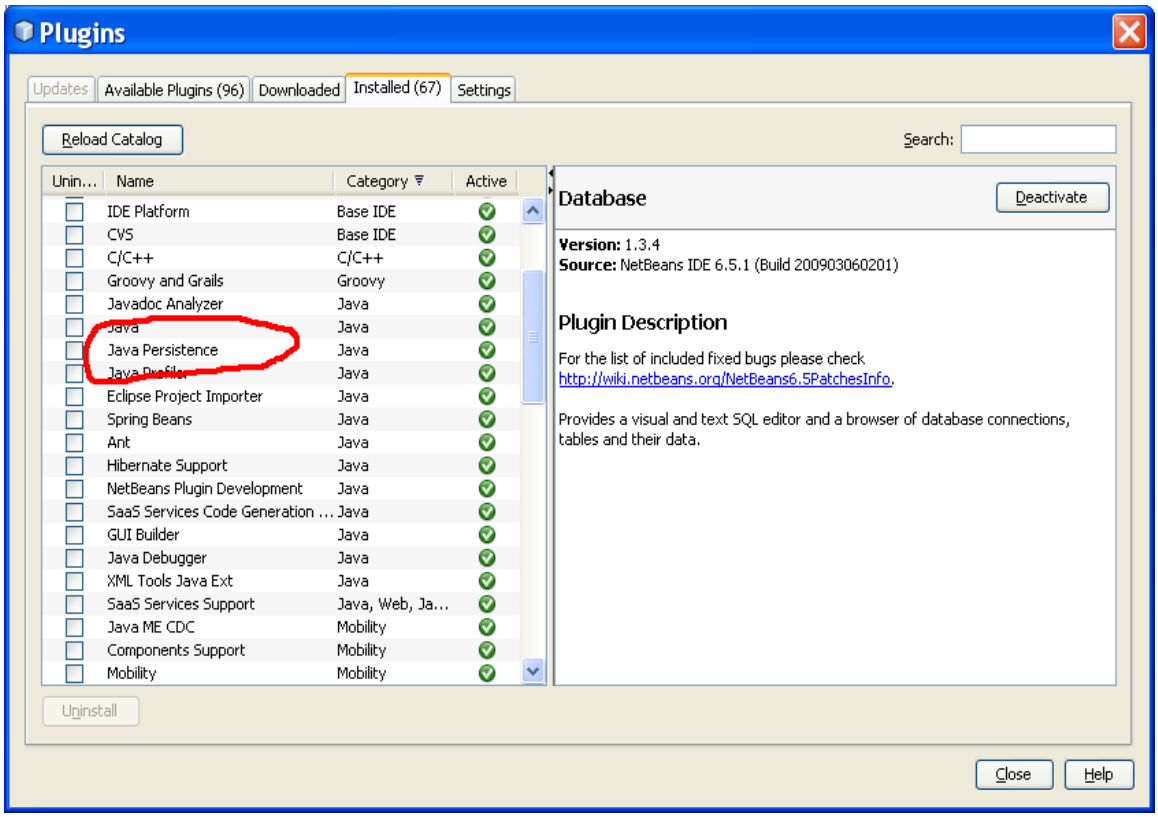
Lezione 11 – Approfondimento JDBC

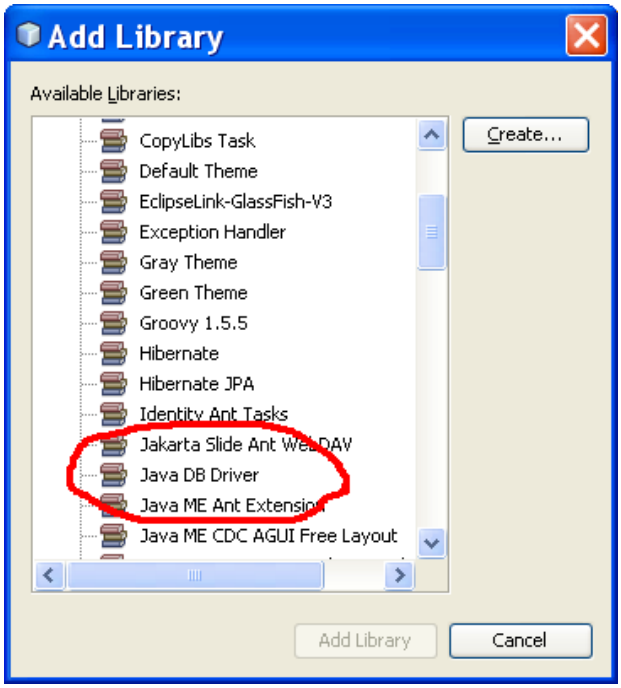
1) Derby

Integrato con Netbeans c'è la possibilità di usare Derby, un database scritto interamente in Java e che nella modalità di utilizzo più semplice non richiede installazione. Tale database può quindi essere usato per lo sviluppo e il test delle applicazioni e successivamente in produzione possiamo scegliere se installarlo nella versione server oppure cambiare DB. Per usare Derby in Netbeans, controllare che sia installato il plug-in “Java-Persistence” e aggiungere la libreria “Java DB Driver”.

Ulteriore semplificazione per l'installazione è la creazione della base dati dall'applicazione web stessa (vedi esempio sotto). In questo modo, alla prima esecuzione su una nuova macchina, il collegamento al database fallisce e nella gestione dell'eccezione si cerca di creare il database stesso usando l'opzione “`create=true`” nell'url del collegamento al database. Un a volta creato il database vuoto possiamo popolarlo tramite Data Definition Language dell'SQL per creare tutte le tabelle necessarie. Eventualmente, sarà anche necessario prevedere l'inserimento di qualche riga di default in qualche tabella (esempio, aggiungere l'utente amministratore con la relativa password di default).

Particolarità della gestione embedded è quella di ricordarsi di “chiudere” il database prima di uscire dall'applicazione. Per chiudere il DB è necessario creare una nuova connessione con l'opzione “`shutdown=true`” nell'url di collegamento al DB. La chiusura del DB non può ritornare una connessione ma lancia una opportuna eccezione con stato “`08006`”.





Esempio con Derby

```
java.sql.Connection con = null;
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

try {
     con = DriverManager.getConnection("jdbc:derby:sample");
    }
    catch (java.sql.SQLException sqle) {
     con = DriverManager.getConnection("jdbc:derby:sample;create=true");
     con.createStatement().executeUpdate("CREATE TABLE PROVA (nome VARCHAR(250), id
INT NOT NULL, cognome VARCHAR(250))");
     //creazione altre tabelle
    }

PreparedStatement s = con.prepareStatement("INSERT INTO PROVA (nome,id,cognome)
VALUES(?,?,?)");

s.setString(1, “Alessandro”);
s.setInt(2, 0);
s.setString(3, “Roncato”);

s.executeUpdate();

ResultSet rs = con.createStatement().executeQuery("select * from PROVA");
while (rs.next()) {
       System.out.println("nome: " + rs.getString(1) + " id:" + rs.getInt(2));
     }
try {//è una versione embedded quindi va chiusa quando l'applicazione termina
     DriverManager.getConnection("jdbc:derby:sample;shutdown=true");
     } catch (SQLException se) {//Stato = 08006 chiusura OK, altrimenti si è
verificato un errore
     if (!se.getSQLState().equals("08006")) {
         throw se;
        }
    }
```

2) Transazioni

JDBC permette di sfruttare una delle caratteristiche più importanti dei Database ovvero la gestione delle transazioni. Tale gestione si effettua attraverso due semplici metodi della classe `Connection`: `commit` e `rollback`. Il significato dei due metodi è quello che ci si aspetta: il metodo `commit` rende definitive

tutte le modifiche apportate usando la connessione fino al precedente `commit` o `rollback`, viceversa il metodo `rollback` le annulla fino al precedente `commit`. All'apertura di una connessione la connessione stessa può rendere definitiva ogni singola modifica senza bisogno di chiamare esplicitamente il metodo `commit` (Auto Commit). Per fare in modo che la semantica dei metodi `commit` e `rollback` sia quella descritta prima dobbiamo togliere la modalitàAuto Commit nella connessione usando il metodo `setAutoCommit(false)`.
E' chiaro che i metodi per la gestione delle transazioni hanno effetto su tutte e sole le operazioni effettuate sul DB tramite la connessione su cui vengono invocati i metodi stessi.
Ecco un semplice schema di utilizzo per implementare una transazione. In questo caso o entrambe le modifiche al database hanno successo (non vengono generate eccezioni), oppure la prima modifica viene annullata dal `rollback`:

```
Connection conn =null;
try{
    conn = DriverManager.getConnection("...");
    conn.setAutoCommit(false);
    Statement st = conn.createStatement();
    st.executeUpdate("DELETE  ...");
    st.executeUpdate("INSERT ...");
    conn.commit();
}
catch (SQLException sqle)
{
    if (conn!=null)
        try{
        conn.rollback();
    }
    catch (SQLException sqle2)
    {
        //log error
    }
}
```

3) Stored procedure (facoltativo)

Le chiamate a stored procedure sono gestite da JDBC in maniera semplice e simile a quanto appena visto, ecco un esempio:
Supponiamo di avere una funzione SQL definita nel modo seguente:

```
FUNCTION procedura (ent_type IN   VARCHAR(20),  num_errori IN   NUMBER
, errore OUT  NUMBER)  RETURN   BOOLEAN IS
```

Il codice java per invocare la funzione appena vista impostando i valori dei parametri di input e recuperare il valore dei parametri di output:

```
Connection conn = null;
CallableStatement cs = null;
ResultSet rs = null;
try {
     conn = getConnection();
     cs = conn.prepareCall("{? = call procedura(?,?,?)}");

     // Registro i parametri di INPUT
     cs.setString(2, parametro1);
     cs.setInt(3, parametro2);
     // Registro i parametri di OUTPUT
     cs.registerOutParameter(1, java.sql.Types.INTEGER);
```

```
     cs.registerOutParameter(4, java.sql.Types.NUMERIC);
     // Esecuzione della stored procedure.
     rs = cs.executeQuery();
     // Note that you need to retrieve the ResultSet _before_
retrieving OUTPUT parameters.
     if ( rs == null) {
            result = false;
      } else {
         rs.next ();
         // Recupero i parametri di OUTPUT
         int intResult = cs.getInt(1);
         BigDecimal errorNumber = cs.getBigDecimal(4);
        }
}
catch (....)
```

Chiamata di store procedure con parametri di input e parametri di output

4) Metadati (facoltativo)

Tramite JDBC è anche possibile conoscere la struttura della query stessa e quindi conoscere il tipo delle colonne ed eventuali altre informazioni.
esempio:

```
ResultSet rs = ...
ResultSetMetaData md = rs.getMetaData();
md.getColumnCount(); //numero totale di colonne;
for (int i=1; i<= md.getColumnCount();i++){
  System.out.print("Name:"+md.getColumnName(i));//il nome della
colonna
  System.out.print("Type:"+md.getColumnType(i));//il tipo della
colonna
  System.out.println("Table:"+md.getTableName(i);//il nome della
tabella da dove proviene la colonna
}
```

E' anche possibile accedere ai metadati della connessione dai quali si possono ricavare informazioni riguardo il database e il driver dedicato al Database stesso. Ecco un esempio che visualizza i nomi delle tabelle utente del DB:

```
     DatabaseMetaData md = con.getMetaData();
     ResultSet rs=md.getTables(null, null, null, new String[]
{"TABLE"});
     while (rs.next()){
        System.out.println( rs.getString("TABLE_NAME"));
     }
```

5) Chiavi primarie generate dal DB

Come spesso accade per ogni tabella è necessario definire una chiave primaria. Ogni record della tabella dovrà avere quindi un diverso valore per la chiave primaria così definita. Se è possibile definire come chiave primaria una delle colonne (o più colonne) di cui si prevede l'inserimento da parte dell'utente, non ci sono in genere problemi di sorta per la gestione della stessa. Nel caso invece che la chiave debba essere generata in automatico dal Database insorgono delle difficoltà nell'uso delle chiavi stessa da parte di applicazioni esterne al DB. Infatti se tra i dati che inseriamo in un record non ce ne sono che ci permettono di individuare in maniera univoca il record stesso, come facciamo ad interrogare il DB per

farci ritornare il record con i dati appena inseriti? Il DB è in grado di creare la chiave primaria in maniera autonoma, ma dall'applicazione non siamo in grado di collegare la chiave primaria ai dati che abbiamo appena inserito.
Vediamo un esempio:
Supponiamo quindi di avere una tabella Ordini in cui memorizziamo tutti gli ordini dei nostri clienti, nella tabella ordini non ci sono chiavi primarie "naturali" e quindi ne definiamo una che sarà creata dal DB stesso (autoincrement?). Questa chiave la chiamiamo IDORDINE. Gli altri dati memorizzati nell'ordine saranno IDCLIENTE e DATA. Ad ogni Ordine sono associate una o più Righe d'ordine che rappresentano gli articoli e le relative quantità che compongono l'ordine. I record della tabella Righe d'ordine devono per forza avere una chiave esterna alla tabella Ordini che per ogni riga d'ordine a che ordine è relativa. Le colonne della tabella Righe saranno quindi: IDORDINE; IDARTICOLO; QUANTITA. Quindi la nostra applicazione Java dovrebbe fare qualcosa del genere:

```
public boolean insertOrdine(int idCliente, Data data, Righe[] righe)
{
connection = DriverManager.getConnection("url", "utente","password");

PreparedStatement insertOrdine = connection.prepareStatement("INSERT
INTO Ordini (IDCLIENTE,DATA) VALUES (?,?)");
insertOrdine.setInt(1,idCliente);
insertOrdine.setData(2,Data);

res = insertOrdine.executeUpdate();

PreparedStatement insertRiga = connection.prepareStatement("INSERT
INTO Righe (IDORDINE,IDARTICOLO,QUANTITA) VALUES (?,?,?)");

for (int i=0; i<Rgihe.length; i++)
{
insertRiga.setInt(1,???);
insertRiga.setInt(2, righe[j].idArticolo);
insertRiga.setInt(3, righe[j].quantita);
res2 = insertRiga.executeUpdate();
}
insertOrdine.close();
connection.close();
return true;
}
```

La prima soluzione che viene in mente è quella di fare una query per farsi ritornare il valore della chiave appena creata dal DB:

```
public boolean insertOrdine(int idCliente, Data data, Righe[] righe)
{
connection = DriverManager.getConnection("url", "utente","password");

PreparedStatement insertOrdine = connection.prepareStatement("INSERT
INTO Ordini (IDCLIENTE,DATA) VALUES (?,?)");
insertOrdine.setInt(1,idCliente);
insertOrdine.setData(2,Data);
res = insertOrdine.executeUpdate();

PreparedStatement readIdOrdine = connection.prepareStatement("SELECT
IDORDINE FROM Ordini WHERE IDCLIENTE=? AND DATA=?");

readIdOrdine.setInt(1,idCliente);
readIdOrdine.setData(2,Data);
```

```
ResultSet rs = readIdOrdine.executeQuery();
int idOrdine = -1;
if (rs.next())
idOrdine=rs.getInt(1);

else return false; //qualcosa di meglio
PreparedStatement insertRiga = connection.prepareStatement("INSERT
INTO Righe (IDORDINE,IDARTICOLO,QUANTITA) VALUES (?,?,?)");

for (int i=0; i<Rgihe.length; i++)
{
insertRiga.setInt(1,idOrdine);
insertRiga.setInt(2, righe[j].idArticolo);
insertRiga.setInt(3, righe[j].quantita);
res2 = insertRiga.executeUpdate();
}
insertOrdine.close();
connection.close();
return true;
}
```

Ma cosa succede se il result set trova più di un Ordine con lo stesso IDCLIENTE e DATA? Dato che queste due colonne non sono univoche (altrimenti le avrei scelte come chiave primaria) l'ipotesi di trovare più ordini con lo stesso chiente e stessa data non è da escludere.

5.1) Sequenze

Quindi qual'è una soluzione sicura del problema? Le sequenze sono state pensate per risolvere questo tipo di problemi. Nei DB professionali esistono delle piccole differenze nella gestione delle sequenze ma sostanzialmente il problema viene risolto allo stesso modo. Sfortunatamente Accesse e MySQL non implementano le sequenze, ma solo l'autoincrement.
Le sequenze sono particolari entità del DB che permettono di gestire dei contatori. La sintassi per creare ed utilizzare una sequenza cambia da DB a DB.
La sintassi Postgress:

- creazione: `CREATE SEQUENCE progressivoOrdine START 1;`
- selezione: `SELECT NEXTVAL('progressivoOrdine').`

La sintassi Oracle:

- creazione: `CREATE SEQUENCE progressivoOrdine START WITH 1 INCREMENT BY 1 CACHE 20 ORDER;`
- selezione: `SELECT progressivoOrdine.NEXTVAL FROM DUAL.`

Ecco come dovrebbe essere implementata l'inserimento di un ordine per il Database Postgress:

```
public boolean insertOrdine(int idCliente, Data data, Righe[] righe)
{
connection = DriverManager.getConnection("url", "utente","password");

PreparedStatement readIdOrdine = connection.prepareStatement("SELECT
NEXTVAL('progressivoOrdine')");

ResultSet rs = readIdOrdine.executeQuery();
int idOrdine = -1;
if (rs.next())
idOrdine=rs.getInt(1);
else
```

```
throws new Exception("Error: NEXTVALsequence progressivoOrdine");


PreparedStatement insertOrdine = connection.prepareStatement("INSERT
INTO Ordini (IDORDINE,IDCLIENTE,DATA) VALUES (?,?,?)");
insertOrdine.setInt(1,idOrdine);
insertOrdine.setInt(2,idCliente);
insertOrdine.setData(3,Data);

res = insertOrdine.executeUpdate();

PreparedStatement insertRiga = connection.prepareStatement("INSERT
INTO Righe (IDORDINE,IDARTICOLO,QUANTITA) VALUES (?,?,?)");

for (int i=0; i<Rgihe.length; i++)
{
insertRiga.setInt(1,idOrdine);
insertRiga.setInt(2, righe[j].idArticolo);
insertRiga.setInt(3, righe[j].quantita);
res2 = insertRiga.executeUpdate();
}
insertOrdine.close();
connection.close();
return true;
}
```

Nota: le sequenze possono simulare l'autoincrement di MySQL e Access: Ad esempio creando una tabella Studenti nel seguente modo:
`CREATE TABLE studenti ( idStudente INTEGER PRIMARY KEY DEFAULT NEXTVAL('sequenza'), ...)` dove sequenza è un nome di un'opportuna sequenza.

5.2) getGeneratedKeys

Il problema è molto frequente tanto che le API JDBC stesse prevedono la possibilità di recuperare in automatico la chiave creata. Purtroppo pochi sono i driever disponibili che implementano questa funzionalità. Fortunatamente il database Derby lo permette. Ad esempio:

```
 java.sql.Connection con = null;
 Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

try {
       con = DriverManager.getConnection("jdbc:derby:sample");
    } catch (java.sql.SQLException sqle) {
       con =
DriverManager.getConnection("jdbc:derby:sample;create=true");
          con.createStatement().executeUpdate("CREATE TABLE PROVA
(nome VARCHAR(250),
           id  INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START
WITH 1, INCREMENT BY 1),
           cognome VARCHAR(250))");
     }

PreparedStatement s = con.prepareStatement("INSERT INTO PROVA
(nome,cognome) VALUES(?,?)",Statement.RETURN_GENERATED_KEYS);

s.setString(1, “Alessandro”);
s.setString(2, “Roncato”);
```

```
s.executeUpdate();
ResultSet rs = s.getGeneratedKeys();
if (rs.next()){
     System.out.println("generated id "+rs.getInt(1));
}


ResultSet rs = con.createStatement().executeQuery("select * from
PROVA");
while (rs.next()) {
     System.out.println("nome: " + rs.getString(1) + " id:" +
rs.getInt(2));
       }
try {//embedded quindi chiudiamo il DB quando l'applicazione termina
     DriverManager.getConnection("jdbc:derby:sample;shutdown=true");
     } catch (SQLException se) {//Stato = 08006 chiusura OK, altrimenti
si è verificato un errore
       if (!se.getSQLState().equals("08006")) {
           throw se;
        }
  }
```

6) Perché Class.forName (facoltativo)

La classe DriverManager deve per forza gestire al run-time il collegamento tra url e Driver relativo. Ricordiamo che JDBC permette di cambiare il tipo di DB server (e quindi il relativo Driver) successivamente alla compilazione è chiaro il collegamento tra url e relativo Driver debba essere fatto al run-time. Per questo il DriverManager pubblica il metodo statico registerDriver(Driver d) che permette a un Driver di registrarsi presso il DriverManager. Ma chi chiama questo metodo. E' compito del Driver stesso. Ma come fa il Driver a invocare questo metodo? Basta che nella classse del Driver sia definito il seguente codice:

```
public classs MyDriver implements Driver {

static {

DriverManager.registerDriver(new MyDriver());

}

…//resto del driver

}
```

Quando viene eseguito questo codice (fuori da ogni metodo della classe)? Al momento della prima uso della classe. Dato che il Driver risiede su un file jar esterno all'applicazione, il primo uso lo induciamo con il metodo Class.classForName(“MyDriver”).

Successivamente, il Driver manager farà qualcosa di simile per implementare il getConnection:

```
public class DriverManager {

Set<Driver> drivers = new ListSet<Driver>();

public Connection getConnection(String url) throws … {

for (Driver d: getDrivers()){

if (d.acceptsUrl(url))

return d.connect(String url, Properties info)

}

throw new …;

}

public void registerDriver(Driver d){

driver.add(d);

}

}
```

7) Vantaggi JDBC

- comandi in SQL (standard);
- driver indipendenti dal codice: per usare un DB diverso cambio i driver ma non ricompilo;