Insegnamento di Laboratorio di Basi di Dati

Lezione 7: Accesso a una base di dati PostgreSQL da un programma Python/Java

Roberto Posenato

ver. 2.1, 17/03/2018

Sommario

Accesso al DBMS in un formato standard

- DB-API v2.0 è la Application Program Interface (API) ufficiale (Python Enhancement Proposals (PEP) 249, https://www.python.org/dev/peps/pep-0249) che descrive come un modulo Python deve accedere a una base di dati esterna.
- Diversi gruppi di sviluppo hanno reso disponibili moduli (librerie) DB-API per diversi tipi di DBMS.
- Alla pagina https://wiki.python.org/moin/DatabaseInterfaces c'è l'elenco aggiornato dei moduli disponibili.
- Per PostgreSQL ci sono 10 implementazioni diverse di DB-API v2.0.

Descrizione dell'applicazione

DB-API più corretta per accedere al dbms

- In questa lezione si introduce DB-API v2.0 considerando l'implementazione psycopg2 (http://initd.org/psycopg/).

Tutti i moduli che ci sono funzionano senza dover cambiare nulla... l'unica cosa che cambia magari sono le performance

Connect -> connettersi ad un DBMS return tipo connector (rappresenta la connessione del nostro DBMS che abbiamo richiesto)
Non si può cambiare la connessione che viene con il database devo rifarla

L'accesso a un database avviene tramite un oggetto di tipo `Connection`. Il metodo `connect(...)` accetta i parametri necessari per la connessione e ritorna un oggetto `Connection`.

```
connector=psycopg2.connect(host="dbserver.scienze.univr.it", \
database="db0", user="user0", password="xxx" )
```

Classe Connection: metodi principali (1/4)

Testo

Sottomettere delle query ed accedere ai risultati delle query

1. `cursor()`: ritorna un *cursore* della base di dati. Un oggetto cursore permette di inviare comandi SQL al DBMS e di accedere al risultato del comando restituito dal DMBS.
   Canale dove si può inviare e ricevere piu query e posso avere più cursor sullo stesso canale
   Indipendentemente tra di loro (lo possono essere ma condivisione del canale)
   Sincronizzazione e gestione di questa ci pensa alla libreria
2. `commit()`: registra la transazione corrente. Attenzione! Normalmente una connessione apre una *transazione* al primo invio di comandi. Se non si esegue un `commit()` prima di chiudere, tutte le eventuali modifiche/inserimenti vengono persi.

Ogni volta che devo ricevere e inviare query ovvero devono lavorare su una transizione e viene fatto il begin e end e tutti i comandi stanno dentro la transazione
Finchè non do il comando commit() oppure rollback
Comunicazione resta aperta e verrà creata un'altra transazione

Classe Connection: metodi principali (2/4)

3. `rollback()`: abortisce la transazione corrente.
4. `close()`: chiude la connessione corrente. Implica un `rollback()` automatico delle operazioni non registrate. Chiude la connessione corrente
5. `autocommit`: proprietà r/w. Se `True`, ogni comando inviato è una transazione isolata. Se `False` (default) il primo comando inviato inizia una transazione, che deve essere chiusa con `commit()` o `rollback()`. Definiti come campi Possibili configurarli prima di aprire la connessione Primo comando della connessione apre la transazione Se metto a true io ogni comando si fa begin e commit perciò ad ogni update si registrano in automatico
6. `readonly`: proprietà r/w. Se `True`, nella sessione non si possono inviare comandi di modifica dati. Il default è `False`.
7. `isolation_level`: proprietà r/w. Modifica il livello di isolamento per la prossima transazione. Valori leciti: 'READ UNCOMMITTED', 'READ COMMITTED', 'REPEATABLE READ', 'SERIALIZABLE', 'DEFAULT'. **Meglio assegnare questa variabile subito dopo la creazione della connessione.**

```
connector = psycopg2.connect(...)
connector.isolation_level = 'REPEATABLE READ'
```

L'oggetto che si usa per le varie interrogazioni

Un `cursore` gestisce l'interazione con la base di dati: mediante un cursore è possibile inviare una comando SQL e accedere all'esito e ai dati di risposta del comando.

Per sapere se ho fatto l'update corretto

Classe Cursore: metodi principali (1/4)

1. `execute(comando, parametri)`: prepara ed esegue un 'comando' SQL usando i 'parametri'. Parametri devono essere passati come tupla o come dict.
   Il comando ritorna `None`. Eventuali risultati di query si devono recuperare con il `fetch*()`.

Execute primo comando

Query

Parametri per completare la query

Intero (se non metto niente è il primo numero intero disponibile)

```
cur.execute("CREATE TABLE test (id SERIAL PRIMARY KEY, num \
integer, data varchar)")
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)", \
(100, "abc'def"))#psyconpg2 fa le conversioni!
```

Per segnare il segnaposto

Passo i valori con quelli che definisco

CONTA L'ORDINE

Execute si preoccupa lui di convertire in modo corretto i valori che indico come i parametri senza dover farlo io

Se devo mettere un solo parametro values(%s) , (Pippo,) usa la virgola

Riesegue la stessa query per tutti i parametri specificati

Classe Cursore: metodi principali (2/4)

2. `executemany(comando, parametri)`: prepara ed esegue un 'comando' SQL per ciascun valore presente nella lista 'parametri'.

```
cur.executemany("INSERT INTO test (num, data) VALUES (\%s, \%s)",
    [(100, "abc'def"), (None, 'dada'), (42, 'bar')]
)
```

Meno efficiente

Riesegue l'inserimento per tutte le tuple scritte

Nota!

Più efficiente fare questo metodo

Per come è attualmente implementato, `executemany()` è meno efficiente di un ciclo `for` con `execute()` o, meglio ancora, di un unico insert con più tuple:

```
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s), \
(%s, %s), (%s, %s)", (100, "abc'def", None, 'dada', 42, 'bar'))
```

Per accedere ai risultati

Classe Cursore: metodi principali (3/4)

3. `fetchone()`: ritorna una tupla della tabella risultato. Si può usare dopo un `execute("SELECT ...")`. Se non ci sono tuple, ritorna `None`.

Accede alla risposta

Ogni volta che viene invocato restituisce una tupla partendo dalla prima

```
>>>cur.execute("SELECT * FROM test WHERE id = %s", (3,))
>>>cur.fetchone()
(3, 42, 'bar')
```

4. `fetchmany(<numero>)`: ritorna una lista di tuple della tabella risultato di lunghezza max <numero>. Si può usare dopo un `execute("SELECT ...")`. Se non ci sono tuple, ritorna una lista vuota.

Lista di tupla che è una riga del risultato con un numero definito dal numero che inseriamo noi

```
>>>cur.execute("SELECT * FROM test WHERE id < %s", (4,))
>>>cur.fetchmany(3)
[(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
>>>cur.fetchmany(2)
[]
```

Tuple

Una lista

Lista vuota

Classe Cursore: metodi e campi principali (4/4)

5. Dopo un `execute("SELECT ...")`, il cursore è un iterabile sulla tabella risultato. È possibile quindi accedere alle tuple del risultato anche con un ciclo

```
>>>cur.execute("SELECT * FROM test WHERE id < %s", (4,))
>>>for record in cur:
...print(record, end=", ")
(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar'),
```

Il cursore è visto come una lista di tuple ed è come un oggetto dentro al for come un iterabile

Quante righe ci sono sulla tabella

6. `rowcount`: di sola lettura, = numero di righe prodotte da ultimo comando. -1 indica che non è possibile determinare il valore.
7. `statusmessage`: di sola lettura, = messaggio ritornato dall'ultimo comando eseguito.

Utile sopratutto con l'inserimento che mi restituisce come Postgres

```
>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)", \
     (42, 'bar'))
>>> cur.statusmessage
'INSERT 0 1'
```

`cursor.execute*` accetta solo `%s` come indicatore di posizione parametro. La conversione dal tipo Python al dominio SQL è automatica per tutti i tipi fondamentali.

```
>>> cur.execute("INSERT INTO test1 (id, date_val, item) VALUES \
    (%s, %s, %s)", (42, datetime.date(2005, 11, 18), "O'Reilly"))
```

Converte lui anche datetime senza nessun problema

è convertita in SQL

```
INSERT INTO test1 (id, date_val, item) VALUES (42, '2005-11-18',
    'O''Reilly');
```

Float non riesce a convertirlo in decimal (errore)

Python	PostgreSQL	Python	PostgreSQL	Python	PostgreSQL
None	NULL	bool	BOOL	float	REAL DOUBLE
int	SMALLINT INTEGER bigint	Decimal	NUMERIC	str	VARCHAR TEXT
date	DATE	time	TIME timetz	datetime	TIMESTAMP timestamptz

Maggiori dettagli: http://initd.org/psycopg/docs/usage.html

- Psycopg2 non fa parte della distribuzione standard.
- Python3 ha un meccanismo di installazione semplice (`pip`) dei moduli che sono registrati presso Python Packaging Index.
- `pip` può installare a livello di sistema (richiede diritti amministratore) o a livello utente. Dettagli: `https://docs.python.org/3/installing`.

Installazione Psyconpg2 su Ubuntu 14.04 Desktop

1. Si assume che Python3 sia già installato come pure il programma `pip3`. (Dalla versione Ptyhon 3.5, pip è già presente nella distribuzione).
2. Da una shell, si avvia l'installazione di Psycopg2 con il comando:
   ```
   pip3 install --user psycopg2
   ```
3. Dopo qualche compilazione, il modulo viene salvato in `~/.local/lib/python3.4/site-packages/` ed è disponibile automaticamente in ogni sessione Python3.

Lo schema tipico di un modulo Python che comunica con un DBMS PostgreSQL via Psycopg2 deve:

1. aprire una connessione tramite `conn = psycopg2.connect(...)`,
2. eventualmente modificare le proprietà di livello di isolamento, autocommit, readonly.
3. creare un cursore tramite `cur = conn.cursor()`,
4. eseguire le operazioni previste,
5. se la sessione non è in autocommit, eseguire un `conn.commit()` se si sono dati comandi SQL di aggiornamento (registra le modifiche) (o `conn.rollback()` per annullare).
6. chiudere il cursore, `cur.close()`, e la connessione, `conn.close()`.

Dalla versione 2.5 della libreria, la gestione dei `close` e dei `commit` è semplificata se si usa il costrutto `with`:

- Quando si usa una connessione con il `with`, all'uscita del blocco viene fatto un `commit` automatico e la connessione non viene chiusa.
- Quando si usa/crea un cursore con il `with`, all'uscita del blocco viene fatto un `close` automatico del cursore.

Try (file=open()) {} in java comunque chiude la risorsa anche se non lo faccio io

```
1  conn = psycopg2.connect(...)
2  with conn:
3      with conn.cursor() as cur1:
4          ...
5      print("Qui cur1 è stato chiuso")
6  print("Qui è stato fatto solo un commit: conn è ancora aperta!")
7  with conn:
8      with conn.cursor() as cur2:
9          ...
10     print("Qui cur2 è stato chiuso")
11 print("Qui è stato fatto un commit: conn è ancora aperta!")
12 conn.close()
```

Costrutto with è l'equivalente del try source in java (vedi sopra)

Metodo che crea un oggetto

Nome oggetto

Se incontra una eccezione viene chiusa cur1 in automatico comunque

Il blocco quando è finito viene chiuso cur1.close()

Quando esco dal cur1 ho finito la transazione significa che viene fatto il commit() ma la connessione non è chiusa

Così pero usando un'altra volta with conn faccio due transazioni con una sola connessione e poi alla fine devo chiuderla

Altrimenti se uso un solo with conn: e with conn.cursor() chiude la transazione facendo commit in automatico e la close in automatico

Si deve porre attenzioni alla combinazione di cursori sulla medesima connessione

- Aprire una connessione costa in tempo (e spazio). Meglio aprire/chiudere poche connessioni in un'esecuzione.
- Con un oggetto connessione si possono creare più cursori. Questi cursori condividono la connessione.
Non posso fare piu canali all'interno della stessa cnnesione
- Psycopg2 garantisce solo che le istruzioni inviate dai cursori vengono sequenzializzate.
- Quindi non si possono gestire transazioni concorrenti usando diversi cursori sulla medesima connessione.
- Regola pratica: usare più cursori sulla medesima connessione quando si fanno transazioni in auto-commit o solo transazioni di sola lettura.

Esempio di modulo basato su Psycopg2 (1/3)

```
1  """Created on 16 apr 2016
2  Gestione semplice tabella Spese su PostgreSQL
3  @author: posenato
4  """
5  from datetime import date
6  from decimal import  Decimal
7  import psycopg2
8
9  conn = psycopg2.connect(host="dbserver.scienze.univr.it", \
       database="psnrrt07", user="psnrrt07")
10 with conn:
11 ____with conn.cursor() as cur:
12 ________cur.execute("CREATE TABLE Spese (id SERIAL PRIMARY KEY, \
       data DATE, voce VARCHAR, importo NUMERIC)")
13 ________print("Esito creazione tabella: ", cur.statusmessage)
```

Esempio di modulo basato su Psycopg2 (2/3)

```
14      print("Esito creazione tabella: ", cur.statusmessage)
15      cur.execute( "INSERT INTO Spese (data, voce, importo) \
            VALUES (%s,%s,%s), (%s,%s,%s), (%s,%s,%s), (%s,%s,%s)", \
16              ( date(2016, 2, 24),"Stipendio",Decimal("0.1"), \
            date(2016, 2, 24),'Stipendio "Bis"',Decimal("0.1"), \
            date(2016, 2, 24),'Stipendio "Tris"',Decimal("0.1"), \
            date(2016, 2, 27),"Affitto",Decimal("-0.3") ) )
17      print("Esito inserimento tabella: ", cur.statusmessage)
18  #In questo punto il with precedente è chiuso:
19  #cursore chiuso e fatto un commit
20
21  with conn:
22      with conn.cursor() as cur: #si apre un nuovo cursore
23          cur.execute("SELECT id,data,voce,importo FROM Spese")
```

Esempio di modulo basato su Psycopg2 (3/3)

```
24 ____#Dentro il width interno, si stampa la tabella
25 ____print('=' * 55)
26 ____print("| {:>2s} | {:10s} | {:<20} | {:>10s} |".format( \
       "N", "Data", "Voce", "Importo"))
27 ____print('-' * 55)
28 ____tot = Decimal(0)
29 ____for riga in cur:
30 ________print("| {:>2d} | {:10s} | {:<20} | {:>10.2f} \
       |".format( riga[0], str(riga[1]), riga[2], riga[3]))
31 ________tot += riga[3]
32 ____print('-' * 55)
33 #In questo punto cur è chiuso ed è stato fatto un commit!
34 conn.close()# connessione chiusa!
35 #Si stampa il totale degli importi della tabella.
36 print("{:>40s}   {:10.2f}".format("Totale", tot))
```

Esempio di modulo basato su Psycopg2

Esito di una esecuzione:

```
Esito creazione tabella:  CREATE TABLE
Esito inserimento tabella:  INSERT 0 4
================================================================
|  N | Data       | Voce                    |    Importo |
----------------------------------------------------------------
|  1 | 2016-02-24 | Stipendio               |       0.10 |
|  2 | 2016-02-24 | Stipendio "Bis"         |       0.10 |
|  3 | 2016-02-24 | Stipendio "Tris"        |       0.10 |
|  4 | 2016-02-27 | Affitto                 |      -0.30 |
----------------------------------------------------------------
                                    Totale          0.00
```

- Java DataBase Connectivity (JDBC) è la Application Program Interface (API) ufficiale che descrive come un programma Java deve accedere a una base di dati esterna. Api per soddisfare le varie parti del programma (caratteristiche)
- Diversi gruppi di sviluppo hanno reso disponibili librerie JDBC per diversi tipi di DBMS.
- Alla pagina `http://www.oracle.com/technetwork/java/index-136695.html` c'è l'elenco aggiornato delle società che forniscono driver JDBC riconosciuti dalla Oracle.
- Per PostgreSQL, ci sono 2 implementazioni diverse.
- In questa lezione si considera la libreria JDBC v4.2 di PostgreSQL.org (`https://jdbc.postgresql.org/`). Si assume che la libreria sia stata scaricata e sia presente nel class path usato dall'interprete Java.

In un programma Java che voglia accedere a una base di dati di PostgreSQL deve:

1. Caricare il driver con l'istruzione `Class.forName(...)`. In questo modo, il driver viene registrato dentro la classe `DriverManager`.
2. Accedere a un database mediante la creazione di un oggetto di tipo `Connection`. Il metodo `DriverManager.getConnection(<uri>,<user>,<pw>)` accetta i parametri necessari per la connessione e ritorna un oggetto `Connection`.

Simile al connect in py

Lo stesso oggetto che restituisce in py solo che cambia la struttura essendo in java

```
DriverManager.getConnection(
"jdbc:postgresql://dbserver.scienze.univr.it/db0", "user0",
"secret")
```

3. La classe `Connection` contiene i seguenti metodi fondamentali:
   Specie di cursore
   - `createStatement()`: ritorna un oggetto `Statement`, che permette di inviare query statiche.
   Si passa una query con " si può istanziari i valori nella query
   - `prepareStatement("query")`: ritorna un oggetto `PreparedStatement`, che rappresenta la *query* ma che permette di reinviare la stessa più volte ma con parametri diversi. (Dettagli più avanti).
   Non ci sono i cursori ma dei specifici oggetti cambiando query
   - `commit()`: registra la transazione corrente.
     Attenzione! Normalmente in JDBC le connessioni sono in auto-commit, quindi un commit è sempre eseguito automaticamente dopo ogni esecuzione di comando.
   - `rollback()`: abortisce la transazione corrente.
   - `close()`: chiude la connessione corrente.

Differenza tra il tipo `Statement` e il tipo `PreparedStatement`:

- Un oggetto di tipo `Statement` è sufficiente per inviare query semplici senza parametri.

In py abbiamo execute. Qui sono divise in due le query; quelle semplice (statement) e l'altro fa prima una sorta di compilatori (una specie di pianificazione) Permette l'esecuzione specifica delle query più velocemente

- Un oggetto di tipo `PreparedStatement` è da preferire quando una stessa query deve essere riusata più volte con parametri diversi o anche, più semplicemente, non si vuole fare la conversione esplicita dei valori dei parametri da Java nei corrispondenti SQL.

Nota!

In questa lazione si considerano solo connessioni che usano oggetti di tipo `PreparedStatement`.

Classe `PreparedStatement`: metodi principali (1/2)

1. Un oggetto di tipo `PreparedStatement` è creato solitamente con il comando `prepareStatement(`*query*`)` di `Connection`, dove *query* è un comando SQL che contiene il carattere '?' in ogni posizione dove deve essere inserito un valore.

```
Connection con = DriverManager.getConnection(...)
PreparedStatement pst = con.prepareStatement("INSERT INTO Spese
    (data, voce, importo) VALUES (?,?,?), (?,?,?), (?,?,?),
    (?,?,?)")
```

2. I valori vengono inseriti invocando dei metodi `set...(<indice>, <valore>)` di `PreparedStatement`. Ci sono metodi per tutti i tipi supportati da PostgreSQL.

Ogni tipo deve essere compatibile con il tipo ch richiede la query (sopra)

```
pst.setDate(1, new Date(2016,02,24));
pst.setString(2, "Stipendio1");
pst.setBigDecimal(3, new BigDecimal("0.1"));
...
```

Per inserire i campi dentro pst uso .set

Classe `PreparedStatement`: metodi principali (2/2)

3. La query viene inviata con il comando `executeQuery()` se interroga o `executeUpdate()` se aggiorna, inserisce o cancella.
   - `executeQuery()` restituisce un oggetto di tipo `ResultSet` che contiene lo stato e l'eventuale tabella risultato della query.

```
PreparedStatement pst = con.prepareStatement("SELECT * FROM Spese
    WHERE id < ?");
pst.setInt(1, 4);
ResultSet st = pst.executeQuery();
```

Restituisce la tabella risultato

Meglio usare ? E usare i metodi set per i valori

Usato con select

   - `executeUpdate()` restituisce il numero di righe che sono state modificate.

```
PreparedStatement pst = con.prepareStatement("UPDATE Spese SET
    importo=importo*1.1");
int nRighe = pst.executeUpdate();
```

Restituisce numero di righe aggiunte modificate oppure cancellate

Esempio vedi slide prima (insert)

Classe `ResultSet`: metodi principali

- Nel caso in qui la query restituisce una tabella, l'oggetto `ResultSet` è un cursore sulla tabella risultato.
  Tabella iteratore e ResultSet è un iteratore
- Il metodo `next()` di `ResultSet` posiziona il cursore alla prossima riga non letta della tabella e restituisce vero se esiste, falso altrimenti.
  Next return una tupla della tabella ogni volta he viene chiamato
- I metodi `get<tipo>(<index>)` e `get<tipo>(<nome>)` di `ResultSet` permettono di recuperare il valore della colonna <index>/<nome> della riga corrente.
- Esempio per accedere alle righe risultato:

```
while (rs.next()) {
    System.out.print("Data: "+ rs.getData(1));
    System.out.print("Voce: "+ rs.getString("voce"));
    System.out.print("Importo: "+ rs.getBigDecimal(3));
}
```

Get restituisce il valore all'interno della riga (in una data posizione) ed invocare il metodo corretto (es. data)

Posso usare la posizione della colonna o il nome della colonna

Iterator legge il resultSet a buffer e se trovo il valore che ho cercato uso solo certi valori e non tutti i valori che non mi servirebbero

Esempio di modulo basato su JDBC (1/7)

```
1  package lezione08;
2  import java.math.BigDecimal;
3  import java.sql.*;
4  import java.text.*;
5  /**
6   * @author posenato
7   */
8  public class AccessoPostgreSQL {
9     /**
10     * Semplice classe demo di interazione con PostgreSQL.
11     * Solo alcune eccezioni vengono gestite.
12     * Le risorse vengono rilasciate in ogni caso.
13     * Richiede Java >= 7.
14     */
15    public static void main(String[] args)
16          throws ClassNotFoundException, SQLException,
        ParseException {
17
18       // Caricamento driver
19       Class.forName("org.postgresql.Driver");
```

Esempio di modulo basato su JDBC (2/7)

<type>://<url>/nomedatabse
Jdbc:postgresql://dbserver.scienze.univr.it:5432/db0 (id178kua)

Indirizzo URI

```
20      // Creazione connessione
21      try (Connection con = DriverManager.getConnection(
        "jdbc:postgresql://dbserver.scienze.univr.it:5432/db0",
        "user0", "xxx" )) {// try-with-resources
22
23          try (Statement st = con.createStatement()) {
24              st.executeUpdate( "DROP TABLE IF EXISTS Spese" );
25              // Creazione tabelle
26              st.execute( "CREATE TABLE Spese (id SERIAL PRIMARY
        KEY, data DATE, voce VARCHAR, importo NUMERIC)" );
27              System.out.println( "Esito creazione tabella: " +
        st.getUpdateCount() );
28          } catch (SQLException e) {
29              System.out.println( "Problema durante creazione
        tabella: " + e.getMessage() );
30              return;
31          }
```

Esempio di modulo basato su JDBC (3/7)

```
32          SimpleDateFormat sdf = new SimpleDateFormat(
      "dd/MM/yyyy" );
33          // Inserimento dati
34          try (PreparedStatement pst = con.prepareStatement(
      "INSERT INTO Spese (data, voce, importo) VALUES
      (?,?,?),(?,?,?),(?,?,?),(?,?,?) " )) {
35              pst.clearParameters();
36
37              pst.setDate( 1, new Date( sdf.parse( "24/02/2016"
      ).getTime() ) );
38              pst.setString( 2, "Stipendio" );
39              pst.setBigDecimal( 3, new BigDecimal( "0.1" ) );
40
41              pst.setDate( 4, new Date( sdf.parse( "24/02/2016"
      ).getTime() ) );
42              pst.setString( 5, "Stipendio \"Bis\"" );
43              pst.setBigDecimal( 6, new BigDecimal( "0.1" ) );
```

Esempio di modulo basato su JDBC (4/7)

```
44              pst.setDate( 7, new Date( sdf.parse( "24/02/2016"
    ).getTime() ) );
45              pst.setString( 8, "Stipendio \"Tris\"" );
46              pst.setBigDecimal( 9, new BigDecimal( "0.1" ) );
47
48              pst.setDate( 10, new Date( sdf.parse( "27/02/2016"
    ).getTime() ) );
49              pst.setString( 11, "Affitto" );
50              pst.setBigDecimal( 12, new BigDecimal( "-0.3" ) );
51              int n = pst.executeUpdate();
52              System.out.println( "Inserite " + n + " righe" );
53          } catch (SQLException e) {
54              System.out.println( "Errore durante inserimento dati:
    " + e.getMessage() );
55              return;
56          }
```

Qui comunque se vada bene o no l'esecuzione dello statement viene chiusa

Esempio di modulo basato su JDBC (5/7)

```
57          // Interrogazione e stampa
58          try (Statement st = con.createStatement()) {
59              ResultSet rs = st.executeQuery(
60                "SELECT id,data,voce,importo FROM Spese" );
61              System.out.println(String.format("%055d",
        0).replace('0', '='));
62              System.out.println(String.format("| %-2s | %-10s |
        %-20s | %10s |", "Id", "Data", "Voce", "Importo" ) );
63              System.out.println(String.format("%055d", 0).replace(
        '0', '-' ) );
64              BigDecimal tot = new BigDecimal( 0 );
```

Esempio di modulo basato su JDBC (6/7)

```
65                  while (rs.next()) {
66                      System.out.println( String.format(
67                        "| %2s | %10s | %-20s | %10.2f |"
68                        , rs.getInt("id")
69                        , sdf.format(rs.getDate("data"))
70                        , rs.getString("voce")
71                        , rs.getBigDecimal("importo")));
72                      tot.add( rs.getBigDecimal( "importo" ) );
73                  }
74                  //fine tabella
75                  System.out.println( String.format(
76                    "%055d", 0 ).replace( '0', '=' ) );
77                  System.out.println( String.format(
78                    "%036d", 0 ).replace( '0', ' ' )
79                    + "Totale "
80                    + String.format( "%10.2f", tot ) );
```

Esempio di modulo basato su JDBC (7/7)

```
81              } catch (SQLException e) {
82                  System.out.println( "Errore durante estrazione dati:
        " + e.getMessage() );
83                  return;
84              }
85          } catch (SQLException e) {
86                      System.out.println("Problema durante la
        connessione iniziale alla base di dati: " + e.getMessage());
87                      return;
88          }
89      }
90  }
```

Come funzionano i commit in java?

Di default non ci sono le transazioni e tutte le query sono in autocommit() in pratica si apre e si chiude una transazione

Se voglio usar il commit e un nuovo livello di isolamento devo cambiare i parametri della connessione quando viene creata

Esempio di modulo basato su JDBC

Esito di una esecuzione:

```
Esito creazione tabella: 0
Inserite 4 righe
=================================================
| Id | Data       | Voce                 |   Importo |
-------------------------------------------------
|  1 | 24/02/2016 | Stipendio            |      0,10 |
|  2 | 24/02/2016 | Stipendio "Bis"      |      0,10 |
|  3 | 24/02/2016 | Stipendio "Tris"     |      0,10 |
|  4 | 27/02/2016 | Affitto              |     -0,30 |
=================================================
                                  Totale      0,00
```