JAVA ET BASES DE DONNÉES

Architecture 3-tiers

- Serveur applicatif situé entre le client et la BD :
  - On peut changer une partie sans toucher au reste.
  - Possibilité de se connecter à différentes BD sans changer le client.
  - Séparation de la présentation, gestion des données, …
- Mais plus complexe à mettre en oeuvre, à configurer, maintenir…

Architecture N-tiers

- On rajoute autant de tiers que nécessaire :
  - Permet de dissocier toutes les parties de l'application.
  - Utilisation d'API (interfaces) pour que les différentes parties communiquent de manière transarente.
  - On peut changer une partie si l'API ne change pas.

- JDBC permet cela.

JDBC

- Java DataBase Connectivity

- API (Application Programming Interface) JAVA de connexion à des bases de données.
- Indépendant de la base de données :
  - On utilise la même syntaxe quelle que soit la BD.
  - Modèle 3-tiers ou N-tiers.

Pilotes

- Utilisation de pilotes de connexion aux BD :
  - JDBC charge le pilote approprié
  - Via le pilote on communique avec la BD

- On peut donc se connecter à n'importe quelle BD à condition d'avoir le pilote associé.
- On doit pouvoir changer de BD sans toucher au code (uniquement le choix du pilote) :
  - Tous les pilotes doivent avoir la même interface.

Pilotes

- On dispose de 4 types de pilote :
  - type 1 : bridge ou passerelle
  - Type 2 : pilotes natifs
  - Type 3 : modèle 3 couches
  - Type 4 : pilote pur

Pilotes de type 1

- Passerelle vers d'autres drivers applicatifs :
  - Généralement vers ODBC (Open Database Connectivity de Microsoft).
  - Les requêtes sont traduites en ODBC.
  - ODBC fait ensuite des requêtes sur l'API de la BD.

- Peu efficace :
  - Conversion en ODBC qui va faire le travail.
  - Il faut installer ODBC (dépend de la plate-forme).
  - À n'utiliser qu'en l'absence de drivers d'autres types.

Pilotes de type 2

- Pilote natifs :
  - Mélange de Java et d'autres langages.
  - Les appels JDBC sont convertis en appel vers l'API native de la BD (généralement dans un autre langage).

- Appels natifs :
  - Rapide car codés de manière efficace.

Pilotes de type 3

- Utilisent un serveur intermédiaire
  - Le pilote traduit les requêtes dans un format standard.
  - Le serveur intermédiaire reçoit les requêtes, effectue les traductions nécessaires et contacte la BD.
  - Un seul pilote permet d'accéder à différentes BD.

- Il faut disposer d'un serveur intermédiaire :
  - Installation/maintenance.
  - Peut faire plus de chose :
    - cache de requêtes, logs, répartition de charge, …

Pilotes de type 4

- Pilote d'accès direct à la BD
  - Peuvent communiquer directement avec la BD :
    - Gain de performances en général.
  - Codés en Java (indépendant de la plate-forme) :
    - Debug potentiellement plus simple car on a tout en main.
  - Utilisent les librairies réseaux de Java.

- Généralement fournis par les éditeurs de BD.
- Très fortement liés à la BD.

Installation

- Trois étapes :
  - Installer un serveur de base de données.
  - Installer JDBC.
  - Créer une connexion à la base et y accéder.

- On va ici se baser sur une base MySQL :
  - Installation d'un serveur mysql :
    - Directement ou via WampServer ou EasyPhp.
    - Installé en salle DANT
  - Téléchargement du pilote jdbc mysql :
    - mysql-connector-java-5.1.23-bin.jar
  - Copie de l'archive dans WEB-INF/lib/

API JDBC

Les classes associées

- DriverManager :
  - Chargement et choix des drivers.
- Connection :
  - Permet d'exécuter plusieurs requêtes à la BD.
- Statement :
  - Un requête à la BD.
- ResultSet :
  - Le résultat d'un requête à la BD.

- Driver :
  - Permet de se connecter à la BD.
  - Pas utilisé directement en général.

Accéder à la base

- Six étapes :
  - Chargement du driver spécifique à la base.
  - Création d'un objet Connection :
    - Offre des moyens de se connecter à la base.
  - Création d'un objet Statement :
    - Permet de préparer l'exécution d'un requête.
  - Exécuter la requête.
  - Traiter le résultat :
    - Via l'objet ResultSet par exemple.
  - Se déconnecter.

Chargement du driver

- Le DriverManager teste tous les drivers enregistrés :
  - Utilise le premier qui fonctionne.
  - Le driver s'enregistre auprès du DriverManager à sa création :
    - Class.forName("com.mysql.jdbc.Driver");
    - Ou Class.forName("…").newInstance() en cas de problème
    - Connection c = DriverManager.getConnection(...);
- Solution alternative :
  - Driver d = new foo.bar.MyDriver();
  - Connection c = d.connect(...);

DriverManager - méthodes

- Connection getConnection (String url, String user, String passd)
  - Se connecte à l'URL avec les identifiants donnés.
  - Retourne un objet Connection.
  - Peut lever une exception java.sql.SQLException.

- Class.forName("com.mysql.jdbc.Driver");
  - Crée une instance du driver.
  - Enregistre le driver auprès du gestionnaire de drivers.

URL de connexion

- Permet de localiser le serveur de BD et son type.
- Format standard :
  - jdbc:sousprotocole:source
  - Chaque pilote a un sous-protocole
  - Chaque sous-protocole a une syntaxe pour la source

- Exemple :
  - jdbc:mysql://localhost:3306/ntw
  - On utilise le pilote mysql
  - On se connecte en local sur le port 3306 et on utilise la base nommée ntw

Exemple de base

```
Connection conn = null;

try {
     Class.forName("com.mysql.jdbc.Driver");
     String url = "jdbc:mysql://localhost:3306/ntw";
     conn = DriverManager.getConnection(url, "root", "");
     ...
} catch(SQLException e) {
     // S'il y a un problème durant la connexion
     e.printStackTrace();
     ...
} catch(Exception e) {
     // Si le driver JDBC ne peut pas être chargé
     e.printStackTrace();
     ...
}
```

Objet Connection

- Une connexion représente une session avec une BD :
  - On peut faire plusieurs requêtes avec une connexion.
- On peut avoir plusieurs connections simultanées avec une même BD (en général).
- La connexion fournit aussi des informations sur la base, les tables et les champs.

Gestionnaire de connections

- Utilisation du gestionnaire de connections :
  - Responsable de la gestion des connexions ouvertes vers le SGBD.
  - Quand l'application a besoin d'une connection, elle la demande au gestionnaire de connexions.

- Evite d'ouvrir et de fermer des connexions en permanence : plus rapide.

Objet Connection - méthodes

- Plusieurs méthodes pour créer des requêtes :
  - Statement createStatement() :
    - Méthode générale.
  - PreparedStatement prepareStatement(String sql) :
    - Un requête exécutée plusieurs fois avec différentes valeurs.
    - Précompilation pour gagner du temps.
  - CallableStatement prepareCall(String sql) :
    - Pour les procédures ou les fonctions stockées par le SGBD.
    - Dépend des SGBD.
- Il faut utiliser le bon objet selon ce que l'on fait :
  - Plus efficace.

Objet Statement - méthodes

- ResultSet executeQuery(String)
  - Exécute une requête qui retourne un seul ensemble de résultat.
- int executeUpdate(String)
  - Exécute une requête qui ne retourne pas de résultat (SQL INSERT, UPDATE, DELETE, …)
  - Retourne le nombre de lignes affectées par la requête.
- boolean execute(String)
  - Exécute une requête qui peut retourner plusieurs résultats.
  - Méthode générale.
  - Voir getResultSet et getMoreResults

Statement - exemple

```
Statement stmt = conn.createStatement();
int nRows = stmt.executeUpdate(
    "INSERT INTO ntw_table1" +
    "VALUES (65, 245)");
```

PreparedStatement - exemple

```
PreparedStatement st = conn.prepareStatement(
    "UPDATE ntw_table1 " +
    "SET val = ? " +
    "WHERE id = ? " );

for( ... ) {
    st.setString(1, val[i]);
    st.setInt(2, id[i]);
    st.executeUpdate();
}
```

CallableStatement - exemple

```
CREATE PROCEDURE simpleproc (OUT param1 INT)
BEGIN
    SELECT COUNT(*) INTO param1 FROM t;
END
```

```
String sql = "call simpleproc(?)";
CallableStatement statement = conn.prepareCall(sql);
statement.registerOutParameter(1,Types.INTEGER);
statement.execute();

out.println(statement.getInt(1));
```

ResultSet

- ResultSet contient les lignes du résultat de la requête :
  - C'est une abstraction de la table SQL résultant de la requête.
    - Tableau à deux dimensions, colonnes et enregistrements.
  - On peut le parcourir simplement avec la méthode next()
    - Les résultats sont récupérés un par un.
    - Pas de retour en arrière.
  - Un seul ResultSet par Statement utilisable à la fois.

```
ResultSet rs = stmt.executeQuery(...);
while (rs.next()) {
    out.println(rs.getString("val"));
}
```

Récupération d'une colonne

- Type get[Type](int columnIndex)
  - Retourne le champs demandé dans le bon type
  - Par exemple : int getInt(5) ; string getString(3)
  - Les champs commencent à 1
- Type get[Type](String columnName)
  - Identique mais en utilisant les noms des champs au lieu de leur indice.
  - Moins efficace mais plus pratique/lisible.

- int findColumn(String columnName)
  - Permet d'obtenir l'indice d'une colonne à partir de son nom.

Classe ResultSet – autres méthodes

- boolean next()
  - Charge la ligne suivante du résultat.
  - Le premier appel charge la première ligne.
  - Retourne faux à la fin.

- void close()
  - Libère le ResultSet.
  - Permet de réutiliser le Statement associé.
  - Appelé automatiquement par certaines méthodes de Statement.

Types SQL et Types Java

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[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Exemple en JSP

```
<%@ page language="java" import="java.sql.*" %>
<%
try {
	Class.forName("com.mysql.jdbc.Driver");
	Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/ntw", "", "");
%>
<table>
<tr><th>Identifiant</th><th>Valeur</th></tr>
<%
	Statement s = con.createStatement();
	ResultSet rs = s.executeQuery("SELECT * FROM ntw_table1");
	while(rs.next()) {
%>
		<tr> <td> <%= rs.getInt("id") %> </td>
		<td> <%= rs.getString("val") %>  </td></tr>
<%	} %>
</table>
<%
} catch (Exception e) {	out.println(e);}
%>
```

Méta-données

- On dispose d'informations supplémentaires dans le resultSet :
  - Objet ResultSetMetaData :

```
ResultSetMetaData rsmd;
rsmd = results.getMetaData();
```

- On peut avoir des informations telles que :
  - Nombre de colonnes
  - Nom de chaque colonne
  - Type de chaque colonne

- Peut permettre d'écrire des classes génériques :
  - Exemple : affichage d'un tableau automatiquement.

Gestion des dates

- Trois classes sont définies pour tenter de standardiser un peu la gestion des dates
  - java.sql.Timestamp
    - year, month, day, hours, minutes, seconds, nanoseconds
  - java.sql.Date
    - year, month, day
  - java.sql.Time
    - hours, minutes, seconds

isNull

- En SQL, NULL signifie qu'un champ est vide
- Avec JDBC, on peut savoir si un champ est null en utilisant wasNull() après avoir accédé au champ.

```
int i = rs.getInt(1);
if (rs.wasNull())
    out.println("le champ était vide");
else
    out.println(i);
```

Timeout

- Permet de fixer une limite au temps d'exécution d'une requête :
  - Si le temps est dépassé, une SQLException est levée.

```
Statement statement = ...
out.println(statement.getQueryTimeout());
// temps exprimé en secondes
statement.setQueryTimeout(2);
```

JSP et JDBC

- On veut généralement dissocier code et présentation :
  - Avoir des requêtes SQL dans le JSP n'est donc pas idéal.
  - Il est préférable de créer des interfaces d'accès à la base.

- Calculs en Java ou en SQL ?
  - Pour des opérations simples, SQL est en général beaucoup plus efficace.
  - Pour des cas complexes, Java peut être beaucoup plus simple/lisible.

Javax.sql

- Extension de java.sql

- Quatre nouveautés par rapport à java.sql :
  - DataSource remplace DriverManager.
  - Gestion d'un pool de connexions.
  - Transactions distribuées (une transaction sur plusieurs sources de données).
  - RowSets remplace ResultSet.

- On fait l'impasse dans ce cours :
  - A aller voir pour ceux qui veulent faire du JDBC de manière plus approfondie.

JDBC ET TRANSACTIONS

Transactions

- Exemple classique : virement bancaire :
  - Retirer de l'argent sur un compte et en ajouter sur un autre.

- Transaction = modification d'une BD avec plus d'une requête :
  - Atomique : la suite de requête n'est pas divisible, soit tout est exécuté, soit rien.
  - Cohérente : le résultat de l'exécution d'une transaction doit être cohérent même en cas d'annulation.
  - Isolée : si deux transactions sont exécutées en même temps, les modifications effectuées par l'une ne sont pas visibles par l'autre tant qu'elle n'est pas validée.
  - Durable : une transaction terminée ne peut pas être annulée ou recouverte par une autre.

Transactions JDBC

- Pas besoin d'ouvrir/fermer une transaction
  - Mode AutoCommit, si vrai (défaut) alors chaque requête est exécutée.
  - Si faux alors chaque requête est ajoutée à une transaction courant.

- Pour modifier l'autocommit :
  - Connection.setAutoCommit(boolean)
- En cas de transaction :
  - Connection.commit() pour exécuter la transaction
  - Connection.rollback() pour annuler la transaction

Exemple

```
try {
    con.setAutoCommit( false );
    statement = con.createStatement();
    statement.executeUpdate( update1 );
    statement.executeUpdate( update2 );
    // tout va bien jusque là
    con.commit();
} catch(SQLException e) {
    // en cas de problème
    con.rollback();
}
```