# Interazione tra basi di dati e applicazioni



ALBERTO BELUSSI

ANNO ACCADEMICO 2018/2019

# Tipi di interazione



- Interazione via cursore (metodo classico): un API dedicata messa a disposizione dal DBMS consente di sottomettere comandi SQL al sistema e ottenere risultati di interrogazioni rappresentati come cursori (iteratori su liste di tuple).
- Interazione via cursore con **API standardizzata**: come sopra ma con una certa indipendenza dal DBMS (libreria JDBC di Java, ODBC di Microsoft)
- **Object Relational Mapping (ORM)**: è una tecnica che consente di gestire nell'applicazione oggetti "persistenti" e di astrarre dalla base di dati relazionale. Consente un livello di interazione con il DB che maschera l'SQL. Esempi sono: Java Persistence API o Hibernate.

# Interazione con DB in Java attraverso JDBC



In Java è possibile interagire con un DBMS attraverso l'uso della libreria JDBC (Java Database Connectivity).

http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html

JDBC API: fornisce un insieme di classi JAVA che consentono un accesso standardizzato a qualsiasi DBMS purché questo fornisca un driver JDBC.

# Driver JDBC



E’ un modulo software in grado di interagire con un DBMS. Traduce ogni invocazione dei metodi delle classi JDBC in comandi SQL accettati dal DBMS a cui è dedicato.

# 7 passi per interagire con un DBMS



## 1° PASSO

Caricare il driver JDBC per il DBMS che si utilizza.

- Per caricare il driver, basta caricare la classe corrispondente:

```
import java.sql.*
…
Class.forName("NomeDriver");
```

Per postgresql il nome del driver è:

```
org.postgresql.Driver
```

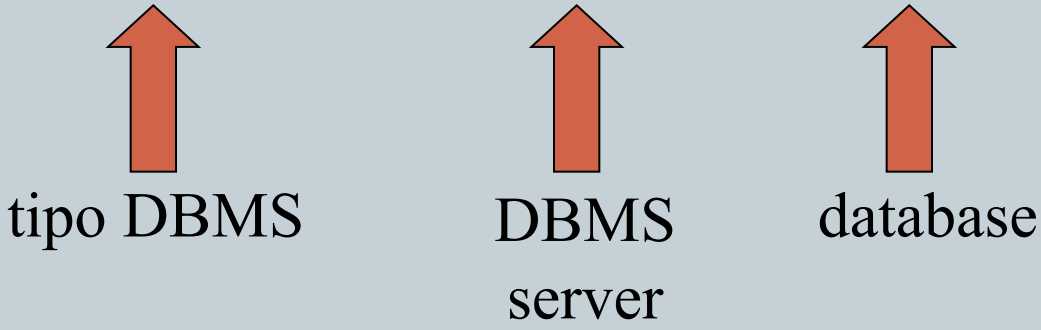
# 7 passi per interagire con un DBMS



## 2° PASSO

Definire una connessione identificando l'URL della base di dati a cui ci si vuole connettere.

```
String URL = "jdbc:postgresql://dbserver/did2013"
```



# 7 passi per interagire con un DBMS



## 3° PASSO

Stabilire una connessione istanziando un oggetto della classe `Connection`:

```
String user = "Utente-postgres";
String passwd = "Password-utente";
Connection con = DriverManager.getConnection(
                    URL, user, passwd);
```

# 7 passi per interagire con un DBMS



**4° PASSO**

Utilizzando l'oggetto della classe `Connection` creato al passo precedente, creare un oggetto della classe `Statement` per poter sottomettere comandi al DBMS:

```
Statement stat = con.createStatement();
```

# 7 passi per interagire con un DBMS



## 5° PASSO

Eseguire un'interrogazione SQL o un comando SQL di aggiornamento di una tabella:

- Interrogazione

```
String query = "SELECT * FROM PERSONA";
ResultSet res = stat.executeQuery(query);
```

- Aggiornamento

```
String update = "UPDATE PERSONA"+
                "SET NOME = 'Rosa' WHERE id = 1";
stat.executeUpdate(update);
```

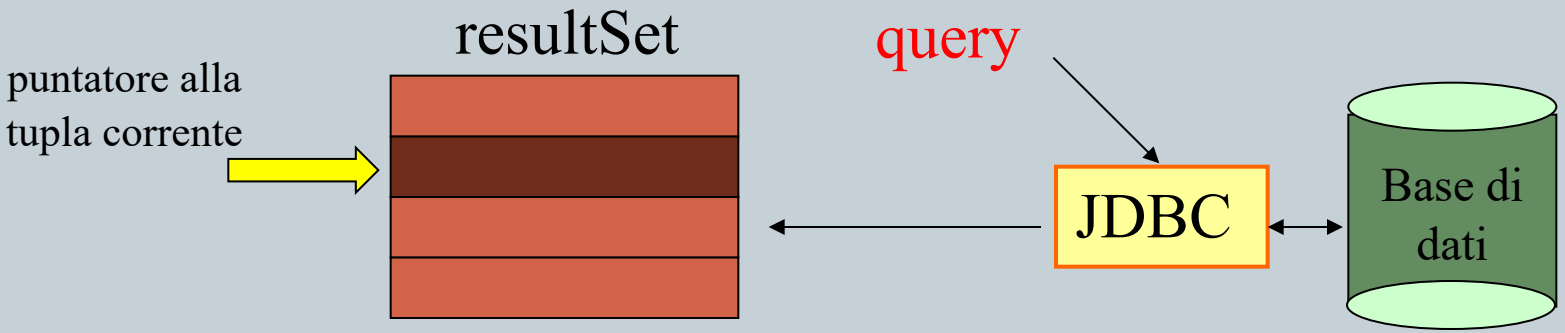
# 7 passi per interagire con un DBMS



## 6° PASSO

Processare il risultato dell'interrogazione attraverso l'applicazione dei metodi della classe `resultSet`.

Il resultSet è un *cursore* che consente di accedere alle tuple risultato dell'interrogazione:



# 7 passi per interagire con un DBMS



## 6° PASSO

E' possibile scandire in modo sequenziale il contenuto di un resultSet:

```
String query = "SELECT * FROM PERSONA";
resultSet res = stat.executeQuery(query);
while(res.next()) {
        ……
}
```

Il metodo `next()` sposta il puntatore sulla tupla successiva.

# 7 passi per interagire con un DBMS



## 6° PASSO

E' possibile accedere alle proprietà della **tupla corrente** di un resultSet con i seguenti metodi:

`getXxxx(par)`: dove `Xxxx` è un tipo base di Java e `par` può essere un indice di posizione o il nome di un attributo della relazione risultato dell'interrogazione; questo metodo restituisce il valore in posizione `par` oppure il valore dell'attributo di nome `par` della tupla corrente.

`wasNull`: si riferisce all'ultima invocazione di `getXxxx` e restituisce true se il valore letto era uguale al valore nullo.

*… molti altri metodi sono disponibili, si veda:*

https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html

# 7 passi per interagire con un DBMS



## 7° PASSO

Chiudere la connessione usando l'oggetto della classe `Connection`:

```
con.close();
```

# 7 passi per interagire con un DBMS



**Variante del 5° PASSO**

E' possibile ottimizzare l'esecuzione di una interrogazione che deve essere rifatta più volte usando la classe `PreparedStatement`.

Tale classe consente di inserire parametri nell'interrogazione e di valorizzarli, attraverso specifici metodi, prima dell'effettiva esecuzione dell'interrogazione stessa.

# 7 passi per interagire con un DBMS



## Variante del 5° PASSO

Esempio:

```
Connection con = DriverManager.getConnection(URL,
    user, passwd);
String q = "SELECT Nome, Cognome"+
          " FROM Persona"+
          " WHERE id = ?";
PreparedStatement pstat = con.prepareStatement(q);
pstat.setInt(1, 15);
ResultSet res = pstat.executeQuery();
```

# Transazioni in JDBC



E' possibile eseguire transazioni in JDBC come segue:

```
…
con.setAutoCommit(false);
preparedStatement ps1 = con.prepareStatement(
    "UPDATE CONTO SET SALDO=SALDO+?"+
    " WHERE NUMERO=? AND FILIALE = 'X'");
ps1.setInt(1, 1000); ps1.setString(2, "358");
ps1.execute();
preparedStatement ps2 = con.prepareStatement(
    "UPDATE CONTO SET SALDO=SALDO-?"+
    " WHERE NUMERO=? AND FILIALE = 'X'");
ps2.setInt(1, 1000); ps2.setString(2, "876");
ps2.execute();
con.commit();
con.setAutoCommit(true);
```

# Transazioni in JDBC



- Durante la transazione è possibile attivare il trasferimento per blocchi di tuple in un `ResultSet`
- L'impostazione di default è che l'intero insieme di tuple risultato dell'interrogazione venga trasferito nel `ResultSet`
- Per alterare tale situazione si usa il metodo `setFetchRow(int rows)` di `Statement`
- Usando tale metodo è possibile fare in modo che il trasferimento delle tuple risultato di un'interrogazione dal DBMS all'applicazione avvenga in lotti di al massimo `rows` tuple (quindi eventualmente prevedendo più interazioni per scaricare tutto il risultato)

# Transazioni in JDBC

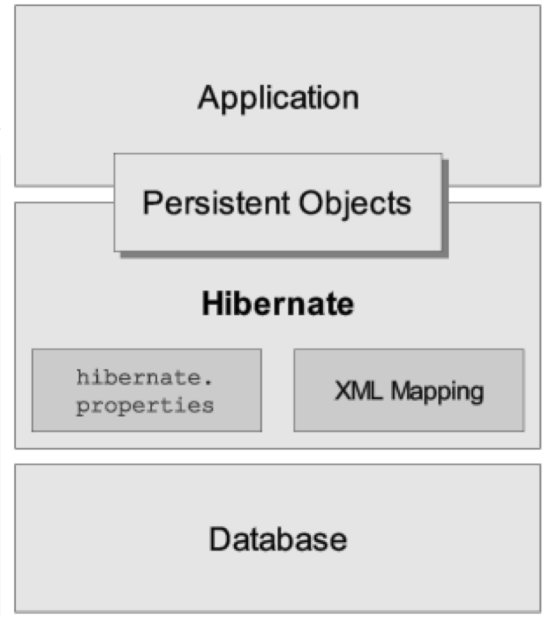


## ESEMPIO (da manuale di postgresql)

```
// make sure autocommit is off
conn.setAutoCommit(false);
Statement st = conn.createStatement();
// Turn use of the cursor on.
st.setFetchSize(50);
ResultSet rs = st.executeQuery("SELECT * FROM mytable");
while (rs.next()) {
   System.out.print("a row was returned."); }
rs.close();
// Turn the cursor off.
st.setFetchSize(0);
```

# Object Relational Mapping






Architettura:

- Uno strato software (gestore della persistenza) si interpone tra l'applicazione e il DB. L'applicazione vede il DB come insieme di oggetti persistenti.
- Viene specificato in un file di mapping (o direttamente sulle classi Java con annotazioni) la corrispondenza tra oggetti persistenti e tabelle del DB.

Vantaggi/Svantaggi:

- Il gestore della persistenza genera parte delle interrogazioni SQL necessarie per generare gli oggetti estraendo dati dalla base di dati in modo trasparente al programmatore.
- La navigazione nei dati segue puntatori tra oggetti
- Le interrogazioni più complesse (che non possono sfruttare i puntatori tra oggetti) vanno gestite ad hoc con SQL nativo.
- Le interrogazioni ad elevata cardinalità possono caricare l'intero DB in memoria (rischio di saturazione della memoria dedicata all'applicazione)
- I meccanismi di generazione automatica degli oggetti riferiti da altri possono caricare l'intero DB in memoria!

# Object Relational Mapping



Esempio di mapping:

Associazione di classe “Event” e tabella “EVENTS”

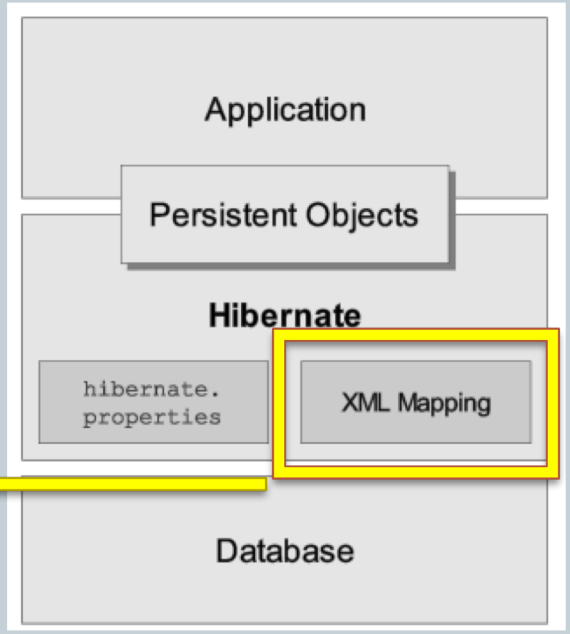
```
<class name="Event" table="EVENTS">
        ...
</class>
```

Dichiarazione di quale colonna della tabella contiene l’id degli oggetti

```
<id name="id" column="EVENT_ID">
    ...
</id>
```

Dichiarazione di alcune proprietà:

```
<property name="date" type="timestamp"
                      column="EVENT_DATE"/>
<property name="title"/>
```



# Altri approcci per l'interazione tra DB e applicazioni



Evoluzioni in diverse direzioni per superare i problemi legati al cosiddetto "*impedance mismatch*"

Lato DBMS (dal relazionale verso nuovi modelli)

- **Object-relational model** (SQL3): tuple (oggetti) con struttura complessa, tabelle (classi) con ereditarietà, navigazione tra le tuple attraverso riferimenti diretti (ref), SQL esteso per interrogare gerarchie di tuple.
- ***Dati semistrutturati***: dati a struttura complessa e a schema variabile, rappresentazione con linguaggi a marca (XML), interrogazioni con X-Path, X-Query.
- ***Document-based models*** (NoSQL database): collezioni di documenti con struttura complessa, dati ridondanti e voluminosi
- ***Sistemi basati su cluster per Big Data***: collezione di dati a struttura complessa e variabile, gestiti da sistemi diversi (dato distribuito), dati voluminosi, interrogazioni distribuite ed eseguite in parallelo.