Chapitre 6

Les bases de données

6.1 Préparation

6.1.1 Choix et installation de la base de données

L'utilisation d'une base de données avec Java présuppose l'existence d'un système déjà installé. Java peut s'interfacer avec pratiquement tous les systèmes courants tels mySQL, SQL-Server, Oracle, InterBase, MSAccess. Il peut aussi utiliser des bases de données plus légères comme Derby, HSQL ou SQLite. Pour des applications destinées à des environnements dans lesquels les utilisateurs n'ont pas les droits ou les compétences pour installer et gérer un serveur lourd, les petites bases que je viens de mentionner sont parfois très intéressantes. Elles permettent de gérer des données dans des fichiers simples, voire même en mémoire centrale. Parmi tous ces systèmes, stratégie commerciale oblige, Sun donne la préférence à Derby, qu'il a développé, et à mySQL qu'il a racheté récemment.

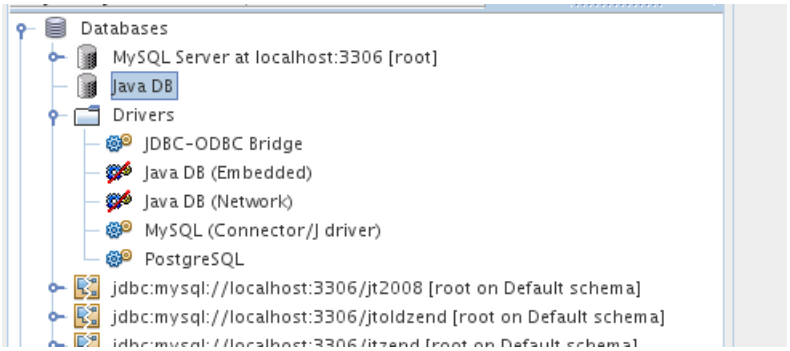
Afin de simplifier les choses, Java propose une interface commune pour la gestion de toutes les bases de données. Inspirée par le modèle ODBC, le standard JDBC permet d'utiliser un grand nombre de fonctionnalités des bases de données. Dans les rares cas où un pilote JDBC ne serait pas disponible, on peut se rabattre sur un pilote ODBC, mais au prix d'une perte de fonctionnalités et de performances. Chaque éditeur fournit un pilote JDBC adapté à son serveur.

Je préconise personnellement l'emploi de mySQL, parce que l'utilisation d'un autre système présuppose un travail d'installation plus conséquent. Oracle dispose de pilotes destinés à Java, mais ils ne sont pas toujours faciles d'accès. Quant à Derby ou SQLite, ils ne disposent pas d'outils de gestion commodes, ce qui peut parfois poser problème en phase de mise au point. Tout est relatif et ces inconvénients ne doivent pas peser sur le choix du système dans le contexte d'une application professionnelle.

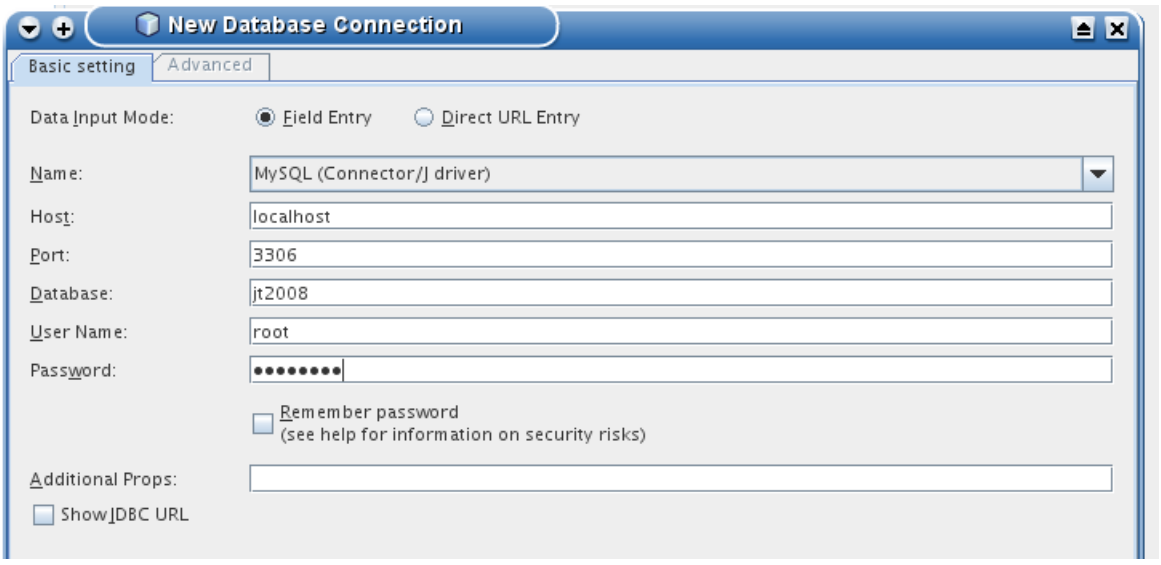
Il reste une remarque à propos de mySQL. Lors de l'installation du serveur, certaines contraintes de sécurité peuvent poser problème dans une exploitation avec Java. C'est le cas notamment du serveur intégré Xampp qui propose une stratégie de sécurité trop sévère pour l'utilisation de NetBeans. Assez curieusement, même dans le cas d'une exploitation sur une seule machine, la directive `skip-networking` interdisant la connexion depuis une autre machine doit être désactivée. Il suffit de placer un '`#`' devant dans le fichier *etc/my.conf*. Ce petit détail, qui a changé depuis la version 6.0 de NetBeans, m'a causé quelques cheveux blancs au moment de la migration.

6.1.2 Installation du pilote

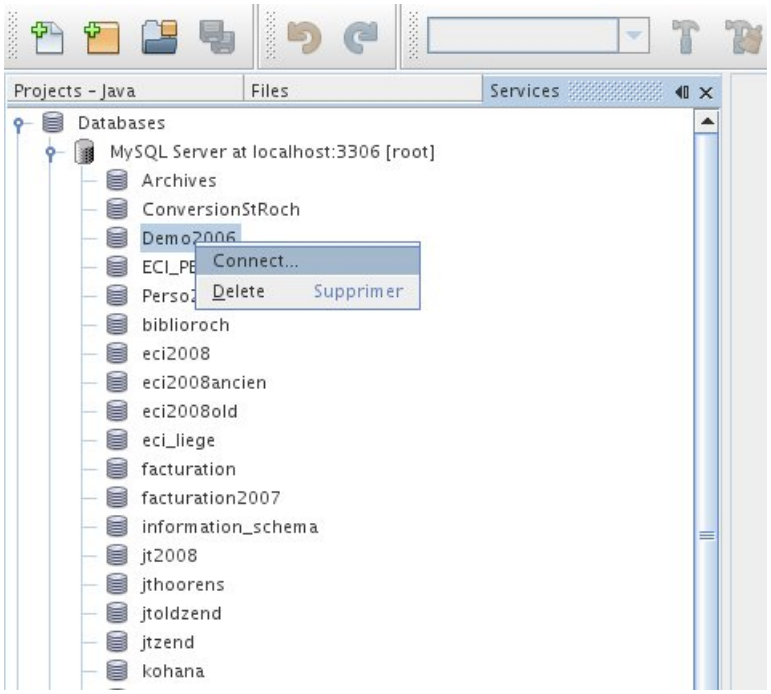
Depuis le rachat de mySQL par Sun, les pilotes JDBC pour mySQL sont directement disponibles dans NetBeans (fichier *netbeans-6.5/ide10/modules/ext/mysql-connector-java-5.1.6-bin.jar*). On peut s'assurer que tout marche bien en allant dans l'onglet Services, sous la rubrique Databases. On y trouve normalement plusieurs pilotes.



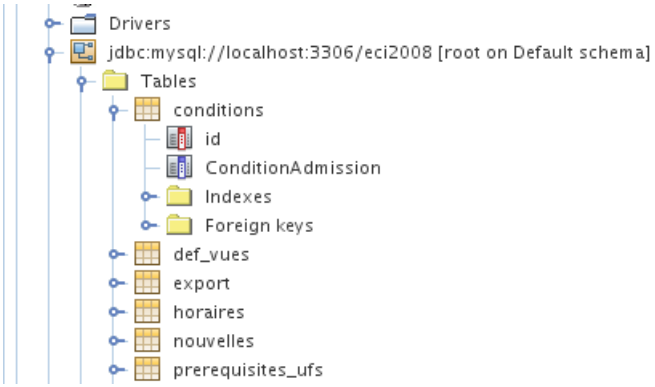
Il suffit ensuite de créer une nouvelle connexion, de sélectionner le pilote et de donner les informations de connexion.



Un seul utilisateur peut être utilisé par serveur, mais on peut ouvrir simultanément plusieurs bases de données.



Une fois connecté, on dispose d'une visualisation des tables et des champs et de la possibilité de modifier la structure de la base et de lancer des requêtes DML Cette intégration de la base dans l'IDE est particulièrement confortable.



6.2 Exploitation d'une base de données

6.2.1 Connexion

Il y a quatre paramètres à définir : le pilote, l'URL (comprenant la base), l'utilisateur et son mot de passe

```
private final String DB_DRIVER = "com.mysql.jdbc.Driver";
private final String DB_URL = "jdbc:mysql://127.0.0.1:3306/messagerie";
private final String DB_USER ="messager";
private final String DB_PASSWD = "codd";
```

L'object fondamental est la `Connection` défini dans *java.sql*.

```
import java.sql.*;
public class .....
{
...
    private Connection connection=null;
```

L'une des méthodes pour se connecter consiste à utiliser un constructeur reprenant les trois derniers paramètres, on notera la gestion obligatoire des exceptions. Le fonctionnement de ce code semble assez obscur. Le choix du pilote se fait par le chargement de sa classe (appel de la méthode statique `forName()` de la classe `Class`). Le chargement de cette classe exécute une méthode statique qui range la classe parmi les drivers JDBC enregistrés. La méthode statique `getConnection()` de la classe `DriverManager` se charge alors de rechercher le driver enregistré et de créer une connexion avec les paramètres nécessaires. Notons que cette connexion fait partie d'un paquetage défini par Sun et que toutes les manipulations de la base de données se font à travers lui. Le choix de la base de données se limite donc à charger le driver voulu au bon moment, dans la suite, le code sera le même pour toute base de données intefacée à l'aide de JDBC.

```
private void retablirDBConnexion()
{
if(connection==null)
{  try{
   // choix du pilote
   Class.forName(DB_DRIVER);
   // Établissement de la connexion
       try{
        connection = DriverManager.getConnection(DB_URL,DB_USER,DB_PASSWD);
          }
       catch(SQLException ex)
          { erreurFatale(1,"Connexion à la base impossible");}
   }
   catch(ClassNotFoundException ex)
      {erreurFatale(2, "Pilote de base de données non trouvé");
      }
}
}
```

6.2.2 Envoi et exploitation d'une requête `SELECT`

L'exploitation d'une requête consiste en fait à obtenir un objet de type `ResultSet` au départ d'une connexion. Cela passe par la création d'un `Statement` dont on activera la méthode `executeQuery()`. Le résultat obtenu est un curseur que l'on peut parcourir à l'aide de la méthode `next()`. Chaque appel de cette méthode fait avancer le curseur. A chaque étape, une série de méthodes `getXXX()` permettent de récupérer des données d'un type donné en passant le nom du champ en paramètre. Dans l'exemple suivant, on utilise une vue qui reprend un utilisateur et ses contacts dans une messagerie.

```
public ArrayList<Utilisateur> getListeContacts(String strNom)
{
  Utilisateur contact;
```

```
  ArrayList<Utilisateur> utilisateurs = new ArrayList<Utilisateur>();
  retablirDBConnexion();
  String strSQL=
      "SELECT␣contactName,connecte␣"+
      "FROM␣VueContacts␣"
      "WHERE␣username='" + strNom + "';";
  try{
      Statement statement = connection.createStatement();
      ResultSet results = statement.executeQuery(strSQL);
      while (results.next())
        {
          contact = new Utilisateur(
             results.getString("U2.username"),
             results.getBoolean("U2.connecte"));
          utilisateurs.add(contact);
        }
      return utilisateurs;
     }
  catch(Exception ex)
    {
      ex.printStackTrace();
      return null;
    }
}
```

6.2.3 Modification d'une donnée

Pour modifier les données, on passera aussi par la création d'un `Statement`. La méthode `executeUpdate()` prendra en paramètre une requête `UPDATE`, `DELETE` ou `INSERT`. Elle renvoie le nombre de lignes modifiées.

```
private boolean modifierDonnees(
      String strTable,
      String strChamp, String strValeur,
      String strChampCond, String strValCond)
{
      ...
      String strSQL =
                "UPDATE␣"+ strTable + "␣" +
                "SET␣" + strChamp + "=" + strValeur + "␣" +
                "WHERE␣"+ strChampCond + "=" + strValCond+";";
try{
      Statement statement = connection.createStatement();
      int iNum = statement.executeUpdate(strSQL);
      return (iNum==1);
     }
  catch(SQLException ex)
    {
       System.err.println(strSQL);
       return false;
    }
}
```

6.3 Exemple de routines génériques (compatibles avec Oracle et mySQL)

L'exemple suivant procure une classe permettant l'utilisation d'une base de données avec les paramètres habituels. Lors de l'instanciation d'un objet, on spécifie le type de serveur (à l'aide d'une constante énumérée). On dispose de cinq méthodes :

- `retablirDBConnection()` rétablit si nécessaire une connexion sur la base de données ;
- `closeDBConnection()` ferme la connexion ;
- `lireDonneUnique()` permet de récupérer sous forme de String une donnnée sur base du nom de la table, du nom du champ, du nom de la clé primaire et d'une valeur[1].
- `modifierDonneeUnique()` réalise une modification d'une donnée dans une table.
- `lirePaire()` renvoie une collection de deux valeurs (sous forme d'un tableau de deux chaînes), associant une valeur de clé primaire et la valeur d'un champ spécifié pour la clé correspondante. Cette collection peut facilement être utilisée dans une liste déroulante afin de réaliser un choix.

Cet exemple a été testé avec des champs numériques et des nombres.

```
/*
 * DataBase.java
 *
 */
package database;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
/**
 *
 * @author jacques@thoorens.net
 */
public class DataBase {

private final int BASE_ERR = 1000;

// caractéristiques mySQL
private final String MYSQL_DB_DRIVER="com.mysql.jdbc.Driver";
private final String MYSQL_DB_URL="jdbcmysql//127.0.0.13306/";

// caractéristiques Oracle
private final String ORACLE_DB_DRIVER="oracle.jdbc.driver.OracleDriver";
private final String ORACLE_DB_URL="jdbcoraclethin@localhost1521";

private String baseName;
private String userName;
private String notPasse;
```

[1] On pourrait déterminer automatiquement le nom de la clé primaire par introspection, mais cela dépasse le cadre de cette petite introduction.

```
    private ServeurType typeServeur;
    private Connection connexion;
    private String baseURL;

    /** Creates a new instance of DataBase */
    public DataBase(ServeurType typeServeur,String baseName,
        String userName, String motPasse) {
        this.typeServeur = typeServeur;
        this.baseName = baseName;
        this.userName = userName;
        this.notPasse = motPasse;
        switch(this.typeServeur){
            case SERVEUR_MYSQL:
                baseURL=MYSQL_DB_URL;
                break;
            case SERVEUR_ORACLE:
                baseURL=ORACLE_DB_URL;
                break;
        }
    }


    public void retablirDBConnexion(){
        if(connexion==null) {
            try{
                // choix du pilote
                switch (typeServeur) {
                    case SERVEUR_MYSQL:
                        Class.forName(MYSQL_DB_DRIVER);
                        break;
                    case SERVEUR_ORACLE:
                        Class.forName(ORACLE_DB_DRIVER);
                        break;
                }
                // Etablissement de la connexion
                try{

                    connexion = DriverManager.getConnection(
                         baseURL+getBaseName(),getUserName(),getNotPasse());
                } catch(SQLException ex) {
                    erreurFatale(1,"Connexion␣à ␣la␣base␣impossible");
                }
            } catch(ClassNotFoundException ex) {
                erreurFatale  (2, "Pilote␣de␣base␣de␣données␣non␣trouvé");
            }
         }
        }

    public Object lireDonneeUnique
       (String strTable,String strColonne,String strCle,String strValeur) {
        Object donneeLue=null;
        Statement stm;
        ResultSet rst;
```

```
        String strQuery =
                "SELECT␣"+ strColonne +
                "␣FROM␣" + strTable +
                "␣WHERE␣" + strCle +
                "␣='" + strValeur + "'";
        retablirDBConnexion();
           try{
                stm = connexion.createStatement();
                rst = stm.executeQuery(strQuery);
                if(rst.next())
                    donneeLue = rst.getObject(strColonne);
                else
                    donneeLue = null;
                // il est conseillé de libérer les ressources
                rst.close();
                stm.close();
            } catch(SQLException ex) {
                System.err.println(strQuery);
                erreurFatale  (3, "Reqyête␣incorrecte");
            } finally {
                return donneeLue;
            }
        }

    public Collection<String[]> lirePaires
      (String strTable,String strColonne,String strCle,String condition) {
        Collection<String[]> paires = new ArrayList<String[]>();
        Statement stm;
        ResultSet rst;
        StringBuilder query= new StringBuilder(
                "SELECT␣" + strCle + "," + strColonne +
                "␣FROM␣" + strTable);
        if(condition.length()>0)
            query.append("␣WHERE␣"+condition);
        retablirDBConnexion();
        try{
            stm = connexion.createStatement();
            rst = stm.executeQuery(query.toString());
            while(rst.next()){
                String[] paire = new String[2];
                paire[0] = rst.getString(strCle);
                paire[1] = rst.getString(strColonne);
                paires.add(paire);
            }
            // il est conseillé de libérer les ressources
            rst.close();
            stm.close();
        } catch(SQLException ex) {
                System.err.println(query.toString());
                erreurFatale  (3, "Reqyête␣incorrecte");
            } finally {
                return paires;
            }
```

```
}

public int modifierDonneeUnique(String strTable,
        String strColonne,String strCle,String strValeur,String strNew){
    Statement stm;
    int ligne=0;
    StringBuilder query = new StringBuilder(
            "UPDATE " + strTable +
            " SET " + strColonne +
            "='" + strNew + "'" +
            " WHERE " + strCle);
    query.append(" ='" + strValeur+"'");
    retablirDBConnexion();
    try{
        stm = connexion.createStatement();
        ligne=stm.executeUpdate(query.toString());
        // il est conseillé de libérer les ressources
        stm.close();
    } catch(SQLException ex) {
        System.err.println(query);
        erreurFatale  (3, "Requête incorrecte");
    } finally{
        return ligne;
    }
}

private void erreurFatale  (int iErreur,String strMessage) {
    iErreur+=BASE_ERR;
    System.err.println(iErreur+" - "+strMessage);
    System.exit(iErreur);
}

public void setBaseURL(final String baseURL) {
    this.baseURL = baseURL;
}
}
```

6.4 Optimisation et `PreparedStatement`

Java fournit une interface qui autorise l'optimisation des ressources, nommée `PreparedStatement`. C'est une variante enrichie des `Statements` vus plus haut. On crée un tel objet en lui affectant une chaîne de caractère correspondant à une requête et en y laissant des emplacements vides à venir remplir par la suite. Ces emplacements sont marqués par `?`. La création de cet objet prépare en fait l'accès à la base de données. Il suffit ensuite de spécifier les valeurs effectives pour pouvoir ensuite effectuer la requête. On dispose pour cela de méthodes nombreuses `setXXX()`, pour chaque type possible, qui précisent une valeur donnée à un paramètre identifié par sa position.

L'exemple qui suit utilise une table comprenant des menus pour un site Web. Chaque item de menu possède entre autres un *Intitule* et une *Position*. On va afficher les intitulés correspondant aux positions 1000, 2000, 3000 et 4000, qui sont supposées exister.

```
try {
    ResultSet rst;
    retablirDBConnexion();
    PreparedStatement prep =  connexion.prepareStatement(
       "SELECT * FROM jt_menus WHERE Position = ?");
    for(int i=1;i<5;i++){
        prep.setInt(1, i*1000);
        rst = prep.executeQuery();
        if(rst.next())
            System.out.println(rst.getString("Intitule"));
        }
    } catch (SQLException ex) {
            System.out.println("Erreur dans la lecture");
    }
```

Cette technique peut fonctionner aussi bien pour des lectures que des modifications.