

Une introduction à JDBC

Table des matières

1	Présentation de JDBC	2
1.1	Les objectifs de JDBC	2
1.2	JDBC dans un client léger	2
1.3	JDBC dans une architecture J2EE	2
1.4	Architecture Logicielle	3
1.5	Les pilotes JDBC	3
2	Utilisation de JDBC	4
2.1	Squelette de notre exemple	4
2.2	Déclaration du pilote JDBC	4
2.3	Connexion à la base de données	4
2.4	Les requêtes en JDBC	5
2.5	Programme principal	6
2.6	L'interface <code>java.sql.ResultSet</code>	6
2.7	Correspondance des types Java / SQL	7
2.8	Correspondance des dates et heures	7
3	Modification de la base	7
3.1	Insertion de lignes	7
3.2	Difficultés à manipuler des données	7
3.3	SQL Préformaté	8
3.4	Appel de procédures stockées en base	8
3.5	Erreurs et warnings	9
3.6	Gestion des transactions	9
4	Obtenir des informations sur la BD	9
4.1	Méta Informations sur les <i>Result Set</i>	9
4.2	Méta Informations sur la B.D.	9
5	JDBC version 2.1	10
5.1	Nouvelle version des <i>ResultSet</i>	10
5.2	Mise en oeuvre de ces « <i>Result Set</i> »	10
5.3	Déplacement dans un « <i>Result Set</i> »	10
5.4	Modification d'un « <i>Result Set</i> »	10
5.5	batch updates	11
6	JDBC 3.0	11

6.1	Les <i>DataSource</i>	11
6.2	Les <i>RowSet</i>	11

1 Présentation de JDBC

1.1 Les objectifs de JDBC

JDBC = *Java Data Base Connectivity*

JDBC **est basé** sur

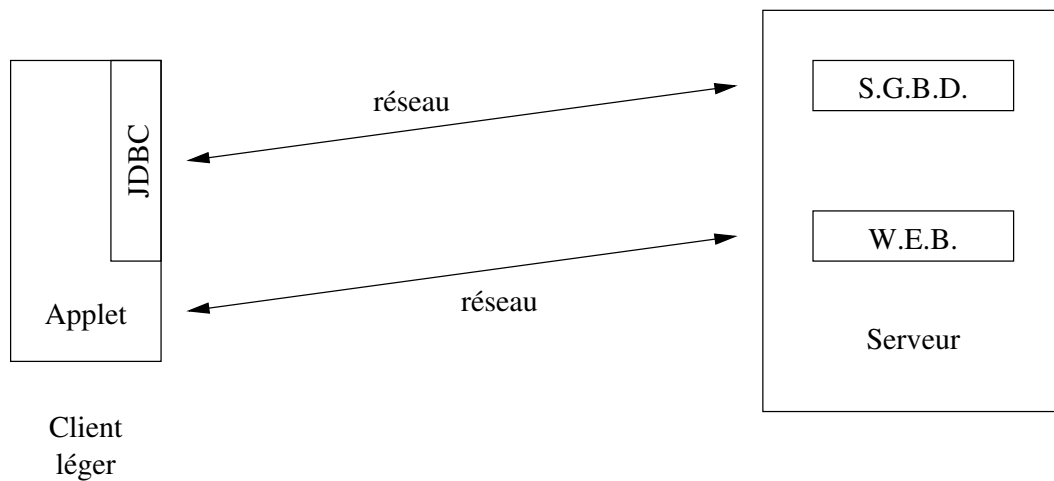
- ANSI SQL-2
- ODBC (Microsoft),
- API Propriétaires,
- SQLX/OPEN CLI (Call Level Interface).

Objectifs :

- Simple,
- Complet (en cours...),
- Portable,
- Modules réutilisables et/ou génériques,
- Intégration aux ateliers de développement.

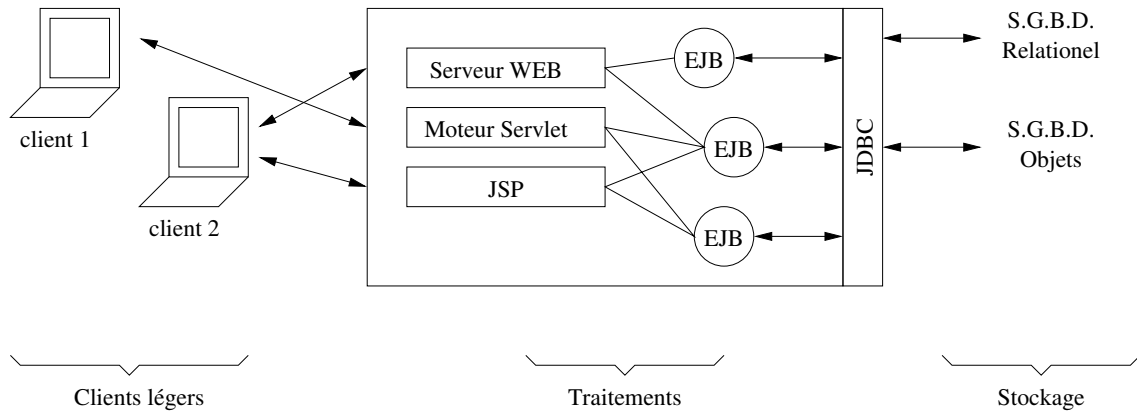
1.2 JDBC dans un client léger

Utilisation de JDBC dans un client léger :

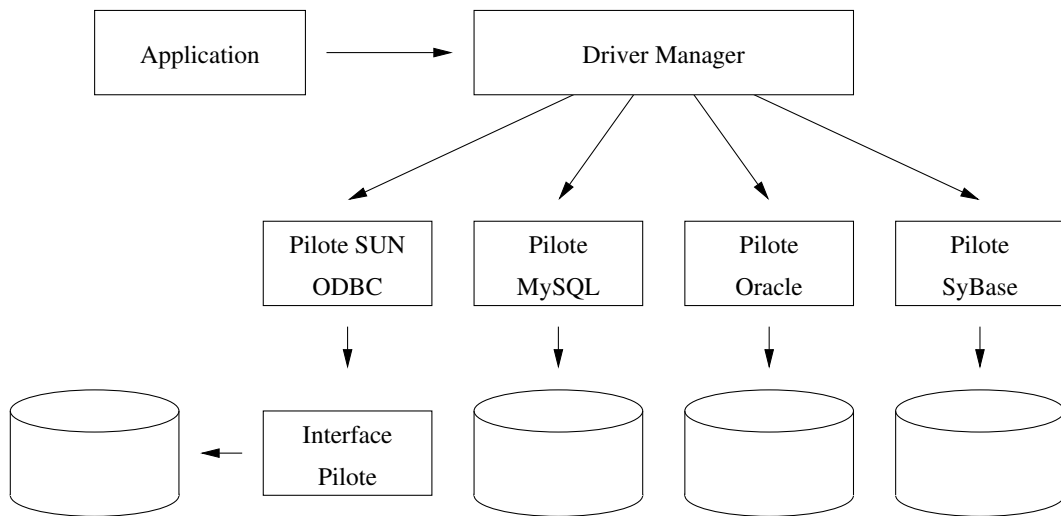


1.3 JDBC dans une architecture J2EE

Architecture d'exécution répartie dans la plateforme J2EE :

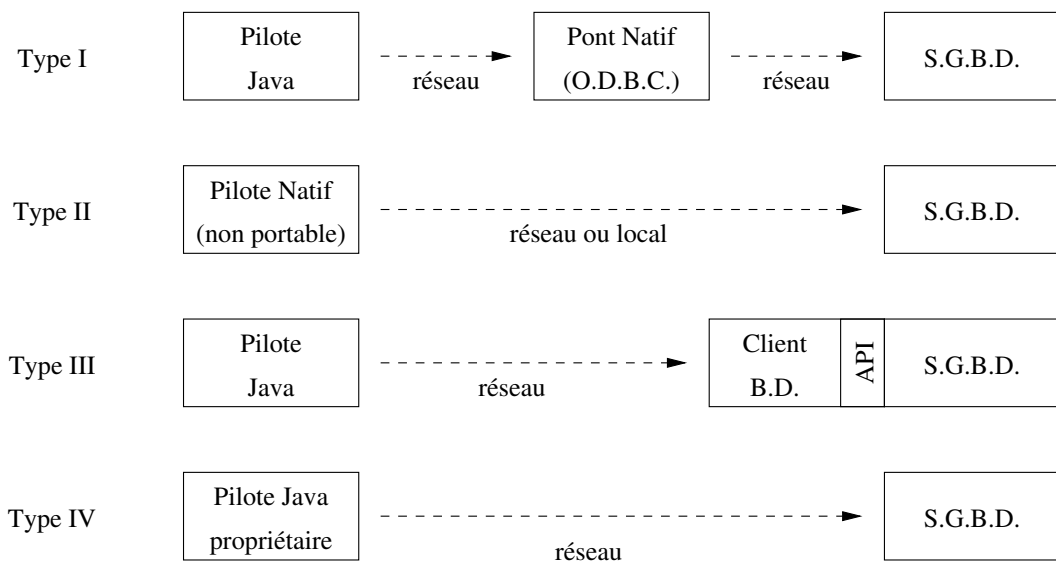


1.4 Architecture Logicielle



1.5 Les pilotes JDBC

Il existe quatre types de pilote :



2 Utilisation de JDBC

2.1 Squelette de notre 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 paquetage `java.sql` regroupe les interfaces et les classes de l'API JDBC.

2.2 Déclaration du 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.

2.3 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`

Quelques exemples (à chercher dans la documentation du pilote) :

```
jdbc:oracle://srv.dil.univ-mrs.fr:1234/dbtest
```

```
jdbc:odbc:msql;USER=fred;PWD=secret
```

2.4 Les requêtes en JDBC

Un exemple d'utilisation :

```
final String PERSONNES = "SELECT nom, prenom, age FROM personne ORDER BY age";

public void listPersons() throws SQLException {
    Connection conn = null;
    try {
        // create new connection and statement
        conn = newConnection();
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery(PERSONNES);

        while (rs.next()) {
            System.out.printf("%-20s | %-20s | %3d\n", //
                rs.getString(1), rs.getString("prenom"), rs.getInt(3));
        }
    } finally {
        // close result, statement and connection
        if (conn != null) conn.close();
    }
}
```

La version **try-with-resources** :

```
final String PERSONNES = "SELECT nom, prenom, age FROM personne ORDER BY age";

public void listPersons() throws SQLException {
    try (Connection conn = newConnection()) {
        // create new statement
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery(PERSONNES);

        while (rs.next()) {
            System.out.printf("%-20s | %-20s | %3d\n", //
                rs.getString(1), rs.getString("prenom"), rs.getInt(3));
        }
    }
}
```

Conseils :

- Évitez d'utiliser `SELECT * FROM ...` (coûteux en transfert),
- Attention à ne pas **dispenser** les noms SQL dans votre code Java. Donnez des noms **locaux** à vos colonnes :

```
SELECT nom AS nomFamille, prenom AS ...
```

- Faites le **maximum** de travail en SQL et le **minimum** en Java.
- **Minimisez** le nombre de connexions ouvertes. Utilisez un *pool* de connexions si possible.
- 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.

2.5 Programme principal

Mise en oeuvre et gestion des erreurs :

```
public static void main(String[] Args) {
    JdbcSample test = new JdbcSample();
    try {
        test.loadDriver();
        test.listPersons();
        ...
    } catch (ClassNotFoundException e) {
        System.err.println("Pilote_JDBC_introuvable!");
    } catch (SQLException e) {
        System.out.println("SQLException:␣" + e.getMessage());
        System.out.println("SQLState:␣␣␣␣␣" + e.getSQLState());
        System.out.println("VendorError:␣␣" + e.getErrorCode());
        e.printStackTrace();
    }
}
```

2.6 L'interface `java.sql.ResultSet`

Accès aux valeurs :

- `TYPE getType(int numeroDeColonne);`
- `TYPE getType(String nomDeColonne);`
- `boolean next();`

Le `TYPE` peut être

<code>Byte</code>	<code>Boolean</code>	<code>AsciiStream</code>
<code>Short</code>	<code>String</code>	<code>UnicodeStream</code>
<code>Int</code>	<code>Bytes</code>	<code>BinaryStream</code>
<code>Long</code>	<code>Date</code>	<code>Object</code>
<code>Float</code>	<code>Time</code>	
<code>BigDecimal</code>	<code>TimeStamp</code>	

2.7 Correspondance des types Java / SQL

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

2.8 Correspondance des dates et heures

Correspondance des dates

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

3 Modification de la base

3.1 Insertion de lignes

Un exemple :

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

int nb = st.executeUpdate(
    "INSERT INTO personne(Nom, Age) " +
    "VALUES(' " + nom + " ', " + age + ")"
);

System.out.println(nb + " ligne(s) insérée(s)");
st.close();
```

Ce principe est aussi utilisable pour les instructions `UPDATE` et `DELETE`.

3.2 Difficultés à manipuler des données

Un exemple :

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

int nb = st.executeUpdate(
    "UPDATE personne " +
    "SET Age=" + age + " " +
    "WHERE Nom='" + nom + "'"
);
```

Inconvénients : solution coûteuse (boucle) et difficile à mettre en oeuvre.

Injection SQL : si la variable `nom` 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.

3.3 SQL Préformaté

Code SQL avec partie variable :

```
PreparedStatement st = conn.prepareStatement(
    "UPDATE personne SET Age=? " +
    "WHERE Nom=? "
);

for( ... ) {
    st.setInt(1, age[i]);
    st.setString(2, nom[i]);
    st.execute();
}
```

Avantages : compilation unique et paramètres binaires plus faciles à passer.

3.4 Appel de procédures stockées en base

Un exemple :

```
CallableStatement st = conn.prepareCall(
    "{call ma_procedure[(?,?)]}" );
    // ou {? = call nom_de_fonction[(?, ..., ?)]}

    // fixer le type de paramètre de sortie
    st.registerOutParameter(2, java.sql.Types.FLOAT);

    st.setInt(1, valeur); // fixer la valeur du paramètre

    st.execute();
    System.out.println("résultat=" + st.getFloat(2));
```

Avantages :

- efficacité (moins de transfert de données),
- compilation des procédures

Inconvénient : pas de norme !

3.5 Erreurs et warnings

La classe `java.sql.SQLException` enrichit la classe `java.lang.Exception` :

- `SQLState` : description de l'erreur au format XOPEN,
- `getNextException()`

La classe `java.sql.SQLWarning` enrichit la classe `java.sql.SQLException` :

- `getWarnings()` : *Warning* suivant (il réalise des appels répétés).

3.6 Gestion des transactions

Le mode par défaut est « *Auto Commit* » :

- `connexion.setAutoCommit(false);`
- `connexion.commit();`
- `connexion.rollback();`

4 Obtenir des informations sur la BD

4.1 Méta Informations sur les Result Set

Exemple :

```
ResultSetMetaData m = rs.getMetaData();
```

Informations disponibles :

- nombre de colonnes,
- Libellé d'une colonne,
- table d'origine,
- type associé à une colonne,
- la colonne est-elle *nullable* ?
- etc.

Avantages :

- code indépendant de la requête,
- code réutilisable !

4.2 Méta Informations sur la B.D.

Exemple :

```
DataBaseMetaData dbmd = connexion.getMetaData();
```

Informations disponibles :

- tables existantes dans la base,
- nom d'utilisateur,
- version du pilote,
- prise en charge des jointure externes ?,
- etc.

5 JDBC version 2.1

Contenu :

- *Core JDBC 2.1* : extension de `java.sql`,
- *JDBC 2.0 optional package* : nouveau package `javax.sql`,

5.1 Nouvelle version des ResultSet

Il existe quatre types de `ResultSet` :

- Réglage de la connexion à la base de données :
 - ▷ **Scroll-insensitive** : vision figée du résultat de la requête au moment de son évaluation (JDBC 1.0).
 - ▷ **Scroll-sensitive** : le *Result Set* montre l'état courant des données (modifiées/détruites).
- Réglage des mises à jour :
 - ▷ **Read-only** : pas de modification possible (JDBC 1.0) donc un haut niveau de concurrence.
 - ▷ **Updatable** : possibilité de modification donc pose de verrou et faible niveau de concurrence.

5.2 Mise en oeuvre de ces « Result Set »

```
Statement stmt = con.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
ResultSet rs = stmt.executeQuery("SELECT_*_*FROM_Personne");
```

Ce `ResultSet` est **modifiable** mais il ne reflète pas les **modifications** faites par d'autres transactions.

5.3 Déplacement dans un « Result Set »

- `rs.first()`;
- `rs.beforeFirst()`;
- `rs.next()`;
- `rs.previous()`;
- `rs.afterLast()`;
- `rs.absolute(n)`;
- `rs.relative(n)`;

5.4 Modification d'un « Result Set »

Modification :

```
rs.absolute(100);  
rs.updateString("Nom", "Fred");  
rs.updateInt("Age", 30);  
rs.updateRow();
```

Destruction :

```
rs.deleteRow();
```

Insertion de lignes :

```
rs.moveToInsertRow();  
rs.updateString("Nom", "Fred");  
rs.updateInt("Age", 30);  
rs.insertRow();  
rs.first();
```

5.5 batch updates

Regroupement de plusieurs mise à jour :

```
connexion.setAutoCommit(false);  
Statement st = connexion.createStatement();  
  
st.addBatch("INSERT_...");  
st.addBatch("INSERT_...");  
  
int[] nb = st.executeBatch();
```

On peut combiner des `PreparedStatement` et des « Batch updates ».

6 JDBC 3.0

Améliorations :

- ▶ nouveau package `javax.sql.*`
- ▶ *Save point* : pose de point de sauvegarde.
- ▶ *Connection Pool* : Gestion des ensembles de connexions partagées.
- ▶ Support des séquences (auto génération de valeurs).
- ▶ Augmentation et mise à jour des types (CLOB, BLOB, références SQL3).
- ▶ Prise en compte de SQL-3.

6.1 Les DataSource

L'interface `javax.sql.DataSource` permet :

- d'obtenir une connexion JDBC,
- de gérer un *pool* de connexion,
- de faire disparaître les constantes (placées dans un annuaire JNDI ou un fichier de configuration).

6.2 Les RowSet

L'accès aux données est encapsulé dans un seul Bean :

```
javax.sql.rowset.RowSetFactory factory = RowSetProvider.newFactory();
javax.sql.rowset.RowSet rs = factory.createCachedRowSet();

rs.setUrl("jdbc:mysql://localhost/dbessai");
rs.setCommand("SELECT * FROM personne");
rs.setUsername("massat");
rs.setPassword("...");
rs.setConcurrency(ResultSet.CONCUR_UPDATABLE);
rs.execute();

while (rs.next()) {
    System.out.printf("Nom: %s\n", rs.getString("nom"));
}
rs.close();
```

Il existe trois types de RowSet :

- JDBCRowSet (basé sur JDBC),
- CachedRowSet (déconnecté de la base),
- WebRowSet (échange basé sur des flux XML),