- J2EE

- JDBC
- Java Data Base Connectivity

J2EE: JDBC
Java Data Base Connectivity

Pour les applications web utilisant une ou plusieurs bases de données, les servlets et l'API JDBC offrent une solution :

- efficace, la connexion est maintenue durant tout le cycle de vie de la servlet et non à chaque invocation
- flexible, l'accès à une base de données dépend peu de son fournisseur. Il est facile de porter une application web vers une base de données d'un autre fournisseur

J2EE: JDBC
Principe de fonctionnement

Permet à un programme Java d'interagir
- localement ou à distance
- avec une base de données relationnelle

Fonctionne selon un principe client/serveur
- client = le programme Java
- serveur = la base de données

Principe
- le programme Java ouvre une connexion
- il envoie des requêtes SQL
- il récupère les résultats
- il ferme la connexion

J2EE: JDBC
API

- l'API JDBC se trouve dans le paquetage **java.sql**
- l'API JDBC est constituée d'un ensemble d'interfaces pour interagir avec des bases de données
- l'API JDBC ne fournit donc aucune implémentation des comportements spécifiés par les interfaces
- l'API JDBC permet d'exécuter des instructions SQL et de récupérer les résultats
- La version la plus récente de l'API est la version 3.0

J2EE: JDBC
Pilotes JDBC

Le pilote JDBC a pour rôle de permettre l'accès à un système de bases de données particulier. Il gère le détail des communications avec un type de SGBD
A une API JDBC correspond plusieurs implémentations selon les fournisseurs (un type de driver par SGBD Oracle, Sybase, Access, ...). Il en existe pour la plupart des SGBD relationnels
Le pilote implémente l'interface java.sql.Driver
Il existe un driver générique JDBC-ODBC pour toutes les données accessibles par le standard ODBC

- SGBD : MS Access, SQL Server, Oracle, dBase, ...
- tableurs : Excel, ...
- tout autre application conforme ODBC

J2EE: JDBC
Gestionnaire de pilotes: DriverManager

`DriverManager` est une classe, gestionnaire de tous les drivers chargés par un programme Java

Chaque programme Java charge le (ou les) drivers dont il a besoin

L'enregistrement des pilotes est automatique

J2EE: JDBC
Connexion à une base de données

Disposer de l'URL d'un pilote pour le SGBD

```
com:mysql:jdbc:Driver
Sun:jdbc:odbc:JdbcOdbcDriver
```

Charger le pilote

```
Class.forName("com.mysql.jdbc.Driver");
// création d'une instance du pilote
// enregistrement automatique auprès du
// DriverManager
```

Etablir l'URL de la base

```
String dbUrl="jdbc:mysql://localhost:3306/maBD";
```

Ouvrir la connexion : le DriverManager choisit le pilote approprié à cette URL parmi l'ensemble des pilotes disponibles.

```
Connection dbcon =
DriverManager.getConnection(dbUrl,user,password);
```

J2EE: JDBC
Code de connexion : exemple (1/2)

```
try {
   //Déclaration du driver :
   Class.forName("com.mysql.jdbc.Driver");
   //Url de connexion
   Sting dbUrl="jdbc:mysql://localhost:3306/laboBD";
   //Profil/mot de passe
   String user = "root";
   String password = "19AB5";
   //connexion à la base
   dbcon =
   DriverManager.getConnection(dbUrl,user,password);
}
```

J2EE: JDBC
Code de connexion : exemple (2/2)

```
catch(ClassNotFoundException e ){
      // erreur de chargement du pilote
      }
catch(SQLException e ){
      // erreur d'obtention de la connexion
      }
finally{
      // fermeture de la connexion pour libérer
      // les ressources de la BD
...
      if (dbcon!=null) dbcon.close();
...
}
```

J2EE: JDBC
L'interface ResultSet

Un objet de type **ResultSet** est une table représentant le résultat d'une requête à une base de données

Le nombre de lignes du tableau peut être nul.

Les données de la table sont accessibles ligne par ligne grâce à un curseur

Un **ResultSet** gère un curseur sur la ligne courante de la table des données satisfaisant aux conditions de la requête.

J2EE: JDBC
Exécution d'une requête SQL de type Statement (1/2)

Le type de résultat obtenu dépend de la manière dont on souhaite manipuler le **`ResultSet`**

3 constantes de types :
`TYPE_FORWARD_ONLY` (le curseur se déplace uniquement vers l'avant)
`TYPE_SCROLL_INSENSITIVE` (le curseur se déplace en avant et en arrière mais le Resultset est insensible aux modifications apportées à la BD durant le traitement)
`TYPE_SCROLL_SENSITIVE` (le curseur se déplace en avant et en arrière mais les changements faits à la BD durant le traitement sont reflétés dans le ResultSet)

2 constantes de concurrence :
`CONCUR_READ_ONLY` (les données du ResultSet ne peuvent pas être modifiées)
`CONCUR_UPDATABLE` (des modifications de la BD peuvent être effectuées via le **`ResultSet`**)

Exécution d'une requête SQL de type Statement (1/2) Exemple

Créer un objet de la classe java.sql.Statement

```
Statement stmt = dbcon.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
// indique que le ResultSet est scrollable et non modifiable.
// ResultSet.CONCUR_UPDATABLE indiquerait qu'un seul utilisateur à la
fois peut modifier
// la donnée

Statement stmt = dbcon.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
// lorsqu'on apporte un changement au ResultSet, on peut y accéder par
la suite car le type est "scrollable" et on peut récupérer sa valeur car
le type est "SENSITIVE"
```

Pour que les modifications apportées au Resultset prennent effet dans la base, il faut appliquer les méthodes updateRow

J2EE: JDBC
Exécution d'une requête SQL de type Statement (2/2)

Par défaut, le ResultSet n'est ni scrollable, ni modifiable

```
Statement stmt = dbcon.createStatement();

dbcon.createStatement(TYPE_FORWARD_ONLY,CONCUR_READ_ONLY);
```

Construire la requête SQL

Soit la table **personne(nom,prenom,age)**

```
String reqSQL = "select * from personne";
```

Exécuter la requête

```
ResultSet rs = stmt.executeQuery(reqSQL);
```

J2EE: JDBC
Exploiter les résultats d'une requête (1/5)

A la création du **`ResultSet`**, le curseur est placé avant la première ligne (N°1).
Pour le déplacer, on dispose des méthodes :

- **`next()`** déplacement d'une ligne en avant
- **`previous()`** déplacement d'une ligne en arrière
- **`first()`** déplacement vers la première ligne
- **`last()`** déplacement vers la dernière ligne
- **`beforeFirst()`** déplacement avant la première ligne. Si le **`ResultSet`** est vide, cela n'a aucun effet.
- **`afterLast()`** déplacement après la dernière ligne. Si le **`ResultSet`** est vide, cela n'a aucun effet.
- **`relative(int i)`** déplace de i lignes par rapport à la position courante
- **`absolute(int i)`** déplace le curseur vers la i-ème ligne

J2EE: JDBC
Exploiter les résultats d'une requête (2/5)

Déplacement absolu **`absolute(int i)`**
i>0 déplacement vers la ligne numéro i
`rs.absolute(1)` => déplacement vers la ligne 1

i<0 déplacement à partir de la fin
`rs.absolute(-1)` => déplacement sur la dernière ligne
`rs.absolute(-3)` => si 30 lignes, déplacement vers ligne 28

Déplacement relatif **`relative(int i)`**
On spécifie le nombre de déplacements à partir de la ligne courante
`i>0` => déplacements en avant
`i<0` => déplacements en arrière

```
rs.absolute(4); // curseur sur la 4ème ligne
rs.relative(-3); // curseur sur la première ligne
rs.relative(2); // curseur sur la 3ème ligne
```

J2EE: JDBC
Exploiter les résultats d'une requête (3/5)

On récupère le contenu d'une ligne du **`ResultSet`** colonne par colonne
Les méthodes d'accès sont construites sur le modèle : **`getXXX`** (XXX étant un type primitif ou **`String`**)
La colonne est spécifiée soit par son nom, soit par son numéro (la première colonne est numérotée 1)

par son nom

```
String getString(String nomAttribut)
float getFloat(String nomAttribut)
```

par le numéro de la colonne qui lui correspond

```
String getString(int indexColonne)
float getFloat(int indexColonne)
```

idem pour les `int, double, long, boolean, Object`

J2EE: JDBC
Exploiter les résultats d'une requête (4/5)

Pour obtenir le n° de la ligne courante

```
int getRow()

rs.absolute(5);
int i = rs.getRow(); // i=5
```

Pour tester la position dans le **Resultset**

```
isFirst, isLast, isBeforeFirst, isAfterLast.

if (rs.isAfterLast() == false) {
rs.afterLast();
}
```

J2EE: JDBC
Exploiter les résultats d'une requête (5/5)

```
String reqSQL = "select * from personne";
ResultSet rs = stmt.executeQuery( reqSQL );
while (rs.next()){
   String prenom = rs.getString("prenom");
   int age = rs.getInt("age");
}
```

J2EE: JDBC
Mises à jour de table (1/2)

Il est préférable que le **`ResultSet`** soit scrollable lors d'une mise à jour

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

Pour mettre à jour, 2 méthodes:
`updateXXX` avec 2 paramètres (nom et valeur de la colonne)
`updateRow()` rendre effective la mise à jour

J2EE: JDBC
Mises à jour de table (2/2)

Nom	Prénom	Age
Meyer	René	54->37
Legrand	Léa	18
Dupont	Marcel->Robert	43

```
String reqSQL = "select * from personne";
ResultSet rs = stmt.executeQuery( reqSQL );
rs.next();
rs.updateInt("Age",37);
rs.last();
rs.updateString("Prénom","Robert");
rs.updateRow();
```

J2EE: JDBC
Requêtes préparées

L'exécution de chaque requête à une BD nécessite 4 étapes :

- analyse
- compilation
- optimisation
- Exécution

Pour des requêtes identiques, les 3 premières étapes n'ont pas à être effectuées de nouveau.

Avec la notion de requête préparée, les 3 premières étapes ne sont effectuées qu'une seule fois.

JDBC propose l'interface **`PreparedStatement`** qui dérive de l'interface **`Statement`** pour modéliser cette notion.

J2EE: JDBC
Requêtes préparées

Avec l'interface **`Statement`**, on écrivait :

```
Statement smt = dbcon.createStatement();
ResultSet rs = smt.executeQuery("SELECT * FROM personne" );
```

Avec l'interface **`PreparedStatement`**, on écrit :

```
PreparedStatement pSmt =
dbcon.prepareStatement("SELECT * FROM personne" );
ResultSet rs = pSmt.executeQuery();
```

On voit que la requête est préparée à l'avance et n'est donc pas transmise en argument au moment de l'exécution (**`executeQuery()`**).

J2EE: JDBC
Requêtes préparées

Pour préparer des requêtes paramétrées de la forme :
`SELECT nom FROM Personnes WHERE age > ? AND adresse = ?`

On utilise les **PreparedStatement** avec les méthodes
`setType(numéroDeLArgument, valeur)`
`Type` représente le type de l'argument
`numéroDeLArgument` représente le numéros de l'argument (commençant à 1 dans l'ordre d'apparition dans la requête).
`valeur` représente la valeur qui lui est associée

Exemple :

```
PreparedStatement pSmt = dbcon.prepareStatement("SELECT nom FROM
Personne WHERE age > ? AND prenom=?" );
pSmt.setInt(1, 22);
pSmt.setString(2, "Adrien");
ResultSet rs = pSmt.executeQuery();
```

J2EE: JDBC
Requêtes préparées

```
PreparedStatement psmt = dbcon.prepareStatement("update
personne set nom = ? where prenom like ?");
psmt.setString(1, "Bauer");
psmt.setString(2, "René");
int i = psmt.executeUpdate():
```

On a changé en `Bauer` le nom de la personne dont le prénom est `René`
La valeur retournée par `executeUpdate()` est le nombre de lignes mises à jour