Le SQL intégré

Fred Hémery

IUT Béthune
Département
Réseaux & Télécommunications

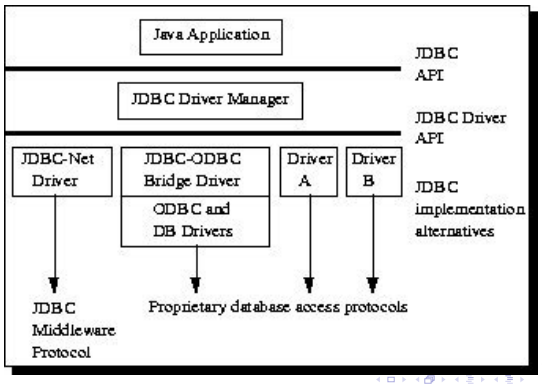
Base de Données (IC5) — 07/08

Le SQL intégré

- Aujourd'hui les applications sont de plus en plus complexe. Un SGBD associé au langage SQL ne peut pas répondre à toutes les questions des utilisateurs.
- La solution adoptée consiste à intégrer des fonctions de liaison dans les langages de programmation classiques pour leurs permettre d'accéder aux données stockées dans un SGBD.
- Il existe des bibliothèques de fonctions ou classes pour les langages
  - C
  - C++
  - JAVA
  - Tcl/Tk
  - Perl ...

Présentation de l'API JAVA : JDBC

- JDBC : *Java Data Base Connection*
- Architecture



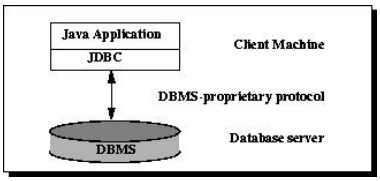
Présentation de l'API JAVA : JDBC

- L'utilisation de l'API JDBC se fait en 4 étapes :
  1. chargement du pilote (driver) JDBC qui fait le lien entre l'application Java et le SGBD
  2. connexion à une base de données du SGBD
  3. utilisation et traitement des données
  4. fermeture de la connexion
- Les classes sont déclarées dans les paquetages
  - java.sql
  - javax.sql

Chargement du pilote

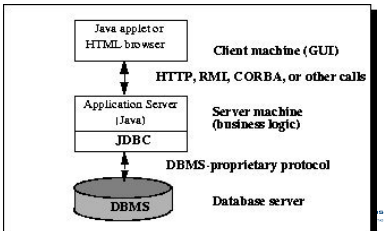
- Il y a quatre types de pilote
  1. La passerelle JDBC-ODBC (Open DataBase Connectivity) qui permet de transformer l'API JDBC vers l'API ODBC qui est très largement utilisée dans le monde Microsoft.
  2. Le pilote qui implémente JDBC en natif dans un langage autre que java, c'est à dire qu'il accède directement au SGBD cible. Il doit exister une version du pilote par type d'architecture qui sera susceptible d'accueillir l'application cliente.
  3. Le pilote JDBC transporte la requête jusqu'à un serveur d'applications (middleware) de manière indépendante du SGBD cible. Ensuite le serveur fournit la requête au SGBD.
  4. Le pilote qui transmet la requête directement vers le SGBD cible, mais le pilote est implémenté en Java.

Chargement du pilote



Architecture 2 tiers

Architecture 3 tiers



Chargement du pilote

- Concrètement pour charger le pilote on tente de créer une instance de la classe qu'il représente en utilisant l'introspection avec l'instruction suivante :

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

Recherche la classe `org.postgresql.Driver` dans la liste des chemins définis dans la variable d'environnement `CLASSPATH` et tente de l'instancier.

Connexion à la Base de données

- Cette étape consiste à utiliser le pilote pour faire la connexion entre l'application et le SGBD cible. Pour cela on utilise l'instruction suivante :

```
Connection connexion = DriverManager.getConnection(
url,
nomLogin,  // identification de l'utilisateur
motPasse); // authentification de l'utilisateur
```

- La variable `url` est de la forme :

```
jdbc:postgresql://machineServeur/nomBD
jdbc:odbc:nomBD
...
```

- La connexion obtenue sera utilisée pour travailler avec le SGBD cible.

Création d'une requête

- Il y a deux types de requête
  - les requêtes d'interrogation, consultation qui renvoient un résultat ; et
  - les requêtes de modification
- Les requêtes de consultation comme les requêtes de modification seront véhiculées par une instance de la classe `Statement`.

```
Statement stmt = connexion.createStatement()
```

Une requête est créée par la méthode `createStatement()` de la classe `Connection` . Il n'y a qu'une seule instance de la classe `Statement` par instance de la classe `Connection`.

Création d'une requête

- On utilise l'instance de la classe Statement pour indiquer le code que l'on veut exécuter.
- Suivant le type de requête :
  - Modification :

```
stmt.executeUpdate("CREATE TABLE agenda" +
                   "(nom varchar(20), prenom varchar(20
                   "adresse varchar(50), tel char(12)")
```

  - Consultation

```
stmt.executeQuery("SELECT * FROM agenda");
```

Contrairement aux requêtes de modification, les requêtes de consultation ont besoin d'accéder au résultat de la requête.

Traitement d'une requête

- Le résultat de la requête est affecté à une instance de la classe `ResultSet`.

```
ResultSet rs = stmt.executeQuery("Select * from client;");
```

- L'instance de la classe `ResultSet` contient la table résultat de la requête.
- Pour récupérer les données dans l'instance, on utilise un pointeur sur une ligne courante et les méthodes du tableau.
- Voir la documentation Java pour avoir l'ensemble des méthodes disponibles.

`int getInt(int i)`	`int getInt(String nomCol)`	Renvoie l'entier stocké dans la ième colonne ou la colonne de nom nomCol
`String getString(int i)`	`String getString(String nomCol)`	Renvoie la chaîne de caractères stockée dans la ième colonne ou la colonne de nom nomCol
`Date getDate(int i)`	`Date getDate(String nomCol)`	Renvoie la date stockée dans la ième colonne ou la colonne de nom nomCol
`boolean next()`		Renvoie vrai si il y a encore une ligne à traiter, faux sinon. Elle fait passer aussi d'une ligne à la ligne suivante
`boolean first()`		Place le curseur sur la première ligne

Clôture de la connexion

On utilise la méthode `close()` de la classe `Connection`.

```
connexion.close();
```


Un exemple complet

```
import java.io.*;
import java.sql.*;
public class TestJDBC {
  private Connection connexion;
  public TestJDBC() {
    try {
      Class.forName("org.postgresql.Driver");
    } catch (ClassNotFoundException e) {
      System.err.println("Pilote d'acces
      postgresql introuvable (" + e.toString()       + ")");
    }
  }
public static void main (String[] args) {
    TestJDBC instance = new TestJDBC();
    instance.ouvreConnexion("jdbc:postgresql://localhost/agenda",
      "duchmol", "secret");
    instance.traitement();
    instance.fermeConnexion();
  }
}
```


Un exemple complet

```
public void ouvreConnexion (String url, String user, String password) {
    try {
      connexion = DriverManager.getConnection(url, user, password);
      System.out.println("Connecte a la base de donnees URL = " + url);
    } catch (SQLException e) {
      System.out.println("Exception: ouverture (" + e.toString() + ")");
    }
  }

  public void fermeConnexion () {
    try {
      connexion.close();
    } catch (SQLException e) {
      System.out.println("Exception: fermeture (" + e.toString() + ")");
    }
  }
```


Un exemple complet

```
public void traitement() {
  Statement stmt = null;
  ResultSet rs;

   try {
     stmt = connexion.createStatement();
     stmt.executeUpdate("drop table telephone");
   } catch (SQLException e ) {}
   try {
     stmt.executeUpdate("create table telephone (" +
"nom varchar(25) NOT NULL," +
"prenom varchar(25) NOT NULL," +
"telephone varchar(15) NOT NULL)");
     stmt.executeUpdate("insert into telephone " +
"values ('Martin', 'Robert', '00 00 00 00 00')");
     stmt.executeUpdate("insert into telephone " +
"values ('Durant', 'Gerard', '00 00 00 00 00')");

     rs = stmt.executeQuery ("select * from telephone order by nom, prenom");

     while (rs.next()) {
       System.out.println("Nom :\t" + rs.getString("nom"));
       System.out.println("Prenom :\t" + rs.getString(2));
       System.out.println("telephone :\t" + rs.getString("telephone") + "\n");
     }
     stmt.close();
   } catch (SQLException e ) {
     System.err.println("erreur execution requete SQL " + e.toString());
     System.exit(1);
   }
 }
```


Résultat d'exécution

```
[...] javac TestJDBC.java

[...] Java -classpath .:/usr/share/lib/pgsql/jdbc7.1-1.2.jar TestJDBC
Connecte a la base de donnees URL = jdbc:postgresql://localhost/duchmol
Nom : Durant
Prenom : Gerard
telephone : 00 00 00 00 00

Nom : Martin
Prenom : Robert
telephone : 00 00 00 00 00
```

Requêtes particulières

- Lorsque vous avez à exécuter plusieurs fois la même requête, il est intéressant de pouvoir la compiler au préalable pour que le SGBD n'ait plus qu'à l'exécuter.
- Dans ce cas on utilise la classe `PreparedStatement`.

```
String sql = "update telephone set telephone = ?
              where nom = ? ";
PreparedStatement changeTel =
                  connexion.preparedStatement(sql);
```

- On remplace, à l'utilisation de la requête préparée, les " ?" par des valeurs actualisées

```
changeTel.setString(1,"03 21 63 71 65");
changeTel.setString(2,"martin");
int i = changeTel.executeUpdate();
```

La méthode `executeUpdate()` renvoie le nombre de tuples ayant été modifiés

Les méta-informations

- `JDBC` dispose aussi de deux classes
  - `ResultSetMetaData` pour obtenir des informations sur une instance de la classe `ResulSet`
  - `DatabaseMetaData` pour obtenir des informations sur la base de données
- En effet `ResultSetMetaData` permet d'obtenir
  - Le nombre de colonnes contenues dans le `ResultSet(getColumnCount())`
  - Le nom d'une colonne (`getColumnLabel(int i)`)
  - Le nom de la table d'une colonne (`getTableName()`)
  - Le type de donnée d'une colonne (`getColumnTypeName()`)
  - La taille en nombre de caractères d'une colonne (`getColumnDisplaySize()`)

Les méta-informations

```
try {
      stmt = connexion.createStatement();

      rs = stmt.executeQuery ("select * from telephone order by nom, prenom");
      ResultSetMetaData meta = rs.getMetaData();
      int nbColonnes = meta.getColumnCount();
      while (rs.next()) {
        for (int i = 1; i <= nbColonnes; i++)
          System.out.println(meta.getColumnName(i) + " :\t" +
                             meta.getColumnTypeName());
      }
      stmt.close();
    } catch (SQLException e ) {
      System.err.println("erreur execution requete SQL " + e.toString());
      System.exit(1);
    }
```


Les méta-informations

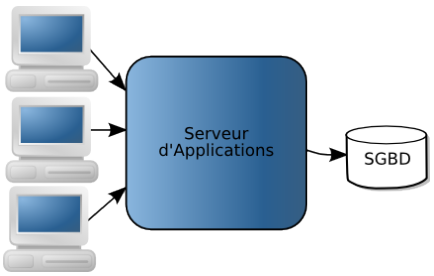
```
try {
      DatabaseMetaData meta = connexion.getMetaData();

      System.out.println("SGBD :\t" + meta.getDatabaseProductName());
      System.out.println("JDBC Driver :\t" + meta.getDriverName());
      System.out.println("Version :\t" + meta.getDriverVersion());

    } catch (SQLException e ) {
      System.err.println("erreur  " + e.toString());
      System.exit(1);
    }
```


Utilisation d'un pool de connexions

- Dans le cas d'une exécution d'un client qui accède au SGBD directement, ce qui vient d'être présenté reste valide ; cependant
- Dans le cas d'une utilisation d'un serveur d'applications dans une architecture 3 tiers, il peut être utilise d'apporter des améliorations.



Utilisation d'un pool de connexions

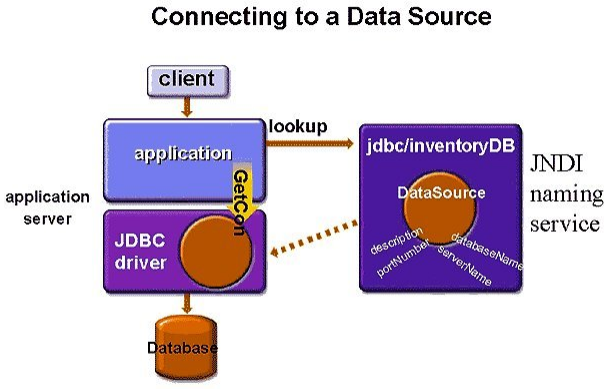
Le serveur d'application gère un ensemble de processus

- Certains de ces processus ont besoin d'une connexion avec la base de données
- Plutôt que de créer une connexion avec le SGBD à chaque utilisation, une solution plus efficace est de gérer un pool de connexions disponibles mais pas affectées.
- Lorsqu'un processus aura besoin d'une connexion, il demandera au gestionnaire du pool de connexion, si il dispose d'une connexion libre. Dans le cas positif, le gestionnaire affectera une connexion au processus. Sinon il bloque le processus en attente d'une connexion.
- Lorsque le processus aura terminé sa transaction avec le SGBD, il la rendra au gestionnaire, qui l'ajoutera au pool des connexions disponibles

Utilisation d'un pool de connexions

- Pour obtenir une connexion disponible dans un pool, il faut utiliser l'interface : `DataSource` qui est définie dans la paquetage `javax.sql`.
- Pour pouvoir utiliser cette interface il faut utiliser une version de `JDBC 2.0` ou `JDBC 3.0`
- Avec `postgreSQL` on dispose d'une implémentation de `JDBC3.0` (c.a.d `java.sql.*`, `javax.sql.*`)
- Le programme client de base qui veut utiliser une connexion de type `DataSource` au travers d'un pool de connexions disponibles va effectuer deux étapes :
  1. Création d'un pool de connexion (étape effectuée par le serveur d'applications, 1 fois à l'initialisation) ;
  2. Obtention d'une connexion dans le pool créé (par le client, à chaque demande de connexion)

Les extensions



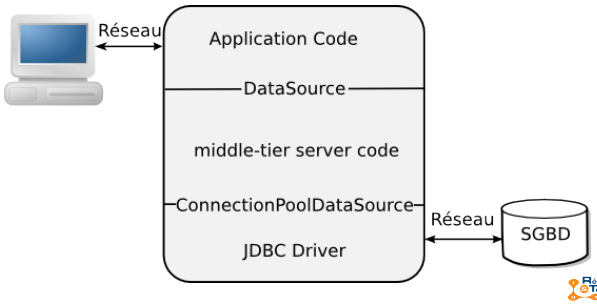
Utilisation d'une DataSource coté client

```
Connection conn = null;
try {
    conn = source.getConnection();
    // use connection
} catch (SQLException e) {
    // log error
} finally {
    if (con != null) {
        try { conn.close(); } catch (SQLException e) {}
    }
}
```

La variable `source` est de type `DataSource`, elle aura été mis à disposition par le serveur d'applications.

DataSource & JNDI

La source de données (`DataSource`) peut avoir été créée par un serveur d'application et ensuite enregistrée dans un annuaire (*repository*). Les ressources disponibles dans cet annuaire sont alors accessibles à partir d'un client qui connaît la référence de la source de données.



Utilisation d'une DataSource & JNDI coté client

```
Connection conn = null;
try {
    DataSource source = (DataSource)new InitialContext().lookup("DataSource");
    conn = source.getConnection();
    // use connection
} catch (SQLException e) {
    // log error
} catch (NamingException e) {
    // DataSource wasn't found in JNDI
} finally {
    if (con != null) {
        try { conn.close(); } catch (SQLException e) {}
    }
}
```

La variable `source` de type `DataSource` est obtenue suite à l'interrogation d'un annuaire (JNDI) qui stocke des références sur des ressources (ici une source de données).

Initialisation d'une DataSource coté serveur : exemple de tomcat

- Il faut ajouter le pilote JDBC dans l'environnement de tomcat (`$CATALINA_HOME/common/lib`)
- Il faut créer un fichier qui définit un contexte particulier pour votre application. Si par exemple votre application a pour nom "`MagasinVirtuel`", il faut créer un fichier `context.xml` qui contient :

```
<Context path="/MagasinVirtuel" docBase="."
   crossContext="true" reloadable="true" debug="1">

<Resource name="jdbc/postgres" auth="Container"
      type="javax.sql.DataSource" driverClassName="org.postgresql.Driver"
      url="jdbc:postgresql://127.0.0.1:5432/mydb"
      username="myuser" password="mypasswd" maxActive="20" maxIdle="10"
maxWait="-1"/>
</Context>
```


Initialisation d'une DataSource coté serveur : exemple de tomcat

- Dans le fichier web.xml il faut ajouter la balise suivante :

```
<resource-ref>
 <description>postgreSQL Datasource example</description>
 <res-ref-name>jdbc/postgres</res-ref-name>
 <res-type>javax.sql.DataSource</res-type>
 <res-auth>Container</res-auth>
</resource-ref>
```

- Enfin pour créer la variable source de données :

```
InitialContext cxt = new InitialContext();
if ( cxt == null ) {
   throw new Exception("Uh oh — no context!");
}

DataSource ds = (DataSource) cxt.lookup( "java:/comp/env/jdbc/postgres" );

if ( ds == null ) {
   throw new Exception("Data source not found!");
}
```