

PostgreSQL per programmatori

PostgreSQL

Gabriele Bartolini

Flavio Casadei Della Chiesa

Luca Ferrari

Marco Tofanari

Associazione Italiana PostgreSQL Users Group

www.itpug.org

Pisa, 8 Maggio 2009



Licenza Creative Commons

Attribuzione

Non commerciale

Condividi allo stesso modo

2.5 Italia

<http://creativecommons.org/licenses/by-nc-sa/2.5/it/>



Scaletta

Nozioni indipendenti dal linguaggio di programmazione

- Connessioni
- Interazioni con il database:
 - Statement
 - ResultSet
 - Prepared Statement (e problemi di SQLInjection)
 - Chiamata a stored procedure
- Transazioni, Isolation level e Savepoint

Esempi di API PostgreSQL per vari linguaggi

- Perl
- Java
- PHP



Schema generico per ogni linguaggio

- Introduzione
- Connessione
- SELECT
- INSERT e UPDATE
- Transazioni
 - Isolation Level
 - Savepoint
- Varie o avanzate



Introduzione

Questo corso ha lo scopo di presentare:

- come sia possibile connettersi ad un database PostgreSQL da diversi linguaggi di programmazione (Perl, Java, PHP)
- quali API utilizzare per eseguire query e, in generale, interagire con il database stesso



Database didattico: corso

```
CREATE TABLE corso
(
  corsopk serial NOT NULL, -- Chiave surrogata della tabella corso.
  corsoid character varying(10), -- Chiave reale della tabella corso.
  descrizione text, -- Descrizione del corso
  requisiti text, -- Elenco dei requisiti richiesti
  data date, -- Data in cui si tiene il corso
  n_ore integer, -- Numero di ore del corso.
  materiale oid, -- Materiale didattico distribuito con il corso
  ts timestamp with time zone DEFAULT ('now'::text)::timestamp,
  CONSTRAINT corso_chiave_surrogata PRIMARY KEY (corsopk),
  CONSTRAINT corso_chiave_reale UNIQUE (corsoid),
  CONSTRAINT corso_n_ore_check CHECK (n_ore > 0 AND n_ore < 8)
)
```



Database didattico: partecipante

```
CREATE TABLE partecipante
(
  partecipantepk serial NOT NULL, -- Chiave surrogata della tabella
  nome character varying(20), -- Nome del partecipante.
  cognome character varying(20),
  ts timestamp with time zone DEFAULT ('now'::text)::timestamp,
  partecipanteid character varying(30),
  CONSTRAINT partecipante_chiave_surrogata PRIMARY KEY (partecipantepk),
  CONSTRAINT partecipante_chiave_reale UNIQUE (partecipanteid)
)
```



Database didattico: j_corso_partecipante

```
CREATE TABLE j_corso_partecipante
(
    j_corso_partecipante_pk serial NOT NULL, -- Chiave surrogata
    corsopk integer NOT NULL,
    partecipantepk integer NOT NULL,

    CONSTRAINT j_corso_partecipante_chiave_surrogata
        PRIMARY KEY (j_corso_partecipante_pk),

    CONSTRAINT j_corso_partecipante_corsopk FOREIGN KEY (corsopk)
        REFERENCES corso (corsopk) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE NO ACTION,

    CONSTRAINT j_corso_partecipante_partecipantepk
        FOREIGN KEY (partecipantepk)
        REFERENCES partecipante (partecipantepk) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE NO ACTION
)
```



Concetti generali per la connettività database

PostgreSQL utilizza un protocollo di rete per lo scambio di dati con il mondo esterno. Tale protocollo identifica i vari *messaggi* che il client e il server devono scambiarsi per fare query, scorrere i risultati e in generale dare istruzioni al database stesso.

Ogni linguaggio propone una propria implementazione di tale protocollo attraverso un **driver**, ovvero un componente (libreria, classe, ...) capace di tradurre le richieste fatte nel linguaggio di programmazione stesso per e dal database.

I driver contengono *facility* di basso livello, e quindi difficilmente portabili fra differenti database (es. MySQL, PostgreSQL, SQL Server,...). Per questo motivo molti driver vengono acceduti attraverso un layer di astrazione generico (es. JDBC, DBI).



Livelli di astrazione generali

Tipicamente tutti i driver e le librerie forniscono i seguenti elementi:

- *Connection* è un componente usato per gestire una singola connessione al database, tipicamente incapsula una sessione “interattiva”. Può essere usato per esempio per regolare le funzioni relative alle transazioni.
- *Statement* è un componente che incapsula un comando SQL (ad es. una query) da inviare al database.
- *PreparedStatement* è simile al precedente, ma permette di essere ottimizzato nel caso di query cicliche.
- *ResultSet* contiene i dati forniti dal database in risposta ad una query (SELECT). Rappresenta un cursore sui dati.



PreparedStatement: hanno anche un'altra caratteristica

- Mai sentito parlare di SQLInjection?



SQLInjection?

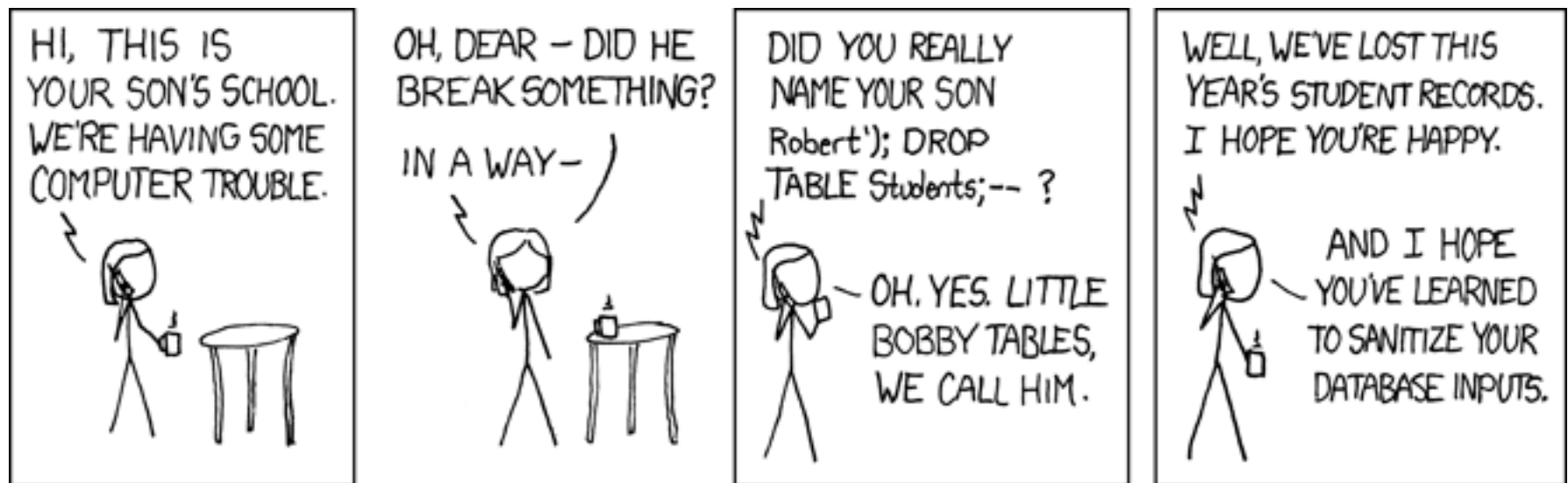


Immagine presa da <http://xkcd.com/327/> (Licenza Creative Commons)

SQLInjection

- Senza una “ripulitura” dei parametri semplici statement generati tramite concatenazione di stringhe possono essere letali:
 - Le query possono restituire troppi/tutti valori → ad esempio è possibile bypassare i meccanismi di autenticazione basati su username e password
 - Le basi di dati (complice una malsana gestione delle grant) possono venire modificate (vedi esempio precedente)



Esempio di SQLInjection

- Aggiungiamo un campo password alla tabella partecipante in modo da realizzare un “rozzo” sistema di autenticazione basato sulle credenziali (partecipanteid, password)
- Esempio (pseudo PHP)
 - `Select * from partecipante where partecipanteid = '$user' and password = '$password'`
 - Cosa succede se al posto della password viene passato
 - `Aaa' or user = 'pippo`
 - Oppure
 - `Aaa' or '1' = '1`
 - La query ritorna il “profilo” dell'utente pippo (anche non conoscendone la password) oppure, nel secondo caso, tutti gli utenti (!)



Soluzioni

- Ripulire i parametri di input prima di eseguire gli Statement
 - E' sempre una buona norma “escapare” gli apici
- Usare PreparedStatement
 - I Prepared Statement “escapano” i parametri in input prima dell'esecuzione sul db (vedremo esempi in seguito)



Stored procedure

- Una stored procedure è un programma scritto in SQL od in altri linguaggi, mantenuto nel database stesso, archiviato nel cosiddetto database data dictionary. Spesso è scritta in versioni proprietarie di SQL, che sono dei veri e propri linguaggi strutturati, come il PL/pgSQL di PostgreSQL o il PL/SQL di Oracle, all'interno dei quali è possibile scrivere codice SQL. (wikipedia)
- Sono invocabili con opportune API messe a disposizione dei vari linguaggi
- Esiste anche una sorta di invocazione da prompt psql:
`select funzione(arg1,arg2, ..., argn)`



Esempio (banale) di stored procedure

```
CREATE FUNCTION recuperapassword (TEXT)
RETURNS TEXT AS
$$
DECLARE
    utente ALIAS FOR $1;
    pass TEXT;
BEGIN
    SELECT INTO pass password FROM partecipante
        WHERE partecipanteid = utente;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Utente % non trovato', utente;
    END IF;
    RETURN pass;
END ;
$$ LANGUAGE 'plpgsql';
```



Stored procedure

- Sono memorizzata all'interno del database
- Possono essere scritte in vari linguaggi:
 - pl/pgsql
 - pl/java
 - pl/perl
 - ...
- Spostano parte della logica applicativa sul server



Transazioni (ACID)

Una transazione è un blocco di istruzioni logicamente correlate ed eseguite in modo da rispettare quattro proprietà fondamentali:

- Atomicità: una transazione deve completarsi nel suo insieme, indipendentemente dalle singole operazioni eseguite al suo interno.
- Consistenza: il database deve rimanere consistente al termine di una transazione.
- Isolabilità: una transazione deve poter essere “protetta” rispetto alle altre istruzioni e transazioni in esecuzione nel database.
- Durabilità: gli effetti di una transazione non possono essere temporanei, ma devono essere definitivi.

In poche parole una transazione è un blocco di istruzioni che viene eseguito nel suo insieme (*atomicità*), garantendo che ogni singola istruzione abbia avuto successo o altrimenti che tutte falliscano (*consistenza*). Per questo una transazione deve poter essere applicata su dati stabili (*isolamento*) e al suo termine i dati devono essere alterati permanentemente (*durabilità*).



Perché servono le transazioni?

In un ambiente single-user e single-thread non si hanno problemi: vi è solo una fonte che accede (e modifica) i dati.

Ma in un ambiente concorrente i dati vengono acceduti da più utenti e thread, e quindi si rischia di avere delle incoerenze.

Le transazioni servono a questo: consentono ad un utente/processo di identificare un set di operazioni da eseguire in modo atomico.

Oltre alla concorrenza ci possono essere problemi di crash dei client che effettuano le singole istruzioni. In questo caso, l'utilizzo delle transazioni garantisce che i dati vengano riportati ad uno stato coerente nel caso di *abort* improvvisi (a tal scopo PostgreSQL conserva un Write Ahead Log).



Transazioni vs Transazioni

Il problema principale nell'implementazione delle transazioni è il mantenimento della coerenza dei dati: piu' processi/utenti potrebbero modificare i dati che una transazione sta per leggere e modificare a sua volta.

E' per questo che vengono definiti degli standard di isolamento ai quali le transazioni devono obbedire.

Con i livelli di isolamento si definisce l'ambiente in cui la transazione agirà, ovvero quali vincoli si avranno nella lettura/aggiornamento dei dati.

Il livello di isolamento di fatto guida il *locking* dei record, e quindi agisce indirettamente sulla concorrenza del database (strict locking implica minor concorrenza ma dati piu' stabili e viceversa).

E' importante per il tuning del database e delle applicazioni!



Livelli di isolamento delle transazioni

PostgreSQL supporta i quattro livelli di isolamento standard, ma internamente implementa tutto con solo due livelli: *serializable* e *read committed* (il default)

<u>Livello di isolamento</u>	<u>Dirty Read</u>	<u>Nonrepeatable Read</u>	<u>Phantom Read</u>	<u>Supportato in PostgreSQL</u>
Read uncommitted	Possibile	Possibile	Possibile	NO
Read committed	Impossibile	Possibile	Possibile	SI
Repeatable read	Impossibile	Impossibile	Possibile	NO
Serializable	Impossibile	Impossibile	Impossibile	SI

Dirty Read: si leggono dati non ancora confermati da un'altra transazione

Nonrepeatable read: dati letti in precedenza sono ora modificati dal commit di un'altra transazione.

Phantom Read: il set di dati che soddisfa una certa condizione è ora cambiato



SAVEPOINT

PostgreSQL supporta i *savepoint*, dei punti definiti all'interno di una transazione che possono essere usati per un rollback parziale.

In sostanza ogni savepoint rappresenta una sottotransazione (ed in effetti i savepoint sono implementati come transazioni annidate) che può essere annullata indipendentemente dalla transazione generale.

Ovviamente la transazione generale ha la precedenza su ogni savepoint, quindi un commit/rollback globale impone lo stesso risultato su tutti i savepoint presenti.



Savepoint: esempio

```

pgdaydb=# begin;
BEGIN
pgdaydb=# insert into corso(corsoid, descrizione) values('PG-A','Basi di postgresql');
INSERT 0 1
pgdaydb=# savepoint sp1;
SAVEPOINT
pgdaydb=# insert into corso(corsoid, descrizione) values('PG-B','Amministrazione postgresql');
INSERT 0 1
pgdaydb=# rollback to savepoint sp1;
ROLLBACK
pgdaydb=# commit;
COMMIT
pgdaydb=# select * from corso;
 corsopk | corsoid |  descrizione  | requisiti | data | n_ore | materiale |          ts
-----+-----+-----+-----+-----+-----+-----+-----
    24 | PG-A   | Basi di postgresql |          |      |      |          | 2009-04-21 08:45:22.278038+02
(1 riga)

```

Si può specificare un punto nella transazione generale che viene usato per un rollback parziale della transazione (anche se questa viene committata).



Perl e PostgreSQL



Cosa è Perl

Creato nel 1987 da Larry Wall

Linguaggio di programmazione ad alto livello:

- creato inizialmente come ausilio ai sistemisti, come linguaggio di manipolazione di testo e file
- ha un insieme di funzionalità ereditate da C, shell unix, awk, sed e da altri linguaggi
- si è evoluto nel tempo (grazie ai moduli) in un linguaggio più generale (elaborazione di immagini, connettività database, networking, Web, ...)
- adatto per la proto-tipizzazione di programmi da implementarsi in altri linguaggi grazie ai tempi di sviluppo più rapidi
- supporta sia il paradigma procedurale che quello ad oggetti



Perché Perl

- + permette di scrivere codice in poco tempo
 - + vasta collezione di moduli
(CPAN 7230 autori 15332 moduli)
 - + pensato per essere pratico
 - non è stato pensato per essere compatto, elegante o minimale (si possono scrivere programmi veramente poco leggibili)
- (+ o –) c'è più di un modo di fare la stessa cosa



Come interfacciare Perl e PostgreSQL

Bisogna installare due moduli Perl: DBI e DBD::Pg

Ci sono 3 strade:

1) installazione dai sorgenti

http://www.cpan.org/modules/by-category/07_Database_Interfaces/DBI/

<http://search.cpan.org/dist/DBD-Pg/>

2) installazione tramite pacchetti della nostra distribuzione preferita

Su Ubuntu

(apt-get install postgresql-8.3 postgresql-client-8.3 postgresql-client-common postgresql-common)

apt-get install libdbi-perl libdbd-pg-perl

3) tramite CPAN

perl -MCPAN -e "install DBI"

perl -MCPAN -e "install DBD::Pg"



Due parole sul DBI (DataBase Interface)

Modulo Perl che fornisce dei metodi di gestione astratta di un Database (è un tipo di dato astratto, come nella filosofia OOP):

- Tutte le operazioni di scambio dati avvengono tramite una serie di metodi che funzionano allo stesso modo con qualunque database
- Deve essere abbinato con i vari driver dedicati che si occupano dei dettagli e delle eventuali differenze (DBD::Pg, DBD::Oracle, DBD::mysql ... etc ...)
- il suo motore interno è scritto in C



Connessione a PostgreSQL tramite DBI

```
#!/usr/bin/perl
use DBI;
use strict;
use warnings;
my $dbh=DBI->connect("dbi:Pg:dbname=scuola",'username','password')
    or die "Errore connessione DB";
... codice perl ...
$dbh->disconnect();
```

La riga di connessione puo' specificare anche dei parametri come nome dell'host,porta del db e altre cose come riportato qui sotto :

```
my $dbh=DBI->connect("dbi:Pg:dbname=scuola;
host='localhost';port=5432",'username','password',
{RaiseError=>0,PrintError=>0,AutoCommit=>1,pg_server_prepare=>1})
    or die "Errore connessione DB";
```



Sequenza di istruzioni tipica per una SELECT

```
prepare,  
    execute, fetch, fetch, ...  
    execute, fetch, fetch, ...  
    execute, fetch, fetch, ...
```

Per esempio:

```
$sth = $dbh->prepare(" SELECT nome,cognome FROM partecipante WHERE nome  
like ? ");  
$sth->execute( 'Marco' );  
while ( @row = $sth->fetchrow_array ) {  
    print "@row\n";  
}
```

Non esiste un metodo per trovare il numero di righe come
`pg_num_rows()` in PHP



Tipi di fetch

```
$sth = $dbh->prepare(" SELECT nome,cognome FROM partecipante WHERE nome like ? ");  
$parametro='Marco';
```

`$sth->fetchrow_array;`

```
$sth->execute($parametro);  
while ( my @nomi = $sth->fetchrow_array ) {  
    print "$nomi[0] $nomi[1]\n";  
}  
$sth->finish();
```



Tipi di fetch (ancora)

`$sth->fetchrow_arrayref;` (- leggibile + veloce)

```
$sth->execute($parametro);  
while ( my $ref1=$sth->fetchrow_arrayref){  
    print "$ref1->[0]  $ref1->[1] \n";  
}  
$sth->finish();
```

`$sth->fetchrow_hashref;` (+ leggibile - veloce)

```
$sth->execute($parametro);  
while ( my $ref1=$sth->fetchrow_hashref){  
    print "$ref1->{nome}  $ref1->{cognome}\n";    <= le chiavi dell' hash sono i  
    nomi dei campi  
}  
$sth->finish();
```



Sequenza di istruzioni tipica per uno statement non di tipo SELECT

```
prepare,  
    execute,  
    execute,  
    execute.
```

Per esempio:

```
$sth = $dbh->prepare("INSERT INTO  
partecipante(partecipantepk,nome,cognome,partecipanteid) VALUES (?, ?, ?, ?)");  
  
while(<CSV>) {  
    chomp;  
    my ($pippo,$pluto,$paperino,$minni) = split /,/;  
    $sth->execute( $pippo, $pluto, $paperino, $minni );  
}
```

Per modifiche “una tantum” posso usare il metodo do()

```
$num_righe=$dbh->do("UPDATE tabella SET conto=conto+1");
```



Crearsi una query di inserimento (con intelligenza)

Per facilitare la scrittura di insert con tabelle composte da molti campi

Questo evita di contare i caratteri ? da inserire

```
my @persona = qw(partecipantepk nome cognome partecipanteid);

my $campi = join ",", @persona;
my $sp = join ",", map {'?'} @persona; # $sp="?,?,?,?";
my $fsql=qq{ INSERT INTO partecipante ( $campi ) VALUES ( $sp ) };
my $sth= $dbh->prepare( $fsql );
my @io= qw(123456 marco tofanari pippol23);
$sth->execute( @io ); # <= execute si aspetta una lista di valori
```



Invocare una stored procedure

```
my $sth=$dbh->prepare(qq{
    select  recuperapassword(?) } );
my $parametro='Marco';
$sth->execute($parametro) or
    die "errore nel sth->execute $DBI::errstr\n";
my $ref1=$sth->fetchrow_hashref;
print "$ref1->{recuperapassword}\n";
```



Gestione transazioni

```
$dbh->{AutoCommit}=0; # abilita transazioni
    # uguale a $dbh->begin_work;
    # da mettere solo se ho iniziato la connessione con AutoCommit=>1
$dbh->{RaiseError} = 1; # il DBI "muore" se uno dei metodi fallisce
    # cosi' non bisogna testare il valore di ritorno di ogni metodo
eval {
    lavoro_di_select(...)      # molto lavoro qui ....
    lavoro_di_insert(...)      # includo insert
    lavoro_di_update(...)      # e updates
    $dbh->commit;
};
if ($@) {
    warn " Transazione abortita causa: $@";
    # adesso rollback per tornare indietro
    # ma anch'essa in eval{} perchè anch'essa puo' fallire
    eval { $dbh->rollback };
    # aggiungo codice per gestire errore
}
```



Savepoint

Per creare un savepoint

```
$dbh->pg_savepoint("mio_savepoint");
```

Per tornare a un savepoint

```
$dbh->pg_rollback_to("mio_savepoint");
```

Per rimuovere un savepoint

```
$dbh->pg_release("mio_savepoint");
```

Metodi specifici del DBD::Pg non del DBI



Transaction Isolation

Non ho trovato nella documentazione del DBI e del DBD:Pg cenni all'implementazione del transaction isolation

Bisogna gestirlo manualmente

```
$dbh->do("set transaction isolation level SERIALIZABLE");
```



Attenzione!

Se nella connessione abbiamo messo AutoCommit=0 deve essere la prima query dopo la connessione al DB altrimenti restituisce il seguente errore

ERROR: SET TRANSACTION ISOLATION LEVEL must be called before any query

Se nella connessione abbiamo messo AutoCommit=1 deve essere la prima query dopo dbh->begin_work altrimenti restituisce l'errore come sopra



Ottimizzazioni - Uso del binding

Senza binding il perl copia i dati dalle variabile interne al DBI alle nostre. Con il binding diciamo al DBI di scrivere direttamente sulle locazioni di memoria delle variabili che usiamo

```
my @nomi=(undef, undef) ;
$strsql="SELECT nome,cognome FROM partecipante WHERE nome like ?"
$sth = $dbh->prepare($strsql);
$sth->execute('Marco');
$sth->bind_columns( \ (@nomi) );    <=== dopo execute prima di fetch
print "$nomi[0]    $nomi[1]\n" while $sth->fetchrow_arrayref;
```

In questo esempio non è stato mai assegnato esplicitamente @nomi ma l'esempio stampa valori corretti => significa che il bind funziona

Una soluzione alternativa avrebbe potuto essere:

```
$sth->bind_col(1, \ $var1);
$sth->bind_col(2, \ $var2);
```

Oppure così:

```
$sth->bind_columns ( \ ( $var1, $var2) );
```



Combinare leggibilità e velocità

```
my %nomi = ( pippo => undef , pluto => undef );
$strsql="SELECT nome,cognome FROM partecipante WHERE nome like ?"
$sth = $dbh->prepare($strsql);
$sth->execute('Marco');
$sth->bind_col(1, \%nomi{pippo});
$sth->bind_col(2, \%nomi{pluto});
print "\$nomi{pippo}  \$nomi{pluto} while \$sth->fetchrow_arrayref";
```

Vantaggi : velocità data da **fetchrow_arrayref** e posso scegliere i nomi delle chiavi dell'hash (scelti apposta pippo e pluto come chiavi al posto di un piu' sensato nome e cognome)

Potevo anche usare **fetchrow_hashref** al posto di **fetchrow_arrayref** (funzionava uguale) ma avrei creato un nuovo hash ad ogni loop.

Con **fetchrow_arrayref** uso sempre il solito hash



Scorciatoie del Perl

Invece della trafila *prepare execute fetch* posso fare:

```
$strsql="select nome,cognome from partecipante where nome like ?";
$nomi=$dbh->selectall_arrayref($strsql,undef,'Marco');
foreach $var (@$nomi){
    print "$var->[0] $var->[1]\n";
}
```

==== > \$nomi e' un riferimento ad un array di riferimenti <====

==== > @\$nomi e' un array di riferimenti <====

Se avessi voluto avere solo il nome

```
@nomi=map{$_->[0]}@{$dbh->selectall_arrayref($strsql,undef,'Marco')};
foreach $var (@nomi){
    print "$var\n";
}
```

Se avessi voluto avere solo il cognome

```
@cog=map{$_->[1]}@{$dbh->selectall_arrayref($strsql,undef,'Marco')};
foreach $var (@cog){
    print "$var\n";
}
```



Per approfondimenti

- man DBI
- man DBD:pg:
- [DBI Recipes](#) di Giuseppe Maxia



Java e PostgreSQL



JDBC

JDBC (**J**ava **D**ata**B**ase **C**onnectivity) è un'API a livello SQL, ossia consente di inglobare comandi SQL nelle proprie applicazioni Java.

JDBC fornisce un'unica API CLI per l'accesso a database relazioni eterogenei. L'applicazione non necessita di adattamenti qualora il motore database cambi!

Applica il famoso concetto “*Write Once, Run Everywhere*” alla connettività database.

Si noti che l'idea non è *nuovissima*: ODBC si propone già come API unificata.

Non è indispensabile utilizzare JDBC!



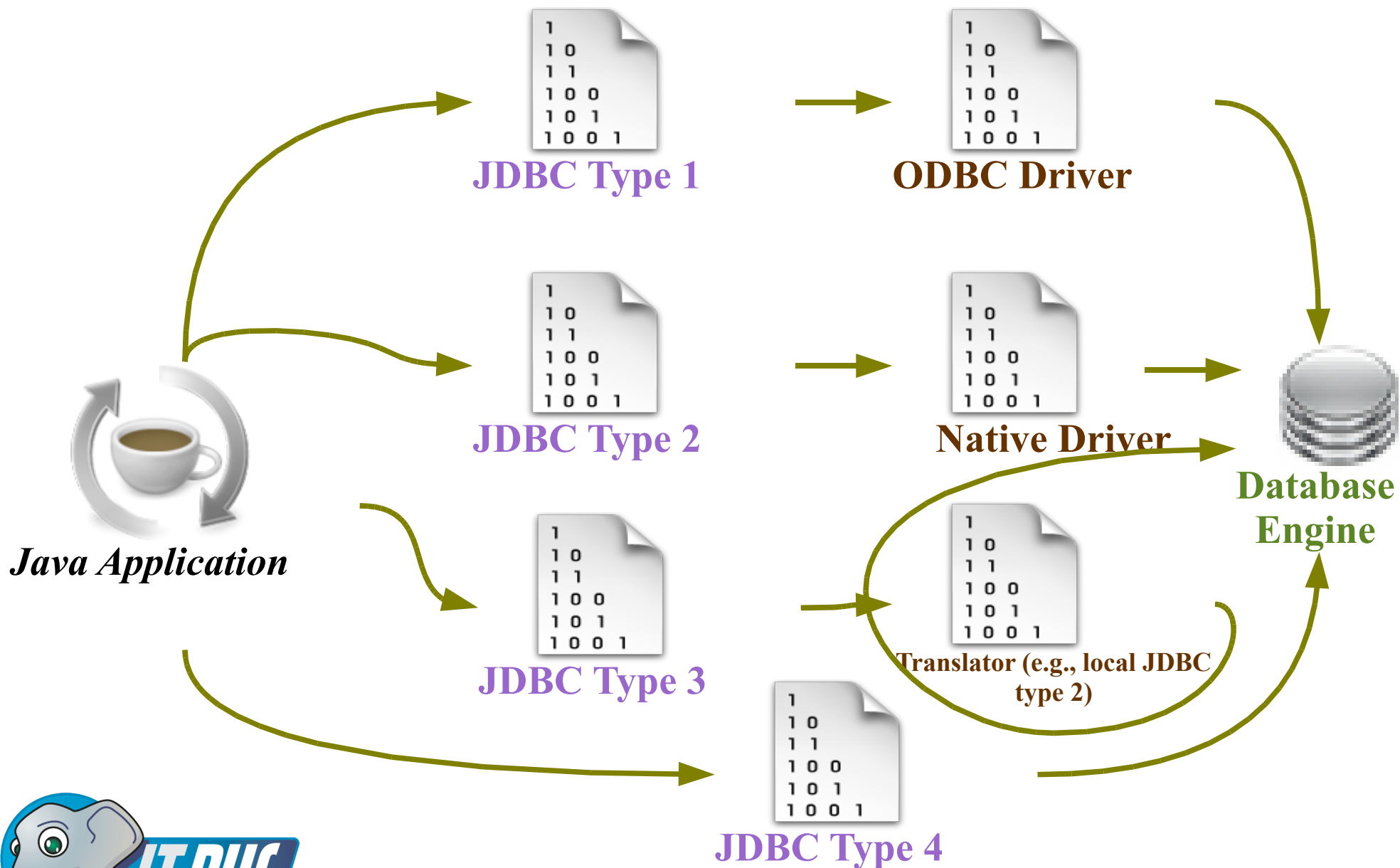
Livelli JDBC

Le Driver API riconoscono quattro modalità di implementazione differenti, chiamate *livelli* o *type* dei driver JDBC:

- type 1 (*JDBC-ODBC bridge*): viene sfruttato un accesso ODBC (che deve esistere!).
- type 2 (*Native API Drivers*): il driver richiama, tramite JNI, codice nativo (es. C/C++) per la connettività.
- type 3 (*Network Drivers*): il driver si collega (via TCP/IP) ad un componente lato server che si interfaccia a sua volta con il database server.
- type 4 (*Pure Java*): il driver si collega direttamente al database server e comunica utilizzando un protocollo opportuno.

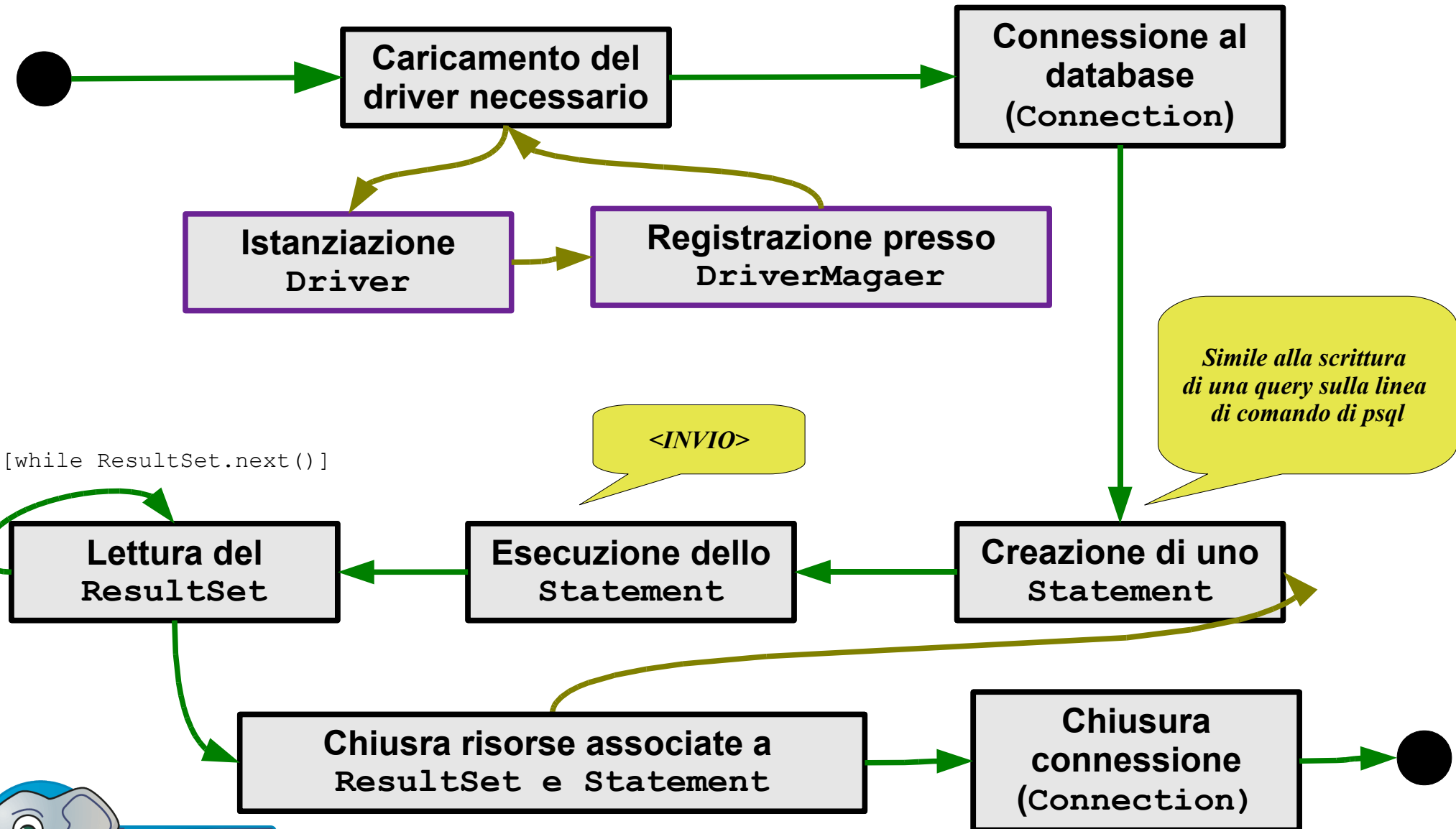


Livelli JDBC: schema riassuntivo



Tipica applicazione JDBC

Simile all'esecuzione del comando
psql -h host -U utente db



Simile alla scrittura
di una query sulla linea
di comando di psql

<INVIO>

`[while ResultSet.next()]`



Connessione al database

```
public static void main(String[] args)
    throws Exception{
    try{
        // classe del driver
        String driverName = "org.postgresql.Driver";

        // url di connessione
        String databaseURL =
            "jdbc:postgresql://localhost/pgdaydb";

        // caricamento della classe del driver
        Class driverClass = Class.forName(driverName);
        // creazione dell'istanza del driver
        Driver driver = (Driver)
        driverClass.newInstance();
```



Connessione al database

```
// a questo punto il driver è registrato
// presso il DriverManager (il driver di PostgreSQL
// si registra automaticamente al DriverManager
// appena la sua classe viene caricata)

// essendo il driver caricato e registrato posso
// ottenere una connessione al database
Connection connection =
    DriverManager.getConnection(databaseURL,
                                "luca",
                                // username
                                "xxx"
                                // password
                                );

// una volta ottenuta una connessione al database
// e' possibile interagire con esso per via di
// oggetti di tipo Statement
```



Creazione di uno Statement

```
// creazione di una query e di uno statement  
// da usare (la query mostra i corsi  
// che contengono java nella descrizione  
// e i relativi iscritti)
```

```
String query = "SELECT c.corsoid, c.descrizione, c.data,  
                c.n_ore, p.nome, p.cognome " +  
                " FROM (corso c LEFT JOIN j_corso_partecipante  
                j ON c.corsopk = j.corsopk) " +  
                " LEFT JOIN partecipante p ON p.partecipantepk  
                = j.partecipantepk " +  
                " WHERE c.descrizione ilike '%java%' ";
```

```
// notare che lo Statement non e' ancora associato  
// a nessuna query
```

```
Statement statement = connection.createStatement() ;
```

```
// effettuo la query
```

```
ResultSet rs = statement.executeQuery(query) ;
```



Analisi dei risultati: ResultSet

```
// visualizzo i dati ottenuti dalla query: l'oggetto
// ResultSet (rs) rappresenta un  cursore  dei risultati
// che può essere scorso.
while( rs.next() ){
    System.out.println("#####");

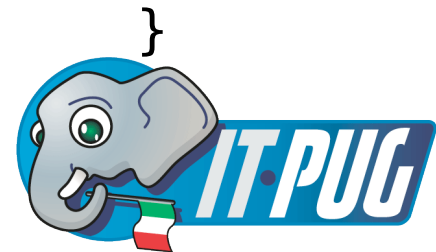
    // posso ottenere i dati di una riga chiamando i
    // metodi getXXX (XXX = tipo di dato) e specificando
    // il numero della colonna (partendo da 1) o il suo
    // nome simbolico
    System.out.println("Id-Corso e descrizione: "
        + rs.getString(1) + " " + rs.getString(2));
    System.out.println("Data e numero ore: "
        + rs.getDate(3) + " " + rs.getInt(4));
    System.out.println("Iscritto: "
        + rs.getString("nome") + " "
        + rs.getString("cognome"));
}
```



Fine del programma e gestione delle eccezioni

```
// chiusura della connessione
connection.close();

}catch(SQLException exception){
    // eccezione SQL (problema nella query, errore del
    // server, etc.)
}catch(ClassNotFoundException ex){
    // classe del driver JDBC non trovata
}catch(InstantiationException ex){
    // errore di reflection (classe driver non
    // istanziabile)
}catch(IllegalAccessException ex){
    // errore di reflection (classe driver non
    // accessibile)
}
}
}
```



```
Console X
<terminated> PGJDBC1 [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.06/bin/java (08/set/2008 18:26:20)
#####
Id-Corso e descrizione: Java08 Corso di base connettività Java.
Data e numero ore: 2008-10-18 1
Iscritto: Enrico Pirozzi
#####
Id-Corso e descrizione: Java08 Corso di base connettività Java.
Data e numero ore: 2008-10-18 1
Iscritto: Luca Ferrari
#####
Id-Corso e descrizione: Java07 Corso connettività Java, compilazione driver, utilizzo di base, transazioni
Data e numero ore: 2007-07-07 1
Iscritto: null null
```

Riassunto:

- **Caricamento del driver PostgreSQL** (necessario solo all'avvio del thread principale dell'applicazione);
- **Creazione di una connessione** specificando l'URL di connessione;
- **Creazione di uno Statement;**
- **Esecuzione dello Statement;**
- **Lettura dei risultati tramite il ResultSet;**
- **Chiusura delle risorse.**



Considerazioni sul ResultSet

Un ResultSet rappresenta un *cursore* sui risultati di una query. Il posizionamento fra le righe del cursore avviene tramite metodi specifici, quali **next()** e **previous()**.

Ogni ResultSet è legato allo Statement che lo ha generato; se lo Statement viene *alterato* il ResultSet è automaticamente invalidato!



Ancora sul ResultSet

Esistono metodi per recuperare il valore di ogni colonna secondo il tipo di appartenenza (String, int, Date, ...):

<tipo_Java> get<tipo_Java>(int colonna)

<tipo_Java>get<tipo_Java>(String nomeColonna)

Ogni tipo SQL viene mappato in un tipo Java (e viceversa) mediante una specifica tabella di conversione (<http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/mapping.html>).

Tale tabella può essere estesa dall'utente per mappare i tipi *user defined*.

Si presti attenzione al fatto che, contrariamente alla logica degli array, le colonne sono numerate partendo da 1!



Esecuzione di un INSERT/UPDATE/DELETE ITPUG - 08-05-2009

```
// creazione di una query e di uno statement
// da usare per inserire un nuovo corso
String query = "INSERT INTO corso(corsoid,descrizione)
                VALUES('Java_adv', 'Corso avanzato su Java/JDBC') ";
Statement statement = connection.createStatement();

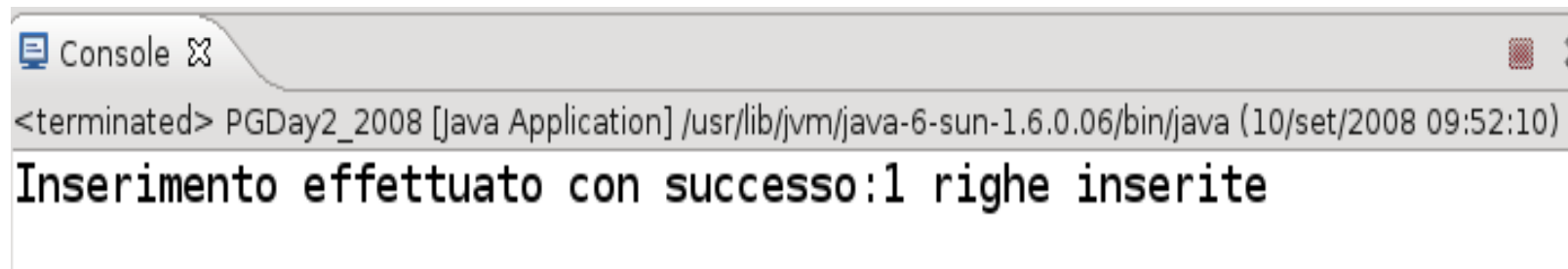
// chiedo al server di eseguire la query e
// di fornirmi il numero di righe "toccate"
// tutte le query che modificano i dati vanno eseguite
// attraverso il metodo executeUpdate(..)
int rows = statement.executeUpdate(query) ;

if( rows > 0)
    System.out.println("Inserimento effettuato con
                        successo:" + rows + " righe inserite");

// chiusura della connessione
connection.close();
```



Esecuzione



```
Console ✕  
<terminated> PGDay2_2008 [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.06/bin/java (10/set/2008 09:52:10)  
Inserimento effettuato con successo:1 righe inserite
```

Riassunto:

- **Caricamento del driver PostgreSQL** (necessario solo all'avvio del thread principale dell'applicazione);
- **Creazione di una connessione** specificando l'URL di connessione;
- **Creazione di uno Statement**;
- **Esecuzione dello Statement**;
- **Lettura del valore di ritorno dell'esecuzione dello Statement**, tale valore indica il numero di righe *toccate* dalla query;
- **Chiusura delle risorse**.



PreparedStatement

Un PreparedStatement è un oggetto che memorizza uno statement SQL precompilato, che può essere parametrizzato e che può eseguire più efficientemente di uno Statement normale.

L'idea è di precompilare un comando SQL *parziale* e di completarlo successivamente, al momento in cui l'esecuzione sia necessaria.

Lo statement precompilato può poi essere riutilizzato più volte, cambiando eventualmente i parametri di esecuzione.

Un PreparedStatement è una sottointerfaccia di Statement!

Essendo uno statement parametrizzabile, PreparedStatement fornisce dei metodi setXXX e getXXX per impostare i parametri di esecuzione.



Prepared Statement

```
// creazione di una query di inserimento e dello  
// statement precompilato per la sua esecuzione  
String query = "INSERT INTO corso(corsoid,descrizione)  
                VALUES(?,?) ";
```

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

```
// inserisco alcuni corsi fornendo gli opportuni
```

```
// parametri al server
```

```
int rows = 0;
```

```
for(int i = 0; i < 10; i++){
```

```
    String corsoid = "corso_" + i;
```

```
    String descrizione = "Nessuna descrizione ";
```

```
    // impostazione dei parametri nello statement
```

```
    statement.setString(1, corsoid);
```

```
    statement.setString(2, descrizione);
```

```
    // eseguo lo statement (non devo specificare una query
```

```
    // poiché lo statement è già stato precompilato)
```

```
    rows += statement.executeUpdate();
```



CallableStatement

- Interfaccia per l'invocazione di stored procedure
- Estende PreparedStatement
- Sintassi per l'invocazione delle stored procedure in maniera indipendente dal database
 - {`?= call <procedure-name>[<arg1>,<arg2>, ...]`} (funzioni con valore di ritorno)
 - {`call <procedure-name>[<arg1>,<arg2>, ...]`} (funzioni senza valore di ritorno)



CallableStatement: esempio

```
CallableStatement cstmt = conn.prepareCall(
    "{? = call recuperapassword (?)}" );
cstmt.registerOutParameter(1, Types.VARCHAR);
cstmt.setString(2, "pippo");
cstmt.execute();
String pass = cstmt.getString(1);
System.out.println(pass);
...
```

Anche qua si parte da 1 con gli indici.



Transazioni

Le transazioni sono gestite a livello di *Connection*.

Per impostazione predefinita tutti i comandi eseguiti tramite uno statement sono eseguiti in “auto-commit”. E' possibile iniziare una nuova transazione con la chiamata al metodo:

```
Connection.setAutoCommit( false );
```

la transazione puo' poi essere terminata con uno dei metodi seguenti:

```
Connection.commit();
```

```
Connection.rollback();
```



Transazioni

E' anche possibile “registrare” un savepoint in una transazione aperta mediante il metodo *setSavepoint(..)*, che ritorna un oggetto *Savepoint* che puo' essere usato per un successivo rollback:

```
Connection connection =  
connection.setAutoCommit( false );  
  
...  
Savepoint sp1 = Connection.setSavepoint( “SP1” );  
  
...  
connection.rollback( sp1 );
```



PHP

- PHP è un linguaggio di scripting per il web molto popolare, fra i più diffusi al mondo
- Dalla versione 5 c'è un supporto minimale – per molti può essere soddisfacente - per la programmazione Object Oriented
- PRO:
 - Relativamente semplice da imparare
 - Ricco di estensioni
 - Ottimo supporto di comunità e larga diffusione a livello di hosting
 - Coesione con Apache
 - Multi-piattaforma
- CONTRO:
 - Prono a errori (non è tipizzato)
 - Carenza di strumenti di debug
 - Per applicazioni di larga scala: non ha un application server alle spalle



PHP e PostgreSQL

- E' possibile accedere a database PostgreSQL da PHP
- Esistono diverse tecniche per accedere a database PostgreSQL, e la scelta dipende da:
 1. Sviluppo di **applicazioni indipendenti da database** (*database independent*): ad esempio un CMS open-source che permetta la selezione del tipo di database a cui connettersi (MySQL, PostgreSQL, Firebird, ecc.)
 2. Sviluppo di **applicazioni per PostgreSQL**: applicazioni che si interfacciano con PostgreSQL e utilizzano funzioni native e uniche presenti su quel DBMS (ad esempio stored procedure)
- In questo intervento, una volta esposto le soluzioni per il lo sviluppo di applicazioni “*database independent*” esamineremo esclusivamente le applicazioni relative al punto 2, quelle esclusive per PostgreSQL



Applicazioni database independent

- Se l'obiettivo è quello di scrivere applicazioni in grado di interfacciarsi con più database, esistono diverse soluzioni per PHP
- Il concetto di fondo è il seguente: utilizzare soltanto le istruzioni standard SQL comuni a tutti i DBMS che desideriamo supportare
- È pertanto vietato utilizzare tutte quelle istruzioni e funzionalità esclusive per un singolo DBMS (ad esempio, integrità referenziale, stored procedure, viste, ecc.)
- PHP offre:
 - PDO (PHP Data Objects) - <http://www.php.net/manual/en/book.pdo.php>
 - MDB2 (tramite PEAR) - <http://pear.php.net/package/MDB2>
 - ODBC - <http://www.php.net/manual/en/book.uodbc.php>
- Ad ogni modo, una volta appresi i concetti di fondo comuni a tutti i linguaggi di programmazione espressi all'inizio dell'intervento, è molto semplice utilizzare una delle soluzioni sopra elencate



Applicazioni per PostgreSQL

- Come negli altri linguaggi di programmazione, per eseguire query su un database, è necessario eseguire i seguenti passi:
 1. Connessione
 2. Esecuzione della query
 3. In caso di set di risultati (*resultset*):
 1. Recupero delle righe e dei campi
 2. Liberazione della memoria (distruzione del *resultset*)
 4. Disconnessione
- I punti dal 2 al 3 possono essere ripetuti più volte
- Per *resultset* che ritornano più di una riga, il punto 3.1 può essere ripetuto più volte (solitamente una volta per ogni riga ritornata)
- Il punto 4 è facoltativo, ma è buona pratica disconnettersi una volta terminate le operazioni sul database (è segno di serietà e attenzione da parte del programmatore!)



Connessione

- Per connettersi ad un database PostgreSQL da PHP è sufficiente utilizzare la funzione `pg_connect($connection_string)`
- Supponendo di avere le variabili di connessione memorizzate in un array denominato `$db_settings`:

```
$connection_string = 'host=' . $db_settings['host']  
    . ' user=' . $db_settings['username']  
    . ' password=' . $db_settings['password']  
    . ' dbname=' . $db_settings['dbname'];
```

```
$connection = pg_connect($connection_string);
```

- Rif: <http://it2.php.net/manual/en/function.pg-connect.php>



Esecuzione di una query

- Per eseguire una query su PostgreSQL è sufficiente utilizzare la funzione `pg_query($connection, $query)`
- La funzione ritorna un resultset oppure `FALSE` in caso di errore

```
$query = 'SELECT * FROM partecipante';  
$result = pg_query($connection, $query);
```

- Rif: <http://it2.php.net/manual/en/function.pg-query.php>



Recupero delle righe

- Esistono diverse alternative per recuperare righe da un resultset, a seconda della struttura dati desiderata per memorizzare la riga. PHP mette a disposizione diverse strutture dati (array sequenziale, array associativo, entrambi, oggetto)

```
$items = array(); // array dei risultati
while ($row = pg_fetch_array($result)) {
    $items[$row['id']] = $row;
}
```

- Rif:
 - <http://it2.php.net/manual/en/function.pg-fetch-array.php>
 - <http://it2.php.net/manual/en/function.pg-fetch-assoc.php>
 - <http://it2.php.net/manual/en/function.pg-fetch-row.php>
 - <http://it2.php.net/manual/en/function.pg-fetch-object.php>



Distruzione del resultset

- Per liberare la memoria detenuta da un resultset è sufficiente utilizzare la funzione `pg_free_result($result)`
- La funzione agisce sul valore restituito da una funzione `pg_query` invocata precedentemente

```
pg_free_result($result) ;
```

- Rif: <http://it2.php.net/manual/en/function.pg-free-result.php>



Disconnessione

- Per disconnettersi è necessario utilizzare la funzione `pg_close($connection)`
- La funzione agisce sul valore restituito da una funzione `pg_connect` invocata precedentemente

`pg_close($connection) ;`

- Rif: <http://it2.php.net/manual/en/function.pg-close.php>



Altri argomenti

- Connessioni persistenti (il loro uso è sconsigliato!):
 - <http://it2.php.net/manual/en/function.pg-pconnect.php>
- Prepared Statement:
 - <http://it2.php.net/manual/en/function.pg-prepare.php>
 - <http://it2.php.net/manual/en/function.pg-execute.php>
- Encoding: fare sempre attenzione alla codifica con cui i dati vengono negoziati con il database
- Escape: fare attenzione effettuare il corretto 'escape' dei caratteri specie per i campi provenienti da form (evitare SQL injection):
 - <http://it2.php.net/manual/en/function.pg-escape-string.php>

