CORSO JAVA DEVELOPER

Java + MySQL

Come comunicano Java e SQL?

JDBC

JDBC (Java Data Base Connectivity) è un'interfaccia che permette di interagire con database relazionali

Con JDBC si possono eseguire in Java operazioni SQL indipendentemente dal tipo di database

Per usare JDBC bisogna installare un driver specifico per il tipo di database

Per connetterci a un database MySQL abbiamo bisogno del driver JDBC per MySQL, chiamato *MySQL Connector/J*.

JDBC con Maven

Come tutte le librerie Java, anche i driver JDBC sono dei **jar**

Possiamo utilizzare il driver JDBC nel nostro progetto Java usando Maven

```xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
</dependency>
```

aggiungendo la dependency nel POM

Maven cerca il jar nel repository locale, seguendo il path mysql/mysql-connector-java/8.0.28

se non lo trova si connette al proprio repository centrale e lo scarica

In Java la connessione al database è... indovinate un po'....

Connessione al database

La connessione a database è un oggetto della classe java.sql.Connection

```java
public static void main(String[] args) {

  String url = "jdbc:mysql://localhost:3306/test";
  String user = "root";
  String password = "rootpassword";

  Connection con = null;
  try {
        con = DriverManager.getConnection(url, user, password);
  } catch (SQLException ex) {
     ex.printStackTrace();
  } finally{
    if(con != null){
      con.close();
    }
  }
}
```

Per ottenere l'oggetto di tipo Connection, dobbiamo chiederlo al driver JDBC, tramite il metodo getConnection della classe java.sql.DriverManager

Al DriverManager dobbiamo indicare a quale db connettersi, attraverso il connection url

Il DriverManager si autentica nel nostro db con username e password

Try-with-resources

Dichiarare una risorsa in un blocco try in modo che venga chiusa dopo l'esecuzione di quel blocco

```

  // istanzio la risorsa nel try
  try (Connection con = DriverManager.getConnection(url, user, password)){

    // qui posso usare la risorsa

  } catch (SQLException ex) {
     ex.printStackTrace();
  }
  // non serve usare finally per chiudere la risorsa
  // finally{
  //  if(con != null){
  //    con.close();
  //  }

```

Connection url

Il connection url è una **stringa** che contiene le informazioni necessarie a rintracciare il database

```
1 String url = "jdbc:mysql://[host]:[port]/[nome_database]";
```

Il formato della stringa è standard e va popolata con i parametri del database

host: è l'indirizzo IP della macchina su cui risiede il database, nel nostro caso localhost o 127.0.01

port: è la porta a cui risponde il database, per MySQL la porta di default è 3306

nome_database: il nome del database a cui ci vogliamo connettere

Il formato del connection url è specifico per il tipo di database

Query al database

```java
try (Connection con
      = DriverManager.getConnection(url, user, password)) {

  // scrivo la query sql
  String sql = "SELECT room_number, floor from stanze";

  // chiedo alla connessione di prepararsi ad eseguire il mio sql
  try(PreparedStatement ps = con.prepareStatement(sql)){
    // eseguo la query che restituisce un result set
    try(ResultSet rs = ps.executeQuery()){
      //posso usare il risultato della query
    }
  }
```

l'oggetto Connection ci permette di preparare delle istruzioni SQL (**java.sql.PreparedStatement**) e di eseguirle

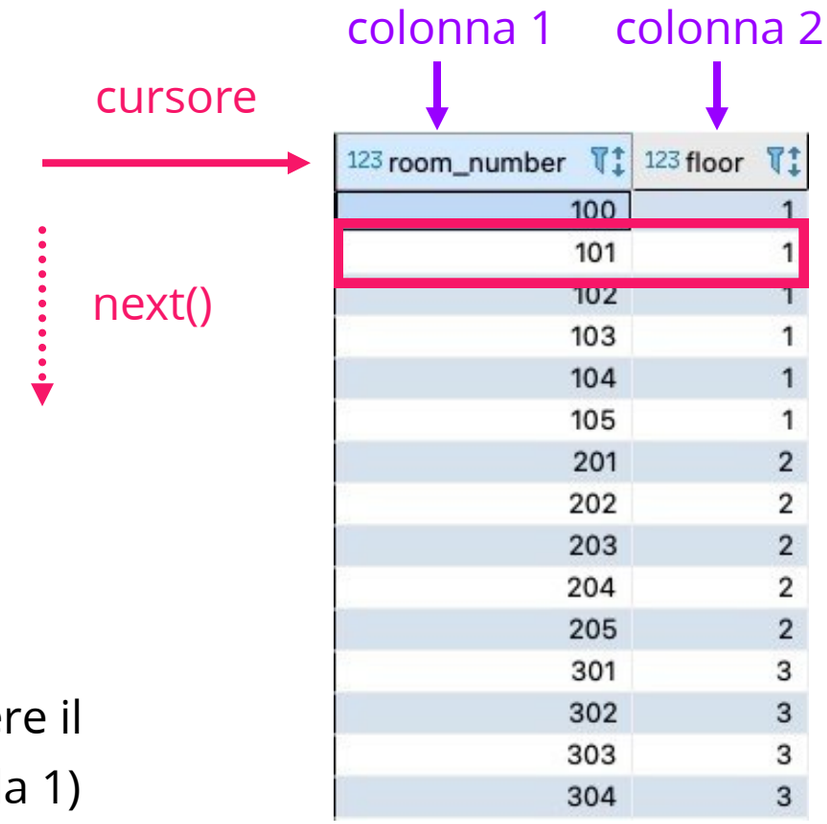
il risultato di **executeQuery** è un oggetto di tipo **java.sql.ResultSet**

Connection, PreparedStatement e ResultSet vengono chiusi automaticamente dal try-with-resources

ResultSet

```java
ResultSet rs = ps.executeQuery();

// itero sul result set
while (rs.next()) {
  int roomNumber = rs.getInt(1);
  int floor = rs.getInt(2);
}
```



java.sql.ResultSet presenta i dati in forma tabellare, con un cursore che ci permette di iterare su di essi riga per riga (**next()**)

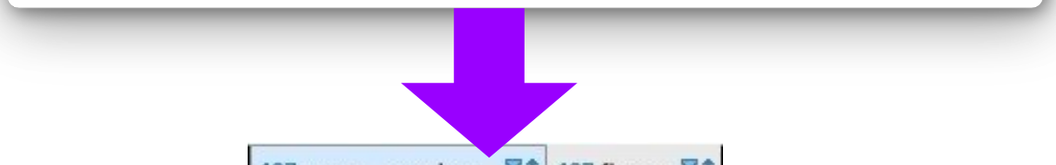
Quando il cursore è posizionato su una riga posso ottenere il valore di ogni colonna, identificata dall'indice (che inizia da 1)

c'è un metodo per ogni tipo di dato : getInt(n), getString(n), getDouble(n),...

ResultSet

= risultato della query

```sql
-- query sql
SELECT `room_number`, `floor` FROM `stanze`;
```



123 room_number	123 floor
100	1
101	1
102	1
103	1
104	1
105	1
201	2
202	2
203	2
204	2
205	2
301	3
302	3
303	3
304	3

la tabella con il risultato della query non corrisponde per forza alla tabella su cui stiamo eseguendo la select

Le colonne del result set sono definite nella SELECT e corrispondono esattamente alle colonne della tabella solo in caso di SELECT *

Le righe del result set sono un sottoinsieme delle righe della tabella, filtrate dalla WHERE, se presente

DANGER - SQL INJECTION

```
1   /* esempio SELECT con parametri */
2   // ...
3
4   String sql = "SELECT * FROM users WHERE name = '" + user "' AND password = '" + password + "';";
5
6   PreparedStatement ps = con.prepareStatement(sql);
7   ResultSet rs = ps.executeQuery();
```

SQL Injections

Hello computer security

```java
void login(String user, String password){
  String sql = "SELECT * FROM users WHERE name = " + user " AND password = " + password + ";";

  PreparedStatement ps = con.prepareStatement(sql);
  ResultSet rs = ps.executeQuery();
  if(rs.next()){
    System.out.println("You can enter the restricted area");
  }
}

```

SQL Injections

Hello computer security

Prima di arrivare al database,
l'input dell'utente **deve essere sempre:**

- validato
- controllato
- sanitizzato
- “escaped”

LIVE CODING

Sql Injection

Come evitare le SQL Injections?

Parameter binding!

Le prepared statements permettono di "preparare" l'istruzione SQL parametrizzata utilizzando dei **segnaposto** per i valori da inserire nella query.
Questi vengono sostituiti con una successiva funzione che riceve come parametri i valori effettivi da inserire nella query e l'indice (a partire da 1) del segnaposto

```java
void login(String user, String password){
  String sql = "SELECT * FROM users WHERE name = ? AND password = ?;";

  PreparedStatement ps = con.prepareStatement(sql);

  ps.setString(1, inputName);
  ps.setString(2, inputPassword);

  ResultSet rs = ps.executeQuery();

  if(rs.next()){
    System.out.println("You can enter the restricted area");
  }
}
```

c'è una funzione di binding per ogni tipo di dato

setString, setInt, setBoolean,...

Come evitare le SQL Injections?

Parameter binding!

Dato che i **parametri vengono passati in un secondo momento**, utilizzando un diverso protocollo, le SQL Injections sono inefficaci e non è nemmeno necessario fare l'escaping dei caratteri pericolosi.

LIVE CODING

INSERT con Java

```java
String sql = "INSERT INTO stanze(room_number, floor) VALUES(?,?);";

PreparedStatement ps = con.prepareStatement(sql);

ps.setInt(1, 404);
ps.setInt(2, 4);

int affectedRows = ps.executeUpdate();
```

Come per la SELECT usiamo un prepared statement con la INSERT SQL

Abbiniamo i valori da passare ai segnaposto con il parameter binding

Per eseguire l'istruzione di **INSERT** si utilizza la funzione **executeUpdate()**

executeUpdate ritorna il numero delle righe che sono state modificate dalla query

UPDATE con Java

```java
String sql = "UPDATE stanze SET beds = ? WHERE room_number = ?;";

PreparedStatement ps = con.prepareStatement(sql);

ps.setInt(1, 2);
ps.setInt(2, 404);

int affectedRows = ps.executeUpdate();
```

Usiamo un prepared statement con la UPDATE SQL

Abbiniamo i valori da passare ai segnaposto con il parameter binding

Per eseguire l'istruzione di **UPDATE** si utilizza la funzione **executeUpdate()**

executeUpdate ritorna il numero delle righe che sono state modificate dalla query

DELETE con Java

```java
String sql = "DELETE FROM stanze WHERE room_number = ?;";

PreparedStatement ps = con.prepareStatement(sql);

ps.setInt(1, 404);

int affectedRows = ps.executeUpdate();
```

Usiamo un prepared statement con la DELETE SQL

Abbiniamo i valori da passare ai segnaposto con il parameter binding

Per eseguire l'istruzione di **DELETE** si utilizza la funzione **executeUpdate()**

executeUpdate ritorna il numero delle righe che sono state modificate dalla query

Transactions

Proprietà acide...remember?

Quando per completare un'operazione dobbiamo eseguire più di una query, per mantenere lo stato del database coerente dobbiamo eseguirle in **transaction**

1. **Atomicità:** la transazione è indivisibile nella sua esecuzione e la sua esecuzione deve essere o totale o nulla, non sono ammesse esecuzioni parziali

2. **Coerenza:** non devono verificarsi contraddizioni (incoerenza) tra i dati archiviati nel DB

3. **Isolamento:** ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione

4. **Durabilità:** (o persistenza): una volta che una transazione abbia richiesto un commit work, i cambiamenti apportati non dovranno essere più persi.

Transactions con Java

```java
try(Connection connection = DriverManager.getConnection(CONNECTION_URL, USER, PASSWORD)) {
    connection.setAutoCommit(false);
    PreparedStatement firstStatement = connection.prepareStatement("firstQuery");
    firstStatement.executeUpdate();
    PreparedStatement secondStatement = connection.prepareStatement("secondQuery");
    secondStatement.executeUpdate();
    connection.commit();
} catch (Exception e) {
    connection.rollback();
}
```

Di default l'oggetto Connection esegue un commit in seguito ad ogni esecuzione

Per gestire manualmente la transaction dobbiamo disabilitare l'autocommit

Al termine dell'esecuzione delle query in transaction eseguiamo il commit

Se si verifica un errore annulliamo tutta l'operazione eseguendo un rollback

LIVE CODING

ESERCITAZIONE