

JDBC – Concetti base



Paolo Merialdo
Università degli Studi Roma Tre



This work is licensed under a Creative Commons Attribution-NonCommercial-Share Alike 3.0 United States
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details

Concetti Fondamentali

- Introduzione
 - Architettura
- Un Esempio
 - Driver e DriverManager
 - Connection
 - PreparedStatement
 - ResultSet

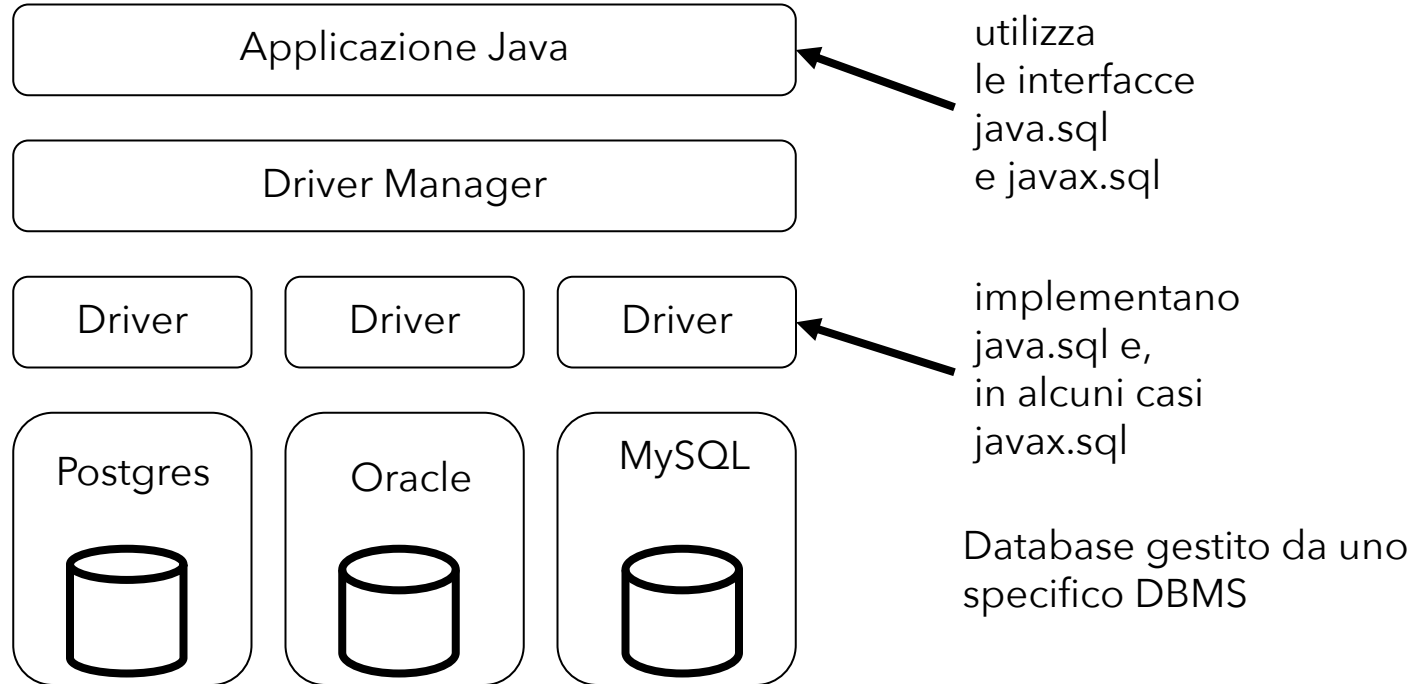
Introduzione

- Un'applicazione che accede (in lettura e/o scrittura) ad una sorgente di dati (nel nostro caso un database relazionale) ha bisogno di fare le seguenti operazioni:
 1. aprire una "connessione" alla sorgente dati
 2. inviare attraverso la connessione istruzioni (di interrogazione o di aggiornamento) alla sorgente dati
 3. processare i risultati ricevuti dalla sorgente dati in risposta alle istruzioni inviate

Introduzione

- Le nostre sorgenti dati sono database (in questo corso relazionali) gestiti da un DBMS
- Ogni DBMS espone una API
- Le applicazioni Java interagiscono con le API del DBMS attraverso un driver
- Possiamo pensare il driver come un interprete che permette all'applicazione di dialogare con il DBMS attraverso un'interfaccia standard (JDBC)

Architettura



Introduzione

- JDBC è una libreria Java che ci permette per interagire con un DBMS
- La libreria offre classi e metodi per
 - Aprire una connessione con il DBMS
 - Inviare istruzioni SQL (insert, delete, update, select) al DBMS
 - Ricevere e gestire i risultati delle istruzioni inviate al DBMS
- E' un libreria di "basso livello"
 - Sopra JDBC sono state costruite varie tecnologie che semplificano ulteriormente l'interazione con il DBMS

Esempio

- Obiettivo: esemplificare l'API JDBC
 - scriviamo codice che ha unicamente lo scopo di evidenziare l'uso della API JDBC
 - non ci preoccupiamo (troppo) della qualità del codice
- Operazioni
 - inserimento di una ennupla nel database
 - cancellazione di una ennupla
 - ricerca di un ennupla per chiave primaria
 - ricerca di un insieme di ennupla (per qualche proprietà)

Esempio

- Consideriamo la classe Studente:

```
package it.uniroma3.bd;
```

```
public class Studente {  
    private String matricola;  
    private String nome;  
    private String cognome;  
    private Date dataNascita;
```

```
    public Studente() {}
```

```
    // seguono i metodi getter e setter, equals, hashCode, toString  
}
```

Esempio

e il database university:

```
CREATE DATABASE university;
```

```
CREATE TABLE studente
```

```
(  
    matricola character(8),  
    nome character varying(64) NOT NULL,  
    cognome character varying(64) NOT NULL,  
    datanascita date NOT NULL,  
    CONSTRAINT pk_studente PRIMARY KEY (matricola)  
)
```

Ambiente

- Ambiente Java standard
- DBMS
 - Noi usiamo PostgreSQL (ma andrebbe bene MySQL, Oracle, etc)
- Driver JDBC per il DBMS scelto
 - con Postgresql, scaricare il driver da <http://jdbc.postgresql.org/>
 - con altri DBMS, scaricare un driver appropriato

In pratica

- Scaricare il driver (è una libreria Java, nella forma di un file .jar) da <http://jdbc.postgresql.org/>
- Salvare il driver in una directory opportuna
 - Nella nostra home (la directory con il nome dell'utente) creiamo la directory **javailib**
- Aprire Eclipse
- Creare il progetto Java **bd-jdbc**
- Tasto destro del mouse sul nome del progetto
 - Build Path
 - Add External Archive...
 - Selezionare il jar di postgresql che abbiamo salvato nella directory ~/javailib
 - Apply and Close

Package Explorer X JUnit *Main.java X

- bd-jdbc
 - dummy
 - siw-jpa-es
 - siw-jpa-es
 - siw-servle
 - siw-spring
 - siw-spring
 - validating

- New
- Go Into
- Open in New Window
- Open Type Hierarchy F4
- Show In ⌘ W
- Copy ⌘ C
- Copy Qualified Name
- Paste ⌘ V
- Delete
- Build Path
- Source ⌘ S
- Refactor ⌘ T
- Import...
- Export...
- Refresh F5
- Close Project
- Close Unrelated Projects
- Assign Working Sets...
- Coverage As
- Run As
- Debug As
- Restore from Local History...
- Team
- Compare With
- Configure
- Properties ⌘ I

1 package it.uniroma3.bd;

c class Main {

public static void main(String[] args) {

- Link Source...
- New Source Folder...
- Use as Source Folder
- Add External Archives...
- Add Libraries...
- Configure Build Path...

@ Javadoc Declaration Console

s, 0 others

Path	Location	Type
------	----------	------

Le classi fondamentali di JDBC

- Classe **DriverManager**
- Interfaccia **Driver**
- Interfaccia **Connection**
- Interfaccia **PreparedStatement**
- Interfaccia **ResultSet**
- Eccezione **SQLException**

Un Esempio

- Nel seguito, con riferimento al nostro esempio, descriveremo le seguenti operazioni
 - Operazione 1: Caricare il driver
 - Operazione 2: Aprire una connessione
 - Operazione 3: Definire l'istruzione SQL
 - Operazione 4: Gestire il risultato

Organizzazione del codice

- In una classe Main creiamo una connessione al database ed eseguiamo alcune operazioni elementari (inserimenti, cancellazione, interrogazioni)
- In una classe StudentRepository mettiamo i metodi che astraggono le operazioni di accesso al database

Organizzazione del codice

- La classe StudenteRepository offre i metodi

```
public void save(Studente studente)
```

```
public void deleteAll()
```

```
public List<Studente> findAll()
```

```
public Studente findByMatricola(String matricola)
```

```
public void deleteByMatricola(String matricola)
```

Operazione 1: Caricare il Driver

- Per caricare il driver usiamo la seguente istruzione:

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

Operazione 2: Ottenere una Connessione

- Ottenere una connessione dal **DriverManager**
 - per farlo è necessario specificare:
 - host, dbms, db
 - utente e password
- URI (indirizzo) della connessione
 - specifica server, porta e database
- Sintassi: **jdbc:<sottoprotocollo>:<parametri>**

Operazione 2: Ottenere una Connessione

- URI per Postgres

`jdbc:postgresql:<baseDati>`

`jdbc:postgresql://<host>/<baseDati>`

`jdbc:postgresql://<host>:<porta>/<baseDati>`

- Esempi

`jdbc:postgresql:university`

`jdbc:postgresql://127.0.0.1/university`

`jdbc:postgresql://193.204.161.14:5432/university`

`jdbc:odbc:university`

Operazione 2: Ottenere una Connessione

- Creazione della connessione

```
Connection DriverManager.getConnection(  
    String uri,  
    String utente,  
    String password);
```

- Esempio

```
Connection connection;  
connection = DriverManager.getConnection(  
    "jdbc:postgresql:university" ,  
    "postgres",  
    "postgres" );
```

Nell'esempio

- Nella classe Main definiamo il metodo **Connection getConnection()** che restituisce una nuova connessione ad ogni richiesta
- E' una soluzione artigianale usata a fini didattici (ma ha un minimo di dignità)

Il metodo `Connection getConnection()`

```
package it.uniroma3.bd;
import java.sql...;

public class Main {

    private String dbURI = "jdbc:postgresql://localhost/university";
    private String userName = "postgres";
    private String password = "postgres";

    public Connection getConnection() throws SQLException, ClassNotFoundException {
        Connection connection = null;
        Class.forName("org.postgresql.Driver");
        connection = DriverManager.getConnection(dbURI, userName, password);
        return connection;
    }

    // vediamo dopo
}
```

Operazione 3: Istruzione SQL

- Vediamo il codice JDBC che esegue istruzioni SQL per:
 - salvare (rendere persistenti) oggetti nel db
 - cancellare oggetti dal db
 - trovare oggetti dal db
- Confiniamo il codice di queste operazioni in un metodo

Operazione 3: Istruzione SQL

- Per eseguire una istruzione SQL è necessario creare un oggetto della classe che implementa **PreparedStatement**
 - creato dall'oggetto Connection invocando il metodo:

```
PreparedStatement prepareStatement(String s);
```

- La stringa s è una istruzione SQL parametrica: i parametri sono indicati con il simbolo ?

- Esempio 1

```
String insert = "insert into studente(nome, cognome, datanascita,  
matricola) values (?, ?, ?, ?)";
```

```
PreparedStatement statement = connection.prepareStatement(insert);
```

- Esempio 2

```
String delete = "delete from studente where matricola=?";
```

```
PreparedStatement statement = connection.prepareStatement(delete);
```

Operazione 3: Istruzione SQL

- I parametri sono assegnati mediante opportuni metodi della classe che implementa **PreparedStatement**
 - metodi setXXX (<numPar>, <valore>)
 - un metodo per ogni tipo, il primo argomento corrisponde all'indice del parametro nella query, il secondo al valore da assegnare al parametro
- Esempio 1 (cont.)

```
PreparedStatement statement;
```

```
String insert = "insert into studente(nome, cognome, datanascita,  
matricola) values (?, ?, ?, ?)";
```

```
statement = connection.prepareStatement(insert);
```

```
statement.setString(1, studente.getNome());
```

```
statement.setString(2, studente.getCognome());
```

```
statement.setDate(3, Date.valueOf(studente.getDataDiNascita()));
```

```
statement.setString(4, studente.getMatricola());
```

Operazione 3: Istruzione SQL

- JDBC usa **`java.sql.Date`**, mentre la classe `Studente` usa **`java.time.LocalDate`**
- L'istruzione

```
Date.valueOf(studente.getDataDiNascita())
```

serve a "convertire" una data da una rappresentazione all'altra

Operazione 3: Istruzione SQL

- Una volta assegnati i valori ai parametri, l'istruzione può essere eseguita
- Distinguiamo due tipi di operazioni:
 - Modifiche (insert, update, delete)
 - modificano lo stato del database
 - interrogazioni (select)
 - non modificano lo stato del database
 - ritornano una sequenza di tuple

Operazione 3:

Istruzione SQL di aggiornamento

- Per le operazioni di aggiornamento (insert, delete, update)
 - vengono eseguiti invocando il metodo **executeUpdate()** sull'oggetto **PreparedStatement**
- Esempio 1 (cont.)

```
PreparedStatement statement;
```

```
String insert = "insert into studente(nome, cognome, dataNascita, matricola)  
values (?, ?, ?, ?)";
```

```
statement = connection.prepareStatement(insert);
```

```
statement.setString(1, studente.getNome());
```

```
statement.setString(2, studente.getCognome());
```

```
statement.setDate(3, Date.valueOf(studente.getDataDiNascita()));
```

```
statement.setString(4, studente.getMatricola());
```

```
statement.executeUpdate();
```

Operazione 3:

Istruzione SQL di interrogazione

- Per le interrogazioni (select)
 - vengono eseguite invocando il metodo **executeQuery()**
 - che ritorna il risultato in un oggetto **ResultSet**
- Esempio 2

```
PreparedStatement statement;  
String query = "select * from studente where matricola=?";  
statement = connection.prepareStatement(query);  
statement.setString(1, matricola);  
ResultSet result = statement.executeQuery();
```

Operazione 4: Gestire il risultato di una query

- Un oggetto della classe **ResultSet** rappresenta la collezione di ennuple restituita da una query SQL (istruzione **SELECT**)
- Per gestire il risultato offre vari metodi:
 - metodo **boolean next()** per scorrere le ennuple (analogo ad un iteratore)
 - metodi **getXXX(String attributo)** per acquisire i valori degli attributi
 - Es.: `int getInt(String attributo);`
 - Es.: `String getString(String attributo);`

Operazione 4: Gestire il risultato di una query

- Il metodo **next ()**

- Moves the cursor forward one row from its current position. A **ResultSet** cursor is initially positioned before the first row; the first call to the method next makes the first row the current row; the second call makes the second row the current row, and so on.
- When a call to the next method returns false, the cursor is positioned after the last row. Any invocation of a **ResultSet** method which requires a current row will result in a **SQLException** being thrown.
- Returns: **true** if the new current row is valid; **false** if there are no more rows

Operazione 4: Gestire il risultato di una query

```
public List<Studente> findAll() throws SQLException {  
    PreparedStatement statement;  
    String query = "select * from studente";  
    List<Studente> studenti = new ArrayList<>();  
    Studente studente;  
  
    statement = connection.prepareStatement(query);  
    ResultSet result = statement.executeQuery();  
    while (result.next()) {  
        studente = new Studente();  
        studente.setMatricola(result.getString("matricola"));  
        studente.setNome(result.getString("nome"));  
        studente.setCognome(result.getString("cognome"));  
        studente.setDataDiNascita(result.getDate("datanascita").toLocalDate());  
        studenti.add(studente);  
    }  
    return studenti;  
}
```