Fabien Coelho
MINES ParisTech
fabien.coelho@mines-paristech.fr

Composé avec LaTeX

JDBC : Java Data Base Connection

présentation du modèle JDBC

driver classe `DriverManager` et URL de connexion

connexion classes `Connection Statement ResultSet`

avancé classes `PreparedStatement CallableStatement`

conversions types SQL ↔ Java

conseils d'utilisation dans une application

Id: jdbc.tex 3985 2017-03-14 08:42:02Z fabien

Systèmes de Gestion de Bases de Données

DBMS : Data Base Management Systems

données volumineuses et structurées

théorie fondée sur l'algèbre relationnelle

services intégrité, cohérence, droits d'accès

SQL langage d'interrogation standard

ODBC interface standard pour C ou C++

acteurs Oracle, Sybase, Informix, IBM DB2, MS SQL Server...

libres PostgreSQL, MySQL, mSQL...

Contenu JDBC

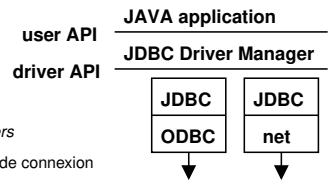
API connexion de JAVA à DBMS

- interfaces et classes prédéfinies
- plusieurs versions !
  JDBC 1.0 SQL 2 : `java.sql`,
  JDBC 2.0 SQL 3 : `java.sql` plus extensions `javax.sql`

standard et portable

- basé sur SQL comme langage de requête !
- modèle offert très proche de ODBC
- gestionnaire de *drivers*
- nommage des bases via une *URL*

les 2 API de JDBC



API drivers

ajout de nouveaux *drivers*

driver = un programme de connexion

interaction avec le *driver manager*

API utilisateur

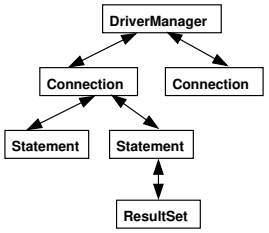
connexion via un *driver* + *URL*

requêtes SQL et résultats

traductions types SQL ↔ JAVA

metadata informations sur la base

Modèle de JDBC



DriverManager gestionnaire

enregistre les différents drivers

ouverture des `Connection`s

Connection connexion ouverte

permet de créer des `Statement`s

Statement une requête

exécution de requêtes SQL

récupération du `ResultSet`

ResultSet un résultat

énumération lignes et colonnes

Survol d'une connexion à une base de données

1. chargement du drivers *à la main*

```
Class.forName("org.postgresql.Driver");
```

2. connexion à la base avec une URL

```
Connection c = DriverManager.getConnection("jdbc:...", ...);
```

3. création d'un statement (plusieurs types de statements)

```
Statement s = c.createStatement();
```

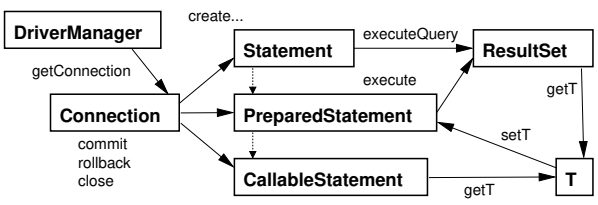
4. exécution d'une requête SQL (selon type de requête)

```
ResultSet r = s.executeQuery("SELECT ...");
```

5. extraction du résultat

```
while (r.next()) col = r.getString(1);
```


Relations principales



Package `java.sql`

classes et interfaces implémentées par les *drivers*

principal `DriverManager Connection Statement ResultSet`

informations `ResultSetMetaData DatabaseMetaData`

utils `Date Time SQLException SQLWarning...`

extensions `Array Blob Clob Ref Struct...`

Package `javax.sql`

— *connection pool*, évènements, `Row...`

Classe de fonctions `DriverManager`

— gestion des différents *drivers*, noms enregistrés auprès de Sun

4 types	binaire	réseau
générique	ODBC	protocole standard ?
		TDS *(Tabular Data Stream)*
spécifique	librairie du DBMS	protocole du DBMS

— enregistrement **automatique** des *drivers*
au chargement `Class.forName(...)` + `CLASSPATH`
zone d'initialisation des classes `static { ... }`

— Property `jdbc.drivers` : classes séparés par des `:`
`shell> java -Djdbc.drivers=org.postgresql.Driver ...`

```
Class.forName("org.postgresql.Driver"); // Postgres
Class.forName("org.gjt.mm.mysql.Driver"); // mysql
Class.forName("twz1.jdbc.mysql.jdbcMysqlDriver"); // z1MySQL
```


Établissement d'une connexion

— trois **fonctions** de `DriverManager`
— utilisation d'URL pour désigner le serveur
— délais maximum d'attente, flux de messages d'erreurs...

```
public class DriverManager {
 static Connection getConnection(String url)
   throws SQLException;
 static Connection getConnection(String url, String login, String p
   throws SQLException;
 static Connection getConnection(String url, Properties info)
   throws SQLException;
 static void setLoginTimeout(int seconds);
 static void setLogWriter(PrintWriter pw);
}
```


URL de JDBC

— protocole, *driver*, machine/port, base, paramètres...
nom du *driver* est facultatif : tous sont essayés !
— interprétation des paramètres par le *driver* : pas de standard !

```
jdbc:odbc:vinsdepays
jdbc:odbc:paye;cachesize=20;extensioncase=LOWER
jdbc:z1MySQL://palo-alto.ensmp.fr/coelho
jdbc:mysql://palo-alto2.ensmp.fr/iar2m
jdbc:postgresql://sablons.ensmp.fr:5432/cinema
```



```
import java.sql.*;
public class SQL01 {
  static public final String
    URL = "jdbc:postgresql://sablons.ensmp.fr/cinema";
  static public void main(String[] args) {
    try {
      String pass = new java.util.Scanner(System.in).nextLine();
      Class.forName("org.postgresql.Driver");
      Connection c = DriverManager.getConnection
        (URL, "coelho", pass);
      // ...
      c.close();
    }
    catch (ClassNotFoundException e) {System.err.println(e);}
    catch (SQLException e) {System.err.println(e);}
} }
```


Exécution de SQL01

```
shell> java SQL01
j'entre un mot de passe au hasard
A connection error has occurred: FATAL 1:
   Password authentication failed for user "coelho"

shell> java SQL01
le bon mot de passe
```


La `Connection`

— il peut y en avoir plusieurs
— y compris vers la même base de données
— changement de base/catalogue, similaire à `cd`

```
String cat = c.getCatalog(); // courant
c.setCatalog("vinsdepays"); // changement
```

— examen des données : tables, structures, procédures...

```
DatabaseMetaData m = c.getMetaData();
```


Utilisation de `Connection`

— `Statement PreparedStatement CallableStatement`
plusieurs statements en parallèle...

```
// avec une connexion
Statement s = c.createStatement();
PreparedStatement ps = c.prepareStatement(requete_sql);
CallableStatement cs = c.prepareCall(requete_sql);

// utilisation de s, ps, cs...

// fermeture de la connexion
c.close();
```

— support des transactions, automatiques ou manuelles

```
c.setAutoCommit(false); // manuel
c.setTransactionIsolation(TRANSACTION_SERIALIZABLE);
// requêtes INSERT/UPDATE/DELETE...
c.commit(); // ok !         OU BIEN
c.rollback(); // pas ok :-(
```

— divers options : consultation des *warnings*, lecture seule...

```
SQLWarning w = c.getWarnings(); // donne le premier
c.clearWarnings();              // on efface !
```


Utilisation d'un `Statement`

— exécution de requêtes **en SQL** selon leur type

```
ResultSet r = s.executeQuery("SELECT ...");
int n = s.executeUpdate("UPDATE ...");
boolean set = s.execute("DROP ...");
```

— suite de commandes SQL *batch* (sauf select)

```
s.clearBatch(); s.addBatch(sql); s.executeBatch();
```

— **options** temps max, tailles max (lignes, attribut), direction d'énumération, échappements, taille des transferts...


```
// connexion
Class.forName("twz1.jdbc.mysql.jdbcMysqlDriver");
Connection c = DriverManager.getConnection(
   "jdbc:z1MySQL://palo-alto2/mysql", "coelho", "pass");

// requête
Statement s = c.createStatement();
s.setFetchSize(1000);
ResultSet r = s.executeQuery("SELECT * FROM user");

// exploitation du résultat...

// on range tout!
r.close();
s.close();
c.close();
```


Utilisation d'un `ResultSet`

— ensemble de lignes résultat, *en partie* sur le client...

— transfert géré par le *driver*

— passer à la ligne suivante : `r.next()`, vrai si ok

— extraction d'un attribut : selon **type** et **nom** ou **numéro**

Correspondances entre les types JAVA et SQL

conversion standard des types définie dans JDBC

souple **tout** peut être une `String`

extraction méthode `getT(...)` où `T` est le nom du type

Correspondances types SQL - Java

entiers java `boolean short int long`

SQL `BIT SMALLINT INTEGER BIGINT`

réels java `float double double BigDecimal`

SQL `REAL FLOAT DOUBLE DECIMAL NUMERIC`

chaînes java `String String AsciiStream`

SQL `CHAR VARCHAR LONGVARCHAR`

binaires java `Byte[] Byte[] BinaryStream`

SQL `BINARY VARBINARY LONGVARBINARY`

temps java `java.sql.Date java.sql.Time`

SQL `DATE TIME TIMESTAMP`

objet java sérialisés dans un attribut binaire ?

Informations sur un `ResultSet`

— à travers la classe `ResultSetMetaData`

— informations sur les colonnes : `m.getColumn...`

```
ResultSetMetaData m = r.getMetaData();
// nombre d'attributs
int nc = m.getColumnCount();
// nom d'un attribut
String n = m.getColumnName(1);
// type d'un attribut (voir java.sql.Types)
int ti = m.getColumnType(1);
String ts = m.getColumnTypeName(1);
```

— etc.


```
Class.forName("org.postgresql.Driver");
Connection c = DriverManager.getConnection(
  "jdbc:postgresql://sablons/cinema", "coelho", "pass");
Statement s = c.createStatement();
ResultSet r = s.executeQuery("SELECT * FROM films");
ResultSetMetaData m = r.getMetaData();
int ncols = m.getColumnCount();

while (r.next()) {
  for (int i=1; i<=ncols; i++)
  System.out.print(r.getString(i) + "\t");
  System.out.println();
}
r.close();
s.close();
c.close();
```


Exécution de `SQL03`

```
shell> java SQL03
mot de passe
1 Les lumieres de la ville 1925-01-01 2 1
2 La rose pourpre du Caire 1985-01-01 1 2
```

Optimisation du cas simple `SELECT ...` ?

- à partir du `DriverManager`
- création d'une `Connection`
- création d'un `Statement`
- exécution d'une requête `executeQuery(...)`
- obtention d'un `ResultSet`
- extraction, conversion des colonnes...
- informations complémentaires : `ResultSetMetaData`
- ouf !

Exécution d'un update

- requête SQL de type `UPDATE INSERT DELETE...`
- retourne le nombre de lignes modifiées
  `int n = s.executeUpdate(sql);`
- ou 0 si non approprié (`CREATE...`)

```
Class.forName("twz1.jdbc.mysql.jdbcMysqlDriver");
Connection c = DriverManager.getConnection(
  "jdbc:z1MySQL://palo-alto2/mysql", "coelho", "pass");
Statement s = c.createStatement();
int d = s.executeUpdate("DELETE FROM user WHERE name='root'");
System.out.println("nombre de lignes effacées : " + d);
s.close();
c.close();
```


Exécution d'une requête générale

- plusieurs résultats : `ResultSet` ou comptes...
- navigation parmis les résultats... si implémenté !

```
boolean res = s.execute(sql_general);
int count = 0;
while (res || count!=-1) {
  if (res) {
    ResultSet r = s.getResultSet();
    // ..
  } else
    count = s.getUpdateCount();
  res = s.getMoreResult(); // set or count
}
```


Lot de requêtes *(batch)*

- requête avec plusieurs commandes SQL (JDBC 2.0)
- commandes courantes : *efface, ajoute, exécute*
- retourne le comptage pour chaque commande
- pas de `ResultSet` par contre !

```
s.clearBatch();
s.addBatch("UPDATE ...");
s.addBatch("INSERT ...");
int[] r = s.executeBatch();
```


Avantage et inconvénient d'un `Statement`

souple toutes opérations, résultats multiples...

dynamique construction de requêtes : concaténation de chaînes de caractères...

optimisation *valeurs* utilisables

danger d'injection SQL

Utilisation d'un `PreparedStatement`

- classe similaire à `Statement` dont elle hérite
  `PreparedStatement ps = c.prepareStatement(...);`
- précise une requête SQL générique, avec jokers `?`
  `SELECT * FROM user WHERE name LIKE ? AND age>?`
- intérêts :

  précompilation mais optimisations selon valeurs...

  sécurité évite injections SQL, mais structure fixée
- valeurs des paramètres comme `ResultSet`

```
// cycle d'utilisation d'une requête préparée
ps.clearParameters();
ps.setString(1, "c%");
ps.setInt(2, 10);
ps.executeUpdate(); // ou .executeQuery() ou .execute()
```



```
Class.forName("twz1.jdbc.mysql.jdbcMysqlDriver");
Connection c = DriverManager.getConnection(
  URL, "coelho", "mon mot de pass");
PreparedStatement ps =
  c.prepareStatement("select * from user where name=?");
String name = in.nextLine();

ps.clearParameters();
ps.setString(1, name);
ResultSet r = ps.executeQuery();

while (r.next())
  System.out.println(r.getString(1) + " " +
                     r.getString(2) + " " +
                     r.getString(3));
r.close(); ps.close(); c.close();
```


Exécution de `SQL05`

```
shell> java SQL05
root
localhost root 566f3e6d598e0c9b
palo-alto root 566f3e6d598e0c9b
palo-alto.ensmp.fr root 566f3e6d598e0c9b
```

Les `CallableStatement`

— appel une procédure stockée *stored procedure*
— hérite de `PreparedStatement`

```
CallableStatement cs =
  c.prepareCall("? = call maProcedure[?,?]");
```

— entrées : comme `PreparedStatement`
— sorties : enregistrement préalable des types

```
cs.registerOutParameter(1, Types.VARCHAR);
```

extraction avec méthode `getT(no)` où `T` est un type

Classe `DatabaseMetaData`

— informations sur la base de données
— retourné par `c.getMetaData()` de `Connection`
— liste des tables : `getCatalogs()`
— quelques centaines de routines d'interrogation...

Classe `Types`

— ensemble de constantes pour décrire les types SQL
— `BIT FLOAT INTEGER TIME DECIMAL VARCHAR...`
— utilisé pour identifier les types : `registerOutParameter(...)`
— et pour l'examen de tables...

Conseils sur l'utilisation de JDBC

— encapsuler (cacher) tout dans une classe spécifique
requêtes diverses
fonctions utilitaires (resultat vers HTML ? vers AWT ?)
gestion des erreurs ? pas forcément...
— `Statement` dangereux, préférer `PreparedStatement` !
— prévoir l'utilisation par des *threads* : synchronisation !
— tester dans un contexte simple *main*
plus difficile avec Servlet/JSP/EJB

Modèle d'utilisation

— réalisation d'applications client-serveur
— serveur lourd (DBMS) stocke les données
sécurité, standard...
— client léger (JAVA) propose une interface
portabilité, apparences...
  — orienté utilisateur, présentation
  — composition de plusieurs services


```
import java.sql.*;
import java.util.*;
/** acces a la base Cinema... */
public class CinemaDB
{
  // ATTRIBUTS
  protected Connection dbc;
  protected Statement sta;

  // CONSTRUCTOR
  public CinemaDB(Connection dbc)
    throws SQLException
  {
    this.dbc = dbc;
    this.sta = dbc.createStatement();
  }
```



```
// METHODS
protected synchronized String[] getList(String sql)
  throws SQLException {
  List l = new ArrayList();
  ResultSet r = sta.executeQuery(sql);
  while (r.next())
    l.add(r.getString(1));
  sta.close(); // ResultSet r is also closed here
  return (String[]) l.toArray(new String[l.size()]);
}
public String[] getGenres() throws SQLException {
  return getList("SELECT nom FROM genres ORDER BY nom");
}
public String[] getTitres() throws SQLException {
  return getList("SELECT titre FROM films ORDER BY titre");
}
```



```
  // TESTS !
  static public void main(String[] args) throws Exception
  {
    Scanner in = new Scanner(System.in);
    System.out.print("login: "); String login = in.nextLine();
    System.out.print("pass: "); String pass = in.nextLine();
    Class.forName("org.postgresql.Driver");
    Connection dbc = DriverManager.getConnection(
        "jdbc:postgresql://sablons/cinema", login, pass);
    CinemaDB cine = new CinemaDB(dbc);
    String[] genres = cine.getGenres();
    String[] titres = cine.getTitres();
    System.out.println(genres[2]);
} }
```


Pooling : mutualisation des coûts

— partage de la connexion (longue et couteuse à établir ?)
une seule suffit ! important pour certaines licences de DB
— partage des `Statement` entre *threads* ?
contexte Servlet/JSP/EJB
— classe `DataSource` + JNDI...
— serveur `pgpool` externe qui cache les connexions
implémente la séquence d'authentification

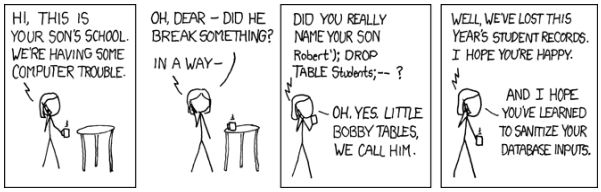
Injection de SQL

— une petit démonstration plutôt qu'un long discours


```
; DROP TABLE "COMPANIES";-- LTD
```

— Samuel Thomas PIZZEY
— Company number 10542519
— depuis le 29 décembre 2016 au Royaume-Uni
— *62020 - Information technology consultancy activities*

Insecure Lab, India



XKCD `http://xkcd.com/327/` *(CC BY-NC 2.5)*

Outil SQLMap `http://sqlmap.org/`

— détection et exploitation automatique d'injections SQL
— contrôle des requêtes
HTTP/HTTPS, GET/POST, Cookie, Referer, User-Agent...
— cibles : Oracle, MySQL, PgSQL, MS SQL Server, IBM DB2...
— extraction des informations roles, tables...
— recherche de mots de passe
— duplication locale (via SQLite) de la base de données

Conclusion

— connexion JAVA-DBMS via JDBC
— conforme aux standards SQL, proche de ODBC
— modèle bien construit `Connection Statement ResultSet...`
— autre ? **XML Torque** et générations automatique de classes !
ORM *Object-Relational Mapping*
se méfier : contraintes ? typage ? fait le design de la base...

List of Slides