCol) ou getString(nomColonne) + getInt (numCol) ou getInt(nomColonne) +

- ❖ **RS.next()** : Déplace le curseur d'une ligne vers l'avant à partir de sa position actuelle.
- ❖ **RS.close()** : Libère la base de données de cet objet ResultSet et les ressources JDBC immédiatement au lieu d'attendre que cela se produise lors de sa fermeture automatique.
- ❖ **RS.get...(...)** : Pour récupérer les données dans chaque colonne, l'interface ResultSet propose plusieurs méthodes adaptées aux types des données récupérées :
 - ❖ getString(NumColonne),
 - ❖ getInt(NumColonne),
 - ❖ getDate(NumColonne), etc.

Exemple : `System.out.println(resultat.getInt(1)+" "+resultat.getString(2));`

4.4 Exécution d'une requête de mise à jour

4.4.1 Première méthode de mise à jour avec l'interface Statement :

Si on veut utiliser une des commandes (insert, update ou delete), on utilisera la méthode **executeUpdate**. Cette méthode retourne le nombre de lignes modifiées.

Exemple :

```
package DAO;

import Gestion.Client;
import Gestion.Facture;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

public class DAOFacture {
```

```
public static boolean ajouter(Facture f)
{
    Connection cn=LaConnexion.seConnecter();

    String requete= "insert into `facture` values ('"+f.getNumFact()+"', '"+f.getDate-
Fact()+"', '"+f.getTotalFact()+"', '"+f.getRefCli()+"')";

    try{
        Statement st=cn.createStatement();
        int n=st.executeUpdate(requete);
        if (n>=1)
            System.out.println("ajout réussi");
        return true;

    }catch(SQLException ex)
    {
        System.out.println("problème de requête"+ex.getMessage());
    }
    return false;
}
```

4.4. Deuxième méthode de mise à jour avec l'interface PreparedStatement:

✚ **L'interface PreparedStatement** définit les méthodes pour un objet qui va encapsuler une requête précompilée. Ce type de requête est particulièrement adapté pour une exécution répétée d'une même requête avec des paramètres différents.

Lors de l'utilisation d'un objet de type **PreparedStatement**, la requête est envoyée au moteur de la base de données pour que celui-ci prépare son exécution.

Un objet qui implémente l'interface **PreparedStatement** est obtenu en utilisant la méthode **prepareStatement()** d'un objet de type **Connection**. Cette méthode attend en paramètre une chaîne de caractères contenant la requête SQL. Dans cette chaîne, chaque paramètre est représenté par un caractère ?.

Un ensemble de méthode **setXXX()** (ou XXX représente un type primitif ou certains objets tels que **String**, **Date**, **Object**, ...) permet de fournir les valeurs de chaque paramètre défini dans la requête. Le premier paramètre de ces méthodes précises le numéro du paramètre dont la méthode va fournir la valeur. Le second paramètre précise cette valeur.

Exemple

```
String requete = "insert into `client` values(?,?,?,?)";

try{
    PreparedStatement pst=cn.prepareStatement(requete);
    pst.setString(1,p1);
    pst.setString(2,p2);
    pst.setString(3,p3);
    pst.setString(4,p4);

    int n=pst.executeUpdate();
    if (n>=1)
        System.out.println("ajout réussi");

} catch(SQLException ex)
{
    System.out.println("problème de requête"+ex.getMessage());
}
```